

Construction Supervision with Augmented Reality

Pablo Fernández Fernández



LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6154
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2022 by Pablo Fernández Fernández. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2022

Abstract

Construction sites are key points regarding implementing robot supervision to create more secure, reliable, and free hand-work workspaces. This project's scope is to develop a complete interpretation of a modern concept of surveillance. For this case, the test scenario is the industrial lab at the Faculty of Engineering, Lund University, known as RobotLab. For its development, an Augmented Reality scenario is recreated in which the work done by the ABB IRB2400 robot is studied at the lab. The use of Unity and ROS as special software programs and systems can generate this concept in a manner that can be easily reproduced in other scenes.

In order to give a more modern concept, the tool for taking care of the vigilance is a robot, or in this case, a simulation of the robot Spot from Boston Dynamics. The reason behind this is to produce a more automatic project so that the person in charge can operate this robot and use its camera for the creation of the Augmented Reality. In general terms, the development of this project is carried out by the use of a large number of separate projects from different developers and research companies. In conclusion, the idea is to provide a software package to be applied and integrated into other constructions scenarios for the recreation of Augmented Reality scenarios.

Acknowledgements

My research would have been impossible without the aid and support of the Department of Automatic Control at Lunds Tekniska Högskola, highlighting the importance of my supervisor Anders Robertsson, Alexander Pisarevskiy, Greg Austin and Anton Johansson.

Contents

List of Figures	9
1. Introduction	12
1.1 Motivation	12
1.2 Objectives	12
1.3 Methods	13
1.4 Limitations	14
1.5 Tools and programs	15
2. State of Art	16
2.1 Industry 4.0	16
2.2 Augmented Reality	17
2.3 Robotics	22
3. Development of the Digital Supervised Construction	29
3.1 Virtualization of an ABB Robot Arm	29
3.2 Spot Supervision	42
3.3 Integration of the sensors	44
3.4 Final integration of the AR in Unity	47
4. Results	49
4.1 General Network Architecture	49
4.2 Generation of the Augmented Reality Scenario	50
4.3 Spot Supervision	57
5. Discussion	59
5.1 Virtualization of the Augmented Reality	59
5.2 ROS infrastructure	60
5.3 Spot Robot	60
5.4 Professional Utility of the project	61
6. Conclusions	62
6.1 Future Work	63
Bibliography	64

A. Code	68
A.1 URDF - ABB IRB2400	68
A.2 ROS script for handling the service communication	74
A.3 C# script for QR detection with AR simulation	75
A.4 ROS Launch File - QR Tracking Integration	76
A.5 ROS Launch File - Vuforia Integration	77
A.6 Sensors implementation with ROS#	78
A.7 ROS Launch File - Gazebo Integration	81

List of Figures

2.1	Architecture of Vuforia Engine [Daraghmi, 2021]	21
2.2	Hololens 1st Generation [Microsoft, 2021a]	22
2.3	ABB IRB 2400 Industrial Robot [ABB, 2021c]	23
2.4	RobotStudio view of the IRB2400	24
2.5	Spot Robot by Boston Dynamics [Boston Dynamics, 2021b]	25
2.6	Sparkfun VL53L1X - Distance Sensor [Sparkfun, 2021]	26
3.1	MoveIt implementation of the ABB IRB2400	30
3.2	ABB IRB2400 built in Unity	31
3.3	List of GameObjects given by Vuforia Engine	32
3.4	Tracking point given by Vuforia's algorithm	33
3.5	AR Simulation of the IRB2400 with the tracking of a QR	33
3.6	Target point of view given by <i>Model Target Generator</i>	34
3.7	Vuforia recognition of a 3D model using Model Target algorithm	34
3.8	Code example of the methods implemented with Unity Robotics Hub in Unity	35
3.9	Code example of a ROS Service	36
3.10	Code example of the methods implemented with ROS [‡] in Unity	36
3.11	Block diagram of the communication between ROS and the ABB Robot Controller [ABB, 2021a]	37
3.12	System Installation of the Virtual Controller	39
3.13	Communication parameters with the remote host PC (ROS)	39
3.14	vcconf.xml file for remote connection with a host PC (ROS)	40
3.15	MKTR Scenario in Unity	41
3.16	ViscAutoTracker ROS Package for a QR recognition	42
3.17	Integration of Spot with RVIZ and Gazebo	44
3.18	Set of three VL53L1X integrated with D1 mini Pro v2.0	45
3.19	ROS message type Sensor_msgs/Range [ROS, 2021e]	45
3.20	Linear potentiometer integrated with D1 mini Pro v2.0	46
3.21	JR3 Force Torque Sensor [NeuroRobotics - NTUA, 2021]	47

List of Figures

4.1	Possible Network Architecture	50
4.2	ROS Nodes used for the application of Unity with the ABB controller	51
4.3	ROS Nodes used for the application of the camera an the QR detection	53
4.4	Integration of an external camera with ROS in Unity	54
4.5	Distance sensors integration in Unity	55
4.6	Linear sensor integration in Unity	56
4.7	Force sensor integration in Unity	56
4.8	QR recognition by Spot's camera simulated in Gazebo	58

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
CAD	Computer Aided Design
DT	Digital Transformation
DoF	Degrees of Freedom
EGM	Externally Guided Motion
ICT	Information and Communication Technologies
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
LTH	Lunds Tekniska Högskola
MRTK	Mixed Reality ToolKit
OS	Operating System
ROS	Robot Operating System
RWS	Robot Web Services
SDK	Software Development Kit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier

1

Introduction

The automation field is growing more and more as technology becomes a more critical factor in our lives. Nowadays, a new trend is to convert all the industrial sector into digitization, with less physical personnel and more computerized work. So is that there is an exciting paradigm on digitizing the supervision of different machinery and construction inside the industries. In this way, one manner of overseeing efficiently if a site is secure is using Augmented Reality and robots, so less human force is needed and better performance is acquired. Along these lines, an investigation and research of how to develop new algorithms and modules for augmented reality and robots seem to be suitable to improve supervision tasks.

1.1 Motivation

The concept of Industry 4.0 incorporates many technological fields into the classic industries that support our society. The integration of Internet of Things (IoT), Mobile Computing, Artificial Intelligence, or Cloud Computing is well known in the industrial world. Their implementation is proportional to a higher impact on the performance and, therefore, the profit.

Here is where this project comes to light. The idea of integrating a robot to carry out supervision tasks at the RobotLab from the Department of Automatic and Control of LTH approaches our reality to the utopian scenario of having a whole industry run by autonomous robots. Furthermore, combining this concept with augmented reality scenarios into the construction fields can generate safer and more profitable sites.

1.2 Objectives

This thesis aims to create a software module for a commercial software, Unity, to recreate an Augmented Reality scenario where a specific robot arm is studied and simulated with the help of ROS. Furthermore, it is studied the integration of

the commercial robot Spot, from Boston Dynamics, for carrying out the automatic mission of supervising the tasks performed by the robot arm.

The mentioned module needs to create an automatic supervision work of the different working robots of a specific industrial site, the RobotLab. For a more functional application, also an augmented reality view from the robot's camera to determine the performance of the mentioned supervision is created.

In this way, it is planned to understand the Application Programming Interface (API) created for the Spot robot and contribute to the research and creation of new modules inside it. For this, the integration of ROS as the software for creating the environment that combines and communicates all the distinguished parts is studied. On the other hand, having an augmented reality scene built from the Spot's camera will satisfy the mentioned supervision.

Another important goal is the contribution to the development of software for the automation of a specific task. Besides the necessity of implementing modules of the API, it will be mandatory to create a fully functional automation job. For this, the autonomous job of the robot is to carry out the checking and supervision of an industrial site. In this case, it will be the LTH's RobotLab, the one understudy, so that this space can be considered "supervised" under the working of the different machinery and robot arms.

One last important concept is the communication in real-time between the robot and the reality created for the augmented world. In this case, it will be a requirement to obtain good performance from the Augmented Reality, where it will be measured the quality, the rate, and the delay of the video.

1.3 Methods

The mentioned project aims to have two different concepts under study; nevertheless, the methodology used in both is similar. With this, the idea is to implement quantitative research where the final results should be operational for future purposes and qualitative research to explain the different approaches and decisions made. A deep traineeship is needed for both concepts to understand the different programs, languages, and theories involved. The use of manuals, books, and tutorials are necessary during the whole project.

Hereafter, every concept will need different kinds of skills and operations of data.

On the one hand, the software module of the Spot robot is implemented in Python by the use of the Software Development Kit (SDK) provided by the company Boston Dynamics. This SDK provides the different functions that the robot can implement, highlighting its control, data acquisition, geolocation, and mission. Furthermore, Boston Dynamics provides a controller for the robot, where the autonomous work is handled at first; that is, the trajectory carried by the robot is represented on the screen of the mentioned controller. Secondly, the idea is to represent

this autonomous task on the screen of a computer. Instead of using the controller, Gazebo and thus Robot Operating System (ROS) are considered. On a second step, the installation of ROS into the robot is set up, regarding if it is possible to actually operate it under this specific software. Finally, as the last step, supervision of a robot arm will be taken into account.

On the other hand, the Augmented Reality is carried out by using software given by the enterprise PTC, called Vuforia Engine, which is implemented in C sharp, under Unity (video game engine). The main idea is to create augmented reality objects of the ABB arm IRB2400 of the RobotLab, wherewith it is made a complete study of the work that it is carrying out. For this, it is created a simulated space where the use of Computer-Aided Design (CAD) tools will generate the augmented reality objects. A second step is implemented on a real-time video of a 3D-scan robot arm to simulate and construct the interface. Finally, the last step is acquiring data from the real robot arm with ROS implementation, where there is communication between computers and the robot.

1.4 Limitations

During the development of this project, several limitations appeared causing a change in the planning and the scope of the different steps intended to be carried out. Even though the final results were not affected, a different approximation was taken into account.

Firstly, the idea of using a unified OS capable of connecting all the different hardware and software has been neglected since the first weeks. The first approach considered was the use of Linux as the main and only OS. The reason for the necessity of using Windows comes from the compatibility between RobotStudio, which is not supported, and Vuforia, which is possible to install but not to run as the camera device from a Linux is not recognizable by the Vuforia Engine.

The second limitation of this project is the type of camera that is implemented. Even though this will be commented on more deeply afterward, the Vuforia API recognizes an actual real device as a camera input. This could sound reasonable, but the concept of using the Spot's camera with ROS is neglected as it is not possible to establish it as a camera device. Because of this, another path will be taken in order to implement the AR with the use of ROS.

Thirdly, the implementation of RWS and EGM on the ABB controller can solve the connection between ROS and the ABB robot. Anyways, these environments present the problem of delay where it is necessary to be careful on the type of software used. On the one hand, RWS seems to be slower than EGM, which uses UDP data transfers, but only a simple joint position can be established.

Last but not least, the Spot robot has presented many complex circumstances. Because of being a product in a released process, parts in the core are still under study, so a significant part of the project the robot has not been considered, and

simulations have been carried out.

1.5 Tools and programs

For the development of this Thesis a big amount of equipment and software needs to be taken into account. For this reason, all the necessary tools needed for this work are commented:

Material	
Hardware	Software
Main Computer	Linux - Ubuntu 18.04 ROS - Melodic Visual Studio 2019
Auxiliary Computer	Windows 10 Unity 2020.3.6 Vuforia 9.8.8 RobotStudio 2021
Router Switch	
SPOT	Boston Dynamics' SDK
USB Camera	
ABB IRB2400 ABB IRC5 Controller	StateMachine Add-In 1.0

2

State of Art

In order to fully understand the development and results of this project, it is necessary to introduce the different technologies that are used these days in industry. As part of the scope, it is commented what is known as Industry 4.0, what is this modern concept of Augmented Reality (AR) and its tools, and finally, how is robotics oriented to this field.

2.1 Industry 4.0

This concept, started in Germany in 2011 and adopted and integrated into society by Klaus Schwab [Schwab, 2015], defines a world where individuals live among the digital world with the inclusion of connected technologies. It is, therefore, essential to highlight the Information and Communication Technologies (ICT), which has a direct impact in the industries with the application of the Internet of Things (IoT) and Artificial Intelligence (AI).

The change has significantly shaped what we understand as an industry compared to what the world thought a couple of decades ago, in terms of the development and introduction of different technologies and the concept of the business models, generating economic growth in many businesses and factories.

Implementing this concept into the construction industry has been well adopted by many companies, embracing the name of Construction 4.0. One of the leading utilities is the use of robots as it introduces scenarios where the production error is reduced considerably compared to human work. In addition, it generates safer workstations as the interaction machine-human is reduced (more focus on production lines) and where the production speed is increased due to the quick response of the robots, the reduction of the delay in the communication between them, and basically, because robots work more hours per day. Furthermore, the increase in the amount of data generated, which could be seen as an inconvenience for its storage and management, engenders a scheme where all the information given, transferred, and processed by every electronic device can be analyzed.

Digital Transformation

Digital Transformation (DT), referred to as "*adopting disruptive technologies to increase productivity, value creation, and the social welfare*" [Ebert and Duarte, 2018], provides an easy understating of what an industry needs to be competent in the market. The introduction of computers, the internet, and communication technology generate a more prosperous and practical value in product development. Indeed nowadays it is hard to find a company that does not integrate this concept into its business core as part of its product value.

For this reason, the construction industry requires to achieve a DT. Even though the use of robotics in this specific field has been adopted for decades, the idea of introducing features such as IoT, AI, and the connection of all the digital machinery in order to have a unique data environment for the acquisition of useful data is being developed nowadays. Going deep into security parameters, the idea of replacing the work carried out by a human with a robot, generates a more significant impact in the reduction of structures errors in buildings, fewer accidents, and higher efficiency in the results.

Robot Supervision of Constructions

As it has been commented, the idea of integrating robots in the construction industry does not bring any doubts for many companies. The benefits of this introduction are numerous, highlighting their use in the construction work itself as it can be the making of walls [FBR, 2021], their use for supervision purposes [Skydron, 2021] with the inclusion of imaging technology, and the making of a higher profit as the human error is reduced.

Here is where Spot from Boston Dynamics takes place [Boston Dynamics, 2021b] as a new tool for constructions and civil works. This tech company has developed a robot with the physical characteristics of a dog so its quadruped shape, which offers good effectiveness in uneven terrain, makes it manageable and suitable for establishing supervision jobs.

In this way, many companies have got their attention to this new concept in robotics, and the *RobotLab* from LTH has been interested in the acquisition of one in order to carry out surveillance and progress logging on a nearby site withing the University campus. Within this thesis work, corresponding surveillance within the RobotLab has been investigated so that inspection can be accomplished without the necessity of being there in person.

2.2 Augmented Reality

Applied to construction sides, the Augmented Reality (AR) can provide significant development in quality and supervisions jobs [Azuma, 1997]. The idea of implementing virtual objects over 3D real objects in real-time generates a new paradigm

in the industry as it is possible to recreate simulations that a human eye cannot perceive and feel at first.

Imagining that a regular operator could see the thermal structure of beams, the forces applied to a wall, the torque from the robots, or if a machine is running in a critical mode (concepts that a regular human eye could not perceive) gives the idea of enormous progress in this field.

For this reason, many companies have raised to fulfill the idea of developing software and hardware tools to allow the community and ordinary people to insert this concept into their lives and factories.

The following set of concepts and software products are one of many cases that helps to integrate AR.

Unity

Unity, created as a video game engine by Unity Technologies, can generate interactive scenarios for designing games and phone applications for all the market devices such as Android, IOS, and Windows platforms [Microsoft, 2021b]. These 2D/3D scenarios can be run by the Integrated Development Environment (IDE) that is given by its Application Programming Interface (API) that runs an internal Unity Code with the addition of external C# (or Javascript) code given by the user.

About the interface, this framework provides a user-friendly tool where the following can be seen:

- **Project:** without entering into detail, Unity gives a directory path where different resources are kept in order to develop your personal scene or game. The most remarkable ones are the *Assets* where all the valuable features for the users are kept and the *Packages* where the different packages compatible with Unity are set, such as Vuforia, ROS, and Mixed Reality Toolkit (which are commented later on). Inside the *Assets* are found a set of other directories that define what the user considers as practical for developing the game scene. Inside this, it is found the *Materials*, where the textures are kept, the *Prefabs* where 3D objects files are defined, *Scene* where it can be set the different scenarios and *Scripts* which contains the C# files responsible for generating the actual game mode.
- **Scene:** it shows the current scene that it is being built. It can be distinguished between the *Game Mode* where the actual scene is seen from the camera point of view; that is, it provides the view that the gamer is going to experience; and the *Play Mode*, which is a running simulation of the actual game.
- **Hierarchy:** is the section where all the *GameObjects* used are referred. In this way, Unity provides a hierarchy structure where a *GameObject* can depend on other *GameObjects* talking about its coordinate positions (relative to one another) and their scale.

- **Toolbar:** it is the selection of the different tools such as *Move*, *Rotate*, and *Scale*, as well as the *Play Mode* where the game is rendered and built by the use of the IDE of the system. In this way it is possible to test how the real game looks like before stepping into the application development.
- **Console:** is the window where the outputs of the debugging, errors, warnings, and prints from the game are shown.

More about what a regular user can develop through this framework, it is necessary to mention what a *GameObject* is as the main utility that can be used in Unity. Virtually, everything set in the framework is considered a *GameObject*, and in the case of this project, it is necessary to mention the following ones as they are the ones that are generated:

- **Directional Light:** it is the main object responsible for generating the artificial light that illuminates the scene. Furthermore, this object is dispensable as the AR is created over a live video or by the use of *Smart Glasses* such as Hololens.
- **Main Camera:** later substituted by *AR Camera* given by the package Vuforia, this object is responsible for generating the point of view of the user and where the actual game is played and rendered. It provides all the possibilities of what a real camera can provide, but in the case of this project, only the rendered distance, that is, the minimum distance in which the objects start to appear, is requested.
- **3D Model:** it corresponds to the object existing and played in the scene. In the case of this project, this corresponds to a model of the robot ABB IRB2400, where the hierarchy of the joints can be found, starting from the base and continuing with all the parts, each being relative in position to the previous one. To make it more dynamic and actually to give it some movement, it is necessary to apply physics laws to the object as all the different joints need to be attached to each other. In this case, gravity and inertia are used so it can be applied different velocities to every section of the robot.
- **Panel:** this object is part of the User Interface (UI) sector. It is intended to be a blank area that can render different textures, and for this, it can be used for many purposes. In this case, and with the help of a special script that is commented afterward, this *GameObject* will display the frames captures from a video source.
- **Text:** also part of the (UI) sector, it provides the addition of text (words and sentences) as part of the AR experience.

- Empty Object: as its name indicates, the user defines the usability of this object. In this case, it is used mainly to create the bridge between this framework and ROS and provide the data for the movement of the robot and the sensors implemented in the real scenario.

Vuforia

Vuforia is a Software Development Kit (SDK) aimed for the development of computer and phone Augmented Reality applications by the computerized recognition of shapes of 2D and 3D objects in real-time [PTC, 2021]. Gathered by PTC Inc, this kit ensures an augmented reality experience for any necessity although with some limitations.

In general terms, an application developed by Vuforia provides the following user experience:

- Text and image recognition: an internal algorithm can detect characteristic points in shapes and curves. In this way, every text or image has a unique set of points that later is used for distinguishing them from other objects, even if they are similar in some way.
- Model recognition: in comparison with image recognition, here it is implemented another type of algorithm that checks the shape and curves of a specific object. For instance, it is not necessary to analyze and, therefore, to recognize inner points in the shape of the object.
- Tracking of the targets: another essential feature of this software is that the algorithm is capable of keeping the tracking of the object even when it is placed in a movement or if the camera is out of view. This is a distinguished spotlight that raises Vuforia over other object recognition softwares.

All these experiences are discussed afterward in the development stage in Section 3.1.

Following the understating of this software, it is necessary to mention the different elements that compose the architecture of Vuforia, of which the main ones are:

- Camera: this section is in charge of capturing the image from the device and sending it to the pixel converter for later analysis by the *Tracker*.
- Database: it could be both local or on the cloud, and it is responsible for keeping the collection of *Targets* for their later recognition by the *Tracker*. This is created by the tool given by Vuforia known as *Target Manager*.
- Tracker: analyses the image captured from the camera, detects the different objects presented in every frame, and compares them with the objects available in the *Database*.

- **Target:** it has the goal of being the object in the real world that the *Tracker* recognizes. There are mainly two types: *Image Targets*, represented as graphic illustrations such as photos, covers of books and magazines, posters, etc; and *Word Targets*, which represent all those objects recognized by texts or words (both for an entire word or characters).

Figure 2.1 shows the general architecture of Vuforia, which explains the sequence of recognition from an image from when the frame is captured until the object is rendered. More detailed, the device captures a scene, such as a live video, by using a camera. As it will be commented, this fact is important as Vuforia does not recognize any capture device that the Operating System (OS) does not recognize as a camera. After this, the SDK takes a frame, that is, a particular one from the sequence of images from the captured scene, and transforms it into a different frame with a special resolution for a later proper read from the Tracker. Later, the SDK analyzes the frame with the *Tracker* and searches for similarities in the *Database* composed by *Targets*. Finally, this software renders some visual content (such as images, 3D objects, models, videos,...) on the device's screen, thereby creating an AR.

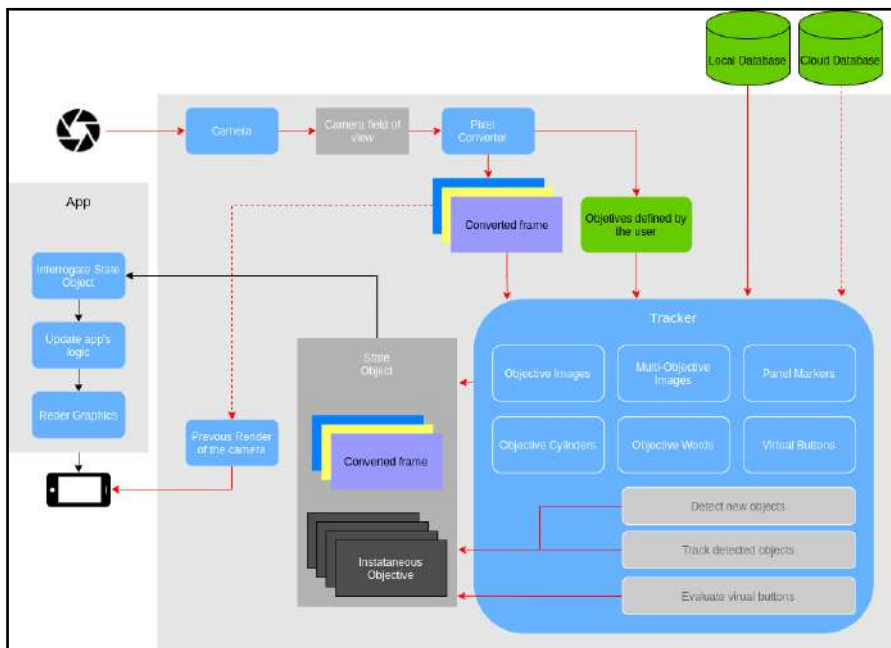


Figure 2.1 Architecture of Vuforia Engine [Daraghmi, 2021]

Hololens

Hololens, see Figure 2.2, enters in the sector of what is called "*Smart Glasses*" as its purpose is to generate a mixed reality or what is known as Augmented Reality. This term adopts the idea of the inclusion of computational sensations such as visual and auditory simulations in the interaction of a real-world environment.



Figure 2.2 HoloLens 1st Generation [Microsoft, 2021a]

This product, developed by Microsoft, collects a large quantity of applications focused more on the educational and medical sector [Microsoft, 2021a]. As a step in this project, the idea of implementing this device in the construction world generates more secure and precise work. One of the main applications in this sector is the one implemented in this project, corresponding to the simulation and recreation of a robot (such as an ABB robot arm) over a specific QR without the necessity of being in front of the real device, that is, without the obligation of exposing a human to a possible danger, by being close to the manipulator.

2.3 Robotics

As it is mentioned previously, robotics had taken over the industry decades ago. So is that now it is impossible to think about an industry that does not depend on any robotic machinery. The evolution of these machines has drastically changed proportionally as the technology advanced; in this way, automation, Machine Learning, AI are concepts present in many modern robotic devices.

This project has followed this path in the sense that it has used robot arms from the company ABB and the new concept integrated by Boston Dynamics, as the Spot robot for supervision purposes that has been released during 2020.

ABB Robot Arms

ABB Robotics is a company dedicated to manufacturing robotics, machine automation, and digital services focused on giving innovative solutions for a range of mul-

multiple industries and factories from the automotive and electronics side to a more logistic scope [ABB, 2021b].

Stepping into what is relevant for this project, Figure 2.3 shows the six Degrees of Freedom (DoF) robot arm that is used as the tool for demonstrating the immersion into the AR. Its industrial name corresponds to *ABB IRB2400*, and its applications are multiple, making it one of the most reliable and used robots in the manufacturing sector, highlighting its performance with welding, cutting material, sealing, and material handling between others [ABB, 2021c].



Figure 2.3 ABB IRB 2400 Industrial Robot [ABB, 2021c]

The data flow and the communication with the robot are operated by the *IRC5 Controller*, responsible also for keeping the system safe and providing an interface for its handling by any operator, both by a computer or by an integrated panel controller. All the software is installed and located in this device, which will communicate the position/trajectories of the joints to the robot between other applications [ABB, 2021d]. It can as well be operated by a high-level programming language known as RAPID.

Lastly, it is mandatory to mention the simulation tool used as a complement to the IRC5 and the robot arms, known as RobotStudio [ABB, 2021f]. This offline programming software substitutes the functions of the ABB controller as well as it can deploy robots with the same functionalities as the real ones. In this way, before testing into the real system, a full simulated deployment can be achieved with RobotStudio.

Figure 2.4 shows the interface for the IRB2400 robot, which, as it can be illus-

trated, provides a toolbar with many different functions. Regarding this project, the *Controller* section is mainly used as it is required to create a specific controller with all the Add-In and software packages for the connection with ROS.

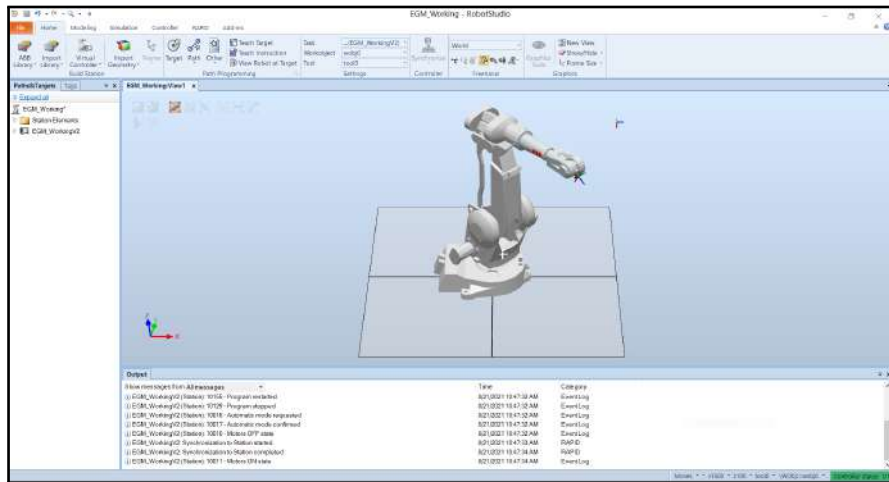


Figure 2.4 RobotStudio view of the IRB2400

Spot - Boston Dynamics

Spot, shown at Figure 2.5, is a manufactured mobile robot whose structure and physical shape contribute in agile motions that enable the robot to navigate difficult terrains, allowing the operator to automate routines and missions for inspection purposes and the collection of data from its different set of sensors [Boston Dynamics, 2021a].

Two features that create a gap between this robot, created by Boston Dynamics, and the rest of the current ones is the dynamical balance that it provides, making it complicated to tear down and proving the handling of up to 14 kilograms of payloads; and its camera integration, capable of generating a 360° perception. Thanks to this performance, Spot is considered ideal for the supervision of construction sites, which provides both challenging terrain and a significant amount of data to handle.

Aside from its physical considerations, Boston Dynamics provides an SDK that gives the control function over the API of the software and, therefore, to the control of the robot [Boston Dynamics, 2021c]. Conceptually, this Python-based SDK gives the possibility of creating automated missions to be done by the robot and handling all the data acquired from the cameras. This functionality generates a perfect paradigm for developers in order to generate whichever control is desired over the robot and under its specifications. One last mention about the system is that it incorporates a controller from which is illustrated the different cameras of



Figure 2.5 Spot Robot by Boston Dynamics [Boston Dynamics, 2021b]

the robot and controls the motion of the body. Also, from this device, it is possible to record the path so that it can be implemented autonomously by Spot.

Sensors

If the structure and motion of the robots could be metaphorically compared to a human's skeleton and muscular system, sensors would be the senses, generating a fully aware system such as the human sensory organs. Robots have the necessity of being integrated with sensors if the idea is to draw on an industrial or any required activity. In this manner, the variety of these electronic devices capable of measuring physical states such as distances, colors, forces, and signals can generate a massive versatility of functions for the same robot.

In this project three types of sensors that provides different functionalities to the IRB2400 robot are considered.

- Distance Sensor: Figure 3.18 shows the distance sensor that is coupled with the robot in order to measure safe distances between an object or the floor and the tool being used. In addition, it is used a total of three sensors per set to take an average between them. A total of three sets for keeping track of different directions for protecting the tool.

In particular, *VL53LIX* is a transceptor capable of measuring distances up to 400cm with a 27° view angle, and with one-millimeter resolution. Its receptor is composed of two photodiodes, and its transmitter is a 950nm laser [STMicroelectronics, 2018].

- Linear Sensor: this sensor measures motion along a single axis. In this case, it is composed of a spring loaded rod that can be compressed, converting this displacement into an electrical signal, therefore, giving an approximation of the compressed distance. For this project, it is applied in the tool position

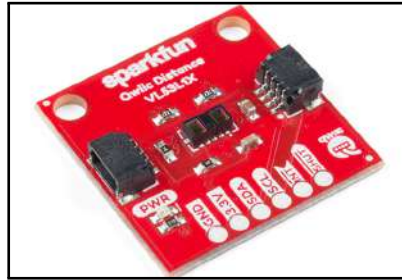


Figure 2.6 Sparkfun VL53L1X - Distance Sensor [Sparkfun, 2021]

to measure how deep the mentioned tool is compressed when it is pressed against any structure. More detailed, the linear sensor used has a maximum elongation of 14 centimeters down to a compression until 4 centimeters.

- **Force Sensor:** the idea of measuring the force being applied to a joint contributes to keeping the robot system from operating under significant torque forces. A standard sensor applied into robotics is characterized by six Degrees of Freedom. Three points are intended to control the x, y, and z-axis, and another set of three for analyzing the torque around each of the mentioned axes. In the case of this project, the *JR3 100M40A-i63* model is applied.

Robot Operating System

"The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications" [ROS, 2021a]. Actually, for a detailed description of what this open-source OS is about and its composition, we refer to the references and only briefly outline some main concepts and features below:

FileSystem: as in any other OS, ROS is divided into folders that describe different functionalities and contain distinct resources. More specifically, ROS is found divided into seven types of folders:

- **Packages:** they might contain libraries, nodes, datasheets, configuration files that form the main unit at an atomic level for organizing software, and they have the corresponding data to create a program.
- **Metapackages:** they serve to collect a group of packages and do not install any code.
- **Package Manifests:** Known as *package.xml*, they provide information about a package, including its name, version, description, and other meta information.
- **Repositories:** collection of packages that share the same version.

- **Message types:** is the description (or data) that a process (node) sends to another process.
- **Service types:** defines the request and response data structures for the services provided by the processes.
- **Workspace:** is the folder that embraces the packages in charge of containing the source files and the environment to process them. That is so that, it is divided into three subfolders: *source (src)* contains *CMakeLists.txt* for compilation purposes of the different packages that describe the processes and actually, the useful data; *build* which has the cache information and configuration of the packages; and *devel* used to test the programs without being installed and being shared to other developers.

Computation Graph level: it defines the network that connects all the processes together. Every process (or node) in the system that has access to the network is capable of interacting with other processes in the way of sending and receiving data. The basis concepts are the following:

- **Nodes:** are the files where the processes and the functions are conducted. Usually, a robot system is composed of many nodes, all targeted by the same network, defining different objectives. For example, one node could be responsible for handling a distance sensor, another node can compute the mentioned data from the sensor, and the last node is responsible for moving a robot arm in response to that computing.
- **Master:** provides the registration of names and the lookup service to the rest of the nodes and connects them. In the definition of the files in this project, it is usually used the command *roslaunch* for launching the Master and the corresponded nodes altogether, described in a script file (*launch file*).
- **Parameter Server:** part of the Master, gives the possibility to store data by the use of keys. With this, it is possible to change the configuration of a node while it is running.
- **Topics:** these are the names given for identifying a type of message. Furthermore, a topic can be published by a node, in which the data message is "transmitted"; and a topic can be subscribed by a node, in which a certain message is "received." In the example above, the node in charge of handling a distance sensor will publish a topic with the information about a measure; the node responsible for processing this data is subscribed to the mentioned topic for acquiring the information.
- **Services:** defined with a unique name, services are given by the nodes to provide synchronous "communication" between them. That is, a service needs a

request message in order to provide a response message to an external node. Imaging that the node responsible for the distance sensor needs to wait to a specific beacon in order to provide its data.

- Bags: used for storing data that is difficult to collect but necessary for developing and testing algorithms.

3

Development of the Digital Supervised Construction

As part of the development of every project, a training and a planning phase is necessary. In this manner, this chapter is dedicated to the explanation of all the considered steps for the achievements of the final results.

As commented beforehand, this project is primarily divided into two sectors: on the one hand, the generation of an augmented reality environment is developed with the simulation in real-time of the ABB arm IRB2400; and on the other hand, the process of automation of the Spot robot for the automated supervision of a scenario is commented .

3.1 Virtualization of an ABB Robot Arm

In this step, the tools and processes necessary for the simulation of a 3D-object ABB IRB2400 arm over a QR are mentioned . In order to recreate this environment, the use of Vuforia loaded into Unity, RobotStudio and ROS is needed. As a general view, ROS is responsible for the communication between Unity and RobotStudio in a sense that a movement generated in the arm run in RobotStudio will generate a movement, first in the game scene of Unity and secondly in an authentic recreation and simulation with the interaction of a real object such as a QR. For a further explanation, this process is described step by step along this section .

Simulation of IRB2400 into Unity

Before entering into simulations, one package of ROS is needed to be commented. In this case, the ROS package *MoveIt* [MoveIt, 2021] is used as the tool for publishing trajectories movements through a specific topic into the Unity software.

So is said, the first step taken in order to generate a movement through ROS in Unity comes with the use of *MoveIt*. This software is an open-source project aimed at encapsulating different robotics functionalities such as movements, kinematics,

and trajectories, all immersed in a ROS environment. In addition, with this tool, it is possible to generate different nodes and subscriptions for many robots, such as the robot arms given by ABB.

The way this tool is used in this project is, as it said, as a first approach to how the ROS communication needs to be built to give the necessary parameters to the Unity scenario. For this reason, an appropriate manner of visual implementation is by the use of *RVIZ*, which will be launched from the ROS setup and where specific files such as *abbirb2400.srdf* for the robot's joint definition and *kinematics.yalm* for simulate "real" movements are called. Figure 3.1 represents how the RVIZ scenario looks.

As a final comment about the image, *RVIZ* provides a handy tool for the planning and execution of trajectories. Furthermore, with the help of the ROS package *movegroup* the different states of the robot can be published as well as the transformation of the coordinates in order to get trajectories from the different joints. The button "Path" is defined as the desired final position, and with the button "Execute" it is triggered the chosen path, where different topics are published. In the case of this project, the ROS Topics *jointState* and *jointTrajectory* are considered as the most useful ones.

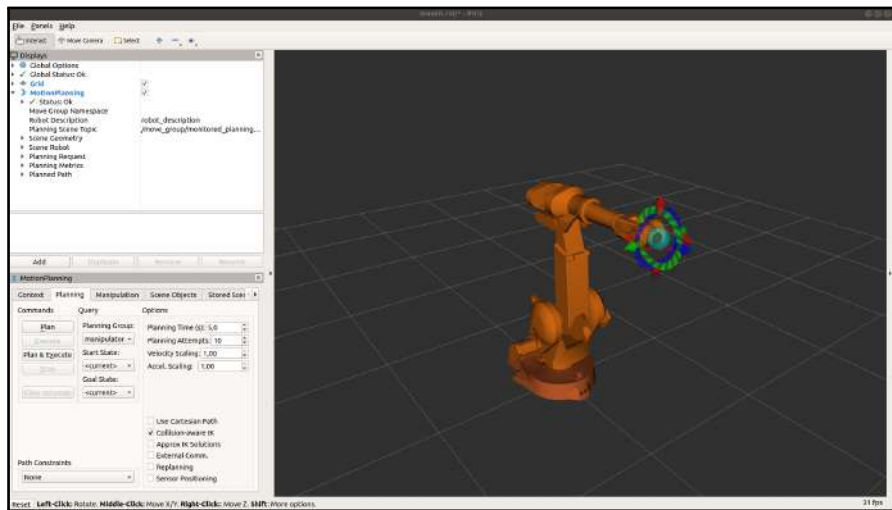


Figure 3.1 MoveIt! implementation of the ABB IRB2400

As a first stage, Unity provides the inclusion of Unified Robot Description Format (URDF) files, which, as its name specifies, collects in an XML format the information about any robot in a way that it is described the different joints, their connections, their hierarchy in the movement and their physical characteristics such as gravity or inertia. In this way, Unity is capable of incorporating it into the game

scene as a *GameObject*, where every segment or joint is described as an independent part. Figure 3.2 shows the implementation of the ABB IRB2400 3D model into Unity. It has to be mentioned that the original URDF file is given by *ROSIndustrial* [ROS-Industrial, 2021e], but even though it defines the *collisions* needed for an appropriate connection between the joints, it needs to be annexed in the URDF file *inertia* parameters for every joint so that it possible to simulate "real" movements in Unity, or at least similar to the ones implemented with *MovelIt*.

Check Annex A.1 in order to see the complete file.

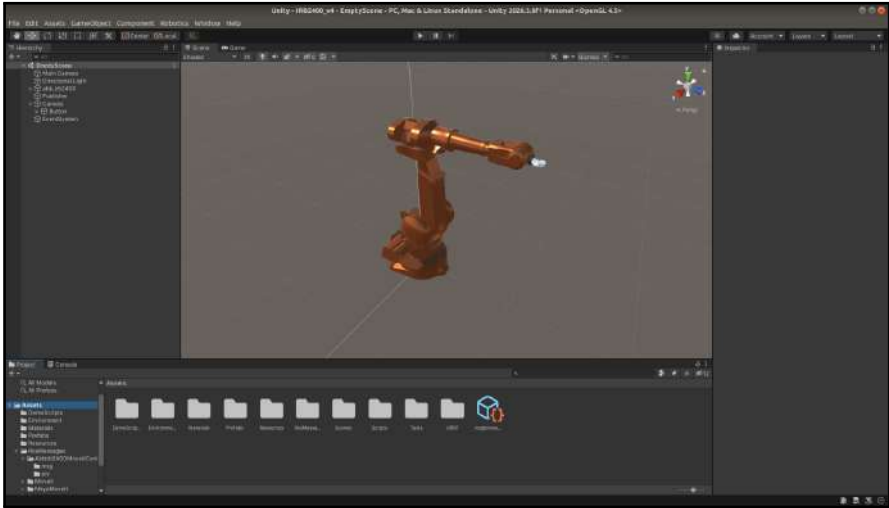


Figure 3.2 ABB IRB2400 built in Unity

Integration of Vuforia

The integration process of the package Vuforia into Unity is as any other Add-In. In the case of this project, the version corresponds to the 9.7.5 regarding the 2020.3 version of Unity. The main reason for this choice is due to the necessity of compatibility between these two versions. As followed at the webpage [Engine, 2021] these two environments have a complete match and non-background error in the importation and functionality.

As a general interpretation of what the Vuforia package contribute, Figure 3.3 shows the list of *GameObjects* that can be added into the Unity environment:

Regarding all this functions, specially in this project the following have been used:

- *ARCamera*: Besides all the options and characteristics that can be used in the regular Unity Camera, this object provides a particular function called

AR Camera	
Image Target	
Multi Target	
Cylinder Target	
Cloud Recognition	>
Model Target	
Object Target (3D scanned)	
VuMark	
Area Target	
Ground Plane	>
Mid Air	>
Session Recorder	

Figure 3.3 List of GameObjects given by Vuforia Engine

Vuforia Behaviour that contributes to the generation of the AR. This method depends on an internal License given by Vuforia whereby it provides object (shape, form, and configuration) recognition by the use of a Camera device, a recording, or from a simulator scenario (not discussed in this project). As a first comment, this method is not supported under Linux software, and this is why a Windows PC is needed. Secondly, having this object recognition being captured by a device camera will have a repercussion on the results of this project. The main reason is that the recognition is desired to be done by using an external input camera, in this case, given by the camera of the Spot robot.

- **ImageTarget:** this GameObject is, in general terms, a filtered image recognized by Vuforia’s algorithm. In order to generate it, an image uploaded and loaded from a Vuforia Database is needed beforehand. For this project, a QR has been used as the algorithm recognized it in full detail. Besides this feature, the Vuforia Engine can give tracking support of the mentioned image; that is, the detection is capable of knowing the orientation and rotation of the image to a certain level. In Figure 3.4, the tracking points that Vuforia’s algorithm is capable of generating from a QR can be seen:

In Figure 3.5 the integration of image recognition is illustrated with the use of a QR (differentiated from the one above). As it seems, the Unity scene generates an AR by implementing Vuforia in which the IRB 2400 model is simulated on top of a real QR. The result is considered sharp and efficient, referred to the delay existing between the recognition and the actual building of the AR.

- **ModelTarget:** similar to the ImageTarget referring it to the algorithm implemented for detecting forms and shapes, a ModelTarget can determine a certain position of an object giving beforehand a specific point of view of the mentioned object. In Figure 3.6 it can be seen an example of the targeting of a



Figure 3.4 Tracking point given by Vuforia's algorithm

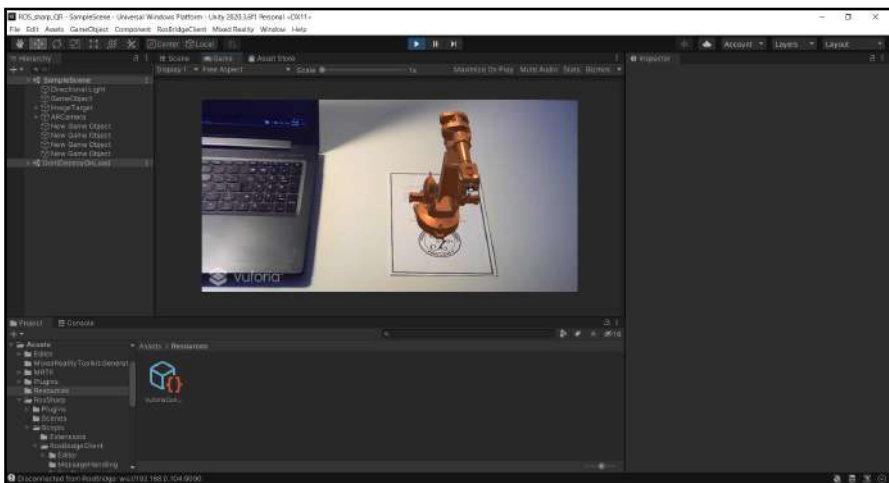


Figure 3.5 AR Simulation of the IRB2400 with the tracking of a QR

point of view of a 3D object IRB2400 giving by the use of a program called *Model Target Generator* [Vuforia, 2021b] given by Vuforia. Furthermore, it was tested and its result can be observed in Figure 3.7, where the generated model view from *Model Target Generator* is implemented and recognised by the software. For this situation, a 3D print model of the ABB IRB2400 robot model was used.

Even though the AR generation seems to be suitable, the lack of real-time feeling given by the existing delay in the object recognition, makes *Image-Target* the proper algorithm to be used in this project.

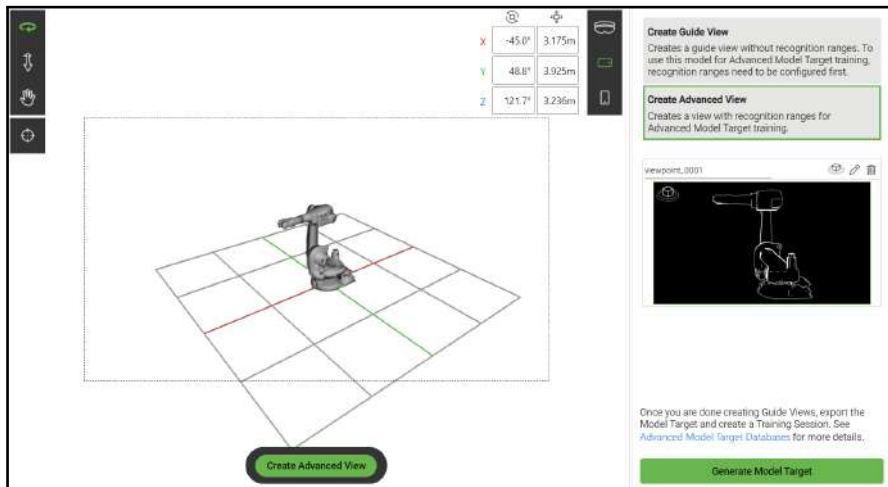


Figure 3.6 Target point of view given by Model Target Generator

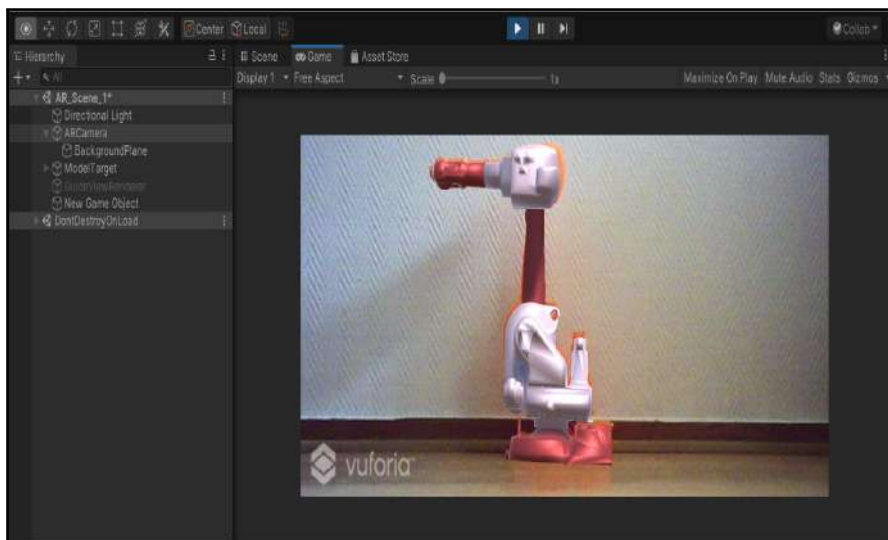


Figure 3.7 Vuforia recognition of a 3D model using Model Target algorithm

Integration of ROS into Unity

Giving more focus on the Unity side, a particular Add-In is needed for the connection between ROS and this environment. In a more detailed manner, the idea is to use a ROS bridge to generate a WebSocket between the two platforms. After several

project analyses, it was discovered the use of two possible Add-Ins:

- **Unity Robotics Hub:** is a repository that encapsulates tools, tutorials, resources, and documentation for the simulation of robots into Unity [Unity, 2021b]. It is a well-explained git repository where all the instructions necessary for the constructions of robots from URDF, and the recreation of movements with the use of written made ROS nodes, such as "Pick and Place" recreations is written. On the Unity side, this Add-In generates a series of methods that allow a TCP bridge connection by using ROS Services. In this sense, a specific GameObject into unity can receive and send ROS messages through the use of ROS Services.

Being more specific, Figure 3.8 shows an example of how to generate a request for ROS and how the response is handled, in this case, for the implementation of trajectories for the different joints obtained from the URDF file. In this defined C++ file is needed a method responsible for making the calls to ROS and for this case is called *StartPose* which calls the request method every given time given by a loop. This, as it will be commented on afterward, will generate a delay problem as the call is slower than the rate of publishing of the specific topic. Figure 3.9 shows the ROS service code in charge of handling the data from the two environments.

```

public void StartPosef()
{
    InvokeRepeating("PublishJoints", 1.0f, 0.5f);
}

public void PublishJoints()
{
    MoverService_V2Request request = new MoverService_V2Request();
    request.joints_input = CurrentJointConfig();
    ros.SendServiceMessage<MoverService_V2Response>(rosServiceName, request, TrajectoryResponse);
}

void TrajectoryResponse(MoverService_V2Response response)
{
    if (response.joint_trajectory != null)
    {
        StartCoroutine(ExecuteTrajectories(response));
    }
}

private IEnumerator ExecuteTrajectories(MoverService_V2Response response)
{
    ...
}

```

Figure 3.8 Code example of the methods implemented with Unity Robotics Hub in Unity

On the other hand, on the ROS side two special nodes, one responsible for the generation of a TCP server and other responsible of handling the ROS Service in charge of transmitting the data are required.

- **ROS \ddagger :** is an open-source project aimed to communicate ROS with .NET applications such as Unity [Siemens, 2021b]. One of the main advantages that

```
IBR2400MoveitJoints joints_input
---
trajectory_msgs/JointTrajectory joint_trajectory
```

Figure 3.9 Code example of a ROS Service

has this package is that it is possible to subscribe a ROS topic to any *GameObject* using a C# file. In the same way, it introduces libraries to create ROS bridges and URDF importations directly from the ROS repositories. Figure 3.10 is an example of part of the code needed to subscribe a *GameObject* formed from joints (such as a robot attached by a URDF file) in which all the joints are bound to the *jointstates* topic.

On the ROS environment side, the communication is held by the usage of a special ROS package called *FileServer* [Siemens, 2021a]. This is responsible for creating a node applying ROS services where the desired topic is requested from Unity and responded to by ROS. As it can be imagined, an external packet is needed to create a communication bridge between the platforms. This is acquired by the package *ROSbridgeServer* [ROS, 2021b] which provides the WebSocket transport layer for the listening and replaying of ROS calls.

```
namespace RosSharp.RosBridgeClient
{
    @UnityScript | 4 references
    public class JointStateSubscriber : UnitySubscriber<MessageTypes.Sensor.JointState>
    {
        public List<string> JointNames;
        public List<JointStateWriter> JointStateWriters;

        8 references
        protected override void ReceiveMessage(MessageTypes.Sensor.JointState message)
        {
            int index;
            for (int i = 0; i < message.name.Length; i++)
            {
                index = JointNames.IndexOf(message.name[i]);
                if (index != -1)
                    JointStateWriters[index].Write((float) message.position[i]);
            }
        }
    }
}
```

Figure 3.10 Code example of the methods implemented with ROS# in Unity

The main difference between these two Add-Ins can be seen in the methods that they have in order to communicate with the ROS nodes. On the one hand, Unity Robotics Hub package defines a C++ file that generates a call to the ROS environment through a ROS bridge and by the use of a ROS service. The desired

ROS node receives the Service call, processes the data, and sends it back to the unity file. On the other hand, ROS \ddagger allows the unity file to receive information of a specific ROS Topic directly. That is, it is not necessary a bilateral communication between the ROS and Unity environment and because of this, the unity file has the same data rate as the publishing topic.

As it can be understood, the main difference is the delay that the Unity Robotics Hub causes for being a bilateral communication.

Communication between ROS and Robot/RobotStudio

One of the main subjects in this project is communicating the ROS environment with the real robot. In order to get a reliable method, the git repository given by ABB and supported by the international communities *Symbiotic*, *ROSin* and *ROSindustrial*, under the name of *abb_robot_driver* [ROS-Industrial, 2021a] is followed. More into detail, this content provides the ROS driver needed to communicate with the controller *IRC5* of the ABB robots. In order to generate such connection, two interfaces are set in the Robot Controller Core known as *Robot Web Services (RWS)* and *Externally Guided Motion (EGM)*, where each one has a dedicated scope of instructions and limitations. Both RWS and EGM are platforms or interfaces that allow developers to create custom applications to fully interact with an ABB robot.

Following the Figure 3.11, the two interfaces can be defined as follows:

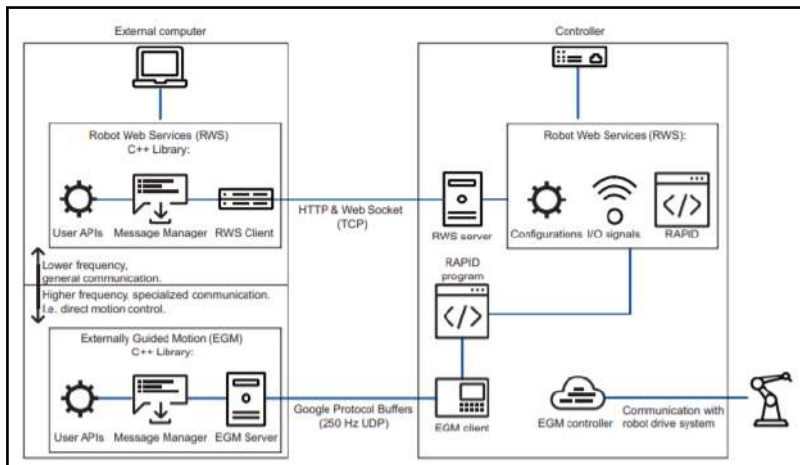


Figure 3.11 Block diagram of the communication between ROS and the ABB Robot Controller [ABB, 2021a]

- Robot Web Services [ABB, 2021e]: As it runs over a Transmission Control Protocol (TCP) connection, its primary purpose is to provide reading and

writing Input-Output (IO) signals and RAPID data to the controller, as well as checking the controller status. This project is used precisely for this purpose; RWS provides the starting and stopping signals to the RAPID program as the controller status is checked continuously.

This interface has limitations. As its primary purpose is to communicate simple signals, a complete status of the joints or the robot's movement can not be controlled. The main reason is in its composition; a Hypertext Transfer Protocol (HTTP) call has permanently attached a delay. Even if it is about a hundred milliseconds, complete real-time control is not supported.

The integration of this interface into ROS can be achieved by the use of the library *abb_librws* [ROS-Industrial, 2021c].

- Externally Guided Motion [ABB, 2021a]: Aimed for a fast User Datagram Protocol (UDP) connection with a top rate of 250Hz, the idea is to have control over parameters that could be considered as real-time behavior. EGM can provide the user a responsive motion control of the robot as well as joint, position, and velocity parameters.

Even though it is not aimed to be used as RWS but rather as a complement, it has some constraints. The IRC5 of the ABB robot is not configured to transmit trajectories but only to convey joint and pose modes. This, as it will be commented on later, will generate some limitations to the user experience in AR.

The integration of this interface into ROS can be achieved by the use of the library *abb_libegm* [ROS-Industrial, 2021b].

As an annex to these platforms, an Add-In complementary to these two called *StateMachine* Add-In [ABB, 2019] is used. Not entering into the detail of its functionality, this Add-In will provide RAPID modules that simplify the interaction of the RWS and EGM with the robot. In this way, and following the manual, the next images show the configurations followed for proper communication with the robot, or by default, with RobotStudio.

Figure 3.12 summarises all the installations that the systems need in order to implemented a complete EGM communication with the ROS environment. As it can be seen, in order to develop this platforms, at least a RobotWare 6.08 (controller version) is required and even though the RWS is not considered in the list, the addition of the *StateMachine* Add-In automatically installs some extra tools such as the mentioned RWS, Multitasking and PC-Interface connection, which are necessary as well for a full development in the ROS communication.

The following Figure 3.13 represents the basic configurations for the connection with the ROS environment that, in this case, is based on an external computer with an Internet Protocol (IP) address *192.168.0.104*.

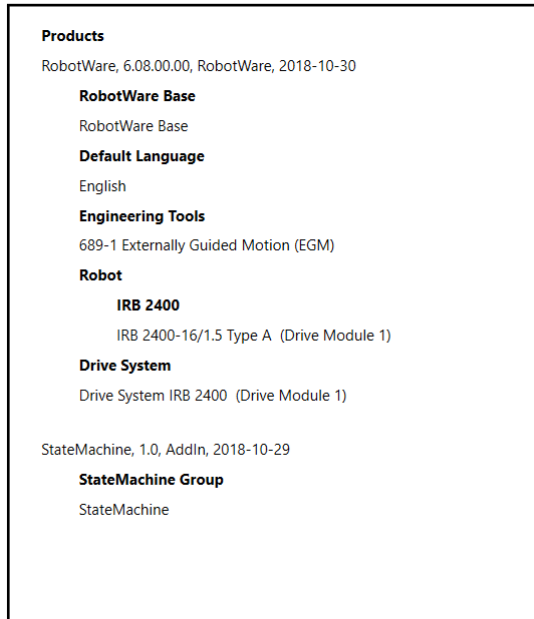
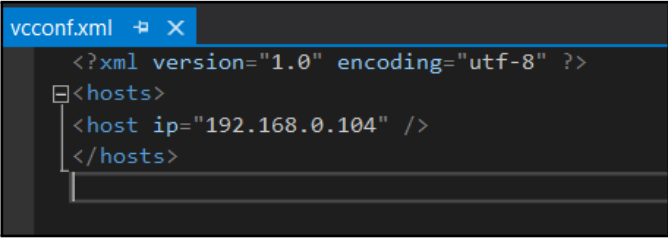


Figure 3.12 System Installation of the Virtual Controller

Type	Name	Type	Serial Port	Remote Address	Remote port number
Application protocol	ROB_1	UDPUC	N/A	192.168.0.104	6511
Connected Services	TCPIP1	TCP/IP	N/A	N/A	N/A
DNS Client	UCdevice	UDPUC	N/A	192.168.0.104	6510
Ethernet Port					
IP Route					
IP Setting					
Serial Port					
Static VLAN					
Transmission Protocol					

Figure 3.13 Communication parameters with the remote host PC (ROS)

Lastly, Figure 3.14 shows an extra Extensible Markup Language (XML) file needed for the configuration of the communication between ROS and the robots controller or by default, RobotStudio. The necessity of adding this file into *Users/AppData/Roaming/ABB Industrial IT/Robotics IT/RobVC* is not found in any manual but it is commented in the ABB forum [ABB Forum, 2021].



```
vcconf.xml
<?xml version="1.0" encoding="utf-8" ?>
<hosts>
  <host ip="192.168.0.104" />
</hosts>
```

Figure 3.14 vcconf.xml file for remote connection with a host PC (ROS)

Hololens

As mentioned in the introduction, the application of the Hololens can provide the user a full immersion in the AR scene. The implantation and the development of this tool into Unity will be the main responsible for changing from the *Unity Robotics Hub* to *ROS*‡ as the first one is not compatible with the package subject to the integration of the Hololens, known as Mixed Reality Toolkit (MRTK).

This open-source package gives the user an easy setup of all the necessary requirements for its integration into Unity and Vuforia and as well a set of features to accelerate the process of its development. As this being a product from Microsoft, the use of a Windows Operating System (OS), and as well as a special setup [Microsoft, 2021c] are required. First of all, the OS machine requires to accept *Hyper-V* setup for the virtualization of the platform. Secondly, it is necessary the use of *Visual Studio* as being the tool in charge of the debugging and the upload of the Unity program into the Hololens devices. In order to achieve it, Unity needs a unique feature called *IL2CPP* [Unity, 2021a] for the assembling of the code and later its debug. Finally, it is required to follow the set of instructions given by Vuforia in order to develop a correct AR scenario [Vuforia, 2021c].

When the MKTR platform is chosen in Unity, a set of changes are made in the scenario. As it can be seen in Figure 3.15, two new *GameObject* are set. *MixedRealityToolkit* is responsible for establishing the general setup of the play mode. With this, certain care is needed with the Camera as it needs to be configured properly to sustain the AR Camera set by Vuforia. Secondly, *MixedRealityPlayspace* is responsible for managing the AR camera as well as all the User Interface (UI); that is, this *GameObject* is in charge of generating the appropriate environment that is seen by the Hololens in the AR.

Within this project, the integration of *Hololens 1st gen* device carried out many problems. For being more precise, the information given was referred in many cases to use the second version of the Hololens. This gives a set of changes that do not fully immerse the AR, which is commented in the results chapter in Section 4.2.

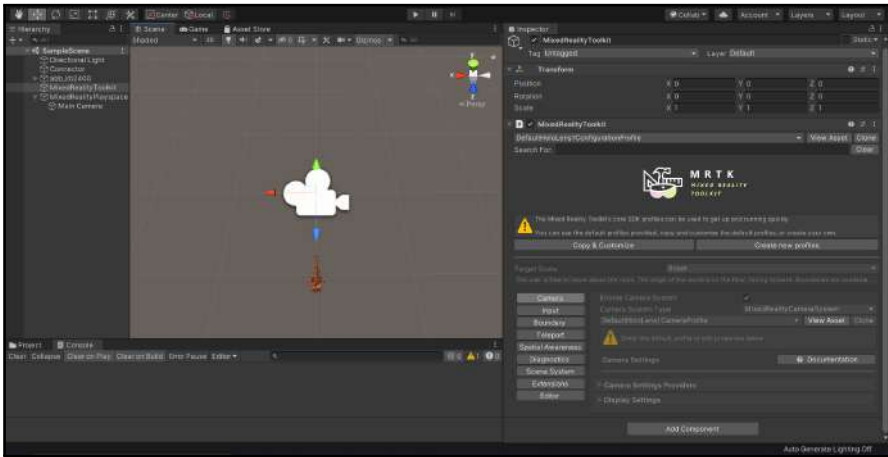


Figure 3.15 MKTR Scenario in Unity

Camera integration into Unity

One last stage concerning the idea of integrating an external camera into the AR scenario is necessary as a solution to the Hologens deployment. In addition, it is known that having a camera input into the Hologens is an idea that cannot be fulfilled as it does not make sense to recreate an external video. At the same time, these glasses are meant to render objects in a natural environment. For this reason, another solution is adopted, taking the real-time video from the Spot robot's camera and inserting it into Unity.

Once this is clear, the concept of integrating an external video while using the different features of Vuforia brings many problems. In general terms, the conclusion that it was achieved is that a "Camera device" is needed to deploy Vuforia's algorithm in image recognition. As the camera integration comes from the subscription of a specific ROS topic, these raw images can not be considered input of any "Camera Device" (as it could be a camera from an Android phone device) that Vuforia or even the OS Windows can provide. The answer that Vuforia brings is the recognition of these images in the cloud that Vuforia on its own provides by the known *Image Recognition Query* [Vuforia, 2021a].

This solution is omitted as the requirement of checking the image recognition via HTTP does not have real-time usage; that is, it causes a delay that can not be afforded in order to have a user-friendly AR scene.

A proper solution is presented in this project as an exchange of Vuforia's algorithm. In this case, the known ROS package *vispauto tracker* [ROS, 2021f] provides an algorithm dedicated to QR recognition, where different features are analyzed but where this project takes advantage of the computation of the center point of the mentioned QR. Once this center is obtained, the idea is to represent the ABB

IRB2400 robot model at that position and simulate it in the same way that is done with Vuforia, continuously tracking the center of the QR.

Figure 3.16 represents the functionality that this dedicated ROS package has for detecting the center of a QR. The result is expressed as a cardinal point in a plane of 20x20 points, where the cardinal zero is at the center of the display. It also gives a depth position (z coordinate) expressed as well as a relative distance to the camera. In this situation, the zeta value is not considered.

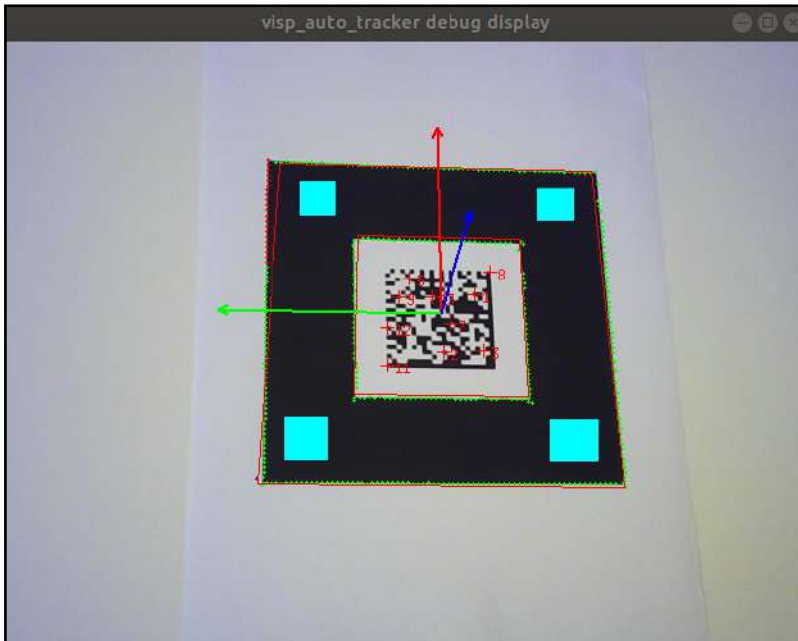


Figure 3.16 ViscAutoTracker ROS Package for a QR recognition

3.2 Spot Supervision

This section is aimed to mention the followed steps made for the incorporation of the Spot robot into the goal of the project. As said, the final scope is to develop ROS with Spot, which means that it is not required to use the main features given by the company Boston Dynamics.

Software Development Kit

The SDK acquired by Boston Dynamics [Boston Dynamics, 2021c] provides the general intentions that the company has in order to use this robot. In this project,

the step presented in the guide and given by the API are not followed, but we will rather give a general understanding of how it works and how the ROS developers made the corresponding packages.

The main feature that this SDK contributes to is the understanding of what a *mission* is. As Boston Dynamics defines, a *mission* is an action taken from the robot in order to do a specific path autonomously. As it can be imagined, this concept is completely integrated as the main idea is to have Spot supervising a scenario.

Besides the contribution of the concept, the following programming principles are followed by the SDK as in the ROS integration.

ROS integration into Spot

As an external functionality given by Boston Dynamics, integrating ROS into Spot's software is an issue that the robotics community hoped to appear. In this sense, the company ClearPath appears to take the responsible for integrating ROS into this robot. In order to install the software, a self-explanatory manual [ClearPath, 2021a] can be followed. The main idea is to install of full ROS package such as *Melodic* into the Spot's Core. This can be considered a tricky step as it is necessary to access remotely by the use of Secure Shell (ssh). The connection needed can be provided by a LAN network given by an Ethernet connection or by the employment of WiFi (which needs a Linux WiFi adapter connected to the Spot's Core).

Once this is applied, a ROS repository is needed to be integrated into the host PC, responsible for being the nexus with the robot. As the last step, it is mandatory to establish communication between the robot and the host computer. In order to achieve this, a bridge in the network interface of Spot (*/etc/network/interfaces*) [ClearPath, 2021b] must be set, detailing the IP addresses that the platforms have.

One feature that is not well fulfilled is the fact that the model given by RVIZ is defined for representing it in a stationary position where no physics laws are applied. This is actually a problem if simulations in both RVIZ and Gazebo are required. Nevertheless, here is where another open-source community comes into play, the known Champ [Champ, 2021], famous for having a development framework for building quadrupedal robots and developing new control algorithms for them in ROS. This provides the required URDF with all the necessary physical parameters and facilities Gazebo integration and algorithms for trajectories purposes such as one aimed to develop the creation of "the shortest path" without colliding with any obstacle.

Figure 3.17 shows the simulation of the Spot robot both with Gazebo and RVIZ. Both of these tools result from having different aspects of their reference positions in their use. On the one hand, RVIZ contributes to the selection of a goal point with the use of the button "2D Nav Goal" in which the shortest and more efficient path is found for a selected position or goal, avoiding collision. On the other hand, Gazebo will recreate a specific scenario; in this case, an environment given by *Champ* is used in order to simulate a construction field. As part of this, a QR is introduced

in this scenario so that the camera of the robot can track it and, with this, simulate the AR in Unity. For the attachment of this QR object into the *.world* model, which defines the Gazebo environment, a python code [Arguedas, 2021] is applied which converts PNG files into Gazebo models.

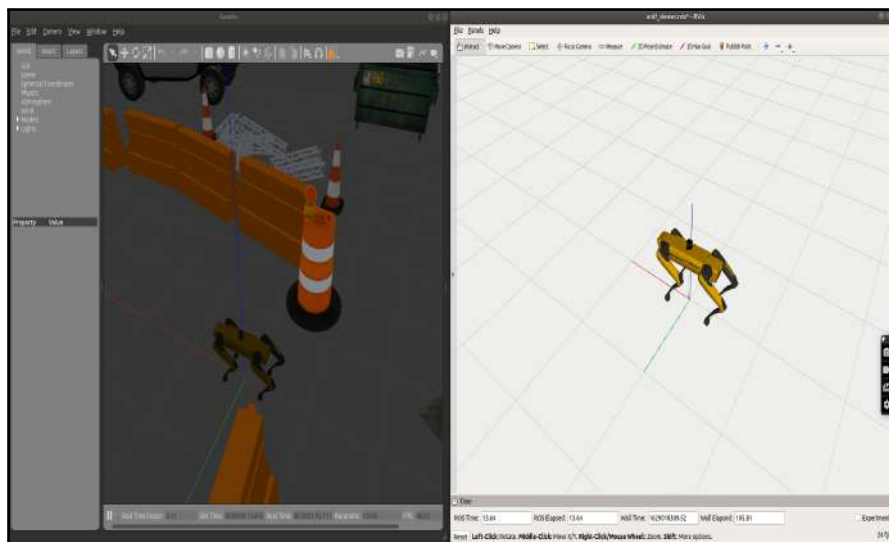


Figure 3.17 Integration of Spot with RVIZ and Gazebo

3.3 Integration of the sensors

This project applies three types of sensors commonly used as they are presented as usual complements to many robots. In this case, these are introduced into the AR in similar ways by the usage of the package *ROS₂*, in which with the help of auxiliary objects (such as a bar or a text), it is possible to represent the different measurements.

Distance Sensor

The distance sensor given for this project is the one that corresponds with Figure 3.18 where three sets of the transceptor VL53L1X are used for giving a more accurate measure of the distance (up to four meters). In addition, an external board module is used as a multiplexer of the different channels and as an I2C bus [Sparkfun, 2020] for its communication with the *D1 mini Pro v2.0* [Wemos, 2021] board which contains the WiFi module responsible for sending the data to the system's network.



Figure 3.18 Set of three VL53L1X integrated with D1 mini Pro v2.0

The integration of this sensor into the environment is done by the Arduino software downloaded into the *D1 mini* board which includes the ROS library necessary for creating and publishing topics through its serial communication. The name of this library is *rosserial_arduino* [ROS, 2021c], and it is well known for the use of Arduino devices as publishers of ROS environments. For this project, the type of messages given by the topics correspond to the sensor package known in ROS as *sensor_msgs* [ROS, 2021d] in which is specified the *Range* as it can be seen in Figure 3.19.

```
Header header
float32 min_range      # minimum range value [m]
float32 max_range     # maximum range value [m]
float32 range
```

Figure 3.19 ROS message type *Sensor_msgs/Range* [ROS, 2021e]

Once the topics from the sensors are published, there is a ROS node from the package *rosserial_server* that is subscribed to the mentioned topic and publishing it into the ROS environment. Furthermore, this is when *ROS#* comes into play as the *rosbridge* node will be communicating with Unity and, therefore, a script is responsible for taking the data published and introduce it into the AR.

The way of how the data is represented into the AR interface is done by the implementation of text that has an update rate of 10Hz and a colored bar that varies its size depending on the value of the measure. In the results chapter in Section 4.2

is explained in more detail.

Linear Sensor

As mentioned previously, the linear sensor is applied to control how deep a tool is compressed when it is pressed against a structure. As Figure 3.20 shows, there is a bar that is compressed so that a proportional electric value is obtained and, for that, a distance measure of how deep it goes. The data is given as an input to the same *D1 mini Pro* board from the other sensor so that it can transmit via WiFi to the system's network.

In the same way, the Arduino code integrated is practically the same as the one commented previously in Section 3.3, with the only difference that, in this case, the range data corresponds to an DC value that needs to be translated into a distance. This is done on the Unity side.

The ROS node used and the topic published by the node are similar to the one of the distance sensor.

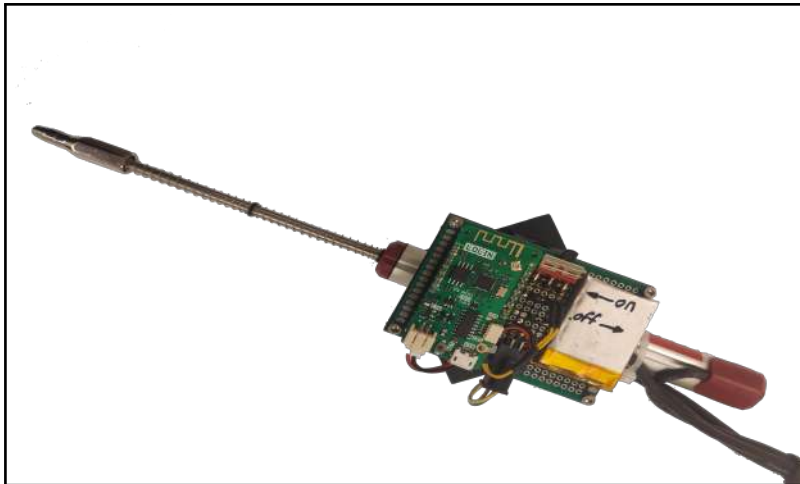


Figure 3.20 Linear potentiometer integrated with D1 mini Pro v2.0

Force Sensor

The introduction and implementation of this sensor in the project have been carried out differently compared to the sensors in Sections 3.3 and 3.3. For this situation, the acquisition of data from the sensor is provided in another computer outside the leading network. With this being said, the ROS call is applied from one machine to another using an ssh connection. The call from a Uniform Resource Identifier (URI) is set up to the primary ROS environment. Once both machines are connected, the

one in charge of taking care of the sensor publishes through a node the data over a topic on the other machine. In this way, the Unity side can directly be attached by the use of ROS#.

Figure 3.21 shows the sensor applied to the project.



Figure 3.21 JR3 Force Torque Sensor [NeuroRobotics - NTUA, 2021]

3.4 Final integration of the AR in Unity

Due to the availability of the different robots and the workstations, and the idea of testing first in safe spaces and not uploading software and programs into robots before being proved, a big part of this project has been carried out by simulations. Having RobotStudio for the development of the communication and movements of the ABB robot; Unity and Vuforia as the platform used to develop the AR without using phone applications or the HoloLens; and finally, using Gazebo and RVIZ as tools for simulating the Spot robot shows that all the steps of this project can be carried out with the use of the computer.

In this way, it can distinguish two scenarios in which the main difference is whether it uses robots or AR devices.

Simulation Scenario

In this situation, the program RobotStudio is implemented to simulate the IRC5 controller and the ABB IRB2400. In this way, it is mandatory to create a Virtual Controller, as was explained previously in Section 3.1, so that this can fulfill what a real controller and the robot can perform.

Secondly, Gazebo and RVIZ would further be used as platforms for the simulation of the Spot robot and its camera. In addition, a scenario or "world," similar to a construction site, where the simulated robot can move freely and where the QR is placed is generated. The robot's camera in this scenario is applied as the input for the AR environment created by Unity.

Lastly, the AR world is implemented on the *Play Mode* option given by Unity. It is known that this method is not the most accurate one regarding the actual reality that a computer screen can produce.

Real Scenario

In this second scenario, the ABB controller IRC5 and the IRB2400 substitute the program RobotStudio, the Hololens is used instead of the Unity game mode for the development of the AR, and finally, Gazebo and RVIZ, which are tools that can complement, are replaced by the real Spot robot.

4

Results

In the last step of this report, and as a continuation of the development process, a section about the results and the system obtained is presented as follows. The general scheme collects the topics discussed previously, where deep analysis is given so that the final system can be perfectly comprehended.

To show the impact of the results, along this section it is shared links to a web video portal where video content has been uploaded. This comes from the difficulty of presenting what actually is a graphical human-computer project, which concludes in being hard to explain through pictures and words.

4.1 General Network Architecture

As a first step in presenting the results of this project, it is compulsory to present a general schema of the general hardware architecture, see Figure 4.1, so that the explanation and a full picture of the whole system can be better understood. In this manner, and having in consideration the figure, the following can be commented:

- **Central computer:** a Linux equipped computer is necessary for the communication between all the parts of the system, which needs the incorporation of the at least three Ethernet ports and, thus, the handling of three UDP LAN connections: one dedicated to the communication with the IRC5 controller of the ABB robot; a second connection to the available WiFi router and giving support for the rest of the equipment; and the last one attached to a switch, responsible of connecting it with an external computer, where RobotStudio and Unity are handled.
- **Router:** responsible for creating the LAN network to communicate all the auxiliary units with the central computer. For this, a specific network is created under the dedicated IP *192.168.101.x*, with a prefix of *255.255.255.0*.
- **Switch:** responsible for connecting the central computer with the auxiliary one. In this case, the dedicated IP is *192.168.100.x*, with a prefix of

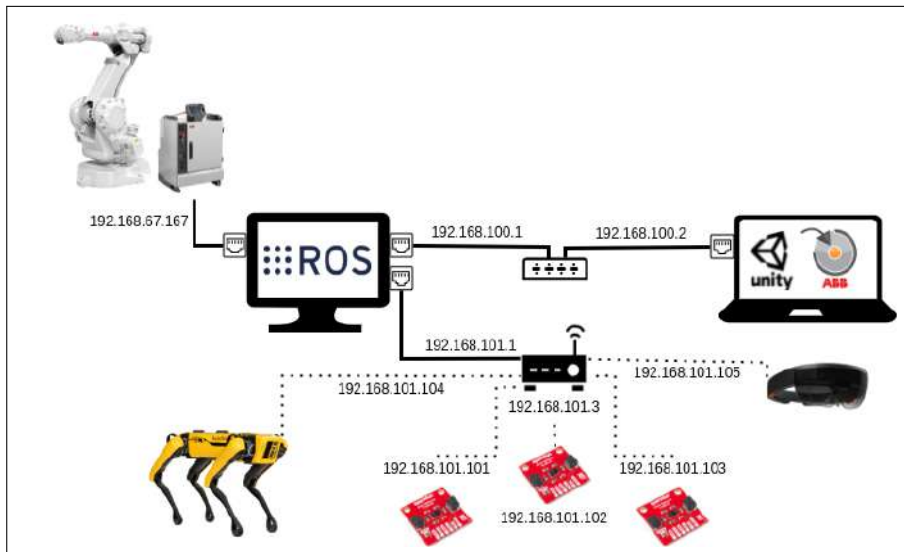


Figure 4.1 Possible Network Architecture

255.255.255.0. Actually, this device might not be necessary for the last step of the project. The main reason for sustaining an external computer is that RobotStudio and Vuforia are not well developed for Linux.

4.2 Generation of the Augmented Reality Scenario

Concerning the global solution in which an Augmented Reality scenario is generated, this project is separated into different steps taken to give a more straightforward explanation and show how it is developed. First of all, it is commented how the communication between the ABB controller and the Unity environment is provided; secondly, it is mentioned the integration of the camera, responsible for generating the AR scene; and lastly, it is added a real development of this application by the use of a commercial device such as the Hololens.

MoveIt implementation into Unity

As a previous step before the integration of the RWS and EGM and the RobotStudio environment, a simulation was required in order to test the communication between ROS and Unity. This step is not considered as the final results of the project, but it is good to know for possible future implementations.

As it is said in the development chapter in Section 3.1, *MoveIt* is a well known package regarding the use of ROS with robotics as it introduces many functions for the movement and position of the joints of different types of robots.

The Unity package for this process corresponds to *ROS TCP Connector* from *Unity Robotics Hub* instead of ROS \ddagger , and for its development, the tutorial from Unity-Robotics [Unity, 2021c] shows all the steps needed for the implementation of a *Pick and Place* mission with *MoveIt* on the ROS side. As the idea was not to follow a given trajectory but to give standalone movements from the *MoveIt* simulation, the code from this source was slightly modified. In Annex A.2 the code of the ROS python script for handling the service communication with Unity can be seen.

The following video shows the implementation of *MoveIt* movements into Unity:

<https://vimeo.com/617122954>

RWS and EGM communication with Unity

The implementation for the communication of ROS and the ABB controller comes from an already implemented project which is capable of transferring joint trajectories between *MoveIt* and the IRC5 controller reference. In this specific scenario, the communication was unidirectional, providing the data from ROS to the robot. As the idea of this project is to present the pose and movements of the robot into the AR, the communication needs to go in the other direction, that is, it is the robot that transfers the data to ROS. As a result of this, and as it is commented previously in Section 3.1, the package *abb_robot_driver* provides this possibility with the use of RWS and EGM.

On the other hand, the ROS environment is in charge of transferring the data given to the AR scenario implemented in Unity. The responsible of providing a connection is the package ROS \ddagger , which installed on the Unity side can obtain data from ROS given by a Websocket connection.

Figure 4.2 provides the main nodes responsible of generating a scene where the movement of the robot simulated in RobotStudio (or the real robot) is replicated by the simulated 3D object presented in Unity.

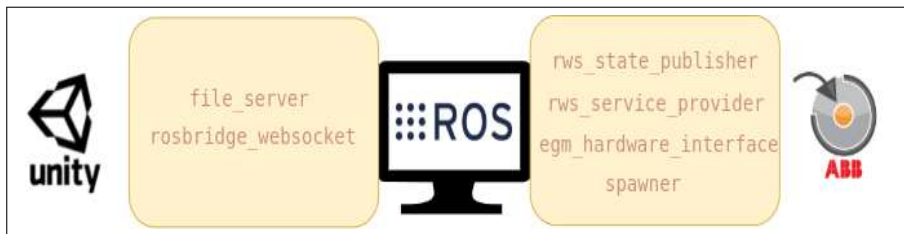


Figure 4.2 ROS Nodes used for the application of Unity with the ABB controller

In order to give a better understanding, the following nodes are commented:

- *rws_state_publisher*: in charge of setting up the connection with the controller in the sense that it checks all the requirements for setting up the communi-

cation. Beyond others, it defines a set of states; for example, it checks if the *StateMachine Add-In* is running and if the motors of the robot are on.

- *rws_service_provider*: defines a considerable number of services that are communicated via RWS. The ones used in this project and the most relevant are the services that start and stop RAPID codes and the ones that control the setting up of the EGM. In addition, it also defines the boolean method that checks if the *StateMachine* and the motors are on, among others.
- *egm_hardware_interface*: responsible for the data communication via EGM between two ports. It also defines the limits of velocity and acceleration of the joints, setting up a set of warnings.
- *spawer*: part of the *controller_manager*, responsible of setting the EGM controller for supporting a six joint robot (from a yalm file) and starting the controller in charge of publishing the joint states, *joint_state_controller*.
- *rosbridge_websocket*: part of the *rosbridge_suite* package, it is in charge of creating the WebSocket, that is, the bidirectional communication layer between clients that connects the ROS environment with the Unity platform.

On the Unity side, two specific scripts given by ROS[‡] are applied for the acquisition of the data given by the *Rosbridge* in which the joint position is applied to each of the joints of the 3D model. Specifically, these scripts are *JointStatePatcher* responsible for targeting the joints given by the URDF as a subscribe variable that is used by *JointStateSubscriber*. This other script is the one that intercepts the joint position topic (*\egm\jointstates*) given by ROS and applies them to the joint variables described by the other script.

In addition, it is remarkable to say that the delay existing between the movements of the real robot (or the one imparted by RobotStudio) compared to the ones showed in Unity is lower than one second, supplying the spectator a real-time feeling.

On the other hand, it is also mandatory to leave the concern of implementing the AR using the Vuforia package. So is that the following video represents the magic behind the creation of a simulated AR scenario given by the lecture of a QR.

<https://vimeo.com/617397834>

Camera integration

The application of an external camera running with ROS has many consequences in the way the AR is given. As it is commented previously, the Vuforia software is no longer helpful, and another concept needs to be integrated. This is where the ROS package *visp_auto_tracker* fulfills this requirement as it runs an algorithm capable of detecting QRs, giving its center point according to the camera scene and point of

view. Figure 4.3 shows the two nodes necessary to track a specific QR, and for this, they are explained more in detail.

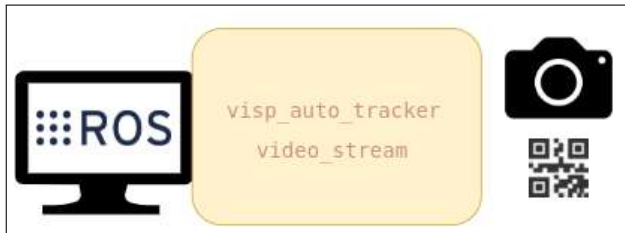


Figure 4.3 ROS Nodes used for the application of the camera and the QR detection

- `visp_auto_tracker`: corresponds to the software in charge of tracking over a QR code, or in this case a flash code, recognizing its shape and given its position linked to the screen of the camera. Furthermore, for better detection, it has been necessary to decrease the threshold for border detection, as it has been seen lately, there is a better detection when the flash code is inserted into Gazebo.
- `video_stream`: responsible for setting the camera that is wanted to be streamed. It gives the option about which stream provider sources it can be used, such as the webcam or an external USB camera, at a specific rate. Actually, the publishing rate frame (in frames per second) has a dual behavior with the `visp_auto_tracker` as the higher the rate, the slower the algorithm track the QR. The final result is a live video that has a delay of around 200 milliseconds.

On the Unity environment, the ROS nodes in charge of communicating and transmitting data are the same as the ones for the RWS and EGM implementation: `file_server` and `rosbridge_websocket`. In addition, the script used follows the same scheme as the one used in the joints positions, but in this case, the topic acquired is `/camera/raw/imagecompression`. This contains the frame's data sent by the camera, in the sense that every frame is published with a rate of 100Hz, contributing to the feeling of a live video in real-time.

Another step in this integration needs to be taken as a result of the incompatibility between the references used to express the position given by the algorithm of the `visp_auto_tracker` and the positioning of the screen of Unity's *Play Mode*. While in ROS, an x/y axis has values between $[0,20]$ in Unity are expressed between $[-1,1]$.

Lastly, a script is implemented whose function is to display the robot in the center position of the QR. For this, the robot is displayed once and only when the QR is detected, making it similar to what it is possible to get if the Vuforia software

is running. The C# Code can be seen in Annex A.3 and the ROS Launch file can be observed in Annex A.4.

For a better understanding of this behavior, the following link shows the cameras' integration into the system:

<https://vimeo.com/618158695>

Figure 4.4 shows the deployment of the rendered robot model centered over a QR when using a USB camera connected to ROS.

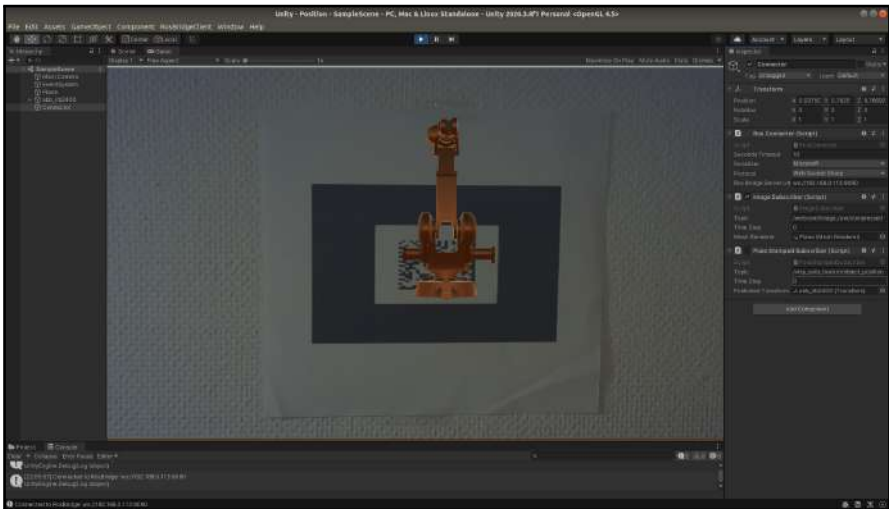


Figure 4.4 Integration of an external camera with ROS in Unity

Sensors integration

The integration of the different sensors into the AR is carried out in similar ways, that is, with the help of the nodes *rosserial* and *websocket* it is possible to be subscribed to the different topics given by the sensors from the ROS environment and capture them by the use of ROS# on the Unity side. Furthermore, a script for the representation of the values into the AR scene is needed. In this case, a text which shows the data is used and a representation of the measurement could be illustrated by a colored bar, for instance. In all the following cases, the Vuforia software for tracking the QR has been used and the ROS Launch file can be observed in Annex A.5.

Even though the development was similar for all the sensors, slightly differences are taken into account:

Distance sensors: For the representation of the three sets of distance sensors (each one has a set of three VL53L1X), a text and a bar are applied for each of them, with a color variation to make it more friendly to the viewer. The script is done according to the specifications: the text is modified with a rate of 100Hz, setting a new distances, and the bar sets its size according to the measurement. Figure 4.5 shows the final result, and in Annex A.6 it is possible to check the code.

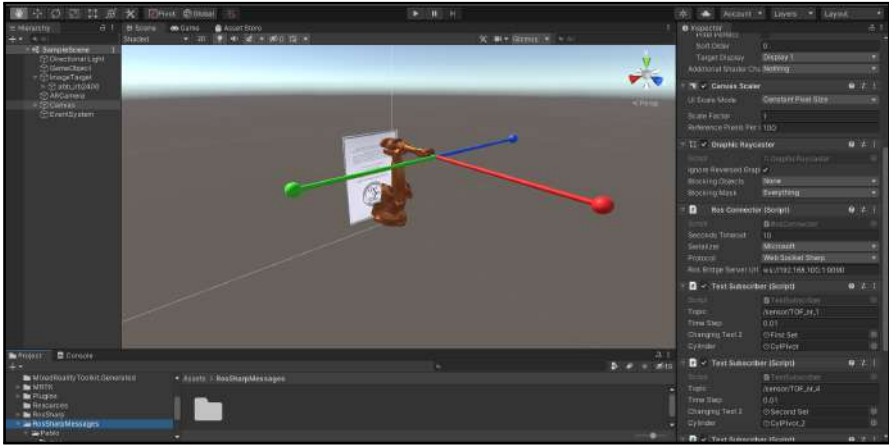


Figure 4.5 Distance sensors integration in Unity

The following link shows a live video of its behavior.

<https://vimeo.com/617397803>

Linear sensor: As it has been commented previously, the idea is similar to the one implemented for the distance sensors. The insertion of a bar to graphically see the changes in the depth measurements makes it ideal and straightforward for its representation. For this, a script runs where an interface text and the bar are adjusted so at the given rate of 100Hz, the same as in the other case, but a calculation is made so that it is shown that the linear sensor is compressing the distance. Figure 4.6 shows the final result, and in Annex A.6 follows the same code structure than the one implemented for the distance sensors.

The following link shows a live video of its behavior.

<https://vimeo.com/617397771>

Force sensor: Following the exact implementation of the other two cases, the manner of representing the force and the torque is given as it can be expressed in Figure 4.7. In this situation, a colored bar provides an idea of the z-axis force that

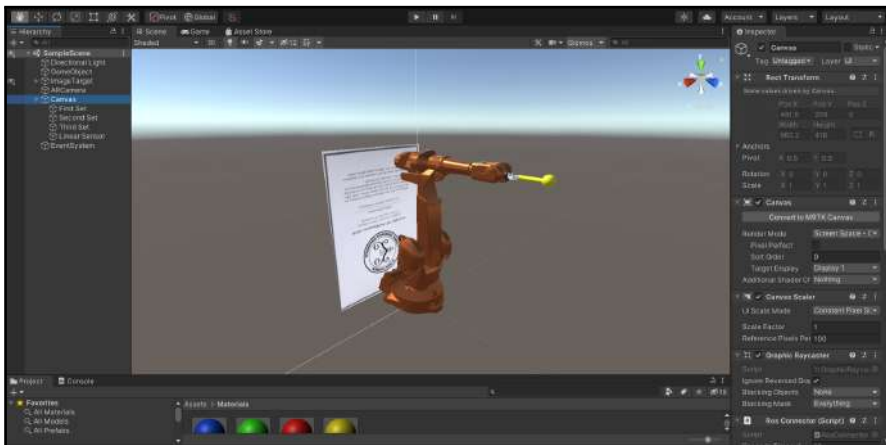


Figure 4.6 Linear sensor integration in Unity

is being applied, and a cylinder represents the rotation of the torque given to the mentioned axis.

Even though the communication between ROS and Unity goes through the same packages (*Websocket* and *ROS#*), the C# script responsible for moving the bar and the cylinder captures a *Float64MultiArray* from the ROS side, giving the opportunity to represent the two missing forces axis and the torques. Annex A.6 contains the corresponding code.



Figure 4.7 Force sensor integration in Unity

The following link shows a live video of its behavior.

<https://vimeo.com/640073495>

Hololens Development

The integration of the Hololens into this caused many problems. First of all, due to the annexing of a needed package called *Microsoft Mixed Reality Toolkit* into the project, the software given by *Unity Robotics Hub* was no longer compatible, so a more deep investigation was done until it was decided to integrate ROS[‡] as its substitute and to result from being a more easy and reliable software.

Secondly, the idea of using an external camera run operated with ROS was discarded as it is not possible to integrate a separate camera view into the Hololens' vision. The reason behind this is simply that the AR Camera needs to be centered in the *Game Scene* of Unity where there is no track of inserting an external view.

Even though the necessity of integrating these smart glasses into the project seems to be doubtful, its application into Unity is relatively straightforward.

4.3 Spot Supervision

The final integration of Spot into the project was not as expected. Due to several brake downs of the robot in which it was necessary to reinstall its Core, it was not possible to set ROS into it withing the allotted project time. For this reason, all the work has been simulated, and was carried out in Gazebo.

Software Development Kit

As far as the SDK given by Boston Dynamics is studied, it has been possible to acquire images from the different cameras from the robot. Nevertheless, the idea of deploying an AR environment in which the real-time sense is critical leaves to the side the concept of sending the camera's frames to the ROS environment if it is not given straightforward.

On the one hand, the intention of uploading the images from Spot into a server was well accepted. Nevertheless, thinking about sending images into the cloud, acquire with ROS, and sending them to the AR environment seems to have a higher delay compared to the idea of having ROS already deployed into the robot.

On the other hand, it has been thought about directly attaching a cable from Spot's networks card into the computer, even though this loses the magic of making this project completely autonomous.

ROS integration into Spot

The simulations carried out with the *Clearpath* and *Champ* packages are reliable and valuable. It has been possible to recreate in Gazebo a construction field in which the robot can move to specific positions automatically.

In addition, the topic responsible for Spot's simulated camera is possible to display by a streaming camera directly from the ROS system. With this said it is

possible as well to run the package *visp_auto_tracker* in order to track a QR displayed in the Gazebo scenario and send its position via a *Websocket* to the Unity environment so that it can be checked what actually the simulated robot is seeing in Gazebo, and therefore, reconstruct the AR scenario over it. The ROS Launch file can be consulted in Annex A.7

Figure 4.8 shows the tracking made to a QR inside the Gazebo scene using the camera of the simulated Spot.

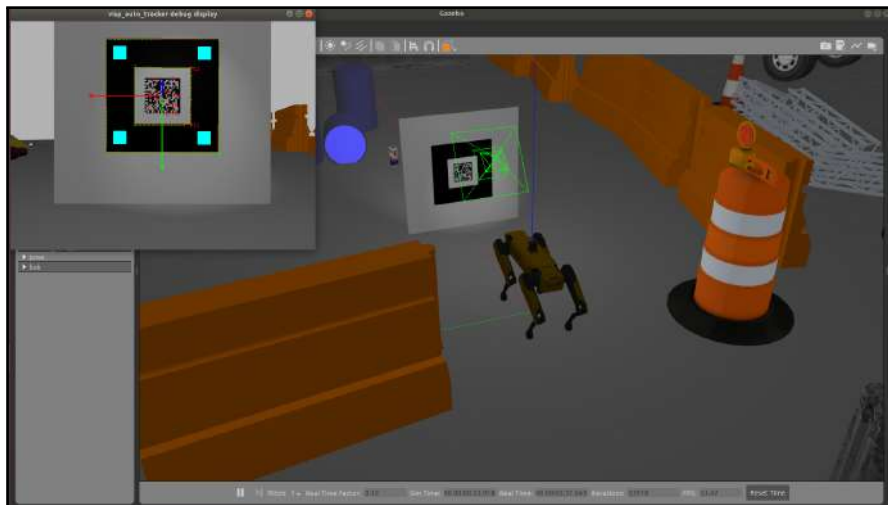


Figure 4.8 QR recognition by Spot’s camera simulated in Gazebo

The following link shows a live video of its behavior.

<https://vimeo.com/656080766>

5

Discussion

This chapter summarizes all the obtained results of the project and provides a general understanding of the main topics and their objectives. Along with those, other explanations are commented with the aim of clarify the conclusions of this project.

In this manner, the discussion is divided into three sections so that future implementations can be guided from each of the parts.

5.1 Virtualization of the Augmented Reality

With virtualization of the AR is meant all the considerations taken from the Unity side. First, it is essential to highlight that Unity provides a logic-friendly framework in which a complete and more advanced Augmented Reality scenario can be set, not only reflecting the significant amount of documentation on the web and the high liability of the software, capable of generating game scenarios in full detail and well rendered.

On the other hand, it is necessary to comment that the use of Vuforia has considerable limitations. As this package was considered since the beginning as the promising software being responsible for tracking real objects for the generation of the AR, it implied a lot time wasted regarding the necessity of tracing a moving object such as an operating robot. With this is said that the real reason behind the use of a simple QR for generating the AR is that it is impossible to track an object that is constantly moving. Also, it is fundamental to discuss that the approach of using a device camera makes it impossible to integrate a camera from the ROS side. In this sense, Vuforia, as a software, should not be that recommendable for this type of AR project, at least for now.

The idea of developing other types of software, such as *visp_auto_tracker* provides an example of how this project could have been the focus on more straightforward ways of reaching the same point.

Another point to discuss is the reason behind the use of *ROS₂* as the bridge between the ROS environment and the Unity side. As it is commented, *Unity Robotics Hub* offers as well a possible solution of this matter, but it is discarded mainly due

to its incompatibility with the *Mixed Tool Reality Toolkit* that Unity offers for the integration of the Hololens. This stroke of luck led to a faster solution given by *ROS* as the communication can be set as unilateral between the ROS and Unity side, compared to the request-answer method given by *Unity Robotics Hub*.

Lastly, there are different ways of graphically showing the integration of the measurements of the three sensors. Even though the results do not show a complex development, there is a reason behind it, no other than proving a simple interface where the data is shown by text and colors so that it is easier to focus, generating quick responses from the supervisor of the robot.

5.2 ROS infrastructure

Three main subjects must be highlighted regarding the integration of ROS in this thesis. On the one hand, the application of the RWS for managing services with the ABB controller and the EGM for the acquisition and submission of joint positions is completely recommendable and reliable regarding any type of project that requires communication with an ABB robot. In this way, the introduction of *abb_robot_driver* as the package for the generation of services under the RWS and EGM makes it a perfect candidate in comparison with using the standalone libraries of *abb_librws* and *abb_libegm*. It consists of a package software that, with the help of a *State Machine* is possible to follow a logical interpretation of how to operate and send instructions to the controller.

The second comment is about the generation of a WebSocket capable of broadcasting the topics to the Unity side. It has been proven that to have reliable real-time communication, a TCP connection seems to be the best candidate over the use of other considerations, such as the applications of web servers.

Lastly, the application of *rosserial* as the tool for the integration of the sensors seems to be easy as the topic is published directly and, in the case that some computation is needed, it is done on the Unity side.

5.3 Spot Robot

The implementation of Spot is delegated into a less critical role. Even though its capabilities are promising and its API functionalities are well defined and achieved, the use of its core for the installation of ROS generates uncertainty as this robot is in the initial steps of its manufacturing, and probably software changes will take place in the future.

Moreover, as the general concept was mainly to use the robot's camera, the idea of simulating the robot is well embraced. Gazebo is capable enough of giving a general idea of how the supervision and automatic route of the robot could work, and because of this, no more extended implementations have been taken into account. On the other hand, it is well known that the introduction of the real robot into the

project would have generated a more impressive result as it seems that a real, more than a simulated, scenario has a better implication over a more functional use.

5.4 Professional Utility of the project

This project completely fulfills what was desired as the final step of my educational career before being an engineer. Regarding all the new concepts and an advance in my software skills, the knowledge learned about ROS provides me a further interest regarding the operation of robots. This project provides professional development of a ROS program. On a secondary level, the use of the Unity framework and different packages such as Vuforia and *ROS#* provide me with the concept of generating some simple video games and Augmented Reality scenarios that I hope will be useful in the near future.

In addition, it can be considered that this project will have a beneficial impact on the Department of Automatic Control. Looking into short-term consequences, the introduction and use of the *abb_robot_driver* package into the different ABB robots of the *RobotLab* have a beneficial impact as it generates a complete communication scheme between the user and the functions of the robots. In the long term, the use of AR as an instrument for monitoring the robots with the integration of different sensors causes more useful implementations for the labor work carried out.

6

Conclusions

In this Master's Thesis, an Augmented Reality scenario of an operating robot is created for its supervision by the use of cameras. More into detail, the arm robot under study is the ABB IRB2400 belonging to the RobotLab of Lunds Tekniska Högskola. RobotLab operates the mentioned robot for educational and research purposes, and the scene is integrated with Unity, a video game engine. The AR scene created integrates an interface in which the IRB2400 is rendered as a 3D object. From this point, several functionalities have been developed.

Firstly, the replications of the movements carried out by the real robot are achieved by the use of the RWS and EGM of the ABB controller by the operation of ROS as the main operating system and the one that is used as the nexus between the ABB robot and the Unity environment. In this sense, RWS is in charge of controlling the main features of the controller, such as the RAPID codes and the starting of the EGM, interface responsible of obtaining the position of the different joints. As it is shown in the report, movements given by RobotStudio and the IRC5 controller are managed from the ROS side in a real-time scheme.

Secondly, two different approaches have been developed in order to produce the AR. On the one hand, Vuforia, a package compatible with Unity which integrates a recognition algorithm for detecting shapes of object by the use of a device camera. In this case, a QR code is used for the virtualization of the scene. On the other hand, a ROS QR detection package generates a similar behavior, so that the virtualization is applied into the center of the QR.

Thirdly, the integration of three different sensors into the interface of the scene have successfully been integrated. In this sense, distances and a linear sensors are represented both in text and in proportional colorful bars which graphically show the measurements. In addition, a 6-axis force sensor which is represented by arrows that indicate the pressure that it is being exerted has also been integrated. The way of publishing the ROS topics corresponds to the use of the *rosserial* package.

Furthermore, two Unity packages have been shown to work as Websocket bridges between the ROS and the Unity environment. Even though *Unity Robotics Hub* offers the possibility of generating a request/answer behavior between the two platforms, *ROS₂* has shown a lower delay as the communication can be kept as uni-

lateral. In this way, it has been shown that it is possible to generate C# codes that read the topics information directly from the socket, giving the possibility of moving the 3D robot arm as the real one. In addition, the implementations of the sensors have been adopted with this package as well.

Lastly, the Spot robot has been studied as the candidate for taking care of the supervision of the IRB2400 robot. Even though its API looks promising, it has been preferred to integrate ROS in its core so that everything is run under the same operating system. Due to the failure and the lack of time, Gazebo simulations have been carried out in order to test how the scenario would look like with the camera of the robot.

6.1 Future Work

As this thesis is considered to have started from scratch, several projects can have the advantage of using it as a start for developing new objectives. Apart from the references taken from the department, a lot of information was obtained from external researchers, manuals, and tutorials. So is that this thesis has collected a number of results that could be well applied to future considerations.

First, the importance of integrating the RWS and EGM packages under the same code in ROS needs to be mentioned. These applied with the StateMachine Add-In of the controller makes it possible to send RWS commands for controlling the RAPID software as the EGM, from which it is plausible to obtain and send the pose and joint states of the IRB2400 robot. By doing so, and considering all the information that is operated on and by the robot, it will be possible to set and correct path trajectories for future applications.

On a second stage, and more referred to the AR, the scene implemented into this project corresponds to a single robot with three sensors. This could be perfectly escalated to the whole RobotLab or to any industry that applies robot arms. With the help and good knowledge of ROS#, ROS, and Unity, it is possible to create much more complex and professional scenarios. Rather than using a QR, the virtualization is applied over other objects.

Lastly, it is pending to incorporate and replicate what Spot's API manages with ROS. Different cameras can be considered in order to recreate different AR scenarios.

Bibliography

- ABB (2019). “User manual Statemachine Add-in 1.1”. URL: <https://robotapps.blob.core.windows.net/appreferences/docs/cd504500-80e2-4cb6-9419-c60ea4ad6d56UserManual.pdf> (visited on 2021-04-20).
- ABB (2021a). “Application manual Externally Guided Motion”. URL: <https://abb.sluzba.cz/Pages/Public/OmniCoreRoboticsDocumentationRW7/Controllers/RobotWare/en/3HAC073318-001.pdf> (visited on 2021-04-10).
- ABB (2021b). *ABB Robotics*. URL: <https://new.abb.com/products/robotics> (visited on 2021-03-05).
- ABB (2021c). *IRB 2400*. URL: <https://new.abb.com/products/robotics/en/robots-industriales/irb-2400> (visited on 2021-03-05).
- ABB (2021d). *IRC5 - Industrial Robot Controller*. URL: https://new.abb.com/products/robotics/en/controladores/irc5_overview/irc5 (visited on 2021-03-10).
- ABB (2021e). *Robot Web Services*. URL: <https://developercenter.robotstudio.com/api/RWS> (visited on 2021-04-06).
- ABB (2021f). *RobotStudio*. URL: <https://new.abb.com/products/robotics/en/robotstudio> (visited on 2021-03-01).
- ABB Forum (2021). *Robot Web Services on multi VC*. URL: <https://forums.robotstudio.com/discussion/10544/robot-web-services-on-multi-vc> (visited on 2021-03-20).
- Arguedas, M. (2021). *Gazebo Models*. URL: https://github.com/mikaelarguedas/gazebo_models (visited on 2021-05-15).
- Azuma, R. T. (1997). “A survey of Augmented Reality”. URL: <https://direct.mit.edu/pvar/article/6/4/355/18336/A-Survey-of-Augmented-Reality> (visited on 2021-02-15).
- Boston Dynamics (2021a). *Spot*. URL: <https://www.bostondynamics.com/spot> (visited on 2021-02-20).

- Boston Dynamics (2021b). *Spot Robot*. URL: <https://www.bostondynamics.com/spot> (visited on 2021-02-25).
- Boston Dynamics (2021c). *Spot SDK*. URL: <https://dev.bostondynamics.com/> (visited on 2021-02-20).
- Champ (2021). *Champ Repository*. URL: <https://github.com/chvmp/champ> (visited on 2021-04-02).
- ClearPath (2021a). *Spot ROS package*. URL: http://www.clearpathrobotics.com/assets/guides/melodic/spot-ros/ros_usage.html (visited on 2021-03-12).
- ClearPath (2021b). *Spot ROS-setup package*. URL: http://www.clearpathrobotics.com/assets/guides/melodic/spot-ros/ros_setup.html (visited on 2021-03-12).
- Daraghmi, E. (2021). “Augmented Reality based mobile app for a university campus”. *ResearchGate* (). DOI: 10.13140/RG.2.2.36356.24962. (Visited on 2021-09-20).
- Ebert, C. and C. H. C. Duarte (2018). “Digital Transformation”. *Software Technology*. Ed. by I. Software. DOI: 10.1109/MS.2018.2801537. (Visited on 2021-07-15).
- Engine, V. (2021). *Vuforia Engine 9.8*. URL: <https://developer.vuforia.com/downloads/SDK> (visited on 2021-02-17).
- FBR (2021). *Hadrian X*. URL: <https://www.fbr.com.au/view/hadrian-x> (visited on 2021-09-20).
- Gazebo Forum (2021). *Why does my robot shake/wobble?* URL: <https://answers.gazebosim.org/questions/16700/revisions/> (visited on 2021-03-15).
- Microsoft (2021a). *Microsoft Hololens*. URL: <https://docs.microsoft.com/en-us/hololens/> (visited on 2021-06-15).
- Microsoft (2021b). *Unity: developing your first game with Unity and C#*. URL: <https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp#> (visited on 2021-02-16).
- Microsoft (2021c). *Windows Mixed Reality installation*. URL: <https://docs.microsoft.com/en-us/windows/mixed-reality/develop/install-the-tools> (visited on 2021-06-20).
- MoveIt (2021). *MoveIt*. URL: <https://moveit.ros.org/> (visited on 2021-03-08).
- NeuroRobotics - NTUA (2021). *Neurorobotics - ControlSystemsLab*. URL: <http://www.controlsystemslab.gr/main/robotics/> (visited on 2021-10-05).
- PTC (2021). *Vuforia: Market-Leading Enterprise AR*. URL: <https://www.ptc.com/en/products/vuforia> (visited on 2021-02-16).

Bibliography

- ROS (2021a). *ROS*. URL: <https://www.ros.org/> (visited on 2021-02-22).
- ROS (2021b). *ROS bridge*. URL: http://wiki.ros.org/rosbridge_server (visited on 2021-04-20).
- ROS (2021c). *Rosserial_arduino*. URL: http://wiki.ros.org/rosserial_arduino (visited on 2021-08-29).
- ROS (2021d). *Sensor_msgs*. URL: http://wiki.ros.org/sensor_msgs (visited on 2021-09-02).
- ROS (2021e). *Sensor_msgs/Range Message*. URL: http://docs.ros.org/en/api/sensor_msgs/html/msg/Range.html (visited on 2021-09-06).
- ROS (2021f). *Visp_auto_tracker*. URL: http://wiki.ros.org/visp_auto_tracker (visited on 2021-07-02).
- ROS-Industrial (2021a). *Abb_driver*. URL: https://github.com/ros-industrial/abb_driver (visited on 2021-03-20).
- ROS-Industrial (2021b). *Abb_libegm*. URL: https://github.com/ros-industrial/abb_libegm (visited on 2021-04-10).
- ROS-Industrial (2021c). *Abb_librws*. URL: https://github.com/ros-industrial/abb_librws (visited on 2021-04-10).
- ROS-Industrial (2021d). *IRB2400 URDF*. URL: https://github.com/ros-industrial/abb/blob/kinetic-devel/abb_irb2400_support/urdf/irb2400.urdf (visited on 2021-03-06).
- ROS-Industrial (2021e). *URDF files*. URL: <https://github.com/ros-industrial/abb> (visited on 2021-03-10).
- Schwab, K. (2015). *The Fourth Industrial Revolution*. Portfolio Penguin.
- Siemens (2021a). *File Server*. URL: https://github.com/siemens/ros-sharp/tree/master/ROS/file_server (visited on 2021-05-30).
- Siemens (2021b). *ROS sharp*. URL: <https://github.com/siemens/ros-sharp> (visited on 2021-05-30).
- Skydron (2021). *Aerial supervision with drones of constructions and civil works*. URL: <https://www.skydron.es/supervision-drones-construccion-obra-civil/> (visited on 2021-09-20).
- Sparkfun (2020). “BOB-16784 - TCA9548A”. URL: https://www.elfa.se/Web/Downloads/_t/ds/BOB-16784_eng_tds.pdf (visited on 2021-09-02).
- Sparkfun (2021). *Sparkfun Distance Sensor Breakout - 4 meter, VL53L1X*. URL: <https://www.sparkfun.com/products/14722> (visited on 2021-09-02).
- STMicroelectronics (2018). “VL53L1X - A new generation, long distance ranging Time-of-Flight sensor based on ST’s Flightsense™ technology”. URL: <https://datasheetspdf.com/pdf-file/1251025/STMicroelectronics/VL53L1X/1> (visited on 2021-09-03).

- Unity (2021a). *Manual IL2CPP*. URL: <https://docs.unity3d.com/Manual/IL2CPP.html> (visited on 2021-06-20).
- Unity (2021b). *Unity Robotics Hub*. URL: <https://github.com/Unity-Technologies/Unity-Robotics-Hub> (visited on 2021-04-05).
- Unity (2021c). *Unity Robotics Hub*. URL: https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/pick_and_place/1_urdf.md (visited on 2021-03-20).
- Vuforia (2021a). *How to perform an image recognition query*. URL: <https://library.vuforia.com/articles/Solution/How-To-Perform-an-Image-Recognition-Query> (visited on 2021-06-25).
- Vuforia (2021b). *Model Target Generator User Guide*. URL: <https://library.vuforia.com/articles/Solution/model-target-generator-user-guide.html> (visited on 2021-02-18).
- Vuforia (2021c). *Working with the HoloLens sample in Unity*. URL: <https://library.vuforia.com/articles/Solution/Working-with-the-HoloLens-sample-in-Unity.html> (visited on 2021-06-21).
- Wemos (2021). *D1 mini Pro*. URL: https://www.wemos.cc/en/latest/d1/d1_mini_pro.html (visited on 2021-09-02).

A

Code

A.1 URDF - ABB IRB2400

This URDF file is obtained with the help of the *ros-industrial* [ROS-Industrial, 2021d] and the *Gazebo* [Gazebo Forum, 2021] communities.

```
<?xml version="1.0" ?>
<!-- ===== -->
<!-- | This document was autogenerated by xacro from irb2400.xacro | -->
<!-- | EDITING THIS FILE BY HAND IS NOT RECOMMENDED | -->
<!-- ===== -->
<robot name="abb_irb2400" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- Conversion were obtained from http://www.e-paint.co.uk/Lab_values.asp
  unless otherwise stated. -->
  <!-- Actual colors -->
  <!-- MCS 2070-Y60R orange, standard robot color 1974 - May 2014 -->
  <!-- RAL 7035 light grey (abb graphite white), standard robot color May 2014 - current -->
  <!-- RAL 9003 signal white (abb white), cleanroom robot color -->
  <!-- RAL 7032 pebble gray, controller -->
  <!-- RAL 7012 basalt gray (abb dark gray) , robot primer -->
  <!-- RAL 7021 black gray (abb black) , robot with dark link color -->
  <!-- Other robot colours available, but not limited too. -->
  <!-- RAL 3001 signal red -->
  <!-- RAL 3002 carmine red -->
  <!-- RAL 5002 ultramarine blue -->
  <!-- RAL 5015 sky blue -->
  <!-- RAL 6002 leaf green -->
  <!-- RAL 9002 gray white -->
  <!-- RAL 9010 pure white -->
  <!-- approximations -->
  <!-- yellow, collision model -->
  <!-- link list -->
  <link name="world"/>
  <link name="base_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/base_link.dae"/>
      </geometry>
      <material name="">
        <color rgba="0.7372549 0.3490196 0.1607843 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/base_link.stl"/>
      </geometry>
      <material name="">
        <color rgba="1 1 0 1"/>
      </material>
    </collision>
  </link>
</robot>
```

```

    <inertial>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <mass value="148"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" izy="0" izz="0.001"/>
    </inertial>
  </link>
  <link name="link_1">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_1.dae"/>
      </geometry>
      <material name="">
        <color rgba="0.7372549 0.3490196 0.1607843 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_1.stl"/>
      </geometry>
      <material name="">
        <color rgba="1 1 0 1"/>
      </material>
    </collision>
    <inertial>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <mass value="127"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" izy="0" izz="0.001"/>
    </inertial>
  </link>
  <link name="link_2">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_2.dae"/>
      </geometry>
      <material name="">
        <color rgba="0.7372549 0.3490196 0.1607843 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_2_whole.stl"/>
      </geometry>
      <material name="">
        <color rgba="1 1 0 1"/>
      </material>
    </collision>
    <inertial>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <mass value="44.8"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" izy="0" izz="0.001"/>
    </inertial>
  </link>
  <link name="link_3">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_3.dae"/>
      </geometry>
      <material name="">
        <color rgba="0.7372549 0.3490196 0.1607843 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_3.stl"/>
      </geometry>
      <material name="">
        <color rgba="1 1 0 1"/>
      </material>
    </collision>
  </link>

```

Appendix A. Code

```
</material>
</collision>
  <inertial>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <mass value="41"/>
    <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
  </inertial>
</link>
<link name="link_4">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_4.dae"/>
    </geometry>
    <material name="">
      <color rgba="0.7372549 0.3490196 0.1607843 1"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_4.stl"/>
    </geometry>
    <material name="">
      <color rgba="1 1 0 1"/>
    </material>
  </collision>
  <inertial>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <mass value="17.9"/>
    <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
  </inertial>
</link>
<link name="link_5">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_5.dae"/>
    </geometry>
    <material name="">
      <color rgba="0.7372549 0.3490196 0.1607843 1"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_5.stl"/>
    </geometry>
    <material name="">
      <color rgba="1 1 0 1"/>
    </material>
  </collision>
  <inertial>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
  </inertial>
</link>
<link name="link_6">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/visual/link_6.dae"/>
    </geometry>
    <material name="">
      <color rgba="0.1882353 0.1960784 0.2039216 1"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/meshes/irb2400/collision/link_6.stl"/>
    </geometry>
```

```

    <material name="">
      <color rgba="1 1 0 1"/>
    </material>
  </collision>
  <inertial>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <mass value="0.3"/>
    <inertia ixx="0.001" ixy="0" izx="0" iyy="0.001" iyz="0" izz="0.001"/>
  </inertial>
</link>
<link name="tool0">
</link>

<link name="gripper_base">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_MainSupport.STL" scale = ".001 .001 .001" />
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_MainSupport.STL" scale = ".001 .001 .001" />
    </geometry>
  </collision>
</link>
<link name="servo_head">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ServoHead.STL" scale = ".001 .001 .001"/>
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ServoHead.STL" scale = ".001 .001 .001"/>
    </geometry>
  </collision>
</link>

<link name="control_rod_right">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_Rod.STL" scale = ".001 .001 .001"/>
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_Rod.STL" scale = ".001 .001 .001"/>
    </geometry>
  </collision>
</link>

<link name="control_rod_left">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_Rod.STL" scale = ".001 .001 .001"/>
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_Rod.STL" scale = ".001 .001 .001"/>
    </geometry>
  </collision>
</link>

```

Appendix A. Code

```
<link name="right_gripper">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ClampRight.STL" scale = ".001 .001 .001"/>
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ClampRight.STL" scale = ".001 .001 .001"/>
    </geometry>
  </collision>
</link>
<link name="left_gripper">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ClampLeft.STL" scale = ".001 .001 .001"/>
    </geometry>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://abb_irb2400_support/Gripper1/G1_ClampLeft.STL" scale = ".001 .001 .001"/>
    </geometry>
  </collision>
</link>

<!-- Disable Collisions -->
<disable_collision link1="right_gripper" link2="gripper_base">
</disable_collision>
<disable_collision link1="left_gripper" link2="gripper_base">
</disable_collision>

<!-- end of link list -->
<!-- joint list -->
<joint name="joint_world" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
  <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
<joint name="joint_1" type="revolute">
  <origin rpy="0 0 0" xyz="0 0 0"/>
  <parent link="base_link"/>
  <child link="link_1"/>
  <axis xyz="0 0 1"/>
  <limit effort="0" lower="-3.1416" upper="3.1416" velocity="2.618"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_2" type="revolute">
  <origin rpy="0 0 0" xyz="0.1 0 0.615"/>
  <parent link="link_1"/>
  <child link="link_2"/>
  <axis xyz="0 1 0"/>
  <limit effort="0" lower="-1.7453" upper="1.9199" velocity="2.618"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_3" type="revolute">
  <origin rpy="0 0 0" xyz="0 0 0.705"/>
  <parent link="link_2"/>
  <child link="link_3"/>
  <axis xyz="0 1 0"/>
  <limit effort="0" lower="-1.0472" upper="1.1345" velocity="2.618"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_4" type="revolute">
  <origin rpy="0 0 0" xyz="0.258 0 0.135"/>
  <parent link="link_3"/>
```



```

<child link="link_4"/>
<axis xyz="1 0 0"/>
<limit effort="0" lower="-3.49" upper="3.49" velocity="6.2832"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_5" type="revolute">
  <origin rpy="0 0 0" xyz="0.497 0 0"/>
  <parent link="link_4"/>
  <child link="link_5"/>
  <axis xyz="0 1 0"/>
  <limit effort="0" lower="-2.0944" upper="2.0944" velocity="6.2832"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_6" type="revolute">
  <origin rpy="0 0 0" xyz="0.085 0 0"/>
  <parent link="link_5"/>
  <child link="link_6"/>
  <axis xyz="1 0 0"/>
  <limit effort="0" lower="-6.9813" upper="6.9813" velocity="7.854"/>
  <dynamics damping="0.7" friction="0.0"/>
</joint>
<joint name="joint_6-tool0" type="fixed">
  <parent link="link_6"/>
  <child link="tool0"/>
  <origin rpy="0 1.57079632679 0" xyz="0 0 0"/>
</joint>
<joint name="joint_6-tool0" type="fixed">
  <parent link="tool0"/>
  <child link="gripper_base"/>
  <origin rpy="1.5707 0 -1.5707" xyz="0.0246 0 -0.01475"/>
</joint>
<joint name="servo_head_joint" type="fixed">
  <parent link="gripper_base"/>
  <child link="servo_head"/>
  <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
<joint name="control_rod_left" type="fixed">
  <parent link="servo_head"/>
  <child link="control_rod_left"/>
  <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
<joint name="control_rod_right" type="fixed">
  <parent link="servo_head"/>
  <child link="control_rod_right"/>
  <origin rpy="0 3.141 0" xyz="0 0 0"/>
</joint>
<joint name="gripper_joint_right" type="prismatic">
  <parent link="control_rod_right"/>
  <child link="right_gripper"/>
  <origin rpy="0 3.141 0" xyz="0 0 0"/>
  <limit effort="1" lower="-0.0257436" upper="0.0257436" velocity="1.0"/>
  <mimic joint = "gripper_joint_left" multiplier = "-1"/>
</joint>
<joint name="gripper_joint_left" type="prismatic">
  <parent link="control_rod_left"/>
  <child link="left_gripper"/>
  <origin rpy="0 0 0" xyz="0 0 0"/>
  <limit effort="1" lower="-0.0257436" upper="0.0257436" velocity="1.0"/>
</joint>
<!-- end of joint list -->
<!-- ROS base_link to ABB World Coordinates transform -->
<link name="base"/>
<joint name="base_link-base" type="fixed">
  <origin rpy="0 0 0" xyz="0 0 0"/>
  <parent link="base_link"/>
  <child link="base"/>
</joint>
</robot>

```

A.2 ROS script for handling the service communication

```
#!/usr/bin/env python

from __future__ import print_function

import rospy

import sys
import copy
import math
import moveit_commander

import moveit_msgs.msg
from moveit_msgs.msg import Constraints, JointConstraint, PositionConstraint,
    OrientationConstraint, BoundingVolume
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory
from moveit_msgs.msg import RobotState
import geometry_msgs.msg
from geometry_msgs.msg import Quaternion, Pose
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list

from abb_irb2400_moveit_config.srv import MoverService_V2, MoverService_V2Request, MoverService_V2Response

joint_names = ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6']

if sys.version_info >= (3, 0):
    def planCompat(plan):
        return plan[1]
else:
    def planCompat(plan):
        return plan

def ReceiveMovement(data):

    global trajectory
    global last_trajectory

    #destination = data.position
    if(data != None):
        trajectory = data
        last_trajectory = trajectory
    else:
        trajectory = last_trajectory

def plan_trajectory(move_group, destination_pose, start_joint_angles):
    current_joint_state = JointState()
    current_joint_state.name = joint_names
    current_joint_state.position = start_joint_angles

    moveit_robot_state = RobotState()
    moveit_robot_state.joint_state = current_joint_state
    move_group.set_start_state(moveit_robot_state)

    print(destination_pose)

    move_group.set_joint_value_target(destination_pose)
    plan = move_group.plan()

    if not plan:
        exception_str = """
        Trajectory could not be planned for a destination of {} with starting
        joint angles {}.
        Please make sure target and destination are reachable by the robot.
        """.format(destination_pose, destination_pose)
        raise Exception(exception_str)

    return planCompat(plan)

def plan_copy_movement(req):
```

```

response = MoverService_V2Response()

response.joint_trajectory = trajectory
return response

def moveit_server():
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('IBR2400_moveit_server')
    rospy.Subscriber("joint_path_command", JointTrajectory, ReceiveMovement)
    s = rospy.Service('IBR2400_moveit', MoverService_V2, plan_copy_movement)
    print("Ready to simulate")
    rospy.spin()

if __name__ == "__main__":
    moveit_server()

```

A.3 C# script for QR detection with AR simulation

```

using UnityEngine;

namespace RosSharp.RosBridgeClient
{
    public class PositionSubscriber : UnitySubscriber<MessageTypes.Geometry.
        PoseStamped>
    {
        private Vector3 position;
        private bool isMessageReceived;

        protected override void Start()
        {
            base.Start();
        }

        private void Update()
        {
            if (isMessageReceived)
                ProcessMessage();
        }

        protected override void ReceiveMessage(MessageTypes.Geometry.
            PoseStamped message)
        {
            if(message.pose.position.z != 0){
                position = GetPosition(message).Ros2Unity();
                isMessageReceived = true;
            } else {
                transform.position = new Vector3(0,0,20);
            }
        }

        private void ProcessMessage()
        {
            transform.position = position;
        }

        private Vector3 GetPosition(MessageTypes.Geometry.PoseStamped message)
        {
            double position_y = (message.pose.position.x)*(-20);
            if (position_y < (-9.9)){
                position_y = (-9.9);
            } else if (position_y > 9.9){
                position_y = 9.9;
            }
            double position_z = (message.pose.position.y)*(-10);
            if (position_z < (-4.9)){
                position_z = (-4.9);
            } else if (position_z > 4.9){
                position_z = 4.9;
            }
        }
    }
}

```

```

        return new Vector3(
            (float)(-8),
            (float)position_y,
            (float)position_z-(float)(0.3));
    }
}
}

```

A.4 ROS Launch File - QR Tracking Integration

```

<?xml version="1.0"?>
<launch>
  <arg name="robot_ip" doc="The robot controller's IP address"/>

  <!-- Enable DEBUG output for all ABB nodes -->
  <arg name="debug" default="false"/>
  <env if="$(arg debug)" name="ROSCONSOLE_CONFIG_FILE" value="$(find
    abb_robot_bringup_examples)/config/rosconsole.conf"/>

  <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch">
    <arg name="port" value="9090"/>
  </include>

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

  <!-- Launch the tracking node -->
  <node pkg="visp_auto_tracker" type="visp_auto_tracker" name=
    "visp_auto_tracker" output="screen">
    <param name="model_path" value="$(find visp_auto_tracker)/models"/>
    <param name="model_name" value="pattern" />
    <param name="debug_display" value="True" />

    <remap from="/visp_auto_tracker/camera_info" to="/webcam/camera_info"/>
    <remap from="/visp_auto_tracker/image_raw" to="/webcam/image_raw"/>
  </node>

  <!-- launch video stream -->
  <include file="$(find video_stream_opencv)/launch/camera.launch" >
    <!-- node name and ros graph name -->
    <arg name="camera_name" value="webcam" />
    <!-- means video device 0, /dev/video0 -->
    <arg name="video_stream_provider" value="4" />
    <!-- set camera fps to (if the device allows) -->
    <arg name="set_camera_fps" value="5"/>
    <!-- set buffer queue size of frame capturing to (1 means we want the
      latest frame only) -->
    <arg name="buffer_queue_size" value="1" />
    <!-- throttling the querying of frames to -->
    <arg name="fps" value="5" />
    <!-- setting frame_id -->
    <arg name="frame_id" value="webcam_optical_frame" />
    <!-- camera info loading, take care as it needs the "file:/// " at the
      start , e.g.:
      "file:///$(find your_camera_package)/config/your_camera.yaml" -->
    <arg name="camera_info_url" value="" />
    <!-- flip the image horizontally (mirror it) -->
    <arg name="flip_horizontal" value="false" />
    <!-- flip the image vertically -->
    <arg name="flip_vertical" value="false" />
    <!-- visualize on an image_view window the stream generated -->
    <arg name="visualize" value="true" />
  </include>

  <!-- ===== -->
  <!-- Robot Web Services (RWS) related components. -->
  <!-- ===== -->

```

```

<!-- RWS state publisher (i.e. general states about the robot controller) -->
<include file="$(find abb_rws_state_publisher)/launch/rws_state_publisher.
  launch">
  <arg name="robot_ip" value="$(arg robot_ip)"/>
</include>

<!-- RWS service provider (i.e. starting/stopping the robot controller's RAPID
  execution) -->
<include file="$(find abb_rws_service_provider)/launch/rws_service_provider.
  launch">
  <arg name="robot_ip" value="$(arg robot_ip)"/>
</include>

<!-- ===== -->
<!-- Externally Guided Motion (EGM) related components. -->
<!-- ===== -->

<!-- EGM hardware interface (i.e. 'ros_control'-based interface for interacting
  with mechanical units) -->
<include file="$(find abb_egm_hardware_interface)/launch/egm_hardware_interface.
  launch">
  <arg name="base_config_file" value="$(find
    abb_robot_bringup_examples)/config/ex2_hardware_base.yaml"/>
  <arg name="egm_config_file" value="$(find abb_irb2400_moveit_config)/config/
    joint_limits.yaml"/>
</include>

<!-- Put 'ros_control' components in the "egm" namespace (to match the hardware
  interface) -->
<group ns="egm">
  <!-- Load configurations for 'ros_control' controllers on the parameter server
    -->
  <roscpp param file="$(find abb_robot_bringup_examples)/config/ex2_controllers.yaml"
    command="load"/>

  <!-- Two 'ros_control' controller spawners (stopped for the controller that
    command motions) -->
  <node pkg="controller_manager" type="spawner" name="started" args=
    "egm_state_controller joint_state_controller"/>
  <node pkg="controller_manager" type="spawner" name="stopped" args="--stopped
    joint_group_velocity_controller"/>
</group>
</launch>

```

A.5 ROS Launch File - Vuforia Integration

```

<?xml version="1.0"?>
<launch>
  <arg name="robot_ip" doc="The robot controller's IP address"/>

  <!-- Enable DEBUG output for all ABB nodes -->
  <arg name="debug" default="false"/>
  <env if="$(arg debug)" name="ROSCONSOLE_CONFIG_FILE" value="$(find
    abb_robot_bringup_examples)/config/rosconsole.conf"/>

  <include file="$(find rosbridge_server)/launch/rosbridge_websocket.
    launch">
    <arg name="port" value="9090"/>
  </include>

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

  <node pkg="roscpp" type="socket_node" name="roscpp_server"
    />

  <!-- ===== -->
  <!-- Robot Web Services (RWS) related components. -->
  <!-- ===== -->

```

Appendix A. Code

```
<!-- RWS state publisher (i.e. general states about the robot controller)
-->
<include file="$(find abb_rws_state_publisher)/launch/rws_state_publisher.
launch">
  <arg name="robot_ip" value="$(arg robot_ip)"/>
</include>

<!-- RWS service provider (i.a. starting/stopping the robot controller's
RAPID execution) -->
<include file="$(find abb_rws_service_provider)/launch/rws_service_provider.
launch">
  <arg name="robot_ip" value="$(arg robot_ip)"/>
</include>

<!-- ===== -->
<!-- Externally Guided Motion (EGM) related components. -->
<!-- ===== -->

<!-- EGM hardware interface (i.e. 'ros_control'-based interface for
interacting with mechanical units) -->
<include file="$(find abb_egm_hardware_interface)/launch
/egm_hardware_interface.launch">
  <arg name="base_config_file" value="$(find
abb_robot_bringup_examples)/config/ex2_hardware_base.yaml"/>
  <arg name="egm_config_file" value="$(find abb_irb2400_moveit_config)
/config/joint_limits.yaml"/>
</include>

<!-- Put 'ros_control' components in the "egm" namespace (to match the
hardware interface) -->
<group ns="egm">
  <!-- Load configurations for 'ros_control' controllers on the parameter
server -->
  <rosparam file="$(find abb_robot_bringup_examples)/config/ex2_controllers
.yaml" command="load"/>

  <!-- Two 'ros_control' controller spawners (stopped for the controller that
command motions) -->
  <node pkg="controller_manager" type="spawner" name="started" args=
"egm_state_controller joint_state_controller"/>
  <node pkg="controller_manager" type="spawner" name="stopped" args=
"--stopped joint_group_velocity_controller"/>
</group>
</launch>
```

A.6 Sensors implementation with ROS₂

C₊₊ Code - Distance and Linear Sensor

```
using UnityEngine;
using UnityEngine.UI;

namespace RosSharp.RosBridgeClient
{
  public class TextSubscriber : UnitySubscriber<MessageTypes.Sensor.
    Range>
  {
    private double distance;
    public GameObject changingText2;
    public GameObject cylinder;
    private double ltemp;
    private bool messageReceived;

    protected override void Start()
    {
      base.Start();
    }

    private void Update()
```

```

    {
        changingText2.GetComponent<Text>().text = "Distance " + distance;

        if (messageReceived==true)
        {
            messageReceived = false;
            ltemp = (20 * distance) / 4;
            cylinder.transform.localScale = new Vector3(1f, 1f, (float)
                ltemp);
        }
    }

    protected override void ReceiveMessage(MessageTypes.Sensor.Range
        message)
    {
        messageReceived = true;
        distance = message.range;
        distance = System.Math.Round(distance,2);
    }
}
}
}

```

C[‡] Code - Force Sensor

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.Threading;

namespace RosSharp.RosBridgeClient
{
    public class ForceSubscriber : UnitySubscriber<MessageTypes.Geometry.
        WrenchStamped>
    {
        private double forceZ;
        private double torqueZ;
        public GameObject changingText;
        public GameObject cylinder;
        private bool messageReceived;
        public GameObject spincylinder;

        protected override void Start()
        {
            base.Start();
        }

        private void Update()
        {
            if (messageReceived == true)
            {
                messageReceived = false;
                if (forceZ < -0.5) {
                    cylinder.transform.localScale = new Vector3(1f, 1f,(float)
                        (forceZ*(-40)));
                }
                if (torqueZ < 0.25)
                {
                    spincylinder.transform.Rotate(0.0f, 45.0f, 0.0f, Space.
                        Self);
                }
                else if (torqueZ > 0.45)
                {
                    spincylinder.transform.Rotate(0.0f, -45.0f, 0.0f, Space.
                        Self);
                }
                else
                {
                    spincylinder.transform.Rotate(0.0f, 0.0f, 0.0f, Space.
                        Self);
                }
            }
        }
    }
}

```

```

    }

    protected override void ReceiveMessage(MessageTypes.Geometry.
        WrenchStamped message)
    {
        messageReceived = true;
        forceZ = message.wrench.force.z;
        torqueZ = message.wrench.torque.z;
    }
}
}
}

```

C# Code - Example of Publishing joint positions to the robot

```

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

namespace RosSharp.RosBridgeClient
{
    public class Warning : UnityPublisher<MessageTypes.Std.Float64MultiArray>
    {
        private MessageTypes.Std.Float64MultiArray message;
        public GameObject cylinder;
        public GameObject spincylinder;
        public GameObject changingText;
        public double[] datarray = new double[6];
        public double y_value = 0;
        public double y_value_old = 0;
        public double count = 0;

        protected override void Start()
        {
            base.Start();
            InitializeMessage();
        }

        private void Update()
        {
            y_value = spincylinder.transform.eulerAngles.y;
            if (y_value - y_value_old > 40)
            {
                count = 0;
                y_value_old = y_value;
                changingText.GetComponent<Text>().text = "WARNING";
                Debug.Log("Right");
                datarray[0] = 0;
                datarray[1] = 0;
                datarray[2] = 0;
                datarray[3] = -0.7;
                datarray[4] = 0;
                datarray[5] = 0;
                message.data = datarray;
                Publish(message);
            }
            else if (y_value - y_value_old < -40)
            {
                count = 0;
                y_value_old = y_value;
                changingText.GetComponent<Text>().text = "WARNING";
                Debug.Log("Left");
                datarray[0] = 0;
                datarray[1] = 0;
                datarray[2] = 0;
                datarray[3] = 0.7;
                datarray[4] = 0;
                datarray[5] = 0;
                //Debug.Log(datarray[4]);
                message.data = datarray;
                //Debug.Log(message.data[4]);
                Publish(message);
            }
        }
    }
}

```



```

else if (cylinder.transform.localScale.z > 700)
{
    changingText.GetComponent<Text>().text = "WARNING";
    Debug.Log("Out");
    datarray[0] = 0;
    datarray[1] = -0.8;
    datarray[2] = 0;
    datarray[3] = 0;
    datarray[4] = 0;
    datarray[5] = 0;
    message.data = datarray;
    Publish(message);
}

else
{
    count += 1;
    if (count > 100)
    {
        count = 0;
        changingText.GetComponent<Text>().text = "";
        datarray[0] = 0;
        datarray[1] = 0;
        datarray[2] = 0;
        datarray[3] = 0;
        datarray[4] = 0;
        datarray[5] = 0;
        message.data = datarray;
        Publish(message);
    }
}

private void InitializeMessage()
{
    message = new MessageTypes.Std.Float64MultiArray
    {
        data = new double[6],
    };
    y_value = spincylinder.transform.eulerAngles.y;
    y_value = y_value_old;
}
}
}

```

A.7 ROS Launch File - Gazebo Integration

```

<launch>
<arg name="robot_name" default="/"> <!-- Change this for namespacing. -->
<!--
<arg name="rviz" default="true"/> <!-- Set to true to run rviz in parallel. -->
<arg name="lite" default="false" /> <!-- Set to true if you're using CHAMP lite version. Only useful for microcontrollers. -->
<arg name="ros_control_file" default="$(find spot_config)/config/ros_control/ros_control.yaml" />
<!-- Path to ROS Control configurations. Do not touch. -->
<arg name="gazebo_world" default="$(find spot_config)/worlds/outdoor.world" /> <!-- Path to Gazebo world you want to load. -->
<arg name="gui" default="true"/>
<arg name="world_init_x" default="0.0" /> <!-- X Initial position of the robot in Gazebo World -->
<arg name="world_init_y" default="0.0" /> <!-- Y Initial position of the robot in Gazebo World -->
<arg name="world_init_heading" default="0.0" /> <!-- Initial heading of the robot in Gazebo World -->

<param name="use_sim_time" value="true" />

<include file="$(find spot_config)/launch/bringup.launch">

```

Appendix A. Code

```
<arg name="robot_name" value="$(arg robot_name)"/>
<arg name="gazebo" value="true"/>
<arg name="lite" value="$(arg lite)"/>
<arg name="rviz" value="$(arg rviz)"/>
<arg name="joint_controller_topic" value=
  "joint_group_position_controller/command"/>
<arg name="hardware_connected" value="false"/>
<arg name="publish_foot_contacts" value="false"/>
<arg name="close_loop_odom" value="true"/>
</include>

<include file="$(find champ_gazebo)/launch/gazebo.launch">
  <arg name="robot_name" value="$(arg robot_name)"/>
  <arg name="lite" value="$(arg lite)"/>
  <arg name="ros_control_file" value="$(arg ros_control_file)"/>
  <arg name="gazebo_world" value="$(arg gazebo_world)"/>
  <arg name="world_init_x" value="$(arg world_init_x)"/>
  <arg name="world_init_y" value="$(arg world_init_y)"/>
  <arg name="world_init_heading" value="$(arg world_init_heading)"/>
  <arg name="gui" value="$(arg gui)"/>
</include>

<include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch">
  <arg name="port" value="9090"/>
</include>

<node name="file_server" pkg="file_server" type="file_server" output=
  "screen"/>
  <!-- Launch the tracking node -->
  <node pkg="visp_auto_tracker" type="visp_auto_tracker" name=
    "visp_auto_tracker" output="screen">
    <param name="model_path" value="$(find visp_auto_tracker)/models" />
    <param name="model_name" value="pattern" />
    <param name="debug_display" value="True" />
    <param name="detector-type" value="dmtx" />
    <param name="tracker-type" value="1" />

    <remap from="/visp_auto_tracker/camera_info" to="/camera/rgb/
      camera_info"/>
    <remap from="/visp_auto_tracker/image_raw" to="/camera/rgb/
      image_raw"/>
  </node>
</launch>
```

Lund University Department of Automatic Control Box 118 SE-221 00 Lund Sweden		<i>Document name</i> MASTER'S THESIS
		<i>Date of issue</i> January 2022
		<i>Document Number</i> TFRT-6154
<i>Author(s)</i> Pablo Fernández Fernández		<i>Supervisor</i> Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden Tore Hägglund, Dept. of Automatic Control, Lund University, Sweden (examiner)
<i>Title and subtitle</i> Construction Supervision with Augmented Reality		
<i>Abstract</i> <p>Construction sites are key points regarding implementing robot supervision to create more secure, reliable, and free hand-work workspaces. This project's scope is to develop a complete interpretation of a modern concept of surveillance. For this case, the test scenario is the industrial lab at the Faculty of Engineering, Lund University, known as RobotLab. For its development, an Augmented Reality scenario is recreated in which the work done by the ABB IRB2400 robot is studied at the lab. The use of Unity and ROS as special software programs and systems can generate this concept in a manner that can be easily reproduced in other scenes.</p> <p>In order to give a more modern concept, the tool for taking care of the vigilance is a robot, or in this case, a simulation of the robot Spot from Boston Dynamics. The reason behind this is to produce a more automatic project so that the person in charge can operate this robot and use its camera for the creation of the Augmented Reality. In general terms, the development of this project is carried out by the use of a large number of separate projects from different developers and research companies. In conclusion, the idea is to provide a software package to be applied and integrated into other constructions scenarios for the recreation of Augmented Reality scenarios.</p>		
<i>Keywords</i>		
<i>Classification system and/or index terms (if any)</i>		
<i>Supplementary bibliographical information</i>		
<i>ISSN and key title</i> 0280-5316		<i>ISBN</i>
<i>Language</i> English	<i>Number of pages</i> 1-82	<i>Recipient's notes</i>
<i>Security classification</i>		

<http://www.control.lth.se/publications/>