Treball de Fi de Grau

Master's in Automatic Control and Robotics (MUAR)

Development of Perception Module for Robotic Manipulation Tasks

MEMÒRIA

27th January, 2022

Autor: Nikola Gazikalović

Director: Prof. Dr. Jan Rosell Gratacòs

Convocatòria: 01/2022

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC

Abstract

Robots performing manipulation tasks require the accurate location and orientation of an object in space. Previously, at the Robotics Laboratory of IOC-UPC this data has been generated artificially. In order to automate the process, a perception module has been developed for providing task and motion planners with the localization and pose estimation of objects used in robot manipulation tasks. The Robot Operating System provided a great framework for incorporating vision provided by *Microsoft Kinect V2* sensors and the presentation of obtained data to be used in the generation of Planning Domain Definition Language files, which define a robots environment. Localization and pose estimation was done using fiducial markers along with studying possible enhancements using deep learning methods. Perfectly calibrating hardware and setting up a system play a big role in enhancing perception accuracy and while fiducial markers provide a simple and robust solution in laboratory conditions, real world applications with varying lighting, viewing angles and partial occlusions should rely on AI vision.

Contents

List of Figures

List of Tables

1 Introduction

The Robotics Lab at the Institute of Industrial and Control Engineering (IOC-UPC) currently, amongst others, operates the follwing robots:

- *YuMi*, a dual arm collaborative robot made by *ABB* and intended for use in manufacturing environments where humans and cobots can work with direct interaction or in close proximity. The safety of a mutual working environment is provided by torque sensors in *YuMi*'s joints and software which don't allow it to exceed certain forces, lightweight materials and rounded edges. At the IOC *YuMi* is placed on a fixed table and perform various object manipulation tasks using its 2 arm manipulators, each one having 7 axis Figure 1.1b.

- *TiaGo* robot manufactured by *PAL Robotics* which has a mobile base, an extendable torso and one arm manipulator. Its sensor suite consisting of a 270 degree lidar, RGB-D camera etc. allow it to perform a wide array of perception, manipulation and navigation tasks. In the laboratory it is used for mapping and various manipulation tasks including object transportation between work areas Figure 1.1a.

- Madar, mobile dual-arm robot manipulator, it consists of an advanced omni-directional mobile base developed at UPC and two UR5 manipulators from Universal Robots on a steel frame along with batteries, an on-board computer and an RGB-D camera for perception Figure 1.1c.

The increasing complexity of everyday manipulation tasks necessitates the development of a robot that is competent, reliable, and autonomous in order to perform a variety of manipulation operations in industrial and free-world situations. To solve complex manipulation challenges, robotic systems must use Task and Motion Planning (TAMP), which identifies a distinct series of symbolic actions and a motion plan solution for each. Task planning alone is unaware of the geometric limits imposed by the environment but it works effectively to generate a symbolic plan for vast state spaces. In order to complete it, motion planning is concerned with determining motion solutions or, in the event of a failure, reporting geometric restrictions for each given task [6]. As a result, a mix of task and motion planning is essential for successfully guiding robots toward optimal manipulation solutions. If an assumption is made that the integration of task and motion planning is perfect, with one of the main challenges becomes to develop an effective method of interacting and exchanging information with the robots surroundings, in particular obtaining accurate locations and orientations of object which will be included in manipulation tasks [26].

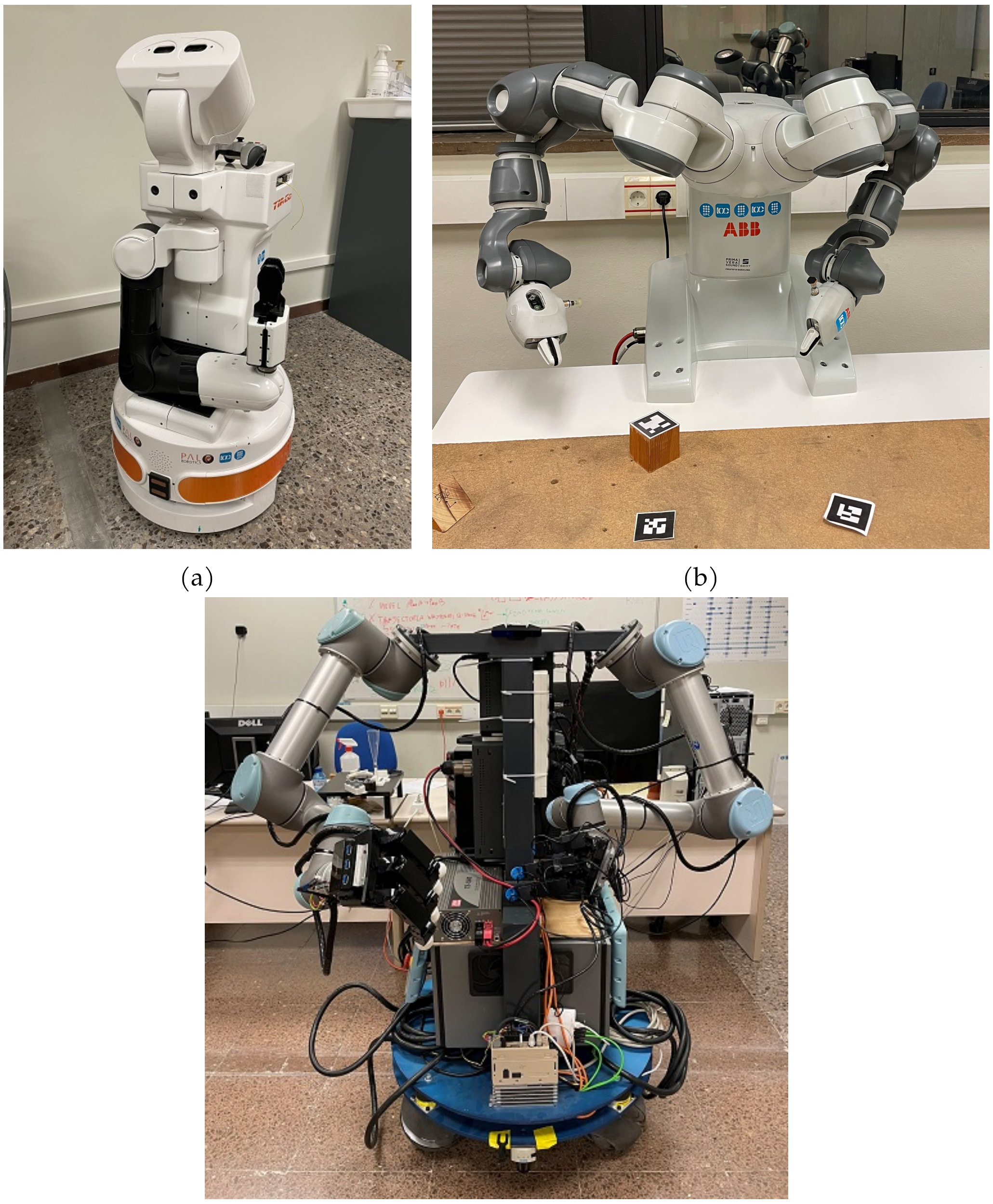


Figure 1.1: Available robots at the IOC robotic lab.

Typically, the task planning domain is described by the Planning Domain Definition Language (PDDL), which is used to standardize the configuration of AI task and motion planning tasks. PDDL files are used to define the world environment, properties of objects, initial states and goals along with ways of changing the world. Objects, Predicates, Initial-State, Goal-Specification, and Actions are the primary components of PDDL. In the planning world, objects are things. Predicates are the true or untrue characteristics of things in the world. The initial-state of the term is the state in which planning starts. The term "Goal-Specification" refers to the objective that must be accomplished via planning. Actions are the means through which the world's status may be altered. In PDDL, planning tasks are organized in two files: a domain file that contains Predicates and Actions, and a problem file that contains Objects, Initial-State, and Goal-Specification [11].

While all of these robots are designed to work in a dynamic environment with some of them having their own perception modules, currently at the IOC lab task planning domain and problem files are designed manually with respect to a task description. This means that the planner is called once at the beginning of execution and the PDDL files stay the same throughout one task implementation.

In order to incorporate a dynamic state environment, change or add tasks and allow the robot to re-generate PDDL files on the go, a perception module is to be developed. The mission of this module will be to provide information regarding the environment, object states, robot localization and to fill in artificially created segments of the PDDL files. One important thing to note is that Lab 2 at the IOC is designed to be a large shared workspace between robots and humans, and thus with people walking freely in the working area and possibly interacting with robots, caution has to be taken to insure that robots are aware of human presence as well as the location and orientation of objects.

1.1 Objectives

The main objective of this project is to develop a perception module to collect information about the robot's initial state, its surroundings and states of objects of interest and present them in the best possible way within the ROS framework, it can be divided into 2 main groups:

- Robot localization with respect to a global fixed reference frame
- Identification and object pose estimation for the objects of interest

The whole system has to be made modular in order to allow addition of more cameras, advanced machine learning algorithms which would provide various data and other sensors. Furthermore, documentation with instructions on use and possible expansion has to be provided.

The secondary objectives is the exploration of deep learning techniques for:

- Detection of humans and congestion levels currently in the working area using an AI vision system
- The detection of a 6-DOF pose estimation of objects

This study aims to provide the motion task planner with all the necessary data regarding a robots environment which would allow it to continuously generate PDDL files and execute tasks without the need for artificial data to be added in order to describe the robot environment. Firstly, enhancement of current methods which use fiducial markers will be undergone, along with developing the general framework of the module and subsequently deep learning algorithms will be implemented to enhance the 6-DOF object pose estimation and to determine laboratory congestion levels. The goal is for the ROS framework to incorporate and control all segments of the robotics lab with the perception module being only one part. It presents the ideal structure to implement separate modules to which all agents could have access.

All of the data will be collected using already available cameras and sensors within the lab.

It is assumed that only one camera will be used for estimating each object pose and different cameras will not have overlapping fields of view which could cause singularities at certain angles.

2 Literature review

2.1 6 Degree of freedom pose estimation

Visual tracking is often restricted in range, prone to mistakes, takes a long time to analyze, and exhibits faulty behavior as a result of mathematical instability [15]. To overcome these flaws, a rigorous methodology is necessary.

The objective of the 6D pose estimation job is to determine an object's rotation and translation in relation to a known coordinate frame (e.g. a robot sensor frame). Because rotation and translation each have three degrees of freedom, the state of a rigid object may be completely represented by six values in the six-dimensional pose representation. This provides robot agents with a compact knowledge of the objects in their surroundings, and 6D posture estimation is critical for a variety of robot manipulation tasks, where pose-relative actions may be performed based on the object's stance.

With the fast advancement of deep learning techniques over the last several years, many deep networks have been developed to predict the posture of 6D objects using RGB or RGB-D observations. Unlike object identification and segmentation, however, 6D posture estimation is an area where traditional approaches based on surface matching remain competitive. And the majority of deep techniques have a limited capacity to generalize to new objects [10].

2.1.1 Pose estimation using fiducial markers

Among the approaches utilized, vision-based techniques provide benefits for augmented reality applications since their registration may be very precise and there is no latency between the movements of actual and virtual scenes. These techniques, however, suffer from a high processing cost and a lack of resilience. To overcome these limitations, one of the solutions is a robust camera pose estimate approach based on monitoring calibrated fiducials in a known three-dimensional environment. The camera location is dynamically calculated using the Orthogonal Iteration Algorithm. [12].

Fiducial markers are used to create a visual reference point inside a scene. They're simple to construct and simple to utilize. When implemented properly, retrieving these markers from a scene may assist in camera calibration, localization, tracking, mapping, and object recognition. These uses are made feasible by the camera geometry [20].

While conventional Fiducial markers provide very precise findings, they may not always operate well in real contexts. These natural settings specify the conditions under which the pose estimation method should function. Possible sources of interference in detection and classification applications include scaling, possible occlusions between the objects, motion blur caused by a non-fixed camera or moving objects, and off-axis viewing, which is defined as viewing a display from an angle greater than one degree away from the center.

Many fiducial marker libraries exist like *ARTag*, *AprilTag*, *ArUco* and *STag* [7], varying in their design, size and robustness, but in the lab at IOC-UPC, currently *ArUco* tags are in use and they will be used in the scope of this project.

2.1.2 Deep learning pose estimation methods

Deep Learning object pose estimation methods rely on large sets of labeled data and 3D computer generated models for objects within those datasets. Unlike conventional methods which locate a known 2D fiducial marker and using the marker size and description along with knowing the intrinsic camera parameters artificially generate a 3D marker frame, deep learning methods provide a 3D bounding box for detected objects and its orientation in space [28].

Using synthetic data for training deep neural networks for robotic manipulation holds the promise of an almost unlimited amount of pre-labeled training data, generated safely out of harm's way. One of the key challenges of synthetic data, to date, has been to bridge the so-called reality gap, so that networks trained on synthetic data operate correctly when exposed to real-world data [29].

A significant technological issue in estimating 6D object posture from RGB-D images is maximizing the use of the two complementing data streams, RGB images and depth fields [5]. Previously published work either extracted RGB and depth information independently or used expensive post-processing procedures, limiting their performance in extremely congested environments and real-time applications.

Although many algorithms exist, trying to bridge the fore mentioned issues, by finding new ways to integrate RGB-D data and create realistic artificial data, one framework has been developed by *Google* called *Mediapipe* which relies solely on RGB data and uses a large set of real data [4].

2.2 MediaPipe

MediaPipe is a Google-developed open-source framework for building multimodal (video, audio, or any time series data) and cross-platform (i.e. Android, iOS, web, and edge devices) machine learning pipelines. It has been tuned for performance with end-to-end on-device inference in mind. Mediapipe is currently in active development and includes a number of demos that can be run immediately upon installation. [25]

While OAK-D uses the *Intel Myriad X* visual processing unit for inference, in order to comply with space limitations, *MediaPipe* models are made to be run on either a computer CPU or on a dedicated GPU for maximum performance. In the case of this project, testing was performed on a student laptop CPU in a *PyCharm* environment while the main implementation is to be on a PC in the lab running Debian OS and using the *Nvidia RTX 2060* graphics processing unit for best performance.

The most significant distinction between the *MediaPipe* and *OAK-D* frameworks is as follows:

1. OAK-D models are developed by training deep learning neural network while MediaPipe uses manually annotated data to train conventional machine learning models.

2. OAK-D uses depth clouds along with RGB images to train their models which are thus more computationally expensive to run in real time, on the other hand MediaPipe uses only precisely labeled RGB data [17]

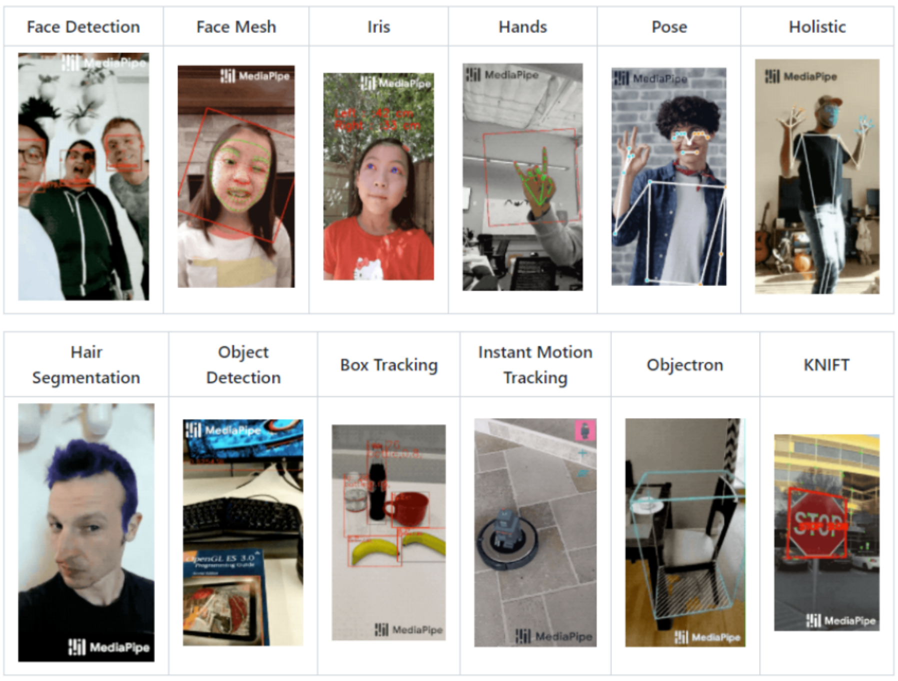


Figure 2.1: Trained Mediapipe models

Some of the available pre-trained models for various applications can be seen in Figure 2.1.Firstly, *MediaPipe Hands* was selected as the model which would be the most useful and innovative for passing inputs to agents within the laboratory using the perception module. It could be a basis for hand gesture control and a form of human-robot interaction and teleoperation as well as the overlay of digital material and information over the actual environment in augmented reality [16]. Moreover, if the algorithm could sense the position and movement of a human hand, an assumption was made that it could be able to preform equally on humanoid robot end effectors. One of the possible doubts towards robustness of this model was that it is a particularly difficult computer vision problem, since hands often occlude themselves or one other (e.g., finger/palm occlusions and hand shaking) and lack high contrast patterns, also there is a difference in skin tone between humans and obvious differences when comparing palms of a human and a humanoid robot. *MediaPipe Hands* is a solution for high-fidelity hand and finger tracking. It uses machine learning (ML) to predict a hand's 21 3D landmarks from a single shot. Whereas existing state-of-the-art systems depend heavily on sophisticated desktop environments for inference, this solution operates in real time on a mobile phone and even scalable to many hands [31].

MediaPipe Hands makes use of a machine learning pipeline comprised of numerous models cooperating:

- A palm detection model that acts on the whole picture and produces a bounding box for an orientated hand.
- A hand landmark model that acts on the palm detector's cropped picture area and provides highly accurate 3D hand feature points.

When the palm detection model finds bounding boxes (anchors) around hands in its field of view, the hand landmark model is deployed which conducts exact keypoint localization of 21 three-dimensional hand-knuckle coordinates inside the observed hand areas by regression, which is a type of direct coordinate prediction. The model acquires a consistent internal representation of hand poses and is hence resilient to partly visible hands and self-occlusions. Figure 2.2 shows hand landmarks used in inference, which were also previously manually annotated on a large number of images in order to have a labeled data set for supervised learning [22].

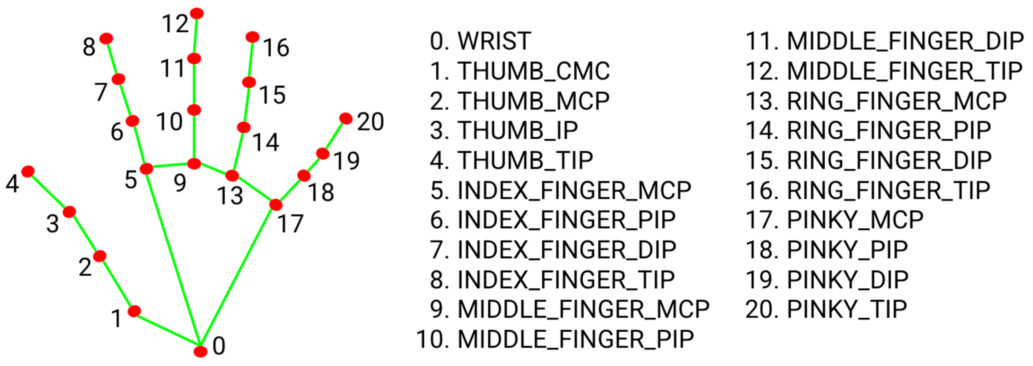


Figure 2.2: MediaPipe Hands hand landmarks

3 Hardware and Software Used

3.1 Hardware

A visual AI system's capacity to make judgments based on two factors is known as spatial AI.

- Visual Perception refers to an AI's ability to visually "see" and "understand" its surroundings. A camera attached to a processor running a neural network for object detection, for example, can detect a cat, a person, or an automobile in the scene the camera is viewing.
- Depth Perception is the AI's capacity to determine how far away objects are. Human eyesight inspired the concept of Spatial AI. We perceive our surroundings with our eyes. Furthermore, people use two eyes (i.e. stereo vision) to judge distances.

Depth perception is obtained using stereo camera pairs, infra red sensors and lidar, one of the most affirmed RGB-D cameras is the *Microsoft Kinect* which has initially been launched more than 10 years ago while one of the most modern is the *Luxonis OAK-D* sensor, featuring a state of the art vision processing unit. These two cameras have been selected to be used in the perception module framework and while their main benefit is the ability to obtain depth fields, their RGB cameras also make them great tools for working with traditional 2D images.

3.1.1 Microsoft Kinect 2

Microsoft Kinect was first designed as a motion controller accessory for *Xbox* video game systems [32], differing from rivals (such as Nintendo's Wii Remote and Sony's PlayStation Move) in that it did not need physical controllers. The devices initially included RGB cameras, infrared projectors, and detectors that map depth using structured light or time of flight calculations, allowing for real-time gesture recognition and body skeleton identification, among other capabilities. Additionally, they had microphones for speech recognition and voice control [9].

