Universidad
Carlos III de Madrid

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO DE FIN DE GRADO

INTEGRACIÓN DEL SIMULADOR GAZEBO CON EL VISUALIZADOR DE DATOS RVIZ DENTRO DE ROS

Autor: Juan Marín Esponera

Tutor: David Álvarez Sánchez

Septiembre 2014

Contenido

ÍNDICE DE FIGURAS

1.- INTRODUCCIÓN.

La robótica es la rama de la tecnología que, mediante el uso de la informática, se encarga del diseño, construcción, operación, manufactura y aplicación de los robots.

El término “robot” proviene de la palabra checa “robota”, que significa trabajos forzados, ésta proviene de la traducción al inglés de la obra teatral R.U.R. (Robots Universales Rossum), escrita por Karel Čapek en 1920 [1].

A pesar de los avances realizados, y de los más de 25 años que lleva estudiándose, a la hora de hablar de robótica tenemos que tener en cuenta que es un campo del que aún se desconocen una gran cantidad de cosas y que se encuentra en pleno desarrollo. En la actualidad, los robots más utilizados son los comerciales e industriales, ya que realizan tareas de forma más exacta o más barata que los humanos. También se les suele utilizar para realizar trabajos que se consideran demasiado peligrosos o tediosos para los humanos . Algunas de las aplicaciones actuales de los robots son en plantas industriales, para manufactura, montaje y embalaje; en exploraciones en el espacio, cirugía, armamento, etc.

A la hora de clasificarlos no es nada sencillo, cada autor atiende a unos criterios para hacerlo [2]. Sin embargo, la forma más común es la siguiente:

- De 1ª Generación (Manipuladores). Este tipo de robots cuentan con un sistema de control en lazo abierto, es decir, sin retroalimentación alguna. Por tanto, no reaccionan a los estímulos de su entorno, puesto que adquieren muy poca o ninguna información de éste. La mayoría de ellos son utilizados en las líneas de producción para realizar labores repetitivas o que requieran de alguna habilidad especial, como el que se puede observar en la Figura 1.1.

Figura 1.1: Robot manipulador.

- De 2ª Generación (Robots de aprendizaje). Repiten una secuencia de movimientos que ha sido ejecutada previamente por un operador humano. Éste enseña los movimientos al robot a través de un dispositivo mecánico, con el que realiza los movimientos requeridos mientras el robot le sigue y memoriza el recorrido.

- De 3ª Generación (Robots con control sensorizado). Esta generación cuenta con un controlador (computadora) que, en función de los datos obtenidos por medio de los sensores, ejecuta las órdenes de un programa.

- De 4ª Generación (Robots inteligentes). Son similares a los anteriores, pero poseen sensores mucho más sofisticados que envían información a la computadora de control y cuentan con estrategias complejas de control para actuar a los estímulos del entorno. Esto permite una toma inteligente de decisiones y el control del proceso en tiempo real. Un ejemplo de este tipo es el robot ASIMO (*Advanced Step in Innovative Mobility*), el cual es capaz de interaccionar con un humano en diversas tareas, como por ejemplo entregando una bandeja a alguien, como puede observarse en la Figura 1.2.

Figura 1.2: Robot inteligente ASIMO.

1.1.- MOTIVACIÓN.

A la hora de desarrollar un nuevo robot, o de evolucionar uno ya existente, es necesario adquirir un conocimiento detallado de los fenómenos físicos que limitan el rendimiento de éste. Este conocimiento detallado no es posible adquirirlo única y exclusivamente mediante los ensayos, en gran parte debido a las limitaciones de tiempo y económicas con las que se cuenta. Además de esto, la simulación también aporta confianza a la hora de realizar el desarrollo y futuros ensayos, puesto que nos da una idea a priori de cómo reaccionará nuestro robot ante situaciones específicas.

Tan importante como puede ser la parte de realizar una buena simulación, y de realizarla en un entorno lo más fiel posible a la realidad, lo es el obtener los datos de los sensores de los que el robot dispone. Muchas veces estas dos necesidades vienen unidas, puesto que no sirve de nada realizar una simulación en un entorno si no somos capaces de averiguar qué es lo que nuestro robot está detectando al encontrarse en esas situaciones determinadas.

La integración entre el robot y los distintos sistemas para la simulación y obtención de datos aparece como una labor primordial a la hora de llevar a cabo el desarrollo de un robot. En muchas ocasiones, dichos sistemas están basados en código libre, es decir, pertenecen a varios colectivos y la labor del desarrollador consistirá en integrar los diferentes componentes y completar el desarrollo con su aportación al sistema.

1.2.- OBJETIVOS.

Como hemos comentado anteriormente, tan importante como es realizar una buena simulación a la hora de diseñar un robot, lo es el realizar una correcta toma de datos de ésta. Para ello nuestro trabajo consistirá en realizar la integración del simulador Gazebo con el visualizador Rviz. Para ello, realizaremos la simulación de un robot en un mundo virtual y seremos capaces de leer los valores de algunos sensores a través de Rviz.

Esta integración será bajo el Sistema Operativo Robótico (ROS, *Robot Operating System*), el cual nos proporcionará una serie de paquetes y servicios que nos ayudarán a la hora de transmitir la información entre las dos aplicaciones. Por tanto, nuestro primer objetivo será el de realizar una aproximación a éste y tratar de comprenderlo de la mejor manera posible.

Una vez realizado esto, nuestra labor será la de hacer una primera aproximación al visualizador Rviz y el simulador Gazebo por separado para, posteriormente, realizar la integración de éstos. Para ello realizaremos la simulación del Robot Personal 2 (PR2) en un entorno determinado, una gasolinera. Lo siguiente será conseguir visualizar en Rviz los valores que nos proporcionan los sensores del robot simulado en Gazebo. Además de esto, también visualizaremos en Rviz el robot y los movimientos que realice éste en el simulador.

2.- TECNOLOGÍAS UTILIZADAS.

A continuación, haremos una pequeña descripción de cada una las tecnologías utilizadas para llevar a cabo los objetivos de nuestro trabajo.

2.1.- ROS.

Robot Operating System [3] es un *framework* para el desarrollo de software para robots, provee de librerías y herramientas a los desarrolladores de software para crear sus propias aplicaciones. Fue creado en el año 2007 bajo el nombre de Switchyard (Subestación), por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR) [4]. Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado.

El motivo por el que se creó ROS fue la dificultad que tiene crear un software realmente robusto en el mundo de la robótica. Desde el punto de vista de un robot, un problema que puede parecer trivial para los seres humanos puede llegar a convertirse en un verdadero reto a la hora de solventarlo. Las variaciones entre las instancias de las tareas creadas y los entornos pueden ser enormes, tanto que ningún individuo, laboratorio o institución puede aspirar a hacerlo por su cuenta. Además de esto, el gran tamaño de código requerido puede llegar a ser desalentador para el desarrollador, ya que debe contener una gran cantidad de software inicial a nivel de controlador comenzando por la percepción, el razonamiento abstracto, etc.

Como resultado, ROS fue construido desde cero para fomentar el desarrollo de software de robótica de colaboración. Es una manera de cooperar entre laboratorios, donde cada uno puede tener expertos en determinados campos y otro laboratorio beneficiarse de ello, y a su vez proporcionar algo que ayude a este anterior. Por ejemplo, un laboratorio podría tener expertos en cartografía de los ambientes interiores y podría construir un sistema de clase mundial para la producción de mapas. Otro grupo podría tener expertos en el uso de mapas para navegar y, además, haber descubierto un algoritmo bastante robusto para la visualización.

A pesar de su nombre, ROS no es un sistema operativo propiamente dicho, de hecho funciona sobre Linux. Lo que hace realmente es proveer los servicios estándar de un sistema operativo, tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes, dejando las labores de procesamiento, manejo de memoria, gestión de interfaces gráficas, etc. a Linux. En la Figura 2.1 puede observarse un esquema de la separación entre los servicios proporcionados por Linux y por ROS.

También debemos tener en cuenta que está basado en una arquitectura de grafos, donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros [5].

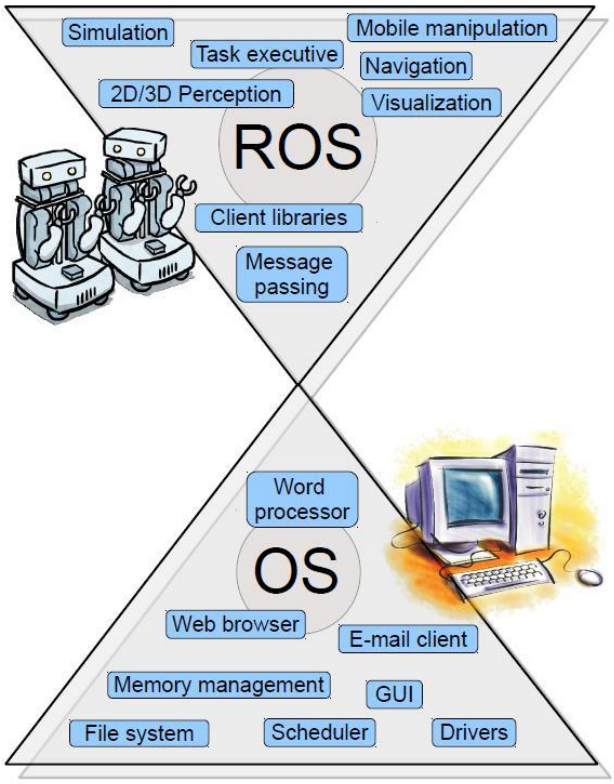

Figura 2.1: Relación ROS y sistema operativo.

La filosofía de ROS puede resumirse con los siguientes conceptos [6]:

- *Peer-to-peer*. Un sistema construido utilizando ROS se compone de una serie de procesos conectados en tiempo de ejecución a un número determinado de anfitriones, utilizando una topología *peer-to-peer*. Este tipo de topología requiere de un *master* que permita que los procesos se encuentren unos a otros. En la Figura 2.2 puede observarse un ejemplo de este tipo de conexión, en el que tenemos varios anfitriones.

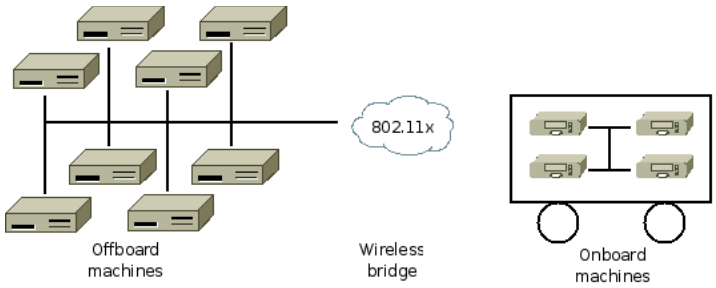

Figura 2.2: Conexión típica de ROS.

- Multilingüe. A la hora de escribir código cada persona, por determinados motivos, tiene unas preferencias por ciertos lenguajes de programación. Para facilitar la integración de éstos, ROS fue diseñado para poderse utilizarse independientemente del lenguaje de programación. En la actualidad es capaz de soportar tres muy distintos, como son C++, Python y Lisp, y cuenta con librerías experimentales en Java y Lua.

- Basado en herramientas. Cuenta con un diseño micronúcleo (*microkernel*), donde un gran número de primitivas son llamadas para crear y ejecutar los diversos componentes de ROS, dejando la gestión de memoria, sistema de archivos, etc. al sistema operativo.

- "Ligero". Todo el código está estructurado en pequeñas bibliotecas independientes, de tal manera que es mucho más fácil reutilizarlo en proyectos distintos al original para el que fueron creados.

- Gratuito y de código abierto: ROS es un software libre bajo licencia estándar BSD de tres cláusulas[1]. Ésta permite la reutilización de los productos comerciales y de código cerrado, entre otras cosas.

Desde su primera distribución, lanzada el 2 de Marzo de 2010 (ROS Box Turtle), han visto la luz 8 versiones en total. ROS Hydro Igloo, la última de ellas, comenzó a distribuirse en Julio de este año y la próxima, ROS Jade Turtle, está previsto que esté lista para Mayo de 2015.

En nuestro caso hemos optado por utilizar la versión de Septiembre de 2013, llamada ROS Hydro Medusa, por ser la más actualizada a fecha de inicio de este trabajo y contar con importantes mejoras en las herramientas que se pretenden integrar, Gazebo y Rviz. Cabe destacar que no todas las distribuciones son compatibles con cualquier versión de Linux. En nuestro caso, ROS Hydro Medusa cuenta con compatibilidad con Ubuntu 12.04 (Precise), 12.10 (Quantal) y 13.04 (Raring).

2.1.1.- RVIZ.

ROS dispone de una herramienta de visualización 3D, llamada Rviz [7], donde se pueden representar robots, nubes de puntos, datos de sensores, etc. que gracias a estar basada en una arquitectura de *plugins*, nos permite no sólo desarrollar nuestras herramientas, si no reutilizar otras ya existentes.

La forma en que funciona Rviz es simplemente leyendo e interpretando los diferentes datos contenidos en mensajes de ROS. Por tanto, es necesario tener un generador externo de estos mensajes, como puede ser un robot real o simulado. En la Figura 2.3 se puede ver un ejemplo de este funcionamiento, en el que el robot genera distintos mensajes con su posición y Rviz se encarga de interpretarlos y mostrarlos por pantalla.

[1] http://opensource.org/licenses/BSD-3-Clause

En nuestro caso, la labor que desempeñará será la de leer los mensajes proporcionados por el simulador Gazebo, con la posición de nuestro robot y las lecturas de sus sensores, y mostrarlos por pantalla.

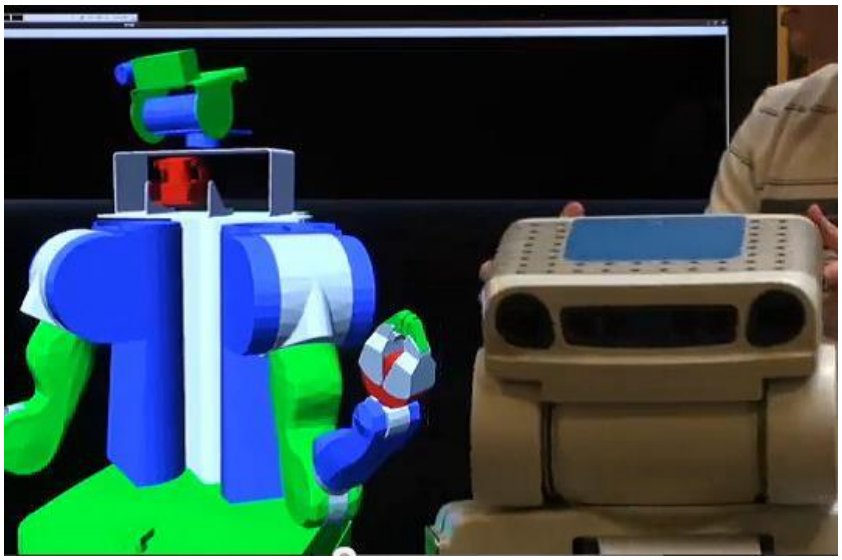

Figura 2.3: Ejemplo de aplicación de Rviz.

A la hora de poder generar un robot en particular, se debe editar un archivo URDF (Unified Robot Description Format) donde se especifiquen las dimensiones del robot, los movimientos de las articulaciones, parámetros físicos, etc. En cualquier caso, ROS cuenta con modelos previamente creados con los que el usuario puede trabajar.

2.2.- GAZEBO.

Gazebo [8] es un simulador 3D, cinemático, dinámico y multi-robot que permite realizar simulaciones de robots articulados en entornos complejos, interiores o exteriores, realistas y tridimensionales. Su desarrollo comenzó en otoño de 2002 en la Universidad del Sur de California a manos del Dr. Andrew Howard y su estudiante Nate Koenig. En la actualidad, y desde 2012, Open Source Robotics Foundation (OSRF) [9] se

encarga de administrar el proyecto, continuando el desarrollo con el apoyo de una comunidad diversificada y activa.

Las principales características con las que cuenta este simulador son las siguientes:

- Al igual que pasa con ROS, Gazebo es un software libre, pudiendo ser reconfigurado, ampliado y modificado.

- Es compatible con ROS y Player. Es posible ejecutarlo desde ROS, posibilitando el envío de información a través de los nodos de éste. Esto nos será de gran utilidad a la hora de poder visualizar los datos de la simulación a través de Rviz.

- Cuenta con una simulación realista de la física de los cuerpos rígidos. Los robots pueden interactuar con el mundo (pueden coger y empujar cosas, rodar y deslizarse por el suelo) y viceversa (les afecta la gravedad y pueden colisionar con obstáculos del mundo). Para este propósito se sirve del motor de física de código abierto ODE (Open Dynamics Engine), el cual cuenta con una serie de bibliotecas de alto rendimiento para la simulación de la cinemática y la dinámica de los sólidos rígidos.

