CHALMERS

Positioning systems for scale model cars

A survey of available systems and an application with Qualisys Track Manager.

Master's thesis in Complex Adaptive Systems

JOAKIM ODENGÅRD

Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2013
Master's thesis 2013:54

MASTER'S THESIS IN COMPLEX ADAPTIVE SYSTEMS

Positioning systems for scale model cars

A survey of available systems and an application with Qualisys Track Manager.

JOAKIM ODENGÅRD

Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2013

Positioning systems for scale model cars
A survey of available systems and an application with Qualisys Track Manager.
JOAKIM ODENGÅRD

Master's thesis 2013:54
ISSN 1652-8557
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Gothenburg, Sweden 2013

Positioning systems for scale model cars
A survey of available systems and an application with Qualisys Track Manager.
Master's thesis in Complex Adaptive Systems
JOAKIM ODENGÅRD
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

Abstract

This report is a brief survey of available positioning systems and techniques that can be applied to tracking a scale model car for the purpose of using it in a test bed for active safety systems in cars. It also treats of setting up the Qualisys Track Manager motion capture system for this purpose and an evaluation of its performance by looking at precision, range and trajectory fill rates of the tracked bodies. As a means of later implementing it into the test bed, a program is written for receiving a stream of positioning data from the motion capture software over a network. It was found that the motion capture system is capable of tracking a scale model vehicle within the majority of the test area with a residual of a few millimeters.

Keywords: Scale model cars, positioning systems, RTLS, Qualisys Track Manager, QTM, local positioning systems

Acknowledgements

I woud like to thank Krister Wolff, the supervisor and examiner of this project, Ola Benderius and the Qualisys support for being very helpful.

Contents

List of Figures

List of Tables

Nomenclature

6 DOF 6 Degrees Of Freedom, page 9
DGPS Differential Global Positioning System, page 6
EKF Extended Kalman Filter, page 4
FOV Field Of View, page 11
FPS Frames Per Second, page 9
GPS Global Positioning System, page 6
IMU Intertial Measurement Unit, page 3
IR Infrared, page 6
LRF Laser Range Finder, page 7
MCU Motion Capture Unit, page 10
MEMS Micro Electro Mechanical Systems, page 3
MP Megapixel, page 8
MVC Model View Controller, page 19
QTM Qualisys Track Manager, page 1
RT Real Time, page 16
RTK Real Time Kinematic, page 6
RTLS Real Time Locating System, page 2
SBAS Satellite Based Augmentation Systems, page 6
SDK Software Development Kit, page 7
SLAM Simultaneous Localization And Mapping, page 7
TCP Transmission Control Protocol, page 16
UDP User Datagram Protocol, page 16
UWB Ultra-Wide Band, page 5

1 Introduction

Active safety systems in vehlices are safety systems that use information about the vehicle's state to avoid accidents or lessen the impact using inventions such as brake assist and traction control. There are also advanced driver assistance systems such as adaptive cruise control and lane departure warning. Furthermore there are systems under development included in this category that literally take autonomous control of the vehicle in order to avoid accidents. To further this research the NG-TEST project at Chalmers University of Technology aims to create a fully functional testing facility for this type of systems using scale model cars which demands a local positioning system.

1.1 Purpose and Aim

The goal of this thesis is twofold:

1. To gain knowledge about what local positioning systems are available today and to implement and evaluate an available candidate systems on a scale model car.
2. To provide instructions and recommendations for using Qualisys track manager (QTM) and similar systems, why the method section partly is written like a manual.

1.2 Scope

This project is limited to testing QTM in one room with one camera configuration and the results may differ under other conditions. The survey is mainly limited to positioning systems that are small and precise enough to be applied on scale model cars and no actual tests are done on other systems than QTM.

1.3 Related Work

There are similar test beds using scale model cars to test autonomous vehicle control systems at KTH Royal Institute of Technology [21] using drones to overlook trucks and the Gulliver project [36] at Chalmers University of Technology for traffic simulations. There has also previously been a project [20] at Chalmers University of Technology aimed at creating a smart-phone app for communication with QTM.

2 Background

2.1 Different Positioning Systems

A positioning system is a system that reports the coordinates in space at which something is located down to a certain precision and sample rate. They are usually called RTLS which is an abbreviation for real-time locating system. There are mainly two types of positioning systems, relative and absolute. The relative positioning systems are incremental methods that keep track of the change in position from the starting position. Absolute systems report the position in relation to reference points such as landmarks or beacons with known position. The precision describes how large the positioning error is, there are different measures for this but for absolute systems it is often the average or standard deviation Euclidian distance between the true location and the reported one. For relative systems the error, called "drift", is specified in percentage of distance travelled. The sample or update rate is how often the system reports the position, usually specified in Hz, the number of times per second.

In this chapter we take a look at different techniques and principles for positioning systems as well as implementations relevant to our application with interesting parameters such as positioning error, update rate, price, range and drawbacks such as sensitivity to noise from the environment or speed limit of the tracked object.

2.1.1 Dead Reckoning

Deduced reckoning is a relative positioning method where the current position and heading is estimated by integrating the velocity or acceleration over a time step and adding that value to the previously known position and heading. The positioning error with this method grows with distance traveled, known as drift, why it has to be combined with an absolute positioning system such as GPS to stop the error from growing infinitely [6, p. 13]. The error for dead reckoning is often measured in percentage of the distance travelled [15, p. 8]. The main advantage of dead reckoning is that it always provides a reading at relatively high refresh rates and isn't blocked out by obstacles in the environment which is the case with many absolute positioning methods that demand a clear line of sight to the vehicle.

Odometry

A simple way to perform dead reckoning is to calculate the displacement along the vehicle's trajectory with an odometer, hence the term odometry [6, p. 13]. In most cases the odometer is a rotary encoder attached to a wheel axis or a motor shaft that outputs a square or sine wave as it turns. The encoders for robot applications are usually optical but there also other types such as brush or magnetic encoders used in ABS brakes. Optical encoders can either be absolute or relative. The relative ones are simpler and cheaper where a light beam is aimed at a photodetector and is interrupted a number of times per revolution. Among the relative encoders single-channel tachometer encoders are the simplest ones that only count the number of revolutions and not the direction. They are unreliable at low speeds because of quantization errors since the number of pulses increase with velocity. Phase-quadrature incremental encoders add a second channel that increases the resolution and outputs a pulse train that is 90 degrees out of phase with the first one and are thus capable of detecting the direction of rotation, illustrated in figure 2.1.1.

Odometry errors occur from tire slippage, tread wear and tire inflation [6, p. 17]. In the case of RC cars tire slippage is a big factor due to the high torque of electric motors. In [6, p. 138] tire slippage already became a big problem at speeds above 0.3 m/s.

Inertial Navigation

Inertial navigation works by sensing acceleration and orientation in the three axes and integrating over time. An inertial measurement unit (IMU) consists of accelerometers and gyroscopes and sometimes magnetometers [15, p. 6]. An accelerometer simply detects acceleration in a certain direction and can be found in many consumer electronics devices today such as smartphones. In an IMU three accelerometers are mounted orthogonally to eachother and if not held perfectly level with the ground components of the gravitation force will be measured why it is important to know the exact orientation of an accelerometer. Three gyroscopes

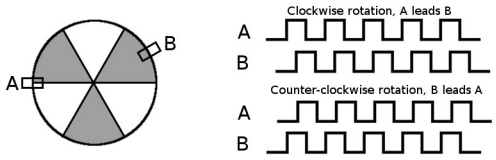

Figure 2.1.1: *Schematic of a quadrature encoder.*

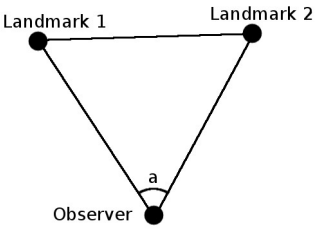

Figure 2.1.2: *Calculating one's position given the position of landmarks 1 and 2 and the angle a between them is an example of triangulation.*

measures orientation and three magnetometers measure the components of the earth magnetic field giving an absolute orientation but is subject to disturbances in the environment. For robot applications micro electro mechanical systems (MEMS) IMUs consisting of microchips on a small printed circuit board are practical to use since they are compact and cheap, an IMU board costs less than 50 EUR at Sparkfun [18]. MEMS IMUs can have relatively good updates rate at 100 Hz [39, p. 5], [15, p. 6], while other units are slower at 10 Hz [9, p. 5].

Intertial navigation has the same disadvantages as odometry with increasing error over time and bad precision at low accelerations [9] but is not subject to wheel slippage and is a lot easier to set up. In [15, p. 8] where they performed dead reckoning with and IMU on a pedestrian they reached an error of 1% of distance travelled.

Visual Odometry

Visual odometry works by determining motion with computer processing of visual information from high-resolution images of the surroundings taken by video cameras. This method is not subject to wheel slippage but suffers from drift like other incremental positioning methods but potentially performs better than inexpensive IMUs. It benefits from fusing with other methods such as wheel encoders and GPS. Algorithms that make use of multiple cameras usually perform better compared to monocular systems [17, p. 1].

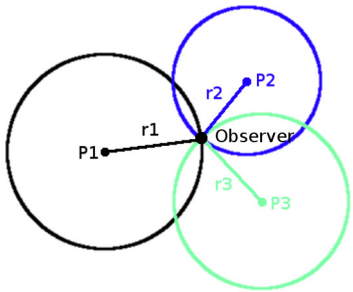

Figure 2.1.3: *Trilateration with 3 beacons at P1, P2 and P3 at distance r1, r2 and r3 from the object. The point where the three circles centered at each beacon intersect gives the position of the observer.*

2.1.2 Active Beacon Systems

Active beacon systems [6, p. 151] consist of active beacons placed at exactly known locations either sending or receiving signals to or from the object being tracked such as a car. Examples of active beacons are GPS satellites or smaller local systems using ultrasound, laser, radio signals or magnetic fields. Triangulation, illustrated in figure 2.1.2, is the process of calculating ones position from angular measurements, the angle of arrival of the signal from two or more beacons with known coordinates [12, p. 5]. With trilateration however, illustrated in figure 2.1.3, the position is calculated from distance measurements, given by the time of flight of signals from three or more beacons [16].

Ultrasound

Ultrasound positioning works by measuring the time it takes for a sound pulse to travel from a beacon to the object tracked or the other way round given three or more beacons and then using trilateration [38, p. 21] to calculate the position. The time travelled by each sound pulse is calculated by comparing the time of arrival to that of a reference such as a radio signal sent out simultaneously. This method can be made very precise however the speed of sound is affected by the medium it travels through so it needs a clear path and parameters like temperature and altitude of the room need to be accounted for. It is also sensitive to false signals from reflections in the environment, high frequency background noise, from electric motors for example, why filtering might be needed. Additionally the range is rather limited, usually just a few meters per beacon, why many beacons are required for covering larger areas making them suitable for indoor use in smaller rooms since the beacons are usually mounted in the ceiling.

Ultrasound positioning systems where the objects being tracked send out ultrasound pulses and have the stationary beacons listen to them as receivers, dubbed an active mobile architecture [4], such as Active Bat [1] and Hexamite Hx19 [19]. It is also possible to have the opposite where the tracked objects listen to the beacons which is called a passive mobile architecture which is normally the case with MIT Cricket. The latter approach with mounting receivers on the objects being tracked makes the system scale better with the number of objects tracked since there will not be any extra communication for every additional one which might limit the sampling frequency. The disadvantage with a passive scheme is that the signals from the beacons do not arrive simultaneously and will be received at different places if the receiver is moving, making the trilateration inaccurate which has to be compensated with least squares or extended Kalman filters (EKF) [30], a non-linear version of a Kalman filter which gives an estimation minimizing the mean-square error of a system's state given the current and previous states under linear and Gaussian conditions. In [4, p. 10] they locate a Lego train moving at 0.7 m/s with the passive MIT Cricket v2 location system combined with an EKF with an error of 20 cm compared to 3 cm for stationary objects [4, p. 7]. In a follow-up article [34, p. 2] they run the MIT Cricket system in a passive and active configuration for comparison, at 0.8 m/s the median positioning error is 10 cm in passive mode compared to 3 cm when it is active, at 1.43 m/s the passive system has a median error of 23 cm compared to 4 cm of the active system.

Table 2.1.1: Table of ultrasound systems.

Name	Accuracy at 0 m/s	Node range	Sample rate	Cost
Active Bat	3 cm [1, p. 3]	1.2 m grid	50	
Hexamite Hx19	9 mm [19]	14 m	20	1000$ for base system
MIT Cricket v2	3 cm [4, p. 7]	1.2 m grid	4	10$ per beacon/receiver [24, p. 1]

UWB and WiFi

There are different methods of using radiowaves for positioning applications. Special antennas can be used for determining the angle of arrival (AoA) of a signal and then triangulating, which requires two or more stations to work. For trilateration one can do time of arrival (ToA) measurements where a signal is sent from the reference station to the mobile device and back again with the exact times of sending and receiving signals being recorded. Time difference of arrival (TDoA) measurements is another approach where the time difference between at least 3 signals sent from reference stations arriving at the receiver are used to calculate the position. A prerequisite for both timing methods to work is that the time is perfectly synchronized at the receiving and transmitting end. A third option is to look at recieved signal strength (RSS) readings from local routers using

the propagation loss of WiFi signals to calculate the distance [12]. This method can be precise down to a few meters [8].

Ultra-wide band impulse radio (UWB IR) works by sending many short, impulse-like pulses in the time domain which creates a large bandwidth in the frequency domain. Having a high bandwidth has many advantages such as better resolution and lower sensitivity to reflections (interference from signals taking different routes to the receiver) compared to a narrow-band wireless system [22, p. 3]. UWB ranging is rather versatile in the sense that it does not need a clear line of sight between the transceivers although it affects accuracy negatively in different degrees depending on what material is blocking the signal. It is generally more accurate outdoors since there are less reflections from walls and other surfaces.

An example of a UWB system is the TimeDomain PulsON P400 and P410 transceivers using two-way time of flight (ToF) measurements [22, p. 5]. According to the specification sheet it has a precision of 2.3 cm standard deviation if the transceivers have a clear line of sight and an update rate of 118 Hz within 60 meters [7, p. 17]. The system is primarily intended for ranging between two transceivers but is used in the Gulliver project at Chalmers University of Technology as an absolute positioning system for 1:8 RC cars with three P400 transceivers used as stationary reference beacons and one on each vehicle [36, p. 23]. Since the software included is only meant for ranging between two transceivers one has to write one's own code for trilateration and switching between reference stations fast enough not to be a problem for moving vehicles. Scalability could also be a concern if different vehicles make simultaneous calls to the reference stations. These problems are addressed with a media access control (MAC) algorithm and reaches an average error of 63 mm moving at 1 m/s at an update rate of 23 Hz for one vehicle and 13 Hz for two [2, p. 31].

Another system utilizing UWB is Ubisense that uses tags transmitting UWB radio pulses received by an array of sensors that determine the location of each tag with time TDoA and AoA measurements down to an error of 15 cm in 3D under ideal circumstances. The tags have an update rate of 10 Hz. The system is used in a project at KTH Royal Institute of Technology for tracking scale model trucks and drones [21, p. 54]. This system also loses sampling frequency with the number of objects tracked.

