FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Automation and Signal Processing	Spring semester, 2017 Open / ~~Confidential~~
Author: Kristian Sletten	.... (signature author)

Instructor: Morten Mossige

Supervisors: Ståle Freyer
Karl Skretting

Title of Master's Thesis:
Automated testing of industrial robots using HTC Vive for motion tracking

Norwegian title:
Automatisk testing av industriroboter med bruk av HTC Vive for bevegelses deteksjon

ECTS: 30

Subject headings: Industrial robots, ABB, HTC Vive, Automated testing, Motion tracking,	Pages: 57 + attached files: Source code, documentation, demonstration video Stavanger,15.06.2017

Abstract

Producers of industrial robots are continuously maintaining and upgrading their software that is running on their robot controllers, to improve and expand the functionalities. Extensive testing is performed to verify that all new and existing features are working as intended before this software can be released. These tests are time consuming and requires an operator to perform, observe, and inspect the results. A failure during these tests can result in very time-consuming troubleshooting, as the error might be caused by changes done several months earlier.

This project has developed a base for automating these tests to reduce the duration of performing the tests. Further, these tests have been designed to be performed at a daily basis, continuously testing the new software while its being developed. These tests will give an early indication of errors that can be rectified at a much earlier stage than during the final tests.

To be able to verify the robot's operations, an external system that can track the robot's motion is required. There are existing systems that can perform this, but they have limited functionalities and often have a very high price, making them undesirable. This project has been challenged with using the virtual reality system HTC Vive, which have shown promising results on its precision on tracking its devices in a three-dimensional space, which have been verified through this project.

This equipment has been used to create a tracker API that uses the HTC Vive as a motion tracking device, and tracks the robot's motion. The tracker captures the robot's path and the tests use this to verify correct operation of the robot's controller software.

The solution in this project have shown that some of the existing tests can be automated, but there are limitations to the HTC Vive that does not give good enough tracking to verify all operations. The solution does however work well on several aspects concerning continuous testing, and will work as a good tool for testing the current state of the development, and give an early warning of failures or errors that might be introduced. The solution has room for further implementations, and there are several more tests that can be created to further improve the testing capabilities.

Contents

1 Introduction **1**

1.1 Testing of industrial robots 1

1.1.1 Tracking the motion of industrial robots 1

1.1.2 Automate testing of industrial robots 2

1.1.3 Defining the tasks 2

1.2 Structure of this project report 3

2 Prerequisites **4**

2.1 Externally tracking of robot 4

2.1.1 The HTC Vive Virtual Reality System 5

2.1.2 Precision of tracked data 8

2.1.3 The OpenVR API 8

2.2 Industrial Robot Testing 10

2.2.1 Test parameters based on ABB's existing test cases 10

2.2.2 The ABB Test Engine and RobApi2 12

2.3 The robot setup 13

2.4 Workign With ABB's Robots 13

2.5 Development environment 14

3 Implementation **15**

3.1 System Overview 15

3.2 Implementing The Tracker API 16

3.2.1 Structure of the Tracker API 16

3.2.2 Representing a robot path 17

3.2.3 Using the Tracker API 18

3.2.4 Implementation of OpenVR 22

3.3 Using the Tracker With the Industrial Robot 23

3.4 Automated Test Cases 24
3.4.1 General Test Implementation 24
3.4.2 Preliminary (ATC-0) 25
3.4.3 General movement (ATC-1) 26
3.4.4 Speed (ATC-2) 29
3.4.5 RAPID Operations (ATC-6) 31
3.4.6 Trigger (ATC-7) 31
3.4.7 ViveTests 32

4 Resulting System 33

4.1 Tracking Precision for The HTC Vive 33
4.1.1 Positional Jitter 33
4.1.2 Positional Drifting Over Time 33
4.1.3 Precision of Tracking Speed 34
4.2 Testcase Performance 36
4.2.1 Preliminary 36
4.2.2 General movement 37
4.2.3 Speed 39
4.2.4 RAPID Operations 40
4.2.5 Trigger 41

5 Discussion 42

5.1 HTC Vive as a Tracking Device 42
5.1.1 System Jitter 42
5.1.2 Position Drift and Changing Coordinates 42
5.1.3 Interference From The Robot 43
5.1.4 Speed 44
5.1.5 Tracking resolution regarding refresh rate 44
5.1.6 Improving the Quality of the HTC Vive Tracking 45
5.2 Quality of the Tracker API 46

5.2.1 Tracker Use Cases 46

5.2.2 Calibrating the Tracker 47

5.2.3 Capturing Digital Inputs with the Tracker 48

5.3 Automated Testing of Industrial Robots 49

5.3.1 Evaluating the new Tests 49

5.3.2 More Potential Test Cases 51

5.3.3 Comparing the new Test Method with Existing Method 52

5.4 Further Work on the Tracker 53

6 Conclusion **54**

A Attachments **57**

Chapter 1

Introduction

1.1 Testing of industrial robots

During development of new controller software for industrial robots, extensive testing is performed to test new functionality, and to ensure the integrity of existing functionality. These tests are quite extensive, covering all the various aspects the robot's functionality, that all must be tested to verify correct operation of the new controller software before it can be shipped.

These tests are mostly manual tests that must be performed by an operator that has to prepare and set up the test equipment. When the tests are being executed, the operator is required to observe the test verifying correct operation during the tests. When the test is done the results must be checked and verified against the required criteria. These tests are time-consuming, and the requirement of an operator limits the testing to daytime during work hours.

When this controller software is being developed, there are usually many involved parties working on the project, and often on different parts of the software. This may often result in the different parts of new features not being properly tested together before the end of development due to the time it takes to perform these tests. If anyone of these tests fail, giving an error or incorrect operation, the cause might be changes done several months prior to testing, making troubleshooting complicated and complicated. These errors will take time to debug and fix, and all tests must be repeated to verify the integrity of the software. Performing these tests and correcting all problems can take several weeks before the software is ready to be released, and be a costly procedure.

This project is tasked by ABB to create a solution for performing automated testing of ABB's industrial robots, without requiring an operator during testing. These tests should be able to be executed frequently, such as during night time and weekends, and work as a solution for continuous integration during development. Mainly this project is target towards the tests regarding the robot's movement, but also extends to various other functionalities that can be integrated into the system.

1.1.1 Tracking the motion of industrial robots

Automating these tests requires an external system that can independently track and record the robot's movement. Tracking systems for robots exists, but usually they have limited degrees of freedom, or they are very expensive. As a cheaper alternative, this project is challenged with using a HTC Vive for tracking.

The HTC Vive is a virtual reality system designed for gaming and other entertainment, which have a price that is within consumers price range. The HTC Vive is delivered with hand controllers used for interacting with the virtual environment, where both position and orientation are tracked. The device tracking for the HTC Vive have been shown to have good precision in space, which is desirable to utilize for this project [1].

1.1.2 Automate testing of industrial robots

ABB want to automate these tests with two goals in mind. Replace the existing testing with more automated solution to reduce the test duration before software releases. And secondly to perform continuous integration during development, by performing simpler automated testing to detect errors and flaws early in the development cycle.

ABB have provided test documents for the existing tests that are relevant for this solution, which should be the basis of the new automated tests. This does not cover all tests being performed prior to release, only tests related to the robot's motion is of interest for this project. The existing tests are mostly based on manual inspection, such as inspecting drawings made by the robot, and observing the movement during testing. With the use of equipment that can track the robot in more than in one plane, new tests can be created with more complex movement, and more thorough tests.

A test engine framework has been provided by ABB that the new tests should be implemented into, that allows the tests to be integrated into their existing testing system. This test engine has predefined functionalities for creating test logs that displays the results, and reports warnings and errors.

1.1.3 Defining the tasks

This project has two parts to the solution that must be implemented to create a solution for automated testing of these industrial robots.

First, using the HTC Vive for capturing the motion of the robot, a tracking system must be created that can extract the data from the Vive hand controllers, and represent that motion as a recorded path. This path must be able to represent the position and orientation of the robot's motion. To verify the quality of the tracking, the solutions must be tested to determine what precision can be expected for the tests.

Second, the automated tests should be based on the existing tests, and be integrated into the provided test engine. It must be determined to what extent the new test is capable to replace the existing tests, and how they could improve the testing. The tests should be designed to be able to be performed frequently during development and expose faults and errors at an early stage.

A complete solution for this project would be far more comprehensive than what is realistic to complete within this project alone. Therefore, this project will focus on creating the basis for tracking with the HTC Vive and to testing the quality of the Vive as tracking

equipment. The automated tests to implement should cover most aspects of possible test areas, to determine if, and how these are working with the tracker.

1.2 Structure of this project report

This project report is divided into six chapters. The first chapter is this introduction covering the reasoning for the project and outlining the tasks. Chapter 2 covers the prerequisites for this project, explaining all technologies and existing solutions used in this project. Chapter 3 explains the implementation of the different solutions that are part of this project. Chapter 4 is presenting all results from testing of the system, and shows how well the solutions worked. Chapter 5 discusses the results, and goes through the solutions explaining the quality, challenges, and how they can be improved for further use and implementation. Chapter 6 presents the conclusion for how the solution worked, and how well the problem presented have been solved.

Chapter 2

Prerequisites

The implementation of the automated test procedure, is based on prerequisites that is both predefined as part of the assignment, and some based on the selected technology for this assignment. This chapter explains these prerequisites, technologies, and existing solutions used in this project.

2.1 Externally tracking of robot

When testing correct operation of a robot's movement, an external system for verifying the motion is required, as the robot system cannot be used to verify its own operation. In the current test cases for ABB's robots, this is done with tracing the movement with a pen on paper, verifying the result by visual inspection and observation. To be able to automate these tests as desired for this project, requires a system that can track the robot's movement and communicate its results into a computer and run together with the test system.

There are systems existing for tracking motion with good precision, but they usually have a very high price tag. [2] Evaluated a range of relative cheap methods for testing positional repeatability for industrial robots. Using cable trilateration where described as one option, where three cables are connected to the end of the robot and each cable feeds through a cable pulley with an encoder to measure the length of all three cables. Another method using a displacement gauge mounted to the robot that measures the distance to a reference surface while moving along that surface, is a method used to verify straight movement. This method is limited to only movements where a reference surface is mounted, and nonlinear movements is not well supported.

Virtual Reality Systems

A virtual reality system allows the user to step into a virtual world by using a headset with a stereoscopic display, showing a virtual environment. The more advanced virtual reality headsets detect the head movements of the user, allowing them to look and move around in the virtual world. Some of the high end virtual reality systems have hand controllers, extending the users experience of the virtual world, by allowing them to interact with the virtual environment. To achieve these functionalities, the position and orientation of these devices must be known in relation to each other, and the room the user are in.

The high precision of these virtual reality hardware, and relative low cost, have made it desirable for other applications [1]. There are several industries that are adopting the

possibilities with virtual reality, for example improving the visualization of a finished product before any physical objects have been created [3].

Prior to this project a HTC Vive set, one of the more popular virtual reality hardware, have been used by ABB for other related projects, and through the experience of using the Vive, it was desirable to examine the utilization of this equipment for tracking the movement of a robot. ABB are considering methods to automate their testing of robots, and want to integrate this tracking with their test procedures. A HTC Vive set have been acquired for this project and the idea is to utilize the hand controllers for tracking the robot's movement, by mounting one directly to the robot's tool mount.

2.1.1 The HTC Vive Virtual Reality System

Figure 2.1: The content of a HTC Vive setup; the Head Mounted Device, two hand controllers, and two Lighthouses.[4]

The HTC Vive is a virtual reality system designed by HTC in conjunction with Valve software, created for use with games and other entertainment. This section explains the basics of how the HTC Vive tracks each device, and how the precision is maintained. The HTC Vive is still being further developed and its functionalities might change or improved, therefore the description in this section is based on the current functionalities. There are several articles describing the functionality of the HTC Vive including interviews and lectures form the designer of the Vive Lighthouse system at Valve Software, Alan Yates [5][6].

A HTC Vive kit consists of three main parts. The headset, also referred to as the Head Mounted Device(HMD), which is a head mounted stereoscopic display that provides the user with a stereoscopic image of the virtual environment. Two Vive hand controllers, designed for the user to interact with the virtual world, through tracking movement and button input. Two base stations, called Lighthouses, mounted in the room that aid the tracked device to locate themselves relative to a stationary point.

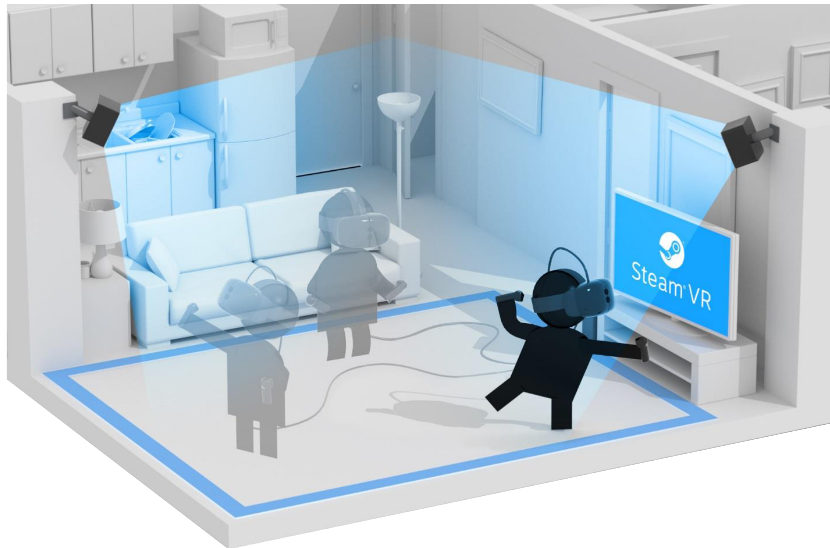

Figure 2.2: Illustrating the HTC Vive used for virtual reality, with its components [7].

2.1.1.1 Determining the pose for each tracked device

Both the head mounted device and the hand controllers are tracked devices, meaning that they can find their position and orientation relative to each other, and relative to the room they are used in. The HTC Vive combines two methods of tracking to determine the actual pose of each device.

Pose from an inertial measurement unit

Each device contains an inertial measurement unit (IMU), which combines the use of accelerometer and gyroscope to determine change in position and orientation of the device. The IMU does not know where in space it is by itself, and only know how it moves by integrating the changes in position and orientation. The movement is susceptible to drifting, and the change in pose determined by the IMU do require to be updated with a position and orientation reference regularly. The inaccuracy in the measurement done by the IMU will accumulate over time, resulting the estimated pose to have an increasing uncertainty. To have a method of correcting the error in the estimate, a stationary reference point is required, which is the purpose of the lighthouse.

Correcting the pose with optical update from Lighthouse

The Lighthouses are mounted stationary in the room where the HTC Vive are in use, and acts as a stationary reference point in the room. The two lighthouses alternate on scanning the room with two infrared(IR) laser lines, one vertical and one horizontal, that the tracked devices uses to determine their pose, and with that correct the estimated pose from the IMU. Figure 2.3 illustrates how the estimated position is corrected with the optical update from the Lighthouses. Note that the graph is only an illustration, and the estimate and timing is not to scale.

