VR-based visual model mediated telepresence using a SLAM generated virtual model

B. (Bart) Lammers

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. D. Dresscher
dr. ing. G. Englebienne

August 2020

031RaM2020
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

SUMMARY

The field of telerobotics is concerned with remote-controlled robots. These robots are sent to a remote location to execute a task. Human expertise is still needed since these tasks are, in many cases, complex and require dexterous manipulation and human cognition. Besides controlling the robot according to the movement of the user and providing haptic feedback in return, visual cues are essential for effective control by the operator.

A simple visual setup only requires a Head-Mounted Display (HMD) and a camera on the remotely controlled robot. A perfect implementation of such a setup would not encounter any delay. However, latency will have an increasing impact if the robot is at a significant distance.

The goal of this research is to design a SLAM-based VR environment that is able to decrease the negative effects of time delays on the operator in a telerobotic setup. To achieve the aforementioned goal, a setup was developed which combines an HMD and a moveable 3D vision sensor. The VR environment for the user is the SLAM map and is constantly updated with new information from the vision sensor by the SLAM algorithm. The user can look around freely in the VR environment without being constrained by the (delayed) movement of the 3D vision sensor itself.

The performance of the aforementioned concept was evaluated in this research. The performance requirements of the system were determined, and shortcomings were identified by using a root cause analysis. Functional tests were performed to improve upon these shortcomings where possible. It is shown that dynamic object tracking is still a severe problem when using a commonly available visual SLAM-based approach (RTAB-map). Segmentation of moveable objects is required in future implementations to generate a virtual model that can cope with dynamic environments.

This thesis is structured in two parts: a paper and additional appendices. The paper gives an overview of the research work and the appendices provide more details about the implementation and additional test results.

Contents

VR-based visual model mediated telepresence using a SLAM generated virtual model

Bart Lammers, Douwe Dresscher, Gwenn Englebienne and Jan Broenink

***Abstract*—Visual awareness has a significant impact on the effectiveness of control for operators in teleoperation tasks. The awareness can degrade when time delays are introduced into the system. In this paper, a new concept is presented that introduces a visual model-mediated approach that can be used in telemanipulation. A dense visual SLAM generated virtual model is created which is asynchronously updated with new information from a moveable RGB-D sensor. The operator is able to look around freely in the generated virtual reality environment. The performance of the aforementioned concept was evaluated in this paper. The performance requirements of the system were determined, and shortcomings were identified by using a root cause analysis. Functional tests were performed to improve upon these shortcomings where possible. It is shown that dynamic object tracking is still a severe problem when using a commonly available visual SLAM-based approach (RTAB-map). Segmentation of moveable objects is required in future implementations to generate a virtual model that can cope with dynamic environments.**

I. Introduction

The field of telerobotics is concerned with remote-controlled robots. These robots are sent to a remote location to execute a task. Human expertise is still needed since these tasks are, in many cases, complex and require dexterous manipulation and human cognition. Telepresence and teleoperations are used to allow the operator to control the robot with remote presence.

Besides controlling the robot according to the movement of the user and providing haptic feedback in return, visual cues are essential for effective control by the operator. Vision systems play an important role in telemanipulation as they can convey an enormous amount of information regarding the remote environment to the operator [1]. A simple visual setup only requires a Head-Mounted Display (HMD) and a camera on the remotely controlled robot. A perfect implementation of such a setup would not encounter any delay. However, latency will have an increasing impact if the robot is at a significant distance.

Time delay in a visual setup causes various issues. Latency in VR is known to cause VR-sickness, with nausea, disorientation, headaches, sweating, and eye strain being the most common discomforts amongst users [2]. The amount of latency that users can handle before they experience VR-sickness varies, but as a rule of thumb, a maximum delay of 20 ms should be taken as a threshold to prevent it [3].

The goal of this research is to design a SLAM-based VR environment that is able to decrease the negative effects of time delays on the operator in a telerobotic setup. To achieve the aforementioned goal, a setup will be developed which combines an HMD and a moveable 3D vision sensor. The VR environment for the user is the SLAM map and is constantly updated with new information from the vision sensor by the SLAM algorithm. The user can look around freely in the VR environment without being constrained by the (delayed) movement of the 3D vision sensor itself.

Previously, SLAM has already been proven to work in a network setup [4]. However, if there are wireless network packet losses present, a SLAM algorithm fails to produce a valid map or even will stop working [4].

In our approach, we asynchronously update the VR environment of the user. The updates are performed based on the output of the SLAM-generated map with a relatively low update frequency. The rendering of the environment occurs at a high update frequency and, therefore, time-delay related issues are expected to have a reduced impact on the user.

This paper is organized as follows; In section II related work on SLAM and SLAM generated virtual models is discussed. Section III elaborates on the experimental setup that was implemented. Next, in section IV the performance of the setup is evaluated. Section V and VI conclude with a discussion and conclusion on this evaluation.

II. Related work

SLAM has been proven to be an efficient algorithm for various goals. These include creating maps of unknown environments, VR implementations and AR-based enhancements [5]. Various implementations of SLAM algorithms are distributed as part of ROS and are commonly used. [6]–[8].

State of the art candidates are the ORB-SLAM2 and RTAB-map algorithms. These algorithms provide an efficient solution for RGB-D type cameras, which can handle high-density point clouds by using efficient map memory management [9], [10]. In the RTAB-map algorithm, for example, a distinction is made between working memory and long term memory which makes it computationally efficient [11]. Da Silva et al. state that RTAB-map is the preferred solution when using 3D stereo SLAM, however, it requires more processing power compared to ORB-SLAM2 [6]. Efficient real-time solutions for SLAM are made possible by [9] and [10] which can be applied in the approach proposed in this work.

While the algorithms stated above provide satisfactory solutions, there are still problems to be solved in SLAM. Dynamical environments are known to cause issues since elements in the generated map of the environment might change. Various solutions have been proposed to deal with environmental changes by updating keyframes [12]–[14]. However, these are all monocular approaches and are not fit for stereo-vision

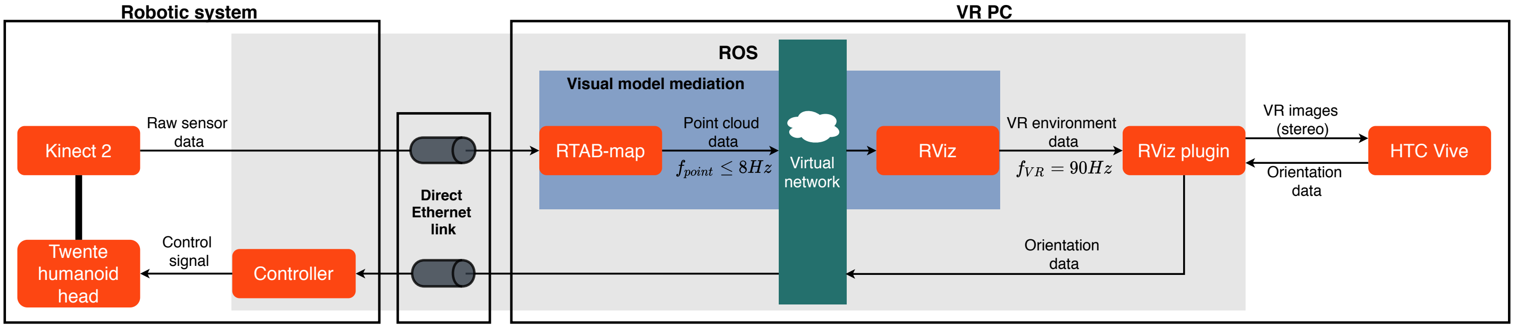


Fig. 1: Schematic overview of the experimental setup.

based solutions. Besides dynamic changes, making reconstructions of outdoor environments can still pose challenges as well due to the lighting conditions [15].

The proposed method of this paper can be considered as a visual model-mediated approach since the user's (delayed) visual perception of the remote environment is achieved through a simulated VR environment. The approach can be considered useful since it eases the real-time constraint of the movement control of the remote robot. That is, the movement of the camera on the remote robot does not need to match the movement of the user exactly due to the model-mediated approach. In literature, similar visual model-mediated approaches are found. Codd-Downey et al. provided a framework based on ROS and Unity to create a simulated SLAM-based VR environment in which the user can look around freely with an HMD [16]. However, the real-time use of their framework was not shown. Rodehutskors et al. worked on a 3D visualization that fuses point cloud data from a laser scanner and HD camera images [17]. The user can observe the VR environment that is unfortunately only updated at 1Hz by using an Oculus Rift HMD. Kim et al. continued with the ROS+Unity framework of Codd-Downey et al. to create a real-time VR environment based on the Turtlebot and GearVR [18]. However, it is based on Octomap and is, therefore, not a SLAM-based solution and without a loop closure implementation. Besides, the resulting VR environment has a low resolution and does not provide color information.

III. Method

Figure 1 shows a schematic view of the experimental setup that is designed to evaluate the proposed approach. It consists of various software and hardware components to interface RGB-D data from a Microsoft Kinect Sensor V2 (also known as Kinect for Xbox One) and the HTC Vive HMD. To create a moveable platform for the Kinect 2 sensor, the so-called Twente humanoid head platform is used which can be seen in Figure 2 and was designed to execute human-like head movements [19]. The Kinect 2 is mounted on top of the platform by using 3D printed brackets that were specifically designed to eliminate any possible flexibility in the enclosure of the Kinect 2 that would otherwise result in unwanted high-frequency motion. The Kinect 2 sensor has angular and linear velocity components since the mounting location is above the rotation point.

The Twente humanoid head platform is controlled based on the desired set point which is updated at 40Hz. On top of the platform, the Kinect 2 is securely mounted. An Intel NUC with an Intel Core i5-4250u CPU @1.7 GHz and 8 GB DDR3 RAM was used to provide for computing power to run the required drivers for the Kinect 2 sensor and the required controller for the Twente humanoid head platform. An Ethernet connection is used to provide for the data communication between the platform and the VR PC which contains an Intel Core i7-6700HQ CPU @ 2.6 GHz and 16 GB DDR4 RAM.

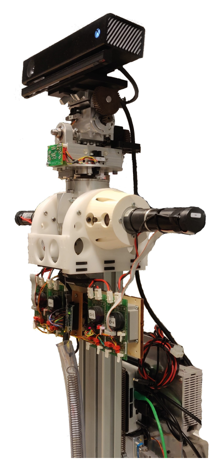


Fig. 2: The Twente humanoid head platform: a moveable platform for the Kinect 2 sensor.

The VR PC is used to run the RTAB-map algorithm on the ROS framework. For computational considerations, only the driver of the Kinect and the controller for the Twente humanoid head were placed on the robotic system. This eliminates the use of point cloud map synchronization between the robotic system and the VR PC since the map is only produced locally on the VR PC. The network is virtually placed before the RViz component and before the orientation data returns to the robotic system. It is possible to introduce manual time delays into the system here.

The RTAB-map algorithm in combination with RViz within ROS can be considered the visual model mediated aspect of the system, where the input is sampled at low frequency and the model of the environment at the output is sampled at high frequency. The output rate (f_{point}) of the RTAB-map algorithm can reach up to 8Hz, while the output rate of RViz (f_{VR}) is rendered to the HTC Vive at 90Hz, regardless of the f_{point} frequency. Thus, the time delay related issues that can occur before the visual model mediation aspect of the system are expected to have a reduced impact on the user's experience.

RTAB-Map is chosen over ORB-SLAM2 since it provides a higher point cloud density and the online output is available through a ROS topic that publishes the point cloud data. The Kinect 2 launch configuration[1] for the RTAB-map algorithm was used with the 'DetectionRate' parameter set to 20 Hz in order to achieve an increased output rate. The RTAB-map algorithm incrementally adds new point cloud data from the Kinect 2 sensor to the map. Also, the algorithm can remove point cloud data from the map. The output of the RTAB-map algorithm is the entire point cloud map and is visualized and updated in RViz in its entirety. RViz is the visualization tool of choice since it has the best support for point clouds and has the best ROS compatibility compared to other tools such as Unreal Engine and Unity3D. From this point, the generated VR environment in RViz is rendered to the HTC Vive as stereo VR images by using a RViz plugin which is designed just for this purpose[2]. Besides rendering, the plugin is also capable of obtaining the orientation data from the HTC Vive which is published to a ROS topic in quaternion format. Some final processing is required to convert the quaternion angles to Euler angles at a rate of 40 Hz which is required by the controller of the Twente humanoid head platform.

IV. Performance evaluation

The performance of the proposed system will be evaluated according to the following structure:

- *Definition of requirements*: identification of functionalities that the system is expected to perform.
- *Identification*: identification of functionalities of the system that perform poorly or correctly through experiments.
- *Root Cause Analysis (RCA)*: enumerate the possible root causes of the identified poorly performing functionalities.
- *Functional testing*: evaluating the above-mentioned root causes by functional testing.

A. Definition of requirements

Below an overview is presented with the baseline performance requirements:

1) Image acquisition from the Kinect 2 and image transport from the robotic system to the VR PC into the ROS framework needs to operate at a sufficient rate such that the RTAB-map node can run effectively. This rate needs to be at least as high as the update rate of the environment in RViz which is required to be 7 Hz (see functionality 3).
2) RTAB-map needs to provide the following two functionalities:
   a) Publish odometry data based on its visual odometry algorithm using the image, depth, and camera info data from the Kinect 2 sensor.
   b) Create a point cloud map of the environment based on the image, depth, and camera info data from the Kinect 2 and the odometry information. The map needs to be able to add new information as well as updating dynamic objects which are captured by the sensors of the Kinect 2.
3) The data from the output topic of RTAB-map needs to be imported into RViz. This needs to happen at a sufficient rate such that the VR environment matches the actual environment as closely as possible. This implies that the dynamic object needs to be able to be tracked in such a way that the movement appears fluent to the user. Busch et al. found that humans have a detection performance of around 7 Hz by studying EEG signals [20]. This finding suggests that humans are processing incoming visual information at 7 frames per second effectively. Therefore, a 7 Hz minimum is taken as the desired threshold for the output topic frequency of RTAB-map.
4) The VR environment from RViz needs to be converted into two images which can be projected onto the left and right eye of the user by using the HTC Vive. To determine which part of the VR environment needs to be displayed, orientation data from the HTC Vive needs to be obtained.
5) The orientation of the Kinect 2 must follow the motion of the HTC Vive. The user will then be able to see new point cloud information from the part of the environment at which he/she is looking after it was incorporated in an update.

B. Identification

From the definition of requirements presented in section IV-A, functionality 2b) performs insufficiently and is explained in further detail below. The remaining functionalities perform according to the specifications.

Dynamic objects:
The main deficiency of the system is the aspect of poor tracking of dynamic objects. If there is a moving object in the remote environment, the current implementation is not able to correctly update the VR environment. This becomes apparent by multiple versions of the same object (artifacts) in the VR environment. Figure 3 shows the dynamic artifacts in the generated SLAM VR environment in the RViz visualization tool.

However, it is observed that when the Kinect is not moving, correct dynamic object tracking is achieved nonetheless. This suggests that the motion of the Kinect induces a decreased tracking performance.

[1] https://github.com/introlab/rtabmap_ros/blob/master/launch/rgbd_mapping_kinect2.launch

[2] https://github.com/getsomatic/rviz_vive_plugin

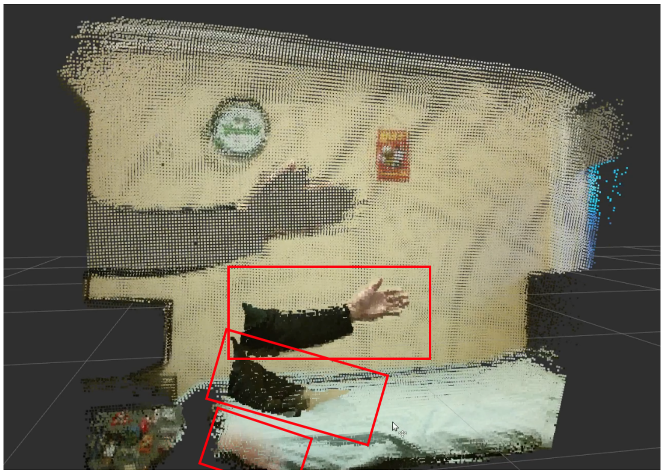


Fig. 3: Identified dynamic object artifacts in the generated VR environment. The red boxes indicate the same dynamic object displayed multiple times.

C. Root cause analysis

In this section, the identified system functionality which performs poorly is analyzed in more detail by using a root cause analysis. Figure 4 gives an overview of the RCA in a structured tree format.

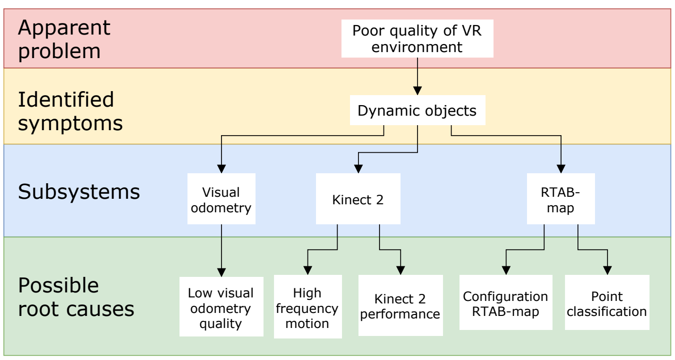


Fig. 4: Overview of the Root Cause Analysis in a structured tree format.

1) Visual odometry:

- *Low visual odometry quality*
  - Low visual odometry quality might have a significant impact on poor tracking of dynamic objects that are present in the system. Visual odometry can provide accurate estimates when sufficient keypoint features can be detected from the sensory image data. However, when not enough features are found, the visual odometry quality degrades. A more robust position estimate can be achieved with the inclusion of additional sensors such as an IMU [21].

2) Kinect 2:

- *Kinect 2 image distortion due to high-frequency motion*
  - High-frequency motion induced by the control signal originating from the HTC Vive might have an impact on the tracking performance of dynamic objects as images from the Kinect 2 can become distorted.
- *Kinect 2 performance*
  - In terms of hardware, the performance of the Kinect 2 sensor might be insufficient for dynamic object tracking capabilities. An increase in the number of (depth) pixels, range, and frame rate might provide the RTAB-map the sensory information it needs to perform adequately. However, increased performance by implementing a improved sensor such as the Azure Kinect[3] is unlikely since dynamic objects can be tracked adequately when the Kinect is not moving - indicating that the performance of the Kinect 2 is sufficient.

3) RTAB-map: The possible root causes listed below are at the core of the RTAB-map algorithm. Even though one of these aspects might be the actual root cause, they are beyond the scope of the work presented in this paper.

- *Configuration RTAB-map* - The RTAB-map core itself might limit the performance of the dynamic object tracking due to the chosen parameter settings. Several settings were attempted already to determine if they would improve the dynamic object tracking capabilities. Examples include changing the odometry strategy from F2M (frame to map) to F2F (frame to frame), changing the feature matching approach, and changing the linear/angular update rates [21]. Although the map quality did improve slightly in some cases, it did not result in improved tracking capabilities. Considering that the configuration is working correctly when the Kinect is not moving, it is unlikely that this would be the root cause of dynamic object tracking issues.
- *Point classification* - The visual odometry of RTAB-map uses RANSAC to determine the ego-motion of the Kinect sensor [21]. Data points that belong to dynamic objects are classified as outliers by the RANSAC algorithm [22]. When the Kinect is not moving, most captured data points are classified as non-moving. As a result, the Kinect camera motion estimate is zero and dynamic points can be easily detected and updated by the core of the RTAB-map algorithm. When the Kinect is moving, dynamic objects can interfere with this process. In that case, it is difficult to differentiate between the movement of dynamic objects and the ego-motion of the Kinect when using the RANSAC approach [22]. In other words, key points that are detected on a dynamic object can mistakenly be used in the ego-motion estimation. It is non-trivial to make this discrepancy between moving and non-moving points in the core of the algorithm and this can make it difficult for the core of the RTAB-map algorithm to track dynamic objects when the camera is moving.

Other subsystems of the RTAB-map algorithm such as the memory management approach was also considered as a possible root cause. However, it was found that the long term memory management component is disabled in the

[3]https://azure.microsoft.com/en-us/services/kinect-dk/

source code by default by the parameters 'MemoryThr' and 'TimeThr' [4]. Therefore, this subsystem of the algorithm cannot contribute to the dynamic object tracking issue.

D. Functional testing

Several functional tests were executed to investigate the identified possible root causes from sections IV-C1 and IV-C2.

1) Visual odometry: First of all, some effort was put into investigating the visual odometry quality of the setup. When the Twente humanoid head platform and the Kinect 2 were kept motionless, it was found that the RTAB-map algorithm is adequately updating dynamic objects in the resulting virtual environment.

When looking at the visual odometry data generated by the RTAB-map algorithm in Figure 5, several deficiencies can be noted. These include discontinuities in the orientation data of the odometry (①) and a noise floor present in the linear velocity data (②). Furthermore, it can be observed that the orientation data already has an offset at ③ compared to the starting position and this is not the case for the linear and angular velocity data at ③. Therefore, the linear and angular velocity data seems to lag the orientation data. Considering the aforementioned deficiencies, it was deemed that the quality of the visual odometry was insufficient.

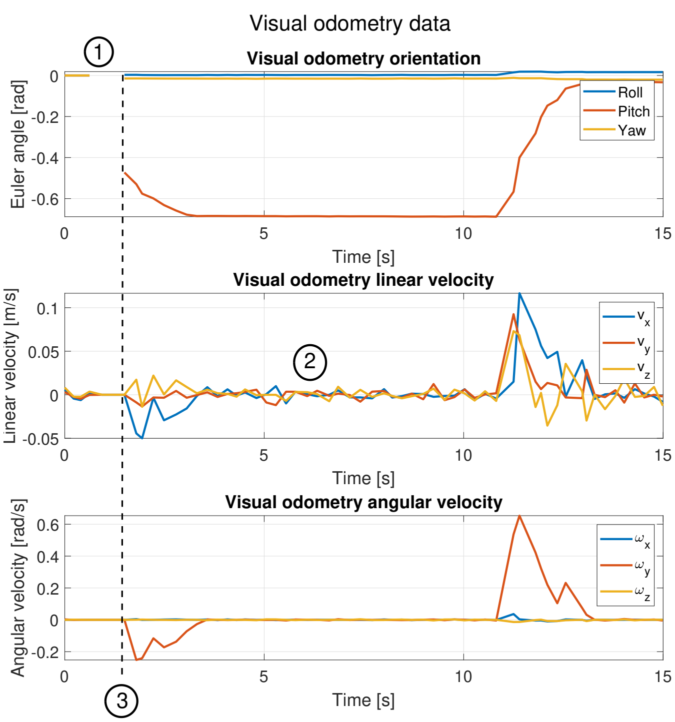


Fig. 5: Visual odometry data from the RTAB-map odometry algorithm. The Kinect 2 is moved between t=1s and t=13s in this time frame. ① - discontinuities in the orientation data. ② - noise floor in the linear velocity. ③ - Linear and angular velocity data seems to lag compared to the orientation data, as seen by the offset compared to the starting position in the orientation data. This offset is not present in the linear and angular velocity data in this time instant.

[4]https://github.com/introlab/rtabmap/blob/master/corelib/include/rtabmap/core/Parameters.h

To improve the odometry an Xsens IMU was added to the implementation and fused with the visual odometry data using an Extended Kalman Filter (EKF). The resulting fused odometry can be seen in Figure 6. Minor improvements can be found such as more continuous data and an increased number of data points overall. Table I gives an overview of the comparison between the standalone visual odometry and the fused EKF odometry when the Kinect 2 sensor is not moving. A decreased variance of the linear velocity can be observed when the fused odometry is used instead of the standalone visual odometry. The remaining variances are mostly in the same order of magnitude when comparing both odometry approaches.

Unfortunately, the improved odometry did not result in an improvement in terms of dynamic object tracking.

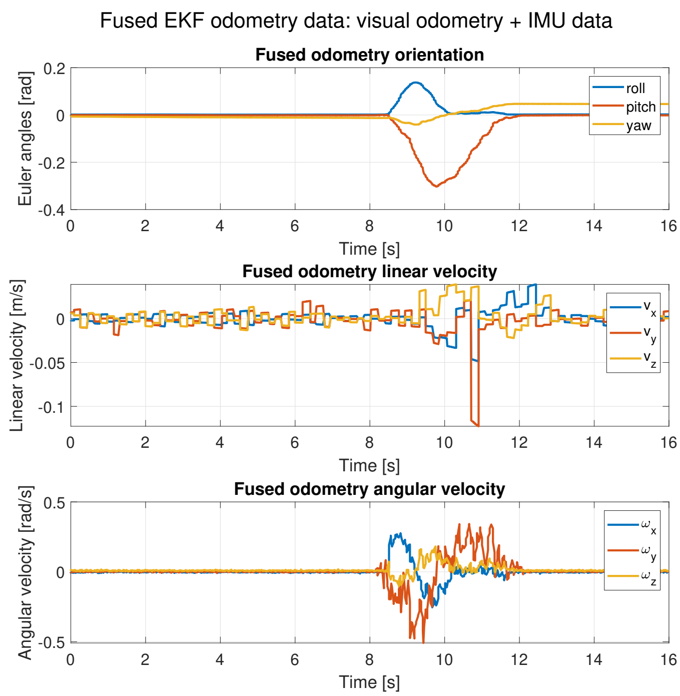


Fig. 6: Fused EKF odometry data produced by the Robot Localization ROS package which combines the visual odometry and IMU sensory data. The Kinect 2 is moved between t=8s and t=12s in this time frame. The resulting environment from this movement can be observed in Figure 3

TABLE I: Comparison of the standalone visual odometry and the fused EKF odometry when no movement is performed. The shown variance values are the average squared difference between the average value of the data points and the data points themselves.

Visual odometry			
	Variance X	*Variance Y*	*Variance Z*
Orientation (Euler)	5.4688e-08	1.4707e-07	2.7692e-07
Linear velocity	1.9888e-05	1.5443e-04	1.3152e-04
Angular velocity	3.3216e-06	7.5997e-06	1.0738e-05

Fused EKF odometery (Visual odometry + IMU data)			
	Variance X	*Variance Y*	*Variance Z*
Orientation (Euler)	2.4355e-08	3.8918e-07	1.2351e-05
Linear velocity	1.5663e-05	6.2316e-05	3.4447e-05
Angular velocity	9.6103e-06	9.8261e-06	7.8788e-06

2) Kinect 2: The proportional gain of the feedback controller was reduced to slow down the movement of the Twente humanoid head platform to investigate possible negative effects of high-frequency motion. Figure 7 and 8 show two different evaluated platform velocities. From these snapshots, it is clear that slow movement does reduce the number of artifacts and streaks present in the virtual environment. However, the dynamic object that was presented in both environments was not properly removed when it was not present anymore in front of the Kinect sensor. Therefore, the problem of dynamic object tracking persisted.

Fig. 7: Snapshot of the virtual environment after *fast* Kinect movement with a presented dynamic object. The red box indicates dynamic object artifacts.

Fig. 8: Snapshot of the virtual environment after *slow* Kinect movement with a presented dynamic object. The red box indicates dynamic object artifacts.

V. Discussion

As seen in section IV, dynamic object tracking is still a persisting problem that affects the performance of the implemented system. In the root cause analysis, several aspects were considered across all the subsystems. Primarily, the main focus was laid on improving the quality of the visual odometry and reduction of high-frequency motion since dynamic object tracking performed adequately when the Kinect 2 sensor was not moving. After the investigations that were performed in this paper, it can be said with a large amount of certainty that the visual odometry is not the root cause of the dynamic object tracking issue.

The Kinect 2 sensor was also taken into consideration in the root cause analysis. First of all, improving the hardware will most likely not result in improved performance since the output of the RTAB-map algorithm does function according to the performance requirement when it is not moving. Also, image distortion due to the high-frequency motion of the Kinect itself was regarded as a possible root cause. An increase in the VR environment quality was observed when the velocity of the platform and possible high-frequency movements in the enclosure of the Kinect 2 itself were reduced. However, it did not improve the dynamic object tracking performance, and it is therefore highly unlikely that the Kinect 2 sensor is the root cause of the dynamic object tracking issue.

As discussed in section IV-C3, the root cause for the dynamic object tracking might be due to the lacking performance of the RTAB-map algorithm itself. Several possible causes were mentioned in the RCA such as faulty parameter settings, long term memory management, and the classification of data points. Based on the adequate dynamic object tracking performance when the Kinect is not moving, the configuration of RTAB-map and the memory management aspect are highly unlikely to contribute to the dynamic object tracking issue. The remaining possible root cause regarding the classification of data points is most likely to be the main possible root cause. More research on RTAB-map related root causes is needed since these aspects are beyond the scope of the work presented in this paper.

More research is needed to make SLAM algorithms robust against dynamic objects and environments. Segmentation of dynamic objects is an approach that is frequently referred to in literature. Although not chosen, ORB-SLAM2 may provide more possibilities to adapt its algorithm and introduce dynamic segmentation compared to the RTAB-map algorithm [14], [22]. Likewise, there also exist algorithms that perform semantic segmentation of dynamic objects by default [13]. However, update rates of 8 Hz which are achieved in the implementation presented in this paper are currently not feasible in those algorithms.

VI. Conclusion

In this paper, a method was presented that decreases the effects of time delays in a telerobotic setup by using a visual SLAM-based VR environment. The experimental setup that was designed did not meet the performance requirements in terms of dynamic object tracking. By using a root cause analysis, it was found that the used SLAM algorithm (RTAB-map) is most likely the bottleneck in the implemented system. Future implementations of VR-based visual model-mediated approaches require improvements in segmentation of dynamic objects.

REFERENCES

[1] K. Shiratsuchi, K. Kawata, E. Vander Poorten, and Y. Yokokohji, “Design and evaluation of a telepresence vision system for manipulation tasks,” in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 4313–4318, 2007.

[2] S. Davis, K. Nesbitt, and E. Nalivaiko, “A systematic review of cybersickness,” in *Proceedings of the 2014 Conference on Interactive Entertainment*, (New York, NY, USA), pp. 8:1–8:9, ACM, 2014.

[3] K. Raaen and I. Kjellmo, “Measuring latency in virtual reality systems,” in *Entertainment Computing - ICEC 2015*, (Cham), pp. 457–462, Springer International Publishing, 2015.

[4] E. Reentov, G. G. Messier, and S. Magierowski, “Evaluating wireless network effects for slam robot map making,” in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pp. 1–5, Dec 2010.

[5] F. Brizzi, L. Peppoloni, A. Graziano, E. D. Stefano, C. A. Avizzano, and E. Ruffaldi, “Effects of augmented reality on the performance of teleoperated industrial assembly tasks in a robotic embodiment,” *IEEE Transactions on Human-Machine Systems*, vol. 48, pp. 197–206, April 2018.

[6] B. M. F. da Silva, R. S. Xavier, T. P. do Nascimento, and L. M. G. Gonsalves, “Experimental evaluation of ros compatible slam algorithms for rgb-d sensors,” in *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, pp. 1–6, Nov 2017.

[7] I. Z. Ibragimov and I. M. Afanasyev, “Comparison of ros-based visual slam methods in homogeneous indoor environment,” in *2017 14th Workshop on Positioning, Navigation and Communications (WPNC)*, pp. 1–6, Oct 2017.

[8] Stanford Artificial Intelligence Laboratory et al., “Robotic operating system.”

[9] R. Mur-Artal and J. D. Tardós, “ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras,” *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[10] M. Labbé and F. Michaud, “Online global loop closure detection for large-scale multi-session graph-based slam,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2661–2666, Sep. 2014.

[11] C. Huang and S. Shih, “Map memory management for real-time displaying in virtual experience,” in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 4132–4137, Oct 2018.

[12] Wei Tan, Haomin Liu, Z. Dong, G. Zhang, and H. Bao, “Robust monocular slam in dynamic environments,” in *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 209–218, Oct 2013.

[13] I. A. Bârsan, P. Liu, M. Pollefeys, and A. Geiger, “Robust dense mapping for large-scale dynamic environments,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7510–7517, IEEE, 2018.

[14] T. Sun, Y. Sun, M. Liu, and D.-Y. Yeung, “Movable-object-aware visual slam via weakly supervised semantic segmentation,” *arXiv preprint arXiv:1906.03629*, 2019.

[15] I. Caminal, J. R. Casas, and S. Royo, “Slam-based 3d outdoor reconstructions from lidar data,” in *2018 International Conference on 3D Immersion (IC3D)*, pp. 1–8, Dec 2018.

[16] R. Codd-Downey, P. M. Forooshani, A. Speers, H. Wang, and M. Jenkin, “From ros to unity: Leveraging robot and virtual environment middleware for immersive teleoperation,” in *2014 IEEE International Conference on Information and Automation (ICIA)*, pp. 932–936, July 2014.

[17] T. Rodehutskors, M. Schwarz, and S. Behnke, “Intuitive bimanual telemanipulation under communication restrictions by immersive 3d visualization and motion tracking,” in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pp. 276–283, 2015.

[18] S. Kim, Y. Kim, J. Ha, and S. Jo, “Mapping system with virtual reality for mobile robot teleoperation,” in *2018 18th International Conference on Control, Automation and Systems (ICCAS)*, pp. 1541–1541, 2018.

[19] R. Reilink, L. Visser, J. Bennik, R. Carloni, D. Brouwer, and S. Stramigioli, “The twente humanoid head,” in *IEEE International Conference on Robotics and Automation, ICRA 2009*, pp. 1593–1594, 8 2009. 10.1109/ROBOT.2009.5152436.

[20] N. A. Busch and R. VanRullen, “Spontaneous eeg oscillations reveal periodic sampling of visual attention,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 37, pp. 16048–16053, 2010.

[21] M. Labbé and F. Michaud, “Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[22] C. Junhao, Z. Wang, H. Zhou, L. Li, and J. Yao, “Dm-slam: A feature-based slam system for rigid dynamic scenes,” *ISPRS International Journal of Geo-Information*, vol. 9, p. 202, 03 2020.

Appendix A: Design space exploration

This appendix provides additional information about the design choices that were made in the Method section of the paper. This was done by using a design space exploration that elaborates on the possible implementation choices.

A. System overview

A system overview is given in Figure 1. A distinction is made between entities and interfaces. An entity can either be a piece of hardware or a distinct piece of software. Interfaces are meant to provide a bridge between entities. For example, the HMD driver interface provides a bridge between the visualization tool entity and the HMD entity. The vision sensor is connected to the platform by definition, therefore no arrow is present. In the next sections, the entities and interfaces will be explained further. These are marked light blue in Figure 1.

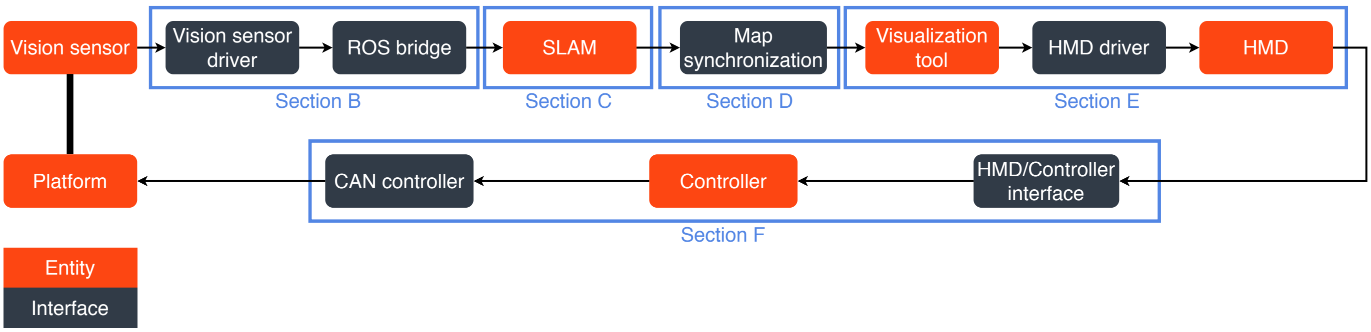


Fig. 1: A system overview. Entities are marked orange and interfaces are marked dark blue. Light blue indicates in which section of the analysis a subsystem is explained.

The system above that will be developed is based on the work of [1] and [2]. [1], investigated how to create an interface between the Kinect 2 and an HTC Vive headset and how to set up a virtual environment. [2], worked on controlling the platform based on the movement of an Oculus headset. There is room for improvement in both interfaces, therefore every aspect of the system will be assessed in this analysis. Finally, the system overview in Figure 1 incorporates both system interfaces and will need to function as a whole to create a working setup where the VR environment is continuously updated with new information from the Kinect 2 on the moveable platform.

Table I provides a component listing of the system depicted in Figure 1 along with their corresponding functionalities.

TABLE I: System components listing

	Type	**Hardware/ software**	**Location**	**Function**
Vision sensor	Entity	Hardware	Remote	Obtains point cloud data from the environment
Vision sensor driver	Interface	Software	Remote	Enables the vision sensor to communicate with the computer's operating system
ROS bridge	Interface	Software	Remote	Provides a bridge between the driver software and ROS
SLAM	Entity	Software	Remote	Creates a virtual environment from the incoming point cloud data
Map synchronization	Interface	Software	Remote + Local	Synchronizes the remote and local point cloud map
Visualization tool	Entity	Software	Local	Displays point cloud data and allows for interfacing with an HMD
HMD driver	Interface	Software	Local	Allows for interfacing between the visualization tool and the HMD
HMD	Entity	Hardware	Local	Allows the user to explore the virtual environment
HMD/Controller interface	Interface	Software	Local/remote	Interfaces the HMD and controller for the platform
Controller	Entity	Software	Remote	Controls the platform by using the incoming HMD data
CAN controller	Interface	Software	Remote	Provides a bridge between the controller and the platform motors
Platform	Entity	Hardware	Remote	Allows the vision sensor to move

B. Vision sensor to ROS

The visual input to the system is obtained by using a vision sensor that can capture point cloud data. For this, the Microsoft Kinect Sensor V2 (also known as Kinect for Xbox One) will be used since there is no other option available at this moment. This is a RGB camera that can capture 1920x1080 images with a corresponding depth image of 512x424 [1]. Currently, there is only one driver available to interface with the Kinect 2 hardware in Linux. Libfreenect2 is part of the OpenKinect project [2] and is open source. To interface the Kinect 2 with ROS where the SLAM algorithm runs, a ROS bridge is needed. This can either be provided by the iai_kinect2 or kinect2_ros package from GitHub. Since the latter is a fork of the iai_kinect2 software and has not had an update since 2015, this option is not preferred. Other interfacing options are not found in the design space exploration. Besides being able to obtain the point cloud data, the iai_kinect2 also provides a calibration tool and a viewer for the point clouds [3].

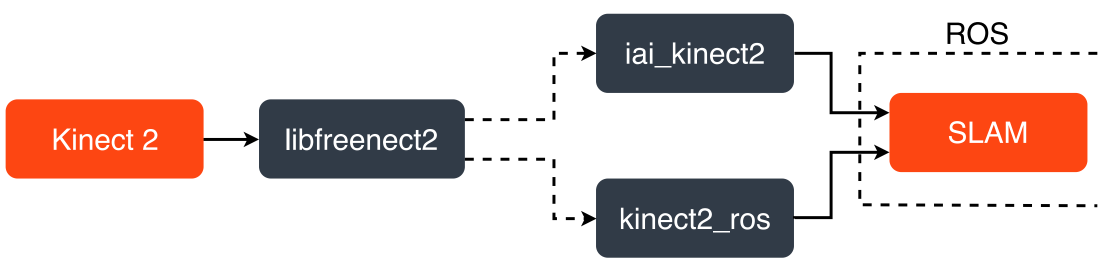


Fig. 2: Possible chains from Kinect 2 to the ROS environment

C. SLAM

SLAM has been proven to be an efficient algorithm for various goals. These include creating maps of unknown environments, VR implementations and AR-based enhancements [3]. SLAM algorithms that are built upon the Robotic Operating System (ROS) are common and can be considered as the de facto standard [4] [5].

In the context of this assignment, a dense 3D map needs to constructed based on point cloud data from a moveable stereo RGB-D camera (Kinect 2) which is mounted on the remote-controlled humanoid head. This will allow the user to explore a virtual environment and get a better sense of the environment in which the robot is placed.

Labbé et al. provided an extensive overview of popular visual SLAM approaches [6]. Based on inputs and outputs which these algorithms can provide, a suitable algorithm(s) can be chosen. Labbé selected 17 ROS-compatible SLAM algorithms:

[1]https://docs.depthkit.tv/docs/kinect-for-windows-v2
[2]https://openkinect.org/wiki/Main_Page
[3]https://github.com/code-iai/iai_kinect2

GMapping, TinySLAM, Hector SLAM, ETHZASL-ICP, Karto SLAM, Lago SLAM, Cartographer, BLAM, SegMatch, VINS-Mono, ORB-SLAM2, S-PTAM, DVO-SLAM, RGBiD-SLAM, MCPTAM, RGBDSLAMv2 and RTAB-Map. State of the art candidates are the ORB-SLAM2 and RTAB-Map algorithms. These algorithms provide an efficient solution for RGB-D type cameras, which can handle high-density point clouds by using efficient map memory management [7] [8]. In the RTAB-Map algorithm, for example, a distinction is made between working memory and long term memory which makes it computationally efficient [9]. [4] states that RTAB-Map is a preferred solution when using 3D stereo SLAM, however, it requires more processing power compared to ORB-SLAM2.

The main focus of ORB-SLAM2 is not to provide dense point clouds. As stated in their recent research, [7]: "Our goal is long term and globally consistent localization instead of building the most detailed dense reconstruction". However, dense reconstruction possibilities are presented in the same paper. Unfortunately, these rely on the backprojection of the depth maps. On GitHub, a dense point cloud reconstruction algorithm is presented based on the ORB-SLAM2 algorithm [4]. The performance of this algorithm is unknown. Regarding online topics with point cloud information generated by the ORB-SLAM2 algorithm in ROS, contradicting information is found. [6] mentions that there is no ROS topic information present on point clouds in the algorithm. [10] points out the contrary.

Handling of point cloud data in ROS can be handled usingf point cloud messages. This is the *PointCloud2* message and Table II gives insight in how the data is structured in such a message[5]. RTAB-Map and ORB-SLAM2 can both produce these types of messages [6], [10]. Other possible format types which can be used for storing point cloud data are *XYZ*, *PLY* and *OBJ* [1]. However, since this is not the common way of interfacing with point clouds in ROS, this will require a lot of unnecessary file conversions.

TABLE II: Message definition of PointCloud2 in ROS

Message field	**Description**
std_msgs/Header header	Time of data acquisition and coordinate frame ID
uint32 height	2D height of the point cloud
uint32 width	2D width of the point cloud
sensor_msgs/PointField[] fields	Describes the channels and their layout
bool is_bigendian	Indication of endianness
uint32 point_step	Length of a point in bytes
uint32 row_step	Length of a row in bytes
uint8[] data	Actual point data
bool is_dense	True if no invalid points are found

1) SLAM algorithm comparison: Table III presents an overview of the comparison between RTAB-Map and ORB-SLAM2. A question mark indicates that the feature of the SLAM algorithm is unknown.

TABLE III: SLAM algorithm comparison table

Feature / **SLAM algorithm**	ROS implementation	Point cloud density	Online point cloud output	Stereo RGB-D support
GMapping	+	?	-	-
TinySLAM	+	?	-	-
Hector SLAM	+	?	-	-
ETHZASL-ICP	+	+	+	-
Karto SLAM	+	?	-	-
Lago SLAM	+	?	-	-
Cartographer	+	+	+	-
BLAM	+	+	+	-
SegMatch	+	+	+	-
VINS-Mono	+	?	+	-
ORB-SLAM2	+	-	+-	+
S-PTAM	+	-	+	+
DVO-SLAM	+	?	-	-
RGBiD-SLAM	+	?	+-	-
MCPTAM	+	-	+	+
RGBDSLAMv2	+	+	+	-
RTAB-Map	+	+	+	+

Conclusion

RTAB-Map will be chosen over ORB-SLAM2 since it provides a higher point cloud density and the online output support is good due to the availability of ROS topics that publish the point cloud data.

[4]https://github.com/bikong2/ORB_SLAM2_dense
[5]http://docs.ros.org/melodic/api/sensor_msgs/html/msg/PointCloud2.html

D. Map synchronization

A significant challenge in this assignment is the notion of map synchronization. The produced map by the SLAM algorithm on the remote robots needs to be transferred to the local robot. In this section, a strategy is proposed which only sends the map difference instead of a continuous stream of the entire map data. Figure 3 gives an overview of the proposed map synchronization strategy for the system under design. The core idea here is to determine the map difference in every update which comes from the SLAM algorithm. The map difference is then sent to the local side and added to the map of the user in the visualization tool which allows for incremental updates for the VR environment of the user. In the following subsection, several aspects of the mapping strategy are discussed and explained.

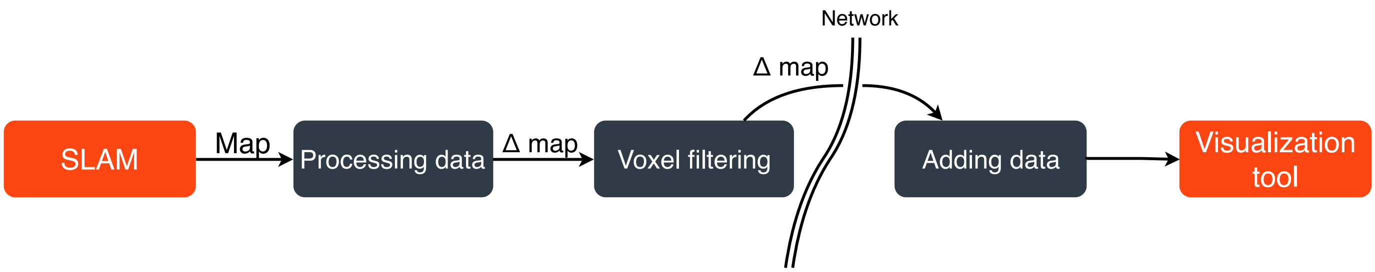


Fig. 3: Proposed map synchronization strategy

1) Processing data: The first step of the process is extracting the data from the SLAM algorithm to determine the difference. In the RTAB-Map, a *sensor_msgs/PointCloud2* message is placed on the rostopic */rtabmap/cloud_map* [6]. This shows that is it relatively easy to obtain the point cloud data from the RTAB-map algorithm. According to the wiki of ORB-SLAM2: "All nodes publish (given the settings) a PointCloud2 containing all key points of the map" [7]. However, it is important to note that is not verified if the published data contains all the point cloud data or only the key points. If ORB-SLAM2 is used in the future, this needs to be examined in more detail.

As discussed before, this PointCloud2 message has a data array which can then be used to determine the difference been the current and previous map. It is important to take into account the case where a single point in the cloud takes on a new RGB value. Also, conversion to PLY format is needed for the visualization tools Unity3d and Unreal Engine 4.

It would be possible to use 3D party software such as CloudCompare, Polyworks or 3DReShaper, but these software programs do not always support Linux or real-time processing [8] [9] [10].

2) Voxel filtering: An important aspect regarding the mapping process is the notion of data reduction. It might be the case that the data stream from the Kinect 2 is too large. This can cause more delay when adding new data to the VR environment of the user. Voxel filtering is a data reduction approach that can be implemented when working with point cloud data. In essence, it reduces the number of points in a point cloud, while maintaining detail. Voxels are the 3D equivalent of pixels and provide a grid representation of a point cloud. Every 'pixel' in a voxel grid has a single intensity value which can be calculated by filtering (e.g. Gaussian filtering). The voxel grid representation is smaller in size and complexity due to the standard geometry of the grid [11], [12]. A graphical representation of this process can be seen in Figure 4.

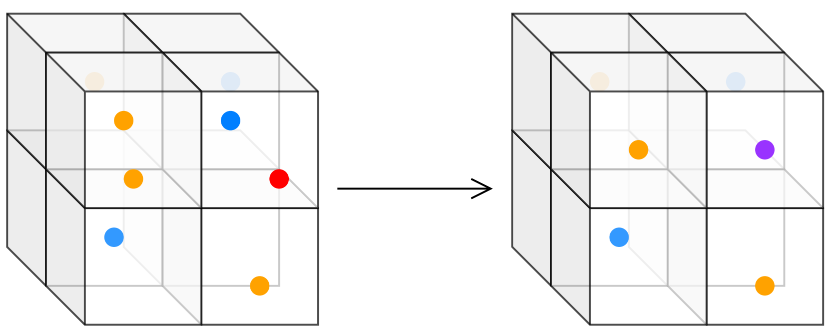


Fig. 4: Visual representation of the voxel filtering process. The boxes indicate the 3d voxel grid and the point cloud is represented by the colored points. If multiple points are present in a single voxel, they are combined to a single point based on location and intensity value.

[6]http://wiki.ros.org/rtabmap_ros
[7]http://wiki.ros.org/orb_slam2_ros
[8]http://github.com/cloudcompare/cloudcomparez
[9]https://www.innovmetric.com/en/products-solutions/polyworksinspectortm
[10]https://www.3dreshaper.com/en/

While this process does improve the speed of transmission of a voxel map is improved, the quality of the map is affected. This is determined by the filter size of the voxel filtering approach. As this filter size gets larger, the quality of the VR environment will deteriorate.

[13], provides an open-source Point Cloud Library (PCL), which can be used for several point cloud manipulations. This also includes voxel filtering of point clouds. ROS provides a PCL package which its allows for a convenient implementation process [11]. PCL voxel filtering is downsampling based on the centroid of a voxel instead of the center. This is somewhat slower, but it results in a better representation of the environment [12].

3) Adding data: The last step of the mapping strategy requires the point cloud data to be added to the visualization tool. This will be dependent on the choice of the software. For example, if RViz will be used, adding data can be done in a similar fashion as extracting the data. First, the difference needs to be determined and then values need to be replaced and/or added to the point cloud in the simulation tool. However, this process gets more difficult when Unity3D is used since this requires the specific PLY-format which needs more data conversion/processing. The same holds for the Unreal Engine 4 software.

E. ROS to HMD

To provide for an interface between ROS and an HMD, several options are available. First of all, two pieces of hardware are available for use; the Oculus DK2 and the HTC Vive. Furthermore, 4 visualization tools are possible as a placeholder for the virtual environment of the user. These are RViz, Unreal Engine 4, Gazebo and Unity3D and were picked because of their great online support and/or ease of implementation in ROS. This section shows the possibilities for the implementation sorted by the choice for the visualization tool.

1) RViz: Figure 5 provides an overview of the possible implementations based on the RViz visualization tool. There are several plugins available for the Oculus in RViz and many of them are based on OgreOculus and OsgOculusViewer. In this implementation, the Oculus is set up as a secondary screen. However, specific support for the Oculus DK2 is limited and likely to be unsupported in the near future [13]. Running the Oculus as a 'Display type' results in segmentation faults. Support for the HTC Vive in RViz seems more promising. Somatic provides the open-source 'rviz_vive_plugin' which sends localization information and controller data and publishes them on topics in ROS. More importantly, it supports the required stereo video output from Rviz to the HTC Vive headset [14]. The RWTH Aachen developed a similar package, however, this only supports stereo video streaming and is therefore not preferred [15]. Finally, point clouds in RViz have native support [16].

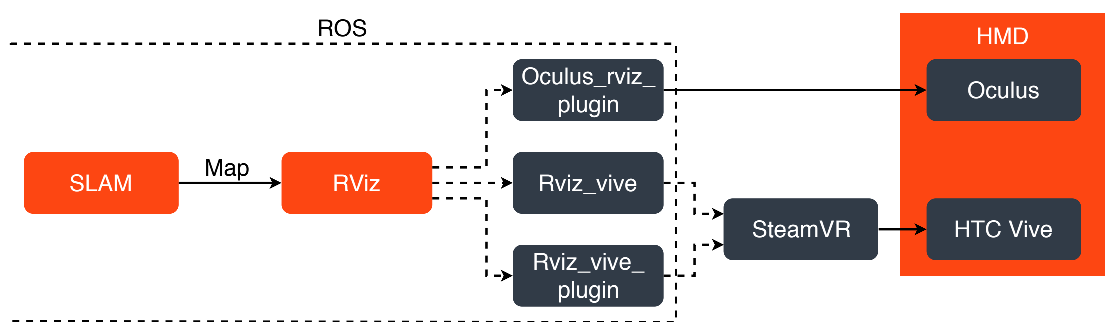


Fig. 5: Possible chains from the ROS environment to either the Oculus or the HTC Vive based on the *RViz* visualization tool.

2) Unreal Engine 4: Figure 6 provides an overview of the possible implementations based on the Unreal Engine 4 visualization tool. To get access to this software a personal account has to be created and a link to GitHub is required [17]. Unreal engine has a plugin for Oculus which can be used as well as a point cloud rendering tool [18]. For the HTC Vive the 'Viveport' software is needed [19].

[11] http://wiki.ros.org/pcl_ros/Tutorials/VoxelGrid%20filtering
[12] http://pointclouds.org/documentation/tutorials/voxel_grid.php
[13] https://github.com/ros-visualization/oculus_rviz_plugins/issues/19
[14] https://github.com/getsomatic/rviz_vive_plugin
[15] https://github.com/AndreGilerson/rviz_vive
[16] http://wiki.ros.org/rviz/DisplayTypes/PointCloud
[17] https://www.youtube.com/watch?v=WseUe9e1xBw
[18] https://www.unrealengine.com/marketplace/en-US/slug/lidar-point-cloud
[19] https://developer.viveport.com/documents/sdk/en/unrealengine.html

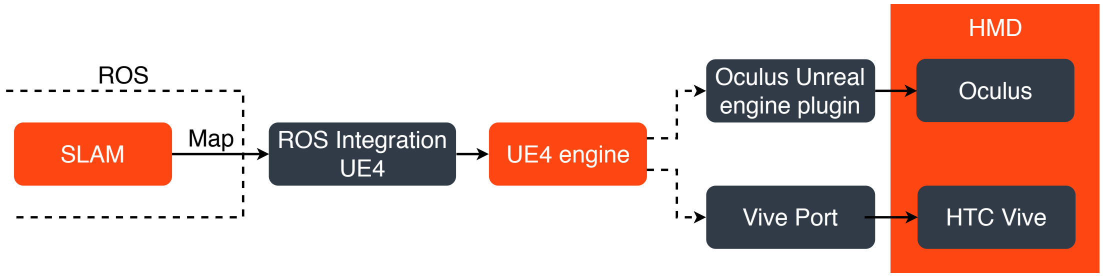


Fig. 6: Possible chains from the ROS environment to either the Oculus or the HTC Vive based on the *Unreal Engine 4* visualization tool.

3) Gazebo: Figure 7 provides an overview of the possible implementations based on the Gazebo visualization tool. Natively, Gazebo is more suited for dynamic simulations and therefore it is not the ideal choice. However, implementation-wise it is a very convenient program to use since it is built into ROS by default and has excellent documentation. Besides that, the Oculus DK2 can easily be configured by using the Oculus VR SDK in Linux [20]. The HTC Vive might require more work since the only available option is the vrui_mdf software which needs SteamVR to function properly [21]. A big downside of using Gazebo is the lack of support for point clouds. If this were to be used, a dedicated plugin needs to be written which is out of the scope of this project.

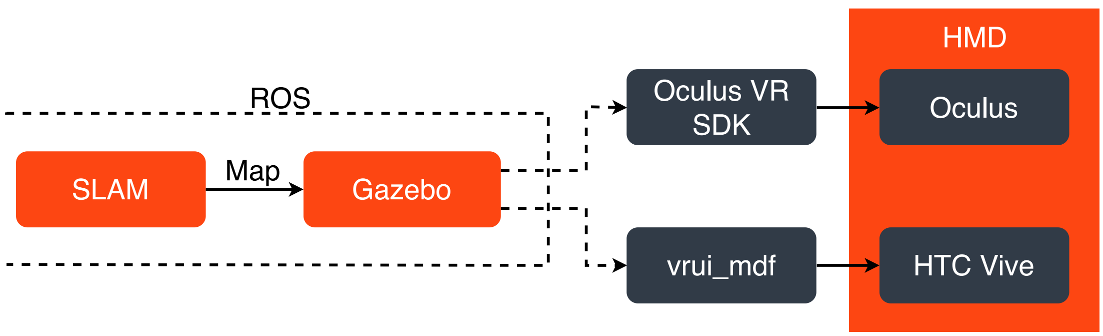


Fig. 7: Possible chains from the ROS environment to either the Oculus or the HTC Vive based on *Gazebo*.

4) Unity3D: Figure 8 provides an overview of the possible implementations based on the Unity visualization tool. Starting May 2019, Unity3d is officially available for Linux [22]. Similar to the Unreal Engine software, Unity is an external piece of software that requires a bridge with ROS. For this purpose, the ROS# (ROS sharp) open-source library can be used. It is specifically tailored to work with Unity3D [23]. The bridge can be used to invoked ROS services between Unity and ROS [14]. The bridge works by using web sockets with JSON API. This can create a web socket server on the 9090 port and allows for interfacing with Unity API [24]. To be able to import point cloud data, a plugin is needed [25]. However, it requires the data to be in PLY-format.

It is convenient to work with Unity3D in combination with HMD's due assets (plugin) which are available. For Oculus, Unity has native support available [26] [27]. For the HTC Vive, the integration works through the SteamVR asset in Unity [28].

[20]http://gazebosim.org/tutorials?tut=oculus&cat=rendering
[21]https://github.com/zhenyushi/vrui_mdf
[22]https://blogs.unity3d.com/2019/05/30/announcing-the-unity-editor-for-linux/
[23]https://github.com/siemens/ros-sharp
[24]https://icave2.cse.buffalo.edu/resources/sensor-modeling/ROS%20and%20Unity.pdf
[25]https://github.com/keijiro/Pcx
[26]https://developer.oculus.com/downloads/package/unity-integration/
[27]https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022
[28]https://www.raywenderlich.com/9189-htc-vive-tutorial-for-unity#toc-anchor-001

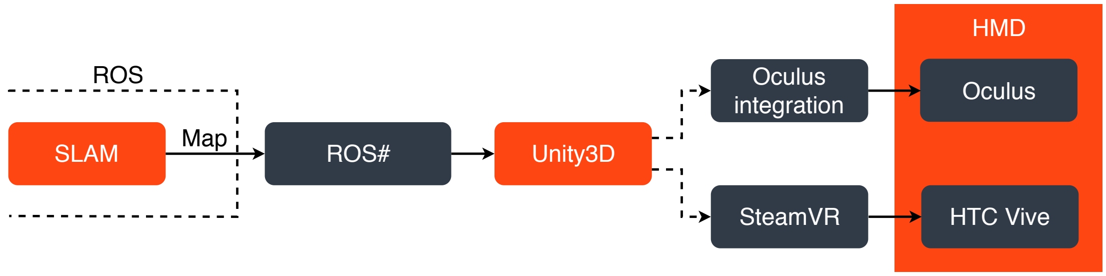


Fig. 8: Possible chains from the ROS environment to either the Oculus or the HTC Vive based on *Unity*.

5) Visualization software comparison: Table IV presents an overview of the comparison between the various visualization tools discussed in the previous subsections.

TABLE IV: Visualization software comparison table

Feature / **Software**	ROS compatibility	HMD support	Point cloud support
RViz	+	+-	+
Unreal Engine 4	-	+	+-
Gazebo	+	-	-
Unity3D	-	+	+-

Conclusion

RViz will be the visualization tool of choice since it has the best support for point clouds and has the best ROS compatibility compared to Unreal Engine 4, Unity3D and Gazebo. The HMD of choice will be the HTC Vive. This is based on the choice for RViz, since this visualization software does not interface well with the DK2 (as discussed in section -E1). Furthermore, the HTC Vive is more up-to-date compared to the Oculus DK2 in terms of online support and the available software and plugins. To interface RViz and the HTC Vive, the RViz_vive plugin in combination with SteamVR will be used.

F. Platform control

The platform setup, designed by [2], will be used as it is and the design will be altered if necessary. It consists of an HMD/controller interface that converts quaternion data to Euler angle data, a controller that uses these angles to produce a control signal and a CAN controller to drive the motors of the platform.

More intelligent control might be needed however. This can entail, for example, filtering the movement of the user if they move around a lot. Or Lidar-like behavior might need to be implemented for scanning the environment if the normal movement of the robotic head based on the movement of the user does not suffice. Based on the choice of HMD, the control software might need a different interface. Currently, orientation data from the Oculus is used for the control of the humanoid head. If the HTC Vive were to be used, the source of the incoming sensory data needs to be redefined.

References

[1] A. Sun, "Building a virtual world in ros based on the robot's perception," August 2019.
[2] R. van der Zee, "Remote control of a robotic humanoid head by using an oculus vr headset," April 2019.
[3] F. Brizzi, L. Peppoloni, A. Graziano, E. D. Stefano, C. A. Avizzano, and E. Ruffaldi, "Effects of augmented reality on the performance of teleoperated industrial assembly tasks in a robotic embodiment," *IEEE Transactions on Human-Machine Systems*, vol. 48, pp. 197–206, April 2018.
[4] B. M. F. da Silva, R. S. Xavier, T. P. do Nascimento, and L. M. G. Gonsalves, "Experimental evaluation of ros compatible slam algorithms for rgb-d sensors," in *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, pp. 1–6, Nov 2017.
[5] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system."
[6] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.
[7] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
[8] M. Labbé and F. Michaud, "Online global loop closure detection for large-scale multi-session graph-based slam," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2661–2666, Sep. 2014.
[9] C. Huang and S. Shih, "Map memory management for real-time displaying in virtual experience," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 4132–4137, Oct 2018.
[10] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
[11] M. Miknis, R. Davies, P. Plassmann, and A. Ware, "Near real-time point cloud processing using the pcl," in *2015 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pp. 153–156, Sep. 2015.
[12] T. He, T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang, "Voxel based object simplification," in *Proceedings of the 6th Conference on Visualization '95*, VIS '95, (Washington, DC, USA), pp. 296–, IEEE Computer Society, 1995.

[13] R. B. Rusu and S. Cousins, “3d is here: Point cloud library (pcl),” in *2011 IEEE International Conference on Robotics and Automation*, pp. 1–4, May 2011.
[14] E. Sita, C. Horvath, T. Thomessen, P. Korondi, and A. Pipe, “Ros-unity3d based system for monitoring of an industrial robotic process,” December 2017.

APPENDIX B: IMPLEMENTATION DETAILS

This appendix gives an overview of the implemented system. It provides more details compared to the Method section of the paper.

A. System overview

In this subsection, an overview of the system will be presented. Figure 1 shows the interconnections between all the software and hardware components of the implemented setup. How to install and run the setup is explained in Appendices A and B.

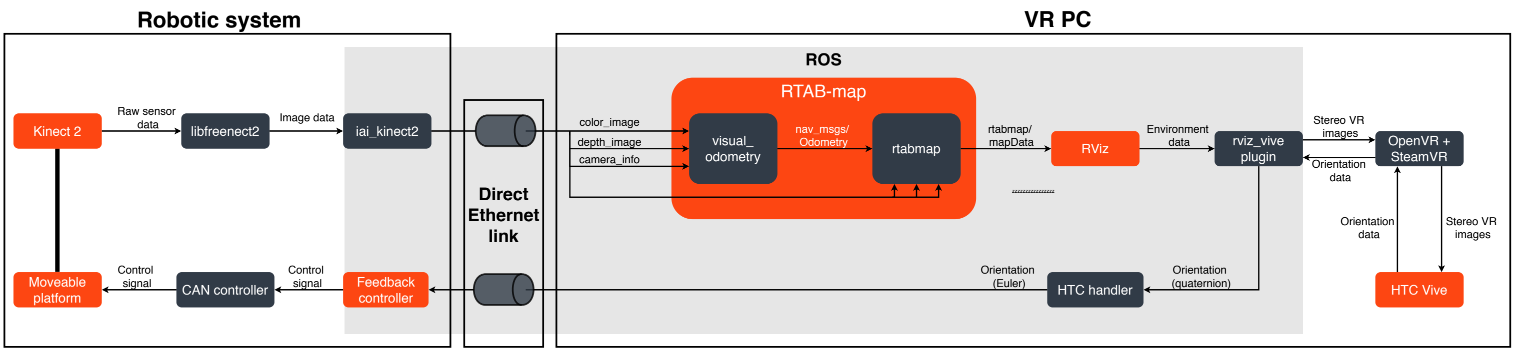


Fig. 1: System overview of the implemented system.

1) Twente humanoid head platform: Figure 2 shows the mechanical realization of the robotic platform for the Kinect 2. Several changes were made compared to the original realization [1]. Originally, the whole structure was placed on a base with wheels which could be used to drive the robot around. This was removed since the goal of this research is to provide a proof-of-concept only and a platform which could provide pitch, jaw, and roll was sufficient to create a SLAM-based map of an environment.

Furthermore, on the back of the Twente humanoid head platform, a plate is mounted which contains the new power unit. This provides 5, 19, and 24 Volt power supplies. AN Intel NUC Kit D54250WYK was introduced to provide for computing power and the Kinect 2 is connected to it by using a USB-serial connection. For now, an Ethernet connection is used to provide for the data communication between the platform and the VR PC. For future additions to the system, such as a new base with wheels which makes the platform drive-able again, a wireless network adapter has to be added since the chosen Intel NUC does not have wireless capabilities.

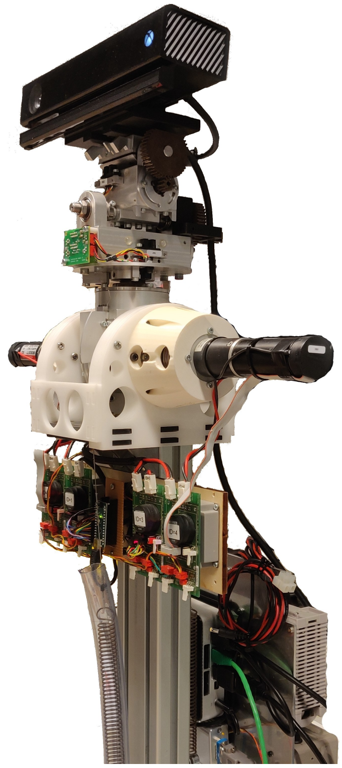


Fig. 2: The Twente humanoid head platform used to accommodate for the movement of the Kinect 2.

Figure 3 shows a render with a model of the Kinect 2 and the connector part that was designed. A single 1/4-inch bolt is used to clamp the Kinect to the part. Next, two M6 bolts are used to mount the connector part to the top of the platform. Also, a small 3D printed cable guide was designed such that the cable does not interfere with the moving parts. The parts were printed using an Ultimaker S5 3D printer with Polylactic acid (PLA) filament as material.

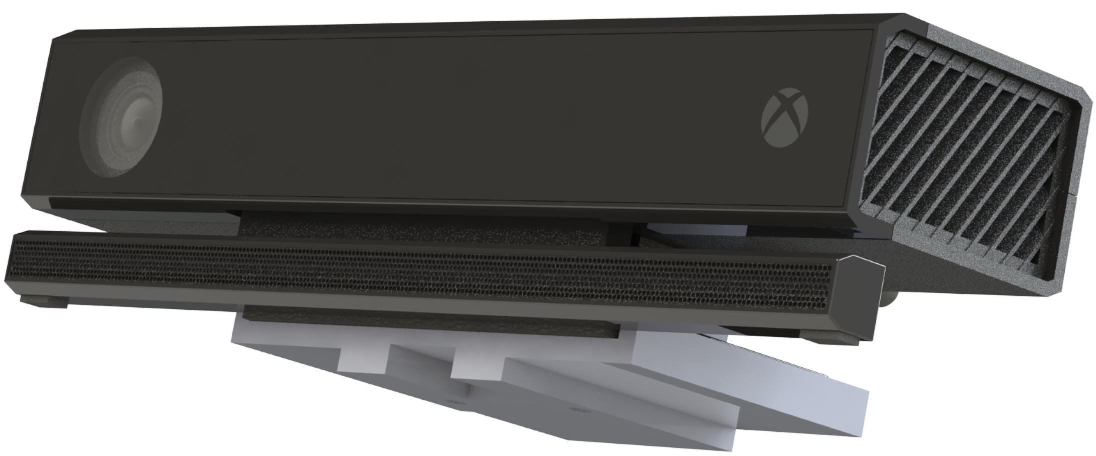


Fig. 3: Render of the connector part for the Kinect 2

The software which is ran on the NUC is kept to a minimum due to the limited CPU power present. In essence, the NUC only obtains the images from the Kinect 2 and sends them over Ethernet within the ROS network. Significant processing power that is needed by RTAB-map is done on the local side of the system. The Kinect 2 launch configuration[1] for the RTAB-map algorithm was used with the 'DetectionRate' parameter set to 20 Hz in order to achieve an increased output rate.
Furthermore, the Twente humanoid head platform has the feedback controller present on the NUC. At a rate of 40 Hz, the platform receives status updates from the local side. That is, a new set point for the platform expressed in Euler angles which are limited by the hardware limits of the platform itself. This is required since the hardware restricts full 360 degrees movement due to cabling.
All the controller components for the platform were written in Python [1]. However, it would be more convenient if written in C++. This is due to the adaptation of general C++ component design in the i-Botics group and the Python language cannot be used for this [2]. Due to the complexity of the feedback controller code structure which would require too much time to comprehend and the delicacy of the mechanical setup, it was chosen not to rewrite the existing code from Python to C++. Nonetheless, the interface between the controller and the orientation input signal needed to match up correctly. This boiled down to two requirements; a refresh rate of 40 Hz and a correct pitch/roll/yaw format for the orientation. The conversion from the HTC Vive orientation data to this required input signal is described in section -D.

B. Map synchronization

Deviations can be noticed when comparing the overview above to the conceptual design. The biggest of them all is the lack of the map synchronization module. In initial feasibility tests, point cloud data was manipulated to determine the difference with the previous point cloud. By using for-loops an incoming point cloud from the Kinect was checked for the existence of a new point compared to the previous point cloud. It was found that not only new points were added, but also the old points were updated or removed. Furthermore, if a whole room was scanned, the total number of points that were present could easily exceed more than 10.000. As a result, a significant portion of the entire map needed to be synchronized in every iteration, and the resulting overhead increased significantly. Simply streaming the point cloud map required less computational power.
Besides the computationally intensive algorithm, the computational power of the Intel NUC on the Twente humanoid head platform was proven to be a bottleneck. At first, the conceptual design was implemented and this required installation of the RTAB-map algorithm on the Intel NUC. This was not feasible due to the limited computing power of the NUC and became apparent with a CPU usage of 100%.
As a result, only the driver of the Kinect and the platform control was placed on the NUC. This eliminated the use of map synchronization since the map is only produced locally on the VR PC. The RTAB-map algorithm in combination with RViz within ROS can be considered the visual model mediated aspect of the system, where the input is low-frequent and the model of the environment at the output is high-frequent. The output rate of the RTAB-map algorithm can reach up to 8 Hz, while the output rate of RViz is rendered to the HTC Vive at 90Hz, regardless of the output frequency of RTAB-map. Thus, the user does not perceive the time delay related issues that can occur before the visual model mediation aspect of the system.

C. HTC vive interface

The HTC Vive interface is provided by the rviz_vive_plugin, OpenVR, and SteamVR as seen in Figure 1. The rviz_vive_plugin is loaded as a plugin in RViz. As a result, the plugin is able to obtain the contents of the virtual RViz environment. A render is made from the environment using OGRE which is an open-source library for rendering 3D graphics [2]. This results in stereo VR images which can be sent to the HTC Vive by using OpenVR and SteamVR. OpenVR provides an API to allows access to the VR headset without having specific knowledge of the headset hardware itself. SteamVR is used to perform a calibration of the HTC Vive setup and it provides a run-time environment.
The software chain described above is also used in the opposite direction. The orientation data originating from the sensors of the HTC Vive is sent through the SteamVR and OpenVR API back to the ROS environment. After that, it can be processed to generate a control signal for the Twente humanoid head platform.
The rviz_vive_plugin needed some small modifications. As it turned out, the default orientation of the HTC Vive rendering orientation did not correspond to the orientation of the map generated by the RTAB-map algorithm. A choice was made to alter the default orientation of the rviz_vive_plugin. To do this a slight change was made in the source code of the plugin and the node was recompiled. This corrected the orientation issue. The exact source code alteration can be found in Appendix A. As an alternative, an additional ROS node could have been implemented. However, this would require much more time to implement and the resulting overhead could have a negative impact on the operation speed of the system. Also, it would unnecessarily complicate the systems' framework. Therefore, this additional ROS node implementation was omitted.

[1] https://github.com/introlab/rtabmap_ros/blob/master/launch/rgbd_mapping_kinect2.launch
[2] https://www.ogre3d.org/

D. Quaternion conversion

From Figure 1 it can be seen that the HTC handler subsystem provides the bridge between the rviz_vive_plugin and the controller for the Twente humanoid head platform. As described in the previous section, the plugin obtains the orientation data from the HTC Vive. This is quaternion data and is published onto a ROS topic. The HTC handler subsystem is then able to process this data and convert it to the required orientation data format for the platform controller. That is, convert the quaternion angles to Euler angles at a rate of 40 Hz.
The same Quaternion to Euler angles conversion principle used by Van Der Zee is used [1], [3]. However, it was converted to the C++ language and the Eigen Library was used to ease the vector implementation of the algorithm [3]. The output topic speed was set to 40 Hz by default to match the input for the controller of the platform. Additionally, changes were made regarding the safety check for the Euler angle output. Originally, the pitch, roll, and yaw values were bounded to prevent the platform from reaching its hardware limits. During testing, it became apparent that oscillations were present in the motion of the platform. To keep these oscillations to a minimum, the safety bounding values for the yaw angle were changed with respect to the original python implementation. See table I for a complete overview of the implemented bounding values.

TABLE I: The implemented bounding values for control signal.

Euler angle / **Bound**	*Pitch*	*Roll*	*Yaw*
Lower bound	-0.611 rad	-0.855 rad	-0.800 rad (Original -1.780)
Upper bound	0.785 rad	0.855 rad	0.800 rad (Original 1.780)

However, this decreases the effective field of view for the user since the maximum yaw angle was decreased. Figure 4 gives a geometric representation of the space which the user can observe. This is based on the pitch and yaw angle bounds presented in Table I and the 4.5 meter depth limitation of the Kinect [4].
Besides the quaternion to Euler angles conversion, the HTC handler subsystem also implements a reset functionality for the created VR environment. This feature can be best described as an added novelty and not so much as an actual contribution to the core functionality of the system. The reset of the environment can be invoked by sending an empty service request to the rtabmap node (<std_srvs::Empty>(”/rtabmap/reset”)). The user is able to trigger this request by pressing the trackpad of the HTC Vive.

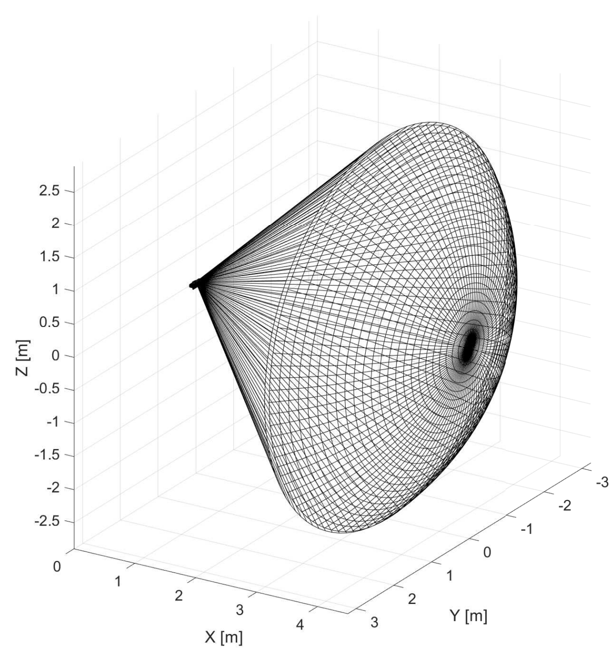


Fig. 4: Effective field of view for the user. With the Kinect present in the origin (0,0,0), the volume in this figure represents the space that the Kinect is able to observe based on the pitch and yaw angle bounds and the 4.5 meter depth limitation of the Kinect 2 sensor.

[3]http://eigen.tuxfamily.org/index.php?title=Main_Page
[4]https://docs.depthkit.tv/docs/kinect-for-windows-v2

E. Delay node

Although not specifically present in Figure 1, there is an additional delay node present in the implemented system. Since the delay functionality is not needed when using the system, it is omitted in the system overview. The delay node is implemented between the rtabmap_ros node and RViz to slow down the rate at which the virtual environment is updated for the user. Furthermore, the delay node also slows down the communication between the HTC handler node and the feedback controller of the Twente humanoid head platform. This decreases the responsiveness of the platform based on the movement of the HTC Vive headset. Together, the implemented delay nodes simulate the effect of large distances on the speed of the communication between the local system and the remote platform. Besides the visual delay, the user is not affected by the negative impact of the time delay.

References

[1] R. van der Zee, "Remote control of a robotic humanoid head by using an oculus vr headset," April 2019.
[2] D. Ellery, "Writing reusable code for robotics," December 2017.
[3] R. M. Murray, Z. Li, S. S. Sastry, and S. S. Sastry, *A mathematical introduction to robotic manipulation*. CRC press, 1994.

Appendix C: Performance evaluation

This appendix provides a more elaborate overview of the experimental procedure - which is carried out on the implemented VR setup - compared to the Performance evaluation section of the paper.

A. Approach

The performance evaluation be will planned and performed according to the approach listed below.

1) *Definition of requirements*: identification of functionalities that the system is expected to perform.
2) *Identification*: identification of functionalities of the system that perform poorly or correctly through experiments.
3) *Root Cause Analysis (RCA)*: enumerate the possible root causes of the identified poorly performing functionalities.
4) *Functional testing*: evaluating the above-mentioned root causes by functional testing.
5) *Functional testing*: by using functional testing (also referred to as experiments), every possible root cause of the RCA will be evaluated. This can either be unit testing, integration testing, or validation testing. Each functional test is designed as follows:
   - *Experiment description:* Overview of the experiment and split up into the following sub-descriptions;
     - *Goal:* Overall goal based on the possible root cause which is set out to be answered in the experiment.
     - *Assumption/hypothesis:* Assumptions that are made in the previously described goal. This can also be a hypothesis for the experiment.
     - *Setup:* Explanation of the setup for the experiment.
     - *Input:* Specification of the data which is used as an input for the system setup.
     - *Tested parameters:* Explanation of the parameters which will be examined in the experiment. These parameters are shown later on in the results section of the experiment.
     - *Question:* A question will be posed here which hopefully can be answered by using the results of the experiment.
   - *Implementation:* Explanation of possible implementation that was needed in order to be able to perform the experiment.
   - *Results:* Show the results of the tested parameters from the experiment description and analyze them.
   - *Conclusion:* Draw conclusions from the results and review the goal and question which were proposed in the experiment description.

B. Evaluation

The approach from section -A is explained in more detail in the following subsections.

1) Definition of requirements: Figure 1 gives an overview of the implemented system. The grey area indicates which software is (partly) implemented within ROS. The components within the black 'Remote' box are implemented on the Twente humanoid head platform and the 'Local' components on a regular PC.

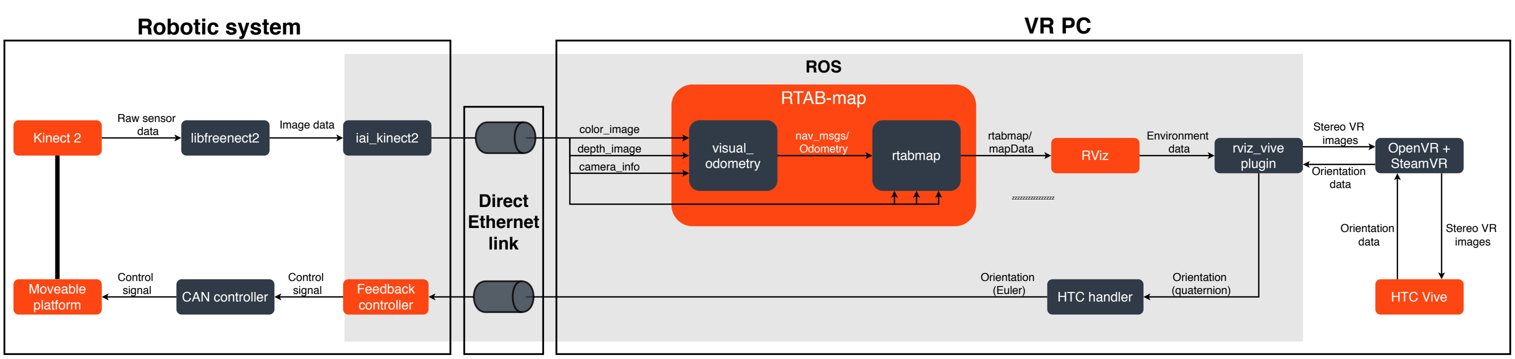


Fig. 1: System overview of the implemented system.

The system needs to include the following functionality as baseline performance:

1) Image acquisition from the Kinect 2 and imported into ROS using libfreenect2 and iai_kinect2. Image data needs to be published at a sufficient rate on ROS topics such that the RTAB-map node can run effectively. This rate needs to be at least as high as the update rate of the environment (7 Hz, see functionality 3). Three topics that are used are the color image, depth image, and camera info.
2) RTAB-map needs to provide two functionalities:
   a) Publish odometry data based on its visual odometry algorithm using the image, depth and camera info data from the Kinect 2.

b) Create a point cloud map of the environment based on the image, depth, and camera info data from the Kinect 2 and the odometry information. The map needs to be able to add new information as well as updating dynamic objects which are captured by the sensors of the Kinect 2. The point cloud map is published on the mapData topic.

3) The data from the mapData topic needs to be imported into RViz. This needs to happen at a sufficient rate as well such that the VR environment matches the actual environment as closely as possible. This implies that the dynamic object needs to be able to be tracked in such a way that the movement appears fluent to the user. Busch et al. found that humans have a so-called 'detection performance' which occurs around 7 Hz by studying EEG signals [1]. This finding suggests that humans are processing incoming visual information at 7 frames per second effectively. Therefore, a 7 Hz minimum is taken as a desired threshold for the mapData output topic frequency.
4) The VR environment from RViz needs to be converted into two images which can be projected onto the left and right eye of the user by using the HTC Vive. To determine which part of the VR environment needs to be displayed, orientation data from the HTC Vive needs to be obtained.
5) The obtained orientation data from HTC Vive, which are quaternions, needs to be converted into Euler angles such that it can be used by the controller of the Twente humanoid head platform.
6) Control the Twente humanoid head platform using the Euler angle data. The orientation of the Kinect 2 must follow the motion of the HTC Vive. The user will then be able to see new point cloud information from the part of the environment at which he/she is looking after it was incorporated in an update.

2) Identification: From the definition of requirements presented in the previous section, functionality 2b) and 3) perform insufficiently and are explained in further detail below. The remaining functionalities perform according to the specifications.

Latency

There is significant latency present in the system and this seems to be related to computational power. To the user, this becomes apparent by a low update rate of the VR environment. The cause of this latency needs to be investigated in the functional tests.

Dynamic objects

Possibly related to the 'latency' issue is the notion of poor tracking of dynamic objects. If there is a moving object in the remote environment, the current implementation is not able to correctly update the VR environment. This becomes apparent by multiple versions of the same object in the VR environment. By using the functional tests the cause of this performance issue needs to be found.

However, it is observed that when the Kinect is not moving, correct dynamic object tracking is achieved nonetheless. This suggests that motion of the Kinect induces a decreased tracking performance.

3) Root Cause Analysis: The next step is to analyse the identified system functionalities which perform poorly from the previous section and find the root cause. The root cause analysis in this section will elaborate on the possible root causes. Based on the implemented subsystems as specified in Figure 14, the possible root causes are categorized. Figure 2 gives an overview of the RCA in a structured tree format. To examine every root cause, several experiments (functional tests) are proposed. The goal, setup, results and conclusions from these experiments are listed in the next section.

Latency

- *Communication channel:*
  - A prime candidate for a latency problem is the communication channel between both PC's in the implemented setup. Through the Ethernet connection, the Kinect image data is sent to the local PC to be used by the RTAB-map algorithm. An assumption is made here that a higher input rate of image data into the RTAB-map algorithm will result in a higher output rate. In *experiment 1* a dual and a single PC setup will be used; with and without the Ethernet connection. In this way, the impact of the Ethernet connection on the latency issue can be investigated.
- *Low visual odometry update rate:*
  - The visual odometry node is a direct input to the RTAB-map algorithm node in ROS. It is assumed here that a low odometry quality might influence the output speed of the RTAB-map node. *Experiment 2* is set up to investigate a possible correlation between a low visual odometry quality and the output frequency of the RTAB-map algorithm. If this is the case, better visual odometry might be required.
- *Configuration RTAB-map:*
  - RTAB-map itself might limit the performance of the output topic frequency. There is a significant number of parameters which can be changed to change the behavior of the RTAB-map algorithm [1]. *Experiment 3* will take a closer look at

[1] https://github.com/introlab/rtabmap/blob/master/corelib/include/rtabmap/core/Parameters.h

these possible parameters which can be changed and the effect in terms of the output topic rate.

Dynamic objects

- *Visual odometry node:*
  - Low visual odometry quality might have a significant impact on poor tracking of dynamic objects that are present in the system. To investigate the impact of the visual odometry node, *experiment 4* is executed. The possible faulty output from the odometry is minimized by eliminating the movement of the Twente humanoid head platform.
  - To further investigate the visual odometry *experiment 5* is performed. Now, the quality parameter of the rtabmap node will be used to investigate a possible correlation with the poor dynamic object tracking performance.
  - As mentioned in the RTAB-map description by Labbé et al., visual odometry can be considered a bottleneck. Visual odometry can provide accurate estimates when sufficient keypoint features can be detected from the sensory image data. However, when not enough features are found, the visual odometry quality degrades. As a result, a more robust estimate from the visual odometry is desired. To achieve this, proprioceptive sensors such as an IMU can be added to the system to increase the quality of the odometry [2]. *Experiment 7* will do exactly this and will incorporate an IMU sensor into the implementation.
- *Kinect 2:*
  - In terms of hardware, the performance of the Kinect 2 sensor might be lacking performance which is required to increase dynamic object tracking capabilities. An increase in the number of (depth) pixels, range, and frame rate might provide the RTAB-map the sensory information it needs to perform adequately. Starting March 2020 a successor to the Kinect 2 is released, the Azure Kinect. A comparison between the two sensors is given in Table I. It is clear that the Azure Kinect outperforms the Kinect 2 in terms of color and depth image resolution. However, the maximum depth range is significantly decreased in the new model and the maximum frame rate cannot be increased. A welcome addition is the IMU capability of the Azure Kinect, which can produce orientation data at 208 Hz [2]. However, increased performance by implementing a improved sensor such as the Azure Kinect is unlikely since dynamic objects can be tracked adequately when the Kinect is not moving - indicating that the performance of the Kinect 2 is sufficient.
  - The HTC handler subsystem is responsible for producing the appropriate control signal to be used by the feedback controller of the Twente humanoid head platform on which the Kinect is mounted. High-frequency motion induced by the control signal might have an impact on the tracking performance of dynamic objects since images from the Kinect can become distorted. To investigate this high frequent motion, *experiment 6* is conducted and the aforementioned motion is reduced as much as possible by using a predefined control signal for the platform. Additionally, the Kinect is securely mounted as well by using a 3D printed bracket, and the proportional gains of the feedback controller are reduced to slow down the movement of the platform.
- *rtabmap node:*
  The possible root causes listed below are at the core of the RTAB-map algorithm. Even though one of these aspects might be the actual root cause, they are beyond the scope of the work presented in this document.
  - *Configuration RTAB-map* - The RTAB-map core itself might limit the performance of the dynamic object tracking due to the chosen parameter settings. Several settings were attempted already to determine if they would improve the dynamic object tracking capabilities. Examples include changing the odometry strategy from F2M (frame to map) to F2F (frame to frame), changing the feature matching approach, and changing the linear/angular update rates [2]. Although the map quality did improve slightly in some cases, it did not result in improved tracking capabilities. Considering that the configuration is working correctly when the Kinect is not moving, it is unlikely that this would be the root cause of dynamic object tracking issues.
  - *Point classification* - The visual odometry of RTAB-map uses RANSAC to determine the ego-motion of the Kinect sensor [2]. Data points that belong to dynamic objects are classified as outliers by the RANSAC algorithm [3]. When the Kinect is not moving, most captured data points are classified as non-moving. As a result, the Kinect camera motion estimate is zero and dynamic points can be easily detected and updated by the core of the RTAB-map algorithm. When the Kinect is moving, dynamic objects can interfere with this process. In that case, it is difficult to differentiate between the movement of dynamic objects and the ego-motion of the Kinect when using the RANSAC approach [3]. In other words, key points that are detected on a dynamic object can mistakenly be used in the ego-motion estimation. It is non-trivial to make this discrepancy between moving and non-moving points in the core of the algorithm and this can make it difficult for the core of the RTAB-map algorithm to track dynamic objects when the camera is moving.

  Other subsystems of the RTAB-map algorithm such as the memory management approach was also considered as a possible root cause. However, it was found that the long term memory management component is disabled in the source

[2]https://docs.microsoft.com/en-gb/azure/Kinect-dk/hardware-specification

code by default by the parameters 'MemoryThr' and 'TimeThr' [3]. Therefore, this subsystem of the algorithm cannot contribute to the dynamic object tracking issue.

TABLE I: Specification comparison between the Kinect 2 and Azure Kinect

	Max. RGB image resolution	Max. depth image resolution	Max. depth range	Max. frame rate	IMU
Kinect 2	1920x1080	512x424	4.5m	30 fps	No
Azure Kinect	4096x3072	640x576	3.86m	30 fps	Yes, 208 Hz

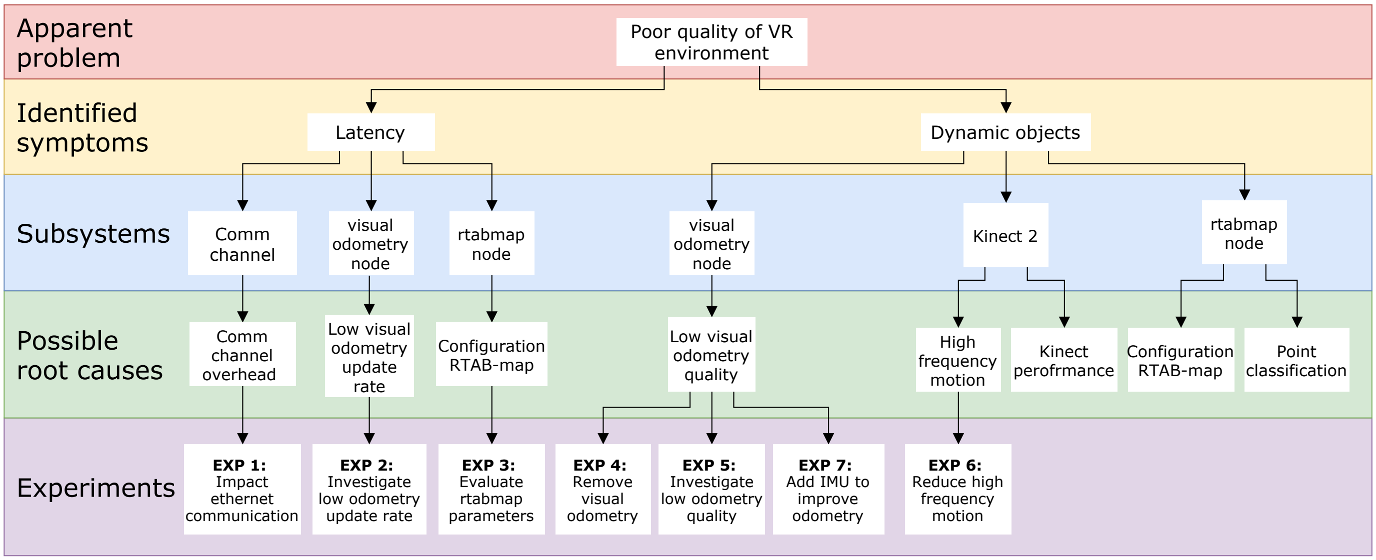


Fig. 2: RCA overview in a tree structured format.

4) Functional testing: The goal of the functional tests is to systematically investigate the previously presented possible root causes for the performance issues in further detail.

The **main questions** that need to be answered are as follows:
1) What is the cause of the slow update rate (latency) of the rtabmap node?
2) What is the cause of improper tracking of dynamic objects?

Experiment 1

- *Goal:* Investigate the influence of the Ethernet connection between the remote and local PC on the input and output topic speeds of the RTAB-map algorithm.
- *Assumption:* The data communication overhead between the remote and local PC causes reduced topic speeds of the Kinect, visual odometry, and output (mapData) topics. Furthermore, it is also assumed that a higher input rate (Kinect images and visual odometry) will result in a higher output rate (mapData) of the RTAB-map algorithm.
- *Setup:* Two setups will be used: a single and dual PC setup. The single PC setup entails that all the hardware and software is interfaced on one PC. In the dual PC setup, the setup is similar to the actual implementation where the Kinect images are obtained on the remote PC and transferred over the Ethernet connection to the local PC where the RTAB-map algorithm is located. Only the visual chain will be used i.e. the Twente humanoid head platform will not be used.
- *Input:* Kinect 2 image data. This can either be SD(512x424), qHD(960x540) or HD(1920x1080).
- *Tested parameters:* ROS topic frequency of the Kinect images (color_image, depth_image, and camera_info), frequency of the visual odometry node (rtabmap/odom) and the output frequency of the resulting map in RViz (rtabmap/mapData)
- *Question:* To what extent does a single PC setup influence the input topic speeds of the RTAB-map algorithm compared to the implemented dual PC setup? This may have an impact on the update rate of the mapData output topic.

[3]https://github.com/introlab/rtabmap/blob/master/corelib/include/rtabmap/core/Parameters.h

Results
The results are shown in Figure II and III. From the dual PC setup, it is clear that the Kinect topic data is limited to 6 Hz when the lowest quality images are used. The odometry topic output from the RTAB-map node publishes at a similar rate as the Kinect topic. The mapData output topic does not exceed the 1 Hz rate of publishing.
In a single PC setup, the Kinect topic and Odometry topic output data do increase significantly. However, the mapData output is still limited to 1 Hz.

TABLE II: Experiment 1 with a *dual PC* setup

Quality	Kinect ROS topic	Odometry ROS topic	mapData output ROS topic
SD	5.9 Hz	4.4 Hz	0.9 Hz
qHD	1.5 Hz	1.8 Hz	0.9 Hz
HD	0.4 Hz	0.4 Hz	0.5 Hz

TABLE III: Experiment 1 with a *single PC* setup

Quality	Kinect ROS topic	Odometry ROS topic	mapData output ROS topic
SD	30.0 Hz	9.4 Hz	1.0 Hz
qHD	8.1 Hz	8.6 Hz	1.0 Hz
HD	2.4 Hz	2.4 Hz	0.8 Hz

Conclusion
In this experiment, an increase in topic speed was found in the Kinect and odometry data for the single PC setup. This can be explained by overhead in the Ethernet communication between the remote and local PC in the dual PC setup. In the single PC setup, there is no Ethernet transfer needed obviously and thus an increased topic speed can be achieved. Furthermore, no improvement was found in terms of the output topic speed of the RTAB-map algorithm.

Experiment 2

- *Goal:* Investigate the correlation between odometry quality and low update rate of the rtabmap node.
- *Assumption:* An insufficient odometry quality (or even lost odometry) influences the performance of the RTAB-map node and thus the output speed of it.
- *Setup:* The single PC setup will be used in this experiment as well as the Twente humanoid head platform. No dynamic objects are presented in the remote environment.
- *Input:* Kinect 2 image data. This can either be SD(512x424), qHD(960x540) or HD(1920x1080).
- *Tested parameters:* The output frequency of the resulting map in RViz (rtabmap/mapData) will be compared for different image qualities. The /rtabmap/odom_info topic reports the quality of the visual odometry according to the documentation of RTAB-map [4]. The quality is based on the number of inliers that the algorithm was able to find between successive images from the Kinect i.e. the number of key points that can be found in both images.
- *Question:* To what extent does a loss in odometry quality correlate with a lower update rate of the output map? This might answer the question of what causes the slow update rate of the rtabmap node.

Results
Figure 3 shows the quality of the visual odometry overtime for SD, qHD, and HD image qualities of the Kinect. The number of inliers indicates the quality of the visual odometry. For the SD and qHD image qualities, it can be observed that the mapData update instances (dashed vertical lines) retain a constant rate of approximately 1 Hz. Therefore, no correspondence can be found between a decrease in the odometry quality and a decreased mapData update rate.
The HD image quality subplot does show a decreased mapData update rate. However, there does not seem to be a correlation with the odometry quality. For example, the mapData update rate decreases after 2 seconds and the number of inliers drops as well. Still, around 8 seconds, the mapData update rate increases again. This reduced update rate might be explained by a lower frame rate of the Kinect at higher image qualities as seen in the first experiment.

Conclusion
In this experiment, no correlation was found between a loss in odometry quality and a lower update rate of the output mapData. More inliers between successive frames do not contribute to a higher frame rate.

[4]http://docs.ros.org/api/rtabmap_ros/html/msg/OdomInfo.html

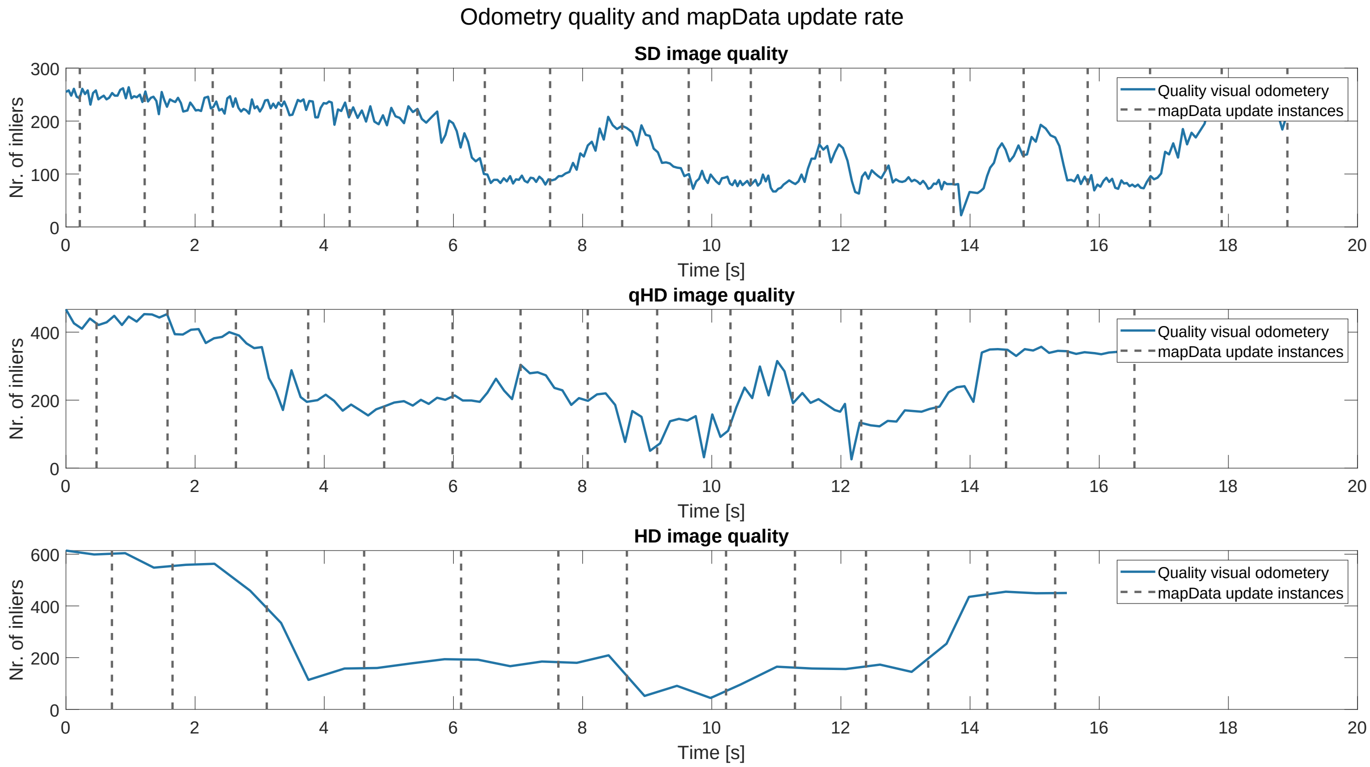


Fig. 3: The quality of the visual odometry over time for SD, qHD and HD image qualities of the Kinect. A new map update are indicated as dashed vertical lines.

Experiment 3

- *Goal:* A parameter performance test will be attempted in this experiment to increase thw publishing rate of the rtabmap node.
- *Hypothesis:* Specific parameters of the rtabmap node influence the output speed performance of the node itself.
- *Setup:* A single PC setup will be used for maximum performance (based on the outcome of experiment 1). There is no focus on dynamic objects in this experiment.
- *Input:* Kinect 2 image data. This can either be SD(512x424), qHD(960x540) or HD(1920x1080).
- *Tested parameters:* There are several parameters of the RTAB-map algorithm which are still untested and which might impact the performance according to the wiki [5]. Parameters that will be investigated are: Rtabmap/DetectionRate, queue_size [6], Grid/CellSize, Vis/MaxFeatures [7].
- *Question:* What is the influence of specific parameter settings of the RTAB-map algorithm on the performance of the system regarding the update rate? Changing the settings may have an impact on the update rate of the rtabmap node.

Results

From the attempted ”Tested parameters” from above, only the Rtabmap/DetectionRate had a significant impact on the output topic speed. The output topic rates for detection rate of 5, 10 and 20 are listed below in Tables IV, V and VI respectively. In all the different cases for the detection rate, it can be observed that the output topic for the mapData does not exceed that of the odometry node. This is logical behavior since the rtabmap algorithm is dependent on the odometry information. A detection rate above 20 did not yield a significant increase in the mapData output topic speed.

TABLE IV: Experiment 3 with Detection rate = 5

Quality	Kinect ROS topic	Odometry ROS topic	mapData output ROS topic
SD	30.0 Hz	9.2 Hz	4.0 Hz
qHD	7.6 Hz	7.8 Hz	3.8 Hz
HD	2.2 Hz	2.2 Hz	2.2 Hz

[5]http://wiki.ros.org/rtabmap_ros
[6]https://github.com/introlab/rtabmap_ros/issues/358
[7]http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning

TABLE V: Experiment 3 with Detection rate = 10

Quality	Kinect ROS topic	Odometry ROS topic	mapData output ROS topic
SD	30.0 Hz	9.0 Hz	6.0 Hz
qHD	6.5 Hz	6.8 Hz	6.5 Hz
HD	2.1 Hz	2.2 Hz	2.2 Hz

TABLE VI: Experiment 3 with Detection rate = 20

Quality	Kinect ROS topic	Odometry ROS topic	mapData output ROS topic
SD	30.3 Hz	8.6 Hz	8.0 Hz
qHD	6.7 Hz	7.0 Hz	7.0 Hz
HD	2.1 Hz	2.3 Hz	2.2 Hz

Conclusion

An increased rtabmap/DetectionRate parameter causes an increase in the output rate of the mapData which is desired. Therefore, the root cause of the latency issue is found. This output rate is limited by the rate of the odometry node. A detection rate above 20 does not yield a significant increase in the output rate of the mapData. The baseline performance of 7 Hz is achieved when SD or qHD quality is used.

Experiment 4

- *Goal:* Investigate the dynamic object tracking issue by eliminating the possible negative effects of the visual odometry node of RTAB-map.
- *Assumption:* Dynamic object tracking issues are caused by inaccuracy of the visual odometry node.
- *Setup:* The two PC setup will be used in this experiment. However, the Twente humanoid head platform will not be controlled and thus left idle. This means that the Kinect does not move during this experiment. Dynamic objects are presented in the remote environment. This will be achieved by the movement of a human arm.
- *Input:* Kinect 2 image data. This can either be SD(512x424), qHD(960x540) or HD(1920x1080).
- *Tested parameters:* Visual inspection of the generated point cloud in RViz. The focus lies on the dynamic objects in the remote environment.
- *Question:* Which part of the movement causes issues with dynamic object tracking? By answering this question, it can be determined which part/element of the system causes the improper tracking phenomena.

Results

Figure 4 shows snapshots from the virtual environment were qHD image quality was used. On the bottom of the left image, an arm can be seen which moved in front of the Kinect. The right image shows the virtual environment after the movement of the arm. It can be seen that all the point from the map which corresponds to the arm are removed properly when the arm was not visible anymore to the Kinect. The movement itself was correctly captured as well. The same resulting behavior of the virtual environment was observed for the SD and HD image qualities.

(a) Snapshot of the environment during dynamic object movement.

(b) Snapshot of the environment after dynamic object movement.

Fig. 4: Snapshots from the experiment where a dynamic object was presented.

Conclusion

When the movement of the platform, and thus the movement of the Kinect itself, was excluded it was shown that the resulting virtual environment correctly updates dynamic objects which are presented in front of the Kinect. Furthermore, this experiment showed that the quality of the Kinect images does not affect the dynamic object tracking performance.

Experiment 5

- *Goal:* Investigate the correlation between the odometry quality parameter of the rtabmap node and the performance of the RTAB-map algorithm in terms of object tracking.
- *Assumption:* The behavior of the odometry quality parameter can be correlated to the dynamic objects which are captured by the Kinect 2. That is, a higher value of the visual odometry parameter indicated a better dynamic object tracking performance.
- *Setup:* Full dual PC setup will be used with presented dynamic objects.
- *Input:* Kinect 2 image data; qHD(960x540) quality.
- *Tested parameters:* The rtabmap/odom_info/inliers parameter will be recorded. This gives information on the quality of the odometry because it indicates the number of inliers that were found between the last two input images coming from the Kinect 2. More inliers result in a better estimate of the odometry [8].
- *Question:* To what extent does the odometry quality parameter from the RTAB-map node indicate poor dynamic object tracking? This might be the explanation that is sought after by the second main question.

Results

Figure 5 shows the visual odometry quality data from this experiment over time. The data is split up into 4 regions and its corresponding testing conditions are explained in Table VII.

TABLE VII: Sections of the experiment and its corresponding testing conditions

Region	Kinect moving?	Dynamic object present?
1	No	No
2	No	Yes
3	Yes	No
4	Yes	Yes

From Figure 5 it can be observed that the odometry quality is relatively high in the first two regions where the Kinect is not moving. In the third region, the quality decreases significantly due to the movement of the Kinect. In this third region, no dynamic object is presented. For the fourth region, the dynamic object is presented while the Kinect is moving. Important to note here is that the average visual odometry shows a slight increase. This might be explained by the additional key points which are introduced by the dynamic object and which are therefore detected by the RTAB-map algorithm. The dynamic object tracking in this fourth region still performs poorly. In the first and second regions, inverse behavior compared to the third and fourth regions can be observed. Namely, in the second region, a slight decrease in the odometry quality is observed when the dynamic object is presented.

Conclusion

Based on the data of the experiment, no correlation could be found between the visual odometry quality parameter and possible poor dynamic object tracking. A higher value of the odometry parameter does not indicate a better dynamic object tracking performance.

[8]http://docs.ros.org/api/rtabmap_ros/html/msg/OdomInfo.html

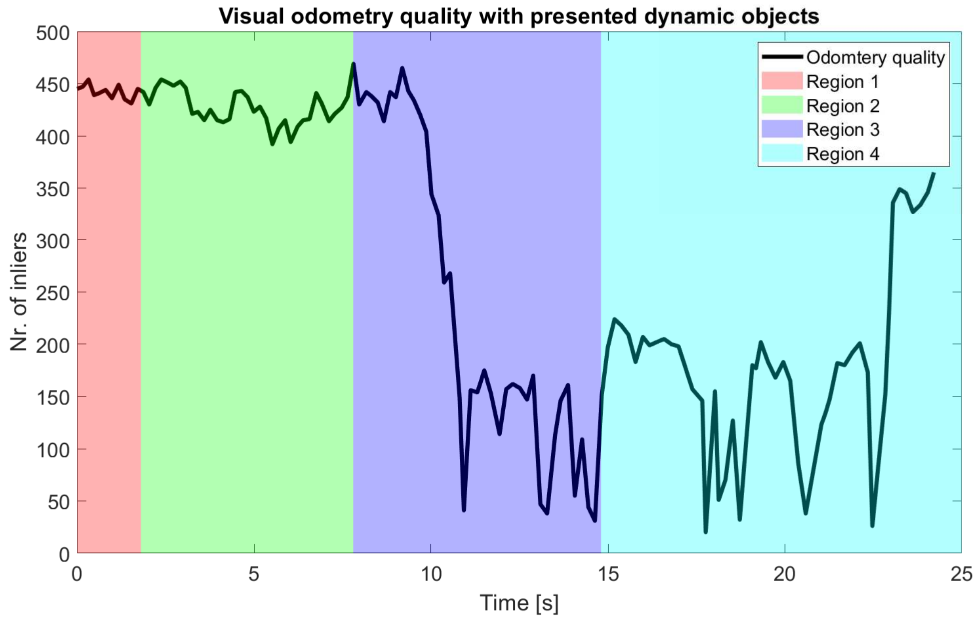


Fig. 5: Visual odometry quality over time. See Table VII for the meaning of each region.

Experiment 6

- *Goal:* Investigate the impact of high frequent movement of the Twente humanoid head platform on the dynamic object tracking performance.
- *Assumption:* High frequent motion has a negative impact on the performance of the visual odometry which may cause reduced object tracking capabilities.
- *Setup:* The dual PC setup is used. Experiment 4 showed that the quality of the Kinect images does not affect the object tracking performance. Therefore, as a trade-off between quality and speed, qHD quality is used. Furthermore, dynamic objects are included. A bracket is included to reduce the flexibility in the Kinect 2 mount. The proportional gain of the feedback controller of the Twente humanoid head platform is changed such that response is slow compared to the given setpoint input. The HTC Vive will be disconnected from the setup and a predefined control signal will be used as an input for the platform to eliminate possible high frequent changes present in the control signal originating from the HTC Vive orientation data.
- *Input:* The platform will only be controlled along the pitch axis. In the first 10 seconds, the platform is left idle. After that, the platform moves up by 0.7 radians and stays there for 10 seconds. Finally, the platform returns to the initial position in the last 10 seconds. Therefore, the duration of the whole cycle is 30 seconds.
  In the first 10 seconds, a static hand is presented and is removed if the platform is moved up by 0.7 radians. If the platform moves to the initial position again in the final 30 seconds of the experiment, the Kinect does not register the static hand anymore and should thus be removed by the rtabmap node.
- *Tested parameters:* The odometry output will be recorded (rtabmap/odom). The pose and twist will be extracted. The pose gives information on the pitch angle. From the twist, the velocity can be determined by taking the Euclidian norm of all the components of the angular velocity vector. Visual inspection of the generated point cloud in RViz will be done. The focus lies on the dynamic objects in the remote environment.
- *Question:* To what extent does high-frequency motion in the setup affect the dynamic object tracking capabilities? Does a predefined path for the Twente humanoid head platform have a significant impact?

Setup implementation
First, to reduce the flexibility in the Kinect 2 mount a 3D printed bracket was designed to prevent pitch movement of the Kinect. The installed white bracket can be seen in Figure 6.

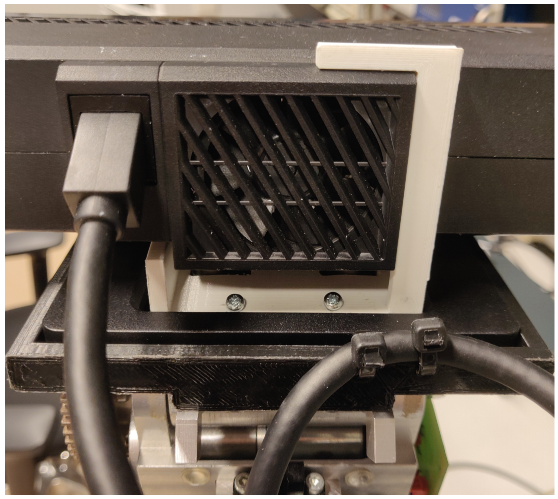


Fig. 6: This figure shows the 3D printed bracket which was installed to prevent the pitch movement of the Kinect taking place.

Additionally, the proportional gain of the feedback controller was reduced to slow down the movement of the Twente humanoid head platform on which the Kinect is mounted. Figure 7 shows two different evaluated platform velocities. In Figure 7a, the proportional gains were reduced to 0.5, 0.5, and 1 for the pitch, yaw, and roll angles respectively. In Figure 7b, the proportional gains were set to the default values of 2, 2, and 3 for the pitch, yaw, and roll angles respectively. Please refer to the implementation explanation of Van Der Zee for the exact implementation details of the feedback controller [4].

(a) Snapshot of the virtual environment after slow Kinect movement with a presented dynamic object.

(b) Snapshot of the virtual environment after fast (default) Kinect movement with a presented dynamic object.

Fig. 7: Snapshot of the virtual environment after slow and fast Kinect movement with a presented dynamic object.

From Figure 7 it is clear that slow movement does reduce the number of artifacts and streaks present in the virtual environment. However, the moving human arms which were presented in both cases were not properly removed when the arm was not present anymore. Therefore, the problem of the dynamic object tracking still persisted even when the bracket for the Kinect and slow movement of the Twente humanoid head platform was implemented.

Results

Next, the results where the predefined path for the HTC Vive was implemented are discussed. Figure 8 shows the orientation in Euler angles, linear velocity, and angular velocity based on the odometry information from the RTAB-map algorithm. Data was obtained by using the rosbag functionality [9] from the /rtabmap/odom topic which publishes an odometry message type [10]. During the experiment, the Kinect was rotated counter-clockwise around the y-axis (pitch).

Several observations can be made. First of all, there is not always odometry information available in the first 10 seconds of the experiment. This becomes apparent by the discontinuities in the graph. Also, the pitch estimate does not return to the initial position in the final seconds of the experiment. Instead, an offset is present of 0.03 rad. Next, there appears to be noise present in the linear velocity data. Namely, in the first 10 seconds, the platform is not moving at all but the linear velocity does not remain constant in this time frame. Finally, it can be observed that the linear and angular velocities start to deviate from 0 at 10 seconds. However, the pitch orientation is already at -0.5 rad at that time instance. Therefore, the linear and angular velocities seem to lag the orientation data.

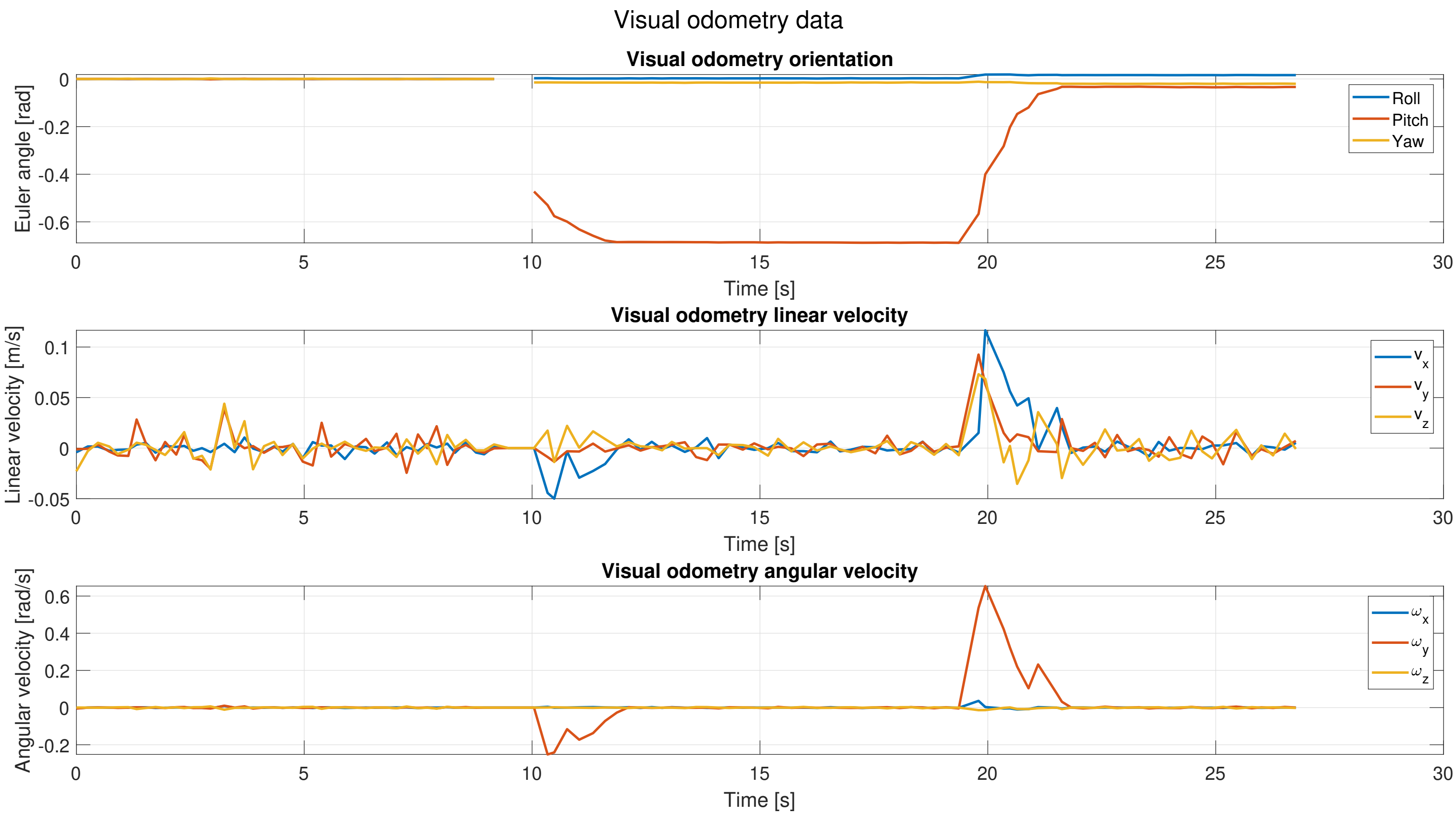


Fig. 8: These figures shows the Euler angle orientations, linear velocity and angular velocity of the Kinect according to the visual odometry data.

Figure 9 shows snapshots of the virtual environment at the start and end of the experiment. The red circles highlight the static object (hand) presented in the environment. The physical hand was not present in the situation of Figure 9b. However, it was not removed from the virtual environment.

[9] http://wiki.ros.org/rosbag
[10] http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html

(a) Environment before movement

(b) Environment after movement

Fig. 9: Snapshots from the start and end of the experiment.

Conclusion

The designed 3D bracket and feedback controller adaptations did not contribute to a reduced number of artifacts and streaks present in the virtual environment. Also, the predefined path for the Twente humanoid head platform does not improve the performance of the rtabmap node in terms of object tracking. Furthermore, it was found that the quality of the odometry was insufficient.

Experiment 7: IMU

- *Goal:* As seen in the previous experiment, there are problems regarding the odometry quality. The goal of this is experiment is to investigate if the addition of an IMU sensor will improve the odometry quality and with that the quality of the VR environment regarding dynamic object tracking.
- *Assumption:* Improved odometry will increase the likelihood that key points of dynamic objects are matched.
- *Setup:* Only the Kinect and an Xsens IMU will be used. Exact details on the implementation are listed in the subsection below.
- *Input:* Kinect 2 image data; qHD(960x540) quality.
- *Tested parameters:* Several parameters will be investigated. Especially the odometry quality of the IMU odometry compared to the visual odometry will be shown. Also, the quality of the VR environment will be visually inspected.
- *Question:* To what extent does the integrated IMU improve the odometry quality? Furthermore, does an increase in odometry quality improve the dynamic object tracking capabilities?

IMU implementation

Figure 10 shows the setup for this experiment were the IMU is mounted on top of the Kinect. The available IMU that was used is the Xsens MTi-300-2A5G4. This IMU sensor integrates an accelerometer, gyroscope, and magnetometer and by using sensor fusion high-quality orientation data can be obtained [11]. For the implementation of this sensor, it is important to know the transform between the coordinate frames of the Kinect and IMU. The origin of the IMU frame is located 13mm behind the front of the sensor on the center line and 7mm up from bottom plate [12]. The origin of the Kinect base which is used by the RTAB-map algorithm is located at the RGB optical sensor [13]. These origin locations are also indicated in Figure 10. The transform between the two frames can be given by a pure translation expressed by Equation 1.

$$\begin{bmatrix} x \\ y \\ z \\ q_x \\ q_y \\ q_z \\ q_w \end{bmatrix} = \begin{bmatrix} -0.02 \\ -0.1 \\ 0.03 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{1}$$

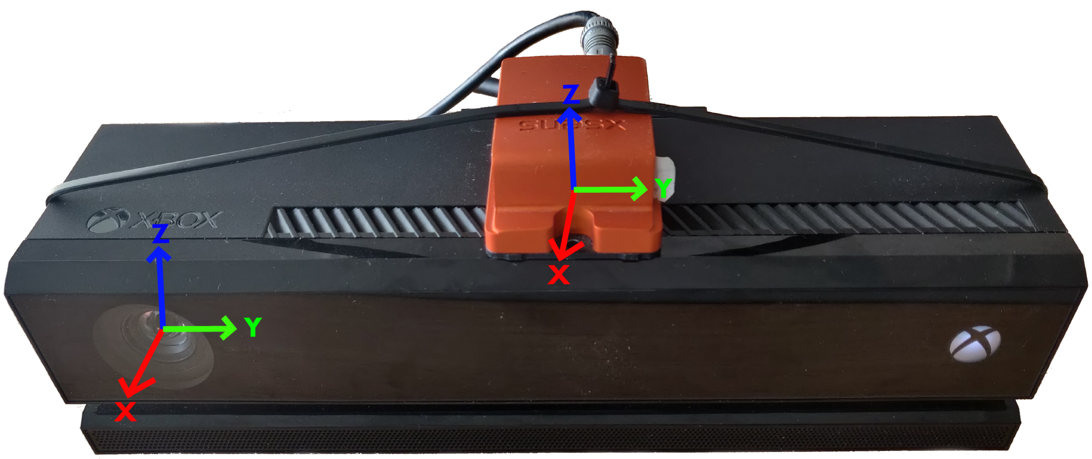


Fig. 10: Setup containing the Kinect and the IMU. A simple cable tie is used to mount the IMU securely on top of the Kinect. The two coordinate frames indicate the origins of the frame of the Kinect and IMU sensor.

The official ROS driver package for Xsens IMU's was used to obtain the IMU data from the sensor and to interface with ROS itself [14]. This allowed for easy integration with the RTAB-map algorithm which also runs on ROS. During initial testing, it was found that the chosen IMU suffered from drift in the z-angle (yaw). This phenomenon can be seen in Figure 11a. During this test the IMU was not moving, however, the yaw angle drifted more than 20 degrees in a 50 second period which is unusable.

It was found that the gyroscope suffered from a bias. This resulted in a constant angular velocity around the z-axis and the observed yaw angle drift over time. Luckily, Xsens provides a functionality which can perform a manual Gyro Bias Estimation [15]. During the estimation the sensor is stationary and therefore the angular velocity bias can be estimated. This functionality was unfortunately not implemented in the ROS package by Xsens. However, by using the Xsens Device API, the Gyro Bias estimation was implemented in the Xsens driver ROS node. After the IMU device was properly calibrated, the IMU orientation is much more consistent which can be seen in Figure 11b.

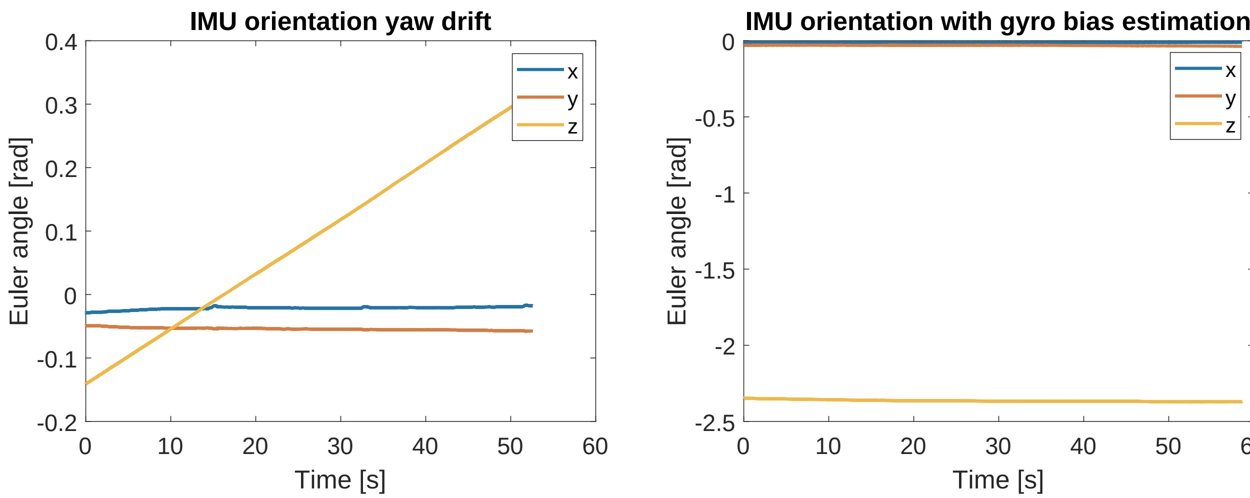
[11] https://www.xsens.com/hubfs/Downloads/Manuals/MTi_familyreference_manual.pdf

[12] https://base.xsens.com/hc/en-us/articles/202294831-MTi-10-100-series-Origin-of-Coordinate-system

[13] http://official-rtab-map-forum.67519.x6.nabble.com/tinkerforge-imu-td1886.html#a3690

[14] http://wiki.ros.org/xsens_mti_driver

[15] https://base.xsens.com/hc/en-us/articles/360002763354-Manual-Gyro-Bias-Estimation



(a) Sensor drift observed in the z-orientation (yaw angle) over time.

(b) IMU orientation after the gyro bias estimation.

Fig. 11: Orientation of the IMU in Euler angles, with and without drift in the z-orientation (yaw angle).

For integrating the sensory data from the IMU with the RTAB-map algorithm, the ROS package Robot Localization was used. Not only does it convert every sensory input to the correct output format (nav_msgs/Odometry), but it also allows for the fusion of several sensory inputs based on an Extended Kalman Filter (EKF) [16]. The translational transform between the sensory inputs is given by Equation 1 and is taken into consideration by the Robot Localization package when it is performing the sensor fusion. Any rotational offset between sensory inputs is automatically corrected for such that all the data is aligned with the base frame orientation [17]. The total transform tree, as seen in Figure 12, shows the necessary transforms between the frames which are present in the implemented system. The ’/ekf_localization’ transform is computed by the Robot Localization package. The remainder of the ’/tf’ transforms are given by the geometrical transforms between the locations of the sensors which do not change over time.

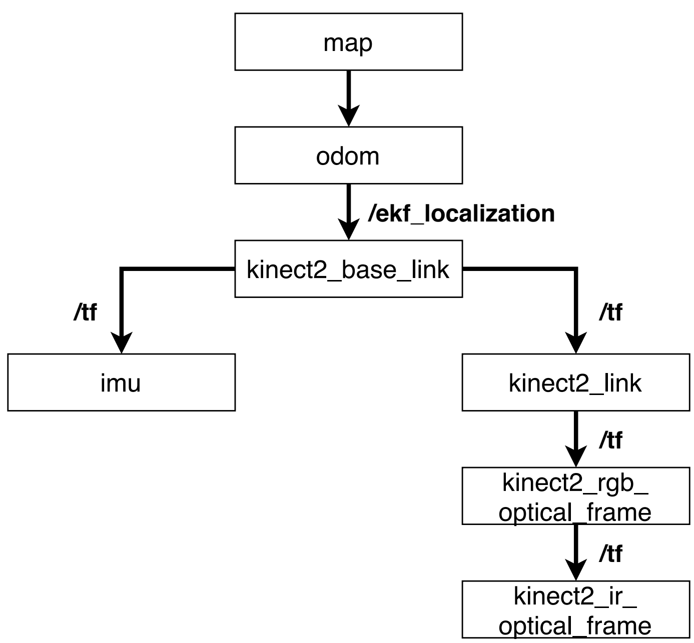


Fig. 12: All the transforms which are present in the implemented IMU + Kinect system.

[16]http://docs.ros.org/melodic/api/robot_localization/html/index.html
[17]http://docs.ros.org/melodic/api/robot_localization/html/preparing_sensor_data.html

In the first iteration of the design, only the sensory data from the IMU was used. Figure 13 shows the output produced by the Robot Localization package. Between the second 22 and 26, the Kinect and IMU were moved. However, since the linear acceleration data of the IMU was not used as an input, the linear velocity output is nonexistent. As a result, the virtual environment has severe shortcomings as can be seen in Figure 14.

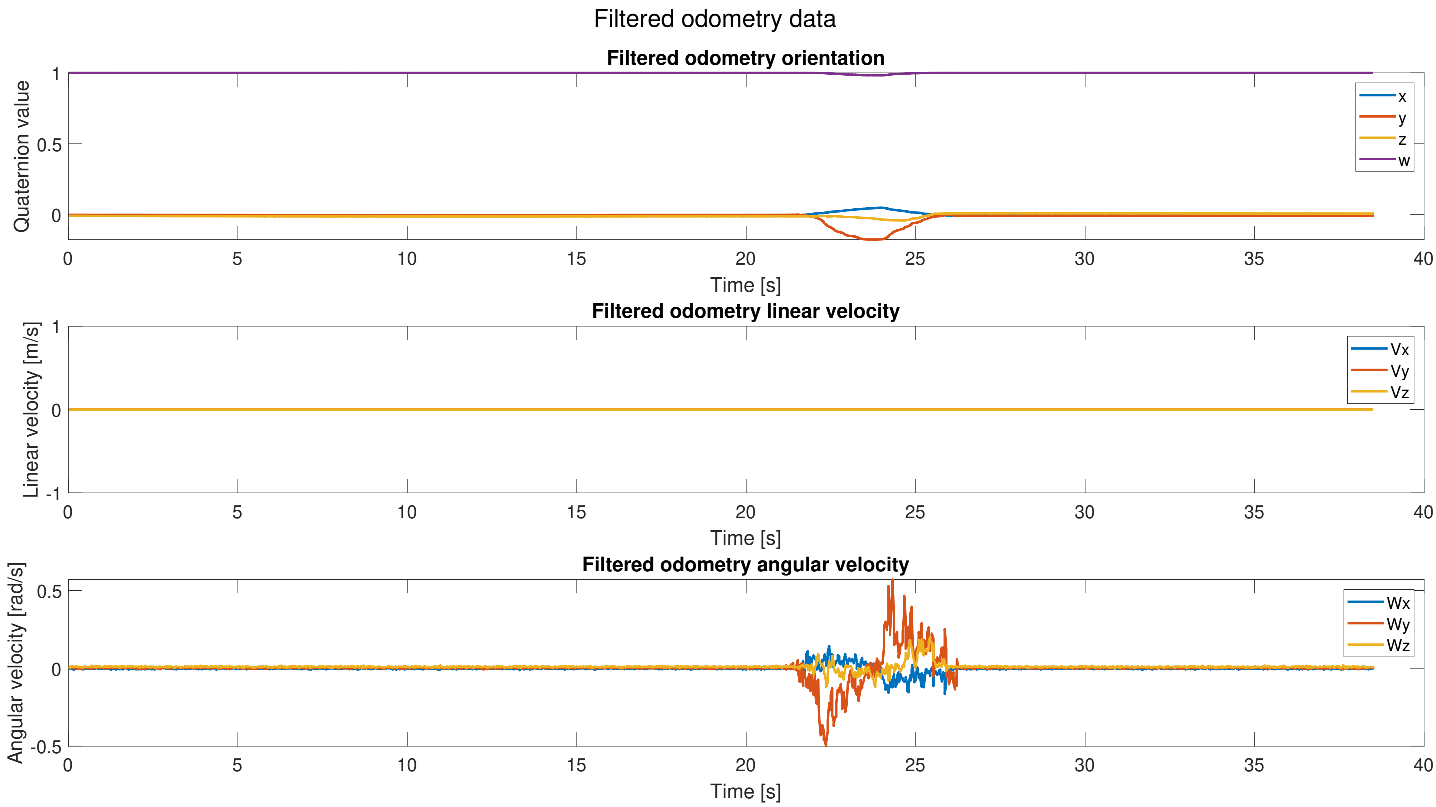


Fig. 13: Odometry data produced by the Robot localization package. Here, only the IMU data was used as an input.

(a) Dynamic object artifacts in the middle left of the virtual environment.

(b) Numerous streaks present in the virtual environment.

Fig. 14: The resulting virtual environment when only sensory data from the IMU is used.

In the second iteration, the linear acceleration data from the IMU was used as an input as well. The built-in gravity compensation of the Robot Localization was used as well [18]. The resulting odometry data can be seen in Figure 15. During this experiment, the IMU was not moving. However, due to the integration needed to produce the linear velocity from the linear acceleration data, significant drift occurs. This results in multiple copies of the mapData in the virtual environment as shown in Figure 16.

[18]http://docs.ros.org/melodic/api/robot_localization/html/preparing_sensor_data.html#imu

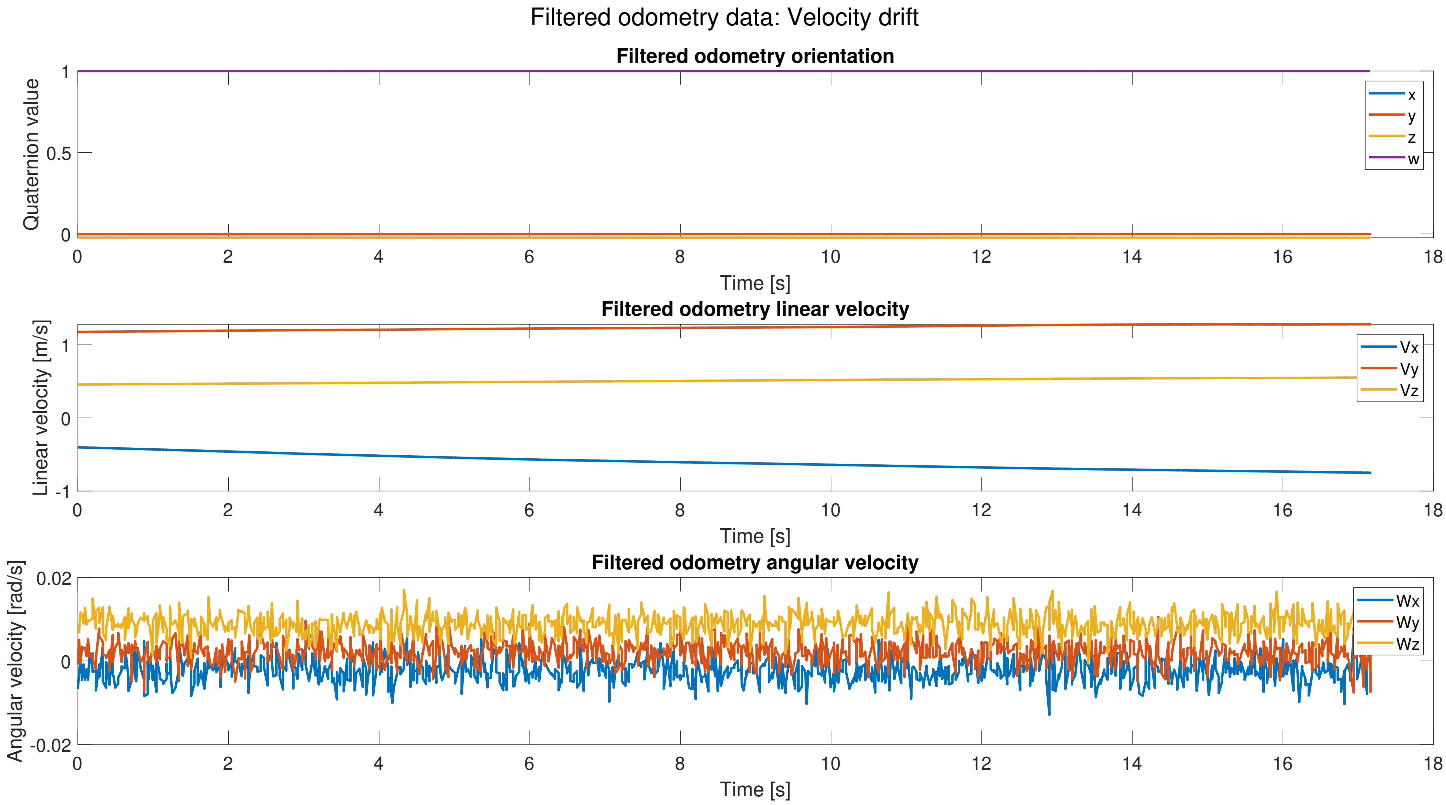


Fig. 15: Odometry data produced by the Robot localization package. Here, only the IMU data was used as an input including linear acceleration data. The Kinect and IMU were not moving during this time frame. A clear drift in the linear velocity can be observed.

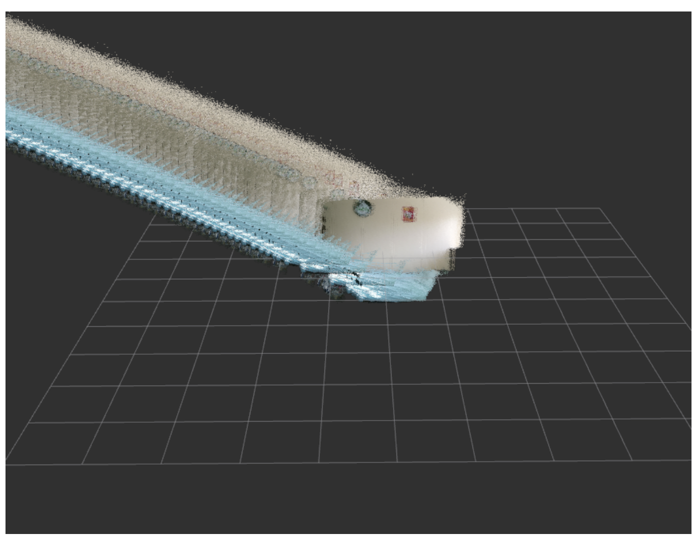


Fig. 16: The resulting virtual environment when only sensory data from the IMU is used including linear acceleration data. A streak of point cloud data can be observed.

In the final iteration, the full capacity of the EKF filter of the Robot Localization package was used. From the IMU, the orientation, angular velocity, and linear acceleration data were used. From the visual odometry node, the position and linear velocity were used as an input. The next section shows the final results.

Results

Figure 17 shows the resulting EKF filtered odometry based on the visual odometry data and IMU data. The recorded time frame can be split up into 4 sections. In the first 8 seconds, the Kinect was not moving and no dynamic objects were presented. In seconds 8 till 15, a human arm was presented as a dynamic object in front of the Kinect. In seconds 15 till 20, the Kinect was moved. In the remaining time frame, the same conditions apply as in the first 8 seconds.

First of all, there are several improvements which can be observed in Figure 17 compared to Figure 8. No more discontinuities are present in the orientation data. Also, due to the high update rate of the IMU, more data points are available for orientation and angular velocity data.

To compare the performance of the standalone visual odometry and the EKF filtered odometry from Figure 8 and 17, the mean and variance are computed according to Equation 2 and 3 respectively [19]. In these equations, the variable x_i represents the data points that are recorded during the first 8 seconds of both experiments when the Kinect was *not* moving.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2}$$

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2 \tag{3}$$

Table VIII gives an overview of the comparison between the standalone visual odometry and the fused EKF odometry when the Kinect is not moving. A decreased variance of the linear velocity can be observed when the fused EKF odometry is used instead of the standalone visual odometry. The remaining variances are mostly in the same order of magnitude when comparing both odometry approaches.

TABLE VIII: Comparison of the standalone visual odometry and the fused EKF odometry when no movement is performed.

Visual odometry						
	Mean X	*Mean Y*	*Mean Z*	*Variance X*	*Variance Y*	*Variance Z*
Orientation (Euler)	7.8395e-05	-5.7218e-05	6.8683e-05	5.4688e-08	1.4707e-07	2.7692e-07
Linear velocity	-2.6561e-04	4.5072e-04	2.0515e-04	1.9888e-05	1.5443e-04	1.3152e-04
Angular velocity	-7.6809e-05	7.0734e-05	-1.5120e-04	3.3216e-06	7.5997e-06	1.0738e-05

Fused EKF odometery (Visual odometry + IMU data)						
	Mean X	*Mean Y*	*Mean Z*	*Variance X*	*Variance Y*	*Variance Z*
Orientation (Euler)	0.0016	-6.0922e-04	-0.0066	2.4355e-08	3.8918e-07	1.2351e-05
Linear velocity	3.1650e-05	1.6303e-04	-2.0168e-04	1.5663e-05	6.2316e-05	3.4447e-05
Angular velocity	-0.0018	0.0019	0.0090	9.6103e-06	9.8261e-06	7.8788e-06

[19]https://en.wikipedia.org/wiki/Variance

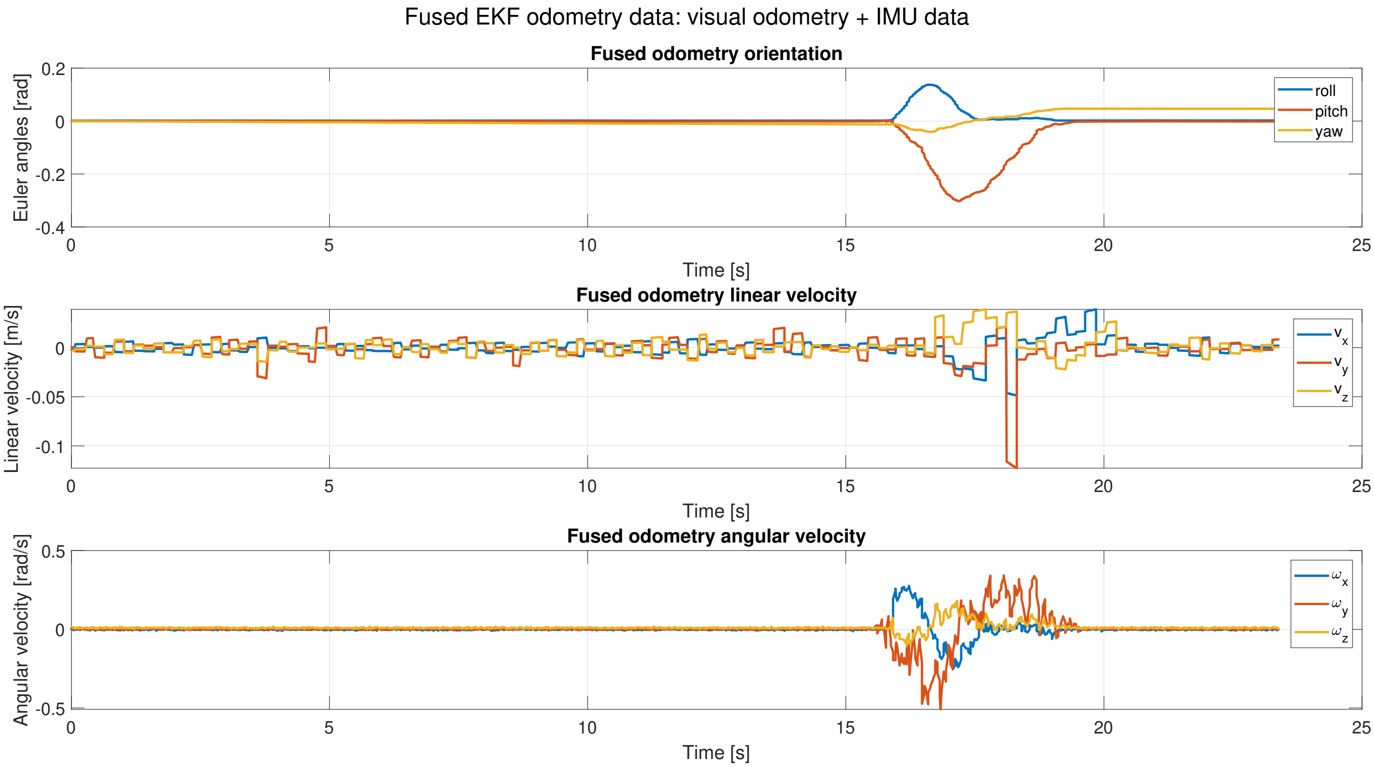


Fig. 17: Filtered EKF odometry data produced by the Robot Localization ROS package. The Kinect is moved between 15 and 20 seconds in this time frame.

Fig. 18: Resulting virtual environment after the Kinect is moved and the dynamic object is presented. Multiple artifacts of the human arm are present after movement of the Kinect.

Figure 18 shows the resulting virtual environment from the executed experiment. Unfortunately, dynamic artifacts are still present in the VR environment after movement of the Kinect sensor. Subjectively, the static objects and background in the resulting environment do seem to be improved compared to the environment were only visual odometry is used as seen in Figure 9b. However, the difference can be considered as minimal.

Conclusion
After careful implementation of the IMU sensor together with the Kinect visual odometry, minor improvements were found such as more continuous data, more data points, and decreased variance in the data for static cases compared to the previous experiment where only visual odometry was used as an input for the RTAB-map algorithm. Unfortunately, the improved visual odometry did not result in an improvement in terms of dynamic object tracking.

5) Concluding remarks: In experiment 3, the root cause for the latency issue was found. The resulting 8 Hz update rate did surpass the baseline performance of 7 Hz that was set as a criterion.
Dynamic object tracking is still a persisting problem that affects the performance of the implemented system. In the root cause analysis, several aspects were considered across all the subsystems. Primarily, the main focus was laid on improving the quality of the visual odometry and reduction of high frequency motion since dynamic object tracking performed adequately when the Kinect 2 sensor was not moving. After the investigations that were performed in this work, it can be said with a large amount of certainty that the visual odometry is not the root cause of the dynamic object tracking issue.
The Kinect 2 sensor was also taken into consideration in the root cause analysis. First of all, improving the hardware will most likely not result in improved performance since the output of the RTAB-map algorithm does function according to the performance requirement when it is not moving. Also, image distortion due to the high-frequency motion of the Kinect itself was regarded as a possible root cause. An increase in the VR environment quality was observed when the velocity of the platform and possible high-frequency movements in the enclosure of the Kinect 2 itself were reduced. However, it did not improve the dynamic object tracking performance, and it is therefore highly unlikely that the Kinect 2 sensor is the root cause of the dynamic object tracking issue.
The root cause for the dynamic object tracking might be due to the lacking performance of the RTAB-map algorithm itself. Several possible causes were mentioned in the RCA such as faulty parameter settings, long term memory management, and the classification of data points. Based on the adequate dynamic object tracking performance when the Kinect is not moving, the configuration of RTAB-map and the memory management aspect are highly unlikely to contribute to the dynamic object tracking issue. The remaining possible root cause regarding the classification of data points is most likely to be the main possible root cause. More research on RTAB-map related root causes is needed since these aspects are beyond the scope of the work presented in this paper.

References

[1] N. A. Busch and R. VanRullen, "Spontaneous eeg oscillations reveal periodic sampling of visual attention," *Proceedings of the National Academy of Sciences*, vol. 107, no. 37, pp. 16048–16053, 2010.

[2] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[3] C. Junhao, Z. Wang, H. Zhou, L. Li, and J. Yao, "Dm-slam: A feature-based slam system for rigid dynamic scenes," *ISPRS International Journal of Geo-Information*, vol. 9, p. 202, 03 2020.

[4] R. van der Zee, "Remote control of a robotic humanoid head by using an oculus vr headset," April 2019.

Appendix D: Installation specifications

This appendix provides the installation specifications for the implemented as presented in the Method section of the paper.

A. Installation guide

*Procedure for Twente humanoid head intel NUC PC (Also known as: **James**):*

- Make sure Ubuntu 18.04 is installed
- Install ROS Melodic: http://wiki.ros.org/melodic/Installation/Ubuntu
- Install Kinect 2 software: libfreenect(driver) and iai_kinect2(ros bridge)
  - Follow these instructions: https://github.com/code-iai/iai_kinect2
  - Make sure to install OpenCL
  - Also perform the calibration
  - Make sure the Kinect is connected to a USB3.0 port
- Install the control for James
  - Create a package in the src of your catkin ws: *catkin_create_package james geometry_msgs rospy*
  - Copy the files to the "james" directory in the /catkin_ws/src/ directory by replacing src/ with the src/ and test/ folder from the i-Botics Git
  - Run *catkin build*
  - Make the node.py file in the James directory executable: *sudo chmod +x node.py*
  - Place the jrc_can.py file from the i-Botics Git in the catkin_ws folder and make executable: *sudo chmod +x jrc_can.py*

Procedure for local PC (Cockpit):

- Make sure Ubuntu 18.04 is installed
- Install ROS Melodic: http://wiki.ros.org/melodic/Installation/Ubuntu
- Install RTAB-map: *sudo apt-get install ros-melodic-rtabmap-ros*
- Install Robot Localization: *sudo apt-get install ros-melodic-robot-localization*
- Install the delay_node package from the i-Botics Git
- Make sure the correct graphics driver is enabled in Ubuntu
  - Go to: settings/detail/about and check if your correct graphics card is enabled
  - If this is not the case, go to: software and updates/additional drivers. Change to the correct drivers here and apply the changes
- Install OpenVR **v.1.3.22**
  - Clone in your *home* folder: git clone –branch v1.3.22 https://github.com/ValveSoftware/openvr.git
  - Build: *cd /openvr && mkdir build && cd build && cmake .. && cmake –build . && make && sudo make install*
- Check the graphics driver requirements here: https://github.com/ValveSoftware/SteamVR-for-Linux
- Install Steam for Linux: https://store.steampowered.com/about/
- Install SteamVR: https://www.steamvr.com/en/
  - Additional info can be found on: https://github.com/ValveSoftware/SteamVR-for-Linux
  - If you encounter issues with curl 4, the following fixes this: https://www.leadwerks.com/community/topic/19252-curl_openssl_4-not-found/
  - The fix:
    * *cd /home/USERNAME/.steam/steam/ubuntu12_32/steam-runtime/pinned_libs_64*
    * *mv libcurl.so.4 libcurl.so.4.bak*
    * *ln -s /usr/lib/x86_64-linux-gnu/libcurl.so.4 libcurl.so.4*
- RViz vive plugin install in catkin ws (https://github.com/getsomatic/rviz_vive_plugin)
  - Install GLEW: *sudo apt-get install libglew-dev*
  - Install libsdl2-dev: *sudo apt-get install libsdl2-dev*
  - The install commands on the github page above are wrong, use the following:
    * First go to the src of the catkin workspace
    * *git clone https://github.com/getsomatic/rviz_vive_plugin.git*
    * *git clone https://github.com/getsomatic/rviz_vive_plugin_msgs.git*
  - Go to the rviz_vive_plugin folder in the src of your workspace. Then to cmake and open the file FindOpenVR.cmake. Change the contents to:
    * set(OpenVR_LIBRARIES /openvr/lib/linux64/libopenvr_api.so)
    * set(OpenVR_INCLUDE_DIRS /openvr/headers)
  - In *vive_conversion.cpp*, change line 14 to:
    * return Ogre::Quaternion(0.5, 0.5, -0.5, -0.5) * orientation;
    * This will correct for the orientation of the HTC Vive w.r.t the Kinect2

  - In *vive_display.cpp*, after line 274 add:
    * hmd.Pose.Orientation = Ogre::Quaternion(0.5, -0.5, 0.5, 0.5) * hmd.Pose.Orientation;
    * This will correct for the orientation of the HTC Vive w.r.t the Kinect2
  - Use *catkin_make* to build the plugin
  - If OpenVR cannot be found, your can change the *vive.h* file of the rviz_vive_plugin manually:
    * Change the openvr include line to: #include "../../../../../openvr/headers/openvr.h"
- Install the HTC_readout package from the i-Botics Git and use *catkin_make*
- Copy the visual_setup folder from the i-Botics Git to the home folder your PC. This contains the startup shell file, the RTAB-map launch file and the rviz config script

B. Running the setup

- Connect all the necessary devices properly: James, HTC Vive, Kinect 2. Use the local network to interface James with the cockpit.
- For using the autorun script, the following software needs to be installed:
  - Install Terminator:
    https://blog.arturofm.com/install-terminator-terminal-emulator-in-ubuntu/
  - Install screen: *sudo apt install screen*
  - Install sshpass: *sudo apt install sshpass*
- Start the setup by using the shell script in the visual_setup folder on the local PC. Navigate to that folder in a Terminator terminal and run:
  - *./su_point.sh*
  - It is advised to first startup the setup without James connected to check if the values from the HTC readout are correct i.e. if they are in bound.
- The script will open several terminals and will start all the software in the correct order. First it will instantiate a SSH connection with the James platform. Then the software will start in the following order:
  1) Roscore
  2) The Kinect 2 ROS bridge
  3) RTAB-map
  4) Delay node (default 0 sec. delay)
  5) Rviz in the steam runtime environment
  6) The CAN interface
  7) The control for James
  8) The readout of the HTC orientation data
- After the full startup is performed, all the terminals can be closed by hitting enter in the same terminal as the ./su_point.sh command was executed.

C. Install for streaming

As an alternative, streaming of image data from the Kinect 2 directly to the HTC Vive is also possible. For this, Unity3D is utilized. *The install from Appendix A has to be executed first.*

- Install ROSbridge: *sudo apt-get install ros-melodic-rosbridge-suite*
- Install Unity3D using unityhub https://unity3d.com/get-unity/download
- Move the UnityHub.AppImage file (from your /Downloads folder to home/Unity/)
- Assets, import asset, and use file asset file from the i-Botics GIT.
- Use the ./su_stream.sh file to run the setup. When Unity starts press the play button. After that the setup can be started.

D. Install for IMU attachment

- Install from: http://wiki.ros.org/xsens_mti_driver
- Find the *xdainterface.cpp* in the catkin workspace and replace with the one on the i-Botics GIT (Changes made after line 218).
- Run using ./su_point_imu.sh