Laser

Laser positioning with beacons usually works by having a rotating laser on the object being tracked and then triangulating by registering the angles of stationary passive reflectors that reflect the beam back on a photodetector [13] mounted above or below the laser or active beacons with photodetectors registering the laser beam. The laser needs a clear line of sight to the beacons and might be disturbed by reflections in the surroundings. It also has to rotate very fast in order for the triangulation to be accurate for moving vehicles. An example of a system using active beacons and a rotating laser with centimeter precision and good for speeds up to 6 m/s is the MTI Research CONAC from 1993 [6, p. 168] although it does not appear that it is available today 20 years later.

GPS

The global positioning system (GPS) consists of 32, as of December 2012 [10], satellites orbiting the earth. Simple GPS receivers calculate their position by comparing the time at which each radio signal travelling at the speed of light is received with the exact time when it was sent which is embedded into the message along with the satellites position from 4 or more GPS satellites. More advanced receivers also do carrier phase analysis from the L1 and L2 channels to improve accuracy. The main drawback of GPS is that the signal can be blocked by tall buildings and trees why it does not work indoors, furthermore the update frequency is normally quite low, a few readings per second, although there are receivers with a 20 Hz update rate such as the Venus638FLPx-L/D. To compensate for the outages and low update rate GPS positioning is normally coupled with other positioning methods like dead reckoning [9].

Simple GPS receivers like those normally included in smartphones have an error of approximately 8 m [11]. To improve upon the default accuracy differential GPS (dGPS) receivers use correction signals compensating for ionospheric conditions and other sources of error. Satellite based augmentation systems (SBAS) ground stations and geostationary satellites such as the regional systems WAAS (North America), EGNOS (Europe) and MSAS (Japan) [37] use the same global standard for communication with dGPS receivers and narrow

down the error to approximately 1-3 m [29]. There are also other, more precise dGPS services depending on country and region utilized by special receivers that reach sub-meter to decimeter accuracy with carrier phase analysis. They are considerably more expensive than standard SBAS dGPS receivers and might require a subscription for the correction signals. The best accuracy is reached with a portable real time kinematic (RTK) base station that sends a very precise correction signal resulting in an error of only a few centimeters [29] as well as repeatability making it possible to return to the exact same spot. Complete RTK systems are very expensive, costing 15 000 $ or more, although there are projects [35] and websites [32] providing code and instructions for building your own RTK station.

Table 2.1.2: Table of different GPS receiver systems.

Name	Accuracy	Cost
Standard GPS	8 m	50 $
Differential GPS with SBAS	1-3 m	100 $
Real-time kinematic (RTK)	< 3-5 cm	15k $

2.1.3 Landmark Navigation

Landmark navigation utilizes the known position of natural or artificial landmarks easily identifiable by the vehicles sensors to determine the position [6, p. 173]. Global vision systems where stationary cameras track the objects are included in this category since the objects themselves can be thought of as artificial landmarks or the patterns or markers attached to them for making tracking easier. They are covered in the next chapter as an introduction to the testing of the Qualisys ProReflex system.

Hagisonic Stargazer

Hagisonic Stargazer(table 2.1.3) is a commercial indoor positioning system made for mobile robot applications. It uses an infrared (IR) camera to look at passive landmarks reflecting IR light resembling QR codes placed in the ceiling. It reports heading as well as 2D position. According to the manufacturer it works at speeds up to 15 m/s and is resistant to interference from lamps and sunlight. An advantage with this system is that there is no loss of sampling frequency with many robots and they can use the same passive landmarks without interference. It has been used for playing Pacman with Roombas [31]. Although it is relatively inexpensive compared to some other systems the system only tracks one vehicle and one additional unit has to be bought for each vehicle.

Table 2.1.3: Hagisonic Stargazer HSG-A-02 [14] specifications.

Property	Value
Framerate	10 Hz
Localization Range	1.6 - 6.3 m in diameter, 2.6 m typical for ceiling height 2.5 m
Precision	1 mm (center area), 10 mm (marginal)
Heading angle resolution	< 1.0 degrees
Max number of landmarks	31 (3x3 type), 4095 (4x4 type)
Price	1000 $

2.1.4 Map Matching

Map matching is mostly used by robots moving at low speeds combined with dead reckoning methods. It works by comparing readings from the robot's sensors to a stored map of the environment and then using a search algorithm to find a match inside the map and senor readings which gives the position and heading of the robot. A common sensor for this type of application is a sweeping laser range finder (LRF) that gives a depth map of the environment in its field of view. The map stored in the robots memory can either be made by the user from e.g. floor plans or generated by the robot itself with simultaneous localization and mapping (SLAM). The advantage of this method is that no modifications to the environment is required, making the robot or vehicle truly autonomous. A drawback is that a lot of computing power is needed to perform the search in

the map limiting the refresh rate and introducing some lag during the search why it is usually combined with dead reckoning. As implemented in [33] with a Hokuyo URG-04LX 2D laser rangefinder scanner and stochastic search algorithm the robot is assumed to stand still during localization. In a newer report [5] a Microsoft Kinect sensor is used and compared to the same LRF with a different algorithm.

Microsoft Kinect

The Microsoft Kinect is an input device for the Xbox 360 video game console developed by Microsoft. It consists of an RGB camera and a depth sensor that consists of an IR laser emitting a pattern that is captured by a CMOS camera with an IR-pass filter [3, p. 6]. Its specifications are listed in table 2.1.4. Multiple Kinects will interfere eachother because they are sensitive to other sources of IR light such as the sun. The depth sensing range is also rather limited, with the OpenNI SDK it is 0.5 m to 9.7 m [3, p. 14] and the resolution decreases with range. Kinect has successfully been used as a cheap alternative to an LRF in map based positioning with an offset error of less than 20 cm compared to less than 3 cm of the LRF and a heading error of roughly 1.5 degrees [5].

Table 2.1.4: Kinect hardware specifications from variuos documents according to [3, p. 6].

Property	Value
Framerate	approx. 30 Hz
Angular Field-of-View	57 horz., 43 vert.
Nominal spatial range	640 x 480 (VGA)
Nominal spatial resolution (at 2m distance)	3 mm
Nominal depth range	0.8 m - 3.5 m
Nominal depth resolution (at 2 m distance)	1 cm
Price	100 $

2.2 Summary of Commercially Available Positioning Systems

See table 2.2.1 for a summary of different commercially available positioning systems that work indoors, for this reason GPS systems are not listed. The sampling rate for the Hexamite, Ubisense and PusON positioning systems are lower when tracking more than one object simultaneously. The cost for the Qualisys system is a rough estimate of what the software and 4 cameras cost. The Ubisense, PulsON and Hexamite systems are starting kits that can be expanded. For the Hagisonic system landmarks need to be bought separetely as well as another unit for each additional vehicle. PulsON is just a ranging system between transceivers that requires a lot of work by the end user to become a positioning system, the data given here is for the system as implemented in the Gulliver project [2, p. 31].

Table 2.2.1: Table of commercial systems sorted by cost. Precision is the average Euclidean error. Sample rate is the number of position readings per second for one vehicle. Cost is for a complete system based on inquiries in April 2013. Range denotes the range of cameras and beacons or the spacing between landmarks. Heading is whether the system reports heading.

Name	Technology	Precision	Sample Rate	Cost	Range	Heading
Hexamite Hx19	Ultrasound trilateration	1 cm	20 Hz	1k $	14 m	No
Hagisonic	IR landmarks in ceiling	1 - 10 mm	10 Hz	1k $	1.6-6.3 m	Yes
TD PulsON P400/P410	UWB RSS ToF	6 cm	23 Hz	10k $	50 m	No
Ubisense	UWB AoA ToA	15 cm	10 Hz	16k $	50 m	No
Qualisys Oqus 1	IR cameras and markers	1 mm	247 Hz	27k $	25 m	Yes

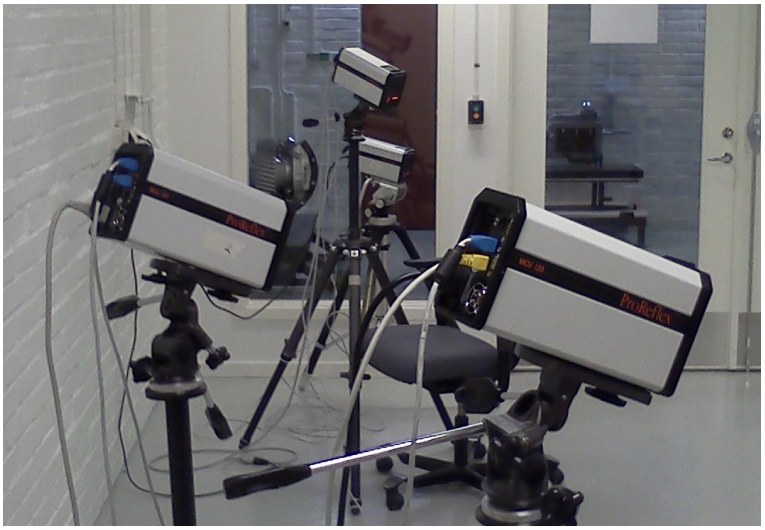

Figure 3.1.1: *A picture of 4 Qualisys ProReflex MCU 120 mounted on tripods.*

3 Method

3.1 Global Vision Systems

Global vision systems use stationary cameras for positioning and have the advantage that they give a good overview of the test area and scale well with the number of objects tracked since one simply adds more markers. A major drawback is that the cameras have limited range and require a clear line of sight. Covering large areas becomes very expensive because it requires many cameras with high resolution.

There are many manufacturers of motion capture systems today such as NaturalPoint, Qualisys, MotionAnalysis and Vicon. Some manufacturers offer smaller budget systems for small volumes such as the MotionAnalysis Osprey and NaturalPoint OptiTrack V100:R2 and Flex 13 [27] with low resolution compared to the really expensive high end cameras with 12 MP resolutions and high framerates, see table 3.1.1 for some example systems.

Table 3.1.1: Table of different motion capture cameras from Qualisys, NaturalPoint and MotionAnalysis.

Property	Oqus 1	Flex 13	Prime 41	Osprey
Max FPS	247	120	180	245
Horizontal field of view	20 to 56 degrees	56 degrees	51 degrees	
Measurement distance	25 m	12.2 m	30.5 m	
Image sensor resolution (pixels)	640x480	1280x1024	2048x2048	640x480
Marker resolution (subpixels)	41000x31000			
Latency	4.2 ms	8.3 ms	5.5 ms	
Camera price	5k $	1k $	5k $	
Software price	7k-24k $	1k $, SDK is free	1k $, SDK is free	

3.1.1 Qualisys ProReflex

The Qualisys ProReflex motion capture system in table 3.1.2 uses infrared cameras pictured in figure 3.1.1 and markers that are either passive reflecting or active sending out IR light themselves that the cameras see. The cameras are connected to a computer running the Qualisys Track Manager software that analyzes the 2D images of the IR markers from the cameras and puts them together into a 3D view of the markers. The accuracy depends on the number of cameras in the system, their resolution as well as the calibration quality and their position and orientation. At least two cameras at any given time must be able to see each marker for

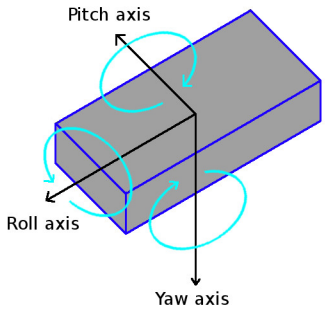

Figure 3.1.2: *The yaw, pitch and roll axes of a rigid body.*

its 3D position to be determined from different angles separated by at least 30 degrees [25, p. 291]. It is also possible to do 2D measurements if the markers move in a plane perpendicular to the cameras.

The cameras have very high resolution and can detect motions as small as 50 microns [28]. The system is versatile in that it is easy to add more cameras with a wide variety of different markers in different sizes and active ones making it possible to cover very large measurement volumes. The update frequency scales well with the number of markers as a camera is able to track 150 markers at 60 Hz. The software has support for 6 degrees of freedom (6 DOF) body tracking reporting the yaw (heading), pitch and roll illustrated in figure 3.1.2 along with the 3D position. The main drawbacks are that the software license and cameras are very expensive (costing between 5 000 - 10 000$ each), the cameras are not so easy to move around outdoors since they need to be connected to each other by network cables and supplied with 230V AC power and protected from water. Additionallly, calibration of the system before a measurement is done becomes increasingly complex for larger measurement volumes. The cameras are sensitive to IR noise in the environment, e.g. shiny objects have a tendency to be falsely identified as markers. The precision decreases with larger measurement volumes if the camera coverage is thin and the 6 DOF body tracking needs clearly separated markers.

QTM has earlier been used for positioning purposes in a thesis work at KTH [21] where they used a labview plugin to communicate with QTM.

Table 3.1.2: ProReflex specifications [28]

Property	Value
Max Frequency (depending on model)	120, 240, 500, 1000 Hz
Horizontal Field-Of-View	10 to 45 degrees
Range	0.2-70 m
Image sensor resolution	658x500 pixels
Effective Resolution	20,000x15,000 subpixels
Max number of cameras	32
Max number of markers	150 at 60 Hz

3.2 Qualisys Track Manager

This section contains instructions and notes on how to set up the system, namely the ProReflex MCU 120 cameras and use the Qualisys software version 2.7 for 6 DOF tracking serving as a summary of the relevant chapters in the QTM 2.7 manual [25].

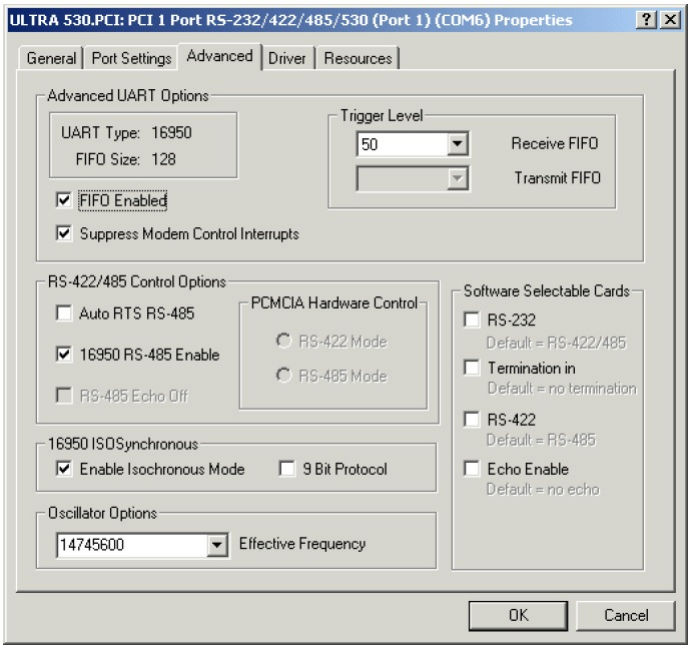

Figure 3.3.1: *Picture of the driver configuration of the Sealevel PCI ULTRA 530.PCI 7101 PCI card [25, p. 505].*

3.3 Installation and Setup

3.3.1 Installation of the PCI Card

The system used is the Qualisys' ProReflex 120 Motion Capture Unit (MCU) with a maximum measurement frequency of 120 Hz. To connect the cameras, one first hast to install a serial communication board which in this case is a Sealevel PCI ULTRA 530.PCI 7101 PCI card that connects the cameras to the measurement computer with Windows 32-bit 2000/XP or newer. This is not entirely trivial since the jumpers on the Sealevel PCI card have to be in their respective positions pictured on page C-3 in the QTM manaual [25, p. 495], although if it has been used before with the cameras it is probably already taken care of. After the PCI card is inserted into a PCI port on the measurement computer one has to install the drivers taken from the board manufacturer's (Sealevel) homepage. When this is done and the computer recognizes the card one has to enter the device manager and configure the card from its drivers, this is described on page C-13 where you copy the settings from a screenshot 3.3.1. This last step is very important because if the serial communication board is not configured properly the QTM software will not detect the cameras.