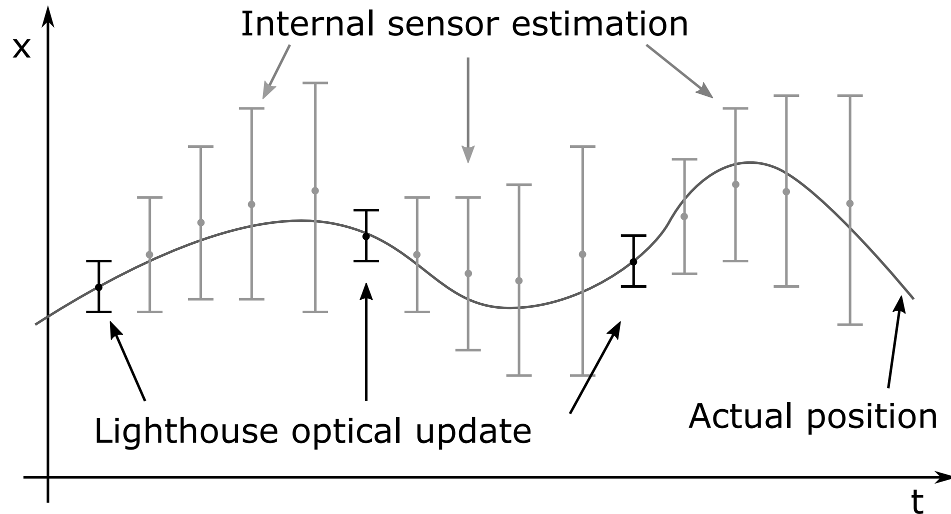

Figure 2.3: The position of a tracked device is calculated by its internal sensor, and the potential error and drift is corrected by the optical update from the Lighthouses. Graph is based on a presentation slide by Alan Yates[5]

To determine the pose of a device based on the infrared scans from the Lighthouses, each device has several separate optical sensors spread around the device, and cover all directions with multiple sensors. These sensors will detect the beams emitted from the lighthouses, and depending on the orientation of the device, the sweeping infrared lines will hit each sensor at different times, and the time difference is used to calculate both orientation and position.

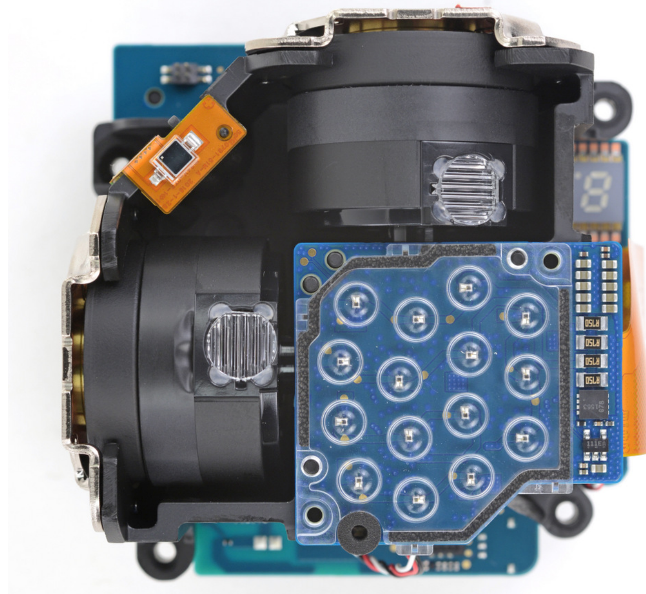

Figure 2.4: The internals of a Vive Lighthouse, showing the two rotating lenses emitting a laser line, and the 15 infrared LEDs for synchronization flash [4].

Each Lighthouse contains one infrared flood light, and two rotating mirrors that projects an infrared laser line, scanning horizontally and vertically from the lighthouse. The flood

light flashes before each scan, and is used to synchronize the timing for each device. The scanners are rotating at a known speed of 60 revolutions per second, and by measuring the time between the flash and the scan hitting each of the sensors on a device, the device can calculate its position and orientation relative to the Lighthouse.

Only one source can emit light at a time, and the two lighthouses are required to synchronize the scans with each other. This synchronization can be done in two different ways depending on the situation. Either by a cable between the two, syncing with an electric signal, or by arranging the two is such a way where they can detect each other's flash, and synchronize optically without any cable connected.

Each scan from a Lighthouse cover a 120° area, and since this sweep is performed within half a rotation, the Lighthouses can perform 120 scans per seconds. Each tracked device will then be able to calculate a new correction to its pose 120 times each second, but each scan will only cover correction in the one direction of the scan.

2.1.2 Precision of tracked data

The performance of the Vive is not specified by HTC, but through testing and digging in the source code, some of its characteristics have been determined in this article [1].

The controllers are shown to update its pose at a rate of 250Hz. While the lighthouse emits scans at 120Hz, this will only update the devices positions in one axis every 8.33ms. As explained in the previous section, the IMU in the device itself will continuously estimate its position and orientation, and update its reference point when detecting a scan from the Lighthouses.

The precision of tracking the controllers, the position at a stationary point have been shown to have a jitter from end to end of 0.3mm. If only one Lighthouse are visible to a device, the system will still work, but this gives the device on axis that is not corrected. This axis is shown to have a jitter of 2.1mm in the direction towards the one active Lighthouse.

2.1.3 The OpenVR API

In conjunction with the release of the HTC Vive, Valve software have released OpenVR, an API allowing developers to interact with several different virtual reality hardware through the same programming interface. The API is a set of virtual functions that have underlying implementations towards different hardware, and when the API is initialized, it detects the type of hardware connected, and initializes the implementation for that specific hardware.

The OpenVR API defines several functionalities for creating virtual reality applications. For this project only functions related to retrieving data from the controllers are in interest. There are only a few functions, and some structures in OpenVR that are required to get the desired data.

GetControllerState() returns the current state of a hand controller, see *VRController-State_t* for details.

GetControllerStateWithPose() outputs the state as *GetControllerState*, but also includes the immediate pose of the controller at the time when the button state was last updated. The purpose of this function is to retrieve the exact pose of the controller at change of state, such as a button press.

GetDeviceToAbsoluteTrackingPose() calculates the pose for each device from their sensors (IMU and Lighthouse tracking). The pose is calculated for the moment the functions is called, and does not just return the last updated pose as *GetControllerStateWith-Pose* does.

GetTrackedDeviceClass() returns the device type for a specific device index.

VRControllerState_t contains the state of all inputs for a controller, including buttons, analog triggers, joysticks, and track pads.

TrackedDevicePose_t is a structure describing the pose of a device. It contains a transformation matrix, the velocity, and angular velocity of the device.

To initialize the OpenVR, one initialization method are called, *vr::VR_Init*. OpenVR allows for starting is different application modes, such as a full 3D scene application, only overlay, and most suitable for this project; in background mode, where the application can run without require starting a rendering window for the display.

When using HTC Vive, OpenVR does not communicate to the hardware directly. SteamVR is the application that connects to the different HTC Vive devices, and maintain their connection and status, and works as a hardware driver for the Vive. When initializing OpenVR, it connects to SteamVR, which then communicates to the devices to retrieve and provide data. This means that the SteamVR application must run in the background when using a OpenVR application with the HTC Vive.

As mentioned, ABB have been using the HTC Vive for other purposes, and a prototype implementation of OpenVR have been provided as an example and basis for further implementation. This code retrieves data from one controller and presents both position and orientation when the application runs. This is a C# that have implemented a C++/CLI interface as a binding between the C++ OpenVR API and the application code. How this interfacing works is further explained in section 2.5

2.2 Industrial Robot Testing

When new software for the ABBs industrial robots are released, there are several tests that must be performed to verify that all functionalities, new and old, are working as intended. For this project ABB have provided with a selection of tests that have their focus on correct movement and operation regarding movement of the robot.

The tests require manual set up of different equipment depending on the tests. For example, several of the movement tests requires a pen to be mounted as a tool, then adjust the height to a table beneath for the robot to draw on. This also requires a paper to be attached to the table, and changed for each test. In general, these test procedures base its verification on manual inspection and observation, requiring a test technician to operate and continuously observe the tests.

2.2.1 Test parameters based on ABB's existing test cases

The existing tests being performed today for checking the continuous functionality for new releases of the robot controller software, is the basis for designing the tests for this project. Each test has several sub test cases covering different test parameters. This section covers the parameters found in these tests that are relevant for adopting into an automated test system that uses motion tracking for verification.

Smooth movement

The two first test are performed with a pen mounted as the robot's tool, and are drawing figures and shapes on a piece of paper. The operator is required to mount the pen and adjust it to be adjusted to a correct high relative to the drawing surface. These tests are intended to verify that the robot is following its instructions correctly while drawing patterns and figures, by checking fine-point overshoot, path accuracy, and smooth movement. To verify these test, the operator must monitor the robot when running, and manually inspect the resulting drawing. The test also specifies that the result should be compared to previous results.

Overshoot may occur when the path contains a fine-point at a corner, where the robot does not manage to stop the tool. This results in the pen in this case to draw the line past the point defined. The operator of the test will inspect the drawn path for any indications at these points for an overshoot.

Path accuracy refers to the robot following the path correctly without any deviations. The result from the test should have the drawn path have the same dimensions as the path defines in the RAPID code, and be equal to the results from previously verified tests.

While the robot is performing its routine, the operator is instructed to observe the movement, which should be continuous without any hiccups or staggering. The patterns and figures drawn should be drawn without errors, and as intended.

Interpolation

Interpolation of circles and zones are covered by two tests. The intention of the first test, it to verify that the robot maintains the same speed through a zone interpolation, with *Wrist SingArea* enabled and disabled. SingArea defines how the robot behaves when operating close to singularity. With SingArea enabled, the robot can deviate its orientation slightly to move past a singular point.

The second test regarding interpolation, is also testing with SingArea enabled and disabled, but in this case, it is a circular path with different orientations. The test is designed to verify that the circle segment is interpolated correctly. The movement speed is not mentioned for this test.

These tests instruct the operator to verify that the speed is not affected during the interpolation. There are not mentioned any other methods of verification other than visual inspection during operation.

Logical instructions

While the robot is following a path several times, a digital output is enabled between two points on the path. The intention of this test is to verify that the output is operated at desired points of the executed RAPID code. The test instructs the operator to verify that the output is enables between these points, and disabled for the remaining path.

Stop and restart of program

One test is intended to check that the program can be stopped and resumed anywhere on a path. The operator is instructed to stop and start the program repeatedly throughout the program. The robot should stop and start without problems at any part of the program.

Path acceleration limitation

The *PathAccLim* is a functionality in RAPID that sets a limit to the allowed acceleration and/or deceleration of the TCP. This functionality is tested by running a program that sets a limit to the acceleration and deceleration, and runs simple path. The test only specifies criteria for the program to complete without internal errors.

Triggering output signals

There is a set of test cases devoted to trigging digital and analog outputs on the system, using the RAPID instructions *TriggIO* and *TriggEquip*. These test cases check that the output is set at the correct position, or at the right time before a given point. The RAPID program in the tests runs trigger movement, where several different trigger times are used. The test requires the recorded times to be within 5ms of the programmed timing. These tests use external equipment and program to handle the signal timing.

2.2.2 The ABB Test Engine and RobApi2

ABB have provided their test engine for creating unit tests that can be run through their test system. The engine is implemented in C#, and each test are created as normal unit tests that inherits the test base class defined in the test engine.

The test engine contains a logger used for logging events through the tests, and is the object used to register failures in a test. The logger has three states of failure:

- **Warning** indicates issues in the testing that isn't necessary a failure.
- **Error** is used when there is a failure in the test, but does not stop the test.
- **Fatal** for when there is a serious failure, and will stop the execution of the test.

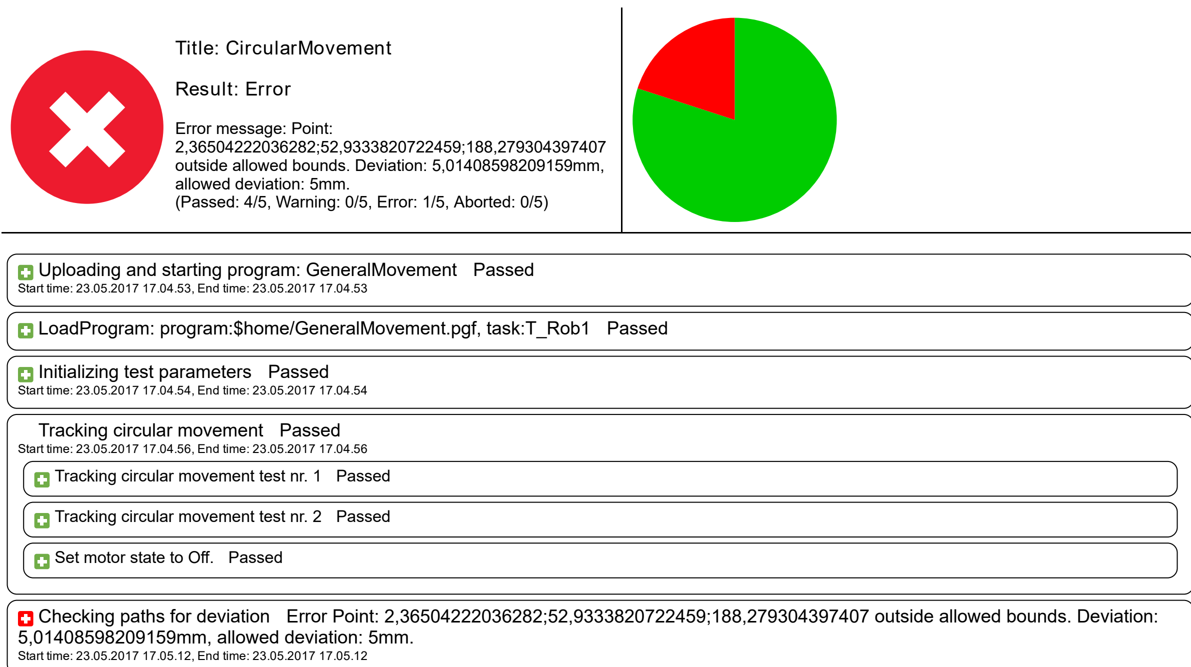

Figure 2.5: An example of the html page generated by the test engine shows each sub test and its result. warnings, errors, and fatals, are reported, and indicated.

The logger is also used to indicate the start of a new sub test, and new operations for that sub test. These are used to display the results in report log generated at the end of the test, as a html page. An example of the generated test result is shown in figure 2.5. General info can also be logged, which are shown by expanding the current operation in on the result page.

The test engine implements the use of RobApi2 as the means to communicate with the robot. To run RAPID programs on the robot, the RAPID module files (**.mod*) that are required for a given program are uploaded to the robot, together with the program file (**.pgf*). Then when the RobApi2 function *LoadProgram* is called, the program is loaded, and ready to start.

2.3 The robot setup

This project is using *Rudolf* for this project, one of the IRB 140 ABB robots on the UiS robot laboratory. This robot has been provided with a 3d-printed mount for one of the HTC Vive hand controllers, with the controller mounted with rubber bands and felt-dampening to reduce the vibration caused by the robot. Figure 2.6 shows the controller mounted to the robot. The HTC Vive Lighthouses are roof mounted, and are approximately 90 degrees on each other at the robot's position to maximize the coverage from the light houses.

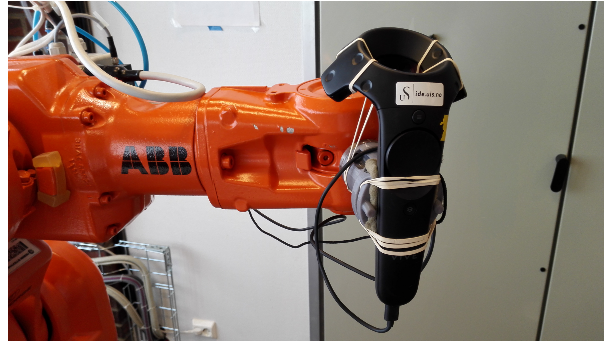

Figure 2.6: The Vive Controller mounted to the IRB 140

The Vive controllers goes into sleep mode if left stationary for more than 10 seconds, a battery saving feature that cannot be turned off. If the controllers are in sleep mode for more than a couple minutes, they will turn off. To work around this, the controller is connected to power through the usb-port, they do not power off, and they return from sleep with in a second. If the USB port on the controller are connected to the PC, it is possible to communicate with the controller without the HTC headset connected.

The automated tests are executed from a computer stationed in proximity of the robot, as it requires to be connected to the HTC Vive equipment. The computer is also connected to the same network as the robot controller, allowing access to the robot controller through RobApi.

When the tests are running on the robot, they may be running for a longer period, and without an operator on site. To ensure that there is no danger to other students, there is a light gate covering the entrance to the working area of the robots that will, if triggered, immediately stop the robots.

2.4 Workign With ABB's Robots

When working with ABB robots, RobotStudio is the tool that allows offline programming, simulation, and interaction with live robots. ABB have created a programming language named RAPID, that is used to program the behavior of their robots. RADID defines several instructions used to perform different operations [8].

When defining positions the robots should move to, a *robtarget* is used, which defines a position and orientation the robots tool should have when located at this target. When

moving the robot to a target, it is performed relative to a work object which defines a coordinate system relative to the robot. this means that the target is relative to the active work object, and a set of targets can be moved together by only changing the work object.

The tool of the robot, called Tool Center Point(TCP), is the defined point on the robot that will be moved to a target. A *tooldata* can be used to offset and rotate the TCP relative to the robot's tool mount.

RAPID have several instructions for moving the robot, including *MoveL*, *MoveJ*, and *MoveC*. Where *MoveL* moves the robots TCP in a straight line from its current position to the specified target, while *MoveJ* does not follow a straight line, but only moves the robot's joints to their required position. *MoveC* is used to create a circular movement by defining the end target, and a target the circular movement must pass through on the path.

2.5 Development environment

The test engine provided from ABB are written with C#, and thus the tests will also be implemented in C# using their template. Since OpenVR is a C++ API, an integration of this into C# is required. C++/CLI is a language created by Microsoft that supports Common Language Infrastructure(CLI), making it possible to create libraries that are compatible with C# applications. This is one way to create an interface that can pass data from OpenVR, a C++ API, to a C# environment such as the test engine. Visual Studio is used as IDE for this project, and support the integration of both C# and C++/CLI projects in one solution, simplifying the build process. The test engine also runs its tests as unit tests from within Visual Studio.

Chapter 3

Implementation

The automated testing application developed in this project contains several aspects, including the test cases them self and tracking of the robot, but also some additional applications used in the process. This chapter will explain each aspect of the project, starting with an overview of the system, and then go into details of each part.

3.1 System Overview

The system created for performing automated tests on industrial robots consists of several parts. The major parts of the system is the implementation of the new test cases that have been designed for this application, and the implementation of a tracker API using the HTC Vive for tracking the robots motion. Besides these two, there are several minor aspects to the project such as applications aiding in representation of results and methods of determining test parameters.

Figure 3.1 shows a block diagram of the different building blocks of the system. The hardware is shown on the left, while all software solutions on the right, divided by the stippled line.

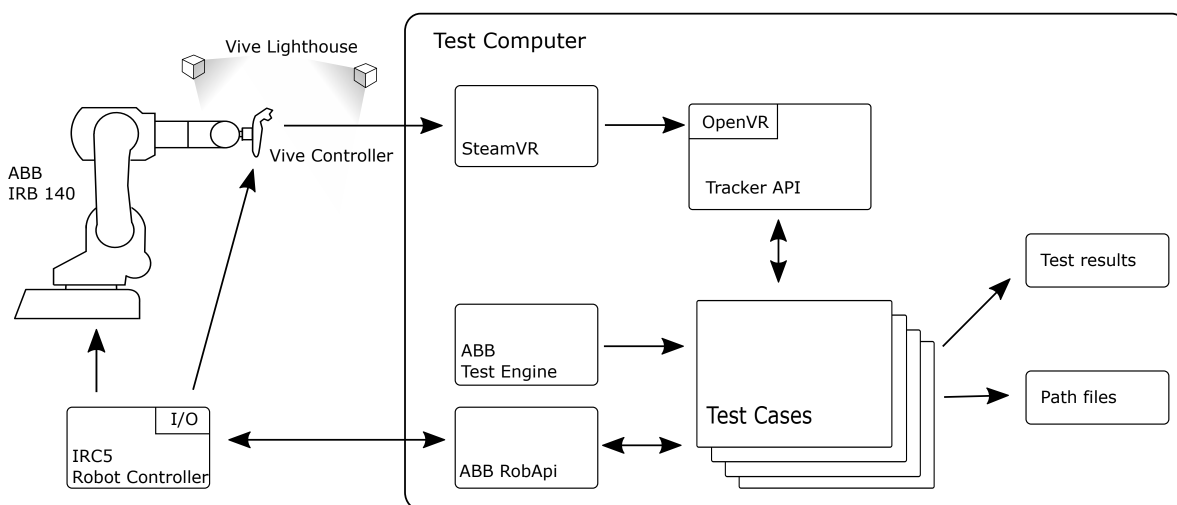

Figure 3.1: System overview

On the hardware side is the robot, an ABB IRB 140 in this case, controlled by the IRC5 robot controller. The Vive controller are mounted to the robot's tool mount, and the Lighthouses mounted in the room as previously described in section 2.3. A digital output from the robot controller are connected to one of the button inputs on the Vive controller.

The test cases running on the test computer, builds on the provided ABB test engine, and uses the developed *TrackerAPI* to retrieve data from the Vive controller. *TrackerAPI* uses the OpenVR API to interact with the Vive controller through SteamVR. RobApi is used to communicate with the robot controller through the network to upload and run the RAPID code for each test case. The tests outputs the results and the associated recorded test paths. The structure of the test result is defined by ABB's test engine, which generates the output files based on the result logged during the test.

3.2 Implementing The Tracker API

The HTC Vive is used in this project as a method of tracking the motion of an industrial robot. The implementation of the required functionalities has been implemented into a C# library named *TrackerAPI*.

This section will explain how this API have been structured, and how the different components work, staring with the interface that the users of the API are concerned with, and then expanding on the inner workings connected to the OpenVR API.

3.2.1 Structure of the Tracker API

The *TrackerAPI* is based on a prototype code provided by ABB, that have been used for retrieving the data from a HTC Vive controller. This prototype uses a method for retrieving data from a Vive controller in a C++ class named *NativeViveTracker*, then uses a C++/CLI class named *ViveTracker* as an interface to the C# test application. This structure was kept for this project, with some modifications to adapt to this use case.

Figure 3.2 shows the overview of the different classes in the API, and how they interact with each other through the different programming languages. The user of the *TrackerAPI* creates an instance of *RobTracker* and is only concerned with the classes within the C# section of the figure 3.2. These classes are defined within a namespace named *TrackerAPI*.

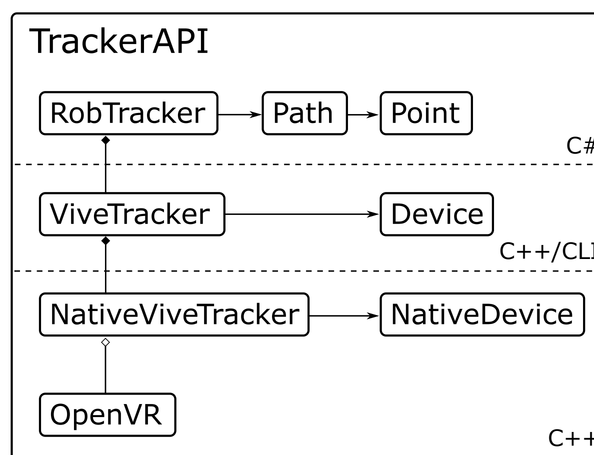

Figure 3.2: Class overview of the TrackerAPI

3.2.2 Representing a robot path

TrackerAPI represents all positional data as discrete points, containing information about a specific point in time, represented with a *Point* object. To represent a movement path, these points are are added to a *Path* that contains a list of points.

3.2.2.1 The Point Class

Point is an object representing one point in space at a given time. A point is represented with position and orientation in three-dimensions, and also the momentary directional and angular speed and acceleration. Positions are represented using a three dimensional vector, while orientations are represented using quaternions. Each point has a time stamp defining the specific time the point data was retrieved. For digital input, an integer is representing the input state of the device at this point.

The object has a few helper functions used when doing operations to the object. There defined static functions that can add and subtract two points, by respectively adding or subtracting the parameters from one point to the other. And a static function that returns a *Point* object interpolated between two points where a scalar value determines where the interpolation should be performed between the two points.

3.2.2.2 The Path Class

The *Path* object represents a path with a series of *Point* objects, stored as a *List*. The *Path* object implements some general functionalities found in common C# containers (eg. *List*), such as *Add* and *Count*. A *GetEnumerator* function, returning the enumerator of the actual list containing the *Point* objects, allowing for example the use of *foreach* on a path.

The class have a set of functions implemented for manipulating and utilizing the *Path* object:

GetPoint() returns the *Point* at a specified index in the *List* containing the points in the *path* object.

GetInterpolatedPoint() returns a *Point* object at a specified time stamp. The points are discrete with a resolution determined by the specified tracking rate, and when interested in a point at a specific time, this function will return a estimated point at the specified time stamp.

GetClosestPoint() finds and returns the *Point* object int the *Path* closest to a specified *Point* or position.

GetAveragePoint() calculates the average of all points in the *Path* and returns that as a new *Point* object.

GetPathSection() returns a new *Path* object from the given start and stop time stamp. Useful when only a section of a tracked path is relevant to verify a test.

ComparePath() is a static function that returns a *Path* object that is the difference between two given *Path* objects. This function compares each *Point* in the first *Path* with the corresponding time interpolated point in the other *Path*.

CalculateVelocity() uses the difference in position and time stamp between each *Point* in the *Path* object to calculate a new directional speed for each *point*. The speed is found from central numerical differentiation, where the speed on one *Point* is based on both the *point* before and after in the *Path*.

StoreAsXml() serializes the *Path* in XML format, and stores that to a file at the specified file path.

3.2.3 Using the Tracker API

When implementing the API in an application or test, an object is created as an instance of the *RobTracker* class. This object contains all functionality for tracking, and connection to the HTC Vive. This subsection will explain how the API is initialized and used.

3.2.3.1 Initializing the RobTracker and OpenVR

When a *RobTracker* object is created, it has to be initialized before in can be used. The initialization is setting up the connection with the HTC Vive through OpenVR, further explained in section 3.2.4.

InitTrackingDevice() is used to initialize the tracker. The tracker can be set to two modes, either Predicted or on update. This refers to which capturing method is used to get the pose through OpenVr. As explained in section 2.1.3, there are two methods in OpenVR to retrieve a pose, *GetDeviceToAbsoluteTrackingPose* and *GetControllerStateWithPose*, where they corresponds to predicted, and on update, respectively.

When using predicted, the *GetDeviceToAbsoluteTrackingPose* returns a new pose on each call, and the frequency of desired new poses should be capped. This is done by a second parameter to the initialization method, that specifies this desired update frequency.

3.2.3.2 Handling Different Tracked Devices

HTC Vive with OpenVR is able to support up to 16 tracked devices, each with their own device index. To access data from a specific device where this is applicable, a *DeviceName* enumerator is defined for each of the devices, that corresponds to the device index that is defined by OpenVR.

3.2.3.3 Retrieving a Single Point From the Tracker

The API have two methods implemented to retrieve the data of a single point, depending on the use case and circumstance.

GetCurrentPoint() is the first method, which returns updated data for the specified device instantly.

GetAveragePoint() is the second method, which is intended for retrieving data for a point when the tracker is stationary, and it is desirable to reduce the jitter. This is done by tracking the device for a short amount of time specified by the user, and then calculating the average of that section of points.

3.2.3.4 Tracking and Retrieving a Tracked Path

The use case of the tracker API requires that tracking a path can be done simultaneously as other parts of the code are running. The tracker is therefore performing path tracking in a separate thread, using *(*StartTracking) and *StopTracking* to respectively start and stop the thread.

StartTracking() is used to start tracking by initiating a new thread that runs the tracking method. When tracking, the *RobTracker* stores each the tracked points in a *Path* object for each device. These paths are emptied when a new tracking is started cleaning up any old recorded paths. The tracking method stars a timer that is used to give each sampled point a time stamp. The timer is also used to maintain the sample rate defined when initializing the tracker.

To handle the data retrieved from the HTC Vive in a thread safe manner, each path is controlled with mutex to avoid accessing the data while the tracker is writing to the path objects. The thread tracking the Vive will run a while loop until a tracking flag is set to false by *StopTracking*.

StopTracking() is called when the tracking should stop, and sets the tracking flag to a low state that will cause the tracking to end when the current tracking sample is completed. This method will by default wait for the tracking to end before returning, ensuring that following code that might try to access the tracked data does not fail.

GetTrackedData() returns the path specified by the provided device enumerator, after the tracking have completed and released the tracker mutex.

3.2.3.5 Calculating offset between the Vive and robot

When tracking the robot, the tests require the tracked positions and orientations to be relative to the coordinate system that the robot is using. The Vive controllers are operating in their own space, and the translation and rotation between the Vive controllers, and the robot must be known to be able to correctly transform the positional data. The center point of the Vive controller are also offset from the robots TCP, and requires to be known to be able to calculate the position of the TCP when changing the orientation of the Vive controller. The latter could be compensated for by creating a tool model in RAPID, and let the robot controller handle this, but the exact center point of the Vive controller is not known, and its position and orientation tends to slightly vary when mounting it to the 3d-printed mount.

There are two procedures performed to find the translations and rotations required to perform this compensation.

CalculateOffset() is used to find the transform between the Vive space, and the robot space. By measuring the position and orientation at four points, one at the point where the new center of tracking should be, preferably at origin of the work object that is used in the RAPID code. Then one point in positive direction of each of the three axes of the work object, one with only positive X, one with positive Y, and one with positive Z. The tracked point in the center will be used as the basis for the translation between the two coordinate systems. Then three vectors are created, one in each direction (X, Y, Z), which are used to create the rotation matrix between the two systems.

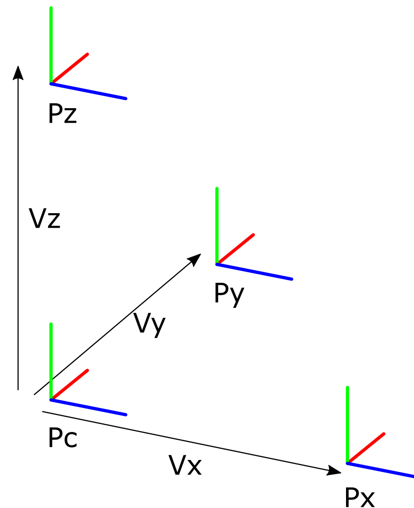

Figure 3.3: The directional vectors, $Vx\ Vy\ Vz$, are found from the four calibration points.

This rotation, and translation is enough to get the tracked pose from Vive transformed to the coordinate system of the robot, if the orientation of the Vive controller are maintained.

CalculateToolOffset() finds the translation offset from the TCP to the center of measurement of the Vive controller. This is required to be able to correct the position of the TCP from the tracked position of the Vive controller.

Figure 3.4 shows a simplified illustration of this process showing only two dimensions. Listing 3.1 shows the C# code performing the calculation.

This process uses three three orientations with the TCP in the same positions. The first should be in the normal orientation which the coordinate system offset were calculated with. The calculated tool offset will be in reference to this first orientation. The second orientation is should be a 180° rotation around the x-axis, such that the measured position in the first two points are at the same position in the x-axis. From the position of these two points, the translation to the point in the middle is calculated by finding the difference and dividing by two, which represents the offset in y, and z-axis.

The next step is to find the offset in the x-axis. This requires a measured point with the tool oriented 90° around the y-axis. Now the difference between the center position found in the last step, and the new position, will have a length in the z-axis which is equal to the offset in x-axis in the first orientation.

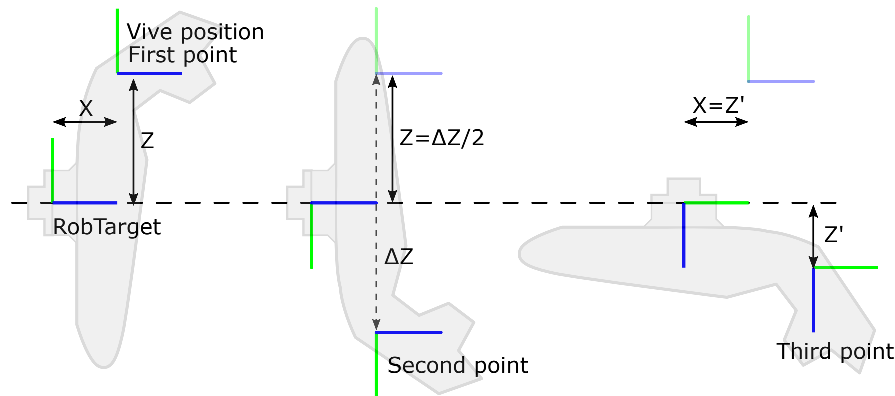

Figure 3.4: The three steps for finding the too offset, here illustrated with two dimensions.

```
Vector3D diff = orient1.Position - orient2.Position;
Vector3D diffToCenter = diff / 2;

Offset.Z = diffToCenter.Z;
Offset.Y = diffToCenter.Y;
Offset.X = orient2.Position.Z + diffToCenter.Z - orient3.Position.Z;
```

Listing 3.1: The C# code calculating the tool offset from three orientations

StoreOffsetToFile() Stores the offsets to file, containing coordinate system translation and rotation, and the rotation of the Vive controller.

LoadOffsetFromFile() Loads the offset data from a file, used after initialization.

UnloadOffset() resets the offset data for *RobTracker* back to initial values. This function is used when the raw data from the Vive controller is required, for example when performing a new calibration.

3.2.3.6 Converting Vive data to the robot coordinates

With the calculated offset transformation and rotation, the positional and orientational data from a Vive controller, or another device, can be converted into the coordinate system of the robot.

Whenever one of the tracking function in *RobTracker* is retrieving a point from a Vive controller, the private function *GetPointFromDevice* is called. This function retrieves the raw data from the Vive device, and applies the required transformation, as shown in listing 3.2. The position is first translated to remove the offset between the Vive and the robot, then rotated around the robot's origin using the rotation quaternion found in section 3.2.3.5.

```
point.Position = Vector3D.Subtract(point.Position, PositionOffset);

Rotation3D rot = new QuaternionRotation3D(OrientationOffsetRobot);
RotateTransform3D rotTransform = new RotateTransform3D(rot);
point.Position = rotTransform.Transform(point.Position);
```

Listing 3.2: The C# code translating and then rotating the position from the Vive to the robot's coordinate system

3.2.4 Implementation of OpenVR

The functionalities required from OpenVR is incorporated through a library named *ViveTracker*. This is created with C++/CLI and is the interface which the C# application uses when retrieving data from the Vive devices. *ViveTracker* have two methods used for retrieving data, *UpdateData* and *GetDevice*, that returns a device object containing all data for a specified device. *ViveTracker* stores the data for all devices, and updates them when a update is called.

3.2.4.1 Initializing *ViveTracker* and OpenVR

When creating a *ViveTracker* object, the constructor creates an instance of OpenVR's *vr::IVRSystem*, and initializes it as a background application by calling *vr::VR_Init()*. *ViveTracker* can track based on two methods, described in section 3.2.4.2, which is by default set to *predicted* if not otherwise specified in the constructor call.

3.2.4.2 Updating the Vive device data

Before any new data can be retrieved from the Vive devices, *UpdateData* in *ViveTracker* must be called, to update the data for all devices.

The ViveTracker are able to get the pose of the Vive devices with two different OpenVR functions, *GetControllerStateWithPose* and *GetDeviceToAbsoluteTrackingPose*. Which one to use is defined with an enumerator, either set at initialization, or changed at a later point. These two different methods of retrieving device pose, are defined as *OnUpdate* when using *GetControllerStateWithPose*, and *Predicted* when using *GetDeviceToAbsoluteTrackingPose*. The code updating the device data implements these two together since much of the code is the same, and uses if statements to determine what method to use where they differ.

When retrieving data with *Predicted* method, *GetDeviceToAbsoluteTrackingPose* outputs an array with the poses for all device indexes. Then the device indexes are looped through, and the controller state for the current index is retrieved with *GetControllerState*. In the case of *OnUpdate* this is where *GetControllerStateWithPose* is used to get both controller state, and pose for the index.

3.2.4.3 Retrieving device data from *ViveTracker*

The position and orientation is outputted as a transformation matrix, which are converted to a vector for position, and a quaternion for the orientation. The position, orientation, directional and angular velocity, and the controller button states, are added to a device object and stored in an array. The data is retrieved from this array when the *GetDevice* method of *ViveTracker* are called.

When retrieving data with *OnUpdate* method, all device indexes are looped through calling *GetControllerStateWithPose* to get the data belonging to that device index, if the device is connected.

3.3 Using the Tracker With the Industrial Robot

With HTC Vive correctly set up in the space where the robot is operating, and the Vive controller mounted to the tool of the robot, the tracker API is ready to track the movement of the robot.

When mounting the controller to the robot, the mount holds the controller firm to reduce any unwanted shift in position. The vibrations from the robot have shown to disturb the measurement of the controller, and some padding is used to reduce the interference. For this project, a mount was 3D-printed, but required to be mounted slightly different than intended by having three millimeter thick felt in between, and fasten the controller with rubber bands and pvc-tape.

The API registers changes in button state on the controller, and to utilize this functionality, one of the buttons have been extended out of the controller with two wires. These two wires are connected through a optocoupler to one of the outputs on the IO device of the robot controller. The optocoupler is there to separate the electrical potential between the controller and its connected device, and the robot controller. The output from the robot controller is a digital signal with 24v. This is fed into the input of the optocoupler through resistors to limit the current to the required 10mA of the optocoupler. Figure 3.5 shows the circuit as it is connected.

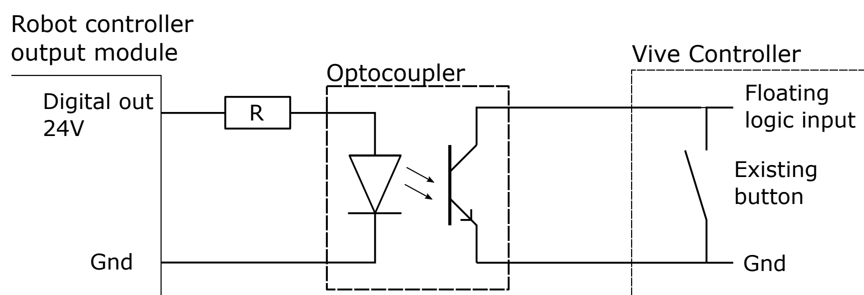

Figure 3.5: Circuit diagram of the trigger connection

To prevent the controller from shutting down when inactive, it is powered through the usb port, which is also used to connect the controller to the computer. This cable and

the cable to the trigger input on the controller are mounted such a way that they are long enough to allow the robot to rotate the controller without tensioning the cables. To ensure that there are no challenges with the cable when running the tests, the RAPID procedures have been run at slow speed while observing the cables when they are designed, before the tests are run at higher speed and unattended.

The controllers are designed to conserve power when not used, and if left stationary for more than 10 seconds, the controller will enter a standby mode, where it does not update its position. The controller leaves this state and reenter active mode when its registers movement again, that this takes approximate 1.5 - 2.0 seconds. When performing tests where the robot is stationary for some time, and in most cases before starting the test, the robot must move the controller and wait at least this amount of time before a tracking can begin.

3.4 Automated Test Cases

Based on the existing tests described in section 2.2.1, several topics have been defined as Automated Test Cases (ATC). Each topic is intended to cover one aspect of the robots functionality that would be subject to testing. For each topic have several tests defined for testing the different functionalities associated to the topic. In total eight topics where drafted as possible test cases, bot this section will only describe the tests that have been implemented.

3.4.1 General Test Implementation

All implemented tests have used a common implementation method, where each topic is created in one test class, inherited from a common base test class. For each test class, a RAPID program are created containing all procedures to be run in the tests.

3.4.1.1 The automated test base class

ABB's Test engine provide a test base class for implementing unit tests with that system. For this purpose, there are several functionalities common for most implemented tests which are defined in a extended test base that inherits the ABB base test.

The *AutoBaseTest* class implements the required test initialization, where the connection to the robot are done by providing the IP address to the robot, Rudolf in this case. A *RobTracker* object is created and initialized, and the appropriate offset parameters are loaded from file. This initialization function is called by the test system, after the test object is created, and the test specific parameters such as the filename for the RAPID program are specified in advance by the test constructor, which is loaded in the initialization function.

AppendToLogfile() is a general-purpose function for logging info to file during testing, fur such as dedicated test output, or measurements.

LoadProgram() is used on initialization and uploads the required RAPID files to the robot. When the .mod .pgd file for the RAPID program have been uploaded onto the robot, the program is loaded on the robot through *RobApi*.

3.4.1.2 Structure of the RAPID programs

Each test case has their own RAPID program that they run, these RAPID programs are created based on the same program structure. A common work object is defined for all the test cases, that ensures that the *RobTracker* follows the same coordinate system for all tests. For this project, the work object used are 45 degrees to the left side of the robot at position $[500, 0, 400]$, to set the origin in the space where the robot can reach.

Throughout a test case there are desirable to run different RAPID procedures at different times, without reloading a new program for each procedure. By running a while loop, with a *TEST case*, that check the value of a variable named *option*, where the value determines what procedure to run. Similarly, each procedure can have a new *TEST case* testing the variable *subOption*, allowing for another level of options to the RAPID code. After a procedure, or sub procedure are completed, *option*, and *subOption*, are set back to value zero.

To interact with the RAPID procedures from the tests cases, several functions have been implemented into the base test class to operate on the RAPID values through the use of *RobApi*.

SetRapidOption() and ***SetRapidSubOption*** is used to set the value of the RAPID variable *option* and *subOption* respectively. *SetRapidOption* has an extra parameter to set the value of *subOption* in the same function call.

WaitForRapidOption() and ***WaitForRapidSubOption*** checks the current value of *option* and *subOption* respectively, in the running RAPID procedure, and waits for the value to be reset by the RAPID procedure. This function is used when the test is required to wait for the running RAPID procedure to finish.

CheckRapidSubOption() checks if the current value of the RAPID variable *subOption* is equal to the provided value.

SetRapidSpeed() sets a speed variable in the RAPID program that updates a *speeddata* that is used by some of the tests where it is desirable to be able to set the movement speed of the robot.

3.4.2 Preliminary (ATC-0)

The preliminary test case covers all tests, or preliminary operations that are performed prior to the rest of the test cases. This is mainly for calibration of the tracking system, and verifying the calibration. When the calibration and verification is performed, the results should be verified before the rest of the tests are initiated, as a failure here will compromise the rest of the tests.

3.4.2.1 Calibrate coordinate system

This test performs a procedure where four points are tracked to calculate the tracker offset as explained in section 3.2.3.5. These four points are at predefined targets, where the first is at $[0, 0, 0]$, the origo of the current work object. The three other targets is each at at positive $300mm$, one in each positive axis direction.

The tracked points must be in reference of the Vive coordinate system, so prior to measuring the points the *UnloadOffset* function is called to revert any offset parameters that are set. These points are passed through the *CalculateOffset* function, which calculates the new offset parameters.

To ensure that the points used for this calibration is correct, the jitter at each recorded point is checked. If the jitter is more than $1.2mm$ the test gives a warning, indication that the calibration might not be precise enough. An additional verification is performed by transforming the measurement points through the calculated offsets, then verifying that the new points are within limits.

3.4.2.2 calculate the vive to tool offset

The purpose of this preliminary routine is to determine the offset between the robot's TCP, and the center position of the HTC Vive controller. This is done with the *CalculateToolOffset* function described in section 3.2.3.5.

This calculation requires three different orientations around the same point. The robot is moved to each of these orientations and the position is tracked by average with the standard $500ms$. The points are used to calculate the tool offset with *CalculateToolOffset*, and then the offset parameters is stored to file.

3.4.2.3 Verify the calibration

This routine is intended to be used to verify that the tracking calibration is still valid, during or after testing. This is simply done by placing the robot in four points, same as used when calculating the coordinate system offset in section 3.4.2.1. These points are tracked, with the current offset parameters, and then checked for offset. The deviation should be within $5mm$ to be accepted, and if they are not, this test will give an error for each point outside allowed deviation. This test also checks that the jitter when measuring the points are within $1.2mm$ to ensure that the measurements are correct.

3.4.3 General movement (ATC-1)

The purpose of the tests of general movement test case, is to test all basic functionalities of the industrial robot.

3.4.3.1 Direction and position

The purpose of this first test is to verify very basic movement, checking that the robot is moving the TCP in the correct direction. And that the robot is moving its TCP to the correct position. Because this test is to determine the correct movement calculation of *MoveJ* RAPID operation, the first rapid instruction use *MoveAbsJ* to set the robot in the initial position based on the robot's joint angles.

When the joint target is reached, that point is tracked with *RobTracker*, and defined as a center point. Then in sequence the robot moves its TCP in each positive X, Y, and Z direction, by calling a *MoveJ* operation with $50mm/s$. While moving, the tracker is retrieving the position by averaging over $100ms$, to reduce the implication of jitter from the tracker. For each result from tracking, the difference from last tracked point is calculated, and then checks that the movement in the current direction is positive. If the difference is not positive, an error is given, indicating that the robot did not move in the correct direction.

After each movement in the different directions, the stationary point is tracked and stored as a point. These points are predefined and compared to determine if the robot ended up in the desired position. If the robot is not within $4.0mm$, which is the defined allowed deviation for this test, an error is given for that sub test.

3.4.3.2 Orientation

To determine if the the orientation of a given target is achieved correctly. This test moves the TCP to four different targets, and records the tracked position of each of them. The recorded orientation is compared with the predefined orientation, and the differential are calculated. The angular deviation limit for this test is set to 5 degrees, and each point tested will result in an error if the deviation is not within this limit. Each tracked point is added to a path object, and stored to a single file when completed.

3.4.3.3 Linear movement

The purpose of testing a linear movement, is to verify the correct operation of the RAPID instruction *MoveL*. This test performs two *MoveL* instruction between two points, one maintaining the same orientation, and another with a continuous rotation to the orientation.

In the RAPID code, four targets are defined, two start targets, and two stop targets, with the same position. They first set of targets have both targets with the standard orientation, while the second set have both different orientation, resulting in a movement where the TCP is following a line, while the orientation is changing through the movement. The targets are positioned such that the TCP is only moving in the Y-axis, keeping both X and Z unchanged.

Both movements are tracked separately, and checked one by one. Due to the simplification of only moving in Y direction, the deviation is found by extracting a two-dimensional

vector from the X and Z positional value for each tracked point. The length of this vector represents the deviation for that particular point. The allowed deviation is set to $5mm$, and if a deviation greater than that, an error is logged.

For each path that is tested, a new path object is created where each out-of-bounds point is copied to. At the end of the test, both the tracked paths, and the out-of-bounds path are stored to file, and can be examined at a later point if desired.

3.4.3.4 Circular movement

Circular movements are performed with the RAPID instruction *MoveC*. This test uses this instruction to move the TCP in a circular movement, and then verify that the tracked path follows that circle.

This test moves in a circle twice, one with the tool orientation stationary, and one with a changing orientation throughout the movement, where both runs are tracked into separate paths. The RAPID targets used for the circular movement, are all set to the same position on the X-axis, where all movement happens in the Y and Z-axis.

When checking deviation from the circle path, it's the distance from each point to the oracle circle, following a perpendicular line to the tangent of the circle. By removing the X-axis value from the 3D position, the result a 2D vector that is on the same plane as the circle. When the tracked position is compensated for the offset to the oracle circle, the length of this 2D vector should be the same as the radius of the circle. This difference and the difference in X-axis, represents a 2D vector with a length of the deviation from the circle.

The allowed deviation is set to 5mm for this test, where an exceeded deviation in a point will be reported as an error. All points with too much deviation are stored into a separate path. Both the tracked paths, and the path with the deviated points, are stored to file at the end of the test.

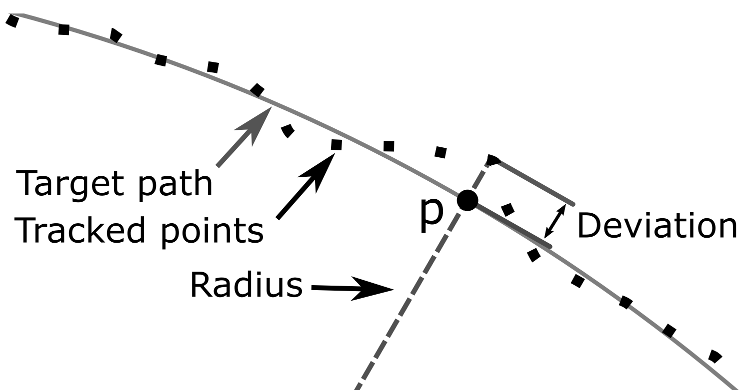

Figure 3.6: Measuring the deviation from a circular path, by determining the distance to point p, which is the point on the circle with the same angle as the tracked point.

3.4.3.5 Path movement

This test is designed to verify correct operation of a longer sequence of RAPID instructions, by comparing the tracked path to a predefined reference. The test prepares the robot by running a procedure that moves the robot to its starting point before starting the tracking. The timing between the tracker and the RAPID procedure is important, as the two are to be compared based on time stamp. When the tracker is started, the RAPID procedure is called immediately after.

The RAPID procedure for the path, have been run through a simulation in RobotStudio, where the simulated path has been recorded with a smart component that records the movement and stores to an xml file. This xml file is loaded into a path in the test, and with the *ComparePath* method of *Path*, the difference between the tracked and the simulated path are calculated.

Each point in the differenced path is checked for its length, which refers to the deviation at a given point. The allowed deviation is set to 10mm, where each point with too high deviation is reported as an error. The logged path, and the differenced path are stored to file.

3.4.4 Speed (ATC-2)

The purpose of this test case is to verify that the robot is moving at the correct speed with different operations.

3.4.4.1 Line Speed

This test performs a linear movement using the *MoveL* RAPID instruction, and repeats the same movement several times with different speeds, verifying each of them.

This test is testing six different speeds, from $50mm/s$ to $800mm/s$. Each run are performed on the same movement between two points, and each rub are tracked individually. This test only checks the speed at the center of the movement, and not the whole run. To get an accurate speed, a $400ms$ section of the path centered around the center point of the path is extracted, and averaged. The speed of this averaged point is compared to the defined test speed, and its deviation compared to the allowed deviation, set to $5mm/s$. If the speed deviation is larger than allowed, an error is reported. Tracked test path for each run are stored to file.

Only six different speeds were used in this implementation, but there is no limit to how many different speeds that can be added to the test. In the RAPID procedure, the speed value is converted into a *speeddata*, not limiting the speeds to any predefined values.

3.4.4.2 Zone speed

The speed through zones should be maintained at the set speed throughout the movement. There are limitations to the maximum speed the robot can move through a zone, due to

the limitations of how fast the robot can change direction.

Six speeds are tested, and for each, the smallest zone that allow for the robot to continuously move at that speed are found for each speed. To determine the zone sizes, the movement is simulated with RobotStudio, where the TCP speed can be plotted with the RobotStudio's signal analyzer. Through experimenting with gradually decreasing the zone size the following combinations of speed and zone have been found.

Table 3.1: The smalles zones each speed are able to move through with constant speed

Speed	Zone size	Zone duration
v1000	z150	150ms
v800	z80	100ms
v500	z30	60ms
v200	z5	25ms
v100	z1	1ms
v50	z0	1ms

Each zone has been programmed in RAPID as different sub-options, that are run from the test. Each run is tracked in separate paths, and analyzed after all speeds are performed. This test checks the speed throughout the movement, and to exclude the parts of the path outside the zone section, the duration of each zone is also determined through the use of the signal analyzer.

The center point of the zone is found by using *GetClosestPoint* to find the point on the path closest to the target where the zone is used. Then to get the relevant section of the path that covers the zone, the time stamp of that point is used with *GetPathSection* to get a section that is plus and minus $100ms$ of that point, totaling $200ms$ that is enough to cover the longest duration which is $150ms$ at $z150$.

For each run, the max deviation is found through the zone, and then compared against the allowed speed deviation that is set to $5mm/s$. If the maximum deviation is outside this limit, it results in an error. All test paths are stored to file.

3.4.4.3 Path speed

This test is verifying that the robot maintains its speed throughout a path. The path is a rectangular path with different zone sizes in each corner, where the test performs a series of runs with different speeds. For each run, the path is tracked, and then the speed for each run is checked throughout the tracked path. To account for the time the robot uses to accelerate and decelerate, and the delay from tracking starts to the robot is moving, $400ms$ of the path is removed from the start and end. The speed is checked at each point on the path, and the maximum deviation is kept track of.

The test runs each speed as a sub test, and gives an error to the sub test if the deviated speed is more than $5mm/s$. Each tracked path is stored to file, named with the run number, and the test speed.

3.4.5 RAPID Operations (ATC-6)

The purpose of the RAPID operation test case, is to check and verify various RAPID instruction that is not directly covered by the other test topics.

3.4.5.1 Start and Restart RAPID Procedure

This test is verifying that a running RAPID program will halt when a stop is called, and resume where it left off when starts is called.

When the test is started, a RAPID program that continuously moves the robot in a path is started. The test then waits for a random amount of time between two and eight seconds, before the program calls stop through RobApi.

The program enters a loop where it calls the tracker for an average point over a $25ms$ period. The point is compared to the previously recorded point, and checks the distance between these two points. With the robot moving with a predefined speed of $200mm/s$, the robot should move approximately $5mm$. If that distance difference is less than $2mm$, its assumed that the robot have stopped.

The time it takes from calling stop with RobApi, to a stop in motion is measured with the tracker, is found. If this time is less than $200ms$, it has passed and no error is logged. If the delay is more than that, the test gives an error informing on the elapsed time. This action is repeated 10 times, checking each time.

3.4.6 Trigger (ATC-7)

The purpose of this test case if to Verify correct operation of the trigger instructions. These tests used the connection from the IO module on the robot controller to the Vive controller as described in section 3.3.

3.4.6.1 TriggIO with different speeds

This test is using TriggIO to trigger a digital output when the robot is moving in a linear movement past a defined point. The test runs the RAPID procedure one time for each speed defined in the test. For each run the robot moves with the determined speed towards a known point, where the digital output should be triggered, and records the movement with the tracker. Each point in the recorded path is checked to find the point where the trigger input to the tracker changes from a low state to a high state. The tracked position at this point is compared to the predefined point, and the positional deviation is compared to the allowed limit of $5mm$.

If the deviation is more than allowed limit the test gives an error. The test also calculates the time difference between the measured trigger point and the desired trigger point. This value is not verified to anything, but added as information to the test result.

The test also checks that there is a registered trigger point in the recorded path, verifying that there is no error to the connection between the output from the robot to the Vive

controller. The path for each run is stored to file, with a name containing the speed, and run number.

3.4.6.2 TriggIO with time offset

Using much the same principles as the previous test using TriggerIO, this test specifies the time before a point to trigger the output. This test runs a similar RAPID procedure, where the triggerIO is created with the time offset specified from the test. The test checks with time offset of $0ms$, $100ms$, and $200ms$.

For each run, the first point in the path where the trigger input is set high on the controller is found. The time difference from this point to the interpolated point in the path closest to the defined trigger point is found. The time offset being tested is subtracted from this difference to find the actual deviation. The test checks if the deviation is less than $25ms$ to verify the correct triggering. If the deviation is more than the allowed deviation, the test gives an error, indicating the deviation. The path for each run is stored to file, with a name containing time offset, and run number.

3.4.7 ViveTests

This is a topic that defines a series of test created to test some of the functionalities of the HTC Vive.

3.4.7.1 Reaction time of rapid start

This test is intended to determine the time it takes from a rapid procedure is started through RobApi until a movement is registered through the HTC VIve.

This is done by starting the RAPID procedure through RobAbi, and starting a timer at the same time. Then the position of the Vive controller is continuously checked, and when the change in position is more than $1mm$, indicating that the robot has started to move, the timer is stopped. The time is logged and stored to file. This procedure is repeated 20 times to verify the accuracy.

3.4.7.2 Checking drift over time

To determine if the HTC Vive drifted over time, this test is created to check a series of points repeatedly over a longer period of time.

The test runs a RAPID procedure that moves the robot to 23 different points covering a large area within the reach of the robot. At each point the test records the average of the point over a period of $500ms$ to reduce the effect of jitter.

When all points are measured, the program waits for five minutes, before repeating. This routine is repeated 192 times to cover 16 hours. Each point is stored to a file containing all recordings for a given point.

Chapter 4

Resulting System

The purpose of this solution is to validate the functionality of an industrial robot. This requires the solution to be precise enough to be qualified to perform these validations. This chapter will present the different measurements performed to determine how well this solution ant the test cases performed.

4.1 Tracking Precision for The HTC Vive

The results presenter in this section covers the different parameters determining the total precision and accuracy for the Vive. These results are based on both tests designed for specifically testing the vive, and from the different test cases.

4.1.1 Positional Jitter

The positional precision of the Vive controller was tested by performing several positional tracking by averaging several stationary points over a $500ms$ period, and checking the jitter at each point, see section 3.4.7.2 for details. The jitter was also recorded while testing the calibration of coordinate system, where each points were averaged over a $200ms$ period.

The average and standard deviation have been found for the jitter of the measurements at each point. When looking at the amount of jitter at these points the results show that some points have consistently more jitter than others. There is also a large gap between the $500ms$ averaging and $200ms$ averaging. Table 4.1 shows jitter for $500ms$, where the highest average jitter was calculated to be $1.17mm$, and the lowest were $0.80mm$. The $200ms$ averaging shown in table 4.2 have a maximum and minimum average at 0.62 and $0.52mm$, respectively. The overall smallest single point jitter with $500ms$ sampling were $0.48mm$, and for $200ms$ the smallest were $0.37mm$

Table 4.1: Maximum and minimum average jitter when tracking a point for $500ms$

	Jitter[mm]	Standard Deviation[mm]
Max	1.17	0.231
Min	0.80	0.183

4.1.2 Positional Drifting Over Time

When testing the positional drift over a longer period of time, the tests showed that the positions at the points tested would deviate over time. The tests showed that drift

Table 4.2: Maximum and minimum average jitter when tracking a point for 200ms

	Jitter[mm]	Standard Deviation[mm]
Max	0.62	0.093
Min	0.52	0.075

would depend on the direction, as shown in table 4.3 there were most drift in the Y-axis and Z-axis, while there were less in X-axis. These measurements were performed over a five-hour period with five minutes between each measurement. Note that these directions are relative to the HTC Vive coordinates, and could be different if the coordinate system were facing another way.

Table 4.3: Maximum and minimum drift in the different direction over a period of five hours

	X	Y	Z
Max	1.67	4.17	3.50
Min	1.26	2.7	2.31

4.1.3 Precision of Tracking Speed

When testing how to well the Vive could track the speed of the robot, the speed was retrieved from the Vive Controller pose, which calculates the speed itself. Figure 4.1 shows the speed plotted, which shows a noisy measurement.

Another approach to determine the speed, were to calculate the speed using the *CalculateVelocity* function described in 3.2.2.2. Figure 4.2 shows the result tracking the same speeds with this method.

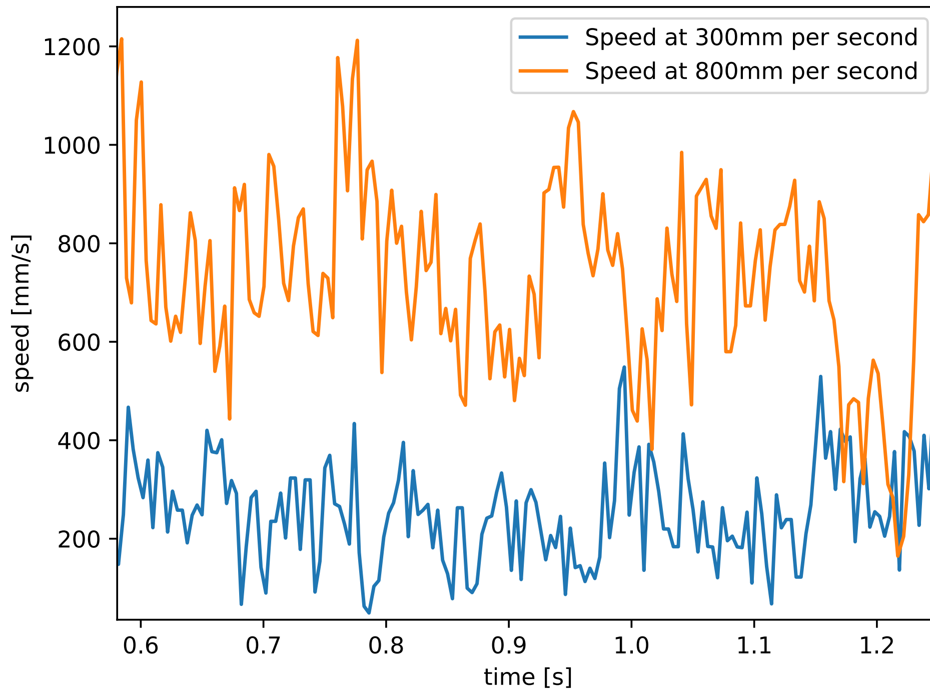

Figure 4.1: Movement at 300mm/s and 800mm/s with speed directly from the controllers

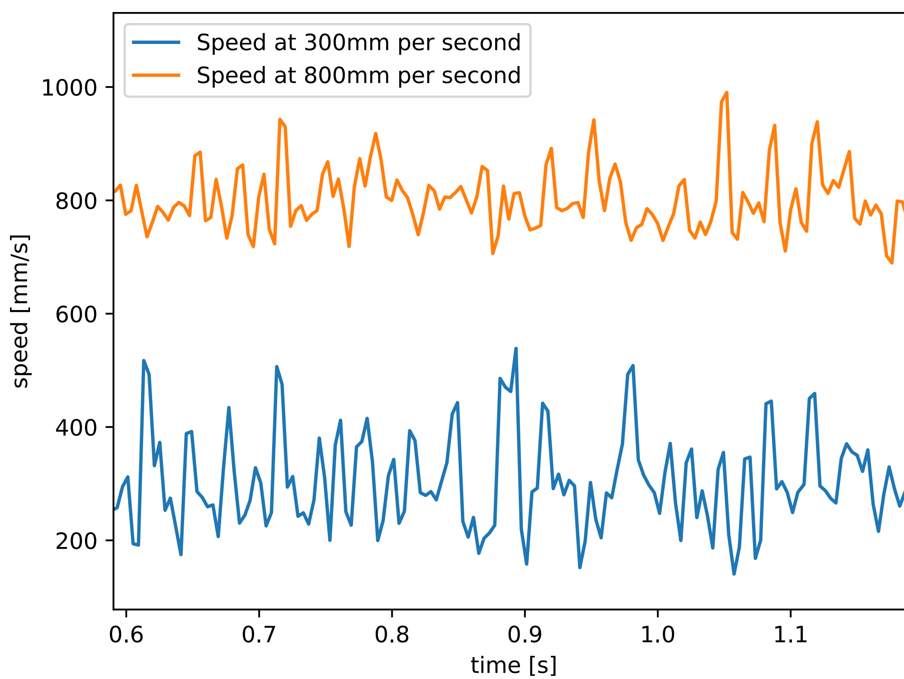

Figure 4.2: Movement at 300mm/s and 800mm/s with soeed calculated from position and time stamp

4.2 Testcase Performance

This section presents the results for each of the test cases.

4.2.1 Preliminary

This section shows how the preliminary test cases (ATC-0) worked for calibrating and checking the HTC Vive.

4.2.1.1 Calibration of Coordinate System

The procedure of determining the positional and rotational offset between the Vive coordinate system and the active work object, have shown to work as intended. Completing the test in 35 seconds. Both calculating offset, and verifying the correctness succeeds, and relevant info regarding deviation is added to the test log.

When a calibration is performed, the resulting error measured at the same calibration points are observed to vary from sub one millimeter, to five millimeters. Table 4.4 show the result from 20 measurements.

Table 4.4: Deviation in validation positions

Position	Min [mm]	Average [mm]	Max [mm]	Standard Deviation
Center	0.03	0.25	1.07	0.256
X-axis	1.53	1.74	1.91	0.126
Y-axis	1.31	1.51	2.24	0.194
Z-axis	1.61	2.62	3.09	0.398

During these 20 measurements, one test run had jitter when measuring its point on the Z-axis, with $25mm$ in maximum jitter. This measurement has been omitted from the data set used in table 4.4.

When the test captures the position of the tracker without any offset loaded, meaning no calculations that alters the positional data from the Vive controller. The distance from the center position, to each of the three other points were measured to determine what distance the HTC Vive gave when moving a distance of $300mm$ with the robot. Table 4.5 shows the results from the same 20 runs.

Table 4.5: Measured distance in Vive space at $300mm$ movement of robot

Position	Min [mm]	Average [mm]	Max [mm]	Standard Deviation
X-axis	300.72	301.90	302.15	0.295
Y-axis	301.39	301.65	303.38	0.423
Z-axis	301.30	301.41	301.75	0.398

4.2.1.2 Calculating Tool Offset

This preliminary test calculates the tool offset from the current TCP by moving the tool to three predetermined orientations. The calculations are performed as expected, and the result are close to within $1.5mm$ from the actual position.

When testing the precision of the calculation, a tool was defined in the rapid code as tool that were $50mm$ in the tool's z-axis, and centered in the others. The calculation of the offset to the center of the Vive controller resulted as shown in equation 4.1. When this offset is compensated for in rapid, the result from running the same calculation again resulted as shown in equation 4.2. Note that the offset shown is relative to the active coordinate system, in this case set to the working work object, and the offset must be rotated to the same orientation as the tool definition.

$$offset = \begin{cases} x: & 27.5mm \\ y: & -2.2mm \\ z: & 85.4mm \end{cases} \quad (4.1) \qquad newOffset = \begin{cases} x: & 0.0mm \\ y: & 0.5mm \\ z: & 0.1mm \end{cases} \quad (4.2)$$

4.2.1.3 Verifying Calibration

The test verifying the calibration works as expected, and tests the same points used to calculate the offset and orientation. When the test completes without any error, there were no inaccuracies with the result, and there is no indication of change in the trackers position.

4.2.2 General movement

This section shows the result from each of the general movement tests (ATC-1).

4.2.2.1 Direction and Position

The test is correctly detecting that the robot is moving in the correct direction. When a movement in the wrong direction is detected an error is given. The positions at the end of each directional test is tested, and detected correctly, giving an error if the position is deviation more than the allowed limit.

4.2.2.2 Orientation

The process of checking the orientation at the predefined orientations works, and results in an error if the angle is deviating more than the allowed limit.

Since calculating the orientation from the HTC Vive is not working properly, it is not possible to execute this test without error in the current state.

4.2.2.3 Linear Movement

The linear movement test completes and succeeds as expected. Moving the TCP in a linear movement, once with the tool in the same orientation, and one with a rotating

orientation. When the robot does not keep the TCP on the line, each tracked point that is outside limits, are reported as an error, informing the detected deviation.

Table 4.6 shows the combined results from 20 tests, where the maximum detected deviation through the two different movements are used. Figure 4.3 plots the paths of both movements for a test performed.

Table 4.6: Max deviation for 20 runs of the test

Orientation	Min[mm]	Average[mm]	Max[mm]	Standard Deviation
Stationary	6.20	3.17	6.20	1.415
Rotating	1.44	3.28	7.70	2.021

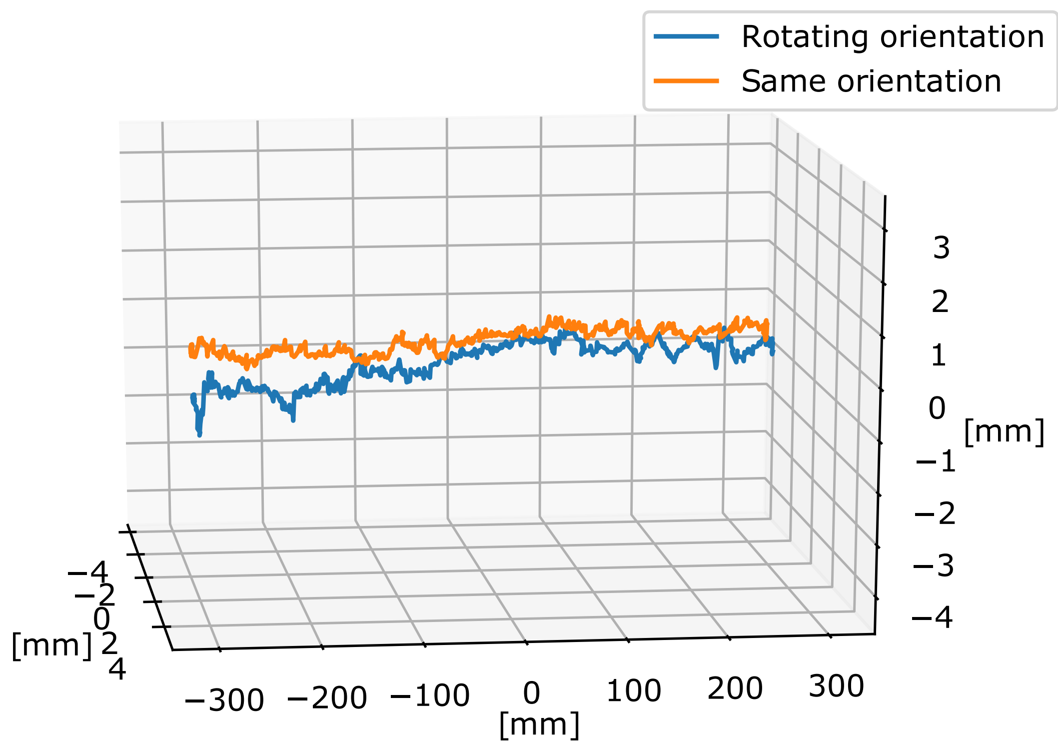

Figure 4.3: Plot of the movement of the two runs. showing the deviation through the movement. Note that the X and Z axis is zoomed in to visualize the deviation of the path.

4.2.2.4 Circular movement

The test of circular movements is executed as expected, and movement with stationary orientation and with changing orientation succeeds as expected within limits. The distance from a measured point to its perpendicular point on the circle is calculated correctly.

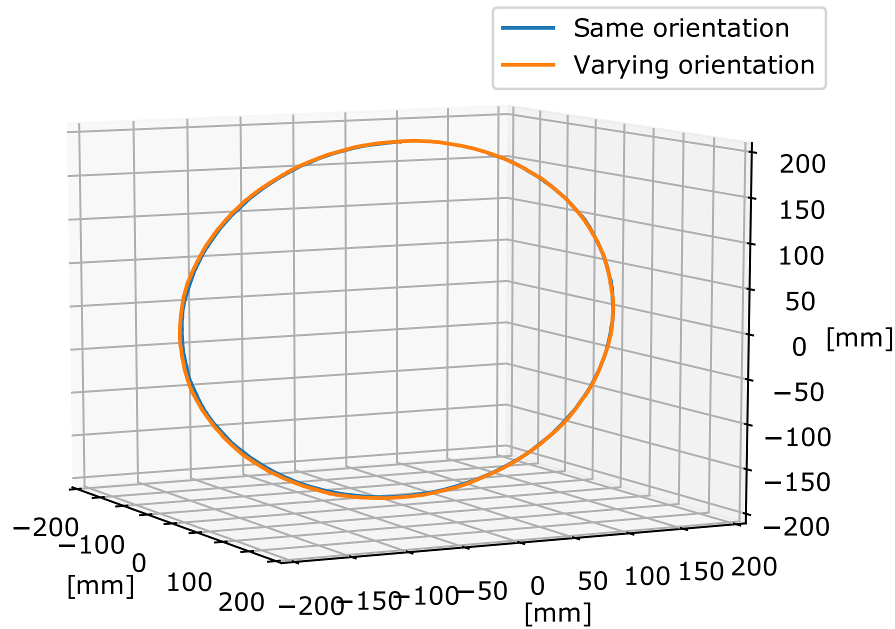

Figure 4.4: Plot of the movement of the two runs. Showing the deviation through the movement.

4.2.3 Speed

These results show the performance of the speed tests (ATC-2).

4.2.3.1 Line speed

Testing line speed completes as expected, testing and verifying all speeds added to the test list. With the deviation limit set to 5mm per second, there are some tests that finishes with an error, as the speed is tracked to a higher deviation, especially at higher speeds. Table 4.7 shows the results when running each speed 20 times, where some of the results show that there were some deviations that were out of limits.

Table 4.7: Speed of robot tool in a linear movement

Speed [mm/s]	Min [mm/s]	Mean [mm/s]	Max [mm/s]	Sd [mm/s]
5	3,71	5,17	6,92	0,84
50	48,38	50,31	56,20	1,69
100	96,57	100,41	104,35	1,59
250	246,57	250,68	255,96	2,27
500	494,81	502,03	507,42	3,40
800	797,45	803,09	807,17	2,27
1000	992,04	1003,26	1015,31	5,30

When testing the listed speeds, the test completed in 7 minutes and 44 seconds. Each move procedure performed, completed in the expected time for that speed over a distance of one meter. At 400mm/s the return from end of line to the start of the next, takes approximately four seconds.

4.2.3.2 zone speed

The test executes the movements correctly, using the correct speed for each zone, and all paths are stored. When checking the speed deviation, each point throughout the zone is calculated. This have shown to not give a successful test, always resulting in several points through the zone being outside the allowed deviation.

Figure 4.5 shows how the speed through a *Z5* zone at $200mm/s$. There is change in the amount of jitter in the speed measurements when the robot translates from a downwards vertical movement to a horizontal movement.

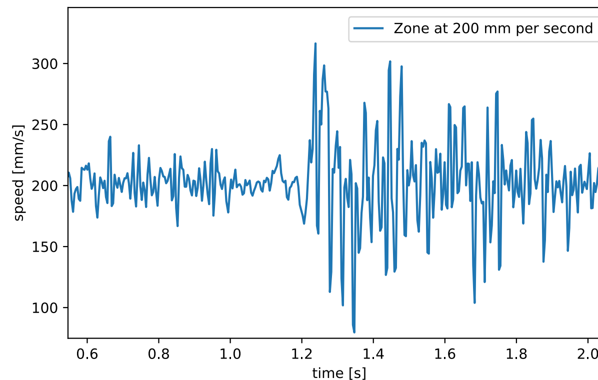

Figure 4.5: Movement at 200mm/s translating from vertical to horizontal movement through a zone at $1.2s$.

4.2.3.3 Path speed

Running the path with different speeds executes as expected, but with the same issues as experienced with zones. Determining the speed deviation from each point in the path does not give results that are accepted by the test, and therefore each test will result in error. The test does execute, and each tracked path is stored to file.

4.2.4 RAPID Operations

This test runs the path continuously as expected, and is able to stop and start the RAPID procedure at any time, completing the test without errors. When the jitter during a measurement is detected to be more than twice the minimum move distance, a warning is logged. The time from a stop have been called through RopApi, to the robot have stopped moving is determined and checked to be within allowed delay. Table 4.8 shows the results from running the test ten times, showing that from a stop is called, it is detected within a maximum of $131ms$.

Table 4.8: Summary of stop times after running the test ten times

Min	Average	Max	Standard Deviation
0.056ms	0.087ms	0.131ms	0.028ms

4.2.5 Trigger

These results show the performance of the trigger tests (ATC-7).

4.2.5.1 TriggerIO

The trigger test using *TriggIO*, performs as expected with the ability to add different speeds to the test. The test correctly detects the point where the trigger input of the controller is set high. The distance from the trigger position is found, and either reports an error if beyond allowed deviation, or added as info to the test log. The time delay for the detected trigger are detected and also added to the log.

The test was performed with three different speeds, 20 times each, where the distance and time deviation is recorded. The resulting time delay values are multiples of 4ms, which is the sample time for the Vive controller. Table 4.9 shows how many occurrences there is for each delay, for the three speeds.

The distance deviation that is measured is not quite correct, as this test compares the distance from the triggered point, to a predefined position. The positional offset in the tracker will add to the measured error.

Table 4.9: Number of occurrences of different delay for registering trigger input at different speeds.

Speed	0ms	4ms	8ms	12ms	16ms	20ms
200mm/s	0	6	7	3	4	0
400mm/s	1	4	6	5	3	1
600mm/s	1	3	5	6	4	1

4.2.5.2 TriggerIO with time delay

Testing the TriggerIO with time delay performed as expected, correctly verifying that the digital output is triggered the correct amount of time before the robot reaches a given point. The test was performed five times each with a delay of $0ms$, $100sm$, and $200ms$. Table 4.10 shows the recorded results at the three different time delays. The time differences were measured from the triggered point to the point in the tracked path that is closes to the desired position.

Table 4.10: The resulting time deviation from five runs on the different time delays.

Run	0ms	100ms	200ms
1	-12.0ms	5.0ms	3.7ms
2	16.7ms	1.6ms	-5.5ms
3	4.0ms	6.8ms	-0.6ms
4	12.0ms	9.3ms	6.1ms
5	0.0ms	9.2ms	0.2ms

Chapter 5

Discussion

5.1 HTC Vive as a Tracking Device

Through the tests performed, several aspects of the performance of the HTC Vive as a tracking device have been demonstrated, mostly giving good results. This section will discuss how well the HTC Vive behaved as a tracking device.

5.1.1 System Jitter

In chapter 2.1.2 some of the precision of the HTC Vive previously found through experimentation, were explained [1], showing among other results that the end to end stationary jitter was on an average of $0.3mm$. In the tests performed during this project, such small jitters were not experienced. When tracking a stationary point, and averaging $200ms$, the jitter were found to be mostly in the range from $0.5mm$ to $0.7mm$. When averaging a tracked point over $500ms$, the results showed that the average jitter increased to range from $0.8mm$ to $1.2mm$.

The Vive set-up for the tests performed by [1] have the two lighthouses mounted 2.4 meter above the ground, where they are four meters apart. This is roughly the same spacing that is used in the robotics laboratory, but there might be other aspects that affects the jitter. The fact that the observed jitter was consistent at the same point in space, but changed from one point to another, indicates that there might be challenges with the room setup.

5.1.2 Position Drift and Changing Coordinates

Drift in the system over time would affect the results from the tests when left running for a longer period of time. To determine if there are any drift, or to what extent there is, a overnight test were performed. The results showed that the system had no continuous drift when used for a longer period of time. The change in position were observed to drift around the same point. Similarly to what was observed with the jitter, some points in room were more susceptible to drift than other. The results also showed that there was a difference in the scale of drift in different directions.

Although no continuous drift was observed, at some points during the test, with several hours between, the coordinates shifted slightly. This happened two times during the test run, with one shift in the X-axis with $6mm$, and later a shift of $35mm$ in the z-axis and $5mm$ in Y-axis. As far as the recorded data shows, this shift happened between two runs,

while the Vive was stationary.

There is no clear explanation of why this happened, but it could be assumed that this is the HTC Vive performing some sort of re-calibration. A similar change in position have been experienced during development where the Vive controller have been disconnected and reconnected again.

5.1.3 Interference From The Robot

When comparing the tracked paths from the robot's movement, and from hand motions, there is a distinct roughness to the robots paths, clearly shown in sharp corners. Vibrations from the robot were a challenge from the beginning, when there was no dampening between the mount and the controller, causing vibrations from the robot to be transferred into the Vive controller. When changing the mounting method for the Vive controller and adding some padding between the controller and the mount, the tracked path was improved to the quality shown in figure 5.1.

During the project, the method used for capturing tracking data from the Vive controller were changed from the *on update* method to *predicted*, which gave a much smoother result for the robot. Figure 5.1 and 5.2 show the same path tracked using *OnUpdate* and *Predicted* method respectively, which shows a clear improvement on the tracked path.

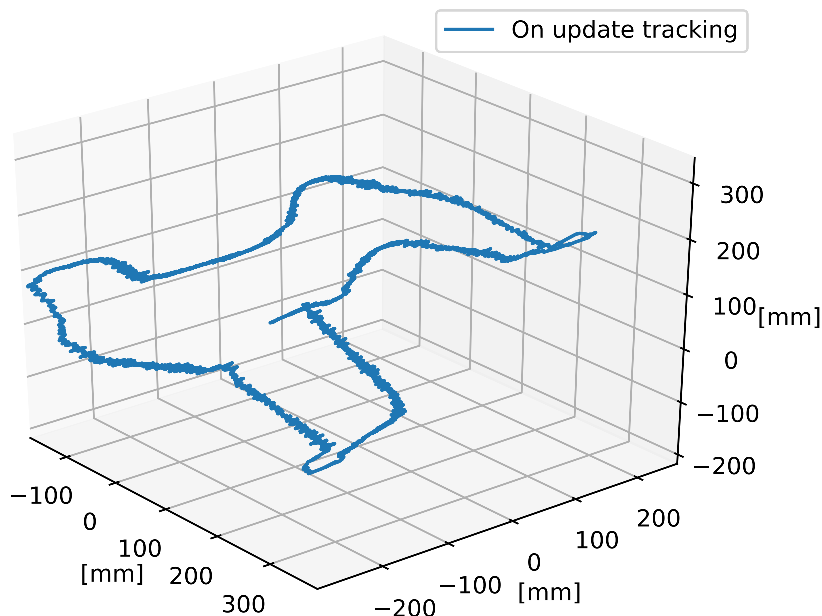

Figure 5.1: Tracking of the robot performed with the tracker configured with *OnUpdate*

When tracking the robot moving through a corner zone, the tests have shown that an abrupt change in direction, such as a small zone, the HTC Vive is not able to track that movement without fluctuations. Figure 5.1 shows this clearly at some of the corners (such as the right most corner zone). A similar behavior can be experienced at the start of a path, when the robot is accelerating.

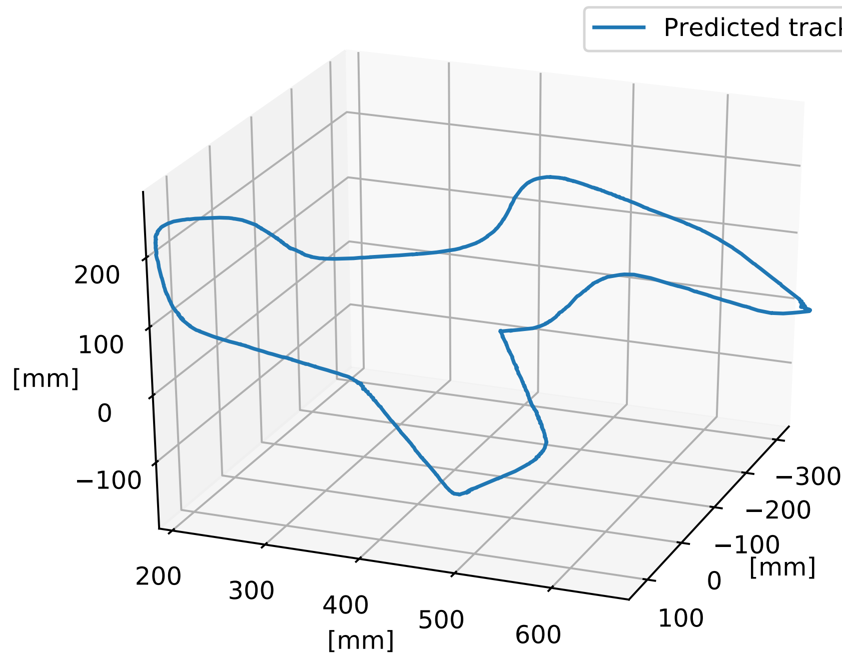

Figure 5.2: Tracking of the robot performed with the tracker configured with *Predicted*

5.1.4 Speed

The speed tests show how that it is possible to determine the speed of the robot from the Vive controller within approximately 7mm/s at speed up to 800mm/s. This were only possible to get when averaging the tracked speed over a 400ms period, where increasing the period would improve the result, and vice versa.

While experimenting on how to get the most accurate measurement of speed, it was shown that the speed of the Vive controller provided when retrieving device pose through OpenVr were not precise, and were heavily fluctuating. Even when averaging this result over time, did not result in the correct speed.

To get a more accurate measurement of speed, it had to be calculated from the difference in position and time between the samples. This speed was also fluctuating, but consistent enough that averaging 400ms of samples gave the desired result with a constant speed.

When observing the tracked speed from a movement by hand, the speed is substantially smoother, not showing the fluctuations occurring with the robot. Figure 5.3 shows the speed recorded when moving the hand controller by hand. This indicates that there are challenges with getting accurate readings while the Vive controller is mounted to the robot, probably as a result of vibrations from the robot.

5.1.5 Tracking resolution regarding refresh rate

The refresh rate of the positional data from the Vive controller is shown to be 250Hz, and as explained in section 2.1.1, the updated position determined by a swipe from the Lighthouses only occur at a rate of 120Hz, and only one direction at a time. The resolution

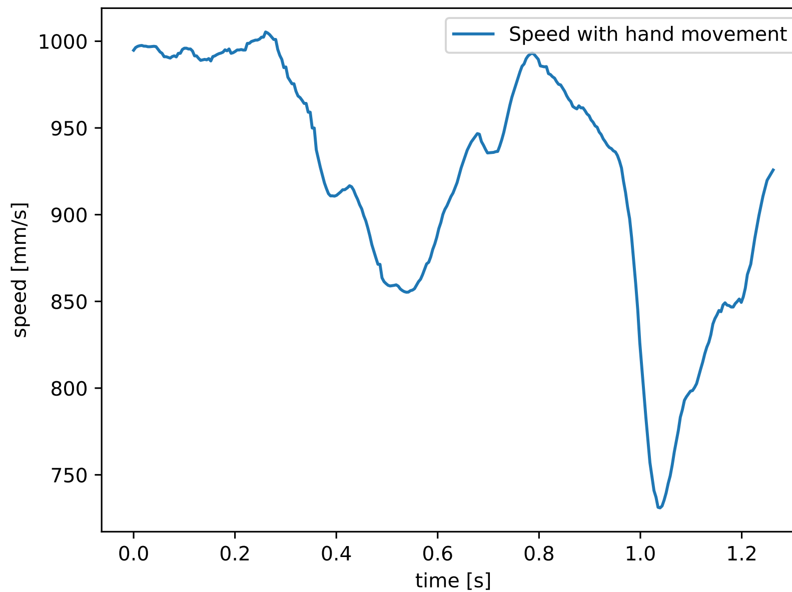

Figure 5.3: The speed tracked when moving the Vive controller by hand does not have the same jittering measurements as when mounted to the robot.

for capturing movements and events, are restrained by the $250Hz$ update rate. Any resolution desired for tests, such as trigger position, or detection of a change in motion, will be limited by this rate.

Although the Lighthouses are the source of accuracy of the tracked position of the Vive controllers, none of the tests and experimentation performed during this project have indicated any limitations due to the Lighthouse update rate being lower than the 250Hz update rate. This is also supported by the findings of [1], where the optical updates with the Lighthouse were only noticeable when the controller were performing fast rotating movements.

5.1.6 Improving the Quality of the HTC Vive Tracking

The precision of the HTC Vive tracking have not been experienced to be as good as indicated in [1], but still acceptable, and have performed well for this project.

There have been occasions where the controller has lost its position of the tracking and returned a point at $[0, 0, 0]$. Without being able to confirm the cause, moving one of the Lighthouses closes to the Vive controller, seemed to reduce the occurrences of error in tracking. Some sources indicate that reflections from surfaces in the room may cause problems for the tracking [9] [10]. This might indicate that there are challenges due to placement of the Lighthouses, and maybe reflections or other interference from the relatively cluttered room with reflective surfaces, including a large TV. A better room setup with less clutter and no reflective surfaces, might improve the quality of tracking.

Vibrations from the robot have been showed to cause problems for the controller, which

might be one of the causes for high jitter. It might also be a cause for the controller to fail to calculate its position and loose tracking. While creating the tests, there where RAPID paths without proper tool configuration causing the robot to rotate the tool all the way around when moving from one point to another quite fast. This caused the controller to lose tracking several times, and might be an indication that such fast movements combined with the vibration, disrupts the Vive controllers ability to maintain its tracking.

To improve the problem with vibrations from the robot, the tracked Vive controller will require a better mounting solution that have enough padding between the Vive and the robot to eliminate as much of the vibrations as possible. This must be done without allowing the Vive controller to move relative to the robot. This might be a challenge with the current controller due to the length of it causing it to lag behind on rotational movements due to the torque and inertia.

Towards the end of this project HTC Vive released their new Vive Tracker, which is a small tracking device designed for mounting on custom equipment used with the HTC Vive [11]. The Tracker is a small puck shaped device at $100mm$ in diameter, and $43mm$ high, with the majority of its mass close to the mounting point. It has a screw mount with a mating surface covering the majority of its diameter.

The Vive Tracker would most probably give a simpler mounting solution, and with the compact size reduce possible movement due to the resulting torque and inertia. With the large mounting surface, a larger piece of dampening could be used without allowing for too much movement between the Tracker and the robot.

The Tracker also features electrical contacts with digital inputs acting as the buttons featuring on the Vive controller. This would make the connection to the digital output of the robot controller simpler than the solution of dissembling the controller done in this project.

If and how this new Vive Tracker would improve the tracking is speculation, but based on how the problems the Vive controller are experienced, and the way the new Tracker is designed. This should be tested to determine the quality of that as an improvement to the solution.

5.2 Quality of the Tracker API

The tracker API have shown to work as expected for all the implemented test cases, and should suffice for other implementations that requires the same functionalities. There are some aspects of the API that does require more work to improve the quality of the tracking.

5.2.1 Tracker Use Cases

There are several ways to use the tracker, and this project have tried utilized them in several different ways in the test cases, that includes retrieving a single point, recording

path, and continuous monitoring.

When using the tracker to get the current position of the Vive controller, two methods have been used, either directly retrieving the current controller data with *GetCurrentPoint*, or using some time to get an average measurement with *GetAveragePoint*. *GetCurrent-Point* have been uses when desiring a point without any delay, such as when moving the controller. This method does have its inaccuracies due to the jitter from the controller precision. The purpose of *GetAveragePoint* is to improve on this inaccuracy, by sampling several points over a given amount of time to get an averaged point that minimizes the effects of the jittering. This function has in most cases been used to get accurate tracking of a stationary point, for example when calibrating the tracker offset, and measuring points.

An alternative use has been to get an average point when the robot is in motion, when continuously checking the position of the robot. When averaging a point over a movement, the purpose has been to minimize the jitter in the directions perpendicular to the movement direction. This method will also reject some jitter in the direction of movement, and the resulting average calculated, will be approximately in the middle of the start and stop position. This have been acceptable in these cases where the resolutions of measurement is given by the time to average, which for example the *DirectionAndPostion* test used $100ms$, for each check, which were good enough for its purpose. When a higher resolution is required for the points sampled during a motion, the averaging time can be reduced, where the sample rate will increase, but also increase jitter error.

The tracking has worked as expected, and is able to track at the sampling rate specified when initializing the *RobTracker*. There have only been required to use the path tracking capabilities as they are, and in that sense no need for another way to track a path. The one feature that would be preferable to have for the tests, is to be able to get the current point, and average point, while tracking a path with the *RobTracker*. This would allow to continuously track the test when *GetAveragePoint* or *GetCurrentPoint* is used.

5.2.2 Calibrating the Tracker

The process of calibrating the tracker is required to be able to track the position of the Vive controller in the same coordinate system as the robot. The results from the test cases, and other tests performed have shown that this calibration have worked, and given a good precision of the tracker, but with some issues.

As the preliminary test case *CalibrateCoordSys* have shown, when verifying the calibration, the results show that the positions used for calculating the rotation, is two to three millimeters off. The points used to calculate the rotation between the two coordinate systems should, when corrected whit the transform found, be correctly placed in the new coordinate systems.

When this error was observed, some testing of the method of calculating the rotation were performed, and the results confirmed that the method does not correctly calculate the

rotation. When the raw points are transformed, the resulting points are approximately in the correct position, but not as precise as desired. The center point is transformed correct as this is only a translation without the need for rotating the points, while the other points does not seem to rotate as expected.

A couple different methods for performing the rotation were experimented with. The rotation transform was tested with both rotation matrix, and quaternions, without getting any difference. Since the solution worked well enough to perform the tests, there were not used too much time on this, but noted that there are inaccuracies here that affects the results when using the tracker.

Another inaccuracy that was shown, were the error when measuring a distance of $300mm$, where the resulting distance were approximately $1.5mm$ longer. This is not a significant error, but it this error is consistent through the vive space, it is a difference that could be compensated with a scale factor added to the positions.

The next part of the calibration is to find the determine the rotations required to transform the orientation of the Vive controller into orientation of the robot's TCP. Both the orientation of a RAPID target, and the orientation of the Vive controller are defined in quaternions, and it were therefore desirable to be able to perform this calculation with quaternion rotations.

To determine the orientation of the robot based on the orientation of the Vive controller, requires finding the required rotation transform to calculate this. Several attempts were done to determine this, and method that seemed to work were found and implemented, but the result was never accurate enough, and not fully implemented into the TrackerAPI. This is probably caused by the same challenges as seen when transforming the positions of each point.

The tracker is transforming the orientation, but does not transform the position of the tracked points with the tool offset. To get the tests which rely on change in orientation, were performed by using the calculated tool offset to change the tooldata in the RAPID programs.

To complete the functionalities that handles the orientation, and compensation of the tool offset, the problem with performing rotational transforms must be found and rectified. Because most of the tests were not affected by this, and using the tooldata as a workaround work perfectly, this problem was not prioritized, as it had already taken too much time without results.

5.2.3 Capturing Digital Inputs with the Tracker

Capturing digital inputs with the Vive worked well, with a simple circuit to handle the voltage difference between the output module of the robot controller, and the Vive controller. The *TrackerAPI* only represents a input from one of the buttons on the Vive controller for simplicity. The device that *RobTracker* retrieves from *ViveTracler* does have the state of all the buttons on the Vive controller, and could be further implemented

if there is an interest in handling several different inputs, for example with other external equipment.

5.3 Automated Testing of Industrial Robots

This section discusses how well the different test cases and the automated tests cases in general have worked. Also looking at what other types tests can be created to further utilize this solution.

5.3.1 Evaluating the new Tests

Most of the new tests have worked as intended, showing good results and being able to test the intended parameters.

5.3.1.1 General Movement (ATC-1)

This test case was designed to cover basic operation of the robot, as an initial test that verifies that there is no major errors when using the robot, that would result in any other test to fail. Such errors could be that the calculation of a work object is not transforming target coordinates correctly, or the robot configurations at different targets are incorrectly calculated.

The first test checks that the robot is moving in the correct direction, which as shown from the results works. This test base its first movement on the use of *MoveAbsJ* to get the robot in a known position. Initially this test should base its movement of the position that the robot is in when the test start, by retrieving its position of each joint and TCP form the robot. A good solution for this were not found, and then used *MoveAbsJ* as an alternative as this instruction does not use a target or a defined work object.

Checking that the position of the TCP corresponds to the defined target, worked as expected. In this case only a few points are tested in close proximity to each other. A more extensive test would be to check the positions at several targets that cover the entire reach of the robot, and with different work objects. This combined with checking different orientations at these points could be extended to a new test case. This would test several combinations of positions, orientations, work objects, and robot configurations.

Checking orientation in this test case were not able to complete, due to the lack of correct calculation of change in TCP orientation from the HTC Vive.

The two tests performing *MoveL* and *MoveC* instructions performed as expected. Performing the tests with change in orientation also verifies that the movement is calculated correctly as the orientation of the TCP is changing.

Testing with following a path the final test of this test case, and the purpose were to extend the existing test cases that draws patterns and figures on a paper to do more complex movement in more than one plane.

This test had to be slightly simplified than first planned, due to the challenges with tracking abrupt changes in direction. This means that this test was not able to verify overshoot, and each corner had to be larger zones, or at slow speeds.

Another challenge was the method of acquiring an "oracle" path to compare the tracked path with. This were done with the *SmartPathLogger* smart component that were loaded into a simulation in RobotStudio. The *OnSimulationStep* function called were spaced at 20ms apart, which were more than often enough for this implementation. If a better quality of tracking is acquired, and sharper turns of the TCP were to be performed, this might not be a good enough resolution.

5.3.1.2 Speed (ATC-2)

As the results in 4.2.3 and the earlier discussion in 5.1.4 shows, there are challenges with getting a good measurement of speed with the current HTC Vive setup, which affects the results of this test case.

With these challenges, there were only possible to verify the speed in a linear movement where the speed was averaged over a distance. Testing speed of linear movement worked well for all speeds tested, and should be capable for even faster speeds. The results showed a higher inaccuracy at higher speed, but still at a lower percentage of error, which could be acceptable.

To get speed at a specific point, or from a smaller section of a path, the inaccuracies of speed measurements from the HTC Vive would need to be improved. The current solution for measuring an accurate speed is to average 400ms of path. To get a good representation of the speed through a path with small zones, the window of such averaging should be reduced to 100ms. The actual value depends on the requirements and the hopefully improved speed measurements of the HTC Vive.

Testing speed in zones were intended to verify that the robot is maintaining its maximum speed through different zones. Several speeds were selected, and then the smallest zone which the robot was maintaining this speed were found through simulation. This test was not able to measure correct speed, due to the high speeds resulting in too much jitter on the measurements, and the slower speed moving through the zone much faster than the sample speed of the HTC Vive. Even with improved measurements from the HTC Vive, this test would be challenging to execute.

5.3.1.3 RAPID Operations (ATC-6)

The RAPID operation test case only has one test implemented, testing correct operation of starting and stopping a RAPID program. This test was designed as a direct implementation of one of the existing tests that verifies that a program can be stopped and resumed.

Similar tests to this could be created for other operations, such as the RAPID instruction *WaitTime*, which were intended but not prioritized to be implemented this time.

5.3.1.4 Trigger (ATC-7)

The trigger tests worked as expected, the connection from the digital output from the robot controller to the HTC Vive worked without any challenges. The precision of the tests was good, well within $20ms$, or $\pm 10ms$ if compensating for the experienced delay.

The existing tests specifies a $\pm 5ms$ delay on the triggered time to be within limits. This test was not able to be give stable results within this time deviation. Some of the inaccuracies of the results might be a result of the tracking precision when moving the HTC Vive, as the result is based on the distance and time difference to the tracked point closest to the trigger point.

When testing *TriggIO* with time offset, the results shows that the timing were between approx $[-1.7, 5.9]ms$ with $100ms$ offset, and $[-5.5, 6.2]ms$ for 200ms offset. These are almost within 10ms, and would probably have increased accuracy if the performance of the tracked position were increased, giving a better measurement of the time difference.

TriggEquip were not implemented in this test case, but would also be applicable, and be tested in much the same way as *TriggIO*, testing combinations of positional offset and equipment lag.

In this test case only *TriggL* were used. In further implementation, there is no restriction to implement other triggering motions, such as *TriggJ* and *TriggC*, further testing the different RAPID instructions.

This test case only connected one digital output to the Vive controller. A further integration would allow for connecting an analog output to the analog trigger input on the Vive controller. This would make it possible to perform these tests with an analog output as well.

5.3.2 More Potential Test Cases

From ABB's test documents, there were other parameters that were categorized into topic that were not created into test cases to limit the scope of this project. This section explains what other tests could be created to cover more of the existing test parameters. The implemented test cases and these topics were all drafted and given names and number.

Acceleration (ATC-3)

This test case is meant for verifying the correct operation of *PathAccLim*, and instruction that limits the acceleration and deceleration of the robot.

Smooth Movement (ATC-4)

ABB's test documents describe several tests covering different aspects of the robots smooth motion. These tests check for overshoot and correct drawing of figures and shapes. This test case is thought to be an extension of the general tests, and perform more exten-

sive tests that for example checks that a path can be performed when moving the work object around the robot.

Interpolation (ATC-5)

The test documents have several tests that verifies correct interpolation of circle and zone movements. These tests more advanced functionalities such as *ConfL* and *SingArea*. To be able to test these requires verifying the orientation of the tool.

This test case would also contain more comprehensive testing of zones and circular movement, by for example testing all predefined zones, and different sizes of circular movement.

RAPID Operations (ATC-6)

In this project only stopping and resuming a path was implemented. The existing tests also tests the *WaitTime* RAPID instruction. This test case could be expanded to test any other instructions that can be detected or verified based on movement.

Trigger (ATC-7)

There are more trigger instructions than *TriggIO* and *TriggL* to be tested, and this test case can be further expanded with *TriggEquip*, and other trigger movements such as *TriggJ*, and *TriggC*.

These tests could also be expanded to test triggering in more than only the straight movements done so far. By utilizing the freedom of movement with this solution triggering could be performed while executing complex paths.

Conveyor (ATC-8)

One large part of the existing tests has to do with synchronizing the robots movement with a conveyor belt. A conveyor belt positioned besides the robot give the robot controller information about its movement through an encoder, such that the robot is able to know its movement speed, and detect when the conveyor stops. These tests verify the correct movement of the robot relative to the conveyor, and is done by drawing with a pen on paper mounted to the conveyor belt.

The *TrackerAPI* supports multiple tracked devices, and can be implemented with the conveyor by mounting one tracked device on the robot, and the other on the conveyor belt. Then the tests can exchange the pen and paper with comparing the paths of the two tracked devices.

5.3.3 Comparing the new Test Method with Existing Method

The long-time goal is to replace the existing tests with automated tests that can perform these tests with the required quality. The secondary goal is to create automated tests

that can be used regularly and perform test that give a good indication if the test will pass or not.

Based on the results shown, the tests are capable of performing a wide range of testing, but few of them have the precision required to actually replace the existing tests. Testing general movement, stopping and resuming RAPID procedures, and verifying linear speed, would seem to be able to replace some parts of the existing tests, but this will have to be quality checked at ABB to be decided. If further improvements can be done to the *TrackerAPI* and the precision of the HTC Vive, more tests would be more qualified to replace the existing.

On the other hand, the tests have shown great potential as an addition to the testing, and allowing for a much more frequent testing, and the current quality of the system would most probably be able to give early warnings on potential errors in the developed controller software.

5.4 Further Work on the Tracker

This project has created a basis for automated tests, and is meant to be further developed and expanded. The tests that have been created does their job, but is also created to test the different functionalities of the tracker API.

To be able to set this project in actual use there are some work required to be done. One operation that must be performed when executing the tests is the calibration of the system. When calculating the offset parameters for the tracker, the robot controller must run a software that is confirmed to work properly, to ensure that the positions used is correct. This means that there must be implemented a system than can perform the calibration procedures and then change the robot controller software to the developing software that are to be tested.

Further as explained in section 5.2.2 the calculation of rotations has not worked properly, resulted in some inaccuracies with the calibration, and problems with determining the actual orientation of the tool. This is calculations that should be corrected to allow for using the tracker in paths with different orientations.

The limitations of the HTC Vive as a tracking device have been shown to not be as accurate as first indicated by some articles [1]. And there are other challenges with the use of this combined with a robot. As explained in section 5.1.6, there are several changes that could improve the quality of the tracking precision of the HTC Vive. These possibilities should be further investigated and tested out, to determine if it is possible to get as good results as shown in the mentioned article [1]. If the jitter can be reduced to $0.3mm$, and the general precision of tracking paths and speeds of the robot can be improved, the total capabilities of this tracking system would allow for more precise and extensive tests.

Chapter 6

Conclusion

This project has utilized the HTC Vive as a tracking equipment for ABB's industrial robots, used for automating test procedures. The Vive have performed as desired, and have shown to be a viable solution for this application.

Although the Vive did not perform as well as some articles would have suggested [1], it still tracks positions to within one millimeter, which have been adequate for this project. The results shown have indicated that the precision of the Vive can be improved with a better and more dedicated test setup.

To utilize the HTC Vive for tracking the industrial robot, a tracking API has been created with the required functionalities to retrieve positional data from the Vive and to store them in a way that represents the movement path of the robot. The API is capable to both track paths, and stationary points, and utilize longer measurement to increase the precision of a tracked point. This API will calculate the transformation required to convert the measured points with the HTC Vive into the same coordinate system used by the robot.

The tracker API has worked well, but there are still more features that could be implemented to increase the possible tests that can be performed. There have been challenges regarding calculating rotations which have given inaccuracies for transforming positions, and compensating for change in orientation of the tool.

The solution for mounting the HTC Vive controller to the robot have shown to be challenging due to the lack any good mounting options. The mounting used have worked well, but there are problems with vibrations from the robot interfering with the measurements in the Vive controller. This have caused the position when moving to be slightly fluctuating up to 1.5mm. The most problematic effect, is the speed calculation that the Vive performs itself, which have shown to be fluctuating with $\pm 200 mm/s$ and even more at higher speeds.

HTC Vive have recently released their new Vive Tracker, which is a small compact tracker device which is intended to be mounted on external equipment for use in a virtual environment. The Tracker have proper mounting point, and would probably be better at handling the vibrations from the robot with a damped mounting surface. This Tracker should be tested with this solution to determine if it can improve on the experienced challenges.

The implemented tests have demonstrated various ways that the tracker can be used to perform different tests. These tests are mostly based on the existing tests ABB performs,

but also expands where this system is capable to perform more complex tests, or perform several repeated tests.

The automated tests implemented in this project might be capable of replacing some of the existing test, and probably even more if the precision of the Vive can be improved. The versatility of the HTC Vive as a tracking device allows for a much wider set of tests, and based on the results shown in this project, these tests could be a very good supplement to the development phase of new controller software.

Utilizing the capabilities of these automated tests, and further implementing more, could improve the development process with easy and quick testing of the software. These tests can be started and performed at any time, without requiring an operator, and any failures or errors will have a high chance of being detected at a very early stage, significantly simplifying troubleshooting and debugging. This will make the process of the final test easier to complete as there will be a much higher chance of completing without significant problems.

Bibliography

[1] Oliver Kreylos. Lighthouse tracking examined. http://doc-ok.org/?p=1478, May 2016. Accessed: 15.01.2017.

[2] Joshua Baker, Timothy Kurisko, Haoran Li, David Poganski, and James Worcester. Development of iso compliant repeatability procedures for evaluating collaborative robots. April 2014.

[3] Graysen Christopher. Is vr ready for business use? six industries getting to grips with virtual reality. http://www.computerworlduk.com/applications/six-business-uses-for-virtual-reality-3641742/, June 2016. Accessed: 26.05.2017.

[4] iFixint https://creativecommons.org/licenses/by-nc-sa/3.0/. Htc vive images. https://www.ifixit.com/Teardown/HTC+Vive+Teardown/62213. Accessed: 27.05.2017.

[5] Alan Yates. The lighthouse tracking system. https://www.youtube.com/watch?v=75ZytcYANTA, December 2016. Accessed: 23.01.2017.

[6] Tested. Steamvr's "lighthouse" for virtual reality and beyond. https://www.youtube.com/watch?v=xrsUMEbLtOs, May 2015. Accessed: 21.01.2017.

[7] Valve Alex Vlachos. Valve lighthouse slide from "advanced vr rendering". http://media.steampowered.com/apps/valve/2015/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf. Accessed: 28.05.2017.

[8] ABB Robotics. *Technical reference manual RAPID Instructions, Functions and Data types.*

[9] Valve Corporation. Steamvr troubleshooting. https://support.steampowered.com/kb_article.php?ref=8566-SDZC-9326, 2015. Accessed: 07.06.2017.

[10] Hewlett-Packard Development Company. Vr room-scale setup htc vive. http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA6-9648ENW.pdf, April 2017. Accessed: 07.06.2017.

[11] HTC Corporation. Htc vive tracking developer guidelines. https://dl.vive.com/Tracker/Guideline/HTC_Vive_Tracker_Developer_Guidelines_v1.4.pdf, 2017. Accessed: 25.05.2017.

Appendix A

Attachments

Source code and TrackerAPI documentation:

Demonstration video: