OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FACULTY OF COMPUTER SCIENCE

Communication and Networked Systems

Communication and Networked Systems

Bachelor Thesis

Teleoperation of Quadrotor Drones using Haptic Devices

Jon-Mailes Graeffe

Supervisor:	Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor:	MSc. Frank Engelhardt

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

May 11, 2022

Abstract

Abstract

Teleoperation is omnipresent in our world – basically everything that is controlled remotely is teleoperation. One application of teleoperation is remotely controlling quadrotor drones, which are gaining more and more popularity in all sectors and are the perfect match for being integrated into the Internet of Things (IoT). Haptic devices allow multi-dimensional input and provide touch sensation as an addition to the conventional visual feedback. Projects already exist that research the combination of all three, the teleoperation of quadrotor drones using haptic devices as input over Wireless Local Area Networks (WLANs) or Wireless Wide Area Networks (WWANs). However, few platforms exist that provide the big picture, or in other words combine all aspects such as control, networking, encoding and compression of haptic data, into one implementation that can in turn be used for testing and evaluation. Also some ways on how to control a drone with haptic devices remain unexplored.

This thesis provides the first iteration of a demonstrator implementation that aims to contribute to the beforementioned problems, and documents the concepts and implementation itself so that it can be used and extended in future research projects.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACK Acknowledgement. 9, 56

API Application Programming Interface. xvi, 1, 18, 25, 26

CI confidence interval. 46, 48

CoAP Constrained Application Protocol. vii, xi, xv, 3, 5, 8–10, 26, 36, 39, 56, 68, 69

CON Confirmable. 9, 36, 56, 69

DoF degree of freedom. 3, 6–8, 13, 14, 57

DTLS Datagram Transport Layer Security. 26

FPS frames per second. 16

FWT Fast Wavelet Transform. ix, xvi, 12, 13, 26, 38, 46–48, 51, 57

HTTP Hypertext Transfer Protocol. xv, 8

IDL Interface Definition Language. 24, 37

IMU Inertial Measurement Unit. 19

IoT Internet of Things. iii, 1

IP Internet Protocol. 9, 10, 56

IPv4 Internet Protocol Version 4. 55, 56

JSON JavaScript Object Notation. 20, 24

MAE Mean Absolute Error. 51, 53

NED North East Down. 21

NON Non-confirmable. 9, 36, 56

OVGU-HC OVGU Haptic Communication Testbed. 3, 58

PCL Point Cloud Library. 26, 34, 43

PID Proportional-Integral-Derivative. 20, 30

POSIX Portable Operating System Interface. xv, 26

RMSE Root Mean Square Error. 51, 53

RPC Remote Procedure Call. 20, 44, 46

RST Reset. 9

RTT round-trip time. 16, 17, 45

TCP Transmission Control Protocol. xv, 9

UAV Unmanned Aerial Vehicle. 1, 11

UDP User Datagram Protocol. xv, 9, 10

URI Uniform Resource Identifier. 8

WLAN Wireless Local Area Network. iii, 1, 17, 18, 56, 57

WMHN Wireless Multi-Hop Network. 3, 57

WWAN Wireless Wide Area Network. iii, 1, 17, 18, 56, 57

Glossary

3D Systems Touch™ Haptic device named Touch™, developed and manufactured by 3D Systems. vii, ix, 7, 8, 17, 23, 27–29, 34, 39

AirSim Open-source simulator focusing on the simulation of quadrotor drones. 19–21, 23, 27, 44, 45

CHAI3D C++ simulation framework focusing on haptic devices as input and output peripherals by the means of force feedback. 21–23, 33, 34, 39, 43, 44

CMake Open-source build system that is able to generate projects for IDEs and toolchains from universal build definitions. 33, 34

Constrained Application Protocol Application layer protocol similar to HTTP, specifically designed for use with constrained nodes and networks. xiii, 3

Datagram Transport Layer Security Encryption protocol that is based on TLS (used by e.g. HTTP, more generally protocols that base on TCP) but that can be used with connection-less protocols such as UDP as well. xiii, 26

Fast Wavelet Transform Mathematical algorithm to efficiently turn a signal time series into a sequence of approximation and detail coefficients, which can be turned back to the original time series with an inverse function. If only the approximation coefficients are transmitted, discarding the detail coefficients, this can be used as a form of compression. xiii, 12

FlatBuffers Open-source cross platform serialization library for C++, Go, Java, Python and lots of other languages. xv, 24, 25, 35, 37, 38, 42, 56

Interface Definition Language A language that describes interfaces in a language-independent way, commonly used for defining data models for cross-language operability. In the thesis, FlatBuffers defines such a language. xiii, 24

JavaScript Object Notation Compact file format that supports nested key-value pairs and arrays. xiii, 20

libcoap Open-source C implementation of CoAP that works on both POSIX systems as well as embedded ones. 26, 36, 68, 69

LIDAR Method for determining distances to an object by targeting it with a laser and measuring the time for the reflected light to return commonly used to scan three-dimensional objects resulting in a digital representation of said objects, in this thesis in a point cloud. 13, 19, 21, 33, 34, 43, 44, 48, 49, 54, 58

MAVLink Messaging protocol commonly used as an interface to flight controllers, such as PX4. Can be used to communicate with drones that support it. 27

North East Down Local coordinate reference system that defines the north axis to be the x axis with positive values being north, east axis to be the y axis with positive values being to the east et cetera. xiii

operator Entity operating (controlling) a teleoperator (e.g. vehicle, robot) remotely, for example a human tasked with controlling a drone. xvi, 1, 5, 6, 11–13, 15, 17, 27–33, 39, 51, 53, 54, 56–58

Point Cloud Library Open-source framework containing numerous C++ libraries for 2D/3D image and point cloud processing. xiii, 26

Portable Operating System Interface Standardized API interfacing between the operating system and applications. Lots of popular operating systems support it. xiv, 26

predictor display Visualization of a prediction of the teleoperator's future state and environment based on operator inputs, before actually enforcing them on the real teleoperator / in reality. 32, 33

PX4 Open-source flight controller (autopilot) for quadrotor drones. 24, 27

Remote Procedure Call Form of Inter-Process Communication; causing procedures to execute in different address space, e.g. a process initiates a call to a function in another process, possibly over the network. xiv, 20

teleoperation Operator operating (controlling) a teleoperator (e.g. vehicle, robot) remotely. iii, 1, 3, 5, 6, 11–13, 15, 30–32, 58

teleoperator Entity that is operated (controlled) by an operator remotely, for example a drone controlled by a human. xvi, 1, 5, 6, 13, 16, 32

vcpkg Open-source cross-platform packet manager that focuses on distribution of C and C++ libraries. 34

wavelib Open-source C implementation of various wavelet-based transformations such as FWT. 26

CHAPTER 1

Introduction

The applications of teleoperation are omnipresent in today's lives for many years, from RC cars that children play with, to robots performing telesurgery [1]. Basically everything you control remotely is teleoperation.

Also quadrotor drones, in the following thesis referred to as drones and commonly called quadcopters, are a type of Unmanned Aerial Vehicles (UAVs) that use four rotors instead of two like a regular helicopter, and gain popularity as a teleoperator in all sectors, be it personal, commercial, governmental or military. For example, they are commonly used personally or commercially for entertainment purposes, such as competitive drone flights and videography. They can also be used for surveillance, search and rescue missions (think finding humans to be rescued in a collapsed building or in avalanche accidents), packet delivery by Amazon or DHL, or even spanning emergency communication networks in response to disasters quickly [2], [3].

Due to their four rotors that provide lots of stability and the ability to safely hover in the air, quadrotor drones are predestined for attachment of payloads such as cameras, sensors et cetera. Figure 1.1 illustrates such a drone on the left with its aerial axes, which will be relevant in the thesis. On the right side of Figure 1.1 you can see a remote control that is typically used to control the drone's axes manually by a human drone operator.

In addition to the four rotors, they have an embedded computer on board that is constrained in its resources regarding battery capacity, memory and network capabilities, which is called the flight controller and usually expose Application Programming Interfaces (APIs) to programmatically control the drone over wireless communication, by becoming a part of WLANs or WWANs. The easy-to-access interfaces make the drone the perfect device to be embedded into the IoT, which does not seem to be falling in interest with the ever-rising number of connected devices that is forecasted to reach 75 billions by 2025 [4].

Haptic devices exist that allow positional and orientational input in three-dimensional space, which exceeds the capabilities of the conventional remote controls of commercially available drones. In addition to that, lots of haptic devices act as an output device too, providing touch sensations as haptic feedback, extending the common visual feedback of control systems. They are vastly relevant to the rapidly emerging *Tactile Internet* and human-machine

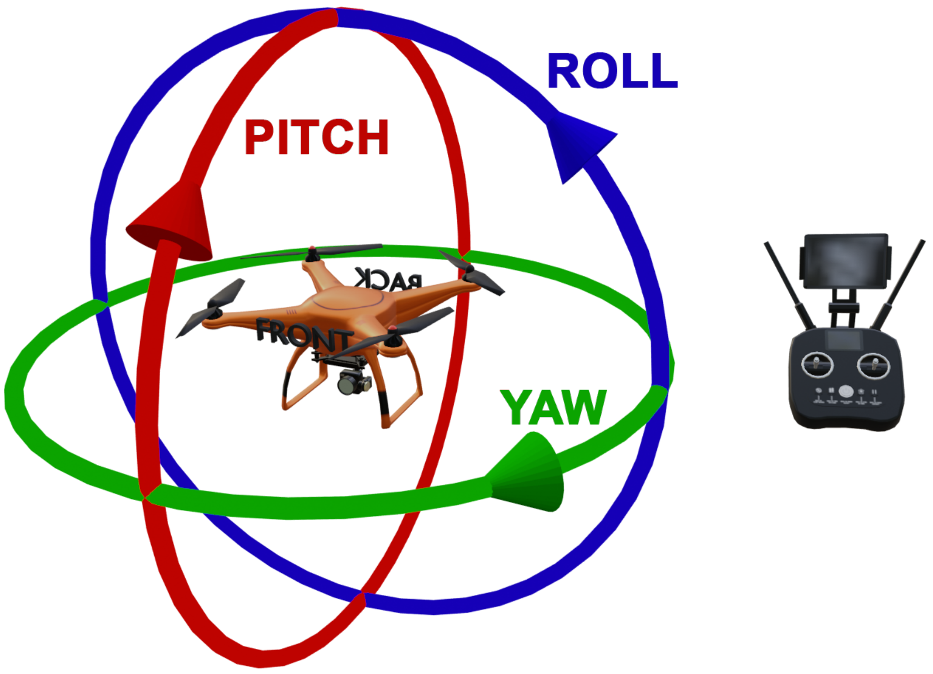


Figure 1.1: Picture of a generic quadrotor drone on the left side, which happens to have a camera on board. The rotational axes are sketched into the picture: the green arrow represents the yaw, the blue arrow the roll and the red arrow the pitch. On the right side sits a typical remote control that is used to control the drones remotely, with a tablet providing a camera feed and additional information.

interaction in general due to their variability and high dimensionality, and already have lots of applications in medicine, design and more (see Section 2.2 for more applications).

No wonder that projects combining all technologies mentioned above sprout currently: the teleoperation of quadrotor drones with haptic devices as the input device, assisting the human pilot by extending his or her visual feedback with haptic feedback.

1.1 Thesis Contribution

In Section 3.4 it will be shown that there are already plenty of demonstrators implementing the teleoperation of quadrotor drones with haptic devices in order to research, test and evaluate the control performance, network requirements and advanced topics such as collision avoidance. However, few projects provide a platform that implements and allows to evaluate a broad spectrum of aspects on control systems – they focus on single, individual aspects such as haptic communication and control performance.

Also, the projects listed in Section 3.4 do all implement their own software platform without providing the implementation itself or details on the implementation, which causes 'reinventions of the wheel' quite often.

That and the urge to explore new ways of using six degree of freedom (DoF) haptic devices to control a drone is why a demonstrator is implemented as the main contribution of this thesis that attempts to cover as much aspects as possible, while being as modular and extendable as possible. It will be part of the OVGU Haptic Communication Testbed (OVGU-HC) at the Otto-von-Guericke University of Magdeburg, which already provides a platform to do data-driven experiments on Wireless Multi-Hop Networks (WMHNs) [5]. The testbed brings many benefits such as an experiment definition language and automatic scheduling of experiments, and it will feature 200 nodes in the future, therefore allowing to run experiments on complex wireless and wired topologies. For now, on the first iteration, the implementation will not be evaluated as part of the testbed for simplicity and due to time constraints.

Said modularity and extendability allows to easily replace and add small parts of the implementation by providing intuitive interfaces with the goal being to provide a platform for further research on newly emerging communication technologies, different control and compression algorithms and a variety of drone hardware. The thesis will act as sort of an out-of-code documentation for the implementation, so that future developers understand the structure of the demonstrator and get to know how to add new hardware drones with minimal effort.

1.2 Thesis Structure

The next chapter, Chapter 2, revisits the fundamental topics such as teleoperation and haptic devices. Also, the Constrained Application Protocol (CoAP) is introduced shortly in Section 2.3.

Afterwards, Chapter 3 introduces some of the work related to the thesis topics like quadrotor drones, haptic devices and haptic communication, and ends with Section 3.4 comparing existing demonstrators with the thesis implementation.

Chapter 4 starts with the definition of the requirements to the implementation, followed by the description of the software architecture and the control system concept in Section 4.3. Then, Section 4.4 goes into detail about the implementation's messaging system and source code.

In Chapter 5 the thesis implementation is evaluated by experiments regarding various latencies and finds out if the requirements set to the implementation are fulfilled. Section 5.2 analyzes the required data rates of the implemented control modes.

A summary and an outlook for possible future work is given in Chapter 6.

CHAPTER 2

Background

Some of the concepts and technologies may not be common knowledge to you or need refreshment, such as teleoperation, haptic devices and the communication protocol used in this thesis. That's why this chapter revisits the terminology of teleoperation in Section 2.1, Section 2.2 introduces what haptic devices are and CoAP is summarized briefly in Section 2.3.

2.1 Teleoperation

Teleoperation is an operator (e.g. a human, in this thesis the drone pilot) operating a teleoperator (e.g. robot or vehicle of some sort, in this thesis it's the quadrotor drone) remotely. It is not new at all and terminology has been well established. This section briefly revisits that terminology.

Controlling a teleoperator can be done in different ways [6]. *Direct control* means that the operator controls the teleoperator directly, so making inputs are directly translated to motion or some other kind of action by actuators on the teleoperator, and information provided by sensors is directly perceived by the operator. In direct control, there is one closed control loop consisting of the operator and the teleoperator. *Supervisory control*, in contrast, is a control scheme in which the teleoperator has the capability of making decisions and controlling actuators based on sensor inputs itself, autonomously [6]. The overall goals of the teleoperator are still dictated by programming of an operator or another entity, though. Instead of only one control loop between the operator and the teleoperator, there are now two control loops – one between the operator and the teleoperator which exists so that the operator can set goals based on what happens (what is perceived from sensors or direct sight), and another one on the side of the teleoperator that controls the actuators based on the dictated goals and sensory information. Figure 2.1 shows the difference between the two control schemes by illustrating the control loops.

Regardless of the control scheme used, teleoperation introduces challenges [1]. Hardware-wise, enough sensors and actuators need to be installed that have enough precision and accuracy, so that the operator and/or the teleoperator can make decisions that result in

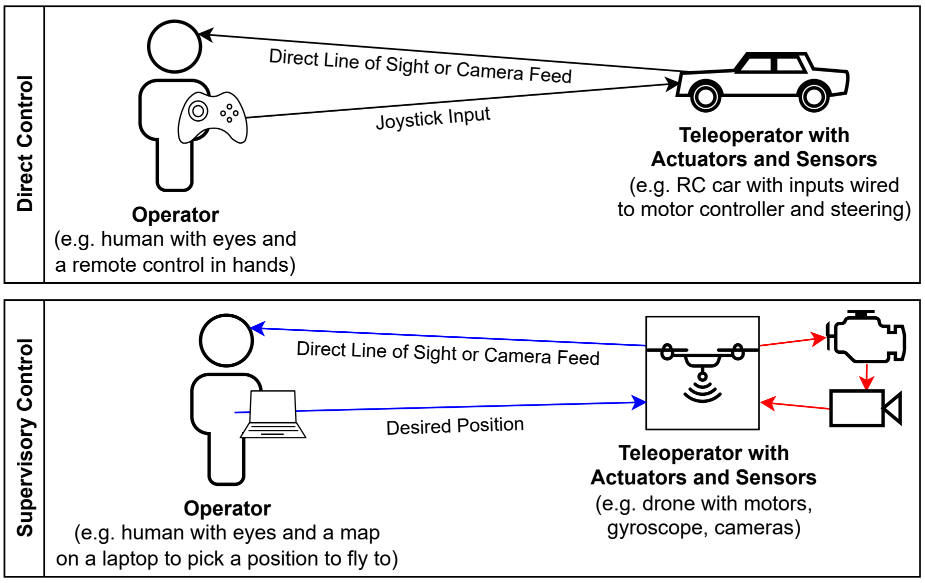

Direct Control
Direct Line of Sight or Camera Feed
Joystick Input
Teleoperator with Actuators and Sensors
(e.g. RC car with inputs wired to motor controller and steering)
Operator
(e.g. human with eyes and a remote control in hands)
Supervisory Control
Direct Line of Sight or Camera Feed
Desired Position
Teleoperator with Actuators and Sensors
(e.g. drone with motors, gyroscope, cameras)
Operator
(e.g. human with eyes and a map on a laptop to pick a position to fly to)

Figure 2.1: Sketch showing the fundamental control schemes in teleoperation.

sensible actions. Also, due to the distance between the teleoperator and the operator, time delays occur which need to be kept low enough to maintain stable control. Those challenges need to be addressed by the implementation of the thesis, which requires research on the nominal values needed at minimum to provide an acceptable control experience.

One popular strategy to at least overcome the issue of time delay is called *Move-and-Wait* [1]. It is very simple: instead of the operator trying to achieve continuous operation (e.g. continuous movement) of the teleoperator, the operator makes only some inputs, then waits for the operations to be done, and observes the resulting state before making the next inputs. This theoretically allows for arbitrary large time delays and is often used in space teleoperation, but only works for teleoperators that do not require continuous inputs. For example, a robotic arm does not have any problem with not moving for a long time, while an aircraft might just crash if no inputs are done for a period of time.

2.2 Haptic Devices

In general, haptic devices are devices which can create an experience of touch, or in other words tactile sensation to living creatures. Haptic feedback can be given in various ways, for example vibrations or applying forces to the user's hand.

There are a lot of different devices that fit into the definition of a haptic device, such as sensory gloves, exoskeletons for arms and bodies, platforms that you can move on et cetera [7]. Technically, a game controller such as those used for making inputs to a gaming console, is a haptic device already if it has a vibrator built into it that provides minimal tactile feedback to its user. Because the vibration feedback is only one dimension that can be sensed (the strength of the vibration), a game controller with vibration has one degree of freedom (DoF).

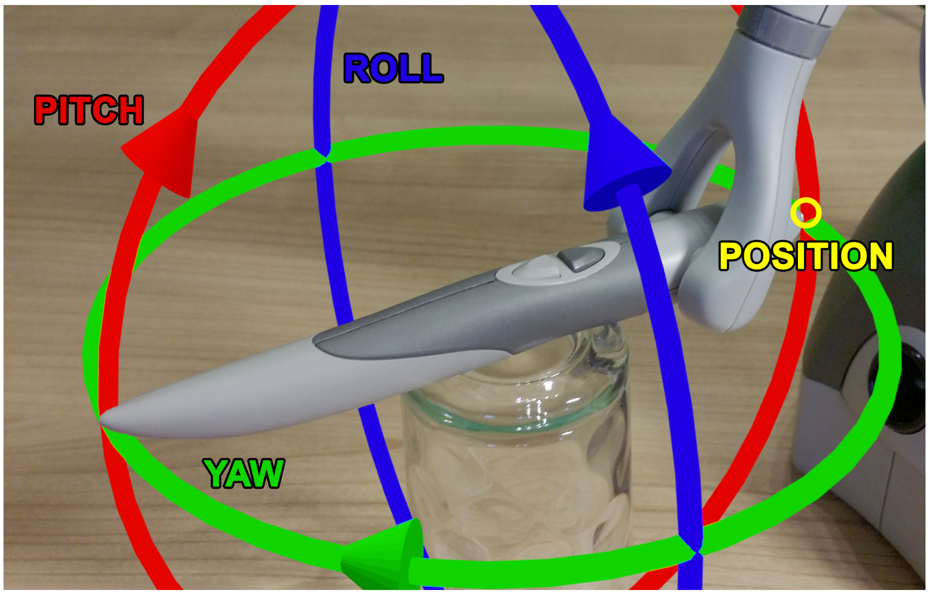


Figure 2.2: Axes provided by the 3D Systems Touch™ haptic device which has six DoF, meaning the the x, y and z coordinates that point to the tip of the haptic tool (stylus) of the haptic device marked by the yellow circle, as well as the yaw, pitch and roll angles represented by the green, red and blue arrows, can be read from the device. The z axis is the vertical one.

However, in this thesis, one refers to a special kind of desktop haptic input and output device that allows to make inputs in the form of a 3D position by moving a haptic tool, such as a pen or a gripper, in three-dimensional space, along with the force vector applied to said haptic tool. Those devices usually also provide haptic feedback by applying forces to the haptic tool and subsequently the user's hand. They come in various configurations with different DoFs. For example, three DoFs mean that the haptic tool can move freely in a 3D workspace area around the haptic device, and usually also apply a force vector three-dimensionally. Devices with six DoFs additionally read the orientations (yaw, pitch, roll angles) of the haptic tool in addition to the positional dimensions which allows for more sophisticated control schemes. Naming the rotational angles yaw, pitch and roll, which are visualized in Figure 2.2, comes primarily from aeronautics. For a typical aircraft such as those used in airliners, yaw is the nose heading left or right about an axis from top to bottom, pitch is the nose up and down about the axis from wing to wing and roll is the rotation on the axis from tail to nose (nose is the front where the cockpit lies, tail is the back where the stabilizers and elevators sit).

Applications of haptic devices are very diverse and interesting, ranging from art over design to medicine [7]. Like mentioned before, vibration is used as a feedback in game controllers to note certain events or situations in video games, and also in aviation to signal stalls by vibrating the pilot's controls. More sophisticated forms of haptic feedback, like with the beforementioned six DoF devices, are used in graphic design to draw on virtual canvases or to form 3D models interactively. There are also medical applications such as surgical simulations, endoscopy and laparoscopy.

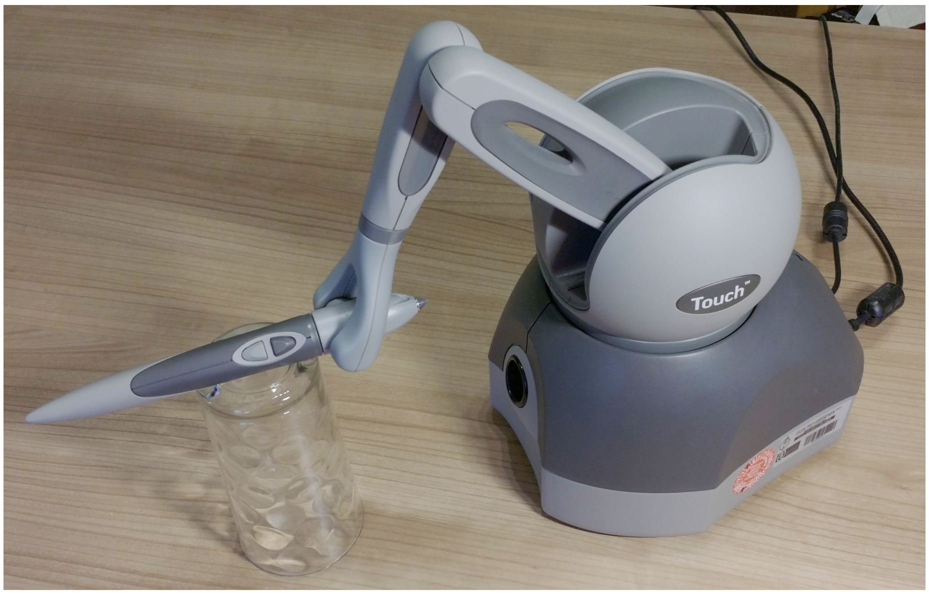


Figure 2.3: Picture of the 3D Systems Touch™ haptic device. The glass is not part of the device and is only utilized as a professional holding solution.