3.3.2 Connecting the Cameras

When the PCI card is installed and configured the cameras are connected according to algorithm 1 [25, p. B-3].

3.3.3 Camera Placement

When placing cameras there are a few things to keep in mind.

- Cameras cannot directly stare into each other unless the IR illumination is turned off which is necessary for tracking passive markers. If the floor is reflective enough they cannot face each other even though they are looking at the floor.

- Try to cover the measurement volume from as many angles as possible. For good 3D data a marker has to be viewed by two cameras from two different angles separated by at least 30 degrees, 60 is preferable. See "Measurement guidelines" [25, p. 291].

Algorithm 1 Connecting the camera system

Steps

1: Connect the blue network cables from the port labeled "next" to the port labeled "prev" on the next camera in the chain leaving the first camera in the chain with an empty prev port and the last one with an empty next port.
2: Connect the data port of the first camera in the chain to the serial port of the PCI board in the computer with QTM.
3: Make sure all the cameras are switched off before connecting power.
4: Connect the power.
5: Switch on the first camera in the chain and the rest of the cameras will do it automatically.
6: Start the QTM software, load or create a new project.
7: Go to "Tools → Project options → Capture → Camera System → Connection", click "Locate system" .
8: Under "Camera System → Linearization", click "Load all from folder" and locate the folder with the linearization files on the CD supplied with the cameras.

- For extended wand calibration to work it is a requirement that at least two cameras can see the enitre measurement wand during calibration at a given time which means that there has to be enough overlap that 3 cameras can view the wand in any angle when it exits the field of view (FOV) of a camera.
- Cover any shiny objects that might falsely be identified as markers.
- A marker should be larger than 4 pixels to be identified, configured in the aim menu.

Algorithm 2 is a procedure for placing the cameras when they are mounted on stands so that they can be freely placed in the room as well as tilted and rotated.

Algorithm 2 Placement of the camera system

Steps

1: Click "File → New" to fill the workspace with black frames of what the cameras are seeing in the 2D view. Markers are represented by white dots.
2: To view what each camera is seeing click "View → Data info window" and select the camera you are interested in and it will list 2D data about every marker it is seeing, namely the coordinates and size.
3: Mark out the desired measurement volume with cones and place markers on them to make them visible by the cameras.
4: Adjust the cameras to see as many of the cones as possible from as different angles as possible.
5: **while** every cone is not visible by at least two cameras **do**
   - move a camera further away from the volume
   - move the outlying cone(s) closer to the visible ones.
6: **end while**
7: Check that the whole measurement wand is visible by at least 2 cameras inside the entire measurement volume meaning 3 in places of overlap between cameras. It helps to remove the markers from the cones in this step and look at the number of identified markers on each camera which is two when the entire wand is visible.

3.3.4 Calibration

There are several ways of calibrating the camera system in QTM described in [25, p. 149]. Included with the system was a wand calibration kit why the focus in this section lies on how to successfully perform a wand calibration described in algorithm 3 and [25, p. 275]. The software switches automatically to extended wand calibration when all cameras cannot see the entire measurement volume.

3.3.5 Making a Measurement

Making a measurement is described in algorithm 4.

Algorithm 3 Wand calibration in QTM

Steps

1: Click "File → New" to open a new window.
2: Place the L-shaped reference frame in the measurement volume where you want the origin of the coordinate system to be and rotate it in the desired direction. The coordinate system can later be translated and rotated under "Workspace options → Capture → Camera system → Calibration → Transformation".
3: Enter the exact wand length labeled on the wand in "Tools → Project options → Capture → Camera system → Calibration" which is 750.7 mm in our case.
4: Click "Capture → Calibrate" to enter the calibration dialog.
5: Set the calibration time to 60 seconds, a time offset that will give you time to move into position and start the calibration.
6: Move the wand back and forth slowly in the entire measurement volume without rotating it ensuring that both ends are visible by at least 2 cameras the entire time. Spend extra time in "important" areas where the cameras overlap and places where the markers will be a lot during the measurement.
7: Rotate the wand slowly over different areas.
8: **if** the calibration fails **then**

- Increase calibration time.
- Don not move the wand outside of the measurement volume and make sure to cover it entirely.
- Check that the cameras are placed correctly and that the places of overlap (for extended wand calibration) are large enough to fit the entire wand in all angles.

9: **end if**

Algorithm 4 Making a measurement in QTM

Steps

1: Click "File → New" to open a new window.
2: Switch between 2D and 3D views by clicking on their respective buttons in the panel to the left.
3: To view the 3D coordinates of markers click "View → Trajectory info window → All" and it will display the 3D coordinates of each tracked marker in the coordinate system defined during calibration.
4: To make a measurement either click the capture icon (the red rec icon) or choose "Capture → Capture" and enter its duration, time offset, etc.
5: Click "Start" to start the capture.

6 DOF Tracking

6 DOF tracking tracks the position in the three axes and the orientation either as a matrix or the 3 Euler angles, see figure 3.1.2, of a rigid object with at least 3 markers attached to it. 6 DOF tracking is described in algorithm 5 from "6 DOF tracking of rigid bodies" [25, p. 331] and "6 DOF tracking" [25, p. 218] in the manual. Figure 3.3.2 shows a screenshot containing the 6 DOF tracking options.

Algorithm 5 Tracking of 6 DOF bodies in QTM

Steps

1: Enable 6 DOF tracking in "Project options → Processing → Real time actions" tick the boxes "Track each frame in 3d" and "Calculate 6DOF".

2: Go to "Tools → Workspace options → Capture → Processing → 6DOF tracking" where one can either acquire, load or add a body manually from labeled markers.

- Load a saved body stored as an xml file saved from an earlier measurement or project. Export a body from a project or recording by clicking "Save Bodies".
- Add a body and import markers from labeled markers (trajectories). If the camera system is not connected and you want to track a body from a recording this is the best way to do it if you cannot load the body definition. To label unidentified trajectories right-click on an unidentified trajectory in the unidentified trajectories window ("View → Trajectory Info Windows → All") and choose "Identify, Labeled". Then in the 6 DOF tracking section click "Add Body" and "Import Markers" and choose the labeled trajectories that belong to the body.
- Click "Acquire body" to add all currently seen markers to the body. Remove markers that do not belong by looking at their coordinates in the labeled trajectories window. Highlight a marker by clicking on it in the 3D view and compare it with the coordinates of the markers included in the body listed under it. Click on the markers that do not belong to the body and press delete.

3: Once the body is newly defined the origo of its coordinate system is usually somewhere far outside of the body so it helps to move it into its geometric center which is done with "Translate body → To the geometric center of the body".

4: Reset its orientation with "reset orientation". It is important to keep track of how the orientation is set when streaming so that they are correctly read by the client. The coordinates are defined according to the 6D Euler standard and the angles according to the QTM standard which can be altered in "6 DOF Tracking → Euler Angles". By default the angles are defined clockwise.

5: Look at the bodies' coordinates and angles by clicking "View → Data Info Window", then right click inside the window and choose "Display 6DOF data".

3.4 Measurement and experiments

3.4.1 Test setup

The camera system was setup according to figure 3.4.1 and figure 3.4.2. As mentioned before, the Qualisys IR cameras cannot stare directly into each other because they radiate IR light in order to illuminate the reflective markers and if the floor is shiny enough they cannot face each other even though they are looking at the floor. Furthermore the limited field of view of a camera requires that the measurement volume has to be a few meters in front of it. In order to maximize the measurement volume inside the rectangular lab room all of the cameras were placed along the shortest wall with two cameras in each corner to maximize separation. The outer cameras capture far away markers and the two inner cameras film close objects. The further away an object is, the separation, the difference in angles between which the cameras see an object, becomes smaller and the precision suffers which is illustrated in figure 4.1.4. All in all there is a tradeoff between the size of the measurement volume and the precision.

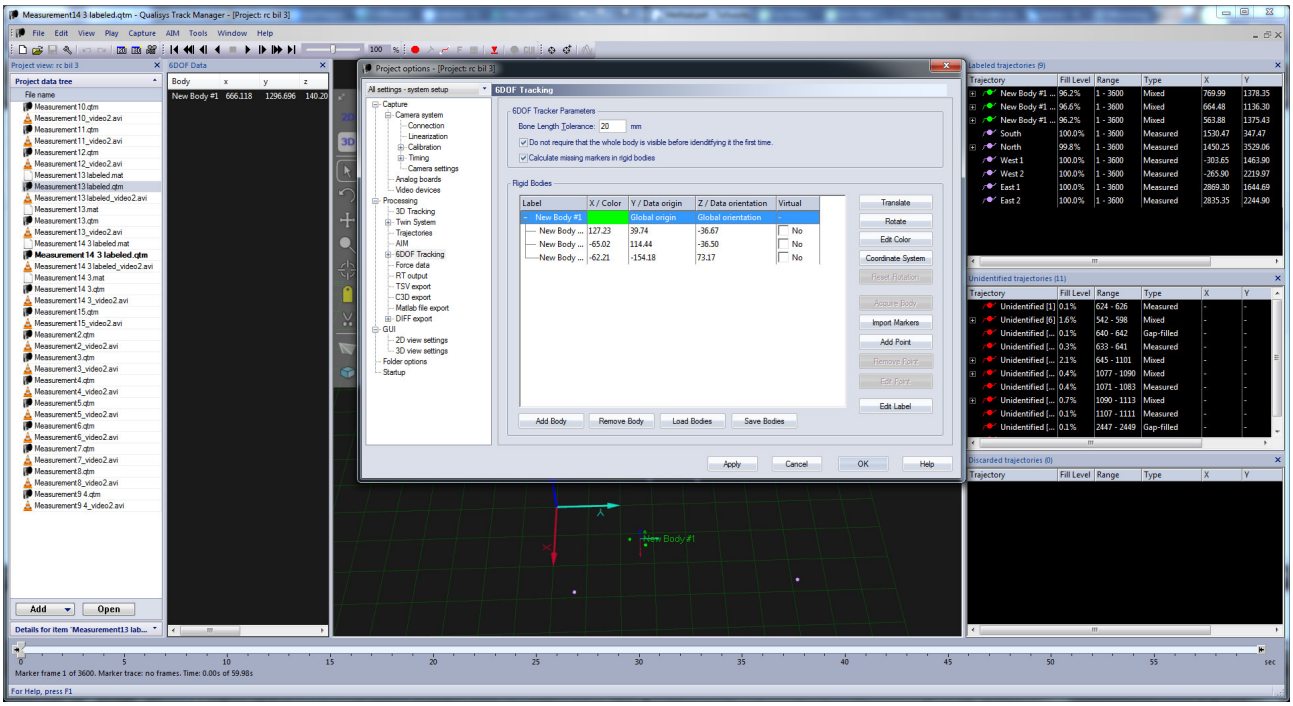

Figure 3.3.2: *Screenshot of QTM and its 6 DOF bodies window as well as the trajectories window to the right.*

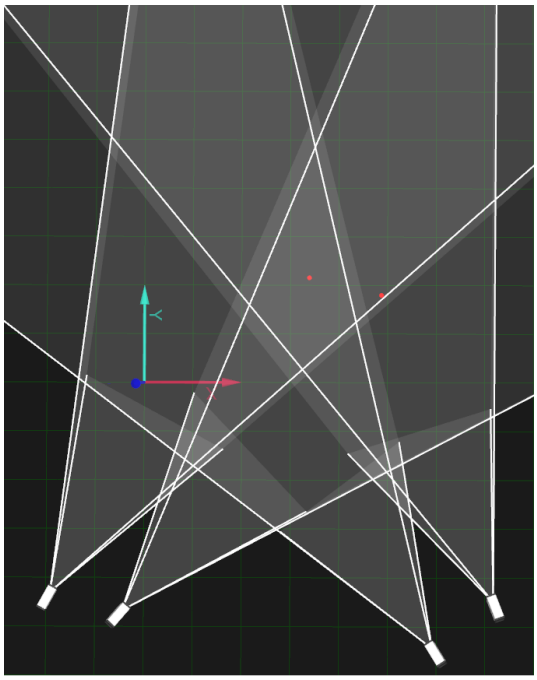

Figure 3.4.1: *Screenshot from QTM of the measurement viewed from above. The shaded white cones are the field of views of the cameras standing against the wall at the bottom of the figure. The two red dots are the markers of the measurement wand. The scale of the grid is 0.5 x 0.5 m.*

Figure 3.4.2: *Picture of the camera arrangement.*

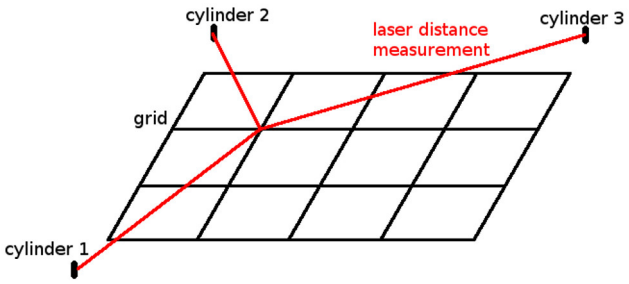

Figure 3.4.3: *Trilateration with a laser distance meter.*

3.4.2 Distance Measurements

First, a test was done to see what the precision of the coordinates is within different places inside the measurement volume by marking out a 5 x 5 grid on the floor with a 20 m long measuring tape using the triangle method, forming a right triangle with 3 spots where the sides have a 3:4:5 ratio. The size of the grid was 3.375 x 4.5 m with an error of a few cm farthest away from the origo since each rectangle wasn't checked with the triangle method and the flexibility of the measuring tape although it was kept taut.

The grid was later checked with a Leica E7400x laser distance meter used to determine the location of each grid point more accurately by trilateration. The cameras were not set up during this measurement since the distance meter uses a class 2 laser and pointing it at the cameras might damage the image sensors. Three solid cylinders were placed far away from each other outside of the grid serving as reference points. For each grid point the distance meter was placed on the floor and the distance was measured to the three reference cylinders, according to figure 3.4.3. The laser would sometimes overshoot the cylinders or hit the floor because of the floor's unevenness so the laser had be to aimed vertically as well. When the measurements were done the XY coordinates were calculated with a program written in Matlab by trilateration. The measured points lie on circles around the reference points and are compared two by two to get the points of intersection. The points of intersection are either two real numbers, one number if the readings are perfect or two complex numbers if they don not intersect at all. Assuming the circles do the intersect at two points one point is wrong and the other is right. The program chooses the "right" point of intersection between the two circles by picking the one with the smallest distance to the rest of the intersection points and then takes the average of the 3 correct intersection points to get the location.

Secondly a fixed distance, the calibration wand or a ruler with 40 mm half spherical markers attached to it, was measured in different orientations inside the measurement volume to check the precision over smaller distances and how it is affected by its orientation relative to the cameras. The reason for using the ruler was that the markers on the wand were too small to be reliably detected by the cameras far away (beyond approx. 6 meters close to the floor) from the cameras in the measurement volume.

