UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA TELEMÁTICA

TRABAJO FIN DE GRADO

Detección y tracking de objetos utilizando visión y láser 3D para vehículos inteligentes

Autor: Carlos Pérez de Rivas.

Tutor: Rafael Barea Navarro.

Tribunal:

Presidente: Cristina Losada Gutierrez

Vocal 1º: Manuel Ocaña Miguel

Vocal 2º: Rafael Barea Navarro

Fecha:

Tabla de Contenido

Índice de Imágenes

Resumen

La **detección** y el **seguimiento** de los objetos presentes en un entorno son dos factores imprescindibles para el guiado de **vehículos** autónomos.

Con este trabajo se pretenden estudiar y aplicar diversas técnicas para detectar los elementos que rodean a un **vehículo** con el objetivo de poder determinar sus características y posicionarlos en el espacio. Tras la **detección** se realizará un **seguimiento** de cada uno de los objetos detectados que permitirá identificarlos en una secuencia de tiempo y predecir su comportamiento.

Se utilizará la información de las imágenes provenientes de las cámaras del vehículo y de las **nubes de puntos** que nos entregará el **LIDAR** situado en el mismo.

Abstract

Object **detection** and **tracking** are essential for autonomous and self-driving **vehicles**.

The objective of this paper is to study and apply different techniques and approaches intended to detect the elements surrounding a vehicle in order to determine their features and location in the space. Once the objects have been detected they will be tracked, which will allow for their identification in a sequence of time and to predict their behavior.

The main data sources that are going to be used are the images taken by the cameras placed in the vehicle and the point clouds provided by the vehicle LIDAR.

Palabras Clave

Detection, Tracking, LIDAR, Pointcloud, Vehicle.

Resumen Extendido

Introducción

La conducción autónoma es una tecnología que no ha parado de evolucionar durante los últimos años, desde coches que lograban moverse solo unos pocos metros en entornos controlados a vehículos que logran desplazarse distancias mucho mayores, controlando gran parte de su entorno y con un grado de libertad mucho mayor.

Los avances que supondría para nuestra sociedad el perfeccionamiento de esta tecnología son incalculables. Eliminar la limitación que supone el factor humano en el transporte actual nos permitiría aumentar tanto la seguridad y velocidad como el rendimiento de todos los transportes que se realizan a diario en nuestra sociedad.

Ligados directamente a la conducción autónoma se encuentran la detección y el seguimiento o *tracking* de los objetos móviles que rodean a los vehículos en cualquier entorno dinámico, puesto que es imprescindible que los vehículos autónomos tengan un conocimiento constante del entorno que les rodea para poder actuar en consecuencia.

Durante los últimos años han surgido numerosas propuestas que intentan abordar este problema y que han proporcionado diferentes resultados a la hora de detectar los objetos del entorno de un vehículo autónomo. Destaca entre estas propuestas el uso de redes neuronales *convolucionales*, que han logrado aportar resultados muy superiores a los obtenidos mediante técnicas puramente deterministas [1][2].

Es importante también mencionar que se han empleado una gran cantidad de fuentes de información diferentes para lograr la detección de los objetos, tales como LASER, RADAR, LIDAR, GPS e información de visión obtenida mediante cámaras tanto mono como estéreo.

En este trabajo se pretende investigar la efectividad de la detección de objetos utilizando la información proveniente de un LIDAR y una cámara. Los datos empleados durante todo el proyecto han sido obtenidos de la base de datos ***The KITTI Vision Benchmark Suite***[1] [3]. Esta base de datos contiene tanto datos de calibraciones de las medidas tomadas como imágenes y datos de nubes de puntos obtenidas mediante el LIDAR ***Velodyne HDL-64E*** de 64 haces. De igual manera la base de datos nos provee con una serie de archivos de *ground truth* que contienen la información real de los objetos que se encuentran en cada uno de los escenarios.

Empleando todos los datos proporcionados directamente mediante la base de datos y la fusión de los datos del LIDAR con la segmentación semántica de las imágenes realizada mediante la red convolucional ***ERFNet*** [4] se pretenden detectar tanto los vehículos como los peatones y ciclistas que se encuentren rodeando al vehículo.

Con el objetivo de poner todo esto en práctica se ha empleado el framework ***ROS*** (Robot Operating System) [5], una herramienta que nos permite controlar de manera eficiente las comunicaciones entre diferentes nodos que se ejecutan en paralelo, y se han programado en C++ varios programas que han permitido realizar los objetivos de este trabajo de fin de grado.

[1] Desde este punto se hará referencia a la base de datos como “base de datos de KITTI”.

Detección de objetos

El primer programa que se ha realizado es el módulo de detección de objetos, encargado de detectar todos los coches, peatones y ciclistas que se encuentran rodeando al vehículo en cada escenario para obtener sus principales características.

Para ello el programa empieza ejecutando una función de manera síncrona cada vez que tiene disponibles todos los datos necesarios para un mismo instante de tiempo, ya que el orden de procesado de algunos de los datos será de gran importancia en el proceso de ejecución.

Tras ello, el primer paso será leer las matrices de calibración de cada instante, ya que estas serán necesarias cada vez que se requiera trasladar puntos de las coordenadas del LIDAR a coordenadas de la cámara y viceversa. El siguiente paso será obtener el ángulo de la carretera mediante la información de la cámara y extraer los objetos que se puedan obtener empleando únicamente la información de la segmentación semántica[2] de la imagen obtenida mediante la red *convolucional*.

Finalmente se realizará el procesado en 3D mediante los datos del LIDAR y la fusión de los mismos con los datos de la red *convolucional*, lo que nos ayudará a determinar el tipo de objeto que estamos tratando y a filtrar sus puntos de los de la nube de puntos total.

Para extraer cada uno de los ***clusters***[3] de los objetos una vez filtrados por tipo de objeto se empleará el algoritmo ***Euclidean cluster extraction based on K-d tree*** [6] y finalmente se procesará cada objeto individualmente.

Al final del procesado se tendrá información sobre el tamaño y posición del objeto, de las coordenadas de todos sus vértices, su ángulo, oclusión, truncamiento y se calculará un parámetro denominado ***score*** que indicará el índice de seguridad que se tiene de que esa detección sea correcta.

Tracking

El ***tracking*** o seguimiento de los objetos se realizará tras la detección de los mismos. Para cada instante de tiempo se almacenará información de los objetos detectados y la posición que ocupan, y mediante el uso del filtro de Kalman se determinará la correspondencia del objeto actual con uno de los objetos detectados en instantes anteriores. El **filtro de Kalman** [8][9] proporcionará una predicción de la posición en la que los objetos detectados en cada instante se deberían encontrar en el siguiente instante de tiempo basándose en el seguimiento de su trayectoria.

Con esto conseguiremos determinar la trayectoria que ha seguido cada objeto durante el transcurso del tiempo en un mismo escenario y podremos tener unas predicciones más o menos precisas del comportamiento que van a tener estos objetos.

[2] La segmentación semántica permite diferenciar los objetos de una imagen otorgándole un color uniforme a cada objeto presente en la imagen.
[3] Término que determina un conjunto de puntos de la nube de puntos que conforman un mismo objeto.

1. Introducción

La detección y el tracking de objetos son dos aspectos de vital importancia para la conducción autónoma. El conocimiento de los objetos móviles que rodean a un vehículo autónomo le brinda la posibilidad de tener en cuenta el movimiento de estos a la hora de realizar sus desplazamientos y saber por qué zonas debe o no debe pasar o qué velocidades debe adoptar para evitar un posible choque.

Los métodos de detección de objetos que tienden a destacar hoy en día se basan en la fusión de datos de **LIDAR** y **cámara**, que se usan conjuntamente para mejorar los resultados obtenidos con una sola de las fuentes de información. Estas aproximaciones usan en la mayoría de los casos redes convolucionales que realizan una segmentación semántica [1] para determinar el tipo de objeto con el que están tratando. La segmentación semántica se aplica por lo general a los datos obtenidos mediante la cámara, aunque existen algunos casos en los que ha sido aplicada directamente sobre datos de LIDAR[10][11].

Una vez se conoce el tipo de objeto se obtienen las características del mismo basándose en los datos proporcionados por el LIDAR. En algunos casos se emplean también redes convolucionales para determinar las características del objeto trabajando con la nube de puntos que proporciona el LIDAR.

El factor diferencial de este trabajo es la **fusión directa** de los datos de **LIDAR** y **cámara**. Tras realizar la **segmentación semántica** de los datos obtenidos de la cámara se **proyectan** todos los datos de la nube de puntos sobre esta imagen y se colorean con el color correspondiente al objeto al que pertenecen, de esta manera al trabajar con la nube de puntos podemos filtrar todos y cada uno de los puntos de la nube por el tipo de objeto al que pertenecen, lo que facilitará en gran medida el proceso de obtención de *clusters*, que será realizado basándonos en el algoritmo ***Euclidean cluster extraction based on K-d tree***.

Uno de los principales problemas que encontraremos a la hora de usar *k-d tree* para realizar la extracción de *clusters* en vez de usar directamente una red convolucional que extraiga todos los *clusters* será determinar el ángulo en el que se encuentra el objeto. Para ello se empleará la ***transformada de Hough*** sobre la nube de puntos del clúster proyectada sobre el suelo.

Finalmente, al igual que en otras muchas aproximaciones de este problema se realizarán una serie de operaciones de **post-procesado** con la intención de mejorar y pulir los datos que se han obtenido en las primeras fases de la detección.

El objetivo final del trabajo será conseguir la mayor precisión posible en cuanto a las detecciones realizadas y también conseguir la mayor cantidad de detecciones evitando una gran cantidad de falsos positivos. Para evaluar los resultados se usarán los tests proporcionados junto a la base de datos de KITTI, que nos darán una idea de la precisión[4] y el "recall"[5] del algoritmo de detección diseñado.

Será necesario generar un fichero de resultados cuya estructura completa será tratada más adelante y que contendrá la información de todos los objetos detectados para poder realizad la

[4] Hace referencia a la calidad de las detecciones que se han realizado.
[5] Indica la cantidad de objetos que se han detectado sobre el total de objetos en un escenario.

comparación con los ficheros de "ground truth"[6] proporcionados junto a la base de datos de KITTI.

Trasladándonos a la parte de ***tracking*** o **seguimiento** de objetos [13][14][15] el objetivo será parecido, realizar el seguimiento de los objetos presentes en las secuencias proporcionadas por la base de datos **KITTI** y comprobar la precisión con los *tests* que nos proporciona la misma base de datos.

El seguimiento de los objetos estará estrechamente ligado a la detección de los mismos puesto que nos basaremos en los datos de las detecciones para determinar las trayectorias que siguen los objetos a lo largo del tiempo.

La aproximación elegida para realizar el seguimiento está basada en el ***filtro de Kalman***. Será necesario etiquetar cada uno de los objetos detectados en cada instante de tiempo e intentar encontrar su correspondencia con uno de los objetos detectados en un instante anterior. Para ello se usarán las predicciones proporcionadas con el filtro de Kalman, que nos darán una idea de donde debería encontrarse cada objeto en el siguiente instante de tiempo. Al igual que en la detección de objetos será necesario realizar un **post-procesado** de los datos para pulir los resultados obtenidos, ya que en algunos casos los falsos positivos podrían alterar la verdadera trayectoria de un objeto.

Para comparar estos datos con los de la base de datos se generará un fichero cuya estructura será abordada en puntos posteriores y que contendrá los objetos detectados en cada instante de tiempo y un identificador de objeto.

Adicionalmente se emplearán varios programas escritos en Matlab que ayudarán a determinar la precisión que tenemos a la hora de realizar las detecciones y a observar que clase de errores estamos cometiendo a la hora de generar los datos de forma que sea más sencillo corregir los fallos que tenga el algoritmo.

Una vez se haya completado el programa de detección de objetos y se hayan obtenido los resultados se tratará de mejorar los mismos mediante el empleo de **Mask R-CNN** [18] y cambiando ligeramente la estructura del programa.

Como se podrá observar en los próximos apartados en muchas ocasiones se ha cambiado la aproximación inicial que se había realizado para resolver algún problema encontrado debido a que otras soluciones proporcionaban mejores resultados o un mejor rendimiento de los algoritmos empleados.

Tras cumplir los objetivos anteriormente mencionados se documentarán debidamente las tareas realizadas y se propondrán posibles mejoras futuras que podrían hacer que el rendimiento de los algoritmos y la precisión de las detecciones aumentasen.

[6] Ficheros que contienen la información real de los objetos presentes en un escenario.

2. Objetivos del Proyecto

En este apartado se explicarán con mayor detalle los objetivos del proyecto exponiendo a modo de resumen los pasos que se seguirán para alcanzar los objetivos finales del mismo.

Los tres objetivos **principales** serán los siguientes:

- Generar un programa capaz de detectar los objetos presentes en un escenario empleando la información en forma de imágenes provenientes de una cámara y las nubes de puntos de un LIDAR.
- Crear un segundo programa basado en el primero que consiga realizar un seguimiento de los objetos detectados en una secuencia de tiempo para determinar su trayectoria.
- Obtener los resultados de la precisión de las detecciones y del seguimiento que se ha obtenido mediante los dos programas mencionados anteriormente.

2.1. Pasos Para Completar los Objetivos y Objetivos Secundarios

Para la realización de ambos programas se emplearán los datos obtenidos del *dataset* de ***KITTI.*** Los datos obtenidos de este *dataset* serán explicados con mayor detalle en apartados posteriores, pero se usarán principalmente las imágenes de las cámaras y las nubes de puntos del LIDAR como se explicó con anterioridad.

Para alcanzar los objetivos principales será necesario realizar diversas **transformaciones** sobre los datos de entrada que nos permitirán finalmente llegar a detectar los objetos y a realizar un seguimiento de los mismos. Estas transformaciones serán las siguientes:

- **Segmentación semántica de las imágenes**: Se empleará la red *convolucional* ***ERFNet*** para obtener una imagen segmentada según el tipo de objeto que estamos detectando en todo momento.
- **Fusión de datos de cámara y LIDAR**: La nube de puntos obtenida mediante el LIDAR se combinará con la imagen segmentada semánticamente para conocer el tipo de objeto al que pertenece cada punto de la nube de puntos.

Una vez conseguidos los datos de entrada necesarios para nuestro programa el siguiente objetivo será realizar detecciones de objetos tanto directamente sobre las imágenes de la cámara como sobre la nube de puntos.

Estas detecciones se combinarán proyectando los datos de la imagen de la cámara sobre la nube de puntos y viceversa. Finalmente, tras haber eliminado las detecciones que sean consideradas incorrectas, se obtendrán los datos finales de las detecciones tanto en 3D como en 2D.

Para cada detección se obtendrán los siguientes datos para cada sistema de coordenadas:

- **Coordenadas 2D:** Punto superior izquierdo e inferior derecho en píxeles de la caja mínima que envuelve al objeto y tipo de objeto.
- **Coordenadas 3D:** Posición en el espacio, tamaño del objeto, tipo de objeto y ángulo en el que se encuentra.

Finalmente, tras haber generado el programa de detección de objetos y el programa de tracking, el objetivo final será obtener los resultados de las detecciones y el seguimiento de los objetos. Para ello se emplearán los test de evaluación proporcionados nuevamente por **KITTI**, que nos permitirán hacernos una idea de la calidad de los resultados que se han obtenido.

Adicionalmente, se tratará de mejorar los resultados mediante el empleo de **Mask R-CNN** cambiando ligeramente la estructura del programa expuesta en este apartado. Será en el apartado ***7. Mejora de Resultados Empleando Mask R-CNN*** donde se explicarán los cambios que se han realizado sobre la estructura del programa con el objetivo de mejorar los resultados y donde se compararán los resultados obtenidos mediante uno u otro método.

3. Estado del Arte

A día de hoy se pueden encontrar una gran variedad de aproximaciones tratando de obtener los mejores resultados posibles en cuanto a la detección y el tracking de objetos aplicado a vehículos inteligentes.

Muchas de estas aproximaciones pueden verse expuestas en la página web de **KITTI** [3], en la que está centrada la realización de este trabajo, y será por tanto en ellas en las que estará basado este apartado.

Para entender el contexto actual en el que se encuentra el ámbito de la detección y tracking de objetos aquí se expondrán de manera breve algunos de los trabajos que actualmente han sido capaces de obtener algunos de los mejores resultados empleando los tests y el dataset proporcionados por **KITTI**.

3.1. F-PointNet

Título del trabajo: Frustum PointNets for 3D Object Detection from RGB-D Data [16].

Esta primera aproximación emplea tres redes neuronales que le sirven para obtener en una última instancia las cajas mínimas de los objetos que se encuentran en el escenario.

La estructura básica del sistema que emplean se puede observar en la siguiente imagen:

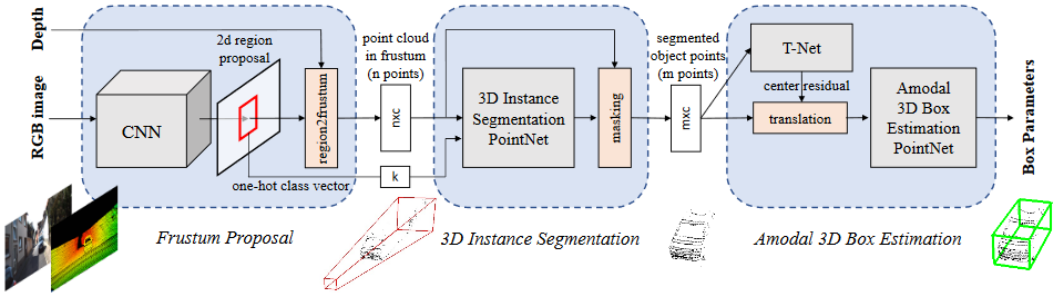


Ilustración 1 - Frustum PointNets for 3D object detection.[7]

El proceso seguido para realizar las detecciones es el siguiente:

1. En una primera fase se realiza una primera detección de objetos sobre la imagen original mediante una CNN 2D que tiene en cuenta tanto los datos RGB de la imagen como la profundidad de esos mismos puntos detectada con el LIDAR.
2. Esa región en 2D de la imagen se traslada al 3D, obteniendo un ***frustum*** en la nube de puntos que corresponde a toda la región de la nube de puntos que queda cubierta por esa zona de la imagen en 2D.
3. La segunda red se encarga de separar los diferentes ***cluster*** de puntos de cada uno de los coches presentes en el ***frustum,*** que servirán de entrada para el último paso.

[7] Esta imagen ha sido extraída del *paper* de este mismo trabajo [16].

4. Finalmente, la última red recibirá como entrada los *clústers* y devolverá las propuestas de las cajas mínimas que corresponden a cada uno de los *clústers*.

La ventaja de emplear una red para separar los diferentes *clústers* de la imagen de todos los puntos obtenidos en el *frustum* es la eliminación de todas las partes del mismo que no forman parte del propio objeto que se está analizando y que llevan a producir una gran cantidad de errores, como se expone en el propio *paper* del trabajo.

3.2. Deep MANTA

Título del trabajo: A Coarse-to-fine Many-Task Network for joint 2D and 3D vehicle analysis from monocular image [17]**.**

A diferencia de otras aproximaciones este trabajo solo emplea la información en 2D de la cámara para realizar las detecciones, y posteriormente, mediante una red neuronal y una serie de modelos de vehículo preestablecidos se reconstruye la información de las detecciones en 3D.

Un esquema del sistema empleado se puede observar en la siguiente imagen:

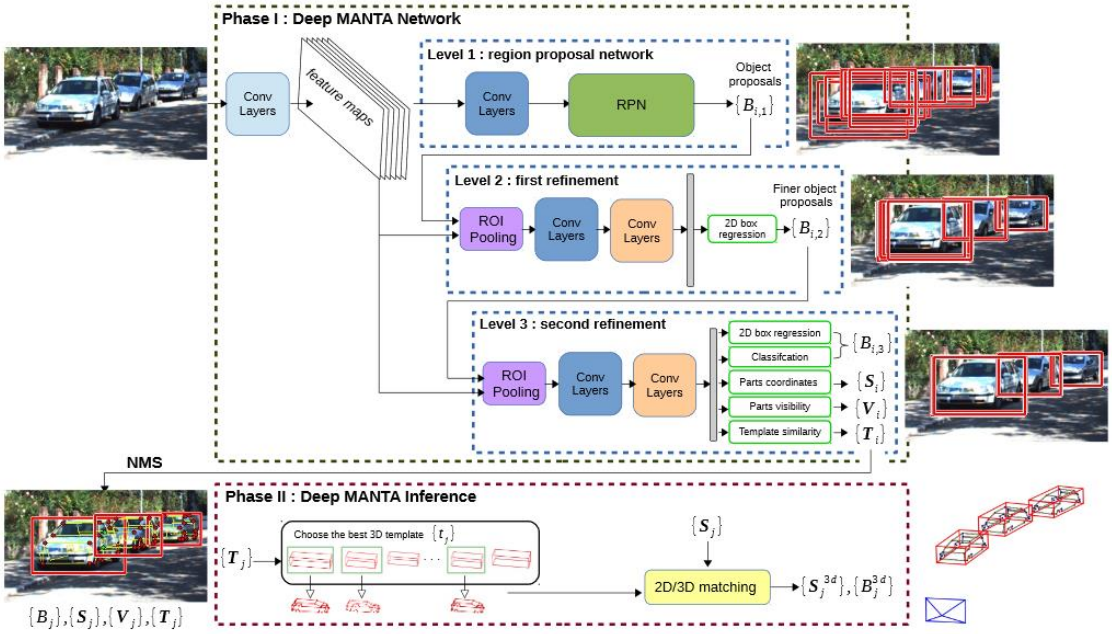


Ilustración 2 - Esquema de la aproximación de Deep MANTA [8].

En un primer paso se realizan las diferentes propuestas de los objetos detectados por la CNN se refinan hasta obtener las propuestas finales.

Tras ello, se realiza un ***template matching*** asignando a cada uno de los coches detectados en 2D el modelo 2D que mejor se le adecúe.

[8] Esta imagen ha sido extraída del *paper* de este mismo trabajo [17].

Finalmente, la segunda red (Deep MANTA Inference) se encarga de asignar al modelo 2D propuesto el modelo 3D que mejor se le aproxime y, empleando un algoritmo de estimación, escala el modelo 3D, lo sitúa en la posición 3D estimada y determina su caja mínima.

Al no emplear información de la nube de puntos este modelo no destaca por su precisión en las detecciones en 3D, sin embargo, si que lo hace por su precisión en detecciones 2D.

3.3. AVOD-FPN

Título del trabajo: Joint 3D Proposal Generation and Object Detection from View Aggregation [2].

Este trabajo emplea tanto datos de la cámara como de la nube de puntos obtenida mediante el LIDAR. A diferencia de otras aproximaciones este trabajo se centra en el uso de los datos de la nube de puntos proyectados sobre el suelo (***Bird eye view***) para la obtención de las detecciones en 3D en vez de trabajar directamente con la nube de puntos.

El esquema de funcionamiento básico de esta aproximación es el siguiente:

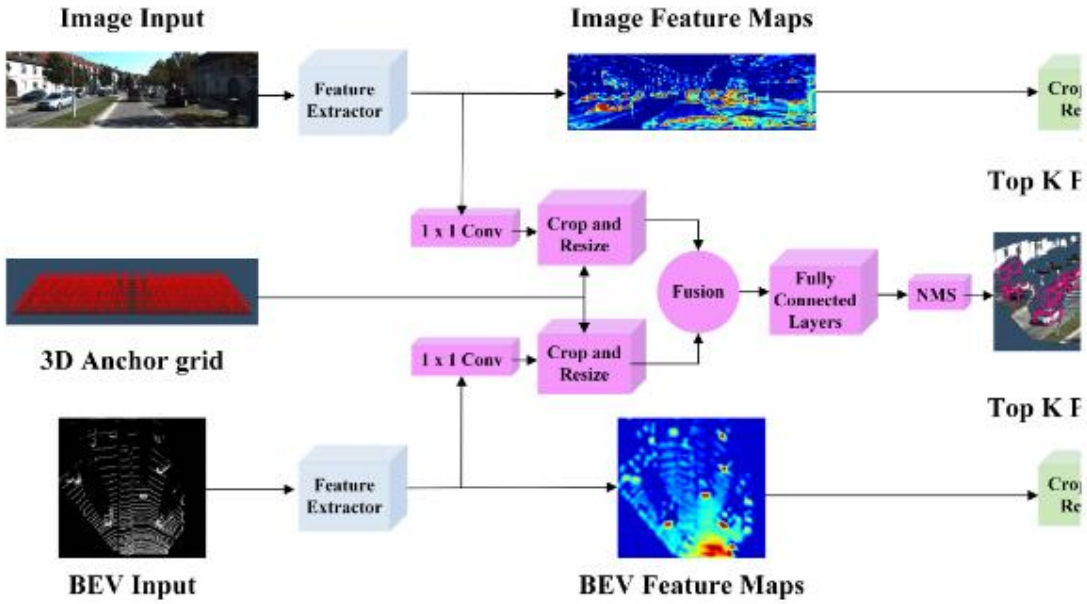


Ilustración 3 - Esquema de la aproximación de AVOD [9].

El primer paso que se lleva a cabo es la extracción de las características tanto de la imagen como de la nube de puntos del LIDAR. Estas características corresponden principalmente con información de color e información de profundidad y altura de los puntos de la nube de puntos.

Tras ello, estas nuevas "imágenes" que contienen la información de las características extraídas son procesadas por la CNN obteniendo una serie de propuestas de las cajas mínimas de los objetos que son finalmente refinadas hasta obtener las propuestas finales de las detecciones.

[9] Esta imagen ha sido extraída del *paper* de este mismo trabajo [2].

Actualmente este es el trabajo que ha logrado obtener los mejores resultados con la base de datos de **KITTI** y que ha publicado información sobre los métodos empleados mediante la publicación de un *paper* y la exposición del código del proyecto en su repositorio público de *github* [10].

[10] Enlace al repositorio donde se puede obtener el código y más información del proyecto: https://github.com/kujason/avod.

4. KITTI Dataset

Como ya se ha mencionado con anterioridad, para el desarrollo de este trabajo se han empleado los datos obtenidos del ***dataset* de KITTI** [3]. En este apartado se explicará con detalle el formato de los datos que hemos usado, el sistema empleado para obtenerlos y se pondrán ejemplos de cada uno de ellos.

El *dataset* contiene distintas bases de datos orientadas a diferentes objetivos, las que se emplearán en este trabajo son la base de datos ***Kitti_dataset_object***, para la detección de objetos en la primera parte del trabajo, y la base de datos ***Kitti_dataset_tracking***, que se empleará en el apartado de seguimiento de los objetos.

Cada una de ellas se explicará por separado en los siguientes subapartados de este punto, pero primero se comenzará explicando el sistema que se ha empleado para obtener los datos.

4.1. Sistema de Sensores

El sistema empleado para obtener los datos de la base de datos de **KITTI** será de vital importancia en el trabajo, ya que determina el tipo de datos que tenemos y en muchos casos habrá que tener en cuenta también el sistema de coordenadas empleado para poder trabajar con sus datos de manera correcta. Por ello se dedicarán las siguientes líneas a explicar los sensores utilizados y los sistemas de coordenadas empleados en la base de datos.

El sistema está compuesto por un **GPS** (***OXTS RT 3003***), un **LIDAR** (***Velodyne HDL-64E***), dos **cámaras** en escala de grises (***Point Grey Flea 2 (FL2-14S3M-C)***), dos **cámaras** a color (***Point Grey Flea 2 (FL2-14S3C-C)***) y 4 **lentes varifocales**. Los sensores de nuestro interés serán principalmente el **LIDAR** y una de las **cámaras** a color.

Todos estos sensores están distribuidos en el coche según se indica en la siguiente imagen obtenida de la página web del *dataset*[11]:

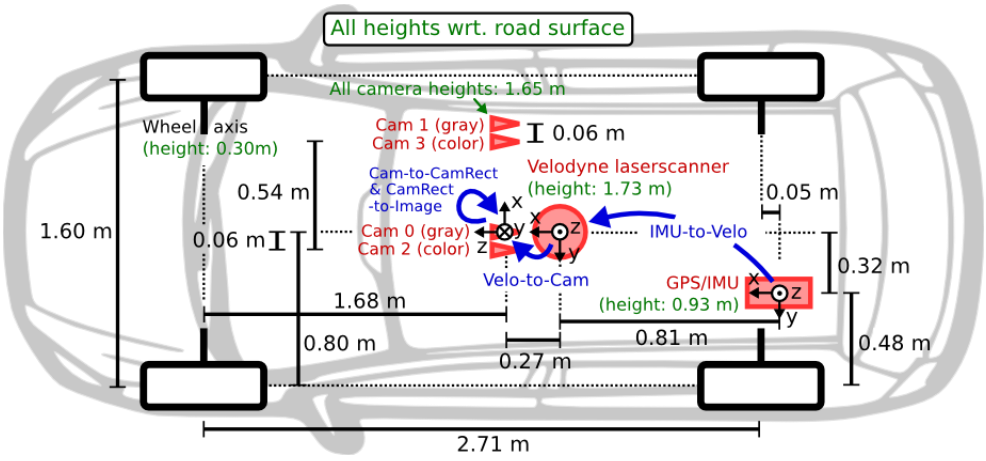


Ilustración 4 - Posición de los sensores de KITTI.

En la imagen se aprecia la posición en el vehículo de cada uno de los sensores anteriormente mencionados. A su vez es necesario mencionar los dos sistemas de coordenadas empleados

[11] http://www.cvlibs.net/datasets/kitti/setup.php

por KITTI. Ambos **ejes de coordenadas** se pueden encontrar en la siguiente imagen también obtenida en la página web del *dataset* de **KITTI**:

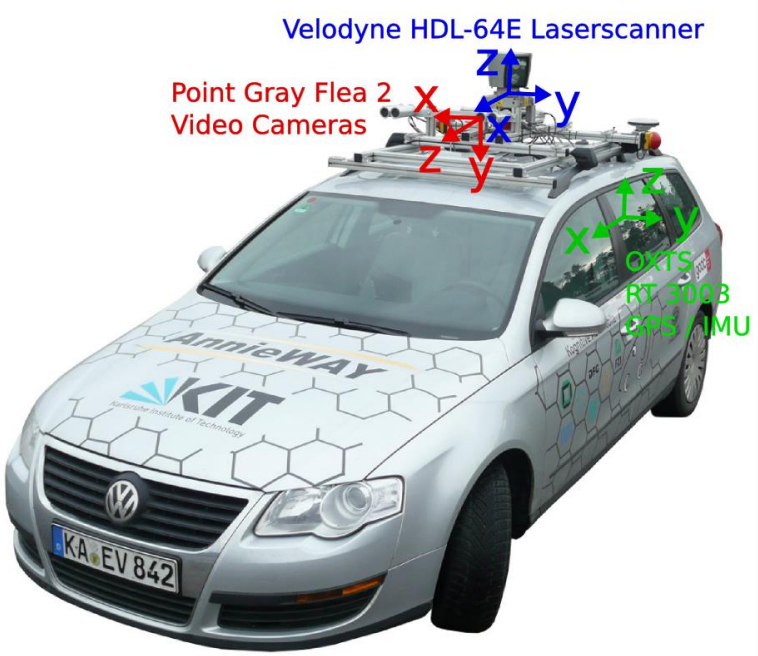


Ilustración 5 - Sistema de coordenadas de KITTI

El eje de coordenadas de color azul corresponde con el usado en las nubes de puntos del LIDAR, mientras que el de color rojo corresponde con el usado en las imágenes de la cámara. En algunas ocasiones los datos proporcionados por la base de datos se encontrarán en coordenadas de la cámara a pesar de tratarse de datos en 3D obtenidos mediante el LIDAR. Esto sucederá por ejemplo en los ficheros de ***ground truth*** y habrá que tener especial cuidado con ello.

4.2. Object Dataset

Una vez aclarados los sistemas de coordenadas se continuará explicando los datos que nos proporciona el *dataset* empleado para realizar el programa de detección de objetos.

Esta base de datos de objetos (***Kitti_dataset_object***) está formada por un conjunto de **7481** muestras independientes de los que se proporcionan datos como **imágenes**, **nubes de puntos** y ficheros de ***ground truth*** entre otros.

A su vez, los datos están divididos entre datos de ***testing*** y de ***training***. Para nuestras comprobaciones usaremos principalmente los datos de ***training***.

La siguiente imagen muestra la estructura de carpetas de la base de datos que contendrán los diversos datos que serán necesarios para la detección de los objetos:

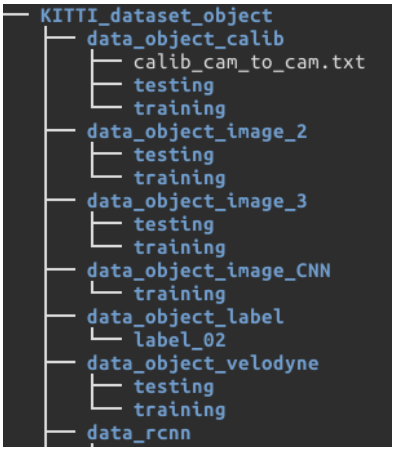


Ilustración 6 - Estructura Base de Datos

A continuación, se explica la lista completa de los datos contenidos en la base de datos así como el formato de cada uno de ellos, su utilidad y la carpeta de la estructura de carpetas donde están contenidos:

1. **Matrices de calibración**: En la carpeta ***data_object_calib***. Existe un fichero de calibración por cada uno de los escenarios contenidos en la base de datos. El fichero de matrices de calibración contiene la información necesaria para realizar los ajustes requeridos a la hora de proyectar datos del LIDAR a la cámara y viceversa. Está compuesto por 7 matrices y tiene el aspecto presentado en la siguiente imagen:

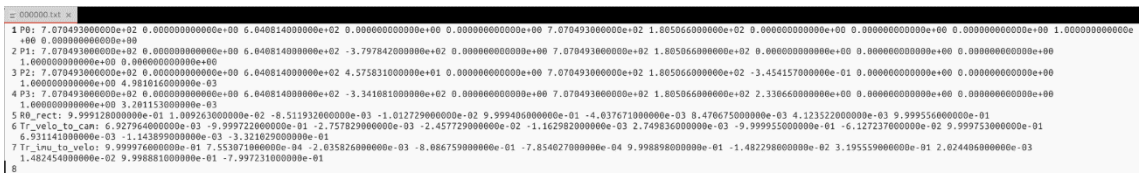


Ilustración 7 - Matrices de Calibración

Para realizar las proyecciones de LIDAR a cámara y viceversa se emplearán las siguientes ecuaciones que usan como datos las matrices mostradas en la imagen anterior:

- Cámara a LIDAR:

$$\text{Punto\_Proyectado} = (\text{Matriz}_\text{R} \times \text{Matriz}_\text{P})^{-1} \times \frac{Altura}{Factor_{Escala}} \times (Matriz_P)^{-1} \times Punto\_A\_Proyectar$$

Siendo el ***Factor_Escala*** el valor obtenido en la posición (1,0) de la matriz obtenida al multiplicar la inversa de la ***Matriz_P*** por el punto a proyectar.

- LIDAR a cámara:

$$Punto\_Proyectado = Matriz\_P \times Matriz\_R \times Matriz\_T \times Punto\_A\_Proyectar$$

2. **Imágenes de la cámara**: En la carpeta ***data_object_image_2.*** La base de datos proporciona la información de dos cámaras situadas a una distancia de 48cm y que toman imágenes del escenario que se encuentra delante del vehículo. Existe una imagen

a color con una resolución de 1224x370 píxeles para cada instante de tiempo y por cada una de las cámaras. Un ejemplo de estas imágenes es el que se muestra a continuación:

Ilustración 8 - Imagen de la cámara a color

3. **Ficheros de *Ground truth*:** En la carpeta ***data_object_label.*** Estos ficheros contienen la información "real" de los objetos que se encuentran en cada uno de los escenarios. Han sido generados de manera manual estableciendo las posiciones y características de cada uno de los objetos para poder comparar los resultados obtenidos con los reales. Nuevamente existe un fichero de texto por cada uno de los escenarios que contiene una línea por cada uno de los objetos presentes en el escenario. La información contenida por cada objeto es la siguiente:
    a. ***Type***: Describe el tipo de objeto del que se trata, puede ser uno de la siguiente lista: "Car", "Truck", "Van", "Cyclist", "Pedestrian", "Person_sitting", "Tram", "Misc" o "DontCare".
    b. ***Truncated***: Indica el grado de truncamiento del objeto. Un objeto estará truncado si se sale de los límites de la imagen captada por la cámara, lo que suele suceder con los coches que se encuentran muy cerca. Se trata de un dato tipo float entre 0 (No truncado) y 1 (Completamente truncado).
    c. ***Occluded***: Se trata de un dato de tipo *int* entre 0 y 3 indicando el estado de oclusión de un objeto. Un objeto se encuentra ocluido cuando existen objetos que lo ocultan parcialmente.
    d. ***Alpha***: Ángulo de observación del objeto entre "pi" y "-pi". Este ángulo se calcula mediante la siguiente fórmula matemática: Alpha = -[(pi + ry) + (pi + beta)], donde ry es el ángulo de giro del coche y beta es el ángulo en el que se encuentra respecto a la posición del origen de coordenadas en el espacio.
    e. ***Bbox***: La ***Bounding box*** que contiene al objeto en coordenadas de la imagen. Se trata de cuatro parámetros que contienen las coordenadas del píxel que está más a la izquierda, más arriba, más a la derecha y más abajo del objeto respectivamente. Mediante ellos se puede reconstruir la *bounding box* en la imagen.
    f. ***Dimensions***: Se trata de 3 datos que indican la altura, anchura y longitud del objeto en metros respectivamente.
    g. ***Location***: Nuevamente 3 parámetros que indican la posición del centroide del objeto en coordenadas de la cámara y en metros.
    h. ***Rotation_y***: La rotación del objeto respecto al eje Y en coordenadas de la cámara.

i. ***Score***: Se trata de un dato tipo *float* que servirá para indicar la seguridad que se tiene a la hora de haber realizado una detección.

