**UNIVERSIDAD POLITÉCNICA DE MADRID**

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

**Trabajo Fin de Grado**

DISEÑO ELECTROMAGNÉTICO DE UN MOTOR SÍNCRONO DE IMANES PERMANENTES PARA EL ACCIONAMIENTO DIRECTO DE LA HÉLICE DE UN BARCO

**Autor**:

Salvador Ollero Martín-Tabernero

**Tutor:**

Claudio Rossi.

Zorana Milosevik

GRADO EN INGENIERÍA DE LAS TECNOLOGÍAS INDUSTRIALES

FEBRERO 2018

**Agradecimientos**

**ÍNDICE**

[RESUMEN 4](#_Toc505331381)

[1. INTRODUCCIÓN Y OBJETIVOS DEL PROYECTO. 8](#_Toc505331382)

[1.1. Evolución histórica y estado del arte. 8](#_Toc505331383)

[1.1.1. Evolución histórica 8](#_Toc505331384)

[1.1.2. Estado del arte. 10](#_Toc505331385)

[1.2. Objetivos del proyecto. 18](#_Toc505331386)

[2. CONCEPTOS PREVIOS. 20](#_Toc505331387)

[2.1. ROS. 20](#_Toc505331388)

[2.1.1. Introducción 20](#_Toc505331389)

[2.1.2. Versiones 21](#_Toc505331390)

[2.1.3. Conceptos 21](#_Toc505331391)

[2.1.4. Instalación de ROS 24](#_Toc505331392)

[2.2. Gazebo 24](#_Toc505331393)

[2.3. RVIZ 26](#_Toc505331394)

[2.4. Blender 27](#_Toc505331395)

[2.4.1. Características 27](#_Toc505331396)

[3. DISEÑO DEL ENTORNO EN GAZEBO. 29](#_Toc505331397)

[3.1. Integración del Velodyne. 29](#_Toc505331398)

[3.1.1. Simulación del Velodyne. 29](#_Toc505331399)

[En la imagen se observa un nuevo láser 3D capaz de cubrir el diámetro completo de las minas, y que se encuentra fijado al cuerpo del robot UNEXMIN. 30](#_Toc505331400)

[3.2. Diseño de las minas. 31](#_Toc505331401)

[3.2.1. Configuración de las minas para su uso en Gazebo. 32](#_Toc505331402)

[4. ELABORACCIÓN DE UN NODO DE OPERACIÓN TELEMÁTICA. 33](#_Toc505331403)

[4.1. Primeros pasos. 33](#_Toc505331404)

[4.2. Realización de un Publisher C++ 34](#_Toc505331405)

[4.2.1. Simulación y comprobación 35](#_Toc505331406)

[4.3. Realización de un Publisher C++ controlado por el teclado. 36](#_Toc505331407)

[4.3.1. Primer intento (iostream) 37](#_Toc505331408)

[4.3.2. Segundo intento (Curses.h) 38](#_Toc505331409)

[5. MAPEADO. 42](#_Toc505331410)

[5.1. Octomap 42](#_Toc505331411)

[5.1.1. Octomap\_server 42](#_Toc505331412)

[5.1.2. Desarrollo del Archivo .launch 43](#_Toc505331413)

[5.1.2.2. Segundo intento. 45](#_Toc505331414)

[5.1.3. Nodo de conversión 47](#_Toc505331415)

[5.1.3.1. Desarrollo. 47](#_Toc505331416)

[5.1.4. Tf Broadcaster y Listener. 49](#_Toc505331417)

[6. Líneas futuras 51](#_Toc505331418)

[7. BIBLIOGRAFÍA 52](#_Toc505331419)

[8. Planificación temporal 56](#_Toc505331420)

[9. Presupuesto. 58](#_Toc505331421)

[10. Índice de figuras 59](#_Toc505331422)

[11.ANEXOS 61](#_Toc505331423)

[11.1 Código del Velodyne 61](#_Toc505331424)

[11.2. Código del primer publisher 63](#_Toc505331425)

[11.3 Código del primer nodo de desplazamiento telemático 64](#_Toc505331426)

[11.4. Código del segundo nodo de desplazamiento telemático. 65](#_Toc505331427)

[11.5. Ejemplo de CmakeList.txt 71](#_Toc505331428)

[11.6. Ejemplo de Package 74](#_Toc505331429)

[11.7. Código del conversor de datos. 77](#_Toc505331430)

[11.8. Código del Broadcaster. 78](#_Toc505331431)

[11.9. Código del Listener. 79](#_Toc505331432)

[11.10. Código del octomap\_server.launch. 80](#_Toc505331433)

# RESUMEN

Los robots móviles son máquinas automáticas capaces de trasladarse en cualquier ambiente dado. Las posibilidades que ofrecen son de elevada consideración en campos como el militar, sanitario, expediciones, inspección de instalaciones y en general en todo tipo de entornos hostiles y de seguridad. Esto los ha llevado a ser un importante foco de la investigación actual.

Es la navegación la que permite guiar el curso de un robot móvil a través de un entorno con obstáculos, de tal forma, que llegue a su destino de manera segura y sin incidentes. Para ello son esenciales las tareas de percepción, planificación y guiado.

La primera de todas ellas se logra a través de los sensores del robot móvil. Gracias a ellos es capaz de conocer el estado del mundo que le rodea y la posición exacta de sus elementos. Un sensor es capaz de convertir una variable física en un impulso eléctrico que recibe nuestro controlador.

El programa es el responsable de interpretar los impulsos que le llegan y de actuar en consecuencia planificando una trayectoria que permita alcanzar el punto de destino seleccionado y guiando el vehículo a través de la referencia establecida. La posibilidad de fallos y errores en la implementación del programa es cuantiosa y sus consecuencias pueden serlo también. Por ello es necesaria una fase previa de simulación, que permita el desacierto sin que haya repercusiones físicas en el modelo robótico.

Diversos softwares se han utilizado para emular el comportamiento de los robots móviles. Tanto es su auge que plataformas tan conocidas como Matlab se intentan introducir en este mercado mediante el lanzamiento de diversas funciones que faciliten la labor. Es en este contexto donde surge Gazebo, un simulador capaz de recrear entornos 3D para evaluar el comportamiento de un robot en un mundo virtual, y gracias al cual se pueden diseñar controladores y mundos virtuales.

Este emulador posee una gran facilidad para sincronizarse con ROS (Robot Operating System), un Framework orientado al desarrollo de software para robots, proporcionando la funcionalidad de un sistema operativo en un clúster heterogéneo. ROS ofrece los servicios estándar de un sistema operativo tales como la abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de menajes entre procesos y mantenimiento de paquete.

El presente trabajo, trata del diseño y programación de un entorno de simulación válido para el robot submarino UNEXMIN mediante el uso de ROS y Gazebo. En concreto, se establecerán algunos de los sensores necesarios y se desarrollarán las herramientas necesarias para realizar un mapeado 3D de las minas de Kaatiala.

En un principio, y previo a comenzar con el proyecto se debe adaptar el entorno de trabajo. Esta etapa consiste en realizar una partición lo suficientemente grande en el ordenador que nos permita trabajar con el sistema operativo de Linux/Ubuntu. Tras ello, es necesario la descarga de Gazebo y los paquetes de ROS que se utilizarán para desarrollar los objetivos propuestos.

Una vez instaladas todos las herramientas, se deben definir los sensores y actuadores necesarios para orientar al robot. El presente trabajo se centra en el láser, eligiendo así, si es necesaria una geometría en 3D o es suficiente con uno en 2D. En el caso actual se opta por un láser Velodyne en tres dimensiones pues el modelo a considerar debe ser capaz de proporcionar información en las todas las direcciones del espacio con rapidez y precisión, siendo este modelo el que ofrece las mejores prestaciones.

Dicho sensor se fija al frente del robot (contrario a los propulsores) y proporciona una nube de puntos que se visualiza en Rviz y al que cualquier nodo se puede subscribir.

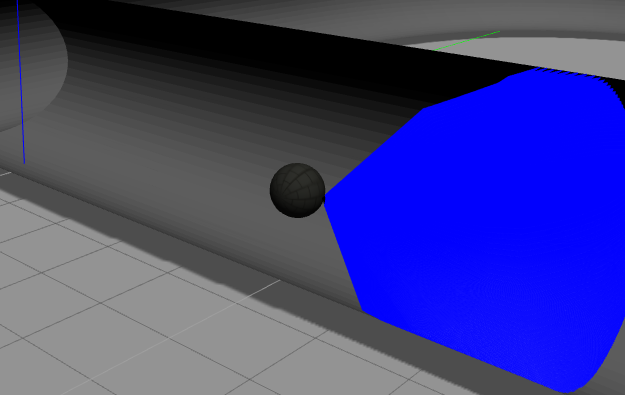


Figura 1: Integración del láser 3D Velodyne en el robot submarino Unexmin.

Con el robot operativo y todos los sensores establecidos se procede a poner en movimiento el vehículo submarino. En esta etapa son necesarios conocimientos necesarios de ROS y de C++ o Python. La primera opción supone una ventaja operativa debido a su uso anterior para diversas aplicaciones del robot.

La primera herramienta necesaria y posiblemente una de las más importantes es un software que proporcione una forma de navegación. Con este fin, se crea un nodo que sea capaz de desplazar al robot mediante el uso del teclado del ordenador. Como paso previo se realiza un código que cíclicamente publique en el topic del robot “*set\_model\_state*” tanto las velocidades lineales como las angulares y vaya actualizando su posición en función de ellas y de la frecuencia del bucle. Con este nodo activo, Gazebo es capaz de subscribirse y obtener dicha información representando el movimiento establecido. La realización de este nodo permite comprobar el funcionamiento del robot ante los distintos comandos que se le puedan enviar y además establece una base para el software de navegación definitivo.

El siguiente paso a seguir es la implementación de un código que permita desplazar al robot mediante el uso del teclado del ordenador. La mecánica no cambia sustancialmente respecto del paso anterior, pero surge el problema que conlleva el uso de un elemento externo. Existen diversas librerías que permiten captar eventos del teclado, pero todas ellas se basan en esperar a que se introduzca un valor, bloqueando el programa hasta que se produce. Esto provoca que la posición del robot no se pueda actualizar en base a la frecuencia del bucle y por tanto, que se tenga que modificar el algoritmo.

El mapeado se realiza gracias a Octomap. Se trata de una serie de paquetes que proporciona una forma viable de realizar un mapa en 3D y que es compatible con ROS y Rviz. Su funcionamiento está basado en la subscripción al topic que proporciona el láser anteriormente instalado (“*/Unexmin/laser/scan*”), obteniendo los datos necesarios para la recreación de las minas en colores RGB.

Para proporcionar un formato de datos adecuando es necesario realizar una conversión de la información obtenida puesto que los datos que proporciona el topic mencionado no corresponde con los que Octomap solicita. Por ello, se realiza otro nodo que se subscribe al tema de Velodyne y realiza una transformación de formato de PointCloud a PointCloud2, publicándolos en otro topic (“*/Unexmin/laser/scan2*”).

Finalmente, es necesaria la creación de un frame fijo al cuál se pueda referenciar el desarrollo del mapa. Con ese fin se realizan los nodos broadcaster y listener. En el primero de ellos se crea el frame fijo “*map*” y se establecen las relaciones pertinentes que permiten las transformaciones de coordenadas que anteriormente estaban siempre referidas a “base\_link”, frame del propio robot. Este código es el conocido como nodo de localización y permite a Octomap establecer un punto fijo a partir del cual se crea el resto del mapa. El segundo nodo simplemente escucha al broadcaster y aplica las transformaciones que en él se realizan.

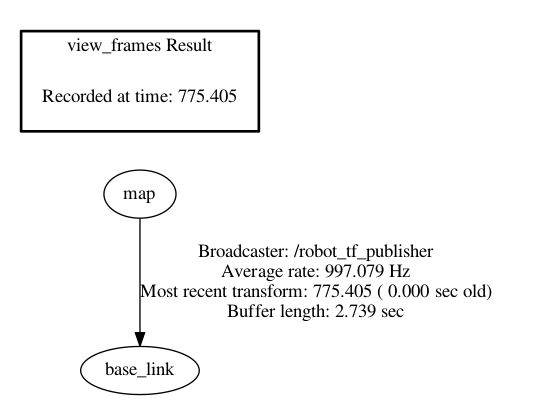


Figura 2: Frames tree creado mediante los nodos broadcaster y listener.

Tras establecer todas las herramientas que permiten al robot topografiar las minas de Kaatiala es necesario proporcionarle un entorno aproximado por el cuál moverse. Gracias a Blender (entorno de desarrollo 3D) se realiza un modelo en tres dimensiones que aproxima la forma de las minas. Blender permite exportar el modelo creado en un formato de extensión .dll que Gazebo puede utilizar.

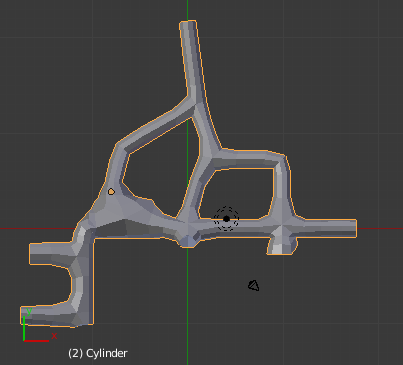


Figura 3: Minas de Kaatiala desarrolladas mediante el software Blender.

# 1. INTRODUCCIÓN Y OBJETIVOS DEL PROYECTO.

## **1.1. Evolución histórica y estado del arte.**

### **1.1.1. Evolución histórica**

Se pueden considerar como primeros robots a todos los artilugios mecánicos que disponían de algún tipo de automatización. Algunos ejemplos son antiguas esculturas egipcias animadas, los primeros relojes y cajas de música.

La palabra robot surgió por primera vez en 1920 cuando el escritor checoslovaco Karel Capek publicó su novela RU; una novela de ciencia ficción. Sin embargo, esta palabra no se hace popular hasta 1950 cuando Isaac Asimov publica el libro “Yo Robot” con las tres leyes de la robótica.

Durante la segunda Guerra Mundial, surgieron los precursores de los robots móviles; que nacieron de la necesidad de dar solución a la problemática que surgía en esta época. Gracias a la investigación en material de informática y cibernética se crearon bombas aéreas e inteligentes que usaban un sistema de guiado que les permitía concluir cuándo y dónde debían detonarse. No fue, sin embargo, hasta 1948 que el neurólogo Grey Walter construye los primeros robots móviles utilizando el sentido común y algo de bricolaje; son las conocidas tortugas de Bristol.

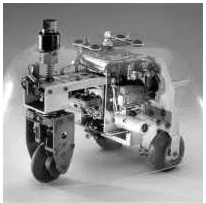


Figura 4: Tortugas de Bristol.

En la década de 1950 se construye el primer robot submarino operado remotamente. Se denomina POODLE y fue creado por Dimitri Rebikoff en Francia.

A principios de los 60´s la universidad John Hopkins desarrolla “Bestia”, un robot submarino que se desplazaba gracias a la ayuda de un sónar. Su principal inconveniente eran sus baterías pues tenían muy poca duración. Posteriormente, los robots subacuáticos empezaron a ganar popularidad, todo ello gracias a los sistemas CURV (vehículos de recuperación submarina por cable) de la marina de los Estados Unidos que, en 1966, recuperaron una bomba atómica extraviada en un accidente aéreo y que posteriormente, en 1973, rescataron a los pilotos de un sumergible en la ciudad de Cork.

En los años 70 se crea a Shakey, una máquina con sensores, cámara y radiolocalización. Es considerado el primer robot capaz de adaptar y tomar órdenes. En adición, en este periodo, la Unión Soviética utiliza un rover controlado a distancia para explorar la luna.

Durante los 80´s surgen diferentes proyectos de robots móviles para diversos usos y desarrollados por distintos países. Sin embargo, no es hasta 1990 cuando John Engelberger, conocido como el padre de los brazos robóticos, trabaja con el objetivo de lanzar los primeros robots en movimiento para el área industrial. A partir de ese momento los robots móviles y desplazables van cobrando fuerza en distintos campos, como por ejemplo la investigación.

Es también en esta década cuando aparecen los primeros vehículos sumergibles operados remotamente, concretamente en una plataforma de perforación de Noruega de la mano de i-Tech, una subdivisión de la gigantesca Subsea. Los robots submarinos siguen creciendo paulatinamente de mano de las firmas comerciales debido a las aplicaciones prácticas que estos tienen en la extracción del petróleo. Así surgieron el RCV-225 y el RCV-150, ambos desarrollados por Hydroproducts.

En los años 90´s Kongsberg Maritime desarrolla el primer vehículo autónomo sumergible, cambiando así el rumbo y el futuro de los robots sumergibles.

En 1991 en paralelo al crecimiento de los robots, en Helsinki, surge Linux.

En el año 1994, la Universidad Carnegie Mellon desarrolla Dante I y Dante II, robots capaces de caminar y que se utilizaron para la investigación de volcanes activos. En 1997, la NASA manda al espacio al rover Mars Pathfinder, con el objetivo de estudiar la superficie de Marte controlado desde la Tierra. Este vehículo estaba diseñado de forma que fuera capaz de evitar los peligros existentes y así abrirse camino a través de la superficie Marciana.

A partir del año 2000, el auge de los robots móviles aumenta y ya no son solo para investigación si no que se desplazan a otros ámbitos como pueden ser las misiones espaciales, los volcanes, el fondo del mar o al interior de fábricas. Llegan también los robots domésticos como Aibo, PackBot, Roomba e IntelliRobots, todos ellos con usos prácticos y fáciles de manejar para el ciudadano de a pie. Surgen también los primeros juguetes para niños.



Figura 5: Robot doméstico Roomba

Finalmente en 2006 surge ROS, originalmente bajo el nombre de Siwtchyard que fue desarrollado por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto STAIR. Desde 2008, el desarrollo continuó primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un proceso de desarrollo federado. Acabó cerrando en 2014.

### **1.1.2. Estado del arte.**

#### **1.1.2.1. Robots: definición y clasificación**

* + **Definición.**

Existen diversas definiciones de la palabra robot. A continuación, se muestran las más representativas.

Robot Institute of America (RIA), en 1979, la define como: “*Un robot es un manipulador reprogramable multifuncional diseñado para mover materiales, partes, herramientas, o utensilios específicos a través de varios movimientos para la realización de una gran variedad de tareas*”.

La Comisión Europea o European Commision proporciona la siguiente definición: “*Un robot es una máquina que actúa de forma independiente y se autocontrola, equipada con herramientas específicas para manipular o mecanizar y cuyos movimientos son programables con respecto a la orientación, posición y secuencia*”.

Sin embargo, la Japanese Robot Association (JARA) divide la definición en 6 clases, que van desde el robot más simple hasta el más inteligente.

* + Clase 1. Dispositivo de manejo manual: Un dispositivo con varios grados de libertad accionado por un operador;
  + Clase 2. Robot de secuencia fija: Un dispositivo de manipulación que lleva a cabo las etapas sucesivas de una tarea de acuerdo con un método predeterminado e inmutable, que generalmente es difícil de modificar.
  + Clase 3. Robot de secuencia variable: El mismo tipo de dispositivo de manipulación que en la clase 2, pero las etapas se pueden modificar fácilmente.
  + Clase 4. Reproducción robot: El operador humano realiza la tarea manualmente controlando el robot, que registra las trayectorias. Esta información se recuerda cuando es necesario, y el robot puede realizar la tarea en el modo automático.
  + Clase 5. Robot de control numérico: El operador humano le proporciona al robot un programa de movimiento en lugar de enseñarle la tarea manualmente.
  + Clase 6. Robot inteligente: Un robot con los medios para comprender su entorno y la capacidad de completar con éxito una tarea a pesar de los cambios en las condiciones circundantes bajo las cuales se realizará.