3.4.3 RC Car Test Runs

This test was performed to see how well this particular experimental setup along with different marker configurations work in practice for the intended application, 6 DOF tracking of an RC car driving around a track marked out with cones within the measurement volume, illustrated in figure 3.4.4. For illustratory purposes a webcam was connected to the measurement computer to film the car during the capture in QTM. QTM has support for video capture devices which are configured in "Project options → Capture systems → Video devices" where one can simply add the connected webcamera and the view from the webcam shows up as another camera window in the 2D view, which can be seen in figure 4.2.8. Each test run was 60 seconds long and consisted of parking between two pairs of adjacent cones to the right and left of the middle of the track then rounding one near and one far away cone and driving in circles in the middle of the track. The maneuvers were performed in random order in different directions to test the tracking from different angles of the body at different places throughout the measurement volume.

3.5 Network Program

3.5.1 QTM RT Protocol

The QTM software has a built-in server called the real time (RT) server that can send the processed motion capture data from the cameras over a TCP/IP or UDP/IP connection under the QTM RT server protocol described in the RT protocol manual version 1.10 [26].

The difference between the UDP and TCP protocols are that TCP is a reliable connection between two programs that makes sure that the data that was sent is received by the other party in the order it was sent. In contrast, UDP works like the postal service and sends packets of data called datagrams which is an unreliable connection as the datagrams are not guaranteed to arrive and might do so in any order. Both protocols use ports to direct data received by the computer to its running processes [23]. UDP is usually

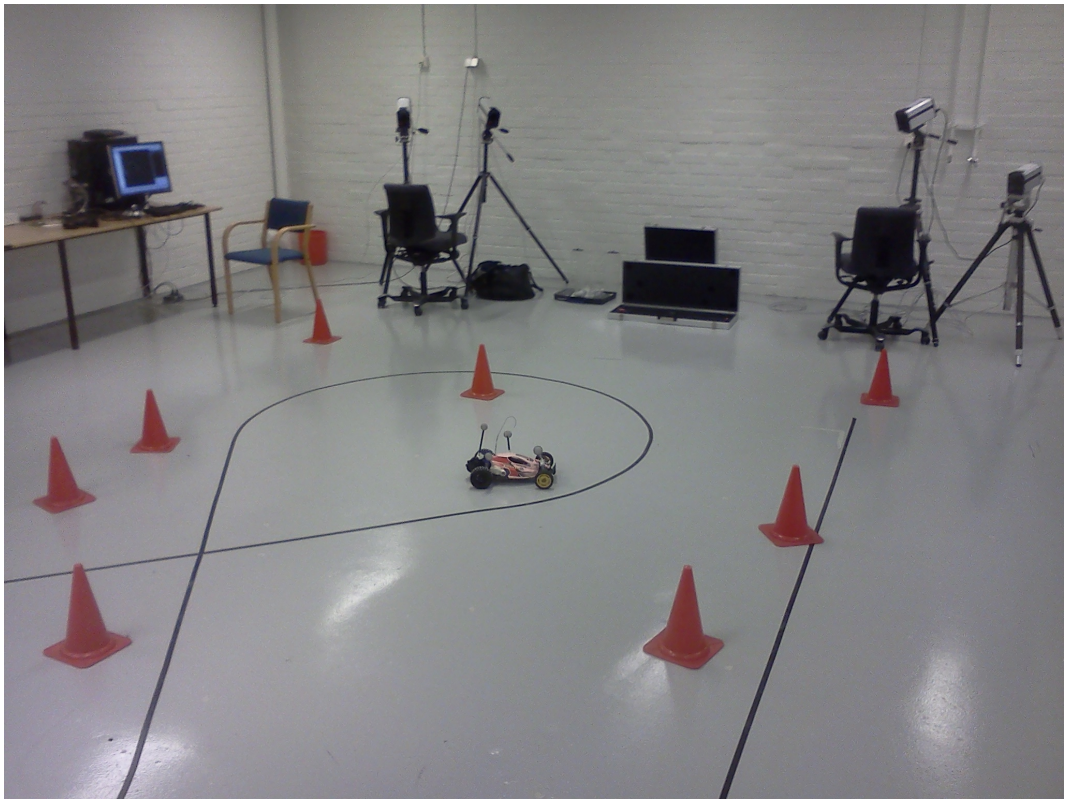

Figure 3.4.4: *Picture of the experimental setup with 6 DOF tracking of an RC car driving around the test track.*

faster with no unnecessary overhead why it can be suitable for streaming applications as when streaming motion capture data where each frame is more or less independent with an individual timestamp and frame number.

The packet structure of the RT protocol is described on page 24. For the communication to work every packet sent between the server and client must follow this format. Each packet starts with a header of 8 bytes denoting two 32 bit integers. The first integer represents the total size of the packet in bytes and the second integer is the packet type. After the header follows the data of the packet which is size - 8 bytes. The data is usually a string with a command and its parameters which is converted into a byte array representing each character and is always terminated with \0. The packet types we are interested in are primarily the command packets and error packets denoted with 1 and 0 respectively. The command packets are commands sent to, or a response from, the server indicating that the last command was successful. An error packet is sent from the server when the last command generated an error and has an error message in the data portion of the packet. An example of the error packet is as follows, where each byte is separated by an underscore.

```
0_0_0_31_0_0_0_0_C_o_m_m_a_n_d_\32_n_o_t_\32_s_u_p_p_o_r_t_e_d_._\0
```

The commands we are interested in are the following, Version and StreamFrames. The version command looks like:

```
0_0_0_21_0_0_0_1_V_e_r_s_i_o_n_\32_1_._1_0_\0
```

and is sent as a byte array (separated with whitespace) like:

```
0 0 0 21 0 0 0 1 86 101 114 115 105 111 110 32 49 46 49 48 0
```

It is the first command that the server expects the client to send after connecting and decides which version of the RT protocol to be used. The lastest version at the time of writing is 10 in version 2.7 of the QTM software so that is what the program is using.

Then there is the streamframes command which can be used as:

```
StreamFrames AllFrames UDP:30000 3DNoLabels
```

The StreamFrames tells the server to start streaming data frames in real time [26, p. 18]. This works if a new window is open in QTM or during an ongoing capture. The AllFrames parameter tells the server to stream every processed real-time frame and can be exchanged with Frequency:30 or another number lower than the real time processing frequency in QTM. The theoretical limit of this number is the measuring frequency of the cameras which in this case is 120Hz but is further limited by the processing power of the computer that the QTM software is running on and the number as well as types of objects tracked. During the measurements in this report the real time frequency was set to 50 Hz. UDP:30000 denotes that the server should send a UDP stream to the UDP port with number 30000 of the client. It is also possible to supply an IP adress like "UDP:address:port" which will make the server stream to that address instead of the address of the client. "3DNoLabels" tells the server which type of data should be sent, in this case it is unlabeled trajectory data, i.e. the 3D position of unlabeled markers. To stream 6 DOF data (3D position coordinates and the three Euler angles) add "6DEuler" as a parameter. This way one can stream many different types of data in each frame, every data type in a frame is called a component.

To make the server stop streaming the stop parameter is sent.

```
StreamFrames Stop
```

There are many more commands listed in the manual where one can take control over QTM over the network but these two commands are the only necessary ones for streaming marker and body data in real time.

The StreamFrames command will stream data packets containing one or more components specified in the count field. Every component has a header just like the packet header. The data header looks like table 3.5.1. They are all 32-bit integers except the timestamp which is a long and denotes the number of microseconds

Table 3.5.1: Data frame header

Name	Size	Type	Time Stamp	Frame Number	Component Count
Size (bytes)	4	4	8	4	4

from the start. The type is 3 and the size is 8 bytes packet header + 12 bytes data frame header + the size of the components and their headers. Component data for markers or bodies look like table 3.5.2. The type is 2 for unlabeled markers and 6 for a 6 DOF Euler component. The 2D drop and out of sync rates are 16-bit integers that range between 0 and 1000. A high value of the drop rate means that the cameras are set at a too high frequency for the network to handle, and a high out of sync rate that the cameras cannot process fast enough. After the component header the data for each marker or body is repeated Marker/Body Count

Table 3.5.2: Marker or body component header.

Name	Size	Type	Marker/Body Count	2D Drop Rate	2D Out Of Sync Rate
Size (bytes)	4	4	4	2	2

times. The marker data in table 3.5.3 consists of the coordinates that are 32-bit floats and an integer ID that identifies each marker between the previous frame and the next.

Table 3.5.3: Unlabeled marker data

Name	X	Y	Z	ID
Size (bytes)	4	4	4	4

For 6 DOF bodies with Euler angles the data repeated for each body looks like table 3.5.4 where every field is a 32-bit float. All in all, first there is the data packet header describing size of the total packet and type, then for each component such as 6 DOF Euler or unlabeled marker data there is the component header followed immediately by the data for each body or marker.

3.5.2 Java Test Program

Java and Eclipse were used for writing the application (Appendix A.3) that communicates with the RT software because Java is platform independent so it can be run on either the computer with QTM which is a windows

Table 3.5.4: 6 DOF Euler data

Name	X	Y	Z	Roll	Pitch	Yaw
Size (bytes)	4	4	4	4	4	4

program or the server gathering data running Linux. Java has easy to use classes for handling network traffic built in as well as array lists for handling large amounts of data.

The test program uses a "simplified" model view controller (MVC) layout where the communication between the views and models is tunnelled through a controller class. In this case the view consists of a Swing window with the user interface as in figure 3.5.1 and the models of objects handling the TCP/IP and UDP communication dubbed the TCP/IP client (Appendix A.1) and UDP client (Appendix A.2). They both have a thread running that listens for packets from the RT Server. The TCP/IP client handles the communication with the server such as sending start and stop stream commands while the UDP client stores the array of streamed frames and saves it on the disk when asked. Each time a new stream is started it starts a new UDP thread listening for UDP packets. The UDP thread saves the data (coordinates, angles and timestamps) into an array and notifies the view through the controller with coordinates of each body or marker on the screen with a specific time interval. Once it detects a timeout it sends the array to the UDP Client object before stopping.

When running the program the first thing that happens once the user presses the "Connect" button is that the TCP/IP client object connects to the RT server over TCP/IP to a specific port with the desired endianness and receives a welcoming message from QTM. QTM has different ports to connect to depending on byte order and protocol version or type. The base port is normally 22222 unless specified otherwise under "Tools → Project options → Processing → RT output" in the program. The base port only supports the 1.0 version of the protocol where some important commands and parameters are missing, such as the UDP-flag for the streamframes command why the 22223 (little endian) and 22224 (big endian) ports are the interesting ones which support every protocol version up to the latest one for that version of QTM. Little endian means that the least significant bit has the largest index when representing numbers and big endian is the opposite. The headers need to be in the appropriate endian depending on what port you connect to (big endian for the big endian port and vice versa).

To connect to QTM with TCP/IP in Java, a TCP/IP socket object is created with the address and port of the QTM server as constructor arguments.

```
Socket client=new Socket("localhost",22224);
```

If the connection was successful the server will respond with the string "QTM RT Interface connected". To receive messages from the server we need to get the input stream of the connection by creating a buffered reader that reads the input stream from the TCP/IP socket.

```
BufferedReader in= new BufferedReader(new InputStreamReader(client.getInputStream()));
```

Each byte from the input stream is read with the read method of the input stream.

```
int lastReadInt = in.read();
```

Because the first 8 bytes constitute the header consisting of the message length and type according to the RT protocol, we read the first 4 bytes, convert it to an integer and we get the length of the message in bytes. Then we call the read method length of message - 4 times. The message starts at byte number 8 and onwards. Since we're streaming data over UDP we only expect server responses to commands or error messages for debugging purposes why there is no type check. It is possible to write a more advanced client that can receive coordinate 3D (c3d) files as well as other types exported by QTM from the server.

To send messages or commands to the server, a printstream object is created that sends messages through the output stream of the connection.

```
PrintStream out=new PrintStream(client.getOutputStream());
```

To send a message use the write method of the output stream with a byte array as input formatted according to the RT protocol documentation and then flush the stream to make sure it is sent.

```
out.write(byteArray);
out.flush();
```

As described in the previous section the message should consist of 8 bytes header, the first 4 bytes are an int32 in little endian denoting the total message length and then an int32 with the type which is usually 1 since it is a command. Then comes the message string which is a char array converted into a byte array which is easily done with the getBytes method in Java.

After the connection has been made and the in- and outputstreams are created the program sends the "Version 1.x" command to change the RT protocol version. If it is not sent then version 1.1 of the RT protocol will be used [26, p. 6]. For version 2.7 of QTM version 1.10 of the RT protocol is the latest. Then the program sends the StreamFrames command which tells QTM to start streaming. The stream is ended by sending the message "StreamFrames Stop". The UDP thread detects this with a timeout of one second and then asks if the user wants to save the data.

To receive the UDP stream of data frames from the RT server a DatagramSocket is created where the address is the address of the QTM server and rPort is the UDP port that the server is asked to stream to.

```
DatagramSocket rsocket = new DatagramSocket(null);
rsocket.setReuseAddress(true);
rsocket.bind(new InetSocketAddress(address, rPort));
```

UDP packets are received with the code below.

```
byte[] buffer = new byte[2048];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
rsocket.receive(packet);
```

Then the contents of the packet are put into a byte array which is formatted according to the RT Protocol documentation, described in the previous section.

```
byte[] buf2 = packet.getData();
```

As an example, the packet size is read with the following code.

```
int packetSize = MyByteOperations.my_bb_to_int(new byte[]{buf2[0], buf2[1], buf2[2], buf2
    [3]});
```

Then it is a matter to loop through the byte array and read each component depending on type and convert the bytes into ints and floats describing the marker or body positions and angles.

3.5.3 Running the Java Program

To run the Java program with the camera system connected, start QTM, load or create a new project, calibrate if necessary, click new to open a new window and turn on the real time processing described in 3.3.5. It is important that a new window is open otherwise it will not stream.

If you want to run the program for debug purposes without the camera system connected, open a project and a measurement and then under "Capture" choose "Run Real-Time Processing on File" and tick the "Track each frame in 3D" box.

When QTM is ready to stream, run the Java program, a screenshot of it can be seen in image 3.5.1. Start by entering the correct IP address of the computer running QTM and click "Connect". Once that is done edit options such as data type, frame rate, the default orientation of bodies (taken counter-clockwise from the x-axis) and click "Start Streaming" to tell QTM to send a UDP stream. It is also possible to send it to another computer by entering a different UDP address in the UDP address box. Click "Stop Streaming" to stop streaming and then "Save" to save the streamed frames to a file. The streamed frames are stored in memory until the next stream is completed. While the stream is running it is possible to alter the offsets and scale in the boxes to the left control the plotting in the graph in the middle. Markers are represented by blue balls and bodies by a blue ball with a blue line indicating its heading (yaw angle).