- Capacidad de desarrollar y simular modelos de robots propios (URDF) e, incluso, cargarlos en tiempo de ejecución. Además, hay una gran cantidad de ellos con software de código libre para poder ser simulados en Gazebo.

- Permite la posibilidad de crear escenarios (mundos) de simulación personalizados, variando las características de los contactos con el suelo, los obstáculos y los valores de la gravedad en las tres dimensiones. También es posible variar las características de contacto de cada unión del robot individualmente.

- Contiene diversos *plugins* que nos brindan la oportunidad de añadir sensores al modelo del robot y simularlos, como pueden ser sensores de odometría (Sistema de posicionamiento global (GPS) y Unidad de Medición Inercial (IMU)), de fuerza, de contacto, láseres y cámaras estéreo.

Como se ha comentado en los objetivos, la labor que desempeñará Gazebo será la de simular un entorno, una gasolinera, y el Robot Personal (PR2) dentro de él, como puede observarse en la Figura 2.4. De tal manera que en los valores de los sensores seamos capaces de observar los distintos elementos de la gasolinera.

Figura 2.4: Simulación de PR2 en Gazebo.

2.3.- PR2.

Aunque a simple vista PR2 parezca un simple robot, como puede apreciarse en la Figura 2.5, su definición más exacta, y la oficial de Willow Garage, es “plataforma robótica de investigación y desarrollo” [10]. El concepto que se persigue con él es el de ofrecer a los investigadores los componentes modulares necesarios para construir robots, programarlos y encontrarles utilidad, olvidándose de las tareas tediosas que ya han sido superadas por otros ingenieros. Es decir, ofrecen la base sobre la que trabajar a futuros investigadores para desarrollar software de control, sin tener que preocuparse por el hardware.

Además, Willow Garage, para facilitar aún más la labor de los desarrolladores, ofrece todo el software del PR2 a través de ROS que, como se ha comentado anteriormente, cuenta con licencias de código abierto. De hecho a la hora de adquirir el robot es posible obtener importantes descuentos si se es capaz de acreditar haber aportado una "dedicación relevante a la comunidad del software libre". Todo esto fomenta, al igual que pasa con ROS, la colaboración entre investigadores y un desarrollo rápido del software.

Figura 2.5: PR2.

Técnicamente el PR2 se encuentra muy desarrollado, cuenta con una gran cantidad de grados de movimiento en sus brazos y pinzas, con una base con rodamientos que le permite hacer todo tipo de movimientos y con una cabeza que puede moverse sin limitaciones. Entre sus sensores cabe destacar la cámara Kinect 3D, que puede observarse en la Figura 2.6, y que en nuestro caso utilizaremos para obtener imágenes y nubes de puntos del entorno simulado.

El dispositivo Kinect cuenta con una cámara RGB, un sensor de profundidad y un micrófono *multi-array* bidireccional que, en conjunto, capturan imágenes y movimientos de los cuerpos en 3D, además de ofrecer reconocimiento facial y aceptar comandos de voz. El sensor de Kinect adquiere imágenes de video con un sensor

CMOS de colores a una frecuencia de 30 Hz, en colores RGB de 32-bits y resolución VGA de 640×480 pixeles. Para calcular distancias entre un cuerpo y el sensor, el sensor emite un haz láser infrarrojo que proyecta un patrón de puntos sobre los cuerpos (nube de puntos). Puede llegar a distinguir la profundidad de cada objeto con una resolución de 1 centímetro, y las estimaciones de la altura y anchura con una exactitud de aproximadamente 3 milímetros.

Figura 2.6: Cámara Kinect del PR2.

El PR2 también cuenta con un escáner láser de posición, mediante el cual realiza barridos periódicamente a su alrededor para poder determinar si hay objetos u obstáculos dentro de su radio de acción. Además de lo ya comentado cuenta con acelerómetros de tres ejes, sensores de presión y de calibración. Todo este hardware hace que el robot pueda hacer casi cualquier tarea que seamos capaces de enseñarle. A la hora de comunicarse y recibir instrucciones y software cuenta con conexión Wi-Fi, bluetooth y un *switch* Ethernet Gigabit, lo que le hace altamente accesible. Cuenta con una autonomía de aproximadamente dos horas en sus baterías, aunque el usuario no debe preocuparse por ello, ya que una de las primeras tareas que le fueron enseñadas es la de recargarse solo en los enchufes que encuentra a su alcance, como puede observarse en la Figura 2.7.

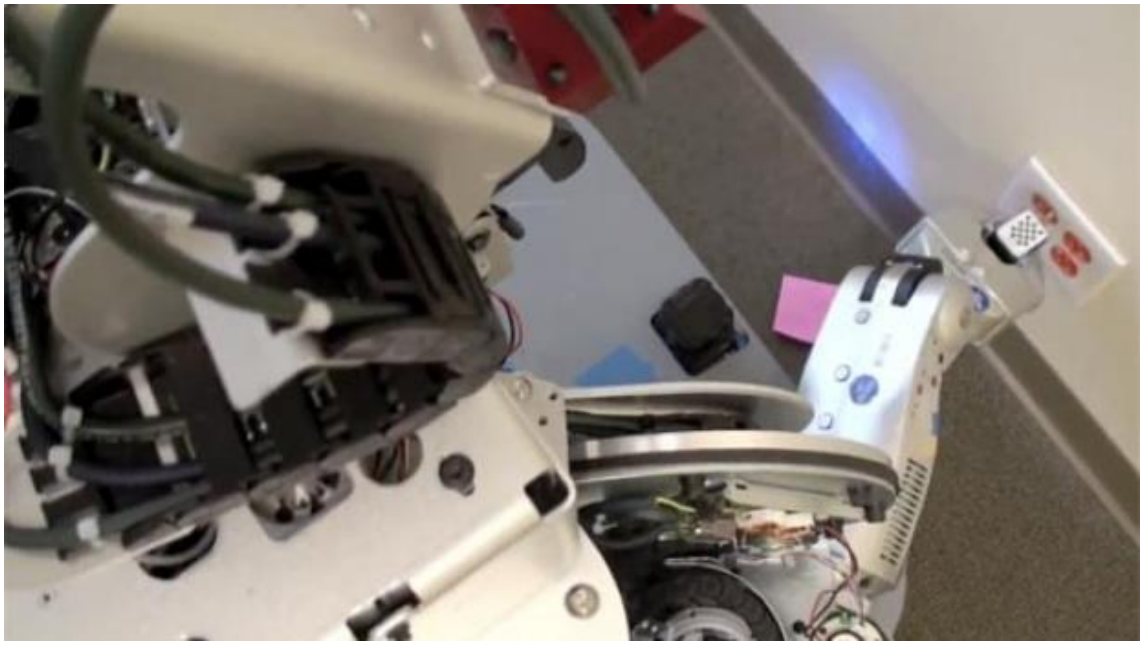

Figura 2.7: PR2 conectándose a la red eléctrica.

3.- EXPERIMENTACIÓN CON GAZEBO Y RVIZ.

A continuación, detallaremos todo el proceso seguido para llevar a cabo la integración de las dos aplicaciones, desde los conceptos básicos de funcionamiento de ROS, hasta la toma de datos desde el visualizador Rviz, pasando por la simulación en Gazebo.

3.1.- CONCEPTOS BÁSICOS DE ROS.

Antes de comenzar nuestro trabajo de simulación y toma de datos, deberemos comprender una serie de conceptos básicos del funcionamiento de ROS Hydro [11] , ya que, como se ha comentado anteriormente, es la versión que hemos utilizado por ser la más actualizada a fecha de comienzo de este trabajo.

La forma con la que cuenta para organizar el sistema de archivos se divide básicamente en tres niveles [5]:

1. Paquetes (*packages*). Se trata del nivel más bajo de organización de archivos. Puede contener toda clase de archivos: ejecutables (nodos), modelos (robots), tipos de mensajes y servicios, herramientas o librerías, como puede observarse en la Figura 3.1.

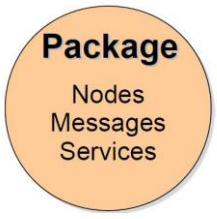

Figura 3.1: Paquetes.

2. Pilas (*stacks*). Son agrupaciones de paquetes, como puede observarse en la Figura 3.2, formando una librería de alto nivel. En cada pila se agrupan paquetes que son complementarios entre sí.

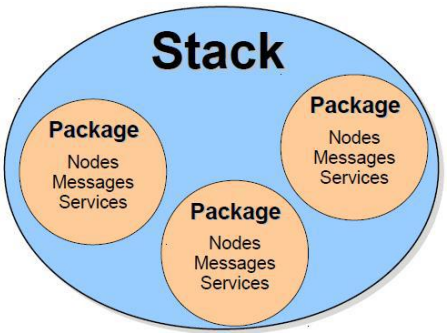

Figura 3.2: Pilas.

3. Repositorios. Son agrupaciones de pilas, como puede observarse en la Figura 3.3, para ser descargadas o instaladas. Funcionan de forma equivalente a los repositorios de Linux, es decir, pueden ser descargados directamente por línea de comandos o mediante el administrador de descargas.

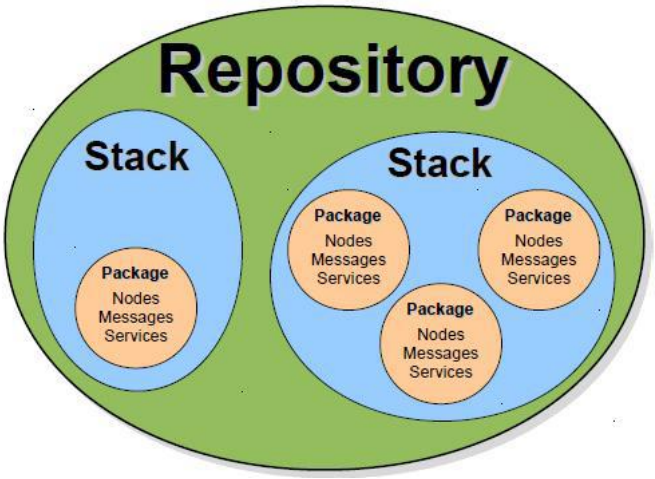

Figura 3.3: Repositorios.

Tanto los paquetes como las pilas, siempre poseen un archivo de información especificando su contenido, la función que tienen y los paquetes de los que dependen. En las pilas el archivo se denomina “stack.xml” y en los paquetes “manifest.xml”.

A la hora de instalar ROS todos los archivos que se crean lo hacen como restringidos, siendo imposible modificar sus paquetes propios y evitando así posibles daños en el sistema. Por este motivo, se crea, en general, una carpeta de trabajo (*workspace*) en la

que se ubican los paquetes desarrollados y que se añade al *path* de ROS para que éste acceda.

A la hora de crear un paquete, lo primero que deberemos tener en cuenta es la organización normal en carpetas de éstos y el tipo de archivos que nos podemos encontrar en cada una de ellas. Las principales que podemos encontrar son:

- Carpeta *bin*. Contiene los ejecutables del paquete, que se podrán lanzar como nodos.
- Carpeta *build.* Contiene los archivos de compilación del paquete. Esta carpeta se crea automáticamente siempre que compilamos el paquete por primera vez.
- Carpeta *src*. Contiene los códigos de los programas (.*cpp*) que al compilarlos crearán ejecutables que se almacenan en *bin*.
- Carpeta *include*. Contiene los archivos de cabeceras (.*h*) propios de los programas del paquete.
- Carpeta *lib*. Contiene las librerías (.*lib* y .*so*).
- Carpeta *launch.* Contiene los archivos de lanzamiento (.*launch*) de aplicaciones completas.
- Carpeta *yaml*. Contiene los archivos de lista de parámetros (.*yaml*).
- Carpeta *msg*. Contiene los tipos de mensajes (.*msg*) definidos en el paquete.
- Carpeta *msg_gen*. Contiene los archivos de cabeceras (.*h*) autogeneradas al compilar los tipos de mensajes definidos en el paquete.
- Carpeta *srv*. Contiene los tipos de mensajes (.*srv*) de los servicios definidos en el paquete.
- Carpeta *srv_gen*. Contiene los archivos de cabeceras (.*h*) autogeneradas al compilar los tipos de servicios definidos en el paquete.

Además de éstas, también es posible encontrar algunas carpetas distintas a las mencionadas. En el caso de paquetes para la simulación en Gazebo es muy común encontrarse con estas dos carpetas:

- Carpeta *worlds*. Contiene los archivos de lanzamiento (*.launch*) de los diferentes escenarios ("mundos") de simulación del paquete.

- Carpeta *robots*. Contiene los archivos de descripción de los robots (*.urdf* y *.xml*).

Junto con este sistema de carpetas, en todo paquete de ROS se encontrarán dos archivos, que son los siguientes:

- Archivo *CMakeLists.txt*. es una lista en la que se especifica al compilador que debe compilar de nuestro paquete: nodos, mensajes, servicios, librerías, etc.

- Archivo *manifest.xml*: contiene una descripción del paquete (función, autor, licencia, etc.) y una lista con los paquetes de los que depende.

3.1.1.- COMPILACIÓN.

Para compilar un paquete en ROS se utiliza CMake, una plataforma de código abierto para la generación, compilación y comprobación de paquetes de software. El sistema que usa es similar al GNU Build de Unix, en el que el proceso es controlado por ficheros de configuración. Su uso es bastante sencillo, basta con que el paquete de ROS a compilar contenga el archivo llamado *CMakeLists.txt* que, como hemos comentado anteriormente, está compuesto por una serie de comandos según lo que se quiera crear y compilar.

Los principales comandos utilizados, además de los que aparecen por defecto para lanzar CMake y configurar la ubicación de las salidas, son los siguientes:

- *rosbuild_add_executable (ejecutable src/programa.cpp).* Crea en la carpeta *bin* el ejecutable de *programa.cpp* descrito en *src*.

- *rosbuild_add_library(librería src/programa.cpp).* Crea en la carpeta *lib* la librería de *programa.cpp* descrito en *src*.

- *rosbuild_genmsg().* Auto-genera todas las cabeceras y archivos necesarios de los tipos de mensaje definidos en la carpeta *msg* del paquete y nos los guardará en una nueva carpeta denominada *msg_gen*.

- *rosbuild_gensrv(). A*uto-genera todas las cabeceras y archivos necesarios de los tipos de servicio definidos en la carpeta *srv* del paquete y nos los guardará en una nueva carpeta denominada *srv_gen*.

A la hora de compilar, podemos elegir hacerlo con un único paquete, para lo que introduciremos en la terminal el comando:

```
$ rosmake nombre_paquete
```

También podremos elegir compilar todos los paquetes contenidos en nuestra carpeta de trabajo, introduciendo en la raíz de nuestro espacio de trabajo el comando:

```
$ catkin_make
```

De estas dos maneras, además de hacerlo con los paquetes deseados, compilaremos todos los paquetes de los que éstos dependan, es decir, los que se encuentran especificados en el archivo *manifest.xml* de la forma:

```
<depend package="nombre_paquete">
```

3.1.2.- NODOS.

Como hemos comentado anteriormente, el sistema de ROS se compone de una serie de procesos conectados en tiempo de ejecución. La forma de comunicarse entre ellos es mediante los nodos. Éstos son básicamente ejecutables que utilizan a su vez otro, que es el núcleo del sistema (*"roscore"*), para comunicarse entre ellos.

La forma de transmitir la información es a través de un *topic*, en el que pueden publicar o subscribirse. En la Figura 3.4 se muestra un ejemplo de dos nodos, *gazebo* y *robot_state_publisher*, en el que el primero está publicando en un nodo, llamado *joint_states*, y el segundo está suscrito a él y, por tanto, lee la información que publica *gazebo*.

Figura 3.4: Comunicación entre nodos

Cada nodo posee un nombre único, que lo identifica y lo distingue de todos los demás nodos en ejecución. De esta forma un robot puede llegar a albergar muchos nodos, siempre que contengan nombres distintos, y que cada uno controle un área del robot. Por ejemplo, un nodo puede controlar un láser medidor de distancias, otro nodo podría controlar los motores de la ruedas del robot, otro podría ser el módulo de localización, otro hacer la planificación de la trayectoria, otro proporcionar una vista gráfica de todo el sistema, y así cada nodo se encarga de una tarea e interactúa con los demás.

Los nodos pueden ser escritos en los lenguajes C++ y Phyton usando la librería correspondiente que ROS nos ofrece (roscpp o rospy, respectivamente), que a su vez también son paquetes de éste. Aunque esto no significa que no puedan comunicarse entre sí, puesto que la ejecución y comunicación es independiente. El único requisito para que dos nodos se comuniquen correctamente es que se cumplan las interfaces entre ellos, es decir, que se envíen y se reciban siempre mensajes del mismo tipo.