Workspace	width 160 mm, height 120 mm, depth 70 mm
Weight	1.42 kg
Nominal Position Resolution	> 450 dpi, ≈ 0.055 mm
Maximum Exertable Force	3.3 N
Continuous Exertable Force (24 hours)	> 0.88 N
Stiffness	X axis $> 1.26\,\mathrm{N\,mm^{-1}}$, Y axis $> 2.31\,\mathrm{N\,mm^{-1}}$, Z axis $> 1.02\,\mathrm{N\,mm^{-1}}$
Inertia (mass at tip)	≈ 45 g
Interface	USB 2.0

Table 2.1: Specifications of the 3D Systems Touch™ haptic device.

In this thesis, the specific haptic device hardware used is the 3D Systems Touch™ pictured in Figure 2.3. It has six degrees of freedom (6 DoF), features a stylus as its haptic tool and is characterized in Table 2.1. Any specification provided about the device is sourced from its user guide, which also gives instructions about connecting the device to a PC, installing required device drivers and how to physically use the device properly [8].

2.3 Constrained Application Protocol

CoAP is an application layer protocol similar to the Hypertext Transfer Protocol (HTTP), specifically designed for machine-to-machine communication and use with constrained devices (e.g. 8-bit microcontrollers with very small amounts of memory) and networks [9]. It provides a request/response model between HTTP-like endpoints that are identifiable via Uniform Resource Identifiers (URIs), which makes the protocol suitable for web applications

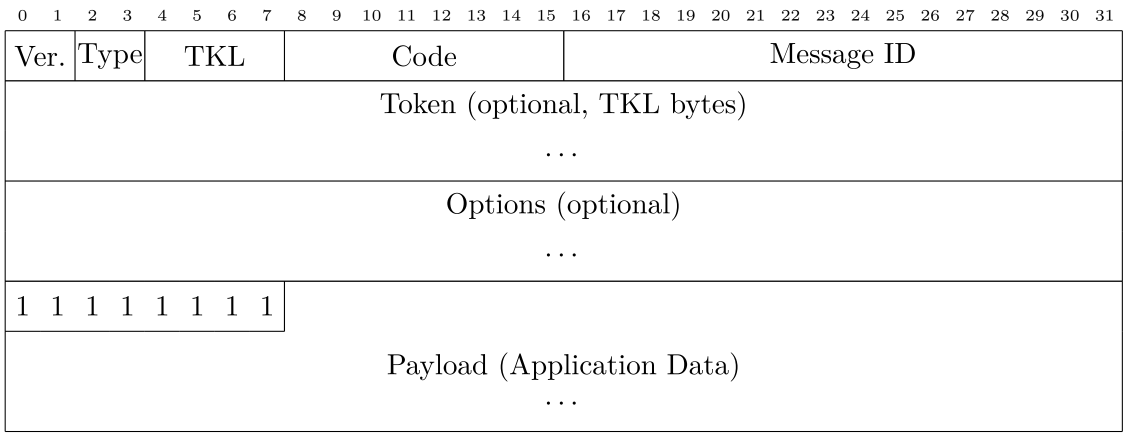


Figure 2.4: Format of a CoAP message, consisting of a four byte header, optional token and options and the payload, which is basically the data that is to be transmitted.

but is not limited to those applications at all:

```
GET coap://<ip-address>/pipe/pressure
POST coap://<ip-address>/cauldron/temperature
...
```

By default, CoAP is supposed to be used with the User Datagram Protocol (UDP) as transport to exchange individual messages, but other means of transportation like TCP or even non-IP-based methods could be utilized, too. Messages can either contain a request, a response or something else.

Figure 2.4 shows the message format used by CoAP, more specifically that messages always contain a header with, among other things, the type of the message, the code (for requests which request method such as `GET`, `POST`, `PUT`, `DELETE`; for responses the status code similar to success and error codes in HTTP), a message ID to identify duplicates and match acknowledgements and a token to correlate requests and responses. There are four types of messages:

- Confirmable (CON),
- Non-confirmable (NON),
- Acknowledgement (ACK) and
- Reset (RST).

CON messages, that can be both a request or a response, must be acknowledged with either an ACK message or be rejected by sending a RST message by the recipient. If such a CON message is not acknowledged in time, it is retransmitted by the sender multiple times until a timeout occurs. So, using this type of message ensures reliable transmission and should be used when it's crucial for the recipient to get the data.

NON messages on the other hand do not need to be acknowledged, thus no retransmissions will be done by protocol implementations on their own. Using this type of message makes sense when reliable data transmission is not required, e.g. when losing some of the messages is not critical or retransmissions are to be avoided to not congest the network anymore.

Apart from the obvious advantages of CoAP such as the low data overhead compared to its alternatives and the small memory footprint of its implementations, which make CoAP very versatile as it can be used on basically any transport or device (think cheap constrained onboard computers for quadrotor drones), it features reliability, congestion control and a common format to be used regardless of the transport [9], [10]. Also, efforts are being made to extend the protocol with features such as message encryption and authentication (even though you can of course use a transport that already provides those features) [11]. Generally CoAP is extendable due to its option system, which already complements the requirement of modularity and extendability that will be introduced in Section 4.1.5. Because of those advantages over other lower layer protocols such as UDP or even raw Internet Protocol (IP) packets it is preferable to use CoAP on top.

CHAPTER 3

Related Work

In the following, papers and books will be introduced that provide research regarding the thesis topics, primarily about controlling quadrotor drones or drones in general in Section 3.1, teleoperation with haptic devices in Section 3.2 and the encoding as well as compression of kinesthetic data in Section 3.3. At the end of the chapter, in Section 3.4, several existing demonstrators will be compared with the thesis implementation to show that there is currently no platform to be found such as the one implemented for this thesis and also that the thesis contributes to the research field by providing new aspects.

3.1 Control of Quadrotor Drones

A lot of research exists regarding the teleoperation of quadrotor drones, from algorithms to control the drone's motors, over strategies that avoid collisions based on visuals and depth maps, to work focusing on the communication between the drone and the operator through networks. That makes sense as quadrotor drones are gaining popularity even for private use (entertainment, hobbyist filming, competitive drone flights et cetera).

It is impossible at this point to cover all of the research about the different aspects of drone control due to the sheer number, but to name a few, Odelga et al. recently presented a quadrotor drone platform that utilizes RGB-D cameras (color cameras with depth value for each pixel) to build a model of the obstacles in the drone environment to autonomously avoid collisions [12].

On the communication side, Riestock et al. performed a user study on the required bandwidth to control drones for two different means of environment representations that are used as visual feedback to the human operator, namely camera images and voxel-based grid maps [13].

For more research regarding controlling mostly quadrotor drones Kangunde et al. provided a broad survey in [14], which additionally includes work about the hardware and operating systems commonly used for quadrotor drones and other UAVs.

3.2 Haptic Devices for Teleoperation

Regarding teleoperation with the help of haptic devices in general, there is also a lot of research available. To get an overview about haptic devices and their capabilities and applications, [7] is suggested as a survey by Laycock et al.

Another contribution was that Wildenbeest et al. did experiments on the impact of haptic feedback on the performance of assembly tasks that are done remotely, and found out that providing haptic feedback substantially improved the overall performance of the operators [15].

Antonakoglou et al. touched a different aspect, the haptic communication, in their survey and discussed several methodologies and technologies regarding haptic communication [16]. One of the main focuses was how 5G wireless technology can be used for haptic communication and will shape the Tactile Internet in the presence and future.

3.3 Compression of Kinesthetic Data

The compression of kinesthetic data such as position, velocity and force vectors or rotation matrices is also a really active topic in research. Hinterseer et al. published several articles regarding the compression of haptic data. For example, in [17] an approach was proposed that reduces haptic data to six to ten percent by using a combination of Kalman filters and model based prediction on haptic signals.

In [18] Hinterseer et al. used a psychophysically motivated compression algorithm that only generates and sends packets if the change of the values of the haptic data exceed a threshold at which differences are noticeable by the human operator, which managed to reduce packet rates by up to 90 percent without any human-perceivable difference.

Another example of a technique that can be used to encode and compress data in haptic communication is proposed by Engelhardt et al. which bases on block-wise Fast Wavelet Transform (FWT) [19]. This method is implemented in the thesis implementation, and will be discussed later in Section 4.4.1.

Steinbach et al. provides both the fundamentals and an overview of the state of the art of haptic codecs, or in other words the ways how haptic and tactile data can be encoded efficiently in the first place and also compressed to lower bandwidth requirements [20], which is a good start for further reading on the topic.

3.4 Comparison of Demonstrator Implementations

Now, let's take a look at work that provides some kind of implementation of a demonstrator or at least the concept of it, which makes it comparable to this thesis. For work like that, the following aspects will be compared to show that no platform is currently to be found that combines all of them:

a. *teleoperation* over networks generally,

b. *direct control* as explained in Section 2.1,

c. *supervisory control* also introduced in Section 2.1,
d. *input by haptic device* to control a teleoperator,
e. *haptic feedback* to complement visual feedback such as a camera feed with sensory information,
f. the *DoF* of the haptic input and possibly feedback if provided,
g. *quadrotor drones used as the teleoperator* specifically as well as
h. *compression* to minimize data flow between components.

As an early example, Lam et al. implemented two different haptic feedback techniques to control some kind of UAV helicopter with an electro-hydraulic aircraft stick (so most likely a two DoF haptic device), of which one applies force based on the distance to an obstacle directly and the other one representing the distance through the springiness of the haptic tool instead to avoid an overshoot of the haptic tool when released by the operator [21].

Rognon et al. used another kind of haptic device – an exoskeleton called the *FlyJacket* which uses inflatable air pouches to render the sensation of air pressure to the operator while flying in order to improve the situational awareness [22]. The FlyJacket seems to be a haptic device with only one DoF and the used drone was not a quadrotor drone but rather a conventional aircraft (no helicopter).

DronePick is a project by Ibrahimov et al. that uses the haptic feedback of an electrical glove to haptically visualize objects that can be grabbed with a magnetic grabber that is attached to a quadcopter, and the glove is also used to control the drone simultaneously by moving the drone to the relative position of the glove to a zero point [23]. The object is scanned by a LIDAR sensor and the goal is to help the operator to find out when the grabber is above the object. It is not entirely clear how much degrees of freedom the glove has, but it probably provides at least six DoF for individual fingers.

Macchini et al. used a glove to control a quadcopter too and performed several user studies on the learning path and control performance of operators [24]. The glove was tracked with a motion capture system and equipped with multiple so-called tactors that can vibrate as force feedback. Due to the fact that a motion capture system was used, it is likely that the input had six DoF.

Another haptic device, more specifically an arm and wrist sleeve, was proposed by Ramachandran et. al that reads the elbow and wrist angles to control a drone's altitude and attitude [25]. The sleeve also allows to block the movement of the joints of the operator's arm and hand as a sort of haptic feedback, so that control into a specific direction can be forbidden if an obstacle is near. Because two angles are read, the device can be classified as a two DoF haptic device.

The paper about the FWT compression mentioned earlier also implements a teleoperation system in which a *WiiMote®* haptic device is used to control the speed of a quadrotor drone [19]. A WiiMote® has six DoF but without force feedback, and in the paper the control is restricted to two DoF for the sake of simplicity, with the goal of evaluating the compression and not the control experience itself.

Table 3.1 illustrates that there are lots of existing projects that implemented drone control

		a	b	c	d	e	f	g	h
[21]	Lam et al.	✓	✓	✗	✓	✓	2 DoF	❍	✗
[22]	Rognon et al.	✓	✓	✗	✗	✓	1 DoF	✗	✗
[23]	Ibrahimov et al.	✓	✓	✗	✓	✓	6 DoF	✓	✗
[24]	Macchini et al.	✓	✓	✗	✓	✓	6 DoF	✓	✗
[25]	Ramachandran et al.	✓	✓	✗	✓	✓	2 DoF	✓	✗
[19]	Engelhardt et al.	✓	✓	✗	✓	✗	2 DoF	✓	✓

Table 3.1: Comparison of related work regarding the aspects a to h defined in Section 3.4. A '✓' means that the aspect is present in the corresponding work, a '✗' means it is non-existent and a '❍' means that it is unclear whether the feature is present or not.

with haptic devices, but since there is no demonstrator that combines all the aspects, the thesis implementation that does exactly that is justified and makes sense as a contribution to the research community.

CHAPTER 4

Thesis Contribution

One of the main results of the thesis is the software. This chapter gets into detail about the requirements to the implementation, its underlying concepts and how the implementation itself is done.

As figured out in Section 2.1, teleoperation introduces challenges in terms of time delays and also haptic devices need to be provided with data in specific frequencies so that haptic perception is smooth. Section 4.1 researches nominal minimums (or maximums, respectively) for those delays and frequencies, and also introduces additional general requirements to the software from a software engineering perspective, such as extendability and modularity. The implementation provided depends on some external software, mostly libraries and frameworks, which are shortly introduced in Section 4.2. Before going into details about the implementation itself in Section 4.4, Section 4.3 conceptualizes how the software is structured generally and how the operator of the drone shall be able to control the drone exactly.

4.1 Software Requirements

In this section the nominal and qualitative requirements to the overall control system and the implementation itself will be analyzed and defined.

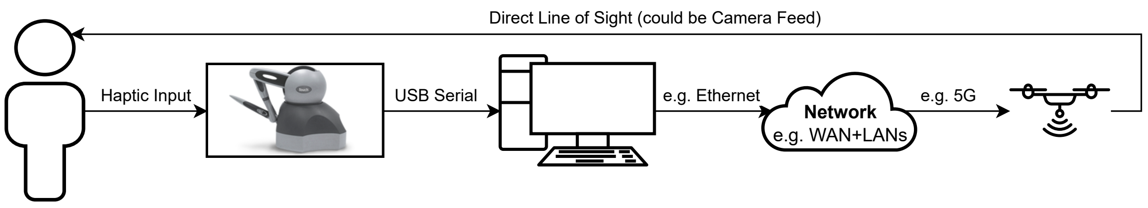


Figure 4.1: Abstract control loop in the grand scheme of things.

4.1.1 Control Frequency

The control loop with the lowest frequency, which most likely will be the bottleneck in terms of control performance and experience, consists of the human teleoperator perceiving how the drone behaves and making inputs to the haptic device based on that, as sketched in Figure 4.1. An upper bound and a lower bound exists for the frequency of that control loop. While the upper bound is determined by the limits of how fast the human eye can perceive changes such as movements of the drone, the lower bound is the frame rate perceived by the human eye at which control performance is acceptable for a specific scenario. For the lower bound, [26] suggests that acceptable psychomotoric performance can be accomplished with frame rates provided to the user as low as 10 Hz, but also mentions that higher frame rates are beneficial. However, the upper bound does not seem to have a clear consensus in science. Generally though, higher frame rates appear to be beneficial in terms of smoothness and subjective quality up to 240 frames per second (FPS), but the improvements stagnate with increasing frame rate [27].

As a consequence, it makes sense to set the minimum required frequency for control of the drone to 10 Hz. To make the movement appear smoother to the teleoperator, higher frame rates are desirable but not required. In the following experiments the frequency will be set to

$$f_{control} = 60\,\mathrm{Hz}$$

because this is the maximum frequency proved to be working well with the simulator that will be used instead of a real hardware drone. Even though higher frequencies would objectively improve user experience and control performance as suggested before, they unfortunately caused visual stuttering which was not the result of a hardware bottleneck but rather implementation of either the flight controller or something else in the physics engine of the simulator used. More details on the simulator itself will be given in Section 4.2.1.

An argument could be made that 60 Hz is also the refresh rate of most computer displays and therefore the frame rate put out by the simulation rendering used for development and experiments if vertical synchronization is enabled, which it is by default. However, it has been proven that in video games frame rates higher than the refresh rate of the display are beneficial in terms of input lag [28], or in other words, changes to the inputs are considered by the physics engine earlier even though those changes are not visually rendered yet. This applies to this project too, so theoretically it would be better to increase the control frequency if the simulator would allow to do that.

4.1.2 Control Latency

The maximum acceptable latency from applying a change to the input of the haptic device by the operator, to the drone actually moving at which a stable flight can be maintained, depends on various factors such as drone speed, desired flight distance, obstacles in the environment et cetera. Therefore, requiring a nominal maximum delay is not trivial. [1] reviewed research on maximum round-trip time (RTT) delays in teleoperated systems. The maximum RTT delay for different control strategies ranged from 400 ms up to one second. For this thesis, the latency, which is the same as the delay in context of the thesis, is required to be lower or equal than

$$T_{max} \approx \frac{400\,\mathrm{ms}}{2} = 200\,\mathrm{ms}$$

to have an approximation of the maximum latency. The value is divided by two as we do not define the RTT but rather unidirectional latency. The reason for that is that the proposed implementation will only communicate unidirectionally, which will not allow the measurement of the RTT.

Please note that the requirement is only applied to the latency introduced by the software itself, as if all components run on a single machine communicating over the loopback interface. Latencies that would be introduced in real-world networks by physical and data link layers are ignored for now. If the proposed implementation shall be connected through a real network, satisfaction of this requirement needs to be evaluated for said network.

4.1.3 Haptics Frequency

The 3D Systems Touch™ haptic device specifies that the device driver requires that haptic device forces are updated at a rate of

$$f_{haptics} = 1000\,\mathrm{Hz}$$

in [8] which therefore requires that the implementation is able to meet the given frequency.

Such a high refresh rate is needed so that the servo motors of the device maintain smooth motions when the haptic tool accelerates or decelerates.

Operation might be possible with a lower frequency down to 500 Hz, as this seems to be the minimum frequency necessary to adequately reproduce all haptic sensations that humans can feel [29]. Nonetheless it makes sense to use the hardware at its fullest potential, thus 1000 Hz is the goal.

4.1.4 Sensitive Data Rate

The required data rate of all communication should be low enough that communication can be done on WLANs such as WiFi and Bluetooth, so that the drone can be controlled at least over a few meters without wired connection. Also, lots of commercially available drones already have a WiFi chip on board. It would be beneficial if WWANs such as 5G, WiMAX, 4G/LTE or even older inferior standards such as 3G/UMTS and HSPA+ can be used too, which would enable operators to control drones over large distances of several kilometers, as long as cell service is provided in the respective area. In Table 4.1 one can see the maximum data rates that can theoretically be reached within WLANs and WWANs

Type	Technology	Maximum Data Rate	Range
WLAN	Bluetooth	$2\,\mathrm{Mbit\,s^{-1}}$	$50\,\mathrm{m}$
	WiFi	$54\,\mathrm{Mbit\,s^{-1}}$	$100\,\mathrm{m}$
WWAN	3G/UMTS	$1.92\,\mathrm{Mbit\,s^{-1}}$	$10\,\mathrm{m} - 150\,\mathrm{km}$
	HSPA+	$84\,\mathrm{Mbit\,s^{-1}}$	$10\,\mathrm{m} - 150\,\mathrm{km}$
	4G/LTE	$300\,\mathrm{Mbit\,s^{-1}}$	$10\,\mathrm{m} - 150\,\mathrm{km}$
	5G	$20\,\mathrm{Gbit\,s^{-1}}$	$10\,\mathrm{m} - 150\,\mathrm{km}$
	WiMAX	$70\,\mathrm{Mbit\,s^{-1}}$	$50\,\mathrm{km}$

Table 4.1: Characteristics of wireless technologies for spanning WLANs [30], [31] and WWANs [32], [33]. The ranges of the WWANs are given as an estimation of the range of a single cell tower. By using a network of cell towers, arbitrary long distances can be realized.

based on different technologies. Strictly speaking, those data rates will not necessarily be reached in real life, which needs to be taken into account on evaluation.

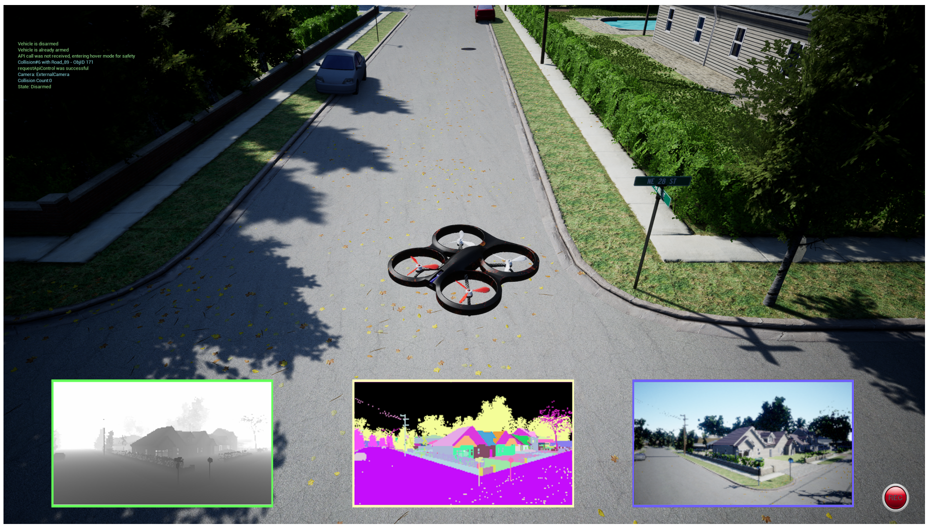
Generally, the lower the required data rate is, the better. When networks are too crowded and communication media is over-utilized, networks may not be able to even provide typical data rates to every station. That's why the implementation should try to actively keep bandwidth requirements low by means of compression.

4.1.5 Extendability and Modularity

APIs of flight controllers and the flight controllers itself, so the things needed to remotely control the drone with own applications, are often proprietary and as a consequence very different from each other. That would require that for each drone, a control application would need to be re-implemented or at least heavily changed for each flight controller or at least for any drone manufacturer.

Also, this thesis initially was planned to work with a real hardware quadcopter from the beginning, but due to COVID 19 regulations that suggested working at home, safety concerns on campus and also to lower development efforts, it was decided that a simulator, more specifically the one described in Section 4.2.1, shall be used. However there might be plans in the future to test the implementation on a real drone.

The beforementioned facts lead to the explicit requirement of developing a system that is extendable so that it's possible to introduce new drone hardware, preferably with as minimal effort as possible. In order to keep the software extendable, an implicit requirement is modularity so parts of the implementation that are specific to hardware or its software are interchangeable without the need to modify the rest of the software.



```
Vehicle is disarmed
Vehicle is already armed
API call was not received, entering hover mode for safety
Collision#6 with Road_89 - ObjID 171
requestApiControl was successful
Camera: ExternalCamera
Collision Count:0
State: Disarmed
```

Figure 4.2: Screenshot of AirSim simulating a quadrotor drone in an urban environment with all available camera views enabled.

4.2 Used Software

4.2.1 Drone Simulation

Instead of using a real drone for testing and evaluation, a simulator is used to lower the effort required to implement the initial iteration of the demonstrator by cutting down any hardware-related specifics. A simulator also removes the risks induced to the well-being of humans and creatures, the integrity of objects and the hardware itself in the fly space by running untested control code.

In this thesis, *AirSim* will be used for physics simulation of the drone and the three-dimensional environment around it. AirSim is an open-source platform that allows to physically and visually simulate autonomous vehicles [34]. It is built on the *Unreal Engine*, a state-of-the-art real-time 3D simulation engine that is primarily used for developing video games, which allows AirSim to run physics simulations of complex scenery and render visually realistic images in real time. That's why AirSim is predestined for testing and evaluating algorithms that are supposed to control autonomous vehicles in the real world, and even more so to be utilized in machine learning and deep learning disciplines, e.g. for generation of training data or for evaluation of artificial intelligence. Details about the vehicle and environment models can be found in [34].

Currently AirSim supports quadrotor drones and four-wheeled cars out of the box, as well as a variety of sensory components that can be configured to be attached to the vehicles, like Inertial Measurement Units (IMUs), magnetometers, GPS receivers, barometers and even LIDAR sensors. Furthermore, it optionally renders several camera views in addition to the normal third-person camera, such as a depth map, a segmentation and a first person view, as can be seen in Figure 4.2. Due to its open-source nature, many 3D environments are provided by the project's community to choose from depending on the specific application, each offering individual characteristics.

