Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de un robot de servicio para la asistencia a personas de la tercera edad.

Proyecto Final de Carrera

Ingeniería Informática

Autor: Jorge Ariño Sustaeta

Director: Ángel Valera Fernández

7/9/2014

Resumen

El presente proyecto consistirá en el diseño e implementación de un robot de servicio que pueda proporcionar asistencia a personas de la tercera edad o discapacitadas. Este robot se basará en una plataforma móvil para interiores, y proporcionará funciones de monitorización del usuario, siendo capaz de detectar una caída y de avisar al servicio de urgencias. El robot también ofrecerá funciones de teleoperación mediante comandos de voz.

Para la construcción de la plataforma móvil se ha utilizado un diseño similar al del robot TurtleBot, aunque con algunas ligeras modificaciones. Este robot se moverá usando la base móvil iRobot Create, y las funciones de reconocimiento visual del entorno y de reconocimiento del habla del usuario se realizarán con el sensor Microsoft Kinect. Estos dos elementos estarán unidos mediante conectores USB a un Netbook, ubicado en la plataforma móvil, donde se realizarán todos los procesos de software.

El software se ha desarrollado utilizando la plataforma robótica ROS, que ofrece la flexibilidad necesaria para poder interconectar todos los elementos del robot. Para el reconocimiento visual se ha hecho un uso extensivo de la librería OpenNI (Open Natural Interaction), que ofrece varias funcionalidades de reconocimiento de usuarios. Y para el reconocimiento y procesamiento de los comandos de voz del usuario se ha utilizado la librería Sphinx, desarrollada por la Carnegie Mellon University.

Palabras clave: robot, servicio, dependencia, asistencia, Kinect, Create, ROS, reconocimiento, voz

Tabla de contenidos

1. Introducción 6
   - 1.1. Motivación 6
   - 1.2. Objetivos 7
2. Desarrollo teórico 9
   - 2.1. La robótica 9
     - 2.1.1. Introducción a la robótica 9
     - 2.1.2. Tipos de robots 11
     - 2.1.3. Robótica móvil 15
     - 2.1.4. Componentes de un robot 21
       - 2.1.4.1. Sensores 21
       - 2.1.4.2. Actuadores 25
       - 2.1.4.3. Control 26
   - 2.2. Estado del arte de la robótica de servicio para la asistencia de discapacitados 28
     - 2.2.1. Plataformas móviles 28
     - 2.2.2. Asistencia al movimiento 30
   - 2.3. ROS 31
     - 2.3.1. Introducción 32
     - 2.3.2. Características principales 32
     - 2.3.3. Arquitectura 34
     - 2.3.4. Herramientas de bajo nivel 35
     - 2.3.5. Transformadas (tf) 39
     - 2.3.6. Robot description lenguaje (URDF) 40
     - 2.3.7. Herramientas de visualización y simulación 42
   - 2.4. OpenNI 46
3. Desarrollo práctico 50
   - 3.1. TurtleBot 50
     - 3.1.1. Presentación del robot 50
     - 3.1.2. Características 53
     - 3.1.3. Sensores 54
   - 3.2. Desarrollo del robot perseguidor 60
     - 3.2.1. Descripción 60

3.2.2. Descripción del software utilizado 61
3.2.3. Instalación del espacio de trabajo 65
3.2.4. Instalación y uso de los paquetes del TurtleBot 67
3.2.5. Instalación y uso de los paquetes de la Kinect 68
3.2.6. Estructura 72
3.2.7. Creación del tracker..76
3.2.8. Nodo de movimiento 85
3.3. Detección de caídas.. 89
3.3.1. Descripción 89
3.3.2. Desarrollo 90
3.4. Reconocimiento de voz 93
3.4.1. Descripción... 93
3.4.2. Instalación 94
3.4.3. Uso de PocketSphinx 96
3.4.4. Estructura utilizada 99
3.4.5. Desarrollo 100
4. Conclusiones y trabajos futuros 101
5. Referencias 103
Anexo 1. Archivo turtlebot_follow_user_commands.launch 105

1. Introducción

1.1. Motivación

Debido al aumento de la esperanza de vida en los últimos años, cada vez existe un mayor número de personas con discapacidad o algún tipo de dependencia. Según los datos extraídos de ***la*** *Encuesta sobre Discapacidad, Autonomía personal y situaciones de Dependencia* (EDAD) [1], con datos de 2008, el 8,5% de la población española (3.847.854 personas) declara alguna discapacidad o limitación para las actividades de la vida diaria. De estas personas con discapacidad, un 55,8% (2.148.548 personas) se consideran dependientes, es decir, que precisan de la ayuda de alguna tercera persona pues no pueden valerse por sí mismas. Además, el 18,1% de las personas mayores de 65 años en situación de dependencia viven solas.

La robótica, al igual que otras muchas ramas de la tecnología, ha logrado numerosos avances en los últimos años. Se han conseguido grandes progresos tecnológicos en los sensores de los robots, siendo ahora capaces de sentir presiones o temperaturas mediante sensores táctiles, o de reconocer diferentes gestos, expresiones faciales o posturas mediante visión artificial. La forma con la que los robots interactúan con el entorno también ha mejorado, perfeccionando los algoritmos de control de brazos y manos para que puedan interactuar en un entorno humano. Asimismo, se han producido muchas mejoras en el ámbito de la interacción entre humanos y máquinas, intentando proporcionar una interfaz más accesible. Prueba de ello son los avances en temas como el reconocimiento de voz, la voz robótica, o las expresiones faciales de los robots, que consiguen simular las expresiones humanas. Por último, cabe mencionar la importancia de los progresos realizados en materia de navegación autónoma, siendo destacables los resultados obtenidos actualmente por algunas empresas en el desarrollo de los coches autónomos.

El objetivo de la tecnología es mejorar y hacer más fácil nuestra vida, siendo especialmente útil en los casos en que existe algún impedimento, discapacidad o dependencia. La robótica de servicio es la rama de la robótica que tiene como objetivo ayudar a los seres humanos, mediante la realización de trabajos peligrosos o repetitivos, o simplemente realizando tareas que una persona no pueda realizar. Esta robótica de servicio es un área de la robótica que aún está dando sus primeros pasos, por lo que creemos que realizar este proyecto de robot de servicio puede contribuir en alguna medida a su desarrollo.

En concreto se ha querido realizar un robot de servicio que ofrezca un cierto elemento de seguridad a las personas con discapacidades que viven solas. Muchas de estas personas son capaces de realizar independientemente la mayoría de las tareas diarias, pero tienen un alto riesgo de sufrir una caída, debido a sus discapacidades relacionadas con la movilidad. Actualmente existen algunos sistemas que permiten a la persona ponerse en contacto con un servicio de urgencias cuando esta persona sufre una caída, pero estos sistemas de alarma suelen consistir en activar algún mecanismo manualmente, por lo que no funcionarían en situaciones donde la persona perdiese la consciencia o no pudiese activar el mecanismo por cualquier motivo. Por esa razón, el robot de servicio que se quiere desarrollar deberá tener como principal misión tanto la detección de las caídas como el aviso al servicio de urgencias de forma autónoma. La concreción de los objetivos del proyecto se especifica en el siguiente apartado.

1.2. Objetivos

- Realizar un seguimiento automático de una persona por interiores, siendo capaz de hacerlo a distintas velocidades de paso, y teniendo como máxima prioridad evitar contactos físicos con la persona.
- Ofrecer una monitorización del estado de la persona, mediante herramientas visuales o sonoras, que permitirá saber si la persona ha sufrido algún accidente o si necesita algún tipo de ayuda o socorro.

- En caso de emergencia, el robot se pondrá en contacto con los servicios de urgencia (112) o con otro servicio privado de emergencias y ofrecerá un estado visual de la persona que necesita ayuda.
- El robot debe ser capaz de entender y procesar diferentes comandos de voz para activar diferentes funcionalidades, tales como iniciar una teleoperación del robot mediante comandos de voz, iniciar el seguimiento automático o realizar una llamada de socorro.

2. Desarrollo teórico

2.1. La robótica

2.1.1. Introducción a la robótica

A lo largo de la historia, los seres humanos han intentado imitar mediante artilugios o maquinas el comportamiento de los seres vivos. Ya los antiguos egipcios crearon sistemas mecánicos para poder imitar el movimiento de los brazos de los seres humanos; estos artilugios estaban en las estatuas de sus dioses. También, los antiguos griegos construyeron estatuas con sistemas hidráulicos para así poder impresionar a la gente que iba a orar a sus templos.

En cambio, hasta el siglo I, no existe ninguna documentación relacionada con la construcción de un autómata. La primera persona que documenta cualquier cosa relacionada con autómatas fue el matemático, físico e inventor griego Herón de Alejandría, que describe múltiples ingenios mecánicos en su libro Los Autómatas, como por ejemplo aves que vuelan, gorjean y beben.

Todos ellos fueron diseñados como juguetes, sin mayor interés por encontrarles una aplicación. Aun así, se describen algunos ingenios que sí presentaban una función útil, como es el caso de un molino de viento para accionar un órgano o un precursor de la turbina de vapor, conocida como Eolípila.

Los ingenios que surgieron en la escuela de Alejandría, se extendieron por diversas culturas, pasando por el imperio romano, posteriormente por los mundos árabes, y por el imperio chino, siendo todo esto, los autómatas antecesores de todo lo que conocemos hoy en día. No fue, hasta la época de la cultura árabe, cuando ya heredados y difundidos los conocimientos griegos, un inventor llamado Al-jazari, dio utilidad a todos esos inventos y además inventó nuevos artefactos de control automático, como fue el cigüeñal o unos relojes mecánicos movidos por pesos y agua.

Sin embargo, no fue hasta 1921 cuando se escuchó por primera vez la palabra robot, utilizada por el escritor checo Karel Capek en su obra de teatro R.U.R

(Rossum´s Universal Robots). La palabra robot viene del vocablo checo 'Robota' que significa "trabajo", entendido como servidumbre, trabajo forzado o esclavitud.

Los primeros robots industriales empezaron a producirse a principios de los años 60 y estaban diseñados principalmente para realizar trabajos mecánicos difíciles y peligrosos. Las áreas donde estos robots tuvieron su aplicación fueron trabajos laboriosos y repetitivos, como la carga y descarga de hornos de fundición. En 1961 el inventor norteamericano George Devol patentaba el primer robot programable de la historia, conocido como Unimate, estableciendo las bases de la robótica industrial moderna.

Este robot industrial era un manipulador que formaba parte de una célula de trabajo en la empresa de automóviles Ford Motors Company, diseñado para levantar y apilar grandes piezas de metal caliente, de hasta 225 kg, de una troqueladora de fundición por inyección.

Debido a los continuos avances en la informática y la electrónica, a partir de 1970 fueron desarrollados diversos robots programables, siendo de gran importancia en la industria mecánica, tanto en las líneas de ensamblaje como en aplicaciones como la soldadura o pintura.

En los últimos años, los robots han tomado posición en todas las áreas productivas industriales. La incorporación del robot al proceso productivo ha representado uno de los avances más espectaculares de la edad moderna. En poco más de cuarenta años, se ha pasado de aquellos primeros modelos, rudos y limitados, a sofisticadas máquinas capaces de sustituir al hombre en todo tipo de tareas repetitivas o peligrosas, y además, hacerlo de forma más rápida, precisa y económica que el ser humano. Hoy en día, se calcula que el número de robots industriales instalados en el mundo es de un millón de unidades, unos 20.000 en España, siendo Japón el país más tecnológicamente avanzado, con una media de 322 robots por cada 10.000 trabajadores.

2.1.2. Tipos de robots

La progresiva y creciente utilización de robots en el proceso productivo industrial ha dado lugar al desarrollo de controladores industriales rápidos y potentes, basados en microprocesadores y PC, así como un empleo de actuadores en bucle cerrado que permiten establecer con exactitud la posición real de los elementos del robot y su desviación o error. La evolución que se ha ido realizando desde los 60's ha dado origen a una serie de tipos de robots, que trataremos de explicar a continuación:

Robots industriales

Es un artefacto mecánico y electrónico destinado a realizar de forma automática determinados procesos de fabricación o manipulación de todo tipo de artículos. Estos aparatos poseen una o más articulaciones que les permiten actuar con precisión Estos robots suelen agruparse dependiendo de los grados de movilidad que tenga. En el extremo de la articulación o brazo robótico pueden tener una ventosa de absorción o una pinza, dependiendo del tipo de material que queremos manipular.

Ilustración 1 Robot Manipulador Fijo (Brazo Robótico)

Estos son actualmente los más utilizados en la industria en el proceso productivo sobre todo donde hay que ordenar piezas en grandes cantidades o mover artículos de gran peso, aunque también tienen adaptaciones como puede ser el realizar el pintado de un vehículo en la industria del automóvil. A estos robots manipuladores también se les suele llamar brazo robot por lo analogía que se puede establecer con las extremidades superiores del cuerpo humano.

Estos robots son sistemas mecánicos multifuncionales que permiten gobernar el movimiento en los siguientes modos:

- **Manual**: Cuando el operario controla directamente la tarea del manipulador.
- **De secuencia Fija**: cuando se repite, de forma invariable, el proceso de trabajo preparado previamente por los programadores
- **De secuencia variable**: Se pueden alterar algunas características de los ciclos de trabajo por parte de algún operario.

Existen muchas operaciones básicas que pueden ser realizadas mediante robots manipuladores, por eso son generalmente muy utilizados cuando las funciones de trabajo a desarrollar son sencillas y repetitivas.

Robots de repetición o aprendizaje

Son parte de la familia de los robots manipuladores, aunque su función no es más que repetir una vez y otra una secuencia programada antes por un humano, haciendo uso de un controlador manual o un dispositivo auxiliar.

Actualmente, los robots de aprendizaje son los más conocidos en algunos sectores de la industria, y el tipo de programación que incorporan recibe el nombre de “gestual”. Este tipo de programación se desarrolla mediante diferentes interfaces, como una pistola de programación, teclas, joysticks o mediante un maniquí.

Ilustración 2 Robot manipulador de repetición

Robots controlados por Ordenador

Son manipuladores o sistemas mecánicos multi-funcionales, que están controlados por ordenador. El control por ordenador dispone de un lenguaje específico de programación, compuesto por varias instrucciones adaptadas al hardware del robot, como las que se puede diseñar un programa de aplicación utilizando solo el ordenador.

Entre las ventajas que nos ofrece este tipo de robots hacen que se vayan imponiendo en el proceso productivo rápidamente, lo que exige la preparación de personal cualificado, capaz de desarrollar programas de control que permitan el manejo del robot.

Ilustración 3 Robot bípedo controlado por ordenador

Robots inteligentes

Son similares a los del grupo anterior, pero tienen la capacidad de poder relacionarse con el mundo que les rodea a través de sensores y de tomar decisiones en función de la información obtenida en tiempo real. De momento se encuentran en fase experimental, donde grupos de investigadores se esfuerzan por hacerlos más efectivos. El reconocimiento de imágenes y algunas técnicas de inteligencia artificial son los campos que más se están estudiando para su posible aplicación en estos robots.

Ilustración 4 Evolución del Robot Asimo

Robots para la educación o Micro-Robots

Son Robots con fines educacionales, para el entretenimiento o investigación. Existen diferentes tipos de robots a un precio bajo, siendo su estructura y funcionamiento muy similares a los de aplicación industrial. Estos robots se han hecho un hueco en universidades y centros de investigación, debido a que son una de las formas más económicas de experimentar con tareas robóticas. El personal de dichos centros ha apostado por este tipo de robots para estimular el interés de sus alumnos por la ciencia y la tecnología.

Ilustración 5 Robot SR1 de un kit de aprendizaje

2.1.3. Robótica móvil

En este apartado vamos a clasificar los diferentes tipos de robots móviles que existen según el grado de movilidad disponible.

Robots rodantes

Son robots que se desplazan mediante ruedas o dispositivos similares siempre mantienen el contacto con el suelo. Según la distribución de estas ruedas, un robot móvil puede tener una gran cantidad de modelos cinemáticos diferentes. A continuación se muestran tres ejemplos de configuración de ruedas:

- **Configuración Triciclo**: Incluyen 2 ruedas motrices fijas y una tercera rueda delantera que tiene una movilidad de 360 grados y se orienta según la tracción que hacen las ruedas motrices.

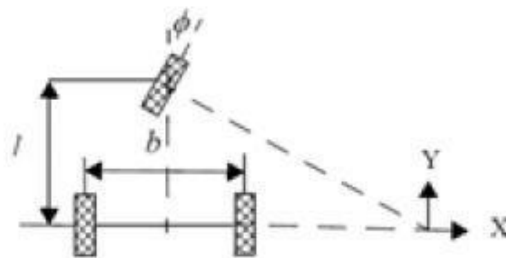

Ilustración 6 Configuración triciclo

- **Configuración Ackerman**, es la configuración que se usa en los coches. En esta configuración dos ruedas delanteras son las encargadas de la dirección mientras que las ruedas traseras se mantienen fijas y son las que se encargan de la motricidad.

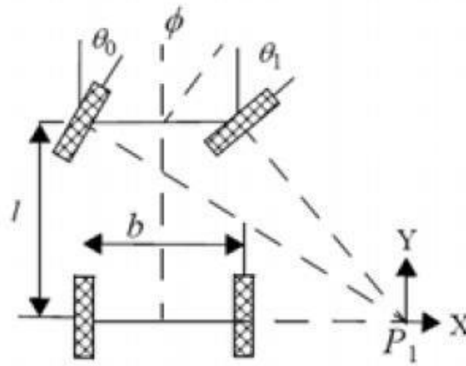

Ilustración 7 Configuración Ackerman

- **Configuración Tipo Oruga o Skid Steering:** Esta configuración es la que usan los tanques, donde el giro a realizar se basara en la diferencia de rodamiento de cada cadena.

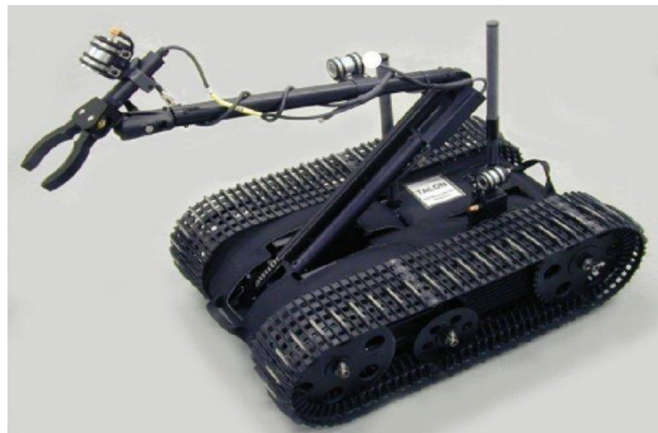

Ilustración 8 Configuración oruga

Robots andantes

Estos robots se desplazan mediante el uso de patas robóticas, intentando imitar el movimiento de ciertos animales. Una de las principales ventajas de los robots andantes respecto a los robots rodantes es que pueden acceder a zonas con suelo irregular, aunque el sistema de control de los robots andantes es más complicado y generalmente ofrece una menor estabilidad. Estos robots los podemos clasificar en función del número de patas que los compone:

- **Bípedos**: Estos robots se mueven como lo hacemos los seres humanos mediante la utilización de dos piernas.

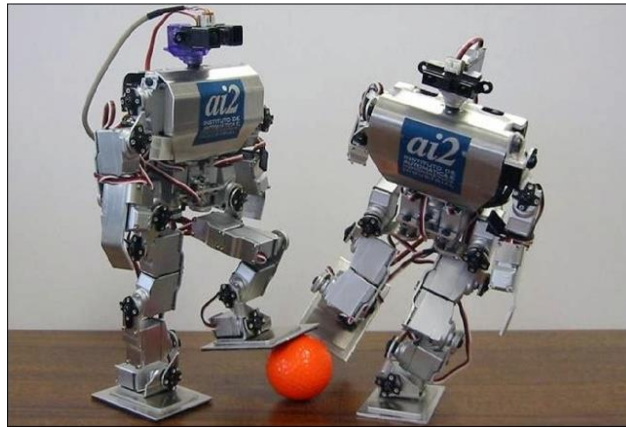

Ilustración 9 Robot bípedo realizado por el ai2 para la robocup.

- **Cuadrúpedos**: Estos simulan los movimientos de los animales de cuatro patas. Este tipo de robots ha despertado un gran interés en las aplicaciones militares, al ser una distribución que puede moverse con gran facilidad además de poder portar peso y no subir grandes desequilibrios. El caso más conocido es el big-dog que ha sido desarrollado por Boston Dynamics con financiación de DARPA.

Ilustración 10 Big Dog desarrollado por Boston Dynamics

- **Hexápodos**: Estos simulan el movimiento de animales de seis patas.