Un ejemplo del formato contenido en este fichero se muestra en la siguiente imagen:

readme.txt × 000025.txt ×

```
1 Car 0.94 3 -2.10 896.11 218.17 1241.00 374.00 1.39 1.44 3.08 2.43 1.68 3.14 -1.49
2 Car 0.00 0 -1.29 351.84 183.19 537.77 308.64 1.47 1.60 3.66 -2.21 1.63 10.42 -1.49
3 Car 0.00 0 1.75 562.48 173.46 618.49 217.36 1.70 1.63 4.08 -0.78 1.75 30.18 1.72
4 Car 0.00 0 -1.69 724.21 178.91 805.39 249.94 1.59 1.59 2.47 3.64 1.75 17.48 -1.49
5 Cyclist 0.00 3 1.60 650.19 175.43 659.99 205.20 1.81 0.55 1.19 2.77 1.98 44.63 1.66
6 Car 0.00 1 -1.62 720.81 187.01 779.98 236.22 1.37 1.59 3.22 4.23 1.83 22.30 -1.44
7 DontCare -1 -1 -10 9.81 201.33 302.24 362.08 -1 -1 -1 -1000 -1000 -1000 -10
8 DontCare -1 -1 -10 683.20 175.57 706.41 202.24 -1 -1 -1 -1000 -1000 -1000 -10
9 DontCare -1 -1 -10 629.78 168.24 647.36 191.59 -1 -1 -1 -1000 -1000 -1000 -10
```

Ilustración 9 - Fichero de Ground Truth

4. **Datos del LIDAR**: En la carpeta ***data_object_velodyne.*** Cada uno de los ficheros contiene información de la posición de todos y cada uno de los puntos que forman la nube de puntos de un escenario. Al igual que con el resto de datos, existe un fichero para cada uno de los instantes de tiempo y se trata de ficheros binarios que contienen la información de uno de los puntos de la nube en cada una de sus líneas y tienen el siguiente aspecto una vez abiertos con un editor de texto:

```
1  7dbf 9342 c520 303d 7d3f 2d40 0000 0000
2  d7a3 9342 cdcc 8c3e b81e 2d40 0000 0000
3  852b 9342 ae47 013f a69b 2c40 0000 0000
4  ac5c 9342 08ac 3c3f 2fdd 2c40 0000 0000
5  1b6f 9342 be9f 5a3f 91ed 2c40 0000 0000
6  df4f 9342 83c0 8a3f cdcc 2c40 0000 0000
7  aa31 9342 ec51 a83f 08ac 2c40 0000 0000
8  9eef 9342 79e9 c63f 0681 2d40 0000 0000
9  cd4c 9442 7d3f e53f 54e3 2d40 0000 0000
10 a6db 9342 e926 0140 a470 2d40 0000 0000
```

Ilustración 10 - Datos del LIDAR

Los puntos visualizados con la herramienta ***RVIZ*** tienen el siguiente aspecto (sin ninguna información de color[12]):

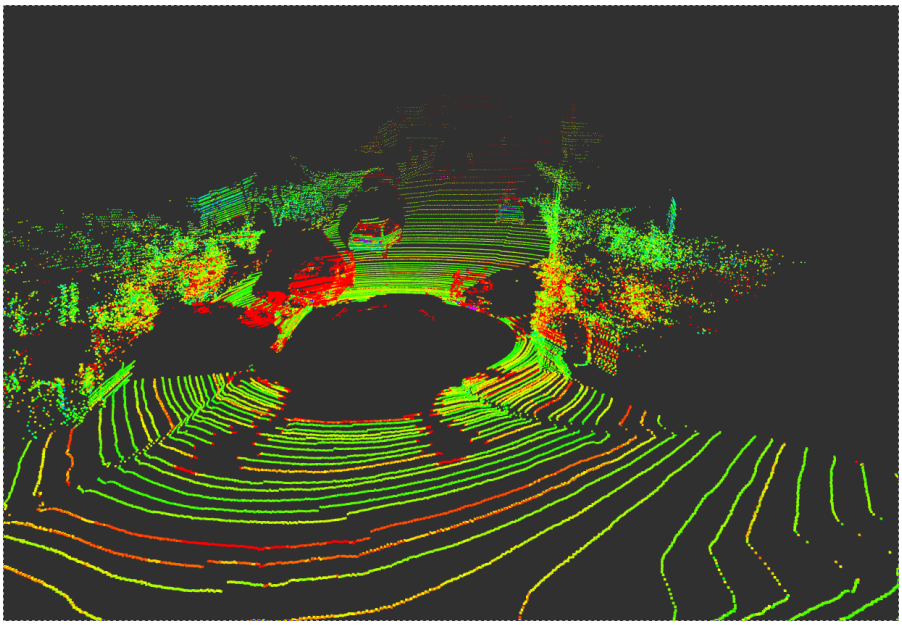


Ilustración 11 - Datos del LIDAR RVIZ

Finalmente, el *dataset* nos proporcionará un programa de evaluación que se encargará de comparar los datos que hemos obtenido con los almacenados en los ficheros de ***ground truth*** y nos proporcionará unos resultados indicando la precisión de nuestras detecciones en forma de gráficas y porcentajes.[13]

4.3 Tracking Dataset

En este caso se trabajará con la base de datos de tracking proporcionada por KITTI (***Kitti_dataset_tracking***).

Esta segunda base de datos emplea un formato de datos muy parecido al empleado en la base de datos de objetos con algunas leves diferencias que serán puntualizadas en este apartado.

La base de datos de objetos presentaba un conjunto de datos procedentes de **7481** instantes de tiempo independientes entre ellos, mientras que esta base de datos está formada por 21 secuencias de datos de ***training*** y 29 de ***test.*** Los datos que emplearemos serán principalmente las secuencias de *training*. Cada una de estas secuencias contendrá los datos de varios instantes de tiempo.

Los datos de esta segunda base de datos son correlativos y por tanto nos permitirán realizar el seguimiento de los objetos a lo largo del desarrollo de las secuencias, cosa que no era posible

[12] Todos los puntos representados en la imagen carecen de color a pesar de la interpretación de color que les ha dado la herramienta RVIZ.

[13] El proceso de evaluación será explicado con mayor detalle cuando se obtengan los resultados del programa realizado en apartados posteriores y en el apartado 9. Manual de Usuario.

empleando la base de datos de objetos, ya que cada instante de tiempo era independiente en ese caso.

Los datos que tendremos serán **nubes de puntos** obtenidas por el LIDAR, **imágenes** de la cámara, **matrices de calibración** y ficheros de **ground truth**.

Todos los datos estarán en el mismo formato que en el caso explicado con anterioridad a excepción de los ficheros de **ground truth**.

El formato de cada uno de los objetos detectados en estos ficheros será igual al de los que se usaban en el test anterior, con la diferencia de que se añadirá un nuevo dato de tipo entero que indicará el número de secuencia del escenario en el que se ha realizado esa detección y otro dato de tipo entero que identificará al objeto cada vez que detectemos que se trata del mismo objeto y será único para cada objeto detectado.

Otra diferencia será que en este caso solo existirá un fichero de datos de ***ground truth*** que contendrá la información de todos los escenarios que formen parte de cada una de las secuencias, a diferencia del caso anterior donde existía un fichero diferente para cada uno de los instantes de tiempo.

De igual manera que en el caso anterior, esta base de datos nos proporcionará un nuevo programa de evaluación que nos permitirá comparar los resultados que hemos obtenido con los ficheros de ***ground truth*** y nos mostrará diversos parámetros indicando la precisión del tracking que estamos realizando.

5. Detección de Objetos

El principal objetivo del trabajo ha sido realizar un sistema de detección de objetos. En esta sección se explicarán detalladamente los datos de entrada que son empleados por la aplicación para realizar las detecciones, todos y cada uno de los pasos que se siguen para realizar la detección de los objetos y los datos de salida obtenidos una vez que la aplicación ha procesado la información de un instante de tiempo.

También se abordarán algunos de los cambios realizados durante el desarrollo del programa explicando los beneficios de estos cambios de aproximación y cómo se llevaron a cabo.

5.1. Datos de entrada (KITTI)

Los datos de entrada que se emplearán en esta parte del trabajo serán los procedentes de la base de datos ***Kitti_dataset_object*** proporcionada por **KITTI**. El formato de estos datos ya ha sido explicado en el apartado **4.2 Object Dataset.** Tal y como se mencionó en ese apartado esta base de datos consta de una serie de datos procedentes de instantes de tiempo independientes que serán empleados para realizar la detección de los objetos.

Los datos presentes en esa base de datos que se han empleado son los siguientes:

- **Datos del LIDAR**: Proporcionarán información en 3D del entorno que rodea al vehículo, de tal forma que podrán ser usadas para obtener las características de los objetos presentes en el escenario.
- **Matrices de calibración**: Proporcionarán la información necesaria de las posiciones relativas de los sensores que nos permitirá realizar proyecciones de los datos de la cámara a datos del LIDAR y viceversa.
- **Imágenes de la cámara**: Serán usadas para obtener la segmentación semántica de las mismas y determinar el tipo de objeto que estamos procesando.
- **Ficheros de Ground Truth**: Contendrán la información real de los objetos presentes en cada escenario y permitirán realizar la evaluación comparando nuestros datos obtenidos con estos datos.

Todos estos datos han sido explicados con mayor detalle en el apartado **4. KITTI Dataset**.

5.2. Requisitos Previos y Arquitectura del Sistema

El esquema que se muestra en la siguiente imagen representa los pasos y transformaciones que se llevarán a cabo sobre las dos fuentes de datos principales que se emplearán para detectar los objetos (Imágenes de la cámara y PointClouds del LIDAR):

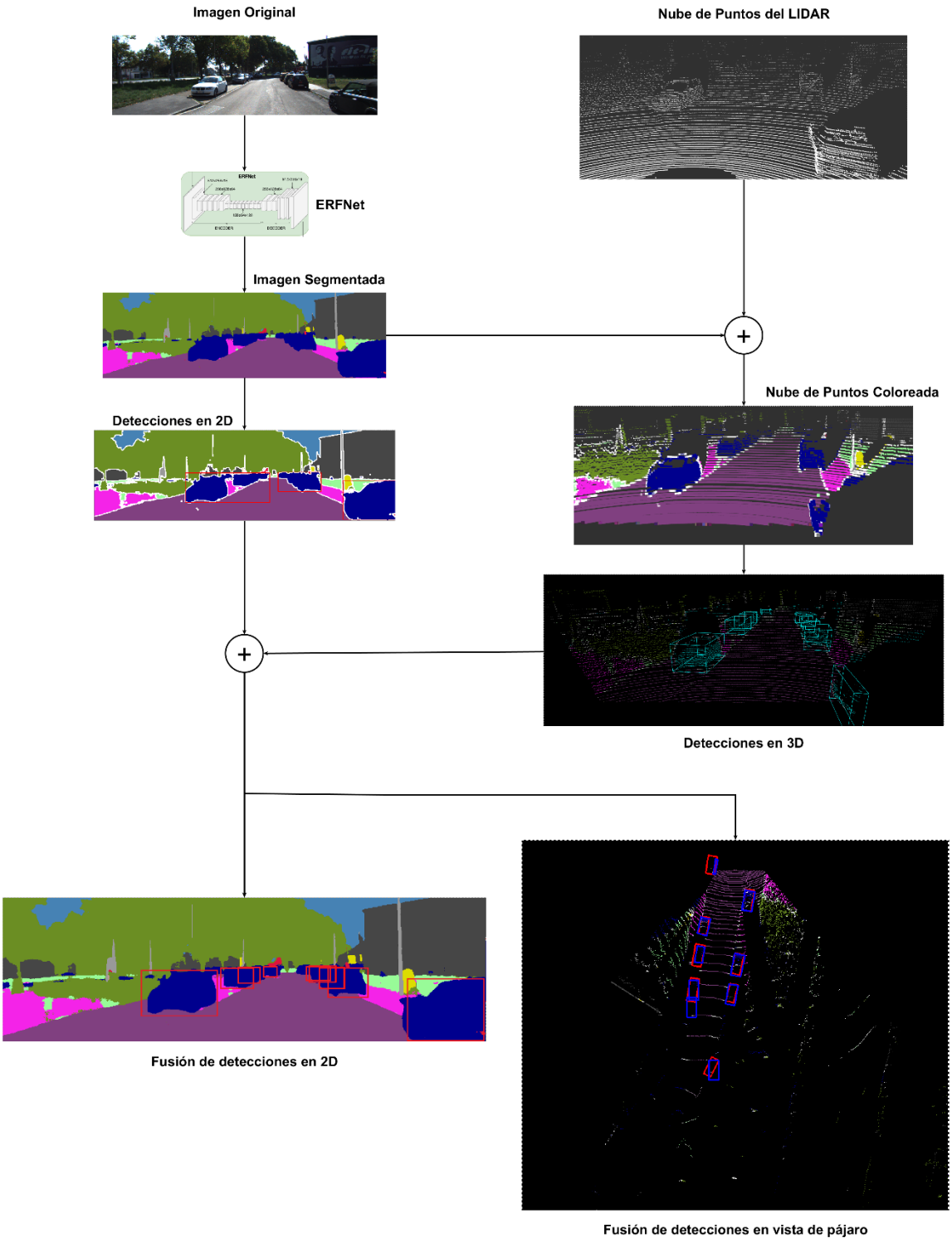


Ilustración 12 - Esquema de funcionamiento del programa.

En el caso del programa destinado al seguimiento de los objetos se aplicará el mismo esquema. Sin embargo, en este segundo programa se realizará un seguimiento de los distintos objetos que se han detectado en varios instantes de tiempo diferentes en vez de limitarse a obtener los datos de los objetos detectados.

5.2.3. ROS

Como ya se comentó con anterioridad, para simplificar la comunicación entre los diversos módulos necesarios para la ejecución del sistema se empleará el *framework* ***ROS***, que nos permitirá publicar datos en una serie de ***topics*** que podrán ser leídos en tiempo real por cualquier otro módulo que se suscriba a ellos.

Estos ***topics*** son canales de comunicación que usan el protocolo ***TCPROS*** basado en el protocolo **TCP/IP** para comunicar sus datos al resto de módulos.

El formato de mensaje empleado en cada *topic* será definido en un fichero con extensión ".msg" que se incluirá en el programa que lo vaya a usar y en los ficheros de compilación. Existen también una serie de tipos de mensaje genéricos que no será necesario crear a mano, como por ejemplo las nubes de puntos, que serán del tipo "*sensor_msgs::PointCLoud2*", o las imágenes que se publicarán con un mensaje del tipo "*sensor_msgs::Image*". En cualquier otro caso, como por ejemplo en el de las matrices de calibración que tenemos que publicar, será necesario crear un archivo para generar mensajes con una estructura personalizada.

Un ejemplo mostrando la estructura de los mensajes que publican las matrices sería el siguiente:

matrices.msg

```
1 Header header
2 float64[12] T
3 float64[9] R
4 float64[12] P
5 int64 S
6
```

Ilustración 13 - Mensaje ROS

Una vez tenemos definido el formato de los mensajes éstos podrán ser publicados en un *topic* mediante el método ***publish()*** y leídos por otro módulo suscrito al mismo *topic* en el que se publican.

En el caso de nuestro programa necesitaremos los siguientes datos:

- Imagen de la cámara segmentada semánticamente por la red convolucional.
- Imagen original de la cámara.
- Matrices de calibración.
- Nube de puntos del LIDAR coloreada.
- Nube de puntos total sin colorear.

Para obtener esos datos será necesario que estemos suscritos a cada uno de los *topics* en los que se publican y que lancemos previamente los módulos que publican estos datos.

5.2.4. Nodos Necesarios Para el Funcionamiento del Sistema

Para que el programa de detección de objetos (***object_color***) funcione correctamente, será necesario haber lanzado previamente una serie de programas o nodos de ROS que serán los encargados de publicar los datos de la base de datos y otros datos que serán leídos y procesados por el programa encargado de la detección.

Estos nodos son los siguientes:

- **Kitti_player_object:** Se trata del módulo que publica tanto los datos de la cámara, como los de la red convolucional, las matrices de calibración y la nube de puntos sin colorear.
- **Coloring**: Se encarga de publicar los datos de la nube de puntos del LIDAR coloreados con la información de la segmentación semántica de la imagen, como se vio en la imagen que esquematizaba el funcionamiento del sistema.

Por tanto, cada vez que queramos hacer que nuestro programa funcione será necesario lanzar previamente esos dos nodos de ROS. El sistema completo tendrá la siguiente estructura, obtenida al lanzar el comando ***rqt_graph*** con todos los nodos ejecutándose:

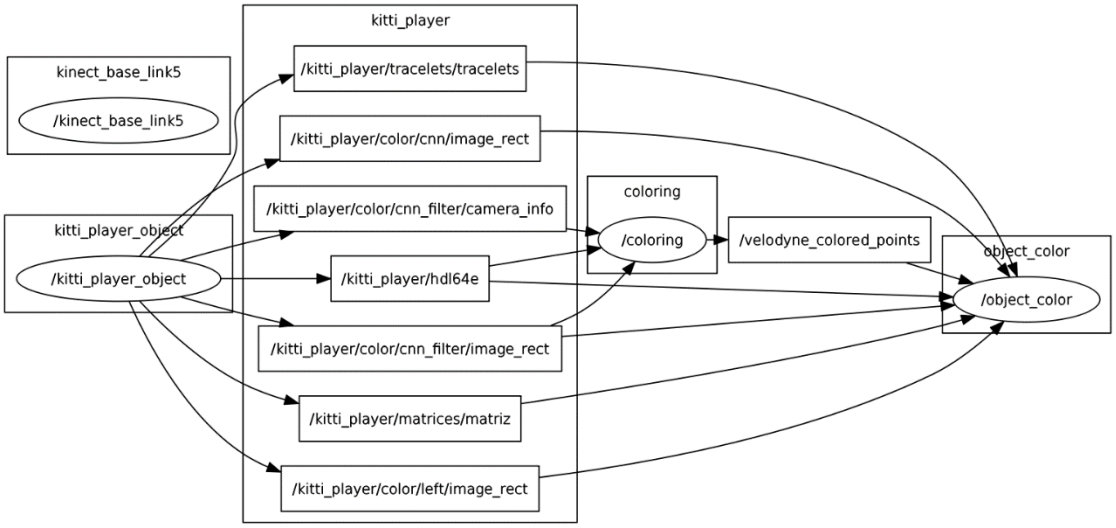


Ilustración 14 - Estructura del sistema.

En el gráfico se pueden ver los ***topics*** generados por el módulo ***kitti_player object*** señalados por una serie de flechas que salen de este módulo. A su vez se puede apreciar los *topics* empleados por el módulo de ***coloring*** y el *topic* que publica con todos los puntos de la nube de puntos coloreados. Finalmente se ve como el programa ***object_color*** está suscrito a los *topics* que se mencionaron con anterioridad.

Esta imagen nos da una visión global de la estructura que tiene nuestro sistema y nos ayuda a comprender el funcionamiento del mismo.

Todos los datos publicados por el módulo ***kitti_player_object*** serán directamente los datos extraídos de la base de datos, y puesto que ya han sido explicados no se les dará mucha importancia en este apartado.

Por otro lado, una de las bases del desarrollo de este trabajo será la nube de puntos coloreada y publicada por el módulo ***coloring***. Esta nube de puntos contendrá solo la información de los puntos de la nube total en 3D que se encuentren dentro de la visión de la cámara, es decir, solo los puntos que se encuentran delante del coche se conservarán a la hora de publicar esta nueva nube de puntos.

La principal transformación que sufre esta nube de puntos respecto a la otra es que pasa a contener la información **RGB** de la cámara segmentada semánticamente por la red convolucional, lo que nos ayudará a filtrar los puntos de la nube por tipo de objeto una vez que estemos trabajando con ella. El añadido de la información del color sobre la nube de puntos se realiza proyectando todos los puntos de la nube de puntos que se encuentran dentro de la visión de la cámara a la imagen de la cámara y dándole a cada punto la información de color que contiene cada píxel en la imagen.

5.2.5. Segmentación Semántica

Para explicar detalladamente el proceso será necesario previamente explicar las bases de la segmentación semántica llevada a cabo sobre la imagen. Para ello usaremos la red **ERFNet (Efficient Residual Factorized ConvNet for Real-time Semantic Segmentation)**[14], que nos proporcionará una imagen en la que cada uno de los elementos que aparecen será coloreado con un determinado color RGB, y que por tanto nos facilitará enormemente trabajar con dichas imágenes para detectar los elementos que se encuentran en ellas.

[14] En ocasiones se hará referencia a la red neuronal convolucional como CNN. https://github.com/Eromera/erfnet

El patrón de color que se ha seguido para colorear cada uno de los objetos es el siguiente:

Ilustración 15 - Esquema de colores CNN

Una vez procesada una de las imágenes la salida obtenida será una imagen que contenga solo esos colores representando los objetos que aparecen en la imagen original. Un ejemplo del funcionamiento se puede ver en la siguiente imagen extraída de la página de *github* desde donde se puede obtener la red neuronal, donde las imágenes de entrada que se encuentran a la izquierda han sido procesadas para obtener la imagen coloreada que se puede apreciar a la derecha:

Ilustración 16 - Ejemplo de segmentación semántica

5.2.6. Fusión de Datos de Cámara y LIDAR

Una vez tenemos presente como son las imágenes que provienen de la red podemos continuar con la explicación de la fusión de datos de cámara y LIDAR. El proceso, como se ha explicado antes, consiste en proyectar los puntos de la nube sobre la imagen segmentada por colores para darle la información de color a cada punto. Este proceso se muestra en las siguientes imágenes, donde nos podemos hacer una idea de los resultados que se obtienen:

Ilustración 17 - Imagen CNN

Esta primera imagen muestra el resultado de la CNN de la imagen de ejemplo que se va a emplear, mientras que la siguiente nos muestra la nube de puntos obtenida mediante el LIDAR del mismo escenario:

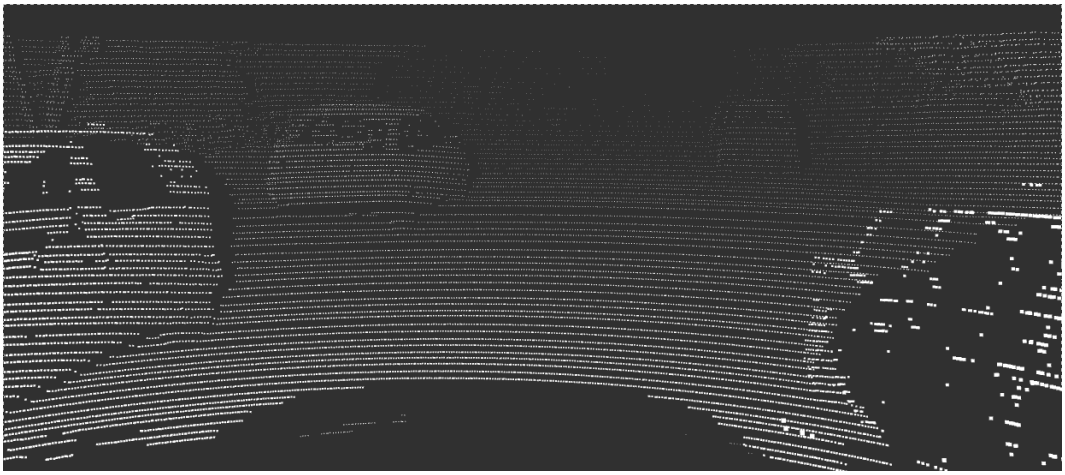


Ilustración 18 - Nube de puntos del LIDAR

Los datos de las imágenes mostradas anteriormente se fusionarán dándole un valor RGB a cada uno de los puntos del LIDAR que queden proyectados dentro de la imagen, obteniendo así una nube de puntos como la que se muestra a continuación:

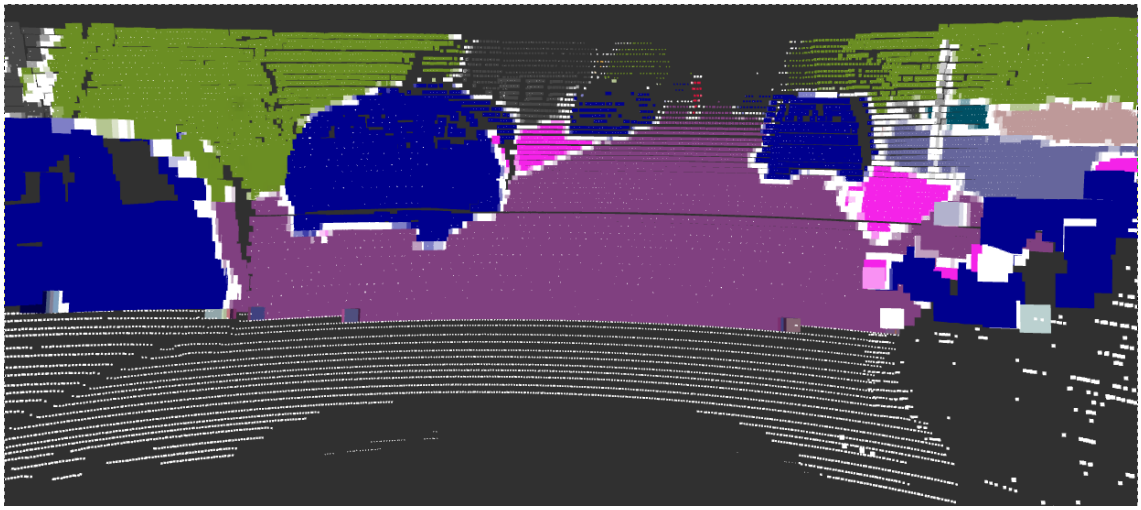


Ilustración 19 - Fusión de datos de la CNN y LIDAR

A pesar de que este método nos sea de gran ayuda tiene ciertos fallos difíciles de corregir que nos dificultarán en cierta medida las detecciones. Como se puede apreciar en las imágenes la proyección de los puntos del LIDAR a la imagen no es completamente precisa y en ocasiones los puntos de algunos objetos no se colorean del color que deberían. Si nos fijamos en el coche de la imagen que se encuentra más a la izquierda la parte trasera de este ha sido coloreada como si se tratase de un árbol o parte de la carretera en vez de un coche. De igual forma, como apreciamos a la derecha de la imagen en ocasiones los coches se acaban proyectando en zonas donde en realidad encontramos una pared u otro tipo de objeto.

Para intentar minimizar estos errores a la hora de realizar las proyecciones se decidió probar a delimitar la separación de cada uno de los objetos mediante líneas blancas, que no representan ninguno de los objetos que consideramos. Esto se llevó a cabo de esta manera debido a que el mayor problema que suponían estos errores era que se proyectaban partes de los coches sobre

la carretera o el suelo y esto generaba una gran cantidad de falsos positivos que reducían la precisión de las detecciones.

En la imagen de la nube de puntos mostrada anteriormente ya se pueden observar estas líneas blancas que separan los objetos, la imagen original que realmente ha sido usada para colorear la nube de puntos en la siguiente:

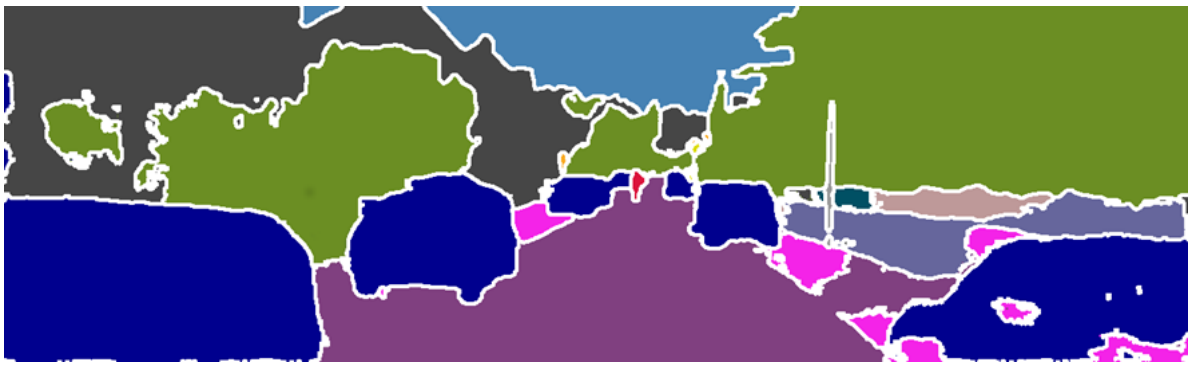


Ilustración 20 - Imagen CNN con reducción de bordes

Para generar estas imágenes a partir de las imágenes originales que nos aportaba la red convolucional se generó otro programa llamado ***quitar_contornos.cpp*** que emplea el filtro ***Canny***[15] de la librería ***OpenCV*** para obtener los puntos donde se juntan los objetos. Con ***Canny*** y la ayuda del método ***Dilate***[16] para ensanchar las líneas que conseguimos obtenemos una imagen en blanco y negro como la siguiente:

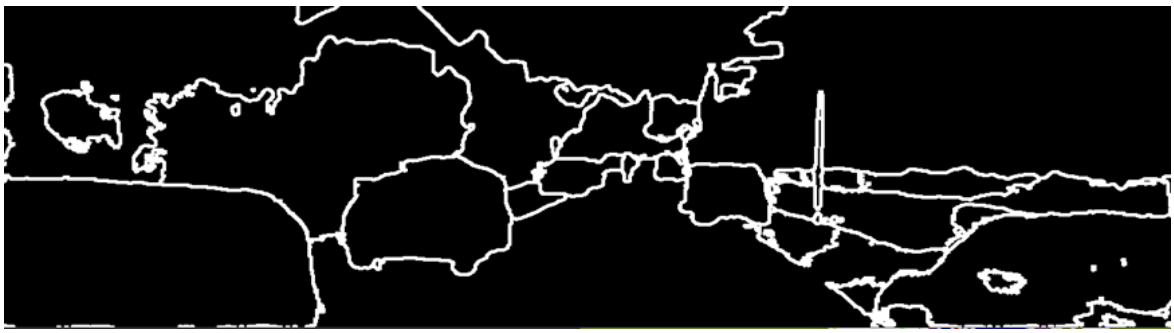


Ilustración 21 - Imagen procesada con Canny

Esta imagen superpuesta con la imagen original de la red es la que nos da el resultado deseado para evitar gran parte las proyecciones y los falsos positivos.

5.3. Detección de Objetos

Este apartado constituye la mayor parte del trabajo realizado en este proyecto, ya que la detección de los objetos es el objetivo principal de la aplicación.

[15] En algoritmo de Canny permite detectar los bordes de los objetos en las imágenes.
[16] Este método expande los puntos presentes en una imagen, en este caso ensancha las líneas de Canny.

Durante el desarrollo del proyecto se han realizado diferentes aproximaciones para la resolución de los diferentes problemas que han ido surgiendo a lo largo del mismo. Para mantener la explicación del proceso de detección que lleva a cabo la aplicación lo más clara y ordenada posible se irá explicando secuencialmente en el orden en el que se realiza cada acción. Por tanto, será durante la explicación de cada una de las funciones cuando se mencionarán las posibles distintas aproximaciones que se llevaron a cabo.

5.3.1. Funcionamiento Básico y Esquema de Detección del Programa

Para explicar las bases del funcionamiento de la aplicación que se ha realizado (***object_color.cpp***) se empezará resumiendo su funcionamiento con la ayuda del siguiente esquema, donde aparecen en forma de grafo las principales funciones de la aplicación y el orden en el que se llama cada una:

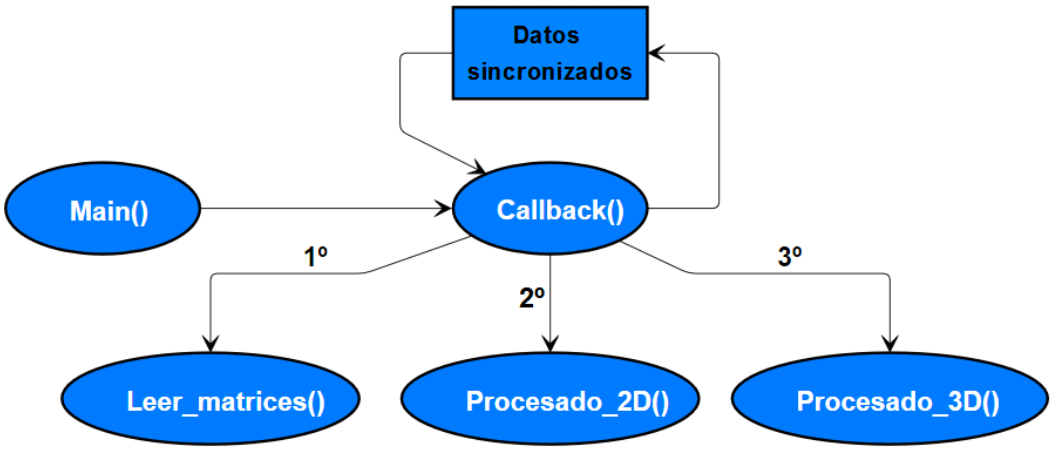


Ilustración 22 - Esquema de ejecución

En la función ***main()*** lo primero que se hace es inicializar el nodo de **ROS**, configurar el estado inicial del visor en 3D que usaremos para analizar los resultados obtenidos y suscribirse a todos los *topics* necesarios, que ya fueron explicados en el apartado anterior. Tras las suscripciones se emplea la clase ***message_filters*** para esperar que tengamos todos los datos sincronizados de un instante de tiempo y una vez están listos se llama a la función ***callback()*** pasándole todos los datos recibidos por los *topics*. A partir de ese momento se realizará una llamada a la función de ***callback()*** cada vez que tengamos todos los datos necesarios sincronizados.

La función de *callback()* se encarga de llamar ordenadamente a el resto de funciones que realizarán el proceso de detección. Es necesario que el orden de llamada sea siempre el mismo debido a que los datos de unas funciones dependen en muchos casos de algunos cálculos que se hayan llevado a cabo en otras, como se verá posteriormente. Es por esto por lo que es tan importante tener todos los datos con los que vamos a trabajar bien sincronizados antes de trabajar con ellos y por lo que no se puede llamar a las funciones en el mismo instante en que tengamos disponibles los datos principales con los que trabajan.

En las primeras implementaciones de la herramienta no se tuvo en cuenta esta sincronización y las funciones se llamaban cada vez que los datos que necesitaban se publicaban por uno de los

topics, lo que llevaba a errores del cálculo del ángulo de la carretera o de las matrices de calibración que se usaban al realizar el paso de datos de 3D a 2D, ya que las matrices y los datos de las cámaras o el LIDAR que se usaban en una misma función podían no pertenecer al mismo instante de tiempo.

Puesto que la función de *callback* es llamada cada vez que se va a realizar de nuevo todo el proceso de detección, lo primero que se hace en ella es refrescar el visor 3D eliminando todas las figuras que se han pintado en la iteración anterior y vaciar los dos vectores[17] de objetos que contienen los objetos detectados mediante información de la cámara y del LIDAR respectivamente.

Tras esto, el siguiente paso será llamar a la función ***leer_matrices()*** pasándole el mensaje que contiene las matrices de calibración publicadas en ese momento. La función se encargará de interpretar este mensaje y guardar las tres matrices de calibración publicadas en cada instante en tres variables globales del tipo ***cv::Mat***, de forma que cualquier otra función pueda leerlas y usarlas en cualquier momento.

Una vez leídas las matrices se llamará a la función ***procesado_2D()***, que procesará las imágenes que han sido publicadas y obtendrá una primera detección de los elementos que se puedan detectar usando solo la información de la imagen, y calculará el ángulo de la carretera, que será necesario para establecer el ángulo de algunos vehículos. Esta función será explicada más a fondo posteriormente.

Finalmente se llamará a la función ***procesado_3D()***, que se encargará de realizar las detecciones de los objetos usando la información del LIDAR, de pintar las cajas de detecciones y las nubes de puntos sobre el visor, de realizar el post-procesado para pulir la información de los objetos detectados y finalmente de guardar los datos en el formato establecido por la base de datos de KITTI y publicarlos en un *topic* en caso de que sea necesario para que puedan ser leídos por otro nodo.

5.3.2. Procesado de Imágenes

Como se ha mencionado anteriormente será la función ***procesado_2D()*** la que se encargará de realizar la mayor parte del procesado en 2D que se llevará a cabo por la aplicación. A continuación, se explicará detalladamente el funcionamiento de esta función, los pasos que sigue, los datos que usa y los que genera.

Esta función recibe como argumentos tres mensajes publicados en distintos *topics* que contienen tres imágenes diferentes: La imagen original en color, la imagen de la CNN y la imagen de la CNN con los bordes de los objetos coloreados de blanco que se explicó con anterioridad.

El primer paso será convertir esos mensajes a imágenes del tipo ***cv::Mat*** mediante el método ***cv_bridge::toCvCopy,*** lo que nos permitirá trabajar con los datos extraídos del mensaje.

Una vez creadas estas tres matrices como variables locales que contienen las imágenes se procederá a guardarlas en variables globales para que puedan ser empleadas posteriormente por otras funciones con el objetivo de poder representar los datos obtenidos sobre ellas.

[17] Se ha empleado el tipo *std::vector* para almacenar los objetos, por eso a lo largo del trabajo se hará referencia al array que contiene los objetos como *vector*.

Los siguientes objetivos serán calcular el ángulo en el que se encuentra la carretera y detectar los objetos solo con la información de la imagen.

5.3.2.1. Cálculo del ángulo de la carretera

El ángulo de la carretera se obtendrá de la imagen de la CNN con los bordes coloreados de blanco.

