HoloLens Robot Controller - Final Report

Arda Düzceker
ETH Zurich
ardad@student.ethz.ch

Jonas Hein
ETH Zurich
heinj@student.ethz.ch

Sophokles Ktistakis
ETH Zurich
ktistaks@student.ethz.ch

Rodolfo Octavio Siller Quintanilla
ETH Zurich
srodolfo@student.ethz.ch

Abstract

The goal of this project was to build an interaction system that would enable a user to control a gardening robot called 'Trimbot' from the Microsoft HoloLens and to display the Trimbot's generated SLAM point cloud as a hologram to the user wearing the HoloLens.

1. Introduction

The objective of this project is to develop an interface for the HoloLens to control the Trimbot and to display the SLAM point cloud generated by the robot as a hologram to the user in the HoloLens. In order to achieve this objective we have to first implement a communication protocol so that the Trimbot and the HoloLens can send information to one another. Then, we have to align the point cloud generated by the robot with the point cloud generated by the HoloLens in order to display the robot's point cloud at the correct places from the viewpoint of the HoloLens. Alignment is necessary also because, we have to combine the coordinate system information gathered from both devices in a common frame to send user-friendly control signals. To solve this alignment problem we use iterative-closest-point algorithm. However, the success rate of the algorithm highly depends on the initial placement of the point clouds. To make the initial placement as close as possible, so-called prior, we place ArUco markers on the robot and integrate/implement a pose estimation algorithm. Thus, we manage to link the independent coordinate system of the Trimbot and the HoloLens while the ArUco markers work as the bridge in our setup.

The overall processing pipeline is depicted in Figure 1. The user has the ability to control the Trimbot directly from the HoloLens. Since the robot does not know its environment at startup, the user initially controls the Trimbot via a holographic joystick and send out direct movement instructions. After the robot starts to construct a map of it's environment, the user starts to see the point cloud of the robot as hologram (green dots in the Figure2) and gains the ability to control the robot by pointing at a location in real world, in other words placing a holographic flag (Figure 14). Trimbot receives the real world location and uses it's already implemented path planning algorithm to drive autonomously to the pointed out target location.

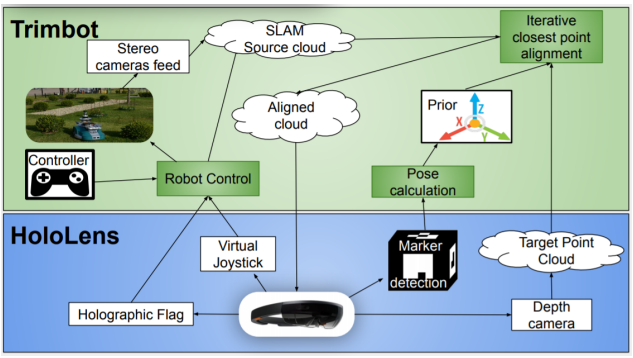

Figure 1: Overview of the processing pipeline

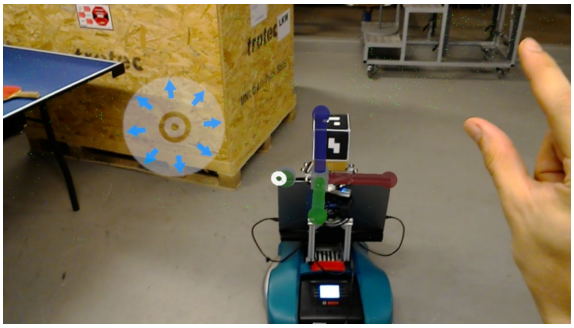

Figure 2: Displaying the holographic joystick and the Trimbot's point cloud to the user

2. Communication

The first task in this project was to establish the communication link between a ROS node running on the Trimbot and a HoloLens app created in Unity. We decided to use standard ROS messages for all communication for a seamless integration in the existing Trimbot software architecture as well as maximum compatibility. The HoloLens app uses the ROS library [1] to communicate with the ROS environment via rosbridge [2], which runs on the ROS master. Here, rosbridge handles the conversion between ROS messages and JSON. Consequently, HoloLens can easily communicate with the Trimbot using JSON streams constructed by ROS. Since the HoloLens runs on the Universal Windows Platform (UWP), we used a customized version of ROS [5], which was built to use correct type of sockets that are compatible with UWP applications.

In order for this communication to work efficiently and robust, we implemented some key technical aspects:
For the communication on the HoloLens, we ran our publishers such as pose, point cloud and images as asynchronous tasks on separate threads. Here, we used the main thread to perform all sensor gathering and calculations on the raw data and 'fire-and-forget-threads' for the communication to publish any processed information. In order to gain control over the publishing frequencies using the 'fire-and-forget-threads', we had to omit Unity's fixed frequency rendering intervals and replace them with a Timer class, that can be accessed by all publishers and measures elapsed time in milliseconds. By being able to control the amount of processing being done in Unity we can decrease it to a degree where the computations run in a stable manner. Also, in line with our goal of reducing processing loads we added the option to control the size of published point clouds, where we had to strike a balance between computational efforts and quality of the point cloud alignment, as sparser point clouds are more difficult to align.

3. Point Cloud Alignment

In order to align the HoloLens and Trimbot maps, we need to calculate a transformation matrix which will rotate and translate one of the coordinate frames and align it with the other. Because the maps differ in the amount and location of the points, we need also a way of measuring the quality of the resulting transformation. The Iterative Closest Point (ICP) algorithm takes two point clouds, a source and a target, and attempts to match them by iteratively computing the closest distance between them. On every iteration, a transformation is calculated which moves the source cloud closer to the target cloud (See figure 3).

The distance computation to the closest point only considers the ones which are below the range of a maximum correspondence limit. After each iteration, the following 3 criteria are evaluated to determine if it has converged:

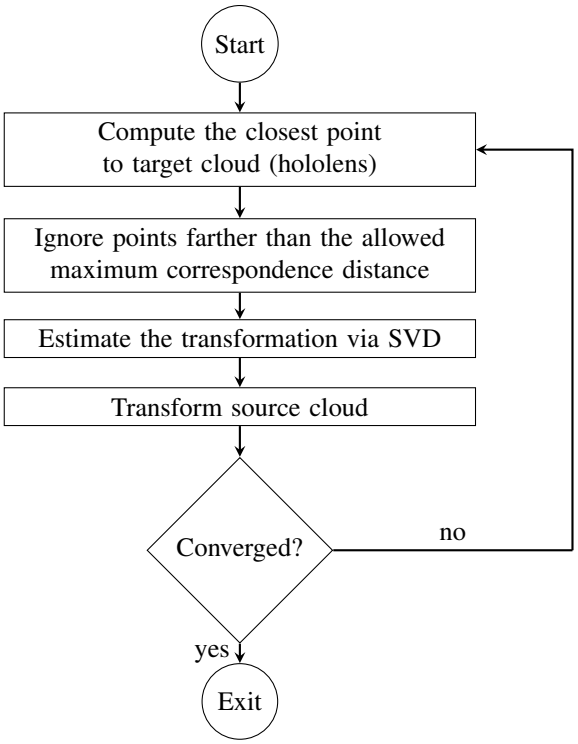

Figure 3: Iterative closest point flow chart

1. The number of iterations has reached the maximum number of iterations set by the user.
2. The difference between the previous transformation and the current estimated transformation is smaller than the value defined by the user.
3. The sum of the Euclidean squared errors is smaller than the value set by the user.

If any of the previous criteria is met, the ICP outputs the resulting transformation which best aligns the source to the target point cloud. At the same time, a fitness score, which is the sum of squared distances from the source to the target point cloud, is also calculated and will help determine the quality of the resulting transformation. Since our goal is to display the Trimbot's map in the HoloLens, the Trimbot's point cloud is treated as the source and the HoloLens point cloud is treated as the target. This will give us a transformation from the Trimbot to the HoloLens coordinate frame.

3.1. Point Cloud Alignment with Synthetic Data

Before having real point clouds from either the hololens or the trimbot, we tested the limitations of the ICP with a synthetic outdoor scene point cloud. The tests took the original synthetic point cloud of about 50k points (target point