Ilustración 11 Robot hexápodo que simula el movimiento de una araña

Robots reptiles

Se desplazan arrastrándose por la superficie de contacto de forma muy similar a como lo realizan los reptiles. Están formados un número elevado de secciones que pueden cambiar de tamaño o posición de forma independiente de las demás, de forma que en conjunto provoquen el desplazamiento del robot.

Ilustración 12 Robot reptador

Robots a hélices

En la actualidad los robots a hélices se pueden dividir en tres grandes familias:

- **Robots Acuáticos**: Estos robots son capaces moverse en un entorno acuático, y están generalmente enfocados a tareas de exploración submarina de zonas donde no es posible llegar con vehículos tripulados. A parte de lo puramente anecdótico, se ha demostrado que la estructura corporal de los peces así como el movimiento que realizan durante su desplazamiento en el agua es uno de los métodos más óptimos de movimiento submarino dado que aprovecha la energía de forma muy eficiente y permite mayor control en la navegación, produciendo mucho menos ruido y turbulencias. Es por todo esto que se

está tendiendo a estudiar y emular en lo posible el comportamiento de estos animales a la hora de crear nuevos robots subacuáticos.

Ilustración 13 Robot subacuático

- **Robots Voladores**: Su Estructura puede ser parecida a la de un Helicóptero RC, aunque últimamente predominan los Quadcopteros como el que se muestra en la Imagen.

Ilustración 14 Robot volador

- **Drones o aviones no tripulados**: Los drones forman parte de los robots voladores. Actualmente muchos ejércitos militares tienen el apoyo de estos aviones donde realizan desde operaciones de logística, tareas de espionaje y reconocimiento cartográfico del terreno.

Ilustración 15 Dron del ejército de EEUU

Dejando de lado la clasificación dependiendo del tipo de movilidad, cualquiera de los anteriores puede ser:

- **Autónomo**: una vez programados tiene capacidad propia de realizar la tarea que se les ha mandado sin necesidad de interacción con el operario.

- **Tele-operado**: Deben ser operados continuamente, de tal forma que si el operario no les manda ninguna acción este no realiza nada, aunque hay casos que tienen programadas operaciones autónomas que el usuario monitoriza.

2.1.4. Componentes de un robot

2.1.4.1. Sensores

Estos elementos son los encargados de adquirir información del entorno y transmitirla a la unidad de control del robot. Una vez esta es analizada, el robot realiza la acción correspondiente a través de sus actuadores. Los sensores constituyen el sistema de percepción del robot, es decir, facilitan la información del mundo real para que el robot la interprete. Los tipos de sensores más utilizados son los siguientes:

- **Sensor de temperatura:** Capta la temperatura del ambiente, de un objeto o de un punto determinado.

Ilustración 16 Sensores de temperatura

- **Sensor de proximidad:** Detecta la presencia de un objeto ya sea por rayos Infrarrojos, por sonar, magnéticamente o de otro modo. Los sensores de proximidad se dividen en dos grandes grupos los Inductivos y los Capacitivos.

    - **Inductivos**: Se basan en el fenómeno de amortiguamiento que se produce en un campo magnético a causa de las corrientes inducidas en materiales situados en las cercanías del material.

    - **Capacitivos**: Se basan en la detección de la capacidad parasita que se origina entre el detector propiamente dicho y el objeto cuya distancia se desea medir.

Ilustración 17 Sensores de proximidad

- **Sensores Magnéticos:** Se utilizan a modo de brújulas para orientación geográfica, su funcionamiento se basa en la variación del campo magnético.

Ilustración 18 Sensores magnéticos

- **Sensores de velocidad, de vibración (Acelerómetro) y de inclinación:** Se emplean para determinar la velocidad de actuación de las distintas partes móviles del propio robot o cuando se produce una vibración. También se detecta la inclinación a la que se encuentra el robot o una parte de él.

Ilustración 19 Sensor acelerómetro

- **Sensores de presión:** Permiten controlar la presión que ejerce la mano del robot al coger un objeto.

Ilustración 20 Sensor de presión

- **Micro interruptores:** Muy utilizados para detectar finales de carrera.

Ilustración 21 Diferentes tipos de microinterruptores

- **Sensor Laser (LIDAR):** Los Sensores Laser (LIDAR, acrónimo del inglés Laser Imaging Detection and Ranging) es una tecnología que permite determinar la distancia desde un emisor laser a un objeto o superficie utilizando un haz laser pulsado. En este tipo de sensores calculan la distancia al objeto e determina midiendo el tiempo de retraso entre la emisión del pulso y su detección a través de la señal reflejada.

Ilustración 22 Sensor LIDAR

2.1.4.2. Actuadores

Los actuadores son los elementos encargados de transformar la información enviada en movimientos que realizara el robot. Hay dos grandes grupos dependiendo de la energía que utilicen.

- **Actuadores eléctricos**: son los motores eléctricos que utilizan la energía eléctrica para que el robot ejecute sus movimientos se emplean en robots de tamaño medio que no necesitan tanta potencia para tener que funcionar con actuadores por impulsión hidráulica. Hay dos tipos de motores eléctricos:

  - **Los motores eléctricos de corriente continua**: se utilizan para proporcionar movimientos giratorios en los que se requiere mucha precisión.

Ilustración 23 Motor de corriente continua

  - **Los motores paso a paso**: permiten controlar de forma precisa el ángulo de giro del motor, haciendo que el robot se coloque en una posición determinada.

Ilustración 24 Motor paso a paso

- **Actuadores hidráulicos y neumáticos.** Son los solenoides, formados por una bobina de hilo de cobre esmaltado, en cuyo interior se mueve un núcleo de hierro cuando se hace circular una corriente por el bobinado.

Ilustración 25 Cilindro hidráulico

2.1.4.3. Control

El control de un robot puede realizarse de diversas maneras, pero casi siempre se suele realizar mediante un ordenador con una capacidad de potencia bastante elevada, el controlador se encarga de almacenar y procesar la información de los diferentes elementos que conforman el robot.

Un sistema de control está definido como un conjunto de componentes que pueden regular su propia conducta o la de otro sistema con el fin de lograr un funcionamiento predeterminado, de modo que se reduzcan las probabilidades de fallos y se obtengan los resultados buscados.

Existen dos tipos de sistemas:

- **Sistemas en bucle abierto:** Es aquel sistema en que solo actúa el proceso sobre la señal de entrada y da como resultado una señal de salida independiente a la señal de entrada, pero basada en la primera. Esto significa que no hay retroalimentación hacia el controlador para que este pueda ajustar la acción de control.

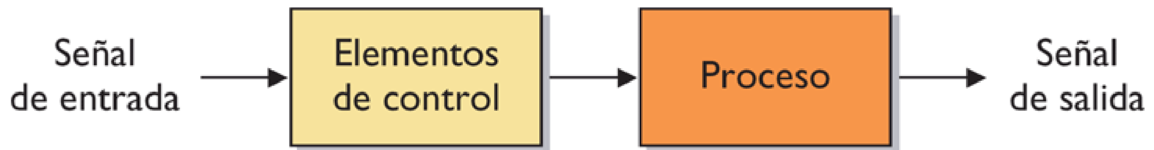

Ilustración 26 Sistema de control en bucle abierto

- **Sistemas en bucle cerrado:** Son los sistemas en lo que la acción de control está en función de la señal de salida. Los sistemas de circuito cerrado usan la retroalimentación desde un resultado final para ajustar la acción de control en consecuencia

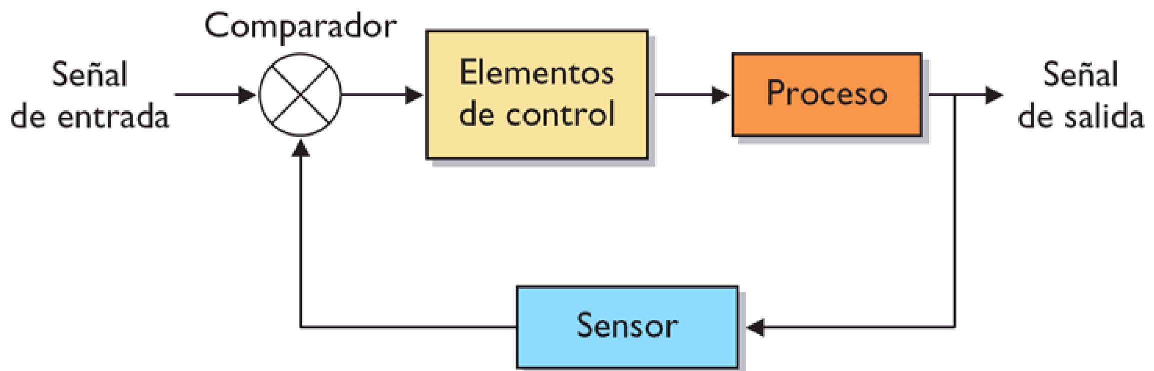

Ilustración 27 Sistema de control en bucle cerrado

El control de robots suele estar realizado mediante sistemas discretos en lazo cerrado, realizados por ordenador. El ordenador procesa la información captada por los sensores y activa los actuadores en intervalos lo más cortos posibles.

2.2. Estado del arte de la robótica de servicio para la asistencia de discapacitados

Los robots de servicio son usados para ayudar en las tareas cotidianas. Las personas ancianas normalmente tienen algunas dificultades realizando estas tareas, así que estas pueden ser ayudadas por los robots de servicio. Estos robots deben adaptarse a las necesidades que se requieren en un entorno adaptado a los humanos. Por ejemplo, deberían ser capaces de moverse por una casa, abrir puertas o realizar tareas simples como recoger objetos pequeños o activar interruptores.

2.2.1. Plataformas móviles

La prioridad de los robots de servicio que se basan en plataformas móviles es que se muevan de forma segura sin ser un peligro para los humanos o los objetos del entorno. **HOSPI-RIMO** [2], desarrollado por Panasonic, es un robot diseñado para trabajar en un entorno de hospital, aunque también puede ser usado en casas. Su principal objetivo es proporcionar un servicio de medicación automática a los pacientes. Utiliza una serie de sensores para evitar los obstáculos mientras se mueve por los diferentes pasillos del hospital. HOSPI-RIMO puede funcionar de forma autónoma o teleoperada por un operario.

Ilustración 28 Robot HOSPI-RIMO

Además de moverse, estos robots deberían ser capaces de efectuar ciertas acciones. La manera más común de realizar acciones es usando brazos y manos robóticas. El **Human Support Robot (HSR)** [3], desarrollado por Toyota es una plataforma móvil diseñada para ayudar a las personas durante sus tareas cotidianas, como recoger pequeños elementos del suelo, mover las cortinas o abrir y cerrar ventanas. HSR está equipado con un brazo robótico con una simple garra que le permite coger objetos pequeños, y una pequeña aspiradora que le permite coger objetos más ligeros como papel o cartas. HSR puede ser controlado mediante una interfaz gráfica o mediante comandos de voz. Debido a que este robot está diseñado para trabajar en un entorno con humanos, su prioridad es la seguridad. Para ello, el brazo robótico se mueve a una velocidad limitada para evitar accidentes, y comprueba el estado de su entorno para evitar colisiones usando una cámara estéreo ubicada en su cabeza. Además, su ligero peso asegura que en caso de colisión el peligro será mínimo.

Ilustración 29 Human Support Robot (HSR)

Uno de los robots de servicio más avanzados, diseñado especialmente para el cuidado de enfermos y discapacitados es el robot **RIBA** [4], desarrollado por la empresa RIKEN. Este robot es una mejora de su último robot, RI-MAN. RIBA es capaz de levantar o bajar a una persona de una silla de ruedas o de una cama. Puede hacer esto debido a sus dos fuertes brazos robóticos con 7 grados de libertad que le permiten levantar un peso máximo de 61Kg. RIBA utiliza un total de 6 paneles de sensores táctiles, ubicados en cada brazo, en cada antebrazo y en cada mano, que pueden ser usados para guiar el movimiento del robot, simplemente presionando suavemente sobre el panel de sensores en la dirección en la que queremos que se

mueva. Estos paneles táctiles también se usan para detenerse en caso de un contacto inesperado. Puede seguir a los humanos, mediante localización de la cara de la persona, usando dos cámaras estéreo. También puede localizar a las personas mediante el sonido, utilizando dos micrófonos, y triangulando la posición de donde viene el sonido. Por último, las articulaciones de RIBA están construidas con materiales suaves, lo que le permite interactuar de forma segura con humanos.

Ilustración 30 Robot RIBA

2.2.2. Asistencia al movimiento

Una de las principales discapacidades de las personas mayores es su dificultad para moverse. Algunos robots están diseñados para dar más independencia a los usuarios. Por ejemplo, el robot **RoboticBed** [5], que es una cama que se puede transformar en una silla de ruedas. El usuario puede activar esta transformación mediante comandos de voz. La silla de ruedas tiene movimiento teleoperado, e incorpora un sensor LIDAR para evitar obstáculos mientras se mueve. Cuando está en forma de cama, RoboticBed ofrece un monitor donde el usuario puede seleccionar diferentes funcionalidades, como ver la televisión, comprobar las cámaras de seguridad o realizar una videoconferencia.

Ilustración 31 Robotic Bed

Otros robots que ofrecen asistencia al movimiento son el **RobuWalker** [6] y el **Monimad** [7]. Estos dos robots son andadores que ofrecen una ayuda extra al usuario a la hora de levantarse, caminar y sentarse. Para ofrecer esta asistencia cuentan con un brazo robótico en el que los usuarios pueden apoyarse mientras se levantan o se sientan.

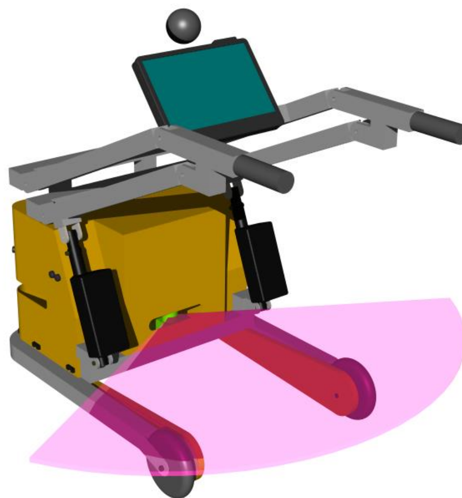

Ilustración 32 RobuWalker

2.3. ROS

2.3.1. Introducción

El desarrollo de software para robots se está convirtiendo en un proceso cada vez más complicado, en parte debido a que los usos de la robótica no paran de crecer, cada vez en ámbitos más dispares. Para cada uso diferente se requiere un tipo de robot diferente, requiriendo distintos tipos de hardware, lo que origina el problema de que se necesita un software nuevo para cada robot, dificultando en gran medida la reutilización del código. Otro problema aparece en la cantidad de niveles distintos en los que el desarrollador de robots tiene que trabajar, a menudo teniendo que programar desde los drivers de las ruedas, hasta la percepción, o el razonamiento de alto nivel del robot. Para superar estos problemas, la industria robótica ha creado diferentes frameworks para facilitar el prototipado y desarrollo de robots, cada uno de estos frameworks intentando solventar algunos de los problemas citados anteriormente, o simplemente han sido creados para adaptar el desarrollo de los robots a las necesidades del programador.

ROS, siglas en inglés de Robot Operating System, es el framework que se utilizará y analizará en este proyecto. ROS fue creado en 2007 por la empresa Willow Garage, con la intención de crear un framework que permita la integración de la robótica en una gran variedad de situaciones. Se puede entender ROS como una gran colección de herramientas, librerías y normas que tienen el objetivo de simplificar la tarea de crear diferentes comportamientos robóticos en una gran variedad de plataformas hardware y software. Además, el hecho de que ROS haya sido creado totalmente bajo la licencia de código abierto BSD (*Berkeley Software Distribution*) ha ayudado a que actualmente ROS sea uno de los frameworks más usados en la comunidad robótica.

2.3.2. Características principales

Las principales características de ROS son:

Sistema distribuido: la funcionalidad de ROS está basado en un sistema distribuido, en el que diferentes procesos, que pueden estar localizados en diferentes máquinas o hosts, se comunican entre sí utilizando una topología punto a punto. Con esta topología se utiliza un canal nuevo para cada comunicación entre dos procesos distintos, y se evita el tener que usar un servidor central para comunicar todos los procesos. ROS ha sido diseñado para funcionar en diferentes tipos de redes, como redes que usen comunicación Ethernet o Wireless al mismo tiempo. Este tipo de redes con una comunicación heterogénea no funcionaría bien con una topología basada en un servidor central, debido al cuello de botella originado por la conexión Wireless. En estas situaciones es en las que el sistema distribuido de ROS funcionará mejor.

Multilenguaje: ROS está programado utilizando principalmente los lenguajes de programación Python y C++. ROS ofrece una extensa API para poder desarrollar en este framework en estos dos lenguajes, aunque también soporta otros lenguajes, como Java, Octave o LISP, entre otros, gracias en parte a la colaboración de los miembros de la comunidad para poder incluir estos lenguajes.

Basado en herramientas: Con el objetivo de controlar la complejidad de ROS, los diseñadores optaron por un diseño de kernel (núcleo), donde se proporciona una gran cantidad de herramientas que acceden a este núcleo para construir y ejecutar los diversos componentes de ROS. Estas herramientas pueden realizar diferentes tareas, desde navegar por los ficheros, obtener y modificar los parámetros de configuración, visualizar la topología de los procesos en ejecución, o realizar la comunicación entre estos procesos.

Gratuito y Open-Source: todo el código fuente de ROS está disponible de manera pública. Con esto se intenta facilitar la depuración de todos los niveles del software que lo componen. ROS está distribuido bajo la licencia BSD, que permite el uso del código tanto de forma comercial como no comercial.

2.3.3. Arquitectura

La arquitectura de ROS está fundamentalmente basada en los siguientes componentes: nodos, mensajes, topics, servicios, parámetros y paquetes.

Un **nodo** es un proceso que realiza un cálculo o cómputo. Los nodos pueden combinarse entre sí utilizando topics, servicios y el servidor de parámetros. Estos nodos están diseñados para operar a una escala mínima, donde un sistema robótico normalmente estará compuesto por varios nodos. Por ejemplo, un nodo se ocupará de controlar sensor LIDAR, otro nodo controlará los motores de las ruedas del robot, otro calculará la localización del robot, etc... El uso de nodos proporciona diferentes ventajas a la arquitectura de ROS, como mayor facilidad para encontrar fallos que estarán aislados en los diferentes nodos, o la reducción de la complejidad del código al no tener que programar el sistema entero en un mismo proceso.

Los nodos se comunican entre sí publicando **mensajes** en los topics. Un mensaje es una estructura de datos, compuesta por campos tipificados. Estos campos pueden ser tipos simples (int, float, boolean…), arrays o estructuras más complejas.

Un **topic** es un bus de datos sobre el que los nodos intercambian mensajes. Estos topics utilizan una semántica de Publisher/Subscriber de forma anónima. En general, los nodos no saben con qué otro nodo se están comunicando. En vez de eso, los nodos que quieren acceder a unos datos concretos se suscriben al topic relevante, y los nodos que quieren generar datos publican en el topic relevante. Puede haber varios publishers o subscribers para un mismo topic.

Los topics están pensados para una comunicación en serie y unidireccional. Si se quiere realizar una llamada procedural a otro nodo, como por ejemplo enviar una petición y recibir una respuesta, deberían utilizarse los **servicios**. Los servicios se definen mediante dos mensajes, un mensaje de petición y otro de respuesta y se basan en un funcionamiento de cliente y servidor. Un nodo de ROS que funcionará

como servidor es el que ofrecerá el servicio, y otro nodo, el nodo cliente, llamará al servicio enviando un mensaje de petición y esperando la respuesta.

La última forma que tienen los nodos para comunicarse entre sí es mediante el uso del **servidor de parámetros**. Este servidor de parámetros es un diccionario nombre-valor compartido que puede ser accedido en toda la red de ROS. Los nodos pueden utilizar este servidor para guardar u obtener parámetros en tiempo de ejecución. Está pensado para guardar datos estáticos y no demasiado complejos, por ejemplo parámetros de configuración.

El último elemento importante de la arquitectura de ROS son los **paquetes**. El software en ROS está organizado en paquetes. Un paquete puede contener nodos, librerías independientes a ROS, ficheros de configuración, software externo a ROS, o cualquier cosa útil para el funcionamiento del módulo software. El objetivo de estos paquetes es proporcionar la funcionalidad de módulos software de una manera sencilla para que el software pueda ser reusado fácilmente.