Other physics simulators exist, like general robot simulators, for example Gazebo [35] or CoppeliaSim [36]. However, for this thesis, AirSim is the perfect match due to its specialization on quadrotor drones, even though only a small subset of its features will be used. AirSim implemented a whole flight controller which is essentially a set of Proportional-Integral-Derivative (PID) controllers configured for controlling the four motors of a quadcopter to accomplish high-level goals such as flying to a specific position or maintaining a specific acceleration vector. In more general robotics simulators, this functionality would be needed to be implemented first in order to allow for evaluation of controllability in real-world scenarios, but AirSim takes away that step – no need to reinvent the wheel.

```cpp
class MultirotorRpcLibClient : public RpcLibClientBase {
  using Client = MultirotorRpcLibClient;

  Client* takeoffAsync(float timeout_sec = 20, [...]);
  Client* landAsync(float timeout_sec = 60, [...]);
  Client* moveByRollPitchYawThrottleAsync(float roll, float pitch, float yaw,
      float throttle, float duration, [...]);
  Client* moveByVelocityAsync(float vx, float vy, float vz, float duration, [...
      ]);
  Client* moveToPositionAsync(float x, float y, float z, float s, [...]);
  [...]

  LidarData getLidarData([...]) const;
  ImuBase::Output getImuData([...]) const;
  BarometerBase::Output getBarometerData([...]) const;
  [...]
};
```

Listing 4.1: Subset of function headers provided by AirSim that can be called by RPC, in shortened C++.

To tell the drone what to do, interfacing with AirSim through Remote Procedure Call (RPC) is necessary. The RPC interface allows interfacing with almost every programming language. AirSim provides straight-forward functions to initiate drone movements and fetch sensory data. A small subset of those functions can be found in Listing 4.1.

Beforementioned sensory components that are attached to the drone in the simulation and several other configuration options effecting the rendering and physics of the simulator can be set by a JavaScript Object Notation (JSON) configuration file. The configuration file `settings.json` is searched for in the folder of the simulator's executable and in `<user>/Documents/Airsim`, but can also be specified explicitly by providing an absolute path to the configuration file:

```
./<AirSim executable> --settings 'C:\path\to\settings.json'

# example for 'Blocks' environment
./Blocks.exe --settings 'C:\path\to\settings.json'
```

Appendix A.2 shows the contents of the configuration file with default values. A subset of the options that can be configured are listed in Table 4.2 with short explanations. For

Option	Explanation
SimMode	Simulation mode ('Multirotor', 'Car', ...)
ViewMode	Camera mode ('GroundObserver', 'Fpv', 'Manual', ...)
Vehicles.<name>.	
VehicleType	Vehicle type ('PhysXCar', 'SimpleFlight')
Sensors.<name>.	
SensorType	Sensor type (6 for LIDAR)
Sensors.<lidarName>.	
NumberOfChannels	Number of channels (individual lasers)
RotationsPerSecond	Rotations per second
HorizontalFOVStart	Horizontal field of view range start (in degrees, -180° to 180°)
HorizontalFOVEnd	Horizontal field of view range end (in degrees, -180° to 180°)
VerticalFOVStart	Vertical field of view range start (in degrees, -90° to 90°)
VerticalFOVEnd	Vertical field of view range end (in degrees, -90° to 90°)
{X, Y, Z}	Position relative to vehicle (NED coordinates, in meters)
{Yaw, Pitch, Roll}	Orientation relative to vehicle (in degrees)

Table 4.2: Small subset of AirSim's configuration options with short explanations.

detailed and complete information, AirSim's documentation has several dedicated pages[1].

The code repository of the thesis[2] contains various settings files that are used for experiments in this thesis. Most of them configure a LIDAR sensor with different scanning parameters, which can be identified by the corresponding file name:

```
drone-controller-airsim\settings\without-lidar.json
drone-controller-airsim\settings\with-lidar-90hfov-90vfov-50ch-10rot.json
drone-controller-airsim\settings\with-lidar-90hfov-90vfov-100ch-12rot.json
drone-controller-airsim\settings\with-lidar-180hfov-180vfov-50ch-10rot.json
...
```

4.2.2 Haptic Device Framework

CHAI3D is a cross-platform open-source C++ simulation framework that focuses on supporting commercially available haptic devices, like the one used for this thesis as introduced in Section 2.2, as input and output peripherals [37].

The library consists of several modules, each containing various classes that can be utilized for own applications. A subset of those modules is depicted in Figure 4.3.

One of the main advantages of CHAI3D is that it takes care of downloading and setting up device drivers for the haptic device in a platform-agnostic manner, and gives developers an

[1] `https://microsoft.github.io/AirSim/settings/`, `https://microsoft.github.io/AirSim/sensors/`, `https://microsoft.github.io/AirSim/lidar/`

[2] `https://code.ovgu.de/comsys-group/haptic-drone-control`

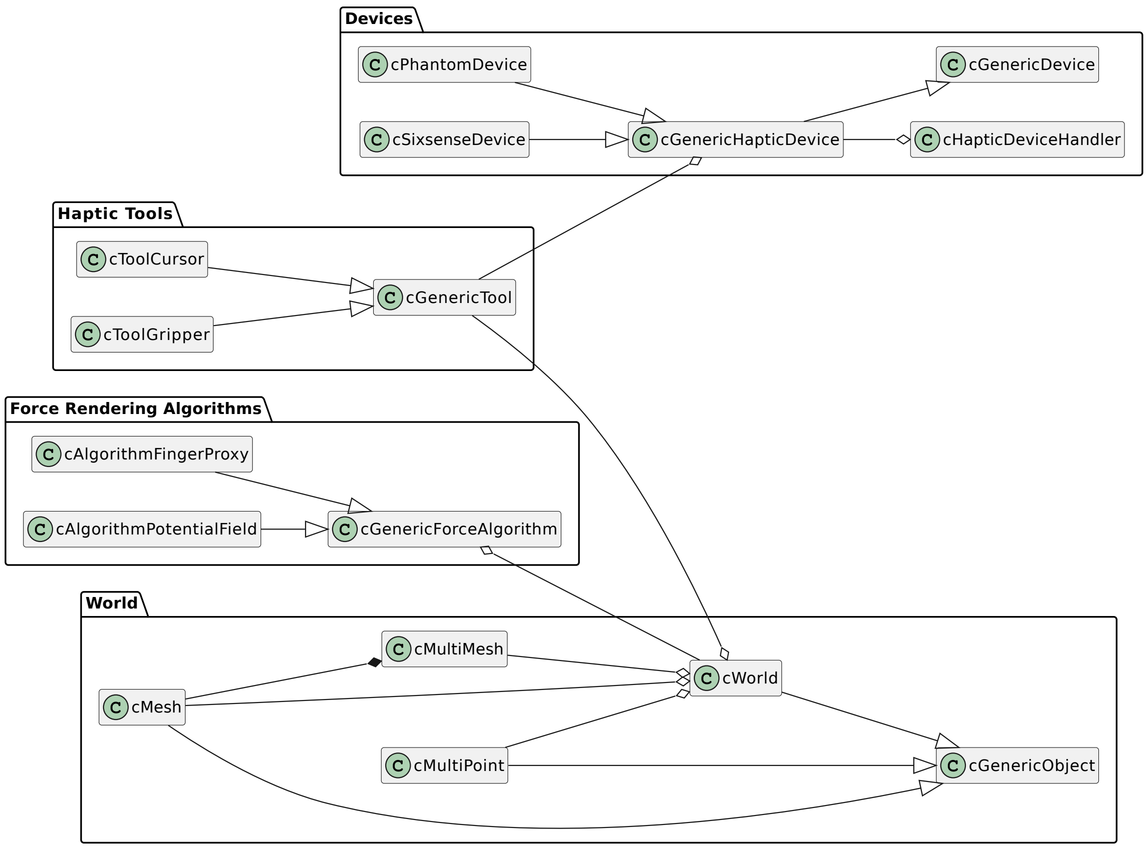


Figure 4.3: UML class diagram of a subset of CHAI3D modules and some of their classes.

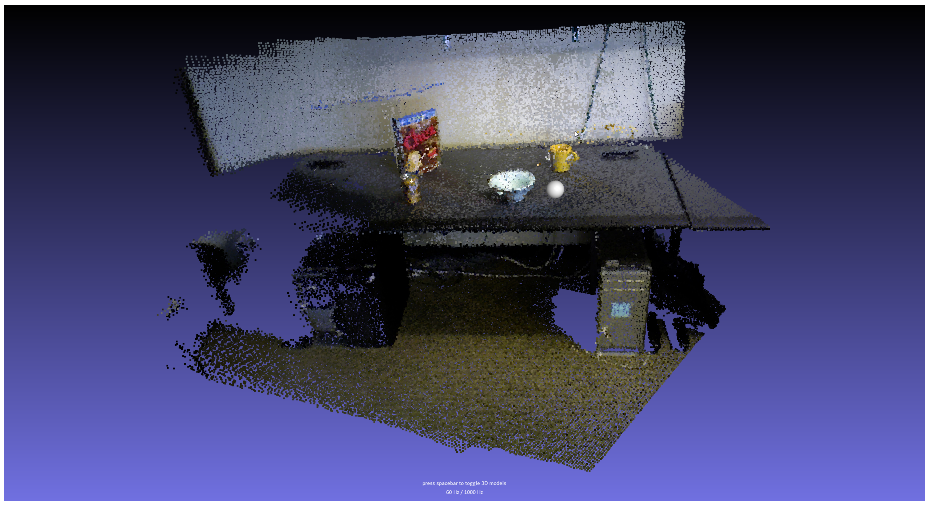


Figure 4.4: Screenshot of a complex scenery (a big point cloud) that is rendered haptically and visually by CHAI3D. The gray ball is the tip of the haptic tool of the haptic device.

abstraction of device interfaces that are different for each manufacturer and appear to be proprietary in most cases.

Thanks to CHAI3D, communication with the haptic device is as easy as calling methods such as `getPosition`, `getRotation` and `getGripperAngleDeg`, which are members of the `cGenericHapticDevice` class, to retrieve the state of the haptic tool. There are also members for applying force feedback to the haptic tool manually with e.g. `setForce`. A typical loop that updates the haptic force feedback through the device driver looks like the pseudocode that can be found in Appendix A.1.

However, to render more complex scenery haptically, such as the one in Figure 4.4, the `cWorld` class exists that represents a virtual world consisting of 3D objects, which can be used in conjunction with collision detection and an implementation of the finger-proxy algorithm proposed in [38] to apply force feedback accordingly. So, when using `cWorld`, the library is actually able to 'abstract away' any mathematical considerations, all force calculations are done internally. Typically the loop that communicates with the device driver changes to the second pseudocode in Appendix A.1, if you choose to use the abstract way with `cWorld` and the finger-proxy algorithm.

Another advantage is that, even though this thesis specializes its design concepts and implementation on a single haptic device, using CHAI3D in the implementation makes it fairly straight-forward to adjust for haptic devices other than the 3D Systems Touch™.

Although CHAI3D also provides modules to implement 3D physics simulations and real-time visualization, it is not utilized for that, because AirSim brings the implementation of a flight controller and focuses on quadrotor drones, as argued in Section 4.2.1. Instead, in the implementation proposed in this thesis, CHAI3D will be used primarily to interface with the haptic device, to apply the implementation of the finger-proxy algorithm and for collision detection of the haptic tool with objects in a virtual world.

4.2.3 Data and Communication

Control data needs to be communicated between various components at some point or another. That's why some kind of protocol for data (de-)serialization and structuring as well as transmission is needed.

Data Serialization and Deserialization Protocol

There are a lot of popular data formats such as JSON, CBOR and XML. All of them are used both for configuration files and actual structured data payloads, even though they introduce lots of overhead in terms of syntactical symbols and flexible layouts (e.g. whole keys are stored for key-value pairs) in exchange for being at least somewhat readable by humans. While that works just fine, optimization is not done to its fullest potential and bandwidth is wasted just to send metadata that is most of the time known by both communication parties already, anyway. That is unfortunate on large scale networks with ever rising node numbers and/or constrained means of communication. Also, bandwidth is theoretically not the only wasted resource. At the end of the day, parsing and generating complex human-readable formats needs more computational power and memory than binary formats or even raw C structs, and the compiled source code needed is also larger [39]. Even though most drones have onboard computers that are not constrained enough that computation, memory or storage conflict with using heavier data formats, it does not hurt to be more efficient in these aspects, too, to save money (less memory chips) and energy (more flight time with same battery capacity).

Using raw C structs is a possibility, but is generally dangerous if one expects to run the software on different platforms due to different endianness and padding requirements. Flight controllers of quadrotor drones often run on ARM-based processors, especially open-source flight controllers such as PX4 [40], while other software components will be developed and may run on x86-based systems. Additionally, language incompatibility is lowered by using C structs or at least more complicated than a well-defined data format with own primitive types, but you might want to implement certain parts of the implementation in other languages than C or C++. Being compatible over language barriers also goes together very well with the extendability requirement set in Section 4.1.5.

The FlatBuffers library tries to solve the beforementioned problems altogether. It is an open-source cross platform serialization library for C++, Go, Java, Python and lots of other languages [41]. Originally created at Google for game development and other performance-critical applications, the library aims to be efficient in terms of computation, speed, memory and code footprint. It does that by avoiding the necessity of allocations other than the binary buffer of the data you want to store, while trying to still be flexible e.g. by supporting optional fields.

To accomplish being usable across different languages, and also to provide the means to develop a clear and structured data model, the FlatBuffers library defines a schema language called Interface Definition Language (IDL) which has a syntax similar to C. IDL can be used to write data schemes in an easy way as a composition of tables (similar to objects), structs (special type of table), arrays, enums, unions and some more [42]. Tables and structs contain fields that can be of the types mentioned above and primitives. Listing 4.2

shows an exemplary schema file that makes use of the most important syntactical elements. Once schema files are present, they can be converted to actual source code for each of the supported languages, with a tool called `flatc` that the library brings with it [43]. Said generated source code can then be used in own application code, including well-defined classes with smart memory management and implementations of (de-)serialization. For C++ which is the language of choice for the thesis implementation, header-only libraries are generated.

```
enum Degree : byte { None, Bachelor, Master }
struct Room { building:short; num:uint; }
table Student { name:string; num:uint; degree:Degree = Bachelor; }
table University { students:[Student]; rooms:[Room]; }
root_type University;
```

Listing 4.2: FlatBuffers schema file that models data of a university and its students as an example.

Including the generated header files together with the FlatBuffers library header gives a software developer two ways of interfacing with buffers and data models respectively: the base API that works directly on the allocated memory, avoiding copies where it can, and an object-based API that can be generated optionally with a compiler flag. The latter provides a more convenient and object-oriented way of using FlatBuffers at the expense of being less memory efficient and more computationally expensive due to packing and unpacking of the values into C++ objects and containers. Both ways are demonstrated shortly in Listing 4.3 and Listing 4.4 by showing the different codes that are used to fill a buffer that is of type `University` (the one that is defined in Listing 4.2). Due to the simplicity of the example, the convenience impact is not really apparent, but for complex models the object-oriented API definitely makes the programmer's life easier. Reading a buffer is similar in both ways, even though mutating values is generally easier when using the object-oriented API.

```
flatbuffers::FlatBufferBuilder builder(1024);

auto name = builder.CreateString("Jon-Mailes Graeffe");
auto degree = Degree_None; // hopefully Degree_Bachelor soon :)
auto studentOne = CreateStudent(builder, name, 219717, degree);
std::vector<flatbuffers::Offset<Student>> studentsVector;
studentsVector.push_back(studentOne);
auto students = builder.CreateVector(studentsVector);

Room rooms_array[] = { Room(29, 333) };
auto rooms = builder.CreateVectorOfStructs(rooms_array, 1);

auto university = CreateUniversity(builder, students, rooms);

builder.Finish(university);
uint8_t* buffer = builder.GetBufferPointer();
size_t size = builder.GetSize();
```

Listing 4.3: Exemplary code on how to use the generated source code of the schema defined in Listing 4.2 to serialize an instance of the `University` type while using the base API (the more efficient one).

```cpp
auto student = new StudentT();
student->name = "Jon-Mailes Graeffe";
student->num = 219717;
student->degree = Degree_None;

auto university = new UniversityT();
university->students.push_back(std::unique_ptr<StudentT>(student));
university->rooms.push_back(Room(29, 333));

flatbuffers::FlatBufferBuilder builder(1024);
builder.Finish(University::Pack(builder, university));
uint8_t* buffer = builder.GetBufferPointer();
size_t size = builder.GetSize();
```

Listing 4.4: Exemplary code on how to use the generated source code of the schema defined in Listing 4.2 to serialize an instance of the `University` type while using the object-based API instead of the base API (the more convenient one from a developer's perspective)

Communication Protocol

For the CoAP implementation, *libcoap* is used which is an open-source C library that aims to be multi-platform, meaning it can be used on Portable Operating System Interface (POSIX) operating systems such as Linux and Windows as well as on operating systems for embedded devices like Contiki, TinyOS, RIOT OS and more [44]. It is the perfect match due to its maturity and lightweightness, with the latter theoretically enabling it to be run on drone flight controllers directly. Both server and client functionality is implemented, and Datagram Transport Layer Security (DTLS) is supported, so adding encryption to the thesis implementation is possible with reasonable effort.

Example implementations of both a CoAP server and client are provided in Appendix A.3 in case the reader is interested. Later in the thesis it will be apparent that the thesis implementation contains an abstraction, which does not necessarily require knowledge of how to implement libcoap even when developing extensions for the implementation, though.

4.2.4 Other Dependencies

In addition to the dependencies listed above, there are mostly two libraries that are used in the thesis implementation.

First in the list is the *Point Cloud Library (PCL)* which is a large open-source framework providing numerous C++ libraries for 2D/3D image and point cloud processing [45]. It is used for the data models it brings for point clouds and 3D meshes, as well as the various implementations that transform point clouds to 3D meshes.

Secondly there is the *wavelib* library which is a small open-source C implementation of various wavelet-based transformations such as FWT [46]. It does not matter too much right now what this is exactly, but it is used for compression of messages in the thesis implementation which is explained in more detail in Section 4.4.1.

4.3 Concept

A connection needs to be established between the haptic device and the drone from an abstract perspective. The 3D Systems Touch™ is connected via USB and its drivers are available for Windows and Linux. So there must be at least one software component, that runs on Windows or Linux, and that the haptic device can be plugged into (wired connection). This software component does need to run a loop that updates force feedback of the haptic device and possibly reads positions and orientations from the device as well, in a frequency of $f_{haptics}$ as explained in Section 4.1.3.

Even though a simulator is used to test the implementation in its first iteration, one needs to think about how interfacing with a real drone could work in the future. Interfacing with a drone to control it wirelessly is different from drone to drone, but is usually done over a 2.4 GHz radio, more specifically over WiFi (IEEE 802.11). Typically, either the drone manufacturer or the drone's flight controller provides software to run on the drone to control it and retrieve sensor data over WiFi directly. For example, drones that run the open-source flight controller PX4 [40] have a MAVLink interface that can be used to control the drone [47], while others may have proprietary ones. Or the drone does not have any interface to control the drone yet, in which case it is needed to add another software component that is able to run on the drone itself.

Because of the possibility that a software component needs to be developed to run on a drone's onboard computer in the future, and also due to the requirement that it should be easy to make the whole thing work on real hardware drones, it makes sense to add a second software component that might or might not run on a real drone in the future. The second component is just used to interface with the drone without any extensive calculations or I/O, which eases adapting to drone hardware because one only needs to implement a small software component that just bridges the gap between the first software component and the drone in use.

So, the implementation shall have two major software components: the *haptics-brain* component which interfaces with the haptic device and does most of the calculations and computationally intensive tasks, thus called the brain of the operation, and the *drone-controller* component. The *drone-controller* component translates commands given by the haptics-brain component so that coordinates or nominal values are converted to fit the drone interface, and sends them to the drone. Because the drone-controller component is specific to AirSim, let's call it *drone-controller-airsim* so that if there will be more implementations, one can distinguish between them.

On a side note, it may be interesting to know that AirSim actually has a MAVLink client that can relay commands sent to AirSim to real hardware MAVLink-compatible drones [48]. In other words, at least for proof-of-concept evaluations, it is possible to run AirSim as a bridge to real drones while utilizing the drone-controller-airsim component without the need to implement another drone-controller component.

Now that both software components have been established: how is the human operator supposed to control the drone? Or in other words, how does the haptics-brain component interpret the input of the 3D Systems Touch™ haptic device and translate it to commands that control the drone? Well, for that, several control modes are invented that let the operator control the drone in different ways. It is not known yet what's the best way on how to use a haptic device as an input device for drone control, and the control modes are evaluated later in the thesis to get an idea about the user experience and advantages and/or disadvantages of the control modes.

The components itself will be revisited in Section 4.4 later on, and the actual structure of the software will be made more clear by various diagrams. Before that though, the control modes will be introduced to get an understanding of the general idea on how to control the drone with the haptic device in Section 4.3.1.

4.3.1 Control Modes

The implementations of the control modes executed in the haptics-brain component are responsible to take the input of the haptic device, mainly the position and orientation of the haptic tool, and generate commands that are then sent to the drone-controller component and are formatted in a common format that all implementations can work with. When the commands arrive on the drone-controller component, all values are converted to sensical values – coordinates in the coordinate system of the haptic device are mapped onto the drone's coordinate system, and other linear values such as thrust are normalized by a scalar so that they are subjectively the same across different drones. For example, it shall not happen that a change of a centimeter of the input to the haptic device lets one drone crash into a wall a few hundred meters away and lets another drone barely move at all.

Said conversions can mostly be expressed mathematically, trivially by multiplication with constant scalars or vectors that are specific to a drone-controller implementation. Let's define the names of those that are used for every control mode now, in Table 4.3 which also contains some short descriptions on what they are used for and the values that are used in the implementation of drone-controller-airsim. The specific values are the result of experimentation and trying out values that seem to provide a good user experience, which obviously are very subjective and can vary greatly depending on the use case, flying environment, haptic device, drone model et cetera. Some of the scalar and vector multiplications used in this thesis and its implementation could be aggregated mathematically by using transformation matrices that contain translations and rotations, but since this is not done in the current implementation and for the sake of simplicity, this is avoided for now.

All control modes implement the two buttons of the 3D Systems Touch™ haptic device. Pressing the first button for a bit arms or disarms the drone respectively, meaning turning it on and off. The second button's long press toggles pausing all inputs to be commanded to the drone, so operators can remove their hands from the haptic device safely.

Currently, four control modes exist: the *Manual Flight* control mode, the *Velocity Joystick* control mode, the *Target Positioning* control mode and, last but not least, an extension of the former, the *Terrain-Aware Target Positioning* control mode. All control modes will be explained in detail below.

Name	Symbol	Value (AirSim)	Usage
Axis Correction	$\overrightarrow{AC}$	$\begin{pmatrix}-1 & 1 & -1\end{pmatrix}^T$	Adjust directions/sign of vector components (axis alignment)
Velocity Scaling	$\overrightarrow{VS}$	$\begin{pmatrix}1 & 1 & 1\end{pmatrix}^T$	Velocity normalization (across different drones)
Position Scaling	$\overrightarrow{PS}$	$\begin{pmatrix}30 & 30 & 30\end{pmatrix}^T$	Distance normalization (across different drones)
Yawing Rate Scaling	YS	1.0	Yawing rate normalization (across different drones)
Haptic Force Multiplier	$\overrightarrow{HM}$	$\begin{pmatrix}-50 & -50 & -50\end{pmatrix}^T$	Multiplier on manual haptic force applications (stiffness, springiness)

Table 4.3: Constant scalars and vectors used for corrections and coordinate mappings in most control modes, with their names used in the implementations, symbols used in mathematics and a brief description for what the value is used.

Manual Flight

The first control mode is really basic as it just takes the pitch and roll of the haptic tool (stylus) and applies them to the drone in the control frequency $f_{control}$ that is specified in Section 4.1.1. Controlling the yaw is a bit different to make flying easier. The problem is, if the operator would also control the yaw with the haptic tool directly, making a 360° turn in terms of yaw would be basically impossible – at least with the used haptic device. The 3D Systems Touch™ does not allow the stylus to be rotated a full 360° in yaw mechanically, and due to the fact that the haptic tool is a stylus that has a specific length in one direction as can be seen in Figure 2.3, it is not possible to apply full or even more than half yaw at every possible position. Because of that, the yaw of the haptic tool translates to a yawing rate once outside of a deadzone, which basically means that before a minimum yaw of the haptic tool, the yawing rate is 0 to make going straight ahead easier. So, the operator can command the drone to do 360° turns in yaw even with less than half of the full yaw applied to the haptic tool, and the higher the absolute yaw value is, the faster the drone will yaw per second.

Throttle is determined by the distance of the haptic tool position to zero on the z axis (vertical) of the haptic device, so speed is very manual in a way that the operator needs to find out the minimum throttle at which the drone becomes airborne, but the operator also should not give maximum throttle as it could lead the drone to uncontrollable speeds in an instant. It bears similarity with driving a car with manual transmission where one needs to balance the clutch pedal to reach a biting point, so basically letting go the minimum distance of the pedal to start rolling, but one must not let go of the clutch pedal too quickly because the car might stall or go forward very abruptly.

Mathematically, all of the above can be expressed as a function ds_{MF} (desired state function for Manual Flight control mode) that takes the position vector and rotation matrix $\boldsymbol{R}_h$ of the haptic tool as the input and that outputs the desired drone orientation in pitch and roll angles as well as a yawing rate yr_d and a throttle scalar $throttle_d$:

$$ds_{\mathrm{MF}} : \mathbb{R}^3 \times \mathbb{R}^{3\times 3} \to \mathbb{R}^4, ds_{\mathrm{MF}} \left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix}, \boldsymbol{R}_h \right) := \begin{pmatrix} \overrightarrow{AC}_x * YS * yr(\boldsymbol{R}_h) \\ \overrightarrow{AC}_y * p(\boldsymbol{R}_h) \\ \overrightarrow{AC}_z * r(\boldsymbol{R}_h) \\ \min\left(\max\left(\frac{z_h}{MT}, 0\right), 1\right) \end{pmatrix} = \begin{pmatrix} yawRate_d \\ roll_d \\ pitch_d \\ throttle_d \end{pmatrix}$$

$$yr : \mathbb{R}^{3\times 3} \to \mathbb{R}, yr(\boldsymbol{R}_h) = \begin{cases} y(\boldsymbol{R}_h) + Y_{min} & \text{if } y(\boldsymbol{R}_h) \leq -Y_{min} \\ y(\boldsymbol{R}_h) - Y_{min} & \text{if } y(\boldsymbol{R}_h) \geq Y_{min} \\ 0 & \text{else} \end{cases}$$

$$y : \mathbb{R}^{3\times 3} \to \mathbb{R}, y(\boldsymbol{R}_h) = \arctan2(\boldsymbol{R}_{h_{21}}, \boldsymbol{R}_{h_{11}})$$

$$p : \mathbb{R}^{3\times 3} \to \mathbb{R}, p(\boldsymbol{R}_h) = \arctan2\left(-\boldsymbol{R}_{h_{31}}, \sqrt{\boldsymbol{R}_{h_{32}}^2 + \boldsymbol{R}_{h_{33}}^2}\right)$$

$$r : \mathbb{R}^{3\times 3} \to \mathbb{R}, r(\boldsymbol{R}_h) = \arctan2(\boldsymbol{R}_{h_{32}}, \boldsymbol{R}_{h_{33}})$$

In the yr function, Y_{min} defines the minimum yaw of the haptic tool that is required in both directions to apply any yawing to the drone. When the yaw of the haptic tool reaches Y_{min}, yaw is applied depending on the yaw value. This constant therefore defines the beforementioned deadzone. MT is the maximum throttle on the haptic device, meaning the maximum value the haptic tool can be positioned on the z axis, so that $throttle_d$ is always a number between zero and one. Extracting the yaw, roll and pitch angles from the rotation matrix that is given by the haptic device is done with the y, p and r functions according to [49].

Haptic feedback is minimal on this control mode. The haptic device forces the tool to always stay on the vertical z axis, which makes it easier for the operator to adjust the tool's angles. Also, negative z positions are forced to be positive because in the current configuration it is not possible to give negative throttle. The force vector applied to the haptic tool and subsequently to the operator's hand can also be written as a function ff_{MF} (force feedback function for Manual Flight control mode), more specifically a function of the position the haptic tool points to:

$$ff_{\mathrm{MF}} : \mathbb{R}^3 \to \mathbb{R}^3, ff_{\mathrm{MF}} \left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix} \right) := \begin{cases} \overrightarrow{HM} \odot \begin{pmatrix} x_h & y_h & z_h \end{pmatrix}^T & \text{if } z_h \leq 0 \\ \begin{pmatrix} \overrightarrow{HM} * x_h & \overrightarrow{HM}_y * y_h & z_h \end{pmatrix}^T & \text{else} \end{cases}$$

$\odot$: element-wise multiplication

As the name of the control mode implies, it is a very manual way of controlling the drone and is similar to direct control in teleoperation. Strictly speaking, direct control is not entirely the correct term though, because there are still PID controllers that are running on the drone to achieve a specific yaw, pitch, roll and throttle by taking inputs of a gyroscope and adjusting the four motors accordingly. It would be direct control entirely if and only if the operator would control the speeds of the four motors directly, which sounds very difficult to do from the perspective of the operator.

Velocity Joystick

Secondly there is the Velocity Joystick control mode which is named according to how it works. It is like a joystick on a game controller, but three-dimensional: the operator steers the haptic tool (stylus) in the direction the drone is supposed to fly, setting the desired velocity the drone is going to and how fast it is. The higher the distance from haptic tool position to zero point on the haptic device, the faster the drone will fly, because the vector will be greater in length.

Mathematically, the vector from zero point of the haptic device to the position the haptic tool points to is just scaled up or down in terms of length, and can be directly commanded to the drone in the control frequency $f_{control}$ that is specified in Section 4.1.1:

$$ds_{\text{VJ}} : \mathbb{R}^3 \times \mathbb{R}^{3\times 3} \to \mathbb{R}^4, ds_{\text{VJ}}\left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix}, \boldsymbol{R}_h\right) := \begin{pmatrix} \overrightarrow{AC}_x * \overrightarrow{VS}_x * x_h \\ \overrightarrow{AC}_y * \overrightarrow{VS}_y * y_h \\ \overrightarrow{AC}_z * \overrightarrow{VS}_z * z_h \\ YS * yr(\boldsymbol{R}_h) \end{pmatrix} = \begin{pmatrix} v_{d_x} \\ v_{d_y} \\ v_{d_y} \\ yr_d \end{pmatrix}$$

$$yr : \mathbb{R}^{3\times 3} \to \mathbb{R}, yr(\boldsymbol{R}_h) = \begin{cases} y(\boldsymbol{R}_h) + Y_{min} & \text{if } y(\boldsymbol{R}_h) \leq -Y_{min} \\ y(\boldsymbol{R}_h) - Y_{min} & \text{if } y(\boldsymbol{R}_h) \geq Y_{min} \\ 0 & \text{else} \end{cases}$$

$$y : \mathbb{R}^{3\times 3} \to \mathbb{R}, y(\boldsymbol{R}_h) = \operatorname{arctan2}(\boldsymbol{R}_{h_{21}}, \boldsymbol{R}_{h_{11}})$$

Feedback in terms of haptics is given in the way that steering into a specific direction becomes harder for the operator, or in other words the counterforce gets higher in value, the longer the distance of the haptic tool is to the zero point. This creates an effect almost like with gas pedals in cars – the operator can judge more easily how fast the drone is commanded to fly without any need for a tachometer and more force is needed to fly faster, which gives an intuitive feeling about the desired velocity. The calculation of the force applied to the haptic device is trivial, as it's just a multiplication of a constant with the current position of the haptic tool:

$$ff_{\text{VJ}} : \mathbb{R}^3 \to \mathbb{R}^3, ff_{\text{VJ}}\left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix}\right) := \overrightarrow{HM} \odot \begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix} \qquad \odot\text{: element-wise multiplication}$$

Additionally to the button events that are present on all control modes, pressing the first button shortly increases the velocity scaling multiplier $\overrightarrow{VS}$, while the second button decreases it.

Compared to the Manual Flight control mode, this mode is less direct control in teleoperation terminology as it is not directly tied to a subset of the aircraft controls, and more supervisory control, because the operator defines a velocity goal the drone is trying to achieve autonomously.

Target Positioning

As a third option, there is the Target Positioning control mode that fundamentally differs from the modes before. When using the Manual Flight or the Velocity Joystick control mode, changing the input to the haptic device results in a change of the behaviour of the drone almost immediately, $f_{control}$ times a second. This is not the case with the Target Positioning control mode, as the operator can move the haptic tool freely without influencing the drone. Once the desired position has been found and is fine tuned, the operator presses the first button on the haptic tool for a short time which commands the drone to autonomously fly to the desired position. Usually operators then wait for the drone to reach the target position, and command to move again once a position is hold stably. The advantage of this Move-and-Wait strategy that was shortly introduced in Section 2.1 is that, regardless of any delay or latency that occurs between operator and teleoperator, movement of the drone will already be done before the next command is given, and new commands do not depend on the immediate state of the drone which might not be known to the operator yet due to the time delays. While the operator chooses the desired position, the drone is hovering, meaning it tries to not move in any direction, maintaining a steady height and stabilizing the aircraft on its own.

To generate a command on request, the position of the haptic tool in the haptic device coordinate system is simply mapped to the local coordinate system around the drone which, depending on the flight controller, can be chosen to be arbitrarily large:

$$ds_{\text{TP}} : \mathbb{R}^3 \times \mathbb{R}^{3\times 3} \to \mathbb{R}^4, ds_{\text{TP}} \left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix}, \boldsymbol{R}_h \right) := \begin{pmatrix} \overrightarrow{AC}_x * \overrightarrow{PS}_x * x_h \\ \overrightarrow{AC}_y * \overrightarrow{PS}_y * y_h \\ \overrightarrow{AC}_z * \overrightarrow{PS}_z * z_h \\ YS * yr(\boldsymbol{R}_h) \end{pmatrix} = \begin{pmatrix} x_d \\ y_d \\ z_d \\ yr_d \end{pmatrix}$$

$$yr : \mathbb{R}^{3\times 3} \to \mathbb{R}, yr(\boldsymbol{R}_h) = \begin{cases} y(\boldsymbol{R}_h) + Y_{min} & \text{if } y(\boldsymbol{R}_h) \le -Y_{min} \\ y(\boldsymbol{R}_h) - Y_{min} & \text{if } y(\boldsymbol{R}_h) \ge Y_{min} \\ 0 & \text{else} \end{cases}$$

$$y : \mathbb{R}^{3\times 3} \to \mathbb{R}, y(\boldsymbol{R}_h) = \text{arctan2}(\boldsymbol{R}_{h_{21}}, \boldsymbol{R}_{h_{11}})$$

As simple as this control mode is, one challenge for the drone operator definitely is to guess to which position in the drone's environment the choice of position in the workspace of the haptic device is mapped to. That's why it makes sense to offer some kind of visual assistance on where the drone is going to fly to. In teleoperation this is called a *predictor display*, as it displays a prediction of the teleoperator's state if the command would be applied. For the thesis implementation, this is trivial because a simulator is used which can just visualize the position the drone would fly to in the rendered 3D world. However, this is not as easy with real drones. One could imagine overlaying a live video feed of a camera attached to the drone that is presented to the drone operator with a mostly transparent picture containing a visualization of the target position in a virtual 3D world.

There is no haptic feedback in this control mode so that the operator can easily move the haptic tool in the workspace of the haptic device. However, a constant upward force $\overrightarrow{GF}$ is applied to counter gravity that pulls down the haptic tool due to its weight:

$$ff_{\text{TP}} : \mathbb{R}^3 \to \mathbb{R}^3, ff_{\text{TP}}\left(\begin{pmatrix} x_h \\ y_h \\ z_h \end{pmatrix}\right) := \begin{cases} \overrightarrow{HM} \odot \begin{pmatrix} x_h & y_h & z_h \end{pmatrix}^T & \text{if reset in progress} \\ \overrightarrow{GF} & \text{else} \end{cases}$$

$\odot$: element-wise multiplication

That's useful because the haptic tool can be positioned somewhere safely without the need to be continuously held by the operator's hand. Also, pressing the second button on the haptic device shortly triggers a reset of the haptic tool to the zero point of the haptic device.

For this control mode, the requirements to control frequency and latency do not apply as they do not impact the operator's control performance. In a way, it removes the human from the control loop and only demands to set an overall goal on what the flight controller should do. This is, without any doubt, supervisory control in its purest form.

Terrain-Aware Target Positioning

The Terrain-Aware Target Positioning control mode is an extension to the Target Positioning control mode. Calculations are the same as for the Target Positioning control mode, and the notes about predictor displays also apply to this mode. All previous control modes only translated inputs from the haptic device, that are read by the haptics-brain component, to commands which in turn are transmitted to the drone-controller component. In addition to that, this control mode involves sensor data from the drone that is gathered by the drone-controller component and sent to the haptics-brain – bidirectional instead of just unidirectional communication.

For this thesis, focus is set onto a LIDAR sensor that is attached to the drone in the simulator, providing a point cloud to the haptics-brain component in a specified frequency. The point cloud is then optionally converted to a mesh, and either the point cloud or the mesh is rendered on the haptic device haptically. Objects or obstructions that are near the drone are scanned by the LIDAR sensor. The resulting point cloud is added to a virtual 3D world that is represented by CHAI3D's `cWorld` class. CHAI3D offers to render this virtual 3D world onto the haptic device's workspace, which means that if the operator moves the haptic tool close enough to an object (in this case, a point of the point cloud or a triangle of the mesh), a resistive counterforce is applied.

Figure 4.5 shows screenshots of the environment being scanned by the LIDAR sensor in the simulator (left picture) and the resulting point cloud that can be felt haptically when present (middle picture), visualized by the already finished implementation of the haptics-brain component. The picture on the right shows the point cloud being converted to a mesh, which closes gaps between the points of the point clouds. More information about the CHAI3D library and how it works fundamentally is provided in Section 4.2.2.

4.4 Implementation

The software that is part of the thesis contribution is written entirely in C++14. As the build system, *CMake* is used and even though the software is specifically developed and

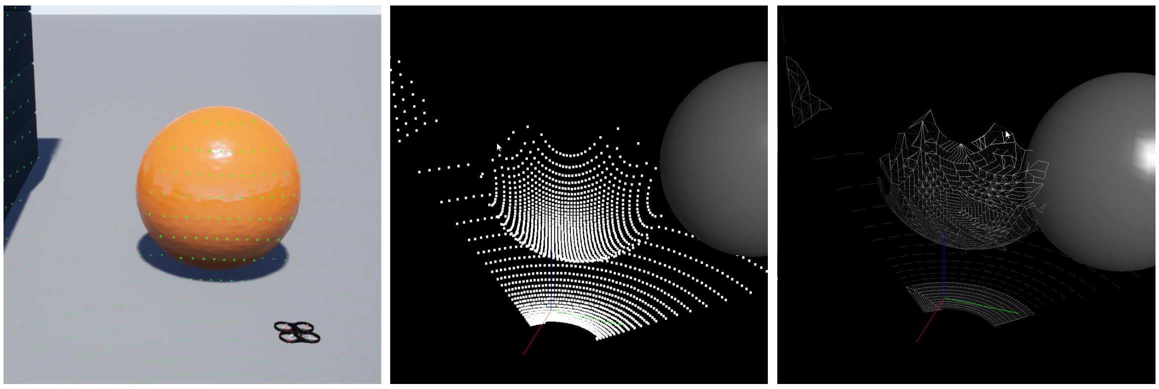


Figure 4.5: Screenshots that show how the drone's LIDAR scanner generates a point cloud representing the environment in the simulator (left picture), which is then sent to the haptics-brain component and haptically rendered by CHAI3D (middle picture). Optionally, the point cloud is converted to a mesh (right picture). The gray ball on the middle and right picture is the tip of the haptic tool of the haptic device.

tested to be working on *Microsoft® Windows 10* because the device drivers of the 3D Systems Touch™ haptic device have proven themselves to be working best on Windows, using CMake will make switching to other platforms easier as it works on e.g. Linux and Mac OS out-of-the-box too. CMake generates ready-to-use projects for various toolchains and IDEs.

For the thesis implementation, the *Microsoft® Visual Studio 2019* IDE and its toolchain (compiler, linker et cetera) is used. Therefore the `CMakeLists.txt` files, that are the project configuration files for CMake, might be partially specific to that IDE, but can be ported to another toolchain or IDE with little effort. Due to the enormous effort of setting up PCL as a dependency manually with CMake, it was convenient to additionally add *vcpkg* to the mix, which is an open-source cross-platform packet manager that focuses on distribution of C and C++ libraries. Vcpkg integrates nicely with CMake, but it is only used for setting up PCL as a dependency. So if you don't want to use the Terrain-Aware Target Positioning control mode, it is possible to spare the hassle of installing vcpkg and downloading PCL as a dependency, which is very large in size. In addition to that, *Git* is used for source code management and versioning.

Figure 4.6 shows how the software is structured, which dependencies the individual software components have and over which protocol they communicate. Boxes with solid borders each represent a component that is structured into its own individual project by CMake, and that is part of the thesis contribution. Dash-bordered boxes represent external components that are also structured into various projects, and are all the dependencies of the implemented components.

As can be also seen in Figure 4.6, the implementation is primarily split into two big components – the *haptics-brain* component and the *drone-controller-airsim* component. Those are individual programs that can be executed independently from each other on different computer systems, and are designed to work across LAN or even WAN networks.

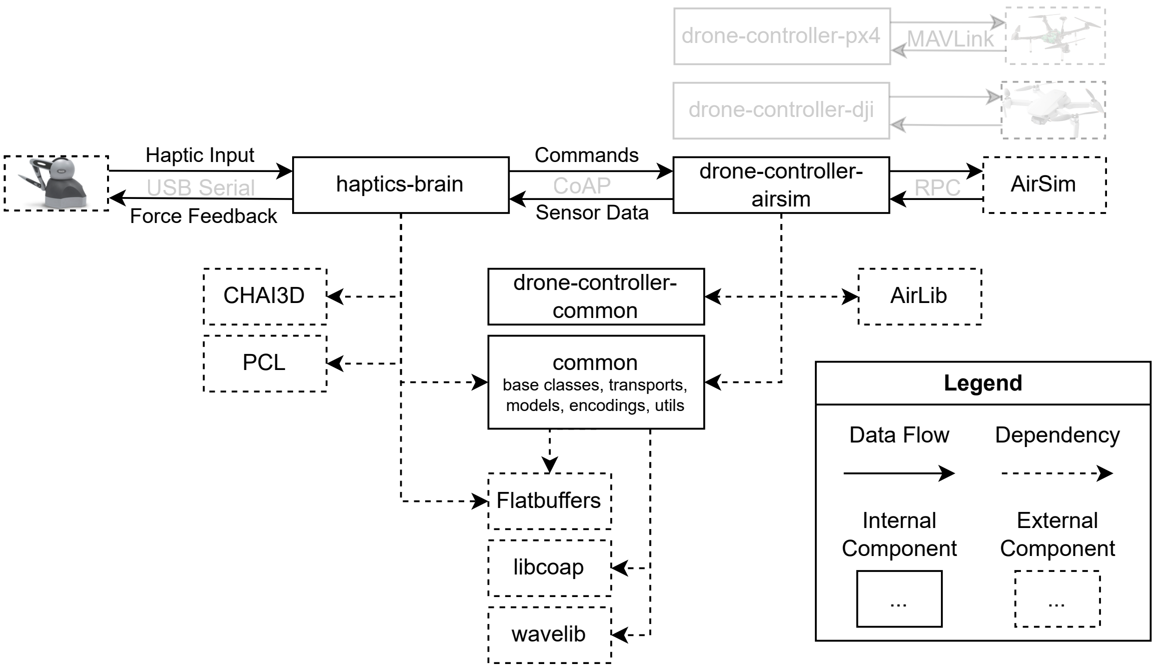


Figure 4.6: Diagram showing the various components of the implementation and their dependencies and communication.

4.4.1 Messaging

Messages are used for the communication between the haptics-brain and the drone-controller components. They contain mostly control data such as desired velocity or position generated by the various control modes that are to be sent unidirectionally from haptics-brain to drone-controller, with the exception of the Terrain-Aware Target Positioning control mode introduced in Section 4.3.1 which also sends sensor data the opposite way occasionally. Said data is modeled by nested FlatBuffers tables and arrays, which are introduced in the next subsection.

Because messages are to be sent and received by both main components, the messaging implementation is included in the *common* project that is linked by both haptics-brain and drone-controller. It consists of an abstract class `Transport` that defines a fairly abstract interface for both sending and receiving messages through simple-to-use functions, as can be seen in Figure 4.7. The receiving logic of the message, more specifically the code using the `receiveMessage` function, is responsible for directing received messages (commands or sensor data) to the corresponding implementation by itself. By having that abstract interface, communication is made more convenient from a developer's perspective and also keeps extendability in mind. It is straight-forward to implement another transport with another communication protocol without changing any code that utilizes messaging.

The messages need to be serialized into a byte array in order to send them via a specific protocol implementation, and also they need to be deserialized on the receiving side, for which FlatBuffers capabilities are used. Data (de-)serialization is implemented by classes implementing the abstract class `Encoding`. Implementations of that class that are part of

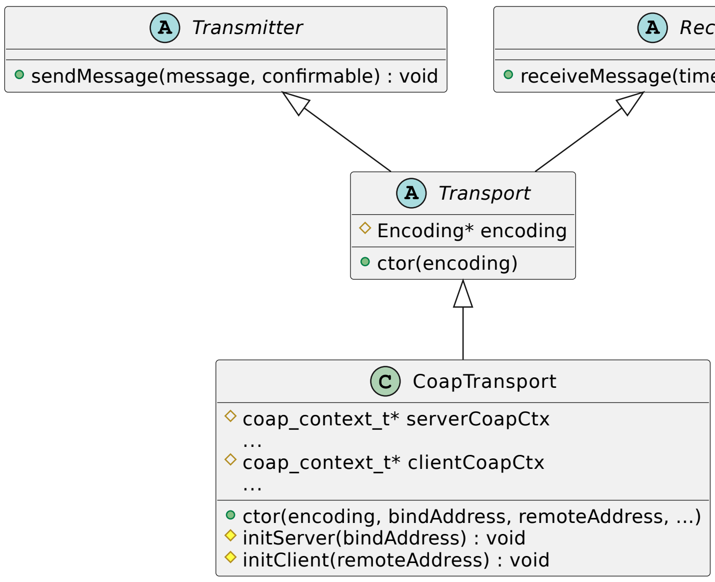

A Transmitter
sendMessage(message, confirmable) : void
A Receiver
receiveMessage(timeoutMs) : MessageT
A Transport
Encoding* encoding
ctor(encoding)
C CoapTransport
coap_context_t* serverCoapCtx
...
coap_context_t* clientCoapCtx
...
ctor(encoding, bindAddress, remoteAddress, ...)
initServer(bindAddress) : void
initClient(remoteAddress) : void

Figure 4.7: UML class diagram showing the class hierarchy of `Transport` class, which is the interface for messaging in the thesis implementation.

the thesis implementation are further investigated in the last subsection of this section. An instance of a specific `Encoding` subclass is passed to the constructor of a `Transport` implementation as sort of a dependency injection.

Currently, one transport is implemented with the underlying protocol being CoAP in the `CoapTransport` class. Libcoap is used as the CoAP implementation, and because CoAP uses a request/response model and the main components may communicate bidirectionally, both the haptics-brain and the drone-controller component act as a server and a client. That's why `CoapTransport` needs an address to bind on and a remote address to send to on instance creation, on which the client gets initialized and the servers start to serve a resource on the following endpoints, respectively:

```
PUT coap://<drone-controller-ip-address>:5685/message
PUT coap://<haptics-brain-ip-address>:5686/message
```

Serialized messages containing commands or sensor data are sent as a payload of a CoAP message by issuing a `PUT` request to the remote endpoint, either of type NON (default) or as a CON message. `PUT` stands for an idempotent request, meaning that it can be issued multiple times with the same effect – this might sound wrong due to the fact that the drone might be in a different state after each request, however commands will have the same effect on the drone even when it's current state changes with every request, e.g. setting a velocity goal or a target position to fly to. The drone's state changes, but not the effect.

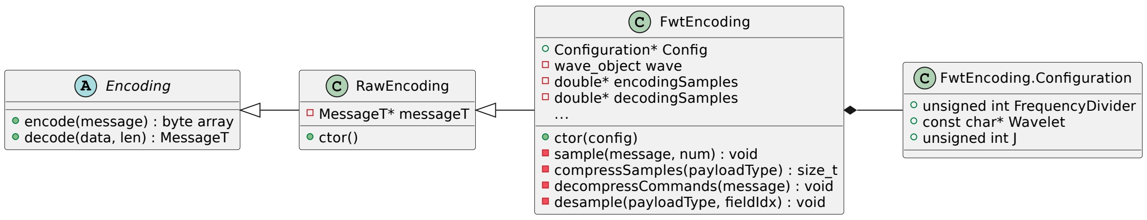

A Encoding
encode(message) : byte array
decode(data, len) : MessageT
C RawEncoding
MessageT* messageT
ctor()
C FwtEncoding
Configuration* Config
wave_object wave
double* encodingSamples
double* decodingSamples
...
ctor(config)
sample(message, num) : void
compressSamples(payloadType) : size_t
decompressCommands(message) : void
desample(payloadType, fieldIdx) : void
C FwtEncoding.Configuration
unsigned int FrequencyDivider
const char* Wavelet
unsigned int J

Figure 4.8: UML class diagram showing the class hierarchy of the classes `RawEncoding` and `FwtEncoding`, whose implementations are responsible for data (de-)serialization.

Data Models

```
include "command.fbs"; include "sensordata.fbs";

union Payload {
  MoveVelocityCommand, ArmDisarmCommand, SetAnglesThrottleCommand,
  MovePositionCommand, SetPositionPreviewCommand, PointcloudSensorData,
  SwitchPointcloudSensorCommand
}

table Message {
  payloads:[Payload];
}

root_type Message;
```

Listing 4.5: Schema file for messages and payloads in FlatBuffers IDL language.

As mentioned before, messages are modeled by tables and arrays as FlatBuffers schemes. More specifically, a message is modeled by a table that contains one or more payloads. The corresponding FlatBuffers schema can be found in Listing 4.5. A payload can be either a command that gives instructions to the drone or sensor data that is gathered by the drone-controller component, and it's represented by a union. The union makes it possible to let the payloads be of different type, even though FlatBuffers tables and structs do not support inheritance whatsoever.

In the source code and in all diagrams regarding the topic, classes or types representing messages, commands et cetera are suffixed with the letter *T*, for example the class `MessageT`. The reason for that is simply that the implementation solely uses the object-oriented API of FlatBuffers that is explained in Section 4.2.3, and the *T* stands for the object representation of the specific type.

Compression

To serialize `MessageT` objects into byte arrays that can be sent by the underlying communication protocol by the `Transport` implementation from the haptics-brain side, and to deserialize byte arrays back to object representation by the drone-controller component, an instance of a class inheriting from `Encoding` is needed that provides encode and decode functions. As can be observed in Figure 4.8, there are two implemented sub classes of `Encoding` called `RawEncoding` and `FwtEncoding`.

Wavelet name	Iteration depth (J)	Compression rate
`sym2`	1	$4/6 = 0.\overline{6}$
`haar`	1	$3/6 = 0.5$
`haar`	2	$2/6 = 0.\overline{3}$

Table 4.4: Compression rates for different configurations of parameters applicable to FWT-based encoding that are provided by the thesis implementation and their resulting constant compression rate (number of commands that are actually sent divided by the number of sampled commands), assuming the number of commands that are sampled before compression is six (`FrequencyDivider = 6`).

Class `RawEncoding` simply implements the `encode` function so that it uses the FlatBuffers library to build a byte array from a message object and the `decode` function to reverse that. How that works fundamentally is already explained in Section 4.2.3.

Section 4.1.4 suggested compression to lower the required data rate. To enable compression in the thesis implementation, class `FwtEncoding` implements a wrapper around `RawEncoding` that compresses a subset of the existing commands by applying the *Fast Wavelet Transform (FWT)* to a series of commands, similar to what is done to kinesthetic data in the case study presented in [19]. This thesis is not going into too much detail about how the FWT works (for that please refer to [19] and its sources), but generally compression is realized by sampling a series of commands and applying FWT with a specific wavelet to their values, effectively reducing the number of commands sent ultimately. So, instead of sending each command directly, a specific number of commands is sampled, then the sampled commands are put into the FWT compression algorithm which outputs approximation coefficients in a number less than the sampled commands, and those approximation coefficients are actually what is sent, after the compression was applied. The receiving side then uses the inverse FWT function to generate the original number of commands with the approximation coefficients as input. This process repeats itself over and over. Advantageous is that the compression rate, which is the ratio between the number of commands that are actually sent and the number of the sampled commands, is constant for a specific value of parameter J (iteration depth of FWT algorithm), the number of sampled commands (determined by `FrequencyDivider` option in source code) and the same wavelet (a wavelet is a named wave function that is used in the FWT algorithm).

Multiple configurations with different wavelets and parameters exist in the thesis implementation, namely the header file of the `FwtEncoding` class, which lead to different compression rates. Table 4.4 shows the compression rates for the configurations provided in the source code, assuming the number of commands to sample before compression (`FrequencyDivider` option in source code) is set to six.

Please note that the `FwtEncoding` class only applies the beforementioned compression to commands that are of type `SetAnglesThrottleCommand` or `MoveVelocityCommand`, which are the commands sent frequently by the Manual Flight and Velocity Joystick control modes, because those are the only control modes that continuously send commands. Compressing the infrequent commands that are sent by the Target Positioning and Terrain-Aware Target Positioning control modes would not make a lot of sense, because sampling is not possible (operators want the drone to execute their commands one at a time) and network bandwidth is not as relevant (commands are only sent on request, and the time of transmission does not matter too much).

4.4.2 Component *haptics-brain*

The haptics-brain component is the one that does most of the computations and communication. The main function of the component initializes the haptics system that is implemented in the `Haptics` class, of which the inner workings are explained in the following subsection. It also instantiates a class inheriting from the `Encoding` class such as the ones shown in Figure 4.8 and passes it to the used transport implementation. Only `CoapTransport` is currently available. Once the `CoapTransport` object has been initialized, or in other words the underlying CoAP client and server are configured, it gets passed to the control mode that is the default one on startup. How the control mode implementations work is also explained in one of the following subsections.

After all the systems such as haptics and the control mode are started, a user interface is presented in a console window, which allows switching control modes on the fly.

Haptics

The starting routine of the `Haptics` implementation starts a thread, referred to as the *haptics thread* in the rest of the thesis. Said haptics thread is responsible for running the CHAI3D haptics loop as explained in Section 4.2.2. In each iteration of the loop, the state of the button switches on the 3D Systems Touch™ haptic device is read. Should a control mode be already initialized and ready to go, it also runs the `calculateHapticFeedback` function that is implemented by the running control mode, which can be observed in Figure 4.10. The function is given to the `Haptics` object by setting a function pointer (as kind of a dependency injection), which is done by the user interface logic of the haptics-brain's main function. In the `calculateHapticFeedback` function, that is in fact run by the haptics thread as shown in Figure 4.9, the respective control mode reads the position and optionally rotation of the haptic tool (stylus) of the haptic device, does its calculations and tells the haptic device driver what force feedback should be applied by utilizing CHAI3D's API. The force feedback is either calculated manually and a force vector is being told to the device driver, or the `cWorld` abstraction is used instead, which does the calculation of the forces by itself.

When taking a look at the source code or the example haptics loop in Appendix A.1, it is noticeable that there is no waiting logic that sleeps a certain duration to achieve the desired haptics frequency $f_{haptics}$ specified in Section 4.1.3, as it is done in the logic of the control modes. That is because applying the force, regardless of it being done manually or with the `cWorld` abstraction, implicitly enforces timing due to the haptic device driver waiting until the next update cycle of the servo loop. Because of that, synchronization between the haptics loop and the haptic device is done implicitly. This needs to be taken into account while evaluating, e.g. interpreting latency measurements.

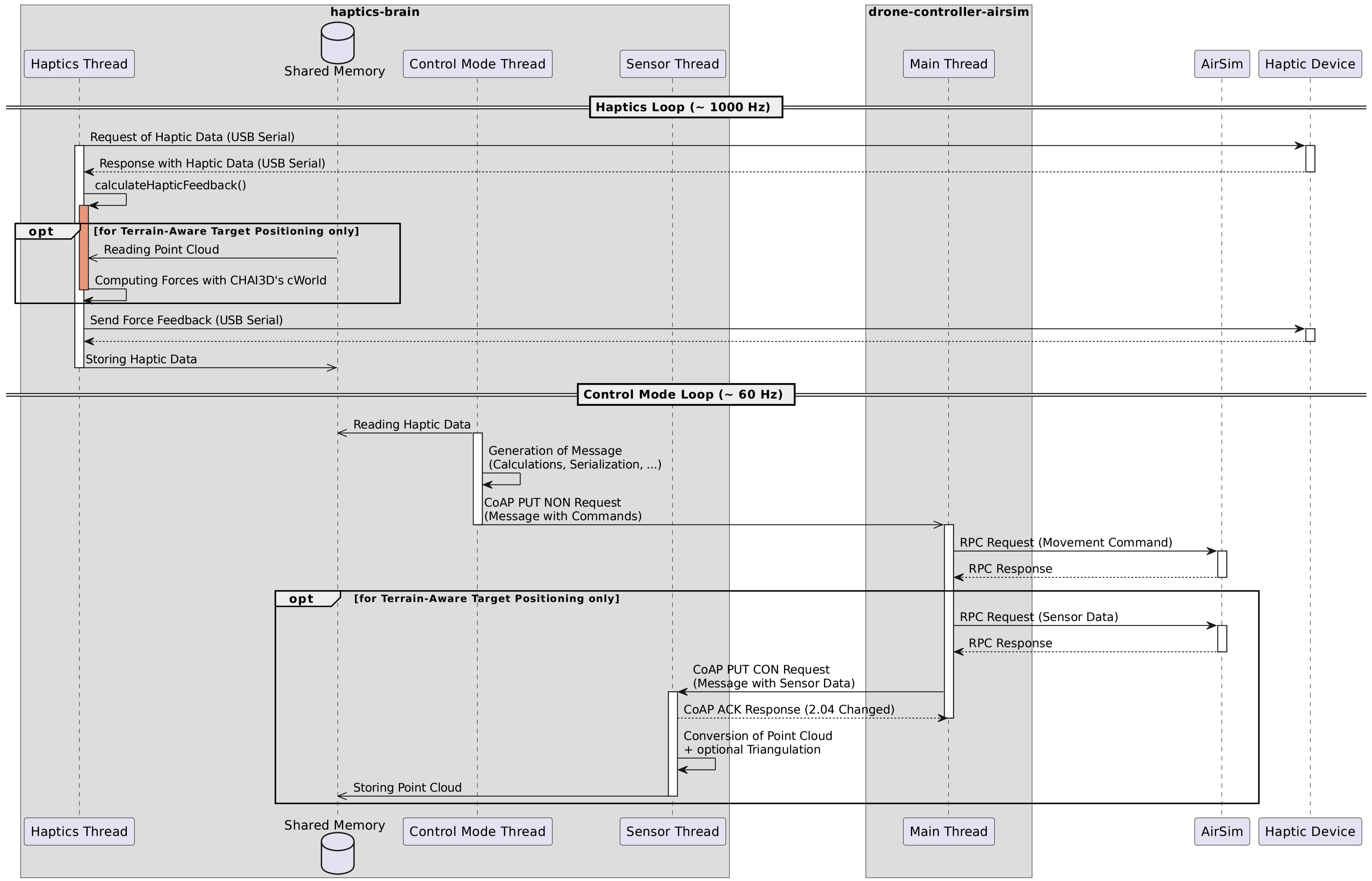
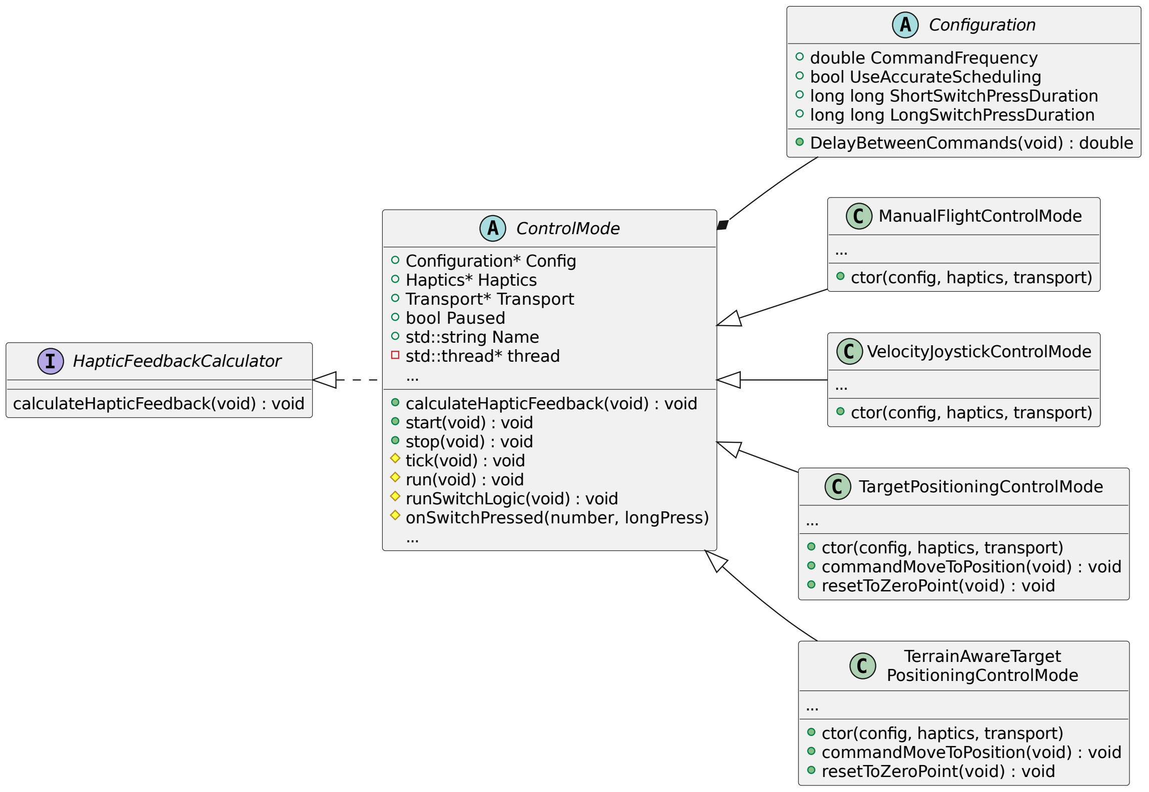

haptics-brain
drone-controller-airsim
Haptics Thread
Shared Memory
Control Mode Thread
Sensor Thread
Main Thread
AirSim
Haptic Device
Haptics Loop (~ 1000 Hz)
Request of Haptic Data (USB Serial)
Response with Haptic Data (USB Serial)
calculateHapticFeedback()
opt
[for Terrain-Aware Target Positioning only]
Reading Point Cloud
Computing Forces with CHAI3D's cWorld
Send Force Feedback (USB Serial)
Storing Haptic Data
Control Mode Loop (~ 60 Hz)
Reading Haptic Data
Generation of Message
(Calculations, Serialization, ...)
CoAP PUT NON Request
(Message with Commands)
RPC Request (Movement Command)
RPC Response
opt
[for Terrain-Aware Target Positioning only]
RPC Request (Sensor Data)
RPC Response
CoAP PUT CON Request
(Message with Sensor Data)
CoAP ACK Response (2.04 Changed)
Conversion of Point Cloud
+ optional Triangulation
Storing Point Cloud

Figure 4.9: UML sequence diagram that shows the communication between components as well as the data flow between threads. The diagram is specific to the `CoapTransport` implementation and drone-controller-airsim as the drone-controller implementation.



Configuration
double CommandFrequency
bool UseAccurateScheduling
long long ShortSwitchPressDuration
long long LongSwitchPressDuration
DelayBetweenCommands(void) : double
ControlMode
Configuration* Config
Haptics* Haptics
Transport* Transport
bool Paused
std::string Name
std::thread* thread
...
calculateHapticFeedback(void) : void
start(void) : void
stop(void) : void
tick(void) : void
run(void) : void
runSwitchLogic(void) : void
onSwitchPressed(number, longPress)
...
HapticFeedbackCalculator
calculateHapticFeedback(void) : void
ManualFlightControlMode
...
ctor(config, haptics, transport)
VelocityJoystickControlMode
...
ctor(config, haptics, transport)
TargetPositioningControlMode
...
ctor(config, haptics, transport)
commandMoveToPosition(void) : void
resetToZeroPoint(void) : void
TerrainAwareTarget PositioningControlMode
...
ctor(config, haptics, transport)
commandMoveToPosition(void) : void
resetToZeroPoint(void) : void

Figure 4.10: UML class diagram showing inheritance tree of control mode classes.

Control Modes

Control modes are responsible for the communication between the haptics-brain and drone-controller components. They implement the calculation of the haptic feedback in the `calculateHapticFeedback` function that is run in the haptics thread. Also they start an own thread when being started, which will from now on be referred to as the *control mode thread*, in which another loop is running. This is illustrated in the haptics-brain box in Figure 4.9.

The loop that is implemented in the `run` function listed in Figure 4.10 is responsible for generating and transmitting commands to the drone-controller component in the frequency $f_{control}$ that is set in Section 4.1.1. In the `run` function that the base class provides to any inheriting sub classes (does not need to be re-implemented for each control mode), each iteration of the loop the button switch logic is applied which may or may not influence further decision making or trigger a command to be sent immediately. Also executed in each iteration is the `tick` function that is implemented by each control mode subclass itself. Implementations of the `tick` function take the inputs of the haptic device such as the position and/or rotation of the haptic tool from shared memory, which is filled by the `calculateHapticFeedback` function that runs in the haptics thread, and using them as inputs to firstly decide if a command is to be sent (e.g. if the inputs changed) and secondly to calculate the values of the command that is sent each iteration – or in terms of the

Target Positioning and Terrain-Aware Target Positioning control modes, no command is to be generated and other housekeeping can be done in the function. For the direct control modes, namely Manual Flight and Velocity Joystick, the values are more or less calculated by applying the inputs to the mathematical functions ds_x with $x \in \{\mathrm{MF}, \mathrm{VJ}\}$ defined earlier, just that they are implemented in C++ and some corrections are also applied. At the end of the loop iteration, the thread is set to sleep for

$$T_{control} = \frac{1}{f_{control}} = \frac{1}{60\,\mathrm{Hz}} = 0.01\overline{6}\ \mathrm{s}$$

so that the frequency is met approximately (approximation is enough for such a relatively low frequency).

Manual Flight

```
table SetAnglesThrottleCommand {
  pitch:float;
  roll:float;
  yaw:float;
  duration:float;
  throttle:float = 0.0;
}
```

The Manual Flight control mode implementation continuously sends messages containing a `SetAnglesThrottleCommand` command to the drone-controller. Appendix A.4 provides UML class diagrams showing the interfaces of all the control mode implementations, and the used command is modeled by the table schema above.

In order to fill the angle and throttle values, the function ds_{MF} that was introduced in Section 4.3.1 is implemented in C++, taking the inputs from shared memory that were obtained by the `calculateHapticFeedback` implementation.

For this control mode, the ff_{MF} function is used to calculate the force vector to apply to the haptic device needed by the implementation of the `calculateHapticFeedback` function.

Velocity Joystick

```
table MoveVelocityCommand {
  velocity:Vec3;
  duration:float;
  yaw:float = 0.0;
}
```

Similar to the Manual Flight control mode implementation, the implementation of the Velocity Joystick control mode sends messages continuously containing a command in the form of a FlatBuffers buffer called `MoveVelocityCommand` modeled by the table schema before.

To calculate the values of the command, the ds_{VJ} function from Section 4.3.1 is used in the same way as before, just that the output is a velocity vector now.

Function ff_{VJ} is used to calculate the force vector, manually applying it in the implementation of `calculateHapticFeedback` that the control mode provides.

Target Positioning

```
table MovePositionCommand {
  position:Vec3;
  velocity:float;
  yaw:float = 0.0;
}
```

In the implementation of the Target Positioning control mode, a listener for a short button press is used to send a `MovePositionCommand` command within a message to the drone-controller, which is modeled by a table with the schema above.

Positional values and the yaw goal is calculated by the ds_{TP} function defined in Section 4.3.1, with the values from the `calculateHapticFeedback` implementation being input.

ff_{TP} is the force feedback function that tells the force vector to be applied to the haptic device in the `calculateHapticFeedback` implementation.

Terrain-Aware Target Positioning

The Terrain-Aware Target Positioning control mode works the same as the Target Positioning control mode (using the same desired state function and button press implementation), but is special in the way that it additionally receives sensor data in form of a point cloud scanned frequently by a LIDAR sensor from the drone-controller-airsim component.

```
table PointcloudSensorData {
  points:[Vec3];
  organized:bool = true;
  height:uint = 0;
  width:uint = 0;
}
```

Receiving the sensor data is done in yet another thread referred to as the *sensor thread* that is started by initializing the `PointcloudSensor` class on the haptics-brain component. It runs a loop that waits for a message containing a `PointcloudSensorData` payload modeled by the table schema above, which contains the points of the point cloud, and constructs the point cloud in CHAI3D representation on each iteration. See Figure 4.9 for an overview about the various threads introduced by now.

The `PointcloudSensor` object is passed to the instance of the Terrain-Aware Target Positioning control mode, and can be used to fetch the point cloud in CHAI3D representation that is constructed in specific intervals in the sensor thread.

If triangulation is enabled in the configuration that can be found in the header of the `PointcloudSensor` class, the sensor thread also constructs a triangle mesh by using an algorithm that is provided by PCL. Available algorithms that proved themselves to construct triangle meshes from the point clouds quite well and fast enough are the 'Greedy Projection' algorithm presented by [50] and the 'Organized Fast Mesh' algorithm introduced by [51].

The latter only works on organized point clouds as the name implies, which means that the point clouds have a width and a height and their points are stored two-dimensionally (e.g. you can obtain a depth value for a specific discrete value on the x and y axes). For the specific LIDAR and point cloud implementation used in this thesis, the point cloud is organized, but that may not be the case for other combinations of drones and sensors.

In contrast to the other control modes that apply a force vector explicitly in a manual way, the Terrain-Aware Target Positioning control mode uses the `cWorld` abstraction of CHAI3D to render the received point cloud (or the generated mesh) haptically. For that, the point cloud provided by the `PointcloudSensor` class is added to the `cWorld` object, after which interaction forces are calculated and sent to the haptic device, similar to how it's done in the second pseudocode in Appendix A.4.

4.4.3 Component *drone-controller-airsim*

The drone-controller-airsim component is the drone-controller implementation for usage with AirSim to simulate the drone. It connects to AirSim's RPC interface and enables remote control of the drone. After that, it runs an endless loop that waits for messages to be received in each iteration. Once a message is received, the contained commands are parsed and several corrections and transformations are applied to the values. For example the axes are switched so that the drone flies in the intended direction, and vectors are mapped to the coordinate system used in AirSim, as conceptualized in Section 4.3.1. Then, the corresponding remote function of the RPC interface is called that leads to the command's desired effect, as can be seen in Figure 4.9. So basically it translates the command into something that is understood by AirSim – or in case of another potential drone-controller implementation, the hardware drone.

If the Terrain-Aware Target Positioning control mode is active, the component is also responsible for reading the LIDAR sensor and sending the resulting point cloud to the haptics-brain component in a specific interval. The LIDAR sensor is implemented by the `LidarSensor` class that is based on the same super classes used in the haptics-brain implementation. A new thread is spawned to do that, similarly to how it's done on the haptics-brain component.

