Proyecto de Excelencia P21-00684. PREMOVE. *Tracking* y predicción del movimiento con BEVFusion, ROS y TensorRT

Contents

1 Generación del conjunto de datos sintético PREMOVE_CARLA

Con el objetivo de evaluar distintas redes neuronales para detectar participantes del tráfico, seguimiento de objetos y predicción de trayectorias, se generó un conjunto de datos sintético mediante el simulador CARLA, denominado `PREMOVE_CARLA`. Para permitir la compatibilidad con arquitecturas de percepción existentes, dicho conjunto de datos se construyó siguiendo un formato similar al del conjunto de datos `nuScenes`, utilizando la herramienta `SimBEV` para la generación y guardado de los datos procedentes de CARLA.
La simulación se llevó a cabo sobre un vehículo sensorizado con una configuración similar a la empleada en la plataforma del proyecto PREMOVE. Esta configuración se compone de un sensor LiDAR 3D y un conjunto de cámaras frontales. En concreto, se utilizaron tres cámaras RGB frontales con una resolución de 1600×900 px y un campo de visión equivalente al de las cámaras ZED 2i. Adicionalmente, se incorporaron cuatro cámaras RGB frontales destinadas a emular sensores térmicos, configuradas con la misma resolución (320×240 px) y campo de visión que las cámaras Seek Thermal Compact Pro. Cabe destacar que, debido a las limitaciones actuales del simulador CARLA (version 0.9.16), no fue posible incorporar cámaras térmicas. Por este motivo, las cámaras térmicas se emularon mediante cámaras RGB, manteniendo las mismas características del sensor real.
El conjunto de datos generado cuenta con un total de ocho escenas, grabadas en el mapa *Town15* de CARLA. De estas escenas, cuatro se destinaron al entrenamiento de las redes neuronales, dos a la validación y dos para pruebas. Cada escena incluye participantes del tráfico, tanto dinámicos como estáticos. Entre los participantes dinámicos se encuentran coches, peatones, camiones, motocicletas y bicicletas, mientras que los participantes estáticos son coches y camiones estacionados.

1.1 Guía para la generación del conjunto de datos sintético

Para la generación del conjunto de datos es necesario partir de un fichero de configuración que define los parámetros de la simulación y la disposición de los sensores. En primer lugar, se ejecutó el script principal para la captura de datos desde CARLA:

```
python3 simbev.py sample_config.yaml --path /mnt/dataHDD/Datasets/
    PREMOVE_v4
```

Una vez completada la captura de datos, se llevó a cabo una etapa de postprocesado para eliminar aquellas muestras que no tuvieran, al menos, un punto en la nube del LiDAR 3D:

```
python3 post_processing.py --path /mnt/dataHDD/Datasets/PREMOVE_v4
```

Finalmente, se emplearon las herramientas de visualización proporcionadas por SimBEV para verificar de forma cualitativa los datos generados, tanto a nivel global como por escena individual. Para la visualización de las imágenes RGB del conjunto completo se utilizó el siguiente comando:

```
python3 visualization.py --mode rgb --path /mnt/dataHDD/Datasets/
    PREMOVE_v4
```

Asimismo, es posible visualizar escenas concretas del conjunto de datos especificando el identificador de la escena:

```
python3 visualization.py --mode rgb --path /mnt/dataHDD/Datasets/
    PREMOVE_v4 --scene 0005
```

2 Entrenamiento de la red neuronal BEVFusion

Para el entrenamiento de la red neuronal BEVFusion, se utilizó una versión modificada y más reciente del modelo original, desarrollada por Goodarz Mehr (https://github.com/GoodarzMehr/bevfusion/tree/develop). Esta versión corrige algunas limitaciones presentes en la implementación original de BEVFusion y está diseñada para ofrecer una mayor flexibilidad en la estructura de los conjuntos de datos, lo cual es de utilidad para trabajar con configuraciones sensoriales personalizadas.

2.1 Modificaciones realizadas en la arquitectura

Con el fin de adaptar el modelo al conjunto de datos generado en CARLA mediante SimBEV, fue necesario realizar varias modificaciones tanto a nivel de dataset como en el flujo de entrenamiento. En primer lugar, se realizaron cambios en el archivo `mmdet3d/datasets/simbev_dataset.py` para soportar un mayor número de cámaras y nombres personalizados. En concreto, se modificó la lista `CAM_NAME` y el bucle que filtraba las cámaras originales, sustituyéndola por los nombres de las cámaras utilizadas en el conjunto de datos sintético. Asimismo, se modificó la línea encargada de cargar las imágenes y los parámetros de las cámaras para que coincidiera con los del *dataset*, pasando de: `data['image_paths'].append(info['RGB-' + camera])` a: `data['image_paths'].append(info[camera])`,
Además, fue necesario asociar la carpeta `dataset/simbev` del repositorio con la ubicación real del conjunto de datos generado. Para ello, se creó un enlace simbólico que apunta al directorio correspondiente:

```
ln -s /media/premovesim/data/Datasets/PREMOVE_v4/simbev ~/bevfusion
    /dataset
```

Dentro del directorio `/dataset/simbev/infos`, se modificaron los ficheros `simbev_infos_{train,val,test}.json` para asegurar que las rutas de las imágenes apuntasen correctamente a su ubicación real. Esta modificación fue necesaria debido a discrepancias en la estructura de discos entre los sistemas Ubuntu 20.04 (usado para entrenar la red neuronal) y Ubuntu 22.04 (usado para generar el conjunto de datos con CARLA).
Durante el entrenamiento, se detectó que al aumentar el número de cámaras del *dataset* y en las últimas etapas del entrenamiento aparecía el error:
`ValueError: matrix contains invalid numeric entries`
Este problema se resolvió ajustando los parámetros de entrenamiento, principalmente reduciendo la tasa de aprendizaje (*learning rate*).

2.2 Guía para entrenar BEVFusion y validar los resultados

El entrenamiento de la red neuronal se llevó a cabo utilizando `torchpack`, partiendo de los pesos preentrenados para la rama del LiDAR.

```
torchpack dist-run -np 1 python tools/train.py \
```

```
configs/simbev/det/transfusion/secfpn/camera+lidar/resnet50/
   convfuser.yaml \
--load_from pretrained/simbev-lidar-only-det.pth \
--run-dir runs/run-PREMOVE_v4-resnet50
```

Una vez completado el entrenamiento, se evaluó el rendimiento del modelo mediante la métrica *mAP* sobre el conjunto de validación:

```
torchpack dist-run -np 1 python tools/test.py \
configs/simbev/det/transfusion/secfpn/camera+lidar/resnet50/
   convfuser.yaml \
runs/run-PREMOVE_v4-resnet50/epoch_20.pth --eval mAP
```

Para el análisis cualitativa de los resultados, se emplearon las herramientas de visualización proporcionadas por el repositorio, añadiendo las cajas de detección sobre las imágenes y nubes de puntos del *dataset* a partir del modelo entrenado:

```
torchpack dist-run -np 1 python tools/visualize.py \
~/bevfusion/runs/run-PREMOVE_v4-resnet50/configs.yaml \
--mode pred-simbev --checkpoint epoch_20.pth \
--split val --bbox-score 0.1 --out-dir viz_PREMOVE
```

Cabe destacar que, para evitar la interrupción del visualizador por saturación de la memoria RAM en grandes *datasets*, se modificó el fichero `mmdet3d/core/utils/visualize.py` y, en su cabecera se añadió la línea `matplotlib.use("Agg")`.

2.3 Guía de aceleración del modelo entrenado con TensorRT

Posteriormente, se aplicó un proceso de *Post Training Quantization* (PTQ) como paso previo a acelerar la inferencia con TensorRT. Para ello, se modificó el script `ptq.py` añadiendo un argumento que permite especificar el directorio de salida del modelo cuantizado. El proceso de cuantización se ejecutó con el siguiente comando:

```
python onnx/qat/ptq.py \
--config configs/simbev/det/transfusion/secfpn/camera+lidar/
   resnet50/convfuser.yaml \
--ckpt onnx/model/PREMOVEv3-resnet50/bevfusion-det.pth \
--calibrate_batch 300 \
--save_path onnx/model/PREMOVEv4-resnet50/bevfusion_ptq.pth
```

Para la exportación del modelo a formato ONNX, fue necesario generar los ficheros de datos de ejemplo (`example-data`), que se utilizan para definir las dimensiones y tipos de entrada del modelo. Dichos ficheros se obtuvieron mediante los siguientes comandos:

```
python example-data/dump-data.py
python example-data/dump-data.py \
--config bevfusion/configs/simbev/det/transfusion/secfpn/camera+
   lidar/resnet50/convfuser.yaml
```

El código fue modificado para que los datos se almacenaran dentro del directorio propio de `example-data`, sustituyendo la línea:
`root = f"dump/{i:05d}"`

por:
`root = f"example-data/dump/{i:05d}"`
Adicionalmente, fue necesario adaptar el tensor de puntos del LiDAR para que fuera compatible con la configuración de entrada del modelo. En concreto, se añadió una quinta columna de ceros al tensor `points.tensor`, mediante el siguiente código de Python:

```
simbev = load('points.tensor')
zeros_column = np.zeros((simbev.shape[0], 1))
simbev_with_zeros = np.hstack((simbev, zeros_column))

simbev_with_zeros_float16 = simbev_with_zeros.astype(np.float16)
save(simbev_with_zeros_float16, 'points_ceros.tensor')
```

Una vez preparados los datos de ejemplo, se procedió a la exportación de los distintos componentes del modelo a formato ONNX. En particular, se exportaron los módulos correspondientes a las cámaras, al bloque de fusión y al *backbone* del LiDAR:

```
python onnx/qat/export-camera.py \
--ckpt onnx/model/PREMOVEv4-resnet50/bevfusion_ptq.pth

python onnx/qat/export-transfuser.py \
--ckpt onnx/model/PREMOVEv4-resnet50/bevfusion_ptq.pth

python onnx/qat/export-scn.py \
--ckpt onnx/model/PREMOVEv4-resnet50/bevfusion_ptq.pth \
--save qat/onnx_int8/lidar.backbone.onnx \
--in-channel 4
```

El parámetro `in-channel 4` corresponde a la configuración definida en el archivo `default.yaml`, donde se especifica que la entrada del backbone del LiDAR consta de cuatro canales.
Finalmente, para la creación de los motores de inferencia optimizados con TensorRT, se ajustó la variable de entorno `DEBUG_MODEL` en el script `environment.sh` para que apuntase al modelo deseado. Una vez configurado este parámetro, se compiló el modelo, generando ya su versión optimizado:

```
bash tool/build_trt_engine.sh
```

3 Detección de participantes del tráfico (BEVFusion-ROS-TensorRT)

Para esta tarea se ha fusionado un LiDAR y siete cámaras, empleando para ello la arquitectura BEVFusion. Con el objetivo se usar dicha arquitectura con datos en línea y aceleración mediante TensorRT, se ha recurrido al paquete BEVFusion-ROS-TensorRT, en su versión para ROS2 Humble. Dicho paquete, originalmente estaba diseñado para emplear un LiDAR y seis cámaras RGB iguales. En el marco del proyecto PREMOVE, se ha modificado dicho paquete para incluir un total de siete cámaras con parámetros distintos y publicar las detecciones en un tópico de ROS2. Una descripción de los cambios realizados se muestra en los apartados 3.1 y 3.2, los pasos para su instalación en el apartado 3.3 y una muestra de los resultados en el apartado 3.4. El código del

paquete se incluye en el repositorio de GitHub https://github.com/jorgemn9/ROS2_BEVFusion.

3.1 Modificaciones para la ampliación del número de cámaras y resoluciones diferentes

Las modificaciones que se muestran a continuación tienen como objetivo ampliar el número de cámaras soportadas por el sistema y permitir el uso simultáneo de cámaras con diferentes. Estos cambios afectan a varias etapas del flujo de procesamiento de datos, como la definición de las entradas, la normalización, la geometría del sistema y la visualización.
En primer lugar, se modificó el array `file_names[]` en el fichero `src/main.cpp` con los nuevos nombres de las imágenes de entrada, permitiendo distinguir entre diferentes tipos de cámaras:

```
const char* file_names[] = {
    "0-RGB_FRONT_LEFT.jpg", "1-RGB_FRONT.jpg", "2-RGB_FRONT_RIGHT.
       jpg",
    "3-THERMAL_LEFT.jpg",   "4-THERMAL_LEFT_CENTRAL.jpg",
    "5-THERMAL_RIGHT_CENTRAL.jpg", "6-THERMAL_RIGHT.jpg"
};
```

Como consecuencia de esta ampliación, se actualizó el número de cámaras definidas en la etapa de normalización y geometría del sistema:

```
normalization.num_camera = 7;
geometry.num_camera = 7;
```

Asimismo, el bucle encargado de la carga de imágenes se ajustó para procesar las siete cámaras disponibles:

```
for (int i = 0; i < 7; ++i) {
    char path[200];
    sprintf(path, "%s/%s", root.c_str(), file_names[i]);
    images.push_back(stbi_load(path, &width, &height, &channels, 0)
       );
}
```

Posteriormente, en el fichero `src_plugin/bevfusion_plugin.cpp`, se adaptó la visualización en formato collage a la nueva configuración de cámaras. Para ello, se modificó la matriz `offset_cameras[]`, ajustando el orden y la posición relativa de cada sensor. En particular, se añadió una posición específica para la cámara adicional, facilitando su identificación dentro del entorno visual:

```
int offset_cameras[][3] = {
    {-content_width / 2 + gap, -content_height / 2 + camera_height
       / 2, 0},
    {-camera_width / 2, -content_height / 2 + gap, 0},
    {content_width / 2 - camera_width - gap, -content_height / 2 +
       camera_height / 2, 0},

    {-content_width / 2 + gap, +content_height / 2 - camera_height
       - camera_height / 2, 0},
```

```
    {-camera_width / 2 - 250, +content_height / 2 - camera_height -
        gap, 1},
    {-camera_width / 2 + 250, +content_height / 2 - camera_height -
        gap, 1},
    {content_width / 2 - camera_width - gap, +content_height / 2 -
       camera_height - camera_height / 2, 1}};
};
```

Con el fin de soportar cámaras con resoluciones distintas, en vez de depender de un único parámetro global de resolución para las cámaras, se introdujo un nuevo array que define la resolución de cada cámara:

```
std::vector<std::pair<int, int>> camera_resolutions = {
    {1600, 900}, {1600, 900}, {1600, 900},
    {320, 240}, {320, 240}, {320, 240}, {320, 240}
};
```

Finalmente, para garantizar una correcta visualización y renderizado de cada cámara, el objeto `image_artist_param` se trasladó dentro del bucle de iteración por cámara. Esto permite que los parámetros gráficos se configuran de forma individual según la resolución de cada sensor:

```
for (size_t icamera = 0; icamera < images.size(); ++icamera) {
    int camera_width_current = camera_resolutions[icamera].first;
    int camera_height_current = camera_resolutions[icamera].second;
    nv::ImageArtistParameter image_artist_param;
    image_artist_param.num_camera = images.size();
    image_artist_param.image_width = camera_width_current;
    image_artist_param.image_height = camera_height_current;

    image_artist_param.image_stride = image_artist_param.
       image_width * 3;
    image_artist_param.viewport_nx4x4.resize(images.size() * 4 * 4)
       ;
    memcpy(image_artist_param.viewport_nx4x4.data(), lidar2image.
       ptr<float>(),
          sizeof(float) * image_artist_param.viewport_nx4x4.size())
             ;

    ...
}
```

3.2 Modificaciones para publicar las detecciones en un tópico

Con el fin de publicar las detecciones generadas por BEVFusion en un nodo de ROS2, se creó un paquete de mensajes denominado `bevfusion_msgs` y se modificó nuevamente el paquete BEVFusion-ROS-TensorRT.

El paquete `bevfusion_msgs` define dos tipos de mensajes: `Prediction.msg` y `PredictionArray.msg`. El primero representa una detección individual, mientras que el se-

gundo permite agrupar múltiples detecciones en un único mensaje. El mensaje `Prediction.msg` contiene información relativa al identificador (ID), clase, posición, tamaño, velocidad, orientación y confianza de cada participante del tráfico detectado:

```
int32 id
string class_name
geometry_msgs/Point position
geometry_msgs/Vector3 size
geometry_msgs/Vector3 velocity
float32 z_rotation
float32 score
```

Por su parte, el mensaje `PredictionArray.msg` agrupa un conjunto de predicciones junto con un encabezado estándar de ROS2:

```
std_msgs/Header header
bevfusion_msgs/Prediction[] predictions
```

Una vez definido el paquete de mensajes, se modificó el paquete BEVFusion-ROS-TensorRT para permitir la publicación de las detecciones. En el archivo `include_plugin/bevfusion_plugin.hpp` se declararon los publicadores necesarios para difundir tanto las detecciones como los marcadores de visualización:

```
rclcpp::Publisher<bevfusion_msgs::msg::PredictionArray>::SharedPtr
    prediction_pub_;
rclcpp::Publisher<visualization_msgs::msg::MarkerArray>::SharedPtr
    marker_pub_;
```

Posteriormente, en el archivo `src_plugin/bevfusion_plugin.cpp`, se inicializan dichos publicadores, definiendo los tópicos ROS 2 correspondientes:

```
if (!rclcpp::ok()) {
    rclcpp::init(0, nullptr);
}

prediction_pub_ = this->create_publisher<
    bevfusion_msgs::msg::PredictionArray>("bevfusion_detections",
        10);

marker_pub_ = this->create_publisher<
    visualization_msgs::msg::MarkerArray>("/bevfusion/markers", 10)
        ;
```

El tópico `bevfusion_detections` se utiliza para publicar las detecciones procesadas, mientras que `/bevfusion/markers` se emplea para la visualización de estas en RViz, junto a su identificador, etiqueta de clase y precisión media.
Durante la ejecución, el proceso de inferencia se realiza en la función `Inference`, donde se obtienen las cajas de predicción tras la llamada al método `core->forward`:

```
auto bboxes = core->forward(
    (const unsigned char**)images_data.data(),
    lidar_half.ptr<nvtype::half>(),
```

```
    lidar_data.size(0),
    stream
);
```

Una vez obtenidas las predicciones, estas se transforman en mensajes ROS2 del tipo `Prediction` y se agrupan en un mensaje `PredictionArray`, que finalmente se publica en el tópico correspondiente:

```
bevfusion_msgs::msg::PredictionArray array_msg;

for (const auto& pred : predictions) {
    bevfusion_msgs::msg::Prediction msg;
    msg.id = pred.id;
    msg.class_name = (pred.id < classes.size()
                      ? classes[pred.id].name
                      : "unknown");

    msg.position.x = pred.position.x;
    msg.position.y = pred.position.y;
    msg.position.z = pred.position.z;

    msg.size.x = pred.size.w;
    msg.size.y = pred.size.l;
    msg.size.z = pred.size.h;

    msg.velocity.x = pred.velocity.vx;
    msg.velocity.y = pred.velocity.vy;
    msg.velocity.z = 0.0;

    msg.z_rotation = pred.z_rotation;
    msg.score = pred.score;

    array_msg.predictions.push_back(msg);
}

prediction_pub_->publish(array_msg);
```

3.3 Guía de instalación

1. Compilar `protobuf v3.6.1` siguiendo las instrucciones del repositorio oficial de dicha versión y anotar la ruta donde ha sido instalado.
2. Modificar la línea 41 del archivo `CMakeLists.txt` del proyecto, actualizando la variable correspondiente con la ruta del protobuf compilado.
3. Crear el espacio de trabajo de ROS2:

   ```
   mkdir -p ros2_ws/src
   ```

4. Clonar el repositorio del proyecto:

```
cd bevfusion_ws/src
git clone https://github.com/linClubs/BEVFusion-ROS-TensorRT.
    git
```

5. Cambiar a la rama de desarrollo para ROS2 Humble:

```
cd BEVFusion-ROS-TensorRT
git checkout humble-devel
cd ..
```

6. Instalar las dependencias de ROS necesarias:

```
cd ..
rosdep install -r -y --from-paths src --ignore-src --rosdistro
    $ROS_DISTRO
```

7. Antes de compilar, es necesario modificar las rutas de CUDA y TensorRT en el archivo `CMakeLists.txt`:

```
# CUDA
set(CUDA_TOOLKIT_ROOT_DIR /usr/local/cuda-11.3)
set(CUDA_INSTALL_TARGET_DIR targets/x86_64-linux)
set(CUDA_INCLUDE_DIRS ${CUDA_TOOLKIT_ROOT_DIR}/${
    CUDA_INSTALL_TARGET_DIR}/include)
set(CUDA_LIBS ${CUDA_TOOLKIT_ROOT_DIR}/${
    CUDA_INSTALL_TARGET_DIR}/lib)

# TensorRT
set(TensorRT_ROOT ~/software/TensorRT-8.5.3.1)
set(TensorRT_INCLUDE_DIRS ${TensorRT_ROOT}/include)
set(TensorRT_LIBS ${TensorRT_ROOT}/lib/)
```

Asimismo, se deben configurar los parámetros del archivo `bevfusion.launch.py`. Para este proyecto, se ha empleado un modelo específico entrenado bajo ResNet50, denominado `PREMOVEv4-resnet50`, entrenado sobre el *dataset* `PREMOVEv4`:

```
# model_name: resnet50 / resnet50int8 / swint
# precision: fp16 / int8
# swint + int8 no es compatible
parameters=[
    {'model_name': 'PREMOVEv4-resnet50'},
    {'precision' : 'int8'}
]
```

8. Modificar los tópicos de las imágenes (`sub_img_XX_`) y de la nube de puntos (`sub_cloud_`) en el fichero `src_ros/bevfusion_node.cpp`.

9. Compilar el proyecto:

```
colcon build --symlink-install
```

10. Cargar el entorno del espacio de trabajo:

```
source install/setup.bash
```

11. Ejecutar el nodo de BEVFusion:

```
ros2 launch bevfusion bevfusion.launch.py
```

3.4 Resultados del paquete BEVFusion-ROS-TensorRT sobre datos del simulador CARLA

En la Figura 1 se muestra el simulador CARLA, sobre el cual se ha cargado el mapa *Town15* y un vehículo sensorizado con una configuración similar a la de la plataforma del proyecto PREMOVE. El vehículo ha sido teleoperado hacia la posición mostrada en la Figura 1 mediante un mando DualShock 4. En la Figura 2 se presenta una visualización en RViz de la nube de puntos del LiDAR publicada por CARLA. Sobre dicha nube se superponen las cajas de detección correspondientes a los distintos participantes del tráfico, generadas por el paquete BEVFusion-ROS-TensorRT. Estas cajas incluyen la clase detectado y su precisión media.

Figure 1: Simulador CARLA con el mapa Town15 y un vehículo sensorizado con una configuración similar a la plataforma PREMOVE.

Por otro lado, en la Figura 3 se muestra la visualización en forma de collage generada por el propio paquete BEVFusion-ROS-TensorRT. En el centro de la imagen se sitúa la nube de puntos del LiDAR, mientras que en la parte superior se representan tres cámaras RGB con resolución 1600×900, correspondientes, de izquierda a derecha, a las cámaras frontal izquierda, frontal central y frontal derecha. En la parte inferior se incluyen cuatro cámaras RGB con resolución

320 × 240, que corresponden, de izquierda a derecha, a las cámaras frontal izquierda, frontal izquierda central, frontal derecha central y frontal derecha. Sobre las imágenes y la nube de puntosse muestra igualmente la clase detectada y la precisión media.

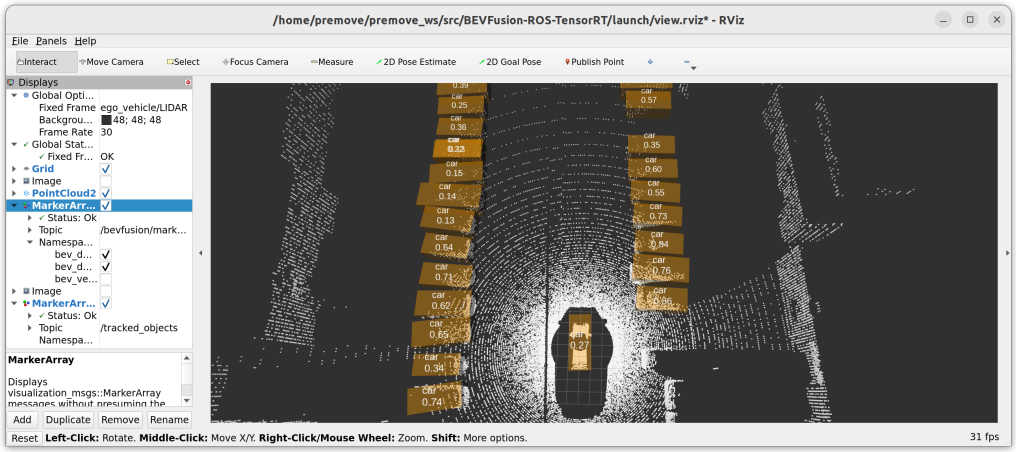

Figure 2: Visualización en RViz de la nube de puntos del LiDAR junto con las cajas de detección generadas por BEVFusion-ROS-TensorRT.

Figure 3: Visualización en forma de collage generada por BEVFusion-ROS-TensorRT, mostrando la nube de puntos del LiDAR y las imágenes de las cámaras RGB con diferentes resoluciones.

4 Seguimiento de las detecciones (*Tracking*)

Una vez adaptado el paquete BEVFusion-ROS-TensorRT a las necesidades del proyecto PREMOVE, se procedió a realizar el seguimiento de las detecciones (*tracking*). Para ello, se ha implementado el módulo de *tracking* de CenterPoint, una arquitectura para la detección de objetos en nubes de puntos. El principal objetivo del seguimiento es asociar detecciones tridimensionales consecutivas y generar identificadores para cada participante del tráfico detectado. Cada objeto seguido se representa mediante un vector de estado definido en el plano BEV, compuesto por la posición y la velocidad del objeto:

$$det = [x, y, v_x, v_y]$$

donde x e y corresponden a la posición del centro del objeto en el espacio vista de pájaro, mientras que v_x y v_y representan la velocidad estimada en dicho plano. Cabe destacar que atributos como la orientación y las dimensiones del objeto no forman parte del estado del filtro ya que no son necesarios para realizar un seguimiento de cada detección. El seguimiento se implementa mediante un filtro de Kalman lineal con un modelo de movimiento de velocidad constante. En cada iteración, el estado del objeto se predice según la siguiente ecuación:

$$x_k = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_{k-1} + w$$

donde Δt representa el intervalo temporal entre frames consecutivos y w modela el ruido del proceso. Este modelo permite predecir la posición futura del objeto y aumentar la robustez del seguimiento frente a pérdidas puntuales de la detección. La asociación entre las detecciones actuales y los objetos previamente seguidos se realiza comparando las posiciones predichas por el filtro de Kalman (x_k, y_k) con las posiciones de las nuevas detecciones (x_{k+1}, y_{k+1}). Para ello, se utiliza la distancia euclídea en el plano BEV como métrica de coste:

$$d = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}$$

A partir de esta métrica se construye una matriz de costes, que se resuelve mediante el algoritmo húngaro (*hungarian assigner*) para obtener la asignación óptima entre detecciones y trayectorias. Las asociaciones cuya distancia supera un umbral predefinido se descartan, evitando emparejamientos erróneos. En cuanto a su funcionamiento, el seguimiento de objetos en CenterPoint usa un identificador único para cada objeto detectado a lo largo del tiempo. Cuando aparece una detección que no puede asociarse a ningún objeto previamente seguido, se crea una nueva trayectoria y se le asigna un nuevo identificador. Si un objeto deja de detectarse temporalmente, por ejemplo debido a una oclusión o a una pérdida puntual de detección, su seguimiento no se elimina de inmediato. En su lugar, el sistema conserva dicho seguimiento durante un número determinado y ajustable de intentos, esperando a que el objeto vuelva a aparecer en una posición coherente con su movimiento previo. Únicamente cuando un objeto no vuelve a detectarse tras superar un umbral temporal, la trayectoria asociada se elimina definitivamente.
En el contexto del proyecto PREMOVE, la lógica del módulo de *tracking* de CenterPoint se ha implementado en un paquete de creación propia denominado `bevfusion_tracker`, que se encuentra en el repositorio de GitHub https://github.com/jorgemn9/ROS2_BEVFusion.

Desde el punto de vista de ROS2, el paquete cuenta con un nodo principal denominado `TrackerNode`, el cual actúa como interfaz entre la detección y el seguimiento. Este nodo se suscribe al tópico `/bevfusion_detections`, donde se reciben las detecciones generadas por BEVFusion, y utiliza internamente la clase `PubTracker` para gestionar el seguimiento, que implementa el algoritmo de CenterPoint
El comportamiento del algoritmo de seguimiento se controla mediante parámetros, como el número máximo de instantes sin seguimiento permitidos (`max_age`), el umbral mínimo de confianza de las detecciones (`detection_th`) y el conjunto de datos utilizado para interpretar las detecciones (`dataset`), siendo `simbev` el valor por defecto.
Como resultado, dicho paquete publica el seguimiento de los participantes del tráfico en el tópico `tracked_objects`, como mensajes del tipo `visualization_msgs/MarkerArray`. Este tópico incluye los siguientes atributos:

- Un identificador único de seguimiento.
- Un identificador de la clase detectada y su etiqueta (heredada de BEVFusion).
- La posición tridimensional y las dimensiones de las cajas de detección de cada participante en coordenadas del mundo (heredada de BEVFusion).
- La velocidad en formato vectorial estimada (heredada de BEVFusion).
- La precisión de la detección (heredada de BEVFusion).

4.1 Predicción de las trayectorias de los participantes del tráfico

Una vez realizado el seguimiento de las detecciones de los participantes del tráfico, se realizó una estimación de su trayectoria. Dicha estimación se basa en un modelo cinemático de velocidad constante, en el que se emplea la velocidad estimada por BEVFusion para cada participante. A partir del estado actual del objeto y asumiendo movimiento uniforme, la posición futura se calcula para un horizonte temporal configurable, que en este trabajo se ha fijado en 2 s. La estimación de la posición futura $(x(t), y(t))$ de cada participante se obtiene según la siguiente expresión:

$$x(t) = x_0 + v_x \cdot t$$
$$y(t) = y_0 + v_y \cdot t$$

donde (x_0, y_0) representa la posición actual del objeto en el espacio BEV, (v_x, v_y) corresponde a la velocidad estimada y t es el horizonte de predicción.

4.2 Resultados del paquete `bevfusion_tracker` sobre datos del simulador CARLA

Para obtener los resultados de seguimiento sobre el simulador CARLA, se ha empleado el mismo vehículo en la misma posición mostrada en la Figura 1. En la Figura 4 se presenta una visualización en RViz de la nube de puntos del LiDAR publicada por CARLA. Sobre dicha nube se superponen las cajas de detección correspondientes a los distintos participantes del tráfico y la flecha con su trayectoria predicha con un horizonte temporal de 2 s. Sobre cada caja se incluye la etiqueta del participante detectado, la precisión de la detección y su identificador de seguimiento. A modo de

comprobación del seguimiento, se puede observar cómo los vehículos estacionados en los laterales no cuenta con trayectoria predicha, ya que no se mueven, mientras que el resto de vehículos en circulación cuenta con una flecha que indica su trayectoria predicha, en sentido y dirección.

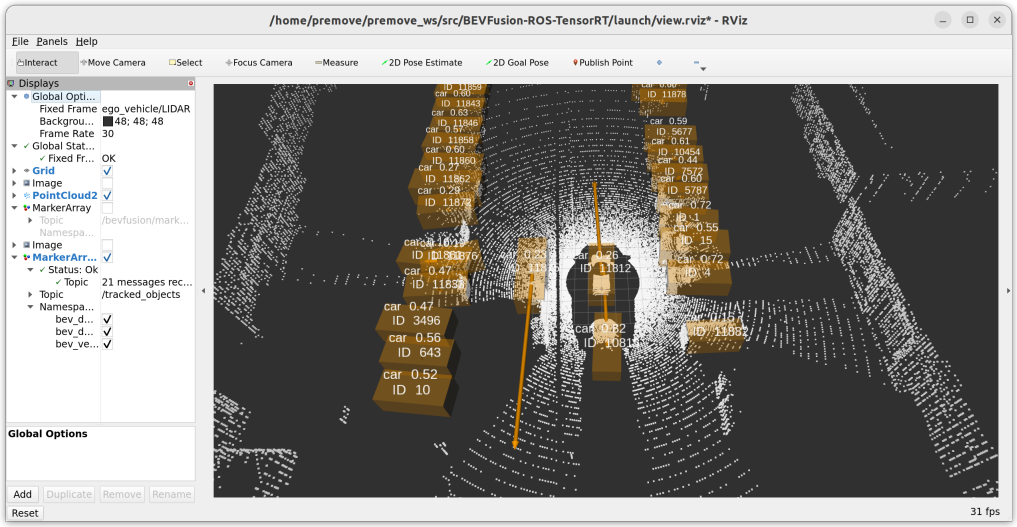

Figure 4: Visualización en RViz de la nube de puntos del LiDAR junto con las cajas, identificador de seguimiento y flechas de predicción de trayectorias generadas por `bevfusion_tracker`.