El proceso de obtención del ángulo será el siguiente:

1. Obtener solo la parte de la carretera de la imagen. Para ello se empleará el método ***inrange()*** de openCV, al que pasándole los límites del color que queremos conservar nos devolverá la parte de la imagen que se corresponde con carretera en blanco y negro. Para hacernos una idea las siguientes imágenes representan una imagen original y la extracción de la parte de la carretera:

Ilustración 23 - Imagen para extracción de carretera.

Ilustración 24 - Imagen con carretera extraída.

2. El segundo paso será obtener solo los extremos del área detectada en color blanco para poder calcular las dos rectas principales y el punto de fuga de la imagen. Para ello se empleará el filtro de ***Canny*** que ya fue explicado con anterioridad, y el resultado será el siguiente:

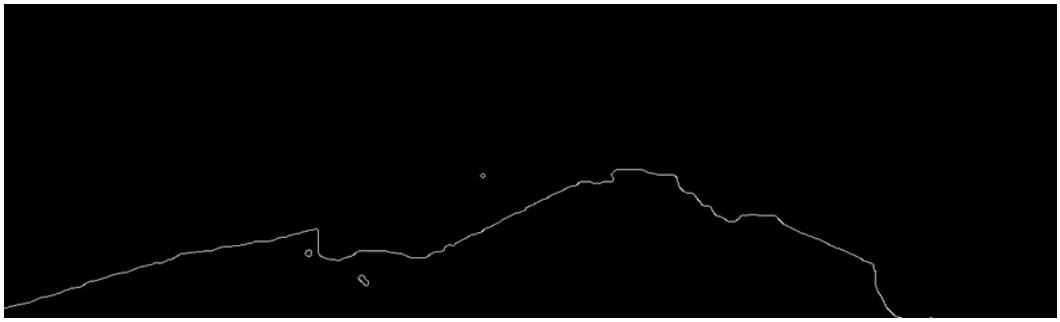


Ilustración 25 - Imagen tras el filtro de Canny.

3. Una vez con la imagen en ese estado podremos calcular las dos rectas principales de la misma empleando la ***transformada de Hough***. Estas dos rectas cortarán en el punto de fuga de la carretera, y conociendo el punto en el que nos encontramos y el punto de fuga de la carretera podremos calcular aproximadamente la dirección en la que se encuentra la misma. En la siguiente imagen se puede ver una representación de las rectas calculadas y el punto de fuga de la carretera dibujadas sobre la imagen de la CNN:

Ilustración 26 - Punto de fuga y ángulo de la carretera.

4. Para el cálculo real del ángulo de la carretera en las coordenadas del LIDAR será necesario realizar la proyección del punto de fuga en la imagen sobre los datos del 3D, lo cual se llevará a cabo empleando las matrices de transformación obtenidas anteriormente.
5. Previamente a la realización del cálculo se comprobará que la altura en píxeles del punto de fuga en coordenadas de la imagen no es inferior a 180 píxeles, ya que ese punto representa una altura superior a la de nuestras cámaras y por tanto la proyección tendería al infinito ya que nunca podría encontrarse con el suelo. En caso de que esto ocurra se fijará esa altura en 180 píxeles, ya que la altura del punto no afectará al ángulo de la carretera.
6. Finalmente, se procederá al cálculo del punto proyectado siguiendo la siguiente fórmula:

$$\text{Punto\_Proyectado} = (\text{Matriz\_R} \times \text{Matriz\_P})^{-1} \times \frac{Altura}{Factor\_Escala} \times (Matriz\_P)^{-1} \times Punto\_Fuga$$

Siendo ***Matriz_R, Matriz_T y Matriz_P*** las tres matrices de calibración, ***Altura*** la altura del LIDAR y ***Factor_Escala*** el valor obtenido en la posición (1,0) de la matriz obtenida al multiplicar la inversa de la ***Matriz_P*** por el punto de fuga obtenido.

7. Una vez obtenida la posición en coordenadas del LIDAR del punto de fuga y considerando que nuestro sensor se encuentra en el origen de coordenadas se puede obtener el valor del ángulo en el que se encuentra la carretera empleando la siguiente fórmula:

$$Ángulo\_Carretera = \tan^{-1}\frac{Punto\_Fuga.x}{Punto\_Fuga.y}$$

Con todo esto se habrá obtenido el ángulo de la carretera, que será empleado para establecer el ángulo de los coches que no hayan sido detectados correctamente y que por tanto no nos sea posible obtener su ángulo mediante la ***transformada de Hough***.

En los casos en los que la detección del ángulo de la carretera no sea posible porque los valores del punto de fuga obtenidos son considerados incorrectos se establecerá este ángulo con un valor de ***-INFINITY***, lo que nos indicará en pasos posteriores que no lo hemos podido obtener, y por tanto consideraremos como ángulo de la carretera el ángulo 0, que es el ángulo hacia el que se encuentra mirando nuestro coche y que corresponderá en la mayoría de las ocasiones con el ángulo de la carretera.

5.3.2.2. Detección de objetos en 2D

Una vez calculado el ángulo de la carretera se llamará a la función ***detectar_objetos_cnn()***, que será la encargada de detectar los objetos que puedan ser obtenidos solo por visión y de guardar su información y características en la estructura de objetos correspondiente.

Para conseguir detectar los objetos con la información de la CNN se utilizará el mismo principio básico que para calcular el ángulo de la carretera. Como los coches están coloreados con un determinado color podremos filtrarlos y calcular sus ***bounding boxes***[18] usando el mismo filtro empleado en el paso anterior.

El proceso llevado a cabo será el siguiente:

1. El primer paso será filtrar el color de los coches, lo que será llevado a cabo nuevamente mediante el método ***inrange()*** al que le pasaremos los valores del color de los coches **(142, 0, 0)**. Tras ello se realizarán un ***dilate()*** sobre la imagen para corregir los posibles huecos que se encuentren dentro de los coches si la CNN los ha pintado incorrectamente y finalmente un ***erode()*** para volver a reducir el tamaño de los coches si se ha ampliado con el dilate. El resultado sería el siguiente, partiendo de la imagen original a la imagen en blanco y negro obtenida:

[18] Cajas de mínimo tamaño que cubren el objeto.

Ilustración 27 - Imagen original para detección 2D.

Ilustración 28 - Imagen procesada para detección 2D.

2. Tras esto, se hallarán los contornos de las cajas mínimas que contienen a los objetos que se encuentran presentes en la imagen empleando los métodos ***threshold()*** y ***findcontours()*** de **openCV**, que nos devolverá un vector de objetos tipo ***Point*** que contendrán la información de los puntos que forman el contorno de la imagen y que por tanto podrán servir para calcular la caja mínima.
3. Mediante el método ***boundingRect()*** se obtendrán la anchura y altura del objeto en coordenadas de la imagen, lo que nos servirá tanto para guardar estos datos en nuestra estructura de datos como para pintarlos sobre la imagen para comprobar que las detecciones se están realizando correctamente.
4. El siguiente paso será pintar las cajas mínimas obtenidas en el apartado anterior sobre la imagen, obteniendo una imagen como la siguiente, que nos dará una idea de si las detecciones se están realizando correctamente:

Ilustración 29 - Ejemplo detecciones 2D.

5. Finalmente, se procederá a introducir los datos obtenidos en la estructura de datos de los objetos 2D detectados. Estos datos serán los cuatro vértices del objeto, el tipo de objeto (**coche, ciclista o peatón**), y la anchura y altura del objeto en píxeles.
6. El último paso será llamar a la función ***proyectar_cnn_3d()***, que se encargará de proyectar el punto más bajo de los objetos sobre el 3D para obtener una aproximación de la posición del objeto en las coordenadas del LIDAR. Este proceso se llevará a cabo de la misma forma que se proyectó el punto de fuga de la carretera para calcular el ángulo de la misma, y como se comentó entonces no será muy preciso debido a que la información de la distancia no la tenemos en el 2D. Sin embargo, será una aproximación bastante acertada que nos dará una idea de la posición del objeto.

Tras realizar este proceso tendremos detectados los objetos de la imagen empleando solo la información de la CNN.

La explicación se ha basado en la detección de los coches, sin embargo, la función que realiza la detección puede ser llamada con diferentes parámetros que indicarán el color que se filtrará, y por tanto la misma función realiza también la detección de peatones y ciclistas si cambiamos sus parámetros de entrada.

El principal problema de esta aproximación es que al no poseer información de la distancia a la que se encuentran los objetos no podremos separar unos de otros cuando se encuentren superpuestos, por tanto, esta información no será de mucha utilidad en escenarios en los que existan una gran cantidad de objetos del mismo tipo.

Un ejemplo del problema comentado sería el siguiente:

Ilustración 30 - Coches sin separar mediante detección 2D.

Como se puede observar en la imagen, al no poseer más información que el color el algoritmo juntará varios objetos en la misma detección.

Sin embargo, esto no será problema ya que más adelante al realizar el procesado con la información del LIDAR seremos capaces de detectar estos objetos de manera correcta.

El procesado en 2D solo nos será de utilidad cuando logremos detectar objetos en la imagen que no consigamos detectar con el LIDAR, lo cual suele ocurrir en objetos que se encuentran a grandes distancias ya que casi no poseemos puntos del LIDAR a esas distancias mientras que la imagen puede detectar los objetos correctamente en algunas ocasiones.

Por tanto, no todos los objetos detectados durante esta fase llegarán a publicarse como objetos reales, ya que en una de las fases de post-procesado que se explicarán posteriormente se realizarán comprobaciones para fusionar los datos obtenidos mediante cámara y LIDAR y se

eliminarán los objetos solo detectados por visión que se superpongan con algunos de los objetos del LIDAR.

5.3.3. Procesado de Datos del LIDAR Coloreados

La última función a la que llamará la función ***callback()*** es la función llamada ***procesado_3d()***. En este apartado se explicará detalladamente el funcionamiento de esta función y de las diversas funciones a las que llama.

El objetivo principal de esta función será detectar los objetos (***coches, ciclistas y peatones***) que se encuentren en el escenario recibido mediante los *topics*, visualizar la información de nubes de puntos publicada por estos *topics* y la información de las cajas mínimas de los objetos obtenida mediante nuestras detecciones, realizar el **post-procesado** de los datos y finalmente guardar estos datos en el formato de **KITTI** y publicarlos en caso de que sea necesario.

La funcionalidad es muy extensa y por tanto se dividirá en diferentes apartados para que la comprensión de su funcionamiento y estructura sean más sencillos.

5.3.3.1. Inicio de la función

La función recibirá como parámetro el mensaje del tipo ***sensor_msgs::PointCloud2ConstPtr&,*** que contiene la nube de puntos coloreada con la información de la CNN, por tanto, lo primero que se llevará a cabo será la declaración de una variable del tipo ***pcl::PointCloud<pcl::PointXYZRGB>::Ptr*** con la intención de convertir el mensaje a una variable de este tipo para poder trabajar con ella.

Esta conversión se realizará mediante los métodos ***pcl_conversions::toPCL*** y ***pcl::fromPCLPointCloud2***, y finalmente tendremos la nube de puntos con la información de color en una variable con el formato deseado que será el que usemos para trabajar con ella durante todo el proceso.

El siguiente paso será utilizar por primera vez el **visor 3D** que creamos en la función ***main()*** para visualizar la nube de puntos, ya que será sobre este visor y sobre la nube de puntos total sobre los que visualizaremos los datos con el objetivo de comprobar si las detecciones que realizamos son correctas o no y para poder encontrar los posibles errores que estemos cometiendo durante el desarrollo del algoritmo.

La nube de puntos con información de color tendrá el siguiente aspecto una vez la mostremos en el **visor 3D**:

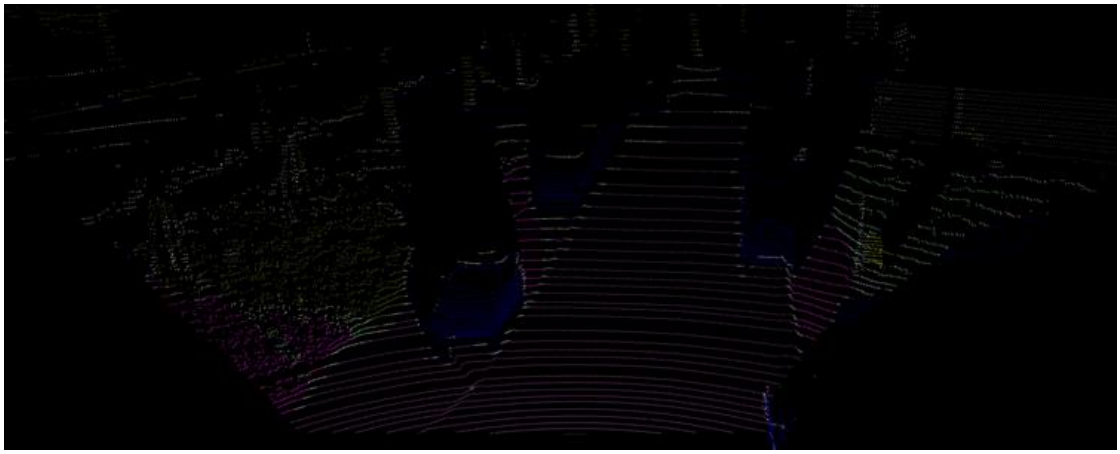


Ilustración 31 - Visor 3D.

Una vez tenemos la nube de puntos el siguiente paso será empezar a filtrar cada uno de los puntos que esta contiene por color y empezar a procesar la detección de cada uno de los objetos presentes en la misma.

Empezaremos realizando la detección de los coches y se continuará con la de los ciclistas y los peatones.

Una vez detectados los objetos se les realizará el post-procesado para pulir los posibles errores cometidos durante la primera detección.

Después de esto se comprobará la oclusión de cada uno de los objetos para poder establecerla en los datos de KITTI.

Tras ello, se realizará la proyección de las detecciones en 3D a la imagen en 2D, ya que será necesaria para determinar donde se encuentra cada objeto en la imagen, y una vez tengamos esta información se procederá a realizar la **fusión de datos** obtenidos mediante 3D y 2D que nos dará como resultado los objetos que serán publicados finalmente.

El último paso será guardar estos datos en el formato de KITTI y publicarlos en un *topic* para que puedan ser obtenidos por otros módulos.

5.3.3.2. Detección de vehículos

Para realizar la detección de cualquiera de los objetos siempre se empezará filtrando la nube de puntos total por color quedándonos solo con el color de los objetos que nos interesen, en este caso los vehículos.

Para ello se creará una nueva nube de puntos y se copiará en ella la nube de puntos completa. Será sobre esta nueva nube de puntos sobre la que se trabajará durante la detección de los coches. Una vez copiada esta nube será pasada por referencia a la función llamada ***Filtrado_Puntos_Color***() junto con el conjunto de valores máximo y mínimo **RGB** que se emplearán para filtrar el color deseado.

En el caso de los coches el color que se quiere filtrar será **0, 0, 142,** y por tanto se pasarán a la función los siguientes valores: **1, -1, 1, -1, 143, 141**.

El funcionamiento de la función de filtrado por color es muy sencillo. Se inicializará una nueva nube de puntos que será utilizada como nube de salida del filtro y se creará un objeto de condiciones del tipo ***pcl::ConditionAnd<pcl::PointXYZRGB>::Ptr***, al que se le pasarán todas las condiciones de los colores mediante el método ***AddComparison()***. Tras ello se creará un filtro del tipo ***pcl::ConditionalRemoval<pcl::PointXYZRGB>*** inicializándolo con el objeto de condiciones generado anteriormente, se le asignará la nube de puntos que queremos filtrar mediante el método ***SetInputCloud()*** y finalmente se filtrará la nube de entrada con el método ***filter()***, que nos devolverá la nube con los puntos que cumplen las condiciones que se han establecido.

La nube de salida de esta función tendrá el aspecto presentado en la siguiente figura:

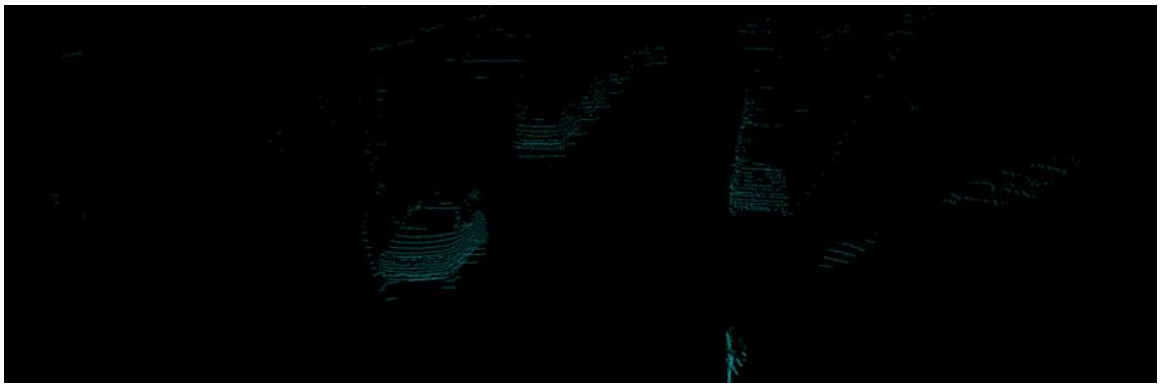


Ilustración 32 - Nube de puntos filtrada.

Comparándolo con la figura anterior se puede ver claramente que se trata de la misma imagen pero esta vez contiene solo los puntos del color que nos interesa. El color de los coches ha sido cambiado con el objetivo de que se aprecien mejor, ya que el color azul con el que están pintados originalmente no se aprecia de manera correcta sobre el fondo negro, sin embargo, este cambio solo se ha realizado a la hora de visualizar la nube, los puntos siguen teniendo la información de color original.

Una vez se ha obtenido la nube de puntos filtrada lo primero que se hará es comprobar si esta nube contiene algún punto, ya que habrá ocasiones en las que no exista ningún coche en la imagen y por tanto no será necesario realizar el resto de pasos de la detección de los coches.

Tras haber cumplido el requisito de que la nube contenga algún punto se pasará a realizar el ***clusterizado***[19] de los objetos que se encuentran en la nube, es decir, la separación en distintos objetos de los conjuntos de puntos que existen en la nube, ya que hasta el momento se ha tratado a la nube como un solo objeto.

Para la extracción de los *clusters* se seguirán los siguientes pasos:

1. Se creará un vector del tipo ***std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr>*** con el objetivo de almacenar en el todos los *clusters* extraídos mediante el algoritmo de extracción para poder trabajar más fácilmente con ellos.

[19] Obtención de los diferentes clusters que forman la nube de puntos completa.

2. Se llamará a la función ***extraer_clusters()*** pasándole como parámetros la nube de puntos de los coches, el vector creado en el paso anterior y tres datos tipo *double* que indicarán respectivamente la distancia mínima con la que el algoritmo considerará que dos puntos pertenecen al mismo *clúster*, que será establecida a 0.8m, el número mínimo de puntos que serán necesarios para considerar que forman un *clúster*, que estableceremos a 7 puntos y finalmente, el número máximo de puntos que pueden conformar un *clúster*, que hemos determinado con un valor de 5000 puntos. Los valores máximo y mínimo de puntos ayudarán en muchos casos a eliminar gran parte de los falsos positivos generados por las proyecciones de los coches sobre los objetos que se encuentran detrás de estos.
   Aunque el límite inferior no es muy importante dado que filtraremos los *clusters* muy pequeños más adelante, el superior si que lo es, ya que ayuda a eliminar las grandes proyecciones sobre paredes que se producen cuando en una imagen encontramos numerosos coches aparcados en fila. Un ejemplo de esto sería el que se mostrará en las siguientes figuras:

Ilustración 33 - Conjunto de coches aparcados en fila.

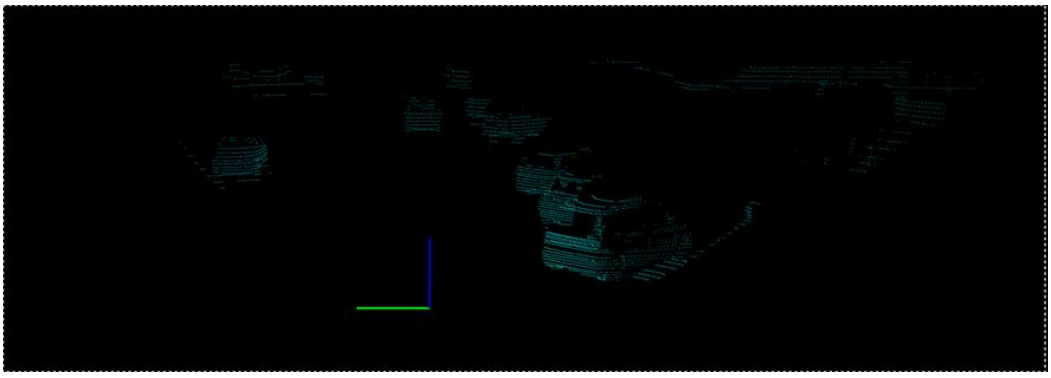


Ilustración 34 - Proyección en la pared que se encuentra tras los vehículos.

Como se puede ver, existe un objeto de grandes dimensiones a la derecha de la imagen debido a que una gran cantidad de coches se han proyectado sobre un edificio y las proyecciones se han juntado.

3. Una vez dentro de la función de extracción de *clusters* se generará un objeto del tipo ***pcl::EuclideanClusterExtraction<pcl::PointXYZRGB>*** al que se le asignarán todos los datos que se han pasado a la función (**nube de puntos, tolerancia y número máximo y mínimo de puntos**) y mediante su método ***extract()*** se almacenarán los índices de cada uno de los *clusters* en una variable del tipo ***std::vector<pcl::PointIndices>.***

4. Finalmente, mediante dos ***bucles for*** se recorrerán todos los elementos del vector de índices y todos los puntos que contiene cada elemento introduciendo cada punto de cada objeto del vector en una nube de puntos auxiliar, y finalmente introduciendo todas y cada una de esas nubes de puntos en el vector de objetos de tipo coches que se le pasó a la función por referencia. De esta manera tendremos un vector de nubes de puntos que contendrá de manera separada las nubes de puntos de cada uno de los *clusters* presentes en la nube de puntos total.

Llegado este punto tendremos preparados todos los elementos necesarios para empezar a realizar las detecciones de los coches, que se llevarán a cabo mediante la función ***car_detection()***, a la que se llamará una vez por cada *clúster* contenido en el vector de *clusters* y se le pasará como parámetro uno de los *clusters* en cada llamada.

5.3.3.3. Función Car_Detection()

Esta función llevará a cabo las tareas principales de la detección de cada uno de los objetos, determinando sus **tamaños**, **posición**, **ángulo**, **score** y demás características.

El primer paso que será llevado a cabo será recorrer la nube de puntos del clúster que se ha recibido como parámetro y almacenar los valores de los puntos máximo y mínimo en los ejes X, Y y Z. Con estos valores podremos obtener el **centroide** del objeto, su **altura**, **anchura**, **longitud** y alturas mínima y máxima.

5.3.3.3.1. Filtrado por Altura

Con esta información básica podemos realizar el primer filtrado para eliminar gran parte de los *clusters* que no nos valen. Este primer **filtrado de *clusters*** estará basado en su altura.

Consideraremos que la altura mínima necesaria para considerar un *clúster* como válido tendrá que ser de al menos 35cm a más de 50 metros y de al menos un metro cuando el objeto se encuentre a unos pocos metros de nosotros, estableciendo de esta manera una altura mínima diferente dependiendo de la distancia a la que se encuentre del origen y siguiendo una progresión aritmética de 15cm por metro.

En las siguientes imágenes podemos ver como actúa este filtro eliminando la detección de coches que en realidad son errores de detección:

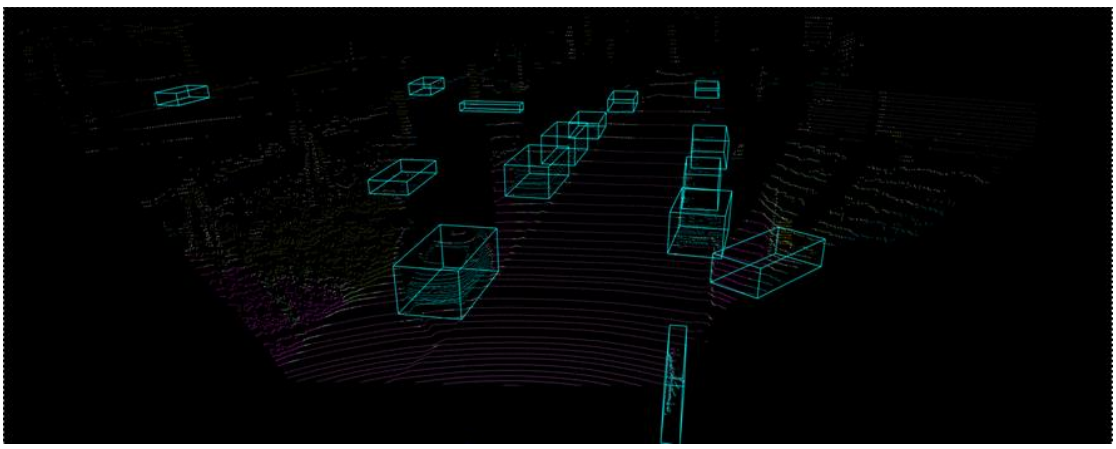


Ilustración 35 - Detecciones antes de aplicar el filtro.

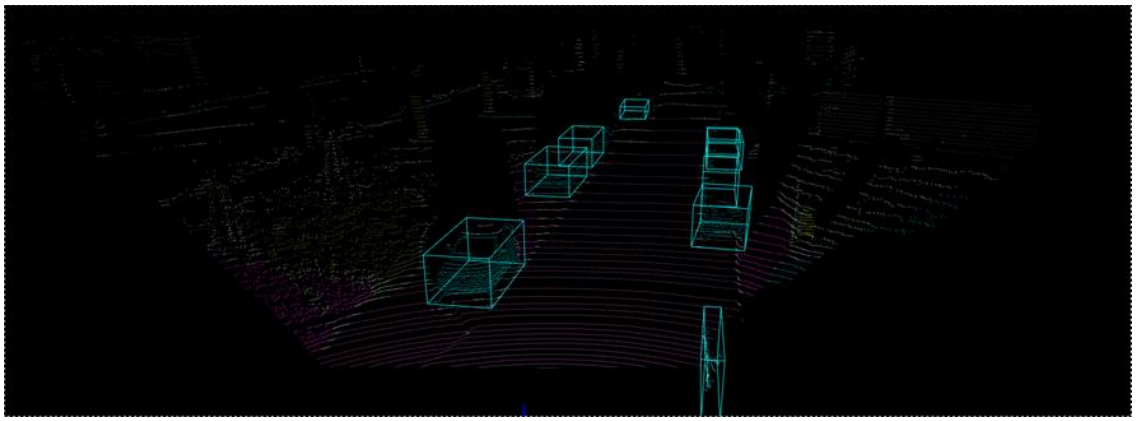


Ilustración 36 - Detecciones tras la aplicación del filtro.

Por lo general este tipo de filtrado nos ayudará a eliminar la mayor parte de los *clusters* que no son coches y que son causados debido a proyecciones de los mismos sobre el suelo. Sin embargo, en algunas ocasiones es posible que se eliminen algunos coches a distancias considerables debido a que la detección de los mismos no haya sido muy buena. En cualquier caso, los resultados de los *tests* con la ayuda de este filtro son muy superiores a los obtenidos si dejamos de filtrar por altura, por lo que la valoración de los resultados nos lleva a pensar que el filtro es imprescindible y que el beneficio que hace eliminando falsos positivos es muy superior a los posibles perjuicios que puede causar eliminando el *clúster* de algún coche a gran distancia.

5.3.3.3.2. Cálculo del Ángulo del Vehículo

Tras haber realizado este primer filtrado se procederá a calcular el ángulo del vehículo al que pertenece el *clúster*. Esto se llevará a cabo aplicando la ***transformada de Hough*** a la imagen que obtendremos tras proyectar la nube de puntos del clúster sobre el suelo.

La ***transformada de Hough*** nos devolverá la recta principal del coche que hemos detectado, que en caso de una buena detección será la dirección en la que se encuentra el vehículo.

Para realizar el proceso completo de detección del ángulo se seguirán los siguientes pasos:

1. Se creará una imagen de tipo ***cv::Mat*** con un tamaño de **500x500** píxeles y se inicializará el valor de todos los píxeles a 0.
2. Comenzaremos a recorrer todos los puntos de la nube de puntos y a **proyectarlos** sobre nuestra imagen. Para ello será necesario conocer el centroide del *clúster*, que ya ha sido calculado en pasos anteriores, de manera que se pueda proyectar el clúster sobre el centro de nuestra imagen. A la hora de proyectar todos los puntos sobre el plano primero se llevarán al centro, restando al valor de todos los puntos obtenidos el valor del centroide del *clúster*. Tras ello se realizará una conversión de unidades en la que cada metro en la nube de puntos corresponderá a 40 píxeles en la imagen, y finalmente se desplazarán los puntos del *clúster* al centro de la imagen para que el *clúster* proyectado aparezca centrado en el resultado final. Un ejemplo de la imagen que obtendremos tras proyectar uno de los *clusters* de los coches es el siguiente:

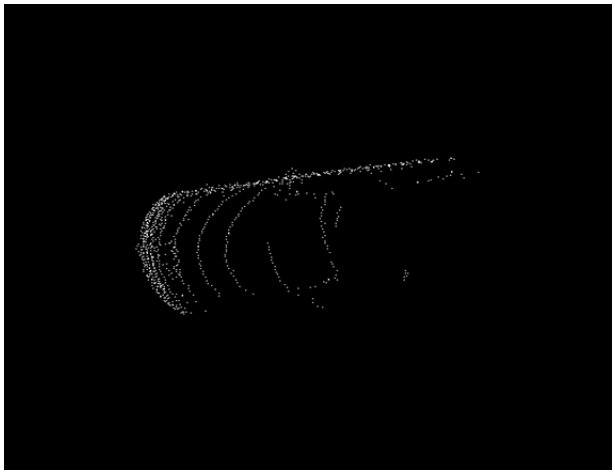


Ilustración 37 - Vehículo proyectado sobre el plano.

3. Una vez tenemos la proyección realizaremos un dilatado de los puntos de la imagen mediante el método **dilate()** de **openCV**, lo cual nos ayudará a mejorar la precisión del cálculo del ángulo principal mediante la **transformada de Hough**, quedando la imagen de la siguiente manera tras haberla dilatado:

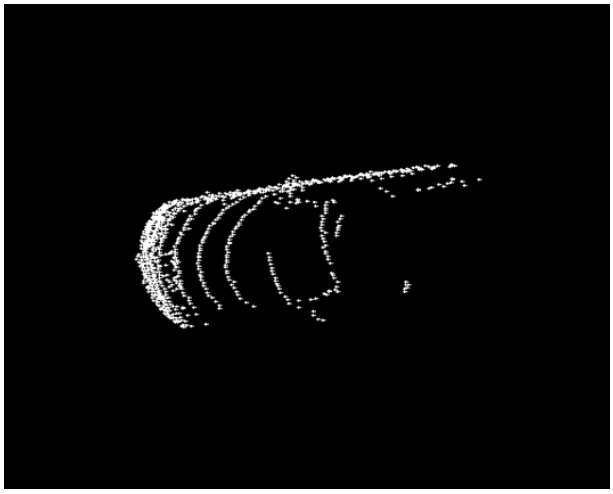


Ilustración 38 - Proyección del vehículo tras dilatar sus puntos.

4. Ahora que ya tenemos la imagen lista, el siguiente paso será aplicarle la ***transformada de Hough*** para obtener la línea principal de la imagen, que corresponderá en la mayoría de las ocasiones con el ángulo en el que se encuentre el vehículo o un ángulo perpendicular a ese si lo que se detecta mejor es la parte delantera o trasera del coche en vez de la lateral. En cualquiera de los dos casos tendremos una buena aproximación del ángulo real del coche. Para el cálculo de la transformada emplearemos el método ***HoughLines()*** de **openCV**, al que le pasaremos la imagen generada previamente y nos devolverá la recta principal de la imagen. Para hacernos una idea del resultado obtenido en la siguiente imagen se muestra la recta devuelta por el algoritmo sobre la imagen mostrada anteriormente:

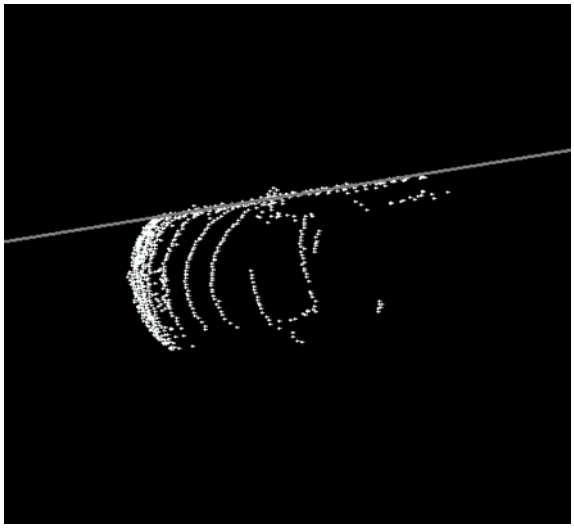


Ilustración 39 - Transformada de Hough.

También se puede observar en la siguiente imagen un ejemplo en el que la ***transformada de Hough*** devuelta corresponde con el ángulo perpendicular al ángulo real del vehículo:

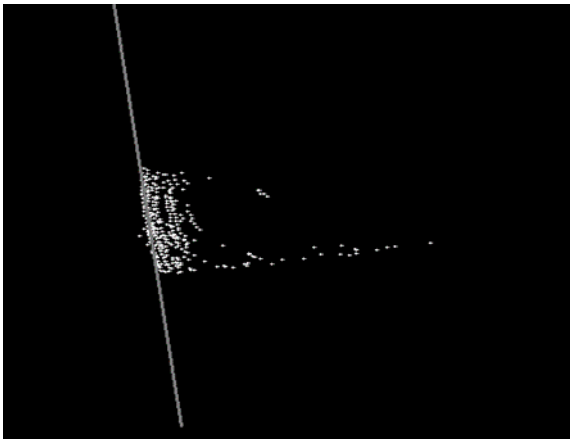


Ilustración 40 - Transformada de Hough perpendicular.

5. Finalmente se realizarán dos comprobaciones que ayudarán a mejorar la precisión de la detección de los ángulos. La primera consistirá en no tener en cuenta el ángulo calculado por la ***transformada de Hough*** si la cantidad de puntos del clúster es muy pequeña, estableciendo en este caso como ángulo del vehículo el ángulo de la carretera que se calculó mediante visión. Un ejemplo de uno de estos casos sería el de la siguiente imagen, donde se puede apreciar que a pesar de tratarse de un clúster de un coche debido a la ausencia de una cantidad adecuada de puntos la ***tranformada de Hough*** no puede calcular el ángulo correctamente:

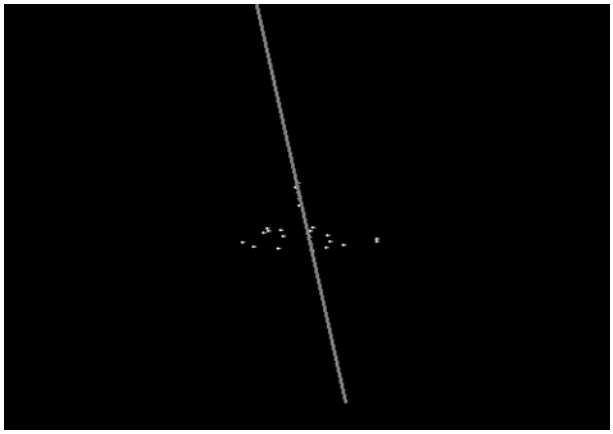


Ilustración 41 - Transformada de Hough no válida.

A pesar de que la imagen representa realmente el clúster de un coche, debido a que la detección es muy mala no nos podemos fiar del ángulo detectado mediante la transformada de Hough.

6. El último caso que será contemplado será comprobar si el ángulo que hemos detectado es el de la parte trasera o delantera del vehículo y por tanto es necesario girar el ángulo 90º. En caso de haber detectado la parte trasera del vehículo como ángulo principal esto será debido a que el coche se encuentra delante de nosotros y avanzando en nuestra misma dirección, ya que en cualquier otro caso, si vemos la parte lateral del vehículo, será esta parte la que conforme la recta principal devuelta por Hough. Es por ello que la primera comprobación que se realizará para saber si el ángulo detectado es el perpendicular será contemplar solo los ángulos que se encuentren dentro de un rango de 25º respecto al ángulo perpendicular a nuestro coche. En cualquier otro caso veríamos una gran parte del lateral del vehículo y por tanto probablemente esa no sería la parte trasera. Y el segundo requisito a contemplar será que el ancho del *clúster* detectado en esa dirección no llegue a los 2,5 metros, ya que si pasa de esta longitud será muy probable que no se trate de la parte trasera de un vehículo.

El cálculo del ángulo del vehículo será la base para la detección de la caja mínima que lo envuelve, por tanto, una vez realizado este paso se procederá a calcular esta caja.

5.3.3.3.3. Cálculo de la Caja Mínima del Vehículo

Nuevamente se volverá a emplear la proyección en el plano del *clúster* para calcular la caja mínima al igual que se hizo con el ángulo.

