Norwegian University of
Science and Technology

Remote Operations using Oculus Rift, Leap Motion and Microsoft Kinect

Ole Magnus Siqveland

Master of Science in Cybernetics and Robotics
Submission date: June 2016
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This thesis and all its related work has been conducted during the 4'th and final semester of the two year master program in Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU). The assignment counts for 30 credits and is carried out at the Department of Engineering Cybernetics during the spring semester of 2016.

The idea of the project originates from my time working as an automation engineer at an industrial automation company. The possibility of performing simple operations at remote locations, without having to be physically present, would be very appealing and practical in many situations during my work. A couple of years later, here I am creating a prototype solution using virtual reality equipment. Hopefully, some of my findings during this semester will prove useful for further development of such systems.

It is assumed that readers of this thesis have a background in technology and a fair understanding of programming and electronics. Experience within rendering, kinematics and virtual reality is an advantage.

Trondheim, 2016-06-20

Ole Magnus Siqveland

Ole Magnus Siqveland

Acknowledgment

I would like to thank my supervisor Professor Tor E. Onshus for good guidance and followup throughout the work on this project. The meetings and discussions we had greatly helped with motivation and maintaining focus on the tasks at hand. I would also like to thank Terje Haugen and Daniel Bogen at the department's workshop for their quick response and excellent quality of requested robot enhancements and repairs.

Fellow student Vegard Lindrup worked on the same robot system as me during the semester, with his own project. I would like to express my gratitude for his cooperativeness and flexibility in regards to sharing equipment. Furtermore, thanks to fellow student Ulrik Utheim Sverdrup for helping me film the demonstration video.

Finally, I am grateful for all support and motivation provided during the semester by my girlfriend, who also gave birth to our beautiful daughter during the final month of the thesis. Her persistence during pregnancy and as a mother greatly helped me maintain focus on my work.

O.M.S.

Summary and Conclusions

Summary in Norwegian

Denne oppgaven utforsker bruken av virtuell virkelighet teknologi som middel for å observere og samhandle med vår fysiske verden via en ekstern robot. En ny tilnærming for å utføre fjernoperasjoner er utviklet i løpet av dette prosjektet, med formål om å gi operatøren en oppslukende følelse av tilstedeværelse på robotens sted.

Et prototype-system er blitt utviklet som gjør det mulig for en operatør å fjernstyre en robot, som består av en manipulator som er montert på en kjørbar plattform, ved hjelp av posisjonen til sin venstre hånd i kombinasjon med en joystick. Venstre posisjon er innhentet ved hjelp av en liten infrarød-kamera enhet montert i fronten på en hodemontert skjerm, som skal anvendes av operatøren. Observasjon fra robotens perspektiv oppnås ved å strømme video fra kameraer montert på roboten til den hodemonterte skjermen. Kameraene følger retningen til operatørens hode ved å knytte en Pan Tilt enhet til sporings sensorene i den hodemonterte skjermen. I tillegg til videostrømmer, er 3D-modeller som representerer ulike attributter av systemet rendret til skjermen ved hjelp av OpenGL, slik at operatøren får viktig informasjon og tilbakemeldinger under drift.

Systemet er delt inn i en klient og en server applikasjon, som kommuniserer trådløst via en nettverkstilkobling. Serveren er montert på roboten, og eksponerer dens påmonterte enheter til klient programmet, mens klienten kjøres på en stasjonær maskin som da skal brukes av operatøren. Et bil batteriet brukes som strømkilde for roboten for å gjøre den fullstendig trådløs.

Joysticken er utpekt som den overordnede styringen av systemet, og er ment å manøvreres av operatørens høyre hånd. Knappene brukes til å velge mellom manipulator moduser og aktivering av ulike funksjoner, mens selve pinnen kan manøvreres for å kjøre roboten. To manipulator moduser er implementert, og de kalles for Direct Mode og Record Mode. Begge modusene baserer seg på posisjonen til operatørens venstre hånd for manipulator bevegelse.

Direct Mode gjør det mulig å styre manipulatoren i sanntid. Enhver bevegelse med venstre hånden utført foran den hodemonterte skjermen resulterer i tilsvarende bevegelse av manipulatoren. Håndposisjoner blir kontinuerlig registrert, og ved hjelp av invers kinematikk finnes de tilsvarende vinklene som brukes som sett punkt for manipulatorens ulike ledd. I tillegg er en virtuell representasjon av manipulatoren laget for visning på skjermen, som da gir umiddelbar tilbakemelding til operatøren på hvordan hans håndbevegelser oversettes til manipulator posisjoner.

Record Mode gjør det mulig for operatøren å først registrere en sekvens av stillinger ved hjelp av sin venstre hånd, før hele sekvensen utføres av manipulatoren. Et 3D-kamera montert i front av roboten gir et punkt sky representasjon av manipulatorens arbeidsområde. I Record Mode blir punkt skyen gjengitt i den hodemonterte skjermen sammen med en virtuell representasjon av manipulatoren som følger bevegelsen av operatørens venstre hånd i sanntid. Den virtuelle manipulator og punkt skyen er plassert i forhold til hverandre, slik at det ser ut som om den virtuelle manipulatoren opererer i det fysiske miljøet. Mens operatøren manøvrerer den virtuelle roboten i punkt skyen, kan han registrere posisjoner i en sekvens. Sekvensen kan deretter bli eksekvert på den fysiske manipulatoren, slik at den utfører de samme handlingene som registrert virtuelt, i den virkelige verden. Altså gjør det føreren i stand til å planlegge og registrere sine fysiske manipulator handlinger i en virtuell representasjon før fysisk utførelse.

Summary

This thesis explores the usage of virtual reality technology as means of observing and interacting with our physical world through a remotely located robot. Proof of concept for a novel approach of performing remote operations is developed during the course of this project, aiming to give the operator an immersive feeling of presence at the robots location.

A prototype system has been developed which enables an operator to remotely control a robot, consisting of a manipulator mounted on a drivable platform, by using the position of his left hand in combination with a joystick. Left hand position is acquired by using a small

infrared-camera device mounted in front of a head mounted display, to be worn by the operator. Observation from the robot's perspective is achieved by streaming video from on-board cameras to the head mounted display. The cameras follow the head direction of the operator by linking a pan tilt unit with tracking sensors of the display. In addition to video streams, 3D models representing various attributes of the system is rendered to the display by using OpenGL, providing the operator with vital information and feedback during operation.

The system is divided into a client and server application, which communicate wireless through a network connection. The server is installed on the robot, exposing its on-board devices, while the client is installed on a desktop machine to be used by an operator. A car battery is used as a power source for the robot in order to make it completely wireless.

The joystick is designated as the master controller of the system, and is intended to be maneuvered by the right hand of the operator. Its buttons are used to select between modes of manipulator operation and toggling of various features, while the stick itself may be moved in order to drive the robot. Two modes of manipulator operation have been implemented, termed Direct Mode and Record Mode. Both of which base manipulator movement on the operators left hand position.

Direct Mode enables the operator to control the manipulator in real-time. Any left hand movement performed in front of the head mounted display, result in corresponding movement of the manipulator. Hand positions are continually registered, and by using inverse kinematics the corresponding joint values are derived and used as target angles for the manipulator. In addition, a virtual representation of the manipulator is rendered to the display providing instant feedback to the operator on how his hand movements translates to manipulator movement.

Record Mode enables the operator to first register a sequence of positions using his left hand, before executing the entire sequence on the manipulator. A consumer grade 3D camera mounted in front of the manipulator provides a point cloud representation of the manipulators work area. During Record Mode, the point cloud is rendered into the head mounted display

together with a virtual representation of the manipulator which follows the movement of the operator's left hand in real-time. The virtual manipulator and point cloud are positioned relative to each other, such that it looks as if the virtual manipulator operates in the physical environment. While maneuvering the virtual robot in the point cloud, its positions may be recorded into a sequence. The sequence can then be executed to make the physical manipulator traverse the same registered positions in the real-world, interacting with its work area as instructed. Hence, it enables the operator to plan and record his physical manipulator action in a virtual representation.

Conclusion

The project culminated in a system which achieves all objectives set for the project. The system performs well under certain circumstances, and enables an operator to control the robotic manipulator with his left hand, while controlling position of the robot itself with a right handed joystick. Observing from the robots point of view by using the Oculus Rift works very well. The stereo view generated by the two cameras provides a sense of depth vision, and the 3D models rendered provide accurate and easy to understand information. When putting on the Oculus Rift it is as if entering an encapsulating control room, feeling to some degree present at the robots location.

Using the left hand to control the position of the manipulator turned out to be a relatively fast and intuitive way of maneuvering the manipulator, compared to traditional single-axis joystick control. However, it is not suitable for operations requiring pinpoint accuracy. The reason being both due to equipment and human constraints, such as inevitable fatigue and hand tremors.

Due to built-in constraints introduced by the robotic manipulator it suffers from large delays, rendering the usage of Direct Mode and real-time manipulator operation very difficult. Hence, Direct Mode became more of a convenient way to change the pose of the elbow mounted cameras to observe the environments from different points of view. Both manipulator modes, however, require a steady hand by the operator and some degree of motion prediction.

Record Mode became the easiest mode to use in practice for interactions with real-world objects. Due to all manipulator motion being planned in advance it is not dependant on rapid response from the manipulator, but a static environment is required. The mode is fairly accurate, however, it suffers from constraints introduced by the Kinect. The Kinect is unable to observe objects within the immediate work area of the manipulator, because of its minimum distance requirement. Thus, when a sequence is registered the robot has to be driven into the point cloud area before the manipulator is engaged. The operator must therefore be certain that the robot can drive straight forward into the point cloud area without hitting anything on its way. Another limitation is the resolution of the Kinect, limiting the amount of details observable in the point cloud.

Overall, the system works as a proof of concept demonstration, but is not a viable solution in practice because of the aforementioned constraints. By investing in better equipment eliminating these constraints, the system and both modes of operation could serve as usable control alternatives for remote operations, where pinpoint accuracy is not a requirement.

Contents

Preface i
Acknowledgment ii
Summary and Conclusions iii

1 Introduction **2**

1.1 Background 2
1.1.1 Motivation 5
1.1.2 Previous Work 5
1.2 Objectives 8
1.3 Limitations 9
1.4 Equipment 9
1.4.1 Oculus Rift DK2 9
1.4.2 Leap Motion 10
1.4.3 Joystick 12
1.4.4 The Robot 13
Intelitek Scorbot ER4u 14
Movable platform 15
Microsoft Kinect for Xbox 360 17
Pan Tilt Unit 18
IP Cameras 19
1.4.5 Software 19

2 Theory **21**

2.1 Linear Transformation Matrices 21

2.1.1 The Rotation Matrix 21
2.1.2 The Translation Matrix 23
2.1.3 The Scaling Matrix 23
2.2 Kinematics 24
2.2.1 Forward kinematics 25
2.2.2 Inverse Kinematics 26
2.3 Network Communication 30
2.3.1 User Datagram Protocol (UDP) 31
2.4 Oculus Rift and Lens Magnification 33
2.4.1 Pincushion Distortion 34
2.4.2 Chromatic Aberration 34
2.4.3 Time Warp 35
2.5 Open Graphics Library - OpenGL 36
2.5.1 Rendering Pipeline 37
2.5.2 The View Matrix 42
2.5.3 The Projection Matrix 43
2.5.4 Shapes 44
2.5.5 Text Rendering 45

3 Realization **46**
3.1 The Server Application 47
3.1.1 Controlling the Robotic Manipulator 49
3.1.2 Serial Communication 51
Driving the Robot 51
The Pan Tilt Unit 53
3.1.3 Monitoring Battery Voltage 54
3.1.4 Kinect Implementation 55
3.2 The Client Application 56
3.2.1 The Heads Up Display 59
Introducing Models 59

Geometric Shapes 61
3D Models 66
Text Rendering 71
3.2.2 The Rendering Loop 71
3.2.3 Rendering the Kinect Point Cloud 73
Calibration 74
Compensating for the Oculus Lenses 77
3.2.4 Leap Motion Implementation 78
Compensating for Head Rotations 79
3.2.5 Joystick Implementation 81
3.3 Communication 83
3.3.1 Client to Server 84
3.3.2 Server to Client 85

4 Results 88
4.1 Remote Control 88
4.1.1 Leap Motion In Action 88
4.1.2 Direct Mode 91
4.1.3 Record Mode 93
4.1.4 Driving 94
4.2 User Interface 95

5 Summary 100
5.1 Discussion 100
5.1.1 Challenges of Direct Control 101
5.1.2 Fragility of Record Mode 101
5.1.3 Lack of Documentation for the Scorbot ER4u 102
5.1.4 Working with Products Under Development 103
5.1.5 Upgrading to the latest Oculus Rift SDK 104
5.1.6 Using the Simple and Fast Multimedia Library (SFML) 105
5.1.7 Network Communication 106

5.1.8 Broken Encoders 106
5.2 Recommendations for Further Work 107
5.2.1 Building a New Controller for the Scorbot 107
5.2.2 Upgrading Outdated Equipment 108
5.2.3 Extending Battery Life 109
5.2.4 Reducing Webcamera Delay 110
5.2.5 Combining Manipulator Movement with the Oculus Rift 111
5.2.6 Adding Support for Object Files 111
5.2.7 Implementing Bi-Directional Audio 113

A Acronyms **114**

B Joystick Configuration **116**

C DVD Contents **117**

D Scorbot ER4u Function Reference **118**

E Installation Instructions **122**
E.1 The Robot 122
E.1.1 Wiring 122
E.1.2 The Server Computer 123
E.1.3 Setup 123
E.2 The Network 124
E.3 The Client Computer 125
E.3.1 Setup 126

F System Interconnections **127**

Bibliography **128**

Chapter 1

Introduction

This thesis is based on previous work done by the undersigned, during the autumn project of 2015 "Remote Vision Using Oculus Rift" [50]. A telepresence system using the Oculus Rift and two Internet Protocol (IP) cameras mounted on top of a Pan-Tilt unit was developed, and it is this system that has been improved, reworked and extended in order to incorporate robot control and supplementing visualizations. Ultimately, the goal has been to remote control a robotic manipulator and the drivable platform on which the manipulator is mounted, through a computer network. Control is to be achieved by using the operators left hand position for the manipulator and a joystick for driving the robot. All while observing from the robots point of view by combining the Oculus Rift and remote cameras.

1.1 Background

Virtual reality (VR) technologies are on the verge of becoming mainstream, and it is estimated that 2016 will be remembered as the year of virtual reality. Big companies like Facebook (owner of Oculus VR), Sony, Google, Microsoft, Valve and HTC are all working on virtual reality headsets and / or augmented reality headsets and related content, investing billions of dollars into the technology.

The concept of virtual reality is not new, and is believed to be first described by the science fiction writer Stanley G. Weinbaum in 1935 [14]. Throughout history many companies have tried to create applicable VR solutions without memorable success, mainly due to the lack of power in the underlying technology within an acceptable price range. However, it seems this history of failed products is about to come to its end [46] [45].

Oculus VR is often credited for the sudden spark of interest in VR. They launched a remarkably successful crowd-funding campaign on the first of August 2012, surpassing their funding goal with a substantial amount in less than 24 hours [37]. This lead to production and distribution of a prototype of the headset, later named the Oculus Rift Developer Kit (DK) 1, which received an overwhelmingly positive response. The success was noticed by other large tech companies, who are now developing alternative and competing VR headsets. First off was Sony, which are working on their own Playstation VR - a VR headset made exclusively for the Playstation 4 [10] - followed by Valve (working in collaboration with HTC) and Google. Facebook also took notice in the massive success of the Oculus Rift and in 2014 they bought Oculus VR for 2 billion dollars, stating that their intention is not to interfere with Oculus VR's focus on the gaming industry, but rather help accelerate their development [43]. At the time of writing, Oculus has just released its first commercially available headset (March 28'th). However, the Oculus Rift DK 2 is the headset used in this project.

In addition to VR headsets many companies are developing VR applicable accessories, mainly for tracking bodily functions into the virtual world. Following is a list of some of the most promising technologies, and a brief description:

- The Virtuix Omni - low friction platform and shoes which registers walking / running [59].
- Myo - wristband registering arm movements by reading electromyographic signals generated by the muscles in the arm [21].
- Perception Neuron - motion capture system capable of registering full body movement [36].

- STEM System - game controllers and tracking sensors for interaction with virtual worlds [20].
- Manus VR - gloves equipped with motion sensors for hand tracking [60].
- Leap Motion - the hand tracking device used in this project, see chapter 1.4.2 for a more thorough description.

Currently, the main target for VR technology is the gaming industry, where the aim of the headset is to induce the feeling of being present inside the game- and virtual universes. The accessories are mainly means of interacting with this virtual world in natural ways.

However, there are many more applications in which such headsets and devices may prove useful. Following is a couple of examples:

- Watching movies and entertainment filmed by 360 degree cameras, providing a new level of presence [61].
- Shopping by browsing stores rendered into a virtual world, for instance enabling shoppers to virtually design their own kitchens [6].
- Concerning health care, virtual realities can be specially designed to cure phobias or to improving life quality for movement impaired individuals [65].
- Boring lectures may be constructed into virtual guided tours, providing limitless opportunities in terms of information presentation and dissemination [29].

This thesis explores the usage of virtual reality technology as a means to control, monitor and interact with remotely located robotics in the real world, as if the operator was present on site. Allowing the operator to sit comfortably in a safe environment while being immersed in operations at remote locations.

1.1.1 Motivation

Before I started on the two year education program leading to this thesis, I worked as an automation engineer. My work then ranged from fixing broken equipment to development of entirely new autonomous solutions. On many occasions I had multiple projects running in parallel, in addition to stressed out customers which needed their broken equipment fixed. Much of the work required my presence on site, which often were in remote locations. Hence, time consuming traveling between customers became part of the weekly routine. In many cases traveling took longer than the actual productive time spent on site, especially in case of simple work such as visual inspections or pushing some buttons. This of course got frustrating at times when time wasted on traveling resulted in delays for other projects.

This frustration serves as the main motivation for choosing the subject of this thesis. Being able to perform tasks at remote locations through a local robot proxy, as if being there, would save me and other people in my situation a tremendous amount of time as physical traveling would no longer be necessary. Additionally, such a system would eliminate any health risks of working in hazardous environments. This is mainly why my thesis aims to implement a proof of concept solution to this issue.

In addition to my work experience, I am also very interested in new technology and its possibilities, which is why I already owned an Oculus Rift DK2 and a Leap Motion device in advance. This thesis presented a golden opportunity to familiarize myself with these two devices on a developer level, and to gain experience which can prove useful in future career.

1.1.2 Previous Work

As mentioned, this thesis is a further development on previous work done by the undersigned, during the autumn specialization project of 2015 [50]. A brief summary of its resulting program is provided below, but for a thorough description please refer to the report itself.

This project resulted in a program which combines two remotely located Internet Protocol cameras with the two logical displays of the Oculus Rift. Each camera got set up with its built in Real-Time Streaming Protocol (RTSP) servers, and the C++ library OpenCV was used in order to grab frames from each camera on the client computer. A Heads Up Display was then drawn on top of the frames from the left and right cameras before they were pasted as textures on top of two squares defined by vertices. Each square was then rendered into a framebuffer, such that they were located directly in front of each eye's view. The framebuffers were then presented in the Oculus Rift, resulting in a stereo camera view with an on top Heads Up Display (HUD).

In addition, both cameras were mounted on top of a Pan-Tilt unit (PTU) whose angle was to mimic the angle of the operators head. This was achieved by continually polling the motion sensors in the Oculus Rift for the operators current head pose, and then transmitting scaled versions of the head's roll and pitch angles as set points for the PTU. As a result, the cameras point in the same relative direction as the users head, giving the operator the ability to look around - head movements result in corresponding camera movements. However, the communication between the computer and the PTU was not implemented through network, thus a local connection between the computer and the PTU controller was necessary.

During the course of the project, functionality originating from this previously developed system have been largely replaced by better solutions. However, four functionalities remain similar.

1. The approach in using OpenCV to grab video frames through RTSP from the IP cameras.
2. Submission of frames to the Oculus Rift. Except for a type change from Direct to EyeFOV frames, the submission process is similar.
3. OpenGL context creation. The WinAPI is still used for OpenGL context creation, in the same manner.
4. The PTU implementation. Serial communication to the PTU is similar, except it is now realized through an instance of a SerialHandler object on the server. The messages sent to the PTU controller, and scaling of the Oculus Rift sensors matches the ones used in the previous project.

The robot used in the project has been the target device for multiple previous projects here at NTNU. Following are the two reports found to be most relevant.

- Aspunvik built the platform on which the manipulator is fastened, and so his documentation has been used extensively in order to understand the wiring and set up of the drivable part of the robot. His report also provides good descriptions of the equipment installed, and why the particular equipment is chosen. [2].

- Kristian Saxrud Bekken created a client- server solution to the Scorbot, such that he was able to remotely control the manipulator. His approach, however, is heavily based on separating the source code originating from Intelitek's own C++ example into two applications [22]. Thus, his solution suffers from not being able to move more than one joint at a time, which does not appeal to this project. Nevertheless, it provided useful insight as to how to use the Intelitek Scorbot in a C++ application [3].

Due to the fact that a project like this was only recently made possible, because of its usage of new technological innovations, there is a very limited amount of similar projects that has been published. Hence, information from a large number of references and literary sources were combined in order to find good solutions to the different problems which arose during the project. Much of the background information is found online, because books are yet to be written on the subjects.

The online SDK documentations for the Oculus Rift, Leap Motion and Microsoft Kinect were used extensively in order to understand how to use this equipment as a developer. In addition, the thesis of Eirik Bjørndal Njåstad contained some useful information concerning the Microsoft Kinect and calibration [35].

The inverse kinematics and related mathematics is heavily based on the works of Spong et al. [53]. The master thesis of Lars Tore Rørlien Carlsen and Philip Røst Wehinger also proved useful in terms of getting acquainted with the field, as they calculate both forward and inverse kinematics on an ABB manipulator in their thesis [44].

1.2 Objectives

The objective of this thesis is to use the telepresence system developed during the autumn project [50] as a basis for further development, and to incorporate control of a remotely located robot consisting of a robotic arm mounted on top of a drivable platform. The control of the robotic arm will be realized by using a Leap Motion device, such that the operators hand movements are translated into robot movements, while the robot platform will be controlled by a joystick.

The main objectives are summarized in the following points.

1. Creating a client/server solution which enables network-communication in between a remotely located robot and a local computer.

2. Reading the left hand position of the operator into the client program, by using the Leap Motion device.

3. Constrain the space in which the operator's hand may move, to a similar space in which the robotic arm is able to move.

4. Transforming hand and fingers positional data into set points for the server program which is to maneuver the arm and claw of the robot accordingly.

5. Update the Heads Up Display created in the previous project, such that the new functionality and its associated data is portrayed intuitively to the operator.

6. Incorporate driving functionality of the robot platform itself by using a joystick.

7. Optimize and conduct tests on whether usage of the Leap Motion and joystick combination is a viable solution for robot control.

During the course of the project, the following objective was incorporated:

- Develop an alternative solution using the Microsoft Kinect for sequence mapping of positions to the manipulator.

1.3 Limitations

Equipment used during this project is limited to what was available at the time. Equipment is provided by the institute, except for The Leap Motion, Oculus Rift DK2 and client computer which are from my own private possession. Some of the provided equipment is outdated and have been superseded by new and better successors, which would increase the overall quality of the resulting system if installed. However, provided equipment is implemented as is. Some potential equipment upgrades are suggested in the Future Work section 5.2.

The development process has been focused on getting the system to work as a whole. Therefore, aspects such as safety, robustness and code efficiency of the system have not been prioritized. The system itself also has some strict requirements. It is developed for Windows machines, and the client computer must be relatively powerful in order to meet the system requirements of the Oculus Rift. Separate rendering is implemented for running the system without an Oculus Rift, however, this defeats the whole purpose of the system and is intended for debugging only.

The screenshots, as well as the demonstration videos, in this thesis is taken of the mirrored Oculus Rift representation from the monitor, which is why they appear double (frames for both eyes are displayed as one). The 3D and sense of depth as seen in the Oculus Rift is not possible to depict properly on paper, therefore the system should be tried out in practice to properly observe and evaluate its design and performance.

1.4 Equipment

1.4.1 Oculus Rift DK2

The Oculus Rift DK2 is a Head Mounted Display (HMD) to be strapped around a users head. The purpose of the device is to give the user a sense of presence in virtual environments. The built-in display is split into two logical halves in software, using separate framebuffers. Magnifying lenses are placed between the display and the eyes of the user, such that each eye sees a

magnified view of one half of the screen each. Contents for both screen halves is intended to be rendered separately with a shifted view into the 3D world, equal to the registered inter-pupillary distance of the user. The resulting effect is an immersive depth vision into the rendered 3D world, with a large field of view (FOV) due to the lens magnification.

Figure 1.1: The Oculus Rift DK2 and its bundled camera.
Image Credits: Oculus VR

The HMD also contains motion sensors which, when combined with the rendering procedure, registers head movement and moves the virtual view accordingly. Ultimately providing the sense of looking around in the virtual world. Two motion detection principals is utilized by the device in order to accurately measure rotational and translational head movement; an inertial measurement unit (IMU) and a static infrared camera reading the position of infrared light diodes built into the device shell [64]. For more details concerning the Oculus Rift, please refer to my previous report [50].

1.4.2 Leap Motion

The Leap Motion controller is a device made specifically for registering the movement of hands and fingers in 3D space, enabling users to interact with computers without having to maneuver some physical object, like a mouse or joystick. Produced by Leap Motion, Inc. it is designed as a small peripheral USB device which initially was meant to be placed on the desk in front of the computer screen.

Figure 1.2: The Leap Motion in desktop mode
Image Credits: Leap Motion, Inc.

Full-scale shipping of the first consumer version started in July 2014. However, further development revealed the potential use in virtual reality (VR) applications, and so the company launched a VR tracking mode for its core software in August 2014. This was designed to provide hand tracking while having the device mounted on virtual reality headsets such as the Oculus Rift, making it possible for users to interact with their own hands in a virtual environment.

Figure 1.3: The Leap Motion mounted on an Oculus Rift
Image Credits: Leap Motion, Inc.

Technology wise, the Leap Motion uses two monochromatic infrared cameras, and three infrared LEDs. It is able to track hands at up to 200 frames per second of reflected data within a 150° field of view. Each frame pair is sent to the computer through USB 2.0 where it is analyzed using internal algorithms to extract hands from the image, as well as their position in 3D space relative to the device. This is achieved by comparing the two 2D camera frames, however, the

exact algorithms used is kept a secret by the producer. [19]

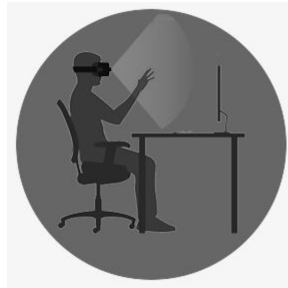


Figure 1.4: The Leap Motion and HMD in action
Image Credits: Leap Motion, Inc.

1.4.3 Joystick

A Logitech Force 3D Pro joystick is used as a controller in the resulting client application. Its main function is to drive the robot, toggle implemented functions, and switch between modes of operation. It is equipped with 12 buttons and a throttle lever, in addition to a Point of View (POV) hat and force feedback functionality. The handle itself may be moved in either direction or twisted clockwise or counter clockwise. The force feedback and twist axis is not implemented during this project.

