# Introducción

Hoy en día, las investigaciones en el ámbito de la robótica están en auge, ya que se

pretende aumentar la calidad de vida del ser humano. Una de las aplicaciones que cobra

importancia en ese aspecto es el brazo robótico, cuya finalidad es manipular los objetos

que lo rodean.

# Entorno software

En esta sección se pretende tratar el entorno software del robot Clank, tratando toda su arquitectura de programación, desde las instrucciones de alto nivel hasta las órdenes de movimiento a los motores, pasando por el tratamiento de la información de sus sensores.

## Sistema Operativo Robótico.

ROS Desde un comienzo se pretendía desarrollar una programación modular, flexible y descentralizada. Es por ello que se ha apostado por usar el Robotic Operating System [10] como entorno de trabajo, en adelante ROS. ROS es un middleware que provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. Provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes y más. ROS está bajo la licencia open source.

A continuación se describen otras características de ROS:

* Liviano: ROS está diseñado para ser lo más ligero posible, esto es para que el código sea fácilmente transportable a otros sistemas o marcos de trabajo. El corolario de este punto sería que ROS es fácil de integrar con otros software de robots, de hecho, ROS ya ha sido implementado con Open Rave, Orocos y Player.
* Librerías transparentes: el modelo de desarrollo preferido es escribir librerías transparentes con interfaces limpias.
* Independencia del lenguaje: El marco de trabajo de ROS permite el desarrollo de programas en diferentes lenguajes de programación . Por el momento están hechas las implementaciones para Python, C++ y Lisp, además se tienen librerías experimentales para Java y Lua.
* Escalado: ROS es bastante adecuado para grandes funciones o para largos procesos de desarrollo.

Entre los objetivos principales de ROS destacamos que fue creado con la idea de reutilizar todo el desarrollo e investigación en robótica. ROS es un sistema distribuido que permite que los programas sean diseñados individualmente y que no sean pisados unos con otros en ejecución. Estos programas están agrupados en paquetes, que pueden ser fácilmente compartidos y distribuidos. ROS también soporta código de repositorios, que habilita a que la colaboración sea también distribuida. El diseño completo de ROS desde los ficheros de sistema hasta el nivel de comunidad permite la manipulación individual sobre desarrollo e implementaciones, pero todo ello se puede hacer a la vez con las herramientas de infraestructura que trae.

Actualmente ROS solo es compatible con plataformas basadas en Linux. El software desarrollado para Ros está principalmente probado sobre distribuciones de Ubuntu, y plataformas Mac OS X. Aunque la comunidad de ROS ha contribuido al soporte para Fedora, Gentoo, Arch Linux y otras plataformas Linux. Aunque se sabe que ROS podría ser compatible con Microsoft Windows, esta opción no ha sido totalmente explorada.

En concreto, la versión de ROS utilizada ha sido Kinetic Kame. En el ordenador portátil externo al robot el sistema operativo es Ubuntu 16.04 de escritorio (amd64), con lo que ROS está incluido en los repositorios oficiales y, por tanto, no hay necesidad de compilarle desde código, esta descentralización se explicará más adelante

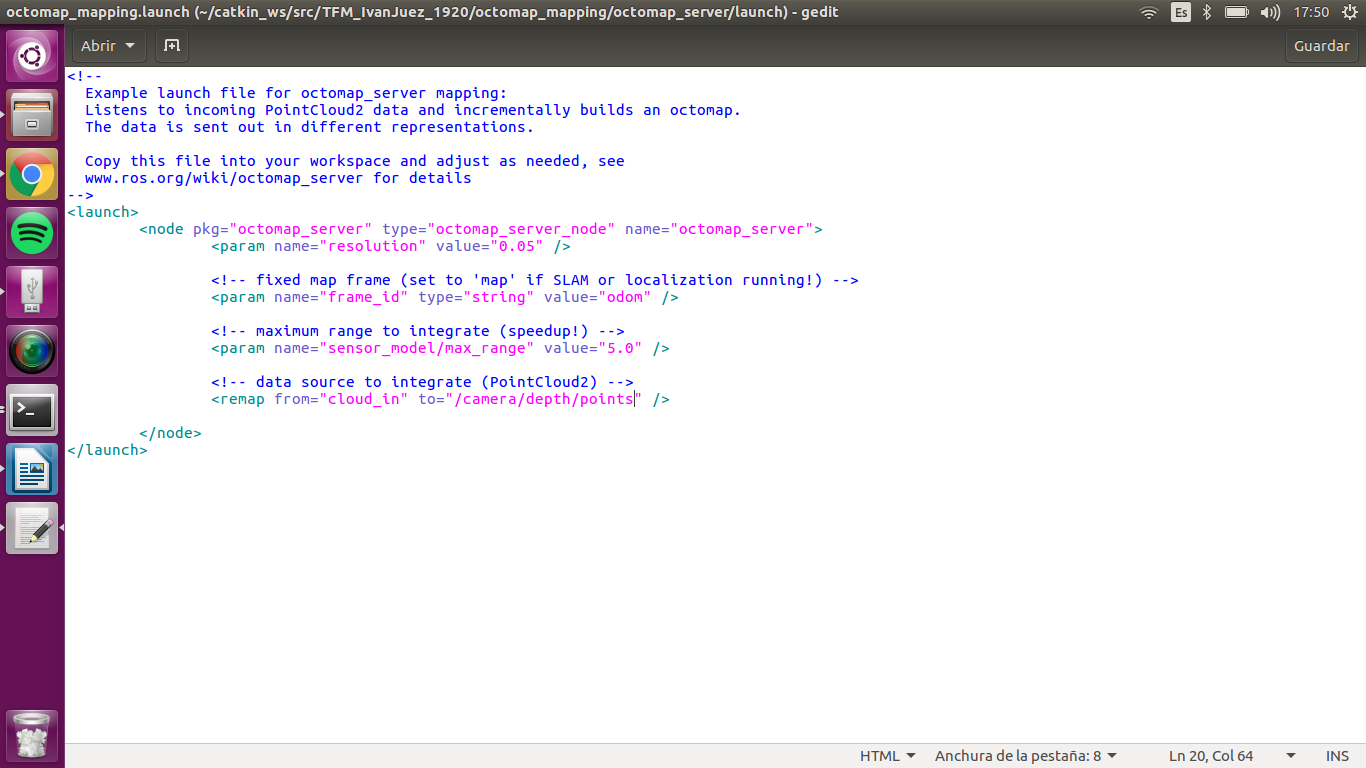
Los archivos ejecutables dentro de un proyecto en ROS pueden ser, hasta el momento, de dos tipos: con extensión .cpp (C++) o .py (Python) y reciben el nombre de nodos. EL nodo es la unidad básica tratamiento de información en ROS. Dichos nodos pueden comunicarse entre sí a través de lo que denominamos topics. Como cabe esperar, a cada topic le corresponde un tipo de mensaje, así por ejemplo, podemos tener topics que trabajen cadenas, enteros, estructuras, vectores, etc [5]. Para coordinar y sincronizar todos los nodos, sus topics etc, existe un nodo maestro que se puede ejecutar con el comando por consola «roscore».

# Detección automática de obstáculos en el entorno a través de la herramienta Octomap

Para poder realizar la detección automática de objetos en el entorno se hace a través de la herramienta Octomap.

(Descripción del Octomap)

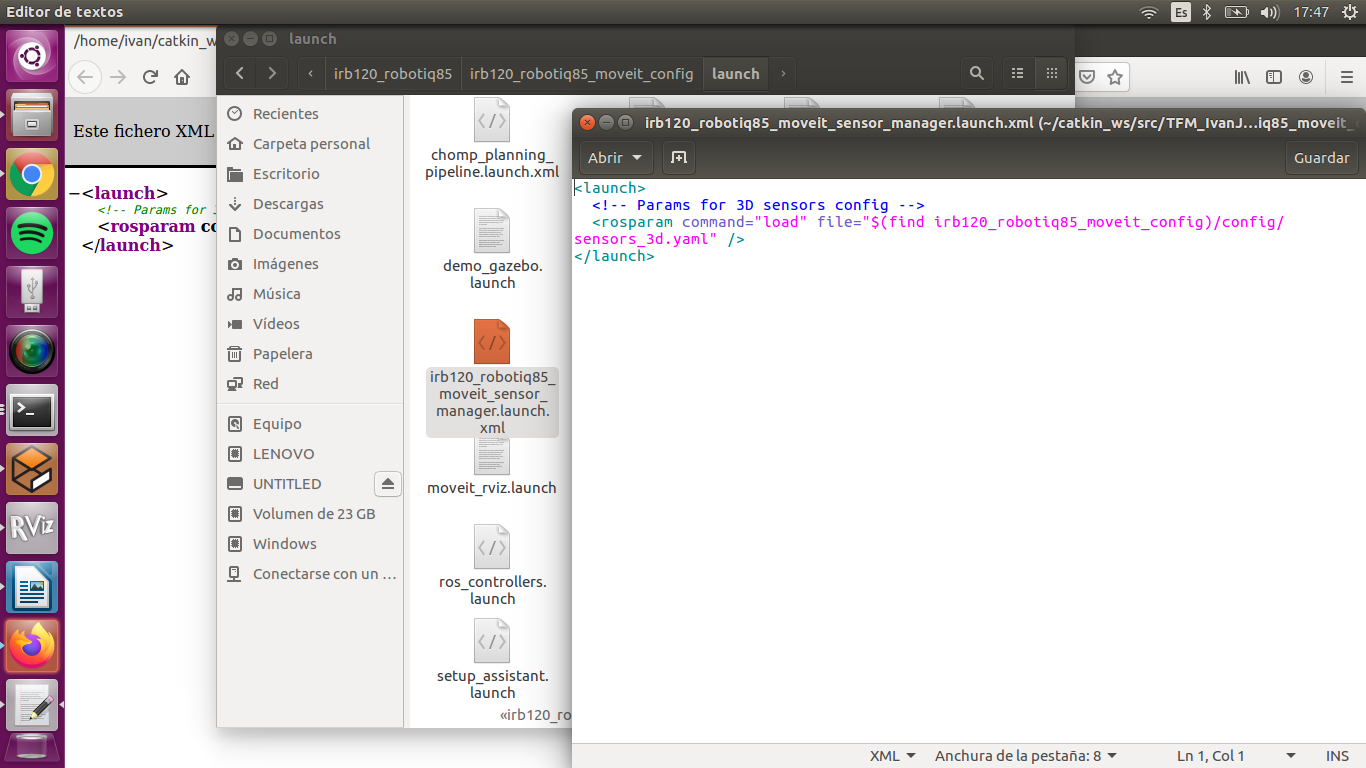
Para ello, es necesario modificar algunos parámetros del Octomap\_server, primero se modifica “fixed map frame” que es el frame donde queremos fijar el octomap, en nuestro caso será base\_link. Después es necesario cambiar el topic donde se almacenan las imágenes, (/camara/depth/points) que es el correspondiente al que genera la cámara Kinect y se puede modificar tanto la resolución como el rango de alcance:



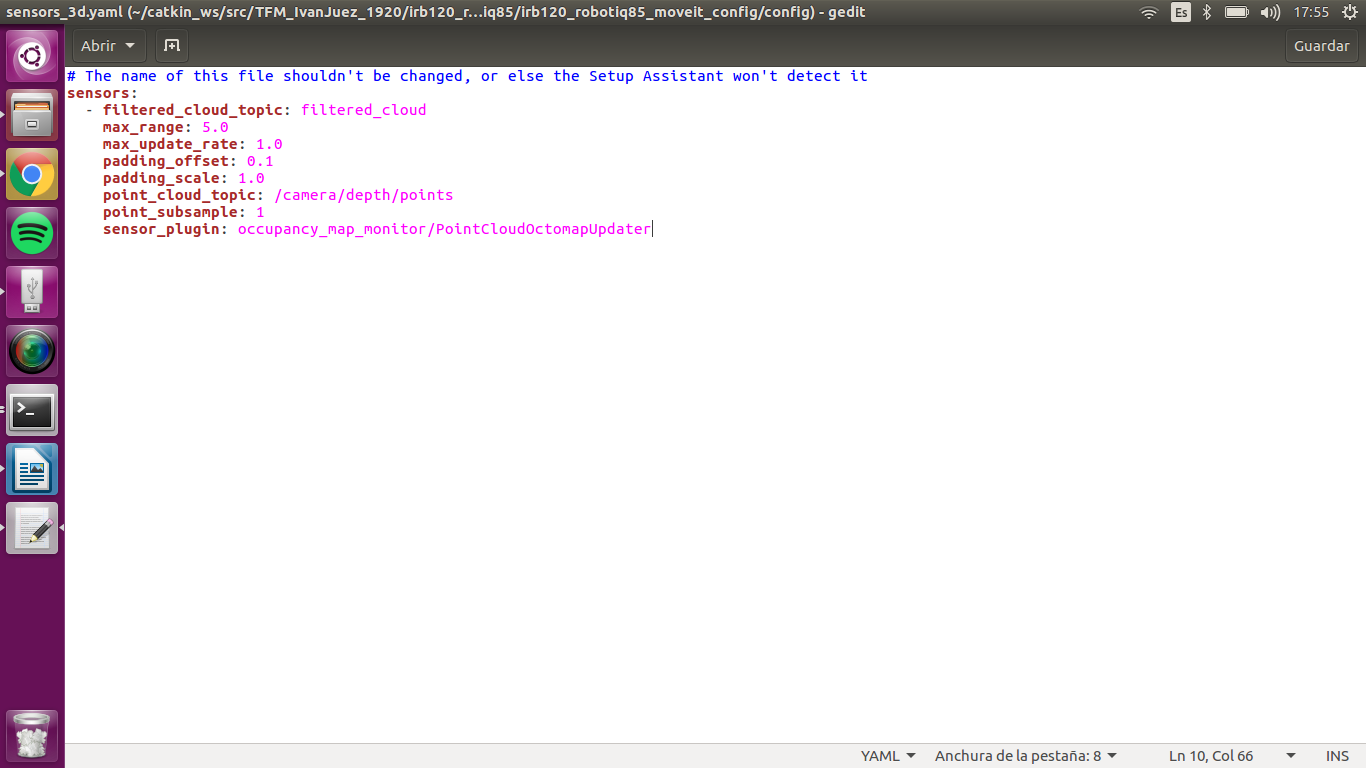
Por último, para que todo funcione se necesita modificar los siguientes ficheros dentro del directorio de irb120\_robotiq85\_moveit\_config:

Dentro del directorio launch:

* irb120\_robotiq85\_moveit\_sensor\_manager.launch.xml:



Dentro del directorio config:

* sensor\_3d.yaml
* 

# RECONOCIMIENTO DE POSICIÓN DE OBJETOS POR IMAGEN

Se estudiará el algoritmo que se aplica sobre la imagen obtenida de la cámara, el control efectuado y la salida de movimiento a los servomotores del robot ABB mediante interfaz hardware.

Según la fuente sensora se puede hacer seguimiento sobre una nube de puntos en 3D para, por ejemplo, obtener la posición del objeto que se tiene delante y así poder efectuar la acción del pick.

**OpenCv**

OpenCv 1 (Open Source Computer Vision) es una librería de visión por computador de código abierto. La librería está escrita en los lenguajes C y C++ y es compatible con Linux, Windows y Mac OS X. Cuenta con un desarrollo activo en interfaces para Python, Ruby, Matlab y otros lenguajes.

OpenCV ha sido diseñado para ser eficiente en cuanto a gasto de recursos computacionales y con un enfoque hacia las aplicaciones de tiempo real, además está escrito y optimizado en C y puede tomar ventaja de los procesadores con múltiples núcleos. La librería OpenCV contiene aproximadamente 500 funciones que abarcan muchas áreas de la visión artificial, incluyendo inspección de productos de fábricas, escaneo médico, seguridad, interfaces de usuario, calibración de cámaras, robótica....etc, porque la visión por computador y el aprendizaje automática van de la mano. OpenCV también tiene una completa librería de uso general de aprendizaje automático (MLL o Machine Learning Library), la cual es muy útil para cualquier problema de aprendizaje automático. Esta sublibrería está especializada en el reconocimiento estadístico de patrones y clustering.

**Módulos de OpenCV:**

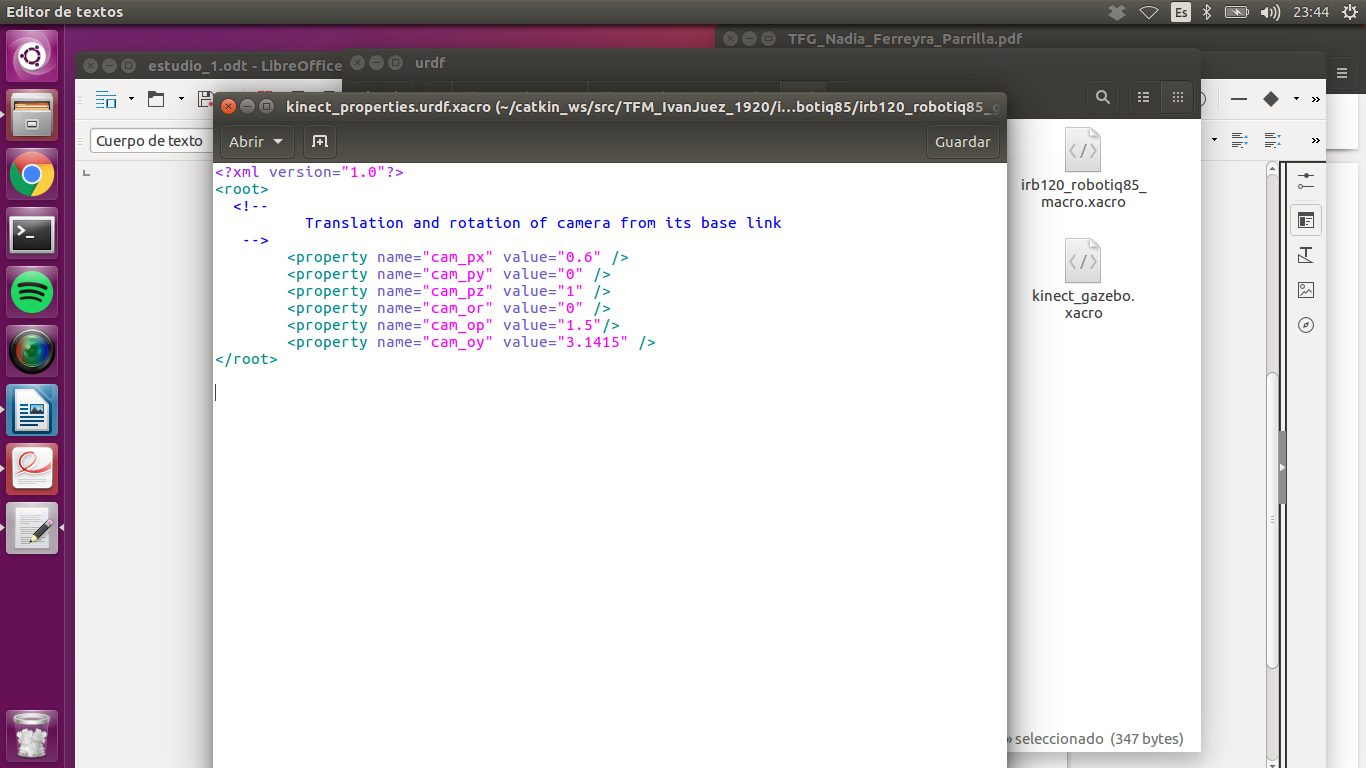
OpenCV tiene una estructura modular. Los módulos principales de OpenCV se listan a continuación :

* Core: Este es el módulo básico de OpenCV. Incluye las estructuras de datos básicas y las funciones básicas de procesamiento de imágenes. Este módulo también es usado por otros módulos como highgui.
* Highhui: Este módulo provee interfaz de usuario, códecs de imagen y vídeo y capacidad para capturar imágenes y vídeo, además de otras capacidades como la de capturar eventos del ratón... etc. Si necesitas capacidades de UI (User Interface) más avanzadas debes usar frameworks tales como Qt, WinFormsetc.
* Imgproc: Este módulo incluye algoritmos básicos de procesado de imágenes, incluyendo filtrado de imágenes, transformado de imágenesetc.
* Video: Este módulo de análisis de vídeo incluye algoritmos de seguimiento de objetos, entre otros.
* Objdetect: Incluye algoritmos de detección y reconocimiento de objetos para objetos estándar.

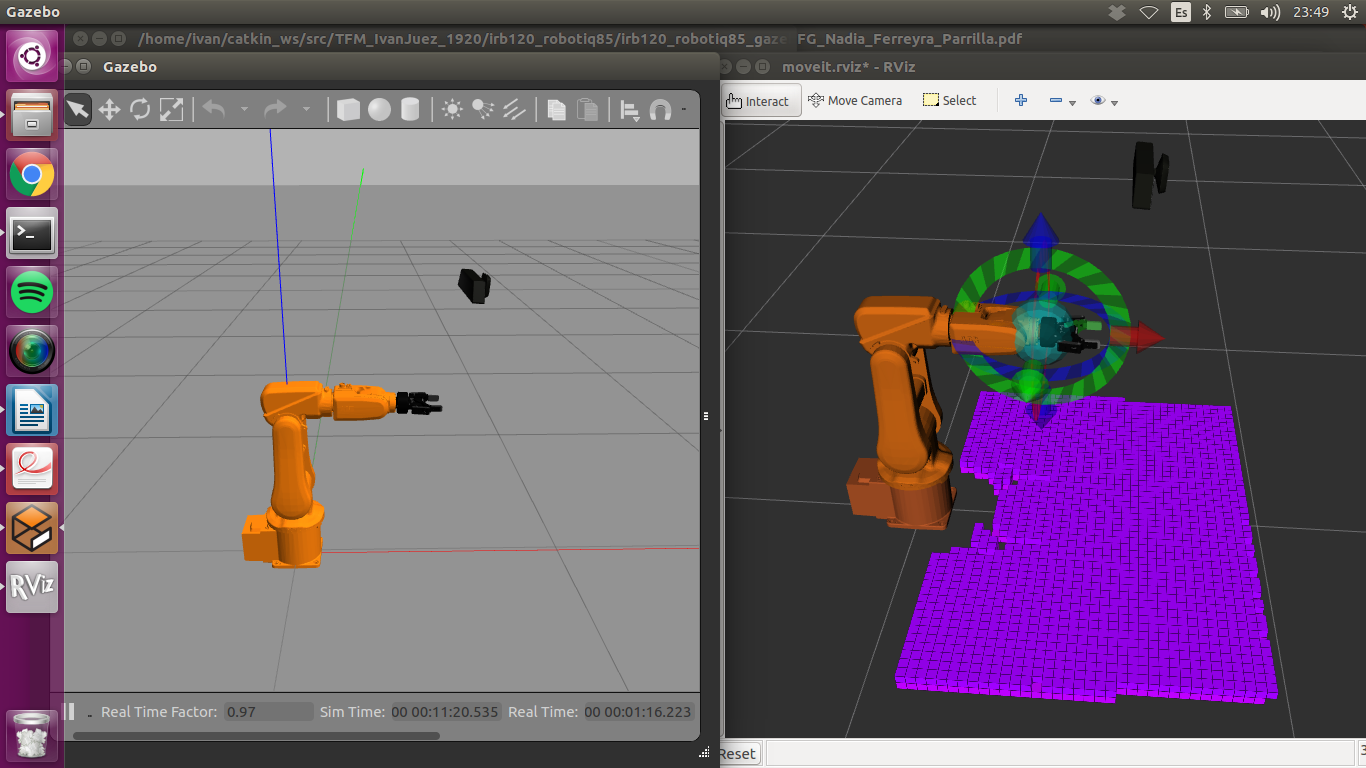
En este proyecto nos vamos a centrar en el módulo de Objdetect, con el fin de encontrar la mejor alternativa de reconocimiento del objeto y su posición se plantean diferentes escenarios de estudio donde finalmente se elegirá el mejor.

# Escenario.1 : Se situá la cámara Kinect en posición cenital para detectar objetos.

Con el fin de encontrar el mejor escenario para poder detectar automáticamente los objetos para poder realizar la acción de pick and place , se ha localizado la cámara Kinect en posición cenital. Para poder realizar este cambio, se modifican los valores asignados dentro del archivo ***kinect\_properties.urdf.xacro :***

  
Ilustración 1: Parámetros de posición y orientación Kinect

De tal forma, que a la hora de lanzar el entorno de simulación Gazebo y Rviz se obtiene lo siguiente:

  
Ilustración 2: Posición Kinect en Gazebo y Rviz

Una vez tenemos la cámara correctamente, vamos a proceder a evaluar la detección de la posición de diferentes objetos con diferentes posiciones y tamaños.

## Esferas de 2 y 3 cm de radio

Primero se va a realizar el estudio con esferas de un radio de 2 y 3 cm situadas en diferentes posiciones, a continuación se muestra la escena simulada:

Captura de pantalla de una computadora

Descripción generada automáticamente

bola\_peq

bola\_peq\_0

bola\_peq\_1

bola\_grd\_0

Captura de pantalla de una computadora

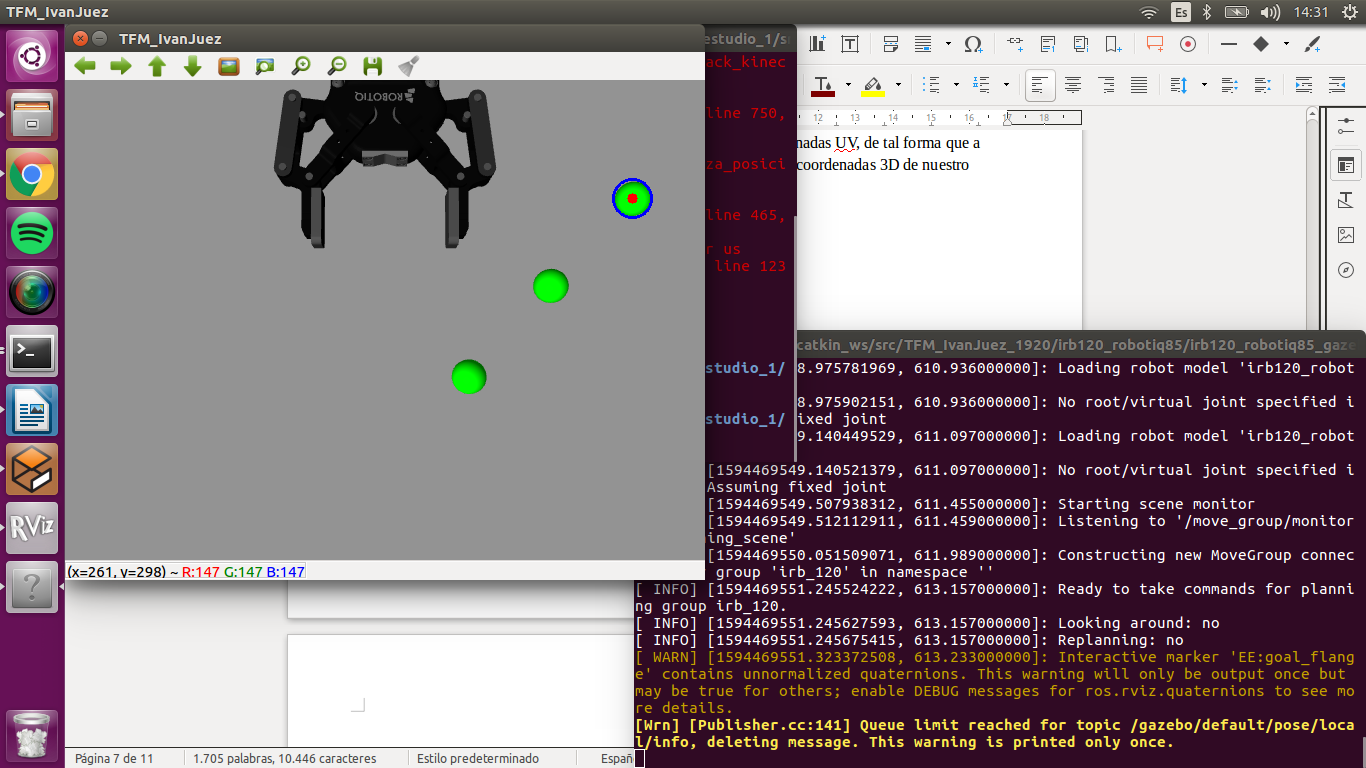
Descripción generada automáticamente

bola\_grd

bola\_grd\_1

De tal forma que se van a obtener las coordenadas a través de dos vías:

* Gazebo: En este caso a través de un servicio de Gazebo *“get\_model\_states”* se obtienen directamente la posición y orientación de los objetos simulados.
* Kinect: En este caso, se ha realizado un procesado de la imagen que consiste en un filtro de color, verde, mediante el cual detecta el objeto y calcula su contorno. Con la cámara se consigue hacer un mapeado UV, que consiste en asignar a cada vértice del polígono que se encuentre en la imagen a un par de coordenadas 2D denominadas UV, de tal forma que a traves de la librería de ROS TF conseguimos trasladar a las coordenadas 3D de nuestro robot *(XYZ).*

  
Ilustración 3: Imagen realizada con Kinect de 3 esferas grandes

Tras realizar esta comparativa se han obtenido los siguientes datos:

| Objeto | Coordenadas Gazebo | Coordenadas Kinect |
| --- | --- | --- |
| bola\_peq |  |  |
| bola\_peq\_1 |  |  |
| bola\_peq\_2 |  |  |
| bola\_grd |  |  |
| bola\_grd\_1 |  |  |
| bola\_grd\_2 |  |  |

## Cajas de 3 y 6 cm de lado

Imagen que contiene interior, electrónica, computadora, monitor

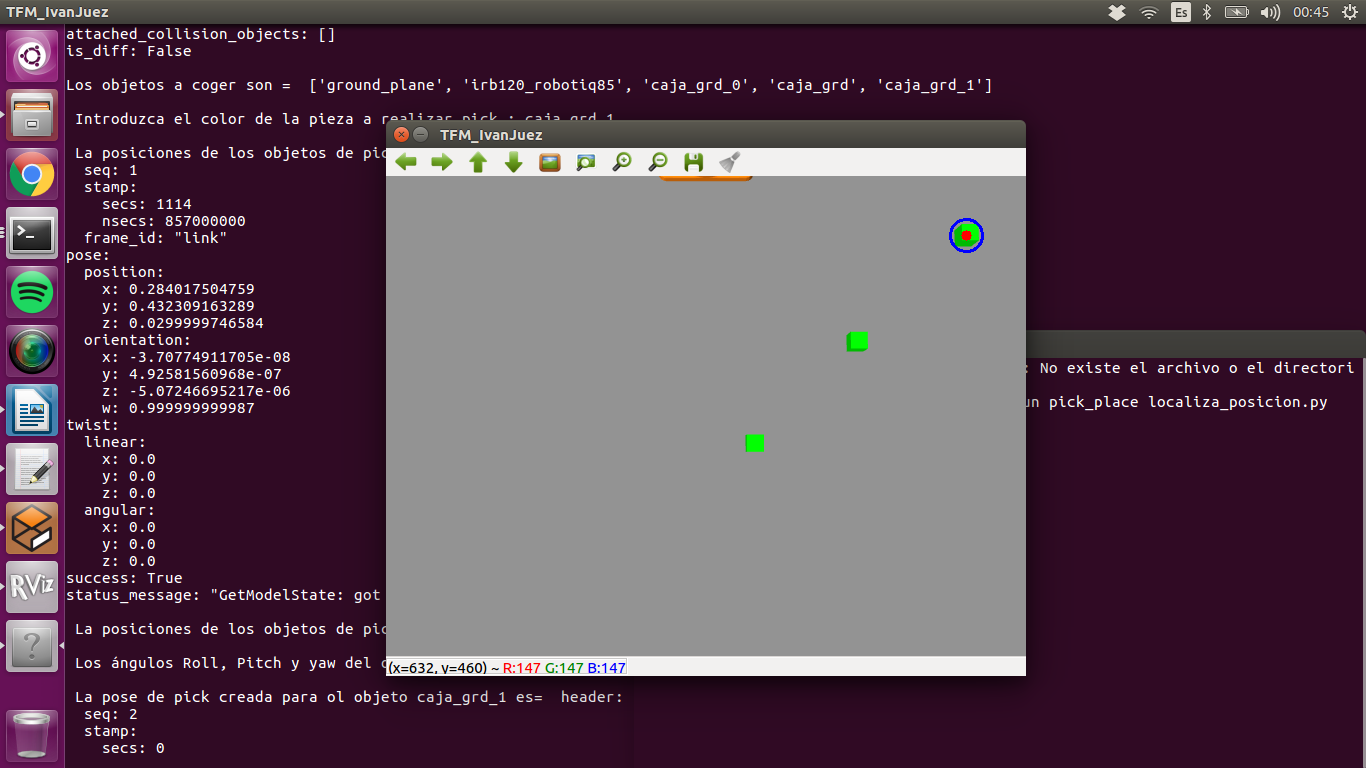
Descripción generada automáticamenteLuego se va a realizar el estudio con cajas de un lado igual a 3 y 6 cm situadas en diferentes posiciones con el fin de averiguar como afecta la forma y tamaño de los objetos en la detección de su posición, a continuación se muestra la escena simulada:

Pantalla de juego de computadora

Descripción generada automáticamente

De tal forma que se van a obtener las coordenadas a través de dos vías:

* Gazebo: En este caso a través de un servicio de Gazebo *“get\_model\_states”* se obtienen directamente la posición y orientación de los objetos simulados.
* Kinect: En este caso, se ha realizado un procesado de la imagen que consiste en un filtro de color, verde, mediante el cual detecta el objeto y calcula su contorno. Con la cámara se consigue hacer un mapeado UV, que consiste en asignar a cada vértice del polígono que se encuentre en la imagen a un par de coordenadas 2D denominadas UV, de tal forma que a traves de la librería de ROS TF conseguimos trasladar a las coordenadas 3D de nuestro robot *(XYZ).*

  
Ilustración 4: imagen obtenida de Kinect para reconocer posición

Tras realizar esta comparativa se han obtenido los siguientes datos:

| Objeto | Posición en el frame | Coordenadas Gazebo | Coordenadas Kinect |
| --- | --- | --- | --- |
| caja\_peq |  |  |  |
| caja\_peq\_1 |  |  |  |
| caja\_peq\_2 |  |  |  |
| caja\_grd |  |  |  |
| caja\_grd\_1 |  |  |  |
| caja\_grd\_2 |  |  |  |

**Análisis de datos**

En este punto, se van a buscar la tendencia del fallo obtenida tanto por forma como por tamaño de los objetos, para encontrar si la desviación entre las dos formas de obtener los datos es constante o si sigue algún tipo de tendencia:

(gráficas de excell)

**Conclusión del estudio**