La intención será rotar la imagen de la proyección del *clúster* con el ángulo del clúster, de manera que este quede posicionado con un ángulo de 0 grados y podamos obtener los puntos máximo y mínimo del clúster en esa posición. Con esto tendremos la anchura y altura máximas del coche, y para calcular sus vértices solo tendremos que volver a rotar la imagen a su posición original y obtener el valor de los vértices calculados anteriormente. En las siguientes imágenes se contemplará todo el proceso para que quede claro como es llevado a cabo:

1. Imagen original:

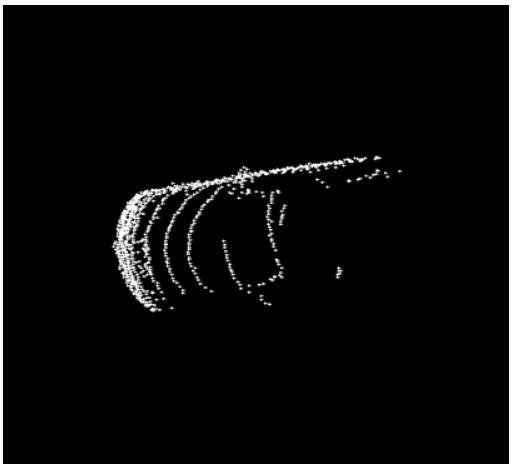


Ilustración 42 - Imagen original para el cálculo de la caja mínima.

2. Imagen rotada:

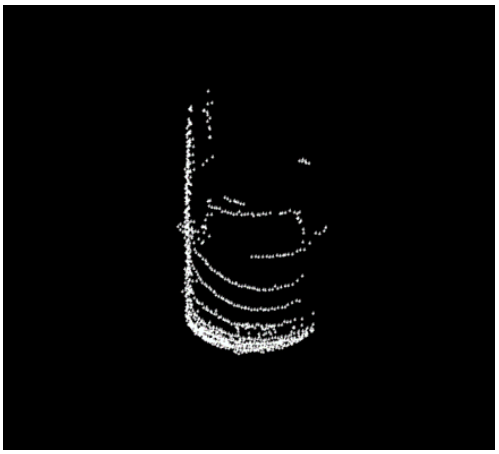


Ilustración 43 - Imagen rotada con el ángulo detectado.

3. Cálculo de los puntos máximo y mínimo en las coordenadas X e Y para obtener la caja mínima que rodea el vehículo, como se puede ver en la siguiente imagen:

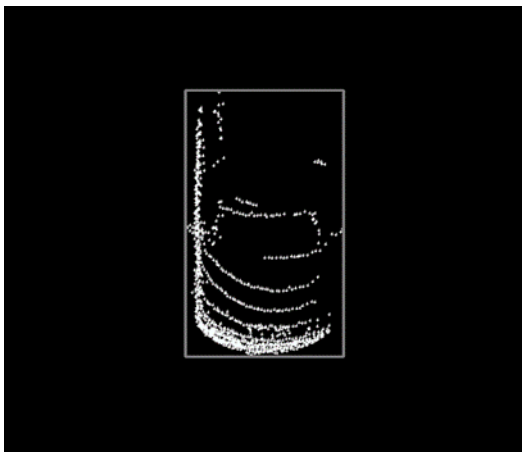


Ilustración 44 - Caja mínima rotada del vehículo.

4. Rotación de los vértices de la caja a la posición original:

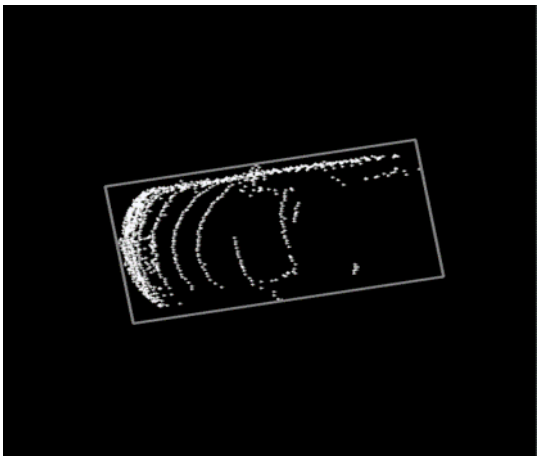


Ilustración 45 - Caja mínima en su posición original.

De esta manera, al igual que se han obtenido los puntos máximo y mínimo del *clúster* cuando su ángulo era 0º podemos volver a obtener los valores reales de los vértices rotando cada uno de ellos sobre el centro de la imagen y como se puede observar obtenemos la caja en la posición en la que realmente está.

Para realizar la rotación de los puntos se empleará la función ***rotacion()***, que se encargará de rotar un determinado ángulo el punto que le pasemos respecto a un centro establecido.

Una vez tenemos el valor de los cuatro vértices de la caja mínima en la proyección del *clúster* solo tendremos que volver a realizar la misma transformación en el sentido contrario para obtener el valor real de los vértices en las coordenadas del LIDAR. Llevaremos los puntos a su posición original sumando el valor del centroide y dividiéndolos por 40, ya que como se expuso anteriormente en nuestra transformación cada metro equivale a 40 píxeles de la imagen. La función empleada para transformar los puntos de la imagen al LIDAR nuevamente es la siguiente:

$$Vértice\_3D = \left(\frac{(Vértice\_2D - Tamaño\_Imagen)/2}{Resolución} \right) + Centroide$$

Por último, una vez que tenemos el valor real de los cuatro vértices, podremos calcular la anchura y longitud reales del vehículo, con las que realizaremos una última corrección del ángulo en caso de que el valor obtenido esté girado 90º, para ello compararemos el valor de la longitud del vehículo y cambiaremos el valor del ángulo en caso de que la anchura obtenida sea mayor que la longitud.

Tras haber realizado todos estos pasos tendremos el valor de los cuatro **vértices** del vehículo junto con su **altura**, **anchura**, **longitud** y **posición**, y podremos por tanto publicar esos datos y representarlos sobre la nube de puntos.

En la siguiente imagen se pueden observar las cajas mínimas que hemos detectado pintadas sobre la nube de puntos y envolviendo el clúster del coche que aparecía en las imágenes anteriores:

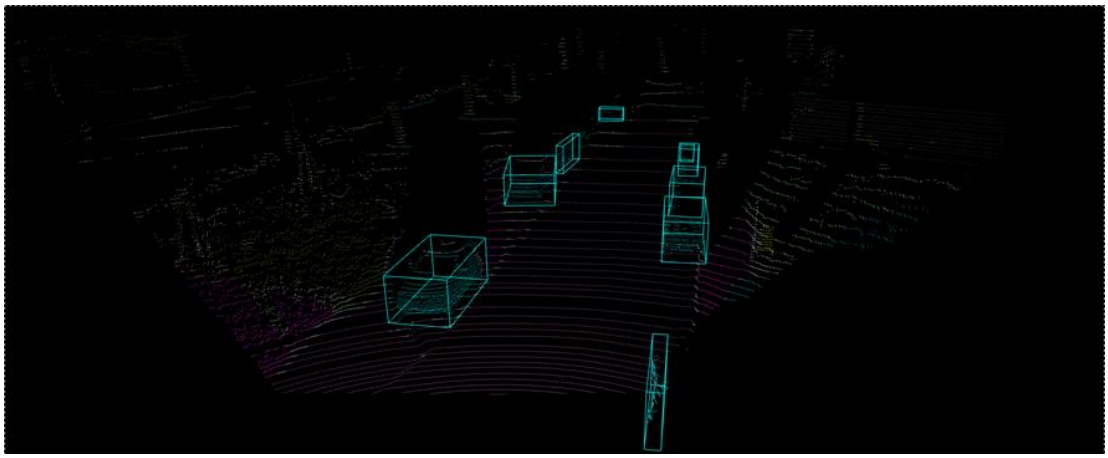


Ilustración 46 - Caja mínima 3D.

5.3.3.3.4. Cálculo de Score y Caja Fija

Sin embargo, antes de dar por terminada la detección será necesario calcular la **puntuación**[20] con la que se ha realizado la detección y calcular una **caja de tamaño fijo** que será establecida en los vehículos con una baja puntuación.

Un vehículo tendrá una mala puntuación en su detección cuando el tamaño de la caja mínima obtenida sea muy pequeño o muy grande para que se trate de un coche. Si el objeto detectado es demasiado grande será tratado en el post-procesado, sin embargo, si es muy pequeño, en vez de usar su caja mínima calcularemos una caja de tamaño fijo que envuelva la misma. Esto es debido a que será imposible que un coche tenga un tamaño menor de un determinado límite, y por tanto el problema de detectar una caja pequeña será debido a la distancia o las oclusiones y en la mayoría de los casos la **caja fija** que le asignaremos se acercará más que la detectada al tamaño real del coche.

Como se puede observar en la imagen anterior existen cajas al fondo que tienen un tamaño demasiado pequeño para ser un coche. Serán estas cajas las que se corregirán con la caja de tamaño fijo.

El proceso para calcular la caja de tamaño fijo será sencillo:

1. Obtendremos los dos puntos del *clúster* más cercanos a ***recta de Hough***, que serán los dos puntos principales de la detección, y seleccionaremos el punto más cercano de esos dos al origen. Será ese punto sobre el que se pinte la esquina más cercana de la caja fija, ya que de esta manera nuestra caja fija siempre envolverá a la caja detectada.

[20] La puntuación, también llamada score es el grado de seguridad con el que se cree que el objeto detectado es correcto.

2. Se calculará una caja fija en el origen de coordenadas, se hallará el punto homólogo al detectado en la caja detectada y finalmente se rotará y trasladará la caja al mismo punto que hemos seleccionado en la caja original.

Para hacernos una idea más exacta de lo que está pasando en la siguiente imagen se muestran varios *clusters* con su caja detectada de color azul y la caja fija calculada de color amarillo:

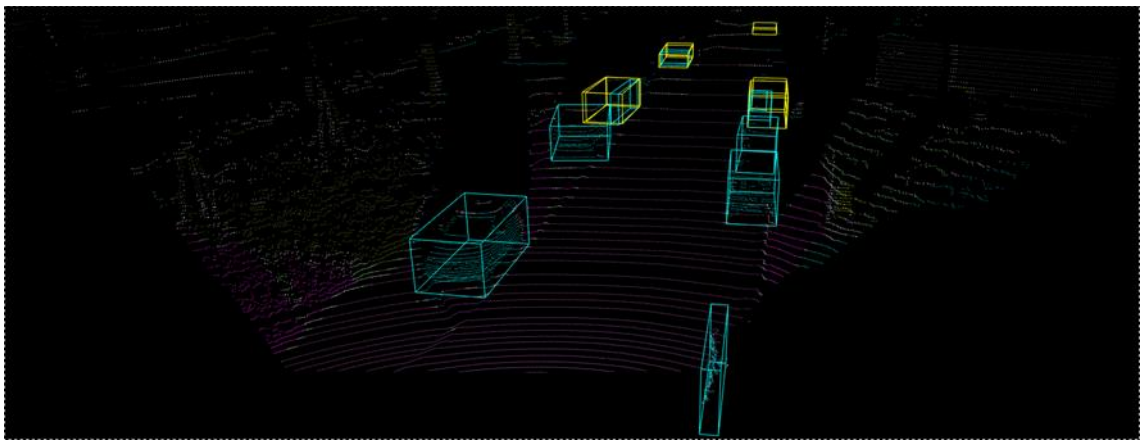


Ilustración 47 - Detección de caja fija.

En cada uno de los casos en los que el coche no alcance determinada puntuación se le pondrá su caja fija en vez de la detectada. Tras realizar las comprobaciones de los resultados y variar el tamaño de la caja fija se llegó a la conclusión de que en todos los casos el hecho de poner la caja fija mejora nuestros resultados a la hora de realizar la comprobación con los resultados de **KITTI**, y se comprobó que el tamaño óptimo de la caja fija es de **1,7** metros de **ancho** y **3,7** metros de **largo**, obteniendo los mejores resultados con este tamaño de caja.

Sin embargo, la caja fija también puede ser el origen de ciertos errores, ya que en algunas ocasiones el coche real no es tan largo como la caja fija y dos objetos pueden llegar a superponerse, como se muestra en la siguiente imagen:

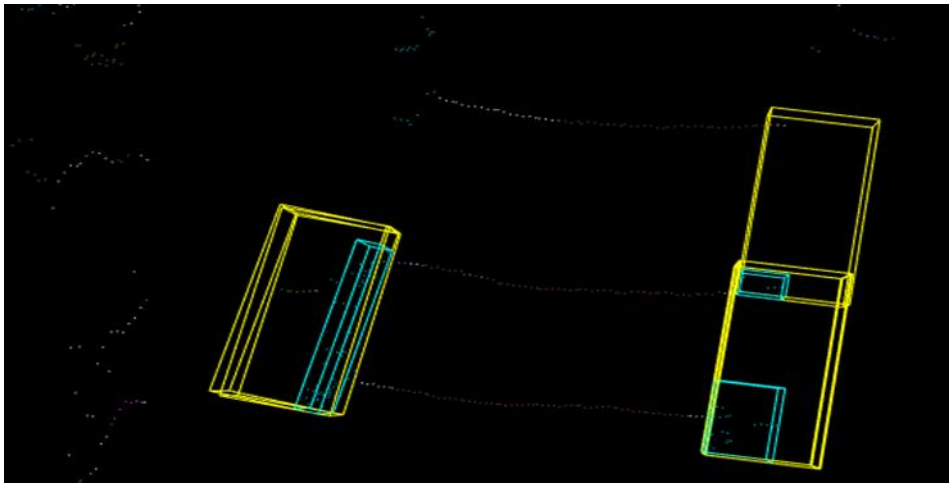


Ilustración 48 - Superposición de cajas fijas.

En todas las cajas de la imagen el cálculo de la caja fija supone una mejora para la detección, sin embargo, en los dos coches de la derecha se produce una superposición debido a que la caja fija es más larga de lo que debería o a que se trata en realidad de un solo coche que ha sido dividido en dos por tener una cantidad de puntos muy baja.

En cualquier caso, la mejora de las detecciones empleando la caja fija es notable, y el hecho de que dos cajas lleguen a superponerse en algunos casos no supone un problema importante.

Una vez realizado el cálculo de la caja fija solo nos queda calcular el ***score*** y decidir cual de las cajas será la empleada para determinar el tamaño de cada objeto de manera definitiva.

Tras realizar varias aproximaciones para calcular las precisiones de las detecciones basadas en el **número de puntos** del *clúster*, la **distancia** o un conjunto de las dos, se determinó que era más efectivo el cálculo de la misma como una aproximación al **tamaño de la superficie detectada** de la caja fija, que es en definitiva muy aproximado al tamaño medio de un coche.

Aun así, el cálculo de la aproximación en longitud y anchura por separado dio lugar a mejores resultados que si solo se tomaba en cuenta la superficie, ya que en algunos casos solo obteníamos el lateral del coche y la superficie total era muy pequeña por tener una anchura muy pequeña a pesar de que la detección era bastante acertada.

Por ello, finalmente se decidió emplear este último método:

1. Primero se calcula el porcentaje de la longitud detectada respecto a la longitud de un coche medio (3,7 metros). Si ese porcentaje de longitud se encuentra entre un 80% y un 125% consideramos que el tamaño detectado es bueno y se le asigna una precisión inicial de 1.
2. En cualquier otro caso se resta a 1 el porcentaje (sobre 1) que se desvía la longitud.
3. Se vuelve a repetir el mismo proceso con la anchura, siendo la anchura de un coche medio establecida en 1,7 metros obteniendo la precisión sobre la anchura.
4. Finalmente, se multiplican las dos precisiones, obteniendo como resultado la precisión total de la detección, que solo será 1 si tanto el ancho como el largo del vehículo detectado se encuentran entre el 80 y el 120% del tamaño medio de un vehículo que hemos establecido.

Será esta medida la que se usará para establecer el ***score*** para la base de datos de ***KITTI*** y también mediante la que se decidirá si se emplea la caja detectada o la caja fija.

Siempre que el score detectado se encuentre por debajo del 0.6 se establecerá por defecto el tamaño de la caja fija, con la única restricción de que la caja fija nunca se establecerá en los coches que se encuentren demasiado cerca del origen y por tanto estén truncados, ya que en estos casos será mejor establecer el tamaño de la caja detectada puesto que solo veremos una parte muy pequeña del vehículo y los resultados empeorarán si se emplea la caja fija.

Finalmente, solo quedará llamar a la función que añade los datos de los objetos al vector de objetos para que queden preparados para realizar el **post-procesado** y posteriormente ser publicados.

5.3.3.4. Función Add_Objects()

Esta función recibirá como parámetros el **tipo** de objeto, los cuatro puntos que corresponden con los **vértices** de la caja mínima detectada, el **ángulo** detectado, la **altura** máxima y mínima, el **score**, el **truncamiento** y la **nube de puntos** que forma el *clúster* del objeto.

La función calculará la **anchura**, **longitud** y **centroide** del objeto mediante los puntos de los vértices, la **altura** mediante las alturas máxima y mínima y la **distancia** del centroide del objeto al origen. Todos estos datos serán almacenados en una estructura del tipo ***Objeto*** que será introducida como un nuevo elemento en el vector de objetos donde se almacenarán todos los objetos detectados.

5.3.4. Detección de ciclistas

El proceso de detección de ciclistas sigue la misma estructura que el de detección de coches con algunas diferencias al calcular la caja mínima que envuelve al ciclista[21].

Nuevamente, se creará una nube de puntos auxiliar del tipo ***pcl::ConditionAnd<pcl::PointXYZRGB>::Ptr*** en la que se copiará la nube de puntos original con el objetivo de no modificar esta.

La nueva nube de puntos será filtrada por color para quedarnos solo con los puntos de la nube que cumplan con los valores **RGB** que corresponden con un ciclista **(255, 0, 0)**. Para ello se empleará nuevamente la función ***Filtrado_Puntos_Color()***, que modificará la nube de puntos que le hemos pasado eliminando todos los puntos que no cumplen con el criterio de color establecido.

El siguiente paso será visualizar la nube en el visor 3D para comprobar que el filtrado se ha realizado correctamente, obteniendo en el proceso una imagen como la siguiente:

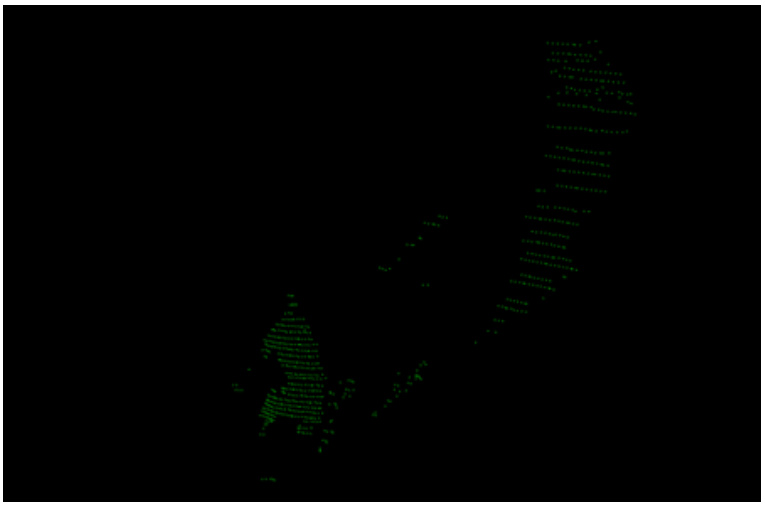


Ilustración 49 - Nube de puntos de ciclistas filtrada.

[21] También variarán algunos parametros empleados en los filtros.

El color de la nube de puntos de la bicicleta ha sido cambiado de rojo a verde para no confundirla posteriormente con las nubes de puntos de los peatones, que tienen un tono de color parecido.

Como se puede observar, en la nube de puntos existe una proyección de gran tamaño sobre el terreno que se encuentra detrás del ciclista, esto es debido a que la imagen obtenida mediante la ***CNN*** no es del todo precisa y al realizar la proyección muchos puntos no se proyectan de manera adecuada. Podemos ver la imagen de la ***CNN*** correspondiente a la nube de puntos mostrada anteriormente en la siguiente imagen:

Ilustración 50 - Imagen CNN ciclista.

Cabe destacar que, como se puede observar, solo se han filtrado los puntos que corresponden al ciclista y no los de la bicicleta. Se podrían haber filtrado ambos colores de manera que en la nube de puntos obtendríamos también los puntos correspondientes a la bicicleta. Esta solución fue probada dando como resultado un aumento considerable de los **errores de detección** y no mejorando en absoluto la detección de los ciclistas.

Esto fue debido a que la CNN dibuja la bicicleta como un elemento mucho más grande de lo que realmente es, sin tener en cuenta que el LIDAR prácticamente no detecta ningún punto de la bicicleta, ya que atraviesa las ruedas y la estructura de la bicicleta que no son mas que unos simples tubos. Viendo el problema de las proyecciones sobre el suelo que teníamos con el ciclista nos podemos hacer una idea de los resultados a la hora de proyectar los puntos de la bicicleta, obteniendo una cantidad mínima de puntos en la bicicleta y unas proyecciones de tamaño considerable detrás de ella.

Una vez tenemos la nube de puntos con la información que nos interesa el siguiente paso será realizar la extracción de los ***clusters*** al igual que se hizo con los coches. Para ello se empleará la misma función ***extraer_clusters(),*** que nos devolverá un vector con los ***clusters*** que ha logrado extraer. En este caso los parámetros que se le pasarán a la función serán una distancia mínima entre puntos de 40cm, un mínimo de 10 puntos y un máximo de 1000.

Finalmente, se recorrerá este vector de nubes de puntos y se llamará a la función ***cyclist_detection()***, que será la encargada de calcular la caja mínima y el resto de características de cada una de las detecciones e introducir los objetos detectados en el vector de objetos.

5.3.4.1. Función Cyclist_Detection()

El primer paso será recorrer la nube de puntos del *clúster* obteniendo los puntos máximo y mínimo de la nube para cada una de las coordenadas en 3D. Con estos datos podremos calcular el **centroide** de la nube de puntos, la **anchura**, **longitud** y **altura** del *clúster* y su **distancia al origen**.

El filtrado que realizaremos en este caso para eliminar parte de las proyecciones será más sencillo que en el caso de los coches. Se eliminarán todos los ***clusters*** que no alcancen los 80 cm de altura, ya que se considera que un ciclista subido en una bicicleta siempre sobrepasará esa altura. También se comprobará que el *clúster* tenga al menos 30cm de ancho y largo.

Una vez realizado el primer filtrado se procesará la nube de puntos de igual manera que con los coches y se proyectará sobre el suelo para calcular la caja mínima que la rodea.

La imagen resultante que obtendremos de un ciclista será como la siguiente:

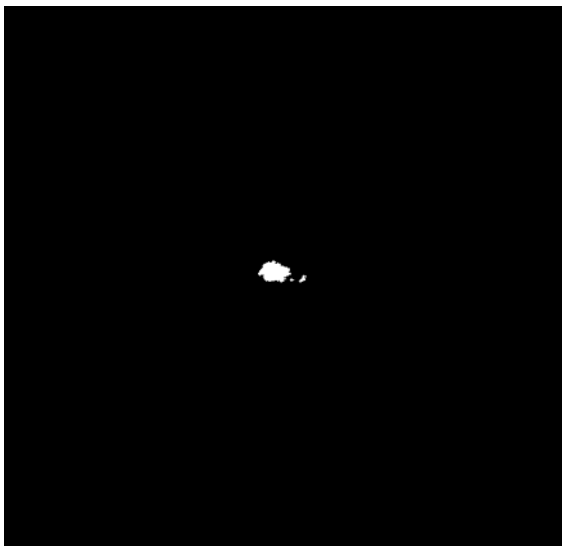


Ilustración 51 - Ciclista proyectado sobre el plano.

Viendo el resultado podremos deducir que en este caso la detección del ángulo por parte de la ***transformada de Hough*** no será muy precisa. En las pruebas que se realizaron calculando el ángulo de esta manera se obtuvieron resultados prácticamente aleatorios del ángulo, por lo que se optó en este caso a calcular directamente el ángulo de los ciclistas mediante el cálculo de la caja mínima que rodeaba a la imagen obtenida.

El cálculo de la caja mínima ha sido llevado a cabo mediante el método ***minAreaRect()*** de **OpenCV**, que nos devuelve directamente los cuatro vértices de la caja que rodea al objeto y que por tanto facilita mucho el proceso de detección.

El resultado de la aplicación de este método se puede ver en la imagen siguiente, donde tenemos detectada la **caja mínima** que envuelve al objeto en 2D:

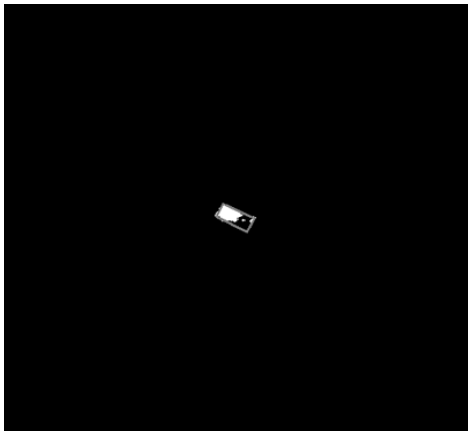


Ilustración 52 - Caja mínima 2D ciclista.

El cálculo de la caja es bastante aproximado como se verá a continuación cuando veamos la caja detectada en 3D.

Sin embargo, antes de pasar a los resultados en 3D conviene explicar por qué no se ha empleado este método a la hora de calcular el ángulo de los coches. A pesar de que pudiese parecer mucho más sencillo aplicar este método y a primera vista los resultados deberían ser bastante buenos existe un problema que ya se ha podido apreciar parcialmente en la anterior imagen.

A la hora de calcular la caja mínima si la detección del coche no es completa, la caja mínima que se detectaría estaría siempre girada y no correspondería con la caja real del vehículo. Y considerando que en la mayoría de los casos las detecciones de los coches son parciales cometeríamos errores en prácticamente todas las detecciones.

Para que aclarar la explicación a continuación se mostrará lo que ocurre si lo aplicamos directamente sobre uno de los ***clusters*** de los coches:

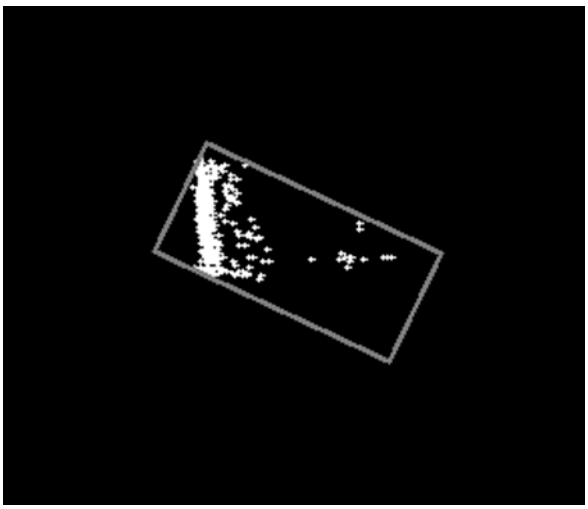


Ilustración 53 - Coche detectado mediante caja mínima en 2D.

Se puede observar como la caja detectada se calcula mal debido a que solo se ha detectado bien la parte trasera y un lateral del coche, que es uno de los casos más comunes de las detecciones que obtenemos.

Tras esta aclaración continuamos con la detección de los ciclistas. Una vez tenemos detectados los vértices en 2D necesitamos realizar la transformación a las coordenadas en 3 dimensiones. Esta transformación será llevada a cabo de la misma manera que se realizó la transformación durante la detección de los coches, para ello se empleará la siguiente fórmula:

$$Vértice\_3D = \left(\frac{(Vértice\_2D - Tamaño\_Imagen)/2}{Resolución} \right) + Centroide$$

Una vez realizada esta operación con todos los vértices del objeto ya tendremos información suficiente para poder pintar la caja detectada en 3 dimensiones, que podremos ver en la siguiente imagen:

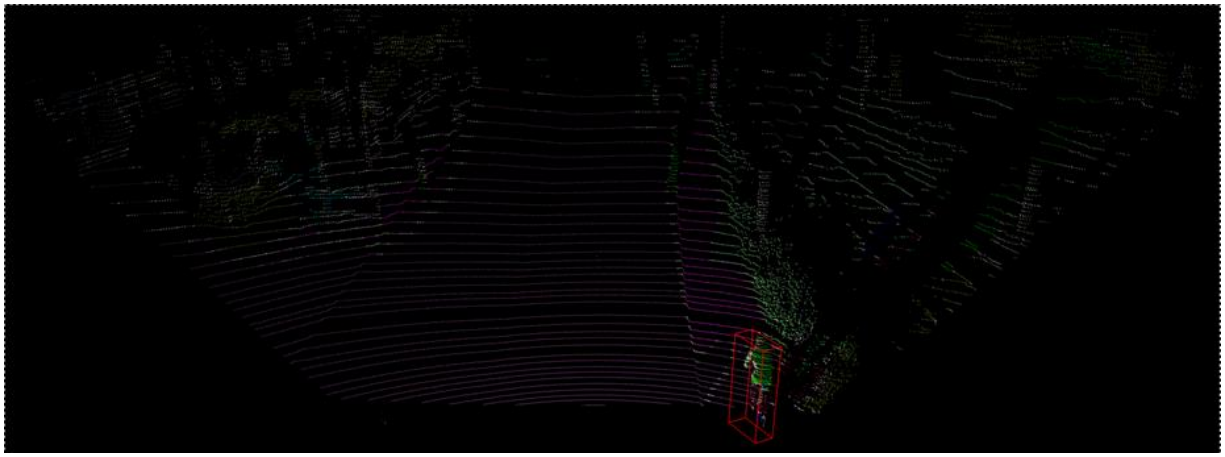


Ilustración 54 - Caja mínima 3D ciclista.

Como podemos observar el resultado obtenido es bastante preciso si tenemos el suficiente número de puntos en el ciclista, y debido a los filtrados que hemos aplicado sobre los ***clusters*** se han logrado eliminar las proyecciones sobre el suelo de manera que no hemos obtenido falsos positivos. Aunque habrá casos en los que sea complicado eliminar estas proyecciones si alcanzan la suficiente altura.

Una vez obtenidos y representados los datos solo queda establecer el ***score*** e introducir los datos en el vector de objetos.

El cálculo del ***score*** será igual que el que se realizó para los vehículos, variando la anchura y longitud con la que compararemos nuestra detección. De igual manera que en los coches se calculará el porcentaje de anchura y longitud detectadas con una anchura y longitud preestablecidas y en función de estos valores se le asignará mayor o menor puntuación sobre 1.

El último paso será llamar a la función ***Add_Objects()*** pasándole todos los datos necesarios al igual que se explicó en el apartado anterior.

5.3.5. Detección de peatones

Para la detección de peatones se empleará exactamente el mismo método usado para detectar a los ciclistas, a excepción de los parámetros empleados para filtrar los ***clusters*** y determinar la score.

Nuevamente, se empezará copiando la nube de puntos original y filtrando los puntos por color para obtener la nube con todos los puntos pertenecientes a peatones. Para ello se empleará la función ***Filtrado_Puntos_Clolor()***, esta vez con los valores de color RGB **220, 20, 60**:

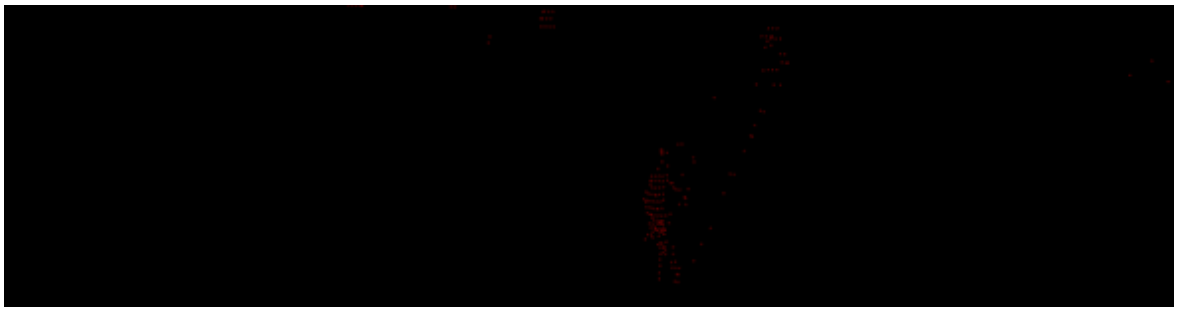


Ilustración 55 - Nube de puntos de peatones filtrada.

Una vez filtrada la nube se extraerán los diferentes ***clusters*** con la función ***extraer_clusters()*** empleando como parámetros **0.4** metros de distancia entre puntos y un mínimo y máximo de **7** y **1000** puntos respectivamente para cada ***clúster***.

Tras finalizar la función tendremos un vector con todos los ***clusters*** pertenecientes a las personas el cual recorreremos e iremos llamando a la función ***pedestrian_detection()***, que es la que se encargará de realizar las detecciones al igual que en el caso de los coches y los ciclistas.

5.3.5.1. Función Pedestrian_Detection()

De igual forma que en las ocasiones anteriores primero se recorrerá la nube de puntos de cada *clúster* para poder obtener los **puntos máximo y mínimo** en cada una de las coordenadas, el **centroide**, la **anchura**, **longitud** y **altura** del *clúster*.

Posteriormente se filtrará de manera similar a la detección de los ciclistas eliminando todos los ***clusters*** que no alcancen al menos 1 metro de altura o 30cm de ancho y largo.

Se proyectará el *clúster* sobre el plano y se obtendrá la caja mínima que rodea al peatón mediante el método ***minAreaRect()*** de **OpenCV**, ya que al igual que pasaba con los ciclistas el cálculo del ángulo mediante la ***transformada de Hough*** es demasiado impreciso. El resultado tras este proceso sería el mostrado en la siguiente imagen:

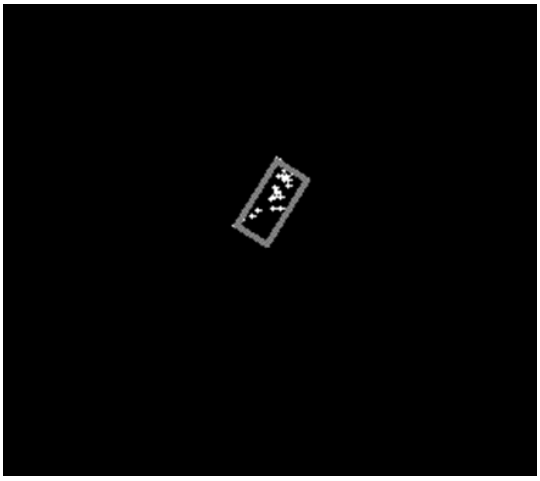


Ilustración 56 - Peatón proyectado sobre el plano.

El resultado es parecido al obtenido cuando detectábamos a los ciclistas. El siguiente paso será transformar los puntos de los vértices de la caja mínima a las coordenadas del LIDAR empleando la misma fórmula que se usó en el caso de los ciclistas:

$$Vértice\_3D = \left(\frac{(Vértice\_2D - Tamaño\_Imagen)/2}{Resolución} \right) + Centroide$$

Una vez tenemos todos los datos en 3D podremos pintar la caja mínima, para hacernos una idea de los resultados obtenidos las siguientes imágenes muestran el escenario original de la cámara y las detecciones en la nube de puntos:

Ilustración 57 - Escenario original de detección de peatones.

La siguiente imagen mostrará el resultado tras procesar el escenario, las **cajas mínimas** de los peatones se muestran en color **verde**:

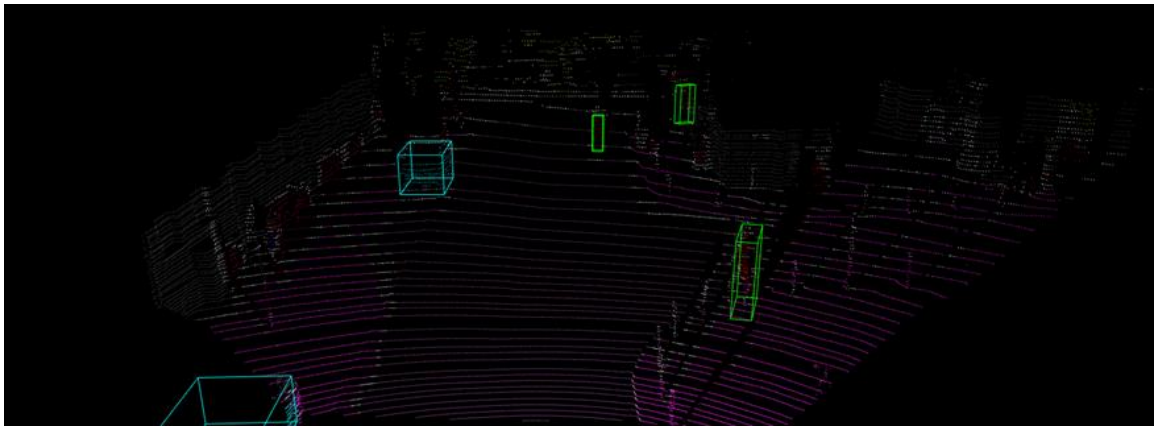


Ilustración 58 - Detección de peatones en la nube de puntos.

Finalmente, solo quedará calcular la ***score*** de igual manera que se hacía con los coches y los ciclistas basándonos en la longitud y anchura del *clúster* detectado.

Una vez tengamos todos los parámetros calculados se le pasarán a la función ***Add_Objects()*** que añadirá el nuevo objeto con todas sus características al vector de objetos.

5.3.6. Mejora de Detecciones

Una vez tenemos todos los datos de los diferentes objetos generados el siguiente paso será realizar un post-procesado sobre los mismos para conseguir mejorar las detecciones.

Uno de los principales problemas que surgen a la hora de realizar la detección de los objetos es el suelo. En la mayoría de las detecciones que hemos estado realizando hasta el momento se ha detectado parte del suelo junto con cada uno de los objetos debido a que al realizar la proyección de la imagen de la CNN sobre el LIDAR el color de los objetos por lo general ocupa más que el objeto real. Esto nos lleva a detectar parte del suelo como si fuese el objeto y por tanto la detección del ángulo del objeto no es del todo precisa y la caja mínima tampoco se corresponde con la caja mínima real del objeto.

En las siguientes imágenes se muestra un ejemplo de estas proyecciones sobre el suelo y de lo que ocurriría a la hora de detectar los objetos si no se realizase el post-procesado:

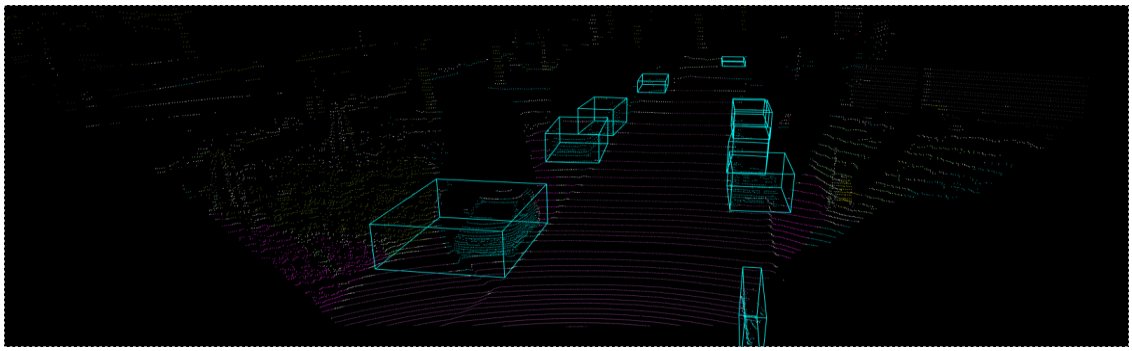


Ilustración 59 - Detección sin post-procesado.

En la siguiente imagen se ve con más detalle uno de los ejemplos más claros que aparecen en la imagen anterior, en el que debido a que existe una proyección sobre el suelo el tamaño del objeto detectado es mucho mayor del que es realmente:

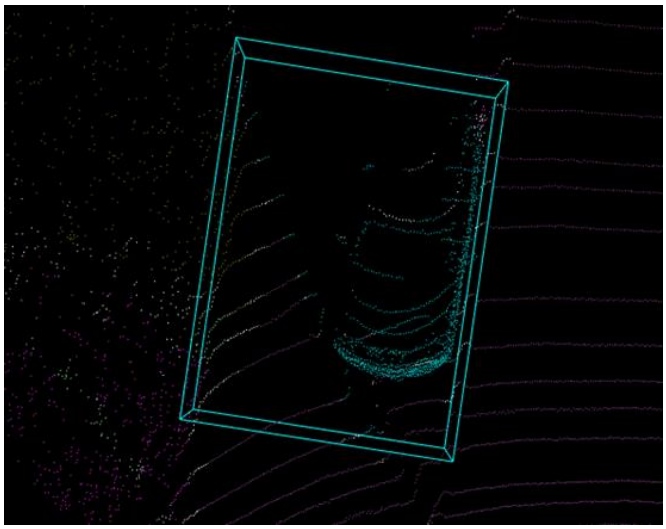


Ilustración 60 - Coche antes del post-procesado.

Si nos fijamos en la proyección sobre el suelo que usamos para determinar el ángulo y tamaño de la caja mínima del objeto, se vería aún más claramente que existe una gran parte del suelo que no podemos separar de nuestro clúster debido a que se encuentra demasiado cerca de este:

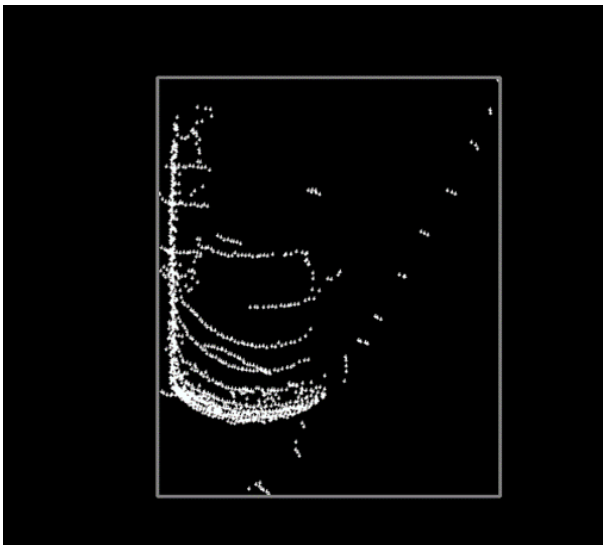


Ilustración 61 - Proyección sobre el suelo antes del post-procesado.

Todos los puntos detectados en el suelo están a la suficiente distancia como para que no podamos separarlos durante la separación de los ***clusters***, lo que acaba sucediendo en una gran mayoría de las imágenes que procesamos.

La primera solución que se tomó para evitar estas proyecciones en el suelo fue eliminar todo el suelo de la nube de puntos mediante una extracción del plano detectado en la nube de puntos.

La extracción planar se realizaba detectando el plano principal de la nube de puntos y filtrándolo empleando métodos de la clase ***pcl::EuclideanClusterExtraction***, para posteriormente eliminarlo. Este primer método no llegó a ser muy eficaz ya que no conseguía eliminar el suelo por completo, y debido a las características del filtro empleado eliminaba algunos puntos de los objetos detectados que no se encontraban en el plano del suelo, lo que acababa empeorando las detecciones en vez de mejorarlas.

La segunda aproximación consistió en realizar un **filtrado por altura**, de manera que se eliminaban todos los puntos del suelo que no llegasen a una altura de aproximadamente 20cm sobre el suelo. Eliminar 20cm de la parte inferior del *clúster* de un coche no supone ningún problema a la hora de realizar su detección, ya que a esa altura solo se elimina una pequeña parte de las ruedas y se queda intacta la mayor parte del coche que es la que nos interesa para realizar la detección. También se decidió realizar una compensación sobre la altura reduciendo nuevamente esos 20cm eliminados, de tal forma que el único parámetro que quedaba afectado por este filtrado volvía a alcanzar un valor muy aproximado al que tenía originalmente.

Este segundo método mejoró considerablemente los resultados en general, sin embargo no era lo suficientemente bueno para mejorar las detecciones de los objetos que se encontraban a gran distancia. Esto es debido a que el terreno no es completamente plano y a la inclinación del LIDAR, que por lo general tiene una pendiente negativa, lo que supone que la parte del suelo que se elimina solo corresponda con las inmediaciones de nuestro vehículo, y a cierta distancia la carretera no se llega a eliminar.

Un ejemplo del resultado se muestra en las siguientes imágenes:

- **Imagen original:**

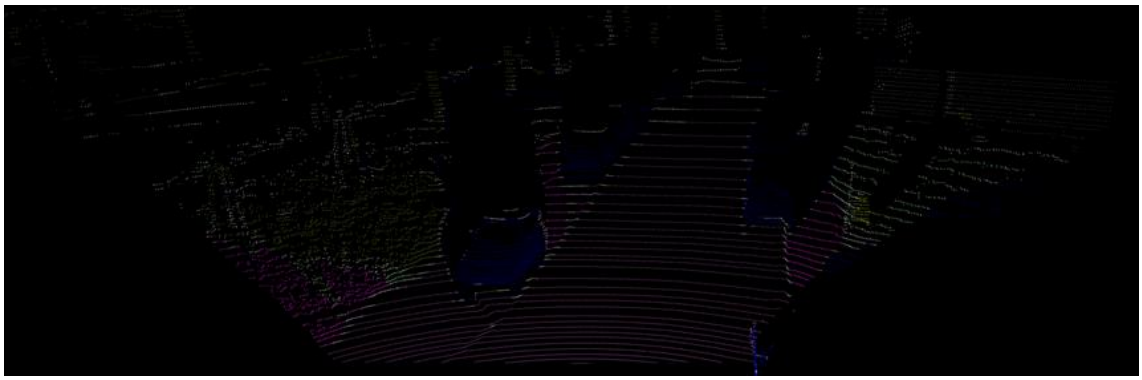


Ilustración 62 - Imagen antes de filtrar el suelo.

- **Imagen filtrada:**

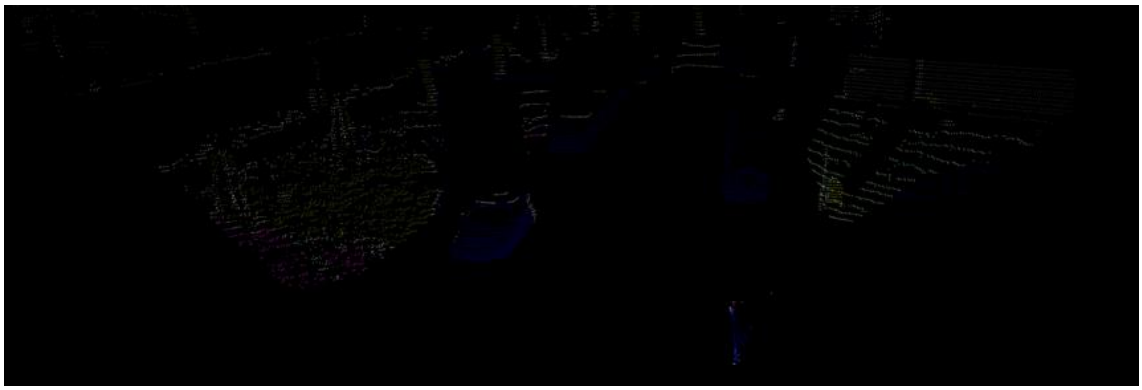


Ilustración 63 - Imagen tras filtrar el suelo.

El suelo queda completamente filtrado cuando nos encontramos cerca del vehículo, pero como se puede apreciar al fondo de la imagen existen partes de la carretera que no han sido eliminadas debido a la inclinación de la misma.

De igual manera, en las ocasiones en las que la inclinación de la carretera fuese mínimamente positiva se acababa eliminando una mayor parte del *clúster* de la que se deseaba y por tanto la detección no llegaba a ser tan precisa como debería e incluso en algunos casos extremos los coches eran eliminados por este motivo.

Finalmente, con el objetivo de adoptar una medida mejor que la anterior se decidió eliminar el suelo de manera independiente para cada uno de los ***clusters***. De esta forma no existen problemas relacionados con eliminar una mayor o menor parte del *clúster* dependiendo de la distancia a la que se encuentre el mismo y de la inclinación de la carretera.

Esta última opción es la que se utiliza en durante el post-procesado. En la primera detección se detectan las cajas mínimas sin eliminar de ninguna forma el suelo, y es durante este segundo procesado cuando se quita el suelo de cada *clúster* y se vuelve a llamar a la función que detecta las cajas mínimas.

Como los objetos ya han pasado por un proceso de detección se tiene información sobre sus alturas máxima y mínima. La altura mínima será siempre una medida válida del vehículo ya que a pesar de haber podido detectar el suelo el vehículo siempre está en contacto con el mismo, lo que mejorará notablemente la detección de la altura respecto a las anteriores aproximaciones que eliminaban siempre una pequeña parte del vehículo.

En este caso se empleará esa altura mínima para tomarla como referencia y eliminar 20cm de la parte inferior del *clúster* con un filtro igual al que se empleó en la aproximación anterior. Eliminar estos 20cm del suelo conseguirá que el *clúster* quede solo compuesto por puntos del objeto a analizar y no del suelo, lo que mejorará considerablemente la detección del ángulo y de la caja mínima que envuelve al mismo, y como teníamos el valor de la altura anterior, que es el valor de altura real podemos recuperarlo tras la segunda detección y de esta manera conseguir una buena detección tanto de la anchura, ángulo y longitud del vehículo como de la altura del mismo, mejorando de esta manera notablemente los resultados obtenidos con cualquiera de los otros métodos que se probaron con anterioridad.

En la siguiente imagen se puede ver la proyección del mismo *clúster* que se mostró anteriormente sin filtrar tras haber realizado la primera parte del post-procesado y haber eliminado el suelo:

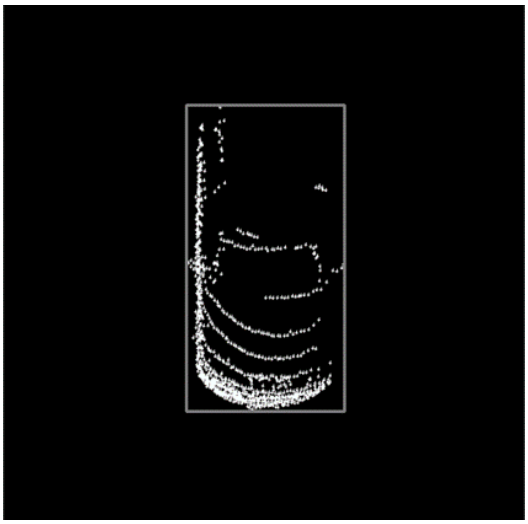


Ilustración 64 - Proyección sobre el suelo tras post-procesado.

Como se puede apreciar en la imagen, esta vez la detección es realmente precisa.

La función de post-procesado realizará una segunda interacción con cada uno de los nuevos objetos detectados comprobando si el tamaño de los objetos se ajusta al de un coche o si se ha detectado un objeto demasiado grande que en ningún caso podría ser un coche. En muchas ocasiones cuando esto ocurre es debido a que dos coches se encuentran demasiado juntos y la función que separa los ***clusters*** los considera como uno solo.

A continuación, se muestra un ejemplo de uno de estos casos. La primera imagen muestra la detección en 3D que hemos realizado donde se puede apreciar claramente como se han juntado dos coches en el mismo *clúster*, la segunda muestra la misma imagen desde una vista de pájaro, donde la caja de color azul es la detección que hemos realizado proyectada sobre el plano:

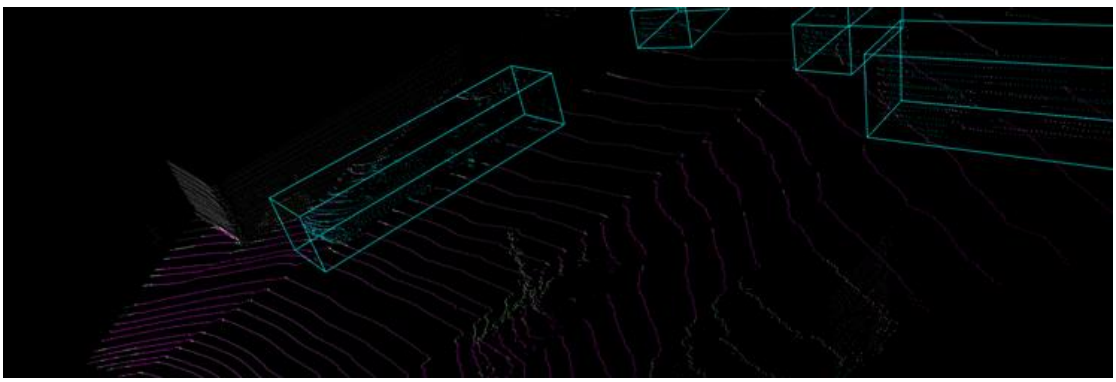


Ilustración 65 - Detección de dos clusters como un mismo objeto.

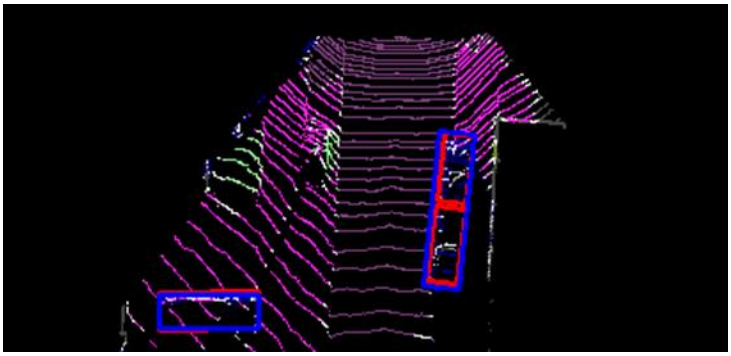


Ilustración 66 - Vista de pájaro.

Para solucionar este problema durante el ***post-procesado*** se han seguido unos pasos muy parecidos a los que se emplearon para eliminar el suelo. En primer lugar, se detectará que ***clusters*** sobrepasan una determinada anchura y longitud y por tanto deben volver a procesarse, tras esto se eliminarán 40cm de la parte inferior de dicho *clúster*, ya que en la mayoría de las ocasiones estos se juntan cerca del suelo debido a alguna variación de la altura del terreno. Una vez filtrado el *clúster* se llamará a la función de extracción de ***clusters*** como se hacía al inicio de las detecciones pasándole esta vez un valor menor de cercanía entre puntos para considerarlos como parte del mismo *clúster* con el objetivo de que le sea más fácil separar los dos *clusters* que se encuentran en la nube de puntos. Esta función devolverá los dos *clusters* que serán pasados a la función de detección y procesados por ella devolviendo la caja mínima que envuelve a cada uno de ellos. Finalmente, se eliminará el objeto original del vector de objetos y se añadirán los dos nuevos objetos detectados.

En las siguientes imágenes se puede apreciar el resultado tras haber procesado nuevamente la nube de puntos del ejemplo anterior. Se puede apreciar como el algoritmo ha separado los dos objetos esta vez:

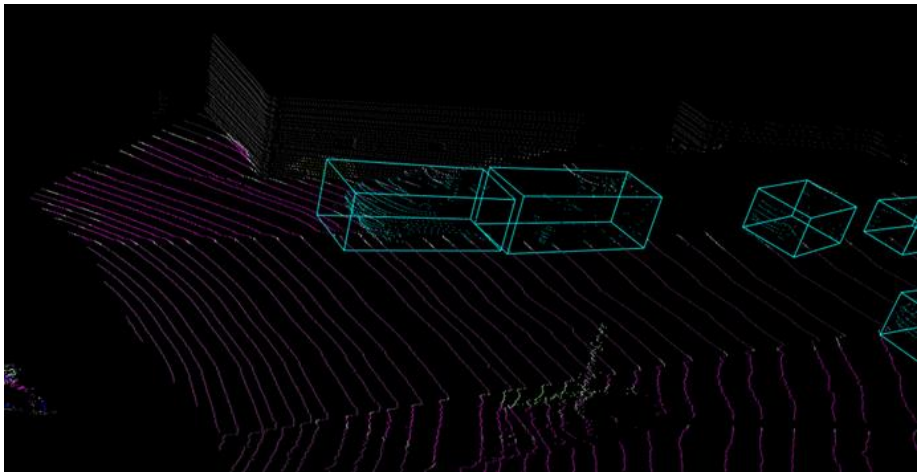


Ilustración 67 - Separación de clusters.

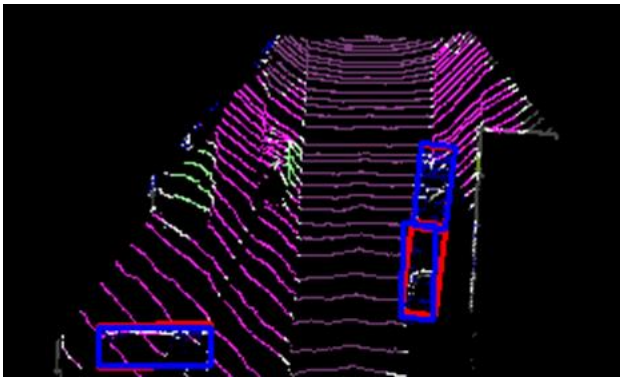


Ilustración 68 - Separación de clusters en vista de pájaro.

El siguiente paso que será llevado a cabo durante la fase de **post-procesado** será la eliminación de los *clusters* que sean excesivamente grandes y que no se hayan podido separar en el paso anterior. También se eliminarán los objetos que se encuentren a menos de 20m del coche pero tengan una altura inferior a 80cm o cualquier otro elemento detectado que tenga una altura inferior a 40cm.

La mayoría de los elementos que sobrepasen las medidas máximas establecidas en 4 metros de anchura o 6 metros de longitud serán proyecciones sobre paredes que habrán sido causadas debido a que un gran número de coches se han encontrado aglomerados muy cerca unos de otros y con una pared detrás. En el caso de la altura, la mayoría de los objetos que se encuentren a menos de 20 metros y tengan menos de 80cm o menos de 40cm en total serán debidos en su mayoría a fallos de la CNN o a proyecciones de los *clusters* de los coches sobre otros objetos que se encuentren en cada uno de los escenarios encontrados.

En algunas ocasiones, el hecho de eliminar alguno de los *clusters* empleando estos criterios nos llevará a eliminar el *clúster* de un coche real que no haya sido detectado correctamente o el *clúster* de dos coches juntos que no hayamos conseguido separar con los procesados anteriores. Sin embargo, al aplicar estos cambios se produce una mejora en las detecciones en general ya que es mucho mayor la cantidad de falsos positivos que se eliminan que la cantidad de coches reales que eliminamos al aplicar este filtrado.

En la siguiente imagen se puede apreciar una de estas proyecciones sobre una pared que ha sido detectada y que tiene un tamaño de más de 5 metros de largo. Esta proyección ha sido causada debido a los errores que existen al **proyectar** el color de la CNN sobre la nube de puntos ya que se han coloreado puntos de la pared que se encontraba detrás de los coches reales:

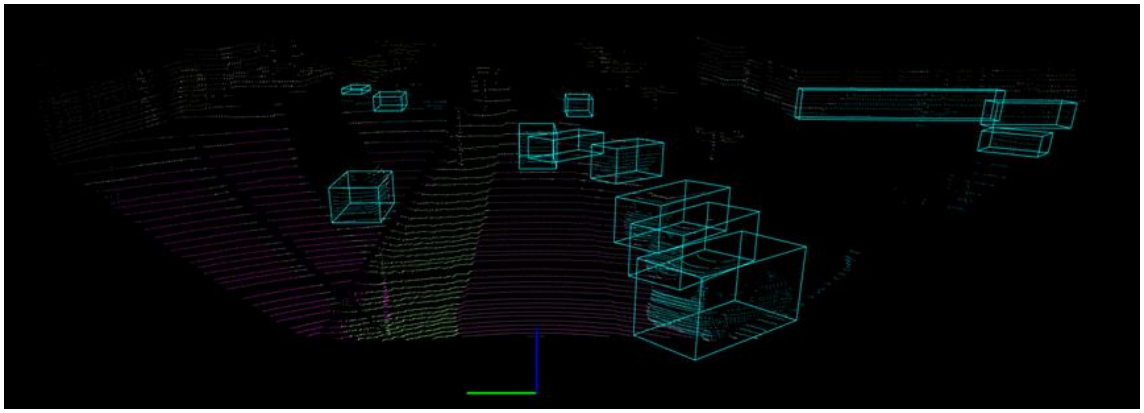


Ilustración 69 - Clusters demasiado grandes antes del post-procesado.

Tras haber realizado el filtrado por tamaño hemos obtenido el siguiente resultado:

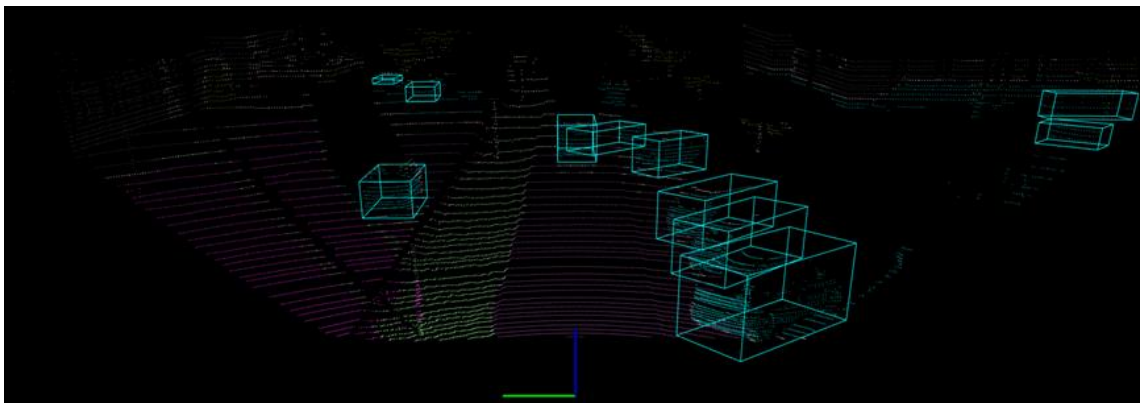


Ilustración 70 - Eliminación de clusters demasiado grandes tras el post-procesado.

La proyección de gran tamaño ha sido eliminada de manera correcta.

Finalmente, el último paso que se realizará durante el ***post-procesado*** será una corrección de la altura de detección de los objetos. En la mayoría de las ocasiones los *clusters* de los objetos no se detectan completamente y falta una parte de la zona superior o inferior del coche debido a la distancia a la que se encuentran los vehículos. Esto lleva a que la altura de los vehículos que detectamos sea por lo general inferior a la real.

Por tanto, durante esta última corrección se tratarán todos los coches cuya altura detectada sea menor de 1.2m, siendo esta altura demasiado pequeña para un coche, y se les añadirá a estos casos la altura máxima necesaria para que el *clúster* alcance al menos una altura de 1.6 metros.

Esta pequeña corrección mejora en todos los casos la precisión de la detección del objeto, con algunas excepciones en las que la altura final establecida es ligeramente mayor que la real, no suponiendo esto ningún problema grave para la detección.

5.3.7. Cálculo del nivel de oclusión

Esta funcionalidad ha sido añadida con el objetivo de obtener el parámetro de oclusión de los objetos que hemos detectado, ya que es necesario para obtener los resultados de las

detecciones con la base de datos de **KITTI**. Además, la oclusión nos servirá para ajustar aún más el parámetro de ***score*** que indica la seguridad que tenemos sobre una de las detecciones que hemos realizado, de forma que se penalizará el ***score*** de los objetos que tengan una oclusión mayor.

Antes de nada, es necesario aclarar que lo que consideramos como **oclusión** es el nivel de **ocultamiento** que sufre un objeto debido a que otros objetos se encuentran en la trayectoria entre este y los sensores impidiendo una correcta detección del mismo.

La base de datos de KITTI determina el nivel de oclusión mediante tres diferentes niveles que se expresan mediante un parámetro de tipo entero y que indican los siguientes niveles de visibilidad del objeto:

- **0**: Completamente visible.
- **1**: Parcialmente ocluido.
- **2**: Muy ocluido.
- **3**: Desconocido.

En nuestro caso solo se han empleado los valores **0**, **1** y **2**, ya que para considerar que un objeto es desconocido debería estar completamente ocluido y por tanto no deberíamos haberlo detectado.

Para realizar este proceso ha sido necesario primero haber detectado todos los objetos, ya que son estos los que se ocluirán los unos a los otros y será necesario conocerlos todos antes de ver como se ocultan entre ellos.

El cálculo de la oclusión que se ha realizado ha consistido en calcular el ángulo en el que se encuentra cada uno de los objetos respecto al centro de coordenadas y su distancia al mismo origen de coordenadas, con el objetivo de determinar que objeto era el que ocluía y cual el ocluido. Finalmente, basándonos en la diferencia del ángulo en el que se encontraba cada objeto se ha determinado que un objeto se encuentra muy ocluido cuando el ángulo que lo diferencia de otro objeto más cercano al centro es menor de 3º, y que está parcialmente ocluido si el ángulo es de 5 º o menos.

Puede parecer un cálculo muy sencillo, ya que no se han llegado a considerar en ningún momento el tamaño de los objetos y la parte total de objeto que queda oculta detrás del objeto que se encuentra más cerca. Sin embargo, cabe destacar que en los casos en los que existe oclusión raramente se detecta la parte del objeto que está ocluida ya que por la propia definición de oclusión esa parte no es detectada por los sensores y por tanto es imposible determinar de esa forma si un objeto está realmente ocluido o no.

En cualquier caso, el cálculo de la oclusión es lo suficientemente preciso llevado a cabo de esta manera para determinar que objetos se encuentran realmente detrás de otros, y dado que el valor de este parámetro no es de vital importancia para la calidad de las detecciones de los objetos no se le ha dado una gran importancia al cálculo del mismo.

5.3.8. Proyección 3D a 2D

Uno de los últimos pasos a realizar para obtener todos los datos que necesitamos para compararlos con la base de datos de **KITTI** y para poder determinar la precisión de las

detecciones es obtener las ***cajas mínimas*** que envuelven a los vehículos en las coordenadas de la cámara.

En uno de los primeros puntos ya se explicó como se obtienen algunas de estas cajas en 2D solo con la información de la CNN, sin embargo, ya se vio que existían ciertas limitaciones si solo se empleaban los datos de la red *convolucional*, ya que al no tener información sobre la distancia varios coches se juntaban en uno solo cuando se intentaba establecer la caja mínima.

Ya se comentó entonces que esa información solo nos serviría para detectar los objetos que se hubiesen detectado exclusivamente mediante técnicas 2D y que no coincidiesen con ningún objeto que hubiese sido detectado mediante la información de la nube de puntos del LIDAR.

Para volver a hacernos una idea del problema encontrado en la siguiente imagen se puede apreciar como a pesar de tratarse de varios coches a la vez, por el hecho de estar todos juntos es imposible que los separemos si solo tenemos en cuenta la información de la CNN, y por tanto el objeto detectado no es válido:

Ilustración 71 - Error de detección en 2D.

Para lograr realizar correctamente la detección de estos vehículos en la imagen será necesario detectarlos primero en el espacio con la información del LIDAR y realizar posteriormente la proyección de los puntos de un sistema de coordenadas al otro ayudándonos de las matrices de calibración.

Para realizar la proyección de los vértices de la caja se seguirán los siguientes pasos:

1. Se almacenarán en una matriz los 8 vértices de la caja mínima que ha sido detectada.
2. Se empleará la siguiente fórmula para obtener una matriz con los puntos en coordenadas de la cámara y en metros:

   $$Punto\_2D = Matriz\_P \times Matriz\_R \times Matriz\_T \times Punto\_3D$$

   Donde las matrices P, R y T serán las matrices de calibración y transformación proporcionadas por la base de datos, **Punto_3D** será la matriz conteniendo los puntos de los 8 vértices de la caja, y **Punto_2D** acabará conteniendo los 8 vértices en coordenadas de la cámara en metros.
3. El siguiente paso será transformar los vértices en 2D de metros a píxeles de la cámara, lo que se logrará gracias al factor de calibración que se habrá calculado y almacenado también en la matriz de salida. Dividiendo cada uno de los puntos por ese factor de calibración se obtendrán las coordenadas **X** e **Y** de cada uno de los vértices de la caja en coordenadas de la imagen y en píxeles.

4. A continuación, será necesario elegir cuales de los 8 vértices se usarán en la proyección al 2D, ya que en 2D solo se tendrá un cuadrado con 4 vértices. Para ello se recorrerán los 8 puntos seleccionando las coordenadas **X** e **Y** máximas y mínimas y quedándonos con ellas para dibujar la caja en dos dimensiones que envolverá al objeto.
5. Se comprobará que ninguno de los índices se salgan de las coordenadas de la imagen, lo que puede pasar si los objetos están muy cerca del origen, ya que algunos de sus puntos pueden ser captados por el LIDAR pero no por la cámara, y se fijarán los puntos máximos o mínimos de la imagen en caso de que esto ocurra.
6. Finalmente, se introducirá el valor de los cuatro vértices detectados en coordenadas de la cámara a la estructura que contiene toda la información del objeto detectado y se dibujarán líneas en la imagen que unan los vértices para visualizar los resultados.

Tras esto, se consigue finalmente la información en 2D de los objetos que hemos detectado en 3D, y se pueden separar los objetos a pesar de que la CNN pinte toda la zona del mismo color, obteniendo los siguientes resultados en la imagen que se mostró al principio de este punto:

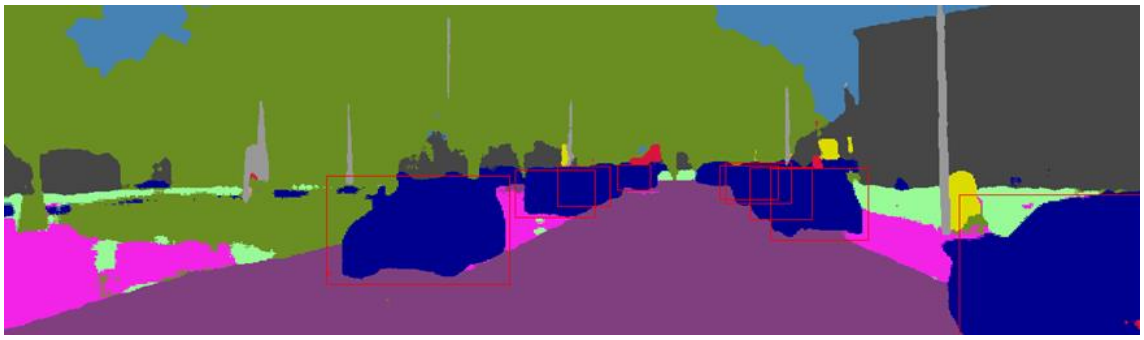


Ilustración 72 - Separación de clusters en 2D con ayuda del LIDAR.

Como se puede ver, los bloques de objetos que no se habían logrado separar empleando solo la información de la cámara han sido correctamente separados en este caso. Para que se pueda apreciar mejor la separación de cada uno de los coches en los puntos donde se juntan varios bloques a continuación se muestra la misma información de las cajas sobre la imagen original que tomó la cámara:

Ilustración 73 - Separación de clusters sobre la imagen de la cámara.

Tras haber realizado este último paso ya tenemos todos los datos necesarios de los objetos detectados para poder publicarlos y evaluarlos, que serán los próximos pasos que serán llevados a cabo.

5.3.9. Fusión de datos 2D y 3D

Antes de publicar los datos será necesario determinar si nos quedaremos con los datos que se han detectado solo mediante la información de la cámara o si esos mismos objetos han sido detectados también por el LIDAR y por tanto serán descartados.

En caso de que un objeto haya sido detectado mediante la información de la nube de puntos será esta información del objeto con la que nos quedaremos y se descartará la detección realizada solo mediante la cámara.

Siempre se obtendrá más precisión en las detecciones cuando han sido realizadas mediante la información del LIDAR, sin embargo, en algunas ocasiones algunos objetos que se detectan en la cámara son eliminados en la detección en 3D debido a la distancia o a una mala detección del *clúster* de la nube de puntos, será en estos casos en los que nos quedaremos con la información de la cámara calculada en los primeros puntos que se han explicado en este trabajo a pesar de que no sea del todo precisa al proyectarla a las coordenadas del LIDAR.

El proceso para decidir si nos quedaremos con el objeto detectado mediante cámara o LIDAR será muy sencillo. Una vez tengamos las proyecciones en la imagen empleando ambos métodos se comparará si alguno de los objetos detectados mediante la cámara se superpone con alguna de las detecciones del LIDAR. En caso de que sea así se eliminará el objeto detectado por la cámara y nos quedaremos con el objeto del LIDAR. En caso de que el objeto detectado por la cámara no se superponga con ningún otro nos quedaremos con él y se le otorgará una ***score*** muy baja, ya que la información en el 3D que tendremos no será muy precisa.

5.3.10. Guardado y publicado de datos

Una vez se tienen todos los datos recopilados el último paso será **publicarlos** en un ***topic*** para que otros módulos de ROS puedan leerlos y guardarlos en un fichero de datos en el formato especificado por **KITTI** para poder comprobar los resultados obtenidos.

El publicado de datos en ROS nos permitirá publicar directamente un array de estructuras de datos con los objetos que hemos detectado.

Para ello será necesario definir dos nuevos tipos de mensaje que serán incluidos en la carpeta ***"/msg"*** dentro del fichero del proyecto. En esa carpeta se introducirán dos ficheros con la extensión *".msg"*. uno contendrá exactamente los mismos datos que la estructura de datos que hemos empleado para guardar todas las características de los objetos, y el otro contendrá un array inicializado con el nombre del fichero anterior.

Una vez definidos los tipos de mensajes que se van a utilizar será necesario inicializar un objeto de este tipo en nuestro programa de la siguiente manera: "***Nombre_Proyecto::Nombre_Mensaje Objeto"***.

Así crearemos un objeto de cada uno de los tipos definidos en los ficheros y cuyos atributos podrán ser accedidos como los de cualquier otro objeto.

El siguiente paso será recorrer el vector de objetos detectados que ya teníamos creado e ir introduciendo los datos de cada uno de los objetos en cada uno de los elementos del array de objetos que será empleado como mensaje. Tras ello, se creará un objeto del tipo ***ros::Publisher*** inicializándolo con el tipo de mensaje que publicará y con un dato del tipo *string*, que contendrá

el nombre del ***topic*** donde se van a publicar los datos. En este caso el ***topic*** será ***"/kitti_pruebas/objetos".***

Finalmente solo quedará emplear el ***Publisher*** para publicar los datos que hemos introducido en el mensaje que hemos generado, esto se realizará mediante el método ***publish()*** del objeto, al que se le pasará como parámetro el mensaje que se va a publicar. Con todo esto nuestro programa empezará a publicar los datos en le ***topic*** que hemos establecido y cualquier otro nodo de **ROS** será capaz de leerlos en tiempo real.

La última acción que realizará el programa será guardar los datos que hemos obtenido en archivos con el formato ***".txt"*** conteniendo la información con la estructura requerida por el test de **evaluación de KITTI**.

Esta estructura ya fue explicada en el apartado de datos de entrada, pero como resumen el fichero tendrá la estructura explicada en los siguientes puntos:

- Existirá una línea de datos en el fichero por cada uno de los objetos detectados.
- Cada fichero tendrá el mismo nombre que la secuencia que se está procesando, por ejemplo "***0000001.txt***" será el nombre del fichero de datos obtenidos para el primer escenario.
- Cada línea contendrá en orden los siguientes datos:
    - Tipo de objeto ("**Car**", "**Pedestrian**", "**Van**", "**Cyclist**", "**Unknown**").
    - Truncamiento.
    - Oclusión.
    - Ángulo Alpha.
    - Píxeles para determinar la caja mínima en coordenadas de la cámara.
    - Dimensiones del objeto (Altura, anchura y longitud).
    - Centroide del objeto (x, y, z).
    - Ángulo de rotación del objeto.
    - Score.