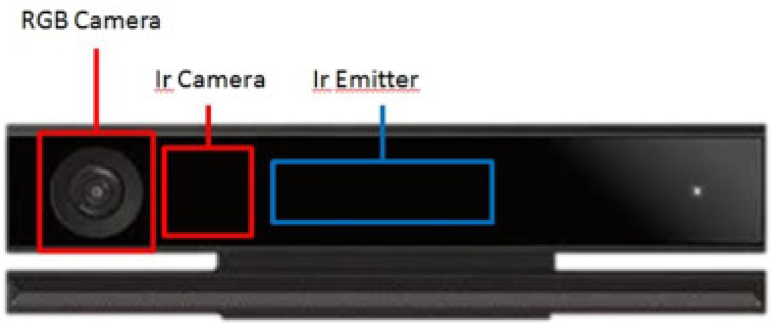


Figure 3.1: Kinect V2 sensor.

The Microsoft Kinect V2 is a 3D sensor that consists of an RGB camera with a resolution of 1920x1080 pixels, an infrared camera with a resolution of 512x424 pixels, and an infrared emitter.

The on-board electronics calculate the distance between the sensor and each point in the scene observed by the camera using Time of Flight (ToF) technology and, in particular, the intensity modulation approach [1]. The precision with which such distances are determined is dependent on the distance itself and may reach values of around 1.5 mm when the point is near to the sensor (about 1 m).

The *Kinect* sensor can be easily integrated with the Robot Operating System environment by using the *IAI Kinect2* package [30], developed by Wiedemeyer, Thiemo, which interprets data obtained from the *Kinect* on-board computer and publishes it in ROS, including the camera view and depth fields.

3.1.2 OAK-D

A company called *Luxonis* recently launched an OpenCV AI kit which includes one of the available cameras, at the IOC lab OAK-D is used, and a built in visual processing unit which when combined with their open source github repository become a very useful tool. At a high level the OAK-D consist of the following important components:

- 4K RGB camera placed at the center and used mainly for visual perception
- Stereo camera pair used for depth perception
- *Intel Myriad X* visual processing unit (VPU) which is essentially the brain of the modules. It is a powerful processing unit designed specifically for running advanced neural network models for visual perception along with creating a depth field using raw data obtained from the stereo pair cameras in real time.

The best thing about utilizing an OAK-D or OAK-D Lite is that it doesn't require any external gear or software. It has a seamless experience thanks to the integration of hardware, firmware, and software. The API (Application Programming Interface) used to program the OAK-D is called Depth-AI. It's cross-platform, so it doesn't matter what operating system is installed. [23] On the other hand, its main competitor, the *Intel Realsense Depth Camera D435* works only as a sensor, requires at dedicated graphical processing unit along with least a mini-PC (ex. Raspberry PI) in order to process the data and doesn't have as many specific ready to use applications.

Unlike other depth cameras that utilize structured light (projecting a grid onto the subject to observe how it deforms) or time of flight (measuring the time it takes light to travel to and from the subject), the Oak-D operates similarly to the way our brain does by measuring the offset from our eyes.

3.2 Software

3.2.1 ROS

Robot Operating System is an open-source middleware suite for robots. It is not a complete operating system but rather a collection of software packages for developing robot applications, it provides services for a heterogeneous computer cluster, including hardware abstraction, low-level device control, implementation of commonly used functionality, message passing between processes, and package management. A graph architecture is used to describe running sets of ROS-based processes. Processing occurs in nodes that may receive, post, and multiplex sensor data, control, status, planning, actuator, and other signals [13]. Despite the critical importance of responsiveness and low latency in robot control, ROS is not a real-time operating system (RTOS). However, it is feasible to connect ROS with real-time programming. The development of ROS 2 focused on overhauling the ROS API that would use contemporary libraries and technologies for basic ROS functionality and include support for real-time code and embedded hardware, although it still doesn't have nearly as many packages as the original, more and more developers are contributing and it is set to replace ROS by 2025.

Main components of a ROS framework are:

- **Nodes** represent single processes running on the ROS graph, every node has a name and is registered with the ROS master before it can start performing actions. Nodes with the same names can exist within different namespaces (domains). Nodes are the center of the ROS framework, they are able to send and receive data, subscribe and publish data to topics, receive and send requests for services and actions.

- **Topics** are referred to as buses because they are the conduits via which nodes transmit and receive messages. Additionally, topic names must be unique inside their namespace. A node must publish to a topic in order to send messages to it, whereas it must subscribe in order to receive messages. The sorts of messages that may be sent on a topic are many and can be set by the user. These messages may include sensor data, motor control directives, condition information, or actuator commands, among other things.

- **Services** are operations that a node may do which produces a single outcome. As such, services are often used for tasks with a specified start and finish point, such as shooting a single-frame photograph [3], rather than processing velocity instructions to a wheel motor or odometer data from a wheel encoder. Nodes advertise and connect to one another's services.

- ROS *tf* is a package that enables the user to maintain a history of numerous coordinate frames. tf stores the connection between coordinate frames in a time-buffering tree structure and enables the user to convert points, vectors, and other objects between any two coordinate frames at any desired time point. *tf* messages have the format of translation in x,y,z coordinates and rotation specified as a quaternion or as roll, pitch and yaw.

In order to incorporate *Kinect2* and *OAK-D* sensors into the ROS environment, ROS bridges needed to be used for each of them. A ROS bridge enables two-way communication between ROS and an external device. The information from the sensors is translated to ROS topics. In the same way, the messages sent between nodes in ROS get translated to commands to be applied in one of the sensors [30].

3.2.2 Aruco ROS and Aruco Broadcaster

In order to be able to work with *Aruco* tags in ROS it was decided to use a package developed by *Pal Robotics* called *Aruco ROS* [21]. Specifically one part of the package was critical, *Aruco marker publisher* identifies fiducial markers of a certain library and specification and publishes related data in a set of ROS topics. It takes the *Aruco* library, marker dimensions, and raw image from camera as inputs while, for this project, the following output data was found the most useful:

- List of identified markers

- Location of markers w.r.t. camera

- Markers orientation in space

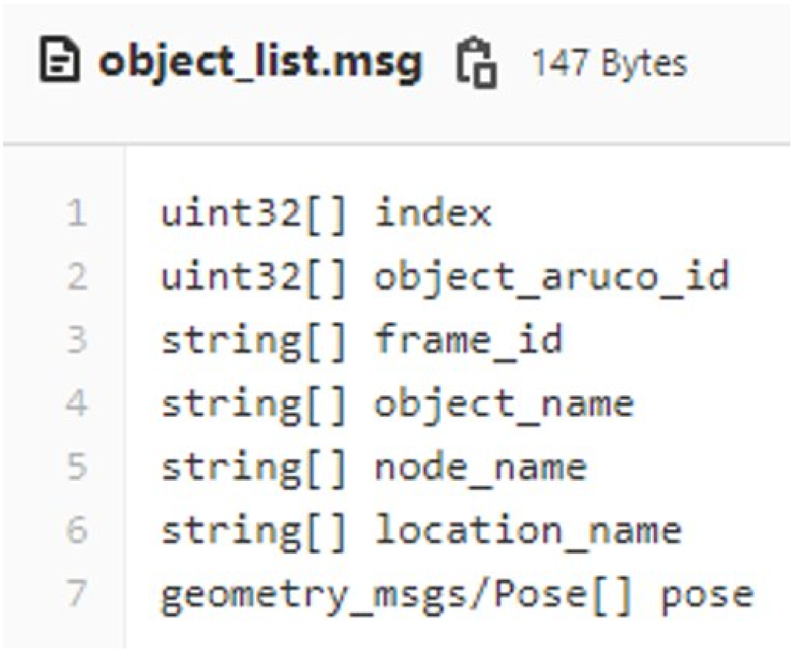
- Resulting image after superimposing marker boundaries and orientation on top of raw image

While *Aruco_ros* publishes a lot of useful data in its topics, another step is needed in order to incorporate fiducial marker locations and orientations into the system. *Aruco broadcaster* [14] is a package used to publish the *tf* between the camera and each fiducial marker in the /*tf* topic so it will be added to the rqt tree and allow each agent in the system to automatically have access to this data. Additionally, visualization in *RVIZ* is a lot more intuitive. This will be explained in more detail in section 4.

4 Localization and object pose estimation using fiducial markers

4.1 Working area description

The data seen in Figure 4.1 should be provided by the perception module in order to comply with the task and motion planning framework currently employed at IOC-UPC [1].



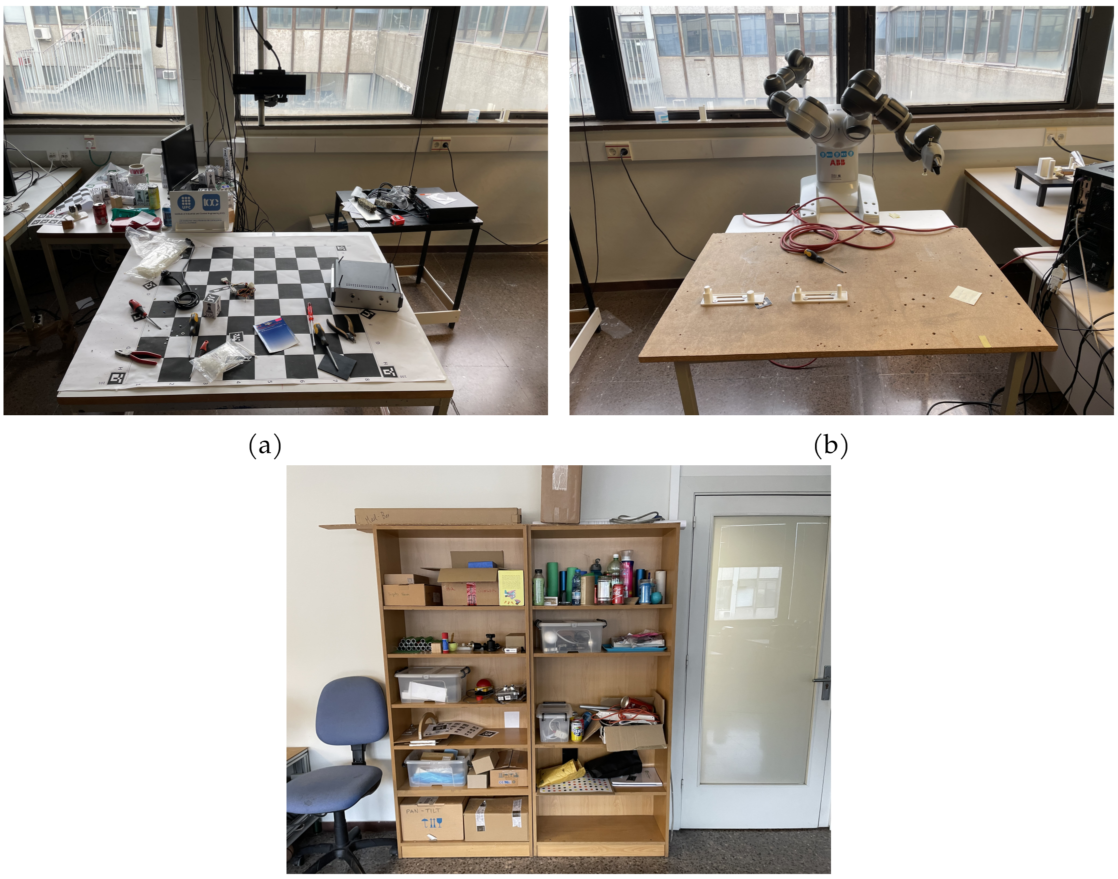
object_list.msg 147 Bytes

```
uint32[] index
uint32[] object_aruco_id
string[] frame_id
string[] object_name
string[] node_name
string[] location_name
geometry_msgs/Pose[] pose
```

Figure 4.1: Object data file used currently at IOC

The IOC Robotics lab is currently divided into 3 working areas:

- Chess table Figure 4.2a
- YuMi table Figure 4.2b
- Bookshelf area Figure 4.2c


(a) (b)

(c)

Figure 4.2: IOC Robotics Lab 2 working areas

The two tables will be covered by *Microsoft Kinect 2* cameras while items on the bookshelf will be located using an RFID module developed by another project as the area is:

- Overcrowded, so it would be difficult for the perception module to take into account an image of the whole are and provide accurate data
- It would be hard to place a camera close enough to provide an acceptable image with not so much noise

4.2 Camera Calibration and Localization

Mapping of the lab was done using a *TiaGo* robot and the */world* frame origin was set as one corner of the room. ROS is organized in such a way that in order for all agents within a system to know their spatial relationship, one */world* coordinate system is defined and connected to the coordinate frames of all agents via transforms, using the *tf* package.

In order to link the */world* frame with the working areas, it was decided to publish static transforms, as the tables had fixed locations and orientations, specifying the relationship between working area origins and the */world* frame origin (aka corner of the room) as in Figure 4.6d

Working area origins were defined as following (represented by red dots on figures):

- Chess table origin - Center of chess board Figure 4.3a
- YuMi table origin - Midpoint of table edge closest to the robot Figure 4.3b

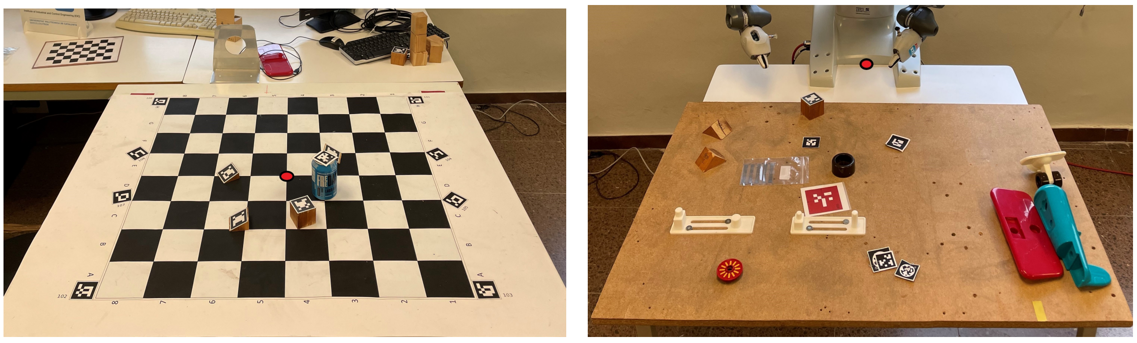

(a) (b)

Figure 4.3: Working area origins (denoted by red dots)

Subsequently, as cameras give the location of detected objects w.r.t. their location, *Kinect* sensors needed to be localized with respect to the working area origins (*ROS tf* automatically generates a transform tree and if frames are connected in any way, their tf is known, thus *camera frame* w.r.t. *world* would also be known).

Before using *Kinect* sensor cameras for obtaining any accurate data, they need to be calibrated. As *Kinects* have an RGB camera, a stereo pair for depth sensing and an IR camera, each module has to be calibrated separately. Calibration is done using a built-in *Microsoft* package by providing it with a series of images of a standardized checkerboard (Figure 4.4). In order to get the best possible precision, 100 images were taken for each module of each camera with the checkerboard being placed in a wide range of positions, orientations and distances from the camera [27].

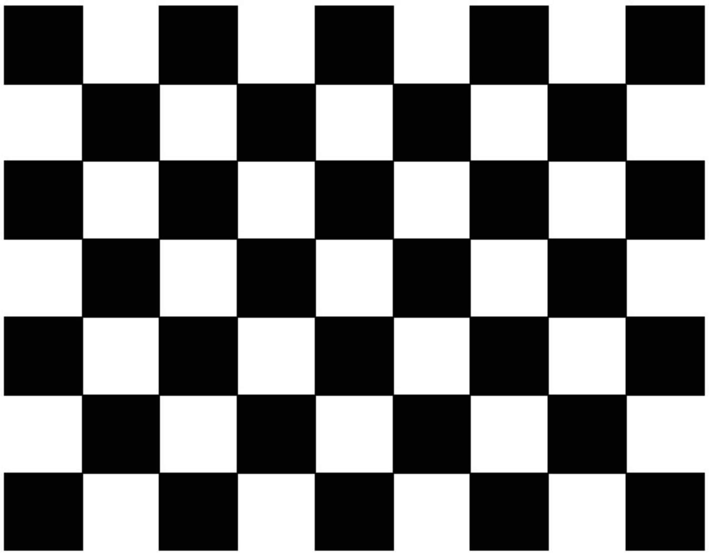
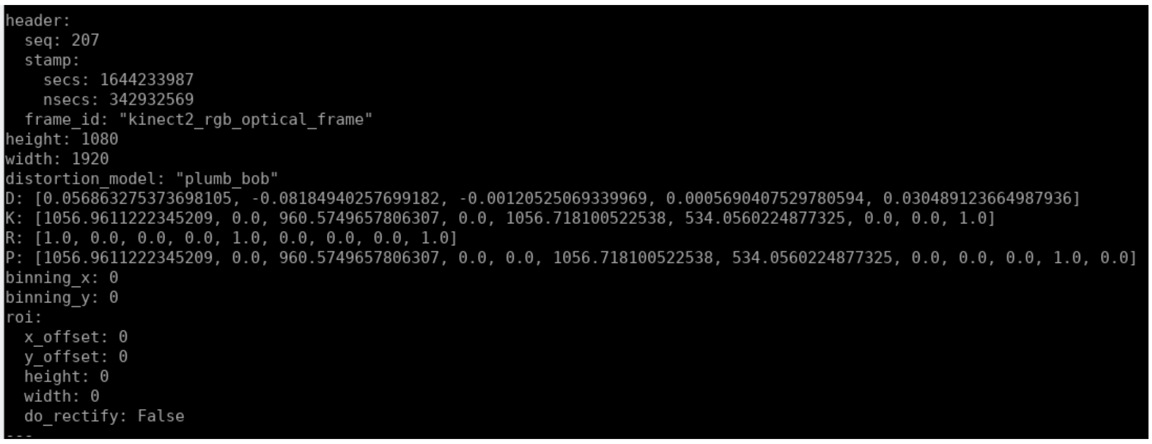


Figure 4.4: Camera calibration checkerboard

Calibration software built by *Microsoft* is then run and it produces configuration files which are loaded at each start of the *Kinect* sensors. In ROS, this data is published by the *kinect2_bridge* in the specific */camera_info* topic for each sensor as can be seen in Figure 4.5.



```
header:
  seq: 207
  stamp:
    secs: 1644233987
    nsecs: 342932569
  frame_id: "kinect2_rgb_optical_frame"
height: 1080
width: 1920
distortion_model: "plumb_bob"
D: [0.056863275373698105, -0.08184940257699182, -0.00120525069339969, 0.0005690407529780594, 0.030489123664987936]
K: [1056.9611222345209, 0.0, 960.5749657806307, 0.0, 1056.718100522538, 534.0560224877325, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [1056.9611222345209, 0.0, 960.5749657806307, 0.0, 0.0, 1056.718100522538, 534.0560224877325, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
```

Figure 4.5: *Kinect* calibration data obtained from */camera_info* topic.