A joystick became the controller of choice, instead of using a mouse and keyboard, because the shape and buttons of the stick is easy to recognize and localize even when wearing a vision-blocking HMD, such as the Oculus Rift. The stick is right hand oriented, and is intended to be maneuvered by the operators right hand. [28]

Figure 1.5: The Logitech Force 3D Pro
Image Credits: Logitech

1.4.4 The Robot

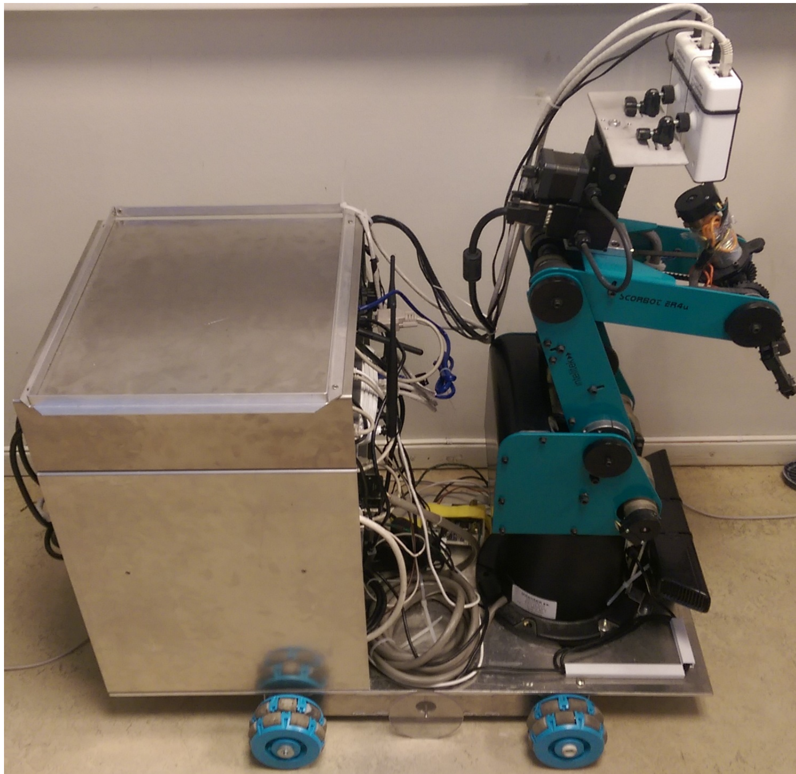
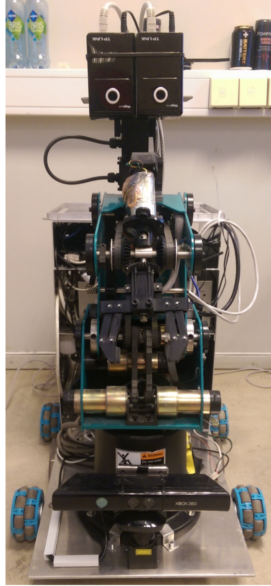
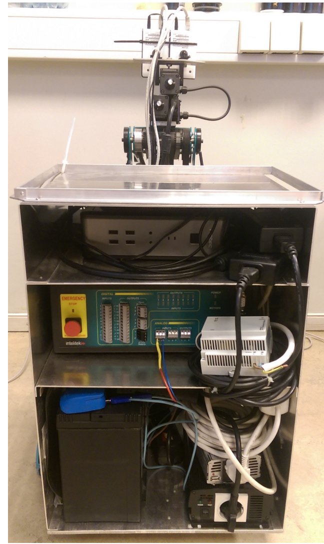


Figure 1.6: The robot viewed from the right side.

The robot which is to be used as the remote proxy mainly consists of two individually functioning parts; a robotic manipulator mounted on top of a movable platform. More precisely it consists of Intelitek's Scorbot ER4u mounted on top of metal frame driven by 4 wheels. Two IP cameras mounted on a Pan Tilt Unit (PTU) are fastened to the manipulators elbow, while a



(a) Front view



(b) Back view

Microsoft Kinect is placed in front. All devices except for the cameras are connected to a local computer, which is to be utilized as the main server for robot control. A wireless router is installed on the robot, in order to make the server and cameras wireless.

A car battery is used as the power-source for the robot. The 12 VDC supplied by the battery goes through two types of inverters, converting it to 230 VAC, which is then used as the operating voltage for all robot equipment. Two inverters are used because a pure sine inverter is necessary to power the manipulator and stepper controllers, while a cheaper modified sine inverter is powering all the rest (computer, router, cameras, Kinect).

Intelitek Scorbot ER4u

The robotic manipulator is manufactured by Intelitek for the purpose of industrial robotics training and education. It has 5 individually controllable axes which are; base-rotation, shoulder, elbow, wrist roll and wrist pitch. In addition, it comes equipped with a gripper which can open and close. The robot itself is interfaced to a computer through a bundled USB controller, which also comes with built in I/O and an emergency stop button.

Figure 1.8: Scorbot ER4u and USB Controller
Image Credits: Intelitek

The robot is meant to be programmed and controlled by using the accompanying software named SCORBASE (or RoboCell, which integrates SCORBASE in addition to software components for 3D simulation). However, an example is provided on Intelitek's download site which demonstrates basic control of the robot's individual axes in a Visual C++ program [22].

Movable platform

The movable platform was constructed and programmed by Petter Aspundvik during his master thesis in 2013 [2].

In short, it consists of a metal plate mounted on top of a metal frame, to which 4 omnidirectional wheels and two encoders are attached. The wheels are mounted in parallel on the platform, which is sub-optimal as the entire platform may uncontrollably slide to either the left or right. Optimally the wheels should be mounted at a 45-degree angle on each corner of the platform, however, by making sure the robot always finds itself on a flat surface, unaffected by external forces, the problem is avoided. The platform requires a 24 VDC power source, which originally was drawn from a separate battery. However, a 24 VDC voltage supply has been installed during this project, in order to manage with just the car battery.

Atmel XMEGA A3BU

Each wheel is connected to its own stepper motor, and each of the motors are controlled by an Atmel XMEGA A3BU microcontroller evaluation board. The motors are controlled by pulse width modulation (PWM) signals from the microcontroller through their designated stepper controllers. In addition to the stepper motors, two separate encoders are connected to the A3BU. The encoders are located on the robots left and right, and enables the measurement of actual distance driven and its derived speed / acceleration. The microcontroller is connected to the local computer through USB, and is programmed in C through the JTAG interface by using Atmel Studio.

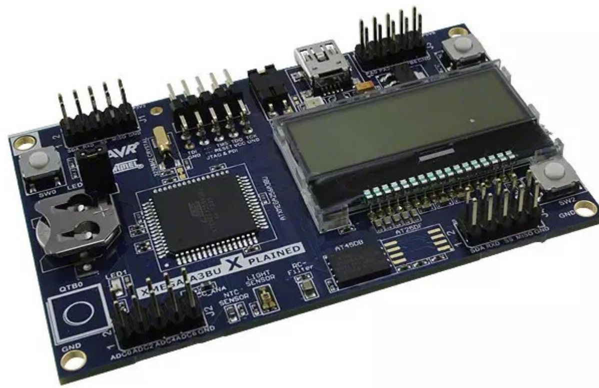


Figure 1.9: Atmel XMEGA-A3BU Xplained
Evaluation Board
Image Credits: Atmel Corporation

The A3BU Evaluation Board contains pin outs for all ports of the A3BU microcontroller itself, in addition to a display, 3 on-board buttons, a battery, a 32k Hz oscillator and other necessary electronics (decoupling capacitors, pull down resistors, voltage regulators etc). Providing easy access to the peripherals of the chip. For details regarding the capabilities of the chip and evaluation board, please refer to datasheets on C.

The interconnections and choice of equipment is kept similar as to what was chosen by Aspunvik. However, instead of using two separate A3BU controllers (one for motor control and another for encoder readings), it has been combined into one microcontroller in this project, as described in Chapter 3.

For details regarding the stepper motors, stepper controllers and encoders please refer to datasheets on C or [2].

Microsoft Kinect for Xbox 360

Microsoft Kinect was launched as an Xbox 360 peripheral device near the end of 2010, followed by a release of official computer support in 2011. It enables users to interact with and control their computer / Xbox 360 by using natural gestures and voice commands.

Figure 1.10: The Microsoft Kinect for Xbox 360
Image Credits: Microsoft

In practice, the Kinect is a camera capable of registering depth in addition to traditional color values. It is also capable of registering bidirectional sound. The device comes equipped with four microphones, a depth sensing IR camera, an IR projector and a Red, Green, Blue (RGB) camera. The IR projector emits infrared light which is continually observed by the depth camera. Using the known distance between the RGB camera lens and the depth camera lens it calculates a corresponding depth of each pixel registered by the RGB camera [17].

The Kinect software development kit (SDK) provides access to the camera and audio streams of the device, enabling programmers to use the Kinect in applications. The SDK also contains a lot of functions for extracting data from these streams. Since the device is originally intended as an Xbox 360 peripheral, a separate adapter is necessary to power the device and to convert the proprietary Kinect connector to USB 2.0. [32]

Pan Tilt Unit

Figure 1.11: The PTU-D46 and controller
Image Credits: FLIR Motion Control Systems

The Pan Tilt Unit (PTU) used in this project is a PTU-D46, now manufactured by FLIR Motion Control Systems after their acquisition of the original producer Directed Perception Inc [18]. The unit consists of two stepper motors, one for each angle, and is controlled by a separate controller that supplies power and control signals for the motors. The controller function as the link between the PTU and a computer, and is the entity containing logic; translating commands into movements. Communication between the computer and the controller is achieved through either RS-232 or RS-485. The same device was also used in my previous project, and so please refer to [50] for more details.

IP Cameras

Two IP cameras, TL-SC3430 produced by TP-Link, are mounted on top of the PTU facing the same direction. The cameras are bound closely together, such that the distance between the lenses are approximately 65mm, matching the average Caucasian inter-pupillary distance (IPD) [55]. Images from the cameras may then be used to generate appropriate stereoscopic views.

Figure 1.12: The TL-SC3430.
Image Credits: TP-Link

The cameras support a maximum resolution of 1280x1024 at 15 frames per second (FPS), and are able to stream video through the Real-Time Streaming Protocol (RTSP). An integrated webserver provides easy access to the cameras features and settings. For more details regarding their capabilities, please refer to datasheet on C and the previous telepresence project where the same cameras are used [50].

1.4.5 Software

Software and SDKs used during the course of this project.

- Microsoft Visual Studio 2013
- Atmel Studio 7.0
- Intelitek Scorbase - For testing the Scorbot, driver installation and parameter setup
- OpenCV - Image processing C++ library
- GLEW - OpenGL function library

- GLM - OpenGL math library
- The FreeType Library - Rendering text
- Oculus SDK v0.8 - Software development kit for interfacing the Oculus Rift
- LeapSDK v3.1.2.40841 - Software development kit for interfacing the Leap Motion
- Kinect SDK v1.8 - Software development kit for interfacing the Microsoft Kinect
- Visual Leak Detector - Checking the software for memory leaks
- Wireshark - Monitoring and verifying network communication
- Putty - Testing serial communication
- MatLab - For testing and verification of mathematical functions

Chapter 2

Theoretical Background

2.1 Linear Transformation Matrices

In order to manipulate the position and orientation of objects defined mathematically in 3D space, transformation matrices have been applied. Transformations are concatenated by multiplying individual transformations, as described in the upcoming sections, together into one matrix.

Operations are performed from right to left, such that when matrices are multiplied in the following order **Transformation Matrix** = **Translation** * **Rotation** * **Scaling**, scaling is applied first, then the scaled object is rotated, before the scaled and rotated object is translated. The order in which composite transformations are multiplied is important, as rotation and scaling is performed relative to the origin of the global coordinate system. If the object is moved (translated) from this origin before applying rotation / scaling, different results will be produced [34].

2.1.1 The Rotation Matrix

Rotating a point or vector in Euclidean space is achieved by multiplying its coordinates with an appropriate rotation matrix. Rotation matrices are used to describe rotations and orientations of rigid bodies in relation to some initial coordinate system. A rotation matrix from frame A to frame B R_b^a may be used to either transform a coordinate vector as expressed in frame B to its

frame A representation, or it may rotate a vector from one orientation to another orientation within the A frame, see Eq. (2.1).

$$\mathbf{p}^a = \mathbf{R}^a_b \mathbf{p}^b \qquad \mathbf{q}^a = \mathbf{R}^a_b \mathbf{p}^a \tag{2.1}$$

Where $\mathbf{p}^a$ is the frame A representation of $\mathbf{p}^b$.
$\mathbf{R}^a_b$ is the rotation matrix which rotates from A to B (transforms from B to A).
$\mathbf{q}^a$ is the rotated $\mathbf{p}^a$ vector within the A frame.

Combining multiple rotations within one rotation matrix is achieved by multiplication:

$$\mathbf{R}^0_i = \mathbf{R}^0_1 \cdot \mathbf{R}^1_2 \cdots \mathbf{R}^{i-1}_i \tag{2.2}$$

Any three dimensional orientation can be expressed by combining three basic rotation matrices, where each matrix rotate the point around one axis of the initial coordinate system. The three basic rotation matrices are given in Eq. (2.3) to (2.5), where each matrix rotates by θ degrees around x, y, z.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{2.3}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.4}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Similarly to when defining transformation matrices, the order in which rotation matrices are multiplied together is important, as all rotations happens according to the world frame and not in relation to the orientation of the object itself. Hence, care must be taken when constructing transformation matrices of rotations around multiple axis, as the outcome will differ depending on the rotation matrix order.

[9]

2.1.2 The Translation Matrix

Moving a point in Euclidean space from one position to another is achieved by multiplying the point vector, using 4 homogeneous coordinates, with a 4x4 translation matrix. The matrix specifies the distance the vector is to be moved in all 3 dimensions, x, y, z. Translation matrices may be multiplied with rotation matrices to form homogeneous transformation matrices, which describes both rotation and translation of a coordinate frame.

In order to translate a point by a vector d, the point can be multiplied by the following translation matrix.

$$T_b^a = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Where, d_x, d_y, d_z represent the x, y, z component of the translation vector, respectively.

[53]

2.1.3 The Scaling Matrix

In order to scale a geometrical object consisting of multiple points, each of its points may be multiplied with a scaling matrix. The matrix then enlarges or shrinks the objects in relation to each axis as described by the scale factors, s_x, s_y, s_z.

$$S_v = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

Where s_x, s_y, s_z are the scale factors of x, y, z, respectively.

[58]

2.2 Kinematics

Kinematics are employed in order to find a geometric description of the motion performed by the robotic manipulator, without consideration of the forces and torques involved in generating the motion. The robot manipulator is composed of a set of links connected by revolute joints, providing one degree of freedom each.

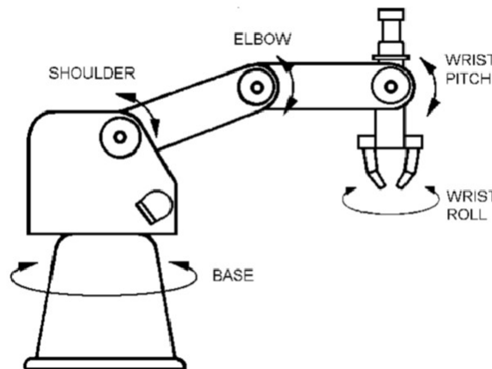


Figure 2.1: The joints of the Scorbot

Axis are defined such that the base joint operates in the global X-Y plane (the floor) while the other joints operate in the plane formed by the global Z-axis (up) and the axis following the arm, see Fig. 2.1. Calculations and information of the upcoming sections are based on material provided by Spong et al. [53] and partly [1].

2.2.1 Forward kinematics

When maneuvering a robot manipulator it is important to know the position of the tool in relation to some global coordinate system. By using the Denavit-Hartenberg (D-H) convention for defining the frames of reference, the tool position in the global coordinate system can be found by knowing the joint angles. The Denavit-Hartenberg procedure starts with defining relative distances and rotations in a table. The table contains a list of all joints and its associated α, a, d, and θ values, which refer to the following:

- α: Link twist angle
  The twist angle between Z_{i-1} and Z_i about X_i, in the Right Hand sense.
- a: Link length
  The distance along the X_i axis between the origin of frame i and where the Z_{i-1} axis intersects.
- d: Link offset
  The distance from where the Z_{i-1} axis intersects with the X_i axis, to the origin of frame $i-1$.
- θ: Joint angle
  The angle of the joint, or more specifically the angle between the X_{i-1} and X_i axis about the Z_{i-1} axis.

The Scorbot ER4u is parameterized in the following D-H table:

$Joint_i$	α_i (rad)	a_i (mm)	d_i (mm)	θ_i (rad)
1	$\frac{\pi}{2}$	16	349	θ_1
2	0	221	0	θ_2
3	0	221	0	θ_3
4	$\frac{\pi}{2}$	0	0	θ_4
5	0	0	145.125	θ_5

These values can then be entered into the general formula (2.8) to find the appropriate transformation matrix between each joint. The transformation matrices may then be multiplied together in order to form the complete transformation matrix from the tool frame to the global

frame.

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1\cos\alpha_1 & \sin\theta_1\sin\alpha_1 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1\cos\alpha_1 & -\cos\theta_1\sin\alpha_1 & a_1\sin\theta_1 \\ 0 & \sin\alpha_1 & \cos\alpha_1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

In case of the Scorbot ER4u the procedure results in the following transformation matrices from the tool frame to the frame defined at the manipulators base:

$$T_1^0 = \begin{bmatrix} \cos\theta_1 & 0 & \sin\theta_1 & 16\cos\theta_1 \\ \sin\theta_1 & 0 & -\cos\theta_1 & 16\sin\theta_1 \\ 0 & 1 & 0 & 349 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_2^1 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & 221\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & 221\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

$$T_3^2 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 221\cos\theta_3 \\ \sin\theta_3 & \cos\theta_3 & 0 & 221\sin\theta_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_4^3 = \begin{bmatrix} \cos\theta_4 & 0 & \sin\theta_4 & 0 \\ \sin\theta_4 & 0 & -\cos\theta_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

$$T_5^4 = \begin{bmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & 0 \\ \sin\theta_5 & \cos\theta_5 & 0 & 0 \\ 0 & 0 & 1 & 145.125 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

$$T_5^0 = T_1^0 * T_2^1 * T_3^2 * T_4^3 * T_5^4 \tag{2.12}$$

2.2.2 Inverse Kinematics

In order to find joint angle set points for both the physical and virtual manipulators from the hand position, the principles of inverse kinematics were applied.

Since the orientation of the robotic wrist is supposed to mimic the orientation of the operators hand, and the Leap Motion device calculates the hand orientation directly, these angles are already known. Hence, only the base, shoulder and elbow angles needs to be derived using inverse kinematics in order to find the robots position representing the position of the hand.

Using the geometric approach finding θ_1, θ_2 and θ_3

The point in space where the manipulator's wrist center is located is defined as O_c which consists of X_c, Y_c and Z_c.

By projecting the vector O_c onto the plane given by the X- and Y- axis we can define the angle of the first robot axis θ_1.

$$\theta_1 = tan^{-1}\left(\frac{Y_c}{X_c}\right) = Atan2\left(\frac{Y_c}{X_c}\right) \tag{2.13}$$

Where Atan2 is the two-argument inverse tangent function which is defined for all $(x, y) \neq (0, 0)$, and equals the unique angle θ such that

$$\theta = Atan2\left(\frac{y}{x}\right), \qquad \sin\theta = \frac{y}{\sqrt{x^2+y^2}}, \qquad \cos\theta = \frac{x}{\sqrt{x^2+y^2}} \tag{2.14}$$

By using two arguments the Atan2 function gather information of the signs of (x, y), in order to select the appropriate quadrant for the angle θ. Finding the appropriate quadrant is not possible by using the single argument inverse tangent function. It also avoids divisions by zero and thus is a suitable choice for computer programs.

Given θ_1 the angle for the elbow θ_3 can be obtained by applying the law of cosines in the plane formed by the second and third links of the manipulator, resulting in the following relationship

$$\cos\theta_3 = \frac{r^2 + s^2 - a_2^2 - a_3^2}{2a_2a_3} \tag{2.15}$$

Where $r^2 = X_c^2 + Y_c^2 - d^2$ and $s^2 = (Z_c - d_1)^2$.

d_1, a_2 and a_3 can be read from the D-H table, while the d value represents the offset in the X- and Y- plane between the shoulder joint and the origo of the global coordinate system. In the case of Scorbot ER4u the magnitude of the offset is equal to a_1 in the D-H table, and it influences

the X_c and Y_c coordinates according to the angle θ_1. Hence, using (2.15) the elbow angle θ_3 can be found

$$\cos\theta_3 = \frac{(X_c - a_1\cos\phi\cos\theta_1)^2 + (Y_c - a_1\cos\phi\sin\theta_1)^2 + Z_c^2 - a_2^2 - a_3^2}{2a_2^2a_3^2} := D \tag{2.16}$$

$$\theta_3 = Atan2(\pm\sqrt{1-D^2}, D) \tag{2.17}$$

The solution $\theta_3 = Atan2(\sqrt{1-D^2}, D)$ provides the joint angles for an "elbow down" configuration, and $\theta_3 = Atan2(-\sqrt{1-D^2}, D)$ the joint angles for an "elbow up" configuration. A solution to the equation can be found by just taking $\arccos(D)$, however, this does not recover both solutions as the $Atan2$ function does. As seen, the d_1 value of the s parameter is excluded from the calculation. This is because the hand coordinates provided by the Leap Motion operate around zero. Hence, a z-value of zero result in angles corresponding to the robot pointing straight out from its shoulder joint, which is similar to how the operators hand is positioned when hand position has a z-value of zero.

When both the θ_1 and θ_3 angles are found they can be used to find the shoulder angle θ_2:

$$\begin{aligned}\theta_2 = Atan2\big(&-Z_c, \sqrt{(X_c - a_1\cos\phi\cos\theta_1)^2 + (Y_c - a_1\cos\phi\sin\theta_1)^2}\big)\\ &- Atan2(a_3\sin\theta_3, a_2 + a_3\cos\theta_3)\end{aligned} \tag{2.18}$$

[53]

Constrain Coordinates within the Frontal Hemiellipsoid

The maximum point the manipulator can reach in any direction is equal to the length of its arm when fully stretched. As shown in 2.2, this can be visualized as a hemiellipsoid in this case. All maximum points reside on a hemiellipsoid and not a hemisphere, because of the distance between the shoulder joint and the z-axis (a_1 in 2.2.1), which make the arm able to reach $a_1 mm$ longer in the horizontal xy-plane than in the vertical yz-plane.

Maximum and minimum limits were found for the x, y, z positional parameters for the wrist by utilizing basic geometry functions.

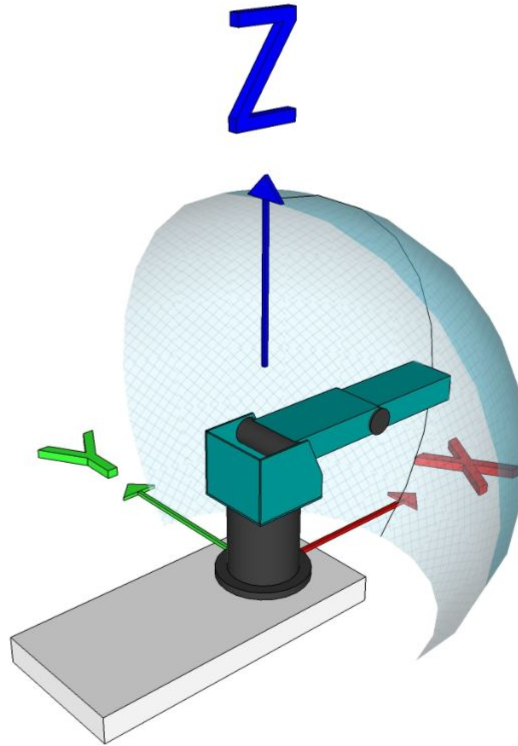


Figure 2.2: A 3D representation of the Scorbot's coordinate system

The maximum reach of the arm is defined as the length of the shoulder, elbow and their offset from the z-axis of the base.

$$V_p = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{2.19}$$

$$L_p = \sqrt{x^2 + y^2 + z^2} \tag{2.20}$$

$$\theta = \arccos(\frac{x}{L_p}) \tag{2.21}$$

$$\gamma = Atan2(x, z) \tag{2.22}$$

$$\phi = \arccos(\frac{\sqrt{x^2 + y^2}}{L_p}) \tag{2.23}$$

$$R_{max} = a_1 \cos\phi + a_2 + a_3 \tag{2.24}$$

$$x_{max} = R_{max} * \cos\theta \tag{2.25}$$

$$y_{max} = R_{max} * \sin\theta \sin\gamma \tag{2.26}$$

$$z_{max} = R_{max} * \sin\theta \cos\gamma \tag{2.27}$$

Where L_p is the length of the input vector V_p.
θ is the angle between the x-vector (robot forward) and the position vector.
γ is the angle of the position vector in the yz-plane.
ϕ is the angle between the horizontal xy-plane at the shoulder joint, and the position vector.
R_{max} is the maximum reach of the robot arm in relation to where the z-axis of the initial frame and the x-axis of the shoulder frame intersect (see 2.2.1 for a_i values).
x_{max} represents the maximum possible x position (forward) at the current angle.
y_{max} represents the maximum possible y position (left / right) at the current angle.
z_{max} represents the maximum possible z position (up / down) at the current angle.

The necessity of coordinate constraining became apparent as the Leap Motion may register positions not reachable by the robotic arm. In order to keep the inverse kinematic formulas from resulting in invalid angles, the coordinates passed on to the formulas is ensured to yield valid joint angles, using this approach.

2.3 Network Communication

As described in [50] the Real-Time Streaming Protocol (RTSP) is the network communication protocol used in order to acquire the camera streams for the visual part of the telepresence system. As this protocol is intended for video and audio streams and is used directly to the cameras, it is not applicable for remote control of the robots motion.

The protocol of choice for controlling robotics over an IP network is not the Transmission Control Protocol (TCP), but rather the User Datagram Protocol (UDP). This is mainly because TCP is a connection oriented protocol with strict reliability requirements. TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is solved by assigning sequence numbers and requiring ACK responses for each packet received by the receiver. If an ACK is not received for a packet within a given timeout, the packet is re-transmitted [24]. In terms of real time control of a remote system, i.e. a remote robot, the re-transmission of packets is undesired, which is why the UDP protocol was

chosen in this scenario.

The paper by Liu et al. [27] suggests a novel scheme for real-time robotics communication named the trinomial protocol. This protocol minimizes the risk of losing data packets due to varying bandwidth availability, as transmission frequency is reduced if available bandwith is reduced. It also shares the available bandwith between nodes using the same connection. However, this paper is written in 2003 when available bandwith was more of a concern than it is today. The decision was therefore made to use the UDP protocol directly. However, the graphs shown in chapter 2 and 4 of [27] also compares TCP and UDP when it comes to transmission rate, delay jitters and packet loss rate, and gives a clear indication as to why UDP is superior to TCP in a real-time control context.

2.3.1 User Datagram Protocol (UDP)

UDP is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks [23]. Unlike the TCP protocol, UDP is a connection-less and transaction based protocol. It does not guarantee the delivery of data-packets, and offers no protection against duplicate packets. However, this is an advantage when controlling a remote system in real time as it is desired that the remote system acts upon the most recent set points. Packets lost during transmission under the UDP protocol are not detected, and no time will be wasted in retransmission of out-dated set points. Instead the transmitter can focus on constantly sending the most recent set points.

The protocol is connection-less meaning it does not maintain a connection to any particular target machine. Simply put, a destination IP address and port must be supplied for each call to the UDP transmit function in addition to the data to be transmitted. Once the transmit function has been called it writes the data contained in the supplied buffers to the network, addressed to the given destination. The function then immediately returns, and does not care whether the destination machine receives the data or not. If necessary, it is up to the programmer to write his own handshaking / watchdog mechanism in-between nodes.

Similarly, when receiving data on the UDP protocol an IP address and port must be supplied which is telling the receive function what address it is supposed to receive data from. Typically a receive function blocks execution until data is received, but timeouts may be added in order to control for how long it should be allowed to block.

Checksums may be generated and included in the UDP packets in order to let the receiver verify data integrity of the packets received. Using checksums is recommended but not required when using UDP. When communicating over larger distances through the internet it is strongly recommended, but may be dropped to speed up implementation in LANs, under controlled circumstances.

Maximum UDP datagram size equals $2^{16-1} = 65535$ bytes. Subtracting 20 bytes for the IP header and 8 bytes for the UDP header, it leaves $65535 - 20 - 8 = 65507$ bytes in which data may be stored [4].

[16] [8]

Maximum Transmission Unit (MTU)

The MTU is the size of the largest possible data unit which may be transmitted in one non-fragmented packet across the underlying network interface. In case of modern Ethernet, which constitutes most of the internet, the MTU is 1500 bytes. This means that no more than 1500 bytes worth of data (including the IP header) may be sent in one data fragment across the physical network. If the data size exceeds 1500 bytes it is fragmented into multiple packets, before they are transmitted across the wire. Packet fragments are flagged as such, enabling the receiving end to reassemble the original data packet from the individually received fragments.

The fragmentation and reassembling processes is carried out by the Internet Protocol in the network layer (layer 3 in the Open System Interconnection (OSI) model), while UDP are categorized as a transport layer protocol (layer 4 in the OSI model). Considering that an OSI layer serves the layer above it, and is served by the layer below it, UDP has no knowledge of an even-

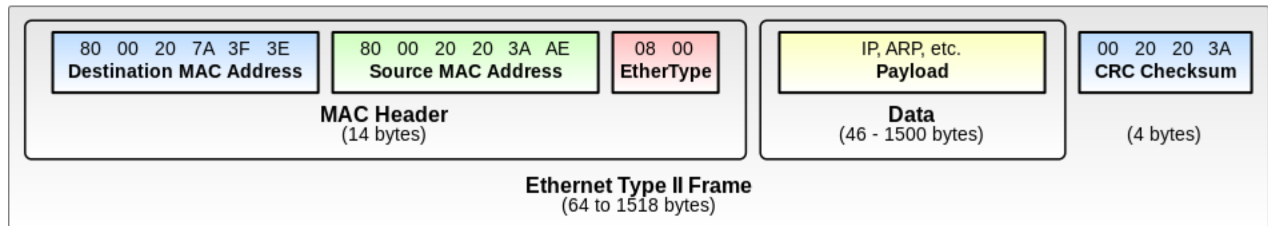


Figure 2.3: The most common Ethernet Frame Format

tual fragmentation / reassembly happening in the IP layer, as UDP simply send and receive complete data buffers to and from the IP layer.

Because of fragmentation, it is not recommended to send more than 1500 bytes of data in one transmission. The main reason being that if a data packet is fragmented and one fragment is lost, the entire data packet becomes invalid and all fragments must be re-transmitted. This is avoided by using non-fragmented packets, as all data will reach its destination if the packet itself does. The fragmentation problem increases the further the packets have to travel across the network, increasing the risk of resulting performance degradation.

Some equipment support the use of Ethernet Jumbo-frames which may contain data payloads between 1501 - 9198 bytes. However, as all equipment on the entire Ethernet network must have the same MTU, Jumbo-frames are usually only seen in special-purpose local networks.

[7]

2.4 Oculus Rift and Lens Magnification

The Oculus Rift achieves its large field of view by magnifying its display with two lenses, one for each eye. This approach introduces two challenges; pincushion distortion and chromatic aberration. Both of these effects are compensated for by algorithms implemented within the Oculus SDK, if the content to display is supplied as Oculus Rift frames and not direct / debug frames [63].

2.4.1 Pincushion Distortion

Since the lenses of the Rift magnifies their respective portion of the display, the images becomes significantly distorted when viewed through the lenses. Therefore, a barrell distortion is applied to each frame before they are presented on the HMD. This is done automatically in the Oculus SDK by post-processing of the rendered frame which is submitted, and effectively zeroes out the pincushion distortion effect caused by the lenses.

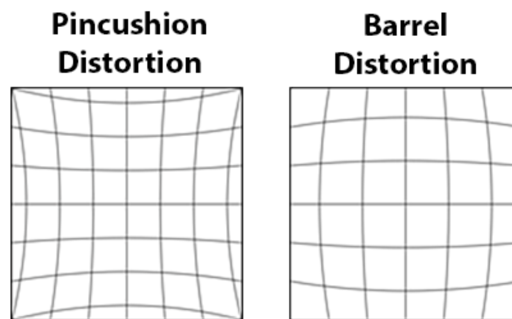


Figure 2.4: Pincushion and Barrell Distortion
Image Credits: Oculus VR

2.4.2 Chromatic Aberration

This is an optical effect resulting from viewing images through a lens which is unable to focus all colors at the same convergence point. It occurs because of lenses having different refractive indices for different wavelengths of light, meaning different wavelengths of light travel at different speeds through the lens resulting in colors being bent / refracted in different angles. The refractive index of a lens decreases with increasing wavelength. Hence, the color blue is bent more on its journey through the lens than red. [30]. There are two types of chromatic aberration, as illustrated in Fig. 2.5.

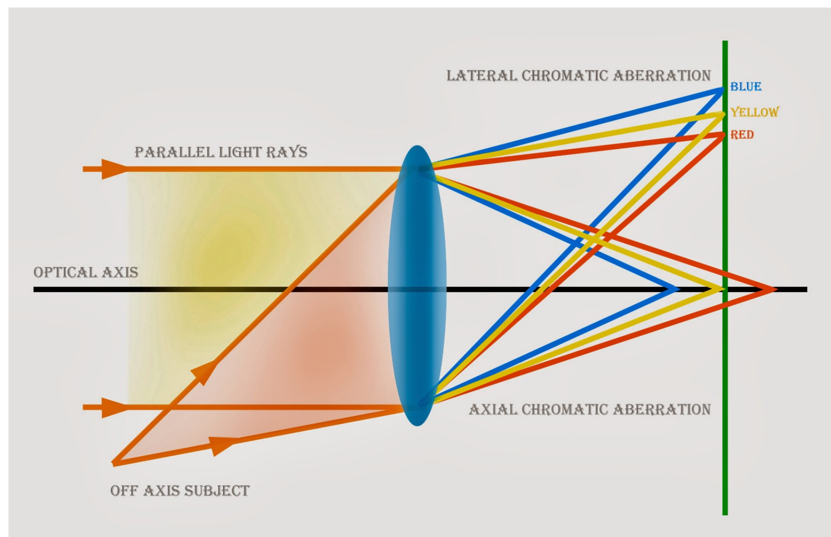


Figure 2.5: Illustration of longitudal chromatic abberation.
Image Credits: School of Digital Photography

Figure 2.6: An image with chromatic abberation.

2.4.3 Time Warp

The Oculus SDK comes with a built-in time warp function, whose purpose is to measure and predict the operators head movement *during* the rendering of the upcoming frame to be displayed. The process of rendering for the Oculus Rift starts off by reading the devices sensors to get the users head position. The 3D geometry is then transformed according to the head angle, and rasterized into separate framebuffers for each eye. The framebuffers are then submitted to the Oculus Rift for presentation to the user.

The process takes some time, during which the user may have changed his head pose ever so slightly from what was originally measured at the start, meaning, the head angle at presentation is not equal to the head angle rendered for. Time warp is an implemented algorithm in the SDK

that measures how much the user's head has moved during rendering of the upcoming frame. The position difference is applied as a last-minute transformation of the virtual view, just before presentation on the display, such that the user virtual view corresponds with the present head pose of the user even during rapid head movements.

[26] [63]

2.5 Open Graphics Library - OpenGL

OpenGL is a low-level graphics application programming interface. Its main purpose is to provide access to features of the graphics hardware, and providing a rendering pipeline for generating frames to be portrayed on attached displays. The library is hardware independent and may be implemented on many different types of systems, or in software, independent of the operating system and windowing system of the computer. Hence, procedures for handling user input and windowing tasks must be programmed separately by using facilities available from the underlying system. Similarly, OpenGL does not provide functionality for describing 3D objects nor reading image files (textures). Any 3D object must be constructed from collections of geometric primitives, such as triangles, lines, or points, as defined by vertices. The process of turning vertices into displayable graphics is achieved through the rendering pipeline. [49]

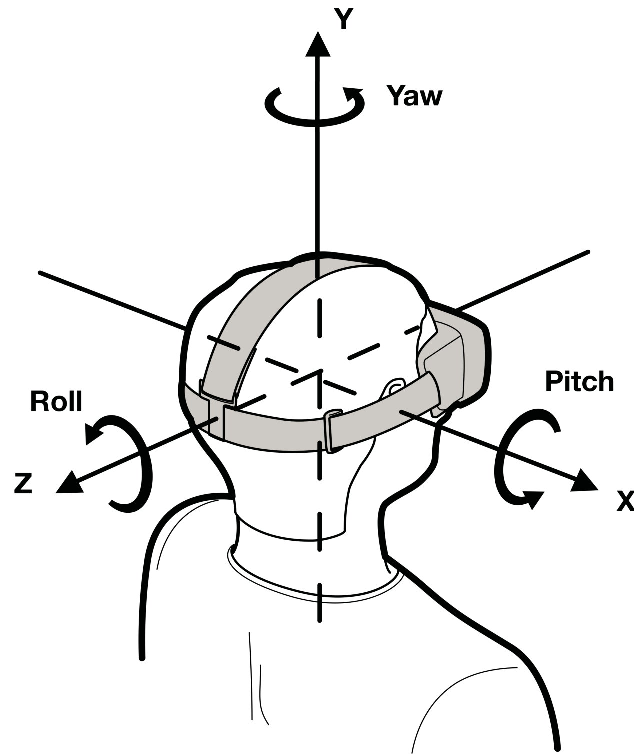


Figure 2.7: The screen coordinate system for the Oculus Rift. As illustrated, the z-axis comes out of the screen, while the y-axis points up and the x-axis points to the right.
Image Credits: Oculus VR

2.5.1 Rendering Pipeline

OpenGL, the rendering pipeline and its stages are huge subjects, and so the descriptions provided in this section are stripped down and superficial. Countless books are written on the subjects, and the field is constantly evolving. Information presented in this section is culminated from the following sources; [49] [38] [41] [42]. In addition, a more thorough description of the rendering pipeline is provided in chapter 2 of my previous report [50]. Please refer to this report for elaborations to some of the information presented below.

The purpose of the rendering pipeline, also called graphics pipeline, is to generate images to display on screen from geometry defined by vertices in code. The process consist of several stages, as illustrated in 2.8.

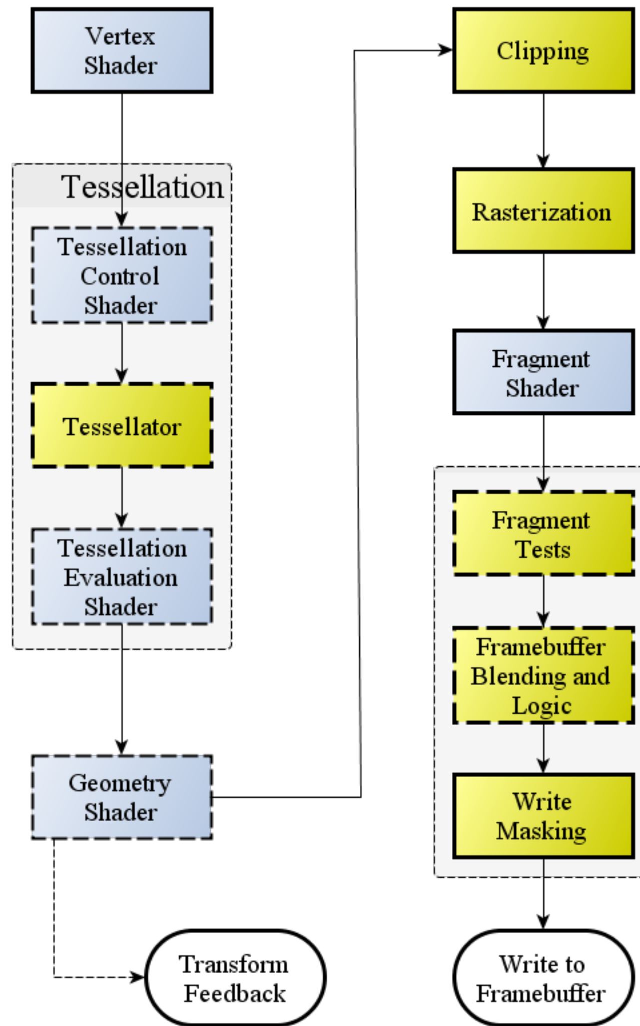


Figure 2.8: Stages of the OpenGL rendering pipeline
Image Credits: OpenGL Wiki

Shaders are small programs written in OpenGL Shading Language (GLSL) that are compiled into one program and loaded into graphics memory during runtime. The shaders run on the Graphic Processing Unit (GPU) while loaded, and process each vertex fed into the pipeline, to form an image to display on screen. The vertex shader and fragment shader are obligatory in any OpenGL program, while the tessellation shader (consisting of three stages) and the geometry shader are optional. The optional stages may be included to automatically generate additional vertices and effects, to increase the quality of the rendered image. However, they are not included in this project.

The vertex shader process every vertex enlisted in the bound vertex buffer when a draw command is executed. Its main goal is to define the position of each vertex in the buffer, however, the shader is user definable and may contain functions for calculating all kinds of vertex specific values, such as color- and lightning effects. The shader stage operates on each vertex individually, and has no knowledge of upcoming or previously processed vertices, and is therefore the natural stage at which vertex multiplication with model-, view- and projection matrices occur.

Listing 2.1: Vertex Shader as implemented in the client application

```
#version 150
in      vec4 Position;
in      vec4 Color;
in      vec2 TexCoord;

uniform mat4 projMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

out     vec4 oColor;
out     vec2 oTexCoord;

void main() {
   oColor = Color;
   oTexCoord = TexCoord;
   gl_Position = (projMatrix * viewMatrix * modelMatrix) * Position;
};
```

As seen from 2.1 three inputs are used. The inputs (in) variables are unique to each vertex, while the uniform variables are constant for all vertices. gl_Position is a built in OpenGL variable to hold the calculated position of the currently processed vertex. As seen from the source, both color and texture coordinates are passed on, unchanged, as output variables in this case. However, running these variables through the vertex shader like this enables interpolation between vertices. For instance, if a triangle is to be drawn by three vertices covering the entire screen, and the leftmost vertex is colored blue while the rightmost vertex is colored red, the resulting triangle color will gradually shift from blue to red (looking from left to right).

After the vertex shader stage, data is forwarded to the tessalation and geometry shaders if implemented, if they are excluded data goes directly to primitive assembly and clipping. Transform feedback may be added to the pipeline to extract data calculated by the aforementioned shader stages. This is useful if the GPU is to be used for mathematical computations in addition to, or instead of, rendering. Since GPUs typically consist of thousands of cores running the shader programs in parallel, it provides great parallel computing power that may, for instance, be utilized by scientists to process data.

The primitive assembler construct triangles, lines, quads or points from the vertices provided, depending on what is instructed by the draw command. The clipping stage removes any vertices located outside viewing volume, as defined by the projection matrix. Any vertices located outside are discarded. Geometric primitives originally consisting of discarded vertices are corrected by adding vertices at the edge of the viewing volume. Clipping is commonly described as a separate stage, but mathematically it happens during the calculation of gl_Position in the vertex shader **while** the point is transformed by the projection matrix [48].

After clipping the geometric primitives, which are now only contained in the unit volume defined by the projection matrix, are forwarded to the rasterization stage. The rasterizer converts the 3D graphics into 2D graphics that can be displayed on screen. This is achieved by projecting the unit volume onto a 2 dimensional plane, dividing all geometry into pixel sized fragments. Fragments are then forwarded to the fragment shader, which is the final programmable stage of the pipeline.

The fragment shader is mandatory, and is responsible for coloring each fragment / pixel supplied by the rasterizer, hence, it processes each fragment individually. The colors are derived from interpolated values calculated during the vertex shader stage. If a texture is bound, the color is extracted from the texture based on the interpolated U, V coordinates.

Listing 2.2: Fragment Shader as implemented in the client application

```
#version 150
in      vec4      oColor;
in      vec2      oTexCoord;

uniform sampler2D Texture0;

out     vec4      FragColor;

void main() {
   FragColor = texture2D(Texture0, oTexCoord) * oColor;
};
```

As seen from 2.2 two inputs are used in this implementation, matching the outputs of the vertex buffer. A uniform sampler2D is used to provide textures to the fragment shader. FragColor is the OpenGL built-in variable to be assigned the color of each fragment. In this case, both color and texture are required. Therefore, if an object is to be textured the color value must be set to pure white ([1.0, 1.0, 1.0]) in order to not interfere with the resulting texture color. Similarly, if an object is not to be textured, a white texture must be provided in order to get the fragments colored as desired.

After the fragment shader is done coloring, the fragments are tested depending on what is enabled. Depth testing and alpha blending is enabled in this project. Depth testing makes sure that if multiple objects are located in front of each other, the front-most object is what will be drawn to the framebuffer. Alpha blending ensures that fragments with an alpha value of less than one are properly blended with underlying fragments, such that the effect of transparency is achieved.

When tests are done all fragments (not masked by an optional viewport restriction) are written into the bound framebuffer, which is a portion of memory to contain pixel values to be displayed on screen.

2.5.2 The View Matrix

This matrix is used to transform points from the world-space into view-space, where view-space defines a space that is relative to the view of the camera through which the 3D scene is to be observed.

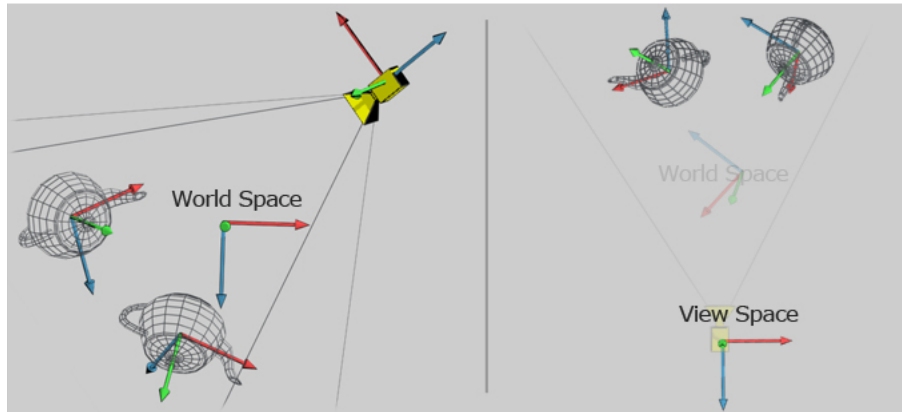


Figure 2.9: The virtual world as seen in world space, and how the vertices of the world space are transformed by the view matrix to result in the view space.
Image Credits: Coding Labs

In computer graphics the camera, or eye through which the scene is observed, is placed at a fixed location and does not move. Rather, the notion of movement in a 3D environment is achieved by transforming vertices of the entire scene with the inverse of the camera's transformation matrix. Meaning, it is not the camera that moves, but the entire world moves around the camera. The inverse of the camera's transformation is what is known as the view matrix.

As every vertex of the scene is multiplied with the view matrix, transforming the vertex into its camera relative position, applying the matrix is usually done by assigning it as a uniform value in the vertex shader of the rendering program.

[57]

2.5.3 The Projection Matrix

The projection matrix is a 4x4 matrix which is multiplied with all points within the camera-space (defined by the view matrix), in order to define which points to project onto a two dimensional canvas during rasterization. Essentially, the matrix defines a viewing frustum into the virtual world from the cameras perspective, where all points that are contained within the frustum is to be displayed on screen. The frustum is defined by desired field-of-view (FOV), aspect ratio, near- and far clipping planes.

Two common types of projection matrices are used in rendering; orthographic and projection. The orthographic matrix equally scales the $[X, Y]$ values of the 3D points, such that the 3D objects distance from the camera does not influence its size. The opposite is true for the perspective matrix; objects further away from the camera appears smaller than similar objects closer to the camera, such as in the real world, see Fig 2.11.

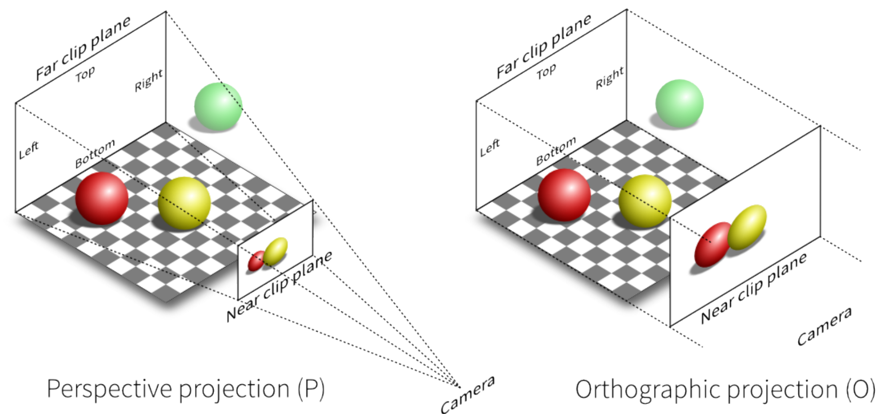


Figure 2.10: Comparison of orthographic and perspective projections
Image Credits: Glumphy

The orthograpic matrix is commonly used for 2D graphics, and thus perspective matrices are used in this project to make 3D models appear as intended.

The projection matrix normalizes its defined frustum such that x, y, z of each point contained within the frustum are scaled between $[-1, 1]$. Each point with a coordinate outside is discarded during the clipping stage of the pipeline. The normalization can be visualized as if the frustum

is truncated into a unit cube, naturally this leads to larger objects near the camera, and smaller objects further away from the camera.

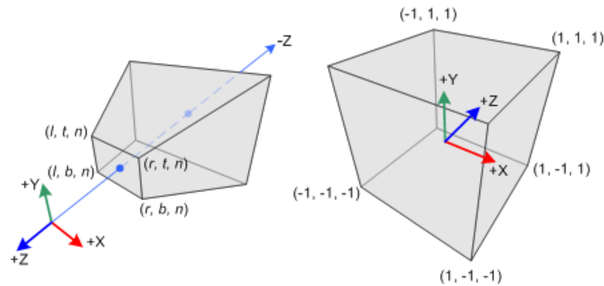


Figure 2.11: The projection matrix maps the frustum to a cube shaped canonical viewing volume *Image Credits: Songho.ca*

A typical perspective projection matrix, as included in older versions of OpenGL can be seen in Eq. (2.28) (in column major order).

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.28}$$

Where

n and f are the distance along the z-axis to the near plane and far plane, respectively

l, r, t and b is the distance from where the z-axis cross the near plane and to the left, right, top and bottom edges of the near plane, respectively.

[48] [12] [52]

2.5.4 Shapes

OpenGL does not include functions for rendering text nor shapes other than geometrical primitives. It is up to the programmer to combine geometrical primitives, such as triangles, points and lines, into more complex shapes and bodies [51].

Legacy OpenGL based its rendering on what is called immediate mode. This mode uses a fixed function pipeline, not containing any programmable shaders, providing low flexibility and poor performance as compared to modern OpenGL, which is based on programmable shaders and vertex buffers. Consequentially, functions related to immediate mode are marked as deprecated as of OpenGL v3.0 [39]. However, immediate mode did include functions for automatically generating vertices for geometric shapes such as circles and rectangles, which, unfortunately, is not compatible with the modern OpenGL rendering approach that is implemented in this project [48]. For this reason individual functions are programmed to generate basic geometric shapes, such as circles, squares, cubes and cylinders as described in 3.2.1.

2.5.5 Text Rendering

As with shapes, OpenGL provides no functions for rendering text. In order to render text, the programmer must supply vertices for each character, color, and optionally texture just as with any other 3D object. Naturally, building every character from geometric primitives (such as triangles) requires a lot of vertices, which again translates to a lot of tedious programming. Text rendering is a very common thing to do with any graphics API, and as a consequence there are many available programming libraries available to help automate the process.

The FreeType Library was chosen for this project, because it is well established and documented. FreeType does not have anything to do with OpenGL in particular, but has the ability to render text on to bitmaps using its own font rasterization engine. The library can read characters directly from most types of vector and bitmap font formats, and produce high-quality glyph bitmaps which then can be used as textures by OpenGL.

[56]

Chapter 3

Realization

This chapter describes the inner workings of the server-, client- and microcontroller application that was developed during the course of the project, and how they communicate. The microcontroller is used as a peripheral for the server application is therefore described in a subsection of the server application. The chapter is intended to be kept an abstraction layer above the actual code implementation, describing mathematical solutions and design choices rather than syntax. Full source code of all applications are provided in Appendix C.

Two concepts of manipulator operation have been implemented, termed Direct Mode and Record Mode.

Direct Mode

The manipulator follows the operators left hand movement in real-time.

Record Mode

A sequence of positions are defined by maneuvering a virtual manipulator in a point cloud representation of the robots work area. The virtual manipulator follows the operators left hand movement, while buttons on the joystick are used for adding / removing positions to the sequence, and to execute a registered sequence. When the sequence is executed the physical manipulator traverse its positions.

3.1 The Server Application

The main purpose of the server application is to control the robot and its peripherals according to data received from the connected client, in addition to supplying the client with requested data. Hence, it initiates and maintains local connections to the USB controller of the robotic manipulator, the microcontroller, the PTU and the Microsoft Kinect. The program is made in C++ using Visual Studio 2013. Due to compatibility issues with the Scorbot driver, it had to be built for x86 environments using Microsoft Foundation Class Library (MFC) [33]. Realization and implementation of the communication between the client and server is covered in 3.3, while the device implementations are described in the upcoming subsections.

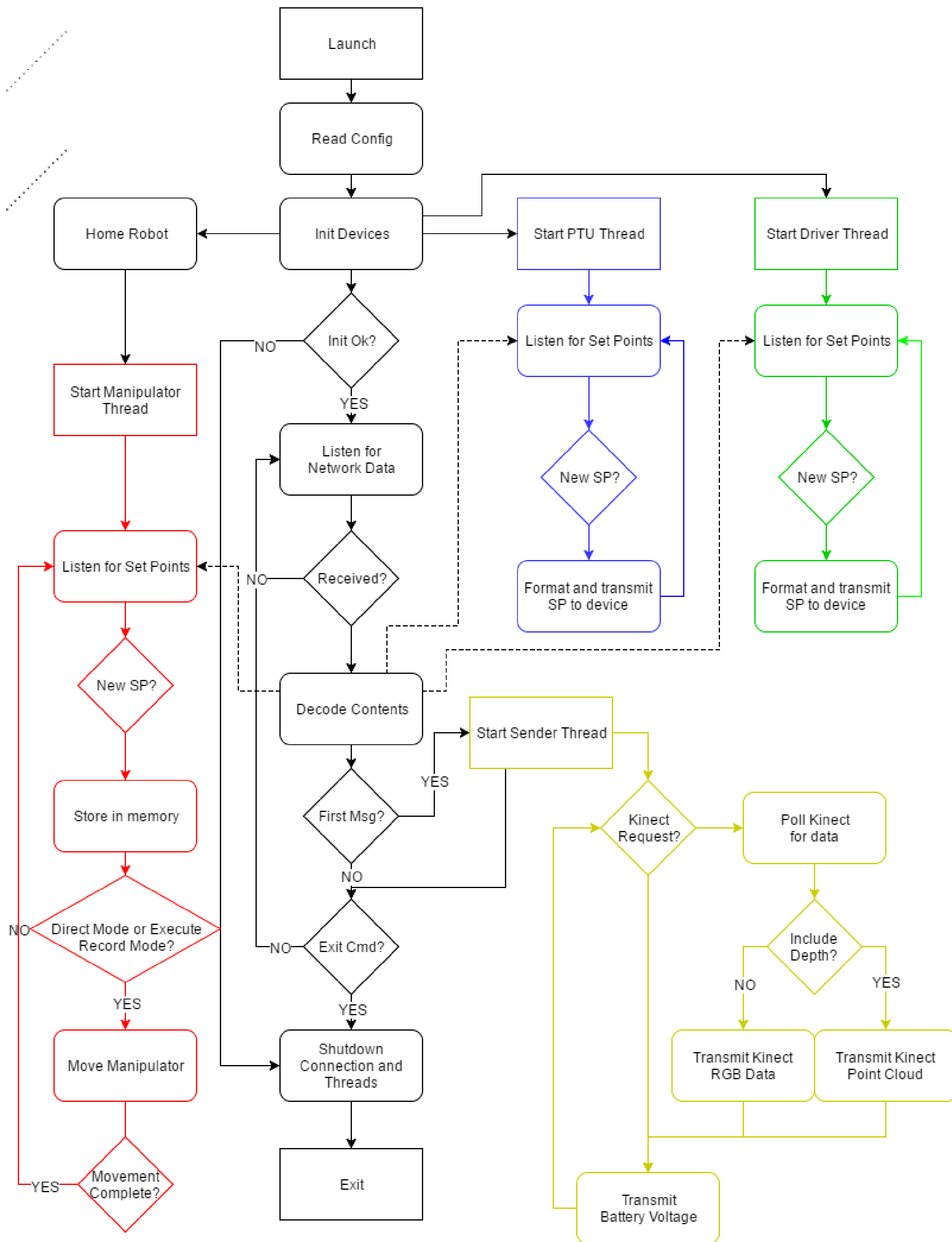


Figure 3.1: A simplified flow chart overview of the server application.
Colors represent separate threads.

3.1.1 Controlling the Robotic Manipulator

Code concerning the manipulator is abstracted into its own class, containing related functions and a private thread which is responsible for maneuvering the manipulator. Public class functions include initialization, reading of analog inputs on the USB controller and setting of position set points. In order to communicate with the USB controller the required files, described in library documents on C, must be put into their appropriate locations. The USBC.dll, USBC.lib, and .H - files must be linked and included under the project settings, and the parameter folder must be added as a sub-directory in the project. Since the default parameters for the robot was found to render it too weak to carry the PTU without causing impact detection errors, the default parameter folder has been replaced by the $3kg parameter folder found under the Scorbot installation directory.

The manipulator initialization starts by establishing a connection to the USB controller for the robotic arm, and setting the control mode to active. If the operation succeeds, three vectors are defined in the controllers memory that are going to store upcoming point coordinates. This reserves the necessary memory in the controller. One vector is intended to be continually updated during Direct Mode, another stores the home position, while the last vector may contain up to 200 predefined points to be mapped during Record Mode.

After memory is reserved, the robot is homed. The homing sequence is predefined in the controller and is a necessary step each startup as the manipulator may drift when not in use, resulting in inaccurate movements. Finally, the initialization process registers callback functions for movement, and starts the thread responsible for moving the robot.

The main thread of the application commands the manipulator thread through a SetPosition function of the manipulator class. This function stores given joint angles in the appropriate vector, depending on which command is passed along. Implemented commands are: A - Add point, R - Remove Point, D - Direct Mode, E - Erase all points, X - Execute sequence, C - Cancel sequence, H - Go Home, where A, R, E, X and C relates to Record Mode, D indicates Direct Mode and H returns the robot to its home position.

As the manipulator is found to be unable to receive a movement command, when motion is already in progress, the thread responsible for manipulator movement waits until motion is complete before handling a new position request. In addition, as gripper motion and joint motion is achieved by using two different functions, the calling of one function disables usage of the other function until its resulting motion is complete. The thread is therefore made to first perform gripper motion, before joints are moved into desired positions. Motions are flagged as complete by a thread of the USBC.dll calling the implemented MotionEnd function. Synchronization between the manipulator thread and the USBC.dll and Main threads of the server program is achieved by using mutexes combined with condition variables.

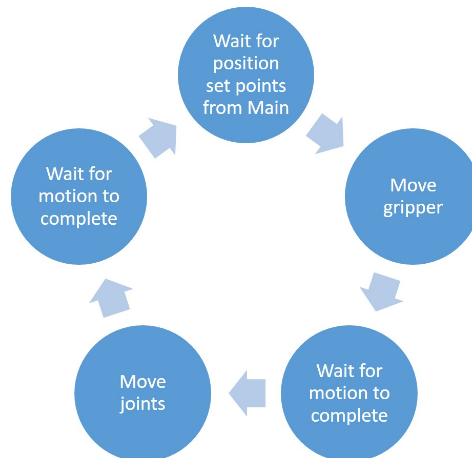


Figure 3.2: The flow of the manipulator thread.

3.1.2 Serial Communication

Both the PTU and the Atmel microcontroller communicate with the server through their own virtual COM ports. Hence, a common SerialHandler class was constructed, which is instantiated once for each serial device. As the name implies, a SerialHandler instance initializes and manages a connection to one COM port, defined by input parameters, by using functions of the Windows API. If the connection is successful a private thread for transmitting and receiving is started.

The thread is synchronized to public write / read functions by using mutex and condition variables, such that whenever the SerialHandler owner calls the write function the thread is awakened and performs the transmission. If the device responds with a message, this message is stored internally and may be accessed through the public read function.

Driving the Robot

The purpose of the Atmel XMEGA A3BU is to drive the wheels of the robot wagon according to set points provided by a connected computer. Originally Peter Aspunvik made a program based on the Atmel Studio USB example project, to do this job [2] as part of his thesis. However, this program turned out to be unstable and provide low flexibility.

There were two main problems presenting themselves during testing of his program.

1. The microcontroller expected sequences of characters to be received, character by character, in the correct order for anything to happen. If a character is lost in transmission, or received out of order, the entire sequence must start from the beginning. In addition, the logic was divided in two; one character sequence were to control the left side and another sequence the right side of the platform. Hence, in order to drive the robot forward, each side of the robot (left / right) had to be activated separately by their own character sequence, which may lead to poor coordination.
   For instance, the right side is told to move by first sending an 'h' (indicating "right side").

Next, either a '+' or '-' sign must be sent depending on the desired direction of travel (forward or backward). Finally, the travel speed is set by sending a number between '0' and '9'. The whole procedure then needs to be repeated for the left side, by starting the sequence with a 'v' instead of an 'h'. If new characters are not received within a deadline the microcontroller will restart its program entirely, which leads to the second problem.

2. The built-in watchdog functionality of the microcontroller is utilized, restarting the microcontroller if the watchdog counter is not continually reset before it reaches a given timeout. If the microcontroller was a standalone unit, not communicating with a host computer through USB, this is often a safe choice. However, when the microcontroller is used as an USB device a restart leads to a broken communication port between the computer and microcontroller. The result is that the microcontroller must be reconnected and the computer application restarted, in order to reestablish communication.

Consequentially, a new program was developed for the Atmel microcontroller, improving stability and eliminating the character sequence requirement. Additionally, functionality regarding both motor control and encoder readings are incorporated into the same microcontroller and not maintained by separate controllers as originally installed. The new program is based on the same Atmel USB example as used by Aspunvik [2], however, it now expects to receive a string containing a command ('SP' or 'T') and either speed, angle and optionally rotation or distance to be driven.

The values in the string are separated by using a token, which is used to extract all different set points from the string and store them into internal variables. Once a string is received its starting command determines which function is to receive the extracted values. 'T' (for travel) calls a function that drives the robot forward while monitoring the encoders, ultimately stopping the robot when desired distance is reached. The implementation makes sure the robot drives straight forward by using the encoder difference as angle set point. For example, if the encoder on the right side is 90 counts ahead compared to its left counterpart, only the motors of the left side are actuated, or if the left side is ahead by 45 counts encoder counts, the motors of the left side rotate half as fast as the motors on the right side.

If the command is ’SP’, one of two functions are called depending on whether the string contains a rotation set point or not. If there is a rotation set point the angle is discarded and the speed is scaled and applied to each pair of wheels in opposite directions, making the robot rotate. Whether it rotates clockwise or counter-clockwise is determined by the rotation set point.

If there is no rotation set point, or if it equals zero, the speed is applied in the same direction to both the left and right, but scaled down for the side towards which the robot is to turn. The angle determines how much the speed is to be reduced on the turn-to side by the formulas (3.2) - (3.4), depending on the quadrant of the angle.

$$Speed_{Steering} = Speed - \frac{Speed}{90} * Angle \tag{3.1}$$

$$Speed_{Steering} = Speed + \frac{Speed}{90} * Angle \tag{3.2}$$

$$Speed_{Steering} = \frac{Speed}{90} * (|Angle| - 90) \tag{3.3}$$

$$Speed_{Steering} = \frac{Speed}{90} * (|Angle| + 90) \tag{3.4}$$

In the server application, code concerning the Atmel controller is abstracted into its own class named RobotDriver. The class is instantiated into a RobotDriver object, accessible from the main part of the application, and functions may be called on the object to send either driving set points or target distances.

Avoiding unintended motion is achieved by implementing a timer as a watchdog in the microcontroller. The timer resets the PWM outputs to zero if no new message is received within 0.5 seconds, by calling a bound interrupt service routine. Using a timer instead of the built in watchdog timer solves the problem mentioned above, present in Aspunvik’s application.

The Pan Tilt Unit

The PTU is controlled by the server, similar to how it was controlled in the previous telepresence project; by transmitting string commands [50]. The device is instantiated in a separate object,

accessible from main, and starts a SerialHandler object when initialized. After initialization a function for setting desired target direction may be called. This will construct a string containing code words and calculated target steps for both the pan and tilt, before forwarding to the SerialHandler.

Resolution of both stepper motors are set to micro stepping of eighth steps. This result a movement resolution of 46.2857 seconds arc per position. Converting seconds arc into degrees yield:

$$1 secarc = \frac{1}{60 minarc \cdot 60 secarc} = 0.0002778^{\circ} \tag{3.5}$$

$$DegreesPerStep = 46.2857 * 0.0002778^{\circ} = 0.01285816746 \tag{3.6}$$

Hence, necessary steps in order to reach a given angle is calculated by the following:

$$Steps_{Pan} = \frac{\theta_{Pan}}{DegreesPerStep} \tag{3.7}$$

$$Steps_{Tilt} = \frac{\theta_{Tilt}}{DegreesPerStep} \tag{3.8}$$

A hysteresis of $\pm 0.01 rad$ is implemented in order to avoid small and involuntary head movements being propagated to the PTU. Please refer to the manual and datasheet of the PTU for elaborations concerning commands and calculations C.

3.1.3 Monitoring Battery Voltage

The USB controller for the Scorbot comes equipped with 4 built-in analog inputs, one of which was used as a battery voltage monitor. These analog inputs read voltages between 0 to 10 VDC with a resolution of 1 byte, such that the voltage range is mapped to values in between 0 and 255.

Since the battery voltage is about 12.5 VDC when fully charged, the voltage was scaled down by using a voltage divider. Two resistors of 20 000 Ω were used, effectively halving the voltage from the battery to the analog input.

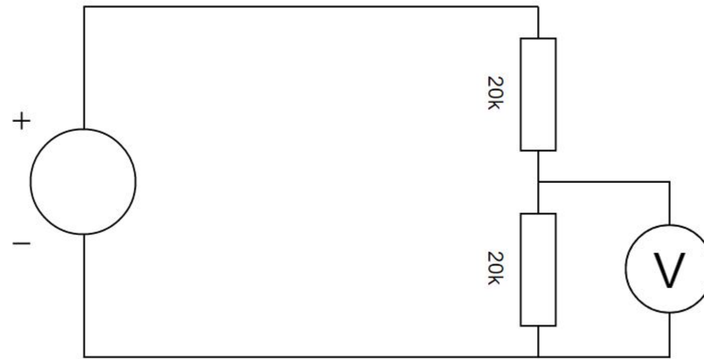


Figure 3.3: Voltage divider schematics

The driver for the USB controller provides two options for reading the analog input; implementing an asynchronous callback function or polling of the analog input. Polling became the solution of choice, since the measurement is only necessary when transmitting the value to the client.

3.1.4 Kinect Implementation

The Kinect is implemented in its own class, which is instantiated at start-up of the server program. The class contains functions for initialization, shutdown and grabbing of image data. Interacting with the device itself is realized by using functions of its SDK, hence, the SDK is linked and included in the application. The device is initialized by first checking if a Kinect is physically connected and operational. If a Kinect is found, a sensor object is instantiated containing two separate data streams; one for retrieving depth images and one for color images. After a successful sensor initialization a separate function may be called to grab Kinect frames.

Frames are grabbed by querying the streams of the upcoming depth and RGB frames, storing them into separate buffers when received. The buffers are then locked (write protected), before iterating over all of the pixels contained in the depth buffer. The depth value and current row / column for each pixel is then passed to a function of the Kinect SDK in order to find corresponding RGB values from the color buffer. Finally, depth and color values are stored sequentially into buffers allocated on the heap, which is to be accessed by the servers UDP transmitter thread. Functionality regarding the network transmitter is covered in 3.3.

The Kinect provides depth in addition to the traditional R, G, B values for every one of its 640x480 pixels, amounting to approximately 1.2 million values for every captured frame. In order to keep frame size at a minimum, the data types used for storing values become important. Originally, the Kinect SDK provides a function for retrieving the X and Y coordinate for each pixel of the depth image as well, such that a complete position vector with $[X, Y, Z]$ is generated for every pixel. The returned position vectors uses float as data type (occupying 4 bytes), resulting in the following size for each frame:

$$640 * 480 * 3 + 640 * 480 * 3 * 4 = 4608000 bytes \tag{3.9}$$

Frame rate is increased by reducing the frame size. This is achieved by implementing the functionality for defining complete position vectors at the client side. The same approach as the Kinect SDK uses is implemented from scratch, so that the client does not require the Kinect SDK.

Since color is retrieved as 3 bytes (values between 0-255 for R, G, B) and depth as an unsigned short (two bytes), two separate buffers are allocated on the heap during initialization; one for depth values and another for R, G, B values.

This result in a total byte consumption for each frame equal to:

$$640 * 480 * 3 + 640 * 480 * 2 = 1536000 bytes \tag{3.10}$$

3.2 The Client Application

This application is what the operator uses in order to interact with the robot, and is where the user interface of the system is implemented. It implements the Oculus Rift, Leap Motion and joystick devices for vision and control of the remote system. The application is structured by dividing related code into different classes and files, as visualized in 3.4.

A brief description of each part in the simple overview:

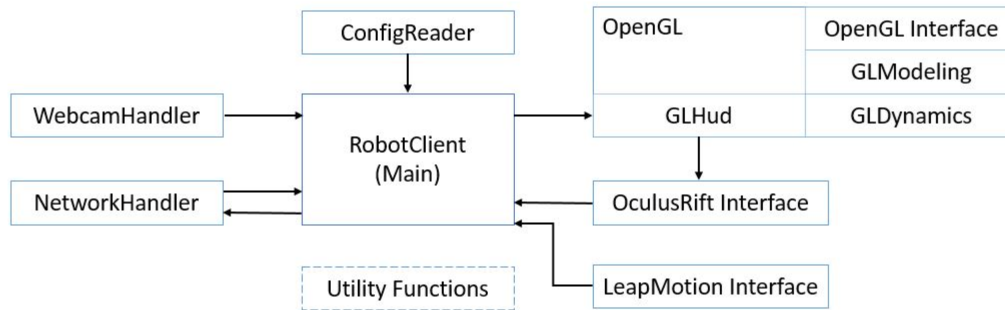


Figure 3.4: Simple overview of the client structure

RobotClient (Main) Ties all parts of the program together. Manages initialization and shutdown of all components, runs the main loop of the program and manages window creation and grabbing of joystick inputs. Also includes functions for calculating inverse kinematics, coordinate constraints and head-rotation compensation for the robot's coordinates.

Leap Motion Interface An instance of this class manages one Leap Motion device. Provides a function which return the position of any registered left hand.

Oculus Rift Interface A class where each instance of the class manages an Oculus Rift. Contains functions specific for the device, such as initialization, reading head-angle measurement and submission of pre-rendered frames to the device.

OpenGL A separate class is instantiated, which manages rendering and modeling of all graphics. The source of the class is large, and is subdivided into separate source files; GLDynamics (containing functions for updating graphic models), GLHud (functions for model initialization, and adding of static geometry), GLModeling (functions for shape- and model construction) and OpenGL Interface (OpenGL specific functions, shaders and rendering), in order to make the code easier to read and maintain.

NetworkHandler A separate class containing a thread for receiving data, and a function for transmitting data on the network. Contains functions for getting last Kinect point cloud or image, and last received battery status.

WebcamHandler A class where each instance manages a webcamera device. Hence, two instances are used in this scenario.

ConfigReader A class for reading the config-file, containing IP-addresses and startup constants.

UtilityFunctions A simple header file, containing generic functions used in multiple classes.

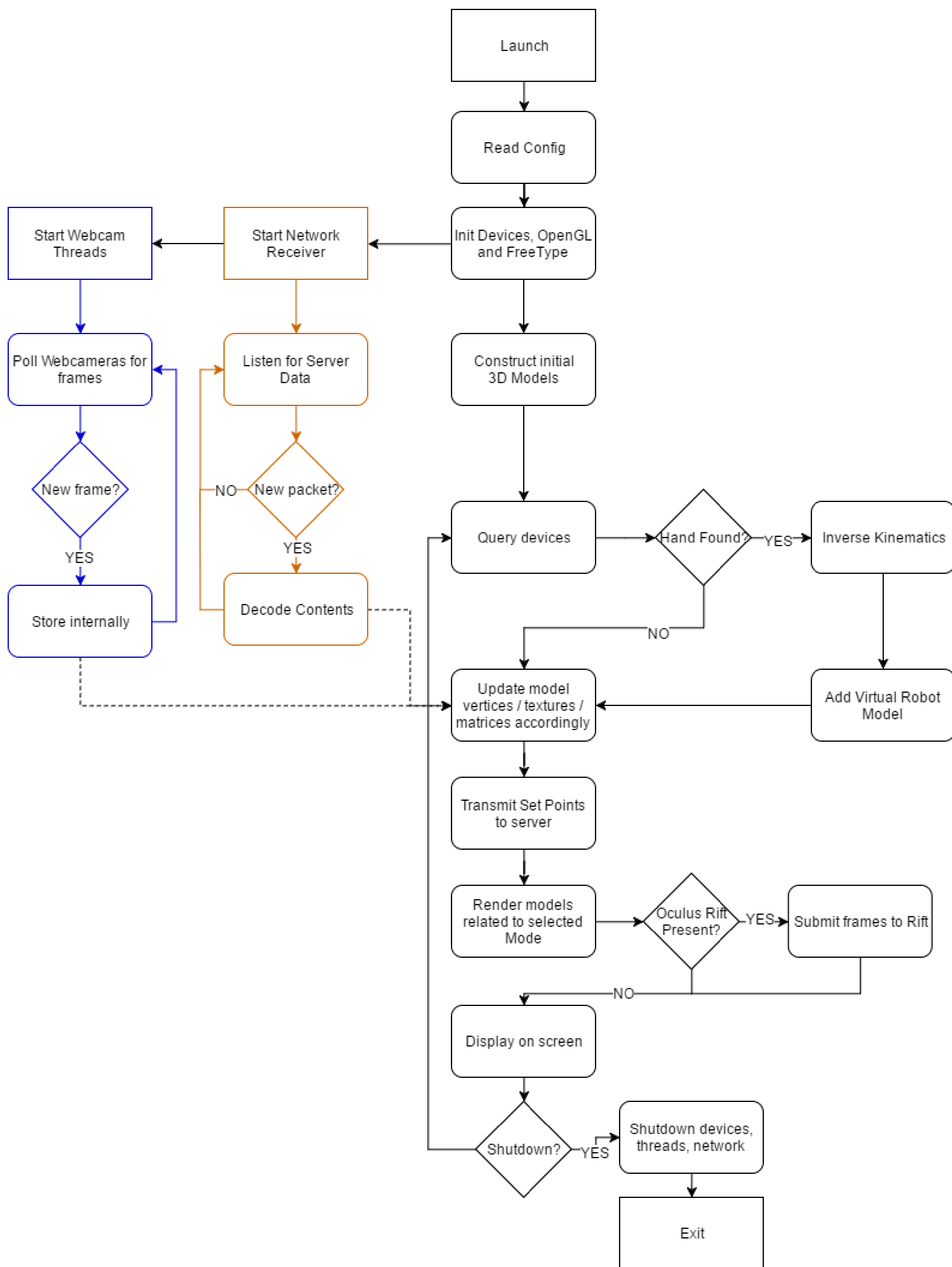


Figure 3.5: A very simplified flow chart overview of the client application. Colors represent threads.

A more detailed walk-through will be provided in the subsequent sections.

3.2.1 The Heads Up Display

The HUD of the previously developed telepresence system was completely renewed. Originally, the application used OpenCV functionality to paint HUD elements directly on top of the 2D camera frames. This, however, turned out to be suboptimal for two main reasons:

1. No functionality for adding HUD elements of 3 dimensions.
2. OpenCV uses the CPU for its paint functions, not utilizing the GPU.

The decision was made to create a 3D model of the robot representing its physical position in the virtual environment of the HMD. This allows the operator to intuitively observe his motion instructions in real time, enabling him to plan his actions before issuing any movement commands. This turned out to be a crucial feature of the system, as mentioned in chapter 4.1.1.

During development of the virtual manipulator extensive CPU usage of the OpenCV drawing functions became an issue, as these functions delayed the rendering loop enough to make 3D model movement appear choppy. In the telepresence solutions previously developed ([50]) this was not observable, as no motion was to be rendered. As a result, the entire OpenCV HUD drawing functionality was scrapped in favor of using OpenGL to render all content.

Introducing Models

Rendering dynamic 3D models by using OpenGL's rendering pipeline introduced major changes to the original telepresence solution. Instead of just being painted on top of the camera frames, every HUD element now require its own vertex buffer, with its associated index buffer, texture buffer and model matrix.

In order to ease implementation of multiple 3D objects a Model structure was created, whose purpose is to encapsulate all necessary data for rendering one 3D model. Models (instances of

this model structure) are then added to a model collection, which are ultimately inserted into a list of model collections as indicated by 3.6.

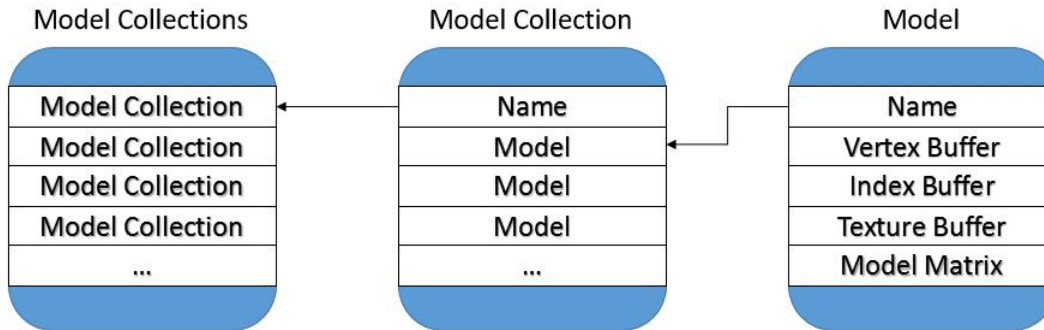


Figure 3.6: Model abstraction as implemented in the client program.

Constructors of the structure requires the caller to supply a name for the model, a valid vertex buffer, an index buffer, a model matrix and optionally a texture. Three unique constructors are implemented to cover 3D models, text models, and point cloud models.

The rendering function then iterates over every model in each model collection, drawing their vertices in the order given by the index buffer. The model is then scaled, rotated and translated into global location by multiplying every vertex with the associated model matrix.

Everything from shape to the color of each model is described by its vertices. One vertex contains 9 floats, $[X, Y, Z, R, G, B, A, U, V]$, where:

- $[X, Y, Z]$ defines the vertex position.
- $[R, G, B, A]$ Red, Green, Blue, Alpha describes the color and opacity of the vertex.
- $[U, V]$ defines the texture coordinates at the current vertex.

In order to ease creation of new 3D models, each with its unique vertices, the vertex generation has been abstracted into separate functions that generate vertices for different geometric shapes like circles, cubes, cylinders, planes and triangles. All 3D models are built as a collection of these geometric shapes.

Geometric Shapes

Shapes are generated by supplying programmed functions with a vertex list, an index list and a shape descriptor. The functions then generate vertices and indices and append them to the provided lists, according to the parameters provided in the shape descriptor. The shape descriptor is a structure containing: height, width, depth, radius, opaqueness, color, translation- and rotation matrix fields, where only the relevant attributes are given a value for each shape (radius for circle, etc).

All shapes are initially generated in regards to a local origin, such that the shape must be transformed into its model-relative position after creation. This is achieved by specifying the rotation- and / or translation matrices in the ShapeAttributes structure.

Flat Surfaces

The simplest shape implemented are the 2D planes used, for instance, to display the camera images. Its vertices and indices are concatenated to provided buffers, such that two triangles are defined to make up a rectangle.

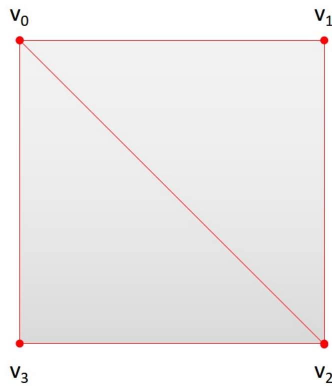


Figure 3.7: A rectangular surface as defined by vertices (red dots).

The four vertices are positioned to match the desired width and height of the surface provided in a ShapeAttributes structure.

$$v_0 = \begin{bmatrix} x_0 \\ y_0 \\ z \end{bmatrix} \quad v_1 = \begin{bmatrix} x_1 \\ y_1 \\ z \end{bmatrix} \quad v_2 = \begin{bmatrix} x_2 \\ y_2 \\ z \end{bmatrix} \quad v_3 = \begin{bmatrix} x_3 \\ y_3 \\ z \end{bmatrix} \tag{3.11}$$

Where v_1 to v_4 represent the top left, top right, bottom right and bottom left vertices of the plane respectively. The following indices are then added to the index buffer in order to paint two triangles from the four vertices: $[0,1,2,2,3,0]$. This essentially tells OpenGL during rendering that v_0, v_1, v_2 make up one triangle, while v_2, v_3, v_0 makes up another triangle, hence, the order of indices is important.

Cubes

Cubes are made up of 6 flat surfaces making up the shell of the cube. The surfaces are sized and spaced relative to each other, depending on provided width, height and depth in a ShapeAttributes structure.

The 6 planes are constructed, two by two, and separated. Meaning, two surfaces are defined and separated in the xy-plane, two in the yz-plane and two in the xz-plane. Together they make up a cube. Therefore, a cube requires $4 \cdot 6 = 24$ vertices, and $6 \cdot 6 = 36$ indices.

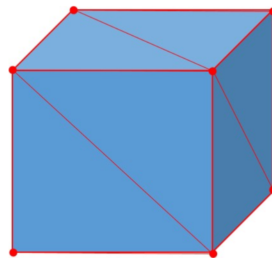


Figure 3.8: A cube as defined by vertices (red dots).

Circles

Vertices for a circle shape is generated by first placing a vertex at $[0,0,0]$, and a second vertex at $[r,0,0]$ (where r is the supplied shape radius). A for-loop is used iterating over 360 degrees, in which each loop-cycle rotates the radius point by 1 degree and adds the rotated vertex to the vertex buffer. $[U,V]$ coordinates are calculated for each vertex, such that textures are displayed

on the circle as illustrated in 3.9.

The following formulas are used to calculate the vertex parameters:

$$\begin{bmatrix} X_v \\ Y_v \\ Z_v \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} X_{v-1} \\ Y_{v-1} \\ Z_{v-1} \end{bmatrix} \tag{3.12}$$

$$\begin{bmatrix} U_v \\ V_v \end{bmatrix} = \begin{bmatrix} 0.5\cos\theta + 0.5 \\ -0.5\sin\theta + 0.5 \end{bmatrix} \tag{3.13}$$

$$\theta \epsilon [0, 1, \ldots, 360] \tag{3.14}$$

Where θ represents the current angle, which is incremented by 1 degree for each loop-cycle.

Indices are added to define triangles representing the circle according to 3.15. As shown, the center vertex is part of every triangle, while the two other corners indices are iterated to match vertices along the circumference.

$$\begin{bmatrix} I_n \\ I_{n+1} \\ I_{n+2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 + \frac{n}{3} \\ 2 + \frac{n}{3} \end{bmatrix} \qquad n \epsilon [0, 3, 6, \ldots, 1080] \tag{3.15}$$

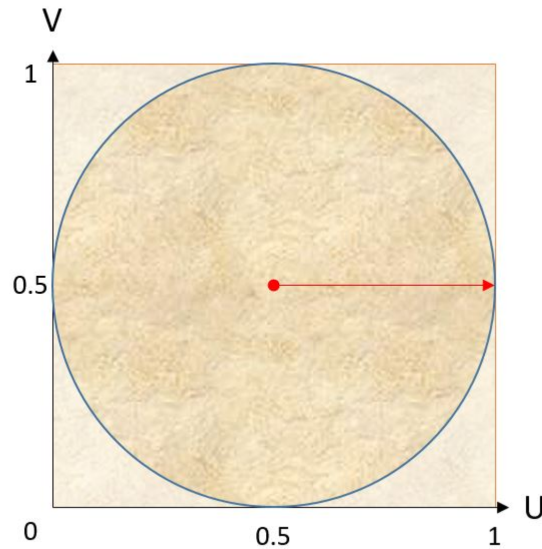


Figure 3.9: Underlying texture as it is mapped to the circle shape, using 3.13. The red line indicates the center- and initial radius vertices.