El código de cualquier nodo en ROS, deberá incluir las siguientes instrucciones:

```
#include "ros/ros.h"


int main(int argc, char **argv)
{
    ros::init (argc, argv, "nombre_nodo");
    ros::NodeHandle n;
    (...)
    return 0;
}
```

Donde *ros.h* es el archivo con las cabeceras necesarias para el uso de las instrucciones más comunes de ROS, sin incluir los tipos de mensajes y otras instrucciones, que deberá hacerse mediante otros archivos.

ros::init (argc, argv, "nombre_nodo") inicializa el nodo en el sistema con el nombre que le asignemos y los argumentos recibidos al ejecutarlo.

Ros::NodeHandle n crea el punto de acceso a las comunicaciones de este nodo con el resto de sistema en ROS. Se trata de una interfaz para crear subscripciones y/o publicaciones de éste con *topics*. Además, este comando, junto con el anterior, inicializa el nodo y lo elimina automáticamente al finalizar la ejecución del programa al que esté asociado. El argumento *n* es el identificador del proceso de este nodo, que siempre debe ser único y distinto al de los demás.

3.1.3.- *TOPICS.*

Como hemos comentado anteriormente, los nodos se comunican a través de unos buses llamados *topics*. El funcionamiento que tienen es muy sencillo, simplemente un nodo debe comunicar al *master* que desea publicar en un *topic* para enviar información, o subscribirse a él para recibir su información. Esta transferencia de información no está restringida a un solo nodo, es decir, pueden existir varios nodos publicando o subscritos al mismo *topic*.

Otra característica que cabe destacar de ellos es que son unidireccionales, esto significa que un nodo que envíe información a través de un *topic* no recibirá información a través de él, ni una confirmación de que sus mensajes están siendo recibidos.

A la hora de declarar un *topic* es necesario definir el tipo de mensaje que se va a enviar a través de él. Esta definición se incluye en el programa mediante las cabeceras generadas a partir de los archivos *.msg*, de los que hablaremos más adelante. Otra cosa que se debe definir es el nombre, que debe ser único y normalmente de la forma: */nodo_que_publica/nombre_topic*, aunque no es obligatorio que siga esta convención.

La forma que nos proporciona *roscpp* para escribir a través de un *topic* desde un programa, y crearlo en caso de que no existiera antes, es incluyendo las siguientes instrucciones en él:

```
#include " paquete/archivo_def_tipo_mensaje.h"


(...)

    ros::Publisher nombretopic_pub = n.advertise<tipo_de_mensaje
    >("nombretopic", 1000);
    Tipo_de_mensaje;
    msg.data = "mensaje";
    nombretopic_pub.publish(msg);

(…)
```

Mediante la primera instrucción se incluyen las definiciones de la clase del tipo de mensaje que se va a utilizar. Un ejemplo sería *#include "std_msgs/String.h"*, donde incluiríamos la clase *String* del paquete *std_msgs*.

Con la segunda instrucción creamos el *topic* "nombretopic" con una cola con capacidad para 1000 mensajes, que puede variar según necesidades del programador. A continuación, se define el tipo de mensaje que se va a enviar, siguiendo el ejemplo anterior podría ser *std_msg::String*, y se almacena en el objeto que se va a enviar con la tercera y cuarta instrucción.

La última instrucción (*publish()*) tiene la finalidad de publicar el mensaje *msg,* almacenado previamente, cuando el programa o nodo esté en ejecución. Para ello, se utiliza el identificador del *topic* devuelto por la segunda instrucción, de la clase *ros::Publisher* y denominado *nombretopic_pub*.

3.1.4.- SERVICIOS.

Los servicios son el método que utiliza ROS para enviar y recibir mensajes de forma bidireccional, en lugar de los *topics* que son unidireccionales.

Este tipo de comunicación es credo por un nodo, de tal manera que cualquier otro nodo puede utilizarlo llamándolo con un mensaje y esperando una respuesta de éste.

La forma de definir el tipo de servicio que se va a ofrecer es mediante los archivos *.srv*, es decir, con ellos se definirá el tipo del mensaje que se va a enviar para llamar al servicio y el tipo del que se recibirá como respuesta. Al igual que pasa con los *topics*, se deben incluir las cabeceras de los archivos *.srv* del paquete en cuestión que contenga las definiciones de las clases del tipo de servicio.

A continuación, se muestra un ejemplo con las funciones básicas que proporciona *roscpp* para crear y ofrecer un servicio:

```
#include "Matematicas/SumaDosEnteros.h"


bool fun_sum(Matematicas::SumaDosEnteros::Request &req,
Matematicas::SumaDosEnteros::Response &res)
{
    res.sum = req.a + req.b;
    Return true;
}

int main(int argc, char **argv)
{
    (...)
```

```
    ros::ServiceServer service = n.advertiseService("
    suma_dos_enteros",fun_sum);

    (...)
}
```

Al igual que en el caso de los *topics*, la primera instrucción incluye el archivo con la clase del tipo de servicio que se va a usar, en este caso *SumaDosEnteros.h* del paquete *Matematicas*.

La última instrucción crea propiamente el servicio, devolviendo su identificador *service* al programa. El nombre que se adjudica al servicio es *suma_dos_enteros* y la función que ejecutará el servicio al ser llamado será *fun_sum*, que debe ser definida previamente en el programa.

Como puede observarse, mediante la segunda instrucción la función *fun_sum* recibe como argumentos las direcciones donde se encuentran el mensaje recibido (*Request*) y donde se debe escribir la respuesta (*Response*).

Equivalente a la creación de este servicio, se debe crear un cliente para éste. Un ejemplo, siguiendo el anterior, sería:

```
#include "Matematicas/SumaDosEnteros.h"


int main(int argc, char **argv)
{
    ros::ServiceClient client = n.serviceClient<Matematicas
    ::SumaDosEnteros>("suma_dos_enteros");

    beginner_tutorials::AddTwoInts srv;
    srv.request.a = 10;
    srv.request.b = -3;

    client.call(srv)
    return 0;
}
```

De nuevo, lo primero que se hace es incluir el archivo de cabecera del tipo de servicio.

La segunda instrucción crea el cliente al servicio *suma_dos_enteros*, creado en el ejemplo anterior, obteniendo el identificador *client* para llamar al servicio.

La tercera, es un objeto de la clase del servicio, que contiene a los miembros *request y response*.

Por último, se crea el objeto, de la clase definida en el mensaje, y se envía, con la función *call* y el identificador *client.*

3.1.5.- MENSAJES.

Como hemos comentado anteriormente, tanto los nodos como los servicios se comunican enviando mensajes. En el caso de la comunicación a través de los topics, éstos no son más que una estructura de datos equivalentes a una estructura de C++, que son definidos en los archivos *.msg*, guardados en la carpeta *msg* de un paquete, en forma de lista tipo-nombre. Un ejemplo de este tipo de archivo, en el que el mensaje contendría dos datos del tipo *int32*, sería:

```
int32 x
int32 y
```

Para la comunicación de los servicios, se definen otros archivos equivalentes a los *.msg*, los *.srv*. La diferencia que existe entre unos y otros es que los *.srv* se componen de dos listas separadas por "---". En primer lugar aparece la lista de la estructura de *request* y, en segundo, la de *response*. Al igual que ocurre con los archivos *.msg*, se guardan en su carpeta correspondiente (*srv*) para que el compilador pueda encontrarlos y cree sus archivos de cabecera. A continuación, se muestra un ejemplo de un servicio que recibe un dato de tipo *string* (*request*) y devuelve otro del mismo tipo (*response*):

```
string str

---

string str
```

Existe la posibilidad de crear variables de tipos de mensajes o servicios dentro del programa, en C++. Para ello simplemente se deberán incluir los archivos de encabezamiento generados y tratarlos como si fueran una variable u objeto más. Por ejemplo, si se crea una variable del tipo h*eader*, para acceder a su variable miembro llamad *stamp*, debemos escribir el comando: *header.stamp*.

3.1.6.- *ROSCORE.*

Los nodos, para comunicarse, requieren de otro llamado *roscore* [12], que funciona como núcleo del sistema, proporcionando una plataforma mediante el registro de nombres, servicios y parámetros. Este nodo "principal" es necesario para que los demás puedan encontrarse unos a otros y, de esta forma, comunicarse. En la Figura 3.5 se muestra cómo actúa *roscore* a la hora de conectar los distintos nodos en ejecución.

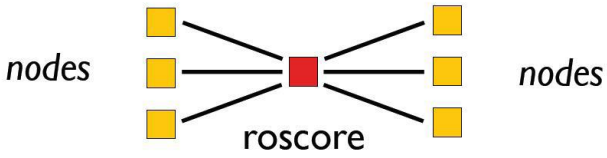

Figura 3.5: Comunicación a través de *roscore*.

Además de cumplir con la finalidad de ser la plataforma de comunicación para los nodos, *roscore* también proporciona un registro de salida al que todos los nodos envían información, denominado */rosout*.

Otra herramienta que también nos proporciona es la del Servidor de Parámetros, que es utilizado por los nodos para almacenar o leer parámetros en tiempo de ejecución, permitiendo además que el usuario pueda leer y/o modificar estos parámetros a través del terminal en tiempo de ejecución.

La ejecución del *master* es obligatoria siempre antes de que se lancen los nodos, de que se comuniquen entre ellos o de que necesiten leer parámetros. Para ello, antes de ejecutar cualquier aplicación en ROS, deberemos introducir el siguiente comando en la terminal:

```
$ roscore
```

3.1.7.- LANZADORES.

Como se verá más adelante, es posible ejecutar nodos, llamar a servicios y leer o publicar en *topics* directamente a través de la línea de comandos de la terminal [13]. Sin embargo, si lo que deseamos es lanzar varios nodos a la vez, cargar una serie de parámetros o ejecutar varias herramientas, existe una forma más sencilla de hacerlo, sin tener que hacerlo uno a uno. Para ello ROS contiene una herramienta denominada *roslaunch* que permite, por ejemplo, lanzar varios nodos con sus argumentos, tomándolos de un archivo con extensión *.launch*. De esta manera para lanzar lo que deseemos simplemente deberemos introducir el comando:

```
$ roslaunch nombre_paquete archivo.launch
```

Los archivos *.launch* están escritos en *xml* y en ellos se especifican los nodos a lanzar, los argumentos de entrada a éstos y los parámetros que son necesarios para crear una aplicación completa. Todo esto se especifica a través de las etiquetas (tags) de las que se compone, como cualquier otro archivo *xml*. A continuación, comentaremos las más comunes, y que hemos utilizado, y su finalidad:

- *<launch>.* Siempre debe aparecer como primer elemento de cualquier archivo *.launch*. El objetivo que tiene es la de identificar el tipo de archivo que es. Además de incluirla en la primera línea del archivo, deberemos hacerlo también en la última en formato de cierre *</launch>*.

- *<node>*. Se utiliza para lanzar un nodo. Lanza una copia del "nodo tipo", por lo que es necesario darle un nombre diferente y especificar el paquete y el nombre del "nodo tipo" de la forma:

```
<node pkg="paquete" name="nombre" type="nombre_nodo_tipo"
args="argumentos" respawn="true"/>
```

- *<param>*. Permite definir un parámetro en el Servidor de Parámetros. Éste puede ser un valor numérico constante y se definiría de la siguiente manera:

```
<param name="nombre_parametro" value="valor_fijado">
```

También se puede definir una cadena de caracteres donde se almacenará el contenido de un archivo:

```
<param name="nombre_parametro" command"archivo">
```

- *<rosparam>*. Esta etiqueta permite el uso de archivos *.yaml* para cargar o eliminar parámetros del Servidor de Parámetros, de la siguiente forma:

```
<rosparam file"archivo.yaml" command="load" />
```

```
<rosparam command="delete" param="mi/parametro" />
```

También se puede usar esta etiqueta dentro de la de <node>, para que un parámetro sea tratado como un nombre privado.

- *<arg>*. Permite recibir un argumento al ejecutar el archivo *.launch* y definirle unos valores por defecto. Un ejemplo en el que recibiríamos el argumento *"paused"* y le definiríamos el valor por defecto *"true"*, sería el siguiente:

```
<arg name="paused" default="true" />
```

- *<remap>*. Permite reasignar el valor del argumento *args* de un nodo previamente creado. La forma de hacerlo es la siguiente:

```
<remap from=" inicial" to=" nuevo"/>
```

- *<include>*. Permite importar y lanzar otro archivo *.launch*. Para lanzarlo deberemos escribir:

```
<include file="archivo.launch">
```

3.1.8.- LÍNEA DE COMANDOS DE ROS.

ROS cuenta con sus propias herramientas de línea de comandos [14] para la búsqueda de paquetes y archivos, la compilación y depuración de código, la ejecución de programas y la obtención de información del sistema en tiempo real. Los más comúnmente utilizados son los siguientes:

- *$ roscore*. Como se ha comentado anteriormente, este comando es utilizado para ejecutar el *master*, el Servidor de Parámetros y el nodo de registro.

- *$ rosstack*. Proporciona información sobre la pila que deseemos. Se utiliza de la siguiente manera:

```
$ rosstack [comando] [pila]
```

- *$ rospack*. Proporciona información sobre un paquete. Funciona de igual manera que *rosstack*, sólo que para el caso de los paquetes:

$ rospack [comando] [paquete]

- *$ roscd*. Permite acceder a un paquete determinado introduciendo el siguiente comando:

$ roscd [paquete]

- *$ rosls*. Permite ver el contenido del paquete que determinemos de la siguiente manera:

$ rosls [paquete]

- *$ roscreate-pkg*. Crea un paquete en la carpeta que nos encontremos. Se pueden crear dependencias a otros paquetes, con el siguiente comando:

$ roscreate-pkg [nombre_paquete] [dependencia1] . . .

- *$ rosmake*. Compila el paquete especificado, y todos de los que dependa, de la siguiente manera:

$ rosmake [paquete]

- *$ rosdep*. Instala el paquete, y todos de los que dependa, introduciendo el comando:

$ rosdep install [paquete]

- *$rorun*. Ejecuta un nodo individual, introduciendo:

$ rosrun [paquete] [ejecutable]

- *$ rosnode*. Permite obtener información de los nodos que se encuentran en ejecución o eliminar uno de la ejecución, introduciendo una serie de comandos que se detallarán más adelante. Se utiliza de la forma:

$ rosnode [comando] [nodo]

- *$ rostopic*. Su funcionamiento es similar a *rosnode*, pero para operar con los *topics*.

- *$ rosservice*. Funciona de igual manera que los dos anteriores pero para servicios.

- *$ rosmsgs*. Permite obtener la información de un tipo de mensaje. Se utiliza de la forma:

$ rosmsg [comando] [tipo_mensaje]

Rosnode, *rostopic* y *rosservice* nos serán muy útiles para observar cómo se comunican los distintos nodos que tengamos en ejecución y ver como se interrelacionan entre ellos a través de los *topics*. Los comandos más utilizados son:

- *Info*. Muestra información de un nodo, *topic* o servicio.

- *List*. Lista los nodos, *topics* o servicios activos.

- *Kill*. Elimina un nodo en funcionamiento.

- *Echo*. Muestra por pantalla los mensajes de un *topic*.

- *Pub*. Publica datos en un *topic*.
- *Type*. Muestra el tipo, de *topic* o servicio, del que es el especificado.

Si se quiere obtener más información acerca de los comandos con los que cuenta cada herramienta y la forma en que se usan, puede escribirse desde una terminal:

```
$ [herramienta] -h
```

3.2.- SIMULACIÓN EN GAZEBO.

Una vez realizada una primera aproximación a ROS, el siguiente paso en nuestro trabajo será el de realizar la simulación de un robot dentro de un entorno simulado en la plataforma.

A partir de esta versión de Gazebo y de ROS Hydro, el simulador ya no cuenta con una dependencia directa de ROS, ahora éste se instala como un paquete independiente de Linux. Por lo que deberemos seguir las instrucciones proporcionadas en la web del simulador para instalarlo[2].

[2] http://gazebosim.org/tutorials?tut=install&ver=1.9&cat=get_started

3.2.1.- INTEGRACIÓN CON ROS.

Para lograr la integración con ROS, existe un conjunto de paquetes llamados *gazebo_ros_pkgs*. Éstos proporcionan las interfaces necesarias para simular un robot en Gazebo utilizando mensajes, servicios y reconfiguración dinámica de ROS. En la Figura 3.6, se muestra una visión general de las funcionalidades que nos proporciona *gazebo_ros_pkgs*.

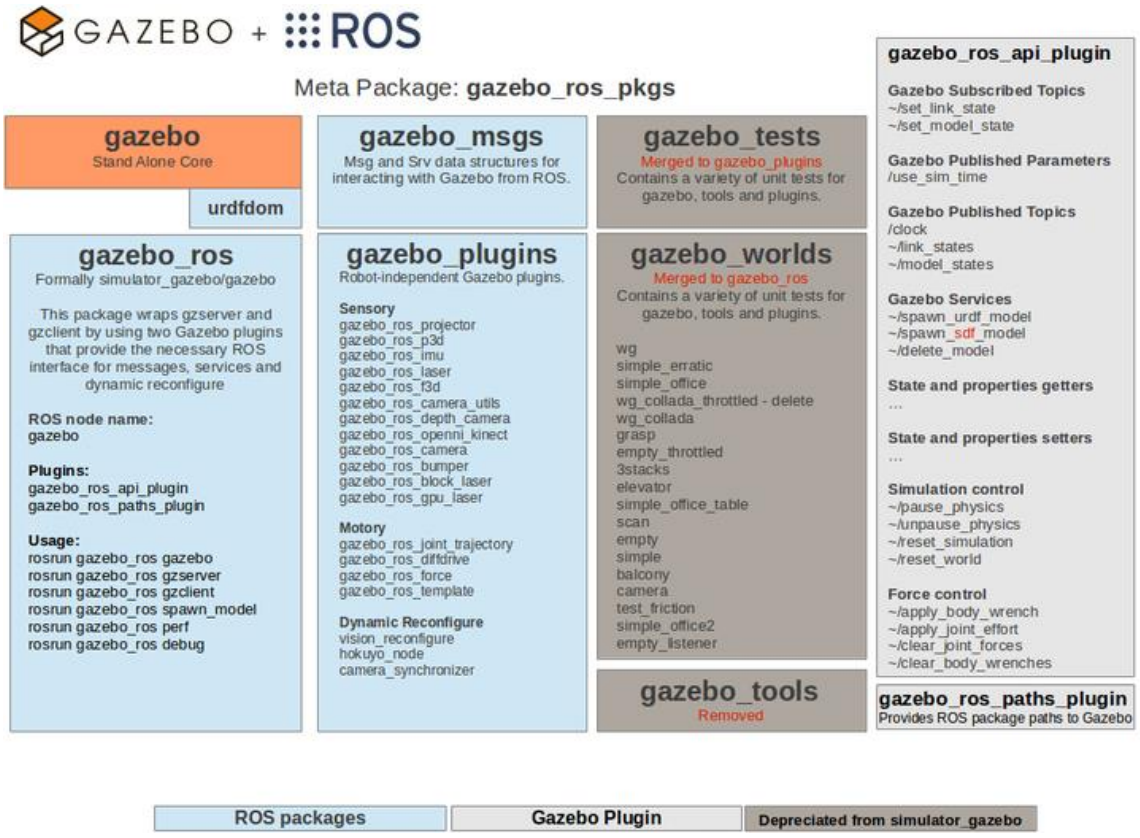

Figura 3.6: Pila *gazebo_ros_pkgs*.

A partir de ahora, para poder realizar una correcta integración con ROS, cada vez que lancemos un modelo desde un archivo *.launch*, deberemos crear el nodo *spawm_urdf* perteneciente al paquete *gazebo_ros*, en lugar del *spawn*, perteneciente a *gazebo*.

Una vez realizada la integración con ROS, cuando carguemos Gazebo a través del paquete *gazebo_ros*, se creará automáticamente el nodo */gazebo* y una serie de *topics*

de los que leerá y a los que se subscribirá para poder comunicarse con ROS. También, la creación de éste, proporcionará una serie de servicios que veremos más adelante.

La creación de todos estos *topics* y servicios, así como la del nodo */gazebo*, se realiza a través del paquete gazebo*_ros_api_plugin,* como puede observarse en la Figura 3.6.

3.2.2.- CONFIGURACIÓN DEL MODELO Y EL ENTORNO DE SIMULACIÓN.

A la hora de realizar cualquier simulación en Gazebo, lo primero que deberemos hacer es cargar un entorno en él. Para ello es necesario, al ejecutar Gazebo, pasarle un archivo de configuración del mundo, en el que se especifican los parámetros del motor físico ODE, como son el tiempo de integración, el valor de la gravedad, etc.; del motor gráfico OGRE (Object-Oriented Graphics Rendering Engine), como son el renderizado, las sombras, el cielo, ambiente, etc., y de los objetos que deben aparecer en el simulador automáticamente.

Para las simulaciones llevadas a cabo en nuestro caso, se ha utilizado la configuración del entorno *empty.world* de Willow Garage, de él tomamos los valores del físicos y del motor gráfico. Aunque estos valores no se vayan a modificar, a continuación comentaremos algunas de las cosas que se están configurando, por si pueden ser útiles a la hora de hacer posibles análisis de las simulaciones. Los parámetros del motor físico ODE, son los siguientes:

```
<physics:ode>
    <stepTime>0.001</stepTime>
    <gravity>0 0 -9.8</gravity>
    <cfm>0.0000000001</cfm>
    <erp>0.2</erp>
    <quickStep>true</quickStep>
    <quickStepIters>10</quickStepIters>
    <quickStepW>1.3</quickStepW>
    <contactMaxCorrectingVel>100.0</contactMaxCorrectingVel>
    <contactSurfaceLayer>0.001</contactSurfaceLayer>
</physics:ode>
```

Donde *stepTime* es el paso de integración del simulador, *gravity* es la gravedad aplicada sobre cada cuerpo; *cfm* (*constraint force mixing*) es un parámetro que varía entre 0 y 1 y que especifica el grado de dureza de las restricciones de los robots en cuanto a los contactos, cuanto mayor sea el valor de este parámetro menores serán las restricciones (permitiendo cierta penetración entre dos cuerpos), y viceversa, y *erp* (*error reduction parameter*), que es otro parámetro que varía también entre 0 y 1 para reducir el posible error en la integración de las articulaciones, que haría que no se cumplieran exactamente las restricciones impuestas sobre ellas. Las etiquetas en las que aparece *quickStep* están relacionadas con un nuevo método de integración, en el que no entraremos en detalle por no ser relevante para nuestras simulaciones, pero cuya labor es agilizar la simulación. Por último, c*ontactMaxCorrectingVel* define la máxima velocidad entre contactos, si no se define es infinita por defecto, y *contactSurfaceLayer* define una profundidad permitida de penetración antes de estabilizarse un robot en el suelo, en caso de que este valor fuera cero podría provocar que no llegara a producirse dicha estabilización.

Además de incluir en nuestro lanzador el archivo *empty.world,* para aprovechar los parámetros físicos de éste, cargaremos un archivo *pr2.world* creado por nosotros, en el que incluiremos los elementos de nuestro entorno, es decir, la gasolinera (*gas_station*), el sol (*sun*) y el suelo (*ground_plane*). La forma de hacerlo es incluyendo el archivo de modelo de cada elemento mediante la siguiente etiqueta:

```
<include>
  <uri>model_file</uri>
  </include>
```

Un archivo de modelo es similar a un *.world* ya que utiliza el mismo formato SDF. El propósito de este tipo de archivos es facilitar la reutilización de éstos y simplificar los

archivos *.world*. Con esta finalidad de reutilización, un gran número de modelos son facilitados en la base de datos de modelos online (*online model database*[3]), permitiendo insertar cualquier modelo de ésta descargándolo en tiempo de ejecución, siempre que se disponga de conexión a Internet. Los modelos que utilizamos en nuestro caso (*gas_station*, *sun* y *ground_plane*) pertenecen a esta base de datos. En la Figura 3.7 puede observarse el entorno de simulación que utilizamos antes de que se cargue ningún robot en él.

Figura 3.7: Entorno de simulación.

Una vez en marcha Gazebo, es posible cargar el modelo de un robot en cualquier momento y en cualquier posición. La forma de hacerlo es utilizando el ejecutable *spawn_model*. Por lo general, los robots se suelen cargar desde el lanzador con el que se ejecuta el simulador, aunque también se puede hacer desde una terminal o desde un programa en C++. En nuestro caso, lo haremos desde el archivo *.launch* en el momento en que cargamos Gazebo con el entorno de simulación.

[3] https://bitbucket.org/osrf/gazebo_models

Para cargar el modelo desde el lanzador, lo primero que debemos hacer es leer el modelo URDF y guardarlo en un parámetro, en nuestro caso lo hacemos en *robot_description*. Una vez hecho esto, crearemos un nodo llamado *spawn_urdf* del tipo *spawn_model* con el que cargaremos el robot PR2 en la simulación. Por tanto, las dos instrucciones a incluir en el lanzador son:

```
<param name="robot_description" command="$(find xacro)/xacro.py $(find
pr2_description)/robots/pr2.urdf.xacro" />

<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -model pr2" />
```

Como puede observarse, hay una serie de argumentos que se introducen para la ejecución del nodo. Con ellos lo que se hace es indicarle el parámetro del servidor donde está almacenada la descripción del robot (*-param robot_description*), el tipo de archivo que es (*-urdf*) y el nombre con el que debe aparecer (*-model pr2*), ya que se puede cargar varias veces el mismo modelo si se desea.

En la figura 3.8 puede observarse como se ha cargado nuestro robot en la gasolinera con el nombre *pr2*.

Figura 3.8: PR2 y entorno de simulación.

En cuanto a la interfaz gráfica del simulador, Gazebo nos ofrece la posibilidad de intervenir en el tiempo de ejecución, es posible pausar y reiniciar la simulación cuando se quiera. También se pueden manipular los modelos que tengamos cargados en él, variando su posición y sus características físicas, modificando los sistemas de referencia inerciales para esto último. En la figura 3.9 puede observarse, en el lado izquierdo de la imagen, la interfaz en la que modificar los valores del modelo y, en la parte inferior, la que nos permite modificar el tiempo de nuestra simulación.

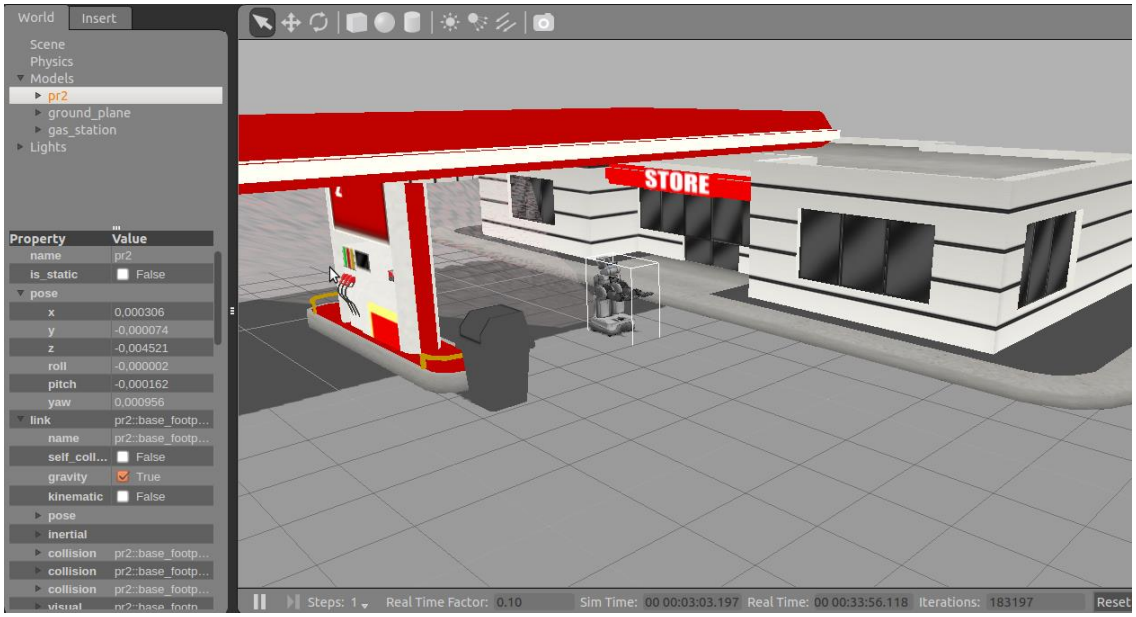

Figura 3.9: Interfaz gráfica de Gazebo.

Como hemos comentado, Gazebo nos proporciona la posibilidad de pausar la ejecución y reiniciarla cuando se quiera, pero este valor también puede ser configurado desde el lanzador. Podemos cargar nuestro entorno con la simulación pausada, asignando un valor (*true* o *false*) al argumento *paused* a la hora de cargar el archivo *.world*, haciéndolo de la siguiente forma:

```
<arg name="paused" value="false"/>
```

Si se está lanzando una simulación en Gazebo, el tiempo de ejecución será el que marque el simulador, es decir, el tiempo de simulación. Éste será el que ROS envíe por el *topic* */clock*, al que se subscriben automáticamente todos los nodos, de forma que comparten el mismo vector de tiempo. Por tanto, debemos tener en cuenta que si se para la simulación, también lo hará el reloj de simulación y, en consecuencia, todos los demás nodos que se estén ejecutando y usando este reloj. Además, también ha de tenerse en cuenta que la velocidad de la simulación puede variar según la carga de cálculos que suponga ésta, por lo que Gazebo también variará la velocidad del reloj de simulación para que todos los demás nodos lo tengan en cuenta.

Para que el simulador utilice el tiempo de simulación, deberemos especificarlo añadiendo al lanzador el siguiente parámetro cuando carguemos nuestro mundo:

```
<arg name="use_sim_time" value="true"/>
```

Los mensajes que Gazebo envíe a través del topic /clock tienen la misma estructura que cualquier otro mensaje de tiempo, dividiendo el tiempo en segundos y nanosegundos.

3.2.3.- COMUNICACIÓN CON ROS.

Como se ha explicado anteriormente, con la integración de Gazebo y ROS, se crea un nodo que publica y se subscribe a una serie de *topics*, al igual que ofrece una serie de servicios, mediante los cuales se comunica con el resto de nodos, como puede ser el *master* (*roscore*). Los *topics* a los que se subscribe y los que ofrece, por defecto, son los que se muestran en la figura 3.10.

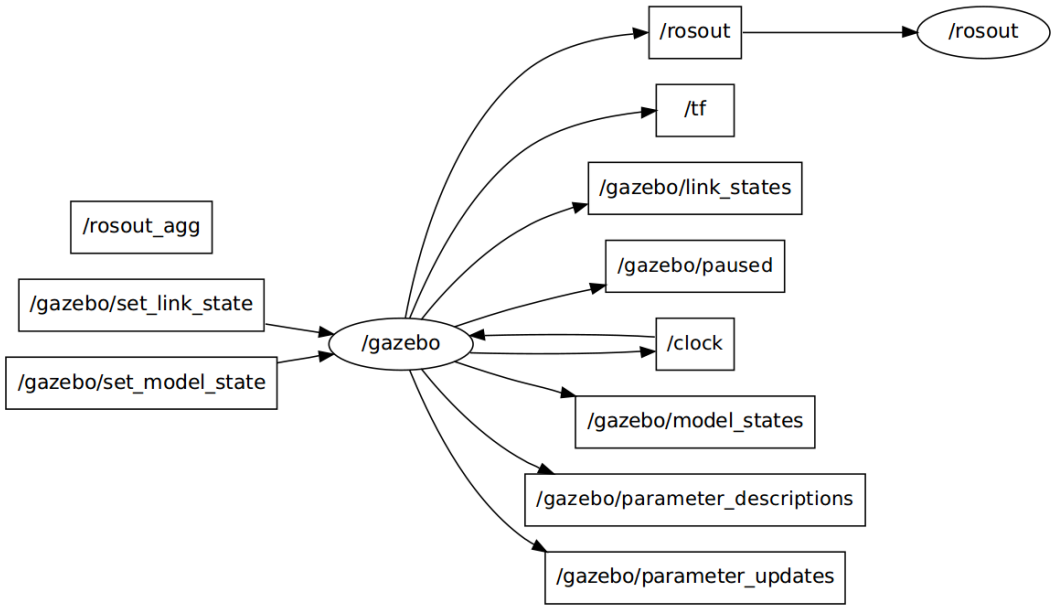

Figura 3.10: *Topics* cargados con Gazebo.

Todos ellos están relacionados con el estado de los objetos existentes en el simulador, es decir, con sus posiciones, parámetros y descripción. En nuestro caso, el único que nos interesará será */gazebo/model_states*, ya que a través de lo que Gazebo publique en él podremos observar en Rviz como se mueve nuestro robot. También se utilizará el *topic /clock*, aunque, como hemos comentado, esto lo hace por defecto el sistema para sincronizar todos los nodos. El resto de *topics* que utilizaremos serán los creados por los distintos *plugins* con los que contará el robot y los de los controladores que usemos para mover a éste.

Además, Gazebo también proporciona una serie de servicios, por defecto, mediante los cuales poder enviar mensajes desde ROS. A continuación, se realiza una descripción de los más comunes, que han sido utilizados en este proyecto para realizar comprobaciones:

- *gazebo/spawn_urdf_model*. Introduce un robot en la simulación, que tiene que estar definido en un archivo URDF.

- *gazebo/delete_model*. Este servicio elimina de la simulación el modelo especificado.

- *gazebo/get_model_properties*. Muestra las propiedades de un modelo presente en la simulación.

- *gazebo/get_model_state*. Retorna el estado de un modelo en la simulación.

- *gazebo/get_joint_properties*. Retorna las propiedades de una articulación (*joint*) que se encuentre en simulación.

