Taavi Oksanen

MOBIILIROBOTIN NAVIGOINTIMENETELMÄ

MOBIILIROBOTIN NAVIGOINTIMENETELMÄ

Taavi Oksanen
Opinnäytetyö
Kevät 2017
Automaatiotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Automaatiotekniikan koulutusohjelma

Tekijä: Taavi Oksanen
Opinnäytetyön nimi: Mobiilirobotin navigointimenetelmä
Työn ohjaajat: Tapio Heikkilä (VTT), Tero Hietanen (OAMK)
Työn valmistumislukukausi ja -vuosi: Kevät 2017
Sivumäärä: 40 + 1 liite

Tässä työssä kehitettiin navigointimenetelmä Probot Oy:n valmistamalle differentiaaliajoiselle mobiilirobottialustalle. Alusta oli tarkoitus saada kulkemaan kartioiden ja seinien määrittämää rataa pitkin, käyttäen apuna VTT:llä kehitettyä 3D-konenäkösovellusta. Työ kokonaisuudessaan toteutettiin VTT:n tiloissa ja se on osa alustalle tehtävää kehitystyötä.

Opinnäytetyö jakautuu pääasiassa teoriaan, toteutukseen ja tuloksien esittelyyn. Teoriaosiossa käydään läpi työssä käytettyjä kehitystyökaluja ja -menetelmiä. Tämän lisäksi tehdään katsaus nykyaikaisiin mobiilirobotteihin ja uuteen tutkimustyöhön aiheesta sekä käsitellään erilaisia kartattomia navigointimenetelmiä. Toteutusosiossa käydään läpi työssä tehtyyn navigointimenetelmään liittyviä suunnitelmia ja sen osatoimintojen kirjoittamista sekä testausta. Siinä erikseen käsiteltäviä alaosioita ovat konenäön liittäminen robottiin, ohjausrakenteen toteutus, navigointimenetelmän ohjelmointi ja testaaminen.

Lopputuloksena onnistuttiin toteuttamaan robotille työn alussa laaditun määritelmän mukainen radanseuranta. Robotti laskee koordinaattipisteet sen havaitsemien maamerkkien ja tasopintojen perusteella. Ohjaaminen pisteitä kohti tapahtuu suhdesäätöä käyttäen. Testien perusteella robotti onnistui suorittamaan hyvällä menestyksellä radan muodon mukaisia liikeratoja. Ongelmia aiheutui lähinnä, jos kartiot aseteltiin liian kauaksi toisistaan.

Asiasanat: robotiikka, navigointi, konenäkö

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Automation Engineering

Author: Taavi Oksanen
Title of thesis: A Mobile Robot Navigation Method
Supervisors: Tapio Heikkilä (VTT), Tero Hietanen (OAMK)
Term and year when the thesis was submitted: Spring 2017
Pages: 40 + 1 appendix

The main goal of this bachelor's thesis was to design and implement a navigation method for a mobile robot platform developed by Probot Oy. Requirement was that the robot should follow a path marked by cones and walls by using a 3D computer vision system. The project was done for VTT and it was a continuation on the previous work done on the platform.

The written part follows a typical structure in engineering: introduction, theory, development, results and conclusions. The theory section deals with software and design principles that this work is based on. It also goes through a few modern examples of mobile robots on the market today, and takes a look at ongoing research in the field. Special attention is paid to reactive navigation techniques developed over the past few decades. The development section is basically divided in three: connecting the vision system to the robot, designing control architecture and finally the navigation method itself. After that, test results and problems are discussed.

In conclusion, the chosen navigation method proved out to be able to complete the specified path following task. Robot was able to drive itself along marked paths, by calculating waypoints from landmarks and walls, and by changing its heading using a simple proportional controller. As long as cones were placed in close distance from each other, the method was able to drive the robot along marked paths with a promising success rate.

Keywords: robotics, navigation, visual servoing

ALKULAUSE

Haluan kiittää VTT:tä ja Oulun ammattikorkeakoulun tutkintovastaava Tero Hietasta kiinnostavasta opinnäytetyöaiheesta. Työtehtävät ja -ympäristö tarjosivat hyvän mahdollisuuden tutustua robotiikkaan ja konejärjestelmien parissa tehtävään kehitystyöhön.

Erityiset kiitokset VTT:n puolella työtä ohjanneelle johtava tutkija Tapio Heikkilälle, sekä konenäöstä vastanneelle tutkija Jari Aholalle. Heidän asiantuntijuus oli korvaamaton apu tätä työtä tehdessä. Kiitos myös tekstin oikolukemisessa avustaneelle Oulun ammattikorkeakoulun lehtori Tuula Hopeavuorelle. Lisäksi haluan kiittää myös muita VTT:llä tutuksi tulleita henkilöitä lämpimästä työilmapiiristä ja jaetuista tiedonjyvistä.

Oulussa 17.5.2017

Taavi Oksanen

SISÄLLYS

SANASTO

CAN	Ajoneuvotekniikassa käytetty väylä tiedonsiirtoon.
C++	Olio-ohjelmointia hyödyntävä ohjelmointikieli.
Debuggaus	Virheenkorjaus.
Karteesinen koordinaatisto	Koordinaatisto, jossa on sen ulottuvuuksien mukainen määrä toisiinsa kohtisuorassa olevia akseleita.
Luokka	Määritelmä siitä, mitä yhteisiä tietorakenteita ja toimintoja kyseiseen luokkaan pohjautuvat oliot sisältävät.
Metodi	Olion yksittäinen toiminto.
Odometria	Robotin sijainnin arviointiin liittyvä termi. Pyrkii selvittämään, minne robotti on liikkunut käyttäen apuna esimerkiksi inertiayksikköä ja renkaissa olevia kulma-antureita.
Olio	Olio-ohjelmointiin liittyvä käsite yksiköstä, joka voi sisältää sekä säilytettävää tietoa että toimintoja.
Pistoke	Ohjelmointirajapinta, jonka avulla voidaan siirtää tietoa kahden eri prosessin välillä. Esimerkiksi TCP- ja UDP-pistokkeet.
Python	Olio-ohjelmointia hyödyntävä ohjelmointikieli. Tarjoaa C++:aan verrattuna mm. helpommin luettavan syntaksin ja mahdollistaa koodin ajamisen ilman kääntämistä.
Sovellus/ohjelma	Tässä työssä käytetään termejä sovellus ja ohjelma kuvaamaan samaa asiaa. Kokonaisuus toimintoja ja tietorakenteina, jotka yhdessä toteuttavat jonkin suuremman toiminnallisuuden.
Toiminto	Koodin muodossa oleva ohjeistus, jonka mukaan tietokoneen tulee toimia.

1 JOHDANTO

Tämän työn tarkoituksena oli toteuttaa Probot Oy:ltä hankittuun mobiilirobottialustaan navigointimenetelmä, jossa konenäön avulla robotti seuraa maamerkkien ja seinien määrittämää rataa. Opinnäytetyö tehtiin Oulun VTT:lle. Työn lähtökohtana oli VTT:n kehittämä konenäkösovellus ja mobiilirobottialustalle aiempana opinnäytetyönä toteutettu koordinaattiohjaus, jonka oli tehnyt Jussi Pulkkinen.