Para guardar estos datos primero será necesario recorrer todo el vector de objetos que hemos ido almacenando, y para cada uno de ellos habrá que realizar una transformación del sistema de coordenadas, ya que el sistema de coordenadas que hemos empleado no es exactamente igual al que emplea **KITTI**, siendo su eje **X** nuestro eje **Y** negativo, su eje **Y** nuestro eje **Z** negativo y su eje **Z** nuestro eje **X**. También será necesario calcular el **ángulo Alpha** definido por KITTI, el cual podrá ser calculado de la siguiente manera:

$$Ángulo\_Alpha = Ángulo\_Vehículo + Ángulo\_Posición\_Vehículo$$

Siendo el ángulo del vehículo su ángulo de rotación calculado mediante la ***transformada de Hough*** y el ángulo de posición del mismo el ángulo respecto al origen de coordenadas en el que se encuentra su centroide, calculado mediante el **arcotangente** de su coordenada **Y** dividida entre su coordenada **X**.

Por último, se creará el fichero donde se van a introducir los datos con su el nombre que se indicó anteriormente y se irán añadiendo los datos de cada objeto detectado uno a uno.

Con este último paso se habrán realizado todas las tareas que realiza nuestro programa y solo faltará analizar los resultados obtenidos, tema que será tratado en el siguiente punto.

Cabe destacar que el fichero de datos deberá ser creado incluso si no se ha detectado ningún objeto en el escenario que se está procesando, ya que el test de evaluación proporcionado por **KITTI** necesitará poder leer todos los ficheros que se van a analizar a pesar de que estén vacíos.

5.4. Resultados

En este apartado se expondrán los resultados que se han obtenido al realizar la evaluación de los datos de las detecciones con las herramientas que nos proporciona KITTI.

Durante la realización del TFG se han ejecutado más de 100 test que han ido mostrando la mejora en la **precisión** de las detecciones cada vez que se implementaba una nueva mejora o cambio en el programa de detección.

La mayoría de estos test han sido realizados con 100 escenarios debido a que una muestra de este tamaño ya es lo suficientemente representativa y la variación al realizar *tests* con más imágenes es muy pequeña. En cualquier caso, también se han realizado varios *tests* con más de 1000 escenarios mostrando resultados muy parecidos a los obtenidos con solo 100 escenarios.

El formato de resultados que nos muestra el test está dividido en tres modalidades, que son ***easy***, ***moderate*** y ***hard***. Estas modalidades se refieren a la dificultad que existe a la hora de detectar los coches que pertenecen a cada una, de manera que un coche que se encuentre a una distancia muy grande pertenecerá a la modalidad ***hard,*** mientras que uno que se encuentre en un punto cercano y donde tengamos una buena visibilidad del mismo pertenecerá a la modalidad ***easy***.

Los resultados serán representados de dos formas diferentes:

- **En forma de gráfica**: Se mostrará una gráfica en la que aparecerán tres líneas de resultados diferentes, correspondiendo cada uno de ellos con las tres dificultades de detección antes mencionadas. La gráfica será representada en base a dos parámetros, la **precisión** de las detecciones y un parámetro llamado **recall**. El parámetro de **precisión** hace referencia a lo bien que se ajustan los resultados de las cajas que hemos detectado con las cajas mínimas reales que envolverían a los coches que se encuentran en los ficheros de ***ground truth*** que nos proporciona la base de datos. El segundo parámetro está relacionado con la cantidad de objetos detectados y los falsos positivos, de manera que este parámetro se verá penalizado si no detectamos como objetos algunos de los que si que están determinados en los ficheros de ***ground truth*** o si cometemos errores indicando que hay coches donde realmente no los hay.
- **En forma de porcentaje**: Esta segunda forma de representar los resultados será en realidad el área bajo la curva de la gráfica con la que se representan los datos. De esta manera tendremos una idea más exacta del conjunto de coches detectados y la precisión de las detecciones que hemos realizado.

Otro aspecto importante será que la evaluación estará dividida en tres tipos de detecciones diferentes:

- **Detecciones en 2D**: Este test solo tendrá en cuenta las cajas que se han detectado en la imagen en dos dimensiones, es decir, las proyecciones de las cajas detectadas en 3D

sobre la imagen en 2D. El test evaluará el nivel de solapamiento entre las cajas que hemos detectado y las que aparecen en los ficheros de *ground truth*, determinando que una detección es correcta cuando al menos el **50%** de la **superficie** de las dos cajas coincide.

- **Detecciones 3D**: Este segundo test analizará los resultados de las cajas mínimas en tres dimensiones que hemos detectado en el espacio. Se considerará que una detección es buena cuando coincida al menos el **50%** del **volumen** de la caja que hemos detectado con la que se encuentra en los ficheros de *ground truth*, siendo de esta forma más complicado obtener buenos resultados en este test que en el anterior debido a que se añade una tercera dimensión que implica una mayor probabilidad de error.
- **Detecciones en vista de pájaro**: El último test mostrará los resultados de la precisión de la detección de los coches si los viésemos desde una vista de pájaro, es decir, solo tendrá en cuenta la proyección de la caja que hemos detectado sobre el plano del suelo eliminando el factor de la altura. Nuevamente se considerará una buena detección cuando las dos cajas coincidan al menos en el **50%** de su **superficie**.

Primero se mostrarán una serie de gráficas correspondientes con los resultados obtenidos de las últimas fases del programa y que por tanto serán los mejores que se ha logrado obtener. Posteriormente, se realizará un análisis de las mejoras de los resultados a lo largo de las diferentes fases de desarrollo del programa.

A continuación, se muestra la primera gráfica de resultados que corresponde con la evaluación de las detecciones en 2D siguiendo los criterios que han sido explicados con anterioridad:

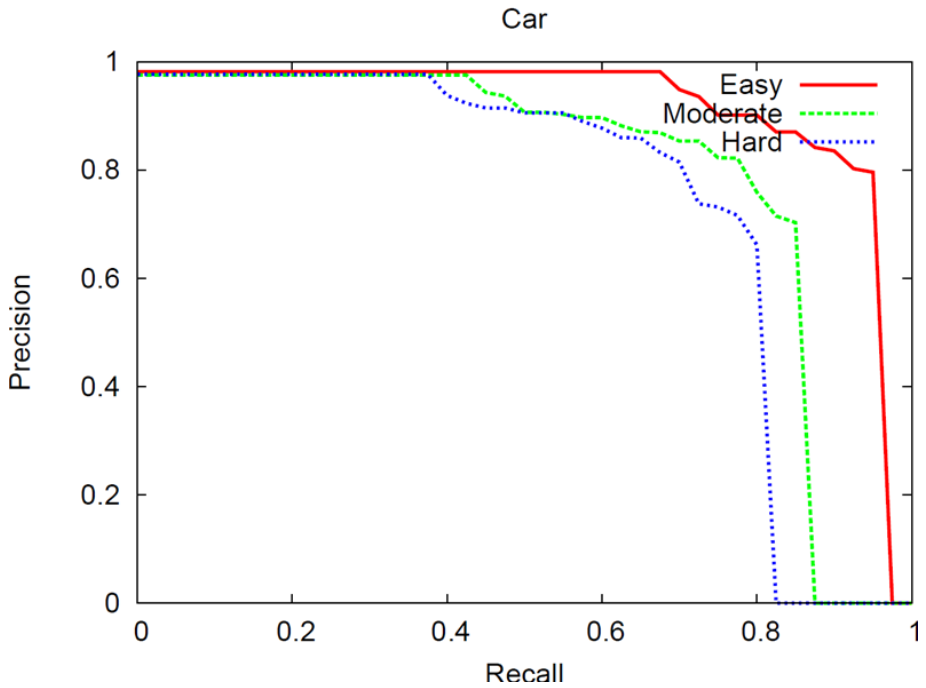


Ilustración 74 - Resultados detecciones 2D.

Las conclusiones que podemos extraer de esta gráfica son que los resultados han sido por lo general bastante buenos para los casos en los que las detecciones han sido consideradas **easy**, empeorando notablemente tanto la **precisión** como el **recall** en las detecciones consideradas **moderate** y **hard**.

La siguiente gráfica muestra los resultados de la evaluación de las detecciones de los coches en tres dimensiones:

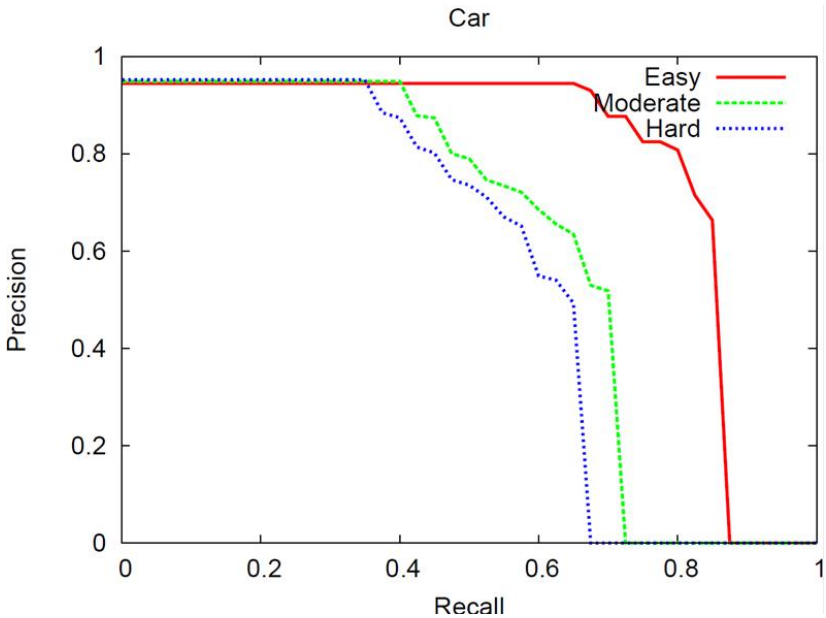


Ilustración 75 - Resultados detecciones 3D.

En este caso los resultados han sido peores que en las detecciones en 2D para todos los casos presentados, lo cual era de esperar debido a que al añadir la tercera dimensión es más fácil que se cometan errores y por tanto más complicado conseguir una precisión tan alta como en 2D. La diferencia entre los casos **easy** y **hard** y **moderate** ha sido mayor en este caso que en el anterior debido nuevamente a que en los casos difíciles el error cometido es mucho mayor por añadir esa tercera dimensión.

Finalmente, la última de las gráficas muestra el caso de las detecciones en vista de pájaro:

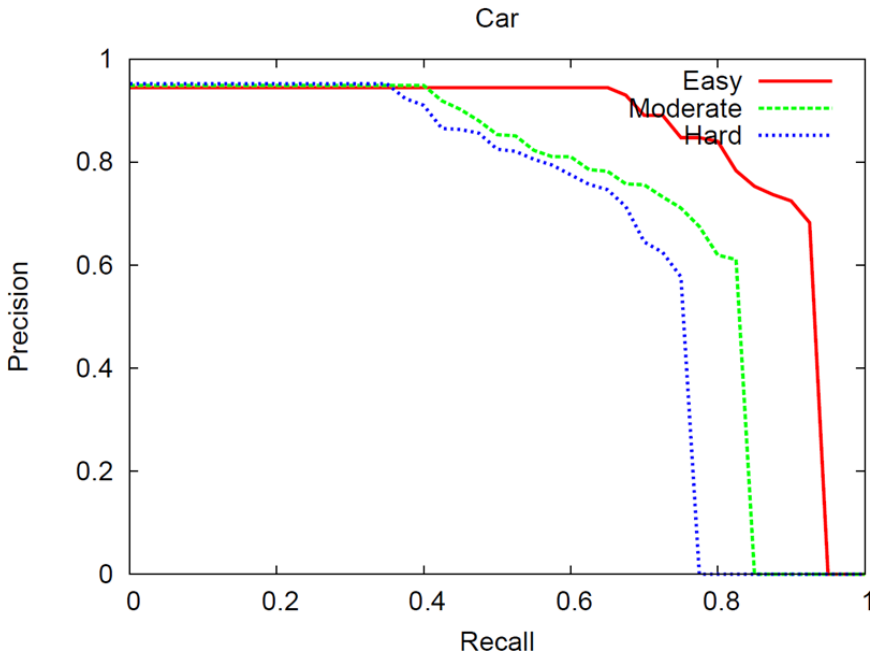


Ilustración 76 - Resultados detecciones en vista de pájaro.

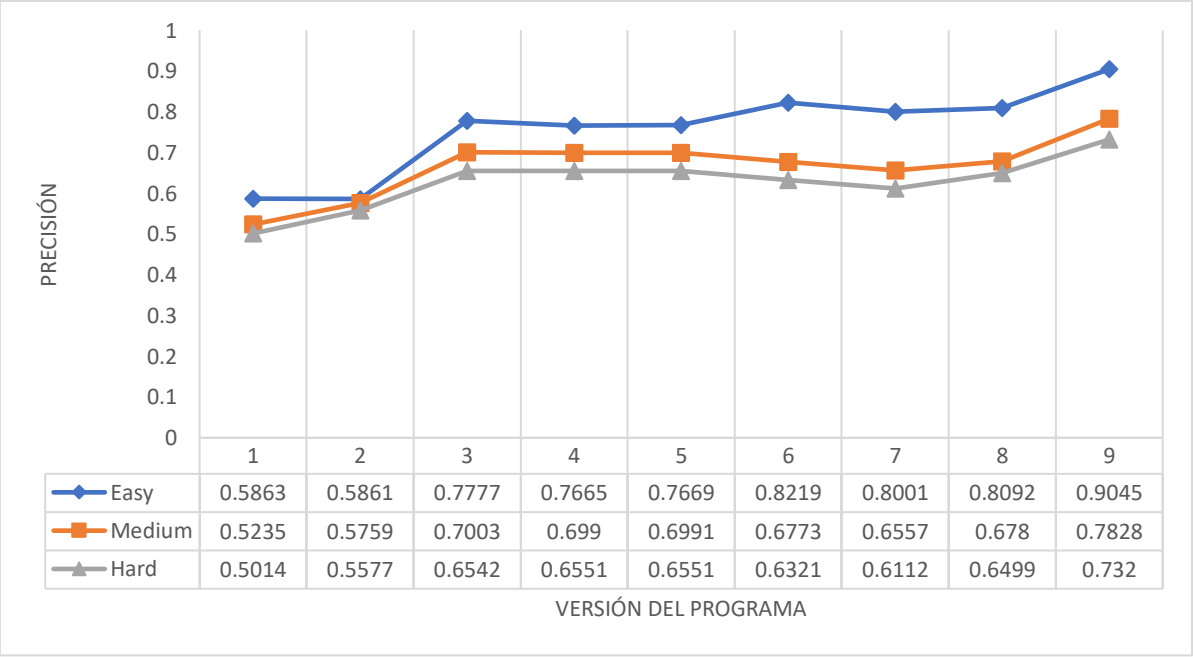
Este último caso nos muestra unos resultados que se encuentran entre los obtenidos en el caso de las detecciones en 2D y en el caso de las detecciones en 3D. Esto es debido a que respecto al 3D se ha eliminado la dimensión de la altura, eliminando de esta manera uno de los posibles factores de error. En cuanto al empeoramiento comparándolo con el 2D es debido principalmente a la detección del ángulo que se realiza mediante la ***transformada de Hough***. Al realizar las proyecciones a las coordenadas de la imagen una ligera variación en el ángulo del vehículo no supone un gran cambio en el espacio que este ocupa al proyectarlo sobre la imagen, sin embargo, esta variación del ángulo si que afecta notablemente cuando se trata de una proyección sobre el suelo como en el caso de la vista de pájaro.

A continuación, se muestra una tabla con los resultados correspondientes a las tres gráficas anteriores en forma de porcentaje. Cada uno de los porcentajes representa el área bajo la curva de cada una de las medidas que representan las gráficas:

Nombre del test/Dificultad	Easy	Moderate	Hard
Detección 2D	0.9045	0.7828	0.7320
Detección 3D	0.7809	0.6023	0.5548
Vista de pájaro	0.8396	0.7172	0.6531

Estos valores ayudan a hacerse una idea del nivel de precisión que estamos teniendo en nuestras detecciones.

Por último, se mostrarán una gráfica y una tabla indicando la mejora de los resultados que se ha ido obteniendo a lo largo del tiempo mientras se han ido realizando diversos cambios en el programa. Esta gráfica dará una idea del progreso que se ha ido obteniendo a lo largo del desarrollo del trabajo:



	1	2	3	4	5	6	7	8	9
Easy	0.5863	0.5861	0.7777	0.7665	0.7669	0.8219	0.8001	0.8092	0.9045
Medium	0.5235	0.5759	0.7003	0.699	0.6991	0.6773	0.6557	0.678	0.7828
Hard	0.5014	0.5577	0.6542	0.6551	0.6551	0.6321	0.6112	0.6499	0.732

Como se puede observar en el gráfico, por lo general se ha conseguido una mejora de los resultados a lo largo del desarrollo del programa, sin embargo, en ciertas ocasiones la mejora de los resultados de tipo “Easy” ha supuesto un empeoramiento en los resultados de los otros dos tipos de datos.

El gráfico solo recoge 9 muestras aproximadamente espaciadas de igual manera de entre todas las evaluaciones que se han realizado durante el desarrollo del programa, pero solo con este pequeño grupo de muestras se puede apreciar como se ha pasado de una precisión menor al **60%** a alcanzar valores cercanos al **90%** en el caso de las detecciones de tipo **Easy**.

Cabe mencionar que se realizaron evaluaciones previas al primer instante que aparece en la tabla antes de haber ajustado el parámetro **score** correctamente y cuyos resultados eran del orden del **30%** de precisión e inferiores, lo que da una idea de la importancia de este parámetro a la hora de realizar los test. También se puede destacar que la gran variación de la precisión que se produce entre las versiones 1 y 3 que aparecen en la gráfica es también debido en su mayoría a este parámetro y a diversos ajustes que se realizaron al efectuar el cálculo del mismo.

Finalmente, la última gran mejora que se observa entre las versiones 8 y 9 fue causada al cambiar el modelo de eliminación del suelo a toda la nube de puntos por la eliminación individual del suelo en cada ***clúster*** de manera separada, como ya se explicó en apartados anteriores.

6. Tracking

En este apartado se explicará la segunda parte del trabajo que se ha realizado en este TFG, que consiste en realizar un **tracking** o **seguimiento** de los objetos que se han detectado empleando los métodos explicados en el apartado anterior.

Para realizar el tracking se ha empleado una modificación del programa usado para la detección de objetos al que se le ha añadido una nueva funcionalidad que es la que se encarga de realizar el proceso de seguimiento.

También ha sido necesario modificar el formato de los datos de salida, ya que el test proporcionado por **KITTI** que nos sirve para evaluar el proceso de tracking emplea un formato de archivo ligeramente distinto y que será explicado a continuación.

Al igual que en el proceso de detección de objetos en este apartado se explicará todo el procesado que realiza el programa para realizar el tracking, sin embargo, en este caso se omitirán todos los pasos que se realizan exactamente de la misma manera que en la detección de objetos y que por tanto ya han sido explicados.

6.1. Datos de Entrada

Los datos de entrada que se emplearán en esta parte del trabajo serán los procedentes de la base de datos ***Kitti_dataset_tracking*** proporcionada por **KITTI**. El formato de estos datos ya ha sido explicado en el apartado **4.3 Tracking Dataset.** Tal y como se mencionó en ese apartado, esta base de datos contiene los datos de una serie de secuencias de instantes de tiempo que nos permitirán tanto realizar una detección como un tracking de los objetos a lo largo del desarrollo de esas secuencias.

Los datos proporcionados por esa base de datos que serán empleados en esta parte del trabajo son los siguientes:

- **Nubes de puntos**: Proporcionarán información en 3D del entorno que rodea al vehículo, de tal forma que podrán ser usadas para obtener las características de los objetos presentes en el escenario.
- **Matrices de calibración**: Proporcionarán la información necesaria de las posiciones relativas de los sensores que nos permitirá realizar proyecciones de los datos de la cámara a datos del LIDAR y viceversa.
- **Imágenes de la cámara**: Serán usadas para obtener la segmentación semántica de las mismas y determinar el tipo de objeto que estamos procesando.
- **Ficheros de Ground Truth**: Contendrán la información real de los objetos presentes en cada escenario y permitirán realizar la evaluación comparando nuestros datos obtenidos con estos datos.

Todos estos datos fueron explicados con más detalle en el apartado **4. KITTI Dataset**.

6.2. Detección de objetos

Todo el proceso de detección de objetos que se realiza en este programa será exactamente igual que el explicado en el apartado de detección de objetos, con la única diferencia de que los datos empleados serán diferentes.

El programa funcionará exactamente igual, detectando los objetos e introduciéndolos en un vector de objetos que será procesado posteriormente por el módulo que se encargará de realizar las correlaciones entre los objetos detectados en cada instante de tiempo.

Todo este proceso ya fue explicado y por tanto se pasará directamente a explicar el proceso de tracking.

6.3. Proceso de Tracking

Una vez hemos detectado todos los objetos que existen en el escenario en cada instante de tiempo se procederá a identificar que objeto corresponde con cada uno de los detectados en el instante anterior o a añadir uno nuevo en caso de que no se pueda realizar esta correlación.

Para ello se ha empleado el **filtro de Kalman**, que nos permite realizar una predicción de la posición que ocupará cada uno de los objetos en el instante siguiente, de esta manera podremos determinar cual de los objetos detectados en el instante actual corresponde con el objeto detectado en el instante anterior.

Se creará un nuevo *vector* de objetos que usaremos para introducir todos los objetos detectados a lo largo del tiempo, de manera que el *vector* que usábamos en el proceso de detección se irá sobrescribiendo para cada instante de tiempo, mientras que el nuevo *vector* conservará todos los objetos hasta que decidamos eliminarlos cuando lleven un tiempo sin ser detectados. En el nuevo *vector* también se introducirán los datos de las **predicciones** realizadas por el filtro de Kalman.

El primer paso para realizar el seguimiento consistirá en calcular la **distancia** que existe entre los objetos nuevos que se han detectado en cada instante de tiempo y la **predicción** que realizó el filtro de Kalman, y se determinará que un objeto puede ser el mismo objeto que se detectó en el instante anterior si la distancia entre esas dos medidas es menor de 3 metros. Tras haber determinado cuales de los objetos podrían ser ese mismo objeto del instante anterior se elegirá entre ellos el objeto candidato cuya distancia a la predicción sea menor, obteniendo de esta manera en la mayor parte de los casos el objeto correcto, aunque en ocasiones se podrán presentar fallos cuando un objeto deje de ser detectado y otro de los objetos candidatos que estén a menos de 3 metros sea considerado como el objeto anterior.

Tras esto, se comprobará que el objeto con el que se le ha relacionado no ha sido ya relacionado con otro objeto en esa misma interacción. Esto puede suceder si dos objetos están muy juntos y el programa determina que ambos podrían ser el mismo objeto detectado en el instante anterior. De esta forma el objeto de los dos candidatos que se encuentre más lejos no será relacionado con este objeto sino añadido como un nuevo objeto detectado que no había aparecido antes.

En el caso de un objeto al que se le haya podido realizar un seguimiento el proceso que se seguirá será sencillo:

1. Primero se comparará la **predicción** del filtro de Kalman con el **valor real** de la posición que hemos obtenido dibujando una línea en el visor 3D entre ambos puntos, con el único objetivo de poder visualizar la precisión de las predicciones que se están obteniendo. En las ocasiones en las que los coches se desplazan en línea recta las predicciones obtenidas suelen ser bastante acertadas, como se puede apreciar en la siguiente imagen:

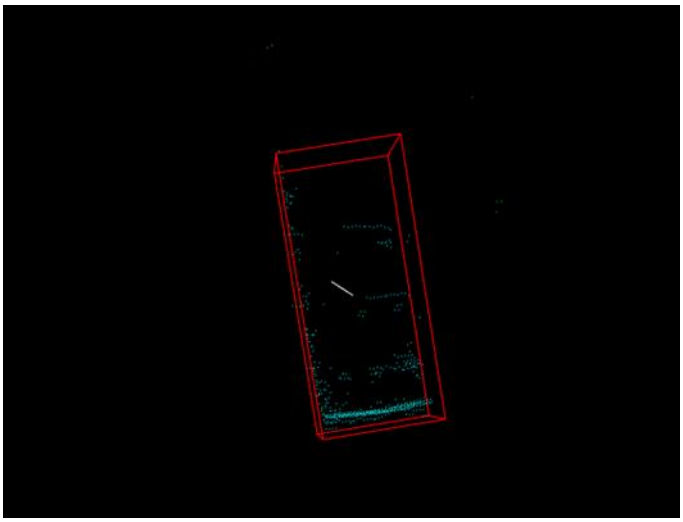


Ilustración 77 - Predicción mediante filtro de Kalman.

La línea blanca que aparece en el centro del *clúster* une el centro del objeto detectado con el centro de la predicción que se había realizado. Sin embargo, los errores se intensifican cuando los coches realizan giros o movimientos fuera de lo normal, como se apreciará en la siguiente imagen tomada del mismo objeto cuando empieza a girar a la derecha:

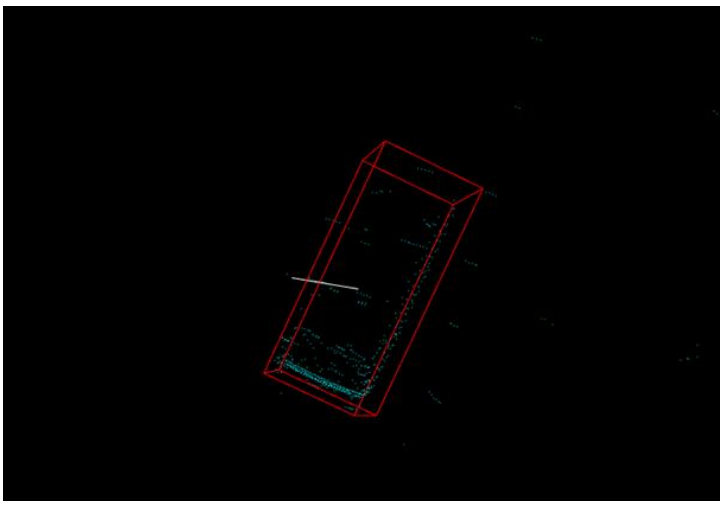


Ilustración 78 - Predicción mediante filtro de Kalman 2.

2. Se introducirá el nuevo valor obtenido en el conjunto de valores que conforman el filtro de Kalman de este objeto para que se pueda predecir la siguiente posición del objeto.
3. Se pintará la caja mínima de color rojo en vez de del habitual color azul para poder identificar los objetos a los que hemos conseguido realizar seguimiento de manera sencilla.

En caso de que el objeto no hubiese sido detectado antes simplemente se creará un nuevo **filtro de Kalman** introduciendo los datos del vehículo que se ha detectado.

También existirá un tiempo de vida de los objetos que se irá decrementado en cada interacción en la que no se detecte el objeto e incrementando cuando este es detectado. Cuando este tiempo de vida llegue a 0 el objeto se eliminará definitivamente del vector de tracking porque consideraremos que lo hemos perdido de vista y por tanto no se volverá a intentar relacionar a ningún objeto nuevo detectado con ese objeto.

En la siguiente imagen, se muestra un escenario en el que se está realizando tracking de todos los objetos que aparecen en una escena excepto del primero, que aparece abajo a la izquierda, que se habrá perdido por una mala detección en algún instante. También se pueden apreciar las líneas blancas que unen los centros detectados con los de las predicciones, que en este caso son bastante precisos debido a que los coches están parados:

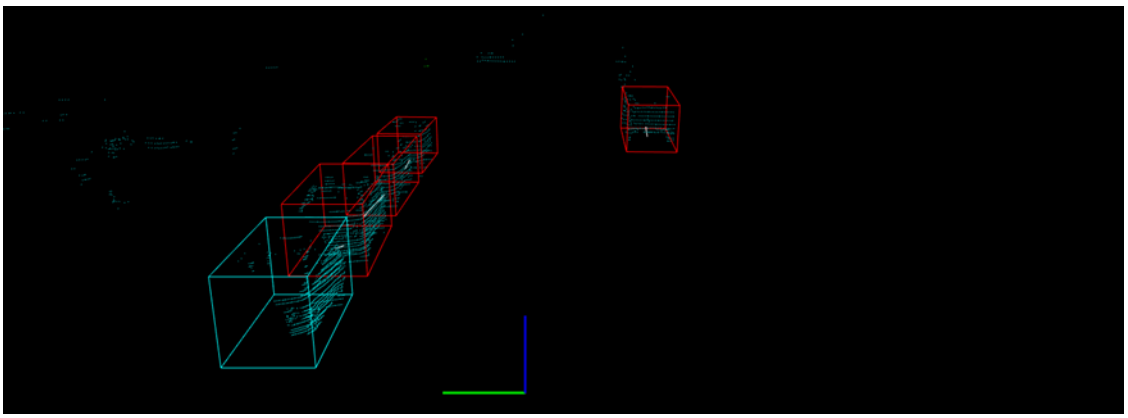


Ilustración 79 - Predicción en escenario completo.

La siguiente imagen muestra la imagen original captada por la cámara, que nos da una mejor visión de lo que está pasando en el escenario anterior:

Ilustración 80 - Imagen de la cámara.

El último paso a realizar será **publicar** los datos en el formato requerido por **KITTI,** al igual que se hizo en el programa de detección de objetos. Se creará un fichero ***".txt"*** que en este caso será único para todo el escenario y que se irá rellenando línea a línea con los datos de los objetos detectados de la manera en la que se explicó en el apartado de datos.

En cada línea se introducirá la información de cada objeto junto con el número de secuencia en el que fue detectado y el identificador de objeto, el resto de datos serán los mismos que se obtuvieron en la detección de objetos.

6.4. Resultados

En este apartado se expondrán los resultados obtenidos a lo largo del desarrollo del programa de tracking al aplicar los ***tests*** proporcionados por la base de datos de **KITTI**.

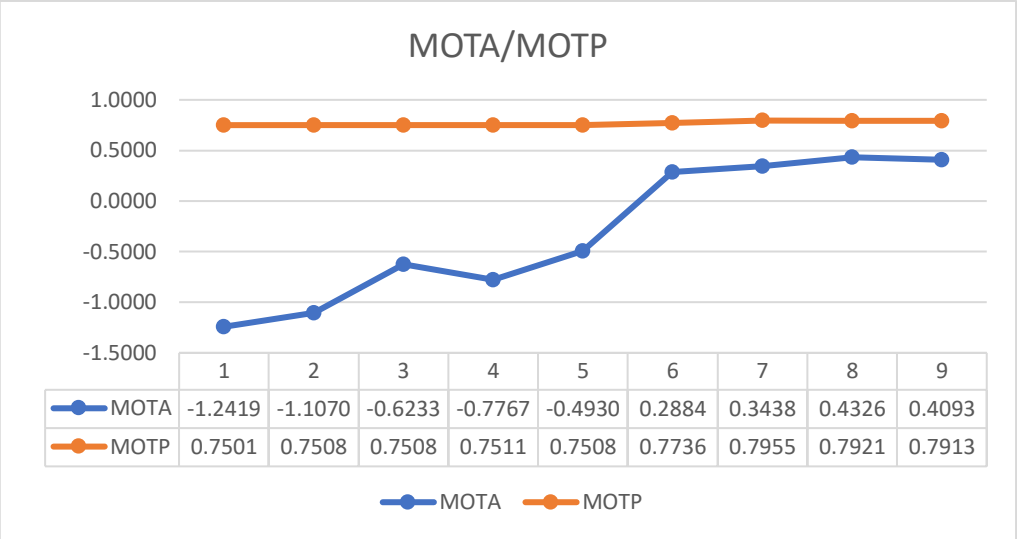
Durante el desarrollo del trabajo se han realizado más de 50 *tests* en los que se ha apreciado una mejora del seguimiento de los objetos según se ha ido mejorando el programa.

En el caso de este test no se nos proporciona una gráfica como salida sino un fichero de texto que contiene los datos que se indican a continuación:

- **MOTA**: Multiple Object Tracking Accuracy.
- **MOTP**: Multiple Object Tracking Precision.
- **MODA**: Multiple Object Detection Accuracy.
- **MODP**: Multiple Object Detection Precision.
- **Recall**: La cantidad de objetos que se han detectado del total presente en el escenario. Este parámetro se ve penalizado por un exceso de falsos positivos.
- **Precision**: La precisión con la que se han detectado esos objetos.
- **Mostly tracked**: Porcentaje de objetos detectados a los que se les ha realizado tracking durante la mayor parte del tiempo.
- **Partly tracked**: Porcentaje de objetos a los que se les ha realizado tracking solo durante una parte del tiempo que aparecían.
- **Mostly lost**: Porcentaje de objetos a los que apenas se les ha realizado seguimiento.

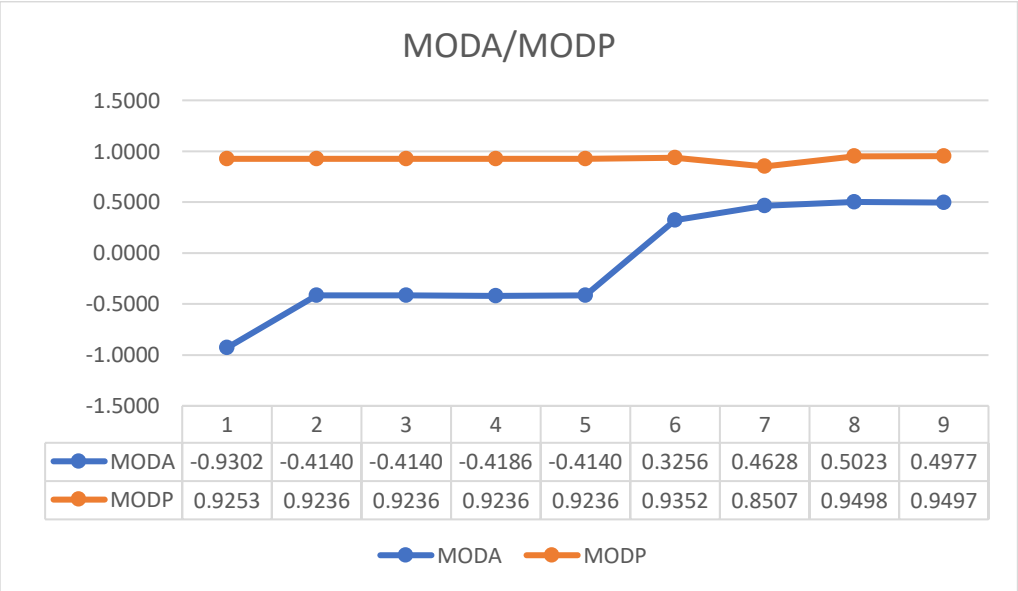
En las siguientes gráficas se mostrarán los resultados tras analizar el primero escenario que nos encontramos en la base de datos de KITTI. Se presentarán los resultados que se han obtenido a lo largo del desarrollo del programa escogiendo algunos de los resultados de todos los test que se han realizado de manera espaciada, ya que la evolución de la mejora de los resultados ha sido progresiva y unos pocos casos muestran correctamente el desarrollo de la misma.

El primero de los gráficos muestra la **MOTA** y **MOTP**. Se puede observar como la **precisión** casi no ha variado a lo largo del desarrollo de la aplicación, lo que es debido a que la parte del programa que detecta los objetos no ha sufrido casi ninguna variación. En cambio, se puede ver como ha mejorado notablemente la calidad de la precisión del tracking, que incluso llegó a ser negativa al principio. Esto es debido a que ha aumentado en gran medida la cantidad de objetos a los que se les hace tracking durante el test y el tiempo durante el cual se les hace el seguimiento:



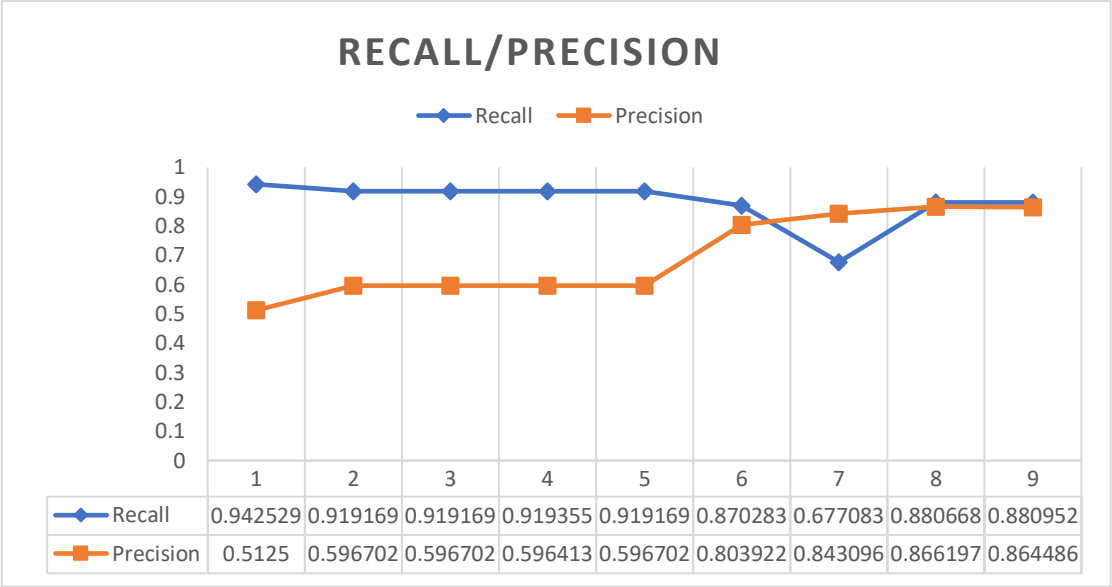
	1	2	3	4	5	6	7	8	9
MOTA	-1.2419	-1.1070	-0.6233	-0.7767	-0.4930	0.2884	0.3438	0.4326	0.4093
MOTP	0.7501	0.7508	0.7508	0.7511	0.7508	0.7736	0.7955	0.7921	0.7913

El siguiente gráfico refleja la **MODA** y **MODP**. Nuevamente la precisión se mantiene constante debido a que casi no ha variado esta parte del programa, pero la cantidad de instantes durante los cuales se realiza seguimiento a los objetos si que ha aumentado considerablemente:



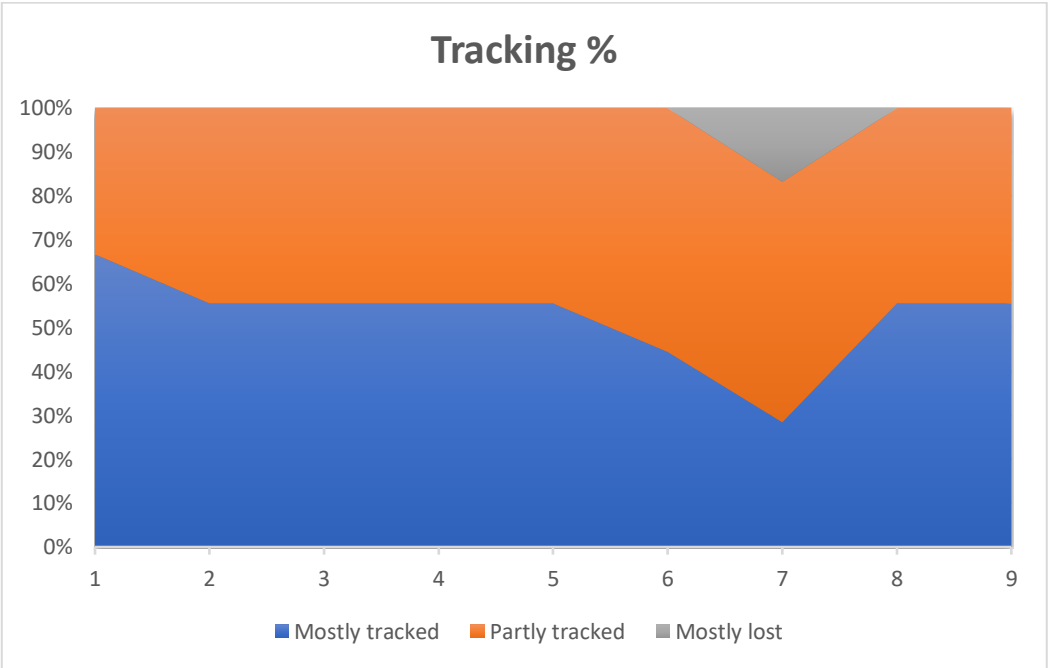
	1	2	3	4	5	6	7	8	9
MODA	-0.9302	-0.4140	-0.4140	-0.4186	-0.4140	0.3256	0.4628	0.5023	0.4977
MODP	0.9253	0.9236	0.9236	0.9236	0.9236	0.9352	0.8507	0.9498	0.9497

En la siguiente gráfica podemos observar el ***recall*** frente a la **precisión** del tracking de los objetos. Como se puede observar al disminuir el ***recall*** se ha aumentado la precisión en las distintas versiones, esto es debido a que a pesar de que antes se detectaban prácticamente todos los objetos que se debían detectar existía una gran cantidad de falsos positivos que disminuían la precisión de las detecciones. Finalmente se alcanzaron valores cercanos al 90% para ambos valores:



	1	2	3	4	5	6	7	8	9
Recall	0.942529	0.919169	0.919169	0.919355	0.919169	0.870283	0.677083	0.880668	0.880952
Precision	0.5125	0.596702	0.596702	0.596413	0.596702	0.803922	0.843096	0.866197	0.864486

En este último gráfico se muestra el porcentaje de objetos detectados a los que se les ha realizado un tracking la mayor parte del tiempo, durante un tiempo considerable o a penas se les ha realizado seguimiento. La disminución inicial es debida al alto ***recall*** que se presentaba en la gráfica anterior, se le realizaba seguimiento a más objetos pero existían muchos falsos positivos que no se reflejan en esta gráfica. Durante el desarrollo, en algún instante se comenzaron a perder objetos debido a que se intentó adoptar unas políticas más restrictivas para eliminar los falsos positivos que afectaron a las detecciones reales. Finalmente, los errores se corrigieron y obtuvimos los resultados finales mostrados en la gráfica:



7. Mejora de Resultados Empleando Mask R-CNN

Tras haber obtenido los resultados empleando la arquitectura mostrada en el punto "2. Objetivos del Proyecto" se decidió seguir realizando mejoras al programa de detección de objetos con el objetivo de conseguir una mejora de las detecciones.

Una de las posibilidades que se plantearon fue el uso de la red neuronal **Mask R-CNN** [22][18], que es capaz de realizar una segmentación de las imágenes que se le pasan dando como resultado las cajas mínimas que rodean a cada objeto, así como el espacio que ocupa el objeto en la imagen. Un ejemplo del funcionamiento de esta red se puede ver en la siguiente imagen:

Ilustración 81 - Imagen procesada por Mask R-CNN.

La gran diferencia entre esta red y la que se estaba usando actualmente (ERFNet) es que la última solo realiza una segmentación de la imagen por tipo de objeto coloreando todos los objetos de un tipo de un mismo color, mientras que **Mask R-CNN** segmenta cada uno de los objetos de la imagen aportándonos directamente como resultado la caja mínima de cada uno de los objetos y separando cada uno de ellos en un color diferente.

El simple hecho de obtener directamente las cajas mínimas en la imagen nos permite realizar la evaluación proporcionada por KITTI para comprobar la calidad de los resultados obtenidos en las detecciones 2D empleando únicamente las imágenes de la cámara y la red *convolucional*. Al comenzar a procesar todas las imágenes destacaba el hecho de que la red detectaba prácticamente todos los vehículos que aparecían en las imágenes incluso si las condiciones para la detección no eran las mejores.

[22] https://github.com/matterport/Mask_RCNN

Los resultados obtenidos fueron los siguientes:

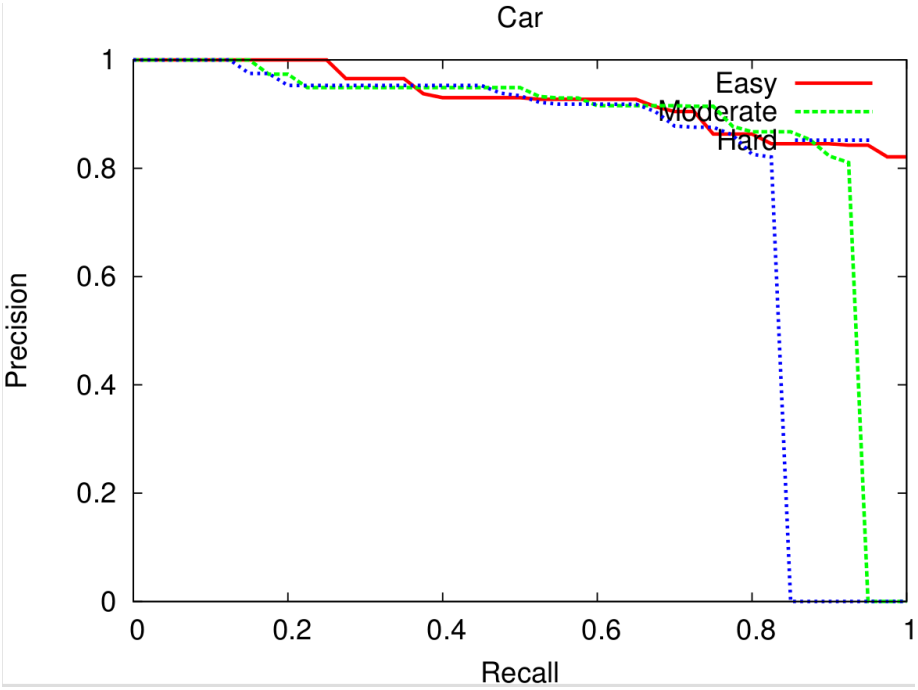


Ilustración 82 - Resultados 2D Mask R-CNN.

Numéricamente estos resultados corresponden con un **92,77%** para las detecciones en modo **easy**, un **86,58%** para las detecciones en **moderate** y un **77,70%** en **hard**. Como se puede ver los resultados son muy buenos teniendo en cuenta que solo se han empleado las imágenes para obtenerlos y que la red no está entrenada con la base de datos de **KITTI**.

Una vez observados los resultados se barajaron varias posibilidades que tenían como objetivo cambiar ligeramente el funcionamiento del programa de detección de objetos añadiendo la funcionalidad de la nueva red con el objetivo de comprobar si se podían mejorar los datos.

La primera posibilidad consistió en la sustitución de la segmentación actual que ya realizaba la red **ERFNet** por la que nos da **Mask R-CNN,** ya que la nueva red realizaba una separación más definida de los objetos y podría evitarnos algunos falsos positivos, y a su vez estaba entrenada con una base de datos distinta y podría brindarnos unos resultados diferentes a los actuales.

La segunda posibilidad se basaba en emplear los resultados de las detecciones en 2D obtenidos directamente por la red *convolucional* para eliminar parte de los falsos positivos que se estaban generando actualmente en nuestro programa. De esta forma se eliminarían todas las detecciones del programa actual que no coincidiesen en el 2D con las de la nueva red.

A continuación, se explicarán con mayor detalle las dos modificaciones que se llevaron a cabo junto con la nueva arquitectura del sistema y se expondrán sus resultados comparándolos con los que se obtuvieron anteriormente sin usar **Mask R-CNN.**

7.1. Primera Modificación: Sustitución de Imagen Segmentada

La primera modificación consistió simplemente en sustituir las imágenes que estaba generando **ERFNet** por las obtenidas de **Mask R-CNN** con algunas modificaciones.

De esta manera el proceso de detección de objetos que se llevó a cabo fue exactamente el mismo que se realizaba antes con la diferencia de que se estaban usando otras imágenes para colorear la nube de puntos.

La estructura de este sistema era la misma que se explicó en el apartado ***2. Objetivos del Proyecto***.

Fue necesario modificar las imágenes de salida de Mask R-CNN haciendo que se colorease cada uno de los objetos detectados por su tipo de objeto siguiendo el mismo patrón de colores que empleaba ERFNet. De esta manera las imágenes obtenidas tenían el siguiente aspecto:

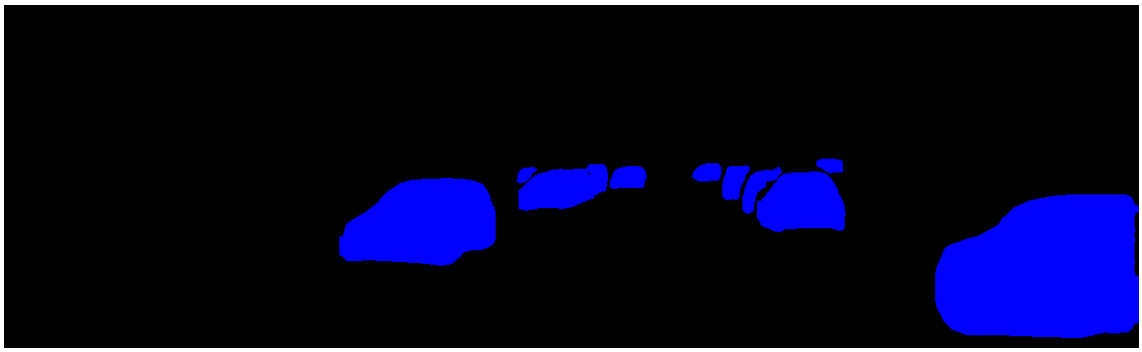


Ilustración 83 - Imagen segmentada con Mask R-CNN.

Como se puede observar comparándolo con la imagen que obteníamos usando **ERFNet** los coches están mejor separados en este caso ya que la red deja un espacio entre ellos aunque estén juntos:

Ilustración 84 - Imagen Segmentada con ERFNet.

Por lo demás el proceso seguido era el mismo que se empleaba antes con la diferencia de que en este caso la nube de puntos no quedaba coloreada con la información del suelo o del resto de objetos presentes en la imagen que no fuesen coches, peatones o ciclistas.

Sin embargo, tras realizar la evaluación de KITTI los resultados mostraron que el uso de la nueva red había llevado a un empeoramiento de los resultados, obteniendo en este caso los siguientes resultados numéricos para las detecciones:

Nombre del test/Dificultad	Easy	Moderate	Hard
Detección 2D	0.8109	0.6798	0.6300
Detección 3D	0.6855	0.4932	0.4363
Vista de pájaro	0.7496	0.5892	0.5243

El hecho de que los resultados empeorasen empleando las nuevas imágenes ocurrió probablemente porque el programa actual de detección de objetos estaba configurado para obtener un mayor rendimiento con las imágenes antiguas, ya que fue con estas con las que se realizó todo el proceso de desarrollo y por tanto se ajustaron los parámetros del programa para obtener un mayor rendimiento con estas imágenes.

De igual forma, tras observar que no se producía una mejora y que los resultados eran muy inferiores se determinó que empleando este método no se podría llegar a alcanzar una mejora sustancial de los resultados y por tanto se decidió optar por el siguiente método.

7.2. Segunda Modificación: Detecciones 2D Mask R-CNN

La segunda posibilidad que se probó consistió en emplear directamente los datos de las cajas mínimas que proporcionaba la red para eliminar los falsos positivos que se obtenían en nuestras detecciones.

Debido a la gran precisión con la que **Mask R-CNN** detectaba los objetos se esperaba poder eliminar las detecciones obtenidas mediante nuestro sistema que no coincidiesen tras proyectarlas al 2D con las que se obtenían con la nueva red.

De esta forma, al igual que ocurría en el programa original se obtendrían las detecciones en 3D empleando la proyección de las imágenes segmentadas usando ERFNet, se proyectarían las cajas obtenidas sobre la imagen 2D y posteriormente se eliminarían todos los objetos detectados cuya proyección en 2D no coincida con la obtenida mediante **Mask R-CNN**.

Para llevar a cabo este proceso se optó por proyectar el centroide del objeto detectado en 3D empleando las matrices de conversión que ya se habían mencionado en apartados anteriores y la siguiente fórmula de proyección 3D a 2D.

$$Punto\_2D = Matriz\_P \times Matriz\_R \times Matriz\_T \times Punto\_3D$$

Tras haber proyectado el centroide de la caja detectada se comprobará si se encuentra dentro de una de las cajas de detección que nos ha devuelto **Mask R-CNN.** Para obtener una mayor precisión solo se considerará que el objeto es válido si esta proyección del centroide en 3D se encuentra a una determinada distancia del centroide de la caja en 2D al proyectarlo, ya que de otra manera algunos falsos positivos podrían colarse si su centroide entraba dentro de una de las cajas detectadas por Mask R-CNN a pesar de que se encontrase en la esquina de una de estas.

Finalmente, tras haber comprobado que objetos coinciden y cuales no se procederá a eliminar los objetos detectados por nuestro sistema que no hayan sido validados por el proceso que se ha explicado.

De esta forma, la arquitectura del sistema quedó modificada levemente sustituyendo las detecciones en 2D que realizábamos anteriormente directamente sobre la imagen segmentada por las detecciones obtenidas mediante **Mask R-CNN**.

Además, con este nuevo método estas detecciones se emplearán para eliminar falsos positivos en vez de añadir detecciones que no se obtenían anteriormente mediante 3D, quedando la arquitectura como se muestra en la siguiente imagen:

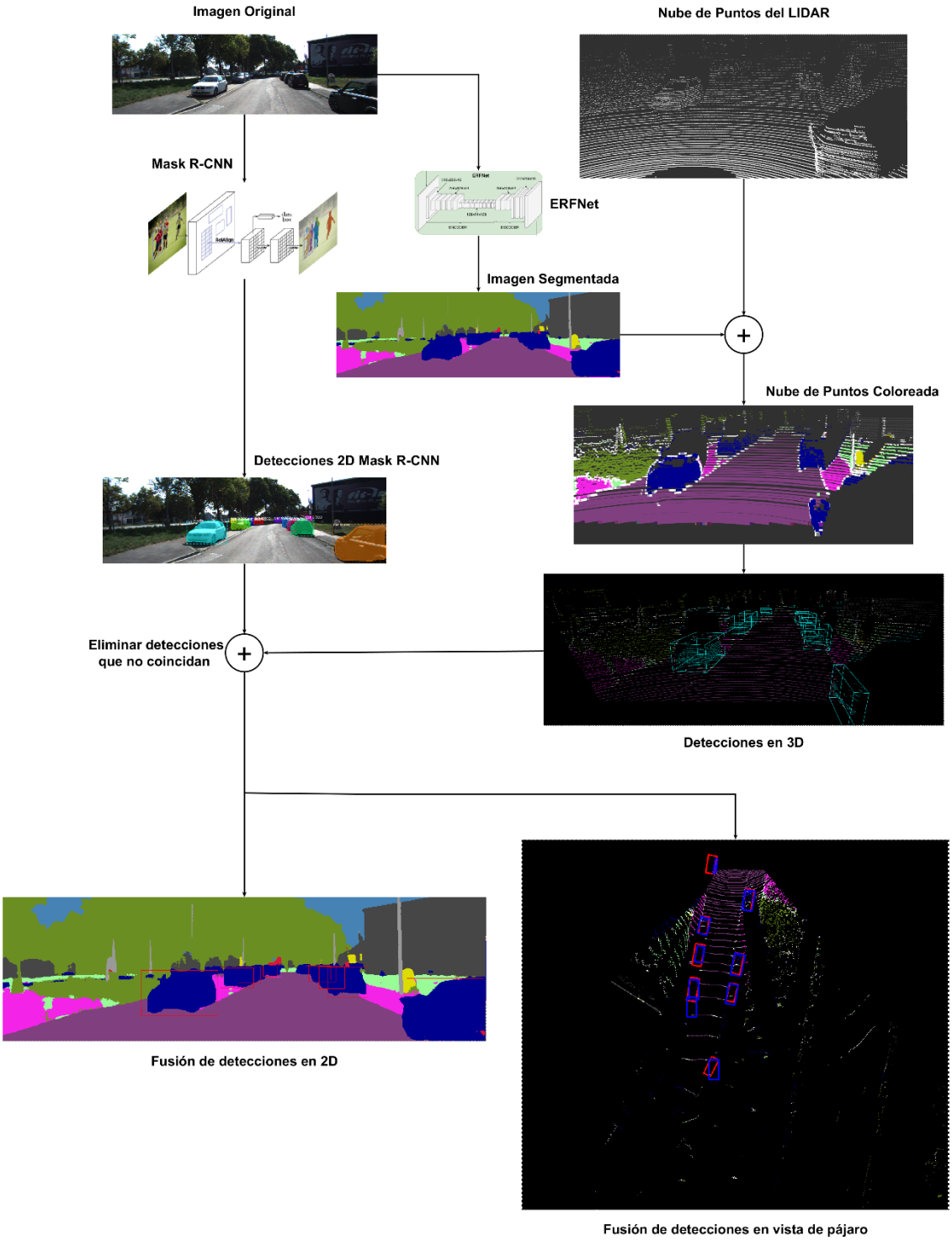


Ilustración 85 - Esquema de funcionamiento tras la modificación.

Tras realizar unas leves modificaciones en el programa de detección que consistieron en "ablandar" la política de eliminación de falsos positivos del programa, ya que la nueva estructura se encargaría de eso, se procedió a obtener los nuevos resultados empleando el test de KITTI.

Estos resultados fueron los siguientes:

-Resultados 2D:

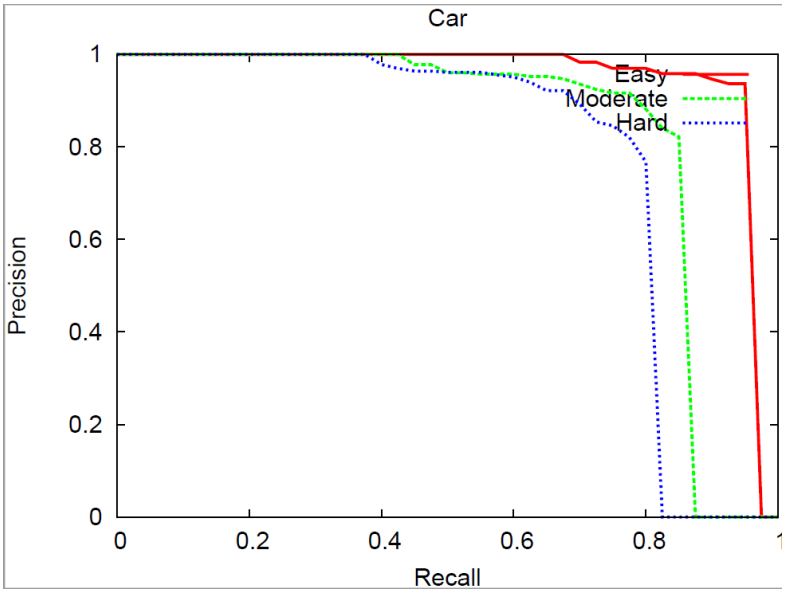


Ilustración 86 - Resultados 2D con Mask R-CNN.

-Resultados 3D:

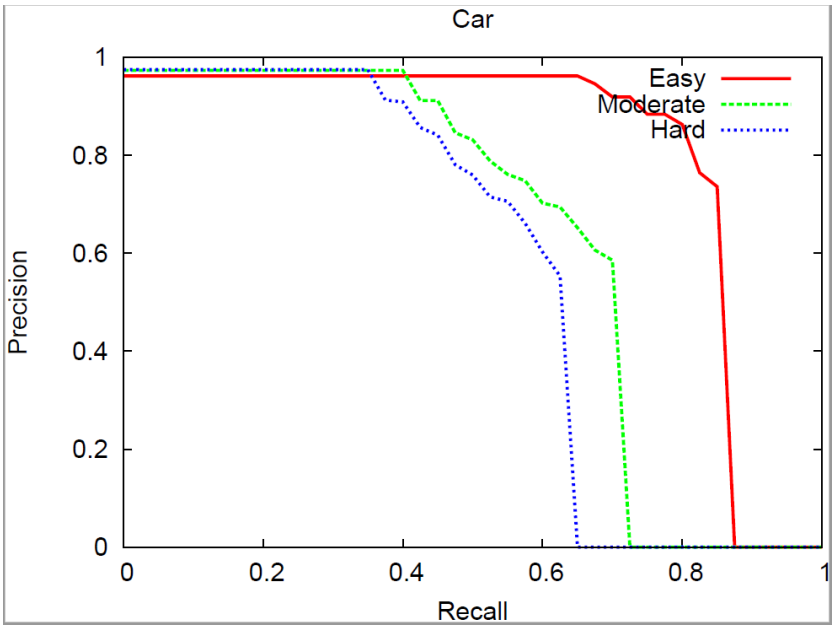


Ilustración 87 - Resultados 3D con Mask R-CNN

-Resultados vista de pájaro:

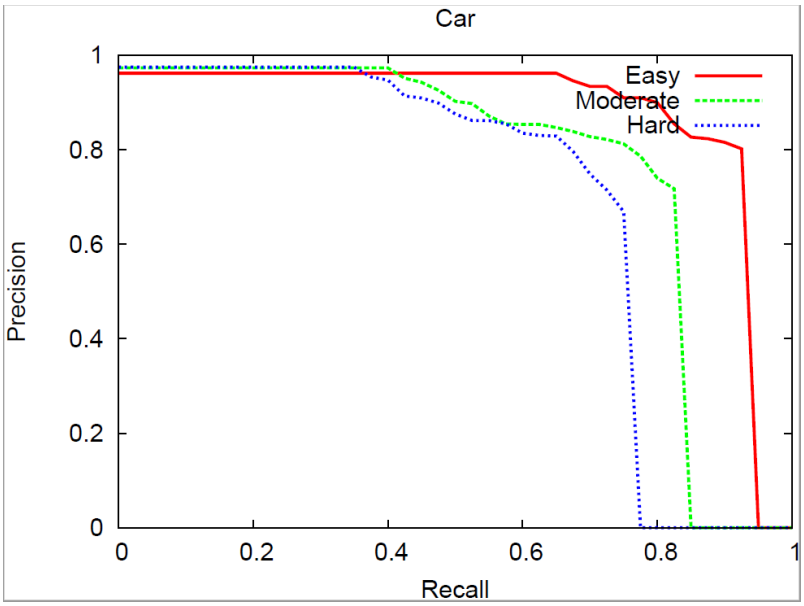


Ilustración 88 - Resultados vista de pájaro con Mask R-CNN.

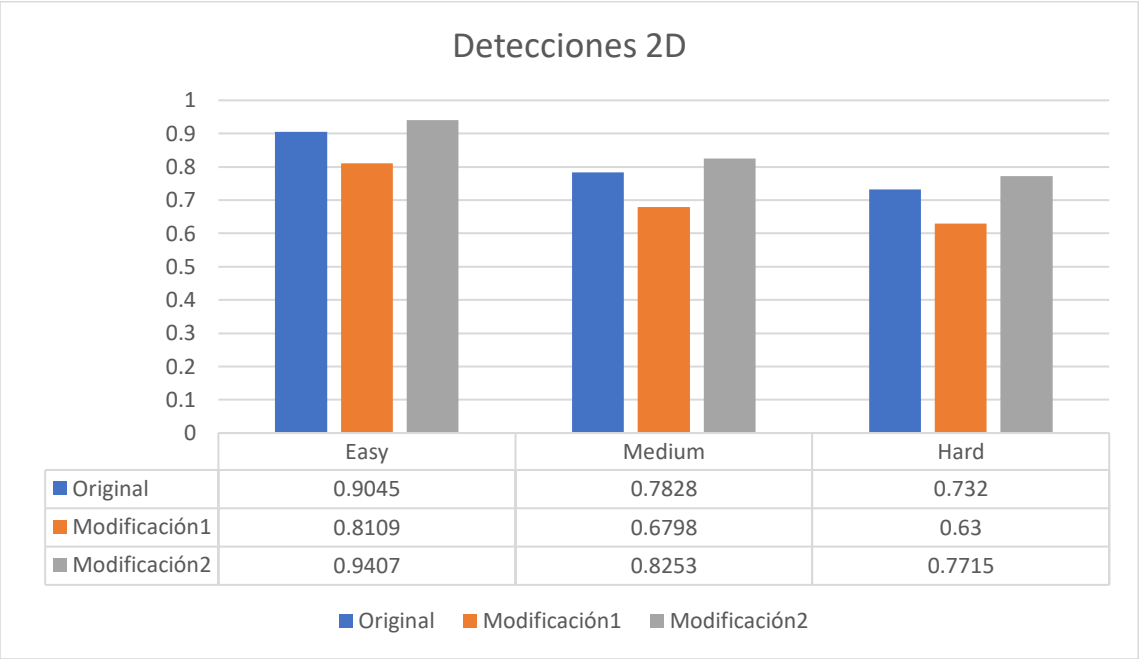
Correspondiendo estos resultados con los siguientes resultados numéricos:

Nombre del test/Dificultad	Easy	Moderate	Hard
Detección 2D	0.9407	0.8253	0.7715
Detección 3D	0.8024	0.6243	0.5593
Vista de pájaro	0.8693	0.7562	0.6863

7.3. Comparación de los Resultados Obtenidos

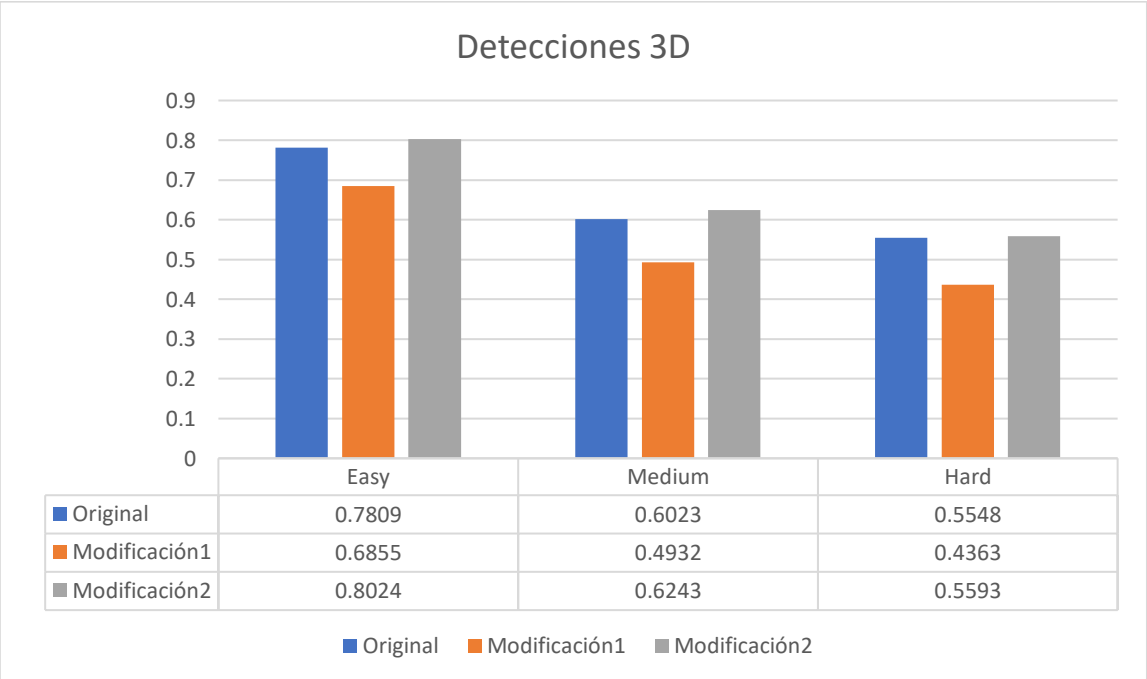
A continuación, se mostrarán una serie de gráficas comparando los resultados originales con los que se han obtenido tras emplear **Mask R-CNN**.

La primera gráfica muestra la variación de la precisión en las detecciones 2D empleando los tres métodos explicados:



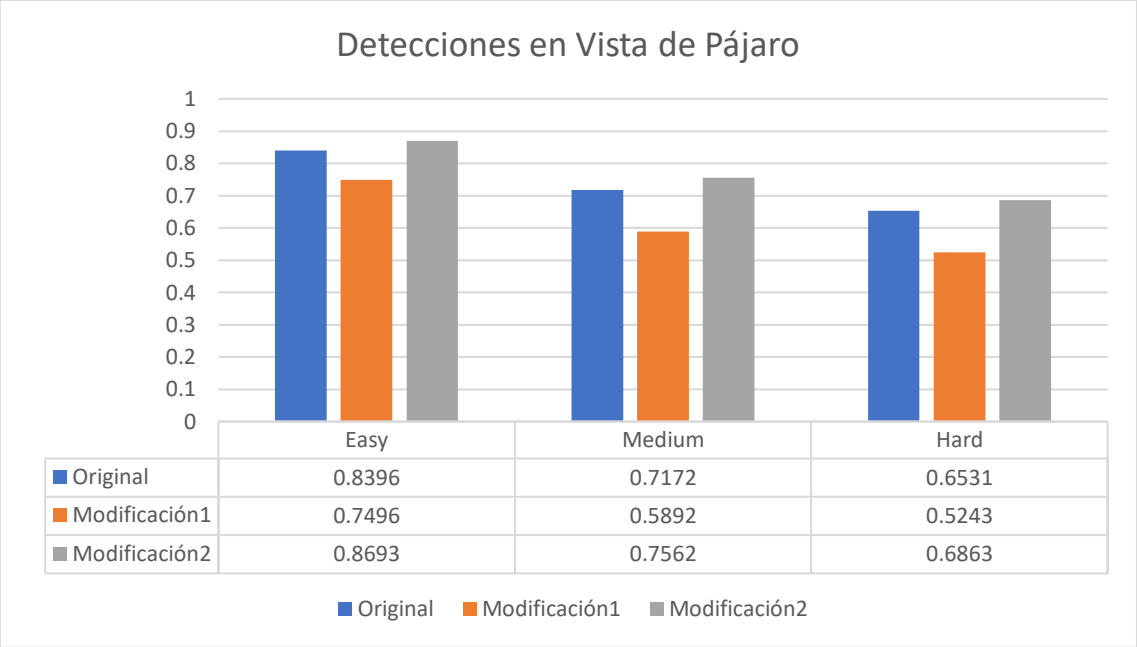
	Easy	Medium	Hard
Original	0.9045	0.7828	0.732
Modificación1	0.8109	0.6798	0.63
Modificación2	0.9407	0.8253	0.7715

La siguiente gráfica muestra los resultados de las detecciones en 3D:



	Easy	Medium	Hard
Original	0.7809	0.6023	0.5548
Modificación1	0.6855	0.4932	0.4363
Modificación2	0.8024	0.6243	0.5593

Finalmente, la última gráfica muestra los resultados de las detecciones en vista de pájaro:



	Easy	Medium	Hard
Original	0.8396	0.7172	0.6531
Modificación1	0.7496	0.5892	0.5243
Modificación2	0.8693	0.7562	0.6863

Como se puede observar en todas las gráficas la primera modificación realizada supuso un empeoramiento de los resultados de las detecciones en todos los tipos de detecciones.

En cambio, si nos fijamos en la segunda modificación, el empleo de Mask R-CNN para eliminar los falsos positivos supuso una mejora notable en los resultados si los comparamos con los originales. Siendo esta mejora de aproximadamente el 4% en las detecciones en 2D, del 2% en las detecciones en 3D y del 3% en las detecciones en vista de pájaro.

8. Conclusiones y Trabajo Futuro

A lo largo de este trabajo se ha tratado de aplicar la **fusión de datos** de **LIDAR** y **visión** para realizar una detección y seguimiento de los objetos presentes en un escenario, estudiando de esta manera los resultados obtenidos al emplear esta aproximación a la hora de enfrentar este problema.

Durante el desarrollo del mismo se ha conseguido mejorar notablemente los resultados que se han ido obteniendo tanto en lo que respecta a detección de los objetos como al tracking de los mismos.

Gracias a los numerosos cambios que ha sido necesario realizar en los parámetros y algoritmos usados en los programas que se han desarrollado para lograr mejorar la precisión de las detecciones y del seguimiento ha sido posible extraer una gran cantidad de **conclusiones** que han ayudado a comprender los principales problemas que afectan a las detecciones e impiden que estas se realicen de una manera correcta.

A pesar de que los resultados obtenidos en las últimas fases del trabajo han sido bastante esperanzadores y han demostrado que el sistema es capaz de realizar correctamente las detecciones de los objetos y el tracking de los mismos, en la mayoría de las ocasiones se han encontrado dos barreras principales que impiden una mejora significativa de los resultados y que podrían ser objetivo de un trabajo futuro.

Uno de los mayores problemas encontrados ha sido derivado del empleo de datos fusionados entre cámara y LIDAR. A pesar de que esto nos concede una gran ventaja a la hora de separar los *clusters* por su tipo de objeto, trae también como consecuencia una gran cantidad de fallos y errores.

En primer lugar, se encuentran los fallos producidos al proyectar la imagen de la CNN sobre los puntos del LIDAR. Al tratarse la cámara y el LIDAR de dos dispositivos diferentes y sumándole que las imágenes de la cámara han sido procesadas por una red *convolucional* nos encontramos que surgen numerosos errores en las proyecciones del color debido a que los datos de ambos dispositivos no se pueden proyectar con total precisión. Estos fallos llevan a la aparición de numerosos falsos positivos que es necesario filtrar causando la eliminación de objetos reales en algunas ocasiones y a la aparición de errores en la detección de los objetos debido a que se juntan con otros objetos que se encuentran cerca de ellos. A estos errores se les pueden añadir también los fallos que se producen directamente en la red *convolucional* a la hora de detectar los objetos.

El segundo factor de error viene derivado del cálculo del ángulo de los *clusters* obtenidos, que nos sirve de ayuda para detectar su caja mínima sobre el plano. Para ello hemos empleado la ***transformada de Hough*** aplicada a la proyección de los *clusters* sobre el suelo, sin embargo, este método no es muy fiable cuando la detección del coche no es demasiado buena y nos lleva a cometer grandes errores en el cálculo de la caja mínima de algunos *clusters*.

Para lograr una mejora de los dos problemas contemplados anteriormente y basándose en otros trabajos actuales se han encontrado métodos que podrían solucionar ambos problemas. Algunas aproximaciones actuales emplean redes neuronales que detectan el tipo de objeto basándose solo en la información del LIDAR, lo que eliminaría el problema de las proyecciones y los falsos positivos causados al fusionar la información de la cámara y el LIDAR. De igual manera, se podrían emplear redes convolucionales entrenadas para ayudarnos a obtener el

ángulo de los coches de un escenario dado, lo que eliminaría en gran medida los problemas derivados del uso de la *transformada de Hough* para el cálculo del ángulo.

9. Manual de Usuario

En este apartado se explicarán los pasos necesarios para poder lanzar tanto los dos programas que se han realizado como los test de evaluación proporcionados por KITTI para comprobar la precisión de los resultados obtenidos.

Para ello primero se mencionarán los requisitos necesarios para poder ejecutar el entorno y finalmente se expondrán los comandos necesarios que habrá que ejecutar para lanzar cada uno de los programas. También se mostrarán algunas imágenes con los resultados de los mismos.

9.1. Requisitos Previos

El principal requisito para poder ejecutar los programas realizados será instalar **ROS**[23] en el sistema, en el caso de este trabajo se ha empleado la versión **ROS Indigo**. Este entorno permitirá ejecutar los diferentes módulos del sistema de manera que puedan comunicarse entre ellos usando diversos mensajes.