CHAPTER 5

Evaluation

In this chapter, the demonstrator will be evaluated. Experiments will be done to show that the implementation meets the requirements such as maximum latencies and minimum frequencies in Section 5.1.1 and Section 5.1.2. Section 5.1.3 evaluates how well the compression performs and answers the question if the applied compression algorithm is suitable for the application. Also, tests are done to find out how well the implemented control modes perform and to give suggestions on what control mode to use in which situation in Section 5.1.4. Last but not least, Section 5.2 analyzes what data rate is required from the underlying communication channel and which overhead is introduced by the various protocols and formats used in the implementation.

All evaluations will be done on a single PC which eliminates additional overhead by network components between the software components and allows for comparability. Table 5.1 shows which hardware and software the system consists of as well as some information about how the software components of the demonstrator are compiled for evaluation. A different compiler is used for the drone-controller-airsim component due to an incompatibility of the AirSim library with the newer MSVC v142 compiler. For time measurements, the Windows API function `GetSystemTimePreciseAsFileTime` for high precision time measurements is used which is advertised to retrieve the system date and time with the highest possible precision that is said to be less than 1 µs [52].

5.1 Experiments

Due to the two threads of the haptics-brain component that run on different frequencies but exchange data in shared memory with each other, it is not trivial to simply measure a RTT and be done with it. Also, as figured out in Section 4.1.2, there is no RTT at all due to the unidirectionality of communication. That's why two different latencies are measured: the latency that is introduced by the control mode in Section 5.1.1 and the latency that is introduced by communication with the haptic device and calculation of the force feedback in the haptics thread in Section 5.1.2. Figure 5.1 shows the directed latency paths for which measurements are taken in the following subsections.

	haptics-brain	drone-controller-airsim
Compiler	MSVC v142 (VS 2019)	MSVC v141 (VS 2017)
	Flags: `/std:c++14 /O2 /Ob2 /fp:precise`	
Hardware Components	CPU: AMD® Ryzen™ 7 2700X	
	GPU: NVIDIA® GeForce RTX 3080 Founders Edition	
	RAM: 2x8GB DDR4-3000 CL15-17-17-35 (Dual Channel)	
Operating System	Microsoft® Windows 10 Professional (SDK 10.0.19041.0)	

Table 5.1: Characteristics of the PC system used for evaluation.

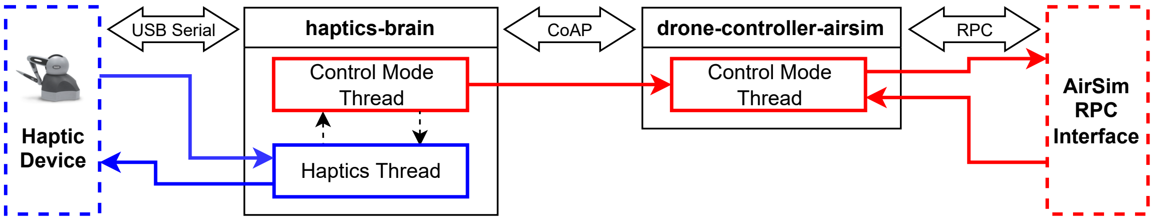


Figure 5.1: Diagram showing different latency paths that work independently but contribute to overall latency.

5.1.1 Control Mode Latency

To build confidence that the latency introduced by the computations and communication of the control mode implementations in both haptics-brain and drone-controller-airsim meet the requirement for the control latency defined in Section 4.1.2, an experiment is done. Also, the computational overhead that comes with the implemented FWT-based compression is of interest.

Measurements of the latencies will be taken for the control modes that continuously send movement commands, namely the Manual Flight control mode and the Velocity Joystick control mode, once with no compression and once with FWT-based compression enabled. Remember that the other control modes do not have a strict control mode latency requirement as explained in Section 4.3.1. That's why only the first two modes are evaluated.

Please consider having a look at the source code[1] to get a better understanding of where the measurements are taken exactly. They are calculated by the difference of an absolute timestamp taken at the beginning of command construction on the haptics-brain component and another one after the last RPC call finished on the drone-controller-airsim component:

$$t_{latency} = t_{end} - t_{begin}$$

To characterize the results of the experiment mathematically, assuming that the latency is *Student's* t-distributed, the expected value $\bar{x}$, the standard deviation s_n as well as the confidence intervals (CIs) for the expected value and for the standard deviation (CI_{s_n}) will be calculated for each sample.

[1] `https://code.ovgu.de/comsys-group/haptic-drone-control`

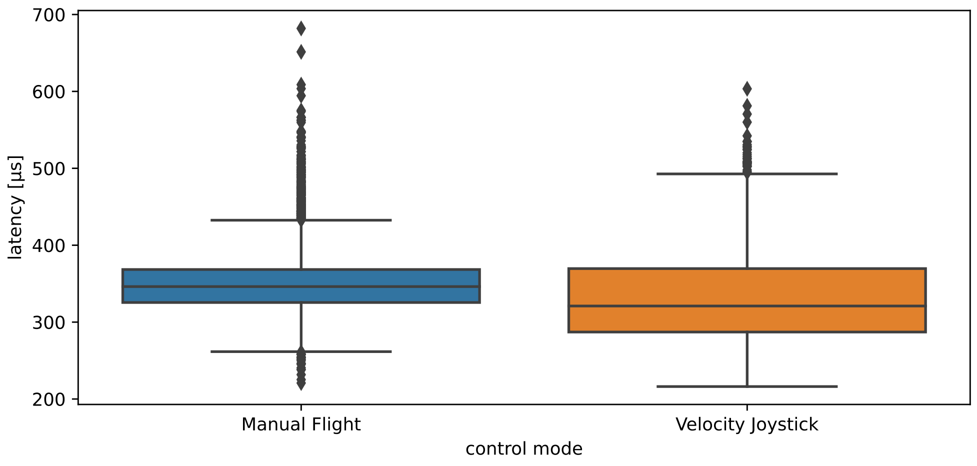


Figure 5.2: Box plots showing $n = 10000$ measurements of the control mode latency for each direct control mode, compression disabled.

Control Mode	Compr.	$\bar{x}$ [µs]	s_n [µs]	CI_{s_n} [µs, µs]
Manual Flight	none	349.43316 ± 0.74436	37.97341	[37.45436, 38.50715]
Velocity Joystick		328.63979 ± 1.01282	51.66916	[50.96291, 52.39541]
Manual Flight	FWT	23411.15497 ± 2.48753	126.90198	[125.16739, 128.68567]
Velocity Joystick		23295.59470 ± 5.56958	284.13300	[280.24925, 288.12667]

Table 5.2: Characteristics of control mode latencies for each direct control mode, no compression versus FWT compression (Symlet *sym2*, $J = 1$). Calculations are based on $n = 10000$ measurements respectively and using a confidence level of $1 - \alpha = 0.95$.

Evaluation

Figure 5.2 shows box plots of $n = 10000$ latency measurements for each of the direct control modes without utilizing any means of compression. To outline the performance impact of FWT-based compression, Figure 5.3 visualizes the same measurements but with compression enabled (Symlet *sym2* and $J = 1$). In Table 5.2 you can see the corresponding expected values, standard deviations and confidence intervals.

All measurements of the control mode latency, regardless of compression, lie far below the maximum $T_{max} \approx \frac{400\,\text{ms}}{2} = 200\,\text{ms}$ that is specified in Section 4.1.2, and leave lots of room for additional latencies that would be introduced in the real world, such as hardware components and network transmission times. The low latencies also easily allow the control frequency $f_{control} = 60\,\text{Hz}$ specified in Section 4.1.1 to be satisfied. FWT-based compression adds computational overhead in the form of latency in order of magnitude of two, as can be expected.

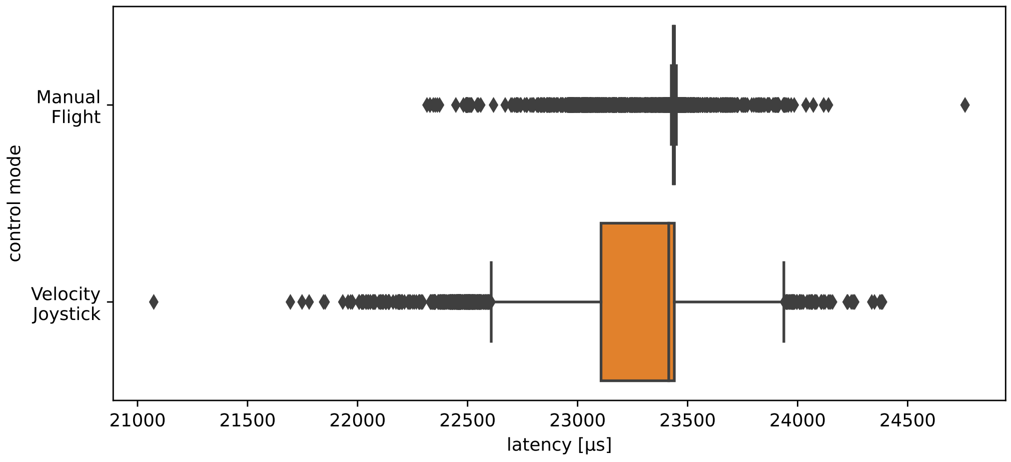


Figure 5.3: Box plots showing $n = 10000$ measurements of the control mode latency for each direct control mode, FWT-based compression enabled (Symlet *sym2* and $J = 1$).

5.1.2 Haptics Latency

Another requirement of the thesis implementation is that it's able to keep up with the haptics frequency $f_{haptics}$ defined in Section 4.1.3. If the execution time is lower or equal than the inter-arrival time

$$T_{haptics} = \frac{1}{f_{haptics}} = \frac{1}{1000\,\text{Hz}} = 0.001\,\text{s} = 1000\,\mu\text{s}$$

of the haptics frequency, the implementation is able to keep up with the update rate of the haptic device. To find out, an experiment will be done in which measurements of the execution time of the `calculateHapticFeedback` implementation will be taken for each control mode. For the Terrain-Aware Target Positioning control mode, different AirSim settings regarding the LIDAR sensor are tested, too.

Once again, please consider having a look at the source code[2] to get a better understanding of where the measurements are taken exactly. They are calculated by the difference of absolute timestamps taken before and after the `calculateHapticFeedback` function call in the implementation of the `Haptics` class, which can be found in the haptics-brain component:

$$t_{latency} = t_{end} - t_{begin}$$

To characterize the results of the experiment mathematically, assuming that the latency is *Student's t*-distributed, the expected value $\bar{x}$, the standard deviation s_n as well as the CI for the expected value will be calculated for each sample.

[2]`https://code.ovgu.de/comsys-group/haptic-drone-control`

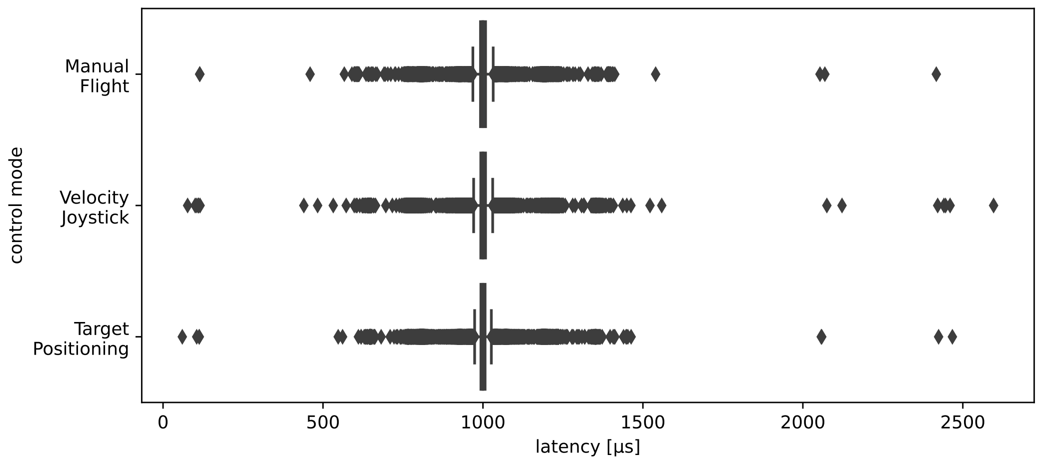


Figure 5.4: Box plots showing $n = 10000$ measurements of the haptics latency for each simple control mode.

Control Mode	AirSim settings	$\bar{x}$ [µs]	s_n [µs]
Manual Flight	-	1000.31664 ± 1.87361	95.58244
Velocity Joystick	-	1000.67837 ± 1.92127	98.01396
Target Positioning	-	1000.40290 ± 1.95436	99.70208
Terrain-Aware	90hfov-90vfov-50ch-10rot	1000.68321 ± 1.76047	89.81078
Target Positioning	90hfov-90vfov-100ch-12rot	1001.87894 ± 2.46986	126.00042
	180hfov-180vfov-50ch-10rot	1000.09301 ± 1.39386	71.10812
	180hfov-180vfov-100ch-12rot	1001.46263 ± 2.34997	119.88388
	360hfov-180vfov-50ch-10rot	1000.88498 ± 1.84505	94.12552
	360hfov-180vfov-100ch-12rot	1002.89444 ± 2.77944	141.79367

Table 5.3: Characteristics of haptics latencies for different control modes and AirSim sensor settings. Calculations are based on $n = 10000$ measurements respectively and using a confidence level of $1 - \alpha = 0.95$.

Evaluation

Figure 5.4 shows box plots that visualize the $n = 10000$ measurements of the haptics latency for the simple control modes, more specifically the Manual Flight, Velocity Joystick and Target Positioning control modes. The measurements taken specifically for the Terrain-Aware Target Positioning control mode and for the different AirSim LIDAR configurations are depicted in Figure 5.5. Numbers that characterize the sample set further are available in Table 5.3.

The expected values basically show the average sum of the execution time of the haptic calculations of the own implementation and the time waited until the next update cycle of

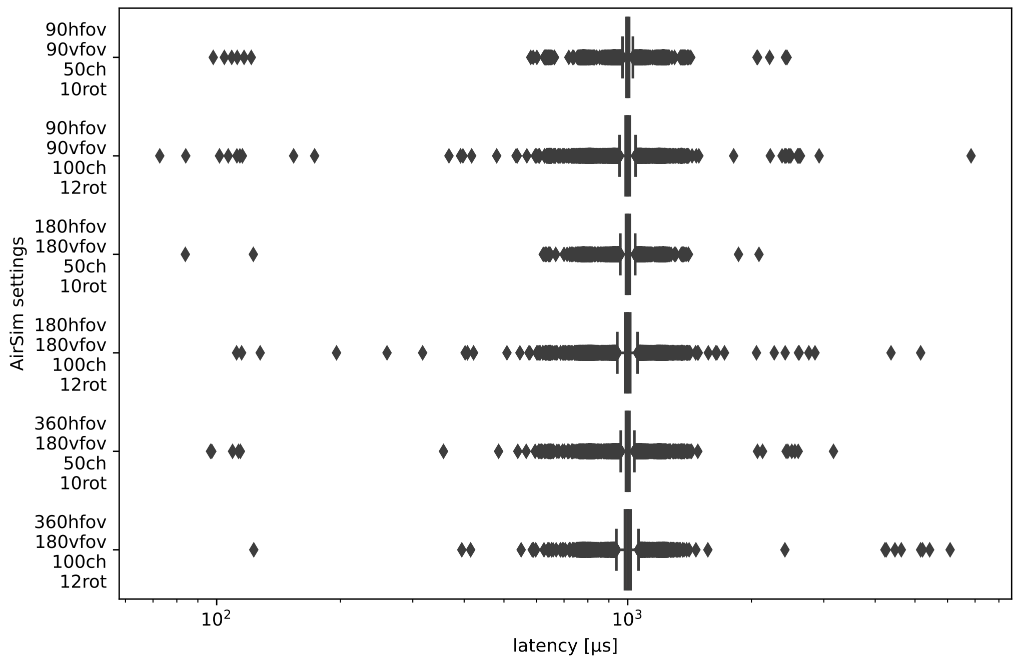


Figure 5.5: Box plots showing $n = 10000$ measurements of haptics latency for Terrain-Aware Target Positioning control mode, for different AirSim sensor settings.

the haptic device. As can be seen, the synchronization between the haptic device and the haptics driver does not seem to be perfect, because there are lots of outliers. It seems that the haptics frequency of $f_{haptics} = 1000\,\text{Hz}$ cannot be achieved reliably. There are outliers that indicates the haptics latency momentarily triples (the haptics frequency getting to as low as a third of the desired frequency). However, the bulk of the latencies are very close to the desired haptics latency $T_{haptics} = 1000\,\mu\text{s}$ and the user experience does not degrade noticeable.

5.1.3 Compression Error

Also of interest is the answer to the question if the implemented FWT-based compression performs well enough to not degrade the operator's experience and control performance too much, or in other words, is applicable to the problem proposed in the thesis. In order to get that answer, an experiment will be done.

The values used in positions and vectors that are calculated by the control mode implementation of the haptics-brain component are collected before compression. Also, the corresponding values that were decompressed by the drone-controller component that are expected to differ from the original values are taken. This is done for different FWT configurations, of which the parameters are explained in Section 4.4.1. Then, the Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) will be calculated to assess the occurring errors.

Evaluation

Table 5.4 shows the calculated errors of a sample size of $n = 10000$ for each control mode and FWT configuration. Symlet *sym2* with $J = 1$ produces lower errors than the *haar* symlet with $J = 1$ for every test case. Also, the errors are low enough so that the FWT-based compression is definitely applicable to the kinesthetic data that is communicated by the thesis implementation, which complements the conclusion of [19] that an FWT-based compression approach is feasible.

Figure 5.6 shows an exemplary cutout from a time series of pitch values, with the original uncompressed values from the haptics-brain component being on the blue continuous line, and the values that were decompressed on the drone-controller component laying on the orange dashed line. Below the time series, the corresponding absolute error is plotted. As can be seen, the graphs barely defer from each other as the error is so low. Such errors will not degrade the performance of the operator too much as it will be barely noticeable, and are low enough so that they can be corrected in time by the operator. What comes to attention is that the absolute error is high when the gradient of the pitch changes drastically, e.g. when the sign changes spontaneously.

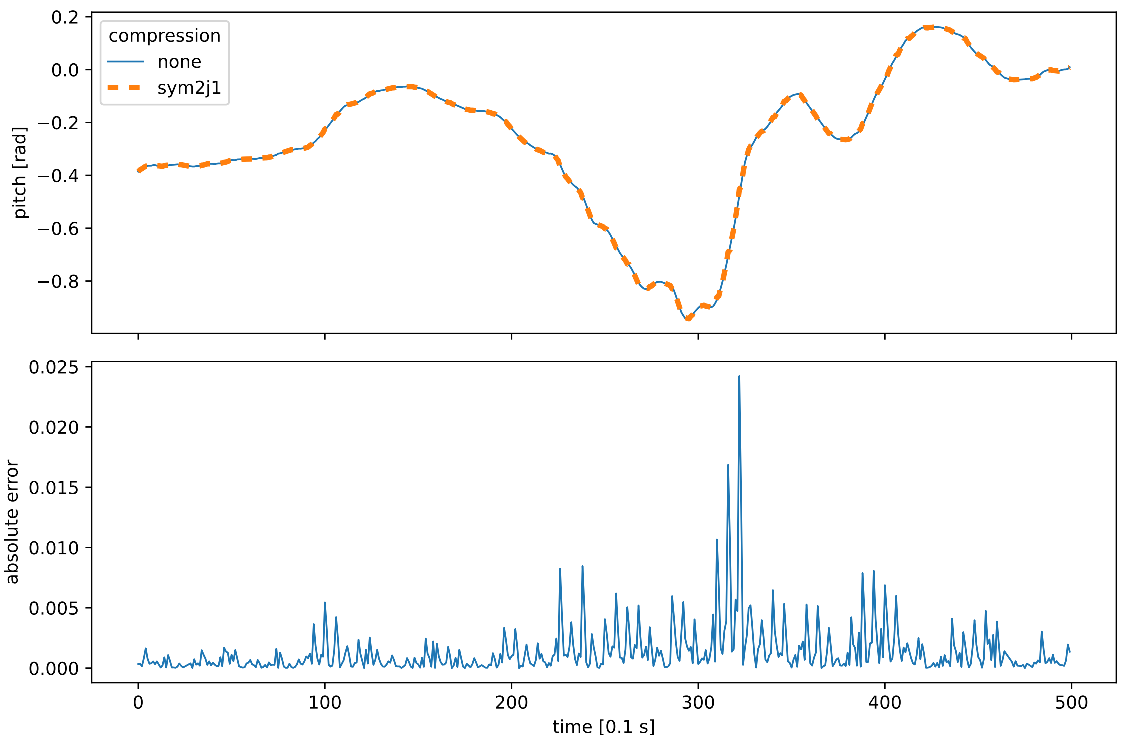


Figure 5.6: Line plot showing arbitrary time series with original and decompressed value, below another line plot showing the absolute error for the corresponding time.

Control Mode	FWT config.	Comp.	MAE	RMSE
Manual Flight	*haar*, $J = 1$	yaw	0.0139381043	0.2401707213
		pitch	0.0024922066	0.0227183118
		roll	0.0054820715	0.0258818107
		throttle	0.0005103184	0.0028538869
	sym2, $J = 1$	yaw	0.0039076283	0.0186814441
		pitch	0.0010339226	0.0043668837
		roll	0.0022474502	0.0054244648
		throttle	0.0004804751	0.0022642765
Velocity Joystick	*haar*, $J = 1$	yaw	0.1074342914	0.3083017231
		v_x	0.0101691560	0.0423465858
		v_y	0.0097758820	0.0318927135
		v_z	0.0135755688	0.0426435064
	sym2, $J = 1$	yaw	0.0504304648	0.1594825163
		v_x	0.0038642132	0.0087465089
		v_y	0.0031162306	0.0078186653
		v_z	0.0044419764	0.0146812443

Table 5.4: Compression errors for different control modes and FWT configurations. Calculations are based on $n = 10000$ measurements respectively.

5.1.4 User Experience of Control Modes

The control modes allow operating the drone in fairly different ways and thus each control mode may be predestined for different applications. An experiment will be done to subjectively evaluate the control modes and to find out which control mode is best for a given scenario.

One person, more specifically the author of the thesis, gets into the role of the operator and operates the drone so that it flies around a cuboid in a virtual world, once in each control mode. After that, the person gives feedback to the user experience by rating certain aspects of the control mode on a scale from one to ten. The aspects are:

- the overall *difficulty*,
- the *maneuverability* meaning how quickly the drone can be turned and how fast the drone reacts to changing inputs,
- the *decision time* the control mode allows between movements,
- the subjective *quality of haptic feedback* and
- the *precision* the operator can achieve.

For example, the latter can be important when operating in an environment that does not allow for a lot of error, where diverging from a specific flight path can be dangerous to the drone's integrity or not be possible at all – imagine tight air ducts or collapsed buildings.

	Manual Flight	Velocity Joystick	Target Positioning
Difficulty	10/10	3/10	1/10
Maneuverability	10/10	5/10	2/10
Decision Time	1/10	5/10	10/10
Quality of Haptic Feedback	1/10	3/10	1(7)/10
Precision	1/10	5/10	10/10

Table 5.5: Subjective user experience of control modes with ratings from one to ten based on various aspects. Ratings done by thesis author.

Evaluation

The Manual Flight control mode is by far the most difficult, as can be seen in Table 5.5, as the axes of the aircraft need to be controlled by the operator itself and thrust needs to be regulated manually. However, it gives the maximum freedom of movement and can be used to achieve hectic and visually impressive flight paths. It is very responsive and allows to fly sophisticated curves continuously. That's why this control mode is predestined for competitive drone races and camera flights that require fast movements, e.g. in the entertainment sector.

While not the easiest in general, the Velocity Joystick control mode is the easiest of the more direct control modes. The operator just needs to push the haptic tool in the direction the drone shall fly to, and the drone does it. Due to the fact that the drone will hover safely while not giving any input, it gives endless decision time optionally, but while flying the operator needs to decide quickly. Other than the Manual Flight control mode, it gives minimal relevant haptic feedback representing the speed the drone is destined to fly. This control mode is the 'jack of all trades' – it combines ease of operation, continuous movement and haptic feedback. It is suggested for every scenario on which the other control modes are not applicable.