- *gazebo/apply_joint_efort*. Aplica un esfuerzo a una articulación especificada, que se encuentre en simulación.

- *gazebo/clear_joint_forces*. Elimina los esfuerzos aplicados sobre las articulaciones en simulación, o sobre una especificada.

- *gazebo/set_model_state*. Mediante este servicio podemos cambiar la posición de un modelo en simulación, en coordenadas cartesianas.

3.2.4.- *PLUGINS.*

Los *plugins* son, por definición, complementos que se añaden a los modelos para proporcionarles funciones y/o sensores. En general, salvo los que son creados para unos paquetes específicos, todos se encuentran en el paquete *gazebo_plugins* en forma de librería.

Por regla general, los *plugins* se añaden a la descripción URDF del robot entre dos etiquetas que identifican que se refieren al simulador(*<gazebo>*) y, posteriormente, mediante la etiqueta *<controller:tipo_plugin>* se le asigna un nombre, se le especifica la librería a utilizar y se seleccionan los valores para los parámetros del *plugin*. Todo esto se hará de la siguiente forma:

```
<gazebo>
<cotroller:tipo_plugin name = "nombre" plugin = "libreria">
<opción_1>...</opción_1>
(...)
<opción_n>...</opción_n>
</controller:tipo_plugin>
</gazebo>
```

Los *plugins* con los que cuenta nuestro PR2 están relacionados, en su mayoría, con sus distintos sensores. La creación de cada uno de ellos va asociada a la creación de uno o más *topics*, a través de los cuales se publican los mensajes con la información que obtienen u otro tipo de mensajes, como pueden ser especificaciones sobre el tipo de datos que se están obteniendo.

3.3.- MOVIMIENTO DE PR2 EN GAZEBO.

Una vez simulado el PR2 y creado el entorno en Gazebo, el siguiente paso será mover el robot. Nuestro trabajo consistirá en mover la base, de tal forma que se pueda

desplazar por el entorno, y moverla cabeza, con la finalidad de que el robot pueda ver a través de su cámara Kinect las cosas que tiene alrededor.

Para facilitar la utilización de los controladores de nuestro robot, utilizaremos la pila de paquetes de ROS *pr2_mechanism*. Ésta está destinado a ofrecer una infraestructura para el control del robot mediante un ciclo de control en tiempo real. En principio, fue creado para el control exclusivo de PR2, pero también puede ser usado con algunos robots similares a éste.

La pila se compone de unos paquetes que dan soporte tanto a robots reales como simulados, y ofrece una interfaz de programación en C++ bastante sencilla. A continuación, haremos una descripción de los paquetes que nos han sido útiles para controlar nuestro PR2 simulado:

- *pr2_controller_interface*. Este paquete ofrece la interfaz C++ para especificar el controlador en tiempo real. Contiene una clase base definida, *pr2_controller_interface::Controller*, de forma que toda clase que hereda de ella puede ser ejecutada por el nodo *Controller Manager* en un ciclo de tiempo real.
  Para crear un controlador basta con crear una clase que herede esta forma, es decir, que sea una clase hija de ella, y modificar la definición de las funciones miembro *init()*, *starting()*, *update()* y *stopping()* con las que cuenta, según lo que deseamos que haga el controlador.

- *pr2_controller_manager*. Este paquete contiene toda la infraestructura para cargar, inicializar, ejecutar y parar en tiempo real múltiples controladores desde ROS. También se encarga de comprobar que la ejecución se haga a 1 KHz y en tiempo real, emitiendo un mensaje de error si esto no se produce. *Controller Manager* puede entenderse como un nodo más del sistema que se encarga de llevar a cabo el programa definido en la clase *Controller,* definida anteriormente, y aplicará las fuerzas que se especifiquen sobre el robot. Para ello, el paquete nos proporciona el nodo *spawner*, que permite cargar y ejecutar los controladores que se le pasen como argumento.
  pr2_controller_manager cuenta con una serie de comandos para interactuar con los controladores, éstos son:

  - *load.* Carga un controlador (constructor e incialización).

- *unload*. Descarga un controlador que ha sido cargado previamente (destructor).

- *start*. Inicia un controlador.

- *stop.* Detiene un controlador previamente iniciado.

- *spawm.* Carga e inicia un controlador.

- *kill*. Detiene y descarga un controlador previamente cargado y/o iniciado.

Para ejecutar estos comandos desde la terminal deberemos introducir en ella:

$ *rosrun pr2_controller_manager pr2_controller_manager <comando> <nombre_controlador>*

También cuenta con una serie de comandos para chequear el estado de los controladores, éstos son:

- *list*. Lista todos los controladores, en el orden en el que han sido ejecutados, y muestra el estado de cada uno de ellos.

- *list-types*. Lista todos los tipos de controladores de los que el *Controller Manager* tiene conocimiento. Si alguno no aparece, no podrá ser cargado e iniciado.

- *list-joints*. Lista todas las articulaciones y actuadores que están siendo utilizados por el *Controller Manager*.

- *reload-libraries*. Actualiza todas las librerías de controladores que están disponibles como *plugins*. Este comando ofrece la ventaja de que, en el momento en el que se está desarrollando un controlador, y es necesario probarlo, permite actualizarlo sin necesidad de resetear el robot cada vez que queramos hacerlo.

- *reload-libraries --restore*. Actualiza todas las librerías de controladores disponibles como *plugins* y restaura todos los controladores a su estado original.

Para ejecutar estos comandos desde la terminal deberemos introducir en ella:

```
$ rosrun pr2_controller_manager pr2_controller_manager <comando>
```

También existe la posibilidad de utilizar *pr2_controller_manager* para cargar e iniciar controladores desde un archivo *.launch*, como veremos más adelante.

- *pr2_mechanism_model*. Este paquete contiene el modelo del robot que es utilizado por los controladores de tiempo real dentro del *Controller Manager*. Cuenta con tres clases, entre otras, que son muy útiles para acceder al robot, en concreto a sus articulaciones, durante la simulación:

  - *pr2_mechanism_model::RobotState*. Esta clase contiene el modelo del robot en formato URDF. También cuenta con una función, *getJointState()*, mediante la cual se puede obtener, a partir del nombre de la articulación, objetos de la clase *pr2_mechanism_model::JointState*.

  - *pr2_mechanism_model::JointState*. Esta clase da acceso, en tiempo de simulación, a una articulación en concreto. Mediante diversas funciones, permite obtener la posición (*position_*), la velocidad (*velocity_*) y el esfuerzo (*measured_effort_*) sobre una articulación. También es posible enviar un esfuerzo a la articulación con la función *commanded_effort_*.

  - *pr2_mechanism_model::Chain*. Esta clase es utilizada para trabajar con cadenas cinemáticas de varias articulaciones.

Debemos tener en cuenta que para poder ejecutar cualquier controlador, es necesario tener lanzada nuestra simulación con el PR2 en ella.

3.3.1.- MOVIMIENTO DE LA BASE.

A la hora de mover la base del PR2, el objetivo que buscamos es el de hacerlo introduciendo las instrucciones desde el teclado en tiempo de ejecución. Para ello, nos apoyaremos principalmente en dos paquetes de ROS, además de los ya comentados anteriormente.

El paquete *pr2_teleop*, contenido dentro de la pila *pr2_apps*, nos proporciona un programa desde el que capturar los comandos introducidos por teclado y traducirlo en una velocidad para los actuadores de la base. También cuenta con un programa que nos permite mover el PR2 desde un mando de Play Station 3, pero al no contar con éste, nos limitaremos a hacerlo desde el teclado.

Para ejecutar el programa que nos interesa, deberemos lanzar el archivo *teleop_keyboard.launch* contenido en el paquete. Para ello deberemos introducir en nuestra terminal la siguiente instrucción:

```
$ rosrun pr2_teleop teleop_keyboard.launch
```

El *pr2_teleop* lo primero que hace es mostrarnos por pantalla las opciones que tenemos para poder mover nuestro robot, como puede observarse en la Figura 3.11. Con las teclas “W”, “S”, “A” y “D” transmitiremos una velocidad lineal (en *m/s*) a lo largo del eje X e Y, y con las teclas “Q” y “E”, será una velocidad angular (en *rad/s*) en un sentido u otro. Introduciendo “W” transmitiremos una velocidad lineal positiva en la dirección del eje X en un sistema de coordenadas situado en la base del robot, es decir, moveremos el PR2 hacia delante; con la “S” transmitiremos una velocidad lineal negativa en esa misma dirección, lo moveremos hacia atrás; con la “A” transmitiremos una velocidad lineal positiva en la dirección del eje Y, lo moveremos hacia su izquierda; con la “D” la velocidad lineal será negativa en la dirección del eje Y, se moverá hacia su derecha; con la “Q” transmitiremos una velocidad angular positiva, es decir, lo moveremos en sentido anti horario, y, por último, con la “E” la velocidad angular será negativa y se moverá en sentido horario. También podremos incrementar las velocidades transmitidas si, a la vez que pulsamos una de las teclas anteriores, mantenemos presionado “*Shift*”.

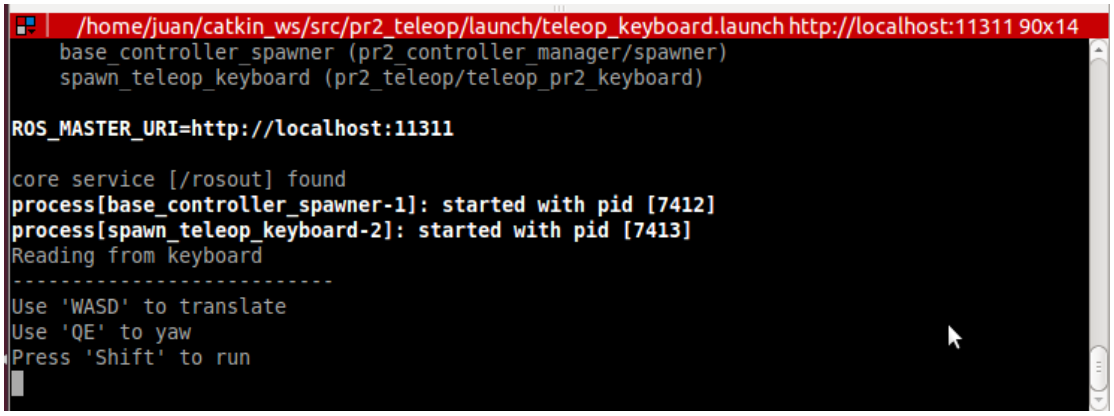

Figura 3.11: Ejecución *pr2_teleop*.

Todas estas velocidades serán enviadas mediante un mensaje del tipo *Twist*[4], contenido dentro del paquete *geometry_msgs* a un *topic* llamado *base_controller/command*. Por tanto, necesitaremos que algún controlador lea estos mensajes del *topic* y aplique las velocidades.

Para ello, utilizaremos un controlador de la pila *pr2_controller*. Ésta cuenta con una serie de paquetes para el control en tiempo real del PR2, pero que pueden ser utilizados, en parte, para otros robots. En concreto utilizaremos el paquete *pr2_mechanism_controllers*, que contiene los controladores *LaserTrajController*, *Pr2BaseController*, *Pr2Odometry* y *Pr2GripperController*.
Como puede deducirse de su nombre, el controlador que a nosotros nos interesa es el PR2BaseController, que se encarga de leer las velocidades enviadas al *topic* *<nombre_controlador>/command* y convertirlas en comandos del motor, para enviárselos a la base móvil del PR2.
Al publicar el *pr2_teleop* en un *topic* llamado *base_controller/command*, necesitaremos que nuestro controlador del tipo *Pr2BaseController* se subscriba a éste. Por tanto, el nombre que debe tener el controlador es *"base_controller"*. Lo haremos

[4] http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html

creando un controlador con ese nombre del tipo que deseamos, introduciendo en la terminal:

```
$ rosparam set base_controller/type pr2_mechanism_controllers/Pr2baseController
```

Lo siguiente que se deberá hacer es configurar los parámetros del controlador, para ello utilizaremos un archivo *.yaml*. En este tipo de archivos se guardan los parámetros que se vayan a necesitar para el controlador y que se cargarán y leerán desde el Servidor de Parámetros, como son el nombre de las articulaciones a controlar (deben ser las mismas que en el archivo URDF) y los valores de los coeficientes de los PID.

En nuestro archivo *pr2_base_controller.yaml*, además de configurar lo que hemos comentado anteriormente, configuraremos los siguientes parámetros:

- *timeout*. Si el tiempo desde la última velocidad recibida es superior al valor configurado en este parámetro, el controlador parará el robot. En nuestro caso fijaremos el valor en 0.2.

- *max_translational_acceleration/x*. Fija la máxima aceleración permitida en el eje X. En nuestro caso fijaremos el valor en 5.0.

- *max_translational_acceleration/y*. Fija la máxima aceleración permitida en el eje y. En nuestro caso fijaremos el valor en 5.0.

- *max_rotational_acceleration*. Fija la máxima aceleración de rotación permitida. En nuestro caso fijaremos el valor en 5.0.

- *state_publish_rate*. Fija la tasa de tiempo con la que se publicará el estado. En nuestro caso fijaremos el valor en 0.25.

- *max_translational_velocity*. Fija la máxima velocidad de translación permitida. En nuestro caso fijaremos el valor en 10.0.

- *max_rotational_velocity*. Fija la máxima velocidad rotacional permitida. En nuestro caso fijaremos el valor en 20.0.

Una vez creado nuestro archivo de configuración, realizaremos la carga de los parámetros desde un lanzador. Para ello, en nuestro caso, realizamos la carga desde el lanzador *teleop_keyboard.launch,* ya que la razón de que utilicemos este controlador es para poder transmitir al robot los valores que envía *pr2_teleop*, en ningún otro momento necesitamos utilizar el controlador de la base. Para realizar la carga de los parámetros introducimos la siguiente etiqueta:

```
<rosparam file="$(find pr2_controller_configuration)/pr2_base_controller2.yaml"
command="load" />
```

Con los parámetros ya cargados, el siguiente paso será iniciar el controlador, incluyendo la siguiente etiqueta:

```
<node name="base_controller_spawner" pkg="pr2_controller_manager"
type="spawner" output="screen"  args="base_controller"/>
```

Una vez iniciado nuestro controlador para la base (*base_controller*) y con *pr2_teleop* funcionando, sólo debemos pulsar las teclas que hemos comentado anteriormente para que nuestro robot se mueva.

A continuación se muestran cuatro capturas de pantalla de una simulación realizada. En ella comenzamos realizando un movimiento rotacional del PR2, como puede observarse en la Figura 3.12. Posteriormente, el movimiento será de avance, es decir, positivo en el eje X, como puede observarse en la Figura 3.13. El siguiente movimiento realizado es hacia la derecha del robot, nuestra izquierda como puede apreciarse en la Figura 3.14, pulsando la tecla “D”. Finalmente, terminaremos la simulación realizando un movimiento del PR2 hacia atrás, como puede observarse en la Figura 3.15.

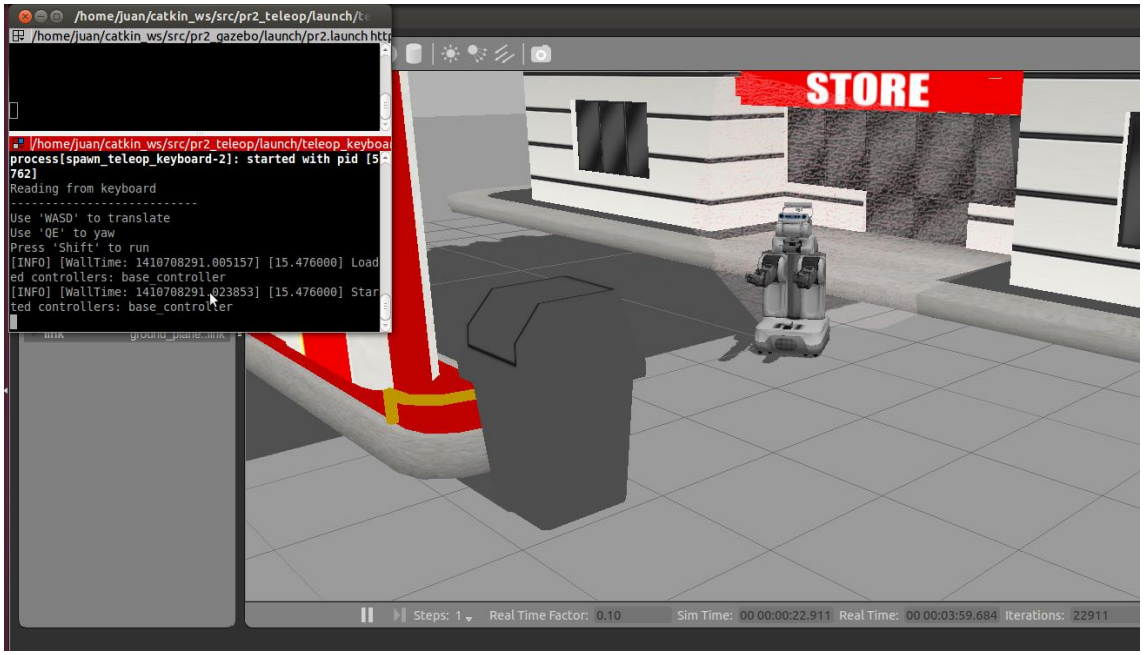

Figura 3.12: Movimiento rotacional 1.