Aiempi koordinaattiohjaus perustui robotin kulkuaikoihin ajettaessa sitä vakionopeudella. Tämä oli ollut ainoa tapa toteuttaa koordinaattiohjaus sen hetkisen anturoinnin puutteen vuoksi. Tässä työssä robottiin lisättiin antureiksi kaksi Kinect-kameraa ja niitä hyödyntävä 3D-konenäkösovellus. Sovellukselta saadaan tietoa mm. sille opetettujen maamerkkien sijainnista ja havaittujen tasopintojen keskipisteistä sekä pintanormaaleista. Itse koordinaattiohjausta ei tässä työssä hyödynnetty. Koordinaattiohjaus oli kuitenkin toteutettu Qt:llä ja tarjosi pohjan, jonka päälle rakentaa tässä työssä toteutettu navigointiohjelma.

Navigointiohjelma toteutettiin pääasiassa Qt Creatorilla, kun taas konenäköohjelmisto oli tehty toimimaan ROS:n (Robot Operating System) päällä. Työhön kuului perehtymistä sekä Qt:hen ja ROS:ään näiden välisen tiedonsiirron toteuttamiseksi. Tämän lisäksi työ sisälsi robotin ohjausrakenteen ja navigointimenetelmän ohjelmoinnin sekä niiden testaamisen ja dokumentoinnin.

2 TYÖSSÄ KÄYTETYT KEHITYSTYÖKALUT JA -MENETELMÄT

2.1 Qt-kehitysympäristö

Qt on ohjelmistojen ja graafisten käyttöliittymien luomiseen tarkoitettu kehitysympäristö. Nykyään sen omistaa ja sitä kehittää The Qt Company. Alun perin Qt:n kehitti norjalainen Trolltech, jolta omistajuus siirtyi Nokialle vuonna 2008. Nokian omistuksesta se siirtyi Digialle vuosien 2011 ja 2012 aikana. (1.) Vuoden 2016 toukokuussa The Qt Company listautui pörssiin ja erkani omaksi yhtiöksi Digian alaisuudesta (2).

Qt Creator on ohjelmointiympäristö, joka sisältää koodieditorin, graafisen käyttöliittymän suunnittelutyökalut ja tuen useille eri käyttöjärjestelmille, kääntäjille, debuggereille ja versionhallintaohjelmistoille. Ympäristön natiivina ohjelmointikielenä on C++, mutta se tukee myös muiden ohjelmointikielten käyttöä sitomalla ne niille räätälöityjen kirjastojen avulla. (3.)

Qt Essentials on useista eri moduuleista koostuva valikoima C++-pohjaisia luokkakirjastoja. Se tarjoaa valmiita luokkia mm. graafisille ominaisuuksille, median hallintaan, tiedonsiirtoon ja kehitettävän ohjelman yksikkötestaukseen.

Tämän työn kannalta oleellisessa asemassa oli Qt:n tarjoamat ominaisuudet reaaliaikatoimintojen toteutukseen. MOC (The Meta-Object System) on Qt:n sisältämä toiminnallisuus, jonka pääasiallinen tarkoitus on toteuttaa olioiden välinen kommunikointi signaalit ja slotit -mekanismin avulla. (4.) Slotit voidaan ajatella perinteisinä olio-ohjelmoinnin metodeina. Ohjelmassa käytettävät oliot määritellään kuuluvaksi MOC:n alaisuuteen ja niiden välille luodaan kytkös, jossa yhden olion signaali yhdistetään toisen olion slottiin. Signaaleista on myös mahdollista tehdä ketjuja, joissa olioiden väliset signaalit yhdistetään suoraan toisiinsa.

Signaaleihin ja slotteihin perustuvat yhteydet voidaan määritellä joko direct- tai queued-tyyppisiksi. Oletusasetuksena säikeiden väliset yhteydet ovat queued-tyypin yhteyksiä. Kummankin olion sijaitessa samassa säikeessä yhteydet ovat direct-tyyppisiä. Erona näiden kahden välillä on, että direct-yhteyksissä olion lä-

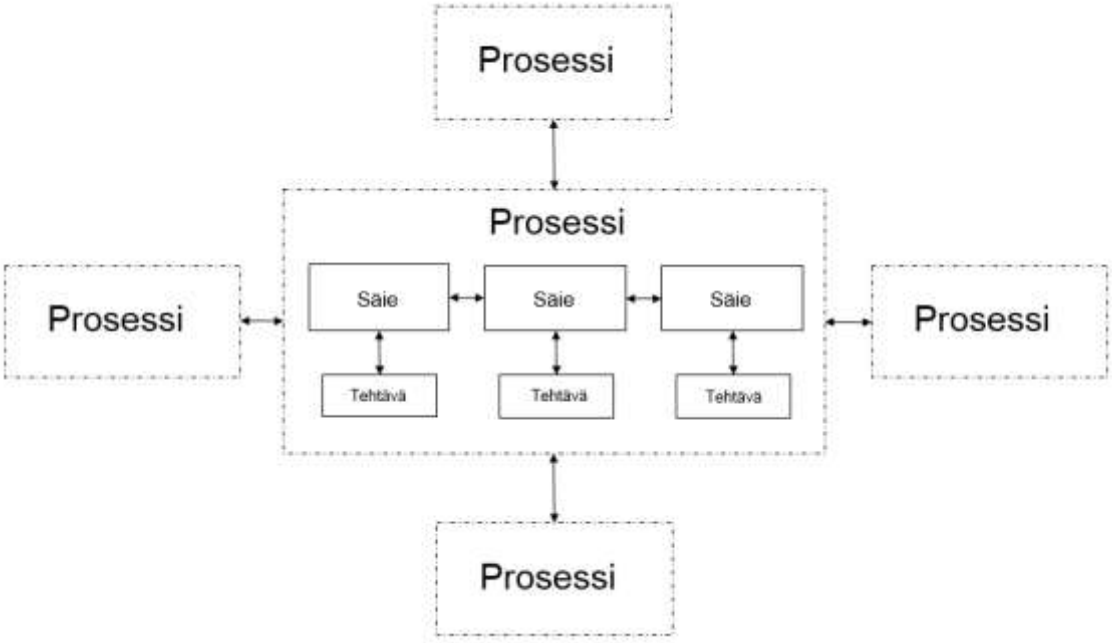
hettämään signaaliin kytketty slotti suoritetaan välittömästi, aivan kuten perinteisessä C++-ohjelmoinnissa metodeita kutsuttaessa. Vasta tämän jälkeen signaalin lähettänyt olio pääsee jatkamaan toimintaansa. Queued-yhteyksissä signaalit siirtyvät odottamaan vuoroaan Qt:n sisäiseen tapahtumaketjuun ja signaalin lähettänyt olio on välittömästi vapaa jatkamaan toimintaansa. Qt tarjoaa omat signaloidut luokat mm. säikeiden, ajastimien ja verkkoyhteyksien luomiseen, mikä helpottaa reaaliaikatoimintojen toteuttamista. Näitä käyttämällä voidaan välttää moniajoisuuden ja säikeistämisen mukanaan tuomia ongelmia, kuten jaettuun muistialueeseen samanaikainen kirjoittaminen ja päivitysaikoihin liittyvät ongelmat. (3; 5; 6; 8, s. 31–32, 35.)

2.2 Reaaliaikajärjestelmät

Reaaliaikajärjestelmissä oleellisena edellytyksenä on niiden taattu aikarajattu toiminta. Reaaliaikajärjestelmillä ei välttämättä tarkoiteta, että järjestelmät olisivat joka hetki täysin ajan tasalla, vaan sitä, että niitä ajetaan niille määritettyjen aikaikkunoiden puitteissa. Tämän pohjalta ne voidaan luokitella joko pehmeään tai kovaan reaaliaikaisuuteen kuuluviksi. Pehmeitä reaaliaikasovelluksia ovat esimerkiksi tietokonepelit ja mukavuussovellukset, joissa näistä aikarajoitteista poikkeaminen ei aiheuta järjestelmän toiminnan kannalta kriittistä virhettä. Koviin reaaliaikasovelluksiin kuuluvat mm. ajoneuvojen turvalaitteet ja sydämentahdistimet, joissa aikarajoitteiden noudattaminen on ensiarvoisen tärkeää. (7, s. 1–2; 8, s. 1.)

Monet reaaliaikajärjestelmät vaativat sekä säikeistämistä että nopeita vasteaikoja. Ne voivat myös olla yksinkertaisia ohjelmia, joiden toiminta tapahtuu yksittäisen ohjelmointisilmukan sisällä. (8, s. 1.) Suuremmat kokonaisuudet voivat koostua useista ajettavista prosesseista ja säikeistä (kuva 1). Yksittäinen prosessori voi suorittaa yhtä tai useampaa prosessia. Useiden prosessien suorittamista kutsutaan moniajoksi. Prosessit voivat taas sisältää useita säikeitä, joilla on omat suoritettavat tehtävänsä. Reaaliaikaisen ja ei-reaaliaikaisen järjestelmän välillä on usein eroja siinä, miten resurssien ja tehtävien jakaminen on toteutettu eri prosessien ja säikeiden välillä. (7, s. 14–19; 8, s. 151–152; 19, s. 23–32.) Ajonvakautusjärjestelmät usein koostuvat erillisistä prosesseista, joita ajetaan niiden

omien mikroprosessorien varassa, esimerkkinä ajoneuvotekniikassa käytetyt sulautetut elektroniset ohjausyksiköt (ECU). Ajonvakautusjärjestelmä voi sisältää omat ohjausyksiköt ajoneuvoon vaikuttavien kiihtyvyyksien ja asennon muutoksien tarkkailuun, ohjauspyörän asennon tarkkailuun, moottorin säätöön ja renkaiden lukkiutumisen estoon (ABS). Jokaisen ohjausyksikön tulee sisältää omat aikarajansa, joiden sisällä niiden halutaan suorittuvan. Esimerkiksi luistonestolta edellytetään vakaata vasteaikaa sille välille, kun pyörien luisto tunnistetaan ja järjestelmä suorittaa tarvittavat korjausliikkeet. Näiden lisäksi voi olla olemassa koko ajonvakautusjärjestelmän yli vaikuttavia aikarajoitteita. (7, s. 3–5; 17.)



KUVA 1. Prosessit ja säikeet moniajoisessa järjestelmässä

Aikarajoitteiden ohella reaaliaikajärjestelmien yleisiä piirteitä ovat moniajoisuus, tehtävien priorisoiminen, keskeytykset ja laiteläheisyys (8, s. 4–10; 19, s. 23–32).

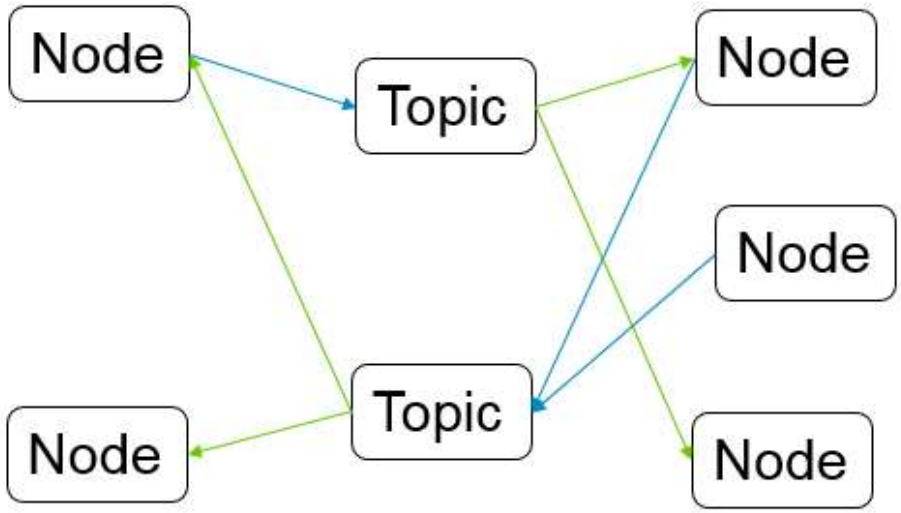
Tässä työssä tehdyssä robotin ohjauksessa hyödynnetään ns. reaaliaikaominaisuuksia jakamalla suoritettavia tehtäviä useille säikeille ja prosesseille, sekä käyttämällä ajastimia päivitystaajuuksien määrittämistä varten. Ohjausohjelmaa ajetaan Ubuntulla, joka on Linux-pohjainen käyttöjärjestelmä eikä suoranaisesti täytä reaaliaikaisuuden asettamia vaatimuksia (9, s. 3–5). Esimerkki reaaliaikaisesta käyttöjärjestelmästä on QNX, jolle Qt:llä tehdyt ohjelmat on mahdollista

kääntää (12). Reaaliaikaisuuden vaatimusten täyttyminen ei ollut edellytys tässä työssä tehdyn ohjauksen kannalta, vaan tässä luvussa selvitettyjen periaatteiden soveltaminen ja navigointisovelluksen toiminta riittävän luotettavalla tavalla.

2.3 Robot Operating System

ROS (Robot Operating System) on useista eri ohjelmistokehyksistä koostuva kokonaisuus robottien käyttöön ja kehitykseen. Se tarjoaa mm. rajapinnan monille laitteille, mekanismeja prosessien väliselle viestinnälle ja pakettien hallinnan. Se sai alkunsa Stanfordin yliopistolla vuonna 2007 nimellä switchyard. ROS-nimellä projekti käynnistyi saman vuoden lopulla ja sitä kehitti Willow Garagena tunnettu robotiikan tutkimukseen erikoistunut yritys. Vuodesta 2013 lähtien sen kehityksestä ja ylläpidosta on vastannut Open Source Robotics Foundation. (10.)

ROS ei myöskään suoranaisesti tue reaaliaikaisuutta, mutta suunnitteilla olevasta ROS 2.0:sta tuki saattaa löytyä (11). ROS:ssä ajettavia prosesseja kutsutaan nodeiksi. Niiden välinen kommunikointi tapahtuu julkaisemalla viestejä topiceille. Nodet voivat vastaanottaa muiden nodejen julkaisemia viestejä tilaamalla vastaavat topicit. (Kuva 2.) ROS tarjoaa myös mahdollisuuden viestintään palvelut-toiminnon avulla lähettämällä pyyntöjä ja vastaanottamalla vastauksia. (13.)

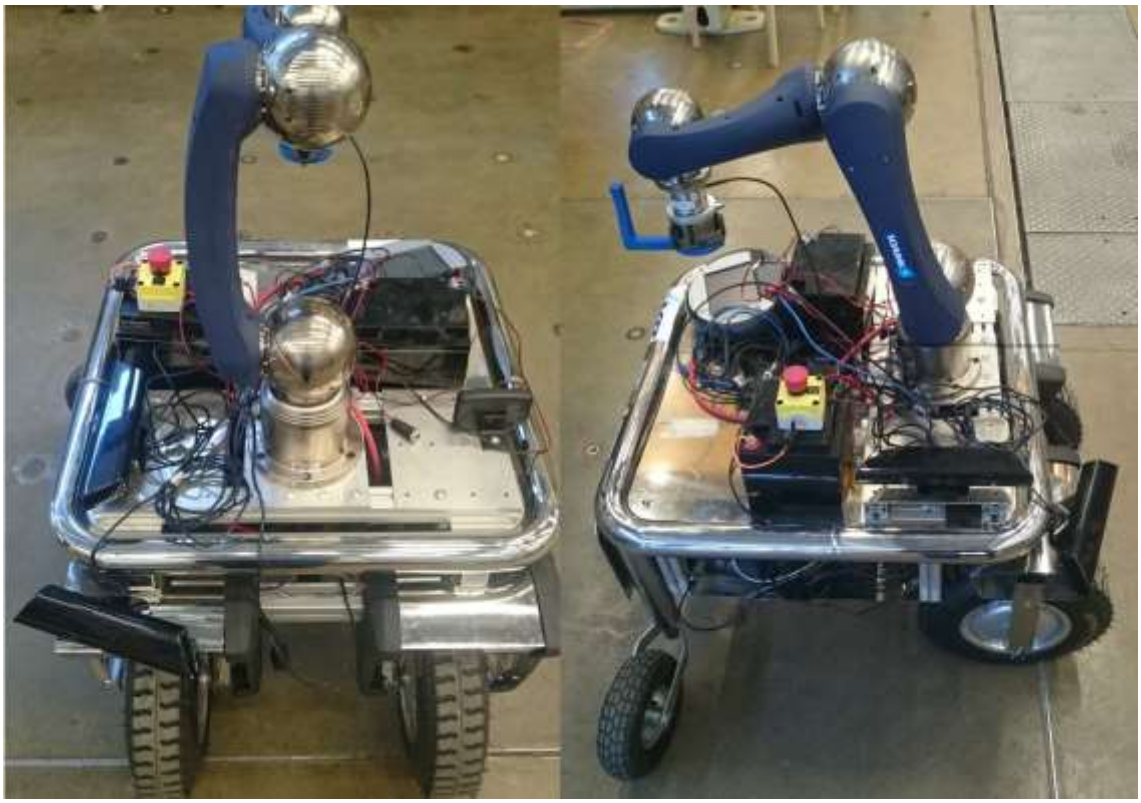


KUVA 2. Prosessien välinen kommunikointi ROS:ssä: julkaisut (sininen) ja tilaukset (vihreä)

Nodeja ympäristöön voidaan lisätä rekisteröimällä prosesseja ROS:n alaisuuteen. Tämä tapahtuu lisäämällä niiden lähdekoodiin tarvittavat kirjastot ja ajamalla komennot ros::init ja ros::NodeHandle. Catkin on yksi työkalu ROS:n käyttämien pakettien luomiseen. Paketit voivat sisältää nodeja, asetuksia sisältäviä tiedostoja, kolmannen osapuolen ohjelmia ja muuta oleellista toiminnallisuutta. (14.) Paketit ajetaan käyttämällä työkalua roslaunch ja määrittelemällä ajettavan paketin nimi sekä siihen liittyvä launch-tiedosto. Launch-tiedostossa määritellään mitä nodeja halutaan käynnistyvän ja mitkä ovat niihin liittyvät parametrit. Tässä opinnäytetyössä käytettävä konenäkösovellus on tällainen paketti, joka koostuu ajettaessa useammasta eri prosessista ja niiden välisistä viesteistä.

2.4 Mobiilirobottialusta

Navigointi toteutettiin nelipyöräiselle Probot Oy:n kehittämälle differentiaaliajoiselle mobiilirobottialustalle (kuva 3). Alustaan kuuluu rungon lisäksi kaksi MobilityModule-yksikköä ja robottikäsivarsi. MobilityModule-yksikkö sisältää moottoroidun renkaan ja moottoriohjaimen. Moottoriohjaimet on yhdistetty CAN-muuntimeen, johon luodaan yhteys Probot Oy:ltä saadun Python-koodin avulla. Qt:llä luodusta ohjauksesta taas luodaan TCP/IP-yhteys Python-koodin käyttämään TCP-pistokkeeseen. Robotin ohjaaminen tapahtuu lähettämällä kahta arvoa tähän pistokkeeseen, ajonopeutta ja renkaiden välistä nopeussuhdetta. Tämän jälkeen Python-koodi ja CAN-muunnin huolehtii näiden arvojen kääntämisestä moottoriohjaimille sopiviksi.



KUVA 3. Mobiilirobottialusta edestä ja sivulta

2.5 Kinect-liiketunnistin

Kinect on Microsoftin kehittämä liiketunnistin Xbox 360 -pelikonsolille. Työssä käytettiin kahta Kinect v1 -mallista kameraa. Kameramallille löytyy OpenKinect-kehitysryhmän ylläpitämä avoimen lähdekoodin ajurikirjasto nimeltä libfreenect, joka mahdollistaa sen käyttämisen PC-sovelluksissa. (15.) Navigoinnin suunnittelun kannalta oli oleellista tietää, kuinka leveä näkökenttä Kinectissä on. Microsoftin mukaan näkökentän kulma pystysuunnassa on 43° ja vaakasuunnassa 57° (16).

3 MOBIILIROBOTIT JA NAVIGOINTI

Tässä luvussa käydään läpi yleisimpiä mobiilirobotteihin liittyviä navigointimenetelmiä ja tehdään katsaus tämänhetkiseen tutkimustyöhön aiheesta. Pääpaino pidetään kartattomassa reaktiivisessa navigoinnissa, jonka alaisuuteen työssä toteutettu menetelmä voidaan katsoa kuuluvan. Työn menetelmässä muodostetaan pisteitä koordinaatistojen avulla, mutta robotilla ei ole kykyä tallentaa näistä minkäänlaista globaalia karttaa muistiin.

3.1 Mobiilirobottien navigointimenetelmät

Pitkäaikainen käsitys kehittyneistä navigoivista roboteista on ollut se, että niiden tulisi sisältää globaali kartta ympäristöstään ja kyky paikallistaa sijaintinsa kartan sisällä. Tähän karttaan perustuen robottien tulisi osata suunnitella kulkureittinsä sen hetkisen sijainnin ja määränpään välille. Tärkeäksi ominaisuudeksi autonomisessa mobiilirobotissa on myös mielletty sen yhtäaikainen kyky rakentaa näitä karttoja tekemiensä havaintojen pohjalta. Tällaista menetelmää kutsutaan nimellä SLAM (Simultaneous Localization And Mapping). (19, s. 107–108.) SLAM:ään perustuvia robotteja on nykyään tulossa markkinoille moniin erilaisiin tehtäviin yksitoikkoisesta tavaroiden kuljettamisesta aina, ihmisten kanssa vuorovaikutuksessa oleviin sosiaalirobotteihin asti. Esimerkiksi Kuri on myöhemmin 2017 myyntiin tuleva interaktiivinen kotirobotti. Sen ominaisuuksiksi on luvattu sosiaalisten vuorovaikutustaitojen ja eräänlaisena mediarajapintana toimimisen lisäksi kyky oppia navigoimaan ympäristössään. (24.) Roomba 980 on vuonna 2015 myyntiin tullut iRobotin valmistama robottipölynimuri. Se on ensimmäinen malli sarjassaan, jonka toiminta perustuu SLAM:ään. (25.) Robottiautot ovat myös yksi pitkäaikaisimmista sitä käyttävistä sovelluskohteista.

Teollisuuden näkökannalta katsottuna mobiilirobottien navigaatio voidaan mieltää yksinkertaisemmaksi. Silloin on usein riittävää, että robotti suorittaa siltä vaaditut tehtävät mahdollisimman kustannustehokkaalla tavalla. (19, s. 109.) Varastoissa ja teollisuudessa tavarankuljetukseen käytettyjä mobiilirobotteja kutsutaan vihivaunuiksi. Niille tärkeimpinä vaatimuksina voidaan pitää niiden turvallista toimintaa ja sitä, että tavarat saadaan helposti ja nopeasti kuljetettua eri pisteiden

välillä. Tällainen vihivaunu on yksinkertaisimmillaan mahdollista toteuttaa jonkinlaisella viivanseurannalla ja muutamalla turvalaitteella sekä rajaamalla sille kuuluvat alueet. Viivanseuranta voi olla värillisen teipin sijasta esimerkiksi radiosignaalia lähettävään kaapeliin tai magneetteihin pohjautuva ratkaisu (19, s. 169–170). Näihin perustuvat vihivaunut voivat olla elektronisesti ja ohjausteknillisesti hyvinkin yksinkertaisia laitteita.

Tutkimuspuolella on taas kehitetty ympäristössään sijaitsevia esineitä uudelleen järjestelevää robottia. Se sisältää fysiikkamoottorin, jota käytetään esineisiin liittyvien simulaatioiden mallintamiseen. Robotin käsitellessä esineitä se muodostaa niistä malleja käytettäväksi simulaatioissa. Näiden simulaatioiden pohjalta taas luodaan suunnitelmat siitä, miten esineiden kanssa tulisi vuorovaikuttaa, että määränpäähän pääsy mahdollistuisi. (21.) Sisätiloissa tapahtuvaan paikannukseen liittyvien ongelmien ratkaisemiseksi on tutkittu geomagneettista menetelmää. Menetelmä perustuu sisätiloissa tapahtuviin maapallon magneettikentän vääristymiin. Nämä vääristymät ovat rakennuksille ominaisia ja riippuvat niiden rakenteesta sekä sisältämästä irtaimistosta. Olettaen, että rakennuksissa ei tapahdu merkittäviä muutoksia, näistä vääristymistä saadut ns. sormenjäljet pysyvät muuttumattomina pitkilläkin aikaväleillä. Paikannuksen tarkkuus on kuitenkin sidottavissa siihen, kuinka paljon rakennukset sisältävät teräsrakenteita ja elektronisia laitteita. Geomagneettisen paikannuksen etuja on, että se ei edellytä sille erikseen suunniteltua infrastruktuuria ympärilleen ja robotin anturoinnissa paikannukseen tarvitaan vain magnetometri. (23.)

3.2 Kartaton navigointi

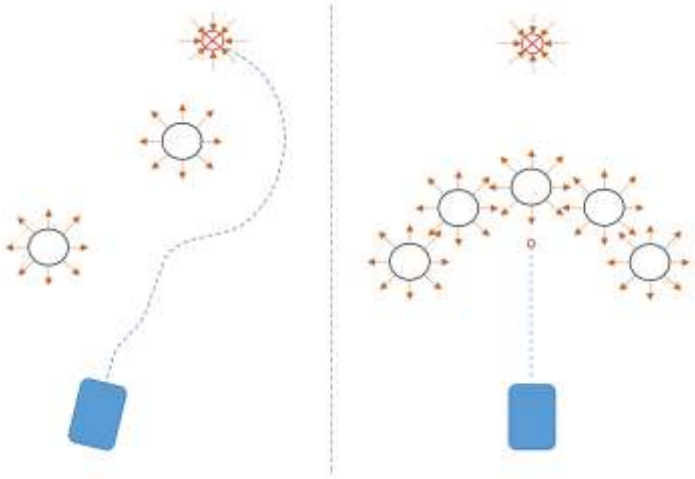
Yksinomaan viivanseurantaan perustuvat vihivaunut navigoivat ympäristössään ilman minkäänlaisia koordinaatistoihin perustuvia karttoja. Vihivaunun ohjaus voidaan toteuttaa esimerkiksi kytkemällä robotin rataa seuraavat anturit suoraan vahvistimen kautta sitä ajaviin moottoreihin. Tällainen ohjaus toimii kuten Braitenbergin ajoneuvot reagoiden välittömästi antureilta saatuihin havaintoihin. Tutkimuspuolella tällaisesta ohjauksesta käytetään termiä reaktiivinen navigointi. Reaktiivisesta navigoinnista puhuttaessa tarkoitetaan yleensä ulkoista maailmaa

kuvaavan (globaalin) kartan ja reitin suunnittelun puuttumista robotin toiminnoista. (26, s. 88–90). Globaalin kartan lisäksi käytetään robotin ympärille rakentuvaa paikallista karttaa, jota päivitetään jatkuvasti robotin anturihavaintojen perusteella. Tämä kartta on verrattavissa osaksi siihen, miten tutka toimii. Koska paikallista karttaa päivitetään tutkan tavoin suhteessa robotin asentoon ja sijaintiin, sen yhteydessä käytetäänkin usein napakoordinaatistoa karteesisen sijasta. (19, s. 121–122). SLAM:ssä tätä paikallista karttaa hyödynnetään myös paikannuksessa vertaamalla sitä muistissa olevaan globaaliin karttaan robotin sijainnin arvioimiseksi (26, s. 109–110). Paikallisella kartalla ei itsessään tarkoiteta, että minkäänlaista karttaa olisi välttämättä muistissa, vaan se kattaa käsitteenä tietyllä hetkellä saadut anturihavainnot. Tässä työssä siihen perustuvat tekniikat mielletään kartattoman navigoinnin alaisuuteen.

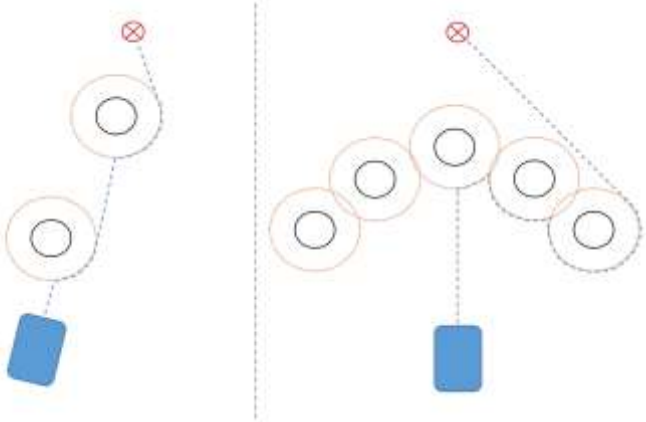
Reaktiiviset navigointimenetelmät voivat olla toteutukseltaan hyvinkin yksinkertaisia, mutta silti toimivia ratkaisuja moniin tarpeisiin. Esimerkkinä on konenäön tunnistamille maamerkeille hakeutuva mobiilirobotti. Tässä opinnäytetyössä hyödynnetyistä 3D-pistepilvistä saadaan tarkkaa tietoa löydetyistä maamerkeistä. Hakeutuminen voidaan toteuttaa ohjaamalla robottia seuraavaa maamerkkiä kohti esimerkiksi suhdesäädön avulla ja etäisyyttä seuraamalla. Yksi ratkaisu käytettäessä paikallista karttaa olisi, että robotille laskettaisiin maamerkille johtava liikerata, jonka robotti toteuttaisi odometrian avulla. Pelkästään odometriaa käytettäessä kertyy kuitenkin virhettä liikeratoja ajettaessa, mikä pitäisi ottaa huomioon. Liikeradan laskemisessa hyviä puolia on, että määrittämällä radan muoto, voidaan ottaa huomioon robotin kinematiikka sekä ajaa se haluttuun loppuasentoon.