Cylinders

A cylinder is constructed by generating two circles (as described above) at different depths, which is defined by the supplied shape attribute structure. No additional vertices are generated, but indices are appended in order to define triangles between the two circles (named front and back) according to 3.16.

$$\begin{bmatrix} I_n \\ I_{n+1} \\ I_{n+2} \\ I_{n+3} \\ I_{n+4} \\ I_{n+5} \end{bmatrix} = \begin{bmatrix} I_{frontcircle}[1+\frac{n}{2}] \\ I_{backcircle}[1+\frac{n}{2}] \\ I_{frontcircle}[2+\frac{n}{2}] \\ I_{frontcircle}[2+\frac{n}{2}] \\ I_{backcircle}[1+\frac{n}{2}] \\ I_{backcircle}[2+\frac{n}{2}] \end{bmatrix} \tag{3.16}$$

$$n\epsilon[0,6,12,\ldots,2160] \tag{3.17}$$

Indices for two triangles are defined each iteration of 3.16, making up one square to be drawn between the circles. This is done for every degree of the circles, resulting in cylinder walls with a resolution of 1 degree.

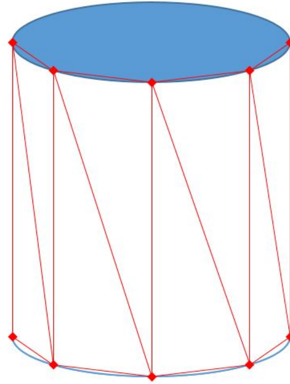


Figure 3.10: A close up view of additional indices for cylinder generation. Red dots represent vertices and red lines show the triangles they form, as defined by the extra cylinder indices. In the program 720 interlocking triangles, making up 360 squares, are indexed in this way per cylinder.

Triangular Prisms

In order to create arrow models, used for instance as part of the clock and speedometer models, a triangular prism shape function was included. The prisms are created by defining vertices for two triangles (top and bottom), which are sized and spaced according to width, height and depth parameters of the supplied ShapeAttributes structure.

24 indices referring to the 6 triangle vertices are then appended to the index buffer, in order to construct both triangles and all three walls of the shape.

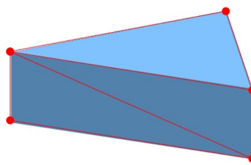


Figure 3.11: A triangular prism as defined by vertices (red dots).

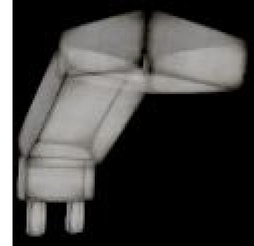
3D Models

All models are created as a collection of multiple shapes translated in relation to each other. Vertices generated by the shape functions are stored in one common vertex buffer, and corresponding indices are stored in one index buffer. After shapes are appended to the buffers, a model structure to contain the buffers, together with a model matrix, texture and coloring information, is instantiated and appended to the model collection to be rendered.

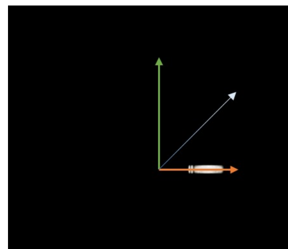
Screenshots displayed in the upcoming sections are taken from the left eye representation displayed on the monitor. Viewed through the Oculus Rift all models appear as 3D objects (i.e. the clock looks like a perfect circle, tilted slightly inwards), however, this is not possible to represent on paper.

The Virtual Robot

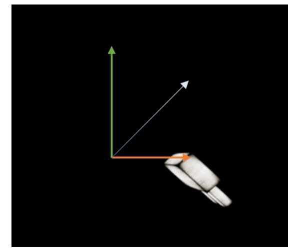
The virtual robot is constructed by adding multiple cubes in series into the same vertex buffer. Two cubes are added as fingers, while 3 other cubes are added to represent the wrist, elbow and shoulder. The base of the robot is intentionally hidden, in order to maintain a better view of the camera / point cloud. Positioning and rotation of the cubes is determined by supplied angle and finger distance parameters. The cubes are added sequentially to the vertex- / index buffers from front to back, i.e starting with the fingers, in order to ease upcoming transformations.



Construction of the virtual robot is illustrated in the screenshots 3.12 - 3.14. Naturally, the robot is only rendered when all vertices are in place, and not for each step as shown in the illustrations. The program was temporarily modified in order to render and display each step. All cubes of the robot are defined according to the Scorbots physical size in millimeters. Since the virtual environment operates in meters, a scaling matrix is multiplied into the model matrix in order to convert millimeters to meters in the vertex shader.

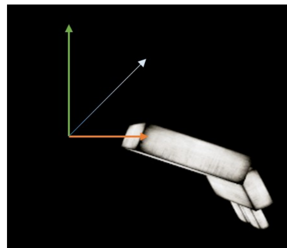


(a) Step 1: Fingers

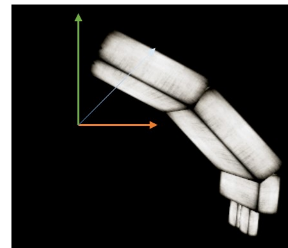


(b) Step 2: Wrist

Figure 3.12: 1: Two finger cubes are added to the vertex buffer (VB), spaced according to provided finger distance, and translated to make room for the upcoming wrist.
2: Wrist cube is added to the VB. All vertices contained in the VB are rotated according to provided pitch angle, and translated to make room for the upcoming elbow.

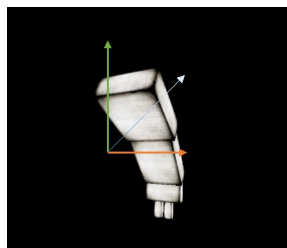


(a) Step 3: Elbow

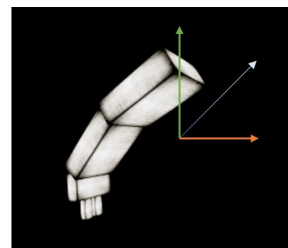


(b) Step 4: Shoulder

Figure 3.13: 3: Elbow cube is added to the VB. All vertices are rotated according to provided elbow angle, and translated to make room for the upcoming shoulder.
4: Shoulder cube is added. All vertices are rotated according to provided shoulder angle, and translated according to base size.



(a) Step 5: Base rotation



(b) Step 6: Model matrix transformation

Figure 3.14: Step 5: All vertices contained in the VB are rotated according to provided base angle.
6: All vertices are positioned, and so a model structure is instantiated and inserted into the model collection (or if a robot model instance is already present, the vertices of that model are updated). The final rotation happen in the vertex shader when multiplying with its model matrix. The matrix rotates it 90° around the Y-axis, to make it appear as if it is viewed from behind.

The Battery

The battery consist of two static cubes, and one dynamic. All three cubes are positioned relative to each other, using the same approach as with the virtual robot, before the final model is positioned and rotated according to its model matrix. The height and color of the dynamic cube is determined by provided capacity-parameter.

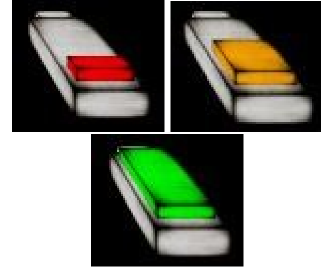


Figure 3.15: The battery model and its color representations.

The Speedometer

The object consist of a textured cylinder beneath a rotating arrow, made by combining a triangular prism with a cube. Additionally, digits are added to the center, indicating current drive speed in percentage of maximum velocity.

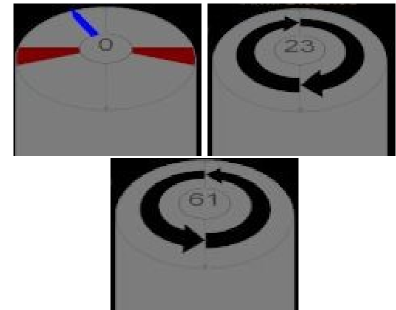


Figure 3.16: The speedometer

Each of the entities mentioned above are instantiated into separate model structures. The cylinder is separated because it is static, i.e. its vertex buffer is not updated during runtime. Since the vertices are equal for every frame, they only need to be loaded into graphics memory once, which is easily achieved by treating it as a separate model from the dynamic arrow. This reduces the amount of data being sent to the graphics memory, essentially improving performance. The applied texture is created in PowerPoint, and changes depending on whether the rotation button is pressed or not.

The arrow is instantiated in a separate model, by combining a triangular prism with a cube shape. Its vertices are rotated according to the angle provided through parameters, and a translation matrix moves it along the rotated axis such that its tip comes at the edge of the underlying

cylinder. The model matrix moves the arrow into its resprective screen position during rendering. When the arrow angle changes all vertices forming the arrow is rotated around the center of the cylinder, and updated in graphics memory.

The digits representing speed is instantiated as a normal text string and translated into the top center of the cylinder. When the speed set point changes the text is instantly updated. Text rendering is covered in the upcoming section 3.2.1.

The Clock

This model is also split into two instances. One model contains the circular background textured with an analog clock, and another model contains the hour- and minute hand. The separation is done for the same reason as with the speedometer; only hands are dynamic, while the circular background is static.

Figure 3.17: The analog clock.

The background circle is generated by using the circle shape function, while the hands are constructed similarly to the arrow of the speedometer (a cube shape + a triangular prism). Vertices of the hands initially added to show a time of 00 : 00 before being multiplied by rotation matrices. The matrices rotates them according to angles derived from current system time (see Eq. (3.18), (3.19)). Model matrices are used in order to position the clock in the bottom left corner of the screen, tilted slightly around the x-axis to make it easier to observe.

$$\theta_{Hour} = H \cdot \frac{360}{12} + \frac{m}{60} \cdot \frac{360}{12} \tag{3.18}$$

$$\theta_{Minutes} = m \cdot \frac{360}{60} \tag{3.19}$$

Where H is the current hour, and m is the current minute.

The Head Orientation Compass

Similar to previous models, this object is split into a static circular background and a dynamic arrow. The arrow consists of vertices generated by a cube shape and a triangular prism shape, and rotates according to the pitch angle derived from the Oculus Rift. The compass is activated when the operator enables the PTU, and always points towards the forward direction relative to the robot. This makes it easier for the operator to know which direction the robot will travel when maneuvering the joystick, as this might get confusing with the PTU activated.

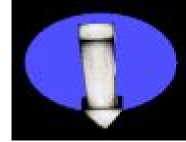


Figure 3.18: The head orientation compass.

Surface Models for Camera Display

In addition to the aforementioned 3D models, 2D surface models are defined and textured with camera images. Two surfaces models are defined and rendered individually to each eye for the IP cameras, such that the operator sees the image from the left camera with his left eye, and the image from his right camera with his right eye. A third surface is textured with the RGB data received from the Kinect. This surface is rectangular and rendered similarly to both eyes, as the Kinect does not provide stereoscopic images. All camera surfaces are sized according to the aspect ratio of the camera images.

Which models to render (i.e. which camera image to display) is selected by the operator during runtime, by using the POV buttons of the joystick B. Zooming is also implemented, by enabling the ZoomTex attribute of the ShapeAttribs structure. This effectively adjust the U, V coordinates applied to the surface vertices, such that only the center part of each camera image is stretched out and used as texture on the surfaces. This require minimal effort from the system, as compared to the previously developed solution using OpenCV funtions for zooming.

Text Rendering

At program initialization the FreeType Library is used to iterate over all glyphs in the font-file (font-file path being defined in the applications config-file), converting them into bitmap representations and loading these into the graphics memory as textures. For each character a structure is instantiated, which contains a reference to the texture in graphics memory, size- and positional information. The character structures are then mapped to a list of characters for later use, using the byte (char) representation of the glyph as map key.

Rendering of text is then achieved by generating one unique model for each character, such that a model collection represents one string. Each glyph is displayed by pasting its corresponding texture on top of a rectangular plane, which is positioned relative to its position in the text and character related size information. Since the glyph textures are stored as alpha values, and alpha blending is enabled, parts of the underlying plane which is not covered by the glyph becomes transparent.

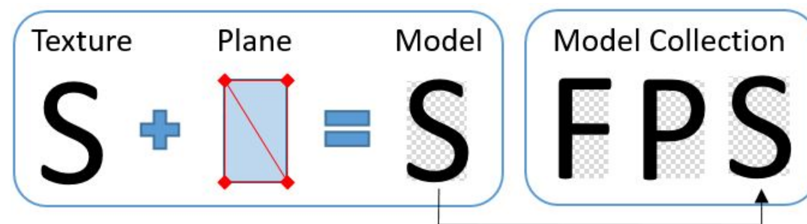


Figure 3.19: Illustration of the implemented text modelling principle

3.2.2 The Rendering Loop

Rendering objects of 3 dimensions to the Oculus Rift required a change in strategy as compared to the previous project. As described under View Matrix in chapter 3.1.2 of [50], the previously implemented solution put the virtual eyes 60 meters apart in order for each eye to only see one plane each - textured with the camera image corresponding to the left and right eyes. In order to perceive 3 dimensional objects in the Oculus Rift as intended (similar to the real world), the IPD of the virtual eyes must match that of the user. Hence, the previously implemented solution

was replaced.

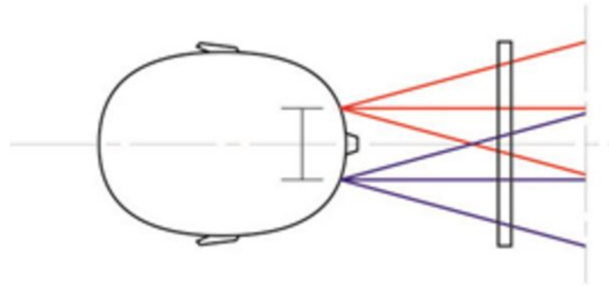


Figure 3.20: The view matrices of the system place the virtual eyes with a similar distance to the IPD of the user, in order to render 3D models correctly.
Image Credits: Oculus VR

The rendering loop is made such that it iterates over every model in every model collection of the model collection list. All models contained in the list are rendered according to the contents of their buffers to both eyes, except for the surface-models which display the IP camera frames. When rendering to the left eye, the surface model textured with the right camera frame is excluded, and when rendering to the right eye, the surface textured with the left frame is excluded. This result in the desired effect; the left eye only see the left camera and the right eye see only the right camera.

Except for the surface exclusions, only the view matrix and framebuffer targets change for each eye rendering. The projection matrices are unique to each eye, and generated by using functions of the Oculus SDK. The view matrices are generated such that it places each virtual eye 55 meters into the scene (along the z-axis), with the IPD distance in between (along the x-axis). The y-axis is set as "up" and the focus point of each eye is set to be parallel to the negative z-axis. Hence, it is rendered as if standing 55 meters along the z-axis looking towards the origin of the virtual coordinate system. The distance of 55 meters is chosen arbitrarily, but all models have been relatively positioned to this view.

3.2.3 Rendering the Kinect Point Cloud

The raw color and depth values of each pixel in the Kinect image is received from the server application through the UDP network connection. The NetworkHandler takes care of constructing vertices out of received Kinect pixel data, and adds them to a local vertex list. Position of the vertices are found according to the formulas of the Kinect SDK.

$$z = d >> 3/1000 \tag{3.20}$$

$$x = \frac{n-640}{2} * \frac{320}{640} * 0.00350100012 * z \tag{3.21}$$

$$y = \frac{-m-480}{2} * \frac{240}{480} * 0.00350100012 * z \tag{3.22}$$

$$Vertex = \begin{bmatrix} x \\ y \\ z + |0.06\cos(\frac{n}{640}\pi)| + |0.06\cos(\frac{m}{480}\pi)| \end{bmatrix} \tag{3.23}$$

Where m is the current row number, n is the current column number, d is the raw depth value and >> represents a binary bitshift of 3. The color values received are scaled from the interval $[0, 255]$ to shader values of $[0, 1]$, before being added as the color values of the vertex.

The formulas for x and y are extracted from the appropriate function in the Kinect SDK (NuiTransformDepthImageToSkeleton()) and implemented separately, in order to avoid including the SDK in the application (reducing the program's requirements and size). The factors added to the z value of the vertex compensates for the radial distortion of the depth camera lens. The values were found during calibration as described in the upcoming section.

Whenever an entire frame has been received a flag is set, indicating that the vertex buffer managed by the NetworkHandler is ready to be rendered. The buffer is then added to a model structure through calling a designated point cloud model constructor, and passed on to the rendering loop. Since the cloud consists of points, and no geometric primitives are to be drawn, it is rendered by using the DrawArrays(GL_ POINTS) function. This renders all vertices as individual

points, and does not require an index array specifying the vertex order (as is the case with all other models).

The point cloud model is only rendered as long as Record Mode is active. Once Record Mode is deactivated the entire model is deleted from the model collection, hence, the point cloud must be reconstructed when Record Mode is reactivated.

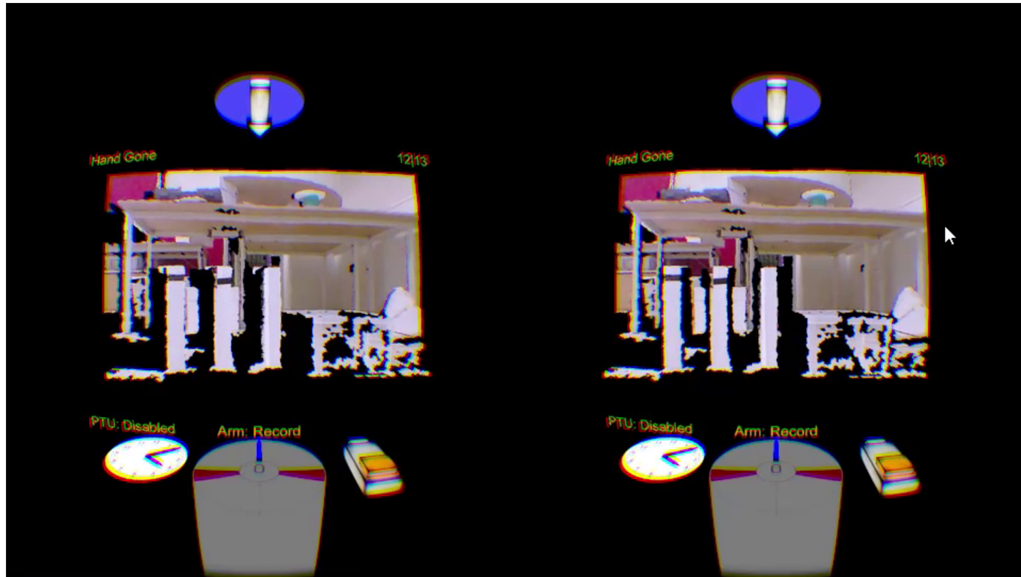


Figure 3.21: The Kinect point cloud being rendered during Record Mode.

The black areas as seen in Fig. 3.21 are caused by poor reflection of IR light (due to the angle of the floor, and the semi-transparent box on the right for instance) and the shadow resulting from the distance in between the RGB lens and depth lens. When viewed in the Oculus Rift the point cloud appears as one 3D representation.

Calibration

The rendered point cloud and manipulator is used as virtual representations of the real world, in which movement coordinates for the physical robot is to be mapped. Hence, it is crucial that the point cloud and the maniputalor is rendered in correct relation to each other. Coordinates registered in the virtual representation must result in the corresponding physical movement. For instance, if the operator registers a sequence of coordinates in which the virtual robot grabs an object in the point cloud, the physical robot must go to the same positions as intended by

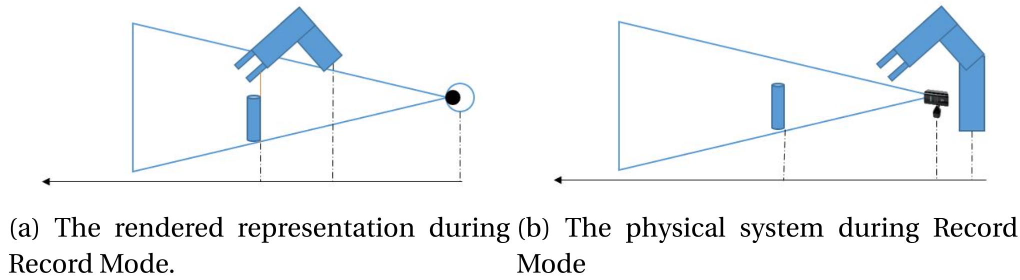
the operator - grabbing the object as registered.

Calibration was performed according to measured distances. There are multiple advanced procedures available on how to calibrate the Kinect to pinpoint accuracy. However, as described by Eirik Bjørndal Njåstad in his thesis [35], the Kinect is well calibrated out of the box, and calibrations result in very small adjustments. Resulting adjustments are too small to be of importance during operation in this project, as pinpoint accuracy is not possible when using hands for maneuvering anyway.

Positioning

The Kinect's depth vision has a minimum distance measured to be approximately 0.8m, such that all pixels with valid depth values are initially out of reach of the physical robot. This is solved by driving the robot forward using the implemented distance driver functionality, such that the physical robot is located at the same position as the virtual robot, relative to the area registered by the Kinect. The point cloud is rendered at the origo of the camera frame, such that the nearest points are 0.8 virtual meters into the scene. The virtual Scorbot is rendered with its center 0.5 meters into the scene, such that it is able to reach objects seen in the point cloud. Since the physical robot is located approximately 0.2 meters behind the Kinect, the robot is driven 0.7 meters at execution of a recorded session, in order to align the virtual and physical manipulators.

The positioning was tuned by testing and error correction. If the robot missed its target, for instance moving 5 cm too far, the point cloud model was translated 5 cm towards the virtual viewpoint, resulting in smaller joint angles stored. Initially the system was tuned relative to the center of the point cloud, and then tested along the edges, which unveiled that the cloud was slightly barrel distorted.



(a) The rendered representation during Record Mode.

(b) The physical system during Record Mode

Figure 3.22: The relative virtual and physical positions of the system during Record Mode

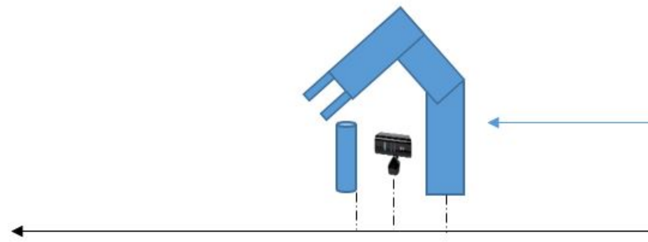


Figure 3.23: When Record Mode is executed the robot is driven into rendered position.

Depth Distortion

During testing positive radial distortion (barrel distortion) in the lens of the depth camera was observed and measured. Observations made it clear that depths registered near the edges of the depth image was registered as slightly more distant to the Kinect than they actually were, as opposed to depth values registered around the center. This was seen as mapping positions for the virtual robot in the point cloud edges resulting in an over extending physical robot movement.

Through measurement, it was found that the manipulator moved approximately 6 cm too far when operating in edges of the point cloud. Meaning, the registered joint angles of the virtual robot became too extensive as a result of the points being rendered too deep into the scene.

To compensate for the observed barrel distortion a simple pincushion distortion was applied to the depth values of the point cloud. Wrapping the edges of the point cloud 6 to 12 cm closer

to the camera view, keeping the center depth values intact.

$$z_{pixel} = z + |0.06\cos(\frac{n}{640}\pi)| + |0.06\cos(\frac{m}{480}\pi)| \tag{3.24}$$

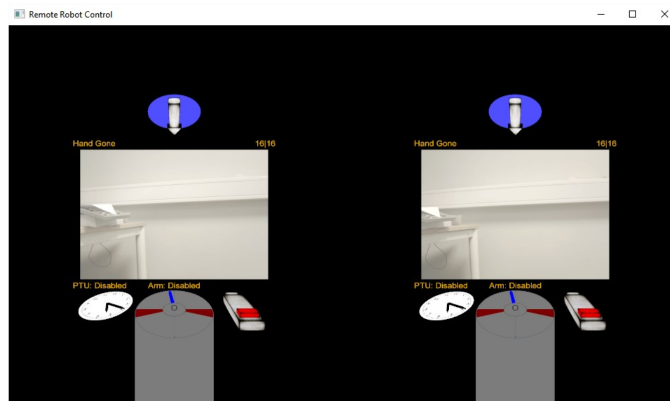
Compensating for the Oculus Lenses

When rendering 3D models by using OpenGL the effect of pincushion distortion and chromatic aberration in the Oculus Rift became visible. These effects were not visible in the previous system, possibly due to the camera images covering the entire field of view. When the camera images were moved into the scene, such that the edges of the image became visible, it was easy to see that the edges of the square were slightly arced. In addition, colors along the edge looked incorrect.

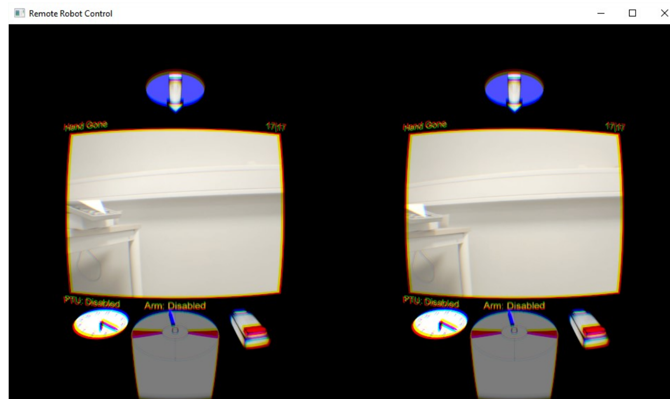
The solution was to no longer submit framebuffers to the Oculus SDK as Direct frames, but rather pass them as Eye FOV frames. When submitting frames as Direct, the SDK is instructed to simply display whatever is contained in the framebuffers supplied on screen. Passing frames as Eye FOV frames, however, enables the SDK's built-in correction algorithms that compensates for both the pincushion distortion and chromatic aberration of the Oculus Rift's lenses. Unfortunately, the built-in time warp function is also enabled, which is undesirable in terms of this application as the view into the virtual world is to be static. Time warp makes the Oculus SDK try to rotate the virtual view according to predicted head rotation in between rendering of frames. Since the scene is rendered using the same view matrix every time (not related to head pose), it looks like the entire scene is shaking when the operator moves his head back and forth.

The time warp effect is observable in the demonstration videos provided on C. It is what cause the image to appear to shake during head movements. The shaking appears more severe on the recording than how it looks in the Oculus Rift, however, due to both eyes being rendered in the same window.

In spite of this, the visual benefit of correcting the pincushion distortion and chromatic aberration surpassed the negative shaking effect induced by the time warp function, and so Eye FOV



(a) Direct / debug frame



(b) Eye FOV frame

Figure 3.24: With and without the pincushion distortion and chromatic aberration correction as performed by the Oculus SDK compositor. Since screenshots are taken of the computer monitor (and not through the lenses) it appears as if debug frames are optimal. However, the opposite is seen through the Oculus Rift.

was implemented.

3.2.4 Leap Motion Implementation

The Leap Motion is encapsulated within its own class, such that a Leap Motion object is created and called upon in the application. Initialization of the device verifies that the Leap Motion is connected and enabled in the Leap Motion control panel, such that it is usable in the application. If that is not the case, manipulator movement will be unavailable and a warning message will appear in the HMD.

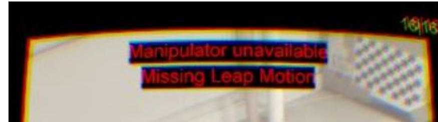
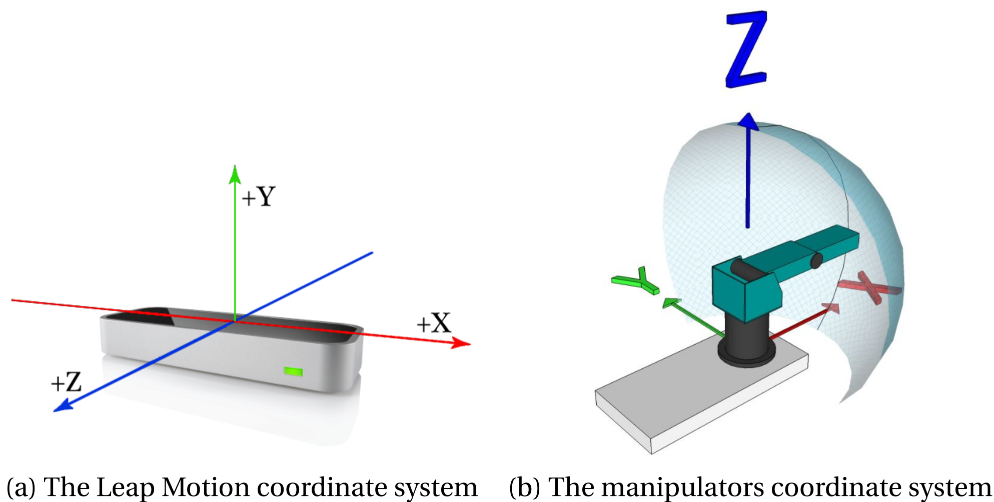


Figure 3.25: The Leap Motion error message.

