Manual de ROS

Manual de ROS

Sitio:	Moodle UA 2017-18
Curso:	2017-18_VISIÓN ARTIFICIAL Y ROBÓTICA_34030
Libro:	Manual de ROS
Imprimido por:	Gonzalo Molina Gallego
Día:	martes, 13 de marzo de 2018, 18:03

Tabla de contenidos

1. Introducción a ROS

ROS (Robot Operating System) proporciona bibliotecas y herramientas para ayudar a los desarrolladores de software crear aplicaciones robóticas. Proporciona una abstracción del hardware, de los controladores de dispositivos, las bibliotecas, visualizadores, paso de mensajes, gestión de paquetes y mucho más.

Este conjunto de herramientas y librerías nos proporciona:

- **Mecanismo de comunicación entre programas:**

Es un estándar para comunicar entre sí diferente programas de un mismo sistema, ya sea en el mismo computador o en varios computadores. La computación distribuida es un recurso muy común donde pequeñas partes de un robot trabajan por separado para conseguir un objetivo común.

- **Reusabilidad de código:**

Los paquetes estándar proporcionados en las distribuciones de ROS implementan muchos de los algoritmos comúnmente usado en robótica que ya han sido depurados y usados de forma estable. Además, el modelo de comunicación y gestión de mensajes de ROS se ha convertido en estándar y muchas de las plataformas robóticas ya implementan interfaces para ser usadas directamente desde ROS. Además de los paquetes estándar, existen muchos otros en la comunidad que implementan interfaces para sus sistemas (robots, sensores, librerías...)

- **Testeado rápido:**

Debido al diseño de comunicación por paso de mensajes, ROS nos permite simular muchos de los dispositivos con los que nuestro sistema trabajará de forma que podemos aislar la funcionalidad de nuestro sistema del código de comunicación entre las diferentes partes del sistema, sensores y actuadores. Uno de los aspectos claves al desarrollar una aplicación es la capacidad de repetir los experimentos y poder simular los sensores y actuadores que nos permite crear conjuntos de prueba.

1.1. Instalación de ROS

En este curso vamos a usar la version de ROS llamada hydro. Haremos uso de las integraciones de las librerías de opencv y Point Cloud Library para tratar tanto con imágenes de color como con nubes de puntos 3D.

Se recomienda a los alumnos instalar ROS en sus portátiles/PCs con el fin de obtener pleno acceso a todos los paquetes de ROS ya que en las máquinas de los laboratorios solo hemos instalado los paquetes básicos de ROS y habrá cosas que no se podrán usar al no tener permisos de superusuario.

Para la instalación se recomienda seguir los pasos indicados en:

http://wiki.ros.org/hydro/Installation

1.2. Conceptos

Tópico (topic):

Son canales de información entre los nodos. Un nodo puede emitir o suscribirse a un tópico. Por ejemplo, stage (simulador de robots) emite un tópico que es la odometría del robot. Cualquier nodos se puede suscribir. El nodo que emite no controla quién está suscrito. La información es, por tanto, unidireccional (asíncrona). Si lo que queremos es una comunicación síncrona (petición/respuesta) debemos usar `servicios`.
Un tópico no es más que un mensaje que se envía. Podemos usar distintos tipos de clases de mensajes.

Paquete (package):

El software en ROS está organizado en paquetes. Un paquete puede contener un nodo, una librería, conjunto de datos, o cualquier cosa que pueda constituir un módulo. Los paquetes pueden organizarse en pilas (stacks).

Nodo (node):

Un nodo es un proceso que realiza algún tipo de computación en el sistema. Los nodos se combinan dentro de un grafo, compartiendo información entre ellos, para crear ejecuciones complejas. Un nodo puede controlar un sensor láser, otro los motores de un robot y otro la construcción de mapas.

Pila (stack):

Conjunto de nodos que juntos proporcionan alguna funcionalidad (por ejemplo, la pila de navegación).

1.3. Comandos de ROS

- **roscd**: cambia a un directorio de paquete o pila (ej. *roscd stage*)
- **roscore**: ejecuta todo lo necesario para que dar soporte de ejecución al sistema completo de ROS. Siempre tiene que estar ejecutándose para permitir que se comuniquen los nodos. Permite ejecutarse en un determinado puerto (ej. *roscore* o *roscore -p 1234*)
- **roscreate-pkg**: crea e inicializa un paquete. Se tiene que ejecutar desde uno de los directorios válidos para que contengan paquetes. El formato de ejecución es: roscreate-pkg paquete [depen1 ...] donde depen1 es una dependencia. Por ejemplo, si el paquete que estamos creando va a usar los mensajes estándar y va a usar código c++, debemos indicar las dependencias std_msgs y roscpp.
- **rosnode**: nos proporciona información sobre un nodo. Disponemos de las siguientes opciones:
  - rosnode info nodo (muestra información sobre el nodo)
  - rosnode kill nodo (mata ese proceso)
  - rosnode list (muestra los nodos ejecutándose)
  - rosnode machine maquina (muestra los nodos que se están ejecutando en la máquina).
  - rosnode ping nodo (comprueba la conectividad del nodo).
- **rosrun**: permite ejecutar cualquier aplicación de un paquete sin necesidad de cambiar a su directorio. Podemos pasarle parámetros con _my_param:=value (ej. *rosrun stage stageros*) stage es el paquete y stageros es la aplicación que ejecutamos.
- **rostopic**: permite obtener información sobre un tópico.
  - rostopic bw (muestra el ancho de banda consumido por un tópico)
  - rostopic echo (imprime datos del tópico por la salida estándar)
  - rostopic find (encuentra un tópico)
  - rostopic info (imprime información de un tópico)
  - rostopic list (imprime información sobre los tópicos activos)
  - rostopic pub (publica datos a un tópico activo)
  - rostopic type (imprime el tipo de información de un tópico)
- **roswtf**: permite chequear si algo va mal. Ejecutamos roscd y después roswtf.

1.4. rviz

rviz es un visualizador de datos muy completo y útil. Lo mejor para aprender su funcionamiento es usarlo, pero hay varias cosas a tener en cuenta:

Frames de coordenadas

Hay dos frames de coordenadas importantes:

1. El `fixed frame`. Es el sistema de coordenadas que tomaremos como referencia para mostrar todos lo datos. Aparece en la parte izquierda en Global options. Debemos asignarlo a un punto estático como el mundo o el mapa. Caso de no tenerlo lo podemos asignar al de odometría. No debemos asignarlo a un frame en movimiento, como la base de un robot.

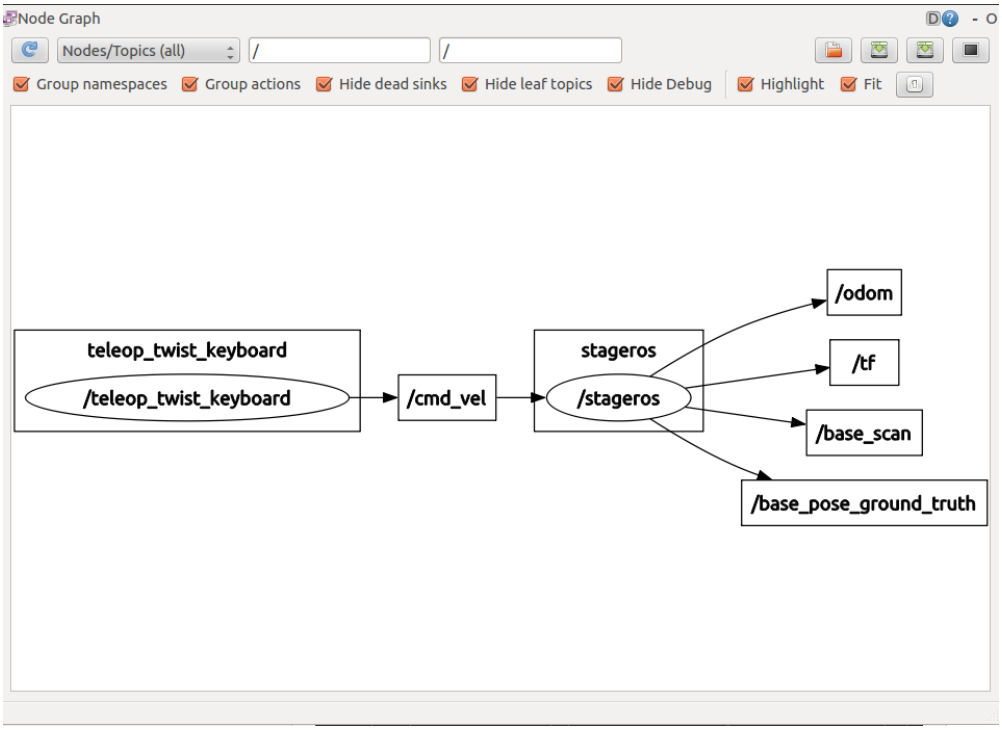
2. El `target frame`. Es el sistema de coordenadas que asignamos a la vista. Este sí que se puede poner a la base del robot, por ejemplo. Podemos tener distintas vistas y así ir cambiando de una a otra en función de lo que queramos ver.