Perinteisiä tapoja etsiä lyhyin reitti päämäärään ovat olleet mm. esteiden kasvattamiseen ja virtuaalisiin voimakenttiin perustuvat tekniikat (19, s. 112–113). Mobiilirobottien laskentakapasiteetti on ollut aiemmin rajallisempaa kuin nykyään, minkä takia on tarvittu kevyempiä menetelmiä navigoinnin toteuttamiseksi. Navigointiin voidaan käyttää ruutupohjaisia karttoja, joissa ruudut merkitään joko esteiksi tai tyhjiksi. Tämän lisäksi kartta sisältää päämäärän, jonne robotin halutaan liikkuvan. Virtuaaliset voimakentät on tähän yhdistetty menetelmä, jossa päämäärä muodostaa ympärilleen puoleensavetävän ja esteet luotaantyöntävän

kentän (kuva 4). Esimerkiksi laskemalla yhteen näiden voimakenttien muodostamat vektorit, saadaan ruudukkoon reitti, jota pitkin robotin tulee navigoida. (20, s. 238–239.) Esteiden kasvattamisessa robotti yrittää liikkua suoraan päämäärää kohti. Havaittujen esteiden ympärille robotti muodostaa rajat, joita sen on käytettävä esteiden kiertämiseen niiden osuessa sen reitille. (19, s. 113; kuva 5.) Voimakenttiin perustuvat menetelmät ovat pääpiirteissään melko yksinkertaisia toteuttaa ja muodostavat esteiden kasvatusta pehmeämpiä liikeratoja, mikä tekee niistä paremmin soveltuvia useimpien robottien ohjaukseen. Yksi niitä käytettäessä huomioitava asia on kuitenkin lokaalien minimien muodostuminen, mikä saattaa jumittaa robotin, kuten kuvassa 4 oikealla.



KUVA 4. Virtuaalisiin voimakenttiin perustuva menetelmä



KUVA 5. Esteiden kasvatukseen perustuva menetelmä

Molemmat navigointitavat ovat alun perin yksinkertaisiin karttoihin pohjautuvia menetelmiä, mutta niiden luonteen vuoksi niitä on mahdollista soveltaa myös kartattomaan navigointiin. Esteiden kasvattamisessa riittää, että robotilla on tieto, missä määränpää sijaitsee kullakin hetkellä. Robotille voidaan täten kirjoittaa algoritmi, joka ohjaa robottia tätä päämäärää kohti. Kohdattaessa esteitä, kierretään ne määriteltyjä rajoja noudattaen, kunnes esiintyy mahdollisuus jatkaa suoraan kohti tiedossa olevaa päämäärää. Voimakenttiin perustuvia menetelmiä on on kehitetty useita ja vaikkakin monet niistä ovat toteutukseltaan karttoihin perustuvia, niin kartattomiakin vaihtoehtoja löytyy.

Eräässä voimakenttiin perustuvassa reaktiivisessa menetelmässä robottia ohjataan sen kulmakiihtyvyyttä muuttamalla kameralta saatujen havaintojen pohjalta. Menetelmässä oletetaan, että suunta ja etäisyys määränpäähän on joko saatavilla suoraan robotin antureilta tai ylemmiltä navigointikerroksilta. Esteistä tieto saadaan etukameran avulla ja robottiin vaikuttavat virtuaaliset voimat lasketaan esteiden havaitun leveyden perusteella. Tällä tavoin sekä esteiden fyysinen koko, että niiden etäisyys kamerasta, saadaan otettua huomioon. Tämän ansiosta ei ole erikseen tarpeellista saada tietoa esteiden todellisesta etäisyydestä. Testeissä menetelmällä on saatu aikaiseksi luonnollisen näköisiä liikeratoja ja sen on todettu soveltuvan hyvin epäholonomisille mobiiliroboteille, kuten tässä työssä käytettävälle alustalle. (22.) Tämä menetelmä voidaan toteuttaa laskennallisesti vaatimattomalla 2D-konenäkösovelluksella. Nykyiset 3D-konenäköjärjestelmät tarjoavat kuitenkin tarkkaa tietoa etäisyyksistä ja kappaleiden koosta sekä ominaisuuksia havaittujen esineiden yksilöintiin. Tämä puolestaan avaa ovia navigoinnin suhteen mahdollistaen aiemmin monimutkaisten algoritmien toteuttamisen yksinkertaisemmilla tavoilla.

Toinen ilman 3D-konenäköä toimiva tekniikka on optiseen virtaukseen perustuva navigointi, joka on saanut alkunsa eläinten havainnointikykyyn liittyvästä tutkimuksesta. Sillä tarkoitetaan visuaalisen informaation näennäisen liikenopeuden pohjalta tapahtuvaa ohjausta. Robotin ollessa liikkeessä sen sivuilla tapahtuva ns. optinen virtaus on sitä nopeampaa mitä lähempänä sitä tarkkaileva anturi fyysisesti sijaitsee. Robotin on mahdollista ohjautua esimerkiksi käytävän keskilinjaa

pitkin, kun se pyrkii PID-säädön avulla pitämään molempien seinien näennäisen liikenopeuden lähellä toisiaan. (20, s. 250–253.)

Kartattoman navigoinnin toteuttamiseksi on myös kokeiltu menetelmiä, joissa robotti tallentaa kuvia ympäristöstään ja assosioi kuviin tiettyjä toimintoja. Yksi tällainen menetelmä on VSRR (View-Sequenced Route Representation), jossa robotti kuvantaa ympäristönsä ja vertaa sitä sen muistista löytyviin kuviin. Jos kuvat ovat tarpeeksi lähellä toisiaan, niin robotti suorittaa kyseiseen kuvasarjaan liittyvät toiminnot. Esineiden tunnistamiseen pohjautuvia symbolisia menetelmiä on myös tutkittu. Näissä robotti voi esimerkiksi suorittaa käskyjä kuten minkä esineen luo seuraavaksi liikkua tai toteuttaa niiden pohjalta määränpäähän johtavan sekvenssin. (20, s. 251–253.)

Navigointimenetelmien kehitys on seurannut hyvin pitkälti saatavilla olevien antureiden ja tietokoneiden laskentatehon kehitystä. Tässä luvussa käsiteltyjä menetelmiä on tutkittu usean vuosikymmenen ajan ja niiden pohjalta on etsitty yhä uusia ratkaisuja. Nykyään laskentateho ja kameratekniikka ovat tuoneet yhä realistisemmaksi vaihtoehdoksi myös 3D:tä käyttävät konejärjestelmät.

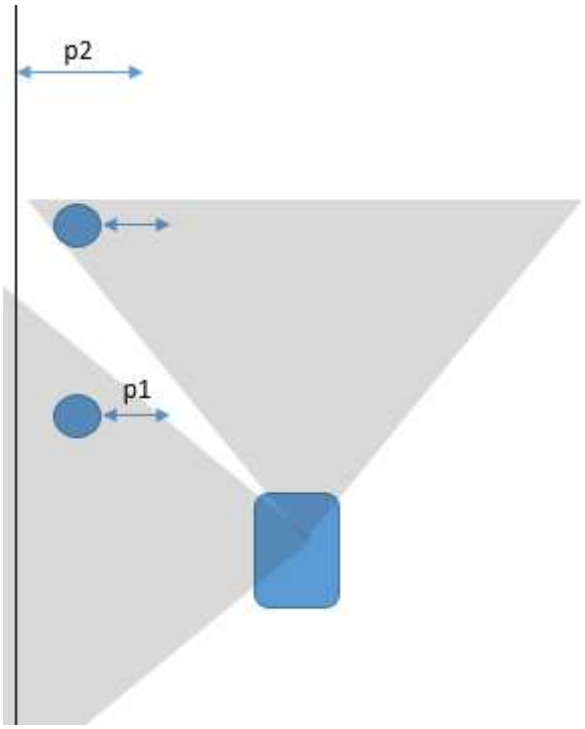
4 OHJAUKSEN RAKENTEEN JA NAVIGOINNIN TOTEUTUS

Toteutukseen käytetty aika jakaantui pääpiirteissään kolmen eri osan kesken. Konenäkösovellus piti saada keskustelemaan ohjausohjelman kanssa. Toisena osana oli robottiin aikaisemmin tehdyn ohjelman rakenteen laajentaminen useampiin kerroksiin. Kolmas osa painottui itse navigointiohjelman toteutukseen. Näitä osa-alueita ei toteutettu järjestyksessä, vaan tarpeen mukaan pienissä palasissa, joissa toiminnallisuutta lisättiin aina niiden välissä suoritettavien testien onnistuessa. Tässä luvussa nämä alueet kuitenkin käsitellään erillisinä, jotta tehdystä työstä muodostuisi selkeämpi kuva.

4.1 Konenäköjärjestelmän liittäminen robottiin

Mobiilirobotin toimintaan ja ohjaukseen tutustumisen jälkeen ensimmäisiä asioita oli saada VTT:llä kehitetty konenäkösovellus toimimaan yhteistyössä robotin ohjausohjelman kanssa. Kyseinen sovellus on toteutettu ROS:lle ja se käyttää PCL:n (Point Cloud Library) tarjoamia luokkia. PCL tarjoaa työkaluja 3D-pistepilvien käsittelyyn. ROS sisältää oman versionsa PCL:stä, joka on myös alun perin Willow Garagelta alkanut projekti. (18.)

Tässä vaiheessa työtä oli jo suunnitelmia navigoinnin mahdollisesta toteutustavasta ja vaikutti siltä, että myöhemmin työssä jouduttaisiin käyttämään useampaa kuin yhtä kameraa (kuva 6). Konenäkösovellus vaati myös sitä ajavalta tietokoneelta huomattavan määrän laskentatehoa. Tämän takia kannettava tietokone, jolla robotin ohjausta ja konenäköä oli tarkoitus ajaa, vaihdettiin tehokkaampaan Dell Precision 5510 -malliin, joka sisältää neliytimisen Intel Core i5-6300HQ -suorittimen. Tietokoneeseen asennettiin uusimmat versiot Ubuntusta ja tarvittavista kehitystyökaluista, kuten Qt Creatorista ja ROS:stä. Tästä kuitenkin aiheutui ongelmia konenäköohjelmiston ylösajamisen kanssa. Aluksi ohjelma ei suostunut kääntymään uudelle alustalle versioristiriitojen takia. Kääntymisen jälkeenkin ongelmia oli vielä ohjelman epävakauden kanssa, kun siihen liitettiin Kinect v1 -kamera. Näitä ongelmia ratkottiin yhteistyössä konenäön ylläpidosta vastanneen Jari Aholan kanssa, kunnes konenäkösovellus toimi vakaasti ja sitä pystyi käyttämään navigoinnissa.



KUVA 6. Alustavia hahmotelmia etu- ja sivukameran sijoituksesta

Ohjausta varten konenäkösovellukselta oli saatava koordinaattiarvot tunnistettujen maamerkkien ja tasopintojen keskipisteistä sekä tasopintojen pintanormaaleista. Tieto näistä tallentuu konenäön tunnistusprosessissa käytetyn ohjelmointikielen (C++) sisältämiin tietorakenteisiin, joita kutsutaan vektoreiksi (ei sama asia kuin myöhemmin navigoinnissa käytettävät matemaattiset vektorit). Tiedonsiirron toteuttamiseen konenäön ja robotin ohjausprosessin välille käytettiin UDP:tä (User Datagram Protocol). Koodissa vektorien sisältö käydään läpi iteroimalla niitä ja niissä oleva tieto tallennetaan merkkijonoksi, johon lisätään tarvittavat välimerkit tietotyyppien erottelemiseksi (kuva 7).

```
buf_l += sprintf(buf+buf_l, ":");

vector<double>::iterator iter_buf_double = g_tr.begin();
i_buf = 0;
while (iter_buf_double != g_tr.end()) {
        iter_buf_double++;
        buf_l += sprintf(buf+buf_l, " %lf", g_tr[i_buf]);
        i_buf++;
}

buf_l += sprintf(buf+buf_l, ":");
```

KUVA 7. Vektoreiden iterointi ja välimerkkien lisäys

Merkkijono lähetetään UDP-pistokkeeseen, josta ohjausprosessi lukee sen käyttämällä QUdpSocket-luokan toimintoja. Sen jälkeen vastaanotetusta merkkijonosta rakennetaan omat tietorakenteet maamerkeille ja tasopinnoille (kuva 8).

```
QNetworkDatagram datagram = m_socket_r5cop->receiveDatagram();
if(datagram.isValid())
{
    m_receivedDatagram = QString(datagram.data());

    m_tmpDatagram = m_receivedDatagram.split(":");
    m_landmark_name = m_tmpDatagram[0].split("#");
    m_landmark_rec = m_tmpDatagram[1].split(" ");
    m_landmark_pos = m_tmpDatagram[2].split(" ");
    m_obstacle_pos = m_tmpDatagram[3].split(" ");
    m_coarseplane_pos_vec = m_tmpDatagram[4].split(" ");
    m_coarseplane_size = m_tmpDatagram[5].split(" ");

    m_landmarks.clear();

    for(int i = 0; i < (m_landmark_rec.length()-1); i++)
    {
        Landmark tmp;
        tmp.rec = m_landmark_rec[i+1].toInt();
        tmp.name = m_landmark_name[tmp.rec+1];
        tmp.x = m_landmark_pos[1+3*i].toDouble();
        tmp.y = m_landmark_pos[2+3*i].toDouble();
        tmp.z = m_landmark_pos[3+3*i].toDouble();
        m_landmarks.push_back(tmp);
    }
```

KUVA 8. Maamerkkien tietorakenteiden luominen vastaanotetusta merkkijonosta

UDP on TCP:tä (Transmission Control Protocol) nopeampi yhteydetön protokolla, mutta ei yhtä luotettava. Tätä ratkaisua päädyttiin käyttämään koska konenäön tunnistusprosessin lähdekoodi ja Qt sisälsivät valmiit luokat sen käyttöönottoon. Vaihtoehtona TCP olisi ollut lähes yhtä hyvä ratkaisu tämän työn kannalta, mutta UDP:n yhteydettömyyden tarjoama yksinkertaisuus nousi etusijalle. Vektoreiden koon tiedettiin myös pysyvän pieninä, minkä takia lähetyksen lisääminen tunnistusprosessin puolelle ei oleellisesti hidastanut sen toimintaa.

Alkuun maamerkkien seuranta toteutettiin pelkästään yhden Kinectin avulla. Seinän seurannassa oli kuitenkin tarkoitus käyttää sivukameraa etukameran lisäksi, mikä tarkoitti koodimuutoksia launch-tiedostoon ja kameran syvyysarvoja lukevaan prosessiin. Konenäkösovellus käyttää libfreenect-kirjastoa syvyysarvojen lukemiseen Kinectiltä, ennen kuin ne muutetaan 3D-pistepilveksi. Kameroille luotiin yhteinen launch-tiedosto. Tiedoston avulla kummallekin kameralle voidaan

käynnistää omat prosessit ja määrittää yksilölliset parametrit. Parametrien avulla määritellään mm. kameroiden käyttämät UDP-pistokkeet ja kerrotaan libfreenectista tuoduille funktioille kumpaa kameraa käyttää syvyystiedon lukemiseen.

Myöhemmässä vaiheessa osaksi navigointia haluttiin vielä törmäyksenesto, jonka vaatimuksena oli navigointiohjelman alimman kerroksen mukainen reaktioaika. Tässä vaiheessa navigointiohjelmasta päätettiin luoda yhteys ROS:n puolelle rekisteröimällä se nodeksi. Tämä sen takia että voitaisiin lukea konenäön tunnistusprosessiin menevää pistepilviviestiä suoraan topicilta. Tämä toiminnallisuus on kuvattu tarkemmin luvussa 4.3.3 Törmäyksenesto.

4.2 Ohjausohjelman rakenne

Aikaisempi koordinaattiohjaus sisälsi käyttöliittymän, johon käyttäjä pystyi syöttämään haluamansa xy-koordinaatit robotin ajamista varten. Käyttöliittymä ja liikeradan laskenta oli ennestään jaettu omille säikeille. Ohjausta lähdettiin tästä laajentamaan useampiin kerroksiin, joiden kesken robotin toiminallisuus jaettiin. Nämä kerrokset on havainnollistettu yleisellä tasolla liitteessä 1.

Kerrokset luodaan käyttämällä apuna Qt:n luokkia QObject, QThread ja QTimer. Luotujen olioiden käyttämät luokat periytetään QObject-luokasta, minkä jälkeen lisätään Q_Object makro niiden välisen signaloinnin mahdollistamiseksi (kuva 9).

```
class CameraControl : public QObject
{
    Q_OBJECT
```

KUVA 9. QObject-luokan periyttäminen

Olioiden jakaminen eri säikeisiin tapahtuu QThread-luokkaa käyttäen. Ensin luodaan uudet oliot ja säikeet, minkä jälkeen oliot siirretään säikeiden alaisuuteen. Säikeet täytyy myös käynnistää erikseen jossain vaiheessa koodia. (Kuva 10.)

```
m_cameraControl = new CameraControl();
m_cameraControlThread = new QThread();
m_cameraControl->moveToThread(m_cameraControlThread);

...

m_cameraControlThread->start();
```

KUVA 10. QThread-luokan käyttäminen

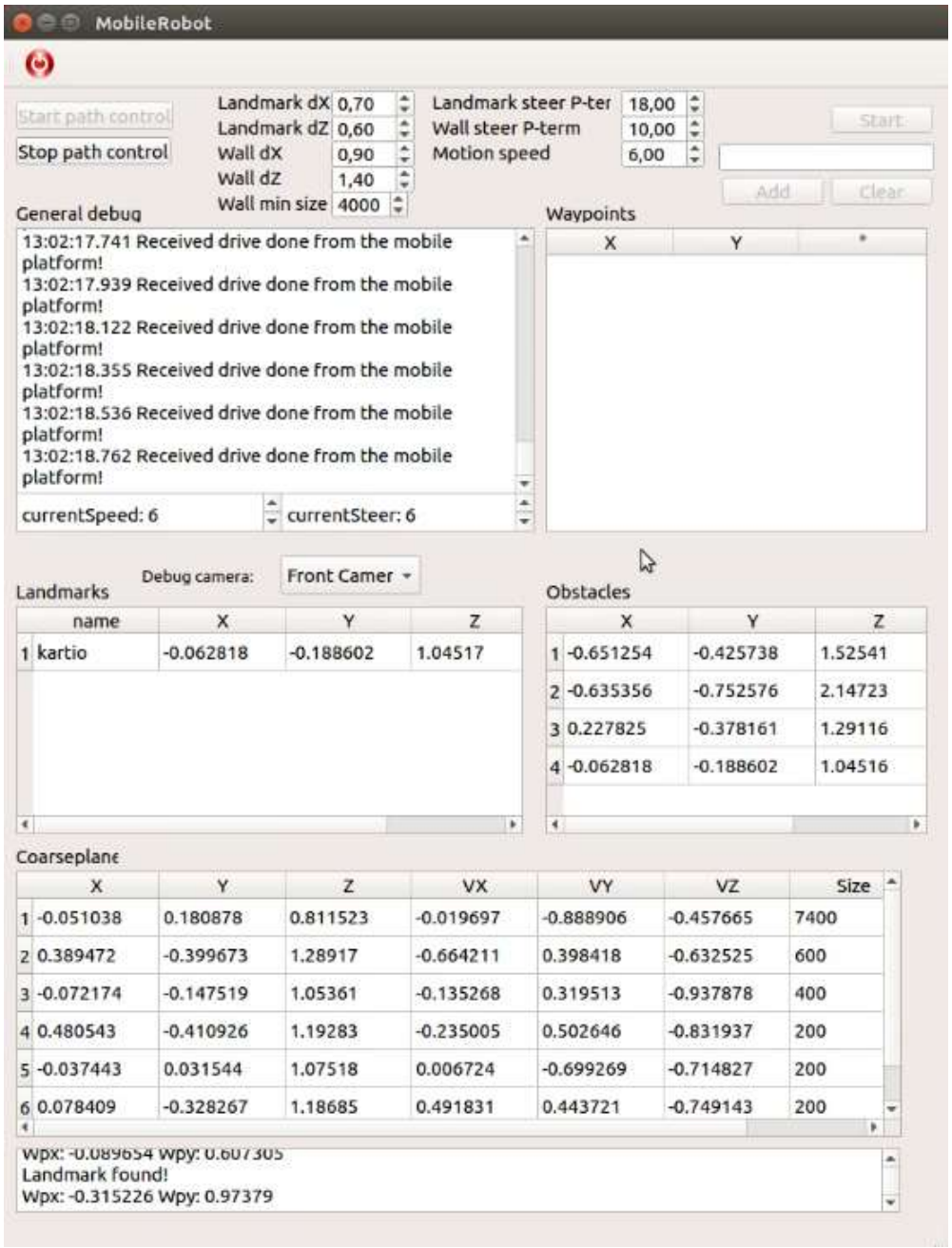
Oleellista oli myös saada eri kerroksien toimintaa