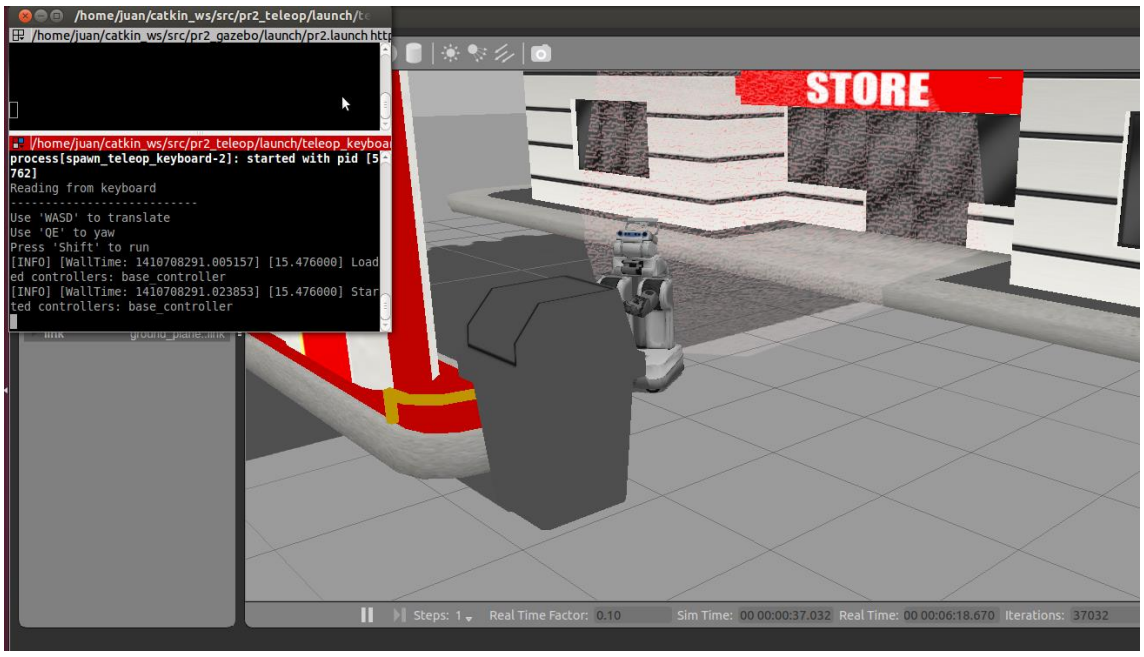

Figura 3.13: Movimiento a la derecha.

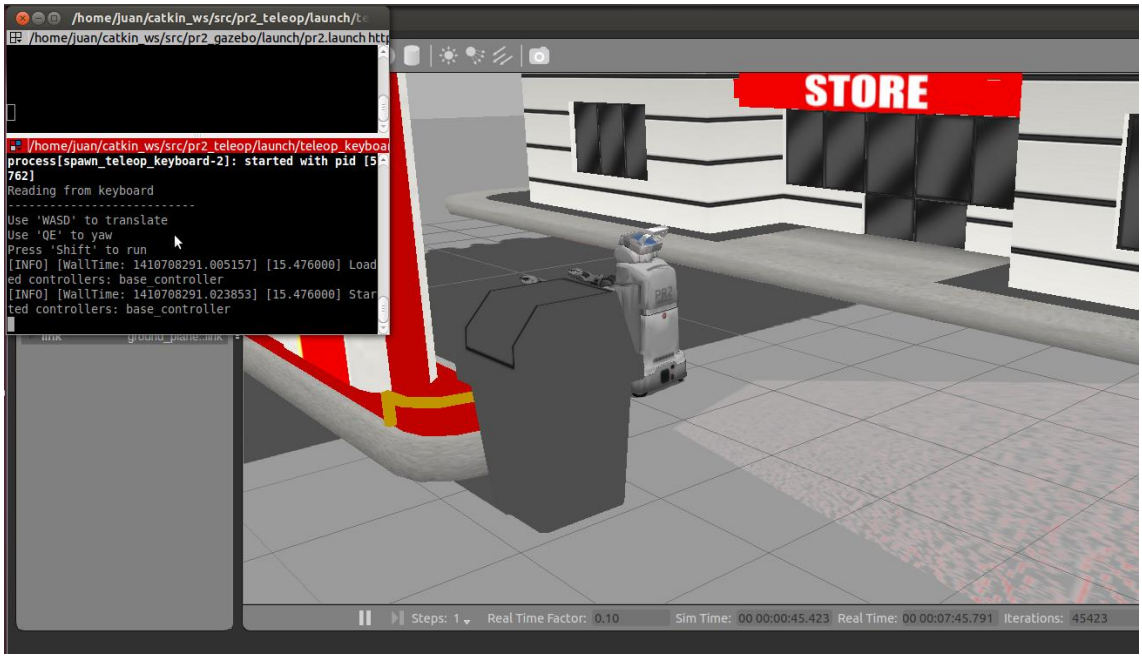

Figura 3.14: Movimiento rotacional 2.

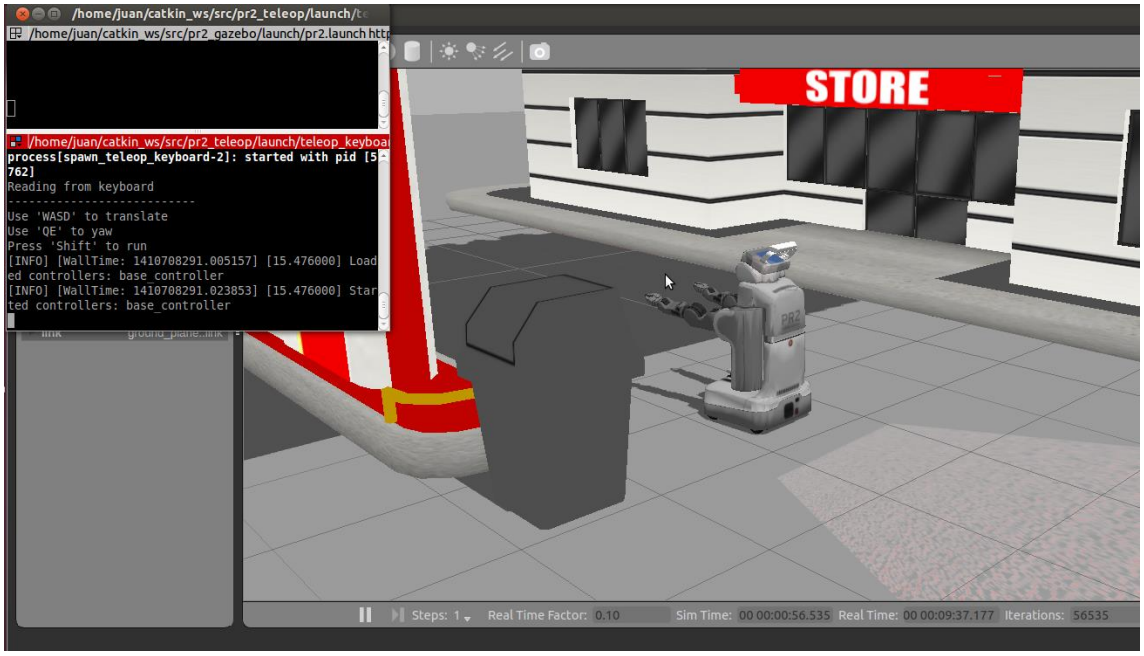

Figura 3.15: Movimiento hacia atrás.

3.3.2.- MOVIMIENTO DE LA CABEZA.

Además del movimiento de la base, también queremos mover la cabeza para que el robot sea capaz de ver, usando de la cámara de su cabeza, lo que tiene a su alrededor.

Para ello crearemos un controlador que usaremos, cargándolo en las dos articulaciones del cuello, para mover la cabeza de un lado a otro y de arriba abajo. Los pasos que seguiremos para crearlo pueden observarse en la Figura 3.16.

Figura 3.16: Pasos para crear un controlador.

Lo primero que haremos será crear un paquete en el que construir el controlador. Nuestro paquete debe depender de *pr2_controller_interface*, que contiene la clase base para todos los controladores; de *pr2_mechanism_model*, para proporcionarle acceso a las articulaciones del robot, y de *pluginlib*, que nos permitirá añadir nuestro controlador como un *plugin* al *Controller Manager*.

Una vez creado el paquete, escribiremos el archivo con las cabeceras. En él crearemos una clase hija de *pr2_controller_interface::Controller*, de tal manera que nuestra clase hija podrá ser ejecutada por el nodo *Controller Manager* en un ciclo del control en tiempo real. Las funciones miembro de la clase *Controller* son las que se muestran en la Figura 3.17, y cuya utilidad es la siguiente:

- La función *init()* es la que se encarga de recibir un objeto *pr2_mechanism_model::RobotState*, que contiene el modelo URDF y da acceso a cada una de las partes de éste. También recibe un objeto ros::NodeHandle asignado para tener acceso al Servidor de Parámetros y poder crear y subscribirse a *topics* y servicios. Esta función se ejecuta fuera del tiempo de ejecución, como puede observarse en la Figura 3.17, al contrario de las siguientes, que lo harán en tiempo real.

- La función *starting()* es llamada una sola vez por Controller Manager, como puede observarse en la Figura 3.17, al inicio del primer ciclo. En ella se deben inicializar todas las variables que vayamos a necesitar, como las posiciones iniciales y el instante de tiempo actual.

- La función *update()* es la función principal del controlador y será ejecutada cada ciclo de control en tiempo real a 1000 Hz, como hemos comentado anteriormente. En ella debe de programarse todo el sistema de control, es decir, los cálculos y acciones de control pertinentes.

- La función *stopping()* es ejecutada una sola vez al finalizar el control, como puede observarse en la Figura 3.17, y es la encargada de detener el controlador.

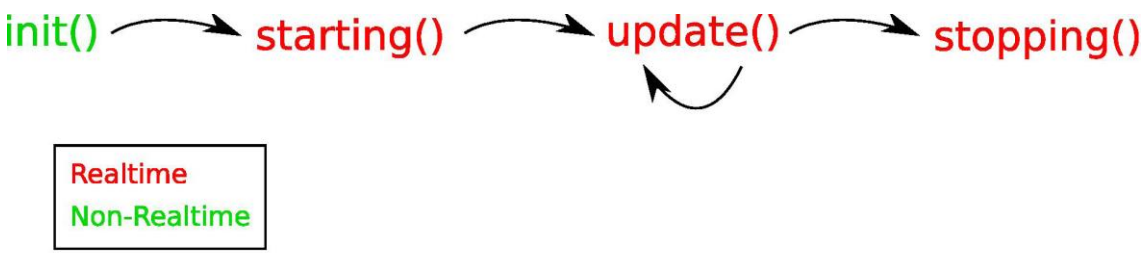

Figura 3.17: Funciones de la clase *Controller*.

El siguiente paso será crear el programa o archivo de alimentación (*.cpp*). En él definimos las funciones miembros de nuestra clase de control definida en el archivo de cabecera, y que será ejecutado por *Controller Manager*. También definiremos las tareas que realizarán las funciones de la clase *Controller*:

- La función *init()* se encarga de recibir el nombre de la articulación a mover y de comprobar que éste sea correcto. También recibe el objeto ros::NodeHandle, comentado anteriormente, e inicializa la amplitud del movimiento que realizará la articulación, que en nuestro caso la fijamos en 1.0.

- La función *starting()* inicializa la variable de la posición de la articulación, recibiendo la posición actual que tiene.

- La función *update()* modifica la posición de la articulación actualizándola, siguiendo la siguiente fórmula:

  nueva_pos = pos_inicial + amplitud * sin(tiempo_ejec)

  De tal manera que la posición al final del ciclo será la suma de la que tenía al inicio del ciclo más un valor que variará entre el de la amplitud y cero.

- La función *stopping()* se encarga de detener el controlador.

Además de estas funciones, en el archivo *.cpp*, también incluiremos una función para crear un servicio, mediante el cual podremos recibir un valor nuevo para la amplitud del movimiento.

Con el programa de nuestro controlador ya escrito, lo que haremos será actualizar el archivo *CMakeList.txt* del paquete para que, al compilar, cree una librería a partir del programa que será ejecutada por *Controller Manager*. Para ello añadiremos al archivo la instrucción:

rosbuild_add_library(my_controller_lib src/my_controller.cpp)

Otro archivo que debemos crear en nuestro paquete es el llamado *controller_plugins.xml*. El motivo de generarlo es para que se cree un *plugin* hacia la librería creada con la instrucción anterior y, de esta manera, *Controller Manager* pueda tener acceso y pueda ejecutarla.

Una vez hecho esto, debemos añadir al archivo *.cpp* las instrucciones para registrar el controlador en *pluginlib*, y de esta manera poder ejecutarlo desde el *Controller Manager*.

Además, también deberemos añadir las instrucciones, mediante la etiqueta *<export>*, al archivo *manifest.xml* para exportar el archivo *controller_plugins.*xml y que sea accesible por *pr2_controller_interface*.

Para terminar de escribir nuestro controlador, lo único que nos queda por hacer es crear el archivo *SetAmplitude.srv*, con la definición del tipo de mensaje que enviará y recibirá el servicio creado para modificar la amplitud. También deberemos añadir, al archivo *CMakeList.txt*, la instrucción para que genere las librerías con las cabeceras del servicio para modificar la amplitud. La instrucción que añadimos es:

```
rosbuild_gensrv()
```

Con nuestro controlador escrito, lo que debemos hacer es crear dos controladores, uno para cada movimiento que queremos realizar. Para ello introduciremos la misma instrucción que en el caso del controlador de la base, una vez para cada controlador:

```
$ rosparam set head_pan_controller/type MyControllerPlugin

$ rosparam set head_tilt_controller/type MyControllerPlugin
```

A partir de ahora, el proceso a seguir será el mismo que con el controlador de la base, crearemos los archivos *head_pan_controller.yaml* y *head_tilt_controller.yaml*, uno para cada controlador. En ellos definiremos el tipo de controlador que son, los coeficientes PIDs y el nombre de la articulación sobre la que actúan (*head_pan_joint*, para el movimiento de izquierda a derecha, y *head_tilt_joint,* para el de arriba abajo).

A la hora de cargar e iniciar los controladores, lo hacemos mediante un archivo *.launch*, en el que introduciremos las mismas instrucciones, salvo que modificando el nombre de los controladores y los archivos a cargar, que añadíamos al archivo *teleop_keyboar.launch* en el caso del controlador de la base.

A continuación, se muestran dos capturas de pantalla de una simulación realizada. En ella se aprecia como en la Figura 3.18 comienza a girar la cabeza hacia la izquierda y hacia abajo, para llegar hasta la posición que tiene en la Figura 3.19.

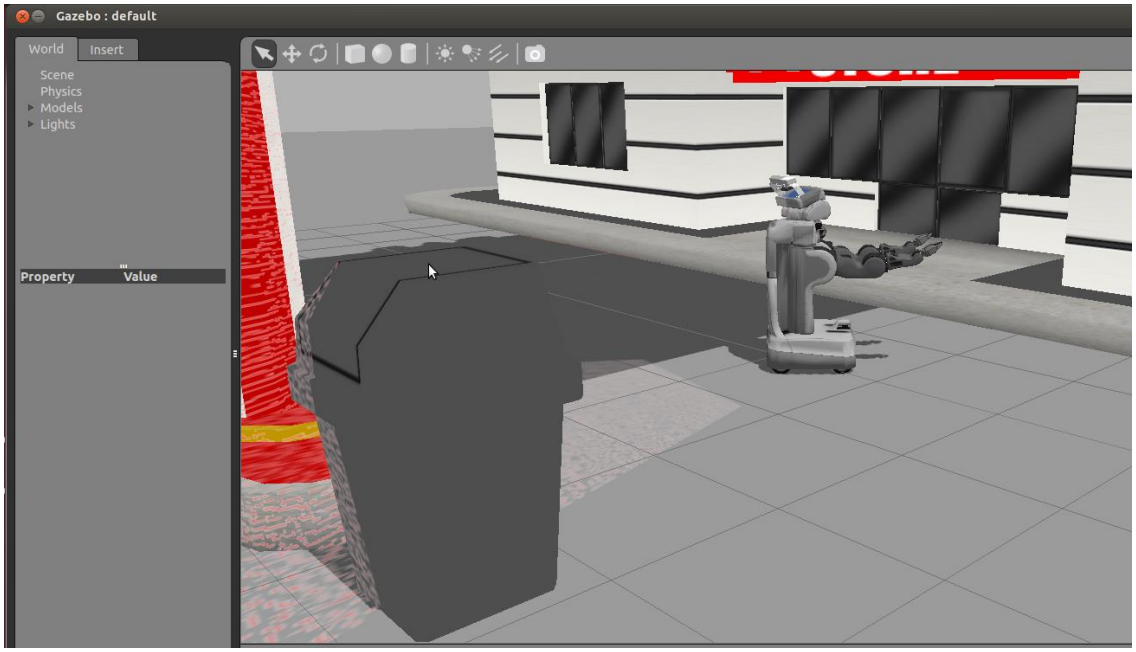

Figura 3.18: Posición inicial de la cabeza.