Modo de operación

Existen diversos modos de operación. Se pueden seleccionar en la parte superior. Por defecto está en interactuar y podemos también mover la cámara, seleccionar elementos (y ver sus valores, por ejemplo coordenadas de puntos), etc.

1.5. rqt_graph

Esta aplicación se ejecuta directamente, sin necesidad de llamar a rosrun. Visualiza los nodos que se están ejecutando en este momento y el paso de tópicos entre ellos. Sirve para visualizar qué está pasando con nuestro programa. Por ejemplo, en la siguiente imagen podemos ver que tenemos un nodo, `stageros`, que publica cuatro tópicos y que escucha uno de ellos el cual es publicado por otro nodo, `teleop_twist_keyboard`



2. Probando un simulador 2D

Vamos a probar el simulador Stage. Stage es un simulador de robots móviles en 2D que podemos utilizar si queremos hacer pruebas iniciales de nuestros algoritmos o bien no disponemos de un robot real. Para nuestro código será transparente el que estemos trabajando con un robot real o con Stage. Stage forma parte de otro controlador/simulador llamado Player/Stage. Stage ha sido incluido en ROS por compatibilidad.

Para empezar a usar ROS, primero debemos ejecutar el comando:

```
roscore
```

en una terminal. Este comando da soporte a todos los mecanismos necesarios para permitir la comunicación entre los distintos nodos de ROS.

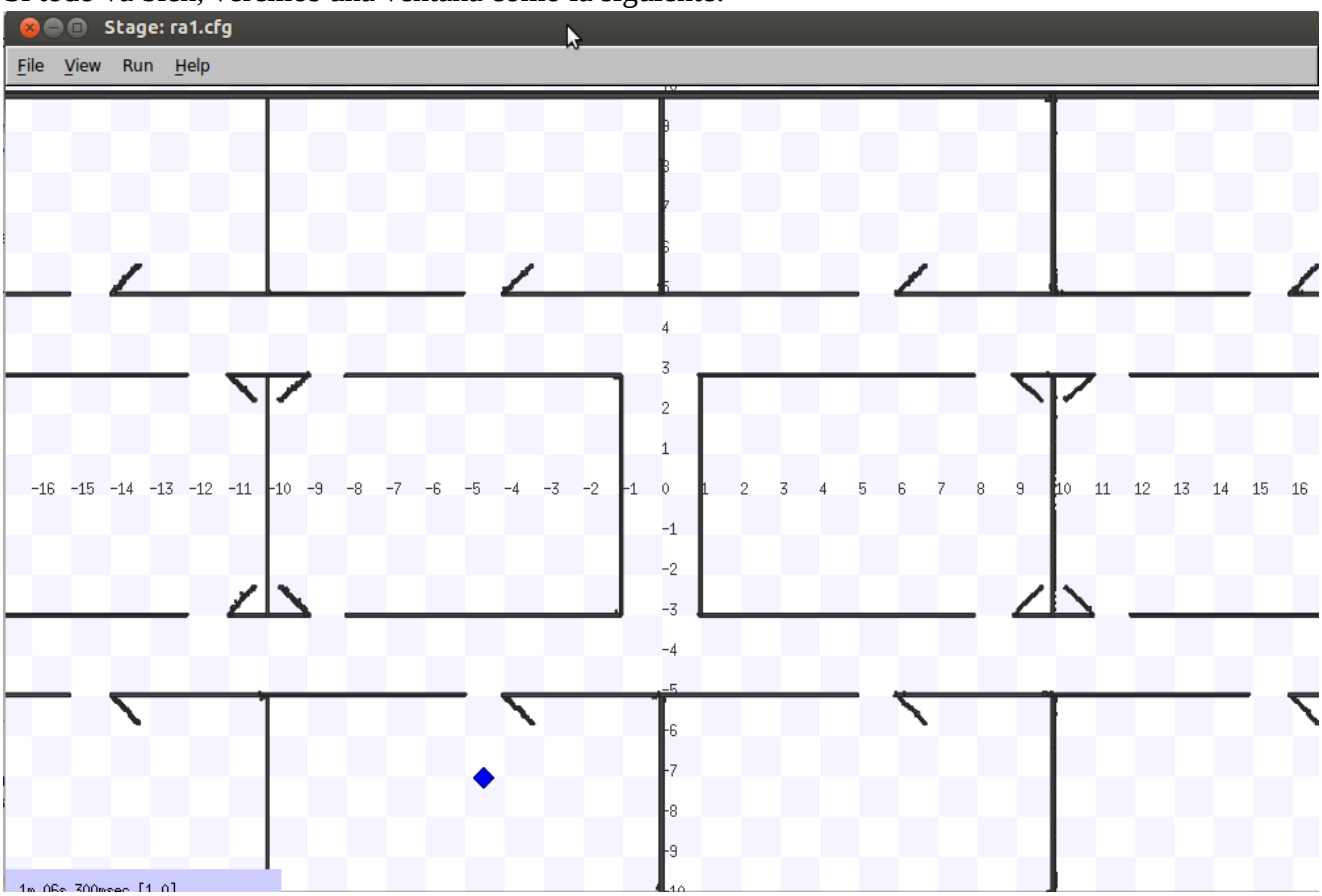
Ejecución de Stage

Una vez arrancado roscore procedemos a poner en marcha Stage. Para ello, nos tenemos que descargar los siguientes ficheros: configuración e imagen del mundo. El primero contiene la configuración del mundo que vamos a simular (incluido el robot) y el segundo contiene el mapa en una imagen. Ahora, desde línea de comandos, ejecutamos:

```
rosrun stage_ros stageros ra1.cfg
```

rosrun permite ejecutar paquetes de ROS. El primer argumento es el nombre del paquete y el segundo es un ejecutable que se encuentra en dicho paquete. El tercero es el nombre del fichero de configuración del mundo. TRUCO: si escribimos `rosrun stage_ros` y pulsamos la tecla tabulador, se nos dirá los ejecutables que tiene ese paquete.

Si todo va bien, veremos una ventana como la siguiente:



Manejo básico de Stage

Tanto Stage como Player utilizan un sistema de coordenadas levógiro, con lo cual los giros "a izquierdas" se consideran de signo positivo, y los que son "a derechas" de signo negativo. Los ejes de coordenadas en Stage son los habituales, es decir la X aumenta hacia la derecha y la Y hacia arriba.

Stage permite tener a más de un robot en su mundo.

Los mundos en Stage son ficheros bitmap, de modo que son sencillos de crear con cualquier herramienta de dibujo. Se puede cambiar la posición del robot arrastrándolo con el botón izquierdo del ratón y su orientación arrastrándolo con el derecho.

Para desplazarse por el mapa, pulsar sobre cualquier punto del mismo con el botón izquierdo del ratón y arrastrar en la dirección deseada.

De los menús que nos aparecen en el simulador, destacamos las siguientes opciones:

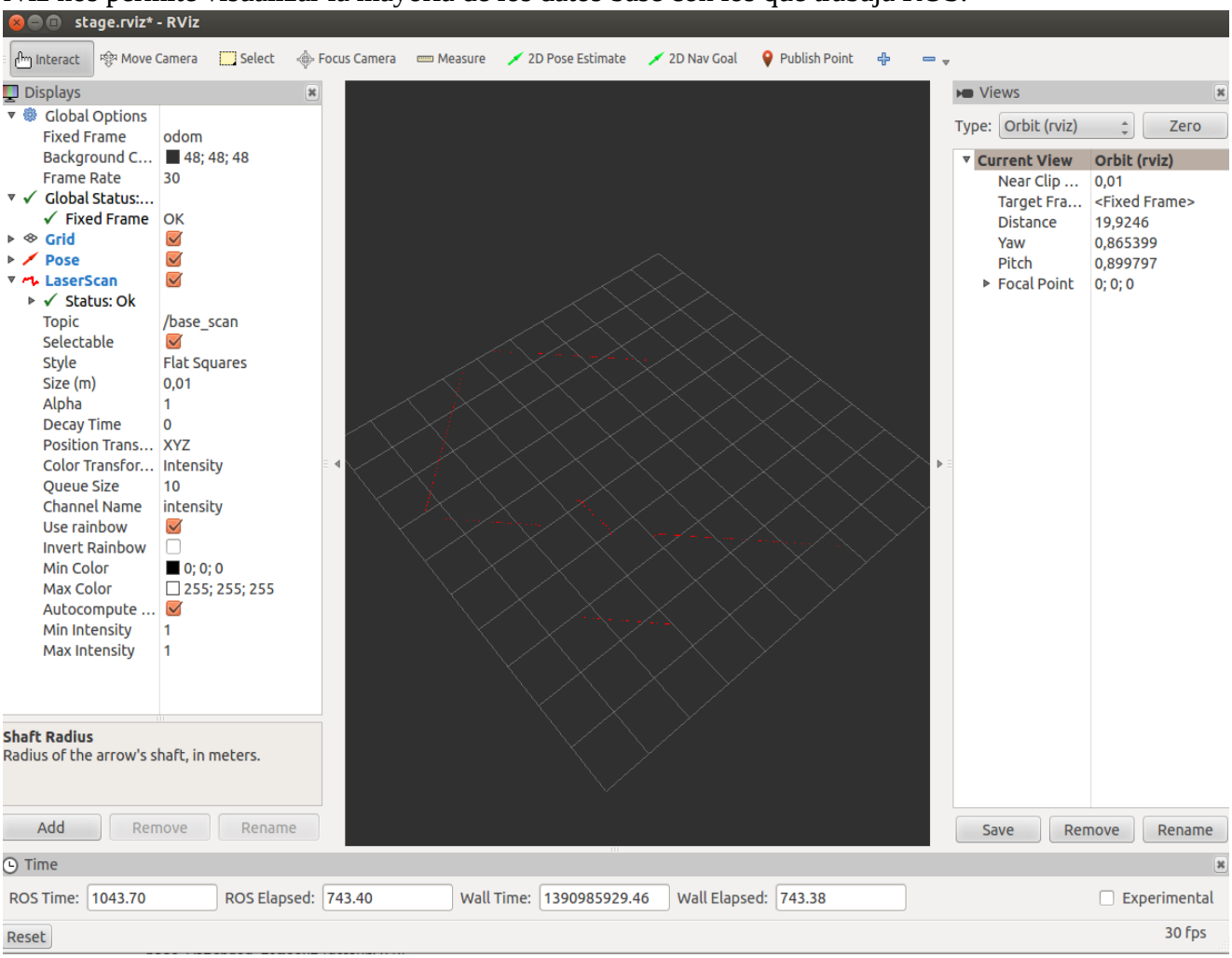
- View->Data: Permite visualizar los datos de los sensores del robot.
- View->Perspective camera: permite ver el mundo en pseudo-3D.
- View->Footprints: nos muestra el camino seguido por el robot.
- View->SaveScreenShots: empezará a guardar instantáneas del mundo, las cuales después pueden ser juntadas para montar un vídeo.

También podemos visualizar los datos del láser ejecutando lo siguiente:

```
roscd stage
```

```
rosrun rviz rviz -d `rospack find stage`/rviz/stage.rviz
```

rviz nos permite visualizar la mayoria de los datos base con los que trabaja ROS.



Existen diversas utilidades tanto desde línea de comandos como aplicaciones gráficas. Probad rqt_grapth. También probad los comandos relacionados con los tópicos y nodos. Haced que se muestre la información del *ground truth* de stage y ved cómo cambia al mover el robot.

```
rostopic echo /base_pose_ground_truth
```

Control del robot mediante teclado

Vamos a ver cómo podemos controlar al robot mediante el teclado. Hay que decir que lo que vamos a hacer aquí se podría hacer perfectamente con un robot real. Vamos a descargar un código compilarlo para poder trabajar. Lo primero es crearnos un espacio de trabajo en nuestro directorio.

- Creamos un directorio, por ejemplo `catkin_ws`
- Creamos un director dentro de este llamado `src`.
- Dentro de ese directorio (`catkin_ws/src`) ejecutamos `catkin_init_workspace`
- Veremos que se han creado varios directorios. En el `src` es donde tendremos nuestros fuentes.
- Nos movemos a `src` y vamos a descargar el código para controlar desde el teclado un robot. Ejecutamos:

```
git clone https://github.com/ros-teleop/teleop_twist_keyboard.git
```

- Este comando descarga el código. Nos movemos a `catkin_ws` y ejecutamos `catkin_make`
- Ya tenemos listo el nodo.

Ahora ejecutamos:

```
source devel/setup.bash
```

Esto lo hacemos para que el sistema sepa que en nuestro directorio `catkin_ws` también tenemos ejecutables de ROS. Y ya por fin:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

3. Programando en ROS

Para nosotros, la programación en ROS va a consistir en crear un paquete que contenga nuestro código. Para ello, vamos a usar algunas instrucciones que nos facilitarán la tarea. Aquí indicamos los pasos a seguir para crear, compilar y disponer de un paquete.

Creando un paquete

Cambiamos al directorio donde queramos crear el paquete. En nuestro caso `~/catkin_ws/src` Ahí, ejecutamos en siguiente comando:

```
catkin_create_pkg wander roscpp geometry_msgs sensor_msgs nav_msgs
```

Lo que viene a continuación de wander son las dependencias del paquete (otros paquetes que vamos a usar). Estas dependencias se pueden editar en el fichero `manifest.xml`. Este fichero contiene información del paquete. OJO: cuando se crea el paquete se usa el nombre del usuario del sistema. Si dicho nombre contiene tildes y/o eñes da problemas de compilación.

`catkin_create_pkg` creará un directorio con todos los ficheros necesarios de configuración. `wander` será el ejecutable que crearemos y lo siguiente son las dependencias de este paquete. Añadimos las siguientes líneas al final del fichero `CMakeLists.txt` que se encuentra en el directorio recién creado.

```
add_executable(${PROJECT_NAME} src/wander.cpp)
target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES} )
```

Descargamos y dejamos en el directorio `catkin_ws/src/wander/src` el fichero con el esqueleto que vamos a usar. En este fichero se explica cómo conectarnos al manejador de nodos y cómo usar el láser para hacer una evitación de obstáculos.

Compilar el paquete

Para compilar el paquete, ejecutaremos `catkin_make` en el workspace de catkin, en nuestro caso, `catkin_ws`.

Ejecutar el nodo

Una vez compilado sin errores, podemos ejecutarlo con

```
rosrun wander wander
```

El primer wander se corresponde con el nombre del paquete y el segundo con el nombre del ejecutable que hemos creado.

Recibir un mensaje de odometría del robot o Stage

Para recibir el mensaje de odometría e interpretarlo, podemos usar el siguiente código:

En el código (wander.cpp) habrá que incluir el fichero donde se definen estos mensajes.

```
#include "nav_msgs/Odometry.h"
```

Luego tenemos que declarar otro subscriptor como miembro de la clase y declarar en el constructor qué método se encargará de recoger este mensaje.

```
ros::Subscriber odometry;
odometry = nh.subscribe("odom", 1, &Wander::commandOdom, this);
```

Aquí tenéis un ejemplo de método que recibe la odometría:

```
void commandOdom(const nav_msgs::Odometry::ConstPtr& msg) {
    ROS_INFO_STREAM("Odometry x: " << msg->pose.pose.position.x);
    ROS_INFO_STREAM("Odometry y: " << msg->pose.pose.position.y);
    ROS_INFO_STREAM("Odometry angz: " << 2*atan2(msg->pose.pose.orientation.z, ms
g->pose.pose.orientation.w)); }
```

Tenemos tres valores: x e y y luego el ángulo. Al estar el giro representado con un quaternion hay que hacer la conversión que se indica.

3.1. Código de ejemplo

Vamos a ver el código pasado como ejemplo para empezar a trabajar.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "sensor_msgs/LaserScan.h"
#include <cstdlib>
#include <ctime>
```

Ficheros a incluir. Los `sensor_msgs` y `geometry_msgs` son necesarios para especificar el tipo de mensajes que vamos a recibir o enviar. Si vamos a usar otro tipo de mensajes, tenemos que incluir otros ficheros.

```
class Wander {
        protected:
        ros::Publisher commandPub;
        ros::Subscriber laserSub;

        double forwardVel;
        double rotateVel;
        double closestRange;
```

Definimos la clase Wander que va a definir nuestro nodo. Declaramos un publicador `commandPub` y un suscriptor `laserSub`.

```
Wander(ros::NodeHandle& nh) {
                forwardVel=1.0; // 1m/s es una velocidad bastante alta
                rotateVel=0.0;
                closestRange=0.0;
                srand(time(NULL));
                commandPub = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);
                laserSub = nh.subscribe("base_scan", 1, &Wander::commandCallback,
this);
        };
```

El constructor de la clase. Recibe como parámetro el manejador de ROS `nh`. Este manejador nos permite acceder al `roscore` al que se conecte este nodo.

Con el manejador "advertimos" que vamos a publicar un mensaje del tipo `geometry_msgs::Twist`, que se llamará `cmd_vel` y con el valor de 1 indicamos que si se acumulan varios mensajes, solo el último es enviado. Con `commandPub` podremos enviar mensajes. No podemos enviar dos mensajes con el mismo nombre.

También nos "suscribimos" a un mensaje `base_scan` y lo asociamos al método `commandCallback`. Esto quiere decir que cuando en el sistema algún nodo publique un mensaje con este nombre, se llamará a dicho método.

```
void commandCallback(const sensor_msgs::LaserScan::ConstPtr& msg) {
                /* VARIOS MENSAJES */
                for (int i=0; i< totalValues; i++) {
                        ROS_INFO_STREAM("Values[" << i << "]:" << msg->ranges
[i]); // Acceso a los valores de rango
                }
        };
```

Este método se llamará cuando se reciba un mensaje `base_scan`. Como parámetro recibe el contenido de dicho mensaje. En este caso, contiene un array de datos con los valores del láser. ROS_INFO_STREAM nos permite imprimir mensajes por la salida estándar de ROS (habitualmente la consola), formateando la salida con un `timestamp`.

```
void bucle() {
                ros::Rate rate(1);
                while (ros::ok()) {
                        geometry_msgs::Twist msg;
                        msg.linear.x = forwardVel;
                        msg.angular.z = rotateVel;
                        commandPub.publish(msg);
                        ros::spinOnce();
                        rate.sleep();
                }
        };
```

Este método nos permite crear un bucle que se irá ejecutando cada cierto tiempo. Con `ros::Rate rate(1);` estamos diciendo que la frecuencia con la que se realizará este bucle es de 1hz o una vez por segundo. Nos metemos en un bucle que se ejecutará mientras el sistema de ROS esté funcionando.

Creamos un mensaje con la velocidad lineal y rotacional a enviar. Este mensaje se mete en el publicador, pero no se publica al resto de nodos hasta que no ejecutemos `ros::spinOnce();` Por último, llamamos a `sleep` para esperar a la frecuencia de publicación definida.

```
int main(int argc, char **argv) {
        ros::init(argc, argv, "wander");
        ros::NodeHandle nh;
        Wander wand(nh);
        wand.bucle();
        return 0;
};
```

Este es el punto de entra a nuestro programa (nodo). Al llamar a `ros::init` estamos conectando con `roscore` y le decimos que nos llamaremos `wander`. Creamos un manejador que le pasaremos al contructor de la clase y por último ejecutamos el bucle.

4. Simulador 3D Gazebo

Gazebo es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD e importar modelos ya creados.

Además, es posible sincronizarlo con ROS de forma que los robots emulados publiquen la información de sus sensores en nodos, así como implementar una lógica y un control que dé ordenes al robot.

Instalación

Gazebo forma parte del bundle de ros "`ros-kinetic-desktop-full`", no obstante el robot que se usará como ejemplo no está integrado. El robot al que nos referimos es Turtlebot, un pequeño robot con una estructura montada sobre una base de Roomba y que integra sensores de odometría y una cámara RGB-D, entre otros.
El proceso de instalación se detalla en la Wiki de ROS, donde además se encuentra toda la documentación y otros tutoriales de interés http://wiki.ros.org/turtlebot_simulator.

Resumiendo, hay que ejecutar este comando para instalar todas las dependencias de Turtlebot:

```
sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic
-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi r
os-kinetic-ar-track-alvar-msgs
```

Esto instala componentes para trabajar con un Turtlebot real, pero nosotros vamos a trabajar con uno simulado. Para descargar la definición del robot para Gazebo y otros ficheros que intervienen en la simulación, descargamos del repositiorio oficial las últimas versiones de estos componentes:

```
git clone https://github.com/turtlebot/turtlebot_simulator
```

Ejecución de Gazebo y Turtlebot

Gazebo puede ejecutarse como simulador `stand-alone`, pero nosotros lo usaremos conjuntamente con ROS para poder consultar desde los nodos que creemos la información de los sensores, así como para mandarles comandos de velocidad. Para lanzar el simulador primero tenemos que arrancar en un terminal el motor de ROS:

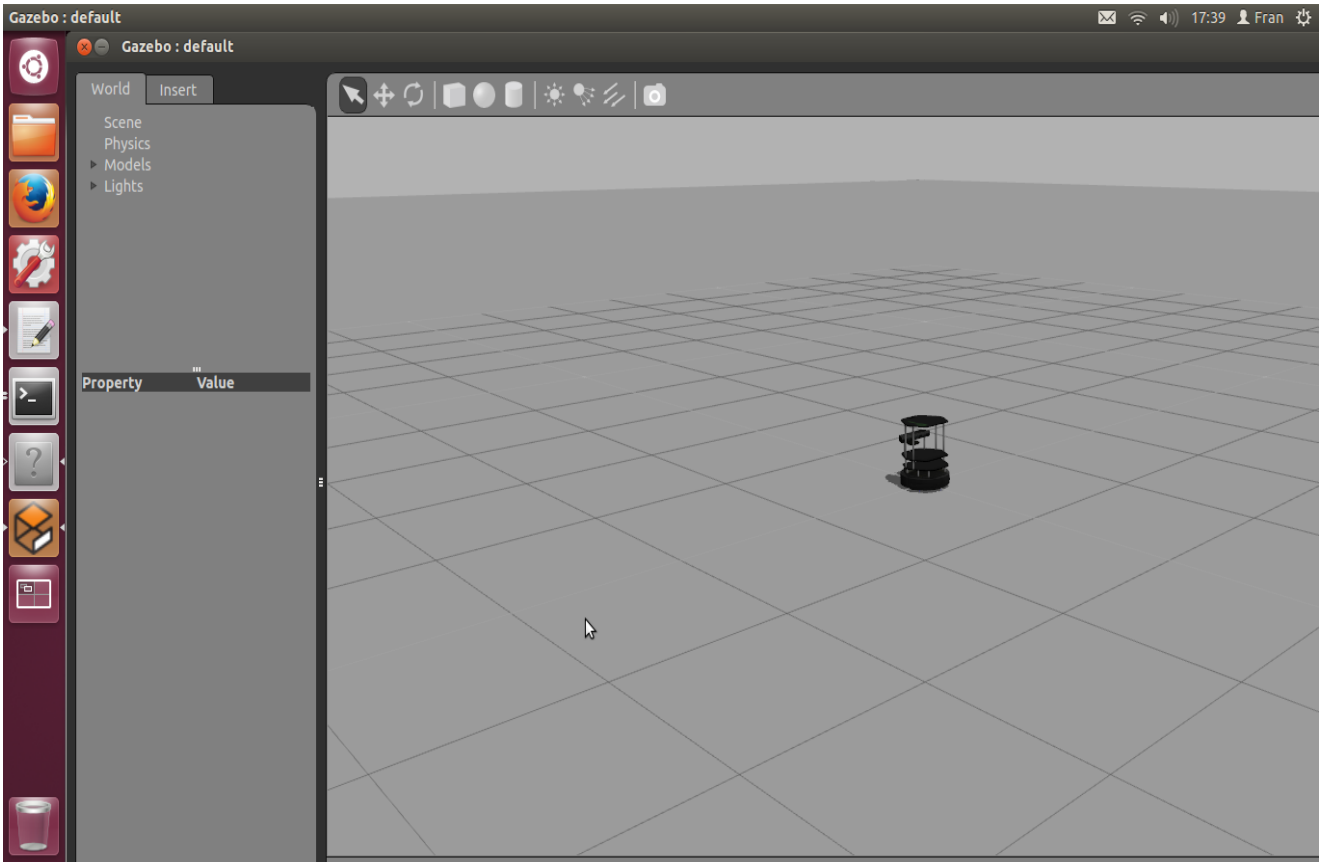
```
roscore
```

En otra terminal lanzamos el siguiente comando:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

`roslaunch` es una herramienta que sirve para lanzar varios nodos de ROS de un paquete de forma automática y con una configuración específica impuesta por un fichero XML con extensión .launch.

Tras ejecutar el comando se abrirá la ventana de Gazebo con un `turtlebot` cargado situado en el centro del mundo.



Control de la cámara

El control del punto de vista de diseño es parecido al que muestran otros programas de diseño o simulación 3D:

- El botón izquierdo del ratón sirve para trasladar la cámara por el mundo
- El botón derecho o la rueda del ratón sirve para hacer zoom en la escena
- El botón central del ratón sirve para rotar la cámara

Diseño de mundos

Gazebo permite diseñar elementos con herramientas CAD, dichas herramientas son muy limitadas, pero suficientes para crear mundos sencillos.
Estas herramientas están situadas en la zona superior del programa y nos permite insertar cubos, esferas y cilindros, así como distintos puntos de luz.

```
Intenta poner un cubo a la izquierda del robot
```

Las otras tres herramientas sirven para seleccionar elementos, trasladarlos y rotarlos.

```
Rota el robot para que la cámara apunte directamente al cubo
```

Los elementos que contiene el mundo se van listando en la parte izquierda del programa, bajo la pestaña "World" y si son seleccionados permiten la edición de algunas propiedades tales como la pose o el nombre (esta característica no está implementada en la versión de que disponemos en el laboratorio).

Visualizar información de los sensores

Por el hecho de tener ROS funcionando y haber usado `roslaunch` para lanzar el simulador, ya tenemos publicada la información que captan los sensores en la simulación. Para visualizarla usaremos otra herramienta de ROS llamada RViz, por lo que ejecutamos en otro terminal

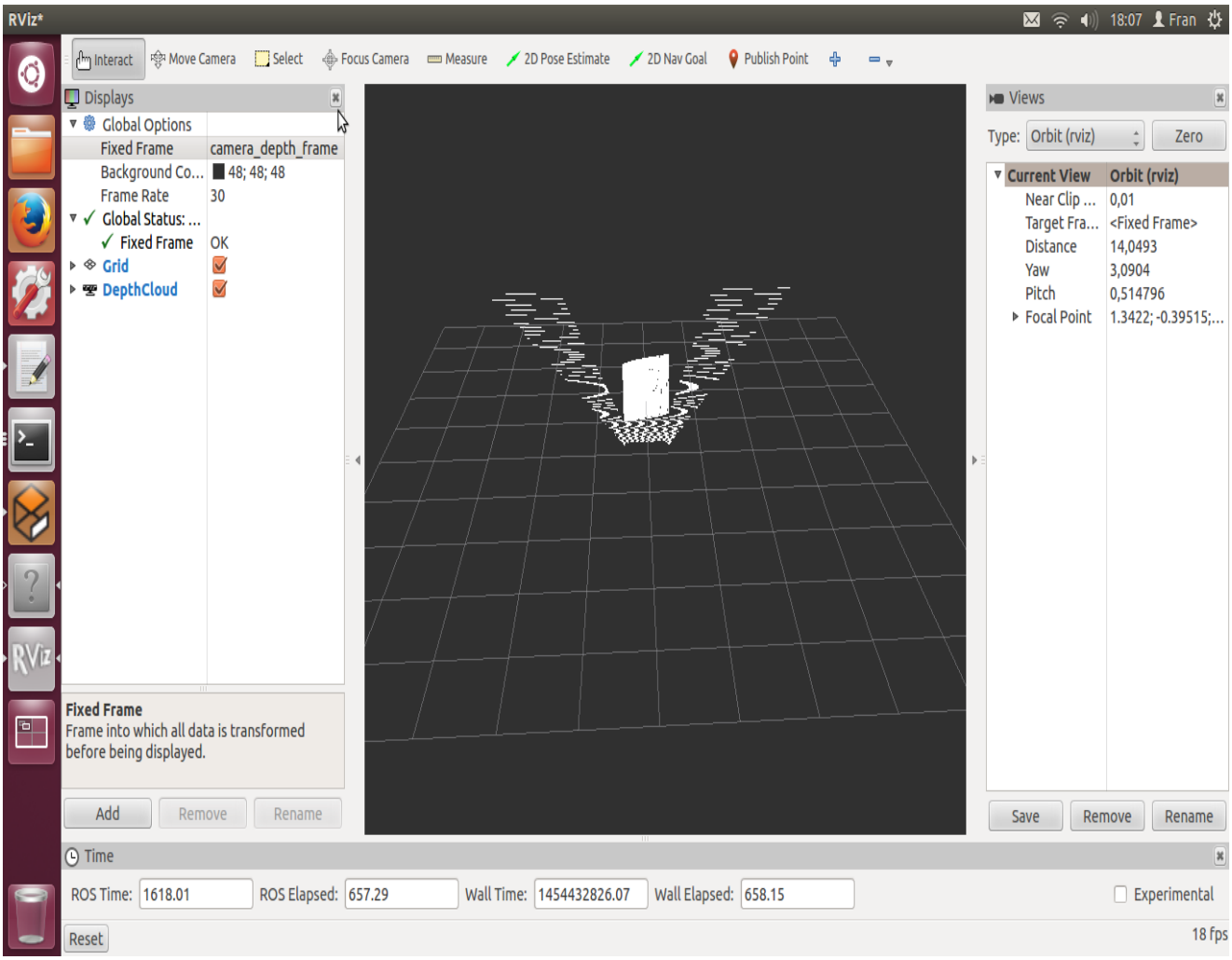
```
rosrun rviz rviz
```

El comando lanza el programa pero aparentemente no muestra información alguna. Esto es porque aún no estamos suscritos a ningún tópico de los del robot.
Antes de suscribirnos a un tópico, vamos a configurar el marco de referencia global del visualizador, que por defecto está en `map`, y lo cambiamos a `odometry`. El marco fijo hace referencia a respecto a qué se va a mostrar la información en el visualizador. Si elegimos la opción `camera_link` la información se representará en base a la posición de la articulación de la cámara. Si elegimos la opción `camera_depth_frame` la información se representará en base a la posición del sensor de profundidad. Lo ideal es que el marco de referencia global haga referencia a un elemento que es fijo.

Ahora sí, para suscribirnos a un nodo pulsamos en el botón "Add" de la parte inferior izquierda. En la nueva ventana que se abre seleccionamos la pestaña "By topic" y elegimos uno de ellos en función de la información que queramos visualizar, por ejemplo "camera/depth/image_raw/DepthCloud" y pulsamos "OK".

El visualizador mostrará la nube de puntos captada por la cámara RGB-D, que si has seguido los pasos consistirá en un cono de puntos a ras del suelo y una pared formada por uno de los lados del cubo.

```
Si es una nube de puntos, ¿Por qué no tenemos puntos por todo el espacio?
```



Importar un modelo 3D al mundo (con varios robots incluidos)

Lo siguiente que vamos a hacer es importar un fichero de configuración que carga el robot en un mundo que hemos diseñado con anterioridad. Como se ha explicado arriba, el paquete que necesitamos es el de `turtlebot_gazebo` por lo que necesitamos ubicarlo. Para ello usamos el comando `rospack`

```
rospack find turtlebot_gazebo
```

Lo siguiente es crear un nuevo paquete de catkin en nuestro workspace y copiar dentro todo el contenido del paquete turtlebot_gazebo. Seguimos cambiándole el nombre del proyecto en el fichero package.xml y CMakeListst.txt para que cuadre con el nombre del paquete que hemos creado. Por último, copiamos dentro de la carpeta launch los ficheros que se proporcionan a continuación turtlebot_gazebo_multiple_files.tar.gz.
Estos ficheros contienen la configuración del simulador y de algunos nodos que se lanzan conjuntamente.

El primer fichero `create_multi_robot.launch` especifica la configuración del mundo que se va a usar. En este caso se toma como base un mundo vacío en el que se inserta el diseño del mundo, especificado en otro fichero que se proporciona `race.sdf`. En el siguiente bloque se incluye la configuración de los robots, que está definida en `robots.launch`.

Aprovechamos para cambiar las rutas del paquete por el nombre del paquete que hayamos creado en el comando `find`.
En `robots.launch` se especifica la configuración de los dos robots que vamos a cargar en el simulador. En el primer bloque se definen ciertas variables como la base del robot, el conjunto de "bandejas" que forma el cuerpo del turtlebot o el tipo de cámara que monta. Por ejemplo, podemos ver que la base que usará el robot se llama kabuki, que va a usar las plataformas hexagonales, en vez de las circulares y que va a montar una cámara kinect.

```
Nota: Por alguna razón, el simulador intenta cargar Turtlebot como si tuviera otr
o modelo de cámara (Asus Xtion) a pesar de especificarle que trabaje con Kinect.
Para solucionarlo cambiamos la línea que dice:

<arg name="urdf_file" default="$(find xacro)/xacro.py '$(find turtlebot_gazebo_mu
ltiple)/launch/$(arg base)_$(arg stacks)_$(arg 3d_sensor).urdf.xacro'" />

por:

<arg name="urdf_file" default="$(find xacro)/xacro.py '$(find turtlebot_gazebo_mu
ltiple)/launch/$(arg base)_$(arg stacks)_kinect.urdf.xacro'" />
```

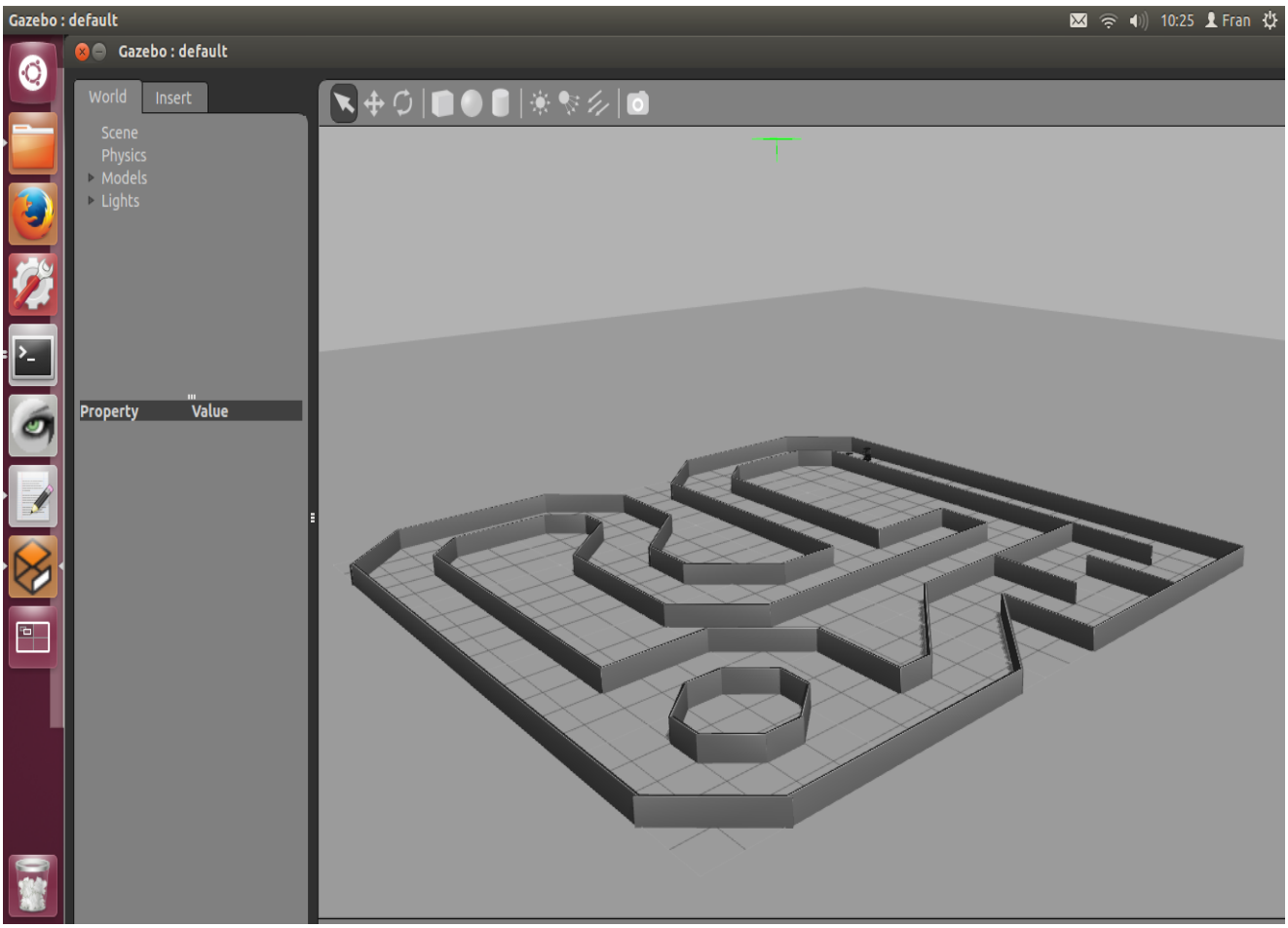
El siguiente bloque carga el modelo del robot, que es el mismo para los dos. Luego hay dos bloques que definen dos robots usando el modelo anterior, pero especificando algunos parámetros únicos como el nombre o la pose inicial. En esta especificación se hace referencia al último fichero `one_robot.launch` que se encarga de hacer aparecer el robot en el simulador y de cargar otros nodos relacionados con la redirección de los tópicos del robot como, por ejemplo, `robot_state_publisher` que se va a encargar de exponer todos los nodos que tiene el robot, con la información de las cámaras, odometría, láser y otros.

También se especifica un nodo que simula la salida de un láser a partir de la imagen de profundidad obtenida por kinect.

Cambiamos las rutas y los nombres de los paquetes por los correctos.

El otro fichero que se proporciona, `race.sdf`, contiene la definición de los elementos que forman el mundo. En este caso es un mundo formado por un sol, que nos prpporciona un sistema de iluminación, un plano que hace de suelo y varios elementos con geometría tipo caja que forman el mundo. Además de las dimensiones también se especifican la pose (traslación y rotación con respecto del sistema de referencia global), algunas propiedades físicas como la inercia o la masa y otras relacionadas con la textura que va a mostrar.
Una vez explicado todo esto, procedemos a construir el paquete y a cargar el mundo y los turtlebot en el simulador con el siguiente comando:

```
catkin_make
source devel/setup.bash
roslaunch turtlebot_gazebo_multiple create_multi_robot.launch
```



Este es el mundo que acabamos de cargar. Consta de una serie de paredes modeladas a partir de las cajas que se veían en el fichero `race.sdf` ubicadas de tal manera que forman un circuito y dos turtlebots situados en la línea de salida.

```
Si utilizamos RViz para visualizar los datos de los robots date cuenta de que al tener dos, cada uno con su conjunto de marcos de referencia, es posible que si seleccionamos un fixed frame correspondiente al robot1 y luego intentamos visualizar datos del robot2 la transformación falle y se muestre un error. Esto es algo de esperar ya que cada robot debe representar los datos en base a sus sistemas de referencia.
```

Mover a turtlebot por el mundo virtual usando comandos de velocidad

La forma de mover los robots, tanto en el mundo real como en el simulador, es enviándoles comandos de velocidad. Los robots tienen un nodo que consumen mensajes de tipo `geometry_msgs/Twist` que son el tipo de mensajes para indicar precisamente eso, valores para velocidades lineales y angulares.
Un mensaje de este tipo acepta como máximo 6 valores: las velocidades lineales en x, y, z y las velocidades angulares de roll, pitch y yaw. Estos valores permiten dar movimiento a robots con hasta 6 grados de libertad, pero turtlebot sólo puede moverse por el plano XY y rotar en el eje Z (yaw).
Pero antes de enviar nada es necesario averiguar qué tópico es que acepta este tipo de mensajes por lo que usamos el siguiente comando para ver qué tópicos tenemos en el sistema

```
rostopic list
```

Para obtener más información sobre algún tópico usamos el siguiente comando

```
rostopic info /robot1/mobile_base/commands/velocity
```

El comando para empezar a mover el robot es el siguiente:

```
rostopic pub -r 10 /robot1/mobile_base/commands/velocity geometry_msgs/Twist '{li
near: {x: 0.1}}'
```

Este comando sirve para obtener información de los nodos, mostrar su feedback y para enviarles mensajes. Nosotros usaremos el modo pub (publish) para publicar mensajes de velocidad. El parámetro -r indica la frecuencia en que se envían los mensajes, en este caso 10Hz. El siguiente argumento indica el tópico donde será lanzado este mensaje. El siguiente hace referencia al tipo de mensaje, en este caso uno de tipo Twist. Y por último se indica el contenido del mensaje, en este caso velocidad lineal de 0.1 unidades (m/s).

```
Si no se reciben mensajes de velocidad continuamente el robot se va a parar. Esto
es una medida de seguridad por si pierde conectividad con el controlador para que
no cause daños.
```

Obtener la imagen RGB de Turtlebot en un nodo de ROS

Lo siguiente que vamos a hacer es crear un nodo de ROS que se suscriba al tópico `robot1/camera/rgb/image_raw` de la cámara de turtlebot para visualizar las imágenes que capta. Para ello primero creamos un nuevo paquete en nuestro workspace de catkin indicando algunas dependecias.

```
catkin_create_pkg listener roscpp sensor_msgs image_transport std_msgs cv_bridge
```

Este comando genera un nuevo paquete vacío, pero antes de meter código es necesario indicar otras dependencias en el fichero CMakeList.txt.
El fichero CMakeList le indica al sistema de compilación de catkin (que internamente usa cmake) los directorios de las librerías que usa el paquete, que dependen de cada máquina, de forma que un solo fichero permita configurar el proyecto para varias máquinas.
Volviendo al proyecto, modificamos CMakeList.txt añadiendo las siguientes secciones

```
find_package(catkin REQUIRED COMPONENTS roscpp sensor_msgs image_transport std_ms
gs cv_bridge)
find_package(Boost REQUIRED COMPONENTS system)
find_package(OpenCV REQUIRED)
```

```
include_directories(${catkin_INCLUDE_DIRS})
add_executable(listener src/main.cpp)
```

```
target_link_libraries(listener
 ${catkin_LIBRARIES}
)
```

Y ahora sí, escribimos un fichero con el siguiente código

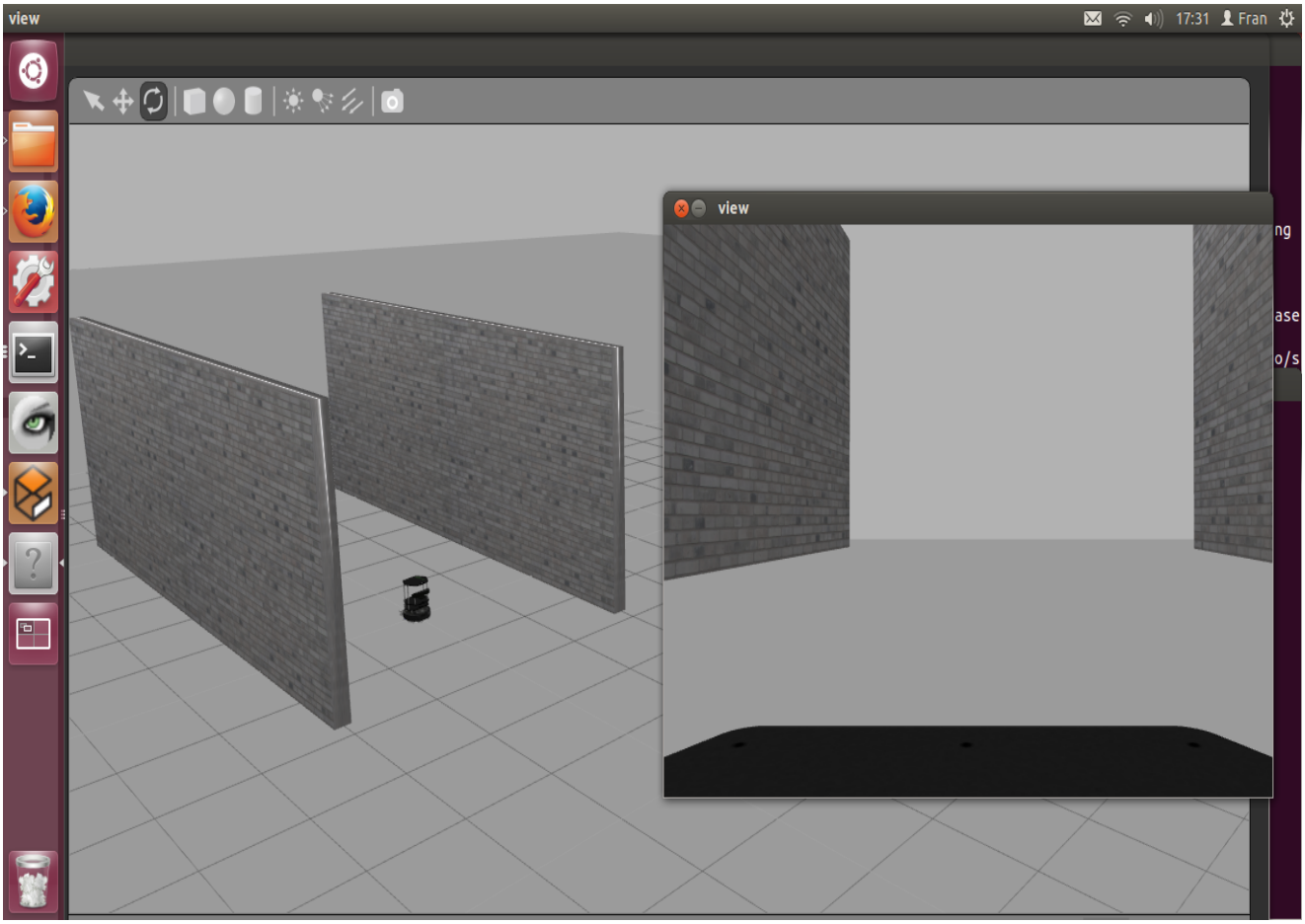
```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>

using namespace std;
void imageCallback(const sensor_msgs::ImageConstPtr& msg) {
    try {
         cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);
         cv::waitKey(30);
     }
     catch (cv_bridge::Exception& e) {
         ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str
());
     }
}

int main(int argc, char **argv) {
     ros::init(argc, argv, "image_listener");
     ros::NodeHandle nh;
     cv::namedWindow("view");
     cv::startWindowThread();
     image_transport::ImageTransport it(nh);
     image_transport::Subscriber sub = it.subscribe("robot1/camera/rgb/image_ra
w", 1, imageCallback);
     ros::Rate rate(10.0);
     while(nh.ok()) {
       ros::spinOnce();
       rate.sleep();
     }
     ros::spin();
     ros::shutdown();
     cv::destroyWindow("view");
}
```

Se comienza iniciando los servicios de ROS así como creando un NodeHandler que es el controlador de comunicaciones del nodo. También se crea una ventana de OpenCV, el objeto que nos permite recibir mensajes que contienen imágenes 'ImageTransport' y el objeto que nos suscribe al nodo que las provee. A este objeto 'Subscriber' se le pasa por parámetro el nombre del tópico, el tamaño de la pila de recepción y la función que se va a ejecutar cuando se reciba un mensaje.
A continuación existe un bucle que lo único que hace es 'spinOnce' y esperar un tiempo. La función 'spinOnce' indica a ROS que puede proceder a recibir, enviar y a hacer las tareas de comunicación propias del sistema. Si no se ejecuta esta función nunca se enviarán ni se recibirán los paquetes desde o hacia el nodo actual.
La función 'imageCallback' se ejecuta cada vez que el tópico publique un nuevo mensaje y se encarga de mostrar la imagen que contenga en la ventana creada anteriormente. La función 'cv::waitKey()' provoca que el sistema espere 30 milisegundos, dándole tiempo al mismo para cargar la imagen en la ventana. Si no se espera un tiempo después de actualizar el contenido de una ventana de OpenCV, el procesamiento será tan rápido que nunca se llegará a ver una imagen cargada en la ventana.

Si compilamos con catkin_make y ejecutamos el nodo con rosrun aparecerá una ventana de OpenCV que muestra las imágenes captadas por la cámara Kinect del Turtlebot en tiempo real.



El paquete completo puedes encontrarlo en el siguiente enlace listener.tar.gz

```
Puedes modificar el código para mostrar la imagen de profundidad?
```

Mover a turtlebot por el mundo virtual usando comandos de velocidad desde un nodo de ROS

En este bloque vamos a aprender cómo mover el turtlebot mandándole comandos de velocidad tal y como hacíamos en la sección `Mover a turtlebot por el mundo virtual usando comandos de velocidad` pero en este caso en vez de mandarle el mensaje directamente desde la terminal lo haremos desde un nodo ROS. Las ventajas de este sistema son evidentes: podremos reunir la información de los sensores para averiguar la situación del robot y el estado del entorno, procesar estos datos y generar los comandos de movimiento necesarios para guiar el robot en función de la sensorización.

```
#include <iostream>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
class RobotDriver
{
    private:
        //The node handle we'll be using
        ros::NodeHandle nh_;
        //We will be publishing to the "/base_controller/command" topic to issue
 commands
        ros::Publisher cmd_vel_pub_;

    public:
        //ROS node initialization
        RobotDriver(ros::NodeHandle &nh) {
            nh_ = nh;
            //set up the publisher for the cmd_vel topic
            cmd_vel_pub_ = nh_.advertise<geometry_msgs::Twist>("/robot1/mobile_ba
se/commands/velocity", 1);
        }

        //Loop forever while sending drive commands based on keyboard input
        bool driveKeyboard() {
             std::cout << "Type a command and then press enter. " <<
                   "Use '+' to move forward, 'l' to turn left, " << "'r' to turn
 right, '.' to exit.\n";
             //we will be sending commands of type "twist"
             geometry_msgs::Twist base_cmd;
             char cmd[50];
             while(nh_.ok()) {
                 std::cin.getline(cmd, 50);
                 if(cmd[0]!='+' && cmd[0]!='l' && cmd[0]!='r' && cmd[0]!='.') {
                     std::cout << "unknown command:" << cmd << "\n";
                     continue;
                 }
                 base_cmd.linear.x = base_cmd.linear.y = base_cmd.angular.z = 0;
                 //move forward
                 if(cmd[0]=='+') {
                     base_cmd.linear.x = 0.25;
                 }
                 //turn left (yaw) and drive forward at the same time
                 else if(cmd[0]=='l') {
                     base_cmd.angular.z = 0.75;
                     base_cmd.linear.x = 0.25;
                 }
                 //turn right (yaw) and drive forward at the same time
                 else if(cmd[0]=='r') {
                     base_cmd.angular.z = -0.75;
                     base_cmd.linear.x = 0.25;
                 }
                 //quit
                 else if(cmd[0]=='.') {
                     break;
```

```
                }
                //publish the assembled command
                cmd_vel_pub_.publish(base_cmd);
            }
            return true;
        };

    int main(int argc, char** argv) {
        //init the ROS node
        ros::init(argc, argv, "robot_driver");
        ros::NodeHandle nh;
        RobotDriver driver(nh);
        driver.driveKeyboard();
    }
}
```

En este caso en vez de suscribirse a un tópico para obtener mensajes, lo que va a hacer es publicarlos para que otros nodos los consuman.
El código especifica una clase que consta de un constructor y una función.
El constructor se ocupa de crear el objeto 'Publisher' a partir del 'NodeHandler' pasándole el tipo de mensajes que envía 'geometry_msgs::Twist', la dirección del tópico y el tamaño de la pila del mismo.
En la función 'driveKeyboard()' se crea un objeto del tipo del mensaje que se va a publicar 'Twist' y a continuación figura un bucle infinito en que se espera que el usuario introduzca un carácter y en función del mismo se realizará una acción u otra: avanzar, girar a izquierda, girar a derecha o detener el programa.
Esto se hace dándole valores a las propiedades pertinentes del mensaje. Por ejemplo, para avanzar, indicamos una velocidad lineal en el eje X de 0.25. Para girar a la derecha se indica una velocidad angular en el eje Z negativa (expresada en radianes) y una velocidad lineal en el eje X positiva.
Por último se publica el mensaje de forma efectiva usando la función 'publish()'.

send_velocity_commands.tar.gz