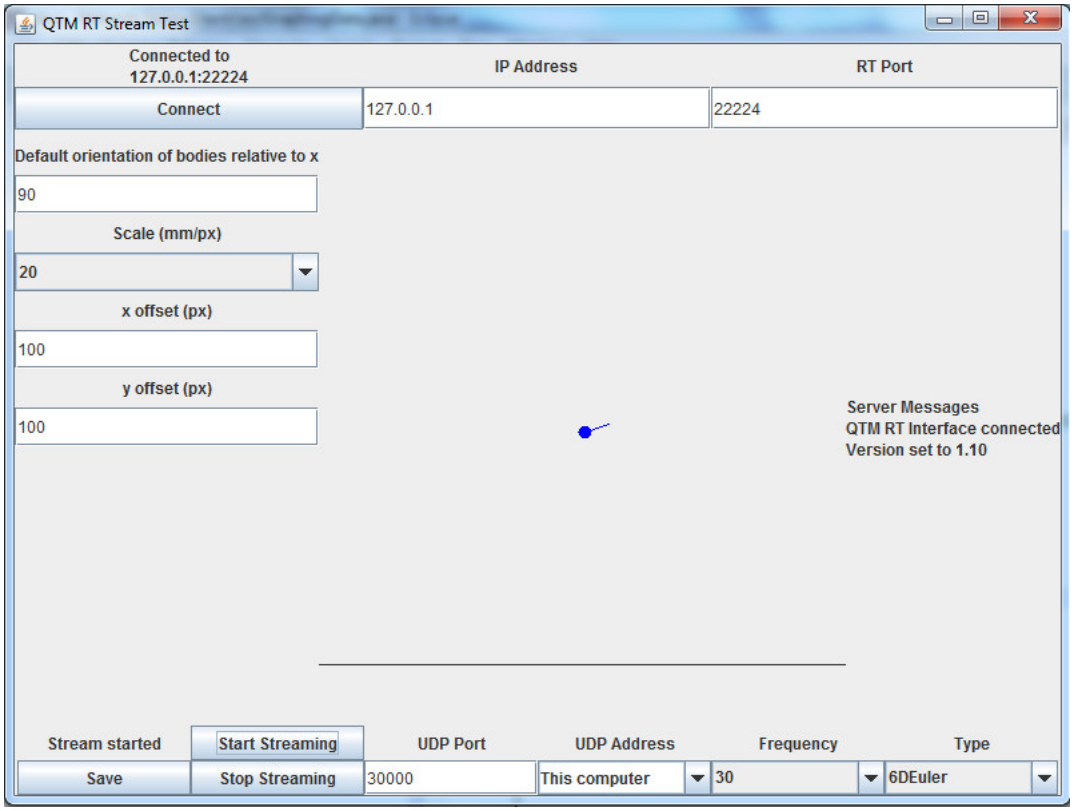

Figure 3.5.1: *Screenshot of the Java program.*

4 Results

A measurement volume of 3.8 x 4.9 m inside the lab room was achieved with all cameras standing aligned against one wall to avoid reflexes from other cameras in the shiny floor illustrated in figure 3.4.2 and figure 3.4.1. There was at least a 30 degree angle between the outermost cameras and enough overlap for the calibration to succeed, the calibration result is shown in figure 4.1.1.

4.1 Distance Measurement Results

Data was taken from the "unidentified marker data" window in QTM after placing a large 40 mm half-spherical shaped marker at each point in a grid marked out with measurement tape one point at a time. After that, the measurement data from QTM and laser measurements were compared to eachother and an ideal grid in Matlab and the differences in the xy-plane are shown in table 4.1.1 and the position coordinates are plotted in figure 4.1.2.

To see how orientation and position in the measurement volume affects the precision, the length of the

Table 4.1.1: Mean, max, median and standard deviation Euclidean distance in the xy plane in millimeters for measurements of a 5x5 imperfect grid marked on the floor. "QTM" denotes data gathered by placing a marker on the grid points, "ideal" means a perfect grid and "laser" refers to measurement of the grid points on the floor with a laser distance meter and trilateration.

Measurement	Mean	Max	Median	Std
QTM vs Ideal	20 mm	45 mm	15 mm	13 mm
Laser vs Ideal	21 mm	53 mm	17 mm	13 mm
QTM vs Laser	9 mm	25 mm	8 mm	7 mm

reference wand and a 200 mm ruler were measured according to figure 4.1.3 with the cameras behind the x-axis

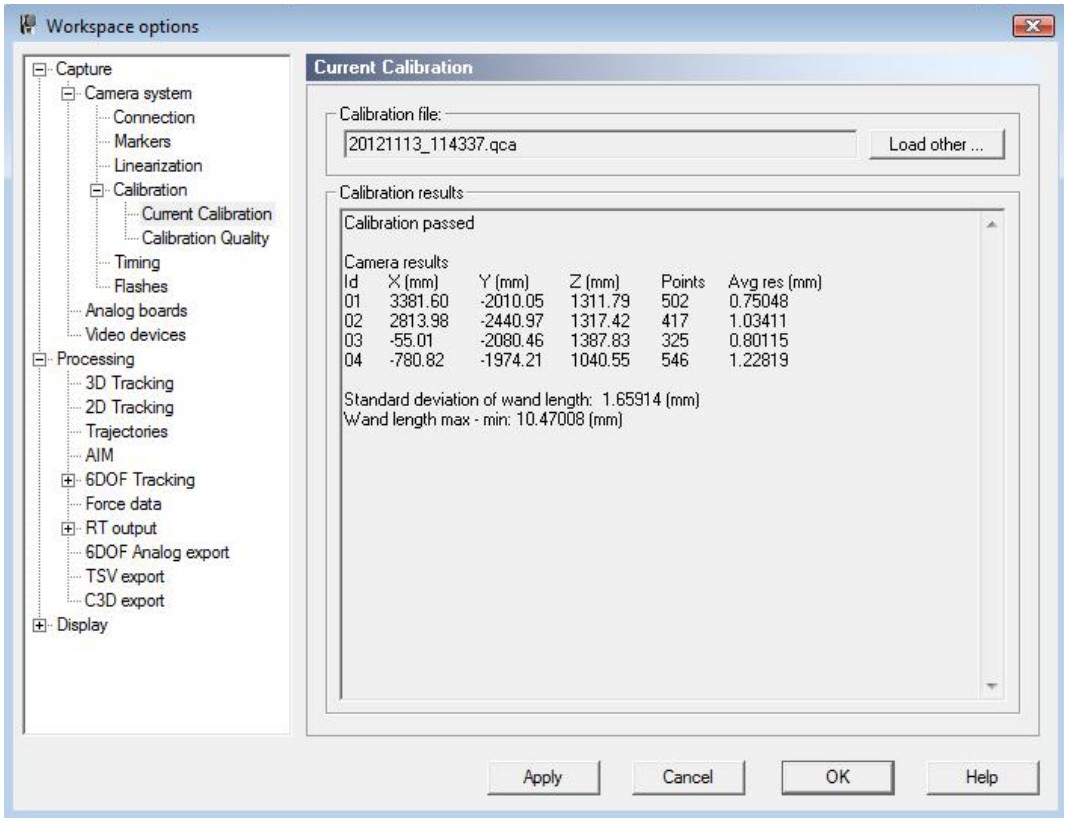

Figure 4.1.1: *A successful calibration result with 4 cameras before the measurement.*

and the results are presented in table 4.1.2. The markers of the wand became too small for the cameras to see at the maximum distance why the same measurements were done with two large half-sphere markers attached to a ruler with their centers spaced 200 mm apart (within 1 mm) far away from the cameras, the results are in table 4.1.3. The data is further illustrated in figure 4.1.5 and figure 4.1.4 with the length error plotted along the z-axis.

Table 4.1.2: Mean, maximum, median and standard deviation errors in millimeters of the length of the 750.7 mm long measurement wand measured in different angles around z taken from the x-axis.

Orientation	Mean	Max	Median	Std
All	1.0 mm	3.3 mm	0.7 mm	0.9 mm
0°	0.7 mm	1.0 mm	0.7 mm	0.3 mm
45°	0.7 mm	1.8 mm	0.5 mm	0.8 mm
90°	1.5 mm	3.3 mm	1.0 mm	1.2 mm

4.2 RC Car Test Runs

Table 4.2.1 presents the minimum marker and 6 DOF body fill levels of different marker layouts on an RC car driving around for 60 seconds on a test track pictured in figure 4.2.4. Figure 4.1.4 indicates that an error tolerance between body markers should be at least 5 mm at a 200 mm distance far away, 6-7 m, from the cameras, the tests were run with 10 and later 20 mm for the measurements with the frame. Figure 4.2.5 shows what the car with the rigid frame looks like with the optimal marker configuration on the car dubbed ”3 half spheres on frame” in table 4.2.1. Figure 4.2.2 and figure 4.2.3 show the measured trajectories with 3 and 4 half-spherical markers (the two uppermost entries in table 4.2.1) and figure 4.2.1 shows a comparison between the tracked trajectories of the two marker configurations by plotting them on top of each other. Figure 4.2.6 and 4.2.7 are more examples of marker setups (entry 3 and 4) on the car used in table 4.2.1. Figure 4.2.8 shows when two markers line up and merge into one tracked marker from the perspective of a camera.

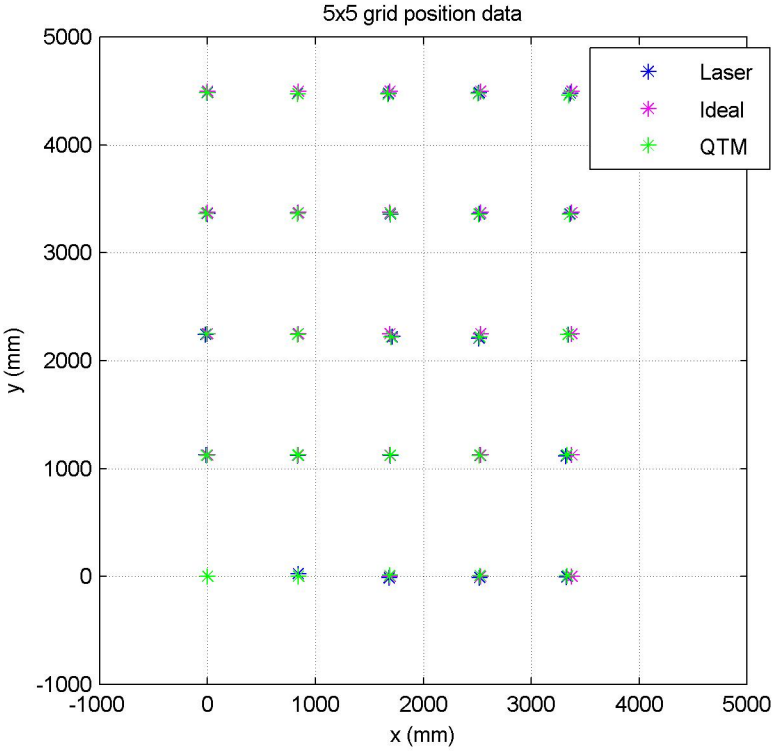

Figure 4.1.2: *Plot of the positioning data from laser measurements and QTM.*

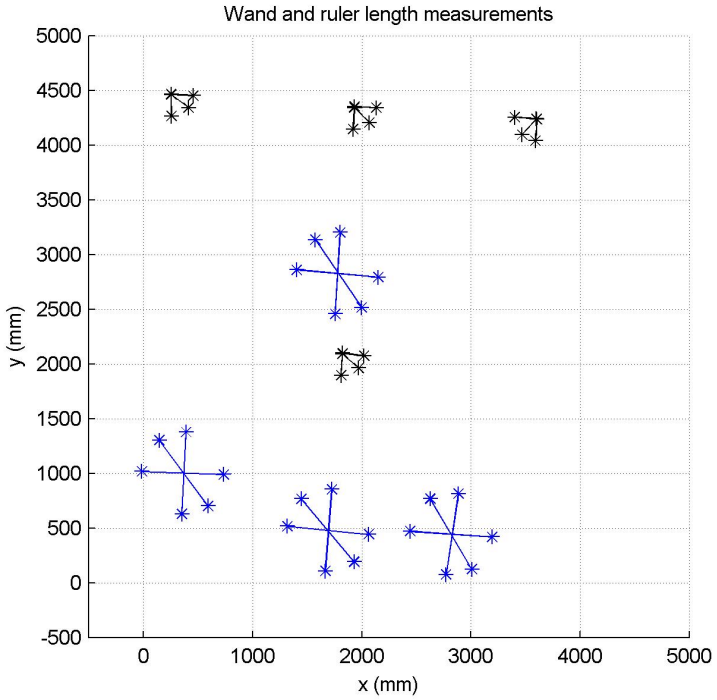

Figure 4.1.3: *Plots of Wand (blue) and ruler (black) position. The cameras are placed along the x-axis a couple of meters behind it as in figure 3.4.1.*

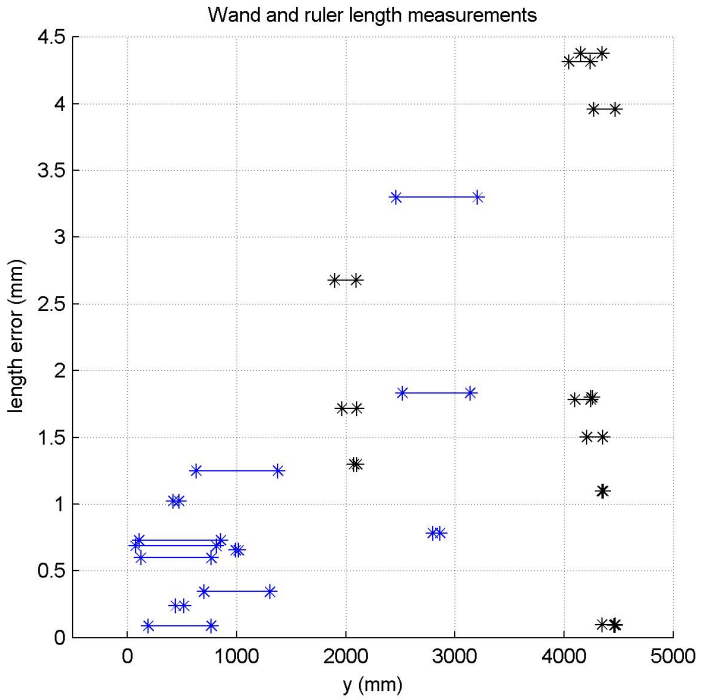

Figure 4.1.4: *Wand (blue) and ruler (black) y position plotted along x vs the length error along the y-axis pointing away from the cameras.*

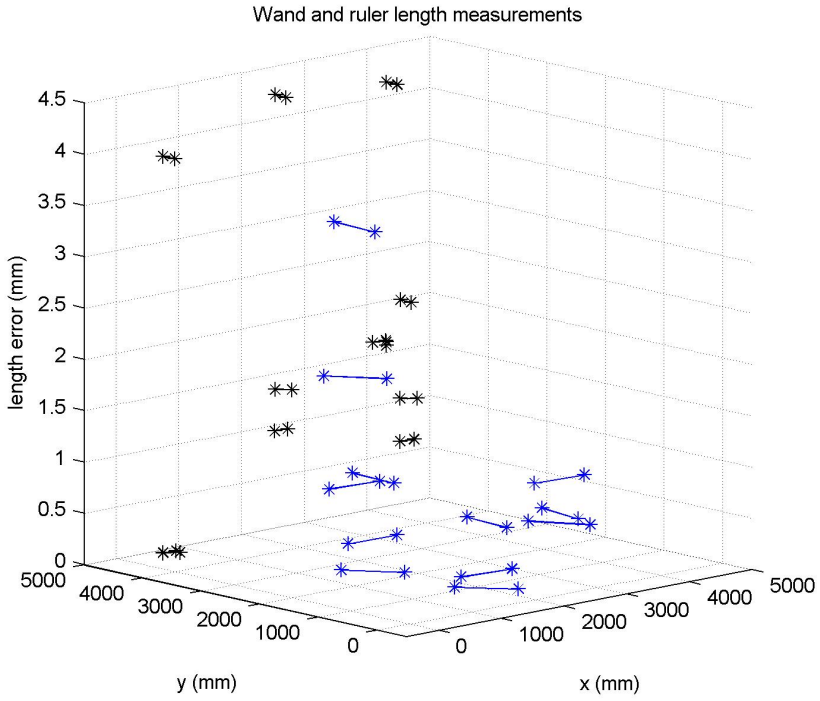

Figure 4.1.5: *Wand (blue) and ruler (black) position plotted at x and y and its length error in z. The x and y coordinates are the same as figure 4.1.3.*