When either Record Mode or Direct Mode is enabled during execution, the Leap Motion will be polled for new left hand positions by using functions of its SDK. If the internal algorithms finds a left hand in front of the device, it returns a Hand instance containing position-, orientation- and velocity information about the hand itself and all of its fingers.

A built in function in the Leap SDK is used to check how confident the algorithm is in that the returned hand object actually represent a left hand. If it is more than 60 % confident, the hand's coordinates is extracted. The coordinates equals the hand's position in millimeters from the device, in the Leap Motion's coordinate system. Hence, the $[X, Y, Z]$ as reported by the Leap SDK is used as $[Y, X, Z]$, to match the coordinate system as defined for the robot.



(a) The Leap Motion coordinate system (b) The manipulators coordinate system

Figure 3.26: The two coordinate systems

Compensating for Head Rotations

Even though the operator rotates his head in order to look around, the manipulator needs to stay in the same relative position as the hand. Since the Leap Motion is mounted in front of the Oculus Rift it follows the head rotation, such that the inertial coordinate system of the Leap

rotates when the head rotates. This head rotation result in changes to hand coordinates, even though the hand does not move.

To account for this the measured head angle from the Oculus Rift was used in order to counter-rotate the hand position around the inertial reference frame of the Leap Motion. Since this angle measurement is according to the Oculus Rift coordinate system the position is transformed into this system, which has its origin approximately 180mm behind the Leap system (center of the head), before being rotated and transformed back to the Leap Motion system again.

$$P_{Leap} = [x, y, z, 1]^T \tag{3.25}$$

$$P_{Oculus} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -180 \\ 0 & 0 & 0 & 1 \end{bmatrix} * P_{Leap} \tag{3.26}$$

$$P_{OculusComp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos\phi & 0 & sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -sin\phi & 0 & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * P_{Oculus} \tag{3.27}$$

$$P_{LeapComp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -180 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * P_{OculusComp} \tag{3.28}$$

Where ϕ is the pitch angle of the head and θ is the roll angle.

A demonstration video is put on C in which the arm is rendered with and without this compensation, showing the effect this correction has in practice.

3.2.5 Joystick Implementation

Implementation of the joystick in the client application was achieved by using the joystick part of the multimedia interface included in the Windows SDK [31].

The joystick is initialized in the program by the callback function WM_CREATE, which is received from the operating system when the application window is being created, and enables polling of the connected joystick. The polling continues throughout the execution of the program, making the operating system issue messages to the application if any movement or button presses is detected. Any received joystick messages is then handled in the Windows callback routine, where boolean flags are set which enables the various functionality of the joystick buttons (as illustrated in B). In case of driving, joystick position values get decomposed into usable values and form the basis of speed and angle calculations to send as set points for the robot.

Calculation of Driving Set Points

The subsystem for driving the robot is set up to be controlled by three variables: speed and angle or rotation.
Messages received from the joystick comes bundled with two parameters that says which (if any) of the two drive related joystick buttons are pressed, and the current x and y position of the stick itself. The joystick is set up to move in a squared plane where positional parameters are described by an unsigned integer of 16 bits, hence $[x, y] = [0, 0]$ is the top left corner, and $[x, y] = [65535, 65535]$ is the bottom right corner.

The $[x, y]$ values can then be used directly as angle and speed parameters for the robot. However, to make the driving experience more intuitive the square plane in which the joystick can move is reduced to a circle in software, where the current joystick position forms a vector with a length and an angle from the center. The vector length then determines the speed relative to the current max speed setting, and the vector angle determines driving direction. The max speed setting is determined by the throttle control on the base of the joystick. This makes the robot

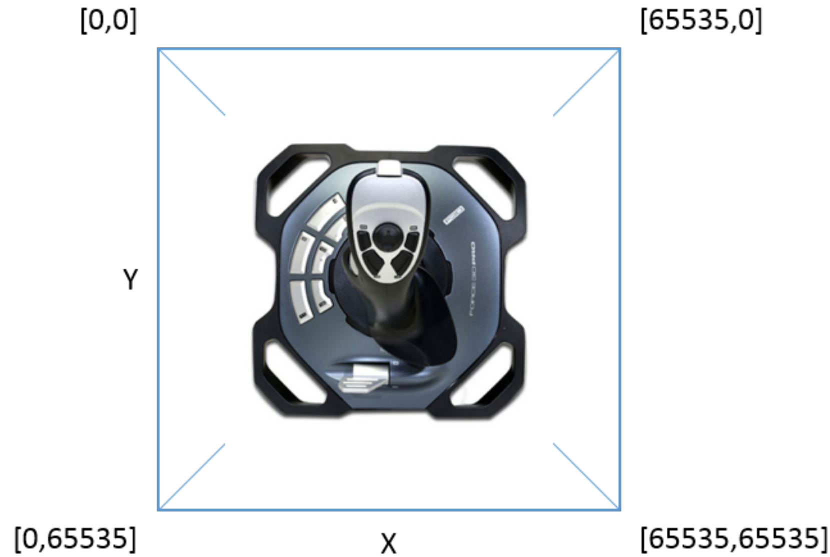


Figure 3.27: Illustration of joystick movement as reported by messages from Windows.

platform drive in the direction in which the joystick is pushed at desired speed.

To achieve this vector / angle control, the following sequence process joystick input.

1. The joysticks throttle position is scaled to a value between 0 and 1000, which translates to 0 - 100 % speed.

$$\frac{(Value - In_{Min})(Out_{max} - Out_{Min})}{In_{Max} - In_{Min}} + Out_{Min}$$

2. The x- and y axis are then scaled from 0 to 65535 to the interval $[-Speed_{Max}, Speed_{Max}]$ (as determined in the previous step) using the same scaling formula.

3. Considering the scaled x and y as coordinates to a vector, the length of this vector is found and used as speed parameter.

$$Speed = \sqrt{x^2 + y^2}$$

   If this length is larger than $Speed_{Max}$, which is the case when operating in the corners of the joystick's movable area, then $Speed = Speed_{Max}$.

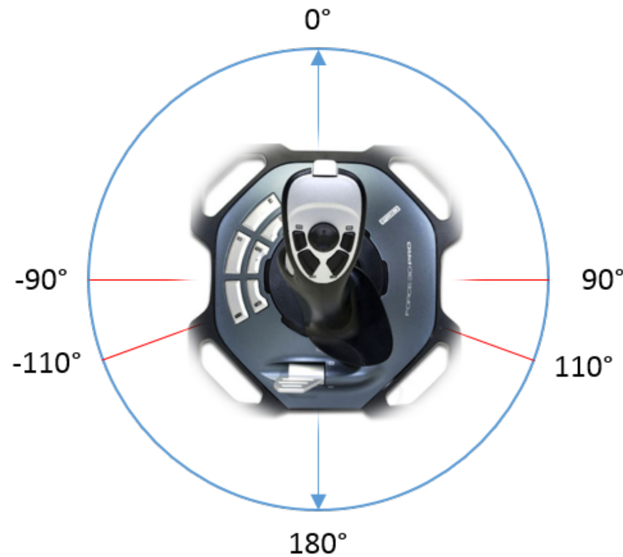


Figure 3.28: Illustration of the altered joystick implementation in software

4. The direction angle of the vector in degrees is then found.

$$Angle = atan2(x, y) \cdot \frac{180}{\pi}$$

   An angle of 0° is straight forward, ∓90° is maximum turning to the left and right respectively, |180°| is reverse, and from ∓110° to |180°| is reversed steering towards the left and right respectively. Dead-bands from |90°| to ∓110° is included in order to avoid an instant transition from fast forward to fast reverse on each steering side, reducing motor stress.

5. A dead-band of 10 % movement is implemented around the center of the joystick in order to minimize unintended movement. Hence, the following statement must hold true for speed to be set:

$$V_{Length} > \frac{Speed_{Max} \cdot 10}{100}$$

3.3 Communication

Communication between the client computer and robot system is realized by using two protocols; UDP and RTSP. RTSP is used to link the web cameras to the client application directly, while UDP is used to transmit set points and data regarding everything else between the client- and

server applications. Both applications have their own threads that listens for incoming packets, while the client have a function for sending data. The server continually sends data to the client during the session by instantiating its own data sending thread.

Communication parameters are provided by a separate Config.cfg file in the application directory of both the server- and client applications. Both applications read the contents of their Config file during start-up, and initializes communication and devices accordingly. IP addresses, Ethernet ports and COM parameters are defined in these files, hence, it is important that these settings are valid before executing the applications.

The client initiates the communication when sending its first packet to the server, as the server initially does not know the address of the client. When the first packet is received by the server it starts a sender thread, with the address of the first received packet as destination for its data. The sender thread then transmits battery measurement and Kinect data until the client is shut down. Kinect data is only transferred when requested by the client.

3.3.1 Client to Server

The client commands the server by transmitting a buffer containing set points and commands in one package. The first byte of the buffer contains the robot command (either A, R, E, X, D, H if an action is to be taken, or N if disabled). The second and third bytes contain Y if the PTU or driving is to be enabled, or N otherwise. The fourth byte tells the server which Kinect data to transmit; D - Depth image, C - RGB image only or N - None. The remaining bytes in the buffer is treated as floats, by using a designated pointer, in which all set points are stored.

Available set points are joint angles for all of the manipulator joints (base, shoulder, elbow, roll, pitch), the gripper distance of the manipulator, the pan and tilt angles of the PTU, in addition to angle, speed and optionally rotation set points of the robot platform itself. After the buffer is populated with values it is transmitted to the IP address and port of the server, which are defined in the config file of the program.

When the server receives the buffer it is passed to a function that extracts its contents into appropriate variables. The set points are then disseminated to the appropriate threads, for all enabled devices (determined by instructions of the first four bytes in the buffer).

3.3.2 Server to Client

Battery measurement and the Kinect image data is being transmitted from the server to the client application. Battery measurement is transmitted as long as a client is connected while the Kinect data is transmitted only when it is requested, in order to save bandwidth. The client may request either full depth image data or RGB image data of the Kinect, hence, three separate transmissions are implemented in the server's SenderThread. Each transmission use its own buffer, such that pixel data from the Kinect and battery measurement is sent in separate network packets.

In order for the client to identify which type of packet it receives, the first byte is set to indicate its contents; ASCII code 'B' for battery measurement, 'K' for Kinect depth image pixel data or 'C' for Kinect RGB image pixel data. The following structures are used for the different transmitbuffers.

The battery buffer contains 'B' and a byte who's value represent the battery voltage.

Buffer:	B	value
Byte:	*0*	*1*

Kinect depth frames are sent row-by-row and reassembled into the full frame at the client. This is done because UDP packets have a maximum size of 65507 data bytes, and an entire frame with depth and color values occupies in total $1xshort(depth) + 3xbyte(RGB) * (640 * 480) pixels$ = 1843200 bytes. Each row, however, occupies $1xshort + 3xbytes * 640 pixels =$ 3840 bytes, which is suitable for transmission. This also means that $65507/3840 \approx 17$ rows could be fit into one UDP package, but in order to avoid heavy packet fragmentation at the link layer

(due to an Ethernet MTU of 1500 bytes) each row is sent in its own packet. Hence, if a fragment is lost in transmission one image row is lost at the client, instead of an entire frame being lost, which is acceptable.

The Kinect depth row buffer contains 'K', followed by row number and its respective pixel values from left to right.

Buffer:	K	Row	D	R	G	B		D	R	G	B		⋯	D	R	G	B
Byte:	*0*	*1*	*4*	*6*	*7*	*8*	*9*	*10*	*12*	*13*	*14*	*15*	⋯	*3838*	*3840*	*3841*	*3842*

As illustrated by the byte numbering Depth (D) occupies two bytes as its data type is short, while the color values R, G, B occupies one byte each. The dummy byte for each pixel is a consequence of using a short pointer to store depth values into the buffer. The pointer, being of type short, may only target even numbered bytes (every second byte - i.e. pShort[1] = pByte[2] pByte[3] and pShort[2] = pByte[4] pByte[5]). Row No. is the current pixel row being stored into the buffer, and occupies one byte for each digit of the current row number.

The Kinect color row buffer is similar to the depth row buffer, except it does not include depth values and starts with 'C'. Since the depth value is excluded, two rows are fitted within one buffer during RGB transfer.

Buffer:	C	Row No.	R	G	B	R	G	B	⋯	R	G	B
Byte:	*0*	*1*	*4*	*5*	*6*	*7*	*8*	*9*	⋯	*3841*	*3842*	*3843*

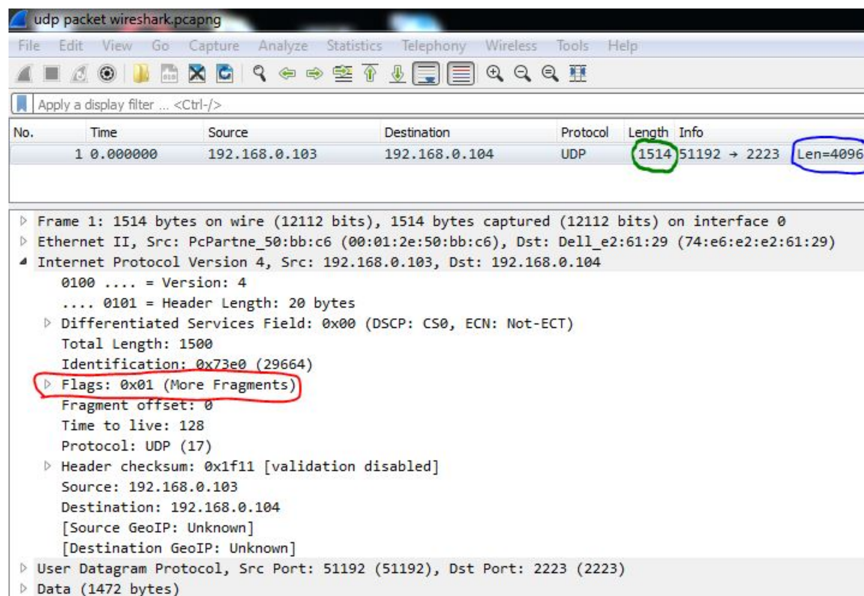


Figure 3.29: Package fragment as observed in the network sniffing tool Wireshark. 1514 is the length of the fragment + overhead, while 4096 is the total length of the packet. Only the first fragment is detected by Wireshark, as the other fragments are not recognized as proper IP packets.

Chapter 4

Results

Overall, the client- and server solution developed works as planned by combining robot control with remote vision using the equipment available. The resulting system and its usability will be presented in this chapter. A video provided on the DVD C shows the system in action. Please refer to this video for a proper demonstration of Record Mode, Direct Mode, driving and overall system functionality. The main result may be considered the source code itself, which is available for all resulting applications on the C.

4.1 Remote Control

The communication between the client and server applications work satisfyingly, enabling remote control as originally intended. There is no noticeable delay in set point transfer from the client to the server, however, the size of each Kinect frame constrain the update rate of the point cloud.

4.1.1 Leap Motion In Action

The Leap Motion performs well in terms of controlling the virtual robot displayed as a part of the HUD. There is no observable delay between left arm movement and resulting rendered robot movement, hence, this way of controlling the virtual robot feels natural and intuitive. The virtual robot turned out to be an important feature, as the instant visual feedback it provides makes

it much easier to predict the movement of the delayed physical robot. Running the system without this model made it practically impossible to control.

Overall, using the Leap Motion as a means to control a robotic manipulator proved to be quite challenging. There are four main reasons for this, aside from the large delay introduced by the Scorbot, which can be summarized as follows:

The tracking algorithm

The tracking algorithm is easily disturbed by external infrared light and white / reflective surfaces. Especially sunlight turned out to interfere with the Leap Motions measurements. As sunlight bounce of white objects in the FOV of the Leap, the algorithm often made the mistake of interpreting the object as a hand, passing on the objects position to the application, Fig. 4.1. Also, when staring at, and maneuvering the hand in front of, a white surface the accuracy deteriorated. This is possibly due to the Leap Motion being blinded by all the IR light reflection, making it hard to extract the hand from its surroundings, see Fig. 4.2.

Most of the time the SDK's confidence check discarded non-hand objects, but that was not always the case. Hence, windows and white / reflective surfaces at the office had to be covered. Optimally, the operator should sit in a dark room, with dark walls, to avoid anomalies regarding hand registration.

The point of view from the Oculus Rift

Having the Leap Motion attached in front of the Oculus Rift lead to difficulties in determining finger positions and hand orientation at certain poses. For instance, when keeping the fingers straight forward while moving the hand up, there comes a point when the palm covers the view of the fingers, see Fig. 4.3. Naturally, if the fingers are moved when not in view of the Leap Motion, the movement will not be registered properly. In addition, the Leap Motion struggles to recognize a hand without observing its fingers, which may result in bad position parameters and anomalies in its hand interpretation. The wrist roll angle reported by the Leap Motion was disabled as it proved too unstable, because of this.

Fatigue

The operator's arm becomes fatigued when maneuvering the manipulator over longer periods of time. This may become a problem if working in Direct Mode over longer periods. The fatigue leads to an increase in hand tremors, which propagates to the manipulator itself, leading to reduced accuracy and increasing the risk of making mistakes. Therefore, it is not recommended to stay in Direct Mode over longer periods.

Using the left hand

Because the left hand is chosen as the hand for manipulator control, it is easier to operate the manipulator in the left half of its work area. Naturally, operations in the right half requires significant stretching of the operator's left arm, leading to an increase in hand tremors and fatigue.

However, apparently practice makes perfect. After putting many hours into test driving and practicing on using the system, I became better and better at controlling it.

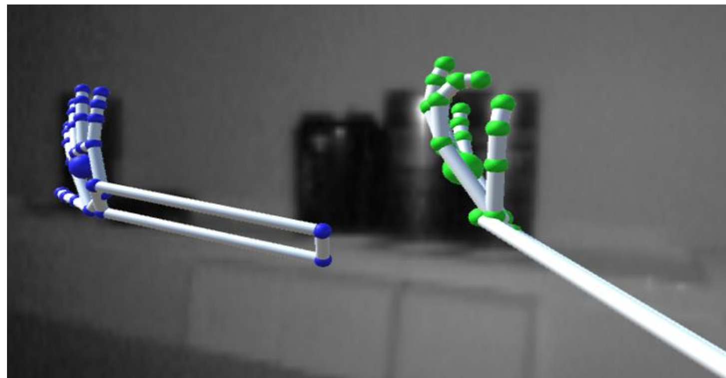
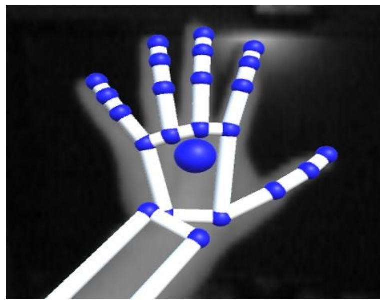
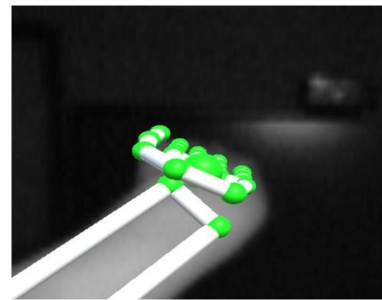


Figure 4.1: Soda cans interpreted as hands, because of their reflective surfaces.

Figure 4.2: Clearly visible hand not registered by the Leap Motion, due to its disturbing bright background.



(a) Hand Open - Good



(b) Hand Closed - Bad

Figure 4.3: As seen from the two screenshots, the Leap Motion fails to show correct finger positions when the hand is closed. Due to the POV from the Oculus Rift, the fingers are not seen when located in front of the palm. The Leap Motion still try to predict their position, but this often fails and lead to unrealistic hand postures.
Screenshots from Leap Motion Control Panel - Visualizer.

4.1.2 Direct Mode

Controlling the manipulator in real-time, which was the initial goal of the project, turned out to be very difficult. The reason for this not being the Leap Motion itself, but the robots lack of functions for direct control.

The Leap Motion works as intended, as it immediately returns registered hand position and orientation to the program when requested. The position is corrected for head rotation (maintaining hand position even during head movement), before inverse kinematics is applied to find corresponding robot joint angles. These angles, and the pitch / roll orientation angles of the

hand, are then used as basis for rendering the virtual robot, after being packed and transmitted to the server which instructs the robot to move its axis accordingly.

The aforementioned operations is completed within a couple of milliseconds. The problem is that after the server instructs the robot to move, it takes approximately one second before the robot reacts. This introduce a large delay to the loop, and makes Direct Mode hard to use accurately. All available movement commands supported by the robot driver was tested, but the result was always the same: approximately 1 second delay between function call and robot movement.

In addition to the delay, the robot turned out to be unable to update its target position during movement. Intentionally, Direct Mode was supposed to continually feed the robot with new target coordinates as the operators hand moved. This was not possible, as the robot had to finish its movement before it could receive and act upon a new target position. Issuing a new move command during motion resulted in errors. Alternatively, a force stop command could be used as new coordinates was accepted after a stop. However, the stop command was very aggressive (immediately halted all motors) and was excluded in order to avoid damaging the Scorbot.

Due to the poor performance of the Scorbot, Direct Mode was rendered difficult to use as a means to pick up and handle objects. It is possible, but requires much training in order to achieve. Personally, I was able to pick up and move objects by the end of the project after a lot of practice. During Direct Mode usage it is crucial to operate from a location free from external disturbances (white walls / sunlight), as faulty coordinates may have a direct consequence of movement.

If the manipulator crash with an object its internal impact detection will activate, disabling further movement and preventing damage. This detection works by comparing actual encoder counts with expected encoder counts for each joint, and if the deviation is above the impact detect threshold (defined in the Scorbot settings file), it will activate. Hence, the manipulator is safe to maneuver as it will protect itself from damage by crashing. However, fragile objects may

be broken if crashed into by the manipulator, and so care must be taken when maneuvering the manipulator.

Despite the difficulties of interacting with objects, it turned out to be a decent way of changing camera views. The IP cameras and PTU are are mounted on top of the Scorbot's elbow, and so moving the Scorbot also moves the cameras.

4.1.3 Record Mode

Record Mode was developed as a consequence of the limitations presented by the robot inhibiting further tuning of Direct Control. Using Record Mode is much easier for the operator in practice as the risk of unintended manipulator movement is minimized, hence, it became the preferred way of performing simple tasks with the manipulator.

By the click of a joystick button, Record Mode and its associated joystick buttons are enabled. The view immediately changes from displaying camera images to displaying the point cloud, as generated from received Kinect data. Whenever the left hand of the operator is moved in front of the Leap Motion / Oculus Rift the virtual representation of the Scorbot ER4u is rendered, according to the hands position. The same kinematic equations as used during Direct Mode is utilized, except now the joint angles are not continually transferred to the server as set points - only the virtual robot moves accordingly.

The point cloud and virtual robot is scaled to match real world sizes, and placed in relation to each other such that the virtual robot may reach out and touch elements in the cloud. As the operator moves his left hand in front of the Leap Motion the rendered robot follows his movement without noticeable delay. Buttons on the base of the joystick may then be used to append points into a sequence, to later be traversed by the physical robot.

When the operator is finished adding points to the sequence, he may push the execute button on the joystick which makes the robot drive into the point cloud area before iterating

Pros	Cons
Responsive - The virtual robot respond to hand movement without noticeable delay	Unable to work in dynamic environments
Intuitive - The concept is easy to grasp, and does not require much training	Drives blindly into work area during execution
Visuals - The rendered point cloud makes up an accurate 3D representation of the work area	Poor Kinect resolution leads to uneven object edges
	The point cloud does not interact with the virtual robot

Table 4.1: Pros and cons of Record Mode

through stored positions. The mode is accurate when calibrated correctly, and consistently interacts with objects as instructed by using the virtual robot and point cloud representations. However, the Kinect impose some limitations which renders the mode heavy dependent on static environments. Being unable to observe objects closer than approximately 0.7 meters, the immediate work area of the manipulator is not observable. Solving this by making the robot drive into the area observed by the Kinect, before engaging the robot, works okay as long as the environment is static.

4.1.4 Driving

Using the joystick for driving the robot turned out to work very well. The implemented solution, where relative speed and angle is derived from stick position, feels natural and requires minimal training in order to master. Using the joystick throttle to set a maximum speed makes the robot easy to maneuver accurately in narrow environments, while also enabling the operator to drive faster in open environments. However, the maximum speed cap is set to half of what is actually possible by the pulse width modulated outputs. Driving the robot at 100 % PWM turned out to be way too fast for safe operation, especially considering the slightly delayed 15 FPS camera feed which is used by the operator for navigation. Hence, the PWM was restricted to 0 - 50 %.

There is no observable delay between joystick maneuvering and robot movement. However, there is a delay in the video from the IP cameras of approximately 1 second which makes

fast driving in narrow environments difficult. The IP camera delay stems from the integrated RTSP solution, and could be reduced by investing in better equipment or changing approach (as suggested in 5.2.4). There is also a slight delay of the Kinect RGB camera feed measured to approximately 0.5 seconds. This could be reduced by compressing the RGB data before transmitting it from the server.

Using joystick buttons as dead-man's switches turned out to work very well. In order to drive or rotate the robot, one of two buttons must be pressed, depending on whether traditional driving or rotation is desired. If the button is released, the robot will immediately stop, preventing unintended movements and easing the driving experience. When the client is disconnected, or the server program is stopped, any driving motion is stopped within 0.5 seconds, as the microcontroller requires a steady flow of commands in order to activate the motors. This works just as intended, and avoids damage in case of malfunction or errors.

The implemented distance driver function, which is used to drive the robot into the rendered point cloud area during Record Mode, also works as intended. It consistently drives the desired distance in a straight line, even when temporarily being blocked by an obstacle on either side. Hence, using the encoder difference as angle set point works well without the need for adding any adjustment factors. A video is located on C which shows the straight forward driving in practice, and how it is affected by obstacles.

4.2 User Interface

Rendering HUD elements by using OpenGL resulted in a much better looking user interface. As opposed to using OpenCV, dynamic models are now added in 3 dimensions to display system parameters relevant during operation. Wearing the Oculus Rift now feels more immersive, as if the operator finds himself in a dedicated control room instead, not just staring at some camera images. What models to create, how to design them and where to place them in the virtual space is basically only limited by the programmers imagination, as any numeric value may be

incorporated as a rotation / translation / scaling / color parameter to any 3D model.

The HUD developed in this case, includes a clock, speedometer, head-orientation compass, remaining battery, frames per second, arm mode, statuses and warnings.

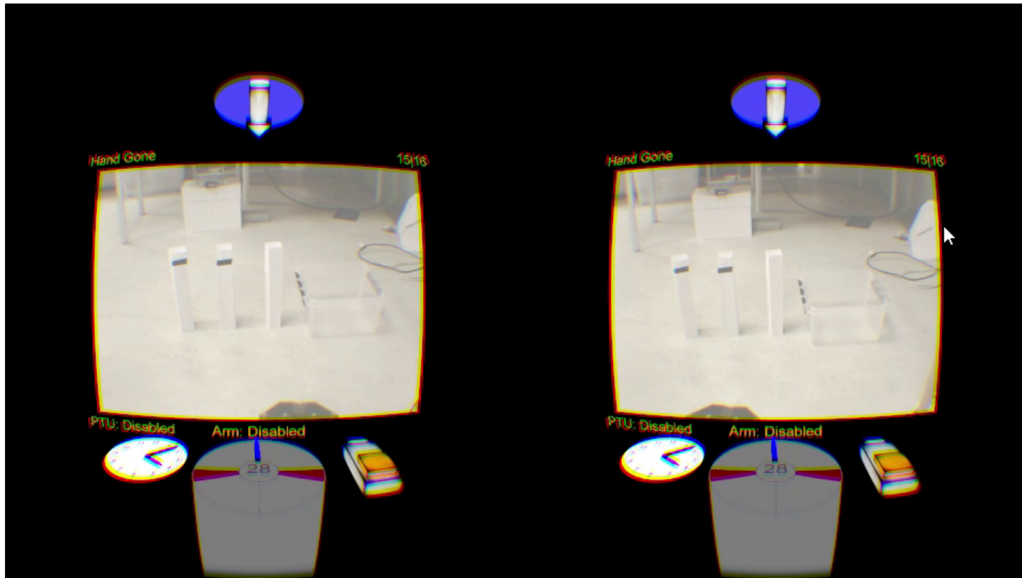


Figure 4.4: The HUD when the stereoscopic camera view is selected.

