TTK4551 - Engineering Cybernetics, Specialization Project
VIVE-Workcell Setup and Calibration

Morten Astad

10th June 2019

Contents

Abstract

The main motivation of this project is to lay the groundwork for a robot cell calibration system, using HTC VIVE - a virtual reality (VR) system with precise room-scale tracking. A VIVE tracking system was installed for Thrivaldi - the KUKA robotics laboratory at NTNU ITK. The main goal was then to calibrate the tracking system for Thrivaldi, and evaluate the calibrated system. A calibration procedure was established based on an algorithm by Park and Martin [1]. HTC VIVE and this algorithm was set up in Robotic Operating System (ROS) - an open-source framework for writing software for robots. Two methods was established for the calibration: a manual and an automated one. The manual method was used because of problems with the current ROS setup for Thrivaldi. The calibrated system has a steady-state error within the robot's workspace, in the range of 0.5 to 1.5 cm and 0.2 to 0.7 degrees for translations and orientations respectively. This error is not a conclusive measure of the VIVE's accuracy, and the error can potentially be reduced by an order of magnitude.

Chapter 1

Introduction

Robotic workcells for various manufacturing processes have to be calibrated such that the robot knows the location of parts, tools and surfaces in the environment. The calibration can be performed by moving the robot to points in its environment, where the points are registered with the robot as a measurement system. These points can then be mapped to their corresponding points in an object model, and the mapping returns the relation between the robot and this object.

The procedure of calibrating complex robot cells can become laborious and expensive. This calibration can be further complicated by mobile robots that change their locations in the production, or processes that should be changed quickly and often. The question then arises: "How can we perform robot cell calibration easier and quicker?". A proposed answer to this question could be the use of an HTC VIVE for rapid robot cell calibration.

The HTC VIVE is a virtual reality (VR) system developed by HTC and Valve corporation. This system has become one of the best VR experiences available to consumers, and has the highest revenue share on the VR market [2]. The system's success is probably rooted in its innovative technology for tracking the users hands and head. It is able to perform room-scale tracking with sub-millimeter precision within a diagonal area up to 5 m. These specifications are impressive for a consumer-grade product such as the HTC VIVE, and it could potentially be used for robot cell calibration if the accuracy is sufficient. The initial plan was therefore to set up an HTC VIVE for use in Robotic Operating System (ROS) - an open-source framework for writing software for robots. This setup can then be used to assess the feasibility of using an HTC VIVE for rapid robot cell calibration.

There was initially four main tasks planned for this project:

1. Set up an HTC VIVE with the Ubuntu operating system
2. Establish an overview of the existing VIVE-ROS bridging attempts
3. Set up an HTC VIVE for use in ROS
4. Evaluate the HTC VIVE accuracy for robot cell calibration

The first three tasks was mostly finished during a summer job at SINTEF Digital, and the last task was also initiated. Using the HTC VIVE with Ubuntu was simple and turned out to be plug and play. However, the existing VIVE-ROS bridging attempts was badly documented or did not work with newer software versions. A ROS node was therefore set up in order to make the features of HTC VIVE devices available in ROS. This node and relevant findings from the summer job will be presented in the introduction, together with the necessary equipment and software for this project. An extra emphasis will also be put on the HTC VIVE and its tracking system, in order to understand the technology and evaluate the system's accuracy.

The intention of this text is not to describe ROS in an in-depth manner, as it is a huge framework, and there are a lot of free and comprehensive resources available online. Enough information will be given to understand the context of how ROS is used in the text, and related resources will also be referenced where appropriate. If the reader is new to ROS, it is recommended to read [3][4] for a general overview of the framework and its tools.

In order to use the VIVE tracking system for rapid robot cell calibration, the position and orientation (pose) of the tracked devices must be known relative to the robot and its environment. This text is therefore primarily concerned with how a calibration procedure can be established for the tracking system. The calibration procedure should generalize with minor adjustments to any industrial manipulator that is supported by ROS. A VIVE tracking system was installed for Thrivaldi - the KUKA robotics laboratory at NTNU ITK. The main goal is then to calibrate the tracking system for Thrivaldi, and evaluate the calibrated system.

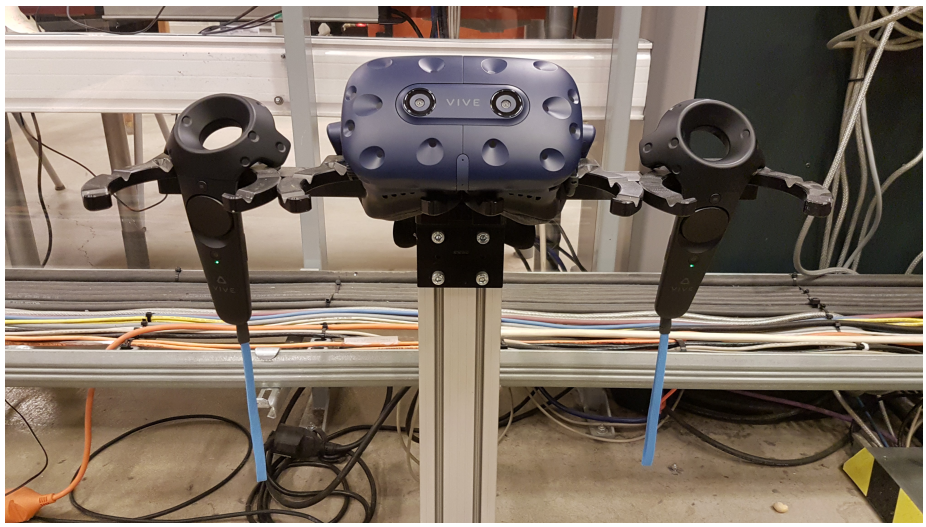


Figure 1.1: HTC VIVE head-mounted display (HMD) and one controller for each hand

1.1 Equipment

1.1.1 HTC VIVE (Pro)

The HTC VIVE is a room-scale VR system, allowing the user to freely walk around a diagonal area of up 5 m and interact with an environment. Steuer [5] defined VR as "a real or simulated environment in which a perceiver experiences telepresence", where telepresence is defined as "the experience of presence in an environment by means of a communication medium". The main medium is in this case a head-mounted display (HMD) that is worn on the users head. This HMD immerses the user in a visualized environment, by displaying a 3D image through lenses in the HMD. There are also controllers and trackers available that allows the user to interact directly with the environment. The trackers are functionally similar to controllers, but have a smaller puck-like form factor and there are no connected inputs. The specifications of the HMD is listed in table 1.1, and a standard HTC VIVE kit is shown in figure 1.1.

Tracking technology

The technology that enables the HTC VIVE to track devices in a room-scaled environment is called SteamVR tracking. This technology sweeps the room horizontally and vertically with 850 nm infrared (IR) laser lines from one or two inertial base stations in the room. These base stations contains a pair of DC motors (Nidec B2044N01) that spins at a rate of 120 Hz, where one motor is turned by 90° and phase shifted by 180° from the other motor. In other words, each motor takes turns sweeping the room horizontally and vertically through a wheel-mounted line lens with a field of view of 120°. The base stations also takes turns sweeping the room in a similar manner, where one of the base stations acts as a master, and the other acts as a slave. This gives the tracking system an update rate of 60 Hz.

The base stations contains an array of 15 IR LEDs (Vishay VSMY3850), which floods the tracking volume with ∼1.8 MHz modulated pulses at the start of each sweep. These pulses are used for time synchronization and transmission of *omnidirectional optical transmitter* (OOTX) data. This data provides the tracked devices with identification of the base stations, their factory calibration data and current status [6].

The tracking works by measuring the time difference between synchronization pulses and line sweeps, from surface-mounted IR photodiodes on the tracked devices. This time difference is used to compute the angle of a base station's motor, and computing both motor angles gives the intersection between two (perpendicular) planes - a line. The line is projected from a base station towards an IR photodiode on the surface of a tracked device.

In normal operation, multiple IR photodiodes are hit by the laser during a single line sweep, and multiple projected lines are computed. The position of the IR photodiodes along these lines are however unknown, but the geometric relationship between the IR photodiodes are known. Estimating the pose of a tracked device based on this knowledge is similar to the Perspective-n-Point (PnP) problem, where n is the number of 3D points (IR photodiode positions). Solving this problem is equivalent to finding the transformation that maps points from a local device frame to a global tracking frame, which respects the constraints that are given by the projected lines.

Accuracy and precision

The SteamVR tracking technology is advertised with sub-millimeter precision, which is an impressive feat for a consumer-grade product such as the HTC VIVE. Niehorster et al. [7] tested the HTC VIVE HMD, and compared its accuracy and precision to a research-grade tracking system. They showed that the average positioning error was 17 mm with 9 mm standard deviation, and the RMS noise levels was below 0.2 mm and 0.02°.

There was however a systematic offset in their measurements, which suggests that the HTC VIVE uses a reference plane that is tilted away from the physical ground plane. These offsets also changed whenever the HMD regained tracking after occlusion, making a calibration procedure to remove the offsets hard in practice. They therefore concluded that the HTC VIVE was not suited for scientific experiments if loss of tracking was likely.

Borges et al. [8] showed a similar precision as Niehorster, but also tested the dynamic accuracy and precision for robotics applications. The dynamic accuracy was shown to be in the millimeter to meter range with best case 2.36 mm and worst case 0.80257 m. They showed that the VIVE's tracking algorithm weighted inertial measurements more to produce smooth trajectories for VR applications. Hence, the tracking algorithm is unsuited for robotics applications, where accuracy and repeatability are key.

Borges et al. also introduced their own tracking algorithm for robotics applications [9]. Their algorithm outperformed the VIVE's algorithm by up to two orders of magnitude in dynamic accuracy. The HTC VIVE algorithm does however outperform their algorithm by over an order of magnitude in the stationary case.

The 60 Hz update rate of the tracking system would cause motion sickness on its own. Each tracked device therefore contains an inertial measurement unit (IMU), providing estimated poses between the tracking updates. The specifications of this IMU is given by table 1.2. A Kalman filter is used for sensor fusion in order to combine the measurements from IMU and tracking system, and the poses are updated at 220-360 Hz depending on the device type [10].

Screen:	Dual AMOLED 3.5" diagonal
Resolution:	1440 x 1600 pixels per eye (2880 x 1600 pixels combined)
Refresh rate:	90 Hz
Field of view:	110 degrees
Audio:	Hi-Res certificate headset Hi-Res certificate headphone (removable) High impedance headphone support
Input:	Integrated microphones
Connections:	USB-C 3.0, DP 1.2, Bluetooth
Sensors:	SteamVR Tracking, G-sensor, gyroscope, proximity, IPD sensor
Ergonomics:	Eye relief with lens distance adjustment Adjustable IPD Adjustable headphone Adjustable headstrap

Table 1.1: HTC VIVE HMD specifications [11]

Range	Gyro Full Scale Range (°/sec)	Gyro Sensitivity (LSB/°/sec)	Gyro Rate Noise °/sec/$\sqrt{\text{Hz}}$	Accel Full Scale Range (g)	Accel Sensitivity LSB/g
0	± 250	131	0.01	± 2	16384
1	± 500	65.5	0.01	± 4	8192
2	± 1000	32.8	0.01	± 8	4096
3	± 2000	16.4	0.01	± 16	2048

Table 1.2: HTC VIVE accelerometer and gyroscope specifications [12]

1.1.2 Thrivaldi

Thrivaldi is a KUKA robotics laboratory at the Department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU). The laboratory consists of a robot cell containing two, 6 degrees of freedom (DOF), KUKA KR 16 industrial robots. One of these robots is mounted on a 3-axis gantry system from Güdel, giving it 9 DOF in total. Thereby the anglicized name Thrivaldi (Þrívaldi), a 9-headed jötunn from Norse mythology.

The setup and interface of the laboratory was documented extensively by Eriksen in his Master thesis [13], and practical documentation is also available from an online repository [14]. MoveIt! is a motion planning library for ROS, and the existing MoveIt! setup for Thrivaldi was used in order to plan and execute trajectories for the robot.

1.2 Software

1.2.1 Robotic Operating System (ROS)

Robotic Operating System (ROS) is an open-source collection of frameworks for writing software for robots, commonly referred to as a middleware. At its core it offers a communication system, which provides a message passing interface between distributed nodes. This infrastructure forces the user to implement clear interfaces between the nodes in their system, making ROS a distributed and modular framework by design [15]. The main motivation behind ROS is code reusability in robotics research and development. ROS has an active community with a large collection of tools and libraries, which simplifies the task of writing complex and robust software for robots.

1.2.2 VIVE bridge

VIVE bridge is a ROS node that makes use of the OpenVR software development kit (SDK) by Valve, allowing access to VR hardware such as the HTC VIVE in a ROS environment. The package exposes the pose of each tracked device as a coordinate frames with respect to an inertial tracking frame, which coincides with the master base station. Linear and angular velocities (twists) from the IMU in each device, and inputs from the axes and buttons on each controller are also published as messages.

The inertial tracking frame is defined relative to some arbitrary inertial frame that is chosen by the user, in order to make sense of the environment. These frames are related by a transformation that is exposed as x, y, z and roll, pitch, yaw (RPY) parameters. A standard interface is provided to change the parameters at any time. The robot cell or any other space is then calibrated by updating these parameters.

The source code is not publicly available at the present time, but the package documentation is attached in appendix A. It is recommended that the reader skims through this documentation for an overview of the capabilities and usage of the package.

Chapter 2

VIVE-workcell setup

The base stations have to be securely set up in the workcell before calibration can be performed. Tripods could be used to install the base stations in a temporary setup for workcell calibration tasks. These tripods have to be extended to over 2 m above ground with standard ¼" UNC threaded camera mounts. This kind of setup could work for Thrivaldi.

The tripods do however take up space and could be moved around unintentionally in the robot cell, invalidating the last calibration. Tripods are therefore unsuited for experimental purposes, where it is important that the setup does not change between experiments. The base stations were for this reason firmly mounted in a fixed setup for Thrivaldi as shown in figure 2.1. The HMD was also kept visible in the robot cell at all times during use, in order to avoid the problems described by Niehorster et al. [7].

The considerations when setting up base stations in a workcell are the same, regardless of the method of choice. A setup should follow the recommendations of the official VIVE support [16], where the base stations should be:

- Above head height, ideally more than 2 m above ground
- Angled down between 30 and 45 degrees

These recommendations were followed for Thrivaldi by mounting the base stations ∼3 m above ground, facing 45° down toward the center of the robot cell. Base stations have a field of view of 120°, leaving 30-45° for adjustment. The optimal setup is therefore not strict in the sense that the base stations have to be placed perfectly. Most importantly, the base stations should be placed such that their view of each other and the workcell is unobstructed. It is also important that the base stations' field of view overlaps as much as possible within the intended tracking volume. This is because the tracking system's update rate is halved to 30 Hz if the tracked device is hit by only one base station.

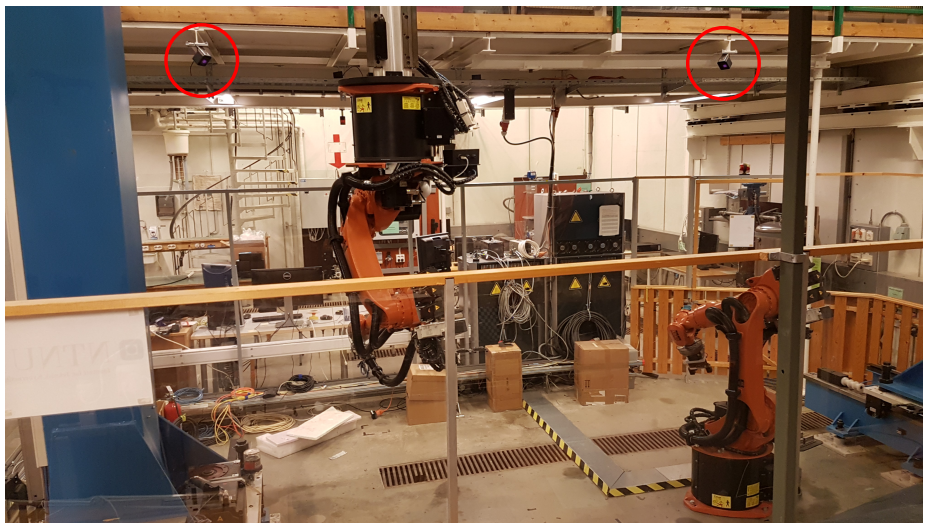


Figure 2.1: Base stations mounted below h-beams in the roof.

Figure 2.2 shows the tracking area of the current setup for Thrivaldi, and covers most of the robot cell. The base station shown in green should ideally be rotated a few degrees anti-clockwise while keeping the HMD within the tracking area. However, the current setup has no problems tracking the area under and in front of the gantry. In addition, the computer desk in front of the robot cell is also tracked with the current setup.

2.1 Internal HTC VIVE calibration

The setup has to be internally calibrated with the SteamVR software for HTC VIVE [17]. This calibration is done via a room setup tool, where a room-scale setup of the HTC VIVE is performed. The calibration process is completed by following simple on-screen instructions and prompts. Both controllers are put on the ground to calibrate the floor, and the boundary of the tracking area is traced with a controller. The traced boundaries are used in a system called Chaperone, in order to warn the user about physical obstructions. This system displays a grid within the HMD whenever the user approaches an obstruction, and it is therefore important for user safety in VR applications using the HMD.

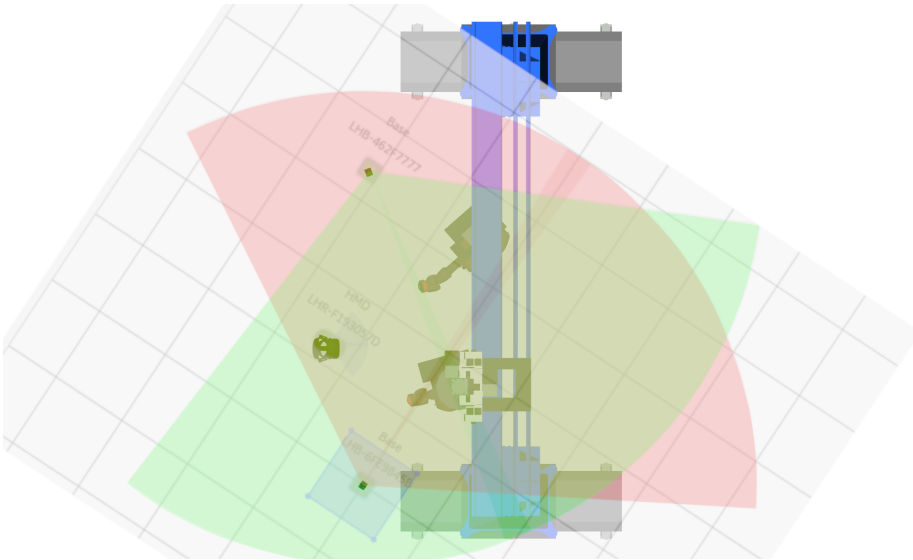


Figure 2.2: Plan view of the robot cell, where the base stations' field of view is shown in red (master) and green (slave). Each square corresponds to $1\,m^2$.

Chapter 3

VIVE-workcell calibration

In order to use the VIVE tracking system for rapid robot cell calibration, the pose of the tracked devices must be known relative to the robot and its environment. This chapter is motivated by a need to establish a calibration procedure for the tracking system. If we are able to find the pose of a single tracked device with respect to the robot, we are also able to find the other devices through their shared inertial frame. We therefore need a method to find the pose of a single tracked device relative to the robot.

Finding the pose of a wrist-mounted sensor with respect to a robot's wrist frame is known as the hand-eye calibration problem. This problem got its name from the robotics community, where a camera (eye) was attached to the robot's gripper (hand) [18][19]. Similarly, it is possible to attach a tracked device to the robot's gripper, a VIVE controller in this case, and find the pose of this device by using one of the many solutions to the hand-eye problem.

Knowing the pose of a tracked device relative to the robot, it is a simple task to compute the transformation between the robot's inertial frame and the tracking system's inertial frame. A method is therefore needed to solve the hand-eye calibration problem, and interestingly, this problem has a lot in common with pose estimation. This is not surprising as we want to estimate the pose of our sensor with respect to the robot.

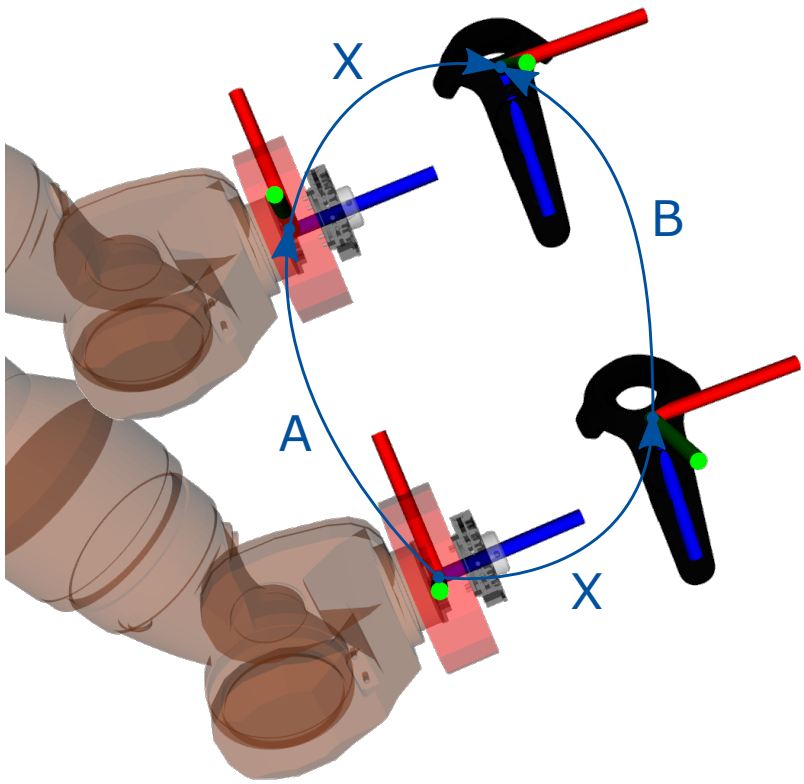


Figure 3.1: Geometric interpretation of the $AX = XB$ problem

3.1 Problem formulation

$$\mathbf{AX} = \mathbf{XB}, \qquad \mathbf{A}, \mathbf{B}, \mathbf{X} \in SE(3) \tag{3.1}$$

The standard hand-eye calibration problem was formulated by Shiu and Ahmad [20]. They stated the problem as (3.1), an equation of homogeneous transformations, where A is a change in the robot's wrist pose, B is the sensor displacement from changing the wrist pose, and X is an unknown transformation relating the wrist frame to the sensor frame. Figure 3.1 shows a geometric interpretation of

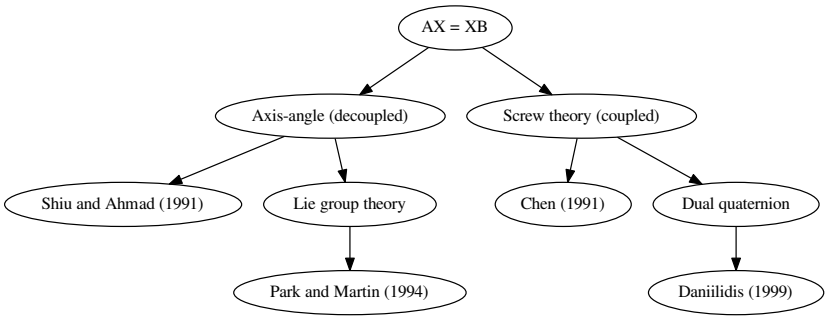


Figure 3.2: A couple of hand-eye problem representations, and associated solutions presented in the literature.

this problem. The unknown transformation X is constant under the assumption that the tracked device is firmly attached to the robot's gripper.

3.2 Overview of solutions

Hand-eye calibration is an established field with a large body of literature. Some of the approaches and contributions in this field will be presented, before one of them is picked for implementation. Shiu and Ahmad [20] used an angle-axis representation of (3.1) and least-squares fitting to solve the rotational part of X, and then solved the translational part with the rotation. They also showed that at least two pairs of A and B are necessary to get a unique solution. Similarly, Park and Martin [1] also decoupled the rotation and translation, and presented a method using Lie group theory and least-squares fitting.

Decoupling the rotation and translation, despite it's simplicity, has the problem that errors in the rotational part could propagate to the translational part [19]. Chasles' theorem, as stated by Chen [21], says that a rigid body displacement can be composed of a translation along a unique screw axis, and a rotation about the same axis. This representation of a rigid body displacement is known

as a screw. Chen [21] was the first that solved both parts of X simultaneously by applying the theory of screws [19]. Similarly, Daniilidis [22] applied unit dual quaternions - an algebraic representation of screws. Interestingly, the hand-eye problem (3.1) can be reduced to solving a second order equation, by using dual quaternions to represent the transformations A, B and X.

The solutions introduced here and their relations are summarized in figure 3.2. There are many other solutions that is not mentioned in this text, and more comprehensive resources are available in [19][23]. Of the mentioned solutions, Park and Martin [1] were chosen specifically for its elegance, and it will be explained, implemented and tested in the forthcoming sections of the report.

Daniilidis' [22] solution was also briefly implemented and tested, and even though it is interesting and computationally efficient, it will not be presented here due to its use of dual quaternions. This decision was mainly the outcome of time constraints. If the reader is interested, it is recommended to read Kenwright [24] for a beginners guide to dual quaternions before reading the paper by Daniilidis [22].

3.3 Solving $AX = XB$ on the Euclidean Group

Park and Martin [1] used Lie group theory to concisely state the conditions for existence and uniqueness of their solutions. The input to their method is measured pairs of homogeneous transformation matrices $(A,\ B) \in SE(3)$, where $SE(3)$ is the special Euclidean group of dimension 3 defined by:

$$SE(3) = \left\{ \mathbf{T} \mid \mathbf{T} = \left(\begin{bmatrix} \mathbf{R} & \vec{r} \\ 0 & 1 \end{bmatrix} \right),\ \mathbf{R} \in SO(3),\ \vec{r} \in \mathbb{R}^3 \right\} \tag{3.2a}$$

$$SO(3) = \left\{ \mathbf{R} \mid \mathbf{R} \in \mathbb{R}^{3\times 3}, \quad \mathbf{R}^T\mathbf{R} = \mathbb{I}_3, \quad \det(\mathbf{R}) = 1 \right\} \tag{3.2b}$$

$$so(3) = \left\{ \boldsymbol{\omega} \mid \boldsymbol{\omega} \in \mathbb{R}^{3\times 3}, \quad \boldsymbol{\omega}^T = -\boldsymbol{\omega} \right\} \tag{3.2c}$$

This group is a smooth manifold with the group structure $SO(3) \times \mathbb{R}^3$, such that the group operations are smooth maps - a Lie group. The most important Lie group property for our purposes, is the existence of a well-defined logarithmic mapping (3.3b) from the manifold to a local Euclidean structure on the manifold. This local structure is associated with the tangent space at the group's identity

element, called the group's associated Lie algebra. The inverse exponential mapping (3.3a) is also well defined.

$$\exp : SO(3) \mapsto \ so(3) \tag{3.3a}$$
$$\log : \ so(3) \mapsto SO(3) \tag{3.3b}$$

With this knowledge in mind, let us start by writing out the standard hand-eye problem as:

$$\mathbf{AX} = \mathbf{XB} \tag{3.4a}$$
$$\begin{pmatrix} \mathbf{R}_A & \vec{r}_A \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_X & \vec{r}_X \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_X & \vec{r}_X \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_B & \vec{r}_B \\ 0 & 1 \end{pmatrix} \tag{3.4b}$$
$$\mathbf{R}_A \, \mathbf{R}_X = \mathbf{R}_X \, \mathbf{R}_B \tag{3.4c}$$
$$\mathbf{R}_A \, \vec{r}_X + \vec{r}_A = \mathbf{R}_X \, \vec{r}_B + \vec{r}_X \tag{3.4d}$$
$$(\mathbf{R}_A - \mathbb{I}_3) \, \vec{r}_X = \mathbf{R}_X \, \vec{r}_B - \vec{r}_A \tag{3.4e}$$

The main result of Park and Martin [1] is that an exact solution to (3.1) exists if and only if:

$$\|\log(\mathbf{A})\| = \|\log(\mathbf{B})\|, \qquad \mathbf{A}, \, \mathbf{B} \in SE(3) \tag{3.5}$$

Where the logarithm of a homogeneous transformation matrix $T \in SE(3)$ is given by:

$$\log(\mathbf{T}) = \begin{bmatrix} \log(\mathbf{R}) & \mathbf{T}_{v \mapsto r}^{-1} \, \vec{r} \\ \mathbb{0}_{1\times 3} & 0 \end{bmatrix} = \begin{bmatrix} \theta[\vec{k}]_\times & \left[\int_0^1 \exp(\theta[\vec{k}]_\times s) \, ds\right]^{-1} \vec{r} \\ \mathbb{0}_{1\times 3} & 0 \end{bmatrix} \in se(3) \tag{3.6}$$

The ordered pair $(\theta, \vec{k})$ in (3.6) is the angle-axis parameters of a rotation matrix R, and $[\vec{k}]_\times$ is the skew-symmetric matrix form of vector $\vec{k}$. These parameters are given by (3.7), which implies that the angle θ is not unique when the trace of rotation matrix $\mathrm{Tr}(R)$ is equal to -1 ($\theta = \pm\pi$). If the angle θ is not unique, then the logarithm in (3.6) is also not unique.

$$\theta = \arccos\left(\frac{\mathrm{Tr}\,(\mathbf{R}) - 1}{2}\right), \qquad [\vec{k}]_\times = \frac{1}{2\sin(\theta)}\left(\mathbf{R} - \mathbf{R}^T\right) \tag{3.7a}$$

$$[\vec{k}]_\times := \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix} \in so(3), \qquad \vec{k} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \in \mathbb{R}^3 \tag{3.7b}$$

The element $T_{v\mapsto r}^{-1}\,\vec{r}$ in (3.6) comes from the fact that the exponential of $\log(T)$ should result in T, leaving behind the translation $\vec{r}$ in this element. It is possible to compute this element as (3.8c) from $T_{v\mapsto r}\,T_{v\mapsto r}^{-1} = \mathbb{I}_3$ as shown in Condurache [25], where $T_{v\mapsto r}$ is given by (3.8b) and $\mathbb{I}_3$ is the 3×3 identity matrix. The mapping $T_{v\mapsto r}$ is in fact a screw that maps a local vector $\vec{v}$ on the manifold to a translation $\vec{r}$. This mapping corresponds to integrating the vector $\vec{v}$ over a rotation R (from $\mathbb{I}_3$ to R), resulting in a screw motion over the rotation.

$$\exp\left(\theta[\vec{k}]_\times s\right) = \mathbb{I}_3 + \sin(\theta s)[\vec{k}]_\times + \left[1 - \cos(\theta s)\right][\vec{k}]_\times^2 \tag{3.8a}$$

$$\mathbf{T}_{v\mapsto r} = \int_0^1 \exp\left(\theta[\vec{k}]_\times s\right) ds = \mathbb{I}_3 + \frac{1-\cos(\theta)}{\theta}[\vec{k}]_\times + \frac{\theta - \sin(\theta)}{\theta}[\vec{k}]_\times^2 \tag{3.8b}$$

$$\mathbf{T}_{v\mapsto r}^{-1} = \left[\int_0^1 \exp\left(\theta[\vec{k}]_\times s\right) ds\right]^{-1} = \mathbb{I}_3 - \frac{1}{2}\theta[\vec{k}]_\times + \left[1 - \frac{\theta}{2}\cot\left(\frac{\theta}{2}\right)\right][\vec{k}]_\times^2 \tag{3.8c}$$

The rotation R_X is assumed to be decoupled from the translation $\vec{r}_X$ in this solution to the hand-eye problem, and it is therefore only the rotational part of (3.6) that is considered. This assumption simplifies (3.5) to its rotational part (3.9a), which can be further simplified as (3.9c) for positive angles and axis vectors of unit length.

$$\|\theta_A\,[\vec{k}_A]_\times\| = \|\theta_B\,[\vec{k}_B]_\times\| \tag{3.9a}$$

$$|\theta_A| = |\theta_B|, \quad \|\vec{k}_A\|, \|\vec{k}_B\| = 1 \tag{3.9b}$$

$$\theta_A = \theta_B, \quad \theta_A, \theta_B \geq 0 \tag{3.9c}$$

It is now simple to show the main result of Shiu and Ahmad [20] from (3.4c), where they have used the identities; $R\exp\left([\vec{k}]_\times\right)R^T = \exp\left(R\,[\vec{k}]_\times R^T\right)$, $R\,[\vec{k}]_\times R^T =$

$[R\,\vec{k}]_\times$ for any skew-symmetric matrix $[\vec{k}]_\times$, and $R = \exp(\theta[\vec{k}]_\times)$ is the angle-axis parametrization of rotation matrix R:

$$\mathbf{R}_A = \mathbf{R}_X \mathbf{R}_B \mathbf{R}_X^T \tag{3.10a}$$

$$\exp\left(\theta_A[\vec{k}_A]_\times\right) = \mathbf{R}_X \exp\left(\theta_B[\vec{k}_B]_\times\right) \mathbf{R}_X^T = \exp\left(\theta_B[\mathbf{R}_X\vec{k}_B]_\times\right) \tag{3.10b}$$

$$\theta_A[\vec{k}_A]_\times = \theta_B[\mathbf{R}_X\vec{k}_B]_\times \tag{3.10c}$$

$$\theta_A\vec{k}_A = \mathbf{R}_X\theta_B\vec{k}_B \tag{3.10d}$$

$$\vec{k}_A = \mathbf{R}_X\vec{k}_B, \quad \theta_A = \theta_B \geq 0 \tag{3.10e}$$

The result in (3.10e) implies that any exact solution to (3.4c) is independent of the angles θ_A and θ_B, except for the special case where both axis vectors $\vec{k}_A$ and $\vec{k}_B$ are collinear. This result is however not true in general, because the measured pairs of A and B are affected by noise. The more general result in (3.10d) is therefore used to estimate a solution to (3.4c) instead. A least squares minimization problem (3.11a) can then be formulated from (3.10d) to find this solution, where the objective is minimized over all the measured pairs of A and B. Similarly, by assuming that the rotation R_X is decoupled and known, a least squares minimization problem (3.11b) can be formulated from (3.4e) to estimate the translation $\vec{r}_X$.

$$\hat{R}_X = \underset{\mathbf{R}_X}{\arg\min} \sum_{i=1}^{n} \|\mathbf{R}_X\theta_B\vec{k}_B - \theta_A\vec{k}_A\|^2 \tag{3.11a}$$

$$\hat{r}_X = \underset{\vec{r}_X}{\arg\min} \sum_{i=1}^{n} \|(\mathbf{R}_{A_i} - \mathbb{I}_3)\,\vec{r}_X + \vec{r}_{A_i} - \hat{\mathbf{R}}_X\,\vec{r}_{B_i}\|^2 \tag{3.11b}$$

Solving the minimization problems in (3.11) is a lot simpler than solving the coupled problem, and their optimal solutions do in fact have a closed form. Park and Martin [1] showed that the optimal rotation $\hat{R}_X$ can be expressed explicitly as (3.12a), where they have used a result by Nadas [26]. This rotation is unique if M^TM, as defined by (3.12a), is non-singular and has no repeated eigenvalues. This will be satisfied in general, making the rotation unique [1]. The optimal translation $\hat{r}_X$ is then also unique, and it is simply given by the linear least squares solution (3.12b), where $C^\dagger$ is the left Moore-Penrose inverse.

$$\hat{\mathbf{R}}_X = \left(\mathbf{M}^T\mathbf{M}\right)^{-\frac{1}{2}}\mathbf{M}^T, \quad \mathbf{M} = \sum_{i=1}^{n}\left(\theta_{B_i}\vec{k}_{B_i}\right)^T\theta_{A_i}\vec{k}_{A_i} \tag{3.12a}$$

$$\hat{r}_X = \mathbf{C}^\dagger\vec{d} = \left(\mathbf{C}^T\mathbf{C}\right)^{-1}\mathbf{C}^T\,\vec{d} \tag{3.12b}$$

$$\mathbf{C} = \begin{bmatrix}\mathbf{C}_1\\ \mathbf{C}_2\\ \vdots\\ \mathbf{C}_n\end{bmatrix} = \begin{bmatrix}\mathbb{I}_3 - \mathbf{R}_{A_1}\\ \mathbb{I}_3 - \mathbf{R}_{A_2}\\ \vdots\\ \mathbb{I}_3 - \mathbf{R}_{A_n}\end{bmatrix} \in \mathbb{R}^{3n\times 3}, \quad \vec{d} = \begin{bmatrix}\vec{d}_1\\ \vec{d}_2\\ \vdots\\ \vec{d}_n\end{bmatrix} = \begin{bmatrix}\vec{r}_{A_1} - \mathbf{R}_X\vec{r}_{B_1}\\ \vec{r}_{A_1} - \mathbf{R}_X\vec{r}_{B_2}\\ \vdots\\ \vec{r}_{A_1} - \mathbf{R}_X\vec{r}_{B_n}\end{bmatrix} \in \mathbb{R}^{3n} \tag{3.12c}$$

The closed-form least squares solution by Park and Martin [1] can now be summarized as algorithm 1, where a singular-value decomposition (SVD) is used to compute the symmetric and positive definite square root $(M^TM)^{-1/2}$. The transformation between the robot's inertial frame and the tracking system's inertial frame is then given by:

$$\mathbf{T}_i^{bs} = \mathbf{T}_s^{bs}\mathbf{T}_w^s\mathbf{T}_b^w\mathbf{T}_i^b \in SE(3), \qquad \mathbf{T}_w^s = \hat{\mathbf{X}} \tag{3.13}$$

Where $\{i\}$, $\{b\}$ and $\{w\}$ are the robot's inertial, base and wrist frames respectively, and $\{s\}$, $\{bs\}$ are the tracking system's sensor and inertial frames respectively. This transformation is used to update the configuration of the vive_bridge node, and the necessary transformations are measured upon completing the calibration.

Algorithm 1: A possible implementation of the closed-form least squares solution by Park and Martin [1].

Data: Pairs of wrist and sensor displacements:
$\{(\mathbf{A}, \mathbf{B})_1, (\mathbf{A}, \mathbf{B})_2, \ldots, (\mathbf{A}, \mathbf{B})_n\}, \quad \mathbf{A}, \mathbf{B} \in SE(3)$
Initialize $\mathbf{M} = \mathbb{0}_{3\times3}$, $\mathbf{C} = \mathbb{0}_{3n\times3}$, $\vec{d} = \mathbb{0}_{3n}$;
for $i = 1$ ***to*** n **do**
  $\mathbf{M} = \mathbf{M} + (\theta_{B_i}\vec{k}_{B_i})^T \, \theta_{A_i}\vec{k}_{A_i}$;
end
Compute SVD of $M^T M = U\Sigma V^T$;
Compute rotation $\hat{\mathbf{R}}_X = U \, \Sigma^{-0.5} \, V^T \, M^T$;
for $i = 1$ ***to*** n **do**
  $\mathbf{C}_i = \mathbb{I}_3 - \mathbf{R}_{A_i}$;
  $\vec{d}_i = \vec{r}_{A_i} - \hat{\mathbf{R}}_X \vec{r}_{B_i}$;
end
Compute translation $\vec{r}_X = \left(\mathbf{C}^T\mathbf{C}\right)^{-1} \mathbf{C}^T \, \vec{d}$;
Result: Transformation from wrist to sensor: $\hat{\mathbf{X}} = \begin{bmatrix} \hat{\mathbf{R}}_X & \hat{r}_X \\ \mathbb{0}_{1\times3} & 1 \end{bmatrix}$

Chapter 4

ROS implementation

Algorithm 1 was implemented in a ROS node with two available services; *add_sample* and *compute_transform*. These services are defined by their request and response message structures, which are shown in code listings 1 and 2, where the separator - - - splits the upper request- and lower response message. Implementing the algorithm in this manner builds on the modularity of ROS, making it easy to use for any ROS related project. Beginner level tutorials on how to interact with ROS services are available in [27] [28].

The intended use of the services is to send measured pairs of A and B, by calling the *add_sample* service with single pairs $(A, B)_n$ as request messages. The service then returns the number of sampled pairs n as a response message. If more than a single pair has been sampled, it is possible to compute the unknown transformation $\hat{X}$. This transformation is computed by calling the *compute_transformation* service with an empty request message. The service then returns the transformation, but only if the computation was successful, which is indicated by the *success* flag in code listing 2. Unsuccessful computations occurs when the logarithm mapping (3.6) is not unique, or there are less than two sampled pairs of A and B.

```
# Wrist transformation between two poses
geometry_msgs/TransformStamped A
# Sensor transformation between two poses
geometry_msgs/TransformStamped B
---
# Number of sampled pairs (A, B)
uint16 n
```

Listing 1: Message definition of the *add_sample* service

```
---
```

```
# Transformation from wrist to sensor
bool success
geometry_msgs/TransformStamped X
```

Listing 2: Message definition of the *compute_transform* service

4.1 Measuring wrist and sensor displacements

$$\mathbf{A}, \mathbf{B}, \mathbf{T} \in SE(3), \quad i \geq 1 \tag{4.1a}$$

$$\mathbf{A}_i = \mathbf{T}_{w_{i+1}} \mathbf{T}_{w_i}^{-1} \tag{4.1b}$$

$$\mathbf{B}_i = \mathbf{T}_{s_{i+1}} \mathbf{T}_{s_i}^{-1} \tag{4.1c}$$

An important element to consider when using algorithm 1, is how the pairs of wrist and sensor displacements are measured, and how these measurements influence the accuracy of the solution. The pairs are initially measured as two wrist and two sensor poses for each pair $(A, B)_i$. These poses are then used to compute their consecutive displacements from (4.1), where T_{w_i} are the wrist poses and T_{s_i} are the sensor poses of measurement i.

$$\mathbf{T}_s = \mathbf{T}_w \mathbf{X} \tag{4.2}$$

It does not matter in which reference frame the measured poses are expressed in (except itself), provided that the wrist and sensor poses are expressed in their respective inertial frames for all poses. The sensor poses are also constrained by (4.2), as a result of the sensor being firmly attached to the robot's wrist. It is therefore only our choice of wrist poses that can influence the accuracy of the solution. Tsai and Lenz [18] showed the following steps to improve the accuracy of hand-eye calibration:

1. Maximize rotations between each consecutive pose
2. Minimize distance between the sensor and its tracking system
3. Minimize translations between each consecutive pose
4. Calibrate the sensor
5. Calibrate the robot

The wrist poses should therefore be chosen such that consecutive rotations are maximized, and consecutive translations are minimized for the wrist and sensor. It is not surprising that the translations should be minimized, as it is a necessary condition for the exact and optimal result in (3.5). The other steps, specifically robot calibration, is outside the scope of this text, and the VIVE tracking system was fixed and calibrated as shown in chapter 2.

In order to measure the pairs of wrist and sensor poses, a VIVE controller was attached to the robot's gripper as shown in figure 4.1. These poses were measured by using tf2 - the transform library in ROS. The wrist poses were received with respect to the robot's base coordinate system, and the sensor poses were received with respect to the tracking system's inertial frame, which coincides with the master base station. Two different methods were implemented as ROS nodes, for measuring the necessary poses and interacting with the *add_sample* and *compute_transform* services:

Method 1: Manual measurements

The manual method is intended for the user to move the robot manually, and sample poses with the VIVE controller. Grip and menu buttons on the controller are mapped to the *add_sample* and *compute_transform* services respectively. The grip button is located on both sides of the controller's handle, and the menu button is on top of the controller's front face. Calibration is performed by manually moving the robot, and the grip button is pressed to measure and sample the necessary poses. The calibration is then completed by pressing the menu button, which automatically updates the configuration of the *vive_bridge* package.

Method 2: Automated measurements

The current implementation of the automated method is based on randomly generated wrist poses. These poses are generated from the normal vector of a sphere with radius $r \in [r_a, r_b)$, polar angle $\theta \in [\theta_a, \theta_b)$ and azimuthal angle $\phi \in [\phi_a, \phi_b)$ drawn from uniform distributions. Two random poses are generated for each wrist pose, where the first pose is used as translation and the second pose is used as orientation. The robot is moved to n randomly generated wrist poses, and the necessary poses are sampled after each move is finished. The calibration is then completed automatically after sampling n points.

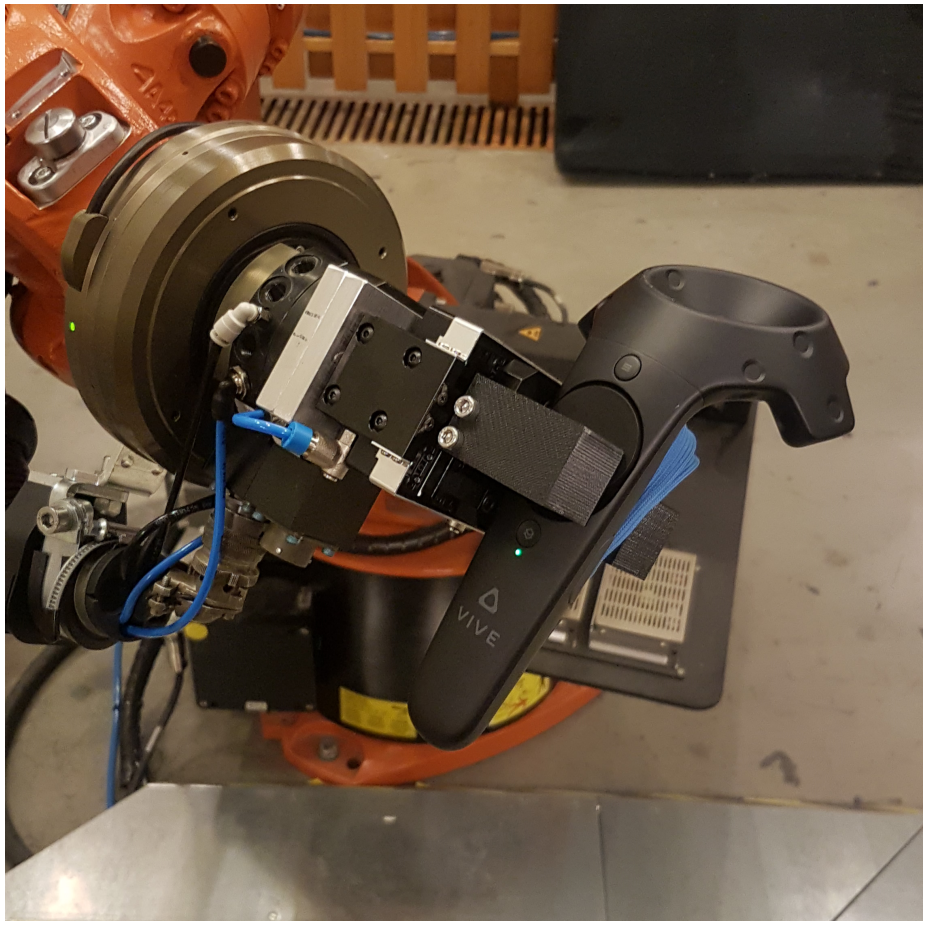


Figure 4.1: VIVE controller firmly attached to the robot's gripper

4.2 MoveIt! issues

The automated method uses the MoveIt! setup for Thrivaldi to move the robot. This setup currently has two problems that causes the robot to stop abruptly while executing trajectories:

- Software end stops in the robot (safety limits for joints)
- Speed singularity when stopping

The online documentation for Thrivaldi states that the joint limits for MoveIt! should be 5° stricter than the software end stops in the robot [29]. These joint limits are not relaxed in the current MoveIt! setup, and causes the robot to stop when the end stops are reached. The robot has to be manually moved away from the end stops whenever one is reached. This problem was solved by simply relaxing the joint limits in the MoveIt! setup.

The speed singularity occurs when the robot is about to stop, and seems to be caused by a discrepancy between the robot and the computer controlling it. An error message is displayed on the robot with the text; "COMMAND VELOCITY EXCEEDED A4 (9243 %)". This error also occurs when the trajectory is parameterized to be slow (10% of normal velocity), which suggests that the robot thinks that it has stopped when it is still moving. The discrepancy could be caused by different ramp down\goal tolerances between the robot and the MoveIt! setup. This hypothesis was not tested due to time constraints, and will have to be assessed in future work.

4.3 VIVE-Thrivaldi calibration

The VIVE calibration for Thrivaldi was performed by using the manual method from section 4.1. This method was chosen because of the MoveIt! problems described in section 4.2. A relatively small number of samples, $n = 11$, was therefore used for the manual calibration. Figure 4.2 shows a visualization of the robot cell after this calibration was performed.

Park and Martin [1] performed simulations with small amounts of added noise (0.7 %) to show the performance of their algorithm. These simulations suggests an error of 0.2 % in the translation for $n = 11$ samples. The performance of the calibrated system will be evaluated further in chapter 5.

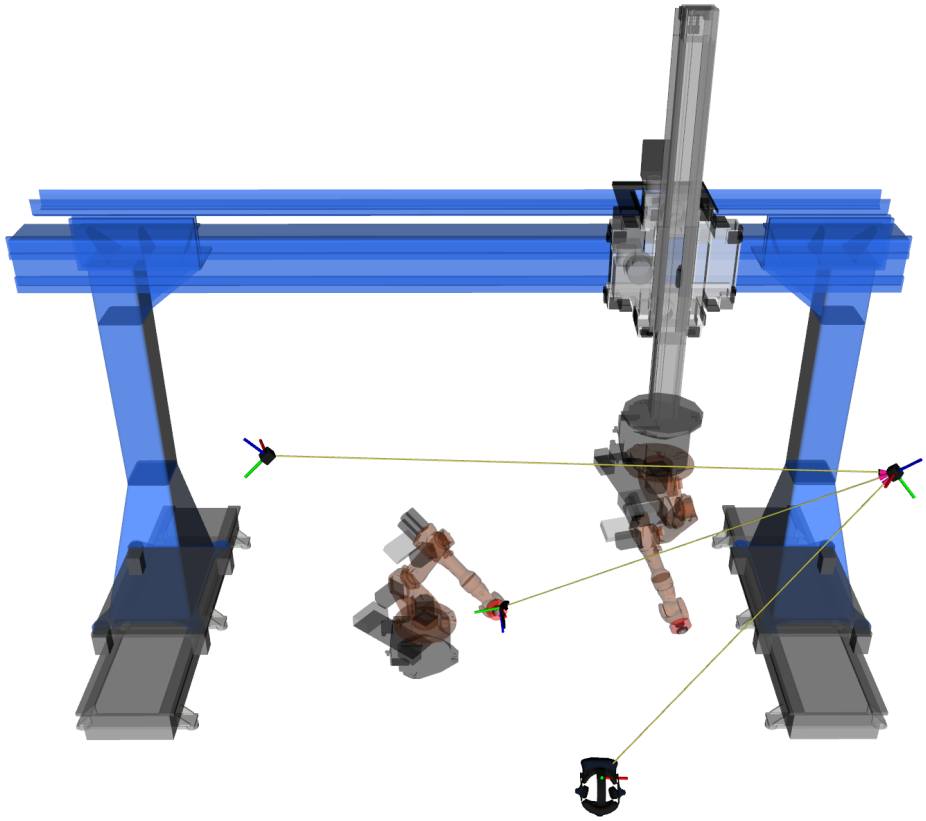


Figure 4.2: Visualization of the robot cell after calibration

Chapter 5

Evaluation, future work and conclusion

The initial plan was to evaluate the calibrated system by performing sets of automated measurements with the robot, where the VIVE controller is kept fixed in the robot's gripper after calibration. These sets would then be measured at points in a 3D grid, similar to the 2D grid measurements by Niehorster et al. [7]. However, the measurements were not automated because of the MoveIt! problems described in section 4.2. A simple test of arbitrary robot displacements was therefore performed instead, in order to show the dynamic behaviour of the calibrated system.

The $\hat{X}$ transformation from the calibration was used together with the forward kinematics of the robot, in order to compute the controller pose. This pose can then be assumed to be an approximation of the ground truth, which based on the robot's repeatability of 0.1 mm indicates the calibrated system's performance. Under this assumption, the error is given by the measured controller pose from the tracking system with respect to the ground truth. This error in translation and orientation is shown for arbitrary robot displacements in figure 5.1 and 5.2 respectively, where Φ_6 is a metric for 3D rotations that is given by [30]:

$$\Phi_6(\mathbf{R}_1, \mathbf{R}_2) = \|\log(\mathbf{R}_1, \mathbf{R}_2)\| \tag{5.1}$$

The arbitrary robot displacements in figure 5.1 and 5.2 are separated by red lines. Each displacement consists of a translation in the range of 1-2 m, and a basic rotation in the range of 45-90°. These displacements shows an indication of the tilted reference plane that was mentioned in section 1.1.1. In other words, the error depends on the location of the tracked device. This spatial dependency is shown as a steady-state error or bias that changes for each dis-

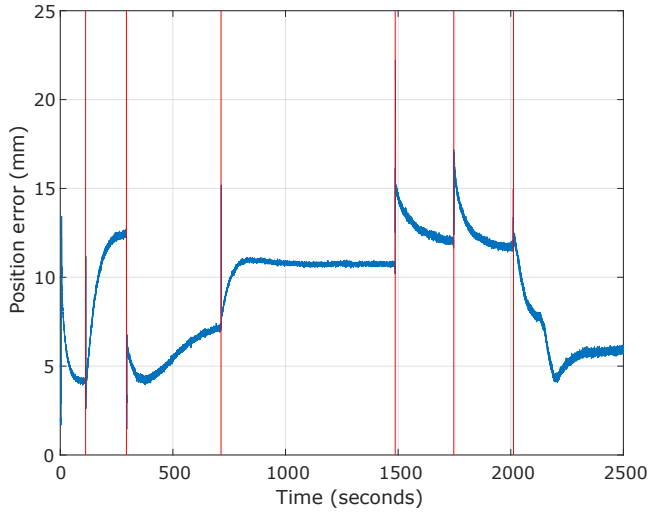


Figure 5.1: Absolute error in the controller translation for arbitrary displacements.

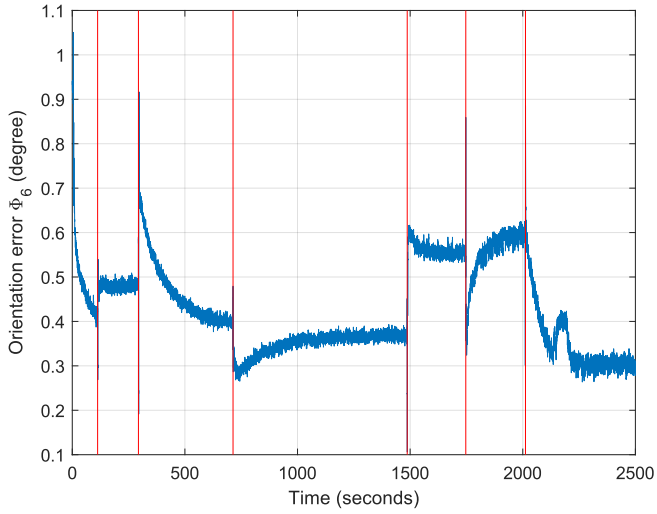


Figure 5.2: Absolute error in the controller orientation for arbitrary displacements.

placement. Transformation (3.13) is therefore only valid for the location where the calibration was completed.

The error dynamics in figure 5.1 and 5.2 shows approximately first order responses with time constant $T \approx 25.5s$. These responses have a steady-state error within the robot's workspace, in the range of 0.5 to 1.5 cm and 0.2 to 0.7 degrees for translations and orientations respectively. Niehorster et al. [7] showed that the accuracy is in the mm range if the tilted reference plane is taken into account. It should therefore be possible to fit the steady-state error of floor measurements to a plane. The main challenge is then to make the procedure of fitting the plane easy and quick for the user, but this challenge will have to be assessed in future work.

5.1 Future work

The calibration procedure presented in this text has a few problems that have a negative impact on the calibrated system. A relatively small number of samples, $n = 11$, was used to manually calibrate the system. The simulations by Park and Martin [1] suggests that this number of samples gives an error of 0.2% in translation. This error can be reduced by using more samples, but the problems in section 4.2 has to be fixed in order to be practical.

Another interface called KUKAVARPROXY (KVP) can be used instead of the MoveIt! setup to control the robot. More importantly, it is important to use a robot interface to properly evaluate the calibrated system. An automated calibration method should also follow the steps by Tsai and Lenz [18] in section 4.1 to improve the accuracy. It should also be possible to estimate the tilted reference plane described by Niehorster et al. [7] from automated measurements. Taking this plane into account can potentially improve the accuracy of the calibrated system by an order of magnitude. Another option could be to make use of libsurvive - an open-source alternative to the OpenVR SDK [31].

Furthermore, there are three main directions that this project can branch into:

1. Robot cell calibration system
2. Real-time tracking for robotics
3. Robot calibration of Thrivaldi

The HTC VIVE has an update rate of 220-360 Hz, which is sufficient for robotics applications. However, Borges et al. [8] concluded that the VIVE's algorithm was unsuited for robotics, because it weighted inertial measurements more to produce smooth trajectories. They also introduced their own algorithm in order to improve the VIVE's dynamic accuracy [9]. This algorithm was implemented by using the libsurvive library. An opportunity can therefore be to implement this library in the vive_bridge node, and try different tracking algorithms. It is also possible to custom fit robots with sensors for the VIVE tracking system. The Shoto SteamVR tracking hardware developement kit (HDK) from Triad Semiconductor can be used to quickly prototype tracked objects [32].

The calibrated system can currently be used to place objects with cm accuracy. This accuracy can be sufficient for calibration purposes, depending on the application. The HTC VIVE can be used to place collisions in a robot cell for instance, and the main challenge of such an application is how the robot cell should be represented. Robot calibration of Thrivaldi is the least likely continuation of this project, because it requires a very accurate tracking system.

5.2 Conclusion

The main goal of setting up and calibrating an HTC VIVE for Thrivaldi was achieved. A calibration procedure was established based on an algorithm by Park and Martin [1], in order to achieve this goal. Figure 4.2 shows the robot cell after the calibration was performed, and it is hard to distinguish between the real and virtual robot cell with the naked eye. The calibrated system has a steady-state error within the robot's workspace, in the range of 0.5 to 1.5 cm and 0.2 to 0.7 degrees for translations and orientations respectively. This error was computed based on a simple test of arbitrary robot displacements. The results are therefore not conclusive on the VIVE's accuracy, and the error can potentially be reduced by an order of magnitude. A tilted reference coordinate system turned out to have the greatest negative impact on the VIVE's accuracy.

Bibliography

[1] F. C. Park and B. J. Martin, "Robot sensor calibration: solving ax= xb on the euclidean group," *IEEE Transactions on Robotics and Automation*, vol. 10, no. 5, pp. 717–721, 1994.

[2] HTC Corporation, "Think vr is dying? it's just getting started," 2018, [Online; accessed 17-November-2018]. [Online]. Available: https://blog.vive.com/us/2018/07/26/think-vr-dying-just-getting-started/

[3] Open Source Robotics Foundation, "About ros," [Online; accessed 9-November-2018]. [Online]. Available: http://www.ros.org/about-ros/

[4] ——, "Is ros for me?" [Online; accessed 9-November-2018]. [Online]. Available: http://www.ros.org/is-ros-for-me/

[5] J. Steuer, "Defining virtual reality: Dimensions determining telepresence," *Journal of communication*, vol. 42, no. 4, pp. 73–93, 1992.

[6] nairol, "Lighthouse reverse-engineered documentation," 2017, [Online; accessed 22-November-2018]. [Online]. Available: https://github.com/nairol/LighthouseRedox/blob/master/docs/Light%20Emissions.md

[7] D. C. Niehorster, L. Li, and M. Lappe, "The accuracy and precision of position and orientation tracking in the htc vive virtual reality system for scientific research," *i-Perception*, vol. 8, no. 3, p. 2041669517708205, 2017.

[8] M. Borges, A. Symington, B. Coltin, T. Smith, and R. Ventura, "Htc vive: Analysis and accuracy improvement."

[9] NASA, "Astrobee robot software," 2018, [Online; accessed 18-November-2018]. [Online]. Available: https://github.com/nasa/astrobee

[10] Oliver Kreylos, "Lighthouse tracking examined," 2016, [Online; accessed 18-November-2018]. [Online]. Available: http://doc-ok.org/?p=1478

[11] HTC Corporation, "Vive pro," 2018, [Online; accessed 9-November-2018]. [Online]. Available: https://www.vive.com/eu/product/vive-pro/

[12] InvenSense, "Mpu-6500 | tdk," 2018, [Online; accessed 9-November-2018]. [Online]. Available: https://www.invensense.com/products/motion-tracking/6-axis/mpu-6500/

[13] I. Eriksen, "Setup and interfacing of a kuka robotics lab," Master's thesis, NTNU, 2017.

[14] NTNU, "itk-thrivaldi documentation," 2017, [Online; accessed 9-November-2018]. [Online]. Available: https://github.com/itk-thrivaldi/Documentation/wiki

[15] Open Source Robotics Foundation, "Core components," [Online; accessed 12-November-2018]. [Online]. Available: http://www.ros.org/core-components/

[16] HTC Corporation, "Tips for setting up the base stations," 2018, [Online; accessed 14-November-2018]. [Online]. Available: https://www.vive.com/eu/support/vive-pro-hmd/category_howto/tips-for-setting-up-the-base-stations.html

[17] ——, "Setting up a room-scale play area," 2018, [Online; accessed 18-November-2018]. [Online]. Available: https://www.vive.com/eu/support/vive-pro-hmd/category_howto/setting-up-room-scale-play-area.html

[18] R. Y. Tsai and R. K. Lenz, "A new technique for fully autonomous and efficient 3d robotics hand/eye calibration," *IEEE Transactions on robotics and automation*, vol. 5, no. 3, pp. 345–358, 1989.

[19] K. Ikeuchi, *Computer vision: A reference guide*. Springer Publishing Company, Incorporated, 2014.

[20] Y. C. Shiu and S. Ahmad, "Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form ax=xb," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 1, pp. 16–29, Feb 1989.

[21] H. H. Chen, "A screw motion approach to uniqueness analysis of head-eye geometry," in *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*. IEEE, 1991, pp. 145–151.

[22] K. Daniilidis, "Hand-eye calibration using dual quaternions," *The International Journal of Robotics Research*, vol. 18, no. 3, pp. 286–298, 1999.

[23] M. Feuerstein, "Hand-eye calibration," 2009. [Online]. Available: http://campar.in.tum.de/view/Chair/HandEyeCalibration#MATLAB

[24] B. Kenwright, "A beginners guide to dual-quaternions: what they are, how they work, and how to use them for 3d character hierarchies," 2012.

[25] D. Condurache and V. Martinusi, "Computing the logarithm of homogenous matrices in se (3)," 2005.

[26] A. Nadas, *Least Squares and Maximum Likelihood Estimates of Rigid Motion.* IBM Thomas J. Watson Research Division, 1978.

[27] Open Source Robotics Foundation, "Understanding ros services and parameters," 2017, [Online; accessed 9-November-2018]. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

[28] ——, "Writing a simple service and client (python)," 2017, [Online; accessed 9-November-2018]. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

[29] NTNU, "itk-thrivaldi joint limits," 2017, [Online; accessed 15-November-2018]. [Online]. Available: https://github.com/itk-thrivaldi/Documentation/wiki/2.2-Joint-limits

[30] D. Q. Huynh, "Metrics for 3d rotations: Comparison and analysis," *Journal of Mathematical Imaging and Vision*, vol. 35, no. 2, pp. 155–164, 2009.

[31] cnlohr, "Lightweight htc vive library," 2018, [Online; accessed 18-November-2018]. [Online]. Available: https://github.com/cnlohr/libsurvive

[32] Triad Semiconductor, "Shoto steamvr tracking hdk," 2018, [Online; accessed 18-November-2018]. [Online]. Available: https://www.triadsemi.com/product/steamvr-tracking-hdk/

Appendix A

VIVE Bridge documentation

vive_bridge

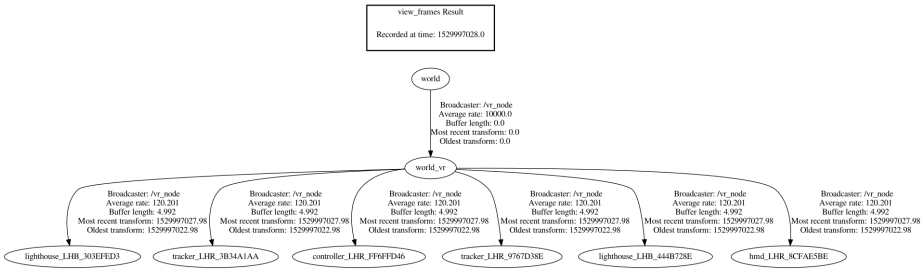
vive_bridge is a Robotic Operating System (ROS) package that utilises the OpenVR SDK by Valve, to make VR devices such as the HTC VIVE available in a ROS environment. The package is inspired by an existing vive_ros package by RoboSavvy, and it exposes a lot of the same functionality. The essentials of the OpenVR SDK is explained in great detail in the OpenVR Quick Start guide by Kevin Kellar. The guide is also saved locally in this package under the `doc/CassieVrControls.wiki` folder.

Features

The package supports the following types of devices:

- HMD (Head-Mounted Display)
- Controller
- Tracker
- Lighthouse

The package exposes the position and orientation (pose) of each device as coordinate frames relative to the *world_vr* frame in the tf tree, which is configurable relative to the *world* frame. The naming scheme of each coordinate frame follows the following structure: <device type>_<serial number>, e.g. *controller_LHR_FF6FFD46*. The serial number is used both internally and externally (in the package) to uniquely identify tracked devices. This results in a structure similar to the tf tree example that is shown below:



The package can also publishes the linear and angular velocities (twists) of tracked devices as a geometry_msgs/TwistStamped message on the */vive_node/twist/<device type>_<serial number>* topic, e.g. */vive_node/twist/controller_LHR_FF6FFD46*. Axes and buttons on controllers can also be published as a sensor_msgs/Joy message on the */vive_node/joy/<device type>_<serial number>* topic, e.g. */vive_node/joy/controller_LHR_FF6FFD46*. Joy messages are only published when the controllers are interacted with. These publishers are not enabled by default, but they are easily enabled during runtime by using the rqt_reconfigure package.

Visualization

It is also possible to visualize the tracked devices by using a MarkerArray display in RViz. The mesh files are defined in the `launch/vive_node.launch` file as parameters for each type of device:

- hmd_mesh_path

- controller_mesh_path
- tracker_mesh_path
- lighthouse_mesh_path

The tracked devices are then visualized by adding `/vive_node/rviz_mesh_markers` as *Marker Topic* in a MarkerArray display.

The mesh files has to be supported by RViz, i.e. .stl, .mesh (Ogre) or .dae (COLLADA).

Requirements

OpenVR SDK

The package requires the OpenVR SDK, which has to be built from the newest available source. It is possible to download and build the source in the correct folder by utilising the following commands:

```
cd ~
mkdir lib
cd lib
git clone https://github.com/ValveSoftware/openvr.git
cd openvr
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make
```

It is also possible to specify which folder the OpenVR SDK should be located in, by changing the *CMakeLists.txt* file in the package directory:

```
set(OPENVR "$ENV{HOME}/lib/openvr")
```

Steam and SteamVR

SteamVR is available through Steam, which is utilised for configuration and room setup. It is also required for running this package by itself, as it depends on the *vrserver* process running in the background. This is a requirement because the OpenVR part of the package runs as a background application (OpenVR API Documentation):

`VRApplication_Background` - The application will not start SteamVR. If it is not already running the call with VR_Init will fail with VRInitError_Init_NoServerForBackgroundApp.

Steam is installed by following the *Getting Started* guide on their Steam for Linux tracker. SteamVR should be installed automatically by Steam if there is any VR devices present on your computer. It is also important to meet the **GRAPHICS DRIVER REQUIREMENTS** and the **USB DEVICE REQUIREMENTS** on their SteamVR for Linux tracker. A complete guide on getting the HTC VIVE up and running in SteamVR is available from: HTC Vive Installation Guide.

Installation

The package is built by cloning this repository into your catkin workspace (catkin_ws/src directory) and then making it with `catkin_make`

Usage

The package is simply run by launching the following launch file: `roslaunch vive_bridge vive_node.launch`

You may have to change the directory paths for Steam and your Catkin workspace in the `/scripts/launch.sh` shell script depending on their location. The package assumes that the directories are in their defuault locations.

```
STEAM_RUNTIME=$HOME/.steam/steam/ubuntu12_32/steam-runtime
CATKIN_WS=$HOME/catkin_ws
```

Applications are generally run through the Steam runtime by running the `run.sh` script. The script is in the steam-runtime folder, and takes the application as an argument for the script.
`~/.steam/steam/ubuntu12_32/steam-runtime/run.sh ~/catkin_ws/devel/lib/vive_bridge/vive_bridge_node`

Interacting with the node

The *vive_node* publishes information about the currently tracked devices to the `/vive_node/tracked_devices` topic. This topic uses a custom `vive_bridge/TrackedDeviceStamped.msg` message that contains information about:

- frame_id - Fixed VR frame (within the message header)
- uint8 device_count - Number of tracked devices
- uint8[] device_classes - Classes of tracked devices (classes are defined within the message):

```
uint8 HMD=1
uint8 CONTROLLER=2
uint8 TRACKER=3
uint8 LIGHTHOUSE=4
```

- string[] device_frames - Child frames associated with each tracked device

The frame names within the `device_frames` field can then be used to find the joy and twist topics of each tracked device, e.g. the twist topic of the first tracked device could be:

`"/vive_node/twist/" + msg_.device_frames[0]`

It is also possible to request this information from the `/vive_node/tracked_devices` service, which requests a *std_msgs/Empty* message, and responds with the same format as the `vive_bridge/TrackedDeviceStamped.msg`. The frame_id is however included as it's own field in the response, instead of being included in the message header.

Controller axes and buttons

The package currently supports all inputs from the HTC VIVE controller, and the sensor_msgs/Joy messages have the following format:

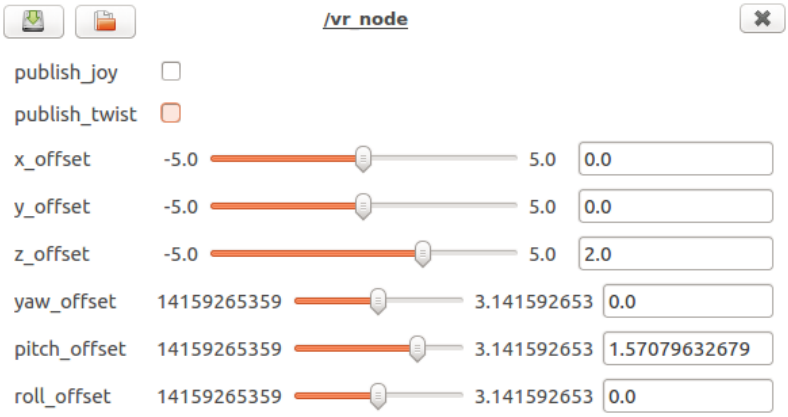
- axes[0] - Trackpad x
- axes[1] - Trackpad y
- axes[2] - Trigger
- buttons[0] - Menu
- buttons[1] - Grip
- buttons[2] - Trackpad
- buttons[3] - Trigger

The package also emulates a numpad when pressing different points on the trackpad button:

- buttons[4] - 1 Left down
- buttons[5] - 2 Center down
- buttons[6] - 3 Right down
- buttons[7] - 4 Left center
- buttons[8] - 5 Center
- buttons[9] - 6 Right center
- buttons[10] - 7 Left up
- buttons[11] - 8 Center up
- buttons[12] - 9 Right up

The x and y values from touching the trackpad is used to find the corresponding numpad key. The header also contains the frame_id associated with the tracked device.

Configuration



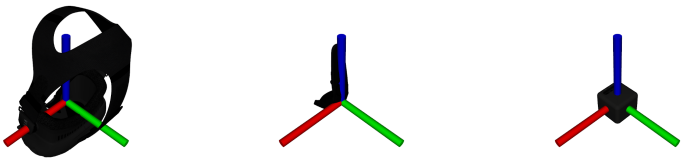
The position and orientation (pose) of each device is defined relative to the *world_vr* frame, which has the same position as one of the lighthouses. This frame has to be defined relative to some defined *world* frame to make sense of the environment. The transformation between these frames are exposed as x-, y-, z- and roll-,

pitch-, yaw- (RPY) offset parameters by the dynamic_reconfigure package. This package provides a standard way to change the offset parameters at any time without having to restart the node, and also provides a graphical user interface (GUI) to change these parameters by using rqt_reconfigure:

`rosrun rqt_reconfigure rqt_reconfigure`.

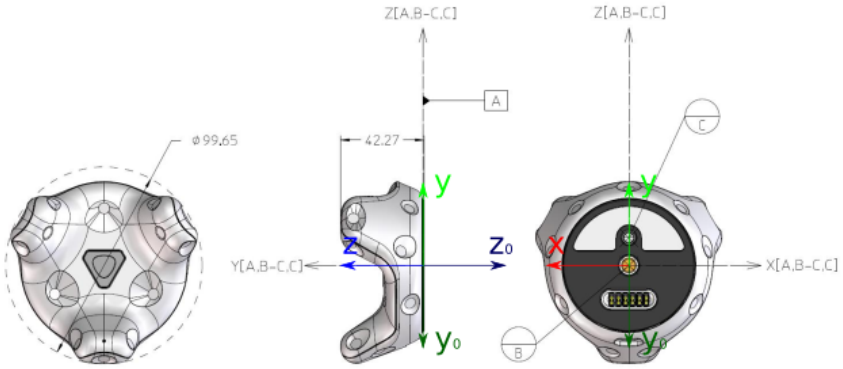
The parameters from dynamic_reconfigure are currently not saved automatically, and they therefore have to be updated manually in the `/launch/vive_launch.launch` file (see param), and in the `/cfg/DynReconf.cfg` file (see How to Write Your First .cfg File).

Coordinate systems



Tracked devices follows the following coordinate system conventions:

- X-axis equates to pitch
- Y-axis is up and equates to yaw (except for the VIVE Tracker, which has Z-axis down)
- Z-axis is approach direction and equates to roll (except for the VIVE Controller, which has Z-axis pointing the opposite way)



The VIVE Tracker coordinate system is rotated 180° around the x-axis such that the z-axis points upwards. This is because we want the tracker to match the orientation of our reference frame (world), when it is placed horizontally on the ground.

Compatibility

The package was tested with:

- HTC VIVE with OpenVR SDK 1.0.15 and Ubuntu 16.04 LTS running ROS Kinetic Kame (1.12.13)

To-do list

- Save and load the parameters that are changed by dynamic reconfigure
- Implement libsurvive - lightweight HTC Vive library as an alternative interface to the OpenVR SDK
- Implement handling of tracked devices with virtual functions for drop-in support of alternative interfaces
- Implement haptic feedback on the VIVE Controllers