Finalmente, la Federación internacional de robótica (IFR) se basa en la definición contenida en el estándar ISO 8373 que dice: “*un robot es un manipulador multiusos controlado automáticamente, reprogramable, programable en tres o más ejes, que puede ser fijo en su lugar o móvil para su uso en aplicaciones de automatización industrial*”.

* + **Clasificación.**

Resulta realmente complicado establecer una única clasificación de los distintos y diversos tipos de robots debido a la gran multitud de parámetros a los que se puede atender para realizarla. Así, si la atención se centra, por ejemplo, a las posibles aplicaciones del robot los podemos catalogar en:

* Robot manipulador: también conocidos como robots industriales (inicialmente fueron utilizados en la industria) son máquinas de manipulación automática, reprogramables y multifuncionales. Tienen la capacidad de orientar y posicionar materiales, piezas, herramientas o dispositivos especiales para realizar las tareas asignadas con éxito. Suelen tener una posición fija pero en ocasiones se pueden desplazar gracias a la ayuda de raíles. Su ambiente de trabajo está totalmente estructurado y por tanto las condiciones en las que realiza las actividades están bajo control, simplificando de manera cuantiosa las tareas que realizan.
* Robot de servicio: se trata de dispositivos electromecánicos dotados con la capacidad de moverse y que usualmente llevan equipados uno o varios brazos mecánicos para la ejecución de actividades no industriales. Pueden llegar a reemplazar al ser humano en entornos poco estructurados.
* Robots de investigación: muchas universidades tienen en sus laboratorios diversos robots que no tienen una finalidad específica más que la de la pura indagación o que son fruto de distintos proyectos de investigación públicos y privados. Estos últimos sí tienen una aplicación determinada
* Robot militar: se trata de máquinas con aplicaciones militares y que asisten al ejército en tareas como pueden ser la vigilancia o búsqueda de minas y que pueden poseer un gran número de morfologías.
* Robots médicos: asisten a los hospitales en campos como la cirugía. Son robots que necesitan de mucha precisión y de múltiples brazos robóticos.
* Robots educacionales: su objetivo es ayudar con la enseñanza. Suelen ser utilizados en escuelas e institutos.

Los robots también podrían catalogarse en función de su autonomía (teleoperados, semi-autónomos y autónomos), por su generación (1G, 2G, 3G, 4G, 5G). Es interesante la clasificación en función de su arquitectura:

* Poliarticulados: engloba a todos aquellos robots que se encuentran fijos en un lugar o en su defecto, tienen desplazamientos limitados guiados por raíles. Están estructurados para mover sus componentes terminales en varios sistemas de coordenadas y con un número limitado de grados de libertad. Pueden ser clasificados según su morfología (Scara,doble Scara, Delta, cartesiano, cilíndrico, antropomórfico…)
* Robots móviles: son máquinas automáticas con la capacidad de desplazarse a través de cualquier ambiente dado. Gracias a los sensores equipados y a la programación puede guiarse por el entorno de forma autónoma, aunque en ocasiones este movimiento pueda ser supervisado o teledirigido por un operario. En este último caso, el robot es capaz de comunicarse con él mediante cables o de forma inalámbrica. Adoptan distintas morfologías en función de la tarea que vaya a realizarse y del entorno en el que la desarrollen. Se podrían clasificar en función del sistema de desplazamiento (con ruedas, oruga, cadenas, patas …) y del medio en el que se mueven (aéreos, terrestres, acuáticos, submarinos o espaciales).
* Androides: son todos aquellos robots que intentan reproducir, de manera aproximada, la forma y el comportamiento de las personas. Se trata de máquinas poco evolucionadas y principalmente dedicadas al estudio y la investigación.
* Zoomórficos: en ciertas fuentes consideran a los androides dentro de esta categoría puesto que reúne a todos aquellos robots cuyos sistemas de locomoción imitan al de los seres vivos. Estos, a su vez, pueden ser agrupados en caminadores y no caminadores. El primer grupo está en pleno desarrollo con vistas a la creación de vehículos capaces de desplazarse por terrenos muy accidentados. El segundo grupo está muy poco evolucionado.
* Híbridos: agrupa al conjunto de robots que pudieran ser clasificados en varias de las categorías anteriores.

#### **1.1.2.1. Robots submarinos.**

Aproximadamente dos terceras partes del planeta están cubiertas por agua. Mares, ríos, lagos, acuíferos…, proporcionan una infinidad de recursos, permiten el transporte de personas y mercancías, y, además, tienen un efecto vital en el clima y el medio ambiente. Por ello no sorprende el interés mostrado por el ser humano en conocer su estado y morfología, tanto en su superficie como en las profundidades.

Los robots submarinos se pueden clasificar en función de su autonomía, el tipo de misión a realizar o su sistema de propulsión, aunque la principal de todas ellas es la primera. Los AUVs son robots completamente autónomos, los ROVs son aquellos que necesitan ser controlados por un operario y los IAUVs se encuentran a medio camino entre los dos anteriores ya que requieren de un control supervisado, es decir, necesitan de un operador que razone las acciones que se deben llevar a cabo, aunque su objetivo es que lleguen a ser completamente autónomos. Todos ellos se explican más adelante.

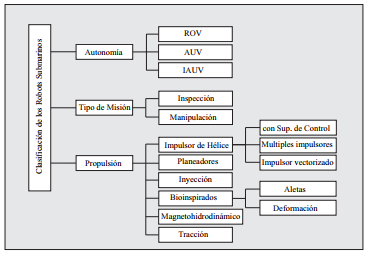


Figura 6: Clasificación de los robots submarinos.

Los robots submarinos también pueden clasificarse en función de la misión u objetivo que desean realizar. Esta clasificación determina qué sensores deberá tener equipado el robot. Así:

* Misiones de Inspección: se realizan mientras el robot está navegando. No suelen necesitar de un brazo robótico ni de ningún mecanismo para interactuar con el ambiente. Suelen ser misiones cuyo objetivo es la captación de imágenes mediante cámaras; la observación del lecho marino, gracias a la cartografía acústica; o sencillamente revisar instalaciones subacuáticas.
* Misiones de manipulación: es común la presencia de brazos manipuladores o herramientas que sean capaces de interactuar con el entorno. Son típicas las tareas de reparación o mantenimiento de instalaciones submarinas, ensamblaje de elementos, la recogida de muestras para diversos estudios, o incluso la intervención en desastres, ya sea retirando material contaminado o en el rescate de personas.
  + **Robots Submarinos Operados Remotamente o ROVs.**

Los robots submarinos operados remotamente son vehículos que están controlados a distancia y que se encuentran conectados a la superficie mediante lo que llaman un cordón umbilical (un conjunto de cables unidos). Este tipo de robot permite que un usuario defina, a través de una interfaz gráfica situada en el ordenador, las órdenes que el propio vehículo deberá realizar. A cambio, el ROV, le da una realimentación continua de las señales que le llegan a sus sensores; de esta forma es capaz de saber el estado del propio robot y del entorno que le rodea.

En los robots operados remotamente la energía eléctrica se divide y distribuye por todos los elementos. Es muy común el uso de propulsores, herramientas de torsión y brazos manipuladores, componentes que necesitan de una bomba hidráulica para ser alimentados. Esta bomba es accionada mediante un motor de alta potencia eléctrica, elemento que consume la mayor parte de la energía eléctrica.

Las cámaras de vídeo y luces son componentes muy habituales. El resto del equipo se suele incluir para ampliar las capacidades del robot. Podemos encontrarnos con sónares, magnetómetros, cámaras fotográficas, herramientas de corte, sistemas para toma de muestras, e instrumentos para medir parámetros.

El auge de los ROVs ha propiciado que su desarrollo tecnológico se acelere y que sus campos de aplicación se vean ampliados. Se puede ver a estos vehículos tanto en simples operaciones de revisión como en complejos proyectos asociados a un gran campo de sectores.

Este tipo de robot submarino es de uso común en las instalaciones petrolíferas dónde las exigencias de tiempos y periodicidad en las tareas de mantenimiento y reparación son muy elevadas. El uso de los ROVs en este tipo de emplazamiento será cada vez más elevado según se vaya extendiendo esta industria a aguas cada vez más profundas. Sin embargo, esto conlleva un problema; resulta que cuanto mayor es la profundidad, mayores son las fuerzas de arrastre que se producen en el cable, reduciendo el control que se ejerce sobre estos robots. Aunque se ha aumentado el diámetro de los cables, producto de la mayor demanda de energía, estas fuerzas de arrastre son debidas en su mayoría por la gran longitud del cable. Como solución se presenta el uso de un Sistema de Manejo de Cable que se ancla en el fondo marino y aguanta las fuerzas de arrastre y permite al robot moverse con mayor facilidad.

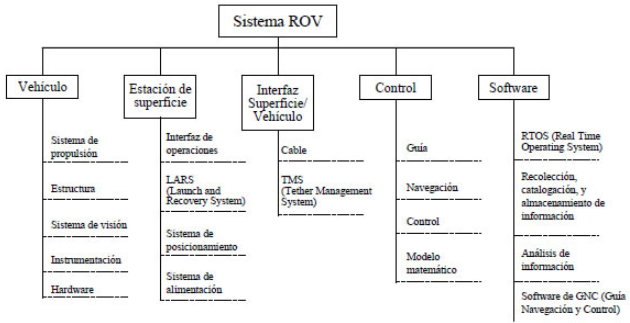


Figura 7: Esquema de los elementos propios de un sistema ROV

* + **Robots Submarinos Autónomos o AUVs.**

Se trata de robots autónomos capaces de realizar misiones sin la supervisión de un operario. No suelen disponer de una línea de comunicación entre el propio vehículo y la superficie, esto es debido a que tienden a estar programados para realizar misiones concretas y predefinidas. Sin embargo, si es necesaria dicha comunicación, se podría realizar mediante ondas de radio o a través de dispositivos acústicos. Estos sistemas, también ofrecen una forma totalmente precisa de evitar posibles golpes o choques contra el fondo.

Otra forma de navegación son los llamados sistemas de navegación inercial que, combinados con un sónar, son mucho más precisos y rápidos que los sensores puramente acústicos, sin embargo, su coste es mucho mayor. Hoy en día, también existe la navegación basada en GPS que ha facilitado en gran medida la labor de posicionamiento de los robots submarinos autónomos, así como la tarea de calcular las trayectorias. El gran problema de esta última tecnología es el hecho de que este sistema requiere de la presencia del vehículo en la superficie para poder actualizar los datos y los algoritmos de navegación; por ello es común ver el GPS combinado con sensores acústicos.

Uno de los puntos más limitantes de este tipo de robot es su autonomía, algo con lo que se está lidiando ya no sólo en este ámbito si no de forma generalizada, siendo la parte que más recursos se lleva en investigación. Su durabilidad y dimensiones son los factores más problemáticos, pues se requiere un mayor aguante, pero un menor tamaño.

Si se piensa en low-cost AUVs vienen a la mente las baterías de Niquel, litio o zinc. Existen algunas propuestas como el aluminio o las células de oxígeno, capaces de aguantar bajas temperaturas. Las energías renovables nos ofrecen algunas posibilidades como puede ser la energía solar.

Estos robots, también ofrecen limitaciones en el tipo de sensores que pueden llevar equipados; esto se debe a que, a parte de los factores que impone el hecho de ser un robot autónomo como pudieran ser la versatilidad, la rapidez, la necesidad de un bajo consumo y un tamaño reducido, con un alto rendimiento, se suman aquellas especificaciones exigidas por el entorno, siendo la presión, la temperatura y el grado de humedad las principales de ellas. Bien es cierto que, gracias al crecimiento de la electrónica, se pueden encontrar sistemas completos de bajo coste que cumplen dichas exigencias

Su principal uso se basa en la exploración, en la recogida de muestras o incluso en la arqueología. Toda la información recopilada por el AUV se almacena en la memoria para ser analizada posteriormente. También son comunes sus usos militares, como la detección de minas e incluso la vigilancia submarina.

Existen concursos para el desarrollo de la robótica submarina, como por ejemplo el AUVSI (Association for Unmanned Vehicle Systems International) que ha llegado a entregar más de cien mil dólares americanos en premios durante un solo año. Se presentan cerca de 200 equipos representados por más de 2000 estudiantes de todo el mundo.

* + **Robots Submarinos Operados para Intervenciones (IAUVs).**

Este tipo de robots fueron inicialmente diseñados para las labores de observación, sin embargo, últimamente ha surgido un creciente interés en dotarles con la capacidad de manipulación. Los IAUVs son una solución económica al problema que presentan los ROVs, en adición la navegación sería más sencilla debido a la libertad que ofrece la ausencia del cordón umbilical.

Ejemplos de IAUVs son los proyectos ALIVE (2003), SAUVIM (2009) y RAUVI (2010). El primero de ellos navega de forma autónoma hasta el área dónde debe realizar la tarea asignada, una vez ha llegado a su destino cambia su modo de operación a Control Supervisado y gracias a la tecnología acústica se comunica con el operario y realiza las operaciones asignadas. En cambio, el robot GIRONA 500 perteneciente al proyecto RAUVI, realiza en primer lugar la exploración del lugar objetivo recopilando datos vía acústica y visual. A continuación, el vehículo asciende a la superficie, donde se reconstruye el escenario a partir de la información obtenida y se selecciona el objetivo que se desea intervenir.

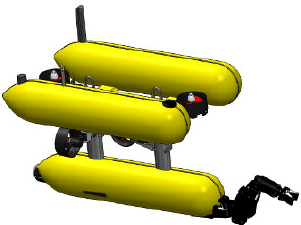


Figura 8: Robot Girona 500

#### **1.1.2.2. ROS.**

ROS se ha implantado como uno de los principales frameworks utilizados para la elaboración de Robots; su infraestructura de comunicaciones, las opciones que ofrece y su sistema basado en nodos, topics, servicios y mensajes hacen de él una herramienta muy poderosa en el desarrollo de la robótica y ha conseguido que desde su aparición en 2006 su uso se haya extendido de forma imparable.

Numerosos son los ejemplos de aplicaciones robóticas basados en este pequeño sistema operativo que corre por encima de los sistemas UNIX. En WillowGarage se pueden encontrar algunos ejemplos como como el WillowGarage PR2 (Personal Robot 2). Este complejo, formado por distintas empresas y bajo el cual se encuentra el custodio de ROS, cerró sus puertas en 2014.

La empresa española Robotnik, especializada en robots móviles programados con ROS, ofrece algunos ejemplos como FREIGHT, una plataforma móvil diseñada para hacer frente a la demanda de la industria de la logística con estación de carga.

En el año 2011, Scott Heath, un ingeniero electrónico de la Universidad de Queensland, participa en el proyecto Lingodroids que busca el desarrollo de "La comunicación efectiva entre agentes con diferentes capacidades cognitivas”. En el artículo A Rat in the Browser describe un robot controlado desde internet inspirado biológicamente y cuya arquitectura está basada en ROS. Este dispositivo consta de los elementos necesarios para poder hacer SLAM (Simultaneous Localization And Mapping) una forma de mapeo muy común.

En 2013, Elvis Ruiz, también ingeniero electrónico, trabaja en la investigación de ROS, Robots móviles y navegación. En colaboración con la Universidad de Simón Bolívar hacen público un esquema de control destinado a un robot Roomba 4400, basado en ROS, que ofrece la posibilidad de comunicar procesos y un Kinect, pudiendo, el robot, comprobar su orientación gracias a la visión proporcionada.

También en 2013, Nuno L. Ferreira junto a algunos componentes del IEEE desarrollan un robot móvil capaz de medir la densidad de gas tóxico y humo, así como el gradiente de temperatura con el objetivo de facilitar las labores de búsqueda y rescate de seres humanos tras un desastre. ROS tiene la tarea de integrar distintos elementos y de interpretar los datos de los sensores y la teleoperación.

Ya en 2014, el investigador en robótica y sistemas inteligentes del Instituto de Sistemas y Robótica de la Universidad de Coímbra, André Araujo, desarrollo el robot Taxbox. Se trata de un robot móvil que tiene una arquitectura totalmente basada en ROS. Su interacción con el entorno esta basado en los sensores ultrasónicos que lleva equipados.

A nivel doméstico está Kobuki que da la opción de implementar un sistema de comportamientos útil para cualquier robot desarrollado con ROS.

#### **1.1.2.2. Simulación de robots.**

La simulación tiene un rol crucial previo en los proyectos robóticos gracias a que brinda la oportunidad de realizar las operaciones de validación y verificación de aplicaciones robóticas antes de la realización física del robot, suponiendo un importante ahorro en tiempo y en dinero. La simulación virtual no solo tiene un carácter preventivo, sino que también tiene una importante aplicación a la hora de entrenar el manejo y entender las características del dispositivo antes de utilizar el robot real.

Simuladores basados en comportamientos robóticos dan la oportunidad de elaborar mundos simplificados con objetos rígidos y programar dispositivos que interactúen con ellos. A veces, un ambiente con condiciones extremas u operaciones en un lugar remoto no ofrecen la opción de verificar todos los procedimientos y problemas físico-mecánicos ni por esa razón testear de una forma únicamente experimental. Pueden, por tanto, darse circunstancias en las que las decisiones a tomar se basen solo en los análisis de resultados de una simulación.

El modelado 3D se ha convertido en una de las aplicaciones más populares; su representación requiere de buenos motores de leyes físicas y elevados gráficos para alcanzar una simulación decente del robot y el entorno de alrededor. Por lo tanto, cada dispositivo tendrá unas propiedades físicas y otras gráficas. Las técnicas de emulación e implementación robótica necesitan de una descripción analítica del sistema. A pesar de ello, hay ocasiones en la que no se puede simular un entorno cien por cien realista pues no se dispone de todos los parámetros. Se usa entonces técnicas de manejo de datos basadas en la probabilidad y en la estadística

Hoy en día, la simulación de robots da una gran diversidad de elementos: distintos tipos de robots como los UAV, UGV, AUV, brazos robóticos, humanoides, avatares, manos robóticas...; actuadores de todo tipo (actuadores de fuerza controlada, cadenas cinemáticas genéricas...), sensores que ayudan con la percepción del robot (Odometría, GPS, IMU, lásers, cámaras, receptores, emisores...), subactuadores, mecanismos, herramientas robóticas, manipuladores.... Existen incluso, simuladores utilizados en tareas en el espacio como satélites o aeronaves.

Se han estado desarrollando una gran variedad de técnicas para el modelado y la emulación, y para ello se han creado diferentes softwares. Morse, por ejemplo, es un simulador de código abierto que tiene el soporte completo en ROS. Utiliza OpenGL como motor 3D y tiene un render realista, que hace que su simulación sea fácil de comprender. El emulador utiliza comandos en línea, pero también puede usarse Python para manejar los robots. Otro simulador parecido es Gazebo, cuyo soporte completo también es en ROS. Gazebo da la oportunidad de simular con precisión y eficacia grandes poblaciones de robots en diversos entornos interiores y exteriores. Proporciona la realimentación de los sensores y las interacciones físicas entre los objetos, tratados como cuerpos rígidos, a través de unos gráficos de alta calidad e interfaces programables. Además, este emulador dispone una comunidad activa. Otra posibilidad es Webots, basado en ODE (Open Dynamics Engine) para la detección de colisiones, ofreciendo una simulación precisa en velocidad, inercia y fricción. V-Rep se trata de otro simulador muy famoso. Esto se debe a su gran versatilidad, no solo porque los controladores pueden ser escritos en, prácticamente, cualquiera de los lenguajes de programación existentes, sino que además tiene una arquitectura distribuida de control: todos los objetos pueden ser controlados independientemente por un script, un nodo de ROS, un plugin, un cliente remoto o el cliente.

## **1.2. Objetivos del proyecto**.

El presente proyecto tiene como objetivo principal:

* Diseñar un entorno y herramientas viables para el mapeado de las minas Kaatiala del robot submarino UNEXMIN para el simulador Gazebo.

Los objetivos parciales que se deben alcanzar para la consecución del objetivo global son:

* Conocimiento y aprendizaje del framework ROS y sus características.
* Conocimiento de las librerías y algoritmos de C++ utilizados en ROS.
* Completar los sensores del robot mediante la adicción de un láser.
* Programar los algoritmos necesarios para establecer el movimiento del robot UNEXMIN obedeciendo a los comandos enviados por el teclado del ordenador.
* Realizar un conversor de formato de datos.
* Establecer los frames necesarios.
* Diseñar las minas del entorno donde el vehículo submarino desempeñará sus tareas.

# 

# 2. CONCEPTOS PREVIOS.

## **2.1. ROS**.

### **2.1.1. Introducción**

Según se define en la página de ROS, se trata de un sistema flexible utilizado para la escritura del software de los robots. Engloba una colección de herramientas, bibliotecas y aplicaciones que tienen como objetivo simplificar la tarea de elaboración de un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas.



Figura 9: Logotipo de ROS

ROS (Robot Operating System) se sustenta en una arquitectura de grafos donde todo el procesamiento tiene lugar en los llamados nodos, que son capaces de recibir, mandar y multiplexar los mensajes recibidos. La librería está originalmente diseñada para los sistemas UNIX, como pudiera ser Ubuntu de Linux, aunque se está llevando a cabo su adaptación a otros sistemas como MAC OS X, Fedora, Microsoft Windows, Arch, Gentoo, Debian, OpennSUSE o Slackware.

ROS se puede dividir en dos partes básicas: la correspondiente al sistema operativo, ya mencionado, y ros-pkg, una agrupación de paquetes que es aportada por la comunidad de usuarios. Se engloban en conjuntos denominados pilas o stacks y son los encargados de proporcionar las funciones de localización, mapeo, planificación, etc.

Proporciona dos opciones para poder acceder a su API (Application Programming Interface), son los lenguajes de programación C++ y Python.

Se trata de un software libre y que se encuentra bajo licencia BSD. Según sus términos concede libertad para el uso comercial y de investigación. Sin embargo, las suites de paquetes aportados por los usuarios se pueden encontrar bajo una gran variedad de licencias diferentes.

### **2.1.2. Versiones**

En mayo de 2016 se lanzó la versión Kinetic Kame de ROS para Ubuntu 16.04, última actualización disponible del sistema operativo de Linux. Kinetic también se puede ejecutar en Ubuntu 15.10 y en otras plataformas como Debian, Android, Windows y MAC OS X.

Dicha versión ofrece una variedad más amplia de paquetes y aplicaciones que sus predecesoras además de modificaciones en funciones como roslaunch y ros::Duration::sleep(). Aparte de otras muchas variaciones no mencionadas cabe destacar la disponibilidad de la versión 7 de Gazebo, la más actual de todas.

En el presente trabajo se han utilizado las versiones, ya mencionadas, de Kinetic kame para ROS, Ubuntu 16.04 para Linux y 7 para Gazebo. Al disponer de las últimas actualizaciones en todos los sistemas y aplicaciones se puede asegurar la compatibilidad del proyecto con futuros trabajos y actualizaciones del software.

### **2.1.3. Conceptos**

El funcionamiento de ROS se sustenta en 3 pilares básicos: los sistemas de archivos, la computación gráfica y la comunidad de usuarios.

En el nivel correspondiente al sistema de archivos (recursos pertenecientes al propio programa) se encuentran:

* Paquetes o packages: son la unidad principal para organizar software en ROS. En su interior se pueden encontrar nodos, bibliotecas, archivos de configuración, conjuntos de datos y en general todo tipo de información que sirve para la organización del software.
* Metapaquetes o metapackages: son paquetes especializados que agrupan un conjunto de paquetes relacionados entre sí.
* Manifiestos del paquete o package manifest: se tratan de archivos xml en los que se proporcionan metadatos de un paquete como su nombre, la versión, descripción, licencia, dependencias, etc.
* Repositorios: son colecciones de paquetes que comparten un sistema VCS (sistema de control de versiones) común.
* Tipos de mensajes: describen los mensajes almacenados en my\_package / msg / MyMessageType.msg. Definen las estructuras de datos para los mensajes enviados en ROS.
* Tipos de servicio: describen los servicios almacenados en my\_package / srv / MyServiceType.srv. Definen la solicitud y las estructuras de datos de respuesta para los servicios de ROS.

La computación a nivel gráfico tiene la responsabilidad de procesar todos los datos. Se basa en los conceptos de nodos, maestro, mensajes, servicios, topics y bags:

* Nodos: los nodos son procesos que realizan algún tipo de cómputo. ROS está desarrollado para ser modular en una escala básica. Un robot basado en ROS tendrá, por lo general, muchos nodos. Generalmente se escriben con el uso de bibliotecas como roscpp o rospy.
* Maestro o máster: proporciona servicios de nombre y registro para el resto de los nodos en el sistema ROS. Realiza un seguimiento de publicadores y subscriptores a los topics y servicios. El papel del Maestro es permitir que los nodos individuales de ROS se encuentren entre sí. Una vez que estos nodos se han ubicado, utilizan entre ellos una comunicación peer-to-peer. El máster se ejecuta comúnmente con el comando roscore, que lo carga junto con otros componentes esenciales.
* Tema o topic: se trata de canales donde los nodos intercambian mensajes. Los topics tienen una semántica de publicación / suscripción anónima, que desacopla la producción de información de su consumo. En general, los nodos no saben con quiénes se están comunicando. Un nodo que esté interesado en un tipo determinado de datos se subscribe al topic. Puede haber varios editores y suscriptores. Los nodos que necesitan realizar llamadas a procedimientos remotos, es decir, recibir una respuesta a una solicitud, deberían usar servicios en su lugar.

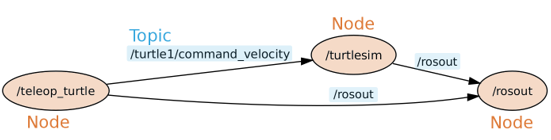


Figura 10:Ejemplo de comunicación entre nodos y topics.

* Servicios: el modelo de publicación / suscripción es un paradigma de comunicación muy flexible, pero su transporte unidireccional de muchos a muchos no es apropiado para las interacciones de solicitud / respuesta, que a menudo se requieren en un sistema distribuido. La solicitud / respuesta se realiza a través de un Servicio, que está definido por un par de mensajes: uno para la solicitud y el otro para la respuesta. Un nodo proveedor de ROS ofrece un servicio con un nombre, y un cliente llama al servicio enviando el mensaje de solicitud y esperando la respuesta. Las bibliotecas cliente normalmente presentan esta interacción al programador como si fuera una llamada de procedimiento remoto. Los servicios se definen utilizando archivos srv, que se compilan en el código fuente mediante una biblioteca cliente ROS. Un cliente puede hacer una conexión persistente a un servicio, lo que permite un mayor rendimiento a costa de una menor robustez.
* Mensajes: los nodos se comunican entre sí mediante la publicación de mensajes. Un mensaje es una simple estructura de datos que comprende campos tipados. Los tipos primitivos estándar (entero, punto flotante, booleano, etc.) son compatibles, al igual que las matrices de estos. Los mensajes pueden incluir estructuras y matrices anidadas arbitrariamente (al igual que las estructuras C). Los nodos también pueden intercambiar un mensaje de solicitud y respuesta como parte de una llamada de servicio ROS. Estos mensajes de solicitud y respuesta están definidos en archivos srv.
* Bags o bolsas: una bolsa es un formato de archivo en ROS utilizado para almacenar datos de mensaje ROS. Las bolsas, llamadas así por su extensión .bag, tienen un papel importante en ROS, y se ha desarrollado una extensa variedad de herramientas para que se puedan almacenar, procesar, analizar y visualizar.

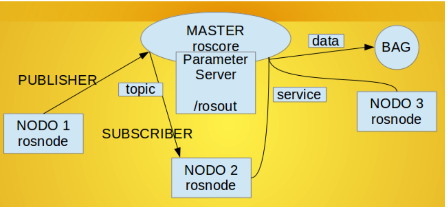


Figura 11: Esquema general de comunicación en ROS.

La tercera base se sustenta en los conceptos pertenecientes a la comunidad de ROS. Se trata de recursos que permiten a las comunidades separadas intercambiar software y conocimiento:

* Distribuciones: las distribuciones engloban a un conjunto de pilas versionadas. Permiten una instalación más rápida y sencilla del software.
* Repositorios: ROS tiene una red federada de repositorios de código en las que los usuarios pueden desarrollar y compartir sus creaciones.
* ROS Wiki: es un foro donde documentarse sobre el sistema. Todo usuario que posea una cuenta puede aportar documentación, tutoriales, correcciones o actualizaciones, etc.
* ROS respuestas: se trata de una página web con formato Q & A sobre ROS.
* Blog: el blog de de Willow Garage ofrece actualizaciones periódicas.

Es necesario conocer las posibles extensiones o formatos que pueden tener los archivos en ROS :

* Launchfiles: son muy comunes en ROS tanto para usuarios como para desarrolladores. Proporcionan una forma sencilla de iniciar varios nodos y un maestro, así como otros requisitos de inicialización como el setting parameter. Para ejecutar estos archivos se utiliza el comando roslaunch especificando el paquete en el que se encuentran los archivos de inicio, seguido del nombre del archivo de inicio o especificando su ruta. Para su escritura se utiliza un formato específico XML. Su extensión es .launch.
* YAML (YAML Ain’t Markup Language): es un lenguaje de serialización de datos diseñado para ser entendible por los seres humanos y que tiene un buen funcionamiento con los lenguajes de programación modernos utilizados en tareas cotidianas comunes. Su extensión es .yaml y es utilizado en la definición de parámetros.
* Xacro (XML Macros): es un lenguaje de macros XML. Con xacro, se pueden construir archivos XML más cortos y más legibles mediante el uso de macros que se expanden a expresiones XML más grandes. Son utilizados principalmente para desarrollar modelos de robots en URDF. Su extensión es .xacro.
* URDF (Unified Robot Description Format): es un formato XML para representar el modelo de un robot. Tiene extensión .urdf.
* Gazebo: es una extensión del formato de descripción del robot URDF, utilizado con fines de simulación en Gazebo. Tienen extensión .gazebo.
* World: la extensión .world definen archivos basados en XML que describen mundos virtuales que se pueden ejecutar con el simulador Gazebo.

### **2.1.4. Instalación de ROS**

La instalación de ROS Kinetic se realiza los pasos indicados por la wiki de su página:

1. El primer paso a seguir es la actualización del sistema, para ello se utiliza el siguiente comando: “*sudo apt-get update*”.
2. Se disponen de varias opciones para la instalación de ROS; para la instalación completa, utilizada en este proyecto, hay que introducir en el terminal lo siguiente: “*sudo apt-get install ros-kinetic-desktop-full*”.
3. Una vez se ha completado la instalación es necesario inicializar rosdep mediante “*sudo rosdep init*” seguido del comando “*rosdep update*”. Rosdep permite instalar fácilmente dependencias del sistema para cualquier fuente que se desee compilar y es necesario para ejecutar algunos componentes principales en ROS.
4. Al tratarse de la primera vez que se utiliza ROS y por tanto no disponer de ninguna versión anterior, añadimos la variable de entorno: “*source /opt/ros/kinetic/setup.bash*”.
5. A continuación, se crea un espacio de trabajo mediante el comando mkdir. Gracias a catkin\_make se crea un fichero CMakeLists.txt que nos permite lanzar los ejecutables. Además aparecen otras dos carpetas ( build y devel) con diversos archivos de configuración.
6. Finalmente incluimos setup.bash: “*source devel/setup.bash*”.

## **2.2. Gazebo**

Gazebo, que nace bajo el nombre de Gazebo Project, es un simulador 3D, cinemático dinámico y multi-robot que permite emular el comportamiento de los robots en ambientes complejos y realistas.



Figura 12: Logotipo de Gazebo.

Sus características principales son:

* Se trata de un software libre, parcialmente financiado por Willow Garage.
* Se puede ejecutar con ROS y Player y utilizar las APIs con el fin de controlar los robots simulados.
* Simulación realista de cuerpos rígidos. Es posible la interacción de los robots simulados con el entorno y del entorno con ellos.
* Se pueden crear y emular modelos de robots (URDF) y cargarlos durante la ejecución.
* Capacidad de desarrollar mundos, personalizando las características de las colisiones, obstáculos e incluso la gravedad.
* Ofrece distintos plugins que se pueden añadir a los modelos como sensores de odometría, de fuerza, de contacto, láseres y cámaras.

Como motores principales utiliza:

* Open Dynamics Engine (ODE): es una biblioteca de código abierto de alto rendimiento para simular la dinámica del cuerpo rígido. Es estable, maduro e independiente de la plataforma, con una API C / C ++ fácil de usar. Tiene tipos avanzados de juntas y detección de colisión integrada con fricción. ODE es útil para simular vehículos, objetos en entornos de realidad virtual y criaturas virtuales. Actualmente se utiliza en muchos juegos de ordenador, herramientas de autoría 3D y herramientas de simulación.
* OGRE (Object-Oriented Graphics Rendering Engine): se trata de un motor de renderizado 3D orientado a escenas. Está basado en C++ y software libre.

Gazebo tiene su propia pila compatible con ROS denominada “*simulator\_gazebo*”. Este stack se divide en una serie de paquetes que poseen diferentes funciones:

* Gazebo: Se ejecuta como un nodo más. En su interior se encuentra la versión actualizada de Gazebo Project para ROS.
* Gazebo\_msgs: contiene estructuras de datos de mensajes y servicios para interactuar con Gazebo desde ROS.
* Gazebo\_plugins: Complementos de Gazebo, independientes del robot, que permiten utilizar sensores, motores y componentes dinámicos reconfigurables.
* Gazebo\_ros: Proporciona complementos de ROS que ofrecen editores de mensajes y servicios para interactuar con Gazebo a través de ROS.

En el presente proyecto es necesario utilizar SITL (Software In The Loop), debido a que no se dispone de hardware. Se trata de un tipo de simulación que permite ejecutar el robot sin ningún tipo elemento físico. Es una compilación del código de piloto automático que utiliza un compilador ordinario de C ++, que proporciona un ejecutable y permite probar el comportamiento del código. Actualmente se puede combinar una simulación en Gazebo con SITL.

## **2.3. RVIZ**

Rviz, abreviatura de ROS Visualitation, es una poderosa herramienta de visualización 3D para ROS. Permite al usuario ver el modelo simulado, registrar la información perteneciente a los sensores del robot y reproducir dichos datos. Al observar lo que el robot está viendo, pensando y haciendo, el usuario puede depurar una aplicación a partir de las entradas proporcionadas por los sensores.

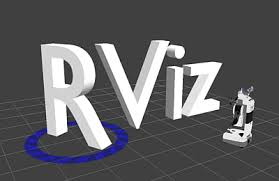


Figura 13: Logotipo de Rviz

Rviz muestra datos en 3D de sensores cómo cámaras estéreo, láser, Kinects y otros dispositivos en forma de nubes de puntos (Point Colud) o imágenes de profundidad. Los datos de sensores 2D de cámaras web, cámaras RGB y láser 2D se pueden reproducir en Rviz como datos de imagen.

Si un robot se está comunicando con una estación de trabajo que está ejecutando Rviz, este plugin mostrará la configuración actual del robot. Los topics de ROS se muestran como representaciones en vivo basadas en los datos publicados por cualquier cámara, sensor de infrarrojos y escáner láser que formen parte del sistema. Rviz proporciona una interfaz gráfica de usuario (GUI, Graphical User Interface) configurable que permite reproducir selectivamente aquellos datos que el operario considere oportuno.

Para poder utilizarlo es necesario conocer sus frames de coordenadas:

* Fixed frame: es el sistema de coordenadas que se considera como referencia para reproducir todos los datos. Se sitúa en la zona izquierda de Global Options. Es necesario asignárselo a un punto fijo, por ello es común utilizar como referencia el mundo en el que se esté reproduciendo la simulación.
* Target frame: Es el sistema de coordenadas que asignamos a la vista. Puede ser la base del robot, ya que sí se permite que se encuentre en movimiento. Se dispone de distintas vistas entre las que el usuario puede escoger en función de lo que quiera visualizar.

## **2.4. Blender**

Blender es un software que posee herramientas de gráficos 3D profesionales, gratuitas y de código abierto que se pueden utilizar para crear películas animadas, efectos visuales, arte, modelos impresos en 3D, aplicaciones 3D interactivas y videojuegos. Las características del programa incluyen modelado 3D, texturizado, edición de gráficos ráster, rigging y skinning, simulación de fluidos y humo, simulación de partículas, simulación de cuerpos, escultura, animación, seguimiento de cámara, rendering, edición de video y composición. También presenta un motor de juego integrado.

Los lanzamientos oficiales de Blender para Microsoft Windows, MacOS y Linux, así como también un puerto para FreeBSD, están disponibles en versiones de 32 bits y de 64 bits.

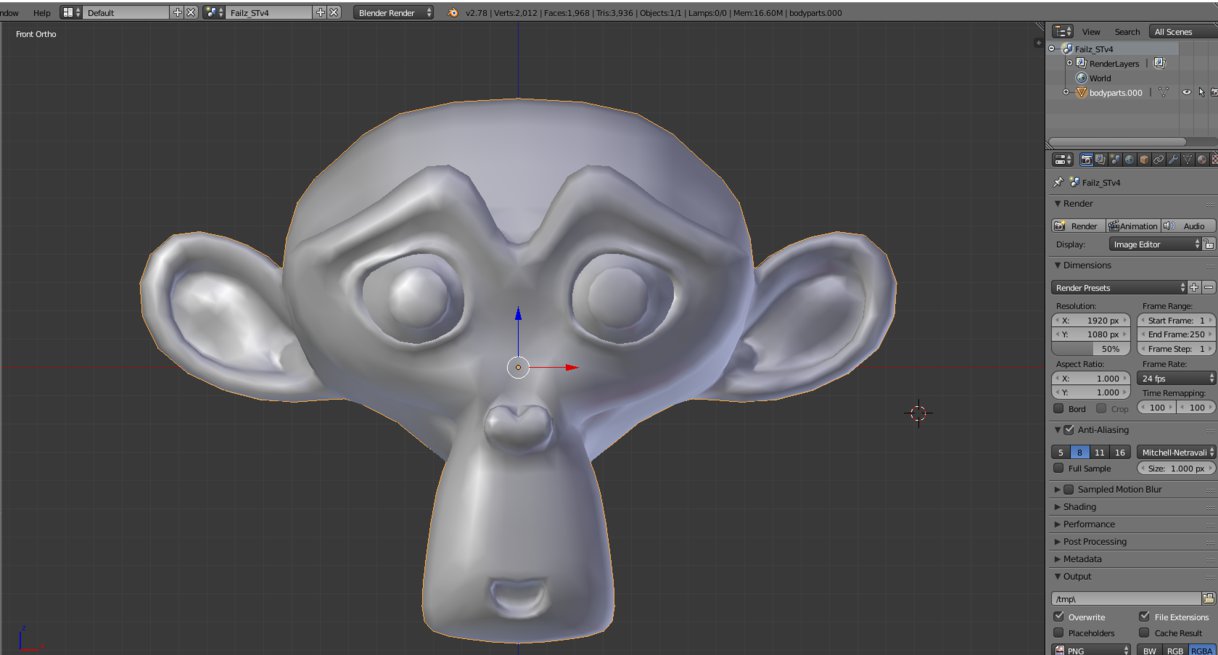


Figura 14: Interfaz y logotipo Suzanne de Blender

### **2.4.1. Características**

El software, realmente contiene características que son propias de las aplicaciones 3D de alta gama. Entre sus capacidades están:

* Soporte para una gran variedad de figuras primitivas geométricas, incluyendo mallas poligonales, modelado de superficie de subdivisión rápida, curvas de Bezier, superficies NURBS, metaballs, icospheres, escultura digital de múltiples resoluciones, fuentes de contorno, y un nuevo sistema de modelado llamado B-mesh.
* Motor de renderizado interno con representación de escaneo, iluminación indirecta y aislamiento ambiental que se puede exportar en una amplia variedad de formatos.
* Un motor de renderizado llamado Cycles, que se aprovecha de la GPU.
* Integración con varios motores de renderización externos a través de plugins.
* Herramientas de animación de fotogramas clave que incluyen cinemática inversa, armadura (esqueleto), gancho, curvas y deformaciones, animaciones de formas, animación no lineal, restricciones y ponderación de vértices.
* Herramientas de simulación para la dinámica del cuerpo blando, incluida la detección de colisiones de malla, dinámica de fluidos, simulación de humo, dinámica del cuerpo rígido, generador oceánico con ondas.
* Un sistema de partículas que incluye soporte para cabello.
* Modificadores para aplicar efectos no destructivos.
* Formatos Python para creación de herramientas y desarrollo de prototipos, lógica de juegos, automatización de tareas y herramientas personalizadas.
* Edición básica no lineal de audio / video.
* Blender Game Engine, un subproyecto, ofrece características de interactividad como detección de colisiones, motor de dinámica y lógica programable. También permite la creación de aplicaciones independientes en tiempo real que van desde la visualización arquitectónica hasta los videojuegos.
* Un compositor basado en nodos totalmente integrado dentro de la canalización de renderizado acelerado con OpenCL.
* Las texturas procedimentales y basadas en nodos, así como diversos tipos de pintura.
* Control en tiempo real de la simulación y renderización de la física.
* Cámara y seguimiento de objetos,

# 3. DISEÑO DEL ENTORNO EN GAZEBO.

En este apartado se explica como se ha procedido para establecer el entorno de simulación, en el cuál se realizan el resto de pruebas y ensayos pertenecientes a este proyecto.

## **3.1. Integración del Velodyne.**

En el presente trabajo se partía de un modelo UNEXMIN que tenía incorporado un láser Hokuyo. Dicho equipamiento proporciona una visualización en 2D cuando se necesitaba un plugin en 3D para el tipo de mapeado que se quería realizar.

Existen dos soluciones posibles a ese problema. La primera se trata de insertar una junta móvil al láser Hokuyo de tal forma que, mediante un movimiento de rotación de dicha junta, se pueda realizar un escaneo 3D. En el segundo caso, se optaría por diseñar un láser que ofrezca directamente una nube de puntos en las dimensiones requeridas, se trata del Velodyne de Gazebo.

El proyecto está basado en un archivo de extensión .world (UNEXMIN\_MINE.world) que se ejecuta en Gazebo y que, en su código, realiza una serie de llamadas a los modelos que definen los distintos componentes (extensión .sdf). Estos a su vez requieren de otros elementos por lo que realizan sus propias llamadas. En el caso actual, la jerarquía establecida se puede resumir de la siguiente forma: UNEXMIN\_MINE.world → UNEXMIN.sdf → Hokuyo\_Unexmin.model.sdf

El último objeto es el correspondiente al láser 2D y por tanto, también el código a modificar. El documento del nuevo láser se escribe en un lenguaje XML y se programa con la ayuda de los tutoriales pertenecientes al nivel intermedio de Gazebo, tomando como base el primer modelo. Se establece así, la posición inicial del objeto, las colisiones, el tipo de láser además de otros aspectos importantes cómo el número de rayos horizontales y verticales que proporciona el Velodyne, los ángulos máximo y mínimo para ambos casos, su resolución y el rango de alcance. Se nombra también el topic sobre el que se publicarán las entradas que recibe en formato PointCloud; en este caso *“/UNEXMIN/laser/scan”*.

Para que el láser que se acaba de configurar se encuentre fijado en el modelo UNEXMIN es necesario cambiar el nombre al apartado <child> e incluir al Velodyne.

### **3.1.1. Simulación del Velodyne.**

Tras diseñar el láser se procede a su simulación con el objetivo de comprobar que se ha desarrollado correctamente, tanto en su apartado visual como en el funcional.

Para ello es necesario introducir el nombre del documento que describe al Velodyne en el fichero de configuración y posteriormente ejecutar Gazebo.

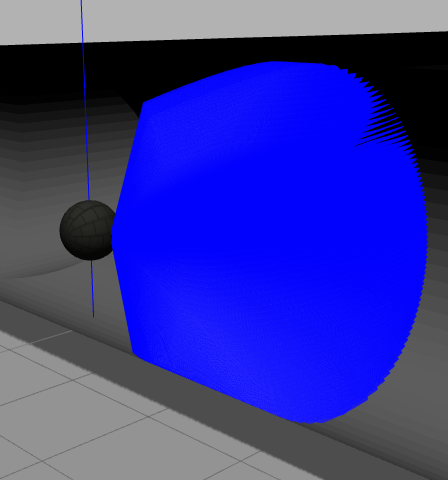


Figura 15: Visualización del láser Velodyne 3D en Gazebo

## En la imagen se observa un nuevo láser 3D capaz de cubrir el diámetro completo de las minas, y que se encuentra fijado al cuerpo del robot UNEXMIN.

Para realizar las comprobaciones de funcionamiento se utiliza el comando “*rostopic info*” de ROS. Gracias a dicho parámetro podemos comprobar si se está utilizando un topic, el tipo de información que contiene, y qué aplicaciones se comunican con él.

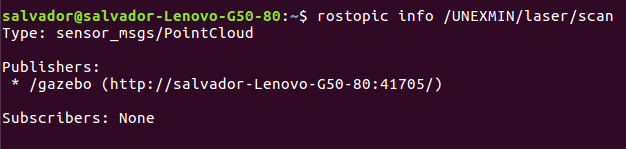


Figura 16: Captura de la terminal de linux ante el comando "rostopic info"

Se observa que el formato de los mensajes que se publican en el topic del Velodyne son del tipo “*sensor\_msgs/PointCloud*”. También comprobamos que la aplicación que publica la información es Gazebo, donde se está ejecutando la simulación del láser.

En Rviz se puede visualizar los datos que proporciona el Velodyne. Para ello se introduce la sentencia “*rosrun rviz rviz*” en la terminal y se introducen los siguientes cambios:

* Se añade un marcador del tipo “*PointCloud*”; con ese fin se siguen los siguientes pasos: Add → By display type → rviz → PointCloud.
* El valor de Fixed Frame se cambia a *“base\_link”*.
* En el apartado PointCloud se establece el topic “*/UNEXMIN/laser/scan*”.

Finalmente, en la pantalla se ve lo siguiente:

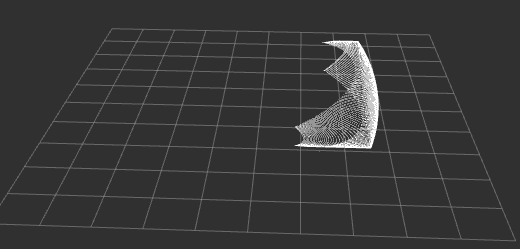


Figura 17: Visualización en Rviz de la nube de puntos correspondiente al topic "/UNEXMIN/laser/scan"

En la imagen se visualiza una nube de puntos que coincide con la intersección del láser con las paredes de las minas.

## **3.2. Diseño de las minas.**

En este apartado se explica el proceso de modelado de las minas. El diseño está basado en las minas de Kaatiala, dónde se llevará a cabo el proyecto real y a las cuáles se ha intentado asemejar el modelo 3D diseñado.

Para ello se ha utilizado el software Blender, un entorno de modelado 3D. El diseño se realiza a partir de cilindros de 10 vértices y distintos tamaños, además es necesario cambiar la opción de “*cap fill type*" a "*nothing*" para que los elementos se encuentren huecos por dentro. En este programa los elementos simples se basan en vértices, aristas y caras, de manera que para cualquier elemento añadido es necesario establecer el número de vértices. Al aumentar la cantidad de estos, la figura se vuelve más real, sin embargo, también es cada vez más compleja.

Para enlazar los cilindros se utilizan Pipe Joints, que poseen distintas formas (en cruz, en Y, en T, y con múltiples brazos). En todos los casos se puede ajustar la longitud de los elementos, así como su radio. Además, en algunos casos, por ejemplo en la junta Y, se puede personalizar el ángulo entre los brazos.

Blender posee varios modos de operación. En el caso presente se utilizan los modos “*Object mode*”, para insertar objetos y utilizar las operaciones básicas de traslación y rotación, necesarias para establecer los elementos en las posiciones requeridas, y el “*edit Mode*”, que nos permite distintas operaciones como seleccionar vértices, desplazarlos o realizar cualquier modificación.

Una vez los cilindros se encuentran en las coordenadas que se quieren, tienen la rotación apropiada, y gracias a la operación “*join*”, se juntan todos los elementos bajo un mismo nombre. Ya, en el modo Edit Mode se cambia la vista de los objetos, de tal forma que solo sean visibles las aristas. Finalmente, y con el objetivo de enlazar todos ellos, se seleccionan una a una todas las aristas y, mediante la opción de “*Bridge Edge Loops*” se establecen caras y aristas entre los objetos. Se recomienda que una vez estén dos cilindros preparados para realizar este proceso, se realice y no se espere a tener todo el mapa ya modelado para enlazar los elementos, ya que la selección de aristas se complicaría debido a la gran cantidad de ellas que se generarían.

Para rellenar aquellas caras que se encuentran vacías (bases de los cilindros) y corresponden a lugares el mapa en los cuales no hay salida, se seleccionan las aristas pertenecientes a la cara que se quiere cerrar y se pulsa la tecla *“F*”.

Se trata de un modelo de distintas profundidades, donde las distancias son aproximadas y no se disponía de todos los datos necesarios para realizar un modelo exacto.

Para concluir con el uso de Blender, y para proporcionar al modelado de las minas un diseño más cuidado se le añade un modificador llamado “*Corrective Smooth*”. Este modificador elimina aristas y vértices, generando pequeñas curvas dónde antes se encontraban los anteriores elementos.

### **3.2.1. Configuración de las minas para su uso en Gazebo**.

Para que el diseño anterior se pueda utilizar en el simulador Gazebo, es necesario exportar el modelo en formato Collada (.dae).

Como en el caso del modelo UNEXMIN, el archivo Mines, es llamado por UNEMIN\_MINE. En el código de extensión .sdf, se llama a los archivos de una carpeta denominada Meshes, donde ubicaremos los datos exportados del modelo diseñado bajo el nombre de tunnels\_final\_meshlab.dae, donde, además, se establece el material de los túneles.

# 4. ELABORACCIÓN DE UN NODO DE OPERACIÓN TELEMÁTICA.

## **4.1. Primeros pasos.**

Con el objetivo de realizar un nodo capaz de mandar comandos, recibidos por el teclado, al robot con el propósito de producir desplazamientos del modelo UNEXMIN por las minas, se realiza un pequeño proceso de investigación con el fin de descubrir qué topics utilizar, qué tipo de mensajes mandar y el comportamiento del robot al escuchar los mensajes utilizados. Para ello se utilizan los comandos rostopic, con todas sus variantes mientras se está ejecutando, de forma paralela, el simulador Gazebo.

El primer paso a seguir es el uso del comando “*Rostopic list*”. Gracias a él podemos ver una lista completa de todos los topics en los que se está publicando información.

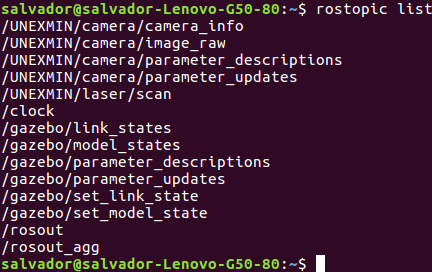


Figura 18: Captura del terminal de linux ante el comando "rostopic list".

Se observan temas de distintas características, entre los que se encuentran el topic correspondiente al láser diseñado (UNEMIN/laser/scan) y uno llamado “*/gazebo/set\_model\_state*”. Este es el topic que permite modificar la posición del robot.

A continuación se debe determinar el tipo de mensajes que utiliza para comunicarse. Para ello se introduce la sentencia: “*rostopic info /gazebo/set\_model\_state*”.

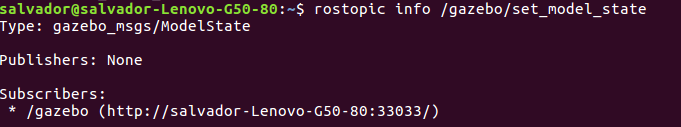


Figura 19: Reapuesta de la terminal de Linux ante el comando "rostopic info /gazebo/set\_model\_state".

Tras la ejecución se observa el tipo del topic, en nuestro caso “*gazebo/msgs/ModelState”*, y el número de subscriptores y editores que se relacionan con él.

Este tipo de mensaje tiene la siguiente estructura:

* string: model\_name
* geometry\_msgs/Pose: pose
* geometry\_msgs/Twist: twist
* string reference\_frame

Finalmente, y con el objetivo de ver cómo se comporta el robot tras publicar en dicho topic, ejecutamos el siguiente comando: *“rostopic pub -r 20 /gazebo/set\_model\_state gazebo\_msgs/ModelState '{model\_name: my\_UNEXMIN, pose: { position: { x: 5, y: 0, z: 0 }, orientation: {x: 0, y: 0, z: 0, w: 0 } }, twist: { linear: { x: 0, y: 0, z: 0 }, angular: { x: 0, y: 0, z: 0}, }}'”*

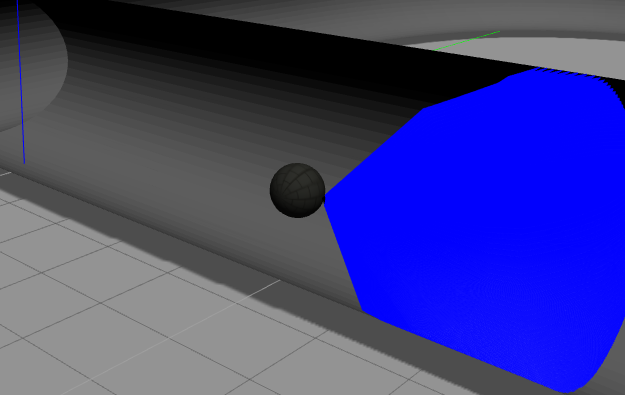


Figura 20: Visualización en Gazebo del desplazamiento de 5 unidades en el eje x

En la figura se muestra al robot desplazado unas 5 unidades en el eje x respecto al origen, como se ha publicado mediante la terminal en el topic set\_model\_state.

## **4.2. Realización de un Publisher C++**

Tras conocer y entender todos los elementos que intervienen en el proceso de publicación de topics, y tras haber conseguido mover el robot mediante el uso del terminal se procede a la elaboración de un nodo capaz de realizar la misma tarea.

Para ello, se crea una carpeta llamada movemente\_ws. Esta carpeta es el workspace destinado para este nodo. En su interior se encuentran los archivos CmakeLists.txt y Package.xml además de una carpeta que alojará el documento .cpp en el cual irá escrito el código del nodo correspondiente.

Como se trata de un programa que no posee demasiada extensión, ni algoritmos complicados, se puede introducir completamente dentro de la función main. En ella, en primer lugar, se llama a la función ros::init para iniciar el nodo. A continuación, se declara un objeto del tipo publisher que permite publicar en el topic que se desee, en este caso, “*Gazebo/set\_model\_state*”.

A continuación, en un bucle, que se ejecuta siempre y cuando ROS no se corrompa, se declara un mensaje del tipo “*gazebo\_msgs::ModelState*”. En él, se introducen los valores que se desea modificar.

En el nodo actual, y con el objetivo de mover el robot, se introduce una velocidad; gracias a dicho parámetro y en función de la frecuencia de ejecución del bucle y de un contador introducido en él, se actualiza el valor de la posición. Finalmente se publica el objeto de tipo mensaje.

Para la correcta compilación del documento es necesario incluir todas las librerías correspondientes a los mensajes utilizados, así como la librería estándar ros.h.

Uno de los casos más comunes de errores en la compilación se produce a la hora de realizar el archivo Cmakelists. En el código es necesario indicar el nombre del proyecto; además, dicho documento, se encarga de recopilar todos los paquetes relativos a las librerías utilizadas. También crea el elemento ejecutable, archivo que realmente realiza las tareas programadas en el código.

El documento Package.xml construye las dependencias indicadas en su código. Actualmente, en ROS, este archivo no se suele modificar de manera cuantiosa debido a que es común utilizar los mismos paquetes.

Para compilar el nodo es necesario encontrarse dentro del workspace e introducir el comando “*catkin\_make*”. Una vez no produzca errores, debe agregarse la fuente setup.bash para que finalmente, y gracias al comando rosrun, se pueda ejecutar correctamente.

### **4.2.1. Simulación y comprobación**

Tras la realización del nodo se comprueba su funcionamiento. Para ello se abren paralelamente dos terminales de Linux, en una se ejecuta Gazebo y en la otra el nuevo programa. Es importante que tras cerrar Gazebo se ejecute la función “*pkill gzserver*” para reiniciar completamente la aplicación.

En primer lugar y mediante el comando info de rostopic se comprueba que el único editor del topic utilizado es el realizado en el apartado anterior.

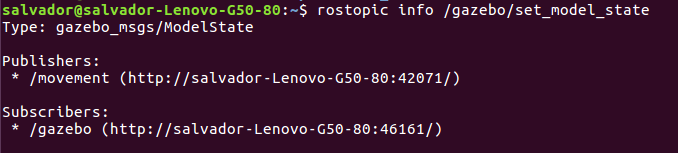
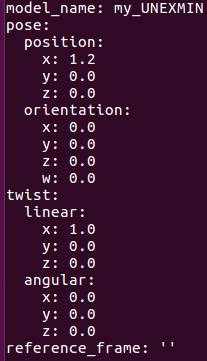
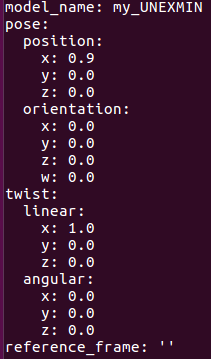


Figura 21: Respuesta de la terminal de linux ante el comando "rostopic info /gazebo/set\_model\_state"

A continuación, se analiza los elementos publicados gracias a la función echo del comando rostopic.

Figura 22: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del nodo de desplazamiento.



Se observa como la posición del robot aumenta en el eje x conforme pasa el tiempo con una velocidad lineal de 1.0 m/s.

## **4.3. Realización de un Publisher C++ controlado por el teclado.**

Para lograr el objetivo de crear un publisher controlado por el teclado es necesario documentarse sobre las distintas librerías que nos proporcionan acceso a los eventos proporcionados por el mismos. Se encontraron principalmente dos librerías, iostream.h y curses.h

### **4.3.1. Primer intento (iostream)**

En esta primera toma de contacto con la librería se intentó realizar un código basado en el anterior, ya que este funciona. Los valores se actualizan en función de un contador y la frecuencia de publicación.

Como principales funciones añadidas, destacan la introducción de initscr() que inicia una pantalla de comandos (donde se introducirán las órdenes deseadas) y la función getch(), que devuelve un carácter de tipo char que podemos almacenar y que corresponde con la tecla que se pulse en el teclado.

#### **4.3.1.1. Simulación y comprobación**

Aunque, como muestran las imágenes, el código cumple el objetivo principal de mover el robot a través del teclado, lo cierto es que se trata de una ejecución poco precisa y estética.

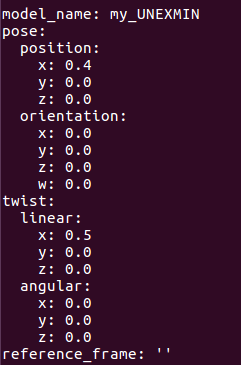


Figura 23:Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del primer nodo de operación telemática (posición final).

El robot UNEXMIN se mueve de forma discontinúa, de manera que se producen desplazamientos instantáneos en ambos sentidos del eje. Sin embargo, la posición del robot sí concuerda con los datos publicados. Esto se debe a que el código está basado en la frecuencia de publicación. Si utilizamos el comando "*echo*” para ver las publicaciones realizadas en el topic, se observa lo siguiente:

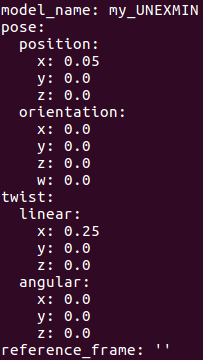
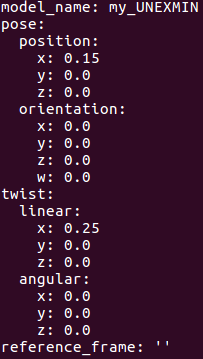
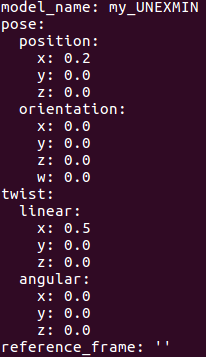


Figura 24:Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del primer nodo de operación telemática.

Tan solo hay 3 publicaciones previas a la posición final del modelo. Se corresponden a los comandos utilizados en este orden: “w-w-s-w”. La tecla w aumenta la velocidad lineal en el eje x hasta que se alcance el valor de 3m/s. Una vez alcanzada dicha magnitud se bloquea el valor; lo mismo sucede con el comando ‘s’ pero en el sentido contrario. El hecho de que tan solo haya 4 publicaciones implica que el valor de la posición no se puede actualizar en función de la frecuencia, ya que la sentencia getch() bloquea el proceso hasta que reciba un evento del teclado, provocando que haya saltos y comportamientos inesperados en la simulación.

Otro factor en contra es la estética de la ventana producida. En dicha ventana aparecen todos los comandos introducidos por el teclado, se debe pulsar la tecla “*Enter*” tras cada comando introducido y además, en ocasiones, se producen rebotes en las teclas creando desplazamientos indeseados.

Como solución a esta situación se plantearon dos caminos; la utilización de otra librería y la implementación de un algoritmo que no se base en la frecuencia de ejecución del bucle principal.

### **4.3.2. Segundo intento (Curses.h)**

Este código está basado en los códigos teleop\_key que circulan por internet. En primer lugar, se definen unas constantes que corresponden a las teclas que pretendemos utilizar. A continuación, se introduce otra de las librerías utilizadas “*termios.h*”.

Las funciones descritas en termios, describen una interfaz general que permite controlar la comunicación con puertos asíncronos, en el caso presente ese puerto es el teclado. Esta librería permite entrar en lo que se conoce como “*raw mode*”. Una vez dentro, se pueden introducir comandos por el teclado sin que sea necesario pulsar la tecla “*Enter*”, que, aunque en algunos programas puede resultar muy útil debido a la opción que ofrece de poder introducir frases enteras e incluso editarlas, evitando así la posibilidad de equivocarte, en el presente trabajo no interesa puesto que se pretende introducir letras aisladas con la intención de controlar un robot de la forma más fluida posible.

Se programa así, una función que permita terminar con la ejecución y salir del “*raw mode*”. Seguidamente, en una función callback, se declaran los objetos de clase nodo y publisher que permiten publicar en un topic. Además, en la misma función, se declara un objeto de tipo “*gazebo\_msgs::ModelState*” y se inicializan los valores fundamentales como puede ser el nombre del modelo.

Los siguientes comandos permiten entrar en el “*raw mode*”. Seguidamente se imprimen por pantalla los controles del robot para que cualquier usuario sea capaz de controlarlo.

En un bucle while, que se ejecuta siempre y cuando no se corrompa ROS, ya sea de manera intencionada o no, almacenamos el evento que venga por teclado gracias a la función read(). Aunque el sistema se encuentre en “*raw mode*” no implica que no se espere que se introduzca un comando por el teclado.

En el resto del bucle, se encuentra el algoritmo para mover el robot en función de la tecla introducida. Para ello, y como no se puede basar el código en la variación de la velocidad y en función de ella establecer la posición, se opta por un código en el que los desplazamientos se producen por pequeñas variaciones de posición. También se introducen velocidades de movimiento reducidas para que los cambios producidos se realicen de forma más fluida. Se introduce un retardo para evitar los rebotes en la introducción de comandos. Finalmente, en la última línea del código se publican los valores.

#### **4.3.2.1. Compilación**

Merece la pena comentar los aspectos de compilación del nodo, debido a que en Internet no se encuentra demasiada información de los pasos a seguir en las nuevas versiones de ROS y Ubuntu, y, como ya se mencionó anteriormente, suele ser uno de los apartados que más errores produce.

Además de los aspectos mencionados en el apartado 4.2 correspondiente a la creación de un publisher en C++, en este caso y debido a la inclusión de las librerías ncurses y termios es necesario introducir las siguientes líneas en el documento CMakeLists a continuación de los componentes de catkin:

* find\_package( PkgConfig REQUIRED )
* find\_package(Curses REQUIRED)

Además, hay que añadir lo siguiente:

* target\_link\_libraries(movement\_keyboard\_model2

${catkin\_LIBRARIES}

${Boost\_LIBRARIES}

${Curses\_LIBRARIES})

* target\_link\_libraries(movement\_keyboard\_model2 ${catkin\_LIBRARIES} ncurses)
* target\_link\_libraries(movement\_keyboard\_model2 ${ncurses++\_LIBRARIES})

Sin estas líneas la compilación falla y nos imposibilita la ejecución del nodo.

#### **4.3.2.2. Ejecución y simulación**

Al igual que en el primer modelo de operación remota mediante el teclado, este nodo publica en el “*topic /gazebo/set\_model\_state*”. Si se ejecuta el nodo en paralelo con Gazebo y lanzamos el comando “*echo*” de rostopic se observa lo siguiente:

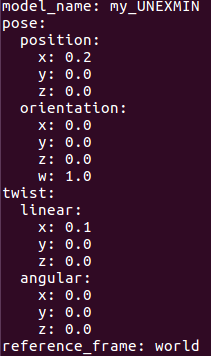


Figura 25: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del segundo nodo de operación telemática. Movimiento lineal.

En dicha imagen se muestra una velocidad lineal de 0.1 m/s (mucho más pequeña que en el primer nodo) con una posición en x de 0.2 m. Las teclas que se han pulsado son las equivalentes a las “w-w-s-w”. Se puede concluir por tanto que se producen desplazamientos mucho menores.

Gracias a este nodo, se consigue una visualización más fluida del desplazamiento del robot. El movimiento también se puede producir en el apartado de la orientación del modelo.

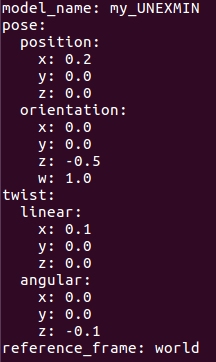


Figura 26: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del segundo nodo de operación telemática. Movimiento de rotación.

En cuanto a la pantalla adicional que se producía con el primer nodo, en este caso se ha eliminado, y en su lugar se ha introducido la siguiente interfaz en la misma ventana del terminal:

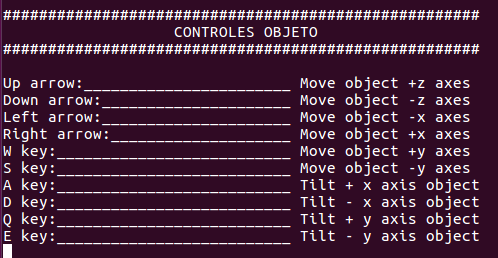


Figura 27: Instrucciones para desplazar telamáticamente el robot Unexmin mostradas en la terminal tras la ejecución del segundo nodo de operación telemática.

En el nodo actual no hace falta introducir la tecla “*Enter*” para lanzar comandos, estos no aparecen en la pantalla y se ha conseguido eliminar los rebotes.

# 5. MAPEADO.

En este apartado se trata el tema del mapeado de las minas diseñadas con el objetivo de obtener un mapa tridimensional que, de forma aproximada, se corresponda con el del modelo. Para ello se ha utilizado Octomap.

## **5.1. Octomap**

La biblioteca OctoMap ofrece un enfoque de mapeo de cuadrícula de ocupación en 3D, proporcionando estructuras de datos y algoritmos de mapeo en C ++ especialmente adecuados para la robótica. La implementación del mapa se basa en un octree y está diseñado para cumplir los siguientes requisitos:

* Es un Modelo 3D completo. Octomap puede modelar ambientes arbitrarios sin suposiciones previas al respecto. Los modelos de representación representan tanto áreas ocupadas como espacio libre. Las zonas desconocidas del entorno están implícitamente codificadas en el mapa. Si bien, la distinción entre espacio libre y ocupado es esencial para la navegación segura de un robot, la información sobre áreas desconocidas también es importante, por ejemplo, para la exploración autónoma de un entorno.
* Actualizable. Es posible agregar nueva información o lecturas de sensores en cualquier momento. El modelado y la actualización se realizan de manera probabilística. Esto tiene en cuenta el ruido del sensor o las mediciones que resultan de cambios dinámicos en el entorno, por ejemplo, debido a objetos dinámicos. Además, múltiples robots pueden contribuir al mismo mapa y un mapa previamente grabado es extensible cuando se exploran nuevas áreas.
* Flexible. La extensión del mapa no tiene que conocerse de antemano. De hecho, el plano se expande dinámicamente según lo necesitado. El mapa tiene varias resoluciones, por lo que, por ejemplo, un planificador de alto nivel puede usar un mapa aproximado, mientras que un planificador local puede operar con una resolución fina. Esto también permite visualizaciones eficientes que van desde vistas generales hasta vistas de primer plano detalladas.
* Compacto. El mapa se almacena de manera eficiente, tanto en la memoria como en el disco. Es posible generar archivos comprimidos para un uso posterior o un intercambio conveniente entre robots incluso bajo restricciones de ancho de banda.

La biblioteca OctoMap está disponible como una distribución de fuente autónoma para Linux (recomendado), Mac OS y Windows. También es compatible con ROS y RVIZ. Fue desarrollado por Kai M. Wurm y Armin Hornung, y actualmente es mantenido por Armin Hornung.

Se trata de paquetes independientes, por lo que para su utilización es necesario su instalación. Si ya se tiene instalado ROS solo hace falta ejecutar el comando “*sudo apt-get install ros-kinetic-octomap*”.

### **5.1.1. Octomap\_server**

Octomap\_server construye y distribuye mapas de ocupación 3D volumétricos como secuencia binaria de OctoMap y en varios formatos compatibles con ROS, como por ejemplo, para evitar o visualizar obstáculos. El mapa puede ser un archivo estático OctoMap .bt (como argumento de línea de comando) o puede construirse de forma incremental a partir de datos de entrada (como PointCloud2). Octomap\_server comienza con un mapa vacío si no se proporciona uno. En general, crea y publica solo sobre temas a los que se encuentra subscrito. En ocasiones se puede requerir de mucho tiempo para compilar mapas grandes, por ello se recomienda subscribirse a topics que realmente se necesitan (por ejemplo, en RViz) y configurar el parámetro "latch" como falso al realizar el mapeo.

Para su instalación, posterior a la de octomap, se ejecuta la siguiente sentencia en la terminal: “*sudo apt-get install ros-kinetic-octomap ros-kinetic-octomap-mapping*”.

### **5.1.2. Desarrollo del Archivo .launch**

Para poder ejecutar octomap\_server es necesaria la creación de un archivo de extensión .launch. El correcto desarrollo de este documento es esencial para el funcionamiento del mapeado, pues en él se especifican el frame fijo que se va a utilizar y a qué topic se va a subscribir para la realización del mapeo. Para ello se realizaron varios intentos.

#### **5.1.2.1.Primer intento**

Como primera opción se opta por modificar el fichero que se encuentra en la ruta /opt/ros/kinetic/share/octomap\_server/launch correspondiente a los paquetes instalados de ROS. Este documento, llamado octomap\_traking\_server.launch. Se trata del fichero que se utiliza por defecto para la ejecución de octomap\_server con el objetivo de utilizarlo como método de aprendizaje para ciertos tutoriales.

En el código se modifica el valor de la resolución, introduciendo la perteneciente al láser utilizado; también se cambia el dato del campo “*rango máximo del láser*”. Es necesario referenciarlo todo a un frame. Finalmente, en el apartado “*<!--remap from="cloud\_in" to= “ " /-→”* , entre las comillas finales, se escribe el topic del láser.

##### **5.1.2.1.1. Ejecución y visualización en Rviz.**

Para la ejecución de octomap\_server bajo ese fichero de extensión .launch se escribe la sentencia “*roslaunch octomap\_server octomap\_tracking\_server1.launch*” en la terminal.

Tras lanzar octomap\_server , aparece el siguiente aviso:

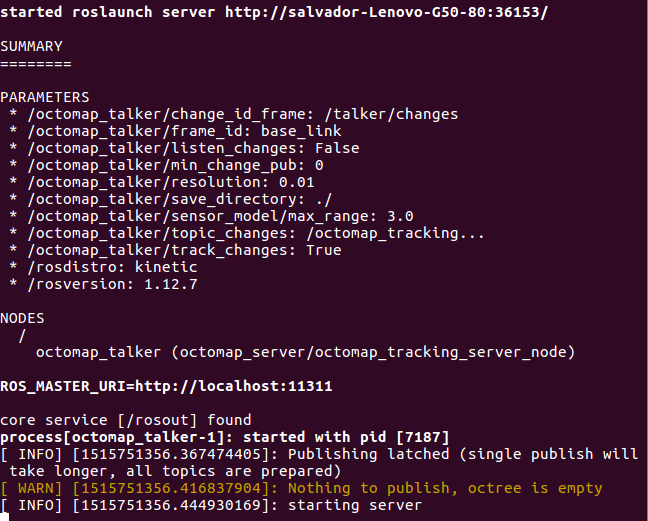


Figura 28: Ejecución de octomap\_server. Primer intento.

Tras ello se ejecuta Rviz y se actualizan los siguientes parámetros:

* Se añade un “*MarkerArray*”, para ello se siguen los siguientes pasos: Add → By display type → rviz → MarkerArray.
* Se cambia el valor de Fixed Frame a *“base\_link”*.
* En MarkerArray se actualiza Marker Topic a “*/occupied\_cells\_vis\_array*”.

Se visualiza lo siguiente:

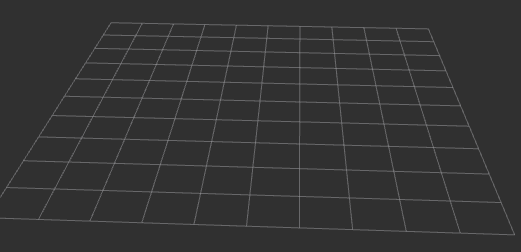


Figura 29: visualización en Rviz de octomap\_server. Primer intento.

El mapa en Rviz se encuentra totalmente vacío, esto, entre otros motivos, se debe a que, como se explica en el apartado de octomap\_server, es necesario actualizar el valor de latch a “False”. Este parámetro no aparece en el archivo predeterminado de octomap.

En esta situación surgen dos caminos posibles; seguir modificando el documento anterior, el cual no se tiene certeza de que acabe funcionando o desarrollar otro archivo de extensión.launch con otro formato distinto. Como en ocasiones anteriores, se opta por la segunda opción puesto que, en caso de que no funcione, tenemos dos opciones viables por las que seguir.

### **5.1.2.2. Segundo intento.**

Para la realización del segundo fichero de formato .launch, partimos de diversos ejemplos que se encuentran en Internet y los adaptamos a las necesidades actuales.

En primer lugar, en el código, se establece el tipo de paquete el que pertenece este documento, estableciéndolo como nodo y dándole un nombre. A continuación, y como en el caso anterior, se establece el frame de referencia.

Como principal cambio, respecto al primer intento se encuentra el parámetro latch. Este parámetro es el que permite mapear, siempre y cuando tenga un valor igual a “False”.

Finalmente se establecen los valores de rango máximo y resolución del láser, además del topic al cuál debe subscribirse. En este caso es /UNEXMIN/laser/scan2 (se explica más adelante).

##### **5.1.2.2.1. Ejecución y visualización en Rviz.**

Se utiliza el mismo comando utilizado en el primer caso, variando, únicamente, el nombre del archivo: “*roslaunch octomap\_server octomap\_tracking\_server2.launch*”. Tras lanzar octomap\_server se observa:

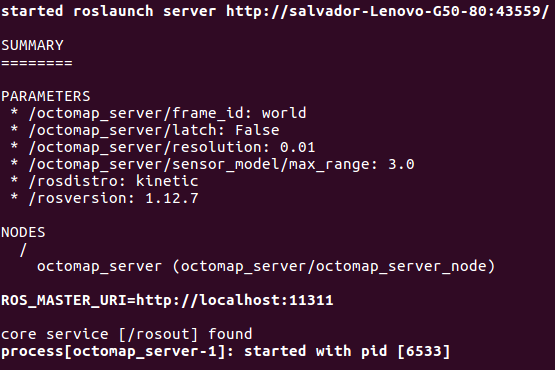


Figura 30: Ejecución de octomap\_server. Segundo intento.

En este caso, no existe ningún warning al que prestar atención, por lo que se procede a la ejecución de Rviz con los mismos parámetros que se utilizaron con el anterior fichero de extensión .launch.

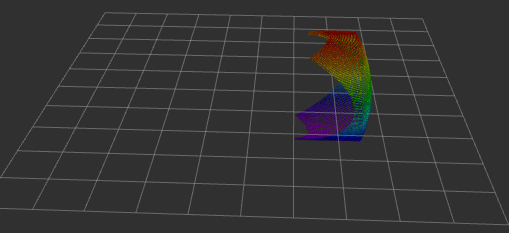


Figura 31: Visualización en Rviz de octomap\_server. Segundo intento.

Se observa cómo se obtiene un mapeado 3D en colores RGB que corresponde con la nube de puntos que proporciona el láser Velodyne. Gracias a esto podemos concluir que octomap\_server está correctamente instalado y listo para utilizarse.

### **5.1.3. Nodo de conversión**

Para que el módulo de octomap\_server funcione, es necesario que el topic al que se encuentra subscrito posea información con el formato PointCloud2. El láser Velodyne configurado publica información de tipo PointCloud en “/*UNEXMIN/laser/scan*”, por lo que sin un cambio de formato sería imposible realizar el mapeado con los paquetes octomap. Por ello se realiza otro nodo, cuya función, en este caso, es la de convertir los datos de formato PointCloud al tipo PontCloud2.

### **5.1.3.1. Desarrollo.**

Como con todos los nodos descritos hasta ahora, se establece una carpeta de trabajo denominada convert\_ws, en la cual se introducen los archivos Cmakelists.txt y Package.xml, así como la carpeta src, que incluye el documento de extensión .cpp que programa el comportamiento del nodo.

El funcionamiento se basa en tres sencillos puntos. El primero de ellos, subscribirse al topic proporcionado por el láser Velodyne y que proporciona información con el formato PointCloud. El segundo paso es transformar dichos datos al tipo PointCloud2 para que finalmente se pueda publicar la información en un nuevo topic al que se llamará, “*/UNEXMIN/laser/scan2*”.

Para poder realizar todos los pasos son necesarias las bibliotecas “*PointCloud.h*”, “*PointCloud2.h*”, que contienen la información necesaria sobre los formatos requeridos, y la librería “*point\_cloud\_conversion.h*”, que proporciona la función necesaria para la transformación; todas estas librerías pertenecen al conjunto de paquetes “*sensor\_msgs*”.

Una vez incluidas dichas bibliotecas se procede a definir una clase, cuyos atributos privados son un objeto de tipo “*publisher*”, que publicará mensajes del tipo PointCloud2, y otro de clase “*subscriber*”, que obtendrá la información en formato PointCloud. En el apartado público se incluyen los tipos de mensajes utilizados, y los topic implicados. También se define una función denominada callback.

Callback es llamada cada vez que se recibe información del topic al que se está subscrito. La función tiene como entrada un puntero que señala a la información en forma de “*sensor\_msgs::PointCloud*” proveniente del topic "UNEXMIN/laser/scan". Los datos del puntero se utilizan como entrada en la función que los convierte al formato “*PointCloud2*”, y que los devuelve dentro de una variable que también se introduce como entrada. Finalmente se publica dicha variable en el topic deseado.

En la función main, principalmente se declara un objeto de la clase definida.

**5.1.3.1.1. Ejecución y visualización en Rviz**.

Para la compilación del nodo no es necesario modificar en exceso los archivos CmakeLists.txt y package.xml, con introducir en el primero los paquetes referidos a los “*sensor\_msgs*” utilizados es suficiente. Una vez, que se ha creado un ejecutable, se utiliza el comando rosrun para lanzarlo.

Con el objetivo de comprobar el correcto funcionamiento del nodo diseñado se procede a usar, entre otras, la función info de rostopic.

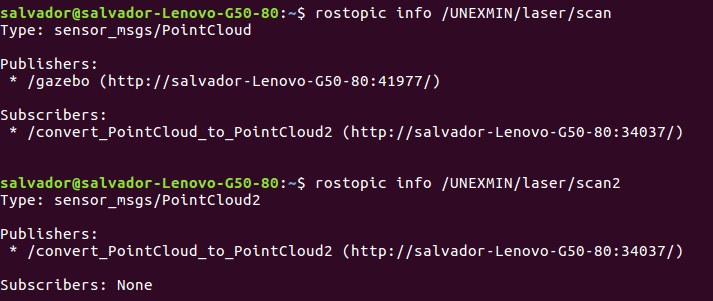


Figura 32: Comparación de los topics "/UNEXMIN/laser/scan" y "/UNEXMIN/laser/scan2"

En la imagen anterior se obtiene información de los topics “*Unexmin/laser/scan*”, tanto del proporcionado por el Velodyne como del creado por el nodo de conversión. En el primero se observa un nuevo subscriptor, cuyo nombre se corresponde con el del código desarrollado. En cuanto al segundo, muestra el tipo de mensajes utilizados, que coincide con el formato pretendido, y además, el nodo de conversión, aparece como único publicador.

Tras comprobar que el nodo realiza las funciones de subscripción en topic y creador y publicador de otro, se procede a revisar que los datos se han convertido correctamente y para ello utilizamos el plugin Rviz.

Por ello, se ejecuta la aplicación y se utilizan los siguientes parámetros:

* Se añade un marcador del tipo “*PointCloud2*”, para ello se siguen los siguientes pasos: Add → By display type → rviz → PointCloud2.
* El valor de Fixed Frame se mantiene en *“base\_link”*.
* En el apartado PointCloud2 se establece el topic “*/UNEXMIN/laser/scan2*”.

Finalmente, en el mapa de Rviz se observa lo siguiente:

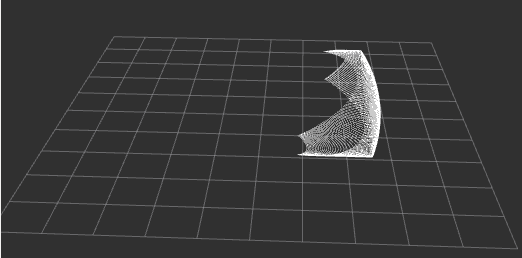


Figura 33: Visualización en Rviz de la nube de puntos correspondiente a "/UNEXMIN/laser/scan2"

La figura muestra una nube de puntos idéntica a la del apartado 3.1, por lo que se concluye que el nodo de conversión está correctamente desarrollado.

### **5.1.4. Tf Broadcaster y Listener.**

Con la intención de realizar un mapeado 3D de las minas, es necesario crear un frame de referencia, denominado “*map*”, en el cual se realizarán ciertas transformaciones respecto el frame ya existente “*base\_link*”.

**5.1.4.1. Broadcaster.**

Este código se encarga de crear/publicar el frame y realizar las transformaciones necesarias, basadas en la librería tf de ROS, que proporciona herramientas muy útiles para conseguir este fin. Estas transformaciones son las que en muchos sitios consideran como localización y que son necesarias en la mayoría de los algoritmos destinados a la realización de un mapa en 3D.

Su funcionamiento depende del topic “*/gazebo/model\_states/*”, donde se publican la posición y orientación del robot Unexmin. Por ello, en una parte del código, subscribe el nodo a dicho tema y utiliza los datos obtenidos para realizar las transformaciones del frame “base\_link” a “map” para posteriormente publicarlas para que un posible listener las pueda utilizar. Es muy importante asegurarse de que el modelo del cual se está obteniendo información es “*my\_Unexmin*” ya que, de no ser así, las transformaciones realizadas no producirían el efecto esperado.

##### **5.1.4.1.1. Ejecución**

Tras revisar que el código desarrollado compila sin errores, es necesario ejecutar y comprobar que los frames creados existen y se comunican, pues es muy común en este tipo de códigos que compilen correctamente pero no realicen su función debido, por ejemplo, a una errónea suscripción al topic deseado, o a una incorrecta relación entre frames.

El comando ”*rosrun tf view\_frames” e*scucha a todos los broadcasters disponibles y desarrolla, a partir de ellos, un PDF que contiene todos los frames disponibles y las relaciones establecidas entre ellos, dando como resultado la siguiente imagen:

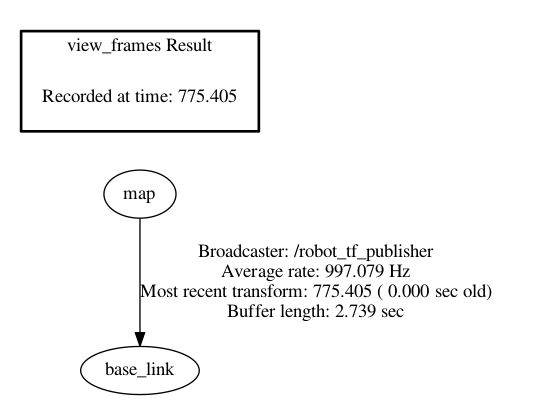


Figura 34: Frame tree tras la ejecución del broadcaster y el listener.

En ella se observan los frames mencionados, con la relación establecida entre ellos; “*base\_link*” se trata del frame child mientras “map” es el parent.

**5.1.4.2. Listener**.

Todo broadcaster necesita de un listener que simplemente escucha y aplica las transformaciones realizadas. Por tanto, se tratan de códigos muy simples en los que basta con utilizar una de las funciones de la librería tf.

La única forma de comprobar su funcionamiento es realizando el mapeado 3D de las minas.

# 6. Líneas futuras

# 7. BIBLIOGRAFÍA

1. *Academia.edu*. (s.f.). Obtenido de http://www.academia.edu/913608/Rob%C3%B3tica\_Estado\_del\_arte
2. *Ardupilot.* (s.f.). Obtenido de http://ardupilot.org/dev/docs/using-gazebo-simulator-with-sitl.html
3. *Ardupilot.* (s.f.). Obtenido de http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
4. ASESMAR, Fundación de estudios del mar. (2013). Propulsion naval. *XXXI SEMANA DE ESTUDIOS DEL MAR.* Vilagarcía de Arousa. Obtenido de www.asesmar.org/conferencias/documentos/
5. *Avance de la robótica*. (s.f.). Obtenido de https://sites.google.com/site/elavancedelarobotica/clasificacion-de-los-robots/clasificacion-de-los-robots-su
6. Bassham, B. A. (2003). *An evaluation of electric motors for ship propulsion.* Calhoun: The NPS Institutional Archive.
7. *bibing.* (s.f.). Obtenido de http://bibing.us.es/proyectos/abreproy/5139/fichero/PFC-+Por+cap%C3%ADtulos%252F3-Entorno+de+simulaci%C3%B3n.pdf
8. Blázquez, F. (2016). *Apuntes de la asignatura de Máquinas Eléctricas II.* ETSII - UPM.
9. *bowlesphysics.* (s.f.). Obtenido de http://www.bowlesphysics.com/images/Robotics\_-\_A\_historical\_perspective.pdf
10. Carol, F., & Dr. Thomas, L. H. (s.f.). *ROS Robotics By Example.*
11. CORDIS, Servicio de Información Comunitario sobre Inveerstigación y Desarrollo. (Junio de 2014). *CORDIS Europa.* Recuperado el 2017, de http://cordis.europa.eu/
12. Daniel Rojas Rodríguez, A. S. (2009). Consideraciones sobre la aplicación de motores eléctricos con tecnología de imanes permanentes para el accionamiento directo de maquinaria rotativa en buques. *Congreso de Ingeniería Naval.* Vigo : Vicus Desarrollos Tecnológicos S.L.
13. Dialnet. (2006). Primer remolcador azimutal tractor de UNV. *Dialnet*.
14. diariodenautica. (2014). *diariodenautica, El digital marítimo de referencia*. Recuperado el 2017, de http://www.diariodenautica.com/los-principales-sistemas-de-propulsion-utilizados-en-los-buques-actuales
15. Díaz, J. G. (2012). *Estudio de motores eléctricos para su uso en motocicletas.* Madrid: Proyecto de Fin de Carrera, Universidad Carlos III Madrid.
16. Diego Valdés, R. (2009). *Estado del arte en robótica submarina.* Valencia.
17. Echave, P. D. (2014). *Estudio Técnico de un sistema de Propulsión Azimutal.* Trabajo de Fin de Grado, Universidad de Cantabria.
18. Edison Felipe, C. C. (s.f.). *Desarrollo de móvil teledirigido basado en ROS.* Granada.
19. García, G. M. (s.f.). *Motores con imanes permanentes.* Recuperado el 2017, de http://www.monografias.com/
20. García, R. (s.f.). *Sistema de Propulsión Schottel (SRP)*. Recuperado el 2017, de Ingeniero Marino: http://ingenieromarino.com/sistema-schottel-srp/
21. Héctor A., M., Roque, S., Lisandro, P., Isela, C., Pedro, C., & César, Á. (2013). Robótica Submarina: Conceptos, Elementos, Modelado y Control. *ScienceDirect*, 17.
22. *IFR*. (s.f.). Obtenido de https://ifr.org/standardisation
23. Joaquín A. Batlle, J. M. (2016). *Guiado de un robot móvil con cinemática de triciclo.*
24. José Andrés, R. M. (2015). *Reconocimiento gestual para interacción humano-robot basado en ROS.* Sevila.
25. Manzaneque Amo, D. (2016). *Guiado de un Robot Móvil basado en ROS y Kinect.* Madrid.
26. MGerNet (MG). (2016). *www.mgar.net*. Recuperado el 10 de 2017, de http://www.mgar.net/
27. Miguel Ángel, L. F., David, S. F., & Carlos, P. V. (2009). *Planificación de trayectorias para un robot móvil.*
28. *Moodle, Universidad de Alicante*. (2015). Obtenido de https://moodle2015-16.ua.es/moodle/mod/book/view.php?id=82546&chapterid=2401
29. Mora, A. J. (2015). *Hélices y timones de maniobra.* La Laguna: Trabajo de Fin de Grado. Universidad de la Laguna, ULL.
30. Moral, G. V. (2016). *Programación de comportamientos de un robot autónomo.*
31. *Open Dynamics Engine*. (s.f.). Obtenido de http://www.ode.org/
32. Padillo, D. R. (2014). *DISEÑO DE UN GENERADOR SÍNCRONO CON IMANES.* Cataluña: Trabajo de Fin de Grado, Universitat Potècnica de Catalunya.
33. (s.f.). *Planificación de trayectorias para robots móviles.*
34. Ponce, J. A. (2015). *SISTEMAS DE PROPULSIÓN Y CLASIFICACIÓN DE BUQUES.* La Laguna: Universidad de la Laguna, ULL.
35. Proyectos Navales. (2015). Azipod, Mermaidpod, Esipod y otros pods. *Proyectos Navales, Revista digital de ingeniería naval*.
36. REVISTA DYNA. (2014). La tecnología de los motores de imanes permanentes . *REVISTA DYNA*.
37. *Robot Submarinos*. (s.f.). Obtenido de http://www-assig.fib.upc.es/~rob/protegit/treballs/Q2\_03-04/submarinos/rov\_his.htm
38. *Robóticapuno*. (s.f.). Obtenido de http://roboticapuno.blogspot.com.es/2013/01/clasificacion-de-los-robots.html
39. Rolls-Royce. (s.f.). *Mermaid podded propulsors* . Recuperado el 2016, de Rolls-Royce: https://www.rolls-royce.com/
40. *ROS Tutorials*. (s.f.). Obtenido de http://www.clearpathrobotics.com/assets/guides/ros/Launch%20Files.html
41. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/Topics
42. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/Master
43. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/Services
44. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/Bags
45. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/xacro
46. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/urdf
47. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/urdf/XML/Gazebo
48. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/gazebo\_ros?distro=kinetic
49. *ROS.org*. (s.f.). Obtenido de http://wiki.ros.org/gazebo\_ros\_pkgs
50. *ROVs*. (s.f.). Obtenido de http://www.rovs.es/que-es-un-rov
51. Sánchez, N. (3 de Noviembre de 2008). *Maquinas de barcos*. Recuperado el 2017, de http://maquinasdebarcos.blogspot.com.es/2008/11/propulsin-elctrica-en-los-buques-i.html
52. SCHOTTEL. (s.f.). *SRP Rudderpropeller*. Recuperado el 2017, de Schottel : https://www.schottel.de/
53. Siemmens. (s.f.). *Siemmens Global Website*. Recuperado el 2017, de https://www.siemens.com/
54. Tomi Veikonheimo, M. T. (2003). CRP Azipod, más eficiencia para la propulsión naval. *Revista ABB*.
55. Turtiainen, M. (2005). Barcos ecológicos. *Revista ABB*.
56. Veslasco-Hernández, G. (2014). *Guía de ROS.*
57. Víctor Hugo, T. G. (2015). *Operación de remachado mediante robot manipulador industrial basado en ROS.* Sevilla.
58. Víctor, M. E. (2009). *Modelado y simulación del robot PASIBOT: estudio de la rigidez y mejora en la prevención del vuelco lateral.* Madrid.
59. Víctor, S., Ricardo, B., José Rafael, S. G., & Dr. Ramón, O. S. (2007). *Robots Móviles: Evolución y Estado del Arte.*
60. Vicusdt, Desarrollos Tecnológicos. (s.f.). *VICUSDT, Desarrollos Tecnológicos.* Recuperado el 2017, de VICUSDT: http://vicusdt.com/
61. *Vinssa*. (s.f.). Obtenido de http://vinssa.com/news/la-historia-y-evolucion-de-los-robots-moviles.html
62. Voith Inline Thruster. (2016). *VOITH RIM DRIVE. TECHNOLOGY IN YACHTS.* VOITH .
63. *Wiki de Robótica*. (s.f.). Obtenido de http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/historia/
64. *Wiki de Robótica*. (s.f.). Obtenido de http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/clasificacion-de-robots/
65. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/Robot\_m%C3%B3vil
66. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/Sistema\_Operativo\_Rob%C3%B3tico
67. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/Historia\_de\_Linux
68. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/ROV\_Sumergible
69. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/Robot\_m%C3%B3vil
70. *Wikipedia*. (s.f.). Obtenido de https://es.wikipedia.org/wiki/OGRE\_3D
71. *Yaml*. (s.f.). Obtenido de http://yaml.org/spec/1.2/spec.html

# 8. Planificación temporal

El presente proyecto se inició a principios de noviembre de 2017. Como primera toma de contacto se instalaron todas las herramientas, sistemas operativos y aplicaciones que se postulaban como indispensables para la consecución de los objetivos propuestos.

A continuación, y de forma paralela, se realizan tanto los tutoriales propuestos por la comunidad de ROS, la familiarización con todos los elementos utilizados y la integración del láser Velodyne en el robot UNEXMIN.

Posteriormente y de forma simultánea se procede a la realización de los nodos de desplazamiento y de los tutoriales que facilitaban dicha labor. Se trata de una de las etapas más importantes y costosas del proyecto.

Tras ello, se comienza la etapa de inclusión de los elementos de mapeado 3D, en concreto de todos aquellos relacionados con Octomap, como son el archivo launch y el nodo de conversión de datos.

Como etapas finales se encuentran el modelado 3D de las minas de Kaatiala y la realización de los nodos Bradcaster y Listener que permiten la adición y transformación de frames.

La redacción del documento se inició tras la finalización del último nodo de desplazamiento telemático y finalizó el 5 de febrero.

Las etapas y sus duraciones quedan explicadas con mayor profundidad en el diagrama de Gantt que se encuentra a continuación.

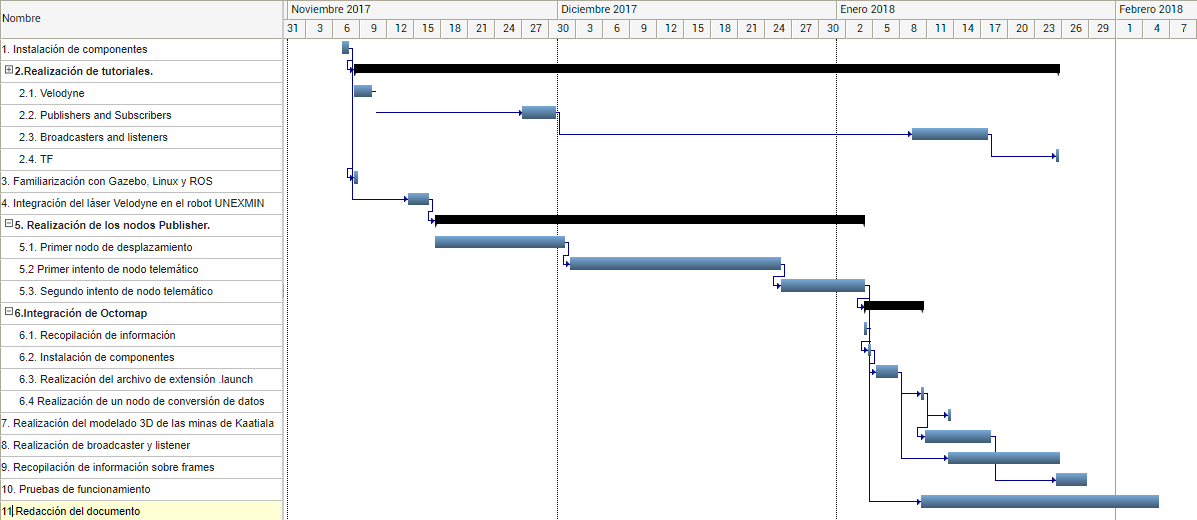


Figura 35: Diagrama de Gantt.

# 9. Presupuesto.

Debido a que los recursos utilizados no han supuesto coste alguno, bien porque ya se disponía de ellos o porque son gratuitos, simplemente se contabilizan las unidades de trabajo.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CANTIDAD** | **DESCRIPCIÓN** | **SALARIO POR UNIDAD** | | | | **TOTAL** |
| Unidades | | Precio unitario | | Importe (€) |
| Cantidad | Unidad | Importe | Unidad |
| 1 | Ingeniero Junior | 360 | h | 15 | €/h | 5.400 |
| 1 | Ingeniero Técnico Superior Senior | 20 | h | 43 | €/h | 860 |
| SUBTOTAL | | | | | | **6.460 €** |
| SUBTOTAL CON IVA | | | | | | **7.816,6 €** |

Tabla 1: Presupuesto de horas empleadas en el trabajo

# 10. Índice de figuras

[Figura 1: Integración del láser 3D Velodyne en el robot submarino Unexmin. 5](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249172)

[Figura 2: Frames tree creado mediante los nodos broadcaster y listener. 6](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249173)

[Figura 3: Minas de Kaatiala desarrolladas mediante el software Blender. 7](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249174)

[Figura 4: Tortugas de Bristol. 8](#_Toc505249175)

[Figura 5: Robot doméstico Roomba 10](#_Toc505249176)

[Figura 6: Clasificación de los robots submarinos. 13](#_Toc505249177)

[Figura 7: Esquema de los elementos propios de un sistema ROV 14](#_Toc505249178)

[Figura 8: Robot Girona 500 16](#_Toc505249179)

[Figura 9: Logotipo de ROS 20](#_Toc505249180)

[Figura 10:Ejemplo de comunicación entre nodos y topics. 22](#_Toc505249181)

[Figura 11: Esquema general de comunicación en ROS. 23](#_Toc505249182)

[Figura 12: Logotipo de Gazebo. 25](#_Toc505249183)

[Figura 13: Logotipo de Rviz 26](#_Toc505249184)

[Figura 14: Interfaz y logotipo Suzanne de Blender 27](#_Toc505249185)

[Figura 15: Visualización del láser Velodyne 3D en Gazebo 30](#_Toc505249186)

[Figura 16: Captura de la terminal de linux ante el comando "rostopic info" 30](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249187)

[Figura 17: Visualización en Rviz de la nube de puntos correspondiente al topic "/UNEXMIN/laser/scan" 31](#_Toc505249188)

[Figura 18: Captura del terminal de linux ante el comando "rostopic list". 33](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249189)

[Figura 19: Reapuesta de la terminal de Linux ante el comando "rostopic info /gazebo/set\_model\_state". 34](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249190)

[Figura 20: Visualización en Gazebo del desplazamiento de 5 unidades en el eje x 34](#_Toc505249191)

[Figura 21: Respuesta de la terminal de linux ante el comando "rostopic info /gazebo/set\_model\_state" 36](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249192)

[Figura 22: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del nodo de desplazamiento. 36](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249193)

[Figura 23:Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del primer nodo de operación telemática (posición final). 37](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249194)

[Figura 24:Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del primer nodo de operación telemática. 38](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249195)

[Figura 25: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del segundo nodo de operación telemática. Movimiento lineal. 40](#_Toc505249196)

[Figura 26: Respuesta de la terminal ante el comando "rostopic echo /gazebo/set\_model\_state" durante la ejecución del segundo nodo de operación telemática. Movimiento de rotación. 41](#_Toc505249197)

[Figura 27: Instrucciones para desplazar telamáticamente el robot Unexmin mostradas en la terminal tras la ejecución del segundo nodo de operación telemática. 41](#_Toc505249198)

[Figura 28: Ejecución de octomap\_server. Primer intento. 44](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249199)

[Figura 29: visualización en Rviz de octomap\_server. Primer intento. 45](#_Toc505249200)

[Figura 30: Ejecución de octomap\_server. Segundo intento. 46](#_Toc505249201)

[Figura 31: Visualización en Rviz de octomap\_server. Segundo intento. 46](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249202)

[Figura 32: Comparación de los topics "/UNEXMIN/laser/scan" y "/UNEXMIN/laser/scan2" 48](file:///C:\Users\devil_000\Downloads\tfg.docx#_Toc505249203)

[Figura 33: Visualización en Rviz de la nube de puntos correspondiente a "/UNEXMIN/laser/scan2" 49](#_Toc505249204)

[Figura 34: Frame tree tras la ejecución del broadcaster y el listener. 50](#_Toc505249205)

[Figura 35: Diagrama de Gantt. 57](#_Toc505249206)

# 11.ANEXOS

## **11.1 Código del Velodyne**

<?xml version=**"1.0"** ?>

<sdf version=**"1.5"**>

<model name=**"velodyne\_UNEXMIN"**>

<pose>**0 0 0.35 0 0 0**</pose>

<link name=**"link"**>

<inertial>

<mass>**1.0**</mass>

<inertia><!-- interias are tricky to compute -->

<!-- http://answers.gazebosim.org/question/4372/the-inertia-matrix-explained/ -->

<ixx>**0.000415**</ixx><!-- for a box: ixx = 0.083 \* mass \* (y\*y + z\*z) -->

<ixy>**0.0**</ixy><!-- for a box: ixy = 0 -->

<ixz>**0.0**</ixz><!-- for a box: ixz = 0 -->

<iyy>**0.000415**</iyy><!-- for a box: iyy = 0.083 \* mass \* (x\*x + z\*z) -->

<iyz>**0.0**</iyz><!-- for a box: iyz = 0 -->

<izz>**0.000415**</izz><!-- for a box: izz = 0.083 \* mass \* (x\*x + y\*y) -->

</inertia>

</inertial>

<collision name=**"collision-base"**>

<pose>**0 0 -0.0145 0 0 0**</pose>

<geometry>

<box>

<size>**0.05 0.05 0.041**</size>

</box>

</geometry>

</collision>

<collision name=**"collision-top"**>

<pose>**0 0 0.0205 0 0 0**</pose>

<geometry>

<cylinder>

<radius>**0.021**</radius>

<length>**0.029**</length>

</cylinder>

</geometry>

</collision>

<sensor name=**"laser"** type=**"ray"**>

<pose>**0 0 0 0 0 0** </pose>

<ray>

<scan>

<horizontal>

<samples>**640**</samples>

<resolution>**1**</resolution>

<min\_angle>**-0.5**</min\_angle>

<max\_angle>**0.5**</max\_angle>

</horizontal>

<vertical>

<samples>**100**</samples>

<resolution>**1.0**</resolution>

<min\_angle>**-0.7**</min\_angle>

<max\_angle>**0.7**</max\_angle>

</vertical>

</scan>

<range>

<min>**0.08**</min>

<max>**3.0**</max>

<resolution>**0.01**</resolution>

</range>

</ray>

<plugin name=**"gazebo\_ros\_head\_hokuyo\_controller"** filename=**"libgazebo\_ros\_block\_laser.so"**>

<topicName>**/UNEXMIN/laser/scan**</topicName>

<frameName>**base\_link**</frameName>

<gaussianNoise>**0.00**</gaussianNoise>

<updateRate>**40**</updateRate>

</plugin>

<always\_on>**1**</always\_on>

<update\_rate>**40**</update\_rate>

<visualize>**true**</visualize>

</sensor>

</link>

</model>

</sdf>

## **11.2. Código del primer publisher**

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include "std\_msgs/Float64.h"

#include "std\_msgs/Float64MultiArray.h"

#include <gazebo\_msgs/SetModelState.h>

#include <stdlib.h>

#include <sstream>

#include <ros/ros.h>

int main**(**int argc**,** char **\*\***argv**)**

**{**

// Se inicia ROS

ros**::**init**(**argc**,** argv**,** "movement"**);**

ros**::**NodeHandle nh**;**

//Se crea el publisher del topic que se necesita

ros**::**Publisher pub **=** nh**.**advertise**<**gazebo\_msgs**::**ModelState**>(**"/gazebo/set\_model\_state"**,** 1000**);**

// Se establece la frecuencia del bucle

ros**::**Rate loop\_rate**(**2**);**

// Se establece un contador que permite modificar la posición

int count**=**0**;**

**while** **(**ros**::**ok**())**

**{**

//Se intoducen los valores de posición y rotación necesarios

gazebo\_msgs**::**ModelState msg**;**

msg**.**pose**.**position**.**x**=**count**\***0.3**;**

msg**.**pose**.**position**.**y**=**0.0**;**

msg**.**pose**.**position**.**z**=**0.0**;**

msg**.**twist**.**linear**.**x**=**1.0**;**

msg**.**twist**.**linear**.**y**=**0.0**;**

msg**.**twist**.**linear**.**z**=**0.0**;**

msg**.**twist**.**angular**.**x**=**0.0**;**

msg**.**twist**.**angular**.**y**=**0.0**;**

msg**.**twist**.**angular**.**z**=**0.0**;**

msg**.**model\_name**=**"my\_UNEXMIN" **;**

count**++;**

// Se publican los valores

pub**.**publish**(**msg**);**

ros**::**spinOnce**();**

loop\_rate**.**sleep**();**

**}**

**return** 0**;**

**}**

## **11.3 Código del primer nodo de desplazamiento telemático**

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include "std\_msgs/Float64.h"

#include "std\_msgs/Float64MultiArray.h"

#include <gazebo\_msgs/SetModelState.h>

#include <stdlib.h>

#include <sstream>

#include <ros/ros.h>

#include <iostream>//\*\* para que me coja el teclado\*/

#include <curses.h>//\*\* para no presionar enter cada vez que pulso la tecla\*/

int main**(**int argc**,** char **\*\***argv**)**

**{**

// Se inicia ROS

ros**::**init**(**argc**,** argv**,** "movement\_keyboard"**);**

ros**::**NodeHandle nh**;**

// Se declara el Publisher

ros**::**Publisher pub **=** nh**.**advertise**<**gazebo\_msgs**::**ModelState**>(**"/gazebo/set\_model\_state"**,** 1000**);**

// Se establece un contador para predecir la posición

int count**=**0**;**

float current\_velocity\_x**=**0.0**;**

/\*\* metemos la función del teclado; con las flechas controlamos el robot \*/

char cmd**;**

initscr**();** //Se captan eventos de un terminal externo

cbreak**();**

// Se establece la frecuencia de publicación

ros**::**Rate loop\_rate**(**2**);**

**while** **(**ros**::**ok**())**

**{**

gazebo\_msgs**::**ModelState msg**;**

msg**.**twist**.**linear**.**x**=**current\_velocity\_x**;**

// Se establece el nombre del modelo

msg**.**model\_name**=**"my\_UNEXMIN" **;**

/\*\*actualizamos posición en x\*/

**if(**current\_velocity\_x**>=**0**){**

msg**.**pose**.**position**.**x**=**count**\***0.2**\***current\_velocity\_x**;**

count**++;**

**}**

**else** **{**

msg**.**pose**.**position**.**x**=**count**\***0.2**\***current\_velocity\_x**;**

count**--;**

**}**

// Se guarda el evento del teclado

cmd**=**getch**();**

//Se modifica la velocidad

**if(**cmd**==**'w'**&&** current\_velocity\_x**<**3 **)** **{**

msg**.**twist**.**linear**.**x**=**current\_velocity\_x**+**0.25**;**

current\_velocity\_x**=**msg**.**twist**.**linear**.**x**;**

**}**

**if(**cmd**==**'s' **&&** current\_velocity\_x**>-**3**)** **{**

msg**.**twist**.**linear**.**x**=**current\_velocity\_x**-**0.25**;**

current\_velocity\_x**=**msg**.**twist**.**linear**.**x**;**

**}**

**if(**cmd **==**'c'**){**

msg**.**twist**.**linear**.**x**=** 0.0**;**

msg**.**pose**.**position**.**x **=** 0.0**;**

count**=**0**;**

**break;**

**}**

pub**.**publish**(**msg**);**

/\*\*

Publicamos las nuevas modificaciones

\*/

ros**::**spinOnce**();**

loop\_rate**.**sleep**();**

**}**

nocbreak**();**

**return** 0**;**

**}**

## **11.4. Código del segundo nodo de desplazamiento telemático.**

#include "ros/ros.h"

#include <gazebo\_msgs/SetModelState.h>

#include <signal.h>

#include <termios.h>

#include <stdio.h>

#define PI 3.14159265

#define L1 104

#define L2 104

#define Lp 60

// Se establecen las teclas que se van a usar

#define KEYCODE\_Xmas 0x43

#define KEYCODE\_Xmenos 0x44

#define KEYCODE\_Ymas 0x41

#define KEYCODE\_Ymenos 0x42

#define KEYCODE\_Zmas 0x77

#define KEYCODE\_Zmenos 0x73

#define KEYCODE\_ZgiroMas 0x61

#define KEYCODE\_ZgiroMenos 0x64

#define KEYCODE\_YgiroMas 0x65

#define KEYCODE\_YgiroMenos 0x71

//Se establece las variables que permiten entrar en el raw mode

struct termios cooked**,** raw**;**

int cont **=** 0**;**

// Se establece un retardo que elimina los rebotes

double retardo **=** 0.1**;**

gazebo\_msgs**::**ModelState objeto**;**

// Se establece una forma de salir del raw mode

void quit**(**int sig**)**

**{**

tcsetattr**(**0**,** TCSANOW**,** **&**cooked**);**

ros**::**shutdown**();**

exit**(**0**);**

**}**

void callback**(**const ros**::**TimerEvent**&)**

**{**

ros**::**NodeHandle n**;**

//Se declara un objeto publisher

ros**::**Publisher object\_pub\_**=**n**.**advertise**<**gazebo\_msgs**::**ModelState**>(**"/gazebo/set\_model\_state"**,** 1**);**

//En seta varible se guarda el nombre del modelo que queremos modificar

std**::**string modelo\_objeto**;**

ros**::**NodeHandle nh**(**"~"**);**

// Se establece el nombre del modelo

nh**.**param**(**"modelo\_objeto"**,** modelo\_objeto**,** std**::**string**(**"my\_UNEXMIN"**));**

ros**::**Time anterior**;**

// Se realizan las inicializaciones necesarias

**if** **(::**cont **==** 0**)**

**{**

anterior **=** ros**::**Time**::**now**();**

**::**objeto**.**model\_name **=** modelo\_objeto**;**

**::**objeto**.**reference\_frame **=** "world"**;**

**::**objeto**.**pose**.**orientation**.**w **=** 1**;**

**::**objeto**.**twist**.**angular**.**x **=** 0**;**

**::**objeto**.**twist**.**angular**.**y **=** 0**;**

**::**objeto**.**twist**.**angular**.**z **=** 0**;**

**::**cont **=** 1**;**

object\_pub\_**.**publish**(::**objeto**);**

**}**

char c**;**

// Comienza el raw mode

tcgetattr**(**0**,** **&**cooked**);**

memcpy**(&**raw**,** **&**cooked**,** **sizeof(**struct termios**));**

raw**.**c\_lflag **&=~** **(**ICANON **|** ECHO**);**

// Se introduce una linea más en el terminal

raw**.**c\_cc**[**VEOL**]** **=** 1**;**

raw**.**c\_cc**[**VEOF**]** **=** 2**;**

tcsetattr**(**0**,** TCSANOW**,** **&**raw**);**

// Permitirá vsualizar los controles por pantalla

puts**(**""**);**

puts**(**"#####################################################"**);**

puts**(**" CONTROLES OBJETO"**);**

puts**(**"#####################################################"**);**

puts**(**""**);**

puts**(**"Up arrow:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object +z axes"**);**

puts**(**"Down arrow:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object -z axes"**);**

puts**(**"Left arrow:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object -x axes"**);**

puts**(**"Right arrow:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object +x axes"**);**

puts**(**"W key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object +y axes"**);**

puts**(**"S key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Move object -y axes"**);**

puts**(**"A key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Tilt + x axis object"**);**

puts**(**"D key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Tilt - x axis object"**);**

puts**(**"Q key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Tilt + y axis object"**);**

puts**(**"E key:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Tilt - y axis object"**);**

**while** **(**ros**::**ok**())**

**{**

// Se capta el siguiente evento por teclado

**if(**read**(**0**,** **&**c**,** 1**)** **<** 0**)**

**{**

perror**(**"read():"**);**

exit**(-**1**);**

**}**

/\*\* no tiene sentido utilizar velocidades con el teclado ya que debemos esperar al evento de la tecla y no se actualiza la posicion correctamente en función de la velocdad(no son ciclos constntes)\*/

//Se establecen los desplazamientos y giros correspondientes a cada tecla

**if** **(**c **==** KEYCODE\_Xmas**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**linear**.**x **=** 0.1**;**

**::**objeto**.**pose**.**position**.**x **=** **::**objeto**.**pose**.**position**.**x **+** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_Xmenos**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**linear**.**x **=** **-**0.1**;**

**::**objeto**.**pose**.**position**.**x **=** **::**objeto**.**pose**.**position**.**x **-** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_Ymas**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))** **{**

**::**objeto**.**twist**.**linear**.**z **=** 0.1**;**

**::**objeto**.**pose**.**position**.**z **=** **::**objeto**.**pose**.**position**.**z **+** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_Ymenos**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**linear**.**z **=** **-**0.1**;**

**::**objeto**.**pose**.**position**.**z **=** **::**objeto**.**pose**.**position**.**z **-** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_Zmas**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**linear**.**y **=** 0.1**;**

**::**objeto**.**pose**.**position**.**y **=** **::**objeto**.**pose**.**position**.**y **+** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_Zmenos**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**linear**.**y **=** **-**0.1**;**

**::**objeto**.**pose**.**position**.**y **=** **::**objeto**.**pose**.**position**.**y **-** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_ZgiroMas**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**angular**.**z **=**0.1**;**

**::**objeto**.**pose**.**orientation**.**z **=** **::**objeto**.**pose**.**orientation**.**z **+** 0.1**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_ZgiroMenos**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

//::objeto.pose.orientation.w = 1;

**::**objeto**.**twist**.**angular**.**z **=** **-**0.1**;**

**::**objeto**.**pose**.**orientation**.**z **=** **::**objeto**.**pose**.**orientation**.**z **-** 0.1**;**

//::objeto.pose.position.y = ::objeto.pose.position.y - 0.005;

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_YgiroMas**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**angular**.**y **=** 0.1**;**

**::**objeto**.**pose**.**orientation**.**y **=** **::**objeto**.**pose**.**orientation**.**y **+** 0.1**;**

//::objeto.pose.position.x = ::objeto.pose.position.x + 0.005;

//revisar el codigo del robot a ver si publica la orintacion o algo todo el rato; restar posicion para que se quede en el sitio.

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

**if** **(**c **==** KEYCODE\_YgiroMenos**)**

**{**

**if** **(**ros**::**Time**::**now**()** **>** anterior **+** ros**::**Duration**(::**retardo**))**

**{**

**::**objeto**.**twist**.**angular**.**y **=** **-**0.1**;**

**::**objeto**.**pose**.**orientation**.**y **=** **::**objeto**.**pose**.**orientation**.**y **-** 0.1**;**

**;**

anterior **=** ros**::**Time**::**now**();**

**}**

**}**

//Publicamos los cambios realizados

object\_pub\_**.**publish**(::**objeto**);**

**}**

**}**

int main**(**int argc**,** char **\*\***argv**)**

**{**

// Se inicia ROS

ros**::**init**(**argc**,** argv**,** "movement\_keyboard\_model2"**);**

ros**::**NodeHandle n**;**

// Se establece el publisher al topic deseado

ros**::**Publisher object\_pub\_**=**n**.**advertise**<**gazebo\_msgs**::**ModelState**>(**"/gazebo/set\_model\_state"**,** 1**);**

signal**(**SIGINT**,**quit**);**

//se crea un timer

ros**::**Timer timer **=** n**.**createTimer**(**ros**::**Duration**(**1**),** callback**);**

ros**::**spin**();**

**return** 0**;**

**}**

## **11.5. Ejemplo de CmakeList.txt**

**cmake\_minimum\_required**(**VERSION** 2.8.3)

**project**(movement\_keyboard\_model2)

## Compile as C++11, supported in ROS Kinetic and newer

# add\_compile\_options(-std=c++11)

## Find catkin macros and libraries

## if COMPONENTS list like find\_package(catkin REQUIRED COMPONENTS xyz)

## is used, also find other catkin packages

**find\_package**(catkin **REQUIRED** COMPONENTS

roscpp

rospy

std\_msgs

genmsg

geometry\_msgs

gazebo\_msgs

)

**find\_package**( PkgConfig **REQUIRED** )

**find\_package**(Curses **REQUIRED**)

## System dependencies are found with CMake's conventions

# find\_package(Boost REQUIRED COMPONENTS system)

## Uncomment this if the package has a setup.py. This macro ensures

## modules and global scripts declared therein get installed

## See http://ros.org/doc/api/catkin/html/user\_guide/setup\_dot\_py.html

# catkin\_python\_setup()

################################################

## Declare ROS messages, services and actions ##

################################################

## To declare and build messages, services or actions from within this

## package, follow these steps:

## \* Let MSG\_DEP\_SET be the set of packages whose message types you use in

## your messages/services/actions (e.g. std\_msgs, actionlib\_msgs, ...).

## \* In the file package.xml:

## \* add a build\_depend tag for "message\_generation"

## \* add a build\_depend and a run\_depend tag for each package in MSG\_DEP\_SET

## \* If MSG\_DEP\_SET isn't empty the following dependency has been pulled in

## but can be declared for certainty nonetheless:

## \* add a run\_depend tag for "message\_runtime"

## \* In this file (CMakeLists.txt):

## \* add "message\_generation" and every package in MSG\_DEP\_SET to

## find\_package(catkin REQUIRED COMPONENTS ...)

## \* add "message\_runtime" and every package in MSG\_DEP\_SET to

## catkin\_package(CATKIN\_DEPENDS ...)

## \* uncomment the add\_\*\_files sections below as needed

## and list every .msg/.srv/.action file to be processed

## \* uncomment the generate\_messages entry below

## \* add every package in MSG\_DEP\_SET to generate\_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder

# add\_message\_files(

# FILES

# Message1.msg

# Message2.msg

# )

## Generate services in the 'srv' folder

# add\_service\_files(

# FILES

# Service1.srv

# Service2.srv

# )

## Generate actions in the 'action' folder

# add\_action\_files(

# FILES

# Action1.action

# Action2.action

# )

## Generate added messages and services with any dependencies listed here

# generate\_messages(

# DEPENDENCIES

# std\_msgs

# )

################################################

## Declare ROS dynamic reconfigure parameters ##

################################################

## To declare and build dynamic reconfigure parameters within this

## package, follow these steps:

## \* In the file package.xml:

## \* add a build\_depend and a run\_depend tag for "dynamic\_reconfigure"

## \* In this file (CMakeLists.txt):

## \* add "dynamic\_reconfigure" to

## find\_package(catkin REQUIRED COMPONENTS ...)

## \* uncomment the "generate\_dynamic\_reconfigure\_options" section below

## and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder

# generate\_dynamic\_reconfigure\_options(

# cfg/DynReconf1.cfg

# cfg/DynReconf2.cfg

# )

###################################

## catkin specific configuration ##

###################################

## The catkin\_package macro generates cmake config files for your package

## Declare things to be passed to dependent projects

## INCLUDE\_DIRS: uncomment this if your package contains header files

## LIBRARIES: libraries you create in this project that dependent projects also need

## CATKIN\_DEPENDS: catkin\_packages dependent projects also need

## DEPENDS: system dependencies of this project that dependent projects also need

catkin\_package(

# INCLUDE\_DIRS include

# LIBRARIES beginner\_tutorials

# CATKIN\_DEPENDS roscpp rospy std\_msgs

# DEPENDS system\_lib

)

###########

## Build ##

###########

## Specify additional locations of header files

## Your package locations should be listed before other locations

**include\_directories**(

# include

${catkin\_INCLUDE\_DIRS}

)

## Declare a C++ library

# add\_library(${PROJECT\_NAME}

# src/${PROJECT\_NAME}/beginner\_tutorials.cpp

# )

## Add cmake target dependencies of the library

## as an example, code may need to be generated before libraries

## either from message generation or dynamic reconfigure

# add\_dependencies(${PROJECT\_NAME} ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

## Declare a C++ executable

## With catkin\_make all packages are built within a single CMake context

## The recommended prefix ensures that target names across packages don't collide

# add\_executable(${PROJECT\_NAME}\_node src/beginner\_tutorials\_node.cpp)

**add\_executable**(movement\_keyboard\_model2 src/movement\_keyboard\_model2.cpp)

**target\_link\_libraries**(movement\_keyboard\_model2

${catkin\_LIBRARIES}

${Boost\_LIBRARIES}

${Curses\_LIBRARIES})

**target\_link\_libraries**(movement\_keyboard\_model2 ${catkin\_LIBRARIES} ncurses)

**target\_link\_libraries**(movement\_keyboard\_model2 ${ncurses++\_LIBRARIES})

**add\_dependencies**(movement\_keyboard\_model2 ${PROJECT\_NAME}\_gencfg ${PROJECT\_NAME}\_generate\_messages\_cpp)

## Rename C++ executable without prefix

## The above recommended prefix causes long target names, the following renames the

## target back to the shorter version for ease of user use

## e.g. "rosrun someones\_pkg node" instead of "rosrun someones\_pkg someones\_pkg\_node"

# set\_target\_properties(${PROJECT\_NAME}\_node PROPERTIES OUTPUT\_NAME node PREFIX "")

## Add cmake target dependencies of the executable

## same as for the library above

# add\_dependencies(${PROJECT\_NAME}\_node ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

## Specify libraries to link a library or executable target against

# target\_link\_libraries(${PROJECT\_NAME}\_node

# ${catkin\_LIBRARIES}

# )

#############

## Install ##

#############

# all install targets should use catkin DESTINATION variables

# See http://ros.org/doc/api/catkin/html/adv\_user\_guide/variables.html

## Mark executable scripts (Python etc.) for installation

## in contrast to setup.py, you can choose the destination

# install(PROGRAMS

# scripts/my\_python\_script

# DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

# )

## Mark executables and/or libraries for installation

# install(TARGETS ${PROJECT\_NAME} ${PROJECT\_NAME}\_node

# ARCHIVE DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

# LIBRARY DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

# RUNTIME DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

# )

## Mark cpp header files for installation

# install(DIRECTORY include/${PROJECT\_NAME}/

# DESTINATION ${CATKIN\_PACKAGE\_INCLUDE\_DESTINATION}

# FILES\_MATCHING PATTERN "\*.h"

# PATTERN ".svn" EXCLUDE

# )

## Mark other files for installation (e.g. launch and bag files, etc.)

# install(FILES

# # myfile1

# # myfile2

# DESTINATION ${CATKIN\_PACKAGE\_SHARE\_DESTINATION}

# )

#############

## Testing ##

#############

## Add gtest based cpp test target and link libraries

# catkin\_add\_gtest(${PROJECT\_NAME}-test test/test\_beginner\_tutorials.cpp)

# if(TARGET ${PROJECT\_NAME}-test)

# target\_link\_libraries(${PROJECT\_NAME}-test ${PROJECT\_NAME})

# endif()

## Add folders to be run by python nosetests

# catkin\_add\_nosetests(test)

## **11.6. Ejemplo de Package**

<?xml version=**"1.0"**?>

<package format=**"2"**>

<name>**movement\_keyboard\_model2**</name>

<version>**0.0.0**</version>

<description>**The beginner\_tutorials package**</description>

<!-- One maintainer tag required, multiple allowed, one person per tag -->

<!-- Example: -->

<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->

<maintainer email=**"salvador@todo.todo"**>**salvador**</maintainer>

<!-- One license tag required, multiple allowed, one license per tag -->

<!-- Commonly used license strings: -->

<!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->

<license>**TODO**</license>

<!-- Url tags are optional, but multiple are allowed, one per tag -->

<!-- Optional attribute type can be: website, bugtracker, or repository -->

<!-- Example: -->

<!-- <url type="website">http://wiki.ros.org/beginner\_tutorials</url> -->

<!-- Author tags are optional, multiple are allowed, one per tag -->

<!-- Authors do not have to be maintainers, but could be -->

<!-- Example: -->

<!-- <author email="jane.doe@example.com">Jane Doe</author> -->

<!-- The \*depend tags are used to specify dependencies -->

<!-- Dependencies can be catkin packages or system dependencies -->

<!-- Examples: -->

<!-- Use depend as a shortcut for packages that are both build and exec dependencies -->

<!-- <depend>roscpp</depend> -->

<!-- Note that this is equivalent to the following: -->

<!-- <build\_depend>roscpp</build\_depend> -->

<!-- <exec\_depend>roscpp</exec\_depend> -->

<!-- Use build\_depend for packages you need at compile time: -->

<!-- <build\_depend>message\_generation</build\_depend> -->

<!-- Use build\_export\_depend for packages you need in order to build against this package: -->

<!-- <build\_export\_depend>message\_generation</build\_export\_depend> -->

<!-- Use buildtool\_depend for build tool packages: -->

<!-- <buildtool\_depend>catkin</buildtool\_depend> -->

<!-- Use exec\_depend for packages you need at runtime: -->

<!-- <exec\_depend>message\_runtime</exec\_depend> -->

<!-- Use test\_depend for packages you need only for testing: -->

<!-- <test\_depend>gtest</test\_depend> -->

<!-- Use doc\_depend for packages you need only for building documentation: -->

<!-- <doc\_depend>doxygen</doc\_depend> -->

<buildtool\_depend>**catkin**</buildtool\_depend>

<build\_depend>**roscpp**</build\_depend>

<build\_depend>**rospy**</build\_depend>

<build\_depend>**std\_msgs**</build\_depend>

<build\_export\_depend>**roscpp**</build\_export\_depend>

<build\_export\_depend>**rospy**</build\_export\_depend>

<build\_export\_depend>**std\_msgs**</build\_export\_depend>

<exec\_depend>**roscpp**</exec\_depend>

<exec\_depend>**rospy**</exec\_depend>

<exec\_depend>**std\_msgs**</exec\_depend>

<!-- The export tag contains other, unspecified, tags -->

<export>

<!-- Other tools can request additional information be placed here -->

</export>

</package>

## **11.7. Código del conversor de datos.**

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include "sensor\_msgs/PointCloud.h"

#include "sensor\_msgs/PointCloud2.h"

#include "sensor\_msgs/point\_cloud\_conversion.h"

class convert\_PointCloud\_to\_PointCloud2

**{**

public**:**

convert\_PointCloud\_to\_PointCloud2**()**

**{**

//Topic nuevo en el que se guarda el nuevo formato de PointCloud2

pub\_ **=** n\_**.**advertise**<**sensor\_msgs**::**PointCloud2**>(**"/UNEXMIN/laser/scan2"**,** 1000**);**

//Topic del que adquirimos los datos con el formato erróneo

sub\_ **=** n\_**.**subscribe**(**"UNEXMIN/laser/scan"**,** 1000**,** **&**convert\_PointCloud\_to\_PointCloud2**::**callback**,** **this);**

**}**

//Función callback de la subscripción

void callback**(**const sensor\_msgs**::**PointCloud**::**ConstPtr**&** input**)**

**{**

sensor\_msgs**::**PointCloud2 output**;**

//Se cnvierten los datos al formato deseado y se guarda en la variable output

sensor\_msgs**::**convertPointCloudToPointCloud2 **(\***input**,**output**);**

//Publicamos dicha variable

pub\_**.**publish**(**output**);**

**}**

private**:**

ros**::**NodeHandle n\_**;**

ros**::**Publisher pub\_**;**

ros**::**Subscriber sub\_**;**

**};**// se termina la clase.

int main**(**int argc**,** char **\*\***argv**)**

**{**

//Se inicia ROS

ros**::**init**(**argc**,** argv**,** "convert\_PointCloud\_to\_PointCloud2"**);**

//Se crea un objeto de la clase creada

convert\_PointCloud\_to\_PointCloud2 SAPObject**;**

ros**::**spin**();**

**return** 0**;**

**}**

## **11.8. Código del Broadcaster.**

#include <ros/ros.h>

#include <tf/transform\_broadcaster.h>

#include <tf/transform\_listener.h>

#include <gazebo\_msgs/ModelStates.h>

#include <geometry\_msgs/Pose.h>

#include <geometry\_msgs/Twist.h>

bool nameSet **=** **false;**

void Callback**(**const gazebo\_msgs**::**ModelStates**::**ConstPtr**&** msg**){**

int model\_order **=** 0**;** // Varaible donde se guarda el nobre del modelo (my\_UNEXMIN)

std**::**vector**<**std**::**string**>** model\_names **=** msg**->**name**;**

std**::**string modelName **=** "my\_UNEXMIN"**;**

// msg->name es un vector que contiene los nombres de todos los modelos, por lo que hay que ir a través del vector buscando el que necesitamos

**if(!**nameSet**)**

**{**

**for** **(**int i **=** 0**;** i **<** model\_names**.**size**();** i**++)**

**{**

**if** **(**model\_names**[**i**]** **==** modelName**)**

**{**

model\_order **=** i**;**

**break;**

**}**

**}**

**}**

// Se declara un Broadcaster

static tf**::**TransformBroadcaster br**;**

// En esta variable se guardará la loaclización del robot

geometry\_msgs**::**Pose pose**;**

// Se guarda la localización

pose **=** msg**->**pose**.**at**(**model\_order**);**

// Se declara un objeto tf

tf**::**Transform transform**;**

// Se realizan las transformaciones tanto de posición como de orientación

transform**.**setOrigin**(** tf**::**Vector3**(**pose**.**position**.**x**,** pose**.**position**.**y**,** pose**.**position**.**z**)** **);**

tf**::**Quaternion q**(**pose**.**orientation**.**x**,** pose**.**orientation**.**y**,** pose**.**orientation**.**z**,** pose**.**orientation**.**w**);**

transform**.**setRotation**(**q**);**

//Se envía la transformada

br**.**sendTransform**(**tf**::**StampedTransform**(**transform**,** ros**::**Time**::**now**(),** "map"**,** "base\_link"**));**

**}**

int main**(**int argc**,** char**\*\*** argv**){**

// Se inicia ROS

ros**::**init**(**argc**,** argv**,** "robot\_tf\_publisher"**);**

ros**::**NodeHandle node**;**

// Se establece la subscripción al topic necesario

ros**::**Subscriber sub **=** node**.**subscribe**(**"/gazebo/model\_states/"**,** 10**,** **&**Callback**);**

ros**::**spin**();**

**return** 0**;**

**};**

## **11.9. Código del Listener.**

#include <ros/ros.h>

#include <gazebo\_msgs/ModelState.h>

#include <geometry\_msgs/Pose.h>

#include <geometry\_msgs/Twist.h>

#include <tf/transform\_listener.h>

int main**(**int argc**,** char**\*\*** argv**){**

// Se inicia ROS

ros**::**init**(**argc**,** argv**,** "robot\_tf\_listener"**);**

ros**::**NodeHandle n**;**

// Se declara un objeto de tipo listener, definido en su propia librería

tf**::**TransformListener listener**;**

**while** **(**n**.**ok**()){**

tf**::**StampedTransform world\_point**;**

//Se aplican las transformaciones

**try{**

listener**.**lookupTransform**(**"base\_link"**,** "map"**,**ros**::**Time**(**0**),** world\_point**);**

**}**

// Se imprime por pantalla los errores

**catch** **(**tf**::**TransformException**&** ex**){**

ROS\_ERROR**(**"Received an exception trying to transform a point from \"base\_link\" to \"map\": %s"**,** ex**.**what**());**

ros**::**Duration**(**1.0**).**sleep**();**

**continue;**

**}**

**}**

**return** 0**;**

**};**

## **11.10. Código del octomap\_server.launch.**

<launch>

<node pkg="octomap\_server" type="octomap\_server\_node" name="octomap\_server">

<param name="resolution" value="0.01" />

<!-- fixed map frame (set to 'map' if SLAM or localization running!) -->

<param name="frame\_id" type="string" value="map" />

<param name="latch" value="False" />

<!-- maximum range to integrate (speedup!) -->

<param name="sensor\_model/max\_range" value="3.0" />

<!-- data source to integrate (PointCloud2) -->

<remap from="cloud\_in" to="/UNEXMIN/laser/scan2" />

</node>

</launch>