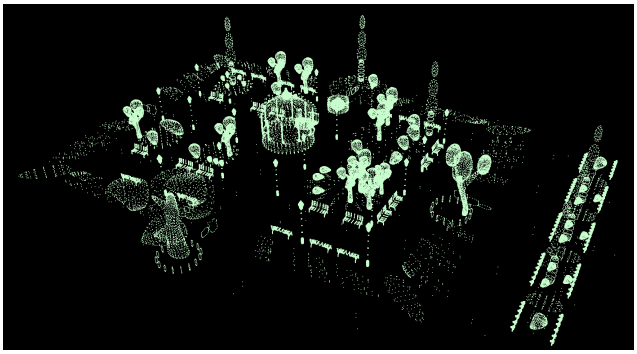

Figure 4: Original synthetic point cloud

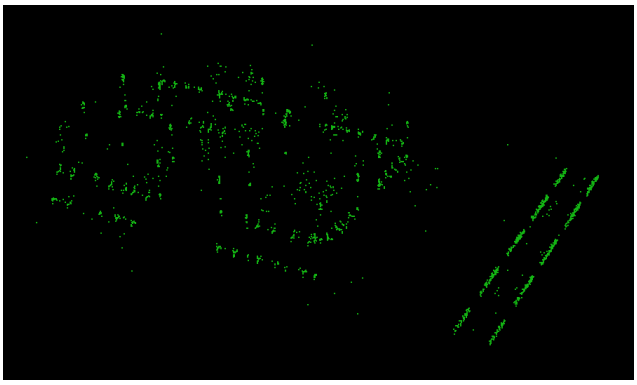

Figure 5: Sparse point cloud with 5% of the points in the target point cloud

cloud, see figure 4), took a scaled copy (source), applied different transformations to it and ran the ICP to measure the resulting computation times, translation and rotation errors. The following tests were carried out:

1. Source cloud with different scaling levels 90% - 99% of the original point cloud. This would give us the maximum point density the ICP was able to take from the trimbot and still yield the desired rate (see Figure 5).

2. Different noise levels added to each point in the source cloud. This would give us the behavior of the worst case scenario if the point clouds were too different and even when taken from the same scene.The maximum distance was 1m offset in all three X-Y-Z axes (see Figure 6).

3. Different translations of the source cloud in X-Y-Z and different starting rotations of the Z axis. This would give us a sense of how inaccurate our prior transformation could be without compromising the alignment (see Figure 7).

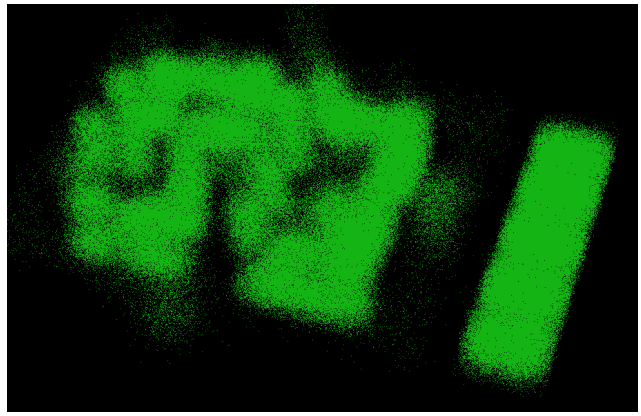

Figure 6: Adding $\pm$1m noise to the source image

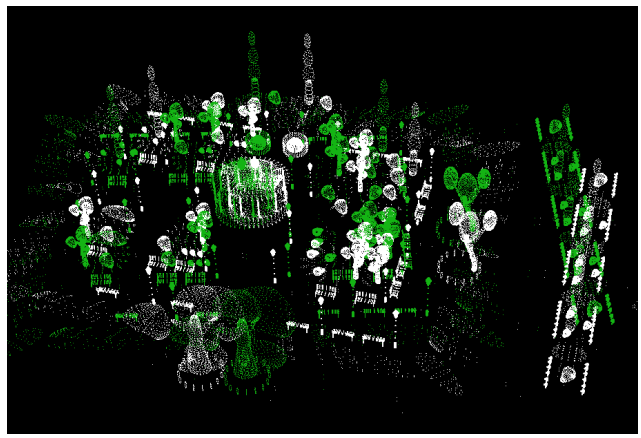

Figure 7: Original image in white, and a slightly rotated and translated transformation of the source in green

After testing with different parameter combinations and visually confirming the alignment of the point clouds, we concluded that the most critical factors to have a fast and effective alignment were the prior rotation and the sparsity level of the source cloud. (see Figures 8 and 9)

3.2. Point Cloud Alignment Implementation

The ICP alignment runs in our ROS node where the Trimbot and the HoloLens maps are received, updated, evaluated and aligned. If the alignment was successful, the aligned map, the transformation and the fitness score are updated. Because we don't know exactly which features each system is mapping, we could have a good alignment but a large fitness score on every new update. By saving the previous fitness score after a successful alignment, it enables the program to compare it with the current score and determine whether to use the resulting transformation or discard it and keep the previous one. Finally, the aligned map is published.

Similar to the Unity publishing threads, the alignment

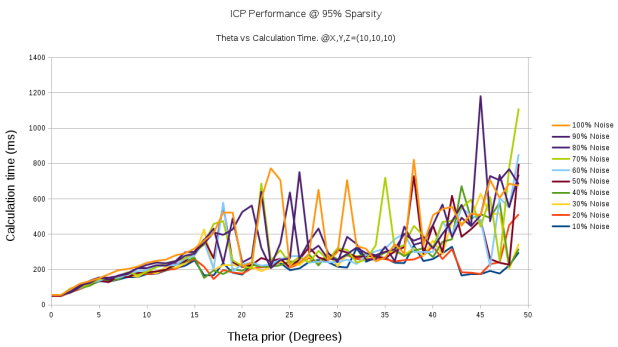

Figure 8: Computation times for different noise levels and rotation angles

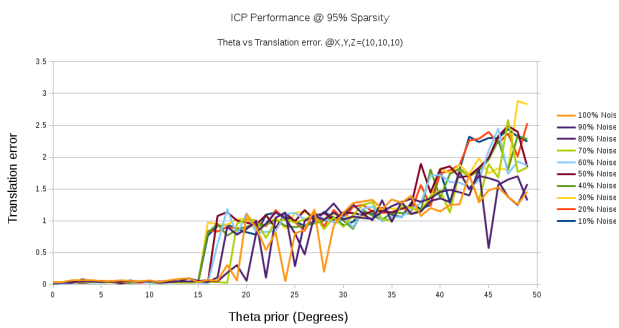

Figure 9: Translation error: sum of squared errors between prior and resulting transformation, with different noise levels.

runs in a separate thread for performance reasons.

4. Robot Pose Estimation by HoloLens

4.1. Camera Calibration

In order to have a functioning marker detection and pose estimation, we first need to calibrate the HoloLens camera by estimating it's mapping of recognized points in the real world to their location in the image plane. For this we used the camera calibration procedure by Zhang [6] as implemented by the OpenCV library and obtained a robust calibration of the camera parameters and distortion coefficients after taking 50 images from a checkerboard in different rotations. An overview of the procedure by Zhang [6]:

The camera calibration parameters are then passed to the marker detection and pose estimation scripts.

4.2. Marker Detection and Pose Estimation

Aligning the HoloLens and robot point clouds requires an estimation of the Trimbot's pose from the HoloLense's point of view, which we implemented with two different solutions. Both solutions estimate the robot pose by detecting

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Figure 10: Homography between known points in the real world and the image plane to obtain camera parameters

$$\breve{x} = x + x[k_1(x^2+y^2) + k_2(x^2+y^2)^2]$$
$$\breve{y} = y + y[k_1(x^2+y^2) + k_2(x^2+y^2)^2]$$
$$\breve{u} = u + (u-u_0)[k_1(x^2+y^2) + k_2(x^2+y^2)^2]$$
$$\breve{v} = v + (v-v_0)[k_1(x^2+y^2) + k_2(x^2+y^2)^2]$$

Figure 11: Pinhole model based equations to obtain distortion parameters

markers that are printed on the sides of a cube placed on top of the Trimbot and inferring the robot's position and orientation by estimating the three-dimensional translation from the detected marker back to the camera's coordinate system.

Our first solution ran the pose estimation directly on the HoloLens as we reasoned that sending images over the communication protocol might put it under too much strain. Here, we used the ARToolkit version by Long Qian[3], which wraps the original C/C++ code base into C# so that it can be run on Unity. The toolkit tracks the poses of 3x3 bit markers with a processing rate of two FPS, where again we had to consider the trade-off between runtime performance of our application and the quality of the point cloud alignment. Since the robot pose only served as a prior to the ICP algorithm, which performed the iterative alignment even with imprecise pose estimations, we felt that the two FPS processing rate served a good balance between performance and quality of alignment. As a final step we had to convert the detected marker poses from Unity's left-handed coordinate system to ROS's right-handed format so that it could be processed by our point cloud alignment.

One of the main drawbacks of the ARToolkit implementation was that we lost control over the specifics of the implementation and thereby over the quality of the marker detection itself, which prompted us to write our own custom maker pose estimation based on OpenCV. We added this as an option to have a more flexible estimation that was easier to customize according to the user's needs. The script, which runs on the Trimbot's ROS system subscribes to a HoloLens image publisher and performs the marker pose estimation in three steps:

1. Candidate Detection: First, we analyze the image for

potential marker candidates using thresholding, contour extraction - and selection. Here, thresholding is performed on multiple window sizes, which have to be adapted to the size of the printed markers and the typical distance from which those markers are viewed from, since windows that are too small or too large can fail to recognize borders between the marker bits. Here, window sizes from 3 to 24 pixels for the marker contour detection worked well for us.

2. Multi-Marker-Identification: Next, a perspective transformation is applied to all marker candidates and after applying Otsu thresholding black and white bits are extracted and markers identified.

3. Pose Estimation: We then estimate the transformation matrix from the marker to the camera coordinate system and use our knowledge of the markers in relation to the robot to infer the pose of the Trimbot.

5. Transformations

For visualizing the Trimbot point cloud as a hologram in the HoloLens, it is necessary to calculate the correct coordinate transformation matrix between the two coordinate frames and update it according to the movements of both devices. Since the Trimbot software architecture already manages its coordinate frames in form of a transformation tree in the ROS TF package [4], we decided to add all necessary coordinate frame transformation to this tree as well.

The transformation tree is visualized in Figure 12. The transformations within the Trimbot's system range from *colmap*, which is the origin of the robot's SLAM system, to *base_footprint*, which is defined as the center of the robot's rear wheel axis. The *base_marker* coordinate frame, which is defined as the centroid of the ArUco markers that were measured by us to determine their transformation matrix. Finally, there is the *hololens_world* frame, which is the fixed coordinate frame in which the HoloLens maps out its environment.

We are mainly concerned with the following transformations:

- $T_{13} = T_3 * T_2 * T_1$, the transformation from the Trimbot map coordinate frame to the *base_footprint* frame
- $T_{15} = T_5 * T_4 * ... * T_1$, the posterior point cloud transformation from Trimbot map frame to HoloLens map frame
- T_5, the transformation from current marker pose frame *base_marker* to HoloLens map frame

Whenever the ArUco marker is detected, we update the corresponding transformation T_5, which is simply the inverse of the estimated pose of the ArUco marker in the HoloLens world frame. As the Trimbot and HoloLens can move independently while the transformation from one system to the other via marker detection T_5 is only updated in fixed time intervals or might even fail altogether if we lose visual contact, the overall transformation T_{15} between the two map coordinate frames becomes invalid as soon as one of the devices changes its pose. To avoid invalid transformations, we only calculate the prior transformation estimate T_{15} immediately after the ArUco marker was detected. As outlined previously, the calculated prior transformation estimate is then given to ICP in order to get an optimized posterior transformation estimate $\hat{T}_{15}$. We want to emphasize the fact that the transformation tree (or rather all transformations based on T_5) are not continuously updated and thus can contain arbitrary large errors.

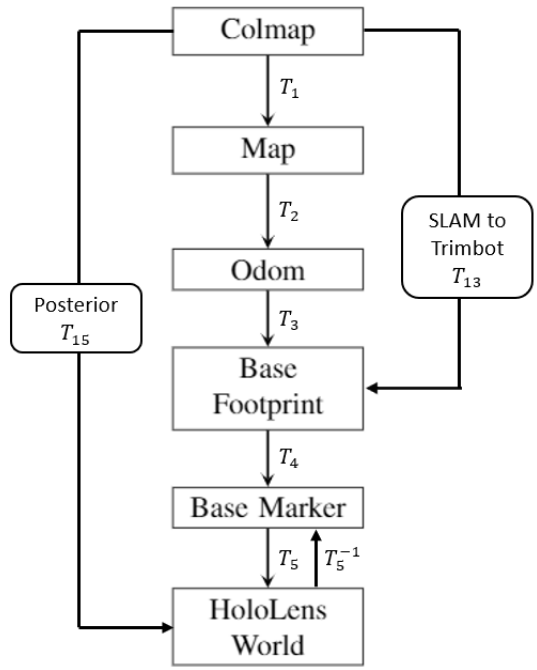

Figure 12: The transformation tree

In order to transform incoming target locations, we keep the last posterior transformation $\hat{T}_{15}$ in memory. As soon as we receive a goal from the HoloLens, we multiply it with the inverse posterior transformation $\hat{T}_{15}^{-1}$ to get the goal from the HoloLens world frame into the Colmap frame, from where it can be accessed by the Trimbot's path finding system.

6. HoloLens User Experience

6.1. Point Cloud Rendering

To display the aligned point cloud to the user, the Trimbot publishes its point cloud which is received by a subscriber in the HoloLens app, which utilizes callback methods to efficiently handle these type of messages in a multi-threaded fashion. In the thread that receives the raw message, it is converted into a renderable Unity vertices structure, packaged and pushed to the internal thread-safe queue. These ready-to-render point clouds are then, dequeued by Unity Engine's main rendering thread, which updates the point cloud object mesh in the scene and render it's vertices. The motivation behind using such multi-threaded system (throughout the whole communication pipeline of the HoloLens) is to reduce the workload on the rendering thread to provide smoother experience to the user. Some key implementation aspects are that we throttle the time between received point clouds to 0.2ms to ensure an acceptable processing load. Also, we set a threshold to process only point clouds that are larger than 500 points to ensure that the user has a visual feedback on when the robot has mapped out it's environment well enough to handle autonomous navigation through click commands on a target location.

Figure 13: Trimbot's point cloud rendered in HoloLens

6.2. Robot Control

The user can control the Trimbot from the HoloLens by either issuing navigation commands to a target location, which the robot will execute autonomously or by issuing low-level control signals via joystick.

In the beginning when the Trimbot has not yet mapped out it's environment and can't execute autonomous navigation commands, the user has to control the robot via holographic joystick commands so that it can further explore it's environment. Every button on the joystick is assigned to a predetermined combination control commands which are expressed in terms of linear (for forward and backward motion) and angular velocity (for clockwise and counter-clockwise turning motion) per second. Such low-level messages are pretty standard messages that are sent to "cmd-vel/" topic in ROS.

Whereas, after Trimbot maps its environment well enough, user can send high-level control commands by simply clicking at a real world location. If it is a reachable location for the robot, the robot will autonomously navigate to the target location following it's already present path planning system. Internally, such click inputs trigger click events in the app. By performing several checks on the triggered event information, we ensure that only the valid scene locations (not on top of another hologram like joystick, inside areas that have already been discovered by the HoloLens) are processed. Then, similarly we publish the target location via utilizing our multi-threaded implementation. After clicking at a location, the user receives a visual feedback on a valid click as we place a visible bright blue flag hologram on the click's location.

Figure 14: Displaying the holographic target flag

7. Results and Discussion

The results can be best demonstrated through the video that we supply with the report. On the top left corner of the video, an overlaid point cloud scene can be seen, which is not present in the HoloLens display. This is only an aid to visualize what is happening at the coordinate frame level. The user can control the Trimbot by sending direct control commands with the holographic joystick. After the Trimbots point cloud was successfully transformed to the Hololens coordinate frame, it is displayed as green points in the Hololens display. From this point on, the goal flag can now be pinpointed to a location in the room to command the Trimbot to move there.

8. Work distribution

The work distribution for our project was as follows:
Constructing the communication interface between Trimbot and HoloLens: Arda and Jonas
Design and coding of the HoloLens app: Arda

Design and coding of the ROS node: Jonas
Point cloud alignment: Octavio
Coordinate transformations: Jonas
Marker detection and robot pose tracking: Sophokles, Arda and Jonas
Poster, Video and Report: Sophokles and Octavio (partially Arda and Jonas)

References

[1] Dr. Martin Bischoff. *ROS-Sharp*. URL: https://github.com/siemens/ros-sharp.

[2] Jonathan Mace. *rosbridge suite*. URL: https://wiki.ros.org/rosbridge_suite.

[3] Long Qian. *HoloLens-ARToolkit*. URL: www.github.com/qian256/HoloLensARToolKit.

[4] ROS. *ROS TF Package*. URL: http://wiki.ros.org/tf.

[5] David Whitney. *ROS-Sharp*. URL: https://github.com/dwhit/ros-sharp.

[6] Zhengyou Zhang. "A Flexible New Technique for Camera Calibration". In: (1998). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf.