Software Architecture For Robotics Project

HRP-Teleoperation

Kinect-Unity3D-Oculus Interface

Student(s): Alberto Ghiotto, Enrico Casagrande
Tutor(s): Alessandro Carfi
Supervisor(s): Alessandro Carfi
Year: 2018 - 2019

Table of Contents

Table of Figures

1. Objective of the Project

The objective of this project is to create a 3D pointcloud 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.

1.1 Introduction

Perceiving a space without effectively being there and being allowed to virtually wonder inside it is a very hard challenge to face, but also very worthwhile/rewarding/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.

Basically, teleoperation allows people to be in two places at the same time, saving time, energy, money and sometimes also human lives.

The efficiency of this way of working relies mostly on the capacity of reproducing, with extreme precision in some applications, the environment surrounding the robot and by localizing it inside the virtual 3D reproduction.

1.2 Hardware and Software

All the things illustrated in the previous paragraph can nowadays be achieved by interfacing many market-available devices: there already are many ways of perceive, elaborate and visualize a 3D environment.

In order to perceive the virtual 3D environment following the real-time behavior of the robot, the SLAM approach (Simultaneous Localization and Mapping) is implemented, which both generates a map and localizes itself in it at the same time. There are many algorithms that allows such a result, from Kalman Filtering to GraphSLAM and particle filtering.
The chosen algorithm is **RTAB-Map** (Real-Time Appearance-Based Mapping), which is an RGB-D, Stereo and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector. It can be used alone with a handheld Kinect, a stereo camera or a 3D lidar for 6DoF mapping, or on a robot equipped with a laser rangefinder for 3DoF mapping.

For this project the algorithm uses data provided by a **Microsoft Kinect v2**.
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. Nowadays it uses a wide-angle time-of-flight camera, and processes 2 gigabits of data per second to evaluate its environment. It also has a color camera recording in 1080p quality.

Feature	Microsoft Kinect v2
Hardware Compatibility	Stable work with various hardware models
USB Standard	3.0
Size	250 x 66 x 67 mm
Weight	3.0 lb
Power Supply	USB + ACDC power supply
Power Consumption	12 watts
Field of View	70° horizontal, 60° vertical
Vertical tilt range	±27°
Frame rate	30 frames per second (FPS)
Color Camera	1920 x 1080 pixels @ 30 FPS
Depth Camera	512 x 424 pixels @ 30 FPS
Maximum Depth Distance	4.5m
Minimum Depth Distance	50cm
OS Platform Support	Xbox One Microsoft Windows Linux MacOS
Programming Language	C++/C# (Windows) C++(Linux) JAVA

Table 1 - Kinect Specifications

The Kinect data are collected, processed by the SLAM module (running on Ubuntu 16.04) and then sent as messages on a ROS topic. **ROS** is a powerful middleware providing libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS Topics are named buses over which nodes exchange messages.

From another laptop (running windows this time) the data are received by subscribing to the ROS topic and then visualized in a **Unity 3D** project. Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005. It gives users the ability to create games in both 2D and 3D, and the engine offers a primary scripting API in C# language as well as drag and drop functionality.

Unity allows the 3D scene to be visible through a VR Oculus, in order to give the user a complete immersive view of what the robot is seeing.

The last link of this chain is the **Oculus DK 2**, a development kit comprehensive of an Oculus Rift which is a goggle 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. The oculus can be united with many complementary devices which improve the experience such as hand controllers and speakers.

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

Table 2- Oculus Specifications

2. The System's Architecture

2.1. Overall 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 a Oculus VR device used for the final visualization.

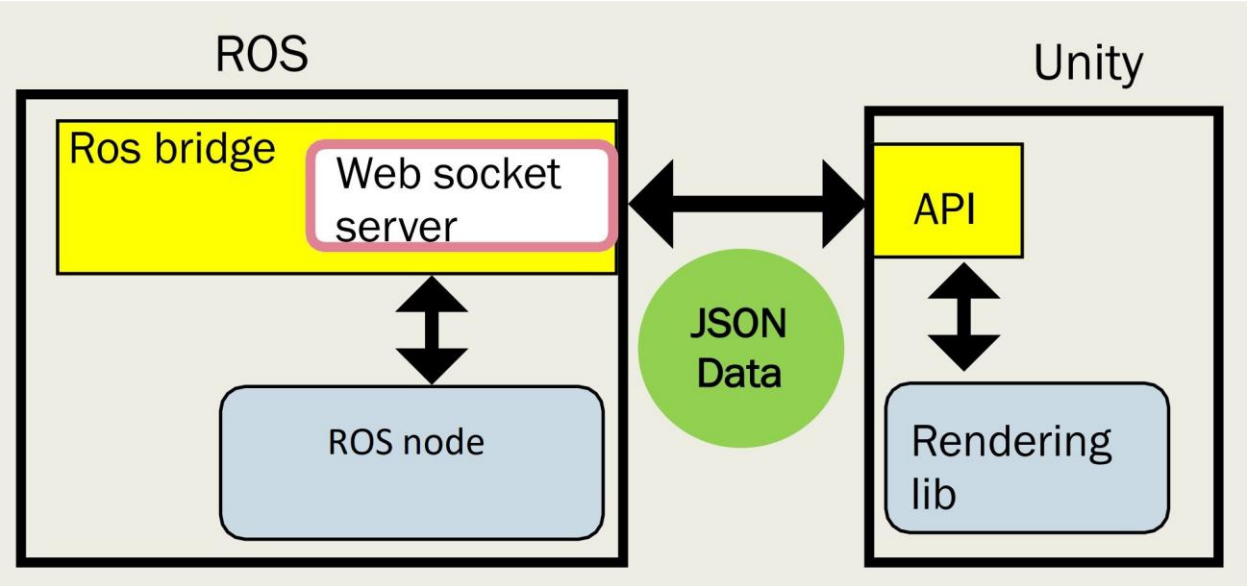

Figure 1-Architecture scheme

2.2 Description of the Modules

2.2.1 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.

Inputs

The inputs are the images acquired by the kinect 2 v2 in RGB-D format, which is a combination of a RGB image and its corresponding depth image.

Internal working

RTAB-Map takes the RGB-D images and publishes them as ROS messages under different topics as shown in the documentation here.

Outputs

The outputs are the messages published on the different ROS topics, in particular we are interested on 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.

2.2.2 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.

Inputs

Unity receives the ROS messages of the `/rtabmap/mapData` topic via the websocket.

Internal working

All the functionalities descripted above are integrated in a custom script which connects to the ROS node, receives the data and then converts the pointcloud 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.

Outputs

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 LINUX SIDE

- 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

3.1.2 WINDOWS SIDE

- Oculus rift Developer Kit 2
- Unity 2018.2.7f1
- Oculus APP
- Unity project files

3.1.3 TROUBLESHOOTING

- Notebook and laptops are apparently not supported by the Oculus kit. This seems to be due to a lack of drivers for portable GPUs, which prevents the Oculus from displaying anything.
- The Oculus must be directly connected to the graphic card of the computer.
- It is recommended to connect the USB cable to a USB 2.0 port. However, in our case the Oculus worked fine as well even when connected to a 3.0 port.
- Before opening the Unity project is always recommended to have already run the Oculus Runtime application (which is also very useful to check the status of the connection of the device).
- If, while launching the unity project, the error **'VR: OpenVR Error! OpenVR failed initialization with error code VRInitError_Init_PathRegistryNotFound: "Installation path could not be located (110)"!'** occurs, it's necessary to install VR Samples from the Unity Assets Store in order to fix it.

3.2 How to run the project

Here it's described step by step how to download and run the project on a new computer.

3.2.1 LINUX SIDE

- Preparation

If you don't already have it, install ROS kinetic as shown in the tutorial

Install rtabmap using the following command:

```
sudo apt-get install ros-kinetic-rtabmap-ros
```

Modify in `/opt/ros/kinetic/share/rtabmap_ros/launch` the file `rgbd_mapping_kinect2.launch`, at row 25 set `default = "true"`, at row 26 set `default = "false"` in order to open the pointcloud map with Rviz.

Modify in `/opt/ros/kinetic/share/rtabmap_ros/launch/config` the file `rgbd.rviz`, at row 76 substitute `/voxel_cloud` with `/rtabmap/cloud_map` in order to visualize the correct topic from the beginning.

Note: remember to open the files as sudo in order to being able to modify and save them.

In order to install freenect2 libraries follow the README's instructions here

In order to install IAI Kinect2 libraries follow the README's instructions here

In order to install the rosbridge_suite use the command below:

```
sudo apt-get install ros-kinetic-rosbridge-suite
```

- Running the node

Now, after having plugged in your kinect, you're ready to launch RTAB-Map. Open a terminal and launch `ROSCORE`

In a new terminal type the command to initialize the RGB and depth sensors:

```
roslaunch kinect2_bridge kinect2_bridge.launch publish_tf:=true
```

In a third terminal type the 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
```

3.2.2 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 pointcloud 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

In this section, the resulting implementation will be shown. All the 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.

4.1 Ubuntu: Kinect and RTABmap

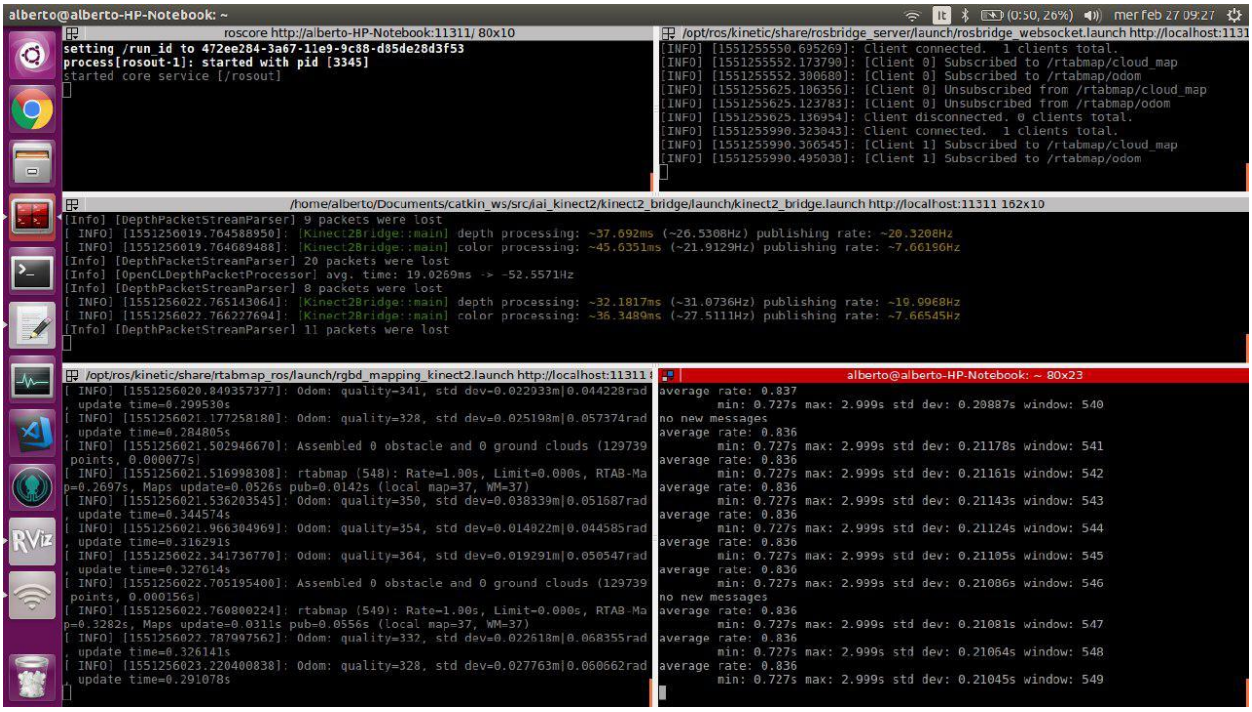

Figure 2 - ROS sending messages

In figure 1 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.

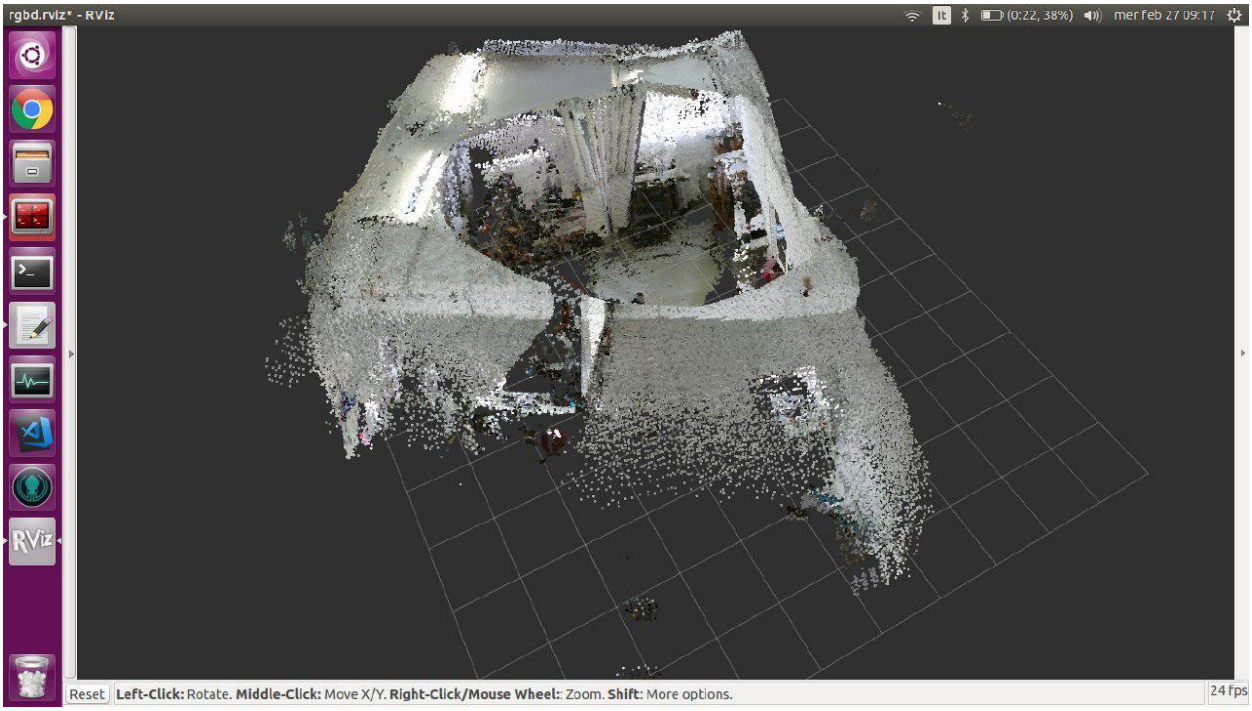

Figure 3 - The 3D point cloud map seen from Rviz

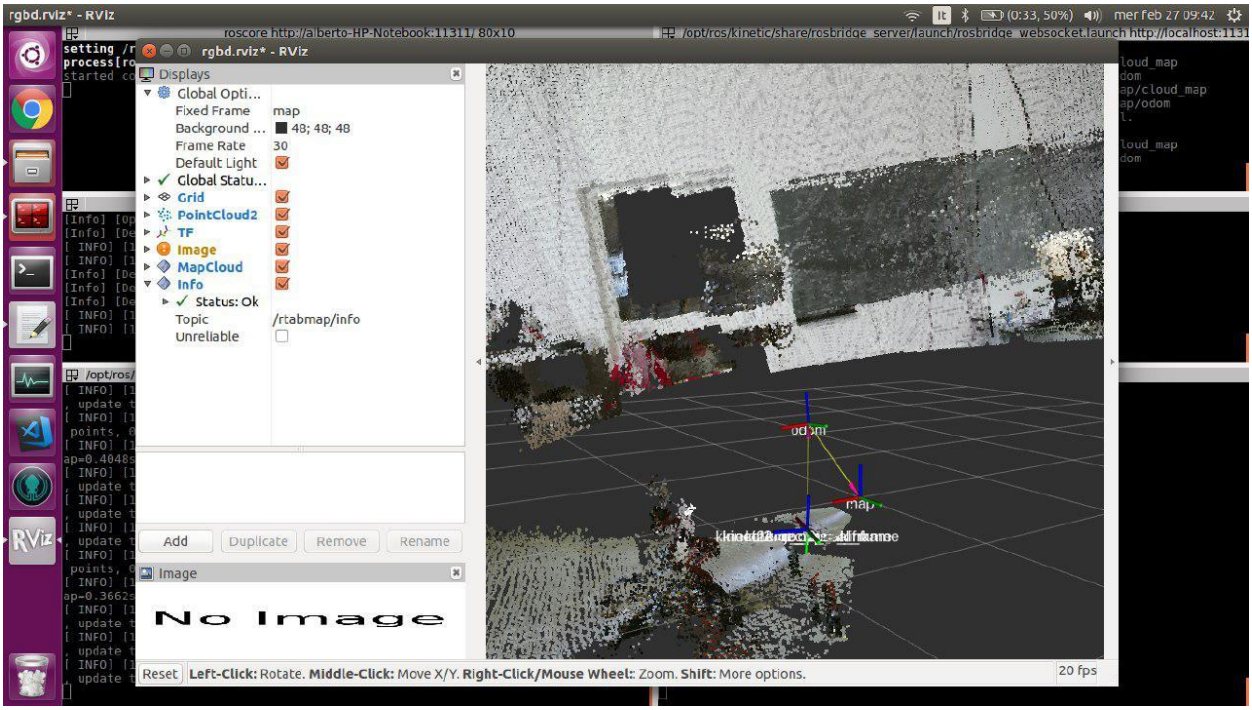

Figure 4 - The 3D point cloud map in Rviz seen from the inside

In figure 2 and 3 it can be seen the 3D point cloud map rendered in Rviz from two different perspective. In figure 3 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 Windows laptop: Unity and Oculus

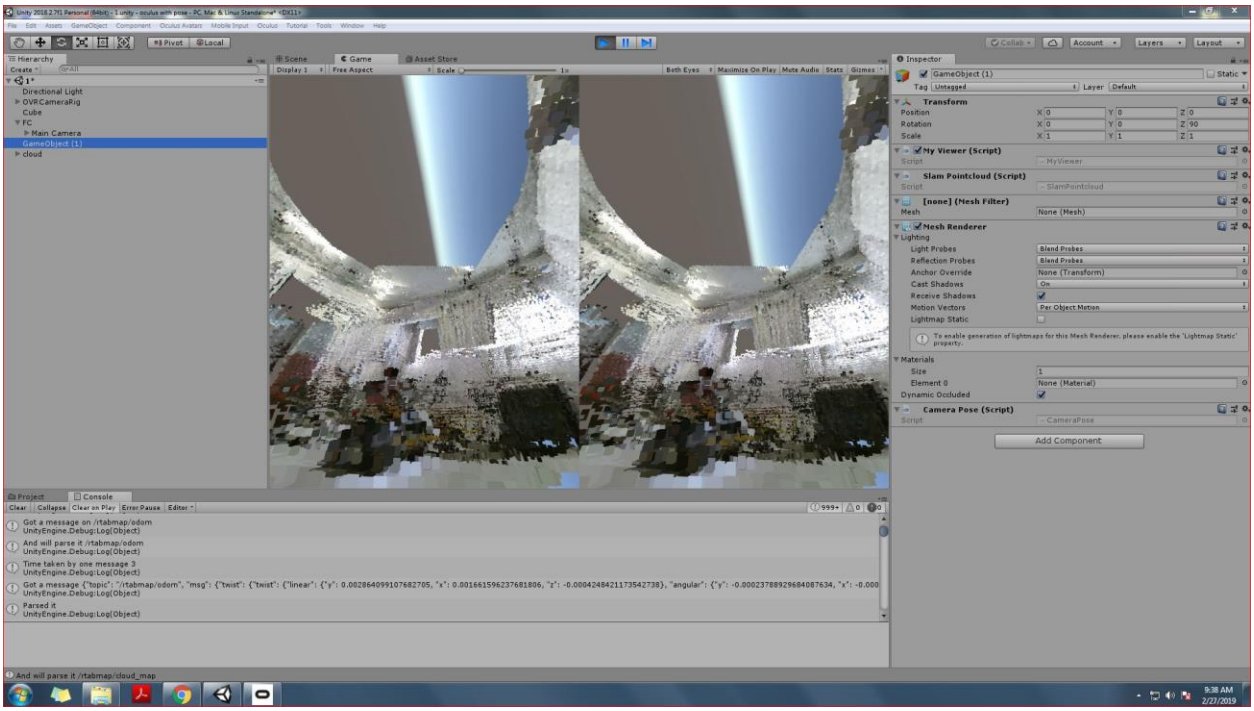

Figure 5-Unity displaying the map

In figure 4 and 5 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.

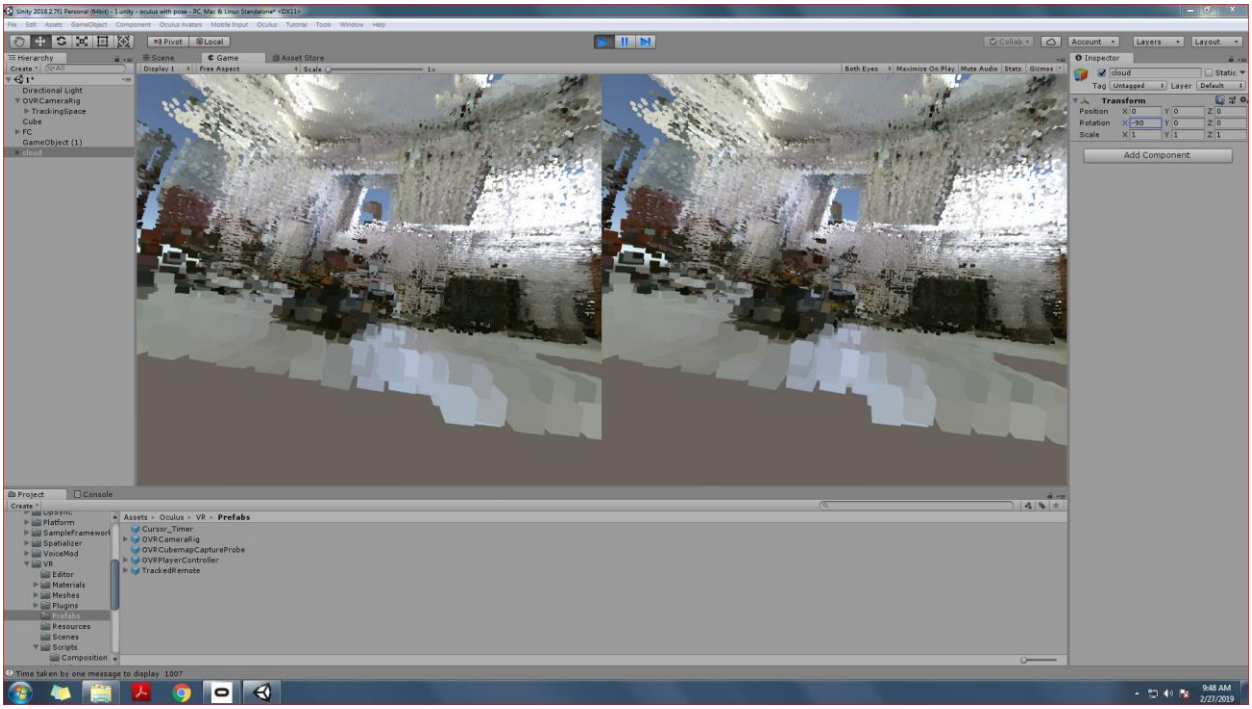

Figure 6 - Unity editor

Figure 7 - Real Emarolab room

In figure 7 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

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 pointcloud 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 pointcloud 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 set 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 Uniy 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 pointcloud 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.