Camera localization was performed using fiducial markers, specifically *Aruco tags* by applying the *table calibrator* ROS package which takes into account a number of specifically placed markers and by knowing their specifications can deduce the camera location, in detail the process went ass follows:

1. 8 *Aruco* tags from *Aruco main* library have been printed alongside chess board in various orientations

2. Aruco tag size and location and orientation known w.r.t. chess board center

3. Data from first two points fed into *table calibrator* package along with raw image from *Kinect* camera

4. *table calibrator* package outputs position and orientation data of camera w.r.t. chess board center

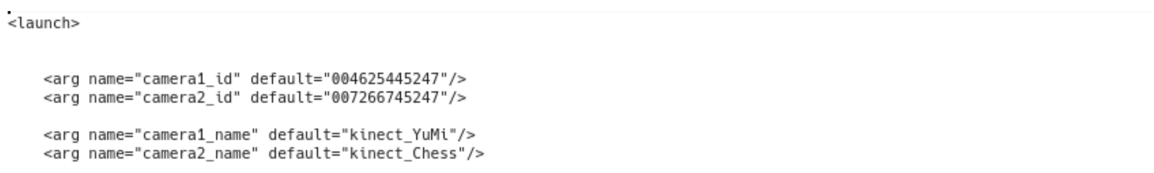
The *Kinect* sensor can not by itself identify and locate *aruco* tags, therefore the following two packages had to be used:

- *Aruco marker publisher* 3.2.2

- *Aruco broadcaster* 3.2.2

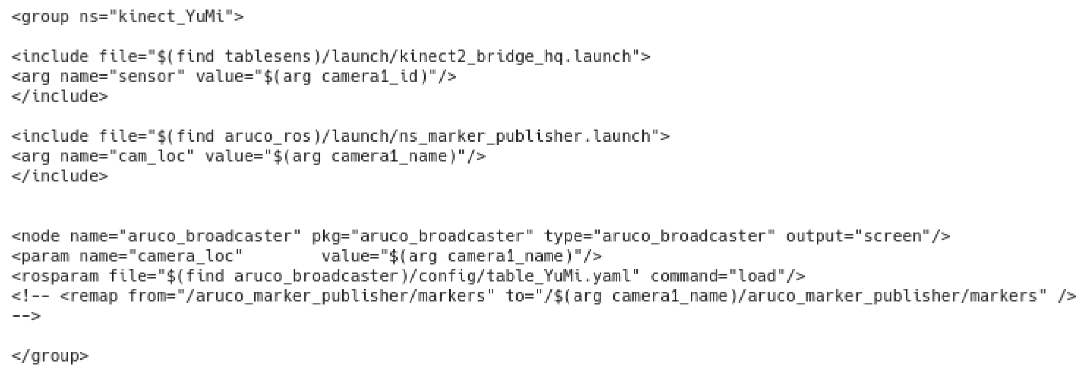
4.3 Perception Module

In order to simplify the perception module initiation, one *launch* file has been created called *perception_module.launch* Figure 4.6. Each segment is explained in the following part:



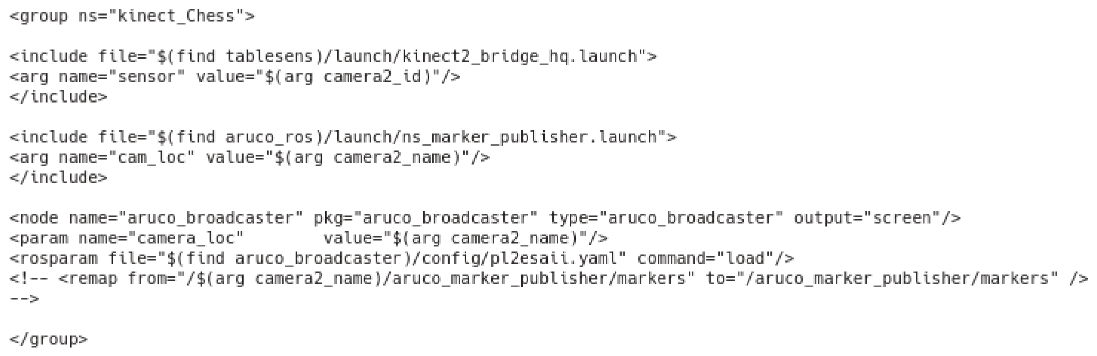
```
<launch>


    <arg name="camera1_id" default="004625445247"/>
    <arg name="camera2_id" default="007266745247"/>

    <arg name="camera1_name" default="kinect_YuMi"/>
    <arg name="camera2_name" default="kinect_Chess"/>
```

(a)



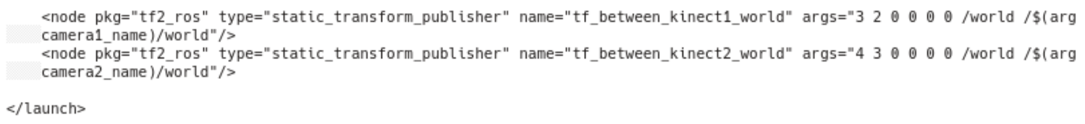
```
<group ns="kinect_YuMi">

<include file="$(find tablesens)/launch/kinect2_bridge_hq.launch">
<arg name="sensor" value="$(arg camera1_id)"/>
</include>

<include file="$(find aruco_ros)/launch/ns_marker_publisher.launch">
<arg name="cam_loc" value="$(arg camera1_name)"/>
</include>


<node name="aruco_broadcaster" pkg="aruco_broadcaster" type="aruco_broadcaster" output="screen"/>
<param name="camera_loc"        value="$(arg camera1_name)"/>
<rosparam file="$(find aruco_broadcaster)/config/table_YuMi.yaml" command="load"/>
<!-- <remap from="/aruco_marker_publisher/markers" to="/$(arg camera1_name)/aruco_marker_publisher/markers" />
-->

</group>
```

(b)



```
<group ns="kinect_Chess">

<include file="$(find tablesens)/launch/kinect2_bridge_hq.launch">
<arg name="sensor" value="$(arg camera2_id)"/>
</include>

<include file="$(find aruco_ros)/launch/ns_marker_publisher.launch">
<arg name="cam_loc" value="$(arg camera2_name)"/>
</include>

<node name="aruco_broadcaster" pkg="aruco_broadcaster" type="aruco_broadcaster" output="screen"/>
<param name="camera_loc"        value="$(arg camera2_name)"/>
<rosparam file="$(find aruco_broadcaster)/config/pl2esaii.yaml" command="load"/>
<!-- <remap from="/$(arg camera2_name)/aruco_marker_publisher/markers" to="/aruco_marker_publisher/markers" />
-->

</group>
```

(c)



```
    <node pkg="tf2_ros" type="static_transform_publisher" name="tf_between_kinect1_world" args="3 2 0 0 0 0 /world /$(arg
    camera1_name)/world"/>
    <node pkg="tf2_ros" type="static_transform_publisher" name="tf_between_kinect2_world" args="4 3 0 0 0 0 /world /$(arg
    camera2_name)/world"/>

</launch>
```

(d)

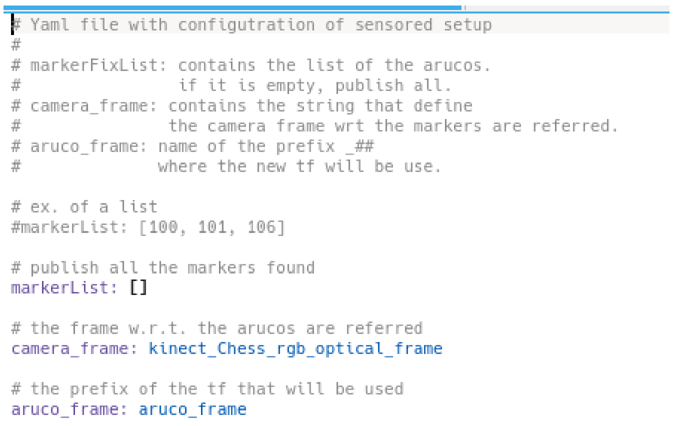
Figure 4.6: Perception module launch file

Firstly, in order to be able to run multiple *Kinect2* sensors on one PC, each one has to have a ROS bridge initiated which is done by using their serial numbers. These serial numbers, along with names for the two cameras currently in use, have been defined as ROS arguments, so they can be specified at launch, with default values corresponding to the current setup (Figure 4.6a).

Following this, we have two similar segments, corresponding to the two working areas (Figure 4.6b and Figure 4.6c), each one having its own namespace to avoid data being sent to the wrong places, allow the same fiducial marker and camera packages to work simultaneously without unwanted interactions and make accessing data more intuitive.

If we break down each one of them, we can see they consist of 3 things:

- *Kinect2* ROS bridge start for arguments specified at the top
- *Aruco marker publisher* executable run
- *Aruco broadcaster* executable run for *Aruco* markers specified and taking into account the camera localization w.r.t. working area origin (specified in .yaml configuration files as in Figure 4.7)



```
# Yaml file with configutration of sensored setup
#
# markerFixList: contains the list of the arucos.
#                 if it is empty, publish all.
# camera_frame: contains the string that define
#                the camera frame wrt the markers are referred.
# aruco_frame: name of the prefix _##
#               where the new tf will be use.

# ex. of a list
#markerList: [100, 101, 106]

# publish all the markers found
markerList: []

# the frame w.r.t. the arucos are referred
camera_frame: kinect_Chess_rgb_optical_frame

# the prefix of the tf that will be used
aruco_frame: aruco_frame
```

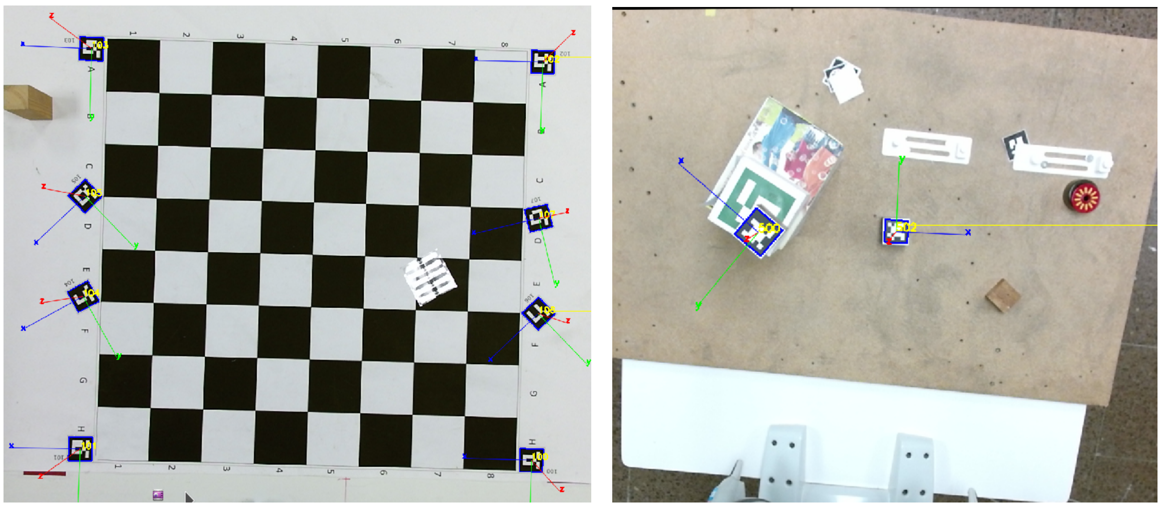
Figure 4.7: *Aruco broadcaster* configuration file for *kinect_Chess*

Also, remapping needs to be done to avoid mixing of data as all packages were previously made to be used only with one source of raw image, camera or sensor.

Finally, static transforms have been published for providing the working area origin locations and orientations with respect to the global */world* origin as can be seen in Figure 4.6d.

When this launch file is run, the following sequence is initiated:

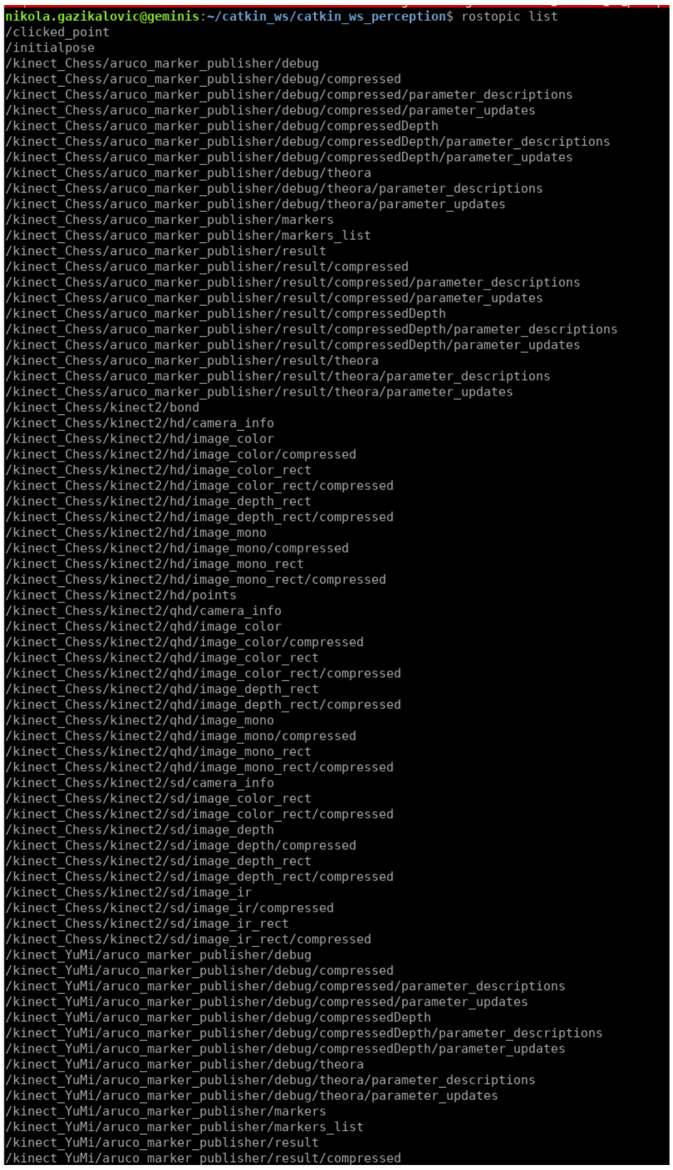
- The resulting image, published by *Aruco marker publisher*, can be seen in *RVIZ* for each of the cameras. It displays all of the detected *Aruco* tags on top of the raw image along with the x,y and z axis of each tag frame in blue green and red respectively Figure 4.8(in these images data for all of the *Aruco* tags is displayed, in order to reduce the load, ones used for localization can be omitted).



(a) (b)

Figure 4.8: Camera view for both tables (*Aruco_marker_publisher/result* topic)

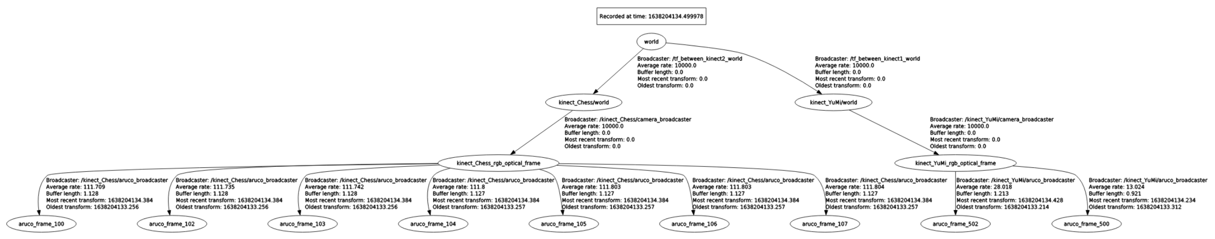
- Topics published by *Aruco marker publisher* for *Kinect2* sensor at YuMi table can be seen in Figure 4.9.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rostopic list
/clicked_point
/initialpose
/kinect_Chess/aruco_marker_publisher/debug
/kinect_Chess/aruco_marker_publisher/debug/compressed
/kinect_Chess/aruco_marker_publisher/debug/compressed/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/debug/compressed/parameter_updates
/kinect_Chess/aruco_marker_publisher/debug/compressedDepth
/kinect_Chess/aruco_marker_publisher/debug/compressedDepth/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/debug/compressedDepth/parameter_updates
/kinect_Chess/aruco_marker_publisher/debug/theora
/kinect_Chess/aruco_marker_publisher/debug/theora/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/debug/theora/parameter_updates
/kinect_Chess/aruco_marker_publisher/markers
/kinect_Chess/aruco_marker_publisher/markers_list
/kinect_Chess/aruco_marker_publisher/result
/kinect_Chess/aruco_marker_publisher/result/compressed
/kinect_Chess/aruco_marker_publisher/result/compressed/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/result/compressed/parameter_updates
/kinect_Chess/aruco_marker_publisher/result/compressedDepth
/kinect_Chess/aruco_marker_publisher/result/compressedDepth/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/result/compressedDepth/parameter_updates
/kinect_Chess/aruco_marker_publisher/result/theora
/kinect_Chess/aruco_marker_publisher/result/theora/parameter_descriptions
/kinect_Chess/aruco_marker_publisher/result/theora/parameter_updates
/kinect_Chess/kinect2/bond
/kinect_Chess/kinect2/hd/camera_info
/kinect_Chess/kinect2/hd/image_color
/kinect_Chess/kinect2/hd/image_color/compressed
/kinect_Chess/kinect2/hd/image_color_rect
/kinect_Chess/kinect2/hd/image_color_rect/compressed
/kinect_Chess/kinect2/hd/image_depth_rect
/kinect_Chess/kinect2/hd/image_depth_rect/compressed
/kinect_Chess/kinect2/hd/image_mono
/kinect_Chess/kinect2/hd/image_mono/compressed
/kinect_Chess/kinect2/hd/image_mono_rect
/kinect_Chess/kinect2/hd/image_mono_rect/compressed
/kinect_Chess/kinect2/hd/points
/kinect_Chess/kinect2/qhd/camera_info
/kinect_Chess/kinect2/qhd/image_color
/kinect_Chess/kinect2/qhd/image_color/compressed
/kinect_Chess/kinect2/qhd/image_color_rect
/kinect_Chess/kinect2/qhd/image_color_rect/compressed
/kinect_Chess/kinect2/qhd/image_depth_rect
/kinect_Chess/kinect2/qhd/image_depth_rect/compressed
/kinect_Chess/kinect2/qhd/image_mono
/kinect_Chess/kinect2/qhd/image_mono/compressed
/kinect_Chess/kinect2/qhd/image_mono_rect
/kinect_Chess/kinect2/qhd/image_mono_rect/compressed
/kinect_Chess/kinect2/sd/camera_info
/kinect_Chess/kinect2/sd/image_color_rect
/kinect_Chess/kinect2/sd/image_color_rect/compressed
/kinect_Chess/kinect2/sd/image_depth
/kinect_Chess/kinect2/sd/image_depth/compressed
/kinect_Chess/kinect2/sd/image_depth_rect
/kinect_Chess/kinect2/sd/image_depth_rect/compressed
/kinect_Chess/kinect2/sd/image_ir
/kinect_Chess/kinect2/sd/image_ir/compressed
/kinect_Chess/kinect2/sd/image_ir_rect
/kinect_Chess/kinect2/sd/image_ir_rect/compressed
/kinect_YuMi/aruco_marker_publisher/debug
/kinect_YuMi/aruco_marker_publisher/debug/compressed
/kinect_YuMi/aruco_marker_publisher/debug/compressed/parameter_descriptions
/kinect_YuMi/aruco_marker_publisher/debug/compressed/parameter_updates
/kinect_YuMi/aruco_marker_publisher/debug/compressedDepth
/kinect_YuMi/aruco_marker_publisher/debug/compressedDepth/parameter_descriptions
/kinect_YuMi/aruco_marker_publisher/debug/compressedDepth/parameter_updates
/kinect_YuMi/aruco_marker_publisher/debug/theora
/kinect_YuMi/aruco_marker_publisher/debug/theora/parameter_descriptions
/kinect_YuMi/aruco_marker_publisher/debug/theora/parameter_updates
/kinect_YuMi/aruco_marker_publisher/markers
/kinect_YuMi/aruco_marker_publisher/markers_list
/kinect_YuMi/aruco_marker_publisher/result
/kinect_YuMi/aruco_marker_publisher/result/compressed
```

Figure 4.9: Topic list

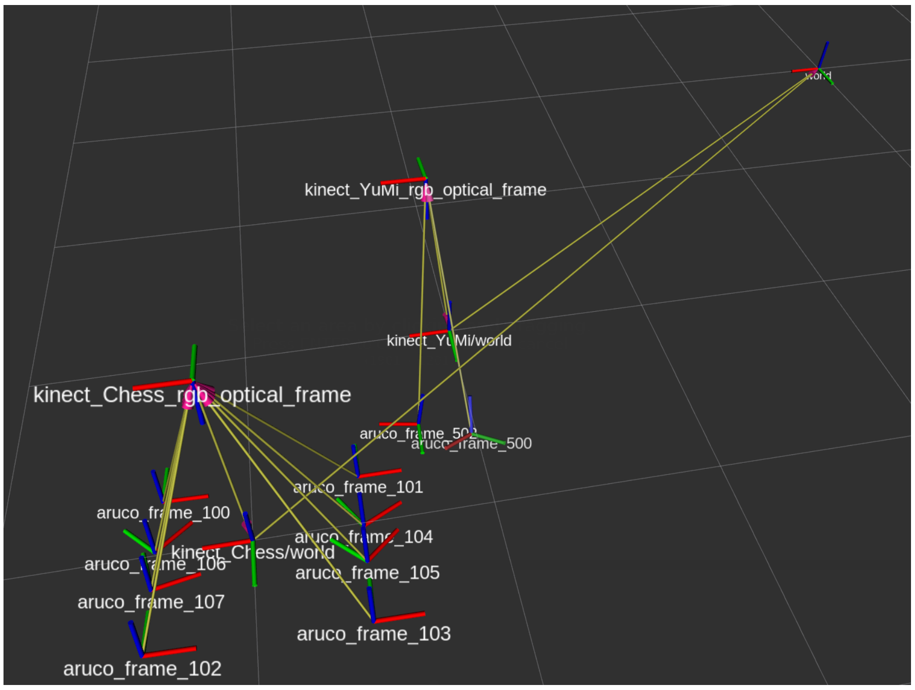
- By implementing ROS *tf*, all of the marker frames, camera frames, local world frames (at each working area) and the global */world* frame have been connected so a robot localized with respect to the *world* frame can know the position of each *Aruco* tag on a table. The rqt tree of connections between frames can be seen in Figure 4.10.



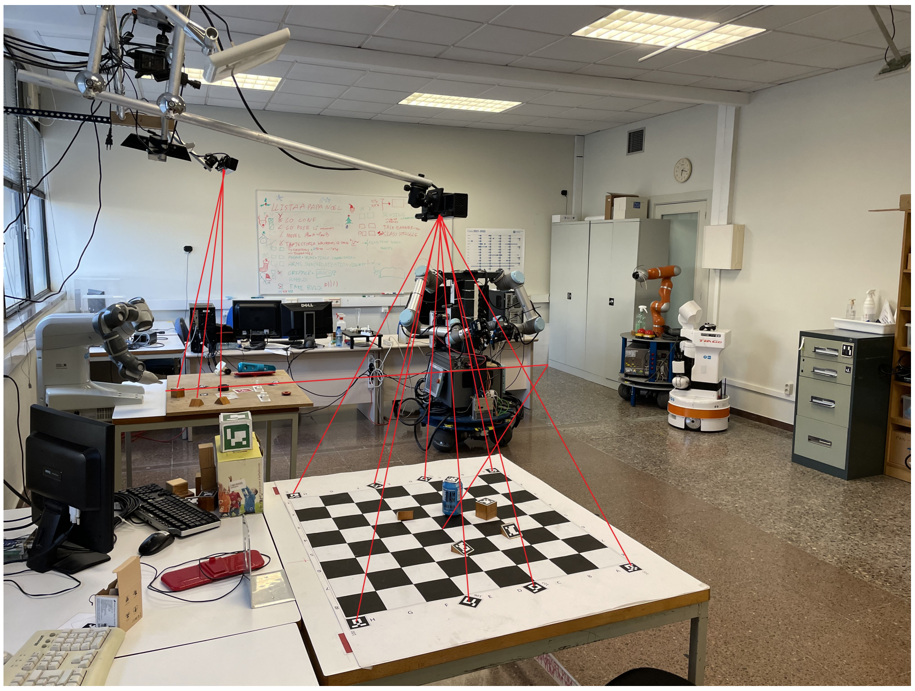
```
Broadcaster: /tf_between_kinect2_world
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0