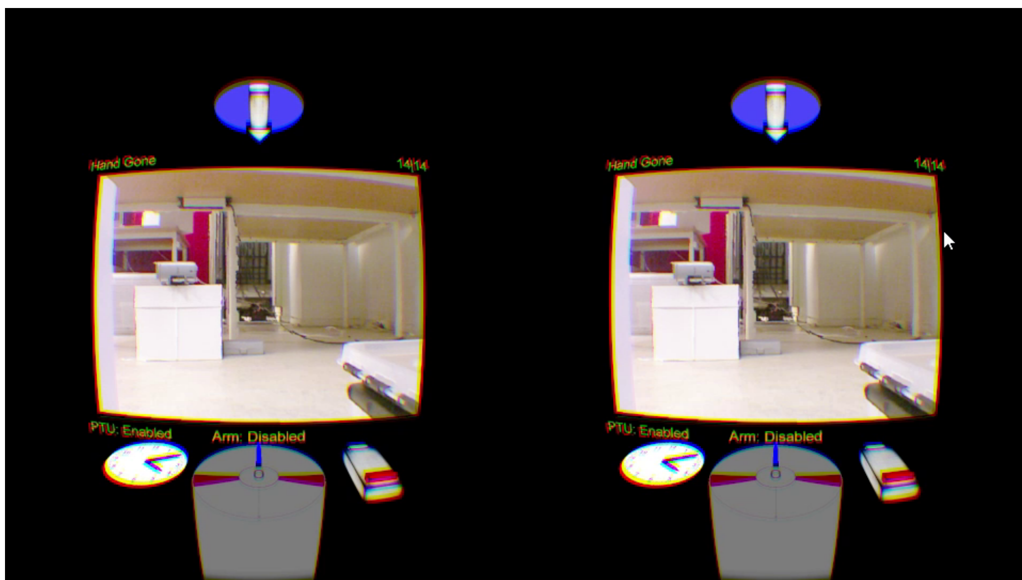


Figure 4.5: The HUD when the Kinect view is selected.

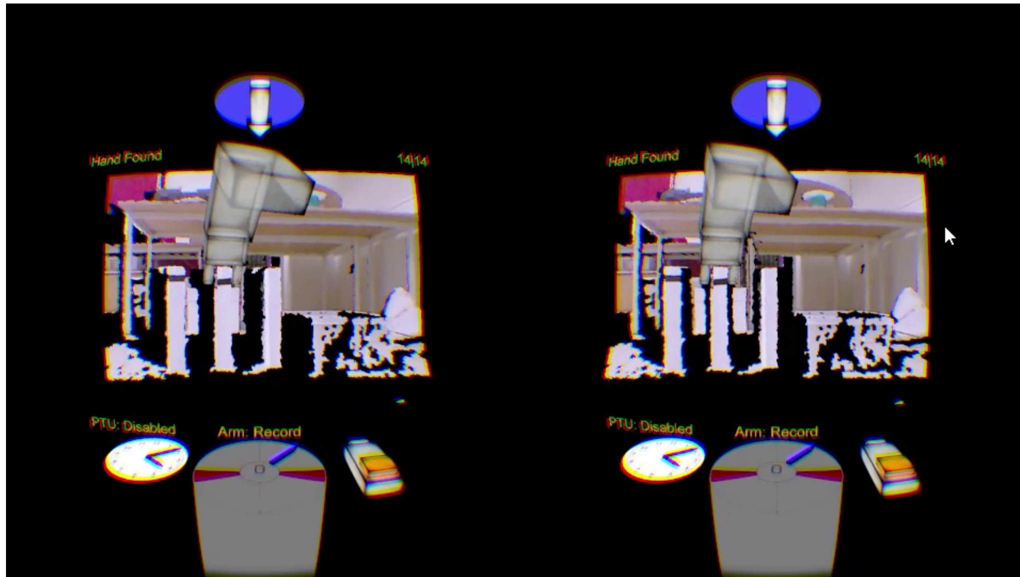


Figure 4.6: The HUD when operating the virtual manipulator during Record Mode.

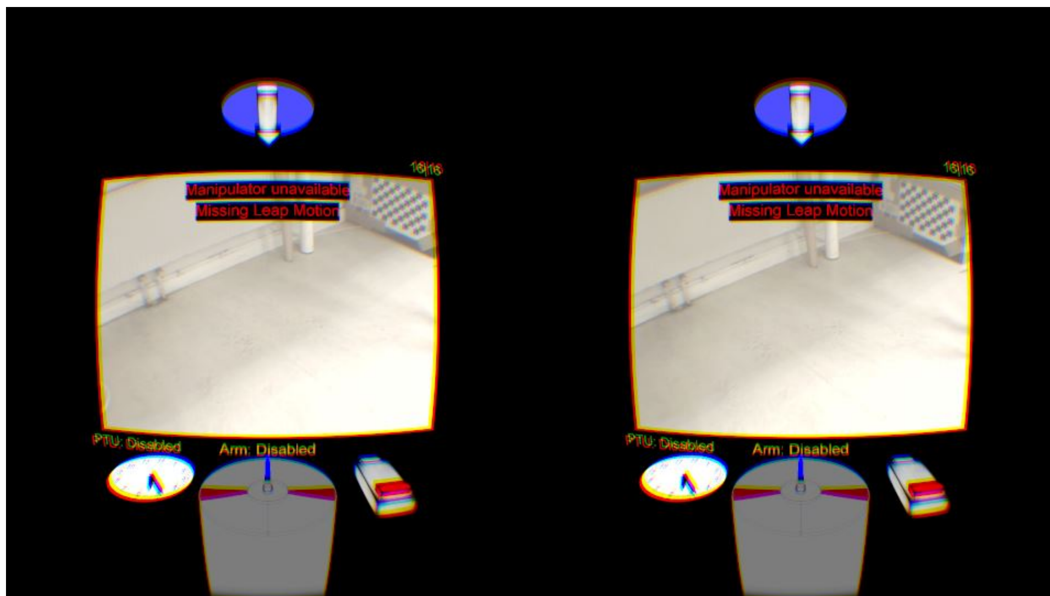


Figure 4.7: The HUD when the Leap Motion is disconnected, as illustrated by the error message.

The clock allows the operator to keep track of time while working on the robot. Naturally, this is a useful feature if for instance the operator is working in a dynamic environment, or has to perform some task within a given amount of time. Rendering an analog clock as opposed to a digital clock is mainly due to aesthetics, but it also removes any doubt regarding its meaning.

The speedometer provides visual feedback to the operator on how he operates the joystick.

This is useful, as the operator is unable to see the joystick when wearing the HMD. Instantly knowing which direction the robot will travel, and at what speed, greatly adds to the feeling of being in control.

The battery indicator intuitively symbolizes how much power is left in the robots 12 VDC battery. The indicator is scaled between full battery voltage and drop out voltage, i.e. the lower voltage limit supported by the inverters. As the voltage of the battery fall towards the lower voltage limit, so does the colored bar of the battery model. The color of the bar also changes according to remaining power. If the battery has more than two thirds of power remaining the color is green, between one and two thirds the color is orange, and red if the remaining power is less than one third. When the indicator turns red the operator needs to finish his session and recharge the battery, as the robot will fail within a couple of minutes, depending on usage.

Changing the camera zoom functionality from using OpenCV, as implemented in the previous project, to using OpenGL and U,V coordinate manipulation, works much better. There is no processing power required, as only the texture coordinates are changed such that only the mid parts of the camera textures gets plastered onto the surfaces. OpenCV on the other hand, uses the CPU to iterate over every pixel of the camera image. Then making an image copy containing scaled up versions of the pixel data contained in the middle of the original image. The reduced processing required with the new implementation is observed as no dynamic 3D models lags when zooming is activated.

An optical effect was discovered when working with text to present in the HUD. When adding text overlaying the stereoscopic camera images, its contents became incredibly hard to read. As if the text was seen in double. An experiment was set up, where a button on the joystick was programmed to toggle between stereo and mono vision from the cameras (either displaying both camera images, or displaying the same camera image to both eyes). It turned out that the text was easily read during monovision or when the cameras were covered, but became almost impossible to read during stereo vision.

Several fellow students participated in the experiment, all having the same issue. It is as

if the brain stitches together the two IP camera images in the Oculus Rift to form one depth image, and then tuning eye focus towards its content. Changing focus from this depth image to the overlaying text was incredibly hard to do, possibly because it conflicts with the perceived depth from the images. However, during monovision the camera image is perceived as a flat surface in the Oculus Rift, making it easy to change focus to the overlaying text. As a result of this "optical illusion" the possibility to add a background color to the text was added. Overlaying the stereo vision using a text with background color counteracted the phenomena (as used on the Leap Motion error message 4.7).

Chapter 5

Discussion and Recommendations for Further Work

5.1 Discussion

Controlling a robot by using a hand and joystick works well in circumstances where pinpoint accuracy is not a requirement. Both manipulator modes implemented require a steady hand by the operator and some degree of motion prediction, while the joystick driving is intuitive and easily achieved by anyone familiar with computer games.
In terms of development, the Leap Motion turned out to be easy to implement and use in the program, thanks to good documentation and a well defined library. The most time consuming parts of the project was definitely to try and make the manipulator behave as intended, and the process of creating the virtual environment.

I am content of the projects outcome, especially considering that I had no experience in either C++ nor OpenGL less than a year ago. The system works and its limited usability is mainly caused by equipment limitations and not the implementation itself. Arriving at a working solution has been a struggle at times, and through the course of the project multiple issues had to be solved and decisions had to be made, some of which are discussed in the upcoming sections.

5.1.1 Challenges of Direct Control

Controlling the robotic manipulator directly by using the left hand turned out to be harder than first imagined. The main reason being the large delay of approximately 1 second from hand movement to robot motion. The delay is too large for direct control to be usable in any practical applications requiring accuracy.

In addition to the delay, the Scorbot does not support immediate update to target position. Instead, it has to complete its current move instruction, before a new target position may be issued. This further deteriorated hand tracking, as the hand (and desired manipulator position) may change constantly.

The bottleneck was found to be the USB controller itself, and its firmware. As a consequence, eliminating these problems became practically impossible. If I knew at the start of the project, what I know now, I would start of by verifying that the manipulator was able to quickly react to set points provided in real-time. If the manipulator was found to not have such features, I would request another device, or if necessary build a separate controller for the Scorbot ER4u, supporting real-time control and faster response.

5.1.2 Fragility of Record Mode

Using the Kinect's point cloud as a representation of the physical robots work environment, requires that the Kinect is securely fastened in relation to the robot. If the angle or position of the Kinect change in any way, the rendered point cloud will change accordingly, such that the virtual robot no longer aligns with the physical robot as intended. Meaning that the physical robot will no longer go to positions as mapped by the operator during Record Mode.

In spite of this, the Kinect was fastened in front of the robot by using regular tape and strips in order to not damage it during the course of this project. Hence, care had to be taken during operation in order to avoid movement of the Kinect. The omni-wheels of the robot did intro-

duce some inevitable vibration during driving that made maintaining a precise Kinect position difficult using this fastening approach, and so slight position adjustments had to be done from time to time. Ideally, the Kinect should be built into the base of the robot, making it impossible to tamper with its relative position to the robot.

Another fragility which makes Record Mode harder to use in practice, is that the Kinect is unable to see objects in the robots immediate work area. Driving the robot into the observed work area before engaging the robot does work, but is not a good solution. This is mainly because of the encoders being the only sensors used during driving. If any of the encoders fail / starts slipping / is tampered with, the robot will no longer drive in a straight line, ultimately missing its intended work area. The robot is also unable to avoid any obstacles during its straight driving, and so the operator must be certain the robot has a free path before hitting the execute button.

Replacing the Kinect with a 3D camera capable of registering depths regardless of proximity would remove the need for any positioning and greatly improve the stability of Record Mode. Making it a very applicable mode of operation in static environments.

5.1.3 Lack of Documentation for the Scorbot ER4u

Interacting with the Scorbot through a Visual Studio application turned out to be difficult. The robot is meant to be controlled and programmed by using Intelitek's own Scorbase application. This program uses its own scripting language in order to get the robot moving between points. In the installation folder of Scorbase, however, a USBC.dll file is located which contains all functions Scorbase use in order to communicate with the USB driver for the USB Controller which translates commands into actual robot movements.

A header file giving access to the methods of the .dll file and a document describing the usage of some of its functions was developed by Intelitek in cooperation with a client, and is accessible on their download site [22]. Additionally, danish researcher Jakob Cornelius Oftebro have written a document that includes description of some additional functions C. However, many of the functions available have no description at all in either document and some descriptions

was very shallow. After sending an e-mail to Intelitek requesting more information about the header-file functions (among other questions, the entire e-mail is available on C), the following response was received:

> "We do not release information at the level for which you're looking unfortunately, as this is a closed-source product covered by our IP."

Hence, there was a lot of trial and error in order to implement some necessary functions available in the USBC header-file. In particular, the Teach() function does not work as described in C for joint set points. After many attempts, it was found that the undescribed SetJoints() function was necessary in order to achieve angle-based movement.

A description of SetJoints() and other implemented functions missing a description has been written and appended to this thesis under Appendix D. Their description was derived through extensive testing, but may contain errors as the producer will not confirm / deny their validity.

5.1.4 Working with Products Under Development

A challenge concerning both the Oculus Rift and Leap Motion, was that both products were under development. This meant that during my work on the application using these devices, new updates to both their runtimes and SDKs were published. Each update containing some new improvement to their functionality, stability enhancements and bug fixes.

Leap Motion published 5 new updates during the course of this project. The SDK / Runtime versions available at project startup was v2.3.1, and so this was implemented in the client application at first. However, its tracking capabilities was fairly disappointing especially when used in combination with the Oculus Rift. Numerous of tracking glitches and inaccuracies in hand recognition was present, effectively ruining my motivation for using the device.

However, during the middle of February Leap Motion released a new version of their software; v3.0.0. The update was named Orion, and was designed specifically for VR applications.

This radically improved the tracking capabilities when used with the Oculus Rift. Greatly increasing accuracy and reliability, as compared to previous versions. The improvement had such an impact on my Leap Motion testing, I made the decision of always keep the client application up to date with the newest version of the Leap SDK, applying changes to the code when necessary. And so, the Leap Motion performance and accuracy has gradually increased during the semester, as new and improving versions of the SDK got incorporated. The final version of the client applications is built using v3.1.2 of the Leap SDK, after first going through v2.3.1, v3.0.0, v3.1.0, v3.1.1.

Oculus VR released v1.3 of their runtime and SDK together with their first commercially available headset on March 28'th. As opposed to the Leap Motion, the decision was made to not upgrade to this latest version, but keep using the v0.8 SDK throughout the semester. The reason being that v0.8 performs well and the main upgrades of v1.3 is centered around the Asynchronous Time Warp (ATW) feature, and ease of focus management in game design [62]. None of which are applicable to this project. Optimally, Time Warp should be disabled in this project (and not improved), as it cause unintended image shaking when rendering the static scene, see (3.2.3.

Another disadvantage of the constant release updates is that documentation becomes obsolete fairly quickly, leading to an abundance of out dated guides and examples available on the internet. Hence, care had to be taken when searching for coding information concerning the devices. It is likely that SDKs and libraries used in this project are also outdated within a reasonable time span.

5.1.5 Upgrading to the latest Oculus Rift SDK

The decision to upgrade the Oculus Rift implementation from the previously developed remote vision program, from SDK v0.6 to SDK v0.8, was made in order to minimize the risk of instability. Several stability improvements have been implemented from v0.6 to v0.8, in addition to the new Direct Driver display mode. Implementing the upgrade required some minor changes to the code; some functions were renamed and a separate enabling of the tracking sensors was no

longer necessary. However, the new direct driver mode required the use of a dedicated graphics card with a GPU newer than Nvidia GTX 600 series and AMD Radeon HD7000 series [64] [54].

This proved to be an obstacle to the project, as NTNU was unable to procure a computer containing such a graphics card. The solution was for me to bring my own powerful private computer to the office, and using it as the computer running the client application throughout the semester. This was annoying as I was no longer able to work on the project from home and testing the system through internet. On a personal note, I find it quite odd that Norway's leading technology university is unable to procure a computer with an up-to-date graphics card, and hope that future students will not be bothered with this inconvenience.

5.1.6 Using the Simple and Fast Multimedia Library (SFML)

SFML is a cross-platform library that is designed to provide a simple application programming interface (API) to various multimedia components, in order to ease development of games and multimedia applications [13]. It is composed of five modules: system, window, graphics, audio and network. The library provides functions for easily creating OpenGL context windows, drawing 2D shapes directly on screen, and reading of textures into graphics memory. Meaning, it could replace the functionality of OpenCV and the WinAPI as used in this project.

Additionally, SFML supports reading data from the internet directly, such as images, by using HTTP GET / POST commands. Easing implementation of the alternative IP camera solution suggested in 5.2.4.

However, SFML does not support creation of 3D objects in itself. Meaning, separate functions must be written in OpenGL to generate vertices for such shapes. Similar to what has been done in this project. Since the previously developed telepresence system already had a working solution for OpenGL context creation using the WinAPI, the decission was made to not spend time on learning SFML as well.

5.1.7 Network Communication

At first, the network communication was implemented by using strings in the same manner as how the server communicates with the Atmel microcontroller. However, this approach resulted in large delays which especially became apparent when the Kinect point cloud was to be transferred from the robot.

All pixel values was appended into a string separated by a token. This resulted in incredibly large strings, requiring a lot of memory due to every digit of every value being considered a char (occupying a byte). In practice, this lead to a very low and inconsistent point cloud refresh rate at the client. Low refresh rate due to the size of each transmit buffer, and inconsistent due to the increased number of fragments, and potential fragment loss, caused by the Ethernet MTU limit.

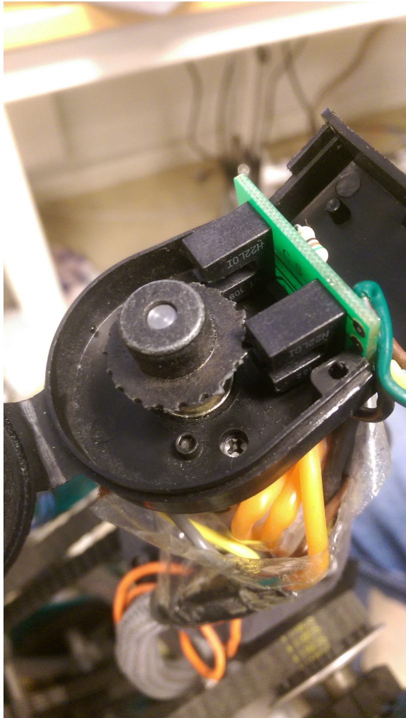
It was clear that the string approach was a poor solution, and so it was abandoned in favor of the currently implemented solution, as described in 3.

Transmission of the Kinect data was implemented from scratch, based on functions available from the WinAPI and Kinect SDK. A more elegant solution could perhaps be achieved by taking advantage of the already existing GStreamer framework, which includes functions for point cloud streaming. However, the implemented solution worked satisfyingly and it was decided to not spend time on GStreamer but rather focus on other aspects of the system [15].

5.1.8 Broken Encoders

After conducting a series of tests on the Scorbot using the USBC function library, the gripper motor started making a strange noise. When testing the available gripper functions of the library, only the functions for complete opening and complete closing worked. Much time was spent on troubleshooting the functions for moving the gripper to a certain distance, and at first it was assumed that the software implementation was wrong because of the poor documentation.

However, the strange noise generated by the gripper motor indicated that something was physically malfunctioning. Eventually, the encoder box was opened and inside was a completely destroyed encoder disc. Luckily the faculty's workshop was equipped with a high end 3D printer. By reassembling the encoder fragments its measurements and properties were derived, and so a new encoder disc was produced in plastic as a replacement. The replacement worked well, enabling the functions in software which open the gripper to given distances as intended.



(a) The encoder disc as mounted on the rotating shaft

(b) The encoder disc and all its fragments

Figure 5.1: The broken encoder disc

5.2 Recommendations for Further Work

5.2.1 Building a New Controller for the Scorbot

In order to get the robot to move to a certain point, first the point coordinates must be transmitted and stored in the USB controller's memory, followed by a move-command referring to

that particular point. Combined, these actions result in a delay of about 1 second from when the controlling PC wants the robot to move, and till the robot actually starts to move. This delay is a major disadvantage in terms of real-time robot control.

Another major disadvantage of the Scorbot is its inability to update target positions during motion, adding to the sensation of lag and poor performance. Being able to continually update the desired target location, and have the manipulator always try to move to the most recent target, would greatly increase usability.

The algorithms and software of the USB controller is made by Intelitek and is not open source, hence there is no way around these limitations by using the USB controller supplied with the Scorbot. By building a USB controller from scratch to interface the robot to a computer, operation delay could be minimized by exposing direct access of the individual motors and sensors to the computer. In order to do this, all the wires of the cable going to the robot must be mapped to its respective device in software, and a microcontroller must be implemented as a bridge between the computer and the devices of the robot. Pin-out of the D50 cable coming out of the robot is available in the Scorbot user manual, provided on C.

5.2.2 Upgrading Outdated Equipment

The robot consists of some relatively old / cheap equipment that could be replaced in order to drastically improve the overall quality of the system.

The wireless routers currently installed communicate through 802.11n, which has a theoretical maximum speed of 450 Mbps. By replacing these with newer 802.11ac routers, the theoretical maximum speed is increased to 1300 Mbps. Hence, more data can be transferred between the client and server applications in a shorter amount of time, leading to an increase in Kinect data renderings per second and smaller delays in set point transmissions [25].

Microsoft has released a new version of its Kinect device. As a result, the Kinect for Xbox 360, used in this project, is outdated and no longer in production. The new device, Kinect Sensor for

Xbox One, has much better specifications than the previous version, such as higher resolutions and faster electronics [32]. Upgrading the system with the new Kinect would therefore result in a more detailed point cloud rendered during Record Mode, and a more detailed camera view when Kinect is selected. Further reducing the risk of poor positioning when working with small or thin objects.

The IP cameras used as remote eyes support a maximum resolution of 1280x1024 at 15 FPS. The resolution is okay, since the Oculus Rift DK2 is unable to present higher pixel densities on its display anyway. However, 15 FPS is quite low and makes the camera vision appear sluggish during operation. This could easily be improved by investing in better cameras, supporting higher framerates.

5.2.3 Extending Battery Life

Currently a fully charged battery can keep the robot operational for approximately two hours in an idle state, before it needs recharging. That is, two hours if no movement commands are issued to the robot. During heavy maneuvering the battery lasts for approximately 20 minutes. One way of extending battery life could be to improve design of the robot's power network.

All of the equipment on the robot is run by 230 VAC provided by the two power inverters installed on the system. However, all equipment except for the manipulator USB controller are powered through their own power adapter which transforms the voltage down again to low level DC. The only item requiring 230 VAC is the USB controller for the Scorbot, which is recomended to be replaced any way. Every power adapter consume standby power, hence, reducing the number of adapters by connecting equipment to common power distributions would increase battery life.

Components and their voltage requirements:

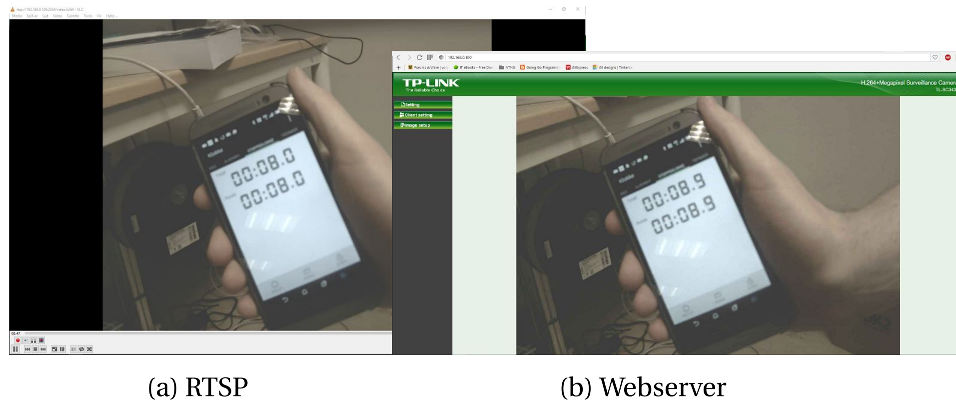
- Computer: 19 VDC

- PTU Controller: 9 - 30 VDC
- Webcameras: 12 VDC
- Router: 9 VDC
- PWM Controllers: 12 - 55 VDC
- Microsoft Kinect: 12 VDC

It is clear to see that most devices could draw their power directly from the battery of 12 VDC, which could be easily achieved by using fused terminals for instance.

5.2.4 Reducing Webcamera Delay

The images retrieved from the IP cameras by using the built in RTSP server functionality lags behind. The lag is observable when comparing the live image displayed on the cameras webserver width the RTSP video feed.



(a) RTSP (b) Webserver

Figure 5.2: Screenshot of the RTSP- and webserver video at the same moment in time.

As observed by the stopwatch in 5.2 the RTSP server lags by approximately 1 second in comparison to the webserver video, hence the webserver provides a more accurate image of the present. Ideally, the same video approach as used by the webserver should be implemented in the client application, in order to improve the operating experience.

The webserver uses javascript to continually load the newest camera image to its front page placeholder, similar to calling http://[Camera IP]/jpg/image.jpg in a while loop. In order to im-

plement this continuous image polling in the client application some mechanism for loading images from an internet address would have to be implemented. OpenCV, which is currently used for the RTSP protocol, does not support reading images from the internet. In other words the webcamera functionality would have to be rewritten, but considering a delay reduction of 1 second it is a recommended improvement.

5.2.5 Combining Manipulator Movement with the Oculus Rift

Having the IP cameras and PTU mounted on top of the Scorbots elbow allows for translational movement of the cameras in addition to the pan and tilt rotations. This means the head position, and not just its orientation, may be extracted from the Oculus Rift and used as set points for camera position. The position could be handled by linear movement of the manipulator, while the PTU handles the orientation.

Head position is derived from the Oculus Rift's infrared camera, which reads positions within its own viewing frustum. The viewing frustum of the camera could be scaled to match a portion of space reachable by the manipulator, such that when the operator moves his head to the left the manipulator moves the cameras accordingly. This could be used to further improve visual immersion experienced through the Oculus Rift and cameras. A similar project was realized by Rø rlien Carlsen and Rø st Wehinger [44] on an ABB manipulator, which provides useful insights on how to create such a solution.

5.2.6 Adding Support for Object Files

Creating the 3D models used as part of the HUD has been a tedious process, and accounts for many lines of code in the client program. The reason being that all models are built from vertices and indices as defined explicitly in code, requiring a healthy dose of imagination as no immediate visual feedback is available - the programmer does not see the resulting model before it is rendered at runtime. Working on this vertex / index level also makes the model creation prone to errors, as each index must match up to its correct vertex, and the vertices must be placed in

correct locations for the geometry to be rendered correctly.

By using a 3D modelling software (such as Blender[5]) models could be created intuitively in a graphic user interface, making it easy to model all kinds of geometric shapes and figures. Most 3D modelling software support the feature of exporting created models to a Wavefront Object file (.obj) [11]. This file typically contain geometric vertices, indices, uv coordinates, color values and vertex normals in a structured manner for the object represented. The Wavefront Object file is open source, hence, exactly which parameters are contained is vendor specific.