2.3.4. Herramientas de bajo nivel

ROS ofrece un conjunto de herramientas para poder trabajar con su arquitectura a bajo nivel. Estas herramientas nos permitirán utilizar los nodos, mensajes, topics, servicios, parámetros y paquetes mencionados anteriormente. Estas herramientas son:

roscore: Es el comando que lanzará los nodos y programas que son prerrequisitos de un sistema ROS. Concretamente, el comando roscore lanzará el ROS master, que ofrece un servicio de nombres y registro para todos los nodos, topics, servicios y parámetros. Roscore también lanzará el servidor de parámetros y el nodo *rosout*, que será el nodo que se encargue de la monitorización del sistema.

rosrun: permite ejecutar un nodo ejecutable de un paquete arbitrario sin tener que estar en la ruta de ese paquete. La forma de usarlo es con la siguiente sintaxis:

rosrun <package> <executable>

roslaunch: es una herramienta que permite lanzar múltiples nodos de manera simultánea con un solo comando. También permite asignar valores a los parámetros del servidor de parámetros, o relanzar un nodo automáticamente si este se cierra de forma inesperada. Roslaunch utiliza ficheros con extensión .launch, que están definidos con el formato XML. La sintaxis de uso de roslaunch es la siguiente:

roslaunch package_name file.launch

rospack: es la herramienta que se encarga de administrar los paquetes de ROS. Se basa en la herramienta de Linux *dpkg*. Con rospack podemos usar los siguientes comandos:

rospack find <package>	Devuelve la ruta absoluta del paquete.
rospack depends <package>	Devuelve una lista de las dependencias del paquete.
rospack depends-on <package>	Devuelve una lista de los paquetes que dependen del paquete determinado.

rosnode: es la herramienta usada para obtener información sobre los nodos ROS, incluyendo información sobre sus publicaciones, subscripciones y conexiones. Podemos usar los siguientes comandos:

rosnode info <node>	Devuelve información sobre el nodo.
rosnode kill <node>	Mata un nodo activo.
rosnode list	Muestra una lista con los nodos activos.
rosnode machine <machine-name>	Muestra una lista con los nodos activos de una máquina concreta.
rosnode ping <node>	Realiza un test de conexión al nodo especificado.

rostopic: es la herramienta que se usa para obtener información sobre los topics de ROS, como los publishers y los subscribers de cada topic, la frecuencia a la que se actualiza un topic o el tipo de mensajes que utiliza un topic. Los comandos que se pueden usar son los siguientes:

rostopic bw <topic>	Muestra el ancho de banda usado por el topic.
rostopic echo <topic>	Muestra los mensajes que se están compartiendo por el topic.
rostopic find <msg-type>	Muestra los topics que están utilizando el tipo de mensaje especificado.
rostopic hz <topic>	Muestra la frecuencia de publicación de un topic.
rostopic info <topic>	Muestra información sobre el topic.
rostopic list	Muestra una lista de todos los topics activos.
rostopic pub <topic> <type> [data]	Publica datos en el topic especificado.
rostopic type <topic>	Muestra el tipo de mensajes del topic.

rosmsg: es la herramienta usada para mostrar información sobre los mensajes de ROS. Se puede usar con los siguientes comandos:

rosmsg show <msg-type>	Muestra los campos del tipo de mensaje.
rosmsg list	Muestra una lista con todos los mensajes.
rosmsg package <package>	Muestra todos los mensajes de un paquete.
rosmsg packages	Muestra una lista con todos los paquetes que contienen algún mensaje.
rosmsg md5 <msg-type>	Muestra la suma md5 de un mensaje.

rosservice: se usa para mostrar y hacer peticiones a los servicios ROS. Se puede usar con los siguientes comandos:

rosservice args <srv>	Muestra los argumentos requeridos por el servicio.
rosservice call <srv> [args]	Realiza una llamada al servicio con los argumentos especificados.
rosservice find <srv type>	Muestra los servicios de un tipo determinado.
rosservice list	Muestra una lista con los servicios disponibles.
rosservice info <srv>	Muestra información sobre el servicio.
rosservice node <srv>	Muestra el nombre del nodo que ofrece el servicio especificado.
rosservice type <srv>	Muestra el tipo del servicio.
rosservice uri <srv>	Muestra la dirección URI del servicio.

rosparam: se usa para acceder al servidor de parámetros de ROS. Se puede usar con los siguientes comandos:

rosparam list	Muestra los nombres de los parámetros activos.
rosparam get <param>	Obtiene el valor del parámetro.
rosparam set <param> [value]	Guarda el valor en el parámetro especificado.
rosparam delete <param>	Borra el parámetro especificado.
rosparam dump <file>	Guarda el contenido del servidor de parámetros a un fichero usando el formato YAML.
rosparam load <file>	Carga el contenido de un fichero YAML al servidor de parámetros.

2.3.5. Transformadas (tf)

Un sistema robótico normalmente tiene varios marcos de referencia 3D que pueden cambiar con el paso del tiempo, por ejemplo la referencia al mundo, la referencia de la base del robot, de la cabeza o de cualquiera de sus elementos. Para mantener información de todos estos marcos de referencia a lo largo del tiempo, ROS utiliza el paquete **tf** (transform). Con este paquete se puede mantener la relación entre diferentes marcos de referencia usando una estructura en forma de árbol, y que está almacenada a lo largo del tiempo. Las herramientas de tf permiten al usuario obtener transformadas (translación y rotación) entre dos marcos de referencia distintos en el tiempo que se requiera. tf funciona como un sistema distribuido, por lo que no hay un servidor central de transformadas, y toda la información de las transformadas de un robot están disponibles en todo el sistema

ROS. En la Ilustración 33 se muestra un ejemplo de cómo quedarían representadas las transformadas en un robot.

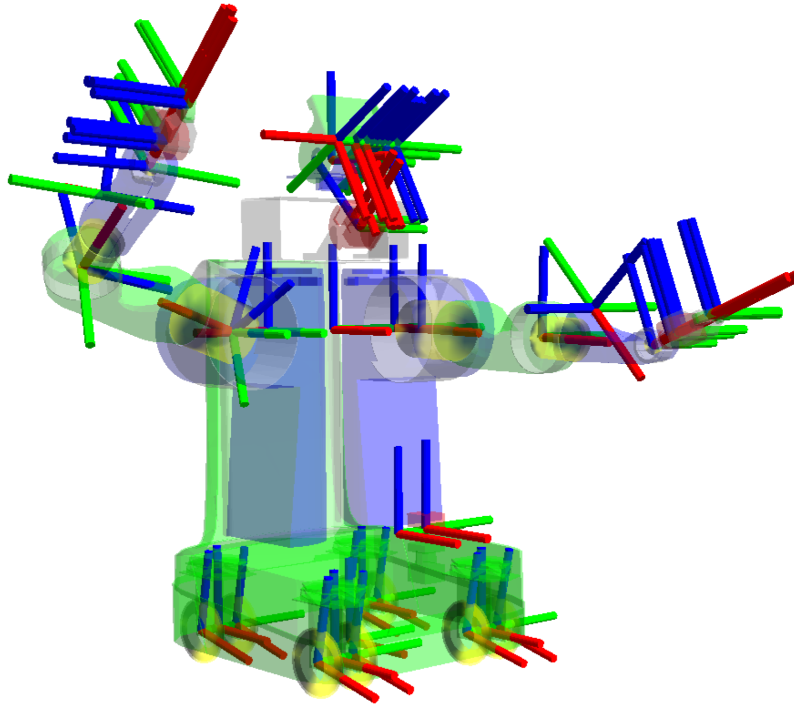

Ilustración 33 Ejemplo de representación de las transformadas en el robot PR2.

Algunas de las herramientas que ofrece el paquete de transformadas son:

tf_monitor	Muestra información del sistema de transformadas actual por la consola.
tf_echo <source_frame> <target_frame>	Muestra información sobre la transformada que existe entre el frame source y el frame target.
static_transform_publisher	Es un nodo que permite publicar transformadas
view_frames	Es un nodo que crea un grafo en PDF del árbol de coordenadas actual.

2.3.6. Robot description lenguaje (URDF)

El paquete **URDF** contiene un analizador sintáctico en C++ para los ficheros con formato URDF (Unified Robot Description Format), que es un formato basado en el lenguaje XML que se usa para representar el modelo de un robot. URDF es el modelo de descripción de robots que se usa en todo ROS, aunque este paquete también intenta ofrecer conversiones y compatibilidad con otros formatos de modelado de robots, como el formato SDF (Simulation Description Format, usado por el simulador Gazebo) o el formato COLLADA, que es un formato más genérico para definir modelos en 3D. La relación de los distintos formatos de modelado de robots en ROS se muestra en la Ilustración 34.

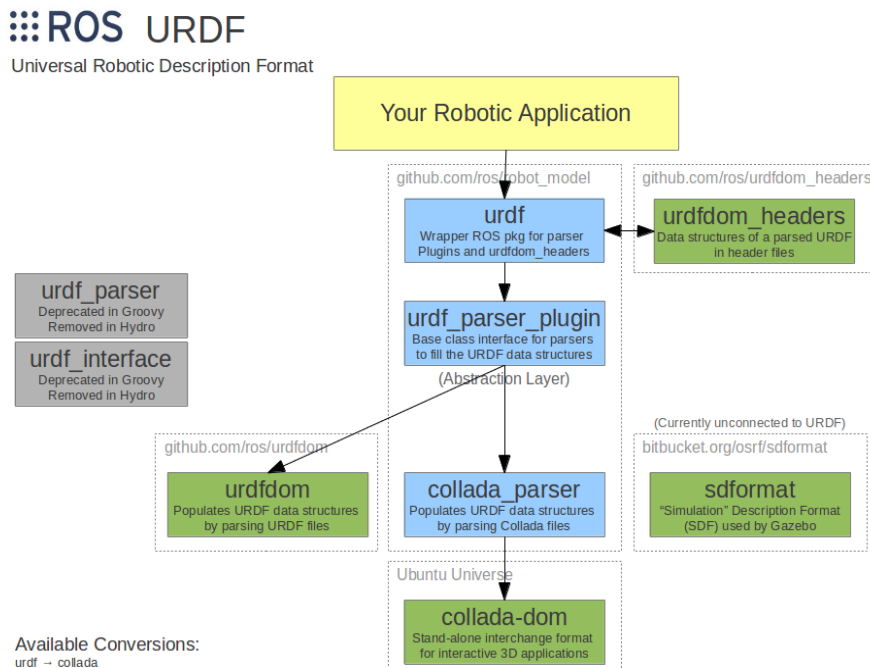

Ilustración 34 Relación entre los diferentes formatos de modelado de robots en ROS.

El formato URDF se usa para definir robots mediante una estructura en forma de árbol, donde los elementos del robot deben ser estructuras rígidas conectadas por articulaciones. URDF define la descripción cinemática y dinámica del robot, la representación visual del robot y el modelo de colisiones del robot. La descripción más típica de un robot en formato URDF consiste en un conjunto de *links*, que definen un cuerpo rígido con inercia y ciertas características visuales, y un conjunto de *joints*, que conectan los links entre sí, y definen las características cinemáticas y dinámicas de la articulación además de limitar los posibles giros y movimientos. En la Ilustración 35 Ejemplo de configuración en formato URDF usando los elementos link y joint. se muestra un ejemplo de una posible configuración de estos links y joints.

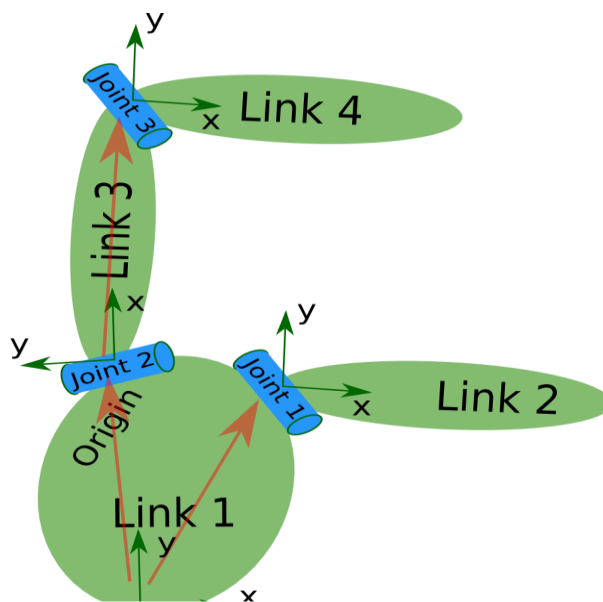

Ilustración 35 Ejemplo de configuración en formato URDF usando los elementos link y joint.

2.3.7. Herramientas de visualización y simulación

A la hora de desarrollar y depurar el software de los robots se hace necesario obtener información sobre el estado del sistema en tiempo de ejecución. ROS ofrece diferentes herramientas para poder visualizar la información que se transmite entre los nodos mediante los topics. La primera y la más básica de estas herramientas es utilizar la herramienta **rostopic**, con el que podemos obtener la información de los topics directamente en la línea de comandos de la terminal, aunque muchas veces mostrar esta información sobre la línea de comandos no es suficiente para depurar el código correctamente. Para ello ROS ofrece el programa **rviz** (Ilustración 36), que es una herramienta de más alto nivel que nos permite visualizar la información de los topics en un entorno 3D. Mediante rviz podemos mostrar por pantalla una gran cantidad de tipos de datos diferentes, como imágenes, nubes de puntos, primitivas geométricas, mapas, posiciones del robot, modelos renderizados del robot, transformadas, etc.

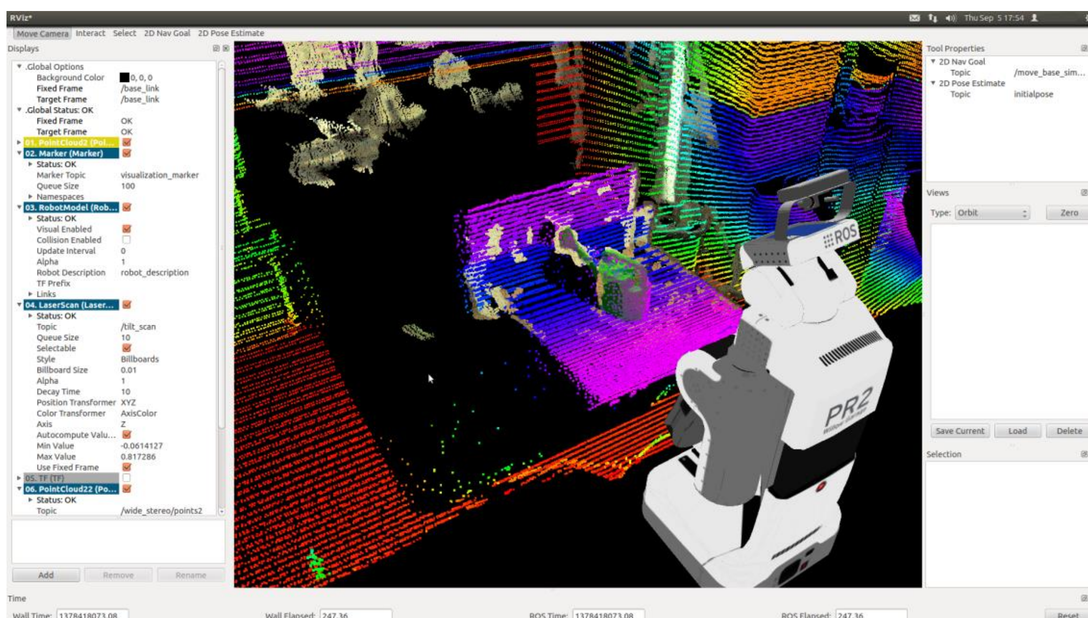

Ilustración 36 Visualización en rviz del modelo del robot pr2 y de la nube de puntos de su escáner 3D

La **simulación** es una parte crítica del desarrollo de cualquier máquina. Para el desarrollo de robots, esta simulación permite probar diferentes comportamientos y algoritmos sin utilizar directamente el robot físico. Una buena herramienta de simulación debería ser capaz de recrear el mundo real con una gran precisión, proporcionando mínimas diferencias entre el robot real y el simulado. Si la

simulación ha sido creada correctamente, estos diferentes comportamientos y algoritmos pueden ser transportados al robot real sin tener que realizar mayores modificaciones.

Una de las herramientas de simulación más utilizadas en ROS es **Gazebo** (Ilustración 37). Gazebo fue inicialmente creado en la University of Southern California [8], pero actualmente está siendo desarrollado por la OSRF (Open Source Robotics Foundation) [9] y está apoyada por una comunidad de usuarios muy activa.

Gazebo utiliza diferentes motores físicos (ODE [10], Bullet [11], Simbody [12] y DART [13]) para realizar una simulación dinámica. Estos motores físicos permiten simular la dinámica de objetos rígidos y articulados, por ejemplo robots móviles con ruedas, brazos o piernas, y proporcionan detección de colisiones. Gazebo ofrece una gran variedad de sensores para poder percibir el entorno simulado, como sensores LIDAR, cámaras 2D y 3D, sensores como la Microsoft Kinect, sensores de contacto y sensores de fuerza y par de torsión. A estos sensores se les puede aplicar un error aleatorio, para simular mejor el comportamiento del sensor en el mundo real.

Para mostrar al usuario este entorno de simulación, Gazebo utiliza el motor gráfico **OGRE** [14]. Con este software, Gazebo ofrece una reproducción realista del entorno a simular, usando iluminación de alta calidad, sombras y diferentes texturas. Los modelos son especificados usando el estándar SDF (Simulation Description Format), que utiliza el formato de etiquetas XML para crear los objetos de la escena, la física de la simulación y los robots. Gazebo también ofrece compatibilidad con el estándar URDF, que actualmente es de los más utilizados en el entorno ROS.

Usando SDF o URDF, Gazebo también ofrece una interfaz para crear diferentes plugins, que permiten especificar el comportamiento del robot ante diferentes situaciones, o especificar las características del control de movimiento del robot. Actualmente hay una extensa comunidad de usuarios que ofrece plugins y modelos para la mayoría de los robots en el mercado.

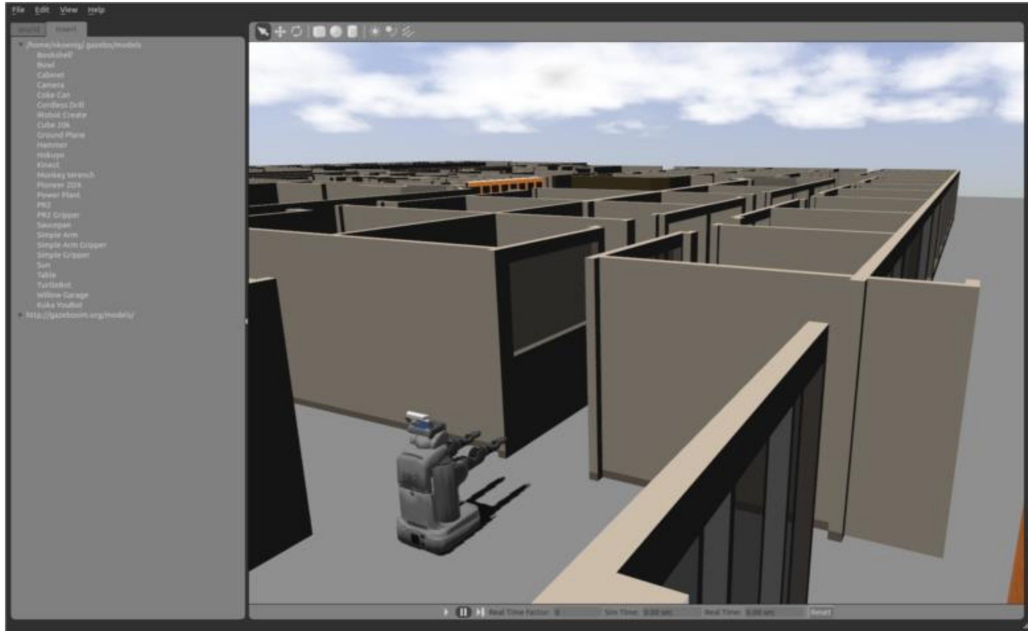

Ilustración 37 Representación del robot PR2 y de la oficina de Willow Garage en el simulador Gazebo.

2.4. OpenNI

El término Natural Interaction (NI) hace referencia a la comunicación humano-máquina basada en los sentidos humanos, principalmente la visión y el oído. Los paradigmas de esta interacción natural pueden dejar obsoletas comunicaciones clásicas como las que ofrecen periféricos como el teclado, el ratón o controles remotos. Algunos ejemplos de esta interacción natural incluyen reconocimiento del habla, donde los dispositivos pueden recibir instrucciones vía comandos verbales, reconocimiento de los movimientos de la mano, donde ciertos gestos y movimientos pueden ser reconocidos e interpretados para activar y controlar distintos dispositivos, o reconocimiento de la posición del cuerpo, donde se puede conocer la posición de todas las articulaciones del cuerpo para enviar instrucciones al dispositivo o para monitorizar el estado del usuario.

OpenNI (Open Natural Interaction) es un framework multilenguaje que ofrece una API para crear aplicaciones que utilicen Natural Interaction. El objetivo principal de OpenNI es crear una interfaz de programación estándar que permita la comunicación entre los sensores de audio o vídeo, y el middleware que analizará la información de estos sensores y la interpretará. OpenNI ofrece una API para los sensores de audio/vídeo, y otra API diferente para el middleware software que analizará los datos. Esta separación de interfaces de programación permite la independencia del hardware respecto del software, facilitando así la tarea de programación y portabilidad, pudiendo programar aplicaciones OpenNI utilizando la información sin procesar del sensor, independientemente del tipo de sensor.

En la Ilustración 38 se ve un esquema de la arquitectura OpenNI, donde la parte superior representa las aplicaciones que implementan algún tipo de comunicación mediante interacción natural, la capa intermedia representa las interfaces de comunicación de OpenNI, y la capa inferior representa a los dispositivos hardware de audio y vídeo.

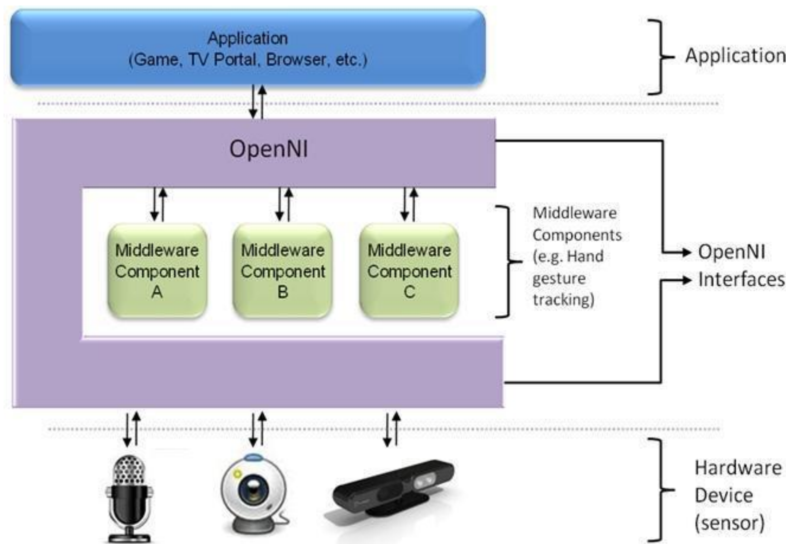

Ilustración 38 Arquitectura de la librería OpenNI

A continuación se detallan algunos conceptos sobre la programación de aplicaciones basadas en OpenNI:

Módulos: La API de OpenNI ofrece una interfaz para dispositivos físicos y para componentes software. Estos dos elementos se definen como módulos, y se usan para producir y procesar los datos del sensor. Los módulos de sensores que permite la librería OpenNI son los siguientes:

- Sensor 3D
- Cámara RGB
- Cámara Infrarroja
- Dispositivo de audio: tanto un único micrófono como una matriz de micrófonos.

Los elementos software que también pueden actuar como módulos son los siguientes:

- Software de análisis del cuerpo: es el componente software que procesa los datos del sensor y genera la información necesaria del

usuario, generalmente información de la posición y orientación de las articulaciones, o el centro de masas del usuario.

- Software de detección de la posición de la mano: analiza los datos del sensor para conseguir la localización de un punto de la mano.
- Software de detección de gestos con la mano: este software permite identificar algunos gestos predefinidos, como por ejemplo identificar un saludo con una mano.
- Software de análisis de la escena: es el software que se encarga de analizar la imagen de la escena para producir información como la separación de elementos de la escena respecto del fondo, las coordenadas del plano del suelo, o la correcta identificación de los elementos de la escena.

Nodos de producción: son los componentes que se ocupan del proceso de crear la información requerida por las aplicaciones OpenNI. Estos nodos utilizan la implementación de los módulos mencionados anteriormente para generar la información especificada. Estos nodos de producción son los elementos fundamentales que la interfaz OpenNI ofrece para generar aplicaciones. Pueden funcionar de forma anidada, donde un nodo de producción genera una información de bajo nivel, y esta información es usada por otro nodo de producción para generar información de más alto nivel. Por ejemplo, para obtener información sobre el movimiento de una figura humana de una escena 3D, primero un nodo generador de profundidad produce el mapa de profundidad, y se lo pasa a otro nodo productor, que obtiene este mapa de profundidad y genera la información del cuerpo de los usuarios. Los distintos nodos de producción que utiliza OpenNI son los siguientes:

Nodos de producción relacionados con los sensores:

- Dispositivo: nodo que representa un dispositivo físico. Se usa principalmente para la configuración del dispositivo.

- Generador de profundidad: genera el mapa de profundidad obtenido de un sensor 3D.
- Generador de imagen: genera imágenes RGB.
- Generador de infrarrojos: genera imágenes infrarrojas.
- Generador de audio: se encarga de generar el audio del micrófono.

Nodos de producción relacionados con el middleware:

- Generador de alertas: genera alertas a la aplicación cuando se detectan gestos predefinidos.
- Analizador de la escena: genera una imagen de profundidad con etiquetas, donde cada pixel tiene una etiqueta en la que se define si el pixel es parte de una figura, o si es parte del fondo de la escena.
- Generador de puntos de la mano: este nodo genera alertas cuando se detecta una mano en la escena, y ofrece información sobre los movimientos de la mano.
- Generador de usuarios: genera una representación de los cuerpos humanos que aparecen en la escena, ya sea que aparezcan de manera parcial o completa.

3. Desarrollo práctico

En esta sección, en primer lugar se trataran en detalle los materiales utilizados durante el desarrollo del proyecto, y a continuación se explicarán los pasos realizados para cada una de los objetivos a conseguir.

3.1. TurtleBot

El presente proyecto se desarrollará principalmente usando el modelo del robot TurtleBot [15]. En las siguientes secciones se realizará una presentación del robot y a continuación se detallarán sus características. Finalmente se hablará sobre los sensores que usa TurtleBot para obtener información del entorno, principalmente el sensor Microsoft Kinect [16].

3.1.1. Presentación del robot

TurtleBot (Ilustración 39) es una plataforma móvil de bajo coste desarrollada por Willow Garage, teniendo como principal objetivo la educación y la investigación. Está programada con código abierto usando las posibilidades del entorno de trabajo para robots ROS [17], y tiene una gran comunidad de usuarios activa que ofrecen una gran variedad de paquetes de software para realizar una gran variedad de tareas [18]. Aun siendo un robot móvil de bajo coste, este permite una gran variedad de tareas a realizar, por ejemplo: explorar una casa de manera autónoma, construir imágenes en 3 dimensiones, tomar imágenes panorámicas o actuar de robot "camarero" trayéndote objetos que puedas necesitar. Usando las posibilidades de ROS se puede adaptar este robot a un sinfín de posibilidades, siendo esta versatilidad a la hora de programar tareas una de las características clave a la hora de elegir el TurtleBot para la realización de este proyecto. La plataforma que se ha usado para unir los componentes del TurtleBot es un poco distinta a la mostrada en la imagen, pero los componentes son los mismos.

Ilustración 39 TurtleBot

TurtleBot 1 es la primera versión de este robot y la que se usará durante este proyecto. Los 3 componentes básicos que forman el conjunto del TurtleBot son:

- Base móvil iRobot Create
- Sensor Microsoft Kinect
- Portátil o Netbook

La base **iRobot Create** (Ilustración 40), creada en 2007 por iRobot [20], es una plataforma móvil de bajo coste pensada para estudiantes, educadores y desarrolladores. Está basada en otro robot de la misma compañía, el iRobot Roomba, pero sustituye los elementos de limpieza y aspiración por una serie de conectores que permiten acoplar diferentes elementos. Por ejemplo, usando estas conexiones podemos adaptar diferentes dispositivos al Create, como sensores, brazos robóticos, conexiones Wireless u otros dispositivos (en el caso de este proyecto, la base Create permitirá acoplar fácilmente un soporte que contendrá el Netbook y el sensor Microsoft Kinect). También ofrece una gran flexibilidad a la hora de programar los diferentes elementos software, gracias a su compatibilidad con el framework ROS, lo que le permitirá conseguir todos los objetivos de este proyecto (moverse libremente o seguir a una persona, mediante identificación visual o comandos de voz) [21]

Ilustración 40 iRobot Create

El sensor principal que se usará durante el proyecto para permitir la interacción del robot con el mundo real será el sensor **Microsoft Kinect** (Ilustración 41). Kinect fue lanzada por Microsoft en 2010 como complemento para su consola de videojuegos Xbox 360. Kinect permite a los usuarios de Xbox 360 interactuar con la consola de una manera diferente al mando clásico, permitiendo controlar a los personajes o avatares de la consola mediante gestos o comandos de voz. En 2012, Microsoft lanzó la compatibilidad de Kinect con Windows 7 y Windows 8, y pronto aparecieron diferentes drivers para otros sistemas operativos como Ubuntu

La cámara Kinect permite calcular distancias mediante un emisor y un receptor de infrarrojos, usando la técnica de la luz estructurada, y ofrece una visión clara mediante una cámara RGB. También dispone de 4 micrófonos para captar diferentes comandos de voz. El bajo coste de esta cámara y sus posibilidades de detección de personas y gestos hacen a Kinect como la mejor opción para cumplir los objetivos de este proyecto.

Ilustración 41 Microsoft Kinect

Para realizar todos los cálculos, algoritmos, tareas y comprobaciones usaremos un Netbook que irá situado en un soporte encima de la base iRobot Create. Encima de esta base irá situado el sensor Microsoft Kinect. El montaje final del robot se muestra en la Ilustración 42. El Netbook usado cuenta con las siguientes especificaciones:

- Procesador Intel Celeron 887@1,50 GHz x2
- Memoria: 1,6 GB DDR
- SO: Ubuntu 12.04 LTS 64bits

Ilustración 42 Montaje final del robot

3.1.2. Características

La configuración de las ruedas de la base móvil iRobot Create está pensada para ofrecer la mayor maniobrabilidad posible. Dispone de 2 ruedas con una configuración diferencial, y una tercera rueda giratoria que actúa como punto de apoyo para ofrecer una buena estabilidad (Ilustración 43). Esta configuración diferencial permite al robot girar sobre sí mismo, simplemente girando las ruedas en diferentes sentidos a la misma velocidad. Esta capacidad de maniobrabilidad es un

punto importante para cumplir los objetivos del proyecto, ya que le permitirá al robot realizar giros cerrados en espacios reducidos. Por ejemplo, con una configuración tipo Ackermann (la configuración de dos ruedas fijas y 2 ruedas móviles que usan los coches) no se podrían realizar estos giros tan pronunciados. [22]

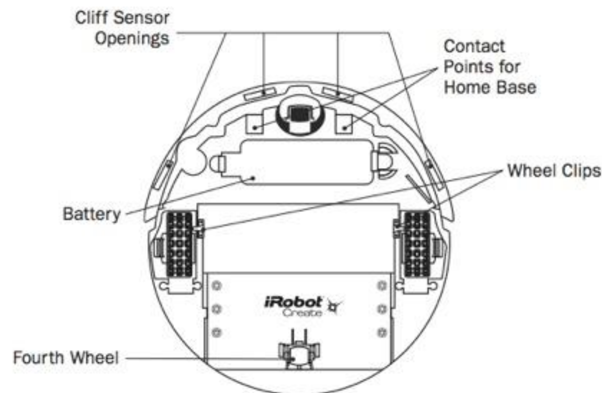

Ilustración 43 Vista inferior del iRobot Create

La tercera rueda es una rueda móvil sin tracción, situada en la parte delantera inferior del robot, que ejerce la función de tercer soporte para la base, dándole la estabilidad necesaria al conjunto del robot. También es posible añadir una cuarta rueda en la parte trasera del iRobot Create, evitando así que el robot pueda inclinarse hacia detrás en caso de tener mucho peso en la parte trasera, o en una aceleración brusca, pero en el caso concreto de este robot no ha sido necesario la inclusión de esta cuarta rueda.

3.1.3. Sensores

La base iRobot Create dispone de una serie de sensores para obtener información del entorno: sensores de contacto y sensores de detección de precipicios. Pero para este proyecto se ha usado exclusivamente los sensores que proporciona la cámara Microsoft Kinect, que ofrecen una información de más alto nivel que los simples sensores de contacto delanteros.

Microsoft Kinect

La cámara Kinect está compuesta por los siguientes sensores:

- Proyector y sensor infrarrojos.
- Cámara VGA
- Conjunto de 4 micrófonos

También dispone de un pequeño motor que permite a la Kinect variar el ángulo de inclinación respecto a la posición de la base. La posición de los sensores y de este motor se muestra en la Ilustración 44.

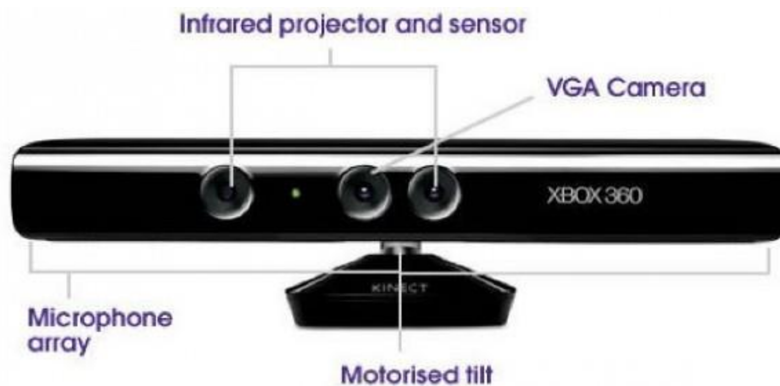

Ilustración 44 Posición de los sensores de la cámara Kinect

La cámara Kinect es capaz de detectar la distancia a los objetos usando la técnica de la luz estructurada. [23]. El principio básico de la luz estructurada consiste en proyectar un patrón conocido sobre la escena a analizar y a continuación inferir la profundidad a partir de la deformación de ese patrón en la escena [24]. El patrón que dibuja sobre la escena el sensor infrarrojo es un patrón complejo de puntos (Ilustración 45 e Ilustración 46). Hay más información sobre cómo se construye este patrón de puntos en [25]

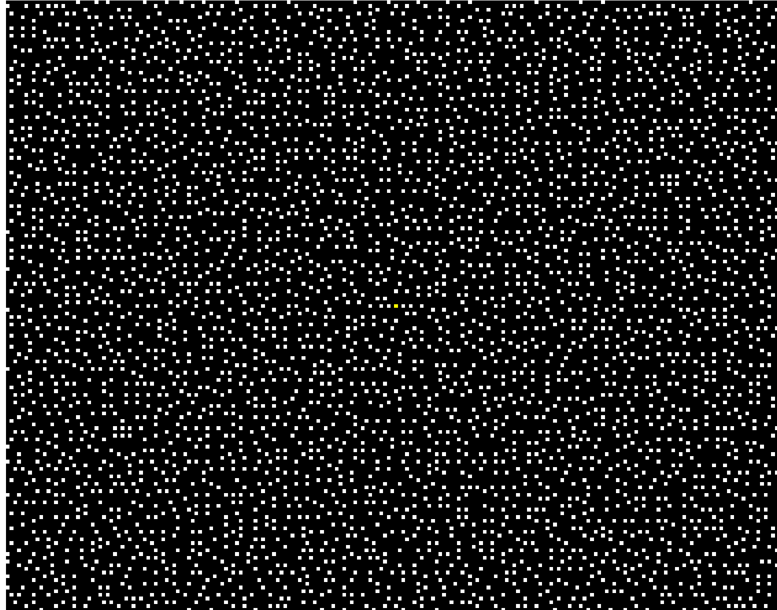

Ilustración 45 Patrón de luz estructurada proyectado por la cámara Kinect

Ilustración 46 Patrón proyectado sobre la escena

Una vez el patrón de puntos ha sido proyectado en la escena, y ha sido recibidos por el sensor infrarrojo, Kinect combina esta luz estructurada con dos técnicas clásicas de la visión por computador para obtener la distancia a cada uno de estos puntos: **la profundidad desde el enfoque**, y la **estereovisión**.

La técnica de la profundidad desde el enfoque se basa en el principio de que los objetos más alejados aparecen más borrosos. Kinect mejora la precisión de este principio clásico utilizando una lente especial para este fin. Esta lente es una lente astigmática, que tiene una distancia focal diferente para el eje vertical y el eje

horizontal. De esta forma, los puntos del patrón de luces, que en la escena se ven como círculos, se proyectan en la cámara como elipses, y sus orientaciones dependen de la distancia de la cámara a los puntos (Ilustración 47) De esta forma es posible calcular la distancia de los objetos de la escena.

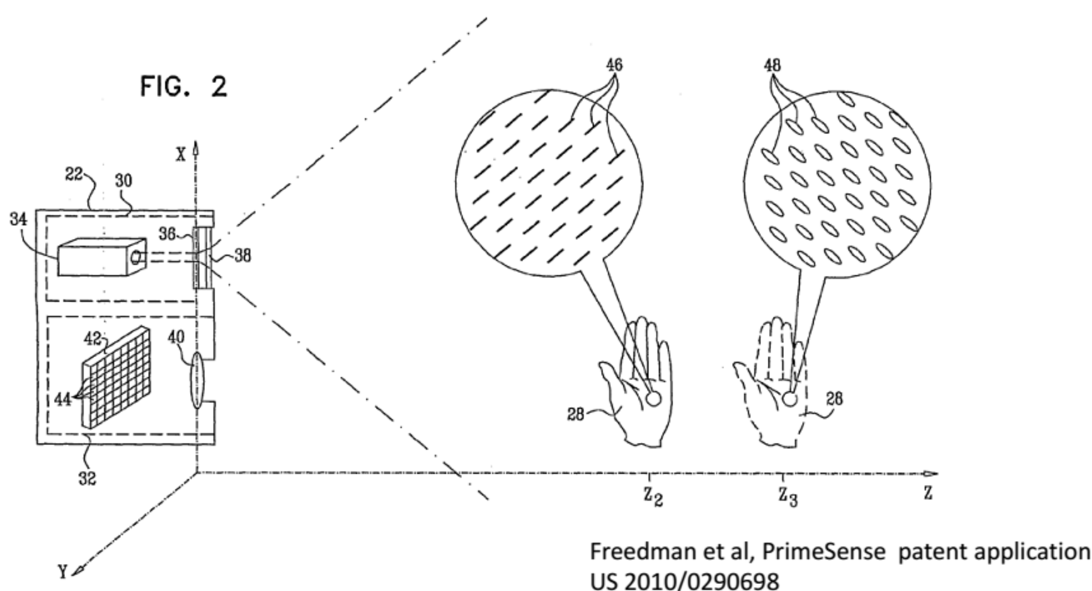

Ilustración 47 Percepción de los objetos de la escena a través de la lente astigmática.

Además de la técnica de la profundidad a partir del enfoque, Kinect utiliza también la técnica de la estereovisión como complemento para calcular distancias a los objetos. Esta técnica se basa en el proceso visual llamado estereopsis, por el cual a partir de dos imágenes ligeramente diferentes obtenidas por cada ojo, el cerebro es capaz de recomponer una imagen tridimensional [26]. En el caso de la Kinect, estas 2 imágenes ligeramente diferentes se consiguen a partir de la luz estructurada, donde la primera imagen corresponde al patrón de puntos proyectado, y la segunda imagen es el patrón de puntos recibido. Cada par de puntos de las imágenes tendrán una ligera diferencia de posición, también llamada disparidad, y esta diferencia de posición será inversamente proporcional a la distancia a la cámara.

La cámara Kinect calcula la disparidad de todos los puntos de luz estructurada y crea una imagen en la que se pueden observar las diferencias entre los puntos,

llamada mapa de disparidad (Ilustración 48) Se puede obtener la distancia exacta a cada punto con la siguiente fórmula obtenida mediante triangulación:

$$z = b\frac{f}{u_l - u_r}$$

Dónde:

- z es la distancia que queremos obtener
- b es la distancia entre el emisor y el receptor infrarrojos
- f es la distancia focal de la lente
- u_l - u_r es la disparidad de los dos puntos

Left image

Right image

Disparity map

Ilustración 48 Mapa de disparidad formado por las diferencias de los puntos de las dos imágenes. Cuanto más alejado esté un objeto, menos disparidad tendrá.

A partir de este mapa de disparidad, la cámara Microsoft Kinect es capaz de distinguir la posición de las personas en la imagen y crear un esqueleto virtual con las

articulaciones de la persona (Ilustración 49). Para obtener la posición de las personas en la imagen, se utiliza un algoritmo basado en un árbol de decisión aleatorio, que ha sido entrenado a partir de 1 millón de muestras de imágenes de personas, con esqueletos conocidos. Estos algoritmos están explicados en detalle en [27].

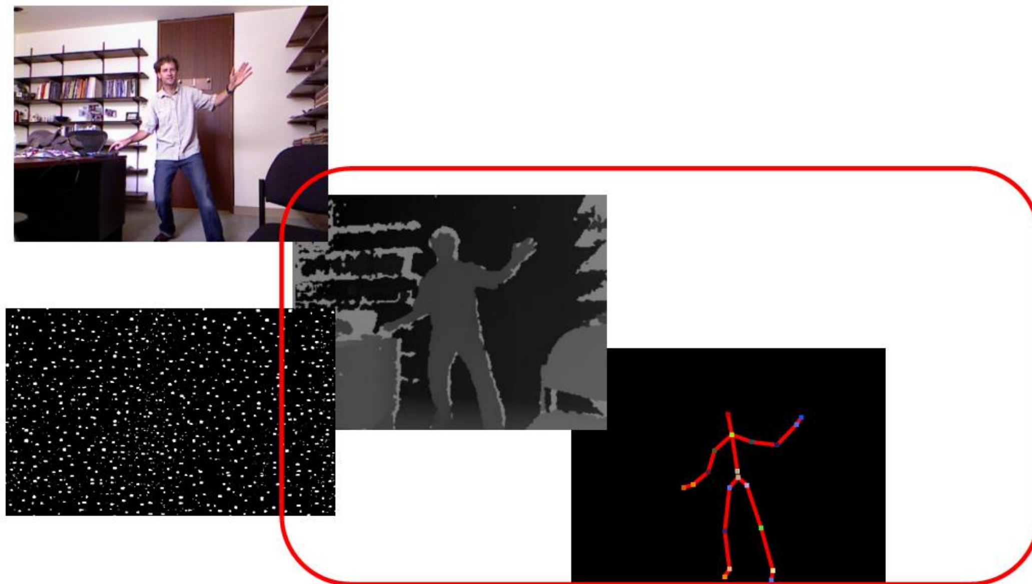

Ilustración 49 Extracción de la posición de una persona y de su esqueleto a partir de la nube de puntos y del mapa de disparidad.

Por último, el sensor Microsoft Kinect cuenta con una serie de 4 micrófonos incorporados en la parte inferior del sensor. Con estos 4 micrófonos se pueden realizar funciones de reconocimiento de comandos de voz, o incluso de detectar la el origen del sonido, triangulando la potencia con la que llega el sonido a cada micrófono.

Sensores del iRobot Create

La base iRobot Create cuenta con otros sensores, como 4 detectores de precipicios situados en la parte inferior del robot, 1 sensor de contacto en la parte delantera, y un receptor de infrarrojos en la parte superior. No se ha visto una

utilidad directa de estos sensores para los objetivos del proyecto, así que estos sensores no han sido utilizados.

3.2. Desarrollo del robot perseguidor

3.2.1. Descripción

El primer objetivo de este proyecto es conseguir que el robot (cuya estructura está definida en la sección 3.1) se mueva de manera autónoma manteniéndose a una distancia constante de la persona a la que se quiera seguir. Este objetivo se puede dividir en dos tareas básicas o subapartados. La primera tarea consiste en analizar la información obtenida por la cámara Kinect y procesar esta información para detectar posibles personas a seguir. La segunda tarea consistirá en realizar los movimientos necesarios en la base móvil para poder seguir los movimientos de la persona y mantenerse a una distancia constante y segura. Durante el desarrollo del software se ha mantenido esta división de tareas en diferentes paquetes de software, facilitando así la reusabilidad del código.

El análisis y procesamiento de la cámara Kinect se basa en los algoritmos desarrollados en la librería de OpenNI (Open Natural Interaction) que se han detallado en el apartado 2.6. Esta librería proporciona las herramientas y algoritmos necesarios para poder localizar a una persona de manera eficaz mediante el sensor Kinect. El trabajo realizado consistirá en adaptar y filtrar la información del sensor para obtener la posición de la persona a la que queremos seguir. Este paso presenta algunos problemas como que los algoritmos de detección de personas pueden tener un error de falso positivo (detectar una persona donde no debería haber ninguna) bastante alto en algunas situaciones. También surge el problema de que los algoritmos de OpenNI están originalmente pensados para funcionar en posiciones fijas, ya que su uso original es el de actuar como sensor de visión para una Xbox, y por tanto no funcionan de manera tan fiable al estar encima de un robot moviéndose constantemente. En los siguientes apartados se detallará como se ha realizado esta tarea y como se han solucionado los problemas destacados.

Una vez obtenida la información requerida del sensor Kinect, la siguiente tarea es aplicar esta información a la base móvil del robot para obtener un robot autónomo. Una manera típica de abordar el desarrollo de un robot autónomo es la trabajar con un mapa, ya sea generado a priori, o a posteriori de manera dinámica. De esta manera el robot puede posicionarse dentro de este mapa y conocer con cierta certeza los posibles obstáculos que rodean al robot. Esta manera de trabajo se ha descartado ya que trabajar con un mapa normalmente requiere el uso de algún sensor de rango activo, por ejemplo un láser 2D, para poder obtener conocimiento de manera fiable del entorno que rodea al robot. Estos láseres suelen ser caros y se salía del presupuesto del proyecto, y por ello se ha descartado la opción de trabajar con un mapa. Por tanto se ha optado por una solución más sencilla pero que cumple con los objetivos del proyecto. El robot recibirá la posición relativa del usuario que tiene que seguir respecto del robot, y este se moverá de manera inmediata en dirección al usuario. Una vez el robot se encuentre dentro del rango de distancia segura al usuario (entre 1 y 2 metros), el robot esperará hasta que el usuario decida moverse otra vez. El movimiento del robot deberá hacerse de manera segura, y con aceleraciones suaves que eviten movimientos bruscos en el sensor Kinect, debido al problema anteriormente comentado de la pérdida de prestaciones del sensor con el movimiento. Este movimiento autónomo se verá complementado con la inclusión de comandos de voz que se explicarán en el apartado 3.4.

3.2.2. Descripción del software utilizado

Paquetes de ROS

En este apartado se muestran en detalle los paquetes de ROS utilizados durante el desarrollo del robot perseguidor. La mayoría de estos paquetes están directamente relacionados con el uso y manejo del sensor Kinect y de la base móvil iRobot Create, con alguna excepción de algunos paquetes que ofrecen herramientas para facilitar la programación. Los paquetes de ROS utilizados para el sensor Microsoft Kinect son los siguientes:

openni_launch: proporciona el launcher de más alto nivel para controlar la cámara Kinect, y es el paquete que se usará cuando se quiera utilizar el sensor. El launcher proporcionado crea una interfaz que permite abstraerse del driver que se usará para comunicarse con el sensor, en este caso el driver del paquete rgbd_launch. Mediante esta interfaz también se pueden controlar diferentes parámetros de los drivers del sensor, por ejemplo indicar que salidas del sensor queremos recibir.

rgbd_launch: este paquete ejecutará el driver del sensor Kinect. Proporciona los launchers necesarios para establecer la conexión con un dispositivo RGB-D (RGB-Depth), en este caso el dispositivo Kinect. También carga una serie de nodelets que permiten convertir los datos sin procesar obtenidos de la Kinect (la imagen RGB, los datos infrarrojos y los datos de profundidad) en información procesada (imagen de profundidad, imagen de disparidad y nubes de puntos).

openni_camera: proporciona algunas funcionalidades de procesamiento RGB-D no cubiertas por rgbd_launch.

openni_tracker: este paquete hace uso de los algoritmos OpenNI de PrimeSense para obtener y publicar el estado y la posición de los usuarios que se detectan a través del dispositivo, así como su esqueleto. Una vez lanzado el nodo openni_tracker, este buscará automáticamente los usuarios que aparezcan por pantalla y les asignará un número identificador. Para poder publicar el esqueleto del usuario, primero deberá hacerse una calibración de la persona. Para ello, el usuario debe hacer la pose PSI (extender los brazos con la forma de la letra griega Ψ) y esperar unos segundos. A continuación, si se ha realizado la calibración correctamente, el nodo empezará a publicar la posición de las articulaciones del esqueleto del usuario. Este esqueleto será publicado como una serie de transformadas con el nombre de la parte del cuerpo correspondiente (head, neck, torso, etc…) (ver Ilustración 50) El código de este paquete se ha usado como referencia para el desarrollo del robot perseguidor y se explicará con más detalle más adelante.

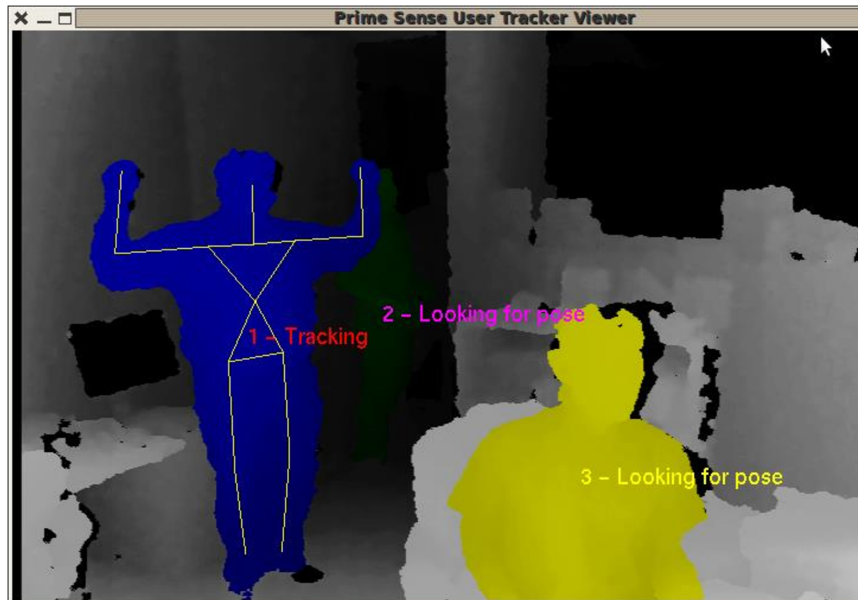

Ilustración 50 Uso del nodo openni_tracker donde se observa a un usuario realizando la pose PSI para calibrar y publicar el esqueleto.

Los paquetes de ROS utilizados para el correcto funcionamiento de la base iRobot Create son los siguientes:

turtlebot_bringup: es el paquete principal que se ocupará de cargar todos los nodos y los parámetros necesarios para preparar las funcionalidades del robot TurtleBot. Para ello proporciona el script ***turtlebot.launch***, que es el que se usará para inicializar el robot. Con este script se lanzarán la mayoría de paquetes y nodos que se describen a continuación.

create_node: es el driver que proporciona la interfaz entre ROS y la plataforma iRobot Create. Está basado en el driver del robot Roomba. Una vez el driver esté lanzado correctamente, este permitirá leer la información de los sensores de la plataforma móvil, y enviarle información de movimiento por el topic "/mobile_base/commands/velocity".

turtlebot_description: contiene el modelo completo en 3D del robot TurtleBot para simulación o visualización. No hará falta modificar directamente este

paquete, pero será usado si se quiere visualizar el estado del robot en el programa RViz, o si se quiere realizar una simulación con el programa Gazebo o VREP.

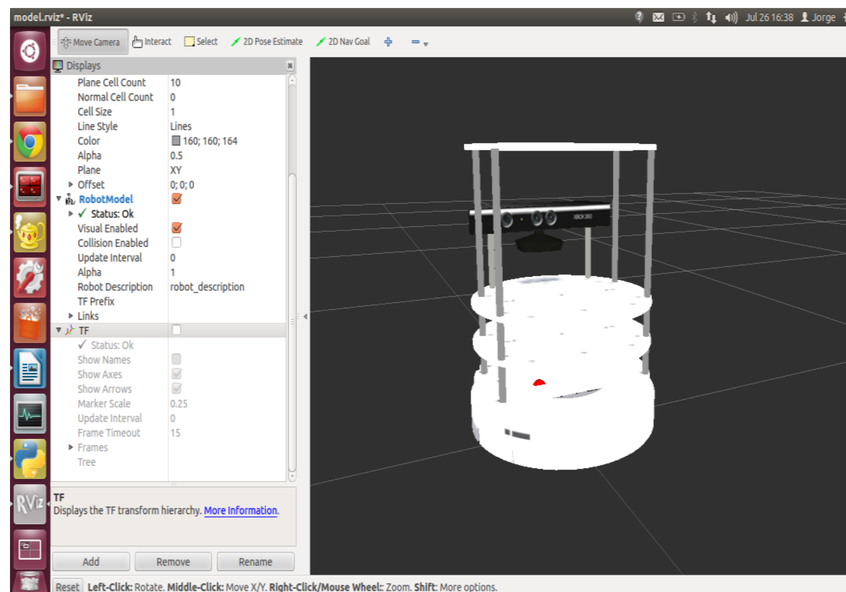

Ilustración 51 Modelo 3D del TurtleBot en el visualizador RViz

robot_state_publisher: es un paquete de ROS usado en muchos robots que permite publicar el estado y la posición del robot mediante transformadas. Para publicar este estado necesita leer el modelo 3D publicado por turtlebot_description.

diagnostics y diagnostic_aggregator: estos paquetes se usan para obtener información de los drivers hardware y del robot para poder realizar tareas de análisis y solución de problemas. La información obtenida será publicada por cada driver en el topic "/diagnostics".

Otros paquetes de ROS que han sido utilizados son:

geometry_msgs: incluye una descripción de los tipos de mensajes más usados. Por ejemplo el mensaje "geometry_msgs/Twist" que se utiliza para dar órdenes de movimiento a la base móvil.

3.2.3. Instalación del espacio de trabajo

En esta sección se detallan todos los pasos que se han seguido para instalar y configurar el software necesario que se ha usado durante el desarrollo del proyecto.

El primer paso es instalar y configurar los archivos y paquetes necesarios de **ROS Groovy**. Cuando se inició este proyecto ya existía la versión de ROS Hydro, pero llevaba poco tiempo desde que se lanzó y existían algunos problemas de compatibilidad con los drivers OpenNI de la Kinect, por eso se ha optado por la versión Groovy. ROS se ha instalado sobre el sistema operativo Ubuntu 12.04 64 bits, aunque la versión groovy de ROS también podría ser instalada sobre las versiones 11.10 y 12.10 de Ubuntu, además de en otras distribuciones de Linux o en Windows.

El primer paso de la instalación es agregar los repositorios al fichero *sources.list*, para que el sistema de paquetes Debian pueda aceptar el software de ROS. Para ello escribimos el siguiente comando en la terminal:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'

El siguiente paso es agregar las llaves de los repositorios que se van a bajar, de esta forma el programa apt (el programa de descarga de archivos en distribuciones de la familia Debian) sabe que los paquetes que va a descargarse son correctos y se asegura de que no han sido alterados. Para ello hay que ejecutar el siguiente comando en la terminal:

wget http://packages.ros.org/ros.key -O - | sudo apt-key add -

Y a continuación actualizamos la base de datos de paquetes Debian con el comando:

sudo apt-get update

Para descargar los paquetes ROS necesarios bajaremos el paquete ros-groovy-desktop-full. Este es un meta paquete que contiene los componentes principales de ROS, además de algunos componentes extras que se utilizan a menudo, como el paquete rqt, el rviz, el simulador gazebo o diversas librerías para el control de robots o uso de sensores. Escribimos el siguiente comando en la terminal:

sudo apt-get install ros-groovy-desktop-full

Esto tardará un buen rato ya que apt tendrá que descargar e instalar una cantidad considerable de paquetes. Una vez se haya completado la instalación de los paquetes ROS, habrá que inicializar el sistema *rosdep*, que se usa para facilitar la instalación de dependencias del sistema a la hora de compilar código. Esto se hace con los siguientes comandos:

sudo rosdep init

rosdep update

El último paso de la instalación de ROS será añadir las variables de entorno de ROS al fichero *.bashrc*, de esta forma se añadirán estas variables de entorno cada vez que abramos una nueva terminal, y no será necesario cargar las variables manualmente. Para ello usamos los siguientes comandos:

echo "source /opt/ros/groovy/setup.bash" >> ~/.bashrc

source ~/.bashrc

Una vez ya tenemos instalado ROS Groovy, pasamos a crear el espacio de trabajo **Catkin**. Este espacio de trabajo se usará para poder compilar y usar nuestros propios paquetes con las ventajas de las funcionalidades que ofrece catkin para poder compilar varios paquetes a la vez de forma fluida. Para ello creamos un directorio llamado *~/catkin_ws/src*, y ejecutamos el script ***catkin_init_workspace*** para inicializar el espacio de trabajo:

mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src

catkin_init_workspace

Para compilar un paquete este deberá estar dentro de la carpeta *~/catkin_ws/src*, y deberá tener asociado un fichero *CMakeLists.txt*, que se usa para indicar a catkin que es lo que tiene que compilar de cada paquete. Se puede realizar una primera compilación del espacio de trabajo aunque aún no haya ningún paquete para comprobar que la instalación ha sido correcta. Se usan los comandos:

cd ~/catkin_ws/

catkin_make

Por último solo falta añadir las variables de entorno del espacio de trabajo al fichero *.bashrc* para que se añadan los paquetes creados al sistema de paquetes de ROS:

echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc

source ~/.bashrc

3.2.4. Instalación y uso de los paquetes del TurtleBot

La instalación de los paquetes del TurtleBot se realizará instalando el meta paquete ***ros-groovy-turtlebot*** *desde los repositorios de* apt. Este meta paquete incluirá todos los paquetes relacionados con el Turtlebot que se usarán en el proyecto.

Para usar el Turtlebot primero habremos de conectarlo al ordenador mediante un adaptador serial-usb. Una vez conectado tendremos que dar permisos de acceso al dispositivo serial conectado. Para ello entramos en el directorio */dev/*, y allí deberá aparecer el dispositivo serial, normalmente se le asignará el nombre /dev/ttyUSB y un número identificador. Una vez localizado el ttyUSB correspondiente al Turtlebot

le proporcionaremos permisos de administrador con el comando ***sudo chmod a+rwx ttyUSBX***, donde la X es el número identificador del dispositivo USB.

Una vez asignados los permisos podemos arrancar los nodos de ROS para activar el funcionamiento del Turtlebot. Para ello primero arrancamos la base iRobot Create pulsando el botón de ON (sonará un pitido indicándonos que se ha arrancado correctamente). A continuación lanzamos los nodos de ROS con el launcher turtlebot.launch del paquete turtlebot_bringup *(**roslaunch turtlebot_bringup turtlebot.launch**) y si se ha iniciado todo correctamente se oirá un segundo pitido desde la base iRobot Create indicando que la comunicación es correcta.*

Si queremos comprobar que el robot está correctamente conectado al portátil y se puede mover sin problemas podemos usar el paquete *turtlebot_teleop, que lo instalaremos con el comando **sudo apt-get install ros-groovy-turtlebot-apps***. Lanzamos el nodo con ***rosrun turtlebot_teleop turtlebot_teleop_key** para iniciar el nodo que moverá la base Create a través del teclado*. Nos aparecerá una simple interfaz en la terminal a través de la cual podremos mover la base, cambiar las velocidades máximas (lineares y angulares) y parar el robot.

3.2.5. Instalación y uso de los paquetes de la Kinect

El uso de la Kinect requiere la instalación de los siguientes paquetes y librerías, todas accesibles desde los repositorios de apt:

- libopenni-nite-dev
- libopenni-dev
- ros-groovy-openni-launch
- ros-groovy-openni-tracker

Una vez instalados, y con la Kinect alimentada y conectada al puerto USB 2.0, podemos lanzar los nodos de la cámara Kinect lanzando el launcher openni.launch del paquete openni_launch (***roslaunch openni_launch openni.launch***). Una vez los

nodos se hayan iniciado se puede comprobar el correcto funcionamiento de la cámara observando que se publican los topics de la cámara con ***rostopic list***.

Estos topics pueden ser visualizados con el programa de visualización *rviz*, donde se podrá observar de manera sencilla la nube de puntos o la imagen RGB de la Kinect. Para ello lanzamos el programa rviz (***rosrun rviz rviz***), y a continuación ponemos como *fixed frame* el frame */camera_link*, para obtener una mejor visualización de la escena. Si queremos visualizar la cámara RGB hacemos click en *Add*, luego en *Image*, y en el nuevo menu desplegable que aparece a la izquierda, seleccionamos como *Image Topic* el topic que queremos mostrar, por ejemplo el topic */camera/rgb/image_color*. De esta misma manera también podremos visualizar la imagen infrarroja de la cámara, y la imagen de disparidad creada por el sensor de profundidad de la Kinect, seleccionando los topics correspondientes. También puede obtenerse la nube de puntos del sensor de profundidad y juntarlos con la imagen RGB para darle color a cada punto 3D. Para ello hacemos click en *Add*, luego en *DepthCloud*, y en el nuevo menú desplegable seleccionamos el topic */camera/depth/image* en *Depth Map Topic*, y seleccionamos el topic */camera/rgb/image_color* en la opción *Color Image Topic*. De esta forma obtenemos una visión en el visor principal del rviz de la nube de puntos con los puntos coloreados, tal como se muestra en la Ilustración 52.

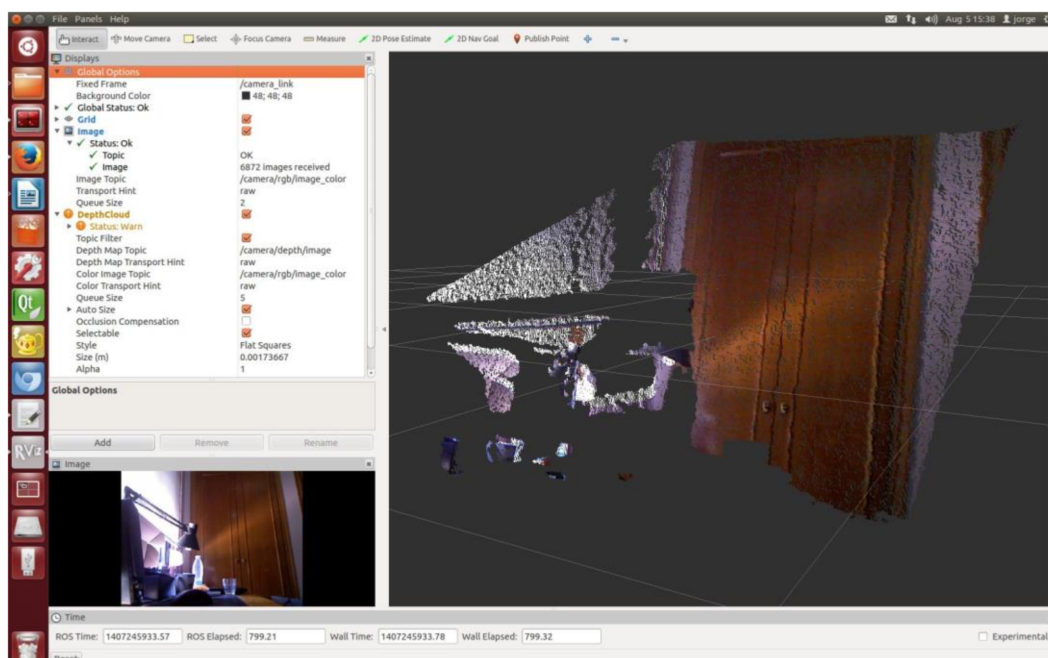

Ilustración 52 Visualización de la Kinect. Debajo a la izquierda la cámara RGB y en el centro la nube de puntos con la imagen RGB superpuesta.

openni_tracker

Podemos lanzar el nodo del openni_tracker con ***rosrun openni_tracker openni_tracker***. Es posible que al lanzar el nodo aparezca un mensaje de error como el siguiente:

[ERROR] [1407222853.912741345]: Find user generator failed: This operation is invalid!

Esto es un bug de incompatibilidad entre OpenNI y las versiones antiguas de ROS, como Electric y Groovy, que es la que se está usando. Hay una discusión sobre este bug en ROS Answers [28], donde ofrecen una solución para solventar el problema. Esta solución requiere reinstalar los drivers de Openni usando la última versión posible del middleware, accediendo a ellos desde *www.openni.org*. Lamentablemente, este sitió cerró el 23 de Abril de 2014, debido a que la empresa Apple compró a la empresa PrimeSense en Noviembre del 2013, y esta era la propietaria de OpenNI [29]. Aun estando esta página cerrada, el código de OpenNI ha sido mantenido por las organizaciones que utilizaban estas librerías, y sigue disponibles en sitios como en GitHub (*https://github.com/OpenNI/OpenNI*), en *http://structure.io/openni*, o en *http://www.openni.ru/*. Podemos descargar los binarios que necesitamos desde:

http://www.openni.ru/wp-content/uploads/2013/10/NITE-Bin-Linux-x64-v1.5.2.23.tar.zip.

Una vez descargado el fichero *NITE-Bin-Linux-x64-v1.5.2.23.tar.zip*, lo descomprimimos y ejecutamos el shell script *uninstall.sh*. Una vez desinstalado, ejecutamos el shell script *install.sh* para reinstalar los drivers OpenNI. Si queremos asegurarnos de que se han instalado correctamente habrá que comprobar que en la carpeta /usr/lib estén las versiones *1.5.2* de las librerías libXnVCNITE, libXnVFeatures, libXnVHandGenerator y libXnVNite. Una vez este todo correctamente instalado, el nodo openni_tracker ya no debería dar ningún error al ejecutarlo.

Una vez lanzado el nodo aparecerán por pantalla informaciones de cuando se detectan nuevos usuarios, pero el nodo no publicará información útil hasta que no calibremos el usuario haciendo la pose *psi*. Una vez se haya completado la calibración (lo indicará con "*Calibration Complete*"), el nodo empezará a publicar las coordenadas del esqueleto del usuario en el topic */tf*. Cada frame de este esqueleto tendrá el nombre de la parte del esqueleto a la que corresponde, y el identificador del usuario al que corresponde, por ejemplo */head_1* (la cabeza del usuario 1), o /left_knee_2 (la rodilla izquierda del usuario 2). Para visualizar este topic lo más sencillo es abrir de nuevo el programa rviz, y una vez dentro hacer click en *Add*, y seleccionar *TF*. También habrá que seleccionar como *fixed frame* el frame */openni_depth_frame*. Si todo va bien obtendremos una visualización del esqueleto como la de la Ilustración 53.

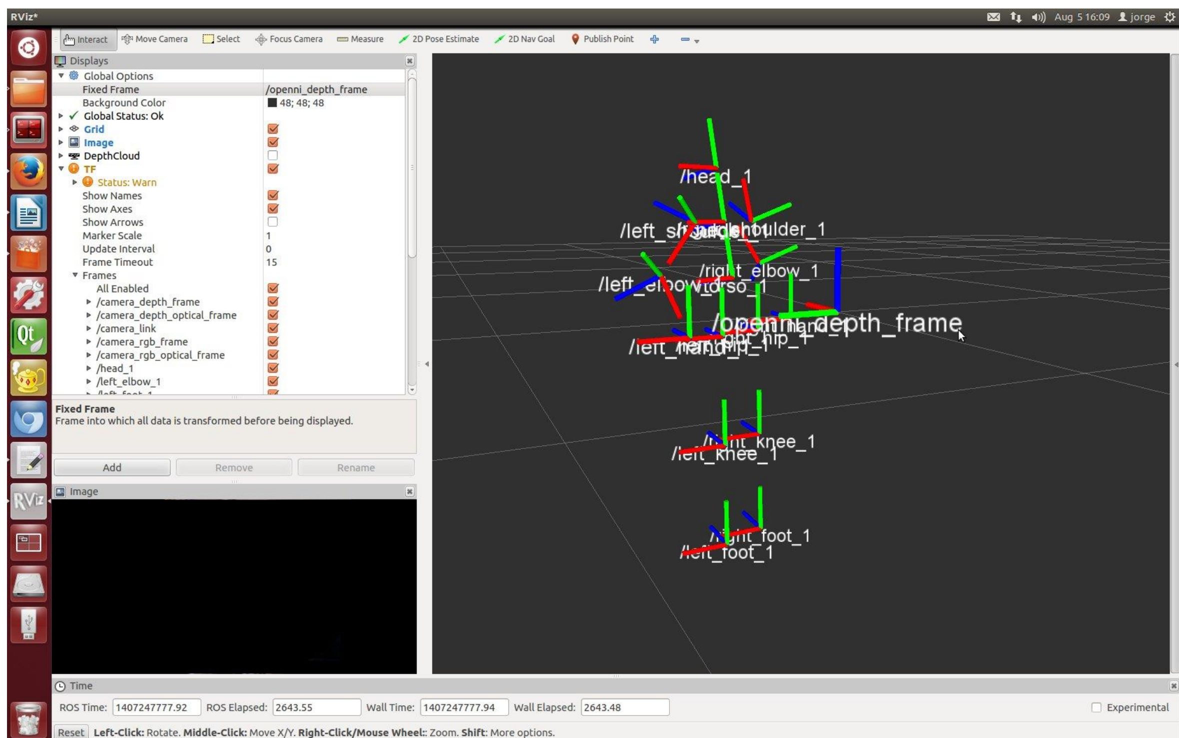

Ilustración 53 Visualización en rviz del esqueleto del usuario obtenido por el nodo openni_tracker

3.2.6. Estructura

Paquetes y nodos

Para el desarrollo del robot perseguidor se han creado dos paquetes diferentes, con tres nodos en total. Estos nodos se han separado en dos paquetes diferentes para facilitar la portabilidad y la reusabilidad del código.

El primer paquete creado es **openni_tracker_follower**. Este paquete contiene un nodo llamado **openni_tracker_ follower**. La función de este nodo es la de ampliar la funcionalidad del nodo openni_tracker para ofrecer la posición 3D de un usuario respecto a la base iRobot Create. También tendrá que realizar diferentes filtros para asegurar la fiabilidad de la información que suministrará. En el apartado 3.2.7 se explicará en detalle la funcionalidad e implementación de este nodo.

El segundo paquete creado es **turtlebot_follow_user**. Este paquete tiene dos nodos: **turtlebot_follow_user** y **turtlebot_follow_user_commands**. El primero será usado para el movimiento del robot perseguidor y el segundo será usado para moverse y al mismo tiempo tener en cuenta los comandos de voz del usuario, por lo que se explicará en detalle en la sección 3.4. El nodo *turtlebot_follow_user* recibirá por un topic la posición del usuario que hay que seguir, proporcionado por el nodo openni_tracker_follower. Una vez reciba esta posición, la función del nodo *turtlebot_follow_user* será la de moverse de manera segura y fiable hacia el usuario, manteniendo una distancia constante al usuario y no realizando movimientos bruscos. Para ello se comunicará mediante un topic con el nodelet de base móvil, enviándole órdenes de movimiento. Este nodo se explicará en más detalle en el apartado 3.2.8.

Para facilitar el uso de estos paquetes y nodos se ha creado un fichero .launch que lanzará todos los nodos necesarios para que el robot se inicie. De esta forma no nos tenemos que preocupar de que nodos estamos lanzando o que terminales

tenemos abiertas, simplemente ejecutaremos el fichero *turtlebot_follow_user.launch* del paquete *turtlebot_follow_user* (***roslaunch turtlebot_follow_user turtlebot_follow_user.launch***). Este fichero launch ejecutará lo siguiente:

- turtlebot.launch (paquete turtlebot_bringup) son los drivers de la base iRobot Create.
- openni.launch (paquete openni_launch) son los drivers de la cámara Kinect.
- openni_tracker_follower (paquete openni_tracker_follower) es el nodo que proporciona la posición del usuario.
- turtlebot_follow_user (paquete turtlebot_follow_user) es el nodo que moverá la base iRobot Create.

Si queremos visualizar todos los componentes que hemos lanzado con el fichero *turtlebot_follow_user.launch* y la comunicación que hay entre los nodos, podemos usar el programa *rqt*, con un plugin para visualizar nodos llamado *Node Graph*. Este plugin viene instalado con la configuración básica de ROS, así que no tendremos que instalar nada más. Para obtener la visualización, una vez lanzado el fichero *turtlebot_follow_user.launch*, ejecutamos el *rqt* con el plugin *Node Graph* escribiendo en la terminal el comando *rqt_graph*. De esta forma arrancará el rqt y automáticamente cargará el plugin. En pantalla veremos todos los nodos y los topics que están actualmente en funcionamiento. El plugin nos da algunas opciones de configuración como mostrar o no mostrar los topics, mostrar solo los nodos/topics que están activos en ese momento, o esconder los topics del Debugger (/rosout y /clock). En la figura 3.17 se muestra la sección más interesante del grafo obtenido. El grafo entero tiene muchos más nodos y topics, la mayoría relacionados con el funcionamiento interno de la cámara Kinect y la base iRobot Create, pero no son importantes para la explicación del desarrollo del robot, y publicar el grafo entero ocuparía varias páginas de la memoria.

Ilustración 54 Sección del grafo proporcionado por rqt Node Graph.

En el grafo los nodos están representados con círculos y la comunicación con topics entre ellos está representada con una flecha. El nombre que aparece encima de la flecha es el nombre del topic que se está usando para la comunicación. En el grafo se puede observar los dos nodos mencionados anteriormente, *openni_tracker_*follower y *turtlebot_follow_user*, además de un tercer nodo, *mobile_base_nodelet_manager*, que es el nodelet que se ocupa de los sistemas de la base móvil Create, como el movimiento de la base.

Topics

El topic /tracker_CoM (Center of Mass) es el topic que informará al nodo *turtlebot_follow_user* donde está el usuario que tiene que seguir. Este topic es del tipo *geometry_msgs/PointStamped*, y tiene la siguiente estructura (obtenida con el comando *rosmsg show geometry_msgs/PointStamped*):

```
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
geometry_msgs/Point point
    float64 x
    float64 y
    float64 z
```

Añadir un header al mensaje es una estructura común en muchos de los tipos de mensajes utilizados en ROS. Dentro del *header* aparece: “*seq*”, que es el número de secuencia del mensaje, “*stamp*”, que indica el tiempo en milisegundos en el cual se ha enviado el mensaje, y “*frame_id*”, que indica el frame del robot desde el cual se ha generado el mensaje. En nuestro caso el *frame_id* será el frame de la cámara

Kinect. El contenido útil del mensaje viene en forma de un *geometry_msgs/Point*, que contiene las coordenadas x, y, z de la posición del usuario que queremos seguir respecto a la cámara, siendo "x" la coordenada que indica si está a la izquierda/derecha, la "y" indica arriba/abajo y la "z" indica la profundidad a la que se encuentra el usuario. Este punto viene dado por el centro de masas del usuario.

El topic */mobile_base/commands/velocity* es el que proporcionará las órdenes de movimiento directamente al nodelet que controla la base iRobot Create. Este topic es del tipo *geometry_msgs/Twist*, que es un tipo de mensaje utilizado a menudo en ROS para indicar el movimiento de un robot. Este tipo de mensaje tiene la siguiente estructura:

geometry_msgs/Vector3 linear

float64 x

float64 y

float64 z

geometry_msgs/Vector3 angular

float64 x

float64 y

float64 z

geometry_msgs/Twist se compone de dos vectores en tres dimensiones, donde el primer vector indicará la velocidad lineal y el segundo vector indicará la velocidad angular. Aunque podemos enviar cualquier valor en cualquier elemento del vector a la base móvil, en realidad los valores de estos vectores están restringidos por el modelo cinemático de la base Create, siendo un modelo cinemático diferencial. Esto quiere decir que el robot solo puede moverse linealmente en el vector x (hacia delante y hacia detrás), y solo puede moverse angularmente rotando alrededor del eje z (movimiento de giro sobre sí mismo izquierda/derecha). Por tanto las componentes "y", "z" del vector linear, y las componentes "x", "y" del vector angular serán ignoradas por la base móvil. Este tipo de mensaje de

movimiento más genérico permite reutilizar mensajes en ROS sin tener que crear un tipo de mensaje de movimiento para cada modelo cinemático.

3.2.7. Creación del tracker

Descripción

La creación del tracker openni_tracker_follower se ha basado en el código del paquete ROS *openni_tracker*. Para ello se ha aprovechado la licencia BSD de los paquetes ROS (licencia que permite el uso del código fuente del programa para software libre o comercial) y se ha realizado un fork (bifurcación en desarrollo de software) del paquete openni_tracker. El código de este paquete esta subido a la web de control de versiones GitHub (como la mayoría de los paquetes de ROS) en el siguiente enlace https://github.com/ros-drivers/openni_tracker . Dentro de la carpeta *src* podemos encontrar el fichero openni_tracker.cpp, que es el que modificaremos para adaptarlo a las necesidades de nuestro proyecto.

openni_tracker.cpp utiliza las funcionalidades del middleware OpenNI a través de un nodo ROS que le permite funcionar independientemente en un entorno de trabajo ROS. De OpenNI utiliza los ya mencionados nodos de producción (no confundir con los nodos de ROS), concretamente utiliza el nodo de producción de profundidad y el nodo de producción de usuarios. Durante las primeras líneas de la función *main* se inicializan y configuran estos nodos de producción, además de unir una serie de funciones *callback* al nodo de producción de usuarios. Las funciones callback que se llamarán cuando se cumplan ciertas condiciones son:

- User_NewUser(…): se llamará cuando el algoritmo de OpenNI detecten un nuevo usuario en la escena.
- User_LostUser(…): se llamará cuando se pierda un usuario de la escena.
- UserPose_PoseDetected (…): se llamará cuando se detecte que un usuario está haciendo la pose PSI para iniciar la calibración.

- UserCalibration_CalibrationStart(…): se llamará cuando se inicie la calibración del esqueleto del usuario.
- UserCalibration_CalibrationEnd(…): se llamará cuando finalice la calibración del usuario, ya sea una calibración correcta o si falla la calibración.

Por otra parte, openni_tracker.cpp funciona como un nodo de ROS, por lo que parte de su código se usa para definir su funcionalidad dentro de la estructura ROS. En el *main* se crea el *nodeHandle*, que se usará para utilizar funcionalidades de los nodos, como topics o parámetros, y a continuación se crea el bucle principal del programa. Este bucle se repetirá a una frecuencia de 30 Hz hasta que se produzca algún problema con la comunicación con ROS (while (ros::ok())). Se utiliza una frecuencia de 30 Hz porque esa es la frecuencia a la que se recibirá información de la cámara Kinect. Dentro de este bucle se llama a la función *WaitAndUpdateAll()*, que actualiza todos los nodos de producción de OpenNI, y después llama a la función *publishTransforms()*, que publicará las transformadas del esqueleto de todos los usuarios de la escena que hayan sido calibrados correctamente.

Mejoras / cambios

Utilizando solamente el nodo de openni_tracker tenemos un programa que detecta todos los usuarios posibles de la escena, y los intenta calibrar para finalmente publicar las transformadas de los esqueletos en el topic /tf. Para el correcto desarrollo del robot perseguidor queremos realizar los siguientes cambios o mejoras:

- El tracker solo tiene que leer usuarios que realmente sean humanos. El problema se origina porque el nodo productor de usuarios de OpenNI puede detectar erróneamente usuarios cuando observa en la escena una forma “similar” a un usuario. Este problema de falsa detección de usuarios se puede dar con objetos tan comunes como con una silla, algunos muebles o incluso con algunas paredes. Por tanto, para el correcto funcionamiento del robot perseguidor, nuestro nodo tendrá que ser capaz filtrar los usuarios y

quedarse con aquellos que realmente son humanos. En la Ilustración 55 se observa un ejemplo de estos falsos reconocimientos de usuarios. Cada usuario reconocido es representado con un color diferente, y el número en el centro es el identificador que se le asigna. Se observa claramente que aparte del usuario humano (rojo), también se detectan otros usuarios no humanos, como una puerta (verde), o una pared (amarillo).

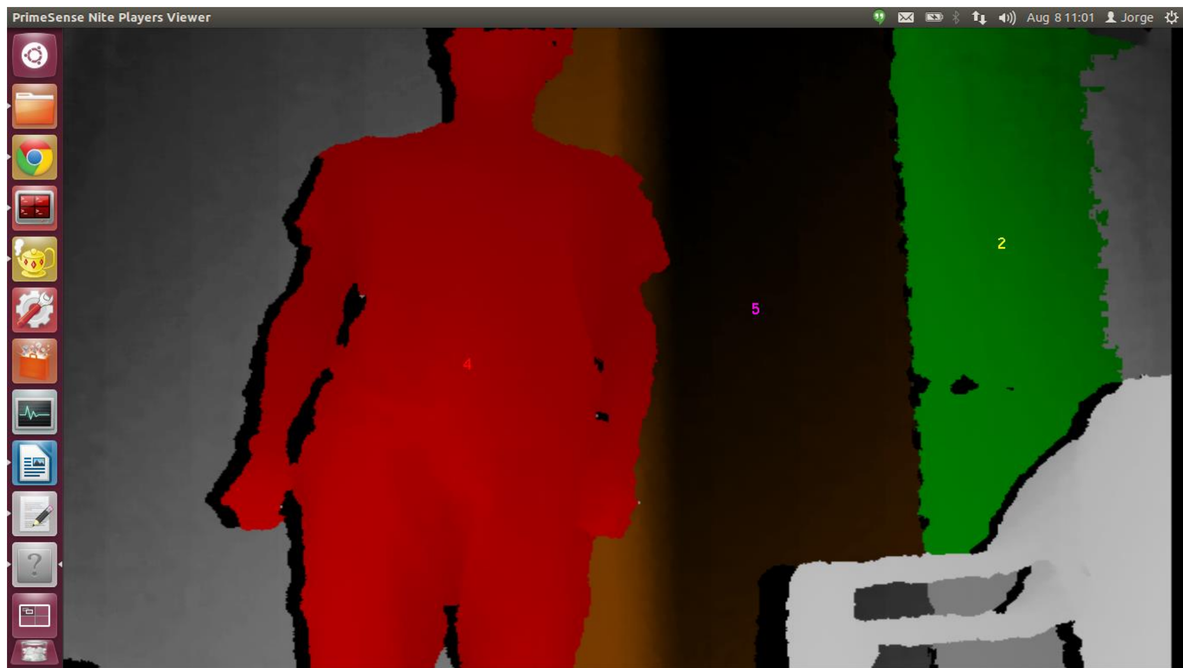

Ilustración 55 Falso reconocimiento de usuarios en los algoritmos de OpenNI.

- Una vez filtrados los usuarios no humanos, se puede dar el caso de que hayan varios usuarios en la escena. En este caso el nodo debe ser capaz de elegir que usuario tiene que seguir.

- El nodo openni_tracker original requiere una calibración haciendo la pose PSI para obtener el esqueleto del usuario. Para la función de seguimiento de usuarios no se va a requerir el esqueleto, ya que se van a seguir usuarios y por tanto no hará falta realizar una calibración. Pero para el siguiente objetivo

del proyecto, que es detectar posibles caídas o accidentes del usuario sí que va a ser importante obtener el esqueleto del usuario. La funcionalidad de tener que realizar la pose PSI y esperar unos segundos para esperar a la calibración no es viable para este proyecto, ya que requiere demasiado esfuerzo extra al usuario y queremos que el funcionamiento del robot sea lo más autónomo posible. Además de que sería necesario realizar una calibración cada vez que un usuario aparece en la escena, no basta con solo una calibración por usuario. Por tanto hay que encontrar una solución para realizar algún tipo de calibración automática de los usuarios.

- Por último, vamos a requerir que se publique un topic que indicará al nodo *turtlebot_follow_user* hacia donde tiene que moverse.

Calibración automática

El problema de evitar realizar una calibración para cada usuario es relativamente fácil de solucionar mediante una calibración automática. Para ello tendremos que guardar en un fichero la calibración del usuario cuando se haya calibrado correctamente, y luego cargar esa calibración cada vez que aparezca un nuevo usuario que necesite calibrarse. La API de OpenNI tiene dos funciones que nos permitirán guardar y cargar los datos de calibración a un fichero. Estas dos funciones son:

```
XnStatus xn::SkeletonCapability::SaveCalibrationDataToFile(
        XnUserID    user,
        const XnChar * strFileName
        )
XnStatus xn::SkeletonCapability::LoadCalibrationDataFromFile(
```

```
XnUserID      user,
const XnChar * strFileName
)
```

Llamaremos a la función SaveCalibrationDataToFile(…) dentro de la función callback UserCalibration_CalibrationEnd(…), y llamaremos a la función LoadCalibrationDataFromFile(…) dentro de la función callback User_NewUser(…). Usaremos el booleano *calibrationSaved* como flag de control para evitar que se cargue una calibración si no hay ningún archivo de calibración guardado. Si se realiza una nueva calibración se sobrescribirá la calibración anterior.

Filtrado de usuarios

Filtrar los usuarios para evitar el problema del falso reconocimiento de usuarios en los algoritmos de OpenNI es un problema bastante más complicado que la calibración automática. A continuación se explican algunos intentos de realizar un filtrado selectivo de usuarios.

Una posible solución para realizar el filtrado es utilizar el esqueleto del usuario. Como hemos implementado una calibración automática, todos los usuarios que aparezcan en la escena tendrán asociado un esqueleto. Esto quiere decir que incluso los usuarios no humanos tendrán asociados un esqueleto, y este será publicado en el topic /tf. El objetivo de este filtrado sería obtener características relevantes del esqueleto, y estudiar si algunas de estas características se dan solo en usuarios humanos. Algunas de las características que se han estudiado han sido las distancias entre algunas articulaciones del esqueleto (por ejemplo, entre el torso y la cabeza), o el ángulo creado entre los elementos del esqueleto, para ver si se asemejan a una forma humana. Finalmente este método se ha descartado debido a

que la calibración que se obtiene de un elemento no humano no es nada fiable, teniendo a veces forma humana y otras veces tomando una forma totalmente aleatoria y caótica. Debido que a veces toman forma de esqueleto humano no se puede realizar una separación precisa de los usuarios humanos de los usuarios no humanos.

Otro acercamiento al problema del filtrado de usuarios ha sido intentar distinguir entre objetos estáticos y dinámicos. Se parte de la base de que un usuario no humano, como una pared o una puerta, en principio no va a estar moviéndose, por tanto podemos calcular un índice de movimiento para descartar los usuarios que no se mueven, y quedarse con los usuarios que si se mueven. Para ello se calcula la velocidad media durante un segundo de la cabeza del esqueleto del usuario (calibrado automáticamente). Para conseguir esta velocidad media primero se guardan las velocidades instantáneas en una lista enlazada, en la que se la lista se rellena completamente cada segundo, y en cada iteración del bucle se calcula la velocidad media calculando la media de los elementos de la lista enlazada. Se utiliza un límite de 0.01 m/s para que un objeto se considere dinámico y por tanto un usuario. Tras varias pruebas este método de filtrado de usuarios tampoco da los resultados esperados y tiene que descartarse. El filtrado de usuarios recaerá en el algoritmo de selección del usuario a seguir, que tendrá que tener en cuenta que tipo de usuario está siguiendo. El método del cálculo de la velocidad del usuario también se ha usado en la detección de caídas del usuario (que se explicará más adelante), para detectar movimientos bruscos de la persona.

Selección del usuario a seguir

Necesitamos un algoritmo que decida que usuario hay que seguir cuando aparecen varios usuarios en la escena. Debe ser capaz de elegir al usuario más importante o intentar seleccionar los usuarios humanos antes que los usuarios no humanos (puertas, paredes, muebles, etc …)

Un primer intento de este algoritmo ha sido crear unos límites espaciales en la escena. El usuario que se encuentre dentro de un rango determinado (se ha usado el eje x) será el usuario que se seguirá. En caso de que aparezcan varios usuarios en el centro, el usuario más centrado será elegido por el algoritmo. Haciendo varias pruebas se observa que este algoritmo no da buenos resultados, ya que el robot no siempre tiene al usuario centrado, por ejemplo en giros bruscos, y muchas veces se queda siguiendo elementos no humanos como una pared.

La solución por la que finalmente se ha optado ha sido la siguiente. En cada iteración del bucle principal se calcula el porcentaje que cada usuario ocupa respecto al total de la escena, y el usuario que supere un cierto umbral de aparición en pantalla será el usuario que se seguirá. Se guardará el identificador de usuario y no se dejará de seguir hasta que desaparezca completamente por pantalla. Cuando se pierda al usuario se volverá a realizar la búsqueda de un nuevo usuario que supere el umbral de aparición. Este sistema evita el problema de perder al usuario cuando se producen giros bruscos, y también evita que el robot siga a elementos no humanos como una pared. Para ello se ha utilizado el siguiente método de la API de OpenNI:

```
XnStatus xn::UserGenerator::GetUserPixels(
        XnUserID user,
        SceneMetaData & smd )
```

GetUserPixels devuelve en smd la información de los píxeles del usuario. Llamando a smd.Data() obtenemos un vector de etiquetas, de tamaño igual al tamaño de la imagen, donde cada elemento del vector pertenece a un píxel y tiene el identificador del usuario al que pertenece ese píxel. Realizando un simple bucle para recorrer este vector, calculando el número de píxeles que pertenece a cada usuario y dividiéndolo por la resolución de la pantalla podemos obtener el porcentaje de ocupación de cada usuario en pantalla en cada instante.

Se ha elegido como umbral un 35% de ocupación. Por tanto el primer usuario que supere este umbral será el usuario que se seguirá, y no dejará de seguirse hasta que no haya ningún píxel del usuario en pantalla. Una vez se pierda el robot esperará hasta que no aparezca un usuario con un 35% de ocupación.

Publicación de topics

Por último el nodo tiene que ser capaz de comunicarse con el nodo *turtlebot_follow_user* por lo que tenemos que crear la comunicación por topics. Para ello, antes de iniciar el bucle principal, inicializamos el *Publisher* que publicará el topic en la línea:

```
ros::Publisher tracker_pub = nh.advertise<geometry_msgs::PointStamped>("tracker_CoM", 1);
```

La función advertise(…) está parametrizada por lo que hemos de indicarle el tipo de mensaje que se publicará en el topic. El primer argumento de la función es el nombre del topic que se publicará y el segundo argumento indica la cola de salida del topic. En este caso no queremos que se forme ninguna cola en el topic, para que no hayan problemas de retrasos de los mensajes, por lo que ponemos el valor de la cola a uno. Advertise(…) es un método del objeto *nh,* que es de tipo ros::NodeHandle.

Ya dentro del bucle principal, tendremos que publicar el punto a seguir solo cuando tengamos un usuario a seguir, para ello utilizamos el flag de control *publishCoM*. Antes de publicarlo realizamos una última comprobación, y es que las coordenadas del punto que se va a enviar NO sean exactamente iguales a las del punto que se envió en la iteración anterior. Esto se hace porque el nodo productor de usuarios sigue enviando información de un usuario cuando ya ha desaparecido de la pantalla durante 10 segundos. Pero esta información que se envía es exactamente igual durante estos 10 segundos. Queremos que el nodo detecte instantáneamente la

desaparición de algún usuario, por lo que realizamos una comprobación de si se está dando el caso de que las coordenadas sean exactamente iguales. Si el usuario sigue en pantalla, aunque este esté prácticamente estático, las coordenadas del centro de masas que se enviarán tendrán una pequeña variación, debido al ruido y a la baja resolución de la cámara, y se enviará la información por el topic correctamente.

Una vez realizadas estas últimas comprobaciones, se publica el mensaje en el topic con el código:

```
tracker_pub.publish(pointStampedCoM);
```

Donde pointStampedCoM es un geometry_msgs::PointStamped con las coordenadas del punto a seguir. Una vez se ha enviado este punto por el topic, el siguiente paso es el movimiento del robot, que se ha realizado con el nodo *turtlebot_follow_user* del paquete del mismo nombre. Este paquete se explica a continuación.

En la Ilustración 56 se muestra una captura de pantalla de los datos obtenidos con el detector de usuarios. Arriba a la izquierda se muestra una visualización de la cámara Kinect, donde se ve al usuario que se tiene que seguir. Arriba a la derecha se muestra el árbol de transformadas del esqueleto del usuario visualizado en Rviz. Debajo se muestran dos terminales donde se observa la salida estándar del nodo openni_tracker_jorge, y a la derecha se muestra el topic donde se publica la posición del usuario respecto a la cámara en coordenadas x,y,z.

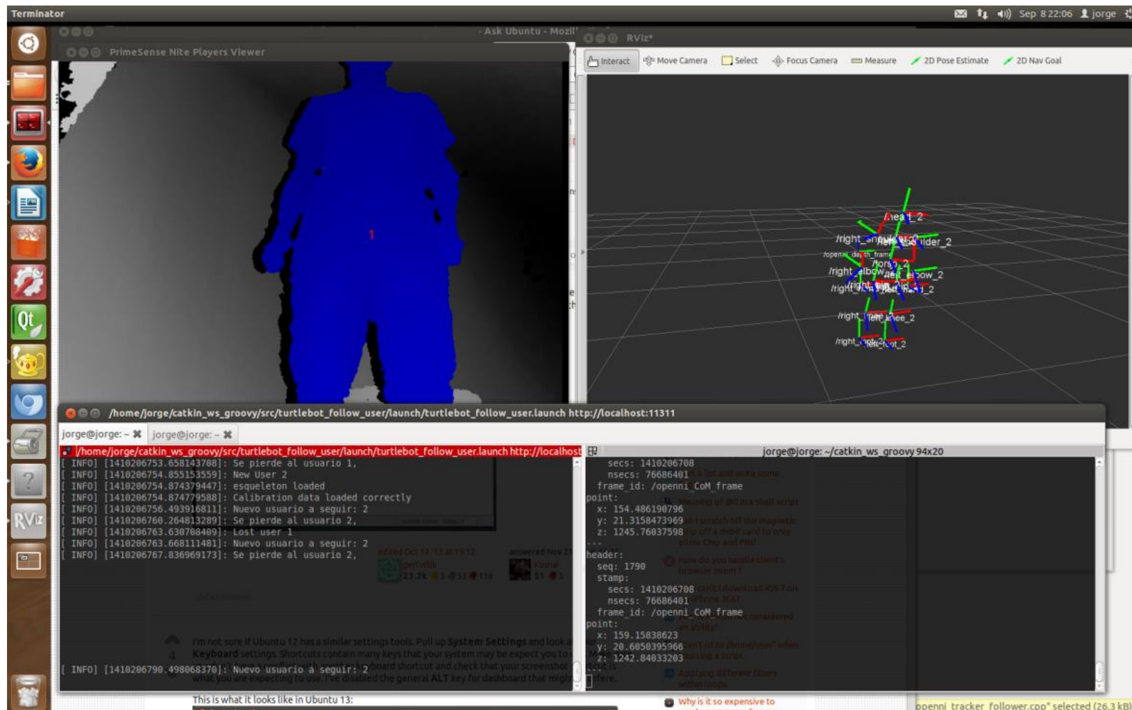

Ilustración 56 Captura de pantalla de la identificación de usuarios

3.2.8. Nodo de movimiento

El nodo turtlebot_follow_user lo definimos y compilamos con el fichero turtlebot_follow_user.cpp. El comportamiento de este nodo funcionará con llamadas callback del topic que envía la posición del usuario, y publicará un topic con un mensaje del tipo geometry_msgs::Twist para definir la velocidad a la que se moverá la base.

Para ello en la función main(…), definimos la subscripción al topic */tracker_CoM*, y definimos el topic que se usará para publicar el topic de movimiento, que será el topic */mobile_base/commands/velocity*, de tipo Twist. A continuación llamamos a la función de ros ros::Spin(). Esta función pone en marcha el nodo y llamará a las funciones callback correspondientes cuando llegue el topic que las activa.

El movimiento de la base se realiza definiendo una distancia objetivo al usuario, que es la distancia que se intentará mantener en todo momento. Se ha elegido una distancia objetivo de aproximadamente 1 metro, que es una distancia adecuada para no molestar al usuario. Si el usuario está más cerca del robot que esta distancia objetivo, el robot se moverá hacia detrás, y su está más alejado, el robot se moverá hacia delante. Estos límites tienen una pequeña holgura para que el robot no se mueva si el usuario no se movido significativamente. Se hace lo mismo para definir el giro del robot. Se definen unos límites en el eje x de la cámara en el cual el usuario tiene que estar situado. Si se sale de esos límites (por la derecha o por la izquierda) el robot girará para mantener al usuario centrado en todo momento.

Un aspecto importante a la hora de definir el movimiento del robot son los movimientos bruscos de la base. Se ha observado que el realizar movimientos bruscos o movimientos que induzcan un pequeño balanceo en la base provoca que los algoritmos de OpenNI fallen. Concretamente falla el nodo productor de usuarios. A veces se da el caso de que el nodo productor de usuarios indica que se ha perdido un usuario cuando el usuario no se ha movido, solamente por inducir un pequeño balanceo. Para solucionar esto se ha añadido una aceleración suave a los movimientos del robot, para tener un aumento de la velocidad constante y así evitar estos movimientos bruscos y balanceos de la base. Para ello, cuando haya que mover la base iRobot Create, en vez de asignarle directamente la velocidad máxima, se aumentará la velocidad progresivamente en cada iteración, hasta llegar a su velocidad máxima. Lo mismo se hace para frenar al robot (que tendrá que frenar más rápidamente para evitar choques con el usuario), y también para los giros del robot.

Por último se envía el topic de tipo geometry_msgs::Twist. Para ello se define la velocidad linear en la componente x del vector linear, y se define la velocidad angular en la componente z del vector angular. Una vez definidos estos componentes se envía el mensaje de movimiento por el topic /mobile_base/commands/velocity.

En la Ilustración 57 y Ilustración 58 se muestran unos frames del video realizado para mostrar el funcionamiento del follower:

Ilustración 57 Robot seguidor. Se puede apreciar que el robot se moverá para mantener una distáncia constante al usuario.

Ilustración 58 Nodo seguidor. En esta secuencia de imágenes se observa como el robot sigue el movimiento del usuario por las esquinas y pasillos.

3.3. Detección de caídas

3.3.1. Descripción

El siguiente objetivo del proyecto es la detección de accidentes o caídas del usuario. El robot, al estar siguiendo a un usuario, debe ser capaz de detectar estas situaciones de peligro de una manera fiable y precisa, ya que la seguridad del usuario podría depender de ello. Una vez detectada esta situación de peligro, habrá que comunicar esta situación a algún teléfono de emergencias o a una persona competente. También será posible activar este llamada de emergencias de forma manual por el usuario usando los comandos de voz que se describirán en el siguiente apartado.

Para la detección de caídas se usarán algunos de los algoritmos de OpenNI utilizados en el apartado anterior del seguimiento del usuario. En concreto se usará el nodo productor de usuarios y la obtención del esqueleto del usuario obtenido de manera automática. No se usará ningún paquete nuevo ni ninguna librería nueva. Toda la nueva funcionalidad creada estará en el fichero openni_tracker_follower.cpp

La detección de la caída del usuario se realizará de dos formas distintas. La primera consistirá en calcular la altura a la que se encuentra el usuario, y situar un límite de altura que si es sobrepasado se detectará como caída. La segunda forma de detección de caídas consistirá en detectar movimientos descendentes del usuario que se produzcan de manera brusca. Estas dos maneras de detectar una posible caída ofrecen mayor fiabilidad a la hora de detectar un posible accidente, debido a la independencia de las dos detecciones. Por ejemplo, una de las dos detecciones podría fallar en caso de accidente del usuario pero la segunda forma de detectar caídas seguirá funcionando.

3.3.2. Desarrollo

Detección de la altura del usuario

El cálculo de la altura a la que se encuentra el usuario funcionará de la siguiente manera. Partimos del nodo desarrollado para el seguimiento del usuario, explicado en el apartado anterior, en el que realizábamos una calibración automática del usuario que queremos seguir. Con esta calibración automática podemos obtener el esqueleto del usuario usando el nodo productor de usuarios de OpenNI. Este esqueleto nos proporciona una transformada por cada elemento del cuerpo del usuario (cabeza, tronco, piernas, brazos, manos, etc…). Esta transformada nos ofrece la posición de los elementos de este esqueleto, mediante una posición absoluta o una posición relativa a la cámara. Estas transformadas y su posición absoluta deberían ser suficientes para obtener la altura del usuario, pero se ha observado que estas posiciones no se asocian directamente a la altura real del usuario. Esto es debido a que los parámetros extrínsecos de la cámara Kinect no están correctamente calibrados, ni pueden ser calibrados con precisión debido al movimiento constante de la base móvil. Por tanto habrá que realizar una corrección de manera manual para obtener la altura real del usuario.

Para obtener esta altura utilizaremos como referencia la transformada de la cabeza del esqueleto del usuario. Se observa que, debido a la mala calibración extrínseca de la cámara, solo se obtienen buenos valores de la altura en un cierto rango de valores de distancia a la cámara. Las medianas de estos valores son:

- Angulo de inclinación (α) = 0.3898
- Distancia a la cámara (x) = 1.4108
- Altura respecto de la cámara (z) = 0.5765

Tomamos estos valores como referencia para obtener la altura del usuario. Lo que se propone es obtener la altura del usuario en cualquier distancia a la cámara, usando el plano de la cámara, y luego aplicarle una corrección a esta altura usando el valor de altura obtenido anteriormente, que sabemos que es un valor real. Para ello aplicamos la siguiente ecuación:

$$altura = z + (x_{head} - x) * (\tan(\alpha - \beta))$$

Donde:

- $\beta = \operatorname{atan}\left(\frac{z_{head}}{x_{head}}\right)$
- z, x, α: son las constantes de la medida de referencia.
- x_head, y_head: son las componentes de la posición de la transformada de la cabeza, obtenida del esqueleto del usuario.

Con esta fórmula podemos obtener unos valores reales de la altura del usuario. Este cálculo de la altura estará restringido por la distancia del usuario a la cámara, ya que dará unos valores bastante alejados de la realidad si el usuario se encuentra muy cerca o lejos de la cámara. En general no se dará este caso ya que el robot siempre se mantendrá a una distancia constante al usuario de aproximadamente un metro. Si el usuario se encuentra más alejado o más cercano que esta distancia de un metro, la base móvil corregirá esta distancia y se seguirán obteniendo valores reales de la altura del usuario.

Como umbral de altura finalmente se ha optado por un valor de 0.35. Esto asegurará que se activará la detección de caídas cuando el usuario realmente se haya caído al suelo, y que no se activará cuando, por ejemplo, el usuario esté sentado en una silla.

Detección de movimientos bruscos

El segundo método que se usará para detectar caídas es la de detectar movimientos bruscos del usuario. Para ello se ha reutilizado parte del código utilizado para medir la velocidad del usuario, explicado en el apartado de filtrado de usuarios, y que se usaba para intentar distinguir un usuario estático de uno dinámico. El nuevo código calculará la velocidad de la cabeza del usuario. Al igual que en el otro cálculo de velocidades, se usará una cola enlazada en la que se van insertando las velocidades instantáneas de la cabeza del usuario. La cola se rellena completamente cada segundo, y en cada instante se calcula la media de los elementos de la cola para obtener la velocidad media en el último segundo. El umbral que se ha elegido para este método es de -0.10 m/s. Así, cuando se detecte que la cabeza del usuario supera este valor (que sea menor que -0.10m/s) se activará la detección de la caída por este método.

Comunicación externa

Una vez tenemos estos dos métodos de detectar caídas, hay que hacer algo cuando se detecte que se ha producido una caída. Tenemos varias opciones para este fin, cada una con sus ventajas e inconvenientes. Por ejemplo, se podría realizar una simple llamada al número de urgencias 112, utilizando como micrófonos los micrófonos que incorpora la Kinect, aunque ello requeriría la incorporación de algún tipo de altavoz al robot para que la comunicación sea posible en los dos sentidos. También se podría utilizar algún software que permita la comunicación de vídeo y voz sobre Internet, por ejemplo Skype. Esta sería una buena opción ya que en principio se podría usar la cámara RGB de la cámara Kinect para ofrecer la transmisión de la imagen. Aunque esta parece una de las mejores opciones, debido a problemas de incompatibilidad con los drivers de la cámara Kinect, la distribución de Linux Ubuntu, y el programa Skype, finalmente no se ha conseguido hacer funcionar de esta forma.

Otra opción sería desarrollar una estación de monitorización externa, en la que un profesional pueda recibir la alarma de que se ha detectado una caída. Entonces este profesional podría ponerse intentar ponerse con el usuario o acceder

de forma remota a las cámaras del robot para comprobar si se trata de una caída o de una falsa alarma. Si detecta que se trata de una emergencia, esta persona llamaría a los servicios de emergencia. Esta opción tiene la desventaja de tener que requerir una persona que esté atenta a estas alarmas, por lo que la comunicación a un servicio de urgencias dejaría de ser completamente automática.

Para estas posibles soluciones se podría utilizar los sistemas de comunicación distribuidos que ofrece ROS. En el nodo de openni_tracker_follower se utiliza el topic */userFall*, de tipo *std_msgs::Bool*, para publicar continuamente si se ha producido una caída o no. Este topic puede ser leído por un tercer nodo que funcione como monitorización, para realizar las acciones necesarias cuando se detecte un estado de alarma. Se ha realizado un ejemplo de cómo se podría realizar este código en el paquete turtlebot_security_camera. Este paquete tiene un nodo que se encarga de leer este topic, y en caso de alarma acceder y mostrar la imagen que ofrece la Kinect. Debido la estructura distribuida de ROS, este nodo no tiene por qué estar ejecutándose en el mismo portátil que el resto de nodos. Puede realizarse una conexión remota desde otro ordenador que puede estar en una red distinta, y desde ahí conectarse al espacio de trabajo de ROS del robot. Después habría que arrancar este nodo para poder acceder a la cámara de forma remota si se produce alguna caída.

3.4. Reconocimiento de voz

3.4.1. Descripción

El último de los objetivos de este proyecto consiste en poder recibir y procesar diferentes comandos de voz para poder realizar una teleoperación del robot mediante órdenes de movimiento, o poder activar el modo del seguimiento automático de usuarios. La posibilidad de comunicarse con el robot de forma hablada es un método más complejo de diseñar que los métodos tradicionales de interacción entre personas y máquinas, como usando botones o interfaces gráficas, pero tiene grandes ventajas si ha sido correctamente diseñado e implementado. Su principal ventaja es que esta comunicación hablada es un método más sencillo e intuitivo para

el usuario final, que puede que no esté tan familiarizado con las máquinas y los ordenadores, y para el que puede ser mucho más cómodo y efectivo poder comunicarse directamente con el robot de forma hablada.

Para poder crear este reconocimiento de voz se aprovecharán los 4 micrófonos que tiene la cámara Microsoft Kinect en su base para obtener el audio del usuario. Una vez obtenido el audio con las órdenes para el robot, se utilizará el reconocedor del habla CMU Sphinx [30], desarrollado por la Carnegie Mellon University, y que tiene licencia open source. La librería CMU Sphinx está incluida en el paquete ROS llamado PocketShpinx, que es una versión reducida de las librerías Sphinx, especialmente diseñada para sistemas empotrados. Se configurará y usará uno de los nodos de este paquete que permitirá reconocer el habla usando un modelo de lenguaje y un diccionario de palabras. Una vez tengamos las órdenes clasificadas, se creará un nuevo nodo que permita el movimiento del robot teniendo en cuenta estas órdenes de movimiento mediante comandos de voz.

Cabe mencionar que antes de escoger el reconocedor del habla CMU Sphinx, se probó exhaustivamente otro software de audio para ROS, llamado HARK [31], desarrollado por la universidad de Kyoto. La programación de HARK se realiza mediante una interfaz gráfica, uniendo y configurando diferentes nodos que representan los procesos del reconocimiento y procesado del audio. Aunque HARK ofrece más funcionalidades que CMU Sphinx, por ejemplo módulos de localización del sonido o separación de diferentes fuentes del audio, finalmente se han utilizado las librerías de CMU Sphinx, debido a la facilidad de uso de este reconocedor, y a la compatibilidad directa con los micrófonos de la cámara Kinect.

3.4.2. Instalación

Para poder utilizar los micrófonos del sensor Microsoft Kinect en la distribución de Linux Ubuntu, primero tendremos que descargar los drivers necesarios. Para ello usamos el repositorio de git “kinect-audio-setup.git” que

incluye los drivers para poder acceder al input de los micrófonos de la Kinect además de una serie de herramientas de configuración. Procedemos a descargarnos el repositorio con el comando:

git clone git://git.ao2.it/kinect-audio-setup.git

Esto nos creará la carpeta *kinect-audio-setup* en el directorio actual, y descargará en ella todos los ficheros del repositorio. Previamente necesitaremos tener instalado el software de control de versiones git. Si no lo tenemos lo podemos instalar con el siguiente comando:

sudo apt-get install git

A continuación compilamos el repositorio con el comando:

sudo make install

Luego tendremos que ejecutar el archivo shellscript *kinect_fetch_fw,* que descargará e instalará el firmware de la cámara Kinect de la página web de Microsoft. Este shellscript necesita los programas *wget* y *7z*. Los podemos instalar con los siguientes comandos:

sudo apt-get install wget

sudo apt-get install p7zip-full

Y ejecutaremos el fichero shellscript con el siguiente comando:

sudo ./kinect_fetch_fw /lib/firmware/kinect

A continuación utilizaremos el programa udevadm con el siguiente comando para actualizar las reglas udev (son las reglas que utiliza Linux para controlar y gestionar los dispositivos):

sudo udevadm control -reload-rules

Por último habrá que reconectar el cable USB de la cámara Kinect al pc para que se reconozcan los micrófonos. Si queremos comprobar si los drivers están bien conectados podemos usar el comando:

arecord -l

Y comprobar que aparece el dispositivo Kinect. También podemos comprobar si se puede acceder a los micrófonos leyendo el archivo /proc/asound/cards, y viendo si aparece el audio de la Kinect. Para un uso correcto de los micrófonos tendremos que comprobar que los micrófonos de la Kinect tienen un volumen de entrada correcto (que no esté desactivado). Eso lo podemos comprobar en Ubuntu accediendo a System Settings →Sound →Input, y comprobando el volumen.

Una vez ya estén instalados correctamente los drivers del micrófono de la Kinect, podemos instalar el paquete de ROS PocketSphinx con el siguiente comando:

sudo apt-get install ros-groovy-pocketsphinx

3.4.3. Uso de PocketSphinx

Dentro del paquete pocketsphinx, se va a utilizar el nodo *recognizer.py*. Este nodo actuará como interfaz entre ROS y las librerías de reconocimiento de voz CMU Sphinx. El nodo *recognizer.py* necesita 2 ficheros para poder funcionar correctamente: un modelo de lenguaje y un diccionario de palabras. El modelo de lenguaje se usa para restringir la búsqueda de palabras, definiendo que palabras pueden seguir a las palabras reconocidas anteriormente, y ayuda a restringir considerablemente el proceso de selección de palabras quitando las palabras que no son probables. El diccionario de palabras contiene los fonemas que componen cada palabra que se va a usar.

Si utilizamos el nodo *recognizer.py* directamente (***rosrun pocketsphinx recognizer.py***) nos informará del siguiente error:

[ERROR] [WallTime: 1408175471.252505] Please specify a language model file or a fsg grammar file

Por lo que primero tendremos que cargar el modelo de lenguaje y el diccionario de palabras en el servidor de parámetros de ROS, en los parámetros */recognizer/lm* y */recognizer/dict*, respectivamente. Para crear estos ficheros se han modificado los ficheros de ejemplo que ofrece el paquete pocketsphinx, dentro de la carpeta *demo*. Se han copiado los ficheros *voide_cmd.dic* y *voide_cmd.lm* de la carpeta demo del paquete pocketsphinx (*/opt/ros/groovy/share/pocketsphinx/demo*) a la carpeta del paquete *turtlebot_follow_user*, que es el paquete donde se creará todo lo relacionado con el reconocimiento del habla. Se ha modificado el fichero voice_cmd.dic para dejarlo con el siguiente subconjunto de palabras y fonemas:

Archivo *voice_cmd.dic* modificado:

BACK	B AE K
FORWARD	F AO R W ER D
LEFT	L EH F T
MOVE	M UW V
RIGHT	R AY T
STOP	S T AA P

Esos son las 6 palabras que se identificarán por el reconocedor, con los fonemas que las componen a la derecha. La forma más fácil de cargar estos dos

ficheros en el servidor de parámetros de ROS es usando un archivo launch. Este launch se define de la siguiente forma:

Archivo *voice_cmd.launch*:

```
<launch>


  <node name="recognizer" pkg="pocketsphinx" type="recognizer.py"
output="screen">

    <param name="lm" value="$(find turtlebot_follow_user)/voice_cmd.lm"/>

    <param name="dict" value="$(find turtlebot_follow_user)/voice_cmd.dic"/>

  </node>


</launch>
```

Si lanzamos este archivo launch (***roslaunch turtlebot_follow_user voice_cmd.launch***) ya tendremos el reconocedor de habla configurado correctamente. Si hablamos por los micrófonos de la Kinect podemos ver cómo van apareciendo las palabras por pantalla, limitadas a las palabras que se han definido en el diccionario de palabras. El reconocedor puede reconocer frases completas, por lo que si se habla continuamente primero aparecerá un reconocimiento parcial de la frase (indicado por la etiqueta *Partial:*), y cuando se deje de hablar, el programa detectará que la frase ha finalizado, y esta aparecerá escrita en forma de un mensaje de ROS_INFO, por ejemplo:

```
[INFO] [WallTime: 1408176565.007596] move forward
```

Este reconocimiento de la frase será entonces enviado por el topic */recognizer/output*, utilizando un mensaje de tipo *std_msgs/String*, siendo un tipo de mensaje muy simple que contiene solo un dato:

std_msgs/String:

string data

3.4.4. Estructura utilizada

Como ya se ha mencionado anteriormente, toda la funcionalidad del reconocimiento del habla se incluirá en el paquete *turtlebot_follow_user*. Se utilizará una estructura similar a la creada para el tracker de usuarios pero con algunos cambios. El primer cambio es que se creará un nodo nuevo, llamado *turtlebot_follow_user_commands*, que sustituirá al nodo *turtlebot_follow_user*. Este nodo está basado en el nodo *turtlebot_follow_user*, pero incluye la funcionalidad de recibir comandos de voz y procesarlos para indicar los movimientos a la base iRobot Create. La funcionalidad e implementación de este nodo se explicará con más detalle en el siguiente apartado. *turtlebot_follow_user_commands* estará subscrito al nodo */recognizer* del paquete *pocketsphinx*, usando el topic */recognizer/output*. El resto de la estructura será prácticamente igual a la creada para el desarrollo del robot seguidor de usuarios. El grafo de esta estructura se muestra en la Ilustración 59, y al igual que otros grafos mencionados anteriormente, este grafo es sólo el subconjunto más importante obtenido con el comando *rqt_graph*.

Ilustración 59 Sección del grafo proporcionado por rqt Node Graph.

Para usar la nueva funcionalidad del reconocimiento del habla, junto con todos los nodos necesarios de la Kinect y la base iRobot Create, se ha creado un fichero launch llamado *turtlebot_follow_user_commands.launch*. Este launcher lanzará y pondrá en marcha todas las funcionalidades que se han creado en este proyecto. El launcher puede leerse en el Anexo 1.

3.4.5. Desarrollo

El código turtlebot_follow_user_commands.cpp es similar al código de turtlebot_follow_user.cpp, por ejemplo utiliza una función callback para leer la posición del usuario que tiene que seguir y realiza los cálculos necesarios para moverse de forma progresiva y segura. Ahora además de eso tendremos que procesar los mensajes del reconocedor de voz que nos llegan por el topic */recognizer/output*. Para ello nos subscribimos a ese topic y creamos una función callback que se llamará cuando llegue uno de estos topics. En esta función callback procesamos el string con el comando del usuario para solo utilizar la primera palabra de la frase que se recibe. De esta forma se simplifica mucho el procesador de las instrucciones de voz, y las órdenes de usuario siempre van a ser de una frase corta de una sola palabra con la instrucción de movimiento.

Con las instrucciones “forward”, “back”, “left” y “right”, lo que se hará es modificar los valores de la velocidad linear y la velocidad angular para adaptarse a estos movimientos. Se utilizará una velocidad más lenta que la utilizada en los anteriores algoritmos para facilitar el control mediante voz. Con las dos instrucciones restantes, “move” y “stop”, lo que se hará es activar o desactivar el seguimiento automático de usuarios. Con “move” el robot empezará a buscar usuarios que puede seguir con la Kinect, y con el comando “stop” el robot se parará y se quedará a la espera de otras órdenes de movimiento. El nodo se moverá automáticamente siguiendo al usuario, o se moverá siguiendo las órdenes de movimiento por voz, pero nunca intentará hacer las dos cosas a la vez.

4. Conclusiones y trabajos futuros

Una vez finalizado el presente proyecto, se puede concluir que la mayoría de los objetivos que se propusieron al principio se han cumplido. El primer objetivo, que es el de realizar un seguimiento automático de la persona, creemos que se ha alcanzado satisfactoriamente, como se puede comprobar en el video incluido como material adicional al proyecto. Este objetivo es al que se le han dedicado más recursos, debido a que llevó más tiempo familiarizarse con el entorno de trabajo ROS, con las librerías OpenNI y con los drivers de la Kinect y de la base móvil Create. Uno de los problemas que surgieron con este objetivo fue el evitar la calibración manual del usuario cada vez que se quería utilizar el programa, para lo que se tuvo que desarrollar como solución una calibración automática.

El objetivo de monitorizar al usuario y detectar caídas también se ha conseguido de forma suficientemente satisfactoria. Ajustar esta detección también nos exigió bastante tiempo hasta que se obtuvo la fórmula y los parámetros correctos para detectar la caída correctamente. El objetivo de ponerse en contacto con un servicio de emergencias externo se ha dejado algo más abierto como posible ampliación, por lo que se considera como uno de los posibles puntos para ampliar en el futuro. Por último, el objetivo de procesar los comandos de voz del usuario finalmente se ha alcanzado suficientemente: se estuvo bastante tiempo experimentando con los paquetes de procesamiento de audio HARK, pero finalmente se descartaron y se optó por los paquetes Sphinx de la Carnegie Mellon University. Esta librería ha dado un buen resultado y, una vez implementada, se consigue que pueda teleoperarse el robot de manera bastante fluida con la voz.

Como conclusión personal este proyecto ha sido un trabajo interesante y estimulante. Realizarlo me ha ayudado a consolidar conceptos nuevos sobre la robótica, y también me ha ayudado a conocer más sobre la comunidad Open Source de la robótica y aprender de ella. Me ha permitido aprender a usar el framework de ROS, que actualmente es el estándar Open Source de la comunidad robótica. También he tenido que estudiar el funcionamiento de la librería OpenNI, y he afianzado mis conocimientos de C++ y Linux.

Como trabajos futuros o ampliaciones se propone lo siguiente:

- Realizar una navegación de la base móvil teniendo en cuenta el entorno en el que se encuentra. Para ello se podría utilizar un LIDAR para obtener información del entorno, y algoritmos de SLAM para la localización y el mapping.
- Intentar hacer la misma navegación que en el apartado anterior, pero usando la nube de puntos 3D del sensor Kinect.
- Mejorar el sistema de aviso al servicio de urgencias, planteando una interfaz intermedia operada por una persona, que pueda atender las llamadas de emergencia del robot.
- Ampliar el robot con algunas funcionalidades propias de los robots de servicio. Por ejemplo, una pantalla encima del robot mejoraría la interfaz con los usuarios, proporcionando diferentes servicios como agendas que recuerden la medicación de cada usuario.
- Añadir un brazo robótico para realizar tareas simples, como poder abrir puertas para mejorar la navegación por interiores.
- Mejorar el sistema de detección de usuarios, con un sistema de identificación de usuarios mediante reconocimiento de caras o voz.
- Actualizar los paquetes creados a la última versión de ROS, que en este momento es ROS Indigo Igloo.

5. Referencias

1. http://envejecimiento.csic.es/documentos/documentos/pm-discapacidad-01.pdf
2. http://panasonic.co.jp/corp/news/official.data/data.dir/en110926-2/en110926-2.html
3. http://www.toyota-global.com/innovation/partner_robot/family_2.html
4. http://rtc.nagoya.riken.jp/RIBA/index-e.html
5. http://panasonic.co.jp/corp/news/official.data/data.dir/en110926-2/en110926-2.html
6. http://www.aal-domeo.eu/index.php/robots
7. http://www.cnrs.fr/cw/dossiers/doshand/decouvrir/sedeplacer/monimad.html
8. http://www.usc.edu/
9. http://www.osrfoundation.org/
10. http://opende.sourceforge.net/
11. http://bulletphysics.org/wordpress/
12. https://simtk.org/home/simbody/
13. http://dartsim.github.io/
14. http://www.ogre3d.org/
15. http://www.turtlebot.com/
16. http://www.xbox.com/es-ES/Kinect
17. http://ros.org/
18. http://wiki.ros.org/Robots/TurtleBot
19. https://www.willowgarage.com/turtlebot
20. http://www.irobot.com/global/es/
21. http://news.cnet.com/2100-1041_3-6147573.html
22. (Siegwart, Nourbakhsh, & Scaramuzza, 2011)Introduction to Autonomous Mobile Robots (2.3.1.2 Wheel geometry)
23. (Siegwart, Nourbakhsh, & Scaramuzza, 2011) Introduction to Autonomous Mobile Robots (pág. 137)
24. http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf

25. http://azttm.wordpress.com/2011/04/03/kinect-pattern-uncovered/
26. Ignacio Tortajada Montañana (2006). Expressió gràfica i infografia. Ed. Univ. Politéc. Valencia. pp. 264–. ISBN 9788483630310
27. http://research.microsoft.com/pubs/145347/bodypartrecognition.pdf , Shotton et al (CVPR 2011)
28. http://answers.ros.org/question/42654/openninite-incompatible-in-fuertegroovy-on-precise/
29. http://www.i-programmer.info/news/194-kinect/7004-openni-to-close-.html
30. http://cmusphinx.sourceforge.net/
31. http://winnie.kuis.kyoto-u.ac.jp/HARK/

Anexo 1. Archivo turtlebot_follow_user_commands.launch

```
<launch>

	<!-- para iniciar el robot comprobar que el tty serial tiene los permisos necesarios-->

	<include file="$(find turtlebot_bringup)/upstart/turtlebot.launch"/>


	<!-- openni drivers -->

	<include file="$(find openni_launch)/launch/openni.launch"/>


	<!-- mi openni tracker -->

	<node pkg="openni_tracker_follower " type="openni_tracker_follower " name="openni_tracker_follower " output="screen"/>


	<!-- Pocketsphinx voice recognizer -->

	<node name="recognizer" pkg="pocketsphinx" type="recognizer.py" output="screen">

		<param name="lm" value="$(find turtlebot_follow_user)/voice_cmd.lm"/>

		<param name="dict" value="$(find turtlebot_follow_user)/voice_cmd.dic"/>

	</node>


	<!-- el nodo que mueve el turtlebot -->

	<node pkg="turtlebot_follow_user" type="turtlebot_follow_user_commands" name="turtlebot_follow_user_commands" output="screen"/>


</launch>
```