Broadcaster: /tf_between_kinect1_world
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0

Broadcaster: /kinect_Chess/camera_broadcaster
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0

Broadcaster: /kinect_YuMi/camera_broadcaster
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.709
Buffer length: 1.128
Most recent transform: 1638204134.384
Oldest transform: 1638204133.256

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.735
Buffer length: 1.128
Most recent transform: 1638204134.384
Oldest transform: 1638204133.256

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.742
Buffer length: 1.128
Most recent transform: 1638204134.384
Oldest transform: 1638204133.256

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.8
Buffer length: 1.127
Most recent transform: 1638204134.384
Oldest transform: 1638204133.257

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.803
Buffer length: 1.127
Most recent transform: 1638204134.384
Oldest transform: 1638204133.257

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.803
Buffer length: 1.127
Most recent transform: 1638204134.384
Oldest transform: 1638204133.257

Broadcaster: /kinect_Chess/aruco_broadcaster
Average rate: 111.804
Buffer length: 1.127
Most recent transform: 1638204134.384
Oldest transform: 1638204133.257

Broadcaster: /kinect_YuMi/aruco_broadcaster
Average rate: 28.018
Buffer length: 1.213
Most recent transform: 1638204134.428
Oldest transform: 1638204133.214

Broadcaster: /kinect_YuMi/aruco_broadcaster
Average rate: 13.024
Buffer length: 0.921
Most recent transform: 1638204134.234
Oldest transform: 1638204133.312
```

Figure 4.10: rqt tree of transforms between frames

- The *tf* connections can better be visualised in *RVIZ*, where the frames are shown in space Figure 4.11a. A clear correlation can be seen if compared to Figure 4.11a, which is a photo of the library taken at the approximate corresponding location to the viewpoint in *RVIZ* Figure 4.11b.



(a) Relationship between frames presented in *RVIZ*.



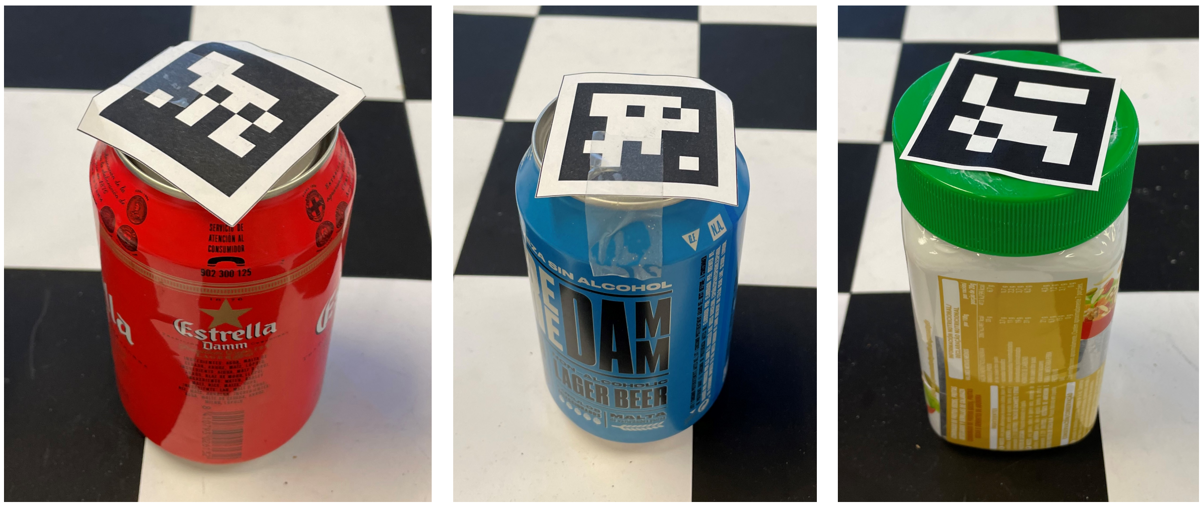
(b) *tf* representation visualized on third person lab image.

Figure 4.11: Frames visualization

In the future, the lab will probably be adding more cameras to the perception module, thus one of the keypoints is that the system is modular and cameras can be added or removed easily. Running the *perception_module.launch* file can be done with default parameters, as it was done in the initial test, where *Kinect_Chess* and *Kinect_YuMi* are selected or by specifying different

Kinect serial numbers and names for the cameras. The camera names define the namespaces and must be consistent throughout the system initialization, including the service which will be described later.

Currently the system is detecting *Aruco* tags and obtaining their pose which is then published as a *tf*. Each of these markers represents an object, as they will be a part of each item meant for robot manipulation. One example is given in Figure 4.12:



(a) (b) (c)

Figure 4.12: Fiducial marker placement on objects.

Use of fiducial markers is a fairly simple way of obtaining high accuracy, especially in the small distances as in the scope of this project. In the case that another strategy would be used to estimate the 6-dof object pose one of the beneficial notions is that the whole system could stay the same, where only the source of object pose estimation would be different and as long as it could be provided in the format of x,y,z translation and roll,pitch,yaw it would be very simple to interchange between the two methods.

An example of a PDDL file which would need to be filled in by data the perception module obtains is shown below. It is an *obj_list.xml* file used previously in testing at the IOC lab which specifies the same data as in Figure 4.1 for each object, in this case 2 objects. As mentioned previously, this data was generated artificially to resemble an object being detected by the *Kinect* cameras, the perception module will automate this process.

Listing 1: Example obj_list.xml file

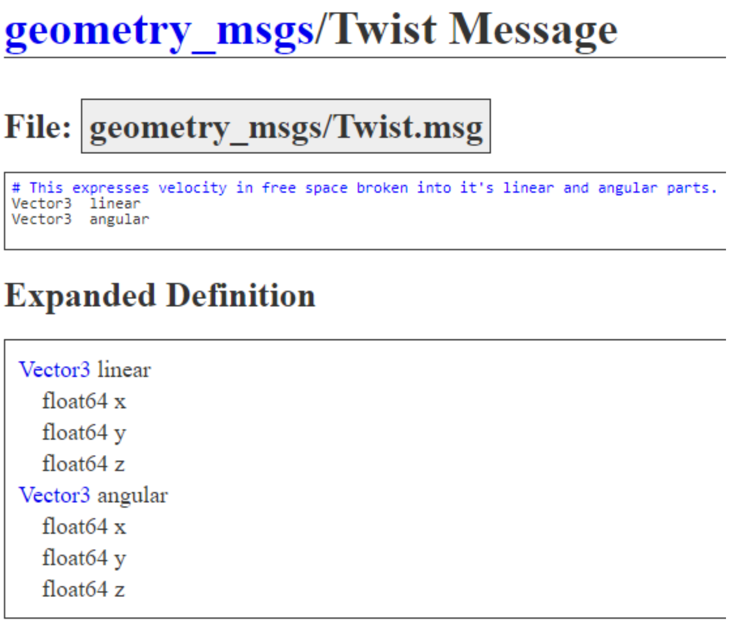
```xml
<?xml version="1.0"?>
<Object >
    <Index>1</Index>
    <ArucoID>101</ArucoID>
    <FrameID>Kinect_Camera</FrameID>
    <ObjectName>OBJECTA</ObjectName>
    <NodeName>Kinect</NodeName>
    <Pose>x=1.032 y=0.123 z=0.245 wx=0.9 wy=0.2 wz=0.3 w=0.6</Pose>
</Object>
<Object>
    <Index>2</Index>
    <ArucoID>102</ArucoID>
    <FrameID>Kinect_Camera</FrameID>
    <ObjectName>OBJECTB</ObjectName>
    <NodeName>Kinect</NodeName>
    <Pose>x=1.032 y=0.123 z=0.245 wx=0.9 wy=0.2 wz=0.3 w=0.6</Pose>
</Object>
```

All parts of the project until now focused on publishing the needed data in ROS topics, but this would not be enough as it would be too complicated for the agent to navigate through the maze of topics and published data each time it would require an object location, pose etc. In order to present the data in a more user friendly way, it was decided to develop a ROS service which would collect all of the desired data. This service would have the task of returning the data in its own message type for each call an agent places for an object.

4.4 Perception Service

The most appropriate way of presenting the data was decided to be using a newly created service message for the perception service (Figure 4.14). The request and response part of the message are separated by the dashed line where the top part represents the request, in this case a name of the object the agent calling the service is searching for, while the bottom part shows the response, assembled by the following parts:

- *twist* variable of the type *geometry_msgs/Twist* Figure 4.13 contains the transform between the camera frame and object frame in the form of a twist with linear displacement and angular rotation components.



geometry_msgs/Twist Message

File: geometry_msgs/Twist.msg

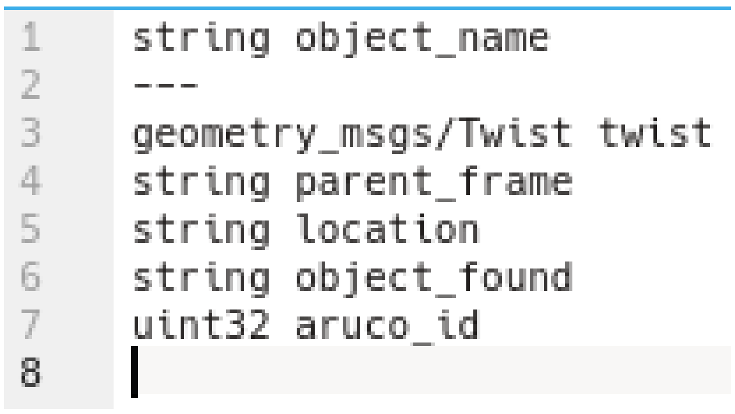
```
# This expresses velocity in free space broken into it's linear and angular parts.
Vector3  linear
Vector3  angular
```

Expanded Definition

```
Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 4.13: geometry_msgs/Twist

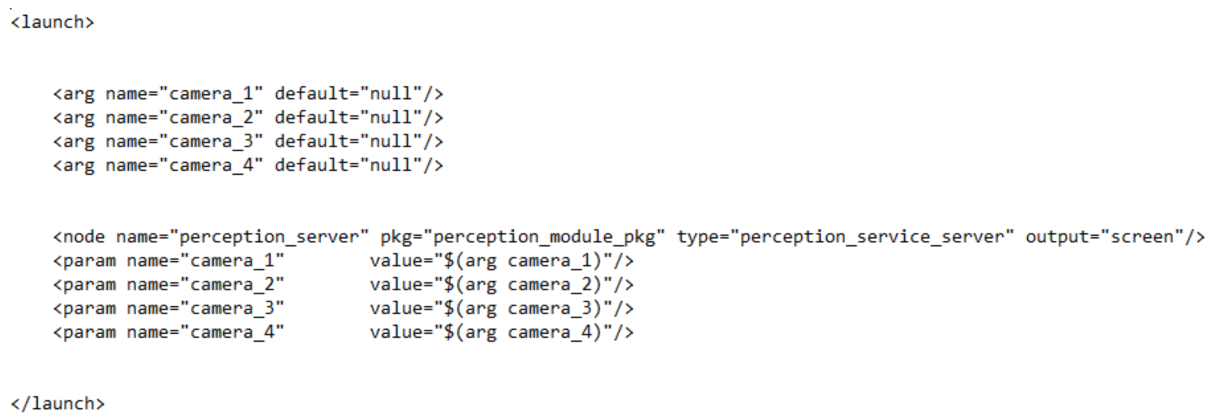
- *parent_frame* variable is a string containing the first predecessor in the rqt *tf* tree (aka the parent frame) of the object frame, in this case it gives us the frame of the camera which located the object.
- *location* variable gives us the name of the working are where the object is located in the form of a string.
- *object_found* tells the agent if the object could be located using the assigned cameras (when starting the perception module) or not, also as a string.
- *aruco_id* variable is an integer which lets the user know what is the *Aruco* tag id assigned to this object.



```
string object_name
---
geometry_msgs/Twist twist
string parent_frame
string location
string object_found
uint32 aruco_id
```

Figure 4.14: Service message construction

The launch file (Figure 4.15) for the perception module server (*perception_service_server.launch*) is relatively simple but it contains key arguments which will be explained in the next part of the report.



```
<launch>


    <arg name="camera_1" default="null"/>
    <arg name="camera_2" default="null"/>
    <arg name="camera_3" default="null"/>
    <arg name="camera_4" default="null"/>


    <node name="perception_server" pkg="perception_module_pkg" type="perception_service_server" output="screen"/>
    <param name="camera_1"       value="$(arg camera_1)"/>
    <param name="camera_2"       value="$(arg camera_2)"/>
    <param name="camera_3"       value="$(arg camera_3)"/>
    <param name="camera_4"       value="$(arg camera_4)"/>


</launch>
```

Figure 4.15: Perception service server launch file

The perception module server currently has the capacity to manipulate data from 4 cameras but this could easily be upgraded with simple changes to the code, if there was a need for expansion, although a hardware bottleneck would exist but this will be discussed in detail in the Conclusions part of the report.

By default all camera arguments are set to *null* which means no cameras are used. If a fixed system is configured this could be changed to start the same cameras every time or the user can specify each time which cameras should be taken into account. Camera selection and activation is done by setting one of the arguments to the camera name, for example in the current fixed system *camera_1* should be set by default to *Chess* while *camera_2* should be set to *YuMi*.

When the perception service server is launched it internally *activates* the cameras whose arguments values have been set to something other than *null* and subscribes to the according *"/kinect_"+cameras[x]+"/aruco_marker_publisher/markers_list"* topic, where *cameras*$[x]$ is the camera id. The above mentioned topic exist for each camera and is used to publish all *Aruco* marker IDs found by each camera.

By knowing which aruco_id is located by which camera and as the camera fields of view do not overlap, the aruco_id location is deduced.

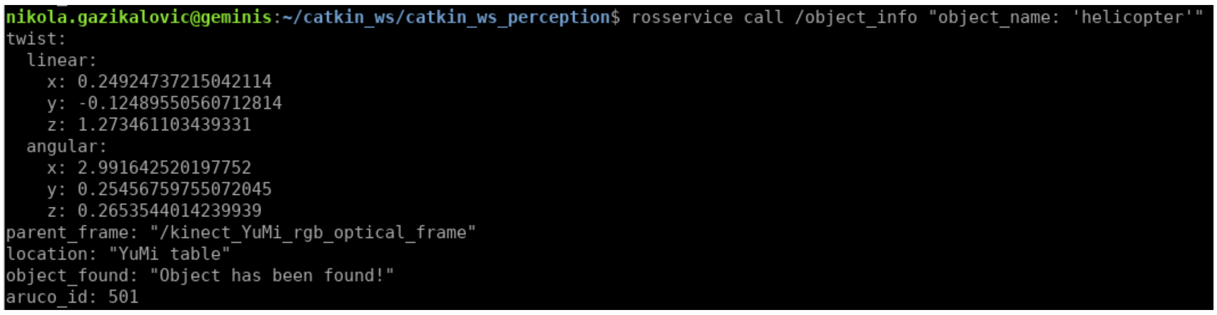
The rqt tree of ROS *tf*s consists, in this case, of a complete tree where each frame is interconnected, although there are some differences in accuracy. Because the table working area frame origins have been determined by manually measuring distances from the Lab corner (*/world* frame origin), and they are on the scale of around 10 meters, if we take into account an error of ± 1.5 percent due to tape measurement miscalibration and human error we end up at an absolute error of around 15cm which is unacceptable in the manipulation tasks. This is why cameras have been located using software and precisely located *Aruco* tags and the accuracy of the camera frame to working area frame *tf*s is a lot more accurate.

The important thing to consider is that working area frames are localized one w.r.t. the other via the */world* frame and thus it is very important to provide agents with the *tf* data from the exact camera which located the *Aruco* tag to the *Aruco* tag itself. Still the problem remains of robot localization with respect to the working area but this will be discussed in the testing segment.

With the above statements considered, the service has been programmed to locate each object and create a listener all *tf*s published between each located *Aruco* id and their parent frame (aka frame of the camera that located it).

