XXXVIII Jornadas de Automática
https://doi.org/10.17979/spudc.9788497497749.0806

SIMULACIÓN DE VEHÍCULOS AUTÓNOMOS USANDO V-REP BAJO ROS

C. Otero, E. Paz, R. Sanz y J. López
Dep. Ing. de Sistemas y Automática, Universidad de Vigo, {omcandido, epaz, rsanz,joaquin}@uvigo.es

R. Barea, E. Romera, E. Molinos, R. Arroyo, L.M. Bergasa, E. López
Dep. Electrónica, Universidad de Alcalá, {rafael.barea, luism.bergasa,elenalopezg}@uah.es,
{eduardo.romera, eduardo.molinos, roberto.arroyo}@edu.uah.es

Resumen

En este artículo se presentan las principales características del entorno de simulación que se está utilizando para el desarrollo de diferentes algoritmos de conducción autónoma. Estos desarrollos forman parte de un proyecto de conducción autónoma de vehículo en el marco del Plan Nacional de Investigación denominado SmartElderlyCar y desarrollado por la Universidad de Alcalá (UAH) y la Universidad de Vigo (UVIGO). Se ha realizado de forma exitosa la simulación de un vehículo comercial en V-REP controlado mediante nodos desarrollados bajo el sistema ROS en el campus externo de la UAH y se ha logrado conducir por sus carriles siguiendo la línea central mediante un algoritmo de seguimiento de trayectoria.

Palabras Clave: Simulación, conducción autónoma, V-REP, ROS.

1 INTRODUCCIÓN

La conducción autónoma es indiscutiblemente una de las tecnologías que mayor auge ha tenido en los últimos años. El reto al que hay que enfrentarse es el desarrollo de un sistema que pueda ser implantado en un vehículo con el objetivo de conducirlo de manera totalmente autónoma. Es en este ámbito donde se desarrolla el proyecto de investigación de los grupos que presentan este trabajo, orientado a la investigación en tecnologías que posibiliten un sistema de conducción inteligente para personas mayores en entornos fundamentalmente urbanos. Para tal fin, estamos utilizando el sistema operativo de código abierto ROS [13], que ha ido creciendo estos años gracias a la contribución de distintos profesionales y aficionados de la robótica. ROS proporciona un sistema de comunicación entre los diferentes programas de nuestro sistema. Los trabajos de investigación sobre navegación autónoma requieren la creación de un entorno de simulación que permita evaluar exhaustivamente los algoritmos desarrollados antes de su validación en condiciones reales. El primer paso en nuestra labor está siendo el desarrollo de un entorno simulado en el que se puedan probar nuestros algoritmos antes de implementarlos en el vehículo real.

Este artículo presenta la etapa de simulación de nuestro proyecto, explicando la justificación para la elección del entorno V-REP [15], las distintas decisiones que se han adoptado y mostrando los avances alcanzados.

El artículo está organizado de la siguiente manera. En la sección 2 se explica la decisión de utilizar V-REP frente a otros simuladores similares, haciendo una breve presentación del mismo y de sus ventajas. La sección 3 detalla la etapa de simulación propiamente dicha. Se explica cómo se obtiene el mapa del entorno, cómo se diseña y controla el vehículo junto con sus sensores y se indica la manera en que se comunica con ROS. En la sección 4 se explica la arquitectura software, que hemos bautizado como *SmartCar*, que consiste de una serie de capas de distinto nivel de abstracción. Aquí se da una pincelada de cada capa, explicando con un poco más de detenimiento los distintos sistemas que intervienen en la capa de control (mapeado, planificación, percepción, etc.). La sección 5 muestra las conclusiones y recoge algunas de las líneas de desarrollo futuras en las que ya se está trabajando.

2 EL SIMULADOR V-REP Y SU INTEGRACIÓN EN ROS

Existen diferentes criterios para la elección de un simulador de vehículos [5]. Aunque generalmente prevalecen criterios técnicos, muchas veces las preferencias personales afectan a la toma de la mejor decisión. En nuestro caso, hemos considerado dos de los simuladores en 3D más empleados en el ámbito de la robótica: V-REP y Gazebo, por su facilidad de integración en el entorno ROS.

Gazebo es un simulador en 3D bastante popular entre los investigadores en robótica, por lo que se pueden encontrar muchos modelos de robots, actuadores y sensores. Es un simulador de código abierto, y tiene una interfaz con ROS nativa y, además, tiene una comunidad de usuarios grande y activa.

Por otro lado, V-REP es un simulador en 3D con un entorno de desarrollo integrado, que se basa en una arquitectura de control distribuido [11]. Esto hace que V-REP sea muy versátil e ideal para aplicaciones multi-robot. V-REP se utiliza para el desarrollo rápido de algoritmos, simulaciones de automatización de procesos de fabricación, prototipado rápido y verificación, educación relacionada con robótica, entre otros.

Para la elección del simulador nos hemos basado en el rendimiento de ambos en un entorno complejo [8]. El rendimiento de un simulador de robot y su precisión son típicamente el resultado de un balance entre el número de robots y la velocidad de la simulación. Se ha tenido en cuenta la facilidad de integración de mapas de entorno y objetos estáticos y dinámicos en el escenario.

2.1 PRICIPALES CARACTERÍSTICAS DE V-REP

V-REP es un software de simulación multiplataforma desarrollado por Coppelia Robotics GmbH. Permite la personalización completa de la simulación mediante diferentes enfoques (complementos, plugins, scripts, etc.) y soporta cuatro motores dinámicos/físicos: Bullet, ODE, Newton y Vortex. Es posible crear modelos personalizados utilizando una amplia variedad de actuadores y sensores incorporados, articulaciones, formas y mallas, scripts, etc. [16].

V-REP está en una etapa de desarrollo avanzada y continua y tiene un foro de soporte muy activo. En cuanto a las condiciones de licencia, está disponible para investigadores y entusiastas de la robótica, una versión gratuita totalmente funcional de V-REP.

2.2 VENTAJAS POTENCIALES DE V-REP

Inicialmente se consideró la posibilidad de utilizar Gazebo, ya que soporta de forma nativa la comunicación con ROS. Sin embargo, V-REP fue finalmente elegido por su facilidad para crear nuevos modelos o modificar los ya existentes. Además, se pueden hacer pequeños cambios de programación utilizando *scripts* en LUA, dejando la carga computacional más pesada a complementos compilados escritos en C++.

V-REP es compatible con muchos formatos de objetos, lo que lo hace perfecto para cargar un entorno de simulación personalizado.

El rendimiento que se ha observado es muy similar entre V-REP y Gazebo, por lo que no ha sido un punto decisivo en la elección final de V-REP.

3 ETAPA DE SIMULACIÓN

Aunque V-REP no soporta nativamente la comunicación con nodos ROS, el *plugin RosInterface* permite esta funcionalidad, replicando de manera natural la API de la librería *roscpp*.

Tanto en simulación como con hardware real, se respetarán las convenciones para los diferentes sistemas de ejes [9][10]. Las diferentes convenciones adoptadas se irán señalando en cada uno de los siguientes apartados.

3.1 ENTORNO

El entorno en el que se probará el vehículo real es el campus externo de la UAH. Se ha modelado el entorno utilizando el servicio abierto *OpenStreetMap*. En este servicio los usuarios pueden mapear una zona y subirla a los servidores para uso público.

3.1.1 Mapa y edificios

V-REP permite importar fácilmente diversos formatos de mallas, por lo que las carreteras y los edificios se han cargado mediante archivos OBJ. Para importar los edificios se ha descargado el archivo OSM correspondiente al campus de la UAH, directamente de la web de *OpenStreetMap*. Posteriormente, este archivo se ha convertido a formato OBJ con la herramienta de código abierto *OSM2World*. Una vez importado este archivo nos quedamos únicamente con los edificios, eliminando el resto de los elementos.

A la hora de importar el archivo OBJ en V-REP, aparece un diálogo para seleccionar la escala y la orientación de la figura. La escala se escoge de tal manera que una unidad equivale a un metro y la orientación se establece con el vector Z apuntando hacia arriba. Si se examina el archivo OBJ con un editor de texto, se comprobará que las cuatro primeras líneas son una cabecera comentada que contiene esta información junto con la coordenada de origen (*origin coordinate*) del mapa local. Esta coordenada es el punto en coordenadas WGS84 (lat,lon,ele) que equivale en el simulador a la coordenada cartesiana (0,0,0).

Las carreteras se importan utilizando otro método. En la sección 4.2 se explica que los carriles se mapean mediante lanelets. Para importar los lanelets en V-REP se ha desarrollado un programa sencillo que convierte los carriles en mallas. Este software es objeto de mejora pudiendo añadir automáticamente otros elementos como señalización y marcado de la carretera en función de la riqueza del mapa de OSM.

Es importante tener en cuenta que el mapeado se realiza en coordenadas WGS84 y que el simulador trabaja en coordenadas cartesianas (UTM) relativas al punto origen anteriormente mencionado. Las transformaciones de un sistema a otro se realizan empleando las librerías implementadas en el paquete ROS *geodesy* y restando el desfase dado por la coordenada de origen del mapa, convirtiéndola previamente también a UTM.

3.1.2 Otros elementos

Para realizar pruebas de percepción con los sensores de visión, se han añadido manualmente elementos comunes encontrados habitualmente en el campus de la UAH (señales, árboles, farolas, etc.). Estos elementos han sido obtenidos de repositorios de archivos CAD libres y convertidos a formato OBJ.

3.2 VEHÍCULO

En la Figura 1 se muestra una captura del vehículo simulado tomando una rotonda. La parte visual del vehículo ha sido importada de un repositorio de CAD, dándole el aspecto de un vehículo comercial. La parte con propiedades dinámicas está formada por *cuboids* (paralelepípedos rectos) a los que se les da una masa para simular la inercia del vehículo. Además, está dotado de cuatro ruedas con amortiguación, motor propulsor, frenos, dirección y sensores.

El sistema de referencia del coche se sitúa centrado en el eje trasero, tomando el eje X apuntando al frente, el Y a la izquierda y el Z arriba. El vehículo es controlado por comandos de velocidad (lineal en X y angular en Z). La velocidad angular en Z se toma positiva según la regla de la mano derecha, por lo que los giros a la izquierda serán positivos. Estos comandos de velocidad son interpretados por el módulo base, obteniendo el ángulo de giro correspondiente, según se esté en modo automático o manual, y enviando al volante la consigna de giro correspondiente.

3.2.1 Control de dirección

Para simular el sistema de dirección del vehículo real con aceleraciones finitas, éste se ha simulado en V-REP como un sistema dinámico (con inercia y fricción) actuado por un motor servo-controlado en posición. Esto quiere decir que la posición del volante es controlada internamente por V-REP mediante un PID. El simulador recibe consignas de posición del volante de la misma manera que el vehículo real. La posición actual del volante determina el ángulo de dirección de una rueda directriz virtual central según el modelo bicicleta [6]. Mediante una serie de relaciones matemáticas, se

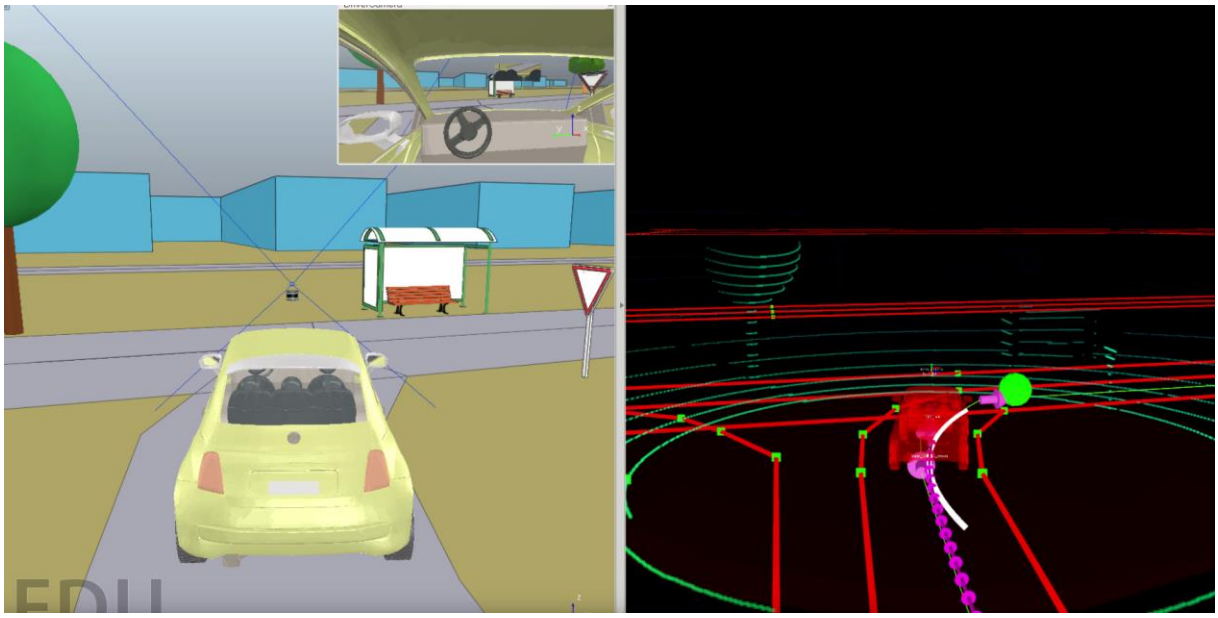

Figura 1: Ejemplo de seguimiento de la línea central de un carril utilizando el algoritmo *pure pursuit*. A la izquierda se muestra el vehículo simulado en V-REP y a la izquierda se visualizan las lecturas del LiDAR, los carriles mapeados, la curvatura instantánea (línea blanca) y el próximo punto objetivo (punto verde)

establece la orientación de las ruedas delanteras de tal manera que se cumpla la disposición Ackermann de las mismas para un centro instantáneo de rotación determinado.

3.2.2 Sensores

Los principales sensores de los que se ha dotado al vehículo son un láser LiDAR 3D, una cámara estéreo y un GPS. Se ha creado un modelo de cada uno de ellos que simula su comportamiento.

LiDAR: colocado sobre la carrocería del vehículo y en el centro del mismo. La implementación más intuitiva sería utilizar un único sensor de N haces (resolución 1xN) y hacerlo girar mediante una junta de revolución. Se ha probado esta implementación y se ha visto que implica leer cientos de veces los datos de visión en cada ciclo de simulación de 50 ms, que en V-REP implica un gran coste computacional, haciendo la simulación inviable. Por tanto, se parte del modelo *velodyne VPL-16* incluido por defecto en el repositorio de sensores de V-REP. Este sensor está modelado como 4 sensores de visión equiespaciados barriendo un ángulo horizontal de 360º y uno vertical de +/-15º con 16 haces diferentes, que es una forma mucho más eficiente de modelarlo.

La información del LiDAR debe ser dada en *PointCloud2*, que es el formato estándar más extendido actualmente para nubes de puntos. Debido a que crear este mensaje desde un *child script* en V-REP implicaría la creación de un gran número de tablas en LUA, lo cual es computacionalmente inviable, se ha modificado el *plugin* que trae por defecto el sensor de V-REP. Los tres cambios realizados son:

- Se publica directamente en ROS en formato *PointCloud2*.
- Se espera a que se hayan barrido 360º para publicar una vuelta completa.
- Las medidas del barrido completo (360º) se obtienen en varios pasos, cada uno referido a la posición en que se encuentra el vehículo. Como al final se transmiten todas juntas, es necesario ajustarlas para corregir el desfase debido al desplazamiento del vehículo.

Cámara estéreo: para modelar una cámara estéreo, colocada en la parte frontal del vehículo a la altura del espejo retrovisor y orientada hacia la carretera, se ha creado un modelo que consta de dos sensores de visión que leen únicamente información RGB. Ambos sensores están separados horizontalmente una distancia interocular fija *d* (línea base). A partir de la publicación en ROS de estas imágenes y de las matrices de calibración de los sensores, se pueden utilizar paquetes de procesamiento de imágenes que obtienen la información de profundidad y devuelvan una nube de puntos 3D.

Adicionalmente, se utiliza un sensor de visión situado en medio de los dos sensores anteriores que devuelve directamente una nube de puntos con gran precisión. Tiene sentido incorporar este sensor en el caso de que la cámara estéreo comercial que incorpore el vehículo real ya incluya internamente un módulo de cálculo de profundidad. Dependiendo de la solución final adoptada, tendrá sentido utilizar el sensor de profundidad o los dos de visión RGB.

Los mensajes de imágenes (de tipo *sensor_msg/Image*) deberían ser publicados con el eje X apuntando hacia la derecha, el Y hacia abajo y el Z hacia adentro en el plano de la imagen. Si se visualiza directamente en ROS una imagen tomada por un sensor de V-REP, se verá que la imagen está volteada horizontalmente. Esto puede solucionarse para las cámaras RGB utilizando filtros integrados en V-REP. Para la cámara de profundidad, ha sido necesario utilizar un *plugin* y crear una función que voltee la imagen.

La imagen de profundidad se publica con codificación 32FC1 (4 bytes) y las imágenes RGB con codificación rgb8 (3 bytes). Es importante tener esto en cuenta a la hora de cubrir los campos del mensaje que contiene la imagen [14].

3.2.3 Control mediante palanca de mando (*joystick*)

Para hacer algunas pruebas, es importante poder mover manualmente el coche. Para ello se ha utilizado un *joystick* y se ha creado un nodo que lee la posición de los ejes de interés (velocidad y dirección), y envía la velocidad y el ángulo deseado de giro de volante. Por simplicidad, se utiliza el campo de velocidad angular en Z del mensaje *twist_vel* para enviar el ángulo de giro del volante. En el simulador, según esté en modo automático o manual, se interpreta el campo *twist.angular.z* como velocidad angular, o como ángulo de giro de volante si está en modo manual.

Se ha utilizado con éxito un mando de PS3 y un volante USB para conducir manualmente el vehículo. Para conectar los *joysticks* con ROS, se instala el paquete *joystick_drivers*. Tras haber configurado la ruta del dispositivo, el nodo *joy_node* publicar el *topic* ROS *joy* con el valor de todos los ejes y botones del dispositivo. Atendiendo al índice de los ejes de interés y de los valores límite de cada uno, se crea un nodo que lee el *topic joy* y publica un comando de velocidad. Se recomienda parametrizar los índices, los desfases y las ganancias de cada eje

para hacer el nodo compatible con distintos dispositivos.

3.2.4 Módulo base

Este módulo ha sido implementado en un *child script* de V-REP y funciona como interfaz entre el vehículo y el sistema. Principalmente, recibe los comandos de velocidad y los convierte en orientación y velocidad de rotación de las ruedas delanteras. Para ello, el módulo interpreta las órdenes de velocidad, según se esté en modo manual o automático, y obtiene el ángulo de giro objetivo del volante y la velocidad lineal deseada del coche. A continuación, lee la posición instantánea del volante. Con ese valor y la velocidad lineal deseada del coche, se calcula la orientación y la velocidad angular de cada rueda delantera según la geometría Ackermann.

Además, en la etapa de simulación, también se utilizan la velocidad del coche y su posición exacta para publicar las transformaciones de posición y orientación entre diferentes elementos.

3.3 Otras funciones

En V-REP hay que actualizar manualmente el árbol de transformaciones entre elementos (el *topic* ROS */tf*). Para ello, primero se calcula manualmente la *pose* relativa entre dos elementos y se publica usando la función *simExtRosInterface_sendTransforms* del *plugin RosInterface*. Todas las transformaciones son actualizadas constantemente. En el caso de vehículos reales, las transformaciones fijas, como las que existen entre el vehículo y los sensores, se suelen publicar utilizando las *static transforms* de ROS.

De forma similar, podemos crear un mensaje del tipo *nav_msgs/odometry*. Esto es especialmente útil en robots móviles si se quiere utilizar el paquete *navigation* de ROS, ya que es una información imprescindible para su funcionamiento. Este mensaje también es útil para leer la posición exacta y la velocidad del vehículo, pero también se puede publicar otros mensajes individuales con esta información.

A menudo, es deseable que los algoritmos de control trabajen en tiempo de simulación. V-REP no publica por defecto el tiempo de simulación, pero la solución es inmediata utilizando el *plugin RosInterface*. Dado que V-REP no asegura en qué orden se va a ejecutar cada *child script* y se quiere que el *topic /clock* se actualice antes de cada ciclo de simulación, se decide modificar el *main script* de la escena para que al inicio de cada ciclo de simulación se publique el *topic* ROS */clock* con los segundos de simulación transcurridos. Es importante poner el parámetro *use_sim_time* a *true* antes de ejecutar V-REP y cualquier nodo que utilice el tiempo de simulación, de otro modo tomará el w*all time* de ROS.

4 ARQUITECTURA SMARTCAR

El sistema desarrollado, que hemos denominado *SmartCar*, se divide en capas, tal y como se muestra en la Figura 2. Las capas se organizan de tal manera que cada capa depende de la anterior, siendo la superior la más abstracta.

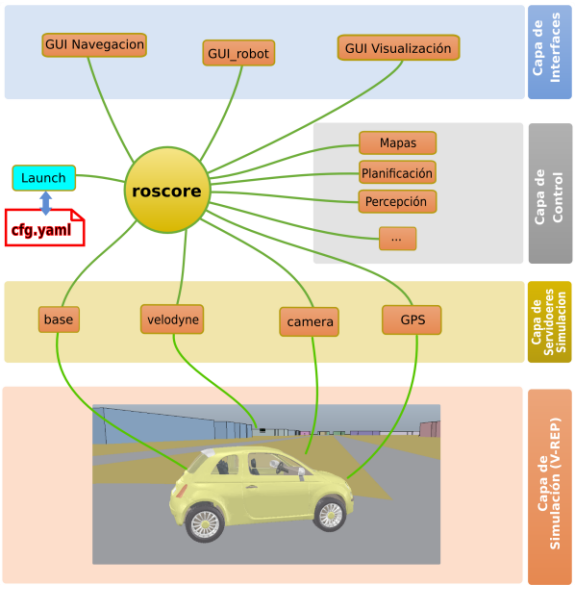

Figura 2: Estructura del sistema *SmartCar* en la etapa de simulación

4.1 CAPAS

Capa de interfaz: es la capa de nivel más alto. Incluye algunas interfaces de usuario para enviar objetivos de destino al vehículo, teleoperarlo y visualizar lecturas como nubes de puntos, posición, velocidad, etc.

Capa de control: es la capa que proporciona inteligencia al vehículo. Las funcionalidades que pertenecen a esta capa son la gestión de mapas, la gestión de comportamientos, la percepción del entorno y la planificación. Estas funcionalidades son descritas con más detalle en los restantes apartados de este artículo.

Capa de servidor hardware/simulación dependiendo del tipo de sistema, real o simulado, se utiliza el término hardware o **simulación**, respectivamente. Esta capa contiene todos los controladores que manejan la capa de hardware/simulación. Los nodos principales son ***base*** (lee objetivos de velocidad y publica los comandos para conducir el coche), ***velodyne*** (lee las medidas de láser y publica una revolución completa), ***camera***

(empareja las lecturas de imagen y las publica en el formato correcto) y ***GPS*** (lee las medidas GPS y las publica en formato estándar).

Capa hardware/simulación: esta es la capa de nivel más bajo. Se trata del vehículo real, en el caso de la capa de hardware, o bien del modelo simulado, en el caso de la capa de simulación.

4.2 GESTIÓN DE MAPAS

Las carreteras han sido mapeadas utilizando *lanelets* [17]. Mediante la herramienta abierta JOSM, se ha accedido a un mapa satélite del campus de la UAH y se han delimitado manualmente los carriles, incluyendo información regulatoria de tráfico, como puntos de parada, intersecciones, etc.

Este mapa topológico que hemos enriquecido para poder conducir con seguridad por las carreteras, es procesado por el *map manager*, que se encarga de planificar la ruta y enviársela al planificador de trayectorias.

4.3 PERCEPCIÓN DEL ENTORNO

El organigrama de funcionamiento del sistema de percepción se muestra en la Figura 3. El sistema de percepción percibe el entorno y obtiene la información correspondiente al LiDAR 3D y la cámara.

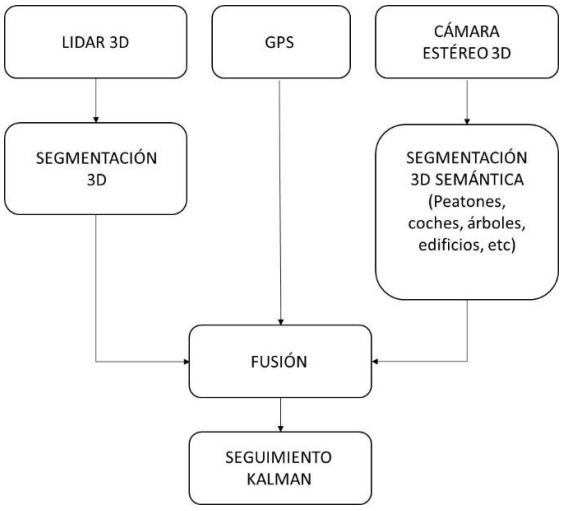

Figura 3: Organigrama del sistema de percepción

A partir de la imagen RGB de la cámara de la derecha se segmentan los diferentes objetos que componen la escena usando un algoritmo basado en color. En la versión sobre el dispositivo real se usará la *Red Neuronal Computacional* (CNN) *ERNet* desarrollada por el grupo de la UAH [12]. Con esta información junto con la nube de puntos obtenida de imagen de profundidad, se obtiene la segmentación semántica 3D de los objetos de la escena. Por otro lado, a partir de la nube de puntos obtenida del LiDAR 3D se realiza también una segmentación 3D de los objetos usando el método referenciado en [1].

Finalmente se hace una fusión de los objetos detectados por visión y por el LiDAR basada en la distancia Euclídea de sus centroides y el solape entre los paralelepípedos que circunscriben las nubes de puntos de cada objeto, tal como se muestra en la Figura 4. De esta forma se obtiene la posición 3D, el volumen y la clase cada uno de los objetos detectados. Para cada uno de ellos se realiza un seguimiento basado en filtro de Kalman [7], con la intención de obtener la predicción de su posición.

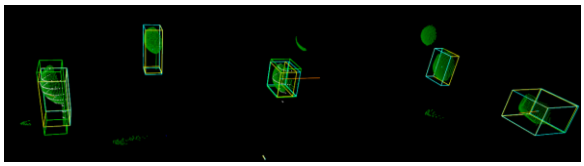

Figura 4: Objetos identificados e inscritos en paralelepípedos para mejor visualización

En la Figura 5 se muestran la carretera y los obstáculos detectados en el *RViz* de ROS, con respecto a la posición del vehículo, representados mediante esferas amarillas. Añadiendo la posición GPS del vehículo se puede referenciar el mapa de obstáculos al mapa global del entorno.

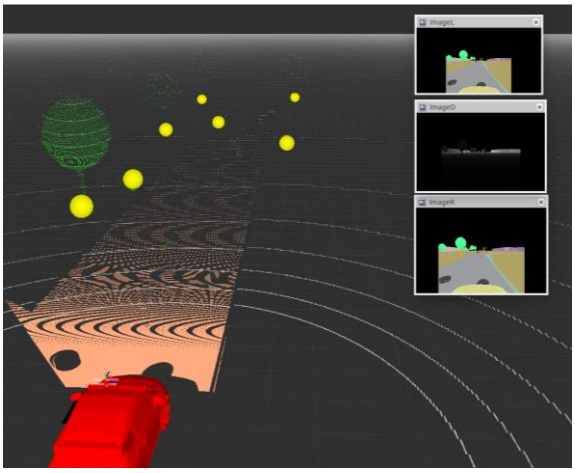

Figura 5: Carretera y obstáculos detectados tras segmentar la imagen

4.4 PLANIFICACIÓN

La planificación de un camino factible es uno de los primeros pasos para lograr una conducción totalmente autónoma. Se ha dividido esta tarea en dos etapas diferentes, una a nivel global y otra a nivel local.

4.4.1 Planificación de rutas

En entornos abiertos, el primer paso para planificar una ruta factible es generar un plan global utilizando una rejilla de ocupación. En el caso de conducción autónoma, se dispone de un mapa detallado de las carreteras, por lo que las restricciones de movimiento (velocidad, puntos de parada, carriles, etc.) están bien delimitadas. Por esta razón, en lugar de un planificador global tradicional, empleamos un planificador de rutas. Este planificador de rutas recibe dos posiciones GPS, inicio y meta, y recupera una lista de nodos del mapa topológico que se debe recorrer en orden para seguir el plan.

4.4.2 Planificación de trayectorias

Una vez que se dispone de la lista de nodos a través de los cuales debe conducir el vehículo, debe ser trazada una trayectoria factible. El enfoque principal es generar una serie de puntos de paso equidistantes en la línea central del carril (*centerline*), que es generalmente la zona de conducción más deseable.
Después de obtener la lista de puntos de paso objetivo, usamos el algoritmo *pure pursuit* [3] para seguir la trayectoria discretizada. Este algoritmo publica en cada ciclo de control un comando de velocidad (tangencial y angular) que describe un radio constante en cada instante infinitesimal. Este radio hace que el robot se dirija hacia el siguiente punto objetivo. El *lookahead* (distancia que establece el punto que "persigue" el vehículo) se ajusta de tal manera que disminuye a baja velocidad y con grandes curvaturas del carril, al igual que haría un conductor. Esta estrategia permite que el algoritmo adapte la distancia objetivo para seguir correctamente la trayectoria.

4.5 GESTIÓN DE COMPORTAMIENTOS

El planificador de trayectorias se encarga de obtener la secuencia de *lanelets* entre dos puntos. No obstante, estos *lanelets* pueden ser de distinto tipo según formen parte de una carretera sin cruces, un cruce de dos vías con preferencia, un cruce con semáforo, etc. En la solución que aquí se propone se define un comportamiento distinto para cada uno de los tipos de *lanelets*. La entrada a estos comportamientos son eventos producidos por una serie de monitores u observadores similares a los presentados en [2].

La secuencia de acciones y eventos que forman parte de un comportamiento se implementa mediante redes de Petri usando una herramienta de la Universidad de Vigo llamada *RoboGraph* [4]. Para ello, cada comportamiento es definido mediante una o varias Redes de Petri usando la herramienta *RoboGraph Editor* mientras que su ejecución la lleva a cabo *RoboGraph Dispatch*.

Dispatch se encarga de la ejecución de las distintas acciones de la capa funcional (acciones básicas), ejecutiva (otras redes de Petri) e interfaces, así como de la sincronización con los eventos producidos. La interacción con los módulos de la arquitectura se realiza mediante la publicación y suscripción de mensajes. De esta manera, los problemas que se puedan producir en un módulo, como un bloqueo, no provocará el bloqueo del *Dispatch*. Incluso mediante un simple mecanismo se puede detectar el error y recuperarse de un fallo en determinadas situaciones.

Dispatch forma parte de la herramienta de gestión de tareas *RoboGraph* (www.webs.uvigo.es/vigobot). *RoboGraph* utiliza redes de Petri jerarquizadas e interpretadas para coordinar la actividad de dichos módulos. Las tareas se definen utilizando un editor de redes de Petri interpretadas y se guardan en un fichero XML. *Dispatch* es el programa de ejecución que se encarga de cargar estos ficheros y ejecutar las redes de Petri cada vez que un nodo del sistema lo solicite mediante el envío del correspondiente mensaje. Por otro lado, se dispone de un programa para monitorizar la evolución de las redes de Petri en ejecución, muy útil para depuración y trazado.

5 CONCLUSIONES Y LÍNEAS FUTURAS

Se ha llevado a cabo de forma exitosa la simulación de un vehículo comercial controlado mediante nodos desarrollados bajo el sistema ROS. Se ha mapeado el campus externo de la UAH y se ha logrado conducir por sus carriles siguiendo la línea central mediante el algoritmo de seguimiento de trayectoria *pure pursuit*. En la Figura 6 se muestra el trazado del vehículo tras realizar un cambio de sentido en una rotonda.

Se sigue trabajando en técnicas que aporten al vehículo información sobre los obstáculos estáticos y dinámicos (anticipándose a su futura posición).

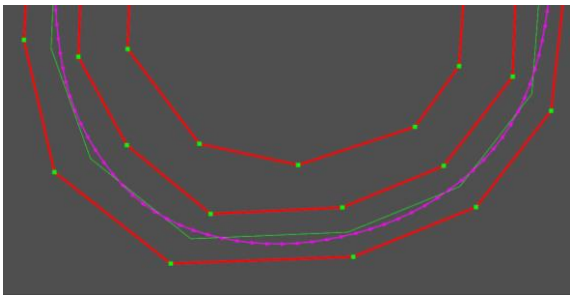

Figura 6: Trazado del coche (fucsia) después de conducir a través de una rotonda. El coche sigue la línea central (verde) del carril (rojo) derecho.

Se está trabajando en un módulo de navegación reactiva que pueda asegurarse de mantener el coche dentro del carril y evitar los posibles obstáculos dinámicos que se encuentre.

Agradecimientos

La investigación presentada en este artículo ha sido financiada por los siguientes proyectos de investigación del Programa Estatal de I+D+i Orientada a los Retos de la Sociedad del Ministerio de Economía y Competitividad: Vehículo inteligente para personas mayores (TRA2015-70501-C2-1-R (MINECO/FEDER)) y *Smartelderlycar*: control y planificación de rutas (TRA2015-70501-C2-2-R (MINECO/FEDER)).

Referencias

[1] Aldoma, A., Marton, Z-C., Tombari, F., Wohlkinger, W., Potthast, C., Zeisl, B., Rusu, R.B., Gedikli, S., and Vincze, M., (2012) "Point Cloud Library: Three-Dimensional Object Recognition and 6 DOF Pose Estimation", *IEEE Robotics & Automation Magazine* pp. 80-91.

[2] Beeson, P., O'Quin, J., Gillan, B., Nimmagadda, T., Ristroph, M., Li, D., and Stone, P., (2008) "Multiagent interactions in urban driving", *Journal of Physical Agents*. Vol. 2(1), pp. 15-29.

[3] Coulter, R., (1992) Implementation of the pure pursuit path-tracking algorithm, Carnegie Mellon University, the Robotics Institute, Pittsburgh, Pa.

[4] Fernández, J. L., Sanz, R., Paz, E., and Alonso, C., (2008) "Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph", in *IEEE International Conference on Robotics and Automation (ICRA 2008),* Vol. I, pp. 1372-1377.

[**5**] Ivaldi, S., Padois, V., and Nori, F., (2014) Tools for dynamics simulation of robots: a survey based on user feedback. *arXiv preprint arXiv:1402.7050*.

[6] Jazar, R., (2014) Vehicle dynamics (2nd ed.), New York, Springer New York.

[7] Kalman filter Opencv tutorial, http://docs.opencv.org/trunk/dd/d6a/classcv_1_1KalmanFilter.html

[8] Nogueira, L., (2014) "Comparative Analysis between Gazebo and V-REP Robotic Simulators", *Seminario Interno de Cognicao Artificial - SICA 2014*, pp. 5.

[9] REP 103 -- Standard Units of Measure and Coordinate Conventions, (2017) Ros.org, retrieved 15 June 2017, from http://www.ros.org/reps/rep-0103.html.

[10] REP 105 -- Coordinate Frames for Mobile Platforms, (2017) Ros.org, retrieved 15 June 2017, from http://www.ros.org/reps/rep-0105.html.

[11] Rohmer, E., Singh, S. P., and Freese, M., (2013) "V-REP: A versatile and scalable robot simulation framework" in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* pp. 1321-1326.

[12] Romera, E., Álvarez, J.Mª. Bergasa, L.M., Arroyo, R., (2017) "Efficient ConvNet for Real-time Semantic Segmentation", in *Intelligent Vehicles Symposium Proceedings,* pp. 1789-1794.

[13] ROS Web page, http://www.ros.org/

[14] sensor_msgs/Image Documentation. (2017) Docs.ros.org, retrieved 15 June 2017, from http://docs.ros.org/indigo/api/sensor_msgs/html/msg/Image.html.

[15] V-REP simulator Web page, http://www.coppeliarobotics.com/

[16] V-REP User Manual (2017), Coppelia-robotics.com, retrieved 15 June 2017, from http://www.coppeliarobotics.com/helpFiles/.

[17] Ziegler, J., Bender, P., Dang, T., and Stiller, C., (2014) "Trajectory planning for Bertha—A local, continuous method", in *IEEE Intelligent Vehicles Symposium Proceedings*, pp. 450-457.