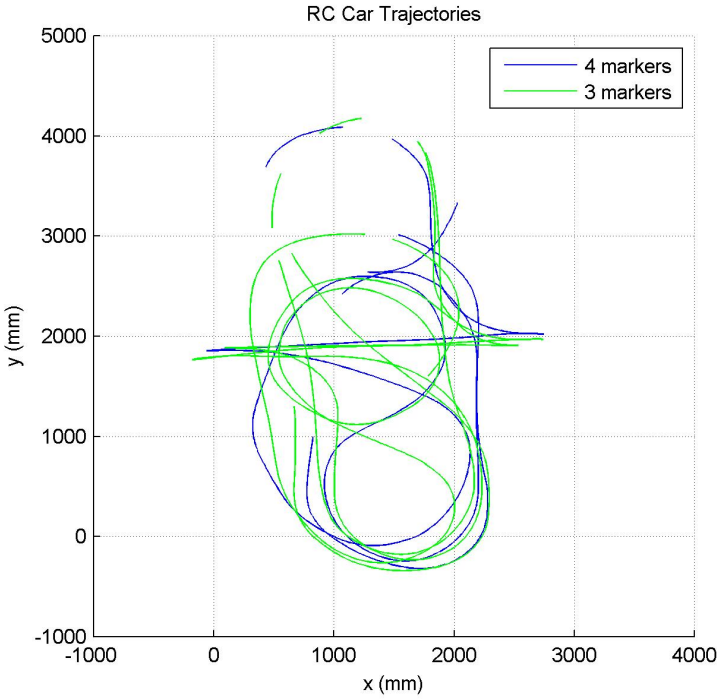

Figure 4.2.1: *Identified body trajectories of a frame with 4 markers vs 3 on the car.*

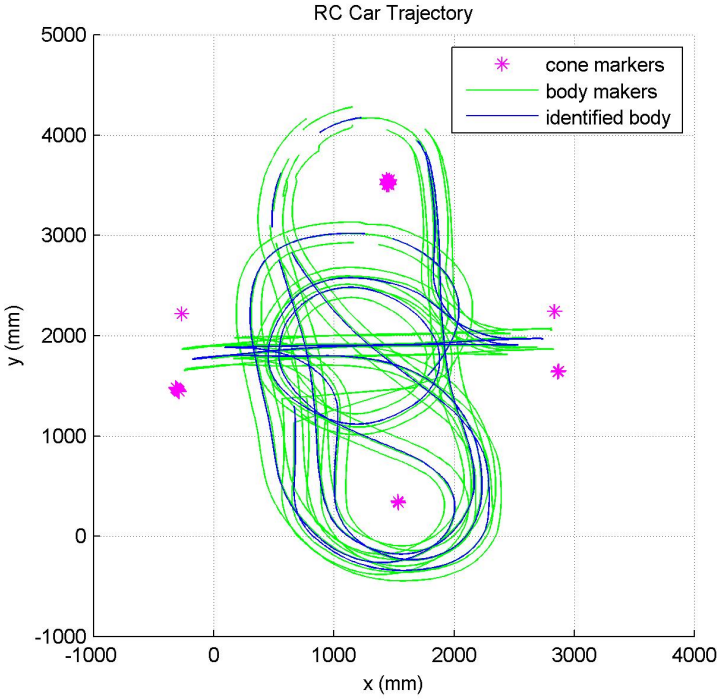

Figure 4.2.2: *Trajectories of the markers and the tracked body of the RC car with 3 half sperical markers on a frame.*

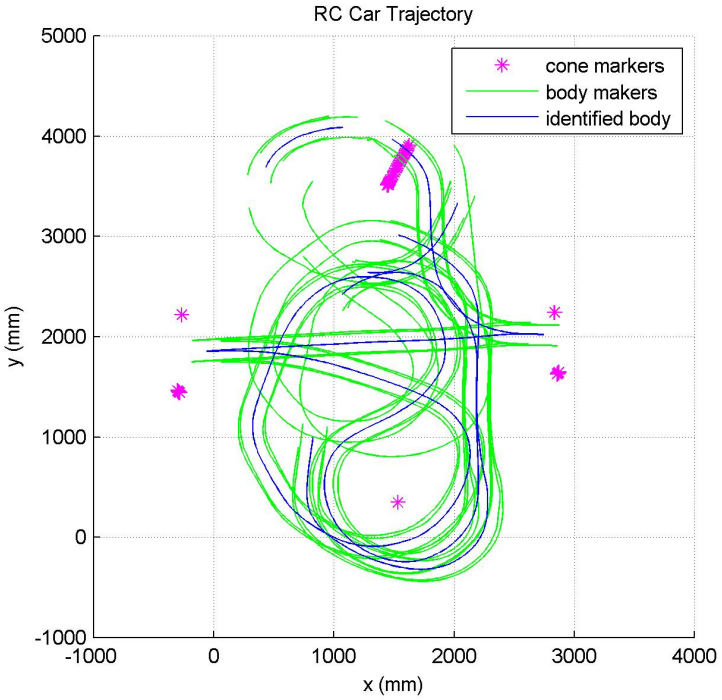

Figure 4.2.3: *Trajectories of the markers and the tracked body of the RC car with 4 half sperical markers on a frame. Notice the drift of the marker belonging to the uppermost cone as the markers on the car crosses its path. In the recording it starts moving up at constant speed and then immediately returns to its original position when the car has finished crossing it.*

Figure 4.2.4: *Test track, approx. 3 x 3 meters big.*

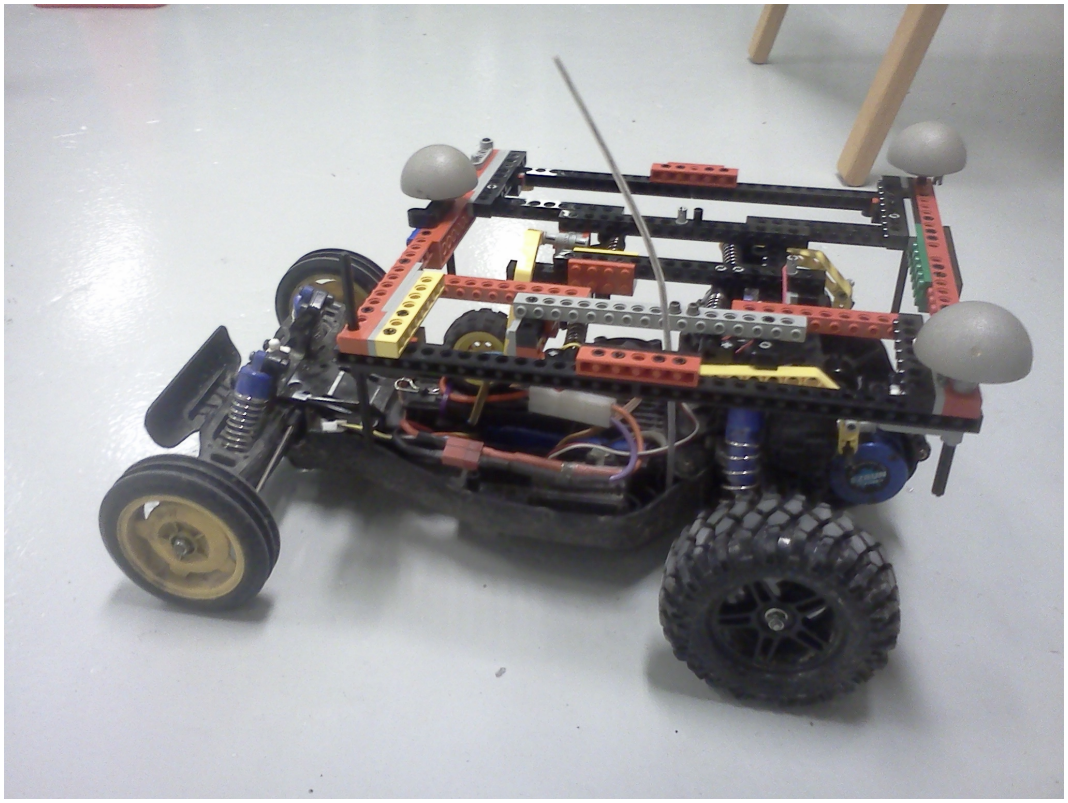

Figure 4.2.5: *3 half spherical 40 mm diameter markers attached to a rigid frame fastened on top of the RC car. When running with 4 markers the middle one in the front is replaced with two markers in the front corners.*

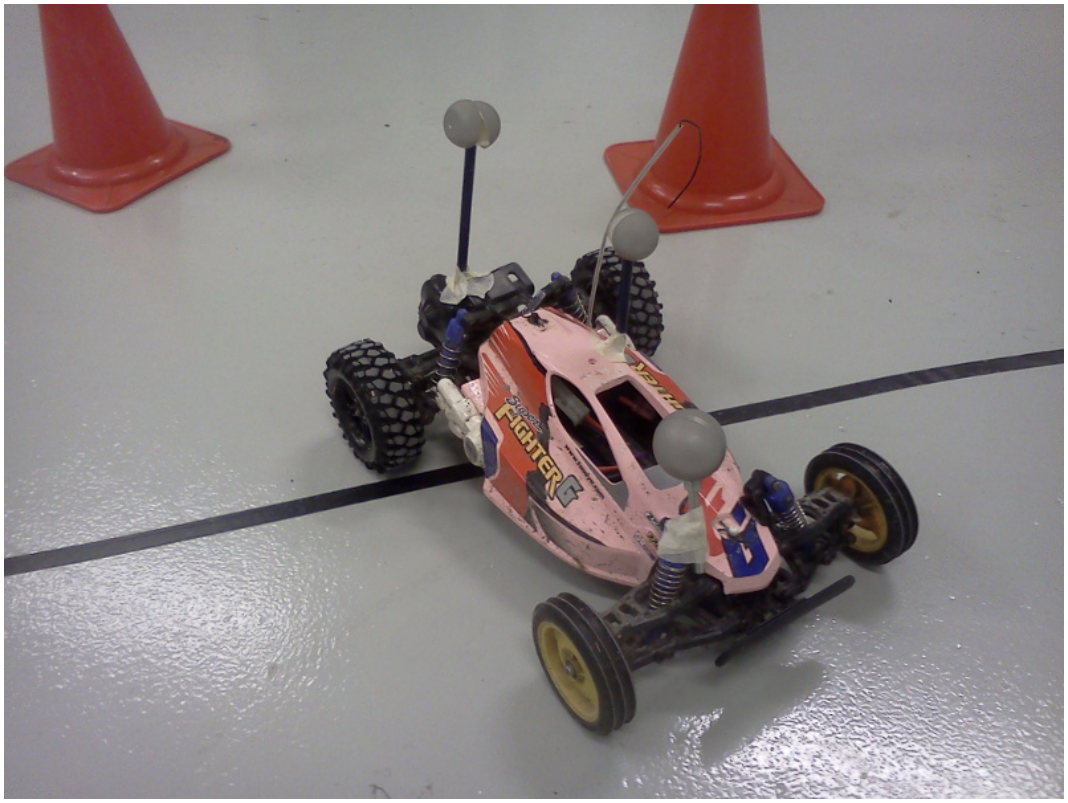

Figure 4.2.6: *3 spherical markers attached to the RC car.*

Table 4.1.3: Mean, maximum, median and standard deviation errors in millimeters of the Euclidean distance between two large (40mm in diameter) half spherical markers attached to a ruler 200 mm apart for different orientations around the z axis where the x-axis is taken as zero.

Orientation	Mean	Max	Median	Std
All	2.1 mm	4.4 mm	1.7 mm	1.5 mm
0°	1.1 mm	1.8 mm	1.2 mm	0.7 mm
45°	1.3 mm	1.8 mm	1.6 mm	0.8 mm
90°	3.8 mm	4.4 mm	4.1 mm	0.8 mm

Table 4.2.1: Trajectory fill levels for 60 second test drives on the track in figure 4.2.4 with different marker setups on the car. Markers are half spheres attached to the body of the car unless specified otherwise. The big markers have a diameter of 40 mm and the small ones 30 mm.

Marker Setup	# Markers	6 DOF Body Fill Level	Minimum Identified Marker Fill Level
3 half spheres on frame	3	93.5%	96.2%
4 half spheres on frame	4	70.4%	70.7%
3 spheres	3	78.5%	82.8%
3 spheres + 2 on side	5	67.2%	74.0%
3 big markers (half spheres)	3	68.1%	77.4%
4 big markers, 2 on side	4	62.6%	66.4%
6 small and big	6	42.9%	44.9 %

Figure 4.2.7: *The same 3 spherical markers at the same position as in figure 4.2.6 but with two additional half spheres, one on each side of the RC car body.*

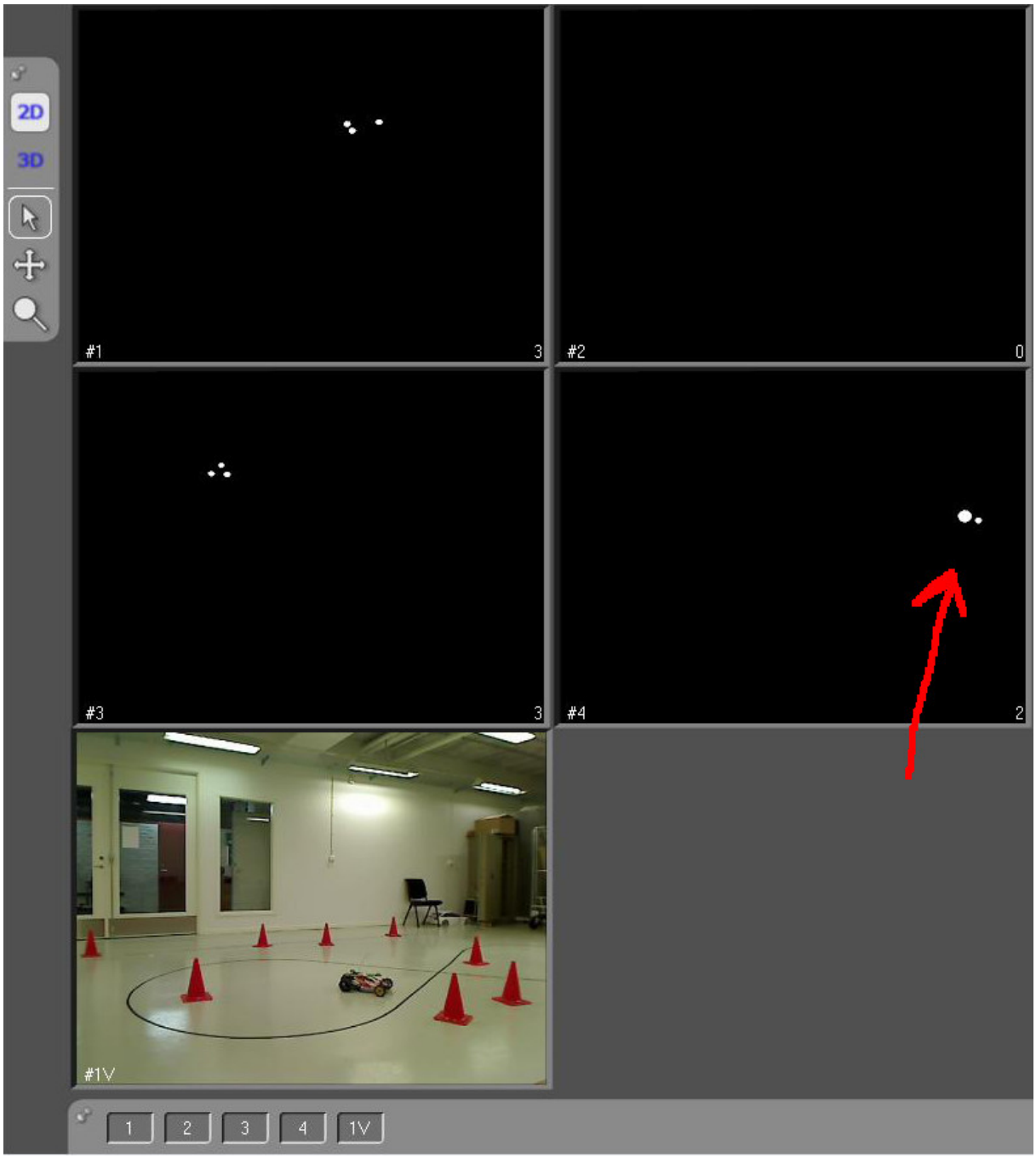

Figure 4.2.8: *The red arrow indicates where camera 4 identifies two markers as one because they are too close from its perspective whereas camera 1 and 3 correctly interpret them as separate markers.*

5 Discussion and Conclusions

5.1 Survey of Positioning Systems

Motion capture cameras are well suited for precise positioning as they have superior update rate, resolution, are capable of tracking fast moving objects and are not affected by signal reflections from (matte) walls and other objects unlike many of the other systems covered in this report. They also report heading and scale well with many tracked objects and are used in other reports such as [21] and [39] to evaluate the precision of other positioning systems. Their main drawback is that they cover relatively small volumes and are very expensive. The Hagisonic Stargazer seems like a good alternative for indoor use that is relatively inexpensive.

