Software Architecture For Robotics Project

HRP-Teleoperation

Sensors-Controller-Robot Simulation
Kinect-Unity3D-Oculus Interface

Students: Noel Alejandro Avila Campos, Nicola De Carli, Angelica Ginnante (Controller and Smartwatch module)
Adam Berka, Nicolas Dejon (Leap Motion Module)
Enrico Casagrande, Alberto Ghiotto (Kinect-Unity3D-Oculus Interface)
Francesca Cantoni, Claudio Curti, Alberto Grillo (Kinect Module)
Tutor: Alessandro Carfi
Supervisor: Alessandro Carfi
Year: 2018 – 2019

Table of Contents

1. Objective of the Project

This project can be divided into two main part. The first one concerns the control of the robot through the usage of the different kind of sensors; instead the second one is focused on the creation of a 3D point cloud and its visualization in an Oculus visor.

More in details, the first part of the project has the aim of controlling a Husqvarna Automower using the arm orientation measured by three different sensors: a smartwatch, a Kinect and a leap motion sensors. The user has to wear the smartwatch and gets in front of the Kinect and the leap motion so that they can analyze the movement of the right arm and move the robot. The orientation of the wrist with respect to the axis along the forearm (roll angle) determines the angular velocity of the robot, anti-clockwise to go left, clockwise to go right. The orientation of the arm with respect to an axis parallel to the ground (pitch angle) determines the linear velocity of the robot, hand over the axis for positive velocity, under for negative velocity (reverse).

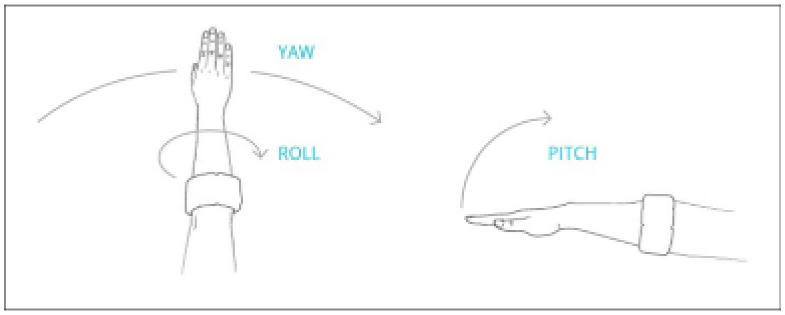


Figure 1 - Roll-Pitch-Yaw Scheme

The objective of the second section of the project is to create a 3D point cloud map from the images acquired by a Kinect in a ROS environment on Linux, to transmit it to a Windows-based Unity project which will tweak and improve the map in order to make it more user-friendly before sending it to the Oculus visor worn by the user. The Kinect could be even mounted on a moving robot in order to create a real-time dynamic map of its surrounding. Perceiving a space without effectively being there and being allowed to virtually wonder inside it is a hard challenge to face, but in the end profitable. There are multiple scenarios that could benefit from the deployment of such a technology: emergency forces could explore spaces occluded or dangerous for humans, people unable to walk could rely on this way of interacting with the environment or with loved ones far from them, the army could exploit it in war zones and many others.

In this report are available all the information about the project developed. How the architecture and the modules were thought and implemented. All the prerequisites that are necessary both on the hardware side and on the software side. What has to be downloaded and installed and how to run the programs.

In the end it can be found all the results achieved and also some recommendations for the future users, helpful to juggle themselves with the different phases of the project.

2. The System’s Architecture

2.1. Overall Architecture

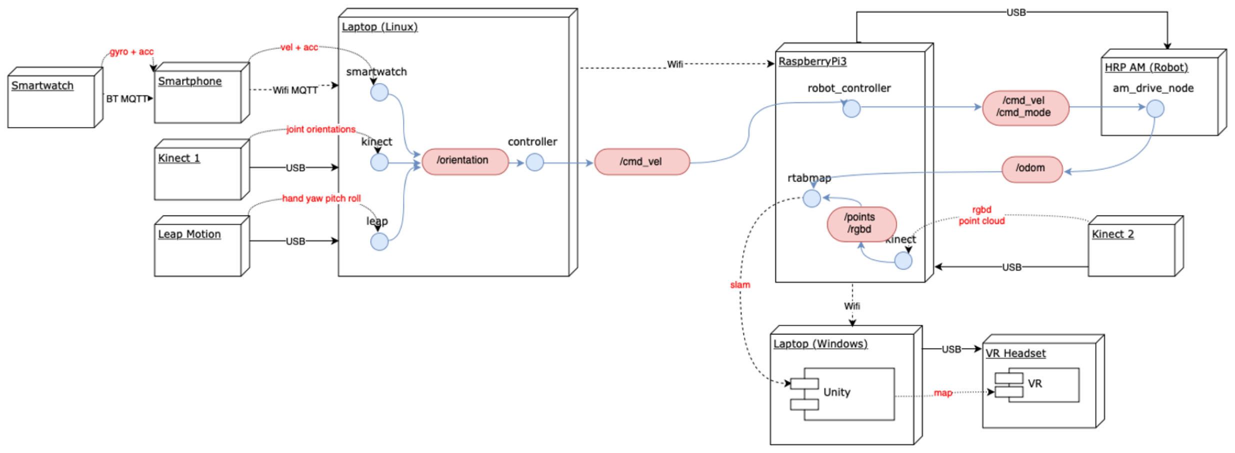


Figure 2 - Complete Project Architecture

2.1.1. First Section Architecture

The three sensors (smartwatch, Kinect and leap motion) get information regarding the arm and interface to the PC using their respective drivers. Since the controller receives RPY angles (roll-pitch-yaw) that are easier to interpret than quaternions, and therefore easier to remap in velocities using gains, there is the necessity for three adapter nodes between the drivers and the controller. These adapter nodes receive the orientation data from the respective sensors and in some way, depending on the sensor, the data are converted to RPY data and then sent to the controller. Eventually the controller node considering orientation data from the sensors provide the desired velocity to the robot.

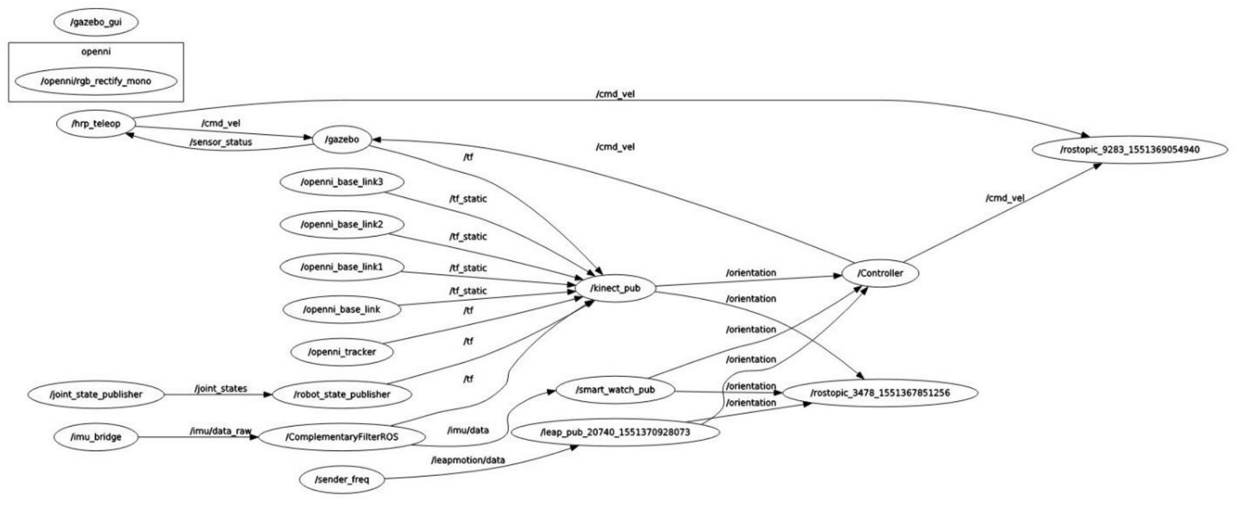


Figure 3 - RQT Graph (sensors to simulation)

Above the *rqt* graph of the whole project with the three sensors connected to the controller is reported. It can be seen that the controller send the actual velocities to the simulation on GAZEBO.

2.1.2. Second Section Architecture

The architecture implementation is conceptually quite trivial. As represented in figure 1, it can be seen how the two principal modules share data with a simple WebSocket. In addition to them there also is an Oculus VR device used for the final visualization.

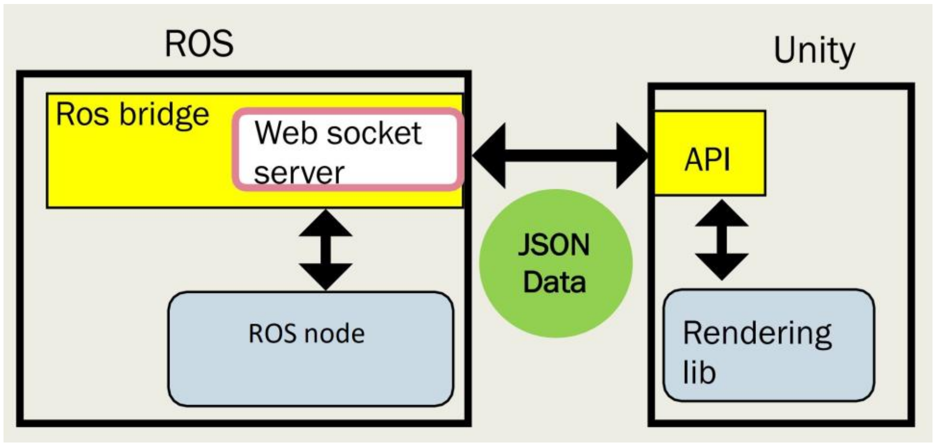


Figure 4 - Architecture ROS-Unity

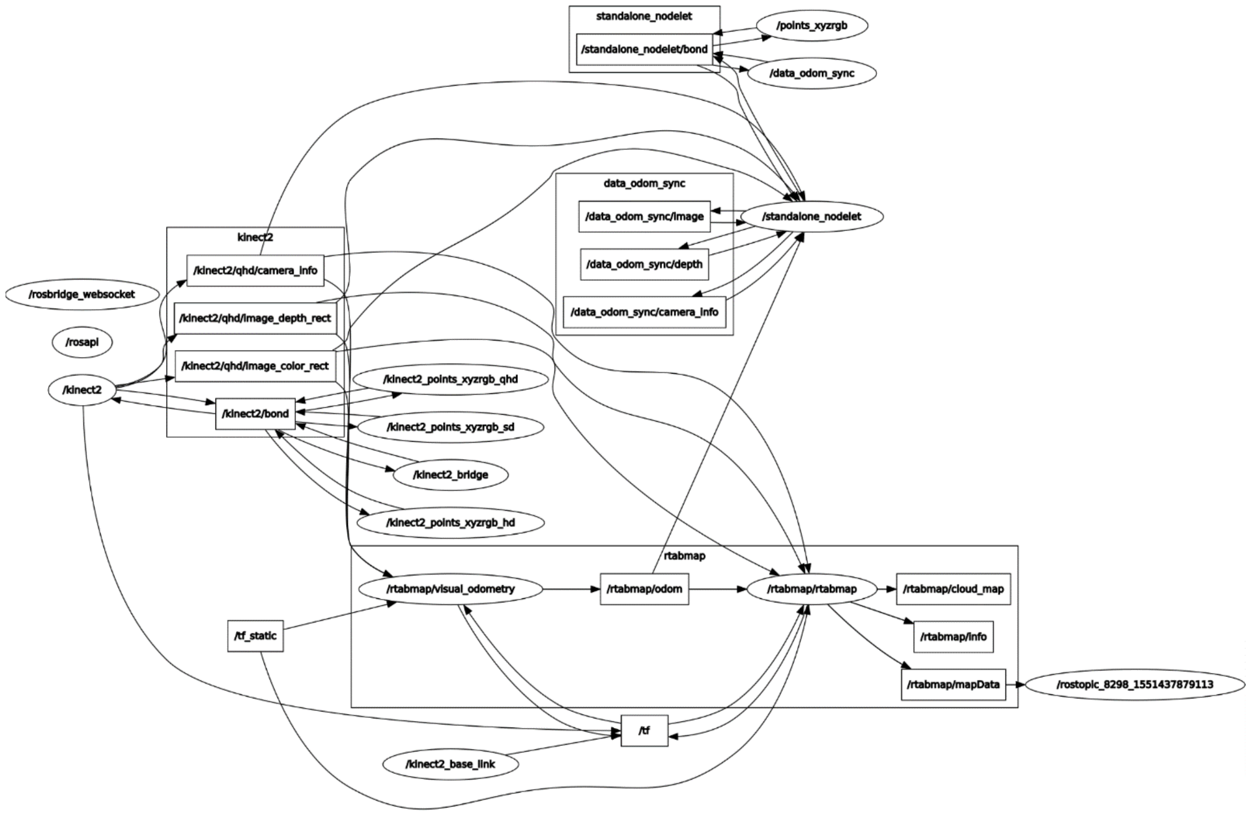


Figure 5 - RQT Graph (3D point cloud part)

2.2. Description of the Modules

On the controller side there are four main modules which communicate together and we will focus on their inputs, outputs and internal working. Instead, regarding the 3D point cloud, we will analyze the SLAM approach and the Unity visualization of a 3D point cloud map.

2.2.1. Controller Module

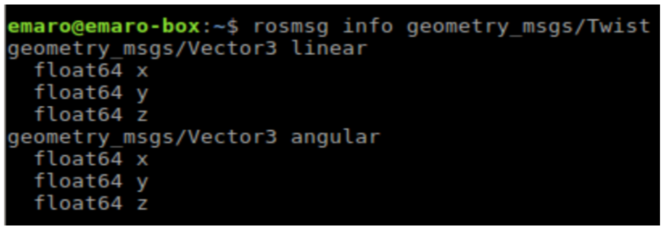
The controller module called *Controller* takes as input the data sent by the other main modules connected to our three sensors. The main aim of this module is remapping the orientation data provided by the sensors to linear and angular velocities using a weighted average (weights are thought to take into account the different reliability of the sensors). These velocities are eventually sent to the Husqvarna Automower, publishing on the topic */cmd_vel*.

The module read the data published on the topic */orientation* by the three modules, called *smart_watch_pub*, *kinect_pub* and *leap_pub*, and saves them in different variables based on the sender. By this action it also takes the time when the new data arrive, saving it in a variable related to the module that sent it, then sets the flag always related to the publisher module to one and if before the flag was zero it increases a sensor counter, a variable that takes into account how many sensors are connected, so that if there aren't it is generated a zero velocity message for the robot to avoid undesired behaviors when all the sensors disconnect. The flag, instead, is used to indicate if the sensor is active or not, when the flag was zero and is set to one sensor counter is increased, when it becomes zero sensor counter is decreased.

In the main part, the controller transforms roll and pitch data acquired into the corresponding linear and angular velocity for each sensor according to the data measured by the specific sensor (Kinect's roll angle is not considered because this sensor doesn't provide this data); then compute the weighted average of the different velocities.

In advanced another checking is implemented in the controller, infact if the same message is sent for more than four times or no data arrives for more than one second from a module it means that probably the sensor was disconnected and so its linear and angular velocities are set to zero.

Once computed the actual velocities to send to the robot, these are published on the topic */cmd_vel* through a message type *geometry_msgs::Twist*. The linear velocity is published only on the variable *x* of the linear part, instead the angular one is published on the *z* of the angular part. The structure of *geometry_msgs::Twist* message is reported below.

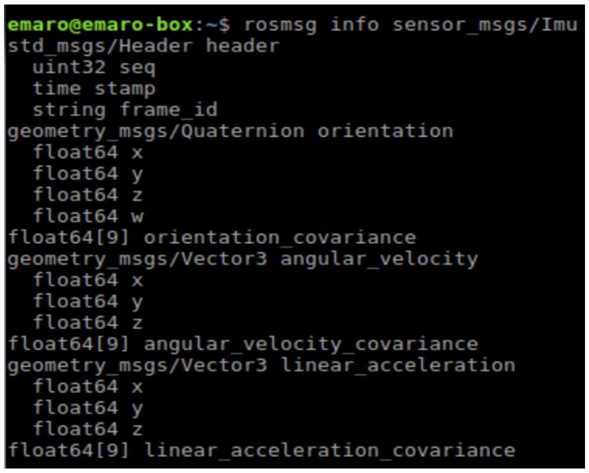


```
emaro@emaro-box:~$ rosmsg info geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 6 - geometry_msgs::Twist

2.2.2. Smartwatch module

The smartwatch module called *smart_watch_pub* takes as input the data sent by the smartphone connected to the smartwatch. These data are of the type *sensor_msgs::Imu* composed by 6 fields, as can be seen in the below figure.



```
emaro@emaro-box:~$ rosmsg info sensor_msgs/Imu
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
  float64 x
  float64 y
  float64 z
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
  float64 x
  float64 y
  float64 z
float64[9] linear_acceleration_covariance
```

Figure 7 - sensor_msgs::Imu

For our project the relevant field is *geometry_msgs::Quaternion orientation*, in order to obtain the positioning of the arm in an RPY representation; but the smartwatch provides only the angular velocity and the linear acceleration.

Through the use of the *complementary_filter_node*, downloaded from the repository, it can be obtained the orientation data, published on the topic */imu/data*, giving as input the actual angular velocity and linear acceleration on the topic */imu/data-raw*.

Once get the right orientation, the developed node *smart_watch_pub* read it on the topic */imu/data* and convert them from *geometry_msgs::Quaternion* to *tf::Quaternion*. Thanks to the function *getRPY* of the class *tf::Matrix3x3*, an analogous RPY representation is given back giving as input *tf::Quaternion orientation* and then it is published on the topic */orientation*.

2.2.3. Kinect Module

This module manages the Kinect sensor data flow and tracks the arm computing its pitch angle, which will be used to determine the linear velocity to send to the robot.

This module uses as input the transformation matrices between shoulder and elbow frames each one with respect to the basis frame placed on the Kinect depth camera, computed from the corresponding topics given by *openni_tracker*. The *openni_tracker* library publishes the Kinect data of the user's pose as a set of transforms (/tf) on several topic each one representing a joint of the body.

Then, using the origin distance vector of these two transformations, the module is able to compute a vector joining elbow and shoulder frames and using a vertical reference vector we compute the pitch angle of the user's arm.

The output consists of customized RPY ROS message but, for practical and implementation reasons, we only set the pitch angle field . Indeed the roll and the yaw angles cannot be estimated with good accuracy by the Kinect so we decided to send zero values to the controller in those fields.

It's important to specify that the pitch values that are sent to the controller have a $-\frac{pi}{2}$ offset in order to comply with the angles produced by the other two sensors and achieve, in such a way, a smoother conversion to velocity values.

The RPY message is published on the topic */orientation* where it will be read by the controller with the other sensors data.

2.2.4. Leap Motion Module

This module has the initial goal of sending the yaw, pitch and roll angles captured by the Leap Motion to the node controller through the topic */orientation*. This module takes as input hands and fingers motions. Then the images taken by two cameras are analyzed to reconstruct a 3D representation of what the device sees. The tracking algorithms interpret the 3D data and infer the pitch and the roll of the nearest objects (hands and fingers). The thresholds set to interpret the hand gestures as commands are:

- Pitch Low: -50 to -20 degrees
- Roll Right: -100 to -50 degrees
- Roll Left: 80 to 130 degrees

The outputs are the commands for the robot. When the gesture command is below the lower pitch threshold and within the pitching range, the value 0.5 (move forward) is sent to the robot as a linear command. When it is above the horizontal position, the value 0 (stop) is sent instead. When the gesture command is below the lower threshold of the roll and within the rolling range, the value -0.5 (turn right) is sent to the robot as an angular command. The value 0.5 (turn left) is sent.

2.2.5. SLAM Approach

This module is implemented on the machine running Linux by exploiting an existing package available on the ROS documentation, a ROS wrapper of RTAB-Map (Real-Time Appearance-Based Mapping). This package can be used to generate a 3D point clouds of the environment as was done in this project. For more information visit the following links:

RTAB-Map

ROS Wiki

GitHub Repo

As the documentation shows, RTAB-Map is composed of different nodes but this project will only require the use of the one called "rtabmap". In order for the kinect 2 v2 to work as needed it is necessary to install the libfreenect2 and the IAI Kinect2 drivers.

To establish the connection between the two machines it was chosen rosbridge which provides a JSON interface to ROS, allowing any client to send JSON to publish or subscribe to ROS topics, call ROS services, and more.

The inputs are the images acquired by the Kinect 2 v2 in RGB-D format, which is a combination of an RGB image and its corresponding depth image. RTAB-Map takes the RGB-D images and publishes them as ROS messages under different topics as shown in the documentation here.

The outputs are the messages published on the different ROS topics, in particular we are interested in the */rtabmap/mapData* topic on which is published the 3D point cloud map of the environment. The map can be visualized in Rviz, which will be launched automatically with the required settings to visualize the map, in order to make a pre-check before starting to send the data stream to Unity via WebSocket.

2.2.6. Unity visualization of a 3D point cloud map

This module is implemented on the machine running windows and provides a platform on which visualize the point cloud map put together by the SLAM module. Unity will visualize the map and provide the connection with the Oculus with all the relative scripts to link the game camera with the movement of the Oculus.

In order to implement the connection another library has been used: RosbridgeLib. It contains C# scripts designed to make possible to communicate with ROS using Rosbridge, such as Subscriber, Publisher and many standard messages, in addition to a general script to establish the connection.

Unity receives the ROS messages of the */rtabmap/mapData* topic via the WebSocket.

All the functionalities described above are integrated into a custom script which connects to the ROS node, receives the data and then converts the point cloud atomic elements into 3D cubes (composed by meshes), in order to make the scene more intuitive and user-friendly.

The cubes are then colored using a particular shader, the “GUI/text shader”, which allows to maintain the original color registered by the RGB camera of the Kinect for every point in space.

The output will be the 3D point cloud map visualized on the Unity scene, composed this time not by points but by colored cubes.

Everything will also be displayed on the connected oculus, giving the possibility to the user to explore the scene just by moving around his head.

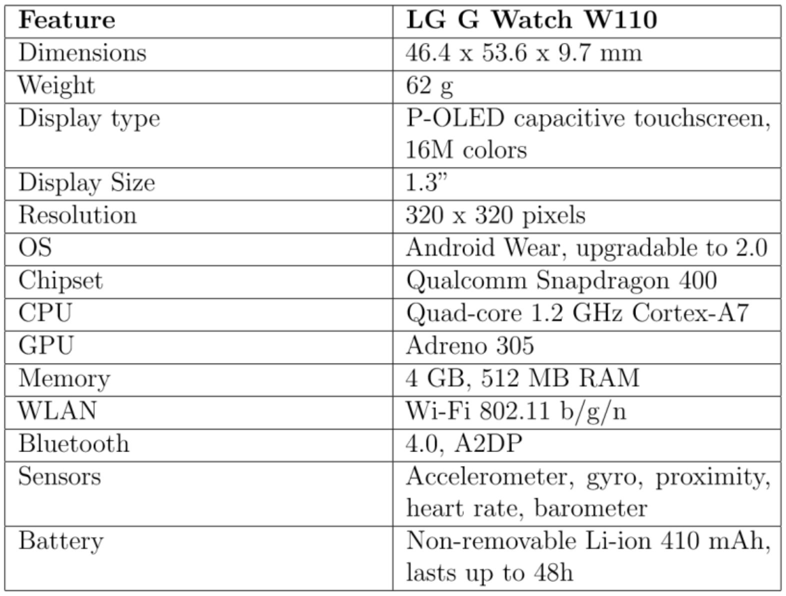
3. Implementation

3.1. Prerequisites

3.1.1. HARDWARE

- LG G Watch R W110

The sensors present on the watch are: accelerometer, gyroscope, proximity, heart rate and barometer. It has Bluetooth 4.0,A2DP and WLAN Wi-Fi 802.11 b/g/n.

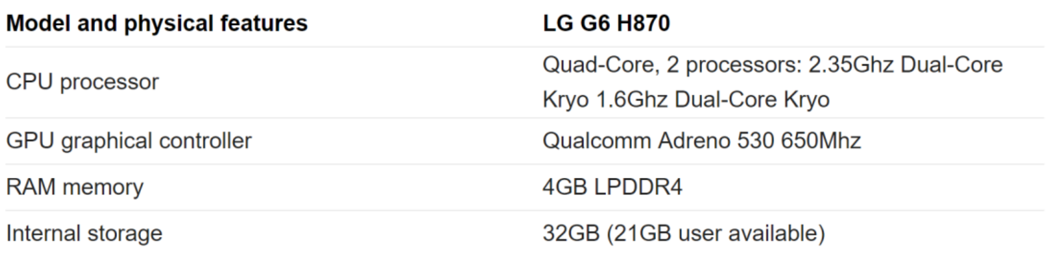


Feature	LG G Watch W110
Dimensions	46.4 x 53.6 x 9.7 mm
Weight	62 g
Display type	P-OLED capacitive touchscreen, 16M colors
Display Size	1.3”
Resolution	320 x 320 pixels
OS	Android Wear, upgradable to 2.0
Chipset	Qualcomm Snapdragon 400
CPU	Quad-core 1.2 GHz Cortex-A7
GPU	Adreno 305
Memory	4 GB, 512 MB RAM
WLAN	Wi-Fi 802.11 b/g/n
Bluetooth	4.0, A2DP
Sensors	Accelerometer, gyro, proximity, heart rate, barometer
Battery	Non-removable Li-ion 410 mAh, lasts up to 48h

Figure 8 - Smartwatch

- LG G6 H870

It uses the operative system Android 7.0 LG UX 6.0 UI Nougat and Bluetooth 4.2 with A2DP/LE/aptX.



Model and physical features	LG G6 H870
CPU processor	Quad-Core, 2 processors: 2.35Ghz Dual-Core Kryo 1.6Ghz Dual-Core Kryo
GPU graphical controller	Qualcomm Adreno 530 650Mhz
RAM memory	4GB LPDDR4
Internal storage	32GB (21GB user available)

Figure 9 - Smartphone

- Kinect

The Kinect is a motion sensing device, developed initially only for gaming purposes as it allowed to interact with the console without the use of any controller but only with gestures. It allows the user to control and interact with the PC and it's capable of providing: full-body 3D motion capture, facial and voice recognition.

Feature	Microsoft Kinect v1	Microsoft Kinect v2
Hardware Compatibility	Stable work with various hardware models	Stable work with various hardware models
USB Standard	USB 2.0	USB 3.0
Size	33 x 16,5 x 12,7 cm	25 x 6,6 x 6,7 cm
Weight	1 kg	1 kg
Power Supply	USB + ACDC power supply	USB + ACDC power supply
Power Consumption	12 watts	12 watts
Vertical tilt range	27°	27°
Field of View	57° horizontal, 43° vertical	70° horizontal, 60° vertical
Frame rate	30 frame per second (FPS)	30 frame per second (FPS)
Color Camera	640 x 480 pixels FPS 30	1920 x 1080 pixels FPS 30
Depth Camera	320 x 240 pixels FPS 30	512 x 424 pixels FPS 30
Maximum Depth Distance	4 m	4.5m
Minimum Depth Distance	80 cm	50 cm
OS Platform Support	Xbox 360 Microsoft Windows Linux MacOS	Xbox One Microsoft Windows Linux MacOS
Programming Language	C++/C# (Windows) C++ (Linux) Java	C++/C# (Windows) C++ (Linux) Java

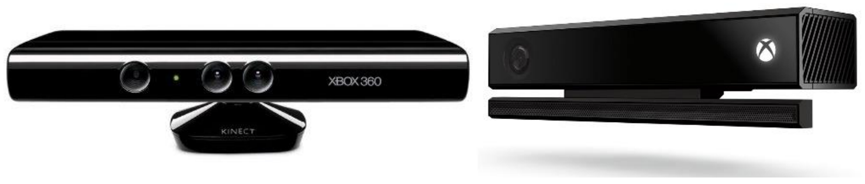


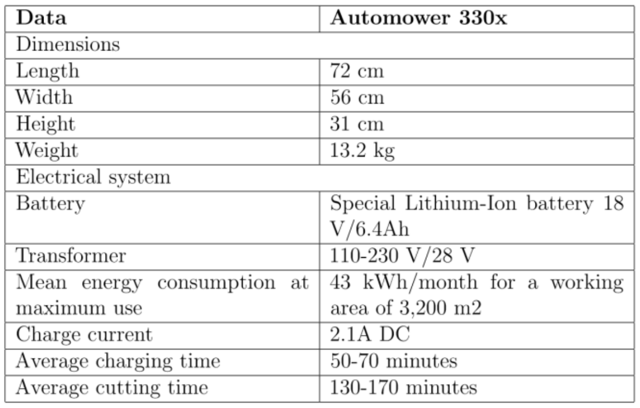
Figure 10 – Kinect v1 and v2 comparison

- Leap Motion

Figure 11 - Leap Motion

- Husqvarna Automower 330X

Husqvarna Automower 330X is one of the automatic lawn mowers from the Husqvarna Automower series by Husqvarna.

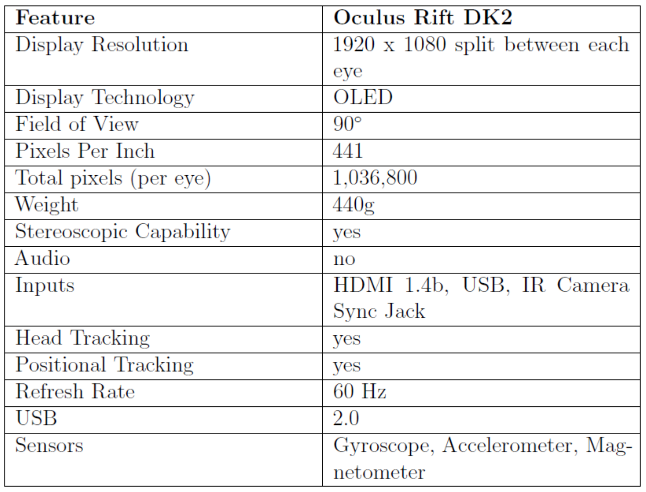


Data	Automower 330x
Dimensions	
Length	72 cm
Width	56 cm
Height	31 cm
Weight	13.2 kg
Electrical system	
Battery	Special Lithium-Ion battery 18 V/6.4Ah
Transformer	110-230 V/28 V
Mean energy consumption at maximum use	43 kWh/month for a working area of 3,200 m2
Charge current	2.1A DC
Average charging time	50-70 minutes
Average cutting time	130-170 minutes

Figure 12 - Husqvarna Automower 330X

- Oculus Rift DK2

A development kit comprehensive of an Oculus Rift which is a google device implementing virtual reality. It was born and shipped for the first time in 2014, with many refinements from the previous version. It incorporates a modified Samsung Galaxy Note 3 screen.



Feature	Oculus Rift DK2
Display Resolution	1920 x 1080 split between each eye
Display Technology	OLED
Field of View	90°
Pixels Per Inch	441
Total pixels (per eye)	1,036,800
Weight	440g
Stereoscopic Capability	yes
Audio	no
Inputs	HDMI 1.4b, USB, IR Camera Sync Jack
Head Tracking	yes
Positional Tracking	yes
Refresh Rate	60 Hz
USB	2.0
Sensors	Gyroscope, Accelerometer, Magnetometer

Figure 13 - Oculus Rift DK2

3.1.2. SOFTWARE

- Microsoft XBOX ONE KINECT 2 V2 with relative connection cable
- Ubuntu 16.04 LTS
- ROS Kinetic
- RTAB-Map package
- libfreenect2 drivers
- IAI Kinect2 drivers
- Rosbridge_suite
- IMU Stream (set of Android applications, mobile and wear)
- GAZEBO
- RViz
- Leap Motion SDK
- Dedicated USB 2.0 bus

3.2. How to Run the Project

3.2.1. Controller Side

In order to run the part of the project that controls the movement of the robot follows these instructions. First of all clone this repository through the command *git clone* and read all the README files in the different modules in order to install all the necessary dependencies.

Then compile your workspace using *catkin_make* and open five different Terminal tab. At this point follow this list of command:

- Kinect:

```
(Terminal 1)  roslaunch openni_launch openni.launch camera:=openni
```

- Smartwatch: Check the Mosquitto broker status:

```
sudo service mosquitto status
```

- Start the Mosquitto broker (if the broker is already active skip this step):

```
(Terminal 2) mosquitto
```

- Leap Motion:

-
  ```
  (Terminal 3) LeapControlPanel
  (Terminal 4) roslaunch leap_motion sensor_sender.launch
  ```

- In another terminal tab launch the controller and all the other nodes (inside the launch file you could comment components not needed, for example those for the simulation):

```
(Terminal 5) roslaunch controller controller.launch
```

- If you desire to run a simulation on GAZEBO, open another Terminal tab and do:

```
(Terminal 6) roslaunch am_gazebo am_gazebo_hrp.launch gui:=true
```

For the simulation on GAZEBO be sure to have installed all the necessary dependencies.

```
sudo apt-get install ros-kinetic-gazebo-ros-control
sudo apt-get install ros-kinetic-joint-state-controller
sudo apt-get install ros-kinetic-hector-gazebo-plugins
sudo apt-get install ros-kinetic-hector-gazebo
sudo apt-get install python-pygame
```

3.2.2. 3D Point Cloud Side

- LINUX SIDE

After having plugged in your Kinect, you're ready to launch RTAB-Map. Open three terminal and follow this list of command:

- command to initialize the RGB and depth sensors:

  roslaunch kinect2_bridge kinect2_bridge.launch publish_tf:=true

- command to start the mapping mode:

  roslaunch rtabmap_ros rgbd_mapping_kinect2.launch resolution:=qhd

The last thing to do is to set up the WebSocket necessary to send the data stream to unity. Please remember that in order for the WebSocket to successfully connect the two machines they must be connected to the same network.

- To launch the WebSocket run in a new terminal:

  roslaunch rosbridge_server rosbridge_websocket.launch

- WINDOWS SIDE

Start by downloading the repository with the unity code here.

Plug the oculus in the computer paying attention to the warnings in the troubleshooting section. The Oculus Sensor must be plugged in alongside the VR, otherwise the Unity scene would not be displayed.

Run then the Unity project by opening the scene contained inside the “/Oculus with pose/Assets” folder.

Once the project is fully loaded, the Oculus properly connected and the WebSocket on the Linux machine up and running, click on play on the editor.

All the relevant information would be displayed on the Unity console such as the configuration and the point cloud messages. After a short time (depending on the amount of data that Unity is receiving) the map will appear onto the scene and putting on the Oculus should be sufficient to give the user a proper 3D immersion into Virtual Reality.

If the Oculus App is not already running it will be opened automatically. If it’s not, the project will run on Unity but the Oculus VR won’t be able to show any image.

4. Results

4.1. Controller Side

The figures below show the simulation developed in the test phase. In the end we can conclude that the robot is fully controllable through the use of the three sensors that we analyzed above.

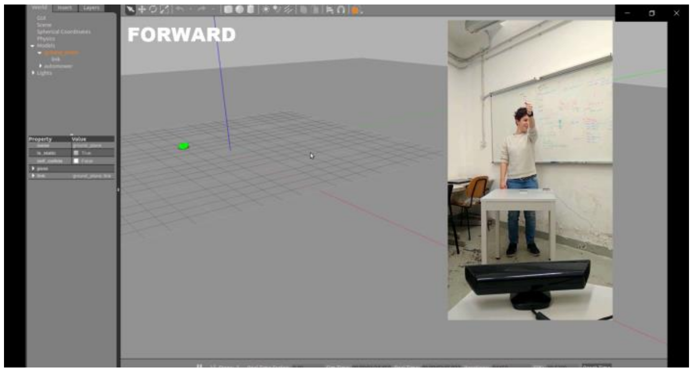


Figure 14 - Robot Moving Forward

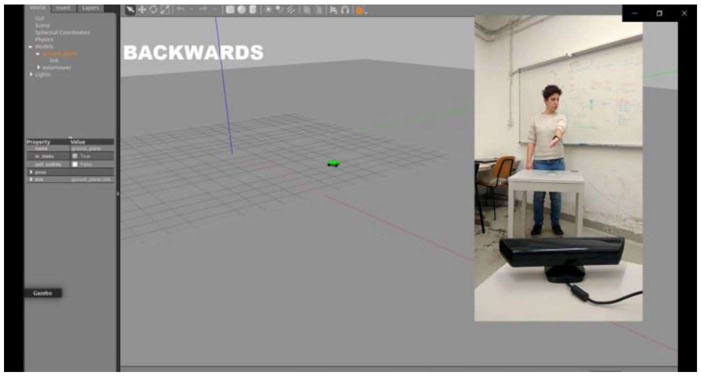


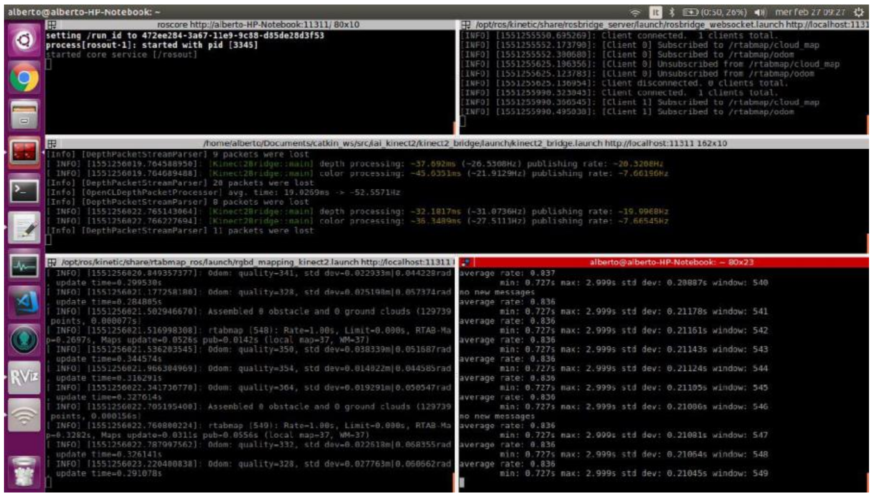
Figure 15 - Robot Moving Backwards

At the following link it is possible to see the complete video of the robot controlled by the user.

4.2. 3D Point Cloud Side

In this section, the resulting implementation will be shown. All three modules (Kinect-Unity-Oculus) have been thoroughly tested and have demonstrated to be fully working. Below are shown some pictures describing the steps performed to get the whole architecture running.
The final implementation allows the user to visualize the entirety of the map in a realistic and dynamic way while the virtual environment keeps expanding as the robot explores its surroundings.

4.2.1. Ubuntu: Kinect and RTABmap



```
alberto@alberto-HP-Notebook: ~

roscore http://alberto-HP-Notebook:11311/ 80x10
setting /run_id to 472ee284-3a67-11e9-9c88-d85de28d3f53
process[rosout-1]: started with pid [3345]
started core service [/rosout]
```

```
/opt/ros/kinetic/share/rosbridge_server/launch/rosbridge_websocket.launch http://localhost:1131
[INFO] [1551255550.695269]: Client connected.  1 clients total.
[INFO] [1551255552.173790]: [Client 0] Subscribed to /rtabmap/cloud_map
[INFO] [1551255552.380680]: [Client 0] Subscribed to /rtabmap/odom
[INFO] [1551255625.196356]: [Client 0] Unsubscribed from /rtabmap/cloud_map
[INFO] [1551255625.123783]: [Client 0] Unsubscribed from /rtabmap/odom
[INFO] [1551255625.136954]: Client disconnected. 0 clients total.
[INFO] [1551255990.323043]: Client connected.  1 clients total.
[INFO] [1551255990.366545]: [Client 1] Subscribed to /rtabmap/cloud_map
[INFO] [1551255990.495038]: [Client 1] Subscribed to /rtabmap/odom
```

```
/home/alberto/Documents/catkin_ws/src/iai_kinect2/kinect2_bridge/launch/kinect2_bridge.launch http://localhost:11311 162x10
[Info] [DepthPacketStreamParser] 9 packets were lost
[ INFO] [1551256019.764588950]: [Kinect2Bridge::main] depth processing: ~37.692ms (~26.5308Hz) publishing rate: ~20.3208Hz
[ INFO] [1551256019.764689488]: [Kinect2Bridge::main] color processing: ~45.6351ms (~21.9129Hz) publishing rate: ~7.66196Hz
[Info] [DepthPacketStreamParser] 20 packets were lost
[Info] [OpenCLDepthPacketProcessor] avg. time: 19.0269ms -> ~52.5571Hz
[Info] [DepthPacketStreamParser] 8 packets were lost
[ INFO] [1551256022.765143064]: [Kinect2Bridge::main] depth processing: ~32.1817ms (~31.0736Hz) publishing rate: ~19.9968Hz
[ INFO] [1551256022.766227694]: [Kinect2Bridge::main] color processing: ~36.3489ms (~27.5111Hz) publishing rate: ~7.66545Hz
[Info] [DepthPacketStreamParser] 11 packets were lost
```

```
/opt/ros/kinetic/share/rtabmap_ros/launch/rgbd_mapping_kinect2.launch http://localhost:11311
[ INFO] [1551256020.849357377]: Odom: quality=341, std dev=0.022933m|0.044228rad
, update time=0.299530s
[ INFO] [1551256021.177258180]: Odom: quality=328, std dev=0.025198m|0.057374rad
, update time=0.284885s
[ INFO] [1551256021.502946670]: Assembled 0 obstacle and 0 ground clouds (129739
 points, 0.000077s)
[ INFO] [1551256021.516998308]: rtabmap (548): Rate=1.00s, Limit=0.000s, RTAB-Ma
p=0.2697s, Maps update=0.0526s pub=0.0142s (local map=37, WM=37)
[ INFO] [1551256021.536283545]: Odom: quality=350, std dev=0.038339m|0.051687rad
, update time=0.344574s
[ INFO] [1551256021.966304969]: Odom: quality=354, std dev=0.014822m|0.044585rad
, update time=0.316291s
[ INFO] [1551256022.341736770]: Odom: quality=364, std dev=0.019291m|0.050547rad
, update time=0.327614s
[ INFO] [1551256022.705195400]: Assembled 0 obstacle and 0 ground clouds (129739
 points, 0.000156s)
[ INFO] [1551256022.760800224]: rtabmap (549): Rate=1.00s, Limit=0.000s, RTAB-Ma
p=0.3282s, Maps update=0.0311s pub=0.0556s (Local map=37, WM=37)
[ INFO] [1551256022.787997562]: Odom: quality=332, std dev=0.022618m|0.068355rad
, update time=0.326141s
[ INFO] [1551256023.220400838]: Odom: quality=328, std dev=0.027763m|0.050662rad
, update time=0.291078s
```

```
alberto@alberto-HP-Notebook: ~ 80x23
average rate: 0.837
        min: 0.727s max: 2.999s std dev: 0.20887s window: 540
no new messages
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21178s window: 541
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21161s window: 542
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21143s window: 543
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21124s window: 544
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21105s window: 545
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21086s window: 546
no new messages
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21081s window: 547
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21064s window: 548
average rate: 0.836
        min: 0.727s max: 2.999s std dev: 0.21045s window: 549
```

Figure 16 - ROS sending messages

In figure 16 there are five terminals opened: one for the "roscore", one for the Rosbridge which implements the WebSocket connection, one which initialize the Kinect driver freenect2 and IAI kinect2, one launching the mapping mode running RTABmap alongside with Rviz and the last which is displaying the frequency of the messages on the *rtabmap/mapData* topic.

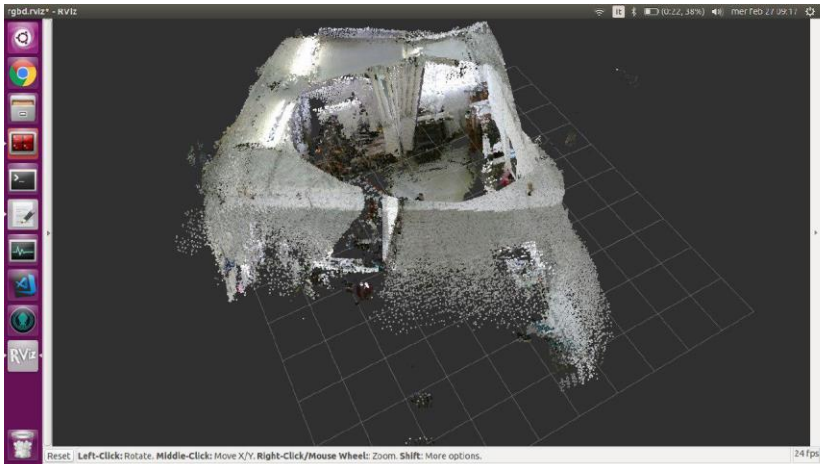
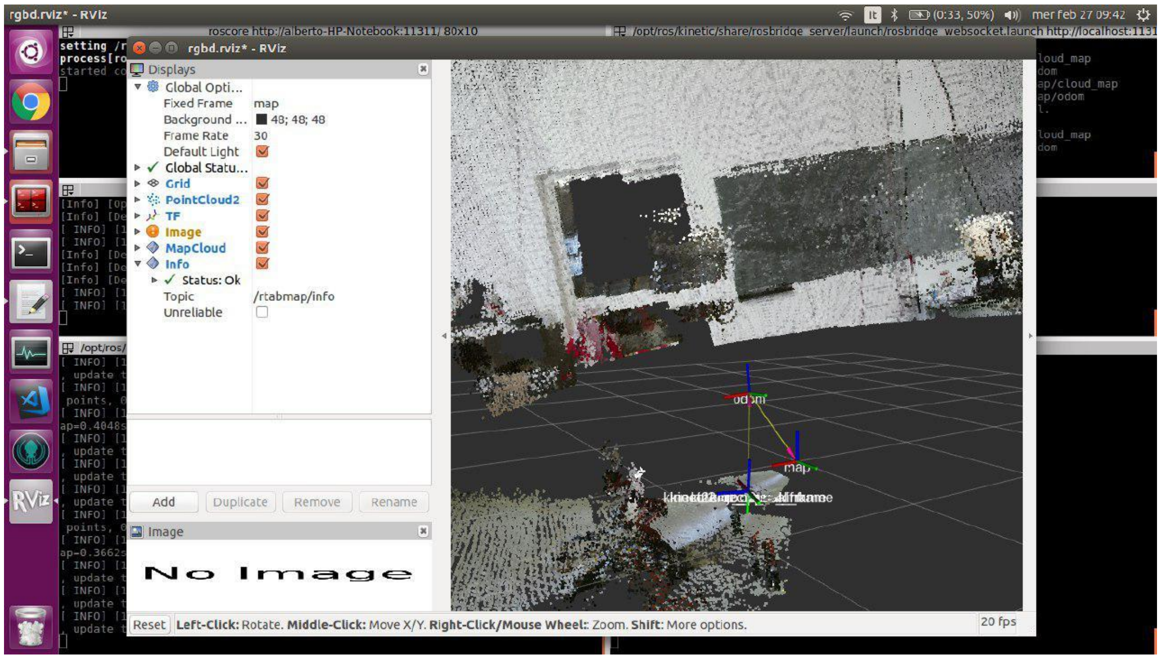

rgbd.rviz* - RViz
(0:22, 38%)
mer feb 27 09:17
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.
24 fps

Figure 17 - The 3D point cloud map seen from Rviz



```
rgbd.rviz* - RViz
roscore http://alberto-HP-Notebook:11311/ 80x10
/opt/ros/kinetic/share/rosbridge_server/launch/rosbridge_websocket.launch http://localhost:1131
```

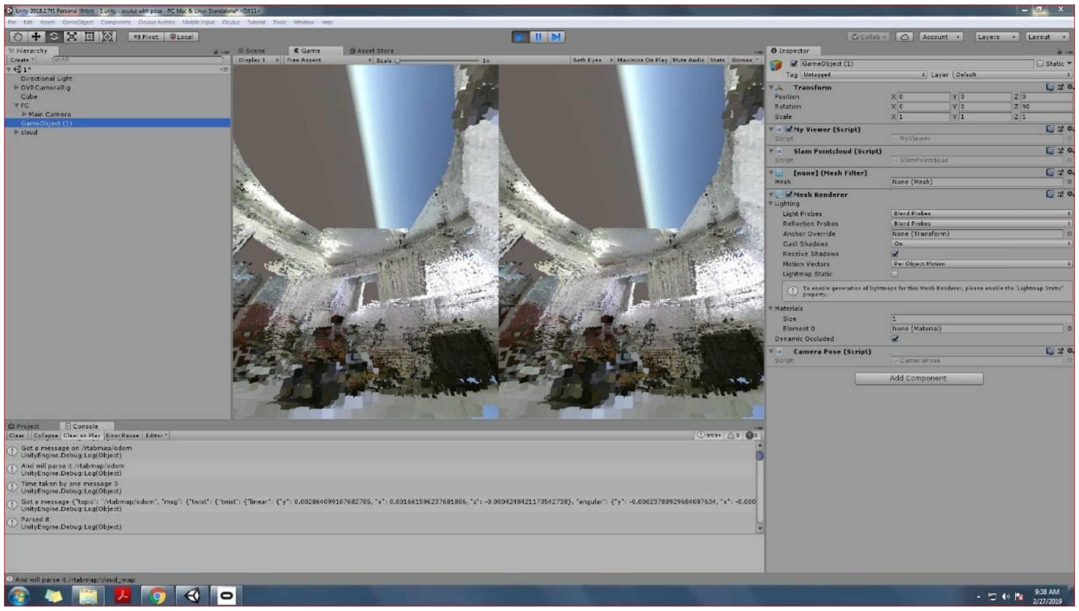
```
Displays
▼ Global Opti...
    Fixed Frame      map
    Background ...   ■ 48; 48; 48
    Frame Rate       30
    Default Light    ☑
▶ ✓ Global Statu...
▶ Grid               ☑
▶ PointCloud2        ☑
▶ TF                 ☑
▶ Image              ☑
▶ MapCloud           ☑
▼ Info               ☑
    ▶ ✓ Status: Ok
    Topic            /rtabmap/info
    Unreliable       ☐

Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   20 fps
```

Figure 18 - The 3D point cloud map in Rviz seen from the inside

In figure 17 and 18 it can be seen the 3D point cloud map rendered in Rviz from two different perspectives. In figure 18 it is highlighted the movement of the camera in space by the transformation frames.

As discussed above Rviz will open automatically with the correct setting already in place.

4.2.2. Windows laptop: Unity and Oculus



```
Got a message on /rtabmap/odom
UnityEngine.Debug:Log(Object)
And will parse it /rtabmap/odom
UnityEngine.Debug:Log(Object)
Time taken by one message 3
UnityEngine.Debug:Log(Object)
Got a message {"topic": "/rtabmap/odom", "msg": {"twist": {"twist": {"linear": {"y": 0.002864099107682705, "x": 0.001661596237681806, "z": -0.0004248421173542738}, "angular": {"y": -0.000237889296840876 34, "x": -0.000
UnityEngine.Debug:Log(Object)
Parsed it
UnityEngine.Debug:Log(Object)

And will parse it /rtabmap/cloud_map
```

Figure 19 - Unity displaying the map

In figure 19 and 20 the Unity editor is running the scripts to receive the point cloud map from ROS and to display through the Oculus lenses. On the console (lowest part of the picture) it's possible to see the messages arriving while on the scene it's clearly recognizable the shape of the EmaroLab room.

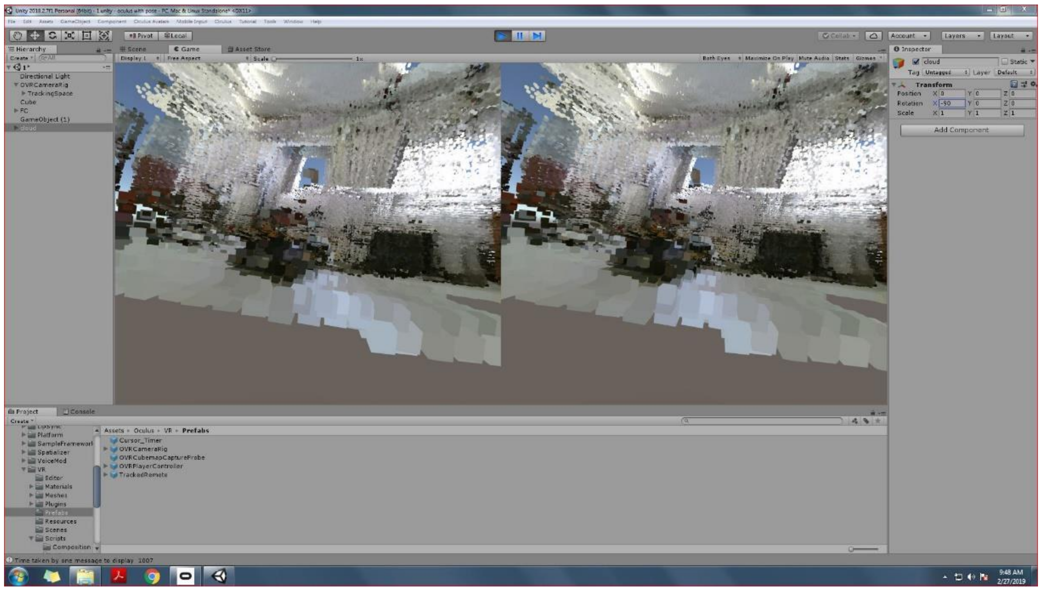


Figure 20 - Unity editor

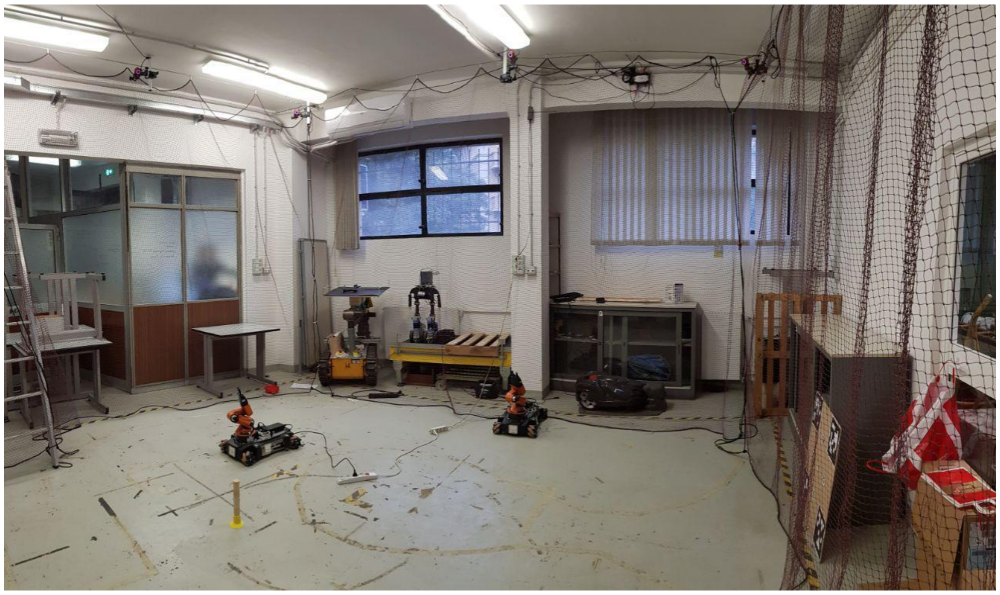


Figure 21 - Real Emarolab room

In figure 21 there is the real Emarolab room, where all the tests were held.
Below it can be found two links to videos showing a working implementation of the architecture.
[Link to implementation example video 1]

[Link to implementation example video 2]

5. Recommendations

5.1. Controller Side

During the simulation of this project some issues came out, like for example the fact that the connection between the smartwatch and the smartphone via Bluetooth and then between the smartphone and the computer via WebSocket introduce a considerable delay that has to be taken into account once the project will be simulated on the real Husqvarna Automower.

Another problem that came out during the implementation of the nodes is that the *complementary_filter_node*, which is discussed above in the second chapter talking about the controller module, needs to be launched with a delay with respect to the *MQTT ROS Bridge*, otherwise it is not able to receive the messages on the topic */imu_data_raw*. We found a solution in the package *timed_roslaunch* that gives the possibility to delay the launch of a roslaunch file; infact going in the file *controller.launch*, it possible to see that *imu_filter.launch* is delayed of two seconds.

While building the system we spotted some limitations in the hand gesture detection due to the accuracy of sensors' work and due to user-defined thresholds. The idea to minimize those limitations is to calibrate the range of thresholds before use by each user, either automatically or by parametrization.

Some recommendations for the use of the Kinect are for example make sure that the user is in the visual field of the sensor. We chose the left arm to take the data from the sensors, but for the Kinect it's the right arm since it's named after his point of view. Limit the persons in the range of the Kinect to one to guarantee a good calibration during the configuration phase and, for better pose estimation, don't move too fast!

Apple stopped public access to NITE (Developed by PrimeSense, now part of Apple) and this prevented us to work with Kinect v2 because NITE2 was not accessible. */left_hand* and */right_hand* are fixed so, during the implementation of the Kinect module, we could not work on quaternions of those frames. There was a bad accuracy for the RPY angles considering the elbow frame and a general lack of documentation for the gesture recognition using the Kinect.

5.2. 3D Point Cloud Side

In this conclusive chapter the choices, the issues and the future ideas for the project are going to be discussed. In particular, the focus will aim to the considerations made during the implementation of every module of the architecture, emphasizing the difficulties encountered and proposing solutions to overcome them.

Initially, the algorithm chosen to produce the 3D point cloud map was RGBDSlam. It seemed a very powerful tool but it started giving issues since the very beginning of the project. First it caused trouble with the catkin compilation: it had dependencies on many complementary libraries (such as g2o, eigen, PCL…) but the integration between all of them was very confused and complicated. For example, g2o could be installed both as a ROS library and from source, although the second was suggested in order to set right dependencies with the eigen. Moreover the required PCL libraries were a newer version with respect to the one pre-existing on Ubuntu 16.04 LTS and so, in order to compile them, it was necessary both to use C++ 2011 support and to modify several lines of code inside "CmakeList.txt" files.

In the end, after correctly following all the steps mentioned above (and precisely described here), the RGBDslam application was able to finally visualize the point cloud map in Rviz using the data provided by the Kinect but there was no apparent way to send those data from a ROS topic to the Unity editor.

So it was decided to switch to a better integrated and maintained algorithm: RTABmap, as described in the above paragraphs.

During the various tests and experiments it was also possible to employ both the Kinect 360 and the Kinect v2, but using two sets of different drivers. For the former "freenect_stack" driver was used, while for the latter the matter has already been discussed above. This put the light on strength and weaknesses of the two version of this device. Truth be told, the two were only slightly different, presenting more or less the same performances, except for a major improvement in the video quality of the camera, which in fact passes from 480p to 1080p. The results of this update can be seen with bare eyes also from Rviz.

For what it concerns Unity, at the beginning it seemed very intuitive to implement the interface with the VR. After getting caught up in many issues (mostly due to lack of compatibility between the devices employed) and after many unsuccessful attempts to get anything to work, it may be that the one between Unity and Oculus is not the best "marriage" and that maybe it's possible to find another software capable of interfacing the two in a simpler and therefore better way.

It should also be noticed that, not every time but quite often, there is a failure in the data acquisition by the RTABmap algorithm from the Kinect: the transmission practically stops and no more messages are sent. This could be due to the massive dimension reached by the point cloud map or to some sudden movement of the Kinect (that for all the experiments has been manually moved around).

After several tests it came out that moving back the Kinect to an already mapped position solves the issue and the acquisition restarts.

Nevertheless, a better understanding of this matter could lead to improvements for the overall performance of the project: maybe reducing slightly the amount of data acquired and sent could make the system lighter and faster.

Last but not least, in order for the overall project to work properly, the actual architecture must be integrated with a way of making the Unity scene move alongside with the movement of the robot.

Otherwise, the user won’t be able to understand the trajectories and so to control the robot. A C# script has already been developed and it only has to be tested with the overall architecture: this will be done in the next future.