El segundo requisito será poseer en el sistema el *workspace* de ROS que contiene los diversos programas que se encargan de colorear las nubes de puntos con la información de la CNN y el que se encarga de leer y publicar los datos de la base de datos de KITTI, los cuales han sido obtenidos de los repositorios del grupo **Robe-safe**. Una vez se tienen ambos *workspaces* en el sistema habrá que compilarlos ejecutando "**catkin_make**" dentro de cada uno de ellos y finalmente ejecutar "**source devel/setup.bash**" con el objetivo de poder lanzar los ejecutables que se encuentran dentro de cada *workspace* sin tener que indicar la ruta completa.

Será necesario descargar los *datasets* de **tracking** y **detección de objetos** de la base de datos de **KITTI** así como los test de evaluación que nos proporcionan los mismos *dataset*. Todos ellos pueden ser descargados directamente de la página web de **KITTI**[24].

Ambos *datasets* deberán ser almacenados en una carpeta con nombre "KITTI" en el *home* del usuario que lo ejecute. También será necesario cambiar en el mismo código de los programas las cadenas de texto que indican las carpetas en las que se almacenarán los ficheros de salida si se quiere cambiar el destino de estos ficheros.

Finalmente, con el objetivo de lanzar el test de evaluación para el *dataset* de tracking será necesario tener instalada en el sistema la versión de **Python 2.7** o superior.

9.2. Detección de Objetos

Para ejecutar este programa y que consiga realizar su cometido será imprescindible seguir los siguientes pasos:

1. Ejecutar el programa que se encargará de publicar los datos de la base de datos de KITTI que serán leídos por nuestro programa:

[23] http://wiki.ros.org/indigo/Installation/Ubuntu
[24] http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d

```
systick@laptop:~/kitti_ws$ roslaunch kitti_player_tracking kittiplayer_object_standalone_fijo.l
aunch
```

Ilustración 89 - Comando para ejecutar el programa de publicado de datos de KITTI.

Este programa empezará a publicar por defecto los datos de la base de datos de objetos de manera seguida y nos mostrará la imagen correspondiente a ese instante de tiempo:

Ilustración 90 - Resultados del comando de publicado de datos de KITTI.

2. Posteriormente será necesario lanzar en otro terminal el programa que publicará la nube de puntos coloreada empleando los datos de KITTI. Este programa no generará ninguna salida visual pero se podrá comprobar su funcionamiento usando la herramienta Rviz que habrá sido instalada durante la instalación de ROS. Para ejecutar el programa habrá que lanzar el siguiente comando:

```
systick@laptop:~/robosafe_ws$ roslaunch but_calibration_camera_velodyne coloring_kitti_tracking
.launch
```

Ilustración 91 - Lanzamiento del programa que colorea las nubes de puntos con los datos de la CNN.

3. Finalmente podremos lanzar nuestro programa, que mostrará por pantalla los datos en 3D de las detecciones, las imágenes de la cámara y de la CNN con las detecciones en 2D dibujadas y la vista de pájaro de las mismas. También publicará los datos de salida en la carpeta “datos” que se encontrará por defecto en el *home* del usuario que lo lanza. Para lanzar el programa de detección de objetos habrá que ejecutar en otro terminal el siguiente comando:

```
systick@laptop:~$ rosrun kitti_pruebas object_color
```

Ilustración 92 - Comando para lanzar el programa de detección de objetos.

La salida obtenida tras ejecutar el programa será la siguiente:

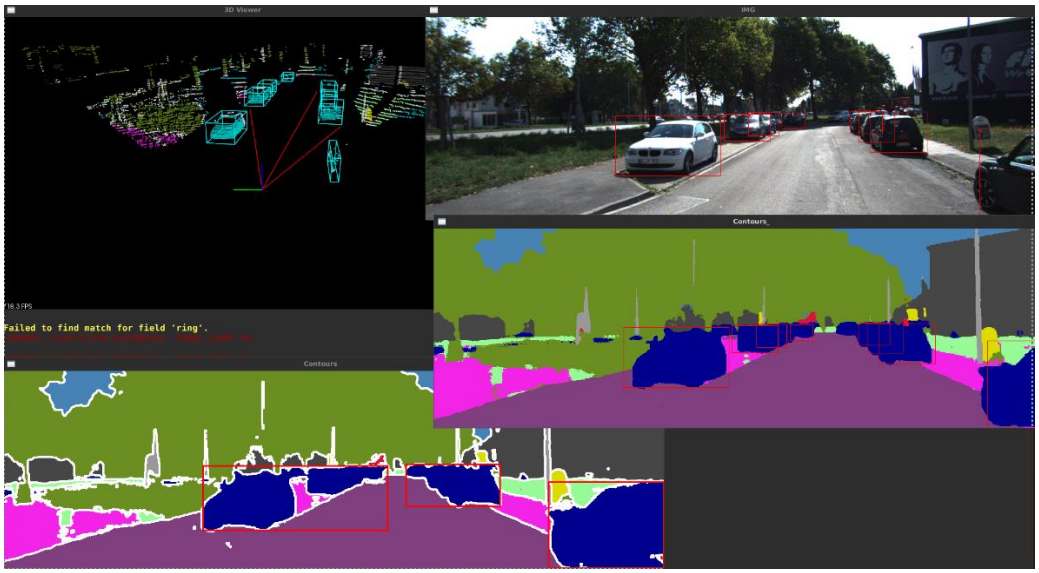


Ilustración 93 - Ejecución del programa de detección de objetos.

También se publicarán los datos en los ficheros de salida en la carpeta ya mencionada anteriormente, que tendrán el siguiente contenido:

```
1 Car 0.00 1 1.53 626.76 164.02 655.27 186.75 1.24 1.70 3.70 3.09 1.60 73.35 1.57 -0.24
2 Car 0.00 1 1.31 774.26 178.34 851.28 219.31 1.33 1.70 3.70 7.11 1.68 26.13 1.57 -0.04
3 Pedestrian 0.00 0 -0.06 646.25 171.93 663.77 205.80 1.45 0.39 0.41 2.71 1.60 44.55 -0.00 0.23
4 Car 0.42 0 -0.91 5.00 199.00 342.08 363.00 1.53 1.75 2.43 -2.77 1.73 3.95 -1.52 0.66
5 Car 0.00 0 -1.29 331.60 183.25 535.85 316.02 1.42 1.72 4.17 -2.24 1.62 10.42 -1.50 1.00
6 Car 0.00 0 -1.65 729.30 180.98 805.89 247.42 1.43 1.52 2.55 3.60 1.77 17.39 -1.45 0.69
7 Car 0.00 2 -1.34 434.72 172.63 516.75 213.82 1.29 1.43 2.38 -4.44 1.43 24.01 -1.52 0.13
8 Car 0.00 0 -1.49 562.48 174.43 614.39 212.33 1.45 1.83 3.70 -0.88 1.60 29.99 -1.52 0.44
9 Car 0.00 2 1.40 703.77 182.91 781.23 228.34 1.23 1.70 3.70 3.94 1.60 22.60 1.57 -0.04
```

Ilustración 94 - Fichero de resultados de detección de obetos.

Una vez se ha ejecutado el programa y se han generado los datos necesarios se podrá pasar a realizar la evaluación de los mismos mediante el test de evaluación de KITTI.

9.2.1. Evaluación de la detección de objetos

Para evaluar los datos obtenidos con nuestro programa será necesario emplear el test de evaluación de objetos obtenido de la base de datos de KITTI.

Antes de poder realizar el test de evaluación tendremos que introducir los archivos de **ground truth** proporcionados por la base de datos en una carpeta con el nombre “data” dentro de la carpeta que contenga el propio test de evaluación.

Este test nos proporcionará un archivo llamado “evaluate_object.cpp” que podrá ser compilado siguiendo las instrucciones del archivo “readme.txt” contenido en el mismo test de evaluación.

Una vez compilado solo será necesario seguir los siguientes pasos para obtener los resultados de la evaluación:

1. Introducir nuestros ficheros de resultados en la siguiente estructura de carpetas dentro del mismo directorio donde se encuentra el test de evaluación: "results/nombre_prueba/data". Todos los ficheros que hemos generado serán introducidos dentro de la carpeta "data" siguiendo la estructura de carpetas mencionada anteriormente.
2. Ejecutar el test de evaluación pasándole como parámetro el nombre de la carpeta donde hemos introducido los datos del test:

```
systick@laptop:~/Downloads/devkit_object/cpp$ ./evaluate_object test
```

Ilustración 95 - Evaluación Detección de Objetos.

Con estos dos pasos el test habrá creado una carpeta "results" dentro de la carpeta "nombre_prueba" junto con varios archivos de texto. Los archivos de texto contendrán diversos datos indicando la precisión de nuestras detecciones de forma numérica, mientras que en la carpeta "results" se habrán generado varios archivos en formato de imagen y "pdf" conteniendo los gráficos que indican la precisión y el *recall* de nuestras detecciones. Ejemplos de estos gráficos ya fueron mostrados en los apartados de resultados de este trabajo.

9.3. Tracking

Para ejecutar el programa que se encargará de realizar el tracking de los objetos y de publicar los datos obtenidos será necesario realizar los siguientes pasos:

1. Ejecutar el programa que se encargará de publicar los datos de la base de datos de KITTI que serán leídos por nuestro programa, en este caso se publicarán los datos del *dataset* de tracking:

```
systick@laptop:~/kitti_ws$ roslaunch kitti_player_tracking kittiplayer_tracking_standalone.launch
```

Ilustración 96 - Comando para el lanzamiento del programa de publicado de datos de tracking.

El programa ejecutará automáticamente una de las secuencias de datos y publicará sus datos publicando por pantalla las imágenes de la cámara como se puede observar a continuación:

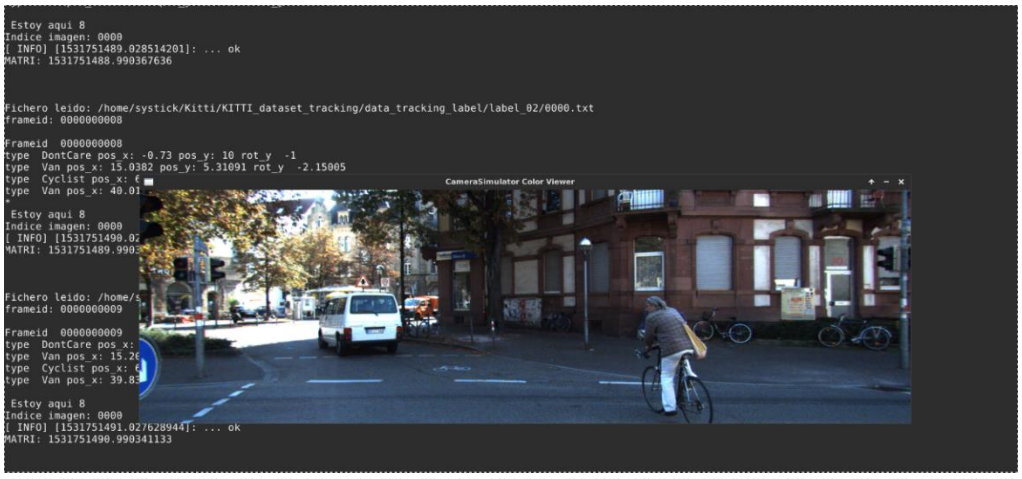


Ilustración 97 - Salida del programa de publicado de datos de tracking.

2. Posteriormente será necesario lanzar en otro terminal el programa que publicará la nube de puntos coloreada empleando los datos de KITTI, de igual manera que se hizo al publicar los datos que se usaban en el programa de detección de objetos:

```
systick@laptop:~/robosafe_ws$ roslaunch but_calibration_camera_velodyne coloring_kitti_tracking
.launch
```

Ilustración 98 - Lanzamiento del programa que colorea las nubes de puntos con los datos de la CNN.

3. Finalmente se lanzará el programa que realizará el tracking de los objetos. Este programa mostrará por pantalla el visor 3D donde se podrán observar los objetos a los que se les está realizando seguimiento marcados con cajas rojas, y también publicará los datos proyectados sobre las imágenes de la CNN en 2D, como se puede observar a continuación:

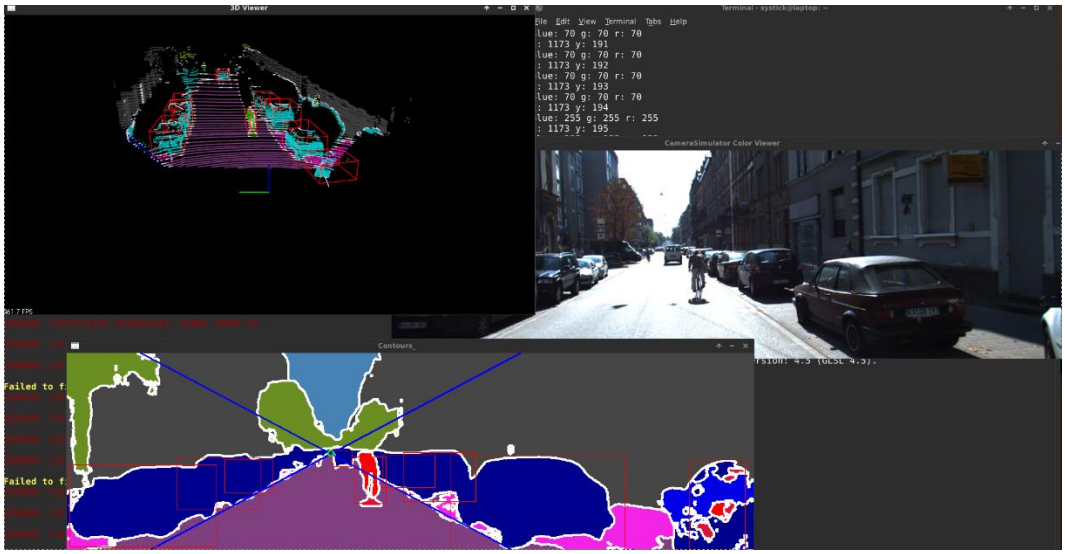


Ilustración 99 - Salida del programa de tracking de objetos.

También se publicarán los datos para realizar la evaluación mediante el test de KITTI en un fichero contenido por defecto en la carpeta "datos_tracking" en el *home* del usuario.

Los datos almacenados en el fichero tendrán el siguiente aspecto:

```
246 68 50 Cyclist 0.00 0 -0.43 921.08 159.59 987.62 335.10 1.70 0.51 0.72 3.59 1.73 7.91 -0.00 0.61
247 68 47 Cyclist 0.00 2 -0.44 931.95 170.70 978.84 227.49 1.60 0.33 1.86 11.02 1.60 23.39 -0.00 -0.42
248 68 49 Car 0.00 0 -1.12 1011.15 159.02 1179.78 248.47 1.79 1.93 3.63 11.42 1.61 17.33 -0.54 1.00
249 69 50 Cyclist 0.00 0 -0.45 946.95 159.49 1008.64 334.50 1.70 0.53 0.79 3.83 1.73 7.95 -0.00 1.00
250 69 47 Cyclist 0.00 2 -0.46 952.62 173.53 999.05 227.19 1.50 0.24 2.24 11.76 1.50 23.56 -0.00 -0.47
251 69 49 Car 0.00 0 -1.12 1028.17 157.21 1198.31 250.33 1.87 1.90 3.65 11.84 1.69 17.32 -0.52 1.00
252 69 51 Car 0.00 0 -2.56 1127.09 154.52 1224.00 264.05 1.75 2.05 3.10 11.85 1.58 13.84 -1.85 0.24
253 70 50 Cyclist 0.00 0 -0.46 962.72 155.05 1018.37 333.78 1.74 0.48 0.70 3.96 1.73 7.93 -0.00 1.00
254 70 47 Cyclist 0.00 2 -0.47 966.11 170.54 999.76 227.04 1.60 0.30 1.58 11.92 1.60 23.43 -0.00 -0.14
255 70 49 Car 0.00 1 -1.14 1041.87 156.98 1212.99 245.27 1.77 1.95 3.60 12.18 1.59 17.34 -0.52 0.90
256 70 51 Car 0.00 0 -1.66 1146.12 148.70 1224.00 245.50 1.50 1.70 3.70 11.37 1.32 14.07 -0.98 0.57
257 71 50 Cyclist 0.00 0 -0.47 951.20 154.96 1047.12 332.40 1.74 0.57 0.92 4.09 1.73 8.05 -0.00 0.72
258 71 52 Cyclist 0.00 2 -0.49 986.83 170.79 1017.78 207.21 1.60 0.34 0.95 20.18 1.46 37.50 -0.00 0.60
259 71 53 Pedestrian 0.00 0 -0.36 865.52 172.34 906.06 236.05 1.58 0.64 1.06 7.60 1.54 20.31 -0.00 0.75
260 71 49 Car 0.00 2 -2.72 1056.32 154.41 1224.00 243.20 1.78 1.95 3.50 12.51 1.51 17.36 -2.09 -0.10
261 71 51 Car 0.00 0 -1.53 1130.21 149.79 1224.00 238.79 1.37 1.70 3.70 11.10 1.19 14.00 -0.86 0.47
262 72 50 Cyclist 0.00 0 -0.49 969.34 155.30 1047.87 332.36 1.74 0.58 0.99 4.29 1.73 8.14 -0.00 0.44
263 72 52 Cyclist 0.00 2 -0.50 982.75 146.50 1029.37 207.98 2.79 0.59 1.95 20.00 1.42 36.83 -0.00 -0.34
264 72 51 Cyclist 0.00 0 -0.67 1178.93 180.86 1224.00 268.87 1.37 0.36 0.76 10.14 1.60 12.68 -0.00 0.40
265 72 53 Pedestrian 0.00 0 -0.34 853.68 178.16 893.39 236.18 1.42 0.38 0.92 7.12 1.52 19.96 -0.00 0.47
266 72 49 Car 0.00 2 -2.74 1066.68 154.19 1224.00 243.37 1.80 1.97 3.18 12.68 1.53 17.40 -2.11 0.00
267 73 50 Cyclist 0.00 0 -0.49 979.72 155.39 1054.30 331.60 1.73 0.62 0.90 4.38 1.73 8.14 -0.00 0.58
268 73 51 Cyclist 0.00 0 -0.68 1175.80 189.56 1224.00 271.56 1.22 0.43 0.98 9.98 1.65 12.46 -0.00 0.46
269 73 52 Cyclist 0.00 2 -0.50 991.90 168.02 1020.78 208.03 1.73 0.44 1.18 19.88 1.39 36.57 -0.00 0.02
270 73 54 Cyclist 0.00 2 -0.52 1011.70 177.88 1041.29 209.50 1.24 0.47 1.16 19.79 1.60 34.66 -0.00 0.19
271 73 49 Car 0.00 2 -2.69 1077.27 151.37 1221.18 232.80 1.65 1.93 2.23 12.61 1.38 17.27 -2.06 0.29
272 74 50 Cyclist 0.00 0 -0.50 981.29 155.92 1060.91 329.18 1.73 0.63 0.75 4.45 1.73 8.20 -0.00 0.84
273 74 54 Cyclist 0.00 2 -0.52 1016.20 177.86 1046.70 210.29 1.25 0.42 0.71 19.50 1.60 33.77 -0.00 0.38
274 74 49 Car 0.00 2 -2.77 1092.88 150.72 1224.00 232.91 1.68 1.80 2.30 12.96 1.41 17.39 -2.13 0.24
```

Ilustración 100 - Datos de salida del programa de detección de objetos.

Una vez se han obtenido los datos se podrá lanzar el test de evaluación de KITTI para tracking.

9.3.1. Evaluación del tracking de los objetos

Este test de evaluación es muy similar al empleado para evaluar las detecciones de los objetos, sin embargo, en este caso el test está programado en Python y los resultados proporcionados por el test tendrán un formato distinto.

Antes de poder realizar el test de evaluación tendremos que introducir los archivos de **ground truth** proporcionados por la base de datos en una carpeta con el nombre "data" dentro de la carpeta que contenga el propio test de evaluación.

Este test nos proporcionará un archivo llamado "evaluate_tracking.py" que podrá ser lanzado para realizar la evaluación de los datos que hemos obtenido, para ello habrá que seguir los siguientes pasos:

1. Introducir nuestros ficheros de resultados en la siguiente estructura de carpetas dentro del mismo directorio donde se encuentra el test de evaluación: "results/nombre_prueba/data". Todos los ficheros que hemos generado serán introducidos dentro de la carpeta "data" siguiendo la estructura de carpetas mencionada anteriormente.
2. Ejecutar el test de evaluación pasándole como parámetro el nombre de la carpeta donde hemos introducido los datos del test:

```
systick@laptop:~/Downloads/devkit/python$ python evaluate_tracking.py test
```

Ilustración 101 - Evaluación Tracking.

El test generará unos ficheros de texto que contendrán los resultados obtenidos dentro de la carpeta donde hemos introducido nuestros datos. En este caso los resultados solo consistirán de datos numéricos indicando tanto la precisión de las detecciones como las cantidades de objetos a los que se les ha realizado tracking o que se han perdido completamente, así como otros datos indicando la cantidad de objetos a los que se les ha realizado tracking del total de objetos.

Ejemplos de los resultados obtenidos fueron expuestos en el apartado de resultados del programa de tracking de objetos de este mismo trabajo.

9.4. Mask R-CNN

El programa encargado de realizar las detecciones empleando los datos obtenidos de ***Mask R-CNN*** es una simple modificación del programa original de detección de objetos al que se le añadió la funcionalidad de leer los datos previamente generados por ***Mask R-CNN***.

Estos datos consistían en unos ficheros de texto que contenían los datos de las detecciones en la imagen original obtenidos mediante **Mask R-CNN** y que posteriormente serían leídos por el programa simulando que la red se ejecutaba en tiempo real.

9.4.1. Obtención de datos de Mask R-CNN

Para obtener los datos de salida de **Mask R-CNN** se empleó el ***Docker***[25] que contenía uno de los repositorios de ***GitHub*** que contenía una implementación de esta red.

Tras clonar el Docker este se ejecuta mediante el siguiente comando:

```
docker run -it -p 8888:8888 -p 6006:6006 -v ~/:/host waleedka/modern-deep-learning
```

Ilustración 102 - Ejecutar Docker.

Una vez dentro del Docker es necesario ejecutar el ***notebook*** de **Jupyter** con el siguiente comando:

```
jupyter notebook --allow-root
```

Ilustración 103 - Ejecutar Jupyter.

Tras ello se puede acceder al servicio web de ***Jupyter*** en el navegador en la dirección ***localhost:8888.*** Dentro del notebook se creó el programa "generar_datos.ipynb", encargado de leer todas las imágenes de la base de datos de KITTI, procesarlas, realizar las detecciones de los objetos de las mismas y generar los ficheros de datos de salida que contienen la información de las ***bounding boxes*** que serían leídos por el programa de detección. El programa de generación de datos se ejecuta usando la interfaz gráfica proporcionada por el servicio web de ***Jupyter***.

Estos ficheros tienen el contenido mostrado en la siguiente imagen, que se corresponde con el formato de datos de los ficheros almacenados por KITTI pero conteniendo únicamente los datos de las ***bounding boxes*** en la imagen y el ***score***:

```
1 Car 0 0 0 3 203 334 373 0 0 0 0 0 0 0 0.99776757
2 Car 0 0 0 354 184 532 304 0 0 0 0 0 0 0 0.995684
3 Car 0 0 0 731 184 801 247 0 0 0 0 0 0 0 0.990739
4 Car 0 0 0 903 228 1238 369 0 0 0 0 0 0 0 0.98872674
5 Car 0 0 0 710 183 742 224 0 0 0 0 0 0 0 0.97463006
6 Car 0 0 0 682 175 709 197 0 0 0 0 0 0 0 0.9725938
7 Car 0 0 0 563 174 617 217 0 0 0 0 0 0 0 0.9671572
8 Car 0 0 0 605 177 624 204 0 0 0 0 0 0 0 0.9588407
9 Car 0 0 0 628 171 650 190 0 0 0 0 0 0 0 0.9345279
10 Car 0 0 0 641 171 659 191 0 0 0 0 0 0 0 0.832704
```

Ilustración 104 - Contenido fichero de datos Mask R-CNN.

Una vez generados los ficheros se puede ejecutar el programa referenciando correctamente en este el lugar de donde se leen, lo que simulará la ejecución del programa si ***Mask R-CNN*** estuviese enviando esos datos en tiempo real a nuestro programa.

9.4.2. Ejecución del programa de detección con Mask R-CNN

Una vez realizados los pasos anteriores bastará con ejecutar el programa de igual manera que se ejecutaba el programa de detección de objetos, cambiando el ejecutable por el que emplea los datos de ***Mask R-CNN***.

Los pasos son los siguientes:

[25] https://hub.docker.com/r/waleedka/modern-deep-learning/

1. Ejecutar el programa que se encargará de publicar los datos de la base de datos de KITTI que serán leídos por nuestro programa:

```
systick@laptop:~/kitti_ws$ roslaunch kitti_player_tracking kittiplayer_object_standalone_fijo.l
aunch
```

Ilustración 105 - Comando para ejecutar el programa de publicado de datos de KITTI..

2. Posteriormente será necesario lanzar en otro terminal el programa que publicará la nube de puntos coloreada empleando los datos de KITTI. Este programa no generará ninguna salida visual pero se podrá comprobar su funcionamiento usando la herramienta Rviz que habrá sido instalada durante la instalación de ROS. Para ejecutar el programa habrá que lanzar el siguiente comando:

```
systick@laptop:~/robosafe_ws$ roslaunch but_calibration_camera_velodyne coloring_kitti_tracking
.launch
```

Ilustración 106 - Lanzamiento del programa que colorea las nubes de puntos con los datos de la CNN.

3. Finalmente se lanzará el programa que mostrará por pantalla las detecciones de igual manera que lo hacía el programa de detección de objetos, con la diferencia de que estas detecciones habrán sido filtradas empleando las detecciones de **Mask R-CNN**, de tal manera que se eliminará una gran cantidad de falsos positivos mejorando el rendimiento de la aplicación. El comando para lanzar la aplicación es el siguiente.

```
systick@laptop:~$ rosrun kitti_pruebas object_color_rcnn
```

Ilustración 107 - Comando para lanzar el programa de detección de objetos con Mask R-CNN.

Tras ejecutar el comando se visualizarán por pantalla los mismos datos que se visualizaban al lanzar el programa de detección de objetos y se generarán los ficheros de detecciones en la misma carpeta donde se guardaban los ficheros del programa de detección de objetos.

Estos ficheros pueden ser evaluados mediante el test proporcionado por **KITTI** exactamente de la misma forma que fue explicada en el apartado **9.2.1. Evaluación de la detección de objetos.**

9.5. Tracking y Detección de Objetos con Rosbag

Finalmente, para lograr ejecutar el programa que realiza ***tracking*** y ***detección*** de objetos empleando datos que han sido grabados en un coche real por el grupo de investigación ***Robe-Safe*** será necesario realizar los pasos que se expondrán a continuación[26]:

1. Ejecutar el comando ***roscore*** para que los nodos puedan iniciarse correctamente y comunicarse entre ellos:

[26] A pesar de que este manual explica como ejecutar los datos del fichero ".bag" que contiene la información de los sensores, estos datos podrían ser publicados de cualquier otra forma y el programa funcionaría siempre y cuando los nombres de los *topics* fuesen los mismos.

```
systick@laptop:~/robosafe_ws$ roscore
```

Ilustración 108 - Ejecutar Roscore.

2. Poner el parámetro "***use_sim_time***" a ***true*** de forma que se use el tiempo simulado en vez de los ***timestamps*** del momento en el que fueron capturados los datos, ya que esto podría llevar a problemas de sincronización:

```
systick@laptop:~/robosafe_ws$ rosparam set use_sim_time true
```

Ilustración 109 - Activar Tiempo Simulado

3. Ejecutar el programa encargado de publicar las transformadas necesarias para colorear de manera correcta la nube de puntos y para referenciar los sistemas de coordenadas de cada uno de los dispositivos con los que se han obtenido datos (LIDAR, cámaras):

```
systick@laptop:~/robosafe_ws$ roslaunch src/velo2cam_calibration-master/launch/genera_tf_real_.
launch
```

Ilustración 110 - Ejecutar Programa de Generación de Transformadas.

4. Tras ello será necesario ejecutar el programa que colorea la nube de puntos con los datos de la imagen segmentada semánticamente:

```
systick@laptop:~/robosafe_ws$ roslaunch src/velo2cam_calibration-master/launch/pcl_coloring_rea
lbarea.launch
```

Ilustración 111 - Ejecutar Programa Para Colorear la Nube de Puntos.

5. Llegados a este punto los datos que se publicarían están referenciados a las coordenadas de la cámara, la cual está inclinada respecto al LIDAR. Por ello será necesario ejecutar el programa de este paso, que transformará los puntos de la nube de puntos y los publicará de manera que estén alineados con la cámara, para evitar que aparezcan inclinados en el programa final:

```
systick@laptop:~/kitti_ws$ rosrun kitti_pruebas local_transform_camera
```

Ilustración 112 - Transformación de Ejes de Coordenadas.

6. El siguiente paso será empezar a publicar los datos que leerá el programa de tracking. En este caso se han empleado los datos contenidos en un fichero ".bag", pero una vez realizados los pasos anteriores bastaría con que los datos se publicasen en los mismos ***topics*** en los que publica el comando ***rosbag*** y el programa de tracking funcionaría de igual manera. La manera de ejecutar el ***rosbag*** para que se use el tiempo simulado es la siguiente:

```
systick@laptop:~/rosbag$ rosbag play 2018-04-05-18-07-10_seg640x360.bag --loop
--clock
```

Ilustración 113 - Ejecución de Rosbag.

7. Finalmente se puede ejecutar el programa de detección y tracking de objetos. Este programa es el mismo empleado en otros apartados para realizar el tracking empleando los datos de KITTI pero con algunas modificaciones. Estas modificaciones se basan en suscribirse a unos ***topics*** diferentes y en pequeños cambios en el filtrado de las detecciones, ya que el LIDAR empleado por el grupo ***Robe-Safe*** es de 16 haces, mientras que el empleado en la base de datos de KITTI es de 64. El programa de tracking de objetos se ejecuta con el siguiente comando:

```
systick@laptop:~$ rosrun kitti_pruebas tracking_velo2cam_rosbag
```

Ilustración 114 - Programa de Tracking con Rosbag.

Una vez realizados todos estos pasos el programa comenzará a realizar las detecciones mostrando por pantalla las cajas mínimas de los objetos detectados como se puede ver en la siguiente imagen:

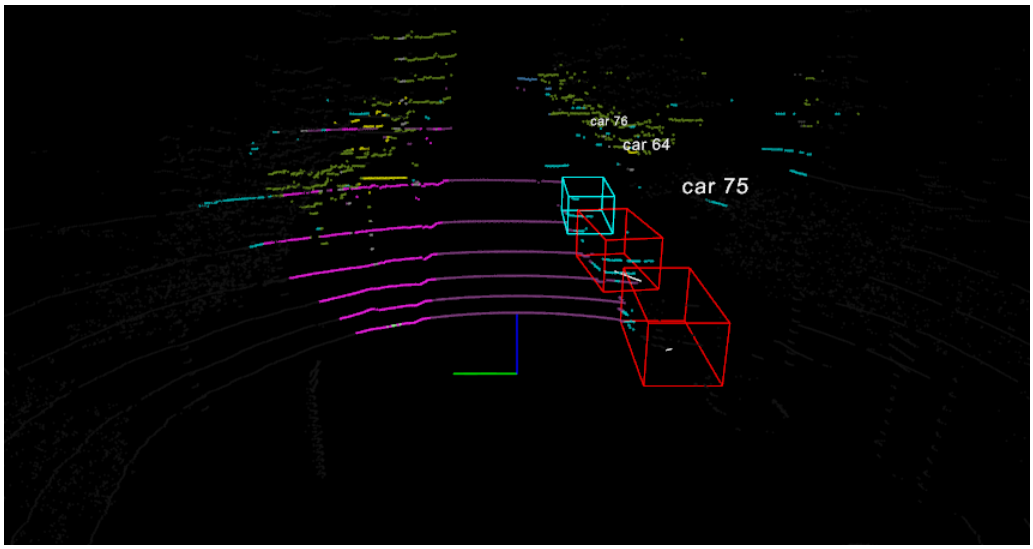


Ilustración 115 - Detecciones Rosbag.

Se puede apreciar claramente una diferencia con la densidad de datos del LIDAR que se obtenía cuando se usaba la base de datos de KITTI, lo que supone un empeoramiento de la calidad de las detecciones a pesar de que se han ajustado los parámetros de filtrado de detecciones y de eliminación de falsos positivos.

En este caso no es posible realizar una evaluación de los resultados ya que no se dispone de datos de ***ground truth***. Sin embargo se puede apreciar que las detecciones son menos precisas por la inferior cantidad de puntos del LIDAR.

10. Pliego de Condiciones

Para la realización de este proyecto se han empleado tanto un **equipo físico** como un conjunto de **herramientas software**.

Hardware

- Ordenador portátil MSI GP60
    - CPU: Intel Core i7 4700HQ (4 núcleos / 8 hilos) a 2.40/3.40 GHz
    - GPU: Nvidia GT840m
    - RAM: 8GB DDR3 1600MHz

Software

- Sistema operativo: Ubuntu 14.04
- Robot Operating System (ROS)
- OpenCV
- Matlab
- Atom
- Office 365
- Programas de evaluación de KITTI

11. Presupuesto

En este apartado se expondrá el **presupuesto** estimado del coste de realización de este proyecto. En el coste se incluirá tanto el valor de los equipos utilizados como el precio de las licencias de los programas que requerían una licencia para ser utilizados.

11.2. Costes Materiales

Concepto	Descripción	Cantidad	Precio (€)	Subtotal (€)
Ordenador portátil	MSI GP60	1	750	750
Ubuntu 14.04	Sistema operativo Licencia GNU	1	0	0
ROS	Framework para el desarrollo de software	1	0	0
OpenCV	Bibiloteca de visión artificial	1	0	0
Matlab	IDE de programación	1	0	0
Atom	Editor de texto MIT License	1	0	0
Office 365	Conjunto de programas de Office	1	0	0
Programas de evaluación de KITTI	Programas para evaluar los resultados obtenidos	1	0	0
Total:				**750 €**

El precio de una gran parte de los programas utilizados ha sido 0 debido a que se trata de software libre o a que las licencias han sido proporcionadas por la universidad, como en el caso de Office 365.

11.3. Coste del Trabajo

En este apartado se incluyen los costes relacionados con el tiempo dedicado a la realización del proyecto.

Concepto	**Tiempo (Meses)**	**Coste (€/Mes)**	**Subtotal (€)**
Ingeniería	8	1350	10800
Escritura	3	650	1950
Total:			**12750 €**

11.4. Coste Total

En este apartado se muestra el coste total del proyecto tras aplicar el IVA correspondiente.

Concepto	Subtotal (€)
Costes Materiales	750
Coste del Trabajo	12750
Subtotal	13500
IVA(21%)	2835
TOTAL:	**16335€**

12. Bibliografía

[1] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, Tian Xia. "Multi-View 3D Object Detection Network for Autonomous Driving". IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2017.

[2] Ku, Jason and Mozifian, Melissa and Lee, Jungwook and Harakeh, Ali and Waslander, Steven, "Joint 3D Proposal Generation and Object Detection from View Aggregation", arXiv preprint arXiv:1712.02294: https://arxiv.org/pdf/1712.02294.pdf.

[3] The KITTI Vision Benchmark Suite: http://www.cvlibs.net/datasets/kitti/index.php.

[4] Eduardo Romera, José M. Álvarez, Luis M. Bergasa, Roberto Arroyo." ERFNet: Efficient Residual Factorized ConvNet for Real-Time Semantic Segmentation". IEEE Transactions on Intelligent Transportation Systems (Volume: 19, Issue: 1, Jan. 2018).

[5] Willow Garage, ROS Indigo Igloo version: http://wiki.ros.org/indigo.

[6] Euclidean Cluster Extraction: http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php.

[7] Hough Line Transform: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html.

[8] Mantripragada, Kiran & Trigo, Flavio & Martins, Flavius & Fleury, A.T.. (2013). Vehicle Tracking Using Feature Matching and Kalman Filtering.

[9] Kalman Filter Implementation: http://opencvexamples.blogspot.com/2014/01/kalman-filter-implementation-tracking.html.

[10] Bo Li. "3D Fully Convolutional Network for Vehicle Detection in Point Cloud".

[11] Yin Zhou, Oncel Tuzel. "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection": https://arxiv.org/pdf/1711.06396.pdf.

[12] Bichen Wu, Alvin Wan, Xiangyu Yue, Kurt Keutzer. "SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud": https://arxiv.org/pdf/1710.07368.pdf.

[13] Abdul Hadi Abd Rahman, Hairi Zamzuri, Saiful Amri Mazlan, Mohd Azizi Abdul Rahman. "Model-Based Detection and Tracking of Single Moving Object Using Laser Range Finder". 2014 5th International Conference on Intelligent Systems, Modelling and Simulation.

[14] Wongun Choi. "Near-Online Multi-target Tracking with Aggregated Local Flow Descriptor".

[15] Yutong Ye, Liming Fu, Bijun Li. "Object Detection and Tracking Using Multi-layer Laser for Autonomous Urban Driving". 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC).

[16] Qi, Charles R and Liu, Wei and Wu, Chenxia and Su, Hao and Guibas, Leonidas J, "Frustum PointNets for 3D Object Detection from RGB-D Data", arXiv preprint arXiv:1711.08488: https://arxiv.org/pdf/1711.08488.pdf.

[17] F. Chabot and M. Chaouch and J. Rabarisoa and C. Teulière and T. Chateau, “Deep MANTA: A Coarse-to-fine Many-Task Network for joint 2D and 3D vehicle analysis from monocular image”, CVPR: https://arxiv.org/pdf/1703.07570.pdf.

[18] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick, “Mask R-CNN”, CVPR: https://arxiv.org/pdf/1703.06870.pdf