It is difficult to know how well a positioning system is going to work before actually testing it because the implementation such as camera placement of a global vision system and noise in the environment affect different systems differently. Scalability is also an important factor as systems like ultrasound and UWB systems lose refresh rate when tracking many objects close to each other. Speed affects odometry sensors because of the increased tire slippage and possibly even ultrasound systems in active configuration judging by the increasing error with speed in [34, p. 2], perhaps due to the Doppler effect. When contacting Hexamite about their ultrasound positioning system Hx19 they replied that they have not performed any measurements on moving objects.

It seems to be common practise to combine an absolute positioning system with dead reckoning from e.g. an IMU to improve the update rate and compensate for signal loss and then filter and fuse the data with an extended Kalman filter. This way one can compensate for some of the weaknesses of the absolute positioning system.

5.2 Experiments

QTM with its 6 DOF tracking serves as a very precise positioning system down to millimeter precision with the setup and limited measurement volume used in this report.

The standrad wand length error in table 4.1.2 is a bit lower than the residuals of the axes reported by the calibration in figure 4.1.1 which probably has to do with that during calibration the wand is moved over the entire measurement volume and sometimes the markers are partially obstructed when it is rotated. The wand measurements were done in the closer part of the volume.

Table 4.1.1 says that the laser trilateration measurements were closer to the values reported by QTM than the ideal grid that was aimed for when marking out the grid with measuring tape. The laser distance meter has a precision of one tenth of a millimeter but the error per measurement is probably a few millimeters for it not being perfectly level with the floor due to its unevenness. When placing the marker on the floor for each position reading with QTM there is also an error of a few millimeters. All in all QTM does a good job of calculating the 3D positions consistently inside the measurement volume as long as a marker is visible by at least two cameras even outside of the area that the software reports as "calibrated volume" in the 3D view.

The shiny and uneven lab floor limited the possible camera configurations since the IR light emitted from the cameras to illuminate the markers were reflected into cameras standing in front of them and were falsely identified as giant markers. The high resolution of the cameras seems to have overcome the less than optimal camera placement and still reach a very high precision.

The difference in the length error between the ruler and wand length measurements in table 4.1.2 and table 4.1.3 has to do with the fact that the ruler is not defined as exactly as the wand, one millimeter compared to one tenth of a millimeter and that the measurements on the ruler are done further away from the cameras where the precision is worse due to the finite resolution of the cameras, judging by the errors increasing with distance from the cameras in figure 4.1.4. The orientation of the ruler and wand is wrong by a few degrees during each measurement, in hindsight it would have been faster and more precise to perform this test by

defining a 6 DOF body of 4 markers and keep the yaw angle the same at every measurement point and then record the 3D coordinates of each marker. All in all the data still yielded the expected results.

Good body tracking is a matter of the markers being visible and discernible from each other by the cameras at all times inside of the measurement volume. An expensive way to improve the body tracking is to add more cameras which will reduce the number of dead angles. There were only 4 cameras set up to cover as large a volume as possible inside of the lab room with just enough overlap for calibration to succeed meaning that there was no redundancy and the markers were usually only seen by two cameras at once.

A solution that is almost guaranteed to help is to raise the cameras further up. The cameras were already at shoulder's height (approx 1.5 m above ground), close to the height limit of the tripods. It would be better to mount the cameras higher up along the walls with permanent wall mounts.

The algorithm for body tracking is rather strict, increasing the "bone length tolerance" from 5 mm to 10 mm to 20 mm did not seem to improve the fill levels, therefore the rigidity of the bodies during the tests was not a problem, although it should help far away from the cameras judging by the difference in ruler length in certain angles. Adding more markers to a body only seemed to worsen the tracking because they would be tracked as one by the cameras from being too close to each other, evidenced by the decreased fill levels in table 4.2.1 for the markers previously there for the marker setups with added markers. The threshold for the number of markers required to identify a body consisting of many markers seems a bit high given that the fill level of the body never exceeds that of the marker with the lowest fill level.

Lessons learned about optimal marker placement is that every body marker must always be visible and not be too close to another marker in the field of view of any camera inside the entire measurement volume. For the camera placement in this report a frame perfectly level with the floor with 3 half spherical markers as far away from eachother as possible was the optimal configuration. If the cameras are mounted closer to the floor spherical markers on rods with different height would be better since they would be separated by their different heights above the floor in all angles. Separating markers heightwise when the cameras are tilted only make them line up in a specific angle.

It might also help to just stream labeled markers and use a more forgiving algorithm for calculating the position and heading of bodies since the one in QTM is rather strict which is evidenced by the discrepancy between the body fill level and trajectory fill levels, or apply a filter to fill in the gaps, seen in figure 4.2.2. This way one could also add some sort of estimation of the heading for when only one marker is visible.

The Java program seems to work rather well. The streaming of marker and 6 DOF body data works, it is possible to save the streamed data and the markers or bodies being streamed are drawn correctly. The RT protocol is well documented and easy to use over a TCP/IP connection.

5.3 Further Work

According to the data sheets the maximum range of the Qualisys cameras is 70 meters, one could try and see how large measurement volume it is possible to achieve with long-range active markers and a limited number of cameras.

It would be interesting to create a global vision positioning system using object detection algorithms with normal web cameras or IR cameras with markers. Alternatively one could implement the MIT Cricket System in active configuration. For an outdoor system covering a large area one could build an RTK GPS base station using rtklib and couple it with IMU odometry.

References

[1] Mike Addlesee et al. "Implementing a sentient computing system". In: *Computer* 34.8 (2001), pp. 50–56.

[2] Alexander Altby et al. *Robust och noggrant positioneringssystem baserat på avståndsmätningar mellan mobila noder*. Tech. rep. Chalmers Tekniska Högskola, 2012.

[3] M Andersen et al. *Kinect depth sensor evaluation for computer vision applications*. Tech. rep. Department of Engineering, Aarhus University (Denmark), 2012.

[4] Hari Balakrishnan et al. *Lessons from developing and deploying the cricket indoor location system*. Tech. rep. MIT Computer Science and Artificial Intelligence Laboratory, 2003.

[5] Joydeep Biswas and Manuela Veloso. "Depth camera based localization and navigation for indoor mobile robots". In: *RGB-D Workshop at RSS*. 2011.

[6] Johann Borenstein, HR Everett, and Liqiang Feng. *Where am I? Sensors and methods for mobile robot positioning*. Tech. rep. The University of Michigan, 1996.

[7] Time Domain. *PulsON 410 Data Sheet*. 2012.

[8] *Enterprise-Wide Location Tracking Explained: How Does Ekahau RTLS Work?* 2012. URL: `http://www.ekahau.com/products/real-time-location-system/overview/how-ekahau-rtls-works.html`.

[9] Yanrui Geng et al. "Developing a low-cost MEMS IMU/DGPS integrated system for robust machine automation". In: *Proceedings of ION-GNSS*. 2007, pp. 25–28.

[10] *Global Positioning System*. 2013. URL: `http://en.wikipedia.org/wiki/Global_Positioning_System`.

[11] *GPS Accuracy*. 2012. URL: `http://www.gps.gov/systems/gps/performance/accuracy/`.

[12] Robin Henniges. *Current approches of Wifi Positioning*. Tech. rep. TU-Berlin, 2012.

[13] Richard T. Vannoy II. *Accurate Autonomous Robot Laser Navigation Using Only Passive Reflectors*. Tech. rep. ITT Technical Institute, San Diego, California, 2007.

[14] *Indoor Localization System: StarGazer HSG-A-02*. 2012. URL: `http://eng.hagisonic.kr/cnt/prod/prod010102?uid=10&cateID=2&fieldName=&orderBy=`.

[15] AR Jiménez et al. "Indoor Pedestrian Navigation using an INS/EKF framework for Yaw Drift Reduction and a Foot-mounted IMU". In: *Positioning Navigation and Communication (WPNC), 2010 7th Workshop on*. IEEE. 2010, pp. 135–143.

[16] Alan Kaminsky. *Trilateration*. Tech. rep. Rochester Institute of Technology, 2007.

[17] Bernd Kitt, Andreas Geiger, and Henning Lategahn. "Visual odometry based on stereo image sequences with ransac-based outlier rejection scheme". In: *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE. 2010, pp. 486–492.

[18] *List of IMU products at Sparkfun*. 2013. URL: `https://www.sparkfun.com/categories/160`.

[19] Hexamite Ltd. *Hx19 flyer*. 2012.

[20] Johannes Martinsson and Reimund Trost. "Implementation of motion capture support in smartphones". MA thesis. Chalmers University of Technology, 2010.

[21] Alejandro Marzinotto. "Cooperative Control of Ground and Aerial Vehicles". MA thesis. KTH Royal Institute of Technology, 2012.

[22] John Dudley Roger Marely Anthony Reid Zane Smith Adrian Talyer Ross McAree Michael Edgar Jonathan Aw. *Characterization of a UWB transceiver for mining applications*. Tech. rep. CRCMining Australia, 2012.

[23] *Networking Basics*. 2013. URL: `http://docs.oracle.com/javase/tutorial/networking/overview/networking.html`.

[24] Nissanka B Priyantha, Anit Chakraborty, and Hari Balakrishnan. "The cricket location-support system". In: *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM. 2000, pp. 32–43.

[25] Qualisys. *QTM Qualisys Track Manager User Manual 2.7*. 2012.

[26] Qualisys. *QTM Real-Time Server Protocol Documentation Version 1.10*. 2012.

[27] *Qualisys ProReflex motion capture camera*. 2008. URL: `http://en.souvr.com/product/capture/Optical/`.

[28] *Qualisys ProReflex motion capture camera*. 2008. URL: `http://en.souvr.com/product/201006/6595.html`.

[29] V. Philip Rasmussen. *High-End DGPS and RTK systems*. Tech. rep. Utah State University, 2010.

[30] Maria Isabel Ribeiro. *Kalman and extended kalman filters: Concept, derivation and properties*. Tech. rep. Institute for Systems and Robotics, 2004.

[31] *Roomba Pacman*. 2009. URL: http://pacman.elstonj.com/index.cgi?dir=home&num=&perpage=&section=.
[32] *RTKLIB: An Open Source Program Package for GNSS Positioning*. 2013. URL: http://www.rtklib.com/.
[33] David Sandberg, Krister Wolff, and Mattias Wahde. "A robot localization method based on laser scan matching". In: *Advances in Robotics*. Springer, 2009, pp. 171–178.
[34] Adam Smith et al. "Tracking moving devices with the cricket location system". In: *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM. 2004, pp. 190–202.
[35] Tomoji Takasu and Akio Yasuda. "Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB". In: *International Symposium on GPS/GNSS, International Convention Center Jeju, Korea*. 2009.
[36] Benjamin Vedder. "Gulliver: Design and Implementation of a Miniature Vehicular System". MA thesis. Chalmers University of Technology, 2012.
[37] *WAAS and EGNOS*. 2009. URL: http://www.kowoma.de/en/gps/waas_egnos.htm.
[38] Bryon Winkler. "An implementation of an ultrasonic indoor tracking system supporting the OSGI architecture of the ICTA lab". PhD thesis. University of Florida, 2002.
[39] Jingang Yi et al. "IMU-based localization and slip estimation for skid-steered mobile robots". In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE. 2007, pp. 2845–2850.

A Java Program

A.1 TCPIPClient.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.util.ArrayList;

//tcp ip client object handling the rt server communication over tcp/ip
public class TCPIPClient {

  private Socket client;
  private boolean connected = false;

  PrintStream out;
  BufferedReader in;

  TCPIPThread subTCPIPThread;

  MyController myController;

  public TCPIPClient(MyController myController)
  {
    this.myController = myController;
  }

  public void connect(String address, String portString)
  {
    if(!connected)
    {
      ArrayList<String> message = new ArrayList<String>();  //for feedback to the connection
          label in the view
      try
      {
        int port = Integer.parseInt(portString);
        //InetAddress address = InetAddress.getByName("localhost");
        //MyByteOperations.useBigEndian = false;

        //ceating the socket to connect to server running on same machine binded on port no
            3000
        //client=new Socket("localhost",22224); //big endian port
        client=new Socket("localhost",port);  //big endian port
        System.out.println("Client connected ");
        connected = true;

        //getting the o/p stream of that connection
        out=new PrintStream(client.getOutputStream());

        //reading the response using input stream
        in= new BufferedReader(new InputStreamReader(client.getInputStream()));
        //start a receiving thread
        subTCPIPThread = new TCPIPThread(in, myController); // create a new thread for
            reading server replies

        out.write(sendMessage("Version 1.10")); //the udp stream command exists in version
            1.1 of the rt protocol and beyond (see protocol changes history in the pdf)
        out.flush();
        //send an event that the connection was successful
        message.add("<html>Connected to<br />" + address + ":" + portString + "</html>");
      }
      catch(Exception e)
      {
        System.out.println(e.getMessage());
```

```
      message.add("Connection to QTM server failed");
    }
    myController.propertyChange(new MyViewEvent("Server", message));
  }
}

public void disconnect()
{
  connected = false;
  try
  {
    client.close();
  }
  catch(Exception e)
  {
    System.out.println(e.getMessage());
  }
}

//send the stream frames command to the rt server and fire a stream started property change
     event to the controller
public void startStream(String frequency, String udpAddress, String udpPort, String
    dataType)
{
  if(connected){
    try
    {
      System.out.println("Streaming to " + udpPort);
      if(frequency.compareTo("AllFrames") != 0)frequency = "Frequency:" + frequency;
      if(udpAddress.compareTo("") == 0)
      {
        out.write(sendMessage("StreamFrames " + frequency +" UDP:" + udpPort + " " +
            dataType));
      }
      else
      {
        out.write(sendMessage("StreamFrames " + frequency +" UDP:" + udpAddress + ":" +
            udpPort + " " + dataType));
      }
      out.flush();
      //notify the view
      myController.propertyChange(new MyViewEvent("Stream", "Stream started"));
    }
    catch(Exception e)
    {
      System.out.println(e.getMessage());
    }
  }
  else
    System.out.println("Not connected");
}