There are two ways to obtain the service output:

1. By using the *rosservice call* command which requires the user to specify the object of interest name and returns all of the data from the response part of the service message, as shown in Figure 4.16.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'helicopter'"
twist:
  linear:
    x: 0.24924737215042114
    y: -0.12489550560712814
    z: 1.273461103439331
  angular:
    x: 2.991642520197752
    y: 0.25456759755072045
    z: 0.2653544014239939
parent_frame: "/kinect_YuMi_rgb_optical_frame"
location: "YuMi table"
object_found: "Object has been found!"
aruco_id: 501
```

Figure 4.16: Data displayed in terminal when perception service is called

2. Another option is to launch the perception service client using the *roslaunch perception_module_pkg perception_service_client* command which also requires the specification of object name along with if the user desires a .csv file to be created containing all of the data for each object. The service client has pre-programmed return commands to present data to the user in a more understandable way for humans.

When the service is called in either way, it loads the *object_data.csv* file and uses it to determine which *aruco* tag id corresponds to the object name input by the user. For example as it can be seen in Figure 4.16, an agent calls the service to locate the object called *helicopter* so the service accesses the database and finds that this name corresponds to *aruco_id_501*, thus the system only the marker id and from then on treats the object as that *Aruco* id so all following actions are performed easily.

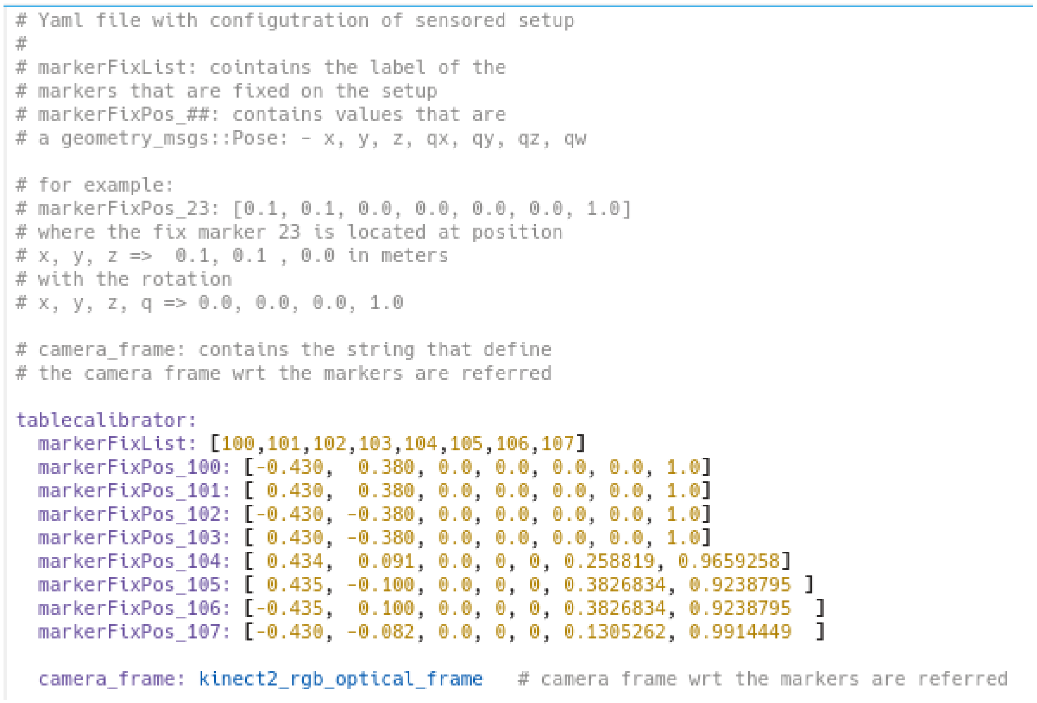
The second way of calling the service generates a .csv file which contains all of the data obtained by the perception module along with data from the *object_data.csv* file. As the perception module works only with fiducial markers which have been glued to certain objects, it can not know the exact dimensions and provide accurate data for the object itself. Robots, when performing grasping tasks, need to do it in such way that the object is secured within the end effector and doesn't slip or fall. Thus, the *object_data.csv* file contains one key field for each object, this is the location and orientation of the *Aruco* tag with respect to each object center of mass.

Finally, when creating a PDDL file, an agent should call the service for an object and proceed to use solely the .csv file which will contain all of the data needed.

4.5 Testing

Before starting the perception module, each camera used in the system needs to be localized using *tablecalibrator* from the *tablesens* package. This has been done in the following steps:

1. The position and list of all fiducial markers to be used in the camera localization have been defined with respect to the working area origin in a .yaml file Figure 4.17.

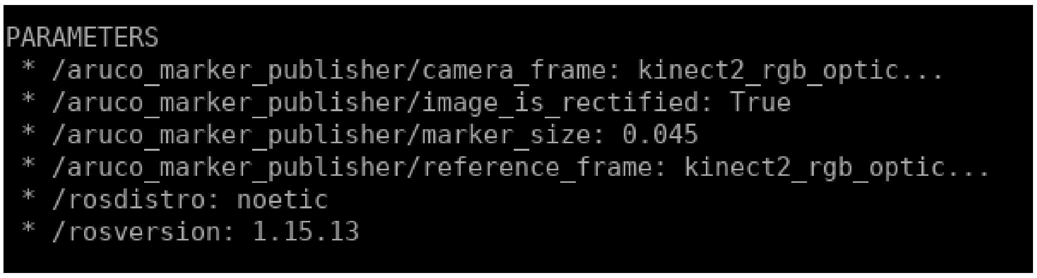


```yaml
# Yaml file with configutration of sensored setup
#
# markerFixList: cointains the label of the
# markers that are fixed on the setup
# markerFixPos_##: contains values that are
# a geometry_msgs::Pose: - x, y, z, qx, qy, qz, qw

# for example:
# markerFixPos_23: [0.1, 0.1, 0.0, 0.0, 0.0, 0.0, 1.0]
# where the fix marker 23 is located at position
# x, y, z =>  0.1, 0.1 , 0.0 in meters
# with the rotation
# x, y, z, q => 0.0, 0.0, 0.0, 1.0

# camera_frame: contains the string that define
# the camera frame wrt the markers are referred

tablecalibrator:
  markerFixList: [100,101,102,103,104,105,106,107]
  markerFixPos_100: [-0.430,  0.380, 0.0, 0.0, 0.0, 0.0, 1.0]
  markerFixPos_101: [ 0.430,  0.380, 0.0, 0.0, 0.0, 0.0, 1.0]
  markerFixPos_102: [-0.430, -0.380, 0.0, 0.0, 0.0, 0.0, 1.0]
  markerFixPos_103: [ 0.430, -0.380, 0.0, 0.0, 0.0, 0.0, 1.0]
  markerFixPos_104: [ 0.434,  0.091, 0.0, 0, 0, 0.258819, 0.9659258]
  markerFixPos_105: [ 0.435, -0.100, 0.0, 0, 0, 0.3826834, 0.9238795 ]
  markerFixPos_106: [-0.435,  0.100, 0.0, 0, 0, 0.3826834, 0.9238795  ]
  markerFixPos_107: [-0.430, -0.082, 0.0, 0, 0, 0.1305262, 0.9914449  ]

  camera_frame: kinect2_rgb_optical_frame   # camera frame wrt the markers are referred
```

Figure 4.17: .yaml file used to define *Aruco* markers used in localization of *kinect_Chess*

2. *Kinect* sensor to be calibrated is started by using the *roslaunch iai_kinect2 kinect2_bridge.launch* command to publish a raw image.

3. In another terminal, *Aruco marker publisher* (subsubsection 3.2.2) from the *aruco_ros* package is run to detect all visible *Aruco* tags (fiducial marker size has to be specified, in this case it is 0.45cm) Figure 4.18.

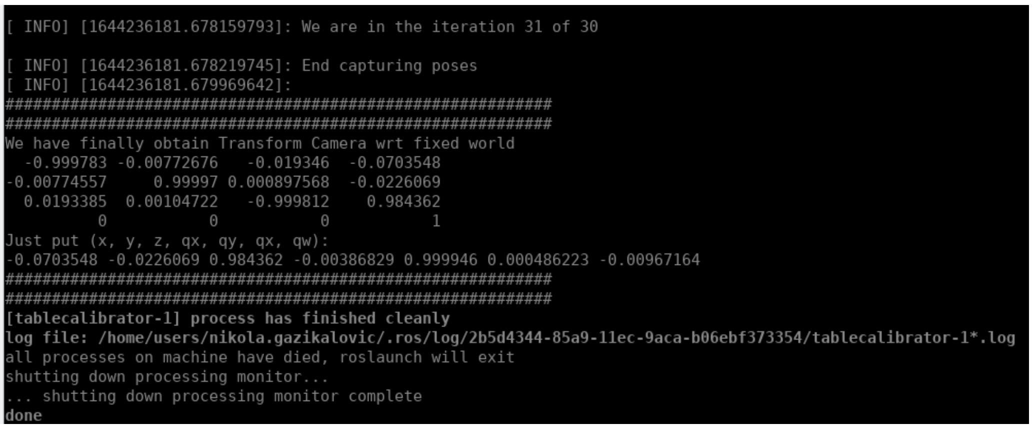


```
PARAMETERS
 * /aruco_marker_publisher/camera_frame: kinect2_rgb_optic...
 * /aruco_marker_publisher/image_is_rectified: True
 * /aruco_marker_publisher/marker_size: 0.045
 * /aruco_marker_publisher/reference_frame: kinect2_rgb_optic...
 * /rosdistro: noetic
 * /rosversion: 1.15.13
```

Figure 4.18: *Aruco marker publisher*

4. In the third terminal, *tablecalibrator* from the *tablesens* package is run which uses the de-

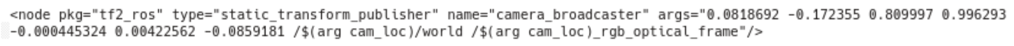
tected marker information and their position in the real world to obtain a transfer between the camera frame and working area frame in the format x,y,z translation and qx,qy,qz,qw quaternion through 31 iteration (Figure 4.19).



```
[ INFO] [1644236181.678159793]: We are in the iteration 31 of 30

[ INFO] [1644236181.678219745]: End capturing poses
[ INFO] [1644236181.679969642]:
##########################################################
##########################################################
We have finally obtain Transform Camera wrt fixed world
  -0.999783 -0.00772676   -0.019346  -0.0703548
-0.00774557     0.99997 0.000897568  -0.0226069
  0.0193385  0.00104722   -0.999812    0.984362
          0           0           0           1
Just put (x, y, z, qx, qy, qx, qw):
-0.0703548 -0.0226069 0.984362 -0.00386829 0.999946 0.000486223 -0.00967164
##########################################################
##########################################################
[tablecalibrator-1] process has finished cleanly
log file: /home/users/nikola.gazikalovic/.ros/log/2b5d4344-85a9-11ec-9aca-b06ebf373354/tablecalibrator-1*.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Figure 4.19: *kinect_Chess* localization output

5. Finally, this transform has to be published using a *static transform publisher*. To automate a launch, this transform is written in the *ns_marker_publisher.launch* file. The static transform is published when the perception module launch file runs the *marker_publisher* node where the *camera_loc* argument specifies which camera location is being defined Figure 4.20.

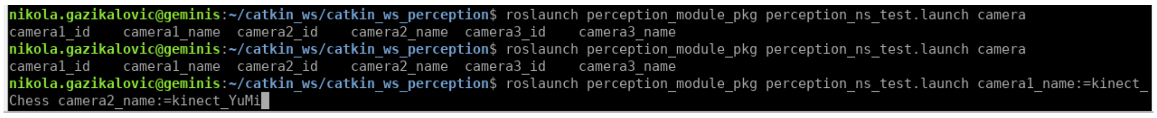


```
<node pkg="tf2_ros" type="static_transform_publisher" name="camera_broadcaster" args="0.0818692 -0.172355 0.809997 0.996293
-0.000445324 0.00422562 -0.0859181 /$(arg cam_loc)/world /$(arg cam_loc)_rgb_optical_frame"/>
```

Figure 4.20: Camera static transform publisher.

Testing of each segment of the perception module has been performed in order to assure its proper functioning:

The Perception module has been launched for cameras *kinect_Chess* and *kinect_YuMi* by setting the parameters *camera1_name* and *camera2_name* accordingly Figure 4.21. Although there is an option to change the *camerax_id*s, currently the setup is constant and thus they have been preset in the launch file itself.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_ns_test.launch camera
camera1_id    camera1_name  camera2_id    camera2_name  camera3_id    camera3_name
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_ns_test.launch camera
camera1_id    camera1_name  camera2_id    camera2_name  camera3_id    camera3_name
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_ns_test.launch camera1_name:=kinect_
Chess camera2_name:=kinect_YuMi
```

Figure 4.21: Launch of perception module and defining camera names

Alternatively, in the case that a system is constantly or most of the time setup in the same way, the perception module can be launched by specifying default parameters within the launch file and running the *roslaunch perception_module_pkg perception_service_server.launch* command without any parameters. The system is currently setup for 2 *Kinect* sensors with the *camerax_id* parameters used to give the *Kinect* bridge information about sensors which have to be started in the form of their serial numbers. In the case that a regular RGB camera needs to be used in the perception module, the user would need to comment the line containing defining the *sensor* from the *camerax_id* parameter in the launch file, and like before, define the camera name within the launch file or as a parameter when running the *roslaunch* command.

Figure 4.22 shows a real time representation of the data published by the perception module in RVIZ including:

- The view from both cameras with each *Aruco* superposed on the image
- Visualization of transforms between all frames

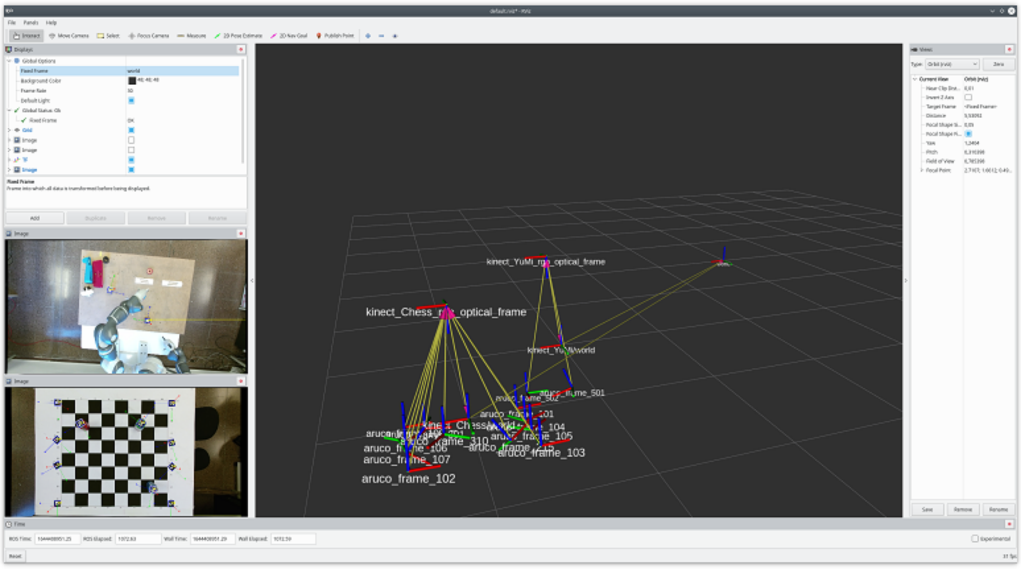
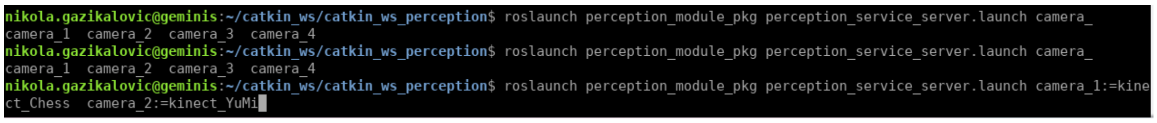


Figure 4.22: Complete RVIZ output.

Next, in another terminal, the perception service server was launched for the cameras started in the previous step Figure 4.23.



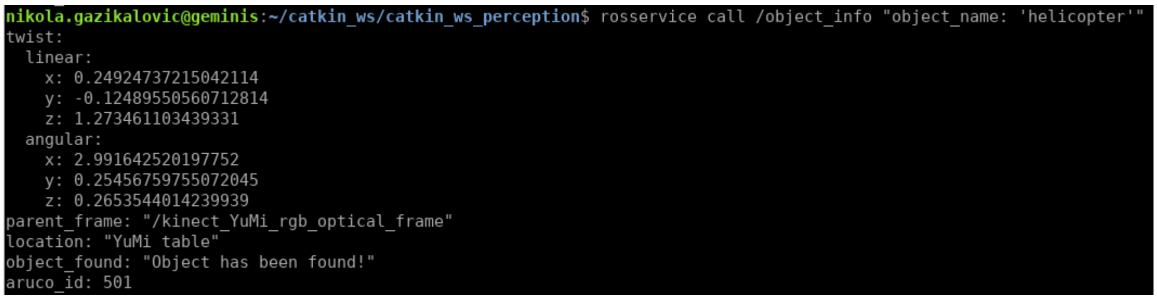
```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_service_server.launch camera_
camera_1  camera_2  camera_3  camera_4
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_service_server.launch camera_
camera_1  camera_2  camera_3  camera_4
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ roslaunch perception_module_pkg perception_service_server.launch camera_1:=kine
ct_Chess  camera_2:=kinect_YuMi
```

Figure 4.23: Launch of perception module and defining camera names

When launching the service server, the user has the options to select in which working ares the service should look for objects by setting *camera_x* parameters to names of cameras in the system, this will then start listeners and subscribers for adequate topics. By default, all cameras are set to *null* which means no cameras should be taken into account and no subscribers or listeners are to be initialized.

In order to make the service as user friendly as possible and functioning continuously (without crashes), it has to be able to respond to a variety of 3 outputs:

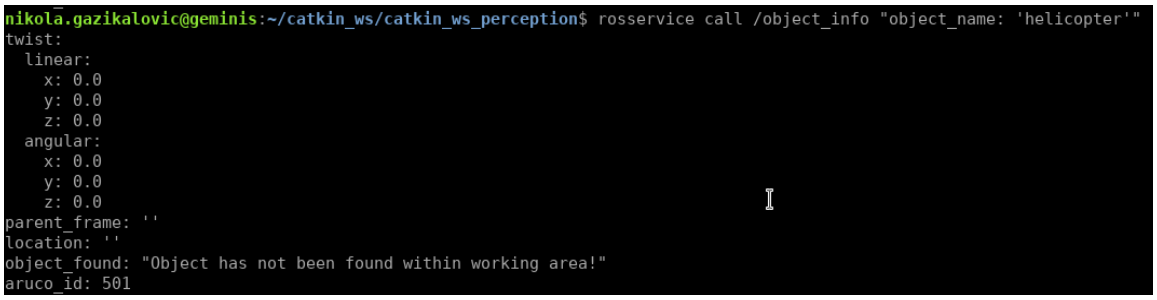
- Case 1: Service called for object existing in database and currently located with previously selected cameras Figure 4.24.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'helicopter'"
twist:
  linear:
    x: 0.24924737215042114
    y: -0.12489550560712814
    z: 1.273461103439331
  angular:
    x: 2.991642520197752
    y: 0.25456759755072045
    z: 0.2653544014239939
parent_frame: "/kinect_YuMi_rgb_optical_frame"
location: "YuMi table"
object_found: "Object has been found!"
aruco_id: 501
```

Figure 4.24: Service call output for found object.

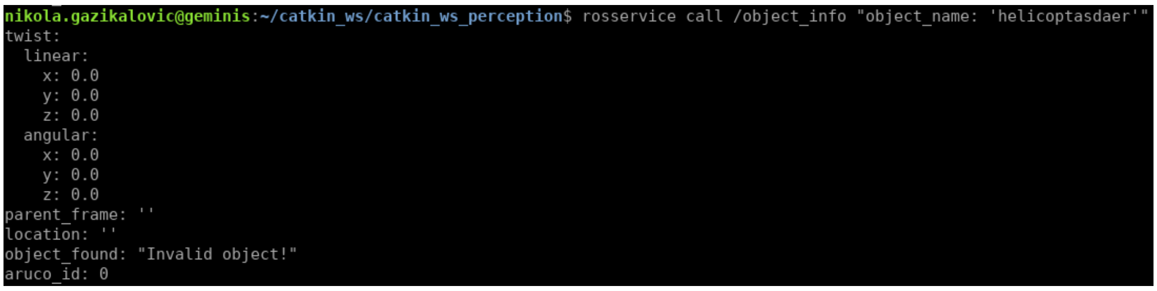
- Case 2: Service called for object existing in database but currently NOT located by selected cameras Figure 4.25.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'helicopter'"
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
parent_frame: ''
location: ''
object_found: "Object has not been found within working area!"
aruco_id: 501
```

Figure 4.25: Service call output for known object that could not be located.

- Case 3: Service called for object not defined in database Figure 4.26.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'helicoptasdaer'"
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
parent_frame: ''
location: ''
object_found: "Invalid object!"
aruco_id: 0
```

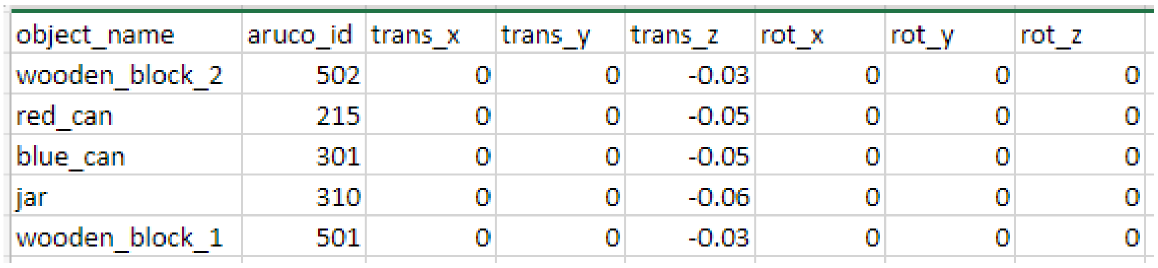
Figure 4.26: Service call output for object that doesn't exist in the database.

4.6 Testing scenarios

The perception module functionalities have been demonstrated using the following setup:

- Perception module launched for *Kinect_YuMi* and *Kinect_Chess*
- Perception service launched for *Kinect_YuMi* and *Kinect_Chess*
- *object_data.csv* from used to provide data to service Figure 4.27.

- Firstly, objects placed in Scenario 1.
- Subsequently, objects placed in Scenario 2.



object_name	aruco_id	trans_x	trans_y	trans_z	rot_x	rot_y	rot_z
wooden_block_2	502	0	0	-0.03	0	0	0
red_can	215	0	0	-0.05	0	0	0
blue_can	301	0	0	-0.05	0	0	0
jar	310	0	0	-0.06	0	0	0
wooden_block_1	501	0	0	-0.03	0	0	0

Figure 4.27: Object data used for testing

All objects used for testing are symmetrical and fiducial markers have been placed on them in such a way that the object COG frame z-axis is exactly the same as the fiducial marker frame z-axis and thus the only adjustment needed is to translate the xy plane. Please not that the camera frame z-axis and fiducial marker z-axis are opposite to each other and object center of gravity frame location has been expressed in its according *aruco_marker* frame and thus is negative.

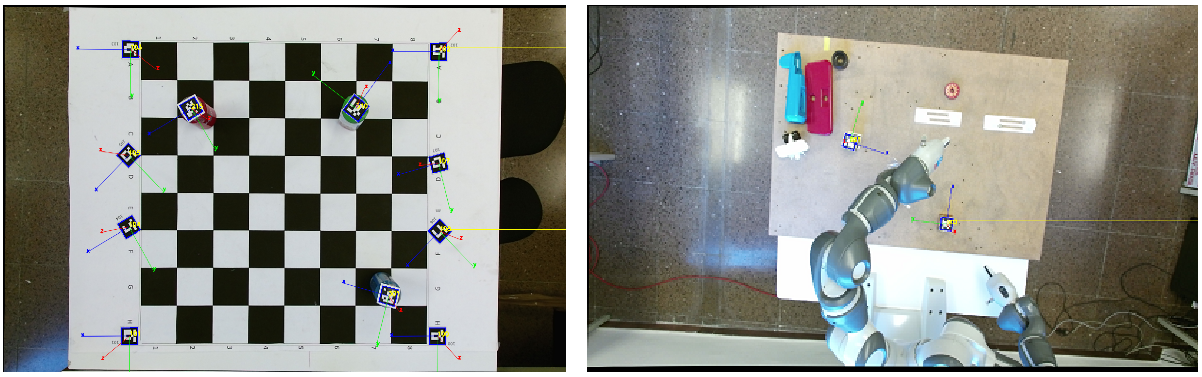
The dataset in Figure 4.27 is only a template used for testing and would be expanded to accommodate more objects and more accurate data.

4.6.1 Scenario 1

Initial setup of objects was the following:

- Objects *red_can, blue_can, jar* placed at Chess table
- Objects *wooden_block_1* and *wooden_block_2* placed at YuMi table

Where the perception module was able to identify each object as can be seen in Figure 4.28 showing camera images with overlaid fiducial marker frames,



(a) (b)

Figure 4.28: Detected objects as seen by the perception module initially.

and the frames were visulized in RVIZ as in Figure 4.29.

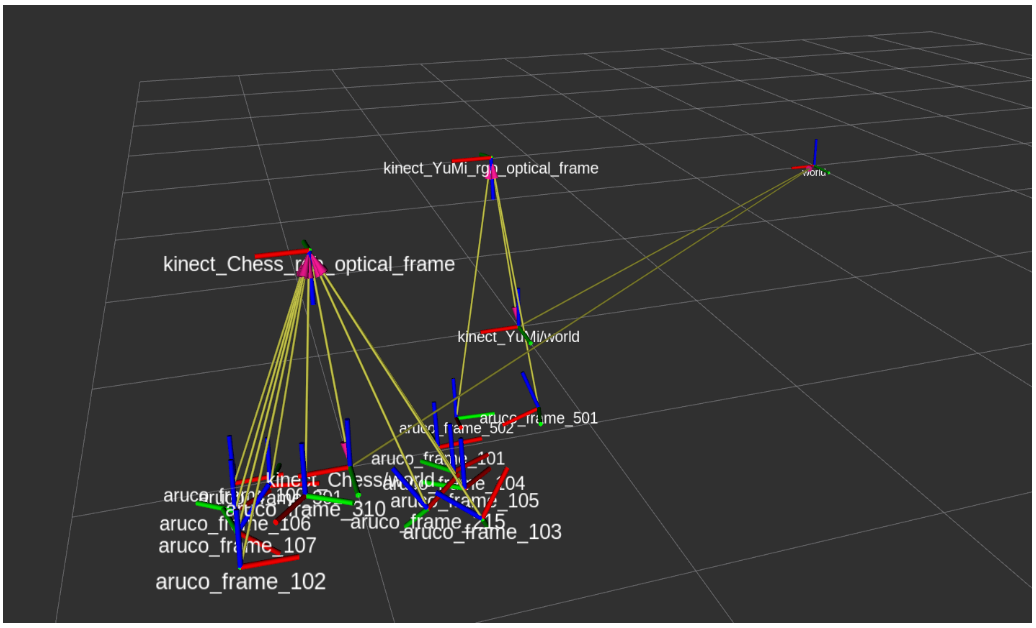
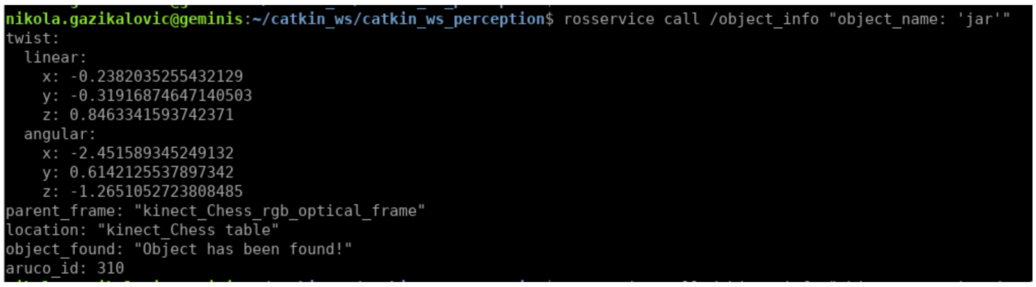


Figure 4.29: Initial setup frames visualized in RVIZ.

The next step was to verify proper functioning of the perception service:

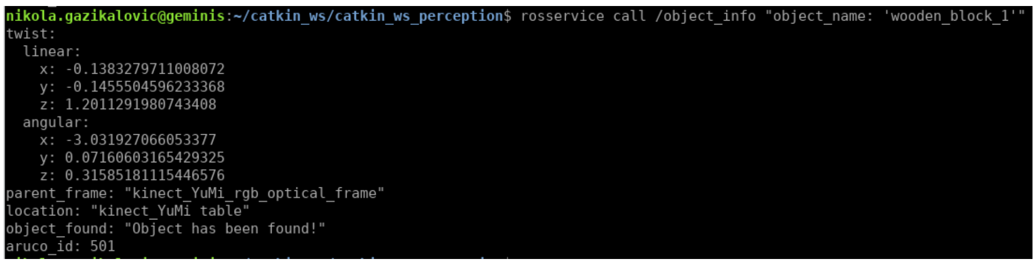
1. Service has been called for an object within *Chess* working area.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'jar'"
twist:
  linear:
    x: -0.2382035255432129
    y: -0.31916874647140503
    z: 0.8463341593742371
  angular:
    x: -2.451589345249132
    y: 0.6142125537897342
    z: -1.2651052723808485
parent_frame: "kinect_Chess_rgb_optical_frame"
location: "kinect_Chess table"
object_found: "Object has been found!"
aruco_id: 310
```

Figure 4.30: Service called for object *jar*.

2. Service has been called for an object within *YuMi* working area.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'wooden_block_1'"
twist:
  linear:
    x: -0.1383279711008072
    y: -0.1455504596233368
    z: 1.2011291980743408
  angular:
    x: -3.031927066053377
    y: 0.07160603165429325
    z: 0.31585181115446576
parent_frame: "kinect_YuMi_rgb_optical_frame"
location: "kinect_YuMi table"
object_found: "Object has been found!"
aruco_id: 501
```

Figure 4.31: Service called for object *wooden_block_1*.

As it can be seen in figures above, the service correctly looks up data from *object_data.csv* and in accordance with the perception module, presents the service message output.

4.6.2 Scenario 2

To determine if the perception module is functioning properly, some of the objects have been moved in the following way:

- Objects *wooden_block_1* placed at Chess table.
- Objects *orange_can, jar* placed at YuMi table.
- Objects *red_can, wooden_block_2, blue_can* placed at YuMi table.

Again, the perception module was able to identify each object as can be seen in Figure 4.32 showing camera images with overlaid fiducial marker frames,

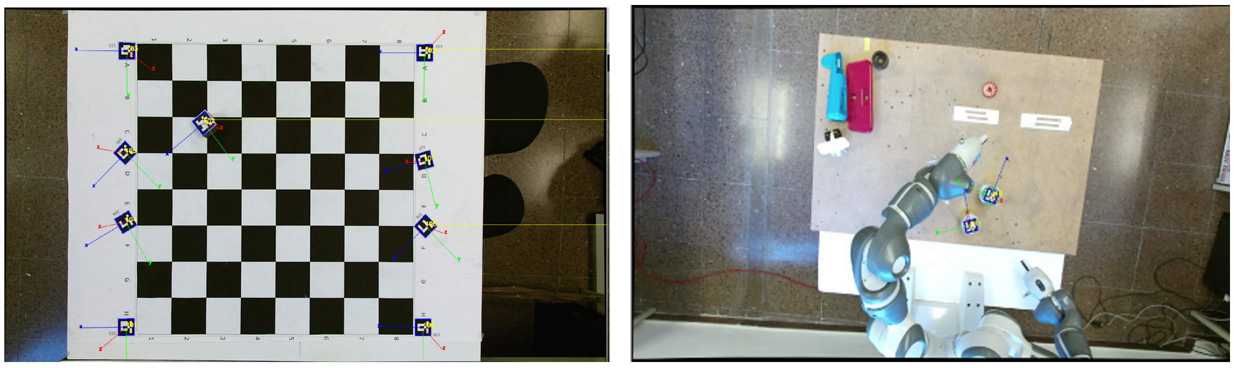

(a) (b)

Figure 4.32: Detected objects as seen by the perception module initially.

and the frames were visualized in RVIZ as in Figure 4.33.

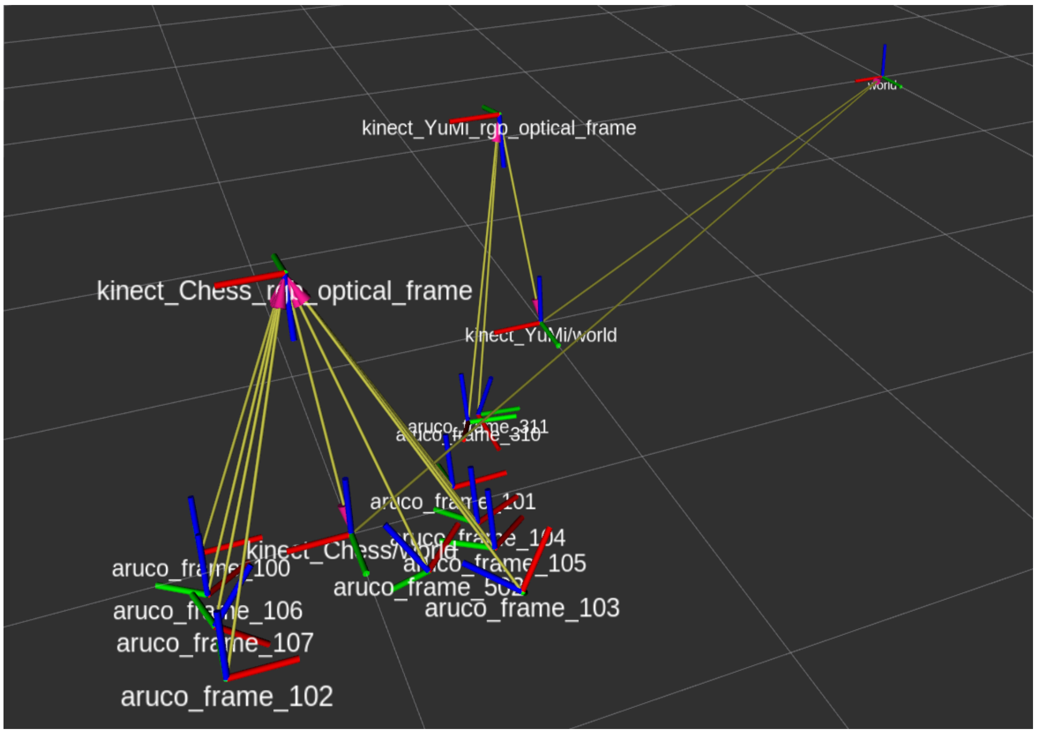
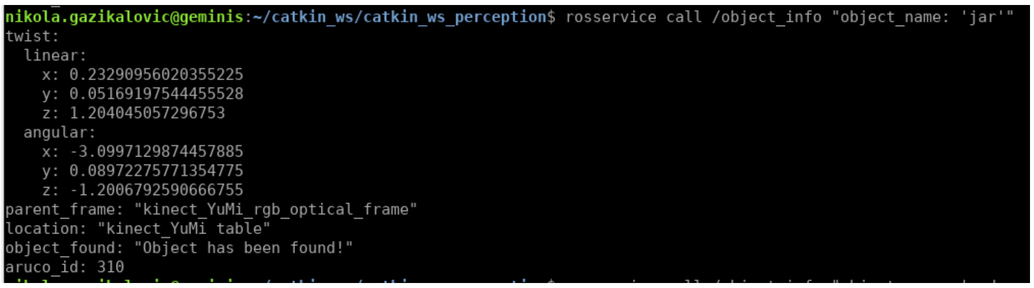


Figure 4.33: Moved setup frames visualized in RVIZ.

The next step was, again, to verify proper functioning of the perception service but there were a few differences:

- Object *orange_can* has not been defined in the *object_data.csv* database file.
- Some of the objects which have been defined in the database, are currently not in any of the working areas.

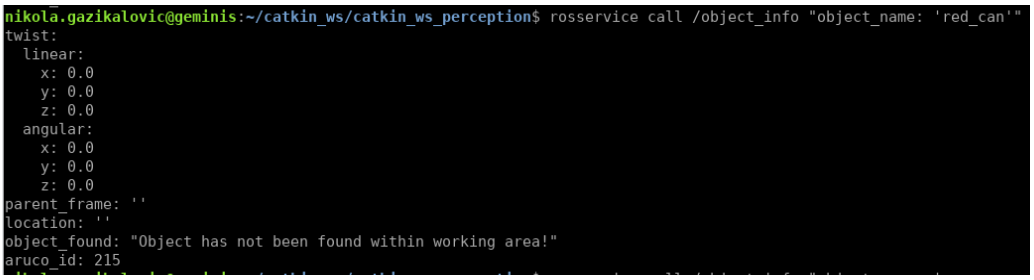
1. Service has been called for the object *jar* which is now in *YuMi* working area.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'jar'"
twist:
  linear:
    x: 0.23290956020355225
    y: 0.05169197544455528
    z: 1.204045057296753
  angular:
    x: -3.0997129874457885
    y: 0.08972275771354775
    z: -1.2006792590666755
parent_frame: "kinect_YuMi_rgb_optical_frame"
location: "kinect_YuMi table"
object_found: "Object has been found!"
aruco_id: 310
```

Figure 4.34: Service called for object *jar*.

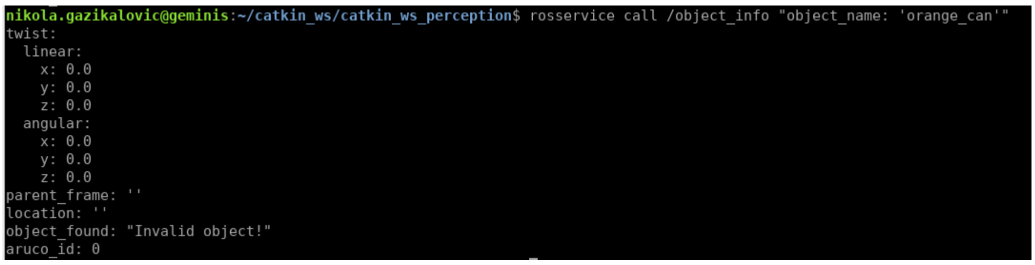
2. Service has been called for the *red_can* object which is currently not in any of the working areas.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'red_can'"
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
parent_frame: ''
location: ''
object_found: "Object has not been found within working area!"
aruco_id: 215
```

Figure 4.35: Service called for object *red_can*.

3. Service has been called for the *orange_can* object which, although apparently visible at table *YuMi*, has not been defined in the object database.



```
nikola.gazikalovic@geminis:~/catkin_ws/catkin_ws_perception$ rosservice call /object_info "object_name: 'orange_can'"
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
parent_frame: ''
location: ''
object_found: "Invalid object!"
aruco_id: 0
```

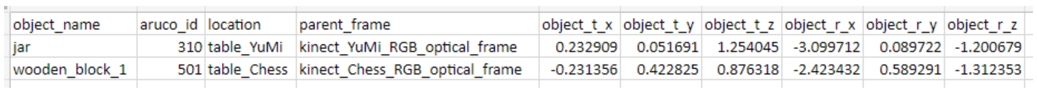
Figure 4.36: Service called for object *orange_can*.

If the user desires that the perception service generates a .csv file containing data for all of the visible objects, the service client must be launched with the following arguments:

- *object_name*="all"
- *generate_csv*="yes"

For scenario 2, the generated .csv file is presented in Figure 4.37. This file contains the most important information, for the agent, about each located object:

- *Object_name* and *aruco_id* which are taken from the object database.
- *location* of the object, meaning the working are at which it has been spotted.
- *parent_frame* with respect to which the location of the object is defined.
- Translational and rotational parameters defining the transform between *parent_frame* and the object center of gravity frame. This data is generated by combining the transform between the camera frame and the fiducial marker frame with the transform between the object COG frame and the fiducial marker frame. As it can be seen in the example of the *jar* object, the only difference is in the z axis translation, positive in the camera frame but negative in the fiducial marker frame as their z axis are opposite to each other.



object_name	aruco_id	location	parent_frame	object_t_x	object_t_y	object_t_z	object_r_x	object_r_y	object_r_z
jar	310	table_YuMi	kinect_YuMi_RGB_optical_frame	0.232909	0.051691	1.254045	-3.099712	0.089722	-1.200679
wooden_block_1	501	table_Chess	kinect_Chess_RGB_optical_frame	-0.231356	0.422825	0.876318	-2.423432	0.589291	-1.312353

Figure 4.37: Data for all located objects.

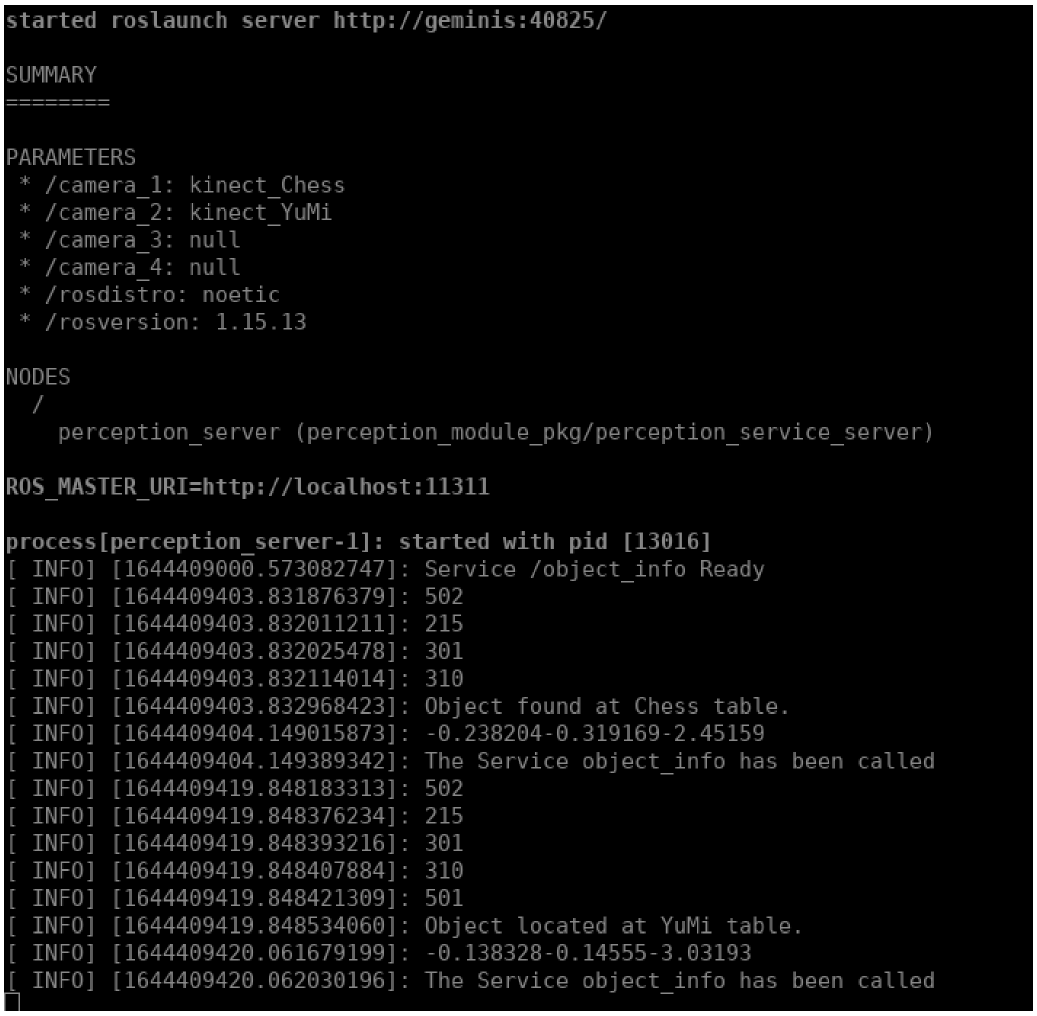
For the sequence of requested objects in subsubsection 4.6.1, the service server gives the following output (Figure 4.38):

1. In the parameter section, displays which cameras are being used.

2. Notifies the user that the service is ready.

3. Goes through the database until the desired item has been found.

4. Lets the user know if and where the object has been located.



```
started roslaunch server http://geminis:40825/

SUMMARY
========

PARAMETERS
 * /camera_1: kinect_Chess
 * /camera_2: kinect_YuMi
 * /camera_3: null
 * /camera_4: null
 * /rosdistro: noetic
 * /rosversion: 1.15.13

NODES
  /
    perception_server (perception_module_pkg/perception_service_server)

ROS_MASTER_URI=http://localhost:11311

process[perception_server-1]: started with pid [13016]
[ INFO] [1644409000.573082747]: Service /object_info Ready
[ INFO] [1644409403.831876379]: 502
[ INFO] [1644409403.832011211]: 215
[ INFO] [1644409403.832025478]: 301
[ INFO] [1644409403.832114014]: 310
[ INFO] [1644409403.832968423]: Object found at Chess table.
[ INFO] [1644409404.149015873]: -0.238204-0.319169-2.45159
[ INFO] [1644409404.149389342]: The Service object_info has been called
[ INFO] [1644409419.848183313]: 502
[ INFO] [1644409419.848376234]: 215
[ INFO] [1644409419.848393216]: 301
[ INFO] [1644409419.848407884]: 310
[ INFO] [1644409419.848421309]: 501
[ INFO] [1644409419.848534060]: Object located at YuMi table.
[ INFO] [1644409420.061679199]: -0.138328-0.14555-3.03193
[ INFO] [1644409420.062030196]: The Service object_info has been called
```

Figure 4.38: Service server output in terminal.

One of the biggest concerns, with regards to the system accuracy, is distance from the camera to the table at *table_YuMi*. In previous figures representing images from the cameras it can be seen that at *table_YuMi* the working area takes only around 60% of the cameras field of view and thus in some occasions the perception module could not identify fiducial markers. With object *jar* placed on the table (Figure 4.39b), the system doesn't recognise it, but when it is placed on top of another object (Figure 4.39a), it is clearly identified. Thus, the obvious solution would be to to position the camera closer to the working area but as in this way it could come within reach of robot *YuMi*, different possibilities will be discussed in the Conclusions section.

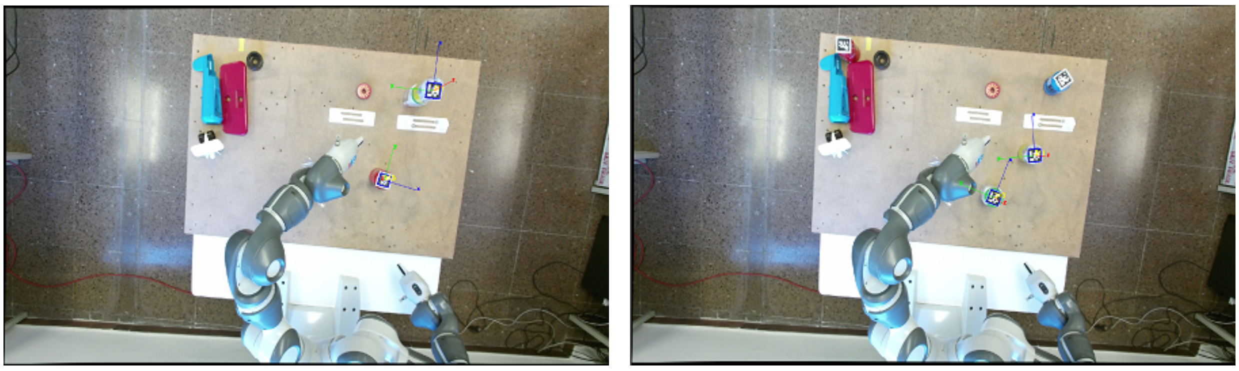

(a) (b)

Figure 4.39: Variability in object detection at *table_YuMi*

5 Estimation of congestion levels within work area using OAK-D framework

The OAK-D, which is built on open source tools such as OpenCV, takes advantage of numerous advancements in computer vision since the release of the Kinect, allowing users to integrate various machine learning models that assist developers in identifying not only poses and locations, but also types of objects, emotions, and other computer vision tasks. *Luxonis* git repository is where all of the trained models, specifically optimized for the OAK-D, are available [18]. The segments *luxonis/depthai*, *luxonis/depthai-python* are where the main, thoroughly tested machine learning models can be found while *luxonis/depthai-experiments* is used for very advanced experimental projects in beta testing. Additionally, there is a section called *luxonis/depthai-ml-training* which contains useful training scripts in the case a user wants to train a NN from scratch in the desired format.

The OAK-D instructions were found to be very intuitive with basic packages and being installed and run without problems, the only required software was *Python3* and *OpenCV*. The required packages have been installed on a PC at the IOC lab and calibration of all cameras was done, as with the *Kinect2* sensors, in order to acquire the best performance from NN models.

Many of the available models could find a use in the whole perception module but the following were shortlisted with consideration to the project scope:

- Human pose detection - Algorithm for estimating the pose of a human which could be very useful for giving directions, signals or other input to a robot or for predicting human motion but unfortunately was not accurate enough and became very computationally heavy when more than one person came into cameras field of view (Figure 5.1).

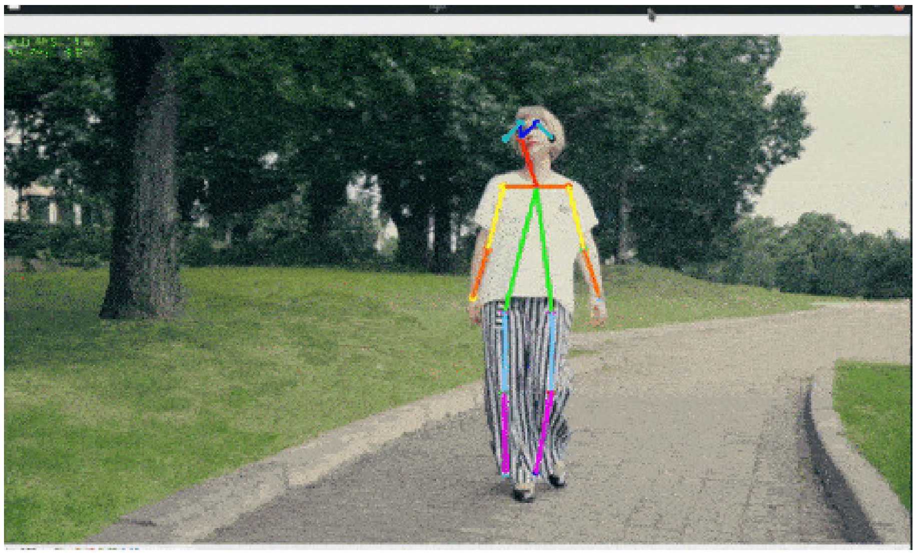


Figure 5.1: Human pose

- People tracker - Trained model for tracking the number of people which enter and leave the area observed by the camera
- Crowd counting - A very computationally expensive algorithm which estimates the num-

ber of people in crowds (Figure 5.2)

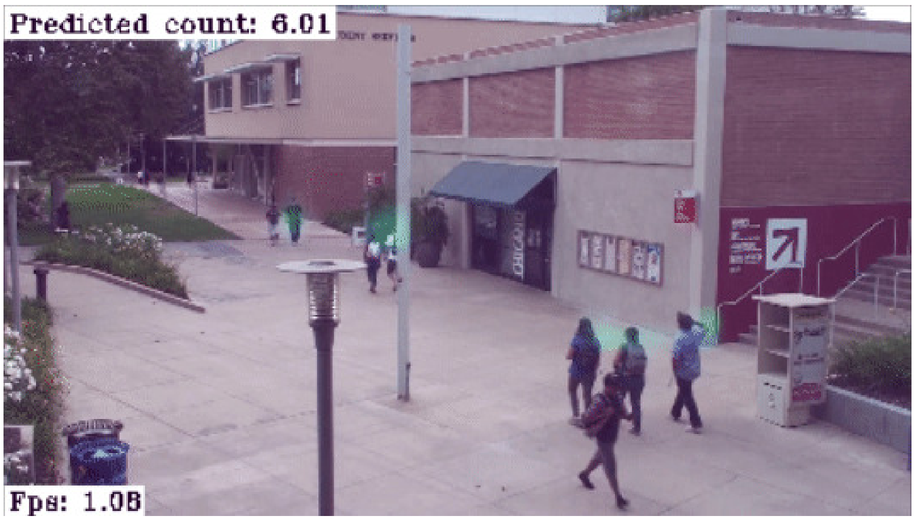


Figure 5.2: Crowd counting

- Mask detection - Algorithm used for detecting all human faces within field of view and then identifying if they are covered by masks

After careful consideration and testing of almost all of the available models it was concluded that although the OAK-D module has amazing capabilities with respect to its size, the processing power available can not be compared to a dedicated GPU in a PC and even with all of the models being maximally optimized for use on such a platform, not all of them are useful in real life applications due to inference speeds being as low as 1 FPS (Crowd counting). The people tracker model was selected to be a useful inclusion for the perception module in order to determine congestion levels within the laboratory based on which robots would adapt their safety margins, acceleration and speed of movement and manipulation. Further testing was undergone in the *PyCharm* environment but as it was considered an additional functionality to the perception module, it was left for future implementation with the current focus being on object pose estimation.

6 6-DOF Object pose estimation by applying MediaPipe machine learning solutions

The *Hands* model has been tested using images from a laptop webcam and running it solely on the laptop CPU which was proven to be substantial enough in order to get a reasonable 25FPS (reduced when markers visualized on image). As can be seen in Figure 6.1, the model works very well in reduced lighting circumstances (a and b), partial occlusions (c and d) and overlaps (a) along with being able to process more than one hand at a time (with a slight reduction in FPS).

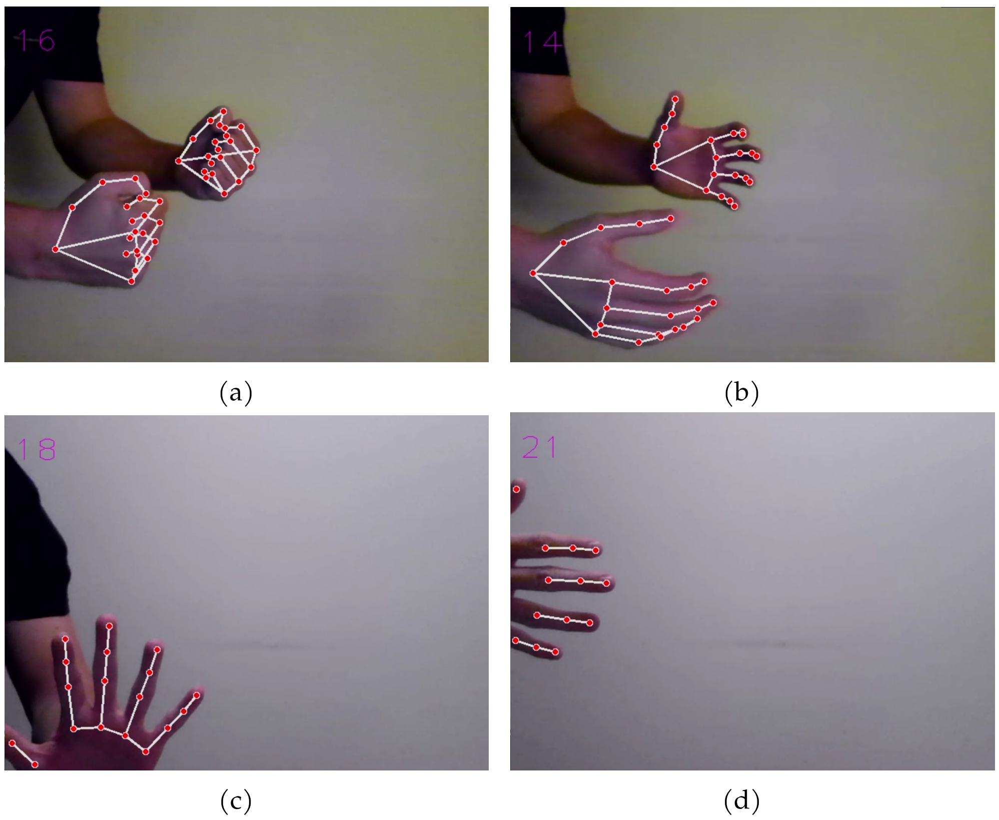


Figure 6.1: *MediaPipe Hands* testing in various situations (framerate shown in top left corners)

Although, *MediaPipe Hands* has many possible implementations, it was decided to focus on improving the previously developed framework for object pose detection by making it more robust and cancelling the need for fiducial markers.

Google and *MediaPipe* provide a tool called *Objectron*, a real-time 3D object identification solution for ordinary items (also can be run on mobile devices). It recognizes and estimates the postures of objects in two-dimensional (2D) photos using a machine learning (ML) model trained on the Objectron dataset (which is also provided). While two-dimensional prediction (a well studied area) gives just two-dimensional bounding boxes, three-dimensional prediction captures an object's size, position, and orientation in the environment, which is the data a robots needs in order to be able to manipulate this object successfully. Whereas 2D object recognition is quite established and frequently used in industry, 3D object detection from 2D photography is a difficult issue to solve owing to a lack of data and the variety of looks and forms of items within a category.

To identify ground truth data, a unique annotation tool has been developed by *Google* for use with augmented reality session data that enables annotators to easily label objects' 3D bounding bounds. Again, all data used to train the ML models has been manually annotated by placing bounding boxes over identified objects in separate frames of example videos.

Objectron currently provides trained models for the following objects:

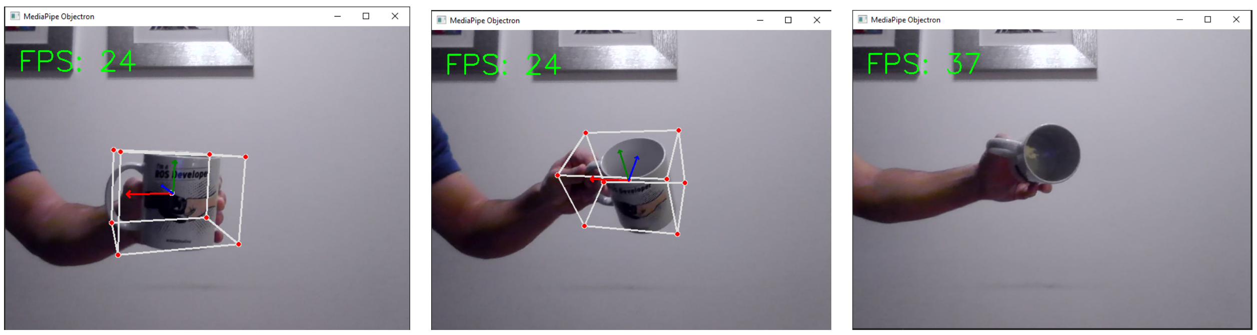
- Shoe
- Chair
- Cup
- Camera

and a data set of 15000 annotated video clips supplemented with over 4 million annotated images for training new models in the following categories:

- Bike
- Book
- Bottle
- Cereal box
- Laptop

Training models can be done in *PyTorch* or *Tensorflow* environments.

While training new deep learning models is out of the scope of this project, the already available ones were put to use. Firstly, each model was tested in the *PyCharm* environment. Figure 6.2 shows testing performed on the most reasonable object for the use case at hand, a cup.



(a) (b) (c)

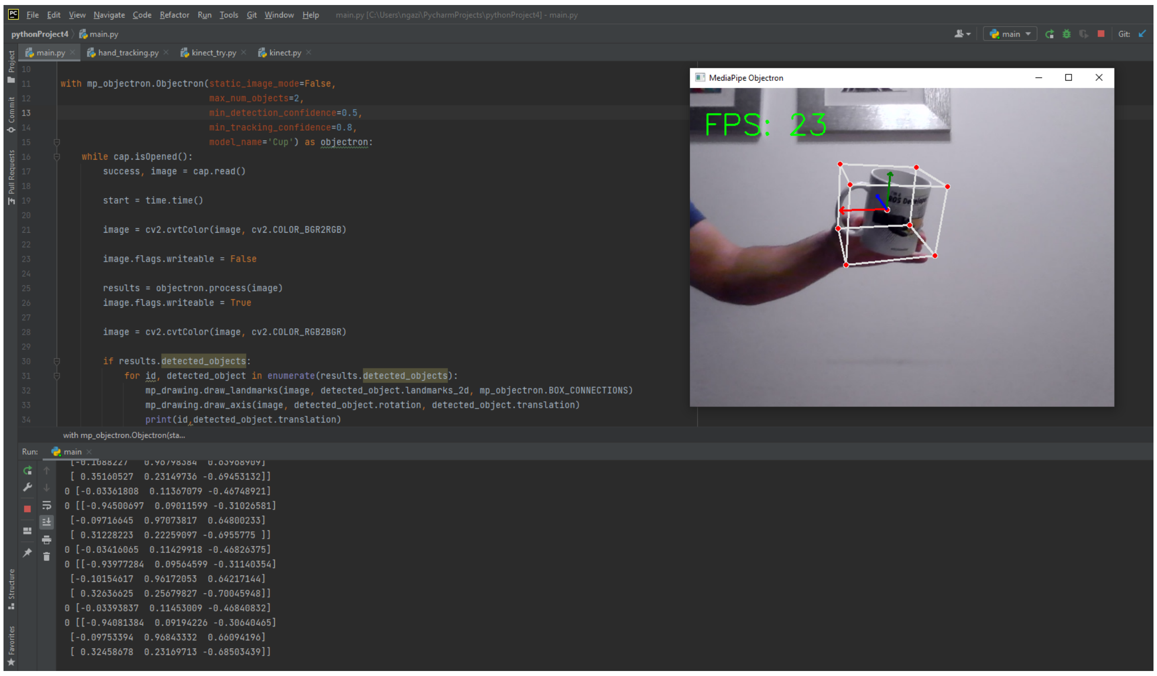
Figure 6.2: *MediaPipe Objectron* testing in various situations for object *Cup*(framerate shown in top left corners)

Obtained results varied with respect to lighting and object orientation as can be seen from the overlaid bounding box orientation. Nevertheless, considering in which circumstances the perception module will be implemented, it was decided to focus on improving the model for manipulation of cups.

While the algorithm performed very well in good lightning and from the most recognizable angles as can be seen in Figure 6.2a and Figure 6.2b, in the specific use case of the IOC Lab, the cameras are placed directly above the working areas, as they have been optimized for working with fiducial markers and from this angle *MediaPipe Objectron* doesn't perform in a satisfactory manner (Figure 6.2c).

When an object and its pose are detected, *Objectron* outputs the bounding box coordinates (Figure 6.3), size and center, this data can be used in ROS to publish a *tf* with respect to the camera frame. In order to use this transform, the following limitations would have to be taken into account and overcome:

- It would need to have a fixed offset because the bounding box center will not be exactly the same as the object center of mass (location where the robot would grasp the object).
- If the object is a cup, it needs to be grasped from a certain side
- A large safety margin will is needed as pose estimation is not as precise as with fiducial markers
- Detection accuracy varies with respect to side from which camera sees the object


File Edit View Navigate Code Refactor Run Tools Git Window Help
main.py [C:\Users\ngazi\PycharmProjects\pythonProject4] - main.py
pythonProject4
main.py
hand_tracking.py
kinect_try.py
kinect.py
MediaPipe Objectron
FPS: 23
with mp_objectron.Objectron(sta...
Run:
main

```
with mp_objectron.Objectron(static_image_mode=False,
                            max_num_objects=2,
                            min_detection_confidence=0.5,
                            min_tracking_confidence=0.8,
                            model_name='Cup') as objectron:
    while cap.isOpened():
        success, image = cap.read()

        start = time.time()

        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        image.flags.writeable = False

        results = objectron.process(image)
        image.flags.writeable = True

        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)

        if results.detected_objects:
            for id, detected_object in enumerate(results.detected_objects):
                mp_drawing.draw_landmarks(image, detected_object.landmarks_2d, mp_objectron.BOX_CONNECTIONS)
                mp_drawing.draw_axis(image, detected_object.rotation, detected_object.translation)
                print(id,detected_object.translation)
```

```
 [ 0.35160527  0.23149736 -0.69453132]]
0 [-0.03361808  0.11367079 -0.46748921]
0 [[-0.94500697  0.09011599 -0.31026581]
 [-0.09716645  0.97073817  0.64800233]
 [ 0.31228223  0.22259097 -0.6955775 ]]
0 [-0.03416065  0.11429918 -0.46826375]
0 [[-0.93977284  0.09564599 -0.31140354]
 [-0.10154617  0.96172053  0.64217144]
 [ 0.32636625  0.25679827 -0.70045948]]
0 [-0.03393837  0.11453009 -0.46840832]
0 [[-0.94081384  0.09194226 -0.30640465]
 [-0.09753394  0.96843332  0.66094196]
 [ 0.32458678  0.23169713 -0.68503439]]
```

Figure 6.3: *Objectron* testing for object of type *Cup* in *PyCharm* environment

The *MediaPipe* framework would be significant upgrade to the current perception module but it would need to be trained for objects manipulated within the IOC Robotic Lab, currently a complying *MediaPipe* dataset exist only for plastic bottles.

During the attempt to integrate *MediaPipe* with ROS, software limitations were encountered:

- *Bazel* [8] compiler would need to be installed in the version 4.7.2+
- *Bazel* version would need to comply to the *gcc* at the computer where it is being installed (as some PCs in the lab are running *gcc 8* while other use *gcc 10*)
- Additional *MediaPipe* packages would need to be installed

and in order to overcome them certain changes would need to be made to one of the computers by the administrators at IOC-UPC.

7 Cost Analysis

This project took 32 weeks to complete, averaging four hours each day in the period from October 2021 until February 2022. During Christmas and the whole month of January, the student was working from home and thus only his working costs and personal laptop consumption are included in this periods estimates. With five working days per week, the total project hours are 450, of which 120 are dedicated to the personal laptop setup and student work hours.

A PC workstation, two Microsoft Kinect V2 sensors, one OAK-D camera, and a personal laptop comprise the hardware. Because the project was developed at the Institute of Industrial and Control Engineering's (IOC-UPC) Robotics Lab, the depreciation cost of the workstation and all sensors and cameras utilized has been included. The useful life of a workstation and a personal laptop is estimated to be five years when used for ten hours a day, five days a week. This results in an 13050 hours of useful life. The workstation PC is used for 50% of the project hours spent in the lab, and the personal laptop is utilized for the other 50% of the project hours spent in the lab and at other places. The student's working hours are regarded to be equal to the overall project hours. ETSEIB proposes that students be paid an hourly rate of 8 euros. Supervision and meetings with the project's director and other laboratory employees will be considered 45 hours in total, at an average cost of 30 €/h.

Electricity usage is calculated for each hardware component that was utilized. The average cost of electricity in 2021 is assumed to be 0.21 €/kWh. The personal laptop's projected energy usage is 120.5W, which equates to 27 kWh during a 225-hour period. The workstation PC's projected energy usage is 550 W, which equates to 123.750 kWh during a 225-hour period. Following that, the Kinect Cameras used 12 W each for 60 hours, which equals 0.72 kWh in total.

COST CALCULATION	Item	Fixed Cost (€)	Life Expectancy	Variable Cost (€)	Utilization Time	Cost to Project (€)
Hardware Equipment	Workstation PC	1,400	13,050	0.107	225	24.075
	Personal Laptop	700	13,050	0.0536	225	12.06
	Kinect Sensors	600	26,280	0.022	120	2.64
Electricity Consumption	Workstation PC	-	-	0.042	225	9.45
	Personal Laptop			0.014	225	3.15
	Kinect Sensors			0.0024	120	0.288
Student Working Hours	-	-	-	8	450	3,600
Supervisor Working Hours	-	-	-	30	45	1,350
Total Cost						4,992.2

Table 1: Cost calculations

Calculations from Table 1 above add up to a total project cost of 4,992.2 €.

8 Environmental and Social Impact

This section debates the social and environmental impacts of robot perception and computer vision topics considered and worked on during this project:

8.1 Social Impact

Throughout history and culture, robots have been envisioned as coworkers. As a result of this idea, a new family of devices called collaborative manufacturing robots enables human and robot employees to collaborate on industrial activities. Their introduction enables us to get a greater understanding of how people interact with and perceive a robot "coworker" in a real-world situation, which may help shape the design of these goods.The findings of a paper by Alison Sauppe and Bilge Mutlu [24] indicates that, even in this high-risk industrial environment, employees see the robot as a social entity and depend on clues to comprehend its activities, which is crucial for workers to feel comfortable while working near the robot. These results advance our knowledge of human-robot interactions in real-world contexts and have significant design implications. Computer vision itself allows robots to understand human cues as well and adapt their behaviour in order to generate a safer environment and more pleasant workspace for everybody.

According to economists, the wide societal benefit of increasing computer vision should be an increase in people's material well-being. Computer vision development comprises incorporating perceptual and interpretation knowledge into new gadgets that expand the extent and depth of human capacity. Although, automating repetitive work using computer vision would burden society with increased population and unemployment, I think people are still the 'ultimate resource. [2]

Some tasks are inevitably performed more efficiently by robots and computer vision is one of the main fields of research widening this spectrum of tasks. Even in this research paper, the aim was to replace data input by humans with data obtained from the perception module.

Another interesting point is that with enhancements in computer vision and robot reasoning, they will begin to understand the world in a more human way and this might become a privacy issue.

8.2 Environmental Impact

Advancements in the field of computer vision will inevitably lead to greater capabilities of industrial and humanoid robots and thus more robots will be produced and operated every year. Industrial robots are substantial consumers of electrical energy but as they are generally able to work in the dark and without heating or cooling, the overall costs of fully automated manufacturing plants could be lower. Moreover, with the implementation of renewable energy sources energy consumption this might not be a thing to worry about.

Bigger problems could arise in the manufacturing and recycling of batteries for mobile robots and vehicles as we are yet to enter a phase of mass production of such batteries to be used every day.

In the scope of this project tests which would require many trials and errors are now eliminated and agents can locate objects successfully, thus less electrical energy is wasted.

Conclusions

The developed perception module manages to fully accomplish all of the objectives set at the project initialisation:

- Successfully identifies objects, localizes them and determines their pose using fiducial markers
- Provides agents with the data needed to generate PDDL files, regarding objects within the working area
- Makes use of ROS framework to publish obtained data and provide a service, thus making it readily available for agents within the laboratory system
- Is of a modular design and provides an intuitive way to add, remove or exchange cameras between working areas
- Is a good basis for expansion and addition of other pose estimation methodologies

With regards to the perception module object pose estimation using fiducial markers, the following could be considered for improvement by future work:

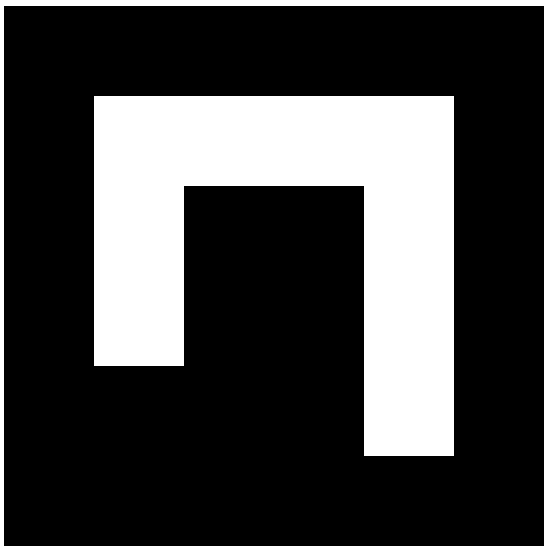
One of the issues discovered while testing the perception module was that the *Kinect* sensor was placed too high for robust fiducial marker detection at *table_YuMi*. In order to solve this problem, the following ideas are proposed:

1. Positioning the camera closer to the working area
2. Using larger fiducial markers
3. Using a different library of fiducial markers

The first option has to be rejected as currently the camera is placed just out of reach for robot *YuMi*s manipulators and thus if the camera was to be brought closer, a chance of it getting hit and damaged would exist.

Regarding the second option, this could be feasible but only for larger objects as the fiducial markers currently being used (size=0.45cm) exactly fit the top of a cans and wooden boxes being manipulated by the robots.

Finally, each *Aruco* marker library divides the size of a marker into white or black boxes (ex. 4x4, 7x7, 8x8 etc), a combination of which is unique for each marker. If the division is higher (Figure 8.1b), the number of possible combinations and unique markers rises but in the case of the object data base at IOC Laboratory there isnt a need for a large number of unique markers and thus the whole system was to switch to *Aruco Dictionary 4x4* (Figure 8.1a) the perception module accuracy would increase.


(a)

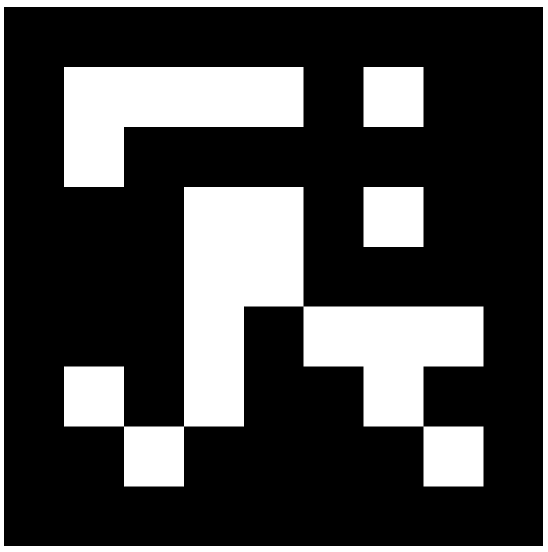

(b)

Figure 8.1: *Aruco* markers from 2 different libraries.

Currently, agents within the Robotic lab are localized by mapping the area and using the room corner (*/world*) frame as global reference frame. As working area local origins have been manually determined with respect to the global reference frame, the accuracy at which the robots are provided with object locations rely on these transforms. Due to this, pose estimation data tends to be less accurate. A solution to overcome this issue would be to have the robots perform the same localization as the cameras do, using fiducial markers printed and fixed on tables, each time they approach a working area to perform a manipulation task.

Many possibilities exist for expansion on the developed framework basis in order to provide more robust object localization and pose estimation along with other functionalities, some of which have been discussed in this project.

In order to incorporate *MediaPipe Objectron*, currently identified steps have been described in section 6 while the incorporation of OAK-D algorithms could be done by installing required packages and using the OAK-D ROS bridge [19] provided by the *luxonis* team, but as mentioned previously, this has to be done by one of the administrators at IOC-UPC as students do not have access.

Acknowledgements

First, I am grateful to my supervisor, Prof. Dr. Jan Rosell Gratacòs for his invaluable advice, continuous support, and patience during my Master's Thesis. I would also like to thank Prof. Leopold Palomo Avellaneda for his technical support on my study. I would like to thank all students at the Robotics Lab of the Institute of Industrial and Control Engineering undergoing their Master's or PhD research. It is their kind help and support that have made my final at ETSEIB wonderfull. Finally, I would like to express my gratitude to my parents and sister, without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my studies.

Appendix

Listing 2: ns_marker_publisher.launch file

```
<launch>

    <arg name="markerSize"       default="0.045"/>     <!-- in m -->
    <arg name="cam_loc"              default="kinect_Chess"/>
    <arg name="ref_frame"        default="kinect2_rgb_optical_frame"/>  <!--
        leave empty and the pose will be published wrt param parent_name -->

    <node pkg="aruco_ros" type="marker_publisher" name="aruco_marker_publisher">
        <remap from="/camera_info" to="/$(arg cam_loc)/kinect2/hd/camera_info"
            />
        <remap from="/image" to="/$(arg cam_loc)/kinect2/hd/image_color_rect" />
        <param name="image_is_rectified" value="True"/>
        <param name="marker_size"         value="$(arg markerSize)"/>
        <param name="reference_frame"     value="$(arg ref_frame)"/>   <!-- frame
             in which the marker pose will be refered -->
        <param name="camera_frame"        value="kinect2_rgb_optical_frame"/>
    </node>

    <node pkg="tf2_ros" type="static_transform_publisher" name="
        camera_broadcaster" args="0.0818692 -0.172355 0.809997 0.996293
        -0.000445324 0.00422562 -0.0859181 /$(arg cam_loc)/world /$(arg cam_loc)
        _rgb_optical_frame"/>


</launch>
```

Listing 3: ns_marker_publisher.launch file

```

<launch>


    <arg name="camera1_id" default="004625445247"/>
    <arg name="camera2_id" default="007266745247"/>
    <arg name="camera3_id" default="null"/>

    <arg name="camera1_name" default="kinect_YuMi"/>
    <arg name="camera2_name" default="kinect_Chess"/>
    <arg name="camera3_name" default="null"/>





    <group ns="$(arg camera1_name)">

    <include file="$(find tablesens)/launch/kinect2_bridge_hq.launch">
    <arg name="sensor" value="$(arg camera1_id)"/>
    </include>

    <include file="$(find aruco_ros)/launch/ns_marker_publisher.launch">
    <arg name="cam_loc" value="$(arg camera1_name)"/>
    </include>


```

```
<node name="aruco_broadcaster" pkg="aruco_broadcaster" type="aruco_broadcaster" output="screen"/>
<param name="camera_loc"          value="$(arg camera1_name)"/>
<rosparam file="$(find aruco_broadcaster)/config/table_YuMi.yaml" command="load"/>
<!-- <remap from="/aruco_marker_publisher/markers" to="/$(arg camera1_name)/aruco_marker_publisher/markers" />
-->

</group>





<group ns="$(arg camera2_name)">

<include file="$(find tablesens)/launch/kinect2_bridge_hq.launch">
<arg name="sensor" value="$(arg camera2_id)"/>
</include>

<include file="$(find aruco_ros)/launch/ns_marker_publisher.launch">
<arg name="cam_loc" value="$(arg camera2_name)"/>
</include>

<node name="aruco_broadcaster" pkg="aruco_broadcaster" type="aruco_broadcaster" output="screen"/>
<param name="camera_loc"          value="$(arg camera2_name)"/>
<rosparam file="$(find aruco_broadcaster)/config/pl2esaii.yaml" command="load"/>
<!-- <remap from="/$(arg camera2_name)/aruco_marker_publisher/markers" to="/aruco_marker_publisher/markers" />
-->

</group>



<!--
<group ns="$(arg camera3_name)">

<include file="$(find tablesens)/launch/kinect2_bridge_hq.launch">
<arg name="sensor" value="$(arg camera3_id)"/>
</include>

<include file="$(find aruco_ros)/launch/ns_marker_publisher.launch">
<arg name="cam_loc" value="$(arg camera3_name)"/>
</include>

<node name="aruco_broadcaster" pkg="aruco_broadcaster" type="aruco_broadcaster" output="screen"/>
<param name="camera_loc"          value="$(arg camera3_name)"/>
<rosparam file="$(find aruco_broadcaster)/config/pl2esaii.yaml" command="load"/>
<remap from="/$(arg camera3_name)/aruco_marker_publisher/markers" to="/aruco_marker_publisher/markers" />


</group>
-->
```

```


    <node pkg="tf2_ros" type="static_transform_publisher" name="tf_between_kinect1_world" args="3 2 0 0 0 0 /world /$(arg camera1_name)/world"/>
    <node pkg="tf2_ros" type="static_transform_publisher" name="tf_between_kinect2_world" args="4 3 0 0 0 0 /world /$(arg camera2_name)/world"/>

    <!-- <node pkg="tf2_ros" type="static_transform_publisher" name="tf_between_kinect2_world" args="4 3 0 0 0 0 /world /$(arg camera2_name)/world"/> -->

</launch>
```

Listing 4: ns_marker_publisher.launch file

```

<launch>


    <arg name="camera_1" default="Chess"/>
    <arg name="camera_2" default="YuMi"/>
    <arg name="camera_3" default="null"/>
    <arg name="camera_4" default="null"/>


    <node name="perception_server" pkg="perception_module_pkg" type="perception_service_server" output="screen"/>
    <param name="camera_1"        value="$(arg camera_1)"/>
    <param name="camera_2"        value="$(arg camera_2)"/>
    <param name="camera_3"        value="$(arg camera_3)"/>
    <param name="camera_4"        value="$(arg camera_4)"/>


</launch>
```

Bibliography

[1] Fatma Nur Arabaci. Perception and reasoning for automatic configuration of task and motion planning problems, September 2021.

[2] Howard Baetjer. Social impact of computer vision. In David H. Schaefer and Elmer F. Williams, editors, *25th AIPR Workshop: Emerging Applications of Computer Vision*, volume 2962, pages 97 – 101. International Society for Optics and Photonics, SPIE, 1997.

[3] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcis Palomeras, Natalia Hurtos, and Marc Carreras. Rosplan: Planning in the robot operating system. *Proceedings of the International Conference on Automated Planning and Scheduling*, 25(1):333–341, Apr. 2015.

[4] Vaishnav Chunduru, Mrinalkanti Roy, Rajeevlochana G Chittawadigi, et al. Hand tracking in 3d space using mediapipe and pnp method for intuitive control of virtual globe. In *2021 IEEE 9th Region 10 Humanitarian Technology Conference (R10-HTC)*, pages 1–6. IEEE, 2021.

[5] Guoguang Du, Kai Wang, Shiguo Lian, and Kaiyong Zhao. Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: a review. *Artificial Intelligence Review*, 54(3):1677–1734, 2021.

[6] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):265–293, 2021.

[7] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[8] Google. bazelbuild/bazel, 2021. https://github.com/bazelbuild/bazel, Last accessed on 27-01-2022.

[9] Diego Guffanti, Alberto Brunete, Miguel Hernando, Javier Rueda, and Enrique Navarro Cabello. The accuracy of the microsoft kinect v2 sensor for human gait analysis. a different approach for comparison with the ground truth. *Sensors*, 20(16), 2020.

[10] Yisheng He, Wei Sun, Haibin Huang, Jianran Liu, Haoqiang Fan, and Jian Sun. Pvn3d: A deep point-wise 3d keypoints voting network for 6dof pose estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11632–11641, 2020.

[11] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5):503–535, 2009. Advances in Automated Plan Generation.

[12] Michail Kalaitzakis, Sabrina Carroll, Anand Ambrosi, Camden Whitehead, and Nikolaos Vitzilaios. Experimental comparison of fiducial markers for pose estimation. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 781–789, 2020.

[13] Anis Koubâa et al. *Robot Operating System (ROS).*, volume 1. Springer, 2017.

[14] Leopold Palomo-Avellaneda. aruco_broadcaster, 2021. https://gitioc.upc.edu/labs/

aruco_broadcaster/-/tree/master, Last accessed on 27-01-2022.

[15] Vincent Lepetit, Julien Pilet, and Pascal Fua. Point matching as a classification problem for fast and robust object pose estimation. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–II. IEEE, 2004.

[16] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for perceiving and processing reality. In *Third Workshop on Computer Vision for AR/VR at IEEE Computer Vision and Pattern Recognition (CVPR) 2019*, 2019.

[17] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for building perception pipelines, 2019.

[18] Luxonis dev team. Luxonis - embedded performant spatial ai and cv, 2021. https://github.com/luxonis, Last accessed on 27-01-2022.

[19] luxonis dev team. luxonis/depthai-ros, 2021. https://github.com/luxonis/depthai-ros, Last accessed on 27-01-2022.

[20] Luis A. Mateos. Apriltags 3d: Dynamic fiducial markers for robust pose estimation in highly reflective environments and indirect communication in swarm robotics, 2020.

[21] Pal Robotics. aruco_ros, 2021. https://github.com/pal-robotics/aruco_ros, Last accessed on 22-01-2022.

[22] Amirmohammad Radmehr, Milad Asgari, and Mehdi Tale Masouleh. Experimental study on the imitation of the human head-and-eye pose using the 3-dof agile eye parallel robot with ros and mediapipe framework. In *2021 9th RSI International Conference on Robotics and Mechatronics (ICRoM)*, pages 472–478. IEEE, 2021.

[23] Satya Mallick and Kukil. Introduction to oak-d and depthai, 2021. https://learnopencv.com/introduction-to-opencv-ai-kit-and-depthai/?utm_source=rss&utm_medium=rss&utm_campaign=introduction-to-opencv-ai-kit-and-depthai, Last accessed on 23-01-2022.

[24] Allison Sauppé and Bilge Mutlu. The social impact of a robot co-worker in industrial settings. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, page 3613–3622, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Seedstudio team. Getting started with mediapipe on reterminal, 2021. https://wiki.seeedstudio.com/reTerminal_ML_MediaPipe/, Last accessed on 27-01-2022.

[26] Mike Stilman. Task constrained motion planning in robot joint space. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3074–3081, 2007.

[27] Carlo Tomasi. Camera calibration. 2002.

[28] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*, 2018.

[29] Chen Wang, Danfei Xu, Yuke Zhu, Roberto Martín-Martín, Cewu Lu, Li Fei-Fei, and Silvio Savarese. Densefusion: 6d object pose estimation by iterative dense fusion. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3343–3352, 2019.

[30] Thiemo Wiedemeyer. IAI Kinect2. https://github.com/code-iai/iai_kinect2, 2014 – 2015. Accessed June 12, 2015.

[31] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. Mediapipe hands: On-device real-time hand tracking, 2020.

[32] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE MultiMedia*, 19(2):4–10, apr 2012.

ETSEIB