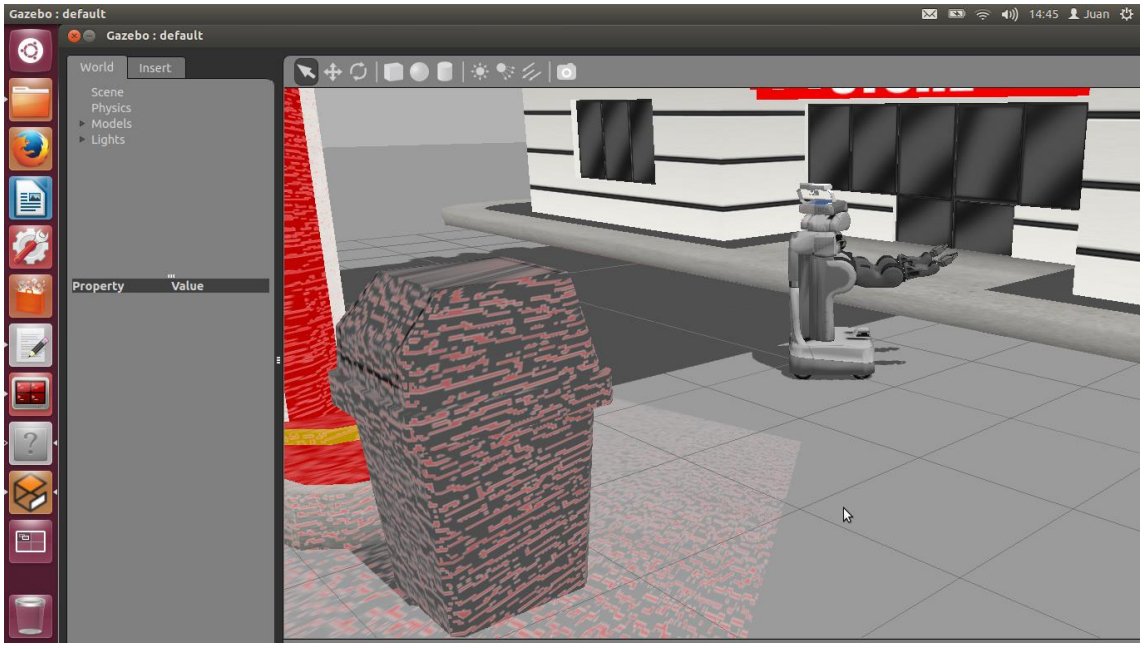

Figura 3.19: Posición final de la cabeza.

3.4.- VISUALIZACIÓN EN RVIZ.

Como hemos comentado anteriormente, Rviz es una herramienta de visualización 3D perteneciente a ROS y, por tanto, se instalará automáticamente cuando hagamos lo propio con éste.

El objetivo que buscamos conseguir con Rviz es visualizar los movimientos de las articulaciones del PR2 desde él y, también, observar lo que detectan sus sensores.

3.4.1.- VISUALIZACIÓN DEL ROBOT.

Lo primero que debemos hacer, para visualizar nuestro PR2, es ejecutar Rviz y cargar el URDF del robot que hemos almacenado en el Servidor de Parámetros, bajo el nombre robot_description, en el momento en que lanzamos Gazebo. Para ello, ejecutaremos un archivo al que hemos llamado *mypr2_rviz.launch* en el que tendremos la siguiente instrucción:

```
<nodename="rviz" pkg="rviz" type="rviz" args="-d $(find
mypr2_description)/mypr2.rviz"/>
```

Donde, como se puede observar, ejecutamos un nodo al que llamamos "*rviz*" del tipo rviz y al que le pasamos el archivo de configuración del visualizador *mypr2.rviz,* para que lo cargue. La forma de hacerlo es con el comando "*-d*", cuya labor es la de cargar este tipo de archivos. Si se desea visualizar otros comandos que se pueden pasar a la hora de ejecutar Rviz, deberemos escribir en nuestra terminal:

```
$ rosrun rviz rviz --help
```

Como ya hemos comentado, los archivos de extensión *.rviz* tienen el objetivo de guardar una configuración de Rviz. Este tipo de archivos siguen el formato *YAML*, que es el mismo que siguen los *.yaml*, que usábamos para los controladores. Son utilizados

desde la versión de ROS Groovy, ya que anteriormente se utilizaban los de extensión *.vcg*, escritos de una manera similar a los de extensión *INI*.

Gran parte de los parámetros, como pueden ser los relativos a las herramientas de rviz o los puntos de vista del visualizador, no van a ser relevantes para nosotros a la hora de visualizar el robot en Rviz, es por ello que estos valores son incluidos en el archivo con su valor por defecto. En nuestro caso, los valores que realmente nos importan de la configuración, son los relativos a los *displays* que se visualizarán. A continuación, comentaremos los que hemos configurado al arranque desde el archivo *myp2.rviz*:

- *Global Options*. En él se fijan los valores genéricos de la visualización, como son el color del fondo de la pantalla de visualización (B*ackground Colour*), el tipo al que son convertidos todos los *frames* antes de ser mostrados (*Fixed Frame*) y el número de *frames* que se mostrarán por minuto (*Frame Rate*). Los valores relevantes para nosotros son el de *Fixed Frame*, en el que configuraremos el *base_footprint*, y el de *Frame Rate*, que fijaremos en 30 *frames* por segundo.

- *Global Status.* Este *display* nos muestra el estado en que se encuentra la visualización, nos indica si se están recibiendo bien los *frames*. En él no es necesario configurar nada, pero nos será útil para confirmar que nuestro visualizador está recibiendo los datos correctamente.

- *Grid*. Muestra una cuadrícula de referencia en el visualizador. Los valores que configuraremos será el tamaño de cada cuadro, que será de 1 metro de lado; el valor del grado de transparencia (*Alpha*), siendo 0 totalmente transparente y 1 opaco, que fijaremos en "0.5" y, por último, el plano sobre el que se mostrará, que será el XY. Para el resto de parámetros se conservarán los valores por defecto.

- *RobotModel*. Con este *display* visualizamos el robot en Rviz. El valor más importante a configurar es el *Robot Description*, en el que debemos introducir el nombre del parámetro en el que almacenamos nuestro robot en URDF al arrancar Gazebo. Como hemos visto anteriormente, a este parámetro lo hemos llamado *robot_description*.
  Otro valor que debemos configurar es el de *Visual Enabled*, que fijaremos en "*true*" para que nuestro robot sea visible. También fijaremos el intervalo de

actualización (*Update Interval*) en “0”, lo cual significa que el estado de nuestro robot se actualizará en cada ciclo de ejecución.
Por útlimo, fijaremos el valor de *Alpha* general en “1”, para que nuestro robot sea opaco, y habilitaremos todas las uniones del robot para que sean visibles y tengan el valor “1” para *Alpha*, en cada una de ellas.

Una vez configurado nuestro menú de *displays*, cada vez que arranquemos Rviz con la configuración *mypr2.rviz*, aparecerá como se muestra en la Figura 3.20.

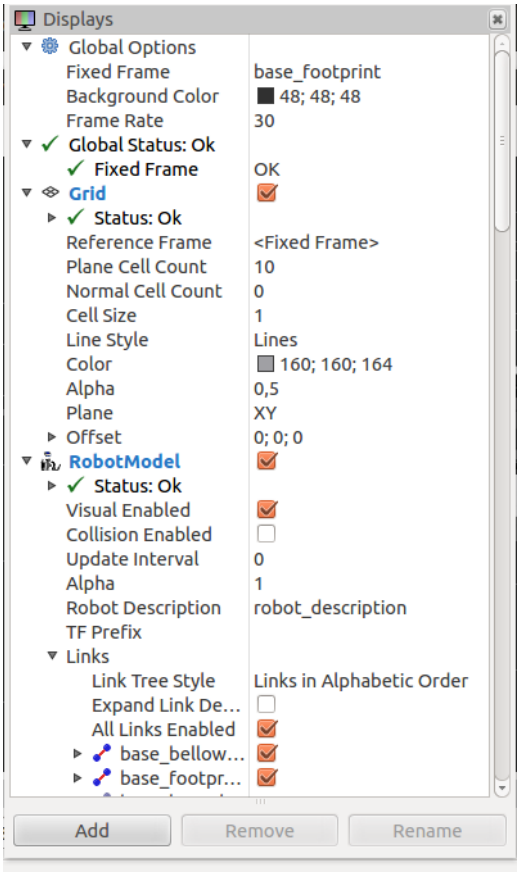

Figura 3.20: Configuración de displays.

Con esta configuración, observaremos nuestro PR2 en el visualizador, como puede observarse en la Figura 3.21, en la misma posición con la que se encuentra en la simulación. Esto se debe a que Gazebo publica el estado en que se encuentran las articulaciones del robot en el *topic /joint_states* y dicha información, tras ser transformada, acabará llegando a Rviz.

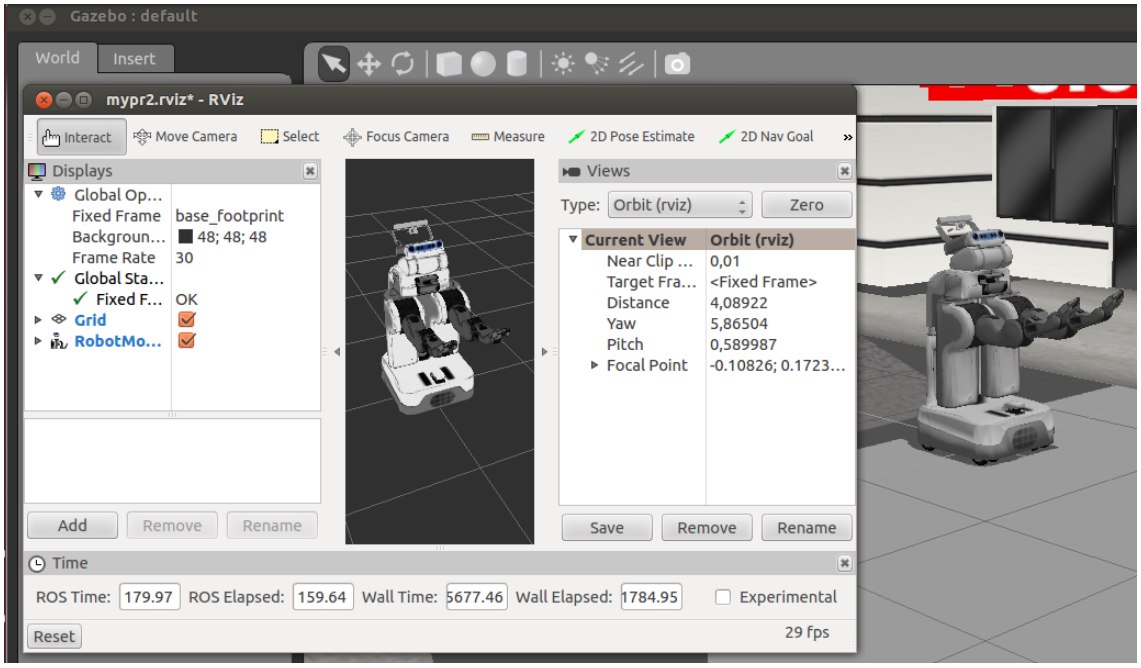

Figura 3.21: Visualización de PR2 en Rviz.

La manera en que transformaremos la información publicada por Gazebo será mediante el paquete *robot_state_publisher*, con el que convertiremos ésta a *tf* y será publicada en el *topic* */tf*, para que pueda ser leída por cualquier programa que se subscriba a él. En nuestro caso, Rviz leerá de éste, tal y como se observa en la Figura 3.22.

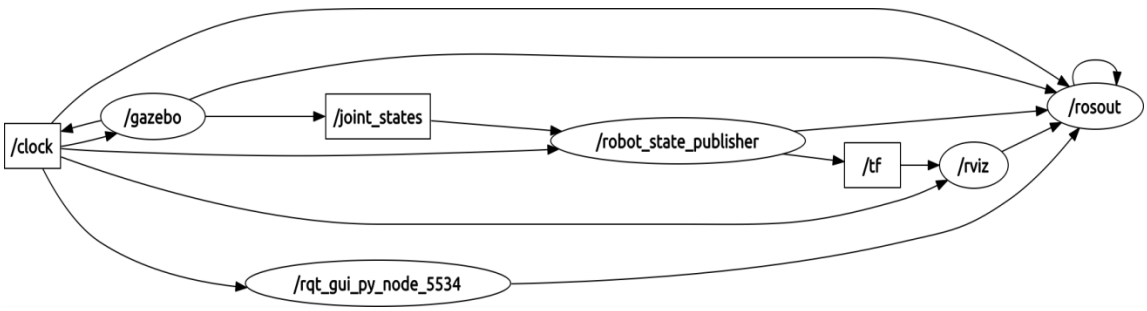

Figura 3.22: Transmisión del estado del robot.

La forma de obtener la posición del robot mediante *tf* consiste en definir la posición de una articulación en relación a otra. De esta manera, al final tendremos un "árbol" de sistemas de referencias en el que cada uno dependerá de otro para definir la posición del anterior. Normalmente, estos sistemas estarán en la base de las articulaciones, de forma que será bastante sencillo obtener la posición de la punta de éstas y, por tanto, la posición del origen de coordenadas de la siguiente articulación, si es que procede. En la Figura 3.23 puede observarse un ejemplo del "árbol" de sistemas de referencias para el PR2.

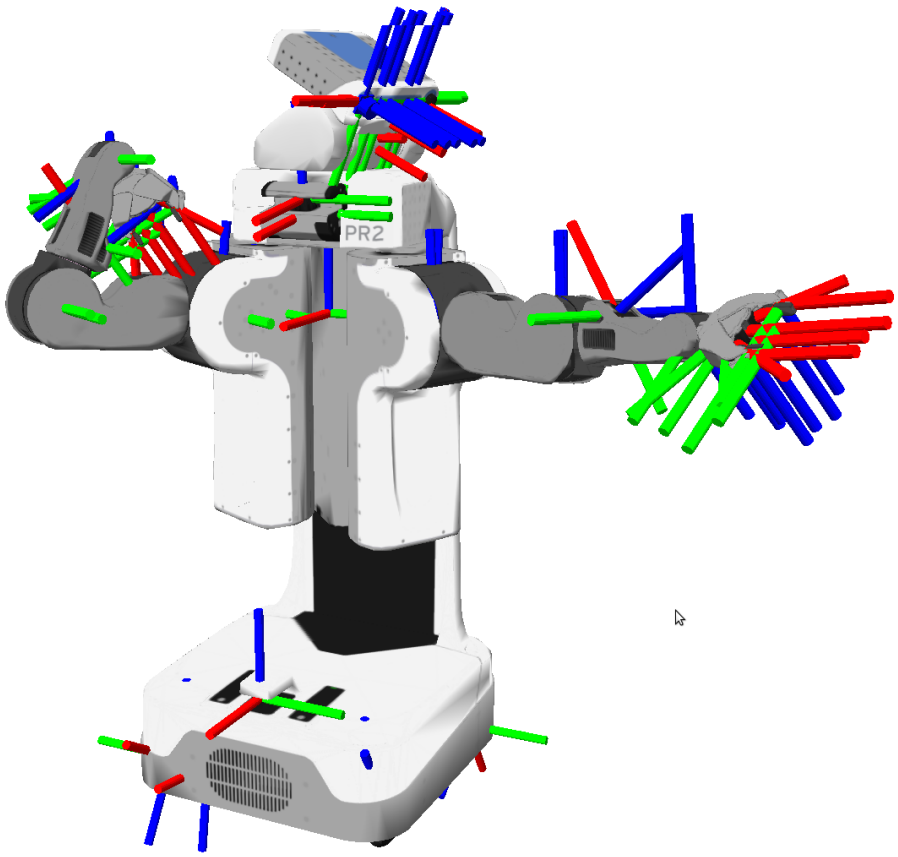

Figura 3.23: Sistemas de referencia del PR2.

3.4.2.- VISUALIZACIÓN DE DATOS DE SENSORES.

Antes de intentar visualizar los sensores en Rviz, lo primero que debemos hacer es comprobar que nuestro archivo URDF contiene los *plugins* de Gazebo de éstos, para que se encuentren en la simulación y, de esta manera, publiquen en los *topics* correspondientes, de los que leerá el visualizador.

Los *plugins* de Gazebo son complementos que se añaden a los modelos proporcionándoles funciones y/o sensores. En general, salvo los que son creados para un modelo en específico, todos los *plugins* se encuentran en el paquete g*azebo_plugins*, en forma de librería (*.so*). La forma de añadirlos a la descripción URDF del robot es entre dos etiquetas que identifican que se refieren al simulador: <*gazebo*>. Dentro de estas etiquetas nos encontraremos otras, <*controller:tipo plugin*>, mediante las cuales se le asignará un nombre, se especifica la librería a la que pertenecen y se encuentran los valores posibles para los parámetros del *plugin.*

En la descripción URDF del robot que utilizamos, la proporcionada por *Willow Garage*, por defecto deben estar todos los *plugins* de los sensores. Sin embargo, es recomendable comprobar que se encuentren añadidos para evitar futuros problemas.

Una vez que hemos comprobado esto, debemos añadir a nuestro visualizador un *display* para mostrar los datos del sensor que deseemos. Para ello, pulsaremos en el botón “Add” del menú de *displays*. En ese momento se nos desplegará una ventana en la que elegiremos el tipo que deseamos añadir. En la Figura 3.24, se muestra un ejemplo para una cámara. De esta forma, ya tenemos añadido el *display* a nuestro visualizador y, únicamente tendremos que configurarlo para obtener los datos del sensor elegido.

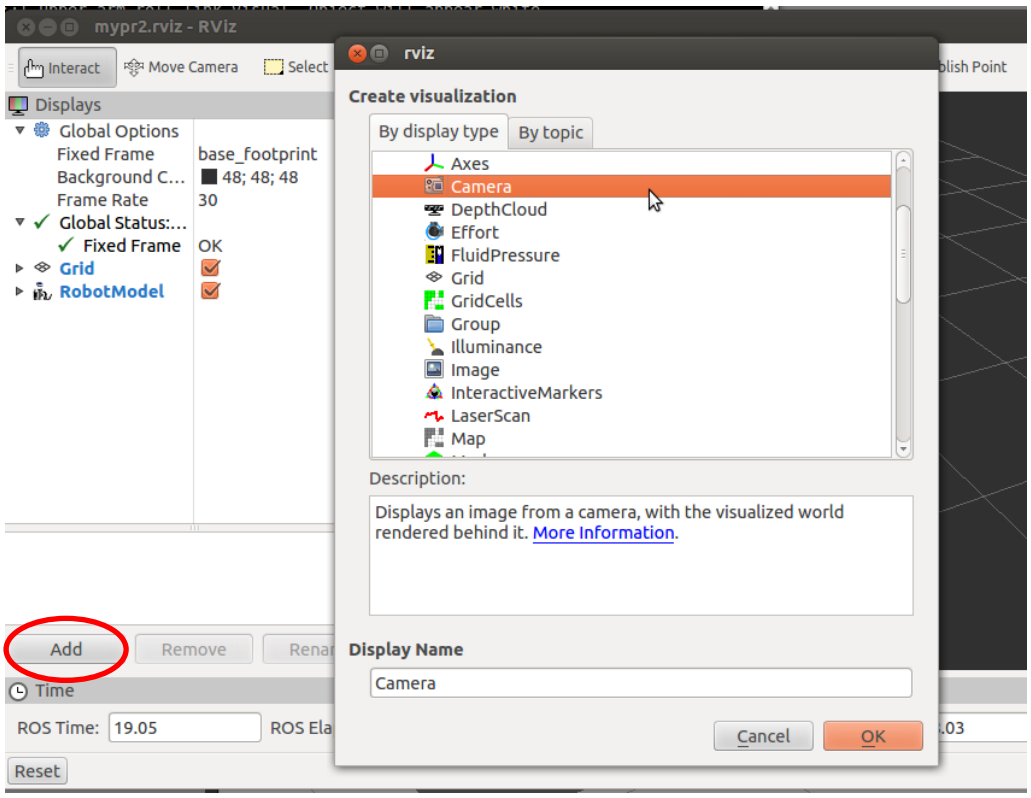

Figura 3.24: Añadir un *display* a Rviz.

Para configurarlo, utilizaremos el menú que nos proporciona Rviz. Lo primero que debemos escoger es el *topic* del que el visualizador leerá los mensajes con la información, éste será en el que publique el *plugin* del sensor que deseemos obtener información. La forma de hacerlo será pulsando sobre el que nos interese del desplegable para la opción *topic* del *display* que hemos añadido. En dicho desplegable, únicamente nos aparecerán los *topics* que contienen mensajes del tipo que podremos visualizar en ese *display,* y a los que Rviz pueda tener acceso.

El resto de parámetros dependerán del tipo de *display* y de las necesidades que tengamos para configurarlas de una manera u otra.

A continuación, en la Figura 3.25 se muestra una captura de una simulación, llevada a cabo en Gazebo, de la que se ha visualizada la imagen RGB obtenida por la cámara del Kinect, utilizando el *topic /head_mount_kinect/rgb/image_raw*; la nube de puntos obtenida por éste, utilizando el *topic /head_mount_kinect/depth/image_raw*, y los datos obtenidos por el escáner láser situado en la base del PR2, mediante el *topic /base_scan*. En la Figura 3.26 se observan, los datos obtenidos en el visualizador.

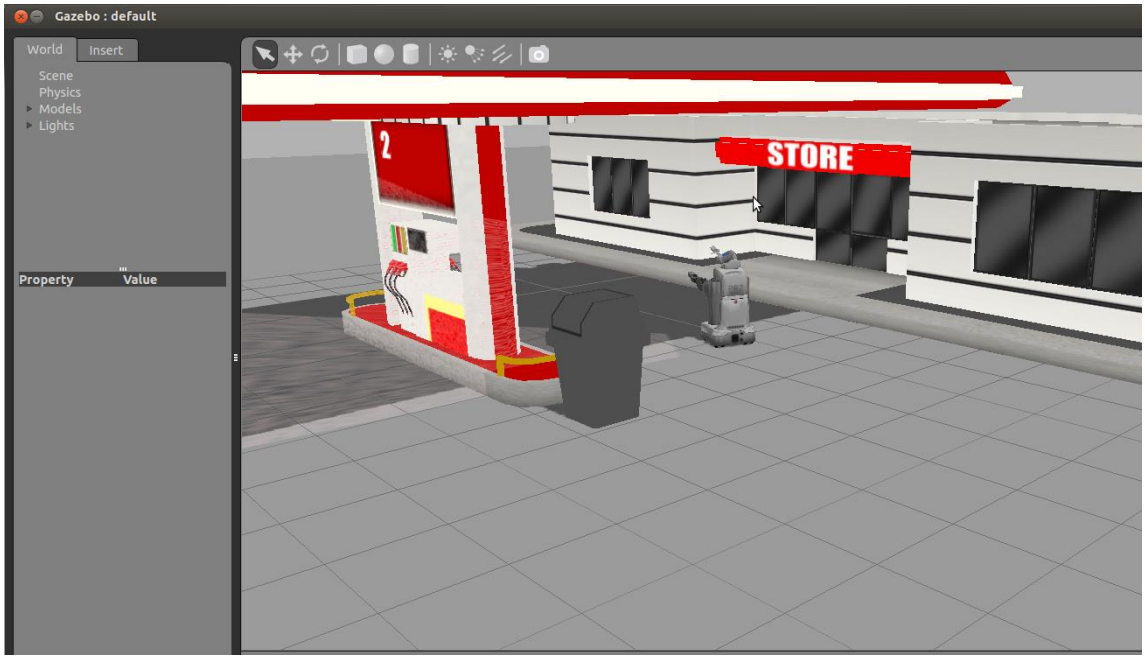

Figura 3.25: Simulación realizada en Gazebo.

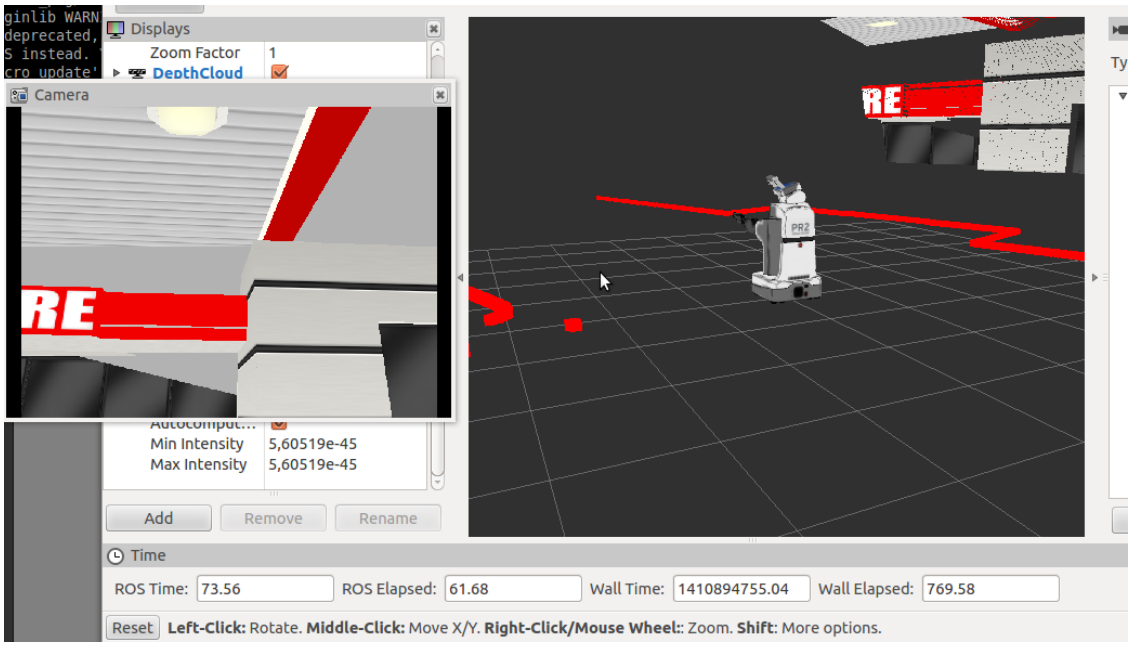

Figura 3.26: Visualización de los datos en Rviz.

4.- CONCLUSIONES Y TRABAJO FUTURO.

El principal objetivo de este trabajo era ser capaces de realizar la integración del simulador Gazebo con el visualizador Rviz y, a la vez, ser capaces de interactuar con el robot simulado desde ROS.

Cuando se comenzó a elaborar este trabajo se desconocía el funcionamiento del Sistema Operativo Robótico (ROS) y, por tanto, de Rviz. Teniendo en cuenta que todo el proceso de elaboración del trabajo ha sido bajo este "sistema operativo", se considera primordial aprender a manejarlo. Después de una labor de investigación y haber trabajado con él todo este tiempo, se consideran adquiridos los conocimientos necesarios acerca de su funcionamiento y de la forma de intercomunicarse de sus procesos, además de haber aprendido a utilizar un gran número de herramientas que nos ofrece.

El siguiente objetivo consistía en familiarizarse con el simulador Gazebo, pues tampoco se conocía su funcionamiento a fecha de inicio del presente trabajo. De igual manera que con ROS, tras la labor de investigación llevada a cabo y el trabajo desarrollado, se considera superado este objetivo. Siendo capaces de simular un entorno y un robot en él correctamente, y comprendiendo el funcionamiento del programa.

Por último, debíamos ser capaces de integrar todas estas herramientas para que interactuasen entre ellas. Como ha quedado demostrado, este objetivo también se ha logrado, siendo capaces de mover el robot en la simulación pasando mensajes desde nodos de ROS y, de igual manera, siendo capaces de transmitir la información de la simulación a través de los *topics* de éste.

Como en todo proyecto de desarrollo, no se considera finalizado el trabajo con Gazebo y Rviz, abriéndose varias vías de trabajo a partir de la elaboración de éste. Una de ellas sería la de elaborar controladores más robustos que nos permitieran mover un número mayor de articulaciones y hacerlo de una manera más intuitiva para el usuario.

También sería interesante incluir un sensor de proximidad al robot simulado, de tal manera que fuera capaz de advertir al usuario en caso de una colisión inminente. Así como ser capaces de aprovechar los mensajes transmitidos por cualquier otro sensor ya presente en el robot, o que se pudiera implementar, para mejorar la labor de los controladores.

5.- PRESUPUESTO

A continuación, se detallará el cálculo del valor aproximado de la realización de este proyecto.

Código	Unidad	Descripción	Cantidad	Precio	Precio total
01		**CAPÍTULO 1: MANO DE OBRA**			
01.01	H	Planteamiento del proyecto por parte de un ingeniero.	90	40,00 €	3.600,00 €
01.02	H	Desarrollo del proyecto por parte de un ingeniero.	120	40,00 €	4.800,00 €
01.03	H	Realización de pruebas de funcionamiento del proyecto por parte de un ingeniero.	15	40,00 €	600,00 €
01.03	H	Realización de la memoria del proyecto por parte de un ingeniero.	90	40,00 €	3.600,00 €
				TOTAL CAPÍTULO 1	12.600,00 €
02		**CAPÍTULO 2: MATERIAL**			
02.01	UD	Ordenador con sistema operativo Linux.	1	750,00 €	750,00 €
02.02	UD	Licencias de software.	1	0,00 €	0,00 €
				TOTAL CAPÍTULO 2	750,00 €
				TOTAL	13.350,00 €
				IVA 21%	2.803,50 €
				TOTAL CON IVA	16.153,50 €

A la hora de realizar el cálculo de la mano de obra, se ha dividido el trabajo en distintas áreas, con el siguiente criterio:

- Planteamiento. Incluye las tareas de familiarización con ROS y el diseño general del proyecto.

- Desarrollo. Incluye la simulación en Gazebo, la elaboración de los controladores y la configuración del visualizador.

- Pruebas. Incluye las simulaciones realizadas para comprobar el correcto funcionamiento de todo el conjunto.

- Memoria. Incluye tanto la redacción, como la revisión de ésta.

BIBLIOGRAFÍA

[1] *Wikipedia*. Robótica, 2014. [Consulta: 20-08-2014]. Disponible en: http://es.wikipedia.org/wiki/Robótica

[2] Asimov, Isaac. *Yo, robot*, 2009. [Consulta 20-08-2014]

[3] *Página oficial de ROS*, 2014. [Consulta: 22-08-2014]. Disponible en: http://www.ros.org/

[4] Y. Ng, Andrew; Gould, Stephen;Quigley, Morgan;Saxena, Ashutosh and Berger, Eric. STAIR: The STanford Artificial Intelligence Robot Project. *Snowbird*, 2008. [Consulta 22-08-2014].

[5] Bogdan Rusu, Radu. *Diapositivas CoTeSys-ROS-Schoo*. Willow Garage, 2010 [en línea]. [Consulta 20-08-2014]. Disponible en: http://www.ros.org/wiki/Events/CoTeSys-ROS-School?action=AttachFile&do=view&target=ros_overview.pdf.

[6] Quigley, Morgan; Gerkey, Brian; Conley, Ken; Faust, Josh; Foote, Tully; Leibs, Jeremy; Berger, Eric; Wheeler, Rob and Y. Ng, Andrew. *Open-source software workshop of the International Conference on Robotics and Automation (ICRA)* ROS: an open-source Robot Operating System, 2009 [en línea]. [Consulta: 26-08-2014]. Disponible en: http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf

[7] *Wiki de ROS.* Rviz, 2014. [Consulta: 27-08-2014]. Disponible en: http://wiki.ros.org/rviz

[8] *Página oficial de Gazebo*, 2014. [Consulta: 29-08-2014]. Disponible en: http://gazebosim.org/

[9] *Open Source Robotics Foundation*, 2014. [Consulta: 29-08-2014]. Disponible en: http://www.osrfoundation.org/

[10] *Willow Garage.* PR2 Overview, 2014. [Consulta 27-08-2014]. Disponible en: http://www.willowgarage.com/pages/pr2/overview

[11] *Wiki de ROS*, 2013. [Consulta: 05-09-2014]. Disponible en: http://wiki.ros.org/

[12] Ventura, Rodrigo. *Diapositivas Introduction to ROS.* Willow Garage, 2011 [en línea]. [Consulta 28-08-2014]. Disponible en: http://mediawiki.isr.ist.utl.pt/images/3/3d/

[13] Goebel, R. Patrick. *ROS By Example Hydro*, 2014. [Consulta 07-09-2014].

[14] *Willow Garage*. ROS Cheat Sheet – Hydro v 1.01, 2014. [Consulta 07-09-2014]. Disponible en: http://www.clearpathrobotics.com/wp-content/uploads/2014/01/ROS-Cheat-Sheet-v1.01.pdf