//tell the rt server to stop streaming and fire a stream stopped property change event
public void stopStream()
{
  try
  {
    out.write(sendMessage("StreamFrames Stop"));
    out.flush();
    //notify the view
    myController.propertyChange(new MyViewEvent("Stream", "Stream stopped"));
  }
  catch(Exception e)
  {
    System.out.println(e.getMessage());
  }
}

private static byte[] sendMessage(String sendString)    //send commands to the rt server of
     type 1.
{
```

```
    //System.out.println("string being sent is " + sendString);
    sendString = sendString + "\0";
    int bufLength = sendString.length() + 8;
    byte[] buf = new byte[bufLength];

    // initialize array
    for (int i = 0; i < bufLength; i++) {

      buf[i] = 0;
    }

    //little endian -> replace this with "my byte operations"
    //buf[3] = (byte)(sendString.length() + 8); //assume that the total command length is
        less than 127 characters :P
    byte[] tempByteArray = MyByteOperations.my_int_to_bb(sendString.length() + 8,
        MyByteOperations.useBigEndian);
    for (int i = 0; i < 4; i++)
    {
      buf[i] = tempByteArray[i];
    }
    //buf[7] = 1;    //message type = 1 i.e. a command
    tempByteArray = MyByteOperations.my_int_to_bb(1, MyByteOperations.useBigEndian);
    for (int i = 4; i < 8; i++)
    {
      buf[i] = tempByteArray[i - 4];
    }

    byte[] message = new byte[sendString.length()];
    message = sendString.getBytes();

    for (int i = 8; i < bufLength; i++) {

      buf[i] = message[i - 8];
      System.out.print(message[i - 8]);
    }

    //System.out.println("the message being sent is");

    for (int i = 0; i < bufLength; i++) {
      System.out.print(buf[i] + " ");
    }

    return buf;
  }

}
```

A.2 UDPThread.java

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import java.io.*;

//thread that receives the udp stream from the rt server, updates the graph in the view with
    coordinates and sends the saved frames to the udp client once the stream has stopped.
// Create a new thread.
class UDPThread implements Runnable {
  Thread t;
  DatagramSocket rsocket;
  UDPClient udpClient;
  MyController myController;
  boolean isRunning;  //if the thread is running
  int frameTimeMs = 100;  //update interval in ms between each call to the graph
  boolean endian = MyByteOperations.useBigEndian;

  UDPThread(DatagramSocket rsocket, UDPClient udpClient, MyController myController) {
    // Create a new, second thread
    this.rsocket = rsocket;
    this.udpClient = udpClient;
    this.myController = myController;
    isRunning = true;

    t = new Thread(this, "UDP Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // entry point for the udp thread started by the udp client that receives udp packets
      containing frame data from the rt server
  // and saves them into an array that it sends to the udp client once the stream has stopped
  public void run() {

    // Create a buffer to read datagrams into. If a
    // packet is larger than this buffer, the
    // excess will simply be discarded!
    byte[] buffer = new byte[2048];

    // Create a packet to receive data into the buffer
    //DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

        //list for saving the frame save data
        ArrayList<StreamFrame> framesList = new ArrayList<StreamFrame>();

        //timer
        long lastPacketTime = System.currentTimeMillis();

        //add some error handling in case of packet loss?
    try {
      //set timeout
      rsocket.setSoTimeout(1000);

      int i5 = 0;
      //add some way to stop this too... send stop command to qtm from the other thread
      while (i5++ < Integer.MAX_VALUE && !udpClient.stopThread) {
      //while(true){
```

```
//udp network stuff
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

// Wait to receive a datagram
rsocket.receive(packet);  //throws an error if it exceeds the timeout

// Convert the contents to a string, and display them
//String msg = new String(buffer, 0, packet.getLength());
//System.out.println(packet.getAddress().getHostName() + ": " + msg);
//System.out.print(packet.getAddress().getHostName() + ": ");

// read the bytes from the packet and put it into a byte array
byte[] buf2 = packet.getData();
//String s = "";
/*
for (int i2 = 0; i2 < buf2.length; i2++) {
  System.out.print(buf2[i2] + " ");
  //s += (char) buf2[i2];
}*/
//System.out.println("was inside the packet!");

// Reset the length of the packet before reusing it.
packet.setLength(buffer.length);

//analyze the data, p.42 in qtm rt protocol.pdf

int packetSize = MyByteOperations.my_bb_to_int(new byte[]{buf2[0], buf2[1], buf2[2],
    buf2[3]}, endian);
//System.out.println("size of data packet is " + packetSize);

//read timestamp at index 8 to 15, anyway, sign is zero because the number isnt big
    enough.... weird time format tho, hmmmn.... i dont care that java uses unsigned
    types ok...
long timeStamp = MyByteOperations.my_bb_to_long(new byte[]{buf2[8], buf2[9], buf2
    [10], buf2[11], buf2[12], buf2[13], buf2[14], buf2[15]}, endian);
//System.out.println("timestamp is " + MyByteOperations.printUnsignedLong(timeStamp)
    + " useconds, i.e. " + 1.0 * timeStamp/1000000.0 + " seconds from start");
int frameNumber = MyByteOperations.my_bb_to_int(new byte[]{buf2[16], buf2[17], buf2
    [18], buf2[19]}, endian);
//System.out.println("frame number is " + frameNumber);
int componentCount = MyByteOperations.my_bb_to_int(new byte[]{buf2[20], buf2[21],
    buf2[22], buf2[23]}, endian);
//System.out.println("component count is " + componentCount);

//read each component
int byteOffset = 24;  //index of the first byte of the component in the package byte
    array
for(int i2 = 0; i2 < componentCount; i2++)
{
  int componentSize = MyByteOperations.my_bb_to_int(new byte[]{buf2[byteOffset], buf2
      [byteOffset + 1], buf2[byteOffset + 2], buf2[byteOffset + 3]}, endian);
  int componentType = MyByteOperations.my_bb_to_int(new byte[]{buf2[byteOffset + 4],
      buf2[byteOffset + 5], buf2[byteOffset + 6], buf2[byteOffset + 7]}, endian);
  int markerCount = MyByteOperations.my_bb_to_int(new byte[]{buf2[byteOffset + 8],
      buf2[byteOffset + 9], buf2[byteOffset + 10], buf2[byteOffset + 11]}, endian);
  //System.out.println("component size is " + componentSize + " component type is " +
       componentType + " and marker count is " + markerCount);

  //header = 16 bytes
  if(componentType == 2)  //unlabeled marker
  {
    //save into array
    framesList.add(new StreamFrame(timeStamp, frameNumber, markerCount, false));

    //offset for each marker is 16 bytes
    int markerOffset = 16;
    int totalOffset = byteOffset;
    //System.out.println("component type is 3D no labels i.e. " + componentType);
    //draw
    ArrayList<float[]> arrList= new ArrayList<float[]>();
    for(int i3 = 0; i3 < markerCount; i3++)
```

```
    {
      //System.out.print("Marker number " + i3);
      totalOffset += 16;  //the header before the components is counted the first
          time and is 16, see p. 46 and 47 in the RT protocol pdf...
      float x = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset], buf2[
          totalOffset + 1], buf2[totalOffset + 2], buf2[totalOffset + 3]}, endian);
      float y = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset + 4],
          buf2[totalOffset + 5], buf2[totalOffset + 6], buf2[totalOffset + 7]},
          endian);
      float z = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset + 8],
          buf2[totalOffset + 9], buf2[totalOffset + 10], buf2[totalOffset + 11]},
          endian);
      int markerID = MyByteOperations.my_bb_to_int(new byte[]{buf2[totalOffset + 12],
           buf2[totalOffset + 13], buf2[totalOffset + 14], buf2[totalOffset + 15]},
          endian);

      //System.out.println(" with ID " + markerID + " has coordinates x:" + x + " y:"
           + y + " z:" + z);

      //System.out.println("array size is " + framesList.size());
      framesList.get(framesList.size() - 1).setComponent(i3, markerID, x, y, z);

      //for plotting
      arrList.add(new float[]{x, y, z});
    }
    //restrict plotting to 15 fps
    if((System.currentTimeMillis() - lastPacketTime) > frameTimeMs)
    {
      //System.out.println("Time diff: " + (System.currentTimeMillis() -
          lastPacketTime));
      lastPacketTime = System.currentTimeMillis();
      myController.propertyChange(arrList);
    }

  } else if(componentType == 6) //6DOF body with Euler angles component (x,y,z
      rotation)
  {
    //save into array
    framesList.add(new StreamFrame(timeStamp, frameNumber, markerCount, true));

    //offset for each body is 12 + 3 * 4 = 24 bytes
    int markerOffset = 24;
    int totalOffset = byteOffset;
    //System.out.println("component type is 3D no labels i.e. " + componentType);
    ArrayList<float[]> arrList= new ArrayList<float[]>();
    for(int i3 = 0; i3 < markerCount; i3++)
    {
      //System.out.print("Body number " + i3);
      totalOffset += 16;  //the header before the components is counted the first
          time and is 16, see p.50
      float x = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset], buf2[
          totalOffset + 1], buf2[totalOffset + 2], buf2[totalOffset + 3]}, endian);
      float y = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset + 4],
          buf2[totalOffset + 5], buf2[totalOffset + 6], buf2[totalOffset + 7]},
          endian);
      float z = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset + 8],
          buf2[totalOffset + 9], buf2[totalOffset + 10], buf2[totalOffset + 11]},
          endian);

      //System.out.println(" has coordinates x:" + x + " y:" + y + " z:" + z + " and
          angles... ");
      //load angles
      float[] angles = new float[3];
      for(int i4 = 0; i4 < 3; i4++)
      {
        angles[i4] = MyByteOperations.my_bb_to_float(new byte[]{buf2[totalOffset + 12
            + i4 * 4], buf2[totalOffset + 13 + i4 * 4], buf2[totalOffset + 14 + i4 *
            4], buf2[totalOffset + 15 + i4 * 4]}, endian);
        //System.out.print(angles[i4] + " ");
      }
```

```
                //System.out.println("array size is " + framesList.size());
                framesList.get(framesList.size() - 1).setComponent6DOF(i3, x, y, z, angles);

                //for plotting
                if(!Float.isNaN(x))arrList.add(new float[]{x, y, z, angles[2]}); //plot
                    nothing if the coordinates are NaN which is the case in frames where the
                    body isn't identified
              }

              //System.out.println("Time diff: " + (System.currentTimeMillis() - lastPacketTime
                  ));
              if((System.currentTimeMillis() - lastPacketTime) > frameTimeMs);
              {
                lastPacketTime = System.currentTimeMillis();
                myController.propertyChange(arrList);
              }

            }
            byteOffset += componentSize;
          }
        }
        System.out.println("Received stop thread command.");
        rsocket.close();
        System.out.println("rsocket closed");
      } catch (Exception e) {
        System.out.println("something went wrong! " + e.getMessage());
        rsocket.close();
        System.out.println("rsocket closed");
      }
      //stream interrupted, send the data to the udp client

      Iterator<StreamFrame> itr = framesList.iterator();
      System.out.println("the saved data has " + framesList.size() + " frames");

      udpClient.setFramesList(framesList); //send the data to the udp client after the stream
          has stopped
      isRunning = false;
      System.out.println("Exiting UDP thread.");
    }
}
```

A.3 UML Diagram

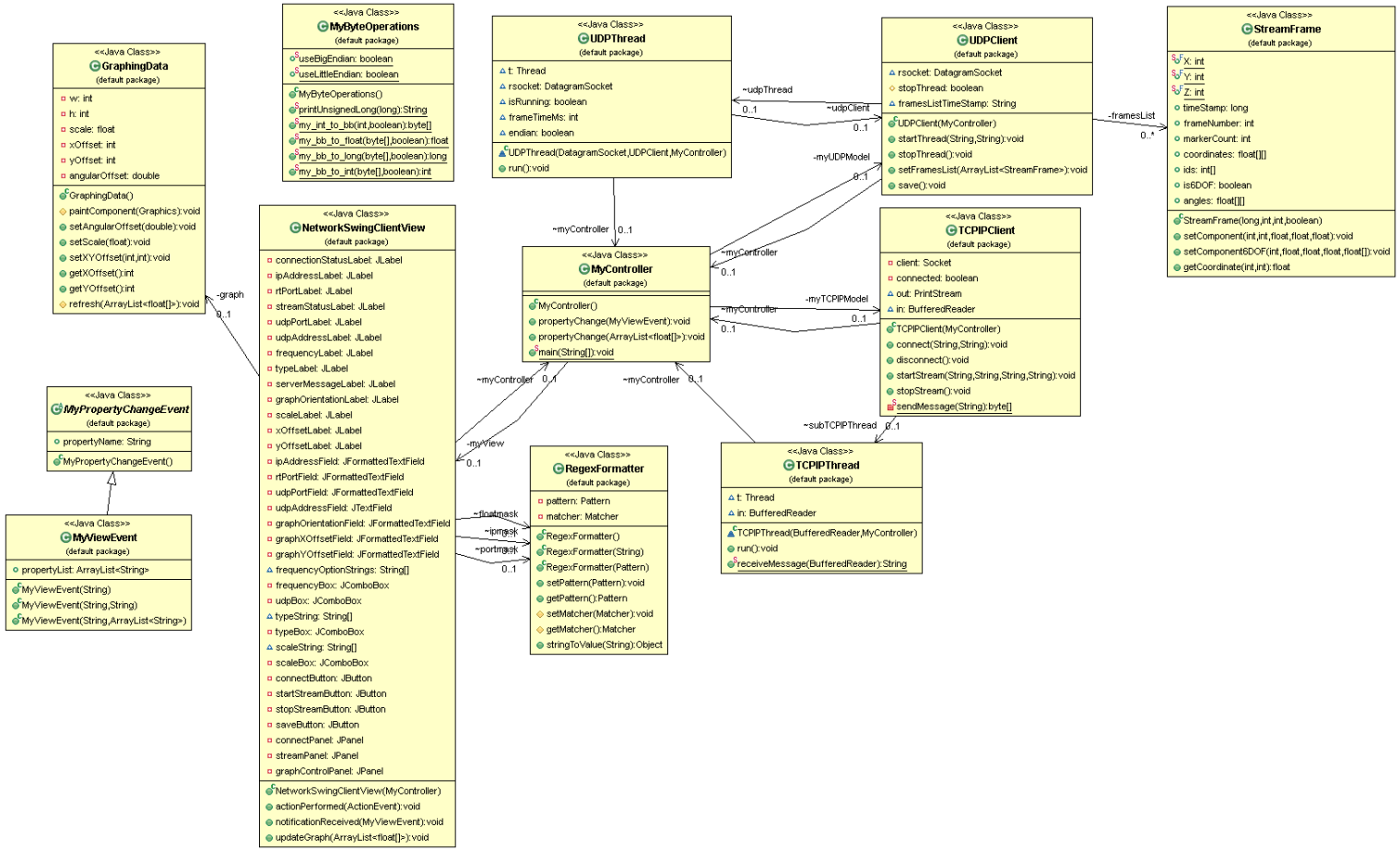

Figure A.3.1: *UML diagram of the Java program.*