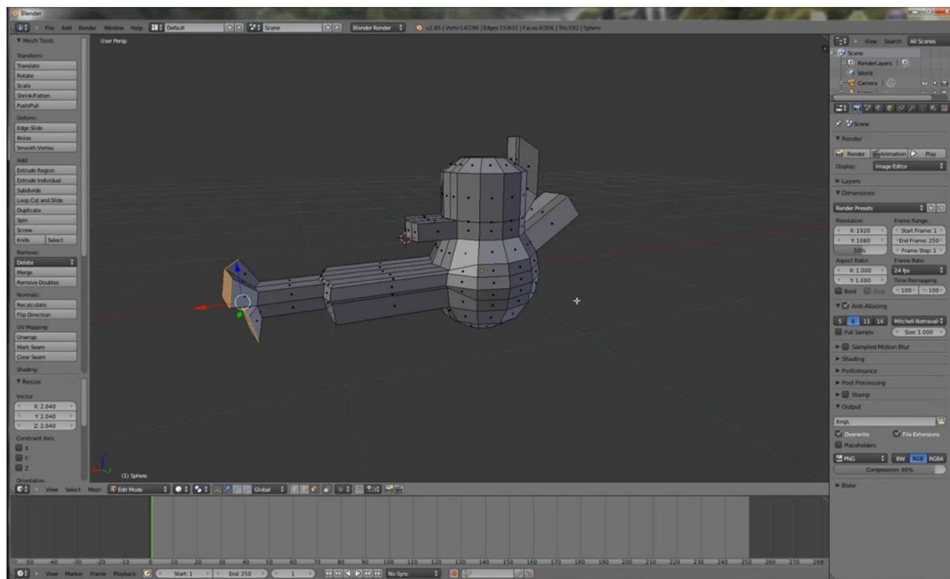


Figure 5.3: Blender 3D software.
Image Credits: TurnTheGameOn's YouTube channel

A function could be created in the client program which reads in such object files, extracting the data necessary to define vertex-, index- and texture buffers, such that a model structure could be initialized. For instance, the approach presented in [40] or [47] could be adapted. Then by simply defining a model matrix, specifying where to place the model in virtual space, and adding the model structure to the global model collection would result in the model being rendered as part of the HUD in the Oculus Rift.

Implementing such a function would make it much easier to create better looking graphics,

while also making the HUD easy to expand and edit.

5.2.7 Implementing Bi-Directional Audio

A natural addition to the system is to implement support for audio, both from the robots location to the operator, and from the operator to the robots location. The latter may be achieved by installing a small speaker on the robot and a microphone at the client computer. The former is doable by using the microphones in the Kinect and IP cameras, and a headset at the client computer. The already implemented network solution may then be expanded to also handle audio packages, in both directions.

Instead of incorporating this functionality into the applications, Skype or similar software could be run in parallel during remote control. Anyhow, such a solution would enable the operator to communicate with personel on site, greatly increasing usability of the system when performing visual inspections or simple tasks with customers present.

Appendix A

Acronyms

API Application Programming Interface

ASCII American Standard Code for Information Interchange

DK Developer Kit

FOV Field of View

GLSL OpenGL Shading Language

GPU Graphics Processing Unit

HMD Head Mounted Display

HUD Heads Up Display

IP Internet Protocol

LAN Local Area Network

LED Light Emitting Diode

MFC Microsoft Foundation Class Library

MTU Maximum Transmission Unit

OSI Open System Interconnection

POV Point of View

PTU Pan Tilt Unit

PWM Pulse Width Modulation

RGB Red, Green, Blue

RTSP Real-Time Streaming Protocol

SDK Software Development Kit

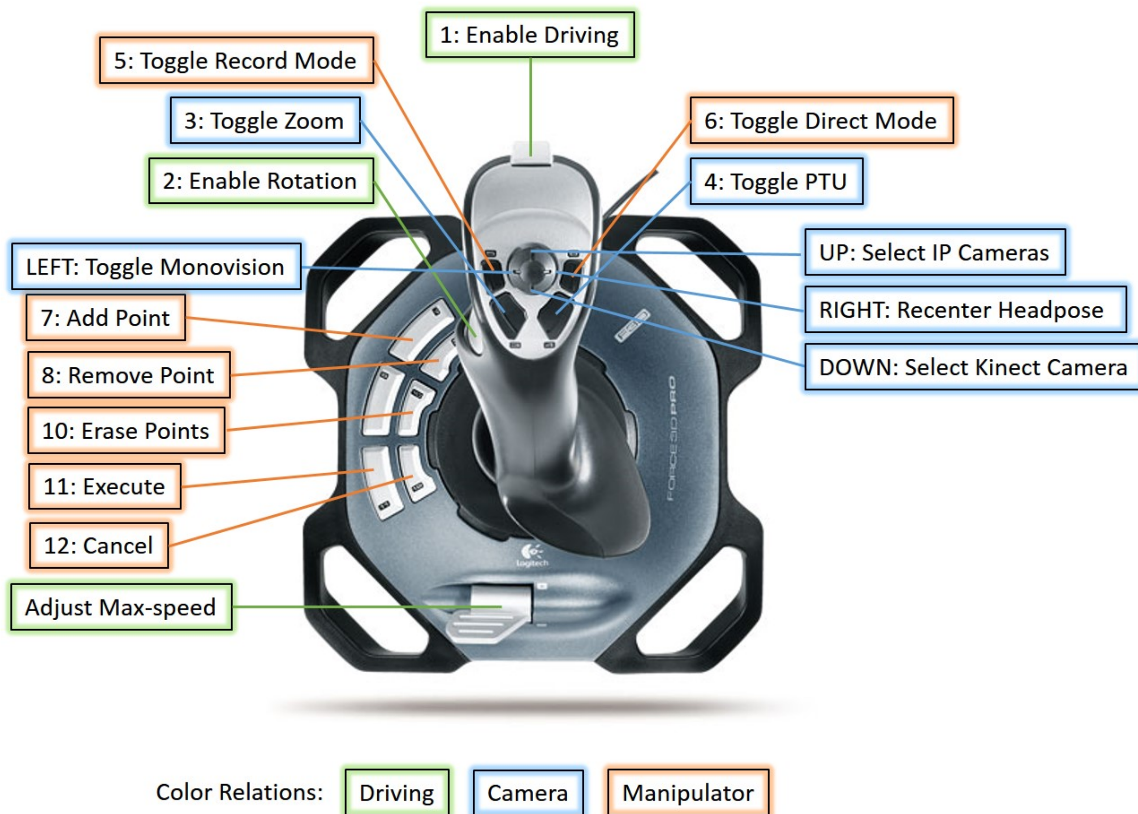
TCP Transmission Control Protocol

UDP User Datagram Protocol

VR Virtual Reality

Appendix B

Joystick Configuration



Appendix C

DVD Contents

This thesis comes bundled with a DVD which contains the resulting application, source code and necessary libraries, in addition to a digital copy of the thesis itself.

Folders and their contents:

ClientSoftware

Contains all software to be installed on the client machine.

ServerSoftware

Contains all software to be installed on the server machine.

Documents

Contains two subfolders:

Datasheets and Manuals

Contains datasheets and manuals for equipment used, in addition to the software manuals and E-Mail response regarding the Scorbot ER4u.

Previous Reports

Contains reports from previous students working on the same robot equipment.

Videos

Contains demonstration videos of the system and features.

Appendix D

Scorbot ER4u Function Reference

The following document provides description to some of the missing functions in Appendix C. All descriptions are derived through extensive testing, as they are undocumented by the producer.

Note:

The .H file contains functions where parameters are set to type TCHAR, however, the underlying USBC.dll is incompatible with the Unicode character set. In order to use these functions in a Unicode project, the parameter types must be changed from TCHAR to char (as TCHAR will automatically assume 16-bit char representations).

SetJoints

Description:

Adds a point to a given vector, specified in joint angles.

Syntax:

bool SetJoints(Char * szVectorName, short sPointNumber, long * plCoorArray, short sCoorArrayDim, long lPointType);

Parameters:

*Char * szVectorName* - Vectorname.

short sPointNumber - Point number in the vector.

*long * plCoorArray* - Array with point information, Joint Values

short sCoorArrayDim - Size of *plCoorArray*. 5 for base, shoulder, elbow, roll, pitch angles.

long lPointType - Point type (ABS JOINT).

Return Value:

Returns true if the point was successfully added to the vector specified, false otherwise.

Notes:

Using Teach() for adding points specified by joint angles, as specified in the Intelitek document, did not work. However, this function does exactly that.

A base angle value of -90 000 rotates the base 90° counter clockwise, while an angle of 90 000 rotates it 90° clockwise.

A shoulder angle value of 0 makes the shoulder point straight forward, while -90 000 makes it point straight up (parallel to the base).

An elbow angle value of 0 makes the elbow point in parallel to the shoulder, while 90 000 makes it point 90° down, relative to the shoulder.

A pitch angle value of 90 000 makes the gripper point 90° down relative to the elbow, while an value of 0 makes the gripper and elbow parallel.

A roll angle value of 90 000 makes the wrist rotate 90° clockwise, while -90 000 rotates the wrist 90° counter clockwise.

ShowPositErr

Description:

Supplies a thread of the USBC.dll with a function to be called at given intervals, to provide position errors from the robot.

Syntax:

bool ShowPositErr(CallBackFun fnPositErr);

Parameters:

CallBackFun fnPositErr - Address of the function to call at certain intervals.

Return Value:

Returns true if the operation was successful, false otherwise.

Notes:

Useful for monitoring and troubleshooting. Values may be compared with impact detection

limits, in order to determine the reason for an impact detection stop.

ClosePositErr

Description:

Stops USBC.dll from calling the supplied position error callback function provided by ShowPositErr.

Syntax:

bool ClosePositErr();

Parameters:

None.

Return Value:

Returns true if the operation was successful, false otherwise.

Notes:

None.

Velocity

Description:

Sets the homing velocity of the robot's axis.

Syntax:

bool Velocity(UCHAR ucAxis, short sPercent);

Parameters:

UCHAR ucAxis - The axis to update. 0 - 5.

short sPercent - Homing velocity in percentage relative to normal velocity during operation.

Return Value:

Returns true if the operation was successful, false otherwise.

Notes:

None.

GetAnalogInput

Description:

Queries the USB controller for measured voltage of an analog input.

Syntax:

bool GetAnalogInput(short sIONumber, UCHAR * ucValue);

Parameters:

short sIONumber - The analog input number to be read. 0 - 3.

*UCHAR * ucValue* - Address to a byte which is to contain the resulting value.

Return Value:

Returns true if the operation was successful, false otherwise.

Notes:

Reads into a byte, hence, measured value ranges from 0v - 10v = 0 - 255.

Appendix E

Installation Instructions

The following document provides a step-by-step guide on how to install and set up the system, for it to work as intended. All software, necessary SDKs and drivers are found on the attached DVD C.

E.1 The Robot

E.1.1 Wiring

Ensure that the following devices are mounted on the robot, and that they are interlinked as described:

Computer

Should be connected to all devices of the robot.

TP Link Router

The computer and the two IP cameras should be attached to the router's LAN ports.

Robotic Manipulator - Scorbot ER4u and USB Controller

The manipulator must be connected to the USB controller, which again is connected to the server computer. Connect a wire in series with a 20k Ω resistor between the Analog Input 1 (AI1) of the USB controller and the positive battery pole. Insert a 20k Ω resistor

between AI1 and Ground, and wire the ground to the negative battery pole. Optionally, mount a capacitor between AI1 and Ground to reduce noise to battery measurement.

Microsoft Kinect

Confirm that the Kinect is mounted in front of the manipulator 8cm above the surface, facing straight ahead. It should be connected to the computer through a separate USB 2.0 bus.

Pure Sine Inverter

Make sure the USB controller and 24 VDC power supply is powered through the pure sine inverter.

Biltema Power Inverter

All other equipment should be powered by the Biltema inverter.

E.1.2 The Server Computer

Note: It is not recommended to setup the system for first time use while using the battery as power source. Plug the computer and devices into a stable 230 VAC outlet instead, to avoid unnecessary shutdowns.

Minimum Requirements

- Windows 7 (or newer)
- 32-bit (x86) or 64-bit (x64) Dual-core 2.66-GHz or faster processor
- 4 GB RAM
- 1x USB 2.0 | 2x USB 1.1 | 1x COM port | 1x Ethernet Port

E.1.3 Setup

1. Make sure the computer is properly secured to the robot, and that the following devices are connected to it:

- Scorbot ER4u USB Controller
- Atmel A3BU Xplained Evaluation Board
- Microsoft Kinect - Located in front of the Scorbot
- Pan-Tilt Unit - Mounted on top of the Scorbot's elbow
- Router (TP-Link TL-WR841N)

2. Install the RoboCell software and Kinect SDK located in DVD\ServerSoftware
3. Vertify that KINECTSDK10_DIR has been added as a system variable, by making the correct SDK path is returned when typing "echo KINECTSDK10_DIR" in Command Prompt
4. Restart the computer
5. Install Atmel Studio, available from http://www.atmel.com/Microsite/atmel-studio/, and upload the program located in DVD\ServerSoftware \MicrocontrollerSW to the Atmel A3BU.
6. Copy the entire DVD\ServerSoftware\RobotServer folder to a local directory on the server computer.
7. Navigate to ..\RobotServer\Debug\Config.cfg and edit the the file in notepad so that each COM port reflects the system assigned ports. Also make sure the servers port number is equal to what is being used in the client applications config file, and that it is not blocked by any firewall.
8. Run RobotServer.exe located in ..\RobotServer\Debug. If everything is set up correctly, the program starts off by homing the robot, when the console window says "Running" the server is ready to connect with a client.

E.2 The Network

All router setup is done through their built-in webserver, which has a default IP of 192.168.0.1 and default username / password equals admin / admin.

Setup the two TP-Link routers to form a wireless bridge (WDS Bridging)

First, set up both routers with a unique SSID under the "Wireless Settings" menu, and make sure "Enable Wireless Router Radio", "Enable SSID Broadcast" and "11n only" is selected. Optionally, go to "Wireless Security" and set up the connection with a password. Second, set up one of the routers to bridge the network of the other router. Do this by selecting "Enable WDS Bridging", and select the SSID of the other router as the connection to bridge. Click "Survey" under "BSSID (to be bridged)" and find the SSID of the other router in the list. Provide the correct password, if the connection is secured. Restart both routers, and verify that both IP cameras, the server, the client computer and internet is connected to the same network.

Address reservation

Go to "Address Reservation" and add the MAC address of all network equipment on the robot (computer and IP cameras). Set the devices up with static IP addresses, and remember to add these IPs into the config file of the client program.

Port Forwarding

In order to control the robot through the internet, ports must be set up and forwarded to their correct IP addresses. This is done in the router which is not located on the robot. Go to "Port Forwarding" and forward one port to the IP address of each IP camera and to the server. Now update the config file of both the client and server applications with the chosen ports.

E.3 The Client Computer

Minimum Requirements

- Windows 7 (or newer) (Desktop PC, Laptops are not supported).
- 64-bit Intel i5-4590 equivalent or greater
- 8 GB RAM
- Nvidia GTX 760 or better
- Oculus Rift Compatible HDMI 1.3 video output

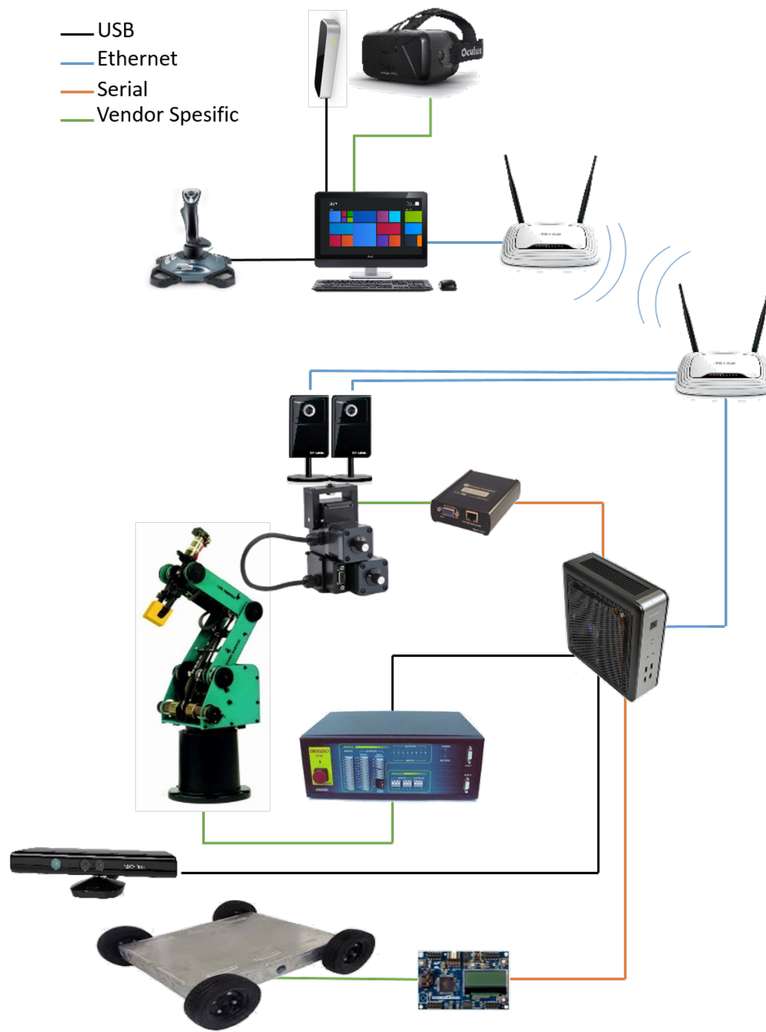
- 2x USB 3.0 | 1x USB 2.0 | 1x USB 1.1

E.3.1 Setup

1. Make sure all drivers are up to date.
2. Install the Oculus (v0.8) and Leap Motion (v3.1.2.40841) Runtimes, located in DVD\ClientSoftware.
3. Restart the computer.
4. Connect the Oculus Rift DK2, the Leap Motion and the joystick to the computer.
5. Calibrate the Leap Motion via its control panel software. Follow the instructions under "Troubleshooting" -> "Device Calibration".
6. Test that the Oculus Rift works properly by running the demo in the Oculus Config Tool.
7. Copy the entire DVD\ClientSoftware\RobotClient and DVD\ClientSoftware\Libraries folders from the DVD to the same local folder on the client computer.
8. Add a system variable named OPENCV_DIR to your system environmental variables, pointing to the folder \Libraries\opencv\build\x64\vc12.
9. Navigate to ..\RobotClient\Debug\Config.cfg. Edit this file in notepad (or similar), such that the parameters match your setup. The IP addresses must point to their corresponding devices, and the server port must match what is set up in the config file for the server application.
10. Verify that the robot is operative and the server is running. Ping the IP of the server from the client PC to make sure the network is ok.
11. Run RobotClient.exe located in ..\RobotClient\x64 \Debug.
12. If everything is done correctly the joystick and Leap Motion may be used to maneuver the robot (see B on how to use the joystick), while cameras and user interface is displayed in the Oculus Rift.

Appendix F

System Interconnections



Bibliography

[1] (2015). *Inverse Kinematics Solution for Trajectory Tracking using Artificial Neural Networks for SCORBOT ER-4u by Rahul R Kumar and Praneel Chand*.

[2] Aspunvik, P. (2013). Robotisert vedlikehold. Master's thesis, Norwegian University of Science and Technology.

[3] Bekken, K. S. (2010). Bevegelsesstyring av robotarm og kamera med kollisjonsunngåelse. Master's thesis, Norwegian University of Science and Technology.

[4] Bianchi, G., Neglia, G., and Manucuso, V. Lecture 5. internet transport layer: User datagram protocol. http://www-sop.inria.fr/members/Vincenzo.Mancuso/ReteInternet/05_udp.pdf. Online: accessed 05.06.2016.

[5] Blender.org. Blender. http://www.blender.org. Online; accessed 10.06.2016.

[6] Boerner, K. (2016). Ikea launches pilot virtual reality (vr) kitchen experience for htc vive on steam. http://www.ikea.com/us/en/about_ikea/newsitem/040516_Virtual-Reality. Online; accessed 17.06.2016.

[7] Cisco (2016). Resolve IP Fragmentation, MTU, MSS, and PMTUD Issues with GRE and IPSEC. http://www.cisco.com/c/en/us/support/docs/ip/generic-routing-encapsulation-gre/25885-pmtud-ipfrag.html. Online; accesed 20.03.2016.

[8] E. Comer, D. (2013). *Internetworking with TCP/IP*, volume 1. Pearson, 6 edition.

[9] Egeland, O. and Gravdahl, T. (2003). *Modeling and Simulation for Automatic Control*. TAPIR Trykkeri.

[10] Entertainment, S. I. (2016). Playstation vr. https://www.playstation.com/en-us/explore/playstation-vr/. Online; accessed 17.06.2016.

[11] FileFormat.info. Wavefront obj file format summary. http://www.fileformat.info/format/wavefrontobj/egff.htm. Online; accessed 10.06.2016.

[12] Glumphy. Modern opengl. https://glumpy.github.io/modern-gl.html. Online; accessed 17.06.2016.

[13] Gomila, L. Simple and fast multimedia library. http://www.sfml-dev.org/. Online; accessed 14.06.2016.

[14] Grauman Weinbaum, S. (1935). *Pygmalion's Spectacles*. Wonder Stories. Gernsback Publication.

[15] GStreamer (2016). Open source multimedia framework. https://gstreamer.freedesktop.org. Online; accessed 14.06.2016.

[16] Hall, B. (2011). *Beej's Guide to Network Programming*. Jorgensen Publishing.

[17] iFixit. Xbox 360 kinect teardown. https://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/1. Online; accessed 07.04.2016.

[18] Inc., F. S. High performance pan-tilts. http://www.flir.com/mcs. Online; accessed 23.05.2016.

[19] Inc., L. M. Leap motion. http://www.leapmotion.com. Online; accessed 01.03.2016.

[20] Inc., S. E. Stem system. http://sixense.com/wireless. Online; accessed 01.03.2016.

[21] Inc., T. L. Myo. https://www.myo.com/. Online; accessed 01.03.2016.

[22] Intelitek. Scorbot c++ example - programming interface.zip. http://www.intelitekdownloads.com/Software/Other/. Online; accessed 10.06.2016.

[23] Jon, P. (1980). User datagram protocol. RFC 768 (Standard).

[24] Jon, P. (1981). Transmission control protocol. RFC 793 (Standard). DARPA Internet Program Protocol Spesification.

[25] Kelly, G. (2015). 802.11ac vs 802.11n - what's the difference? http://www.trustedreviews.com/opinions/802-11ac-vs-802-11n-what-s-the-difference. Online; accessed 14.06.2016.

[26] LaValle, S. (2013). The latent power of prediction. http://www.oculusvr.com/blog/the-latent-power-of-prediction/. Online; accessed 15.04.2016.

[27] Liu, P. X., Meng, M. Q.-H., and Yang, S. X. (2003). Data communication for internet robots. *Autonomous Robots 15, 213-223*, 15(3):213–233.

[28] Logitech. Force 3d pro. http://support.logitech.com/en_my/product/force-3d-pro. Online; accessed 17.06.2016.

[29] LTD, I. V. E. Imagine sitting in a lecture room with einstein as he talks about the theory of relativity. http://immersivevreducation.com/lecture-vr/. Online; accessed 17.06.2016.

[30] Mansurov, N. (2011). What is chromatic aberration? https://photographylife.com/what-is-chromatic-aberration. Online; accessed 06.04.2016.

[31] Microsoft. Joysticks. https://msdn.microsoft.com/en-us/library/vs/alm/dd757116(v=vs.85).aspx. Online; accessed 01.03.2016.

[32] Microsoft. Kinect for xbox one. http://www.xbox.com/nb-NO/xbox-one/accessories/kinect-for-xbox-one. Online; accessed 14.06.2016.

[33] Microsoft. Mfc desktop applications. https://msdn.microsoft.com/en-us/library/d06h2x6e(v=vs.120).aspx. Online; accessed 16.06.2016.

[34] Microsoft. Why transformation order is significant. https://msdn.microsoft.com/en-us/library/eews39w7(v=vs.110).aspx. Online; accessed 17.06.2016.

[35] Njåstad, E. B. r. (2015). Robotsveising med korreksjon fra 3d-kamera. Master's thesis, Norwegian University of Science and Technology.

[36] Noitom. Perception neuron. https://www.neuronmocap.com/. Online; accessed 01.03.2016.

[37] Oculus (2012). Oculus kickstarter campaign. https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game/. Online; accessed 29.02.2016.

[38] OpenGL. Opengl wiki. https://www.opengl.org/wiki/. Online; accessed 11.06.2016.

[39] OpenGL (2015). Legacy opengl. https://www.opengl.org/wiki/Legacy_OpenGL. Online; accessed 06.04.2016.

[40] OpenGL-Tutorial.org. Model loading. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/. Online; accessed 10.06.2016.

[41] OpenGL-Tutorial.org. opengl-tutorial. http://www.opengl-tutorial.org. Online; accessed 10.06.2016.

[42] Overvoorde, A. Open.gl. https://open.gl. Online; accessed 11.06.2016.

[43] Plunkett, L. (2014). Facebook buys oculus rift for $2 billion. http://kotaku.com/facebook-buys-oculus-rift-for-2-billion-1551487939. Online; accessed 29.02.2016.

[44] Rø rlien Carlsen, L. T. and Rø st Wehinger, P. (2015). Remote operations of irb140 with oculus rift. Master's thesis, Norwegian University of Science and Technology.

[45] Robertson, A. and Zelenko, M. Voices from a virtual past. http://www.theverge.com/a/virtual-reality/oral_history. Onlince; accessed 29.02.2016.

[46] Russel, K. (2014). Why virtual reality is happening now. http://techcrunch.com/2014/11/24/why-virtual-reality-is-happening-now/. Online; accessed 29.02.2016.

[47] Scratchapixel. Obj file format. http://www.scratchapixel.com/old/lessons/3d-advanced-lessons/obj-file-format/obj-file-format/. Online; accessed 10.06.2016.

[48] Scratchapixel (2015). The perspective and orthographic projection matrix. http://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix. Online; accessed 15.05.2016.

[49] Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional, 8 edition.

[50] Siqveland, O. M. (2015). Remote Vision Using Oculus Rift.

[51] Sol (2013). Opengl 101: Drawing primitives - points, lines and triangles. https://solarianprogrammer.com/2013/05/13/opengl-101-drawing-primitives/. Online; accessed 06.04.2016.

[52] Songho, H. A. Opengl projection matrix. http://www.songho.ca/opengl/gl_projectionmatrix.html. Online; accessed 17.06.2016.

[53] Spong, M. W., Hutchinson, S., and Mathukumalli, V. (2006). *Robot Modeling and Control*. John Wiley & Sons, Inc.

[54] Staff, O. V. (2015). 0.7 / 0.8 gpu setup. https://forums.oculus.com/vip/discussion/26029/0-7-0-8-gpu-setup-laptops-are-not-supported. Online; accessed 05.06.2016.

[55] Thompson, P. (2002). Eyes wide apart: Overestimating interpupillary distance. *Perception*, 31:651–656.

[56] Turner, D., Wilhelm, R., and Lemberg, W. The freetype project. https://www.freetype.org. Online; accessed 20.02.2016.

[57] Van Oosten, J. (2011). Understanding the view matrix. http://www.3dgep.com/understanding-the-view-matrix/. Online; accessed 17.06.2016.

[58] Virtual Institue of Applied Science (VIAS) (2010). 3d scaling with homogeneous coordinates. http://www.vias.org/comp_geometry/math_coord_homogen_3d_scaling.htm. Online; accessed 12.06.2016.

[59] Virtuix. Virtuix omni. http://www.virtuix.com/. Online; accessed 01.03.2016.

[60] VR, M. (2015a). Manus vr. http://www.manus-vr.com. Online; accessed 20.04.2016.

[61] VR, N. Next vr. http://www.nextvr.com/. Online; accessed 01.03.2016.

[62] VR, O. Oculus pc sdk 1.3.0. https://developer.oculus.com/downloads/pc/1.3.0/Oculus_SDK_for_Windows/. Online; accessed 14.06.2016.

[63] VR, O. (2015b). Rendering to the oculus rift v0.8. https://developer.oculus.com/documentation/pcsdk/0.8/concepts/dg-render/. Online; accessed 23.05.2016.

[64] VR, O. (2016). Oculus rift. http://www.oculus.com. Online; accessed 23.05.2016.

[65] VRMC (2016). The virtual reality medical center. http://www.vrphobia.com. Online; accessed 17.06.2016.