For scenarios that require very precise and thoughtful movements, and where time is not the greatest concern, for example for rescue operations in collapsed buildings or reconnaissance in indoor environments, the Target Positioning control mode is perfect. This mode makes continuous movements very hard to do, which results in a fairly slow pace – the drone gets to its destination safely but slowly. It is not realistically feasible for situations where the flight controller of the drone is not able to hold the drone in position safely, for example in bad weather conditions. Without the addition of some kind of three-dimensional data source, there is no haptic feedback at all. If a LIDAR sensor or stereoscopic vision can be utilized, the Terrain-Aware Target Positioning control mode provides haptic feedback in the form of a haptical presentation of the surrounding environment which, depending on the resolution and quality of the sensor or stereoscopic image, aids in judgement of distances and finding the right target position.

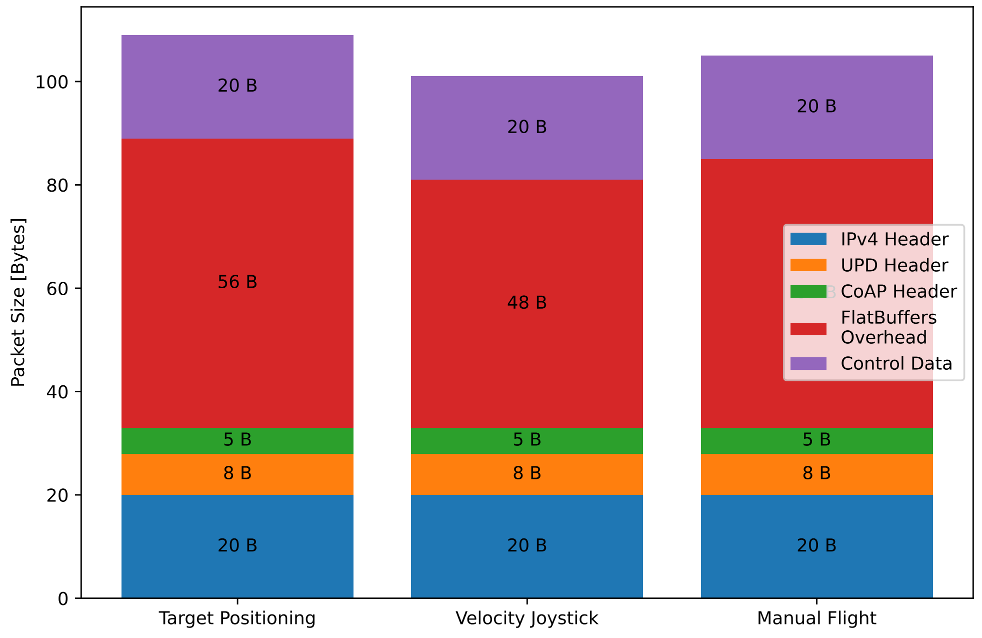


Figure 5.7: Bar plot showing the Internet Protocol Version 4 (IPv4) packet sizes for single messages containing one command that are sent continuously by the corresponding control mode. The stacks visualize the parts the packet consists of and what portion said parts take from the total size of the packet. For the Target Positioning and Terrain-Aware Target Positioning control mode, the size of the message that contains one `MovePositionCommand` is shown instead, as there are no continuous messages to be sent by default.

5.2 Data Rate and Overhead

The communication between haptics-brain and drone-controller-airsim should at least be able to run on WLANs and even better on WWANs as mentioned in Section 4.1.4. CoAP is mainly to be used over IP, and packets can be captured easily with tools like *Wireshark*. Figure 5.7 shows the sizes and compositions of IPv4 packets that contain a message with a single control command generated by the corresponding control mode implementation, which were read from Wireshark captures. Also, the messages that are sent continuously by the implementations of the direct control modes, namely the Manual Flight and Velocity Joystick control mode, are fixed in size due to their static structure defined in the FlatBuffers schema files. With the size of a single UDP packet containing a CoAP message containing a control command, named l_x in the following, and the frequency $f_{control}$ that is defined in Section 4.1.1, the data rate for the direct control modes dr_x with $x \in \{\mathrm{MF}, \mathrm{VJ}\}$ can be calculated trivially. Once again, due to the infrequency of control commands and because latency is not a huge concern, the calculations for the Target Positioning and Terrain-Aware Target Positioning control modes are left out.

$$\begin{aligned} dr_x &= f_{control} * l_x \ [\mathrm{B\,s^{-1}}] \\ dr_{\mathrm{MF}} &= 60\,\mathrm{Hz} * (20\,\mathrm{B} + 8\,\mathrm{B} + 5\,\mathrm{B} + 56\,\mathrm{B} + 20\,\mathrm{B}) = 6.54\,\mathrm{kB\,s^{-1}} \\ dr_{\mathrm{VJ}} &= 60\,\mathrm{Hz} * (20\,\mathrm{B} + 8\,\mathrm{B} + 5\,\mathrm{B} + 52\,\mathrm{B} + 20\,\mathrm{B}) = 6.30\,\mathrm{kB\,s^{-1}} \end{aligned}$$

After calculating the data rates for each control mode like above, it turns out that the thesis implementation should be able to communicate over all technologies that were mentioned earlier, given that IPv4 is used. This conclusion is to be taken with a grain of salt though, since only the theoretical maximum data rates are considered and the data rates of those technologies may be much lower than the maximum in real world environments. Also latencies that are practically introduced by the technologies are not taken into account either, which may or may not lower the user experience and operator's performance to an unacceptable level too, regardless of the bandwidth being sufficient or not.

Please note that the Terrain-Aware Target Positioning control mode causes additional traffic due to exchanging point cloud data frequently (with a size depending on the number of scanned points), but due to the fact that the operator can just wait for the transmission being done, the required data rate does not need to be satisfied – the transmission just needs longer and introduces additional latency. Thus, any further more complex calculations for the Terrain-Aware Target Positioning control mode are avoided at this point. Also, generally the calculations are only done for the communication that consumes the most bandwidth, which are the commands that are continuously sent. Those are transmitted as a CoAP NON message, which does not trigger any acknowledgements to be sent. However, for certain actions such as starting and stopping the drone via pressing the buttons on the haptic device, a CON message is sent which also adds traffic due to the ACK message and potential retransmissions.

CHAPTER 6

Conclusion

This chapter summarizes what was achieved and if the requirements set in Section 4.1 are met by the thesis implementation. Ending the thesis, Section 6.2 gives an outlook on potential future work on the thesis implementation.

6.1 Summary

The thesis results in a software implementation that achieves the goal of implementing a demonstrator that explores new ways of controlling a quadrotor drone with a six DoF haptic device and that provides a platform for further research on drone control and haptic communication.

All requirements that are set in Section 4.1 are met sufficiently by the implementation, and the calculated data rates in Section 5.2 suggest that communication between the components can be established stably by nearly all common WLAN and WWAN technologies, as long as the network is not congested or otherwise restricted by environmental factors.

The implemented control modes work subjectively good enough, however each of them is suitable for different applications. Also, it has been shown that the FWT-based compression proposed by [19] is applicable to the data model of the implementation.

6.2 Future Work

There are a lot of directions further work on the thesis implementation could take. The most obvious one is to re-evaluate the demonstrator with a real hardware quadrotor drone, which would require to implement a new drone-controller component and a way to provide visual feedback, e.g. by transmitting and displaying a camera feed to the operator.

One problem definitely is that the user experience of the control modes evaluated in Section 5.1.4 is highly subjective and only considers the feedback of the thesis author. A proper case study with lots of participants would help to get a better understanding of the quality of the control modes.

Also, it would be interesting to do further evaluations on WMHNs, for example on the

OVGU-HC testbed once a sufficient number of nodes has been added. The effect of lots of hops between the haptics-brain and drone-controller component could be analyzed and may show if the implementation can be used over the internet without too much performance degradation, enabling very long range teleoperation.

While the Terrain-Aware Target Positioning control mode provides somewhat helpful haptic feedback, the modeling of the drone's environment by obtaining a point cloud with a LIDAR sensor could be improved. For example adding some kind of aggregating world construction that continuously adds LIDAR scans from different positions to a model of the world would enable a more complete and wider haptic rendering. Another follow-up idea would then be to add path finding capabilities – if you already have a solid long-range representation of the drone's environment, you might as well add path finding to it.

Last but not least, the demonstrator is open for research on other disciplines. The demonstrator can be used to work on artificial intelligence (e.g. neural networks, reinforcement learning) that could assist a human operator to control the drone, or even operate the drone autonomously.

Bibliography

[1] Stefan Lichiardopol. A survey on teleoperation. 2007.

[2] M. Hassanalian and A. Abdelkefi. Classifications, applications, and design challenges of drones: A review. *Progress in Aerospace Sciences*, 91:99–131, 2017.

[3] Jianguo Zhou, Jintao Yang, and Lu Lu. Research on multi-uav networks in disaster emergency communication. *IOP Conference Series: Materials Science and Engineering*, 719:012054, 01 2020.

[4] IHS. Internet of things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`, 2016. Accessed June 14, 2019.

[5] Frank Engelhardt, Johannes Behrens, and Mesut Güneş. The ovgu haptic communication testbed (ovgu-hc). In *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–6, 2020.

[6] Thomas Sheridan, W. Verplank, and T. Brooks. Human and computer control of undersea teleoperators. 12 1978.

[7] S. D. Laycock and A. M. Day. Recent developments and applications of haptic devices. *Computer Graphics Forum*, 22(2):117–132, 2003.

[8] 3D Systems Touch™ Haptic Device User Guide. `https://de.3dsystems.com/sites/default/files/2017-12/3DSystems-Touch-UserGuide.pdf`, 2017. Accessed April 1, 2022.

[9] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[10] Vasil Sarafov and Jan Seeger. Comparison of iot data protocol overhead. 2018.

[11] Borting Chen, Mesut Günes, and Yu-Lun Huang. Coap option for capability-based access control for iot-applications. In *IoTBD*, 2016.

[12] Marcin Odelga, Paolo Stegagno, Nicholas Kochanek, and Heinrich H. Bülthoff. A self-contained teleoperated quadrotor: On-board state-estimation and indoor obstacle avoidance. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7840–7847, 2018.

[13] Maik Riestock, Frank Engelhardt, Sebastian Zug, and Nico Hochgeschwender. User study on remotely controlled uavs with focus on interfaces and data link quality. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*,

pages 3394–3400, 2017.

[14] Vemema Kangunde, Rodrigo S Jamisola, and Emmanuel K Theophilus. A review on drones controlled in real-time. *International journal of dynamics and control*, 9(4):1832–1846, 2021.

[15] Jeroen G.W. Wildenbeest, David A. Abbink, Cock J.M. Heemskerk, Frans C.T. van der Helm, and Henri Boessenkool. The impact of haptic feedback quality on the performance of teleoperated assembly tasks. *IEEE Transactions on Haptics*, 6(2):242–252, 2013.

[16] Konstantinos Antonakoglou, Xiao Xu, Eckehard Steinbach, Toktam Mahmoodi, and Mischa Dohler. Toward haptic communications over the 5g tactile internet. *IEEE Communications Surveys Tutorials*, 20(4):3034–3059, 2018.

[17] P. Hinterseer, E. Steinbach, and S. Chaudhuri. Perception-based compression of haptic data streams using kalman filters. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 5, pages V–V, 2006.

[18] Peter Hinterseer, Sandra Hirche, Subhasis Chaudhuri, Eckehard Steinbach, and Martin Buss. Perception-based data reduction and transmission of haptic data in telepresence and teleaction systems. *IEEE Transactions on Signal Processing*, 56(2):588–597, 2008.

[19] Frank Engelhardt, Sophie Herbrechtsmeyer, and Mesut Güneş. Kinesthetic coding based on the fast wavelet transform for remote-controlling a quadrotor drone. In *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, pages 157–162, 2022.

[20] Eckehard Steinbach, Matti Strese, Mohamad Eid, Xun Liu, Amit Bhardwaj, Qian Liu, Mohammad Al-Ja'afreh, Toktam Mahmoodi, Rania Hassen, Abdulmotaleb El Saddik, and Oliver Holland. Haptic codecs for the tactile internet. *Proceedings of the IEEE*, 107(2):447–470, 2019.

[21] T. Mung Lam, Max Mulder, and M. M. van Paassen. Haptic feedback for uav teleoperation - force offset and spring load modification. In *2006 IEEE International Conference on Systems, Man and Cybernetics*, volume 2, pages 1618–1623, 2006.

[22] Carine Rognon, Margaret Koehler, Christian Duriez, Dario Floreano, and Allison M. Okamura. Soft haptic device to render the sensation of flying like a drone. *IEEE Robotics and Automation Letters*, 4(3):2524–2531, 2019.

[23] Roman Ibrahimov, Evgeny Tsykunov, Vladimir Shirokun, Andrey Somov, and Dzmitry Tsetserukou. Dronepick: Object picking and delivery teleoperation with the drone controlled by a wearable tactile display. In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 1–6, 2019.

[24] Matteo Macchini, Thomas Havy, Antoine Weber, Fabrizio Schiano, and Dario Floreano. Hand-worn haptic interface for drone teleoperation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10212–10218, 2020.

[25] Vivek Ramachandran, Matteo Macchini, and Dario Floreano. Arm-wrist haptic sleeve for drone teleoperation. *IEEE Robotics and Automation Letters*, pages 1–1, 2021.

[26] Jessie Y. C. Chen and Jennifer E. Thropp. Review of low frame rate effects on human performance. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems*

and Humans, 37(6):1063–1076, 2007.

[27] Y. Kuroki, T. Nishi, S. Kobayashi, H. Oyaizu, and S. Yoshimura. A psychophysical study of improvements in motion-image quality by using high frame rates. *Journal of the Society for Information Display*, 15(1):61–68, 2007.

[28] Are There Advantages to Frame Rates Higher Than the Refresh Rate? `https://blurbusters.com/faq/benefits-of-frame-rate-above-refresh-rate/`, August 2017. Accessed February 9, 2022.

[29] P. Hinterseer and E. Steinbach. A psychophysically motivated compression approach for 3d haptic data. In *2006 14th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 35–41, 2006.

[30] N. Fourty, T. Val, P. Fraisse, and J.-J. Mercier. Comparative analysis of new high data rate wireless communication technologies "from wi-fi to wimax". In *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services - (icas-isns'05)*, pages 66–66, 2005.

[31] Godfrey Anuga Akpakwu, Bruno J. Silva, Gerhard P. Hancke, and Adnan M. Abu-Mahfouz. A survey on 5g networks for the internet of things: Communication technologies and challenges. *IEEE Access*, 6:3619–3647, 2018.

[32] Arun Agarwal. Evolution of mobile communication technology towards 5g networks and challenges. 07 2019.

[33] Mobile Base Stations. `https://mobilenetworkguide.com.au/mobile_base_stations.html`. Accessed May 4, 2022.

[34] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[35] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[36] E. Rohmer, S. P. N. Singh, and M. Freese. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. www.coppeliarobotics.com.

[37] Conti, F. and Barbagli, F. and Balaniuk, R. and Halg, M. and Lu, C. and Morris, D. and Sentis, L. and Warren, J. and Khatib, and Salisbury, K. The chai libraries. In *Proceedings of Eurohaptics 2003*, pages 496–500, Dublin, Ireland, 2003.

[38] Diego C Ruspini, Krasimir Kolarov, and Oussama Khatib. The haptic display of complex graphical environments. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 345–352, 1997.

[39] Flatbuffers: C++ Benchmarks. `https://google.github.io/flatbuffers/flatbuffers_benchmarks.html`. Accessed April 1, 2022.

[40] Flight Controller (Autopilot) Hardware. `https://docs.px4.io/v1.12/en/flight_controller/`, March 2021. Accessed April 1, 2022.

[41] FlatBuffers: FlatBuffers. https://google.github.io/flatbuffers/. Accessed April 1, 2022.

[42] FlatBuffers: Writing a schema. https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html. Accessed April 24, 2022.

[43] FlatBuffers: Using the schema compiler. https://google.github.io/flatbuffers/flatbuffers_guide_using_schema_compiler.html. Accessed April 24, 2022.

[44] libcoap.net. https://libcoap.net/, 2021. Accessed April 24, 2022.

[45] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[46] rafat/wavelib: C Implementation of 1D and 2D Wavelet Transforms (DWT,SWT and MODWT) along with 1D Wavelet packet Transform and 1D Continuous Wavelet Transform. https://github.com/rafat/wavelib, August 2020. Accessed April 24, 2022.

[47] MAVLink Messaging. https://docs.px4.io/v1.12/en/middleware/mavlink.html, March 2020. Accessed April 19, 2022.

[48] AirLib on a Real Drone. https://microsoft.github.io/AirSim/custom_drone/, 2021. Accessed April 19, 2022.

[49] Determining yaw, pitch, and roll from a rotation matrix. http://planning.cs.uiuc.edu/node103.html, April 2012. Accessed May 6, 2022.

[50] Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. On fast surface reconstruction methods for large and noisy point clouds. In *2009 IEEE International Conference on Robotics and Automation*, pages 3218–3223, 2009.

[51] Dirk Holz and Sven Behnke. Fast range image segmentation and smoothing using approximate surface reconstruction and region growing. In Sukhan Lee, Hyungsuck Cho, Kwang-Joon Yoon, and Jangmyung Lee, editors, *Intelligent Autonomous Systems 12*, pages 61–73, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[52] GetSystemTimePreciseAsFileTime function (sysinfoapi.h) - Win32 apps | Microsoft Docs. https://docs.microsoft.com/de-de/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemtimepreciseasfiletime, October 2021. Accessed April 21, 2022.

Appendix

A.1 Pseudocodes for typical CHAI3D update loops

```
// create haptic device handler
cHapticDeviceHandler* handler = new cHapticDeviceHandler();

// get handle to first found haptic device
cGenericHapticDevice* device;
handler->getDevice(device, 0);

// connect to haptic device
device->open();

// haptics loop
while (running) {
  // read position from haptic device
  cVector3d position;
  device->getPosition(position);

  // compute force
  cVector3d force = -25 * position;

  // send force to haptic device
  device->setForce(force);
}

// close connection to haptic device
device->close();
```

Listing A.1: Pseudocode of a typical loop updating the force feedback of the haptic device, using explicit force vector.

```
// create haptic device handler
cHapticDeviceHandler* handler = new cHapticDeviceHandler();

// get handle to first found haptic device
cGenericHapticDevice* device;
handler->getDevice(device, 0);

// connect to haptic device
device->open();

// create virtual 3D world
cWorld* world = new cWorld();

// populate world
world->addChild(new cShapeCylinder(0.25, 0.25, 0.2));
```

```
[...]

// create tool representing the tip of the haptic tool
cToolCursor* tool;
tool = new cToolCursor(world);
world->addChild(tool);

// attach haptic device to virtual tool
tool->setHapticDevice(device);

// haptics loop
while (running) {
    // compute global reference frames for each object
    world->computeGlobalPositions(true);

    // update position and orientation of tool
    tool->updateFromDevice();

    // compute interaction forces
    tool->computeInteractionForces();

    // send forces to haptic device
    tool->applyToDevice();
}

// close connection to haptic device
device->close();
```

Listing A.2: Pseudocode of a typical loop updating the force feedback of the haptic device, using `cWorld` abstraction.

A.2 AirSim configuration file with default values

```
{
  "SimMode": "",
  "ClockType": "",
  "ClockSpeed": 1,
  "LocalHostIp": "127.0.0.1",
  "ApiServerPort": 41451,
  "RecordUIVisible": true,
  "LogMessagesVisible": true,
  "ViewMode": "",
  "RpcEnabled": true,
  "EngineSound": true,
  "PhysicsEngineName": "",
  "SpeedUnitFactor": 1.0,
  "SpeedUnitLabel": "m/s",
  "Wind": { "X": 0, "Y": 0, "Z": 0 },
  "CameraDirector": {
    "FollowDistance": -3,
    "X": NaN, "Y": NaN, "Z": NaN,
    "Pitch": NaN, "Roll": NaN, "Yaw": NaN
  },
  "Recording": {
    "RecordOnMove": false,
    "RecordInterval": 0.05,
    "Folder": "",
    "Enabled": false,
    "Cameras": [
        { "CameraName": "0", "ImageType": 0, "PixelsAsFloat": false,  "
            VehicleName": "", "Compress": true }
    ]
  },
  "CameraDefaults": {
    "CaptureSettings": [
      {
        "ImageType": 0,
        "Width": 256,
        "Height": 144,
        "FOV_Degrees": 90,
        "AutoExposureSpeed": 100,
        "AutoExposureBias": 0,
        "AutoExposureMaxBrightness": 0.64,
        "AutoExposureMinBrightness": 0.03,
        "MotionBlurAmount": 0,
        "TargetGamma": 1.0,
        "ProjectionMode": "",
        "OrthoWidth": 5.12
      }
    ],
    "NoiseSettings": [
      {
        "Enabled": false,
        "ImageType": 0,

        "RandContrib": 0.2,
        "RandSpeed": 100000.0,
        "RandSize": 500.0,
        "RandDensity": 2,

        "HorzWaveContrib":0.03,
        "HorzWaveStrength": 0.08,
        "HorzWaveVertSize": 1.0,
        "HorzWaveScreenSize": 1.0,

        "HorzNoiseLinesContrib": 1.0,
        "HorzNoiseLinesDensityY": 0.01,
        "HorzNoiseLinesDensityXY": 0.5,

        "HorzDistortionContrib": 1.0,
        "HorzDistortionStrength": 0.002
      }
    ],
    "Gimbal": {
      "Stabilization": 0,
      "Pitch": NaN, "Roll": NaN, "Yaw": NaN
    },
    "X": NaN, "Y": NaN, "Z": NaN,
    "Pitch": NaN, "Roll": NaN, "Yaw": NaN
  },
  "OriginGeopoint": {
    "Latitude": 47.641468,
    "Longitude": -122.140165,
    "Altitude": 122
  },
  "TimeOfDay": {
    "Enabled": false,
    "StartDateTime": "",
    "CelestialClockSpeed": 1,
    "StartDateTimeDst": false,
    "UpdateIntervalSecs": 60
  },
  "SubWindows": [
    { "WindowID": 0, "CameraName": "0", "ImageType": 3, "VehicleName": "", "
        Visible": false },
    { "WindowID": 1, "CameraName": "0", "ImageType": 5, "VehicleName": "", "
        Visible": false },
    { "WindowID": 2, "CameraName": "0", "ImageType": 0, "VehicleName": "", "
        Visible": false }
  ],
  "SegmentationSettings": {
    "InitMethod": "",
    "MeshNamingMethod": "",
    "OverrideExisting": false
  },
  "PawnPaths": {
    "BareboneCar": { "PawnBP": "Class'/AirSim/VehicleAdv/Vehicle/VehicleAdvPawn.
        VehicleAdvPawn_C'" },
    "DefaultCar": { "PawnBP": "Class'/AirSim/VehicleAdv/SUV/SuvCarPawn.
        SuvCarPawn_C'" },
    "DefaultQuadrotor": { "PawnBP": "Class'/AirSim/Blueprints/BP_FlyingPawn.
        BP_FlyingPawn_C'" },
    "DefaultComputerVision": { "PawnBP": "Class'/AirSim/Blueprints/
        BP_ComputerVisionPawn.BP_ComputerVisionPawn_C'" }
  },
  "Vehicles": {
    "SimpleFlight": {
      "VehicleType": "SimpleFlight",
      "DefaultVehicleState": "Armed",
      "AutoCreate": true,
      "PawnPath": "",
      "EnableCollisionPassthrogh": false,
      "EnableCollisions": true,
      "AllowAPIAlways": true,
      "EnableTrace": false,
      "RC": {
        "RemoteControlID": 0,
        "AllowAPIWhenDisconnected": false
      },
      "Cameras": {
        // same elements as CameraDefaults above, key as name
      },
      "X": NaN, "Y": NaN, "Z": NaN,
      "Pitch": NaN, "Roll": NaN, "Yaw": NaN
    },
    "PhysXCar": {
      "VehicleType": "PhysXCar",
      "DefaultVehicleState": "",
      "AutoCreate": true,
      "PawnPath": "",
      "EnableCollisionPassthrogh": false,
      "EnableCollisions": true,
      "RC": {
        "RemoteControlID": -1
      },
      "Cameras": {
        "MyCamera1": {
          // same elements as elements inside CameraDefaults above
        },
        "MyCamera2": {
          // same elements as elements inside CameraDefaults above
        },
      },
      "X": NaN, "Y": NaN, "Z": NaN,
      "Pitch": NaN, "Roll": NaN, "Yaw": NaN
    }
  }
}
```

Listing A.3: AirSim configuration file with default values.

A.3 Example of libcoap server and client implementation

```c
#include <stdio.h>
#include <coap3/coap.h>

static uint8_t i = 0;

void handleRequest(coap_resource_t* resource, coap_session_t* session, const coap_pdu_t* request, const coap_string_t* token, coap_pdu_t* response) {
  int ec;
  size_t len;
  uint8_t* data;

  ec = coap_get_data(request, &len, (const uint8_t**) &data);
  if (ec != 1) {
    coap_pdu_set_code(response, COAP_RESPONSE_CODE_NOT_ACCEPTABLE);
    return;
  }

  coap_show_pdu(LOG_INFO, request);

  ec = coap_add_data(response, sizeof(uint8_t), &i);
  if (!ec) {
    printf("Could not add data to CoAP PDU!\n");
    coap_pdu_set_code(response, COAP_RESPONSE_CODE_INTERNAL_ERROR);
    return;
  }

  coap_pdu_set_code(response, COAP_RESPONSE_CODE_CHANGED);

  ++i;
}

int main(int argc, char const *argv[]) {
  int ec;
  coap_context_t* context;
  coap_address_t address;
  coap_endpoint_t* endpoint;
  coap_resource_t* resource;

  coap_startup();
  coap_set_log_level(LOG_DEBUG);

  ec = resolveAddress("127.0.0.1", "5683", &address);
  /* `resolveAddress` is not implemented here as it is specific to the operating system,
   * but this is a function that basically fills the coap_address_t struct
   */
  if (ec < 0) {
    printf("Could not resolve bind address!\n");
    return 1;
  }

  context = coap_new_context(NULL);
  if (!context) {
    printf("Could not create CoAP context!\n");
    return 1;
  }

  endpoint = coap_new_endpoint(context, &address, COAP_PROTO_UDP);
  if (!endpoint) {
    printf("Could not create CoAP endpoint!\n");
    return 1;
  }

  resource = coap_resource_init(coap_make_str_const(""), 0);
  coap_register_handler(resource, COAP_REQUEST_PUT, handleRequest);
  coap_add_resource(context, resource);

  while (1) {
    coap_io_process(context, COAP_IO_WAIT);
  }

  return 0;
}
```

Listing A.4: Minimal CoAP server implementation using libcoap to serve an integer on root path `/`.

```c
#include <stdio.h>
#include <string.h>
#include <coap3/coap.h>

coap_response_t handleResponse(coap_session_t *session, const coap_pdu_t *sent,
     const coap_pdu_t *received, const coap_mid_t mid) {
  int ec;
  size_t len;
  uint8_t* data;

  coap_show_pdu(LOG_INFO, received);

  int* ack_received = (int*) coap_session_get_app_data(session);
  *ack_received = 1;

  return COAP_RESPONSE_OK;
}

int main(int argc, char const *argv[]) {
  int ec;
  int rc = 0;
  coap_context_t* context = NULL;
  int ack_received;
  coap_address_t address;
  coap_session_t* session = NULL;
  coap_pdu_t* pdu;

  coap_startup();
  coap_set_log_level(LOG_DEBUG);

  ec = resolveAddress("127.0.0.1", "5683", &address);
  /* `resolveAddress` is not implemented here as it is specific to the operating
      system,
   * but this is a function that basically fills the coap_address_t struct
   */
  if (ec < 0) {
    printf("Could not resolve remote address!\n");

    rc = 1;
    goto cleanup;
  }

  context = coap_new_context(NULL);
  if (!context) {
    printf("Could not create CoAP context!\n");

    rc = 1;
    goto cleanup;
  }

  coap_register_response_handler(context, handleResponse);

  session = coap_new_client_session(context, NULL, &address, COAP_PROTO_UDP);
  if (!session) {
    printf("Could not create CoAP session!\n");

    rc = 1;
    goto cleanup;
  }

  coap_session_set_app_data(session, &ack_received);

  for (int i = 0; i < 10; ++i) {
    pdu = coap_pdu_init(COAP_MESSAGE_CON, COAP_REQUEST_CODE_PUT,
        coap_new_message_id(session), coap_session_max_pdu_size(session));
    if (!pdu) {
      printf("Could not create CoAP PDU!\n");

      rc = 1;
      goto cleanup;
    }

    uint8_t token[8];
    size_t token_len;
    coap_session_new_token(session, &token_len, token);
    coap_add_token(pdu, token_len, token);

    // dummy data
    uint8_t data[16];
    const size_t data_len = sizeof(data) / sizeof(data[0]);
    memset(data, i, data_len);

    ec = coap_add_data(pdu, len, data);
    if (!ec) {
      printf("Could not add data to CoAP PDU!\n");

      rc = 1;
      goto cleanup;
    }

    ack_received = 0;
    coap_send(session, pdu);
    while (!ack_received) coap_io_process(context, COAP_IO_WAIT);
  }

cleanup:
  if (session != NULL) coap_session_release(session);
  if (context != NULL) coap_free_context(context);
  coap_cleanup();

  return rc;
}
```

Listing A.5: Minimal CoAP client implementation using libcoap to send ten CON requests to root path `/`.

A.4 UML class diagrams for all control modes

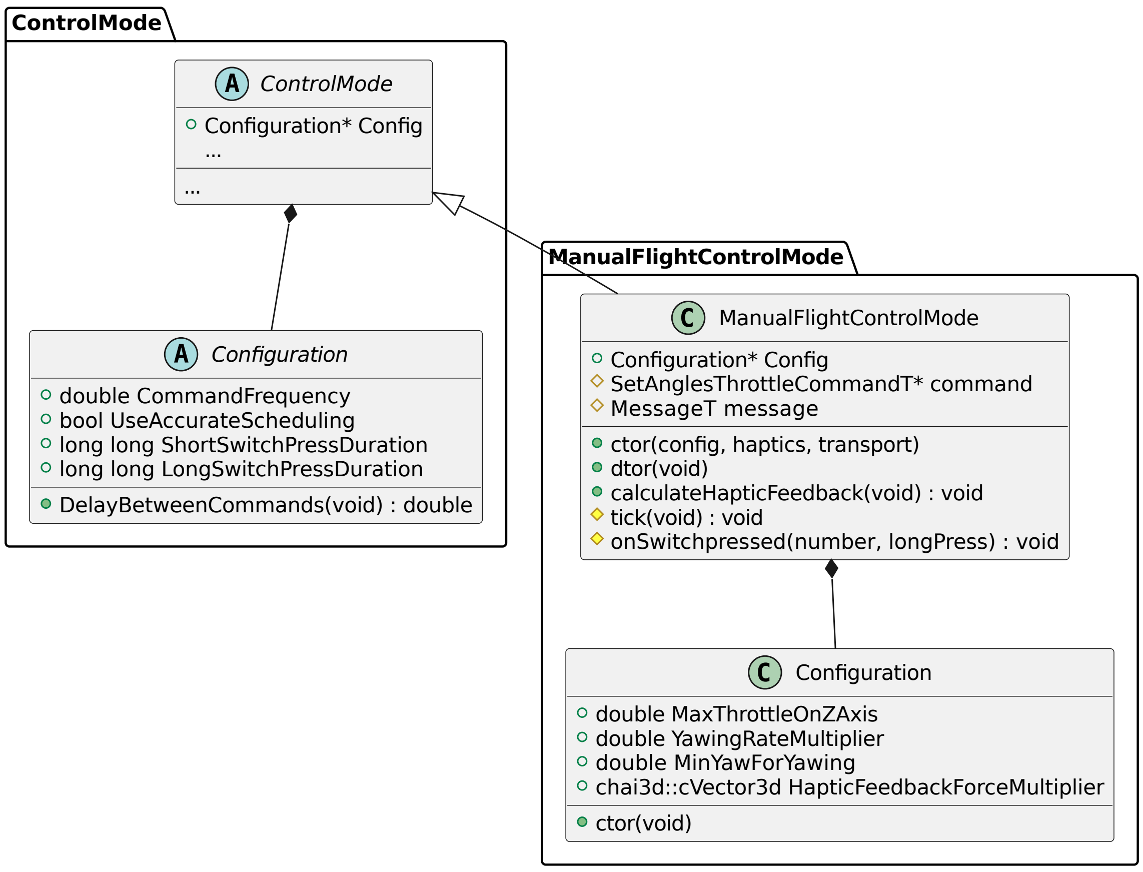

ControlMode
A ControlMode
Configuration* Config
...
...
A Configuration
double CommandFrequency
bool UseAccurateScheduling
long long ShortSwitchPressDuration
long long LongSwitchPressDuration
DelayBetweenCommands(void) : double
ManualFlightControlMode
C ManualFlightControlMode
Configuration* Config
SetAnglesThrottleCommandT* command
MessageT message
ctor(config, haptics, transport)
dtor(void)
calculateHapticFeedback(void) : void
tick(void) : void
onSwitchpressed(number, longPress) : void
C Configuration
double MaxThrottleOnZAxis
double YawingRateMultiplier
double MinYawForYawing
chai3d::cVector3d HapticFeedbackForceMultiplier
ctor(void)

Figure A.1: UML class diagram showing fields and methods of the class that implements the Manual Flight control mode.

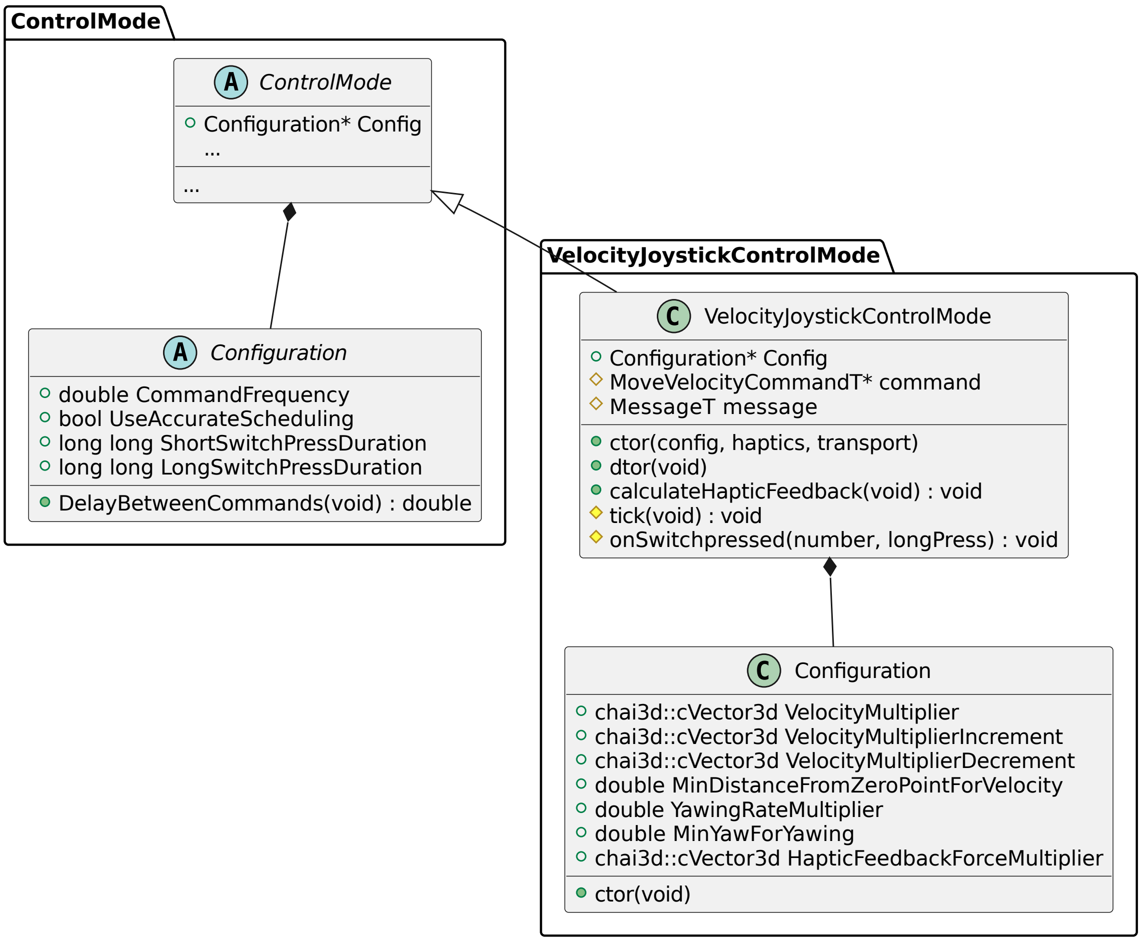
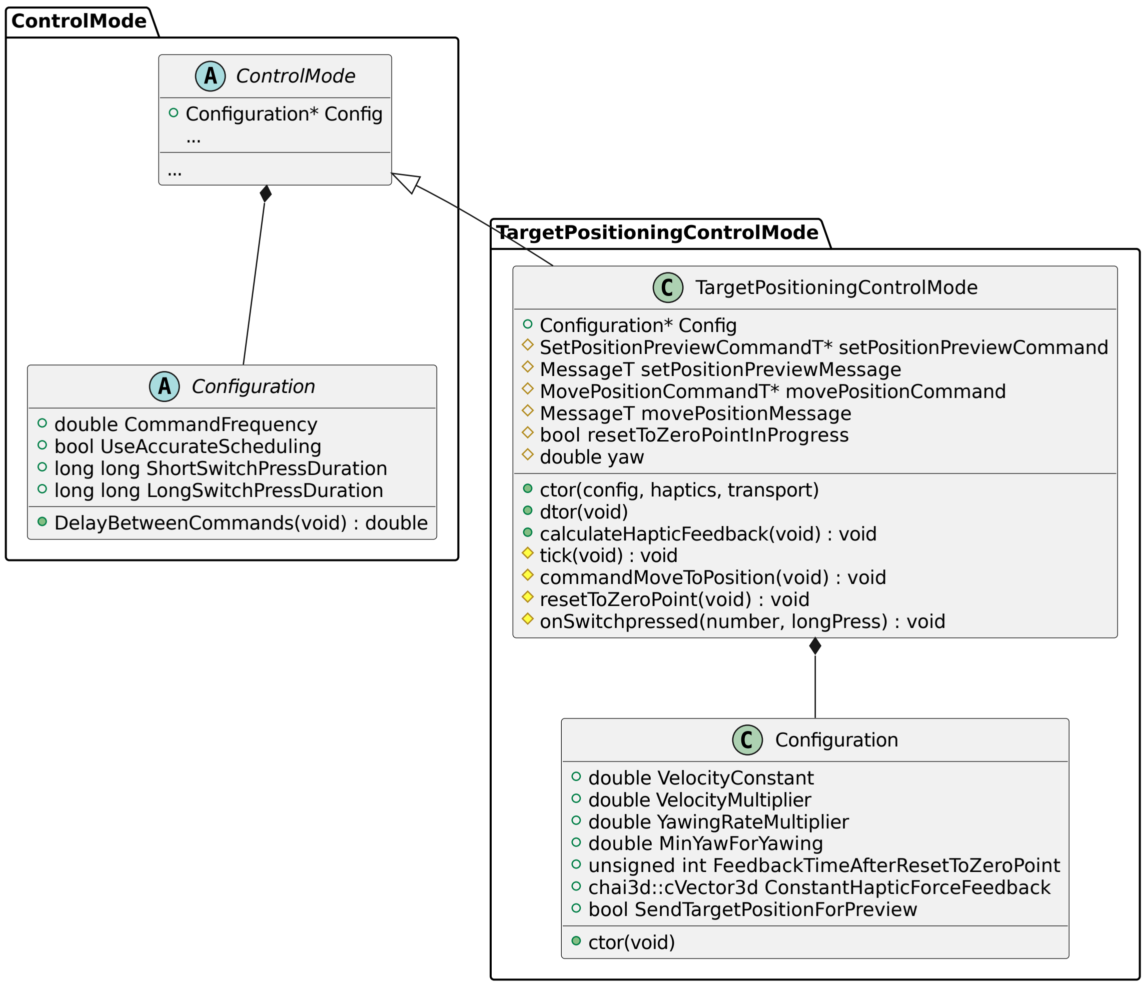

ControlMode
A ControlMode
○ Configuration* Config
...
...
A Configuration
○ double CommandFrequency
○ bool UseAccurateScheduling
○ long long ShortSwitchPressDuration
○ long long LongSwitchPressDuration
● DelayBetweenCommands(void) : double
VelocityJoystickControlMode
C VelocityJoystickControlMode
○ Configuration* Config
◇ MoveVelocityCommandT* command
◇ MessageT message
● ctor(config, haptics, transport)
● dtor(void)
● calculateHapticFeedback(void) : void
◇ tick(void) : void
◇ onSwitchpressed(number, longPress) : void
C Configuration
○ chai3d::cVector3d VelocityMultiplier
○ chai3d::cVector3d VelocityMultiplierIncrement
○ chai3d::cVector3d VelocityMultiplierDecrement
○ double MinDistanceFromZeroPointForVelocity
○ double YawingRateMultiplier
○ double MinYawForYawing
○ chai3d::cVector3d HapticFeedbackForceMultiplier
● ctor(void)

Figure A.2: UML class diagram showing fields and methods of the class that implements the Velocity Joystick control mode.



ControlMode
A ControlMode
Configuration* Config
...
...
A Configuration
double CommandFrequency
bool UseAccurateScheduling
long long ShortSwitchPressDuration
long long LongSwitchPressDuration
DelayBetweenCommands(void) : double
TargetPositioningControlMode
C TargetPositioningControlMode
Configuration* Config
SetPositionPreviewCommandT* setPositionPreviewCommand
MessageT setPositionPreviewMessage
MovePositionCommandT* movePositionCommand
MessageT movePositionMessage
bool resetToZeroPointInProgress
double yaw
ctor(config, haptics, transport)
dtor(void)
calculateHapticFeedback(void) : void
tick(void) : void
commandMoveToPosition(void) : void
resetToZeroPoint(void) : void
onSwitchpressed(number, longPress) : void
C Configuration
double VelocityConstant
double VelocityMultiplier
double YawingRateMultiplier
double MinYawForYawing
unsigned int FeedbackTimeAfterResetToZeroPoint
chai3d::cVector3d ConstantHapticForceFeedback
bool SendTargetPositionForPreview
ctor(void)

Figure A.3: UML class diagram showing fields and methods of the class that implements the Target Positioning control mode.

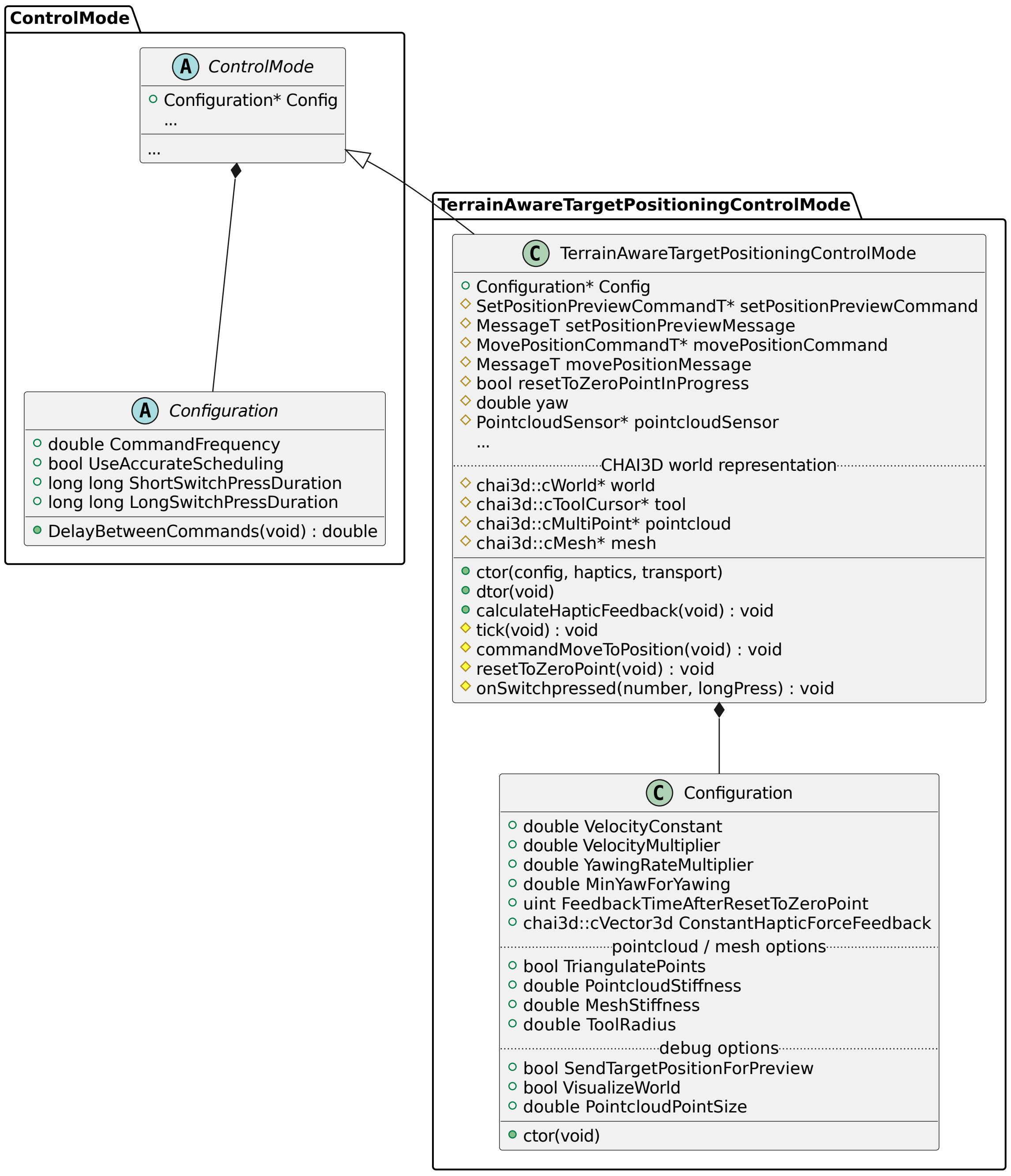

ControlMode
A ControlMode
○ Configuration* Config
...
...
A Configuration
○ double CommandFrequency
○ bool UseAccurateScheduling
○ long long ShortSwitchPressDuration
○ long long LongSwitchPressDuration
● DelayBetweenCommands(void) : double
TerrainAwareTargetPositioningControlMode
C TerrainAwareTargetPositioningControlMode
○ Configuration* Config
◇ SetPositionPreviewCommandT* setPositionPreviewCommand
◇ MessageT setPositionPreviewMessage
◇ MovePositionCommandT* movePositionCommand
◇ MessageT movePositionMessage
◇ bool resetToZeroPointInProgress
◇ double yaw
◇ PointcloudSensor* pointcloudSensor
...
CHAI3D world representation
◇ chai3d::cWorld* world
◇ chai3d::cToolCursor* tool
◇ chai3d::cMultiPoint* pointcloud
◇ chai3d::cMesh* mesh
● ctor(config, haptics, transport)
● dtor(void)
● calculateHapticFeedback(void) : void
◆ tick(void) : void
◆ commandMoveToPosition(void) : void
◆ resetToZeroPoint(void) : void
◆ onSwitchpressed(number, longPress) : void
C Configuration
○ double VelocityConstant
○ double VelocityMultiplier
○ double YawingRateMultiplier
○ double MinYawForYawing
○ uint FeedbackTimeAfterResetToZeroPoint
○ chai3d::cVector3d ConstantHapticForceFeedback
pointcloud / mesh options
○ bool TriangulatePoints
○ double PointcloudStiffness
○ double MeshStiffness
○ double ToolRadius
debug options
○ bool SendTargetPositionForPreview
○ bool VisualizeWorld
○ double PointcloudPointSize
● ctor(void)

Figure A.4: UML class diagram showing fields and methods of the class that implements the Terrain-Aware Target Positioning control mode.

I herewith assure that I wrote the present thesis titled *Teleoperation of Quadrotor Drones using Haptic Devices* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, November 10, 2022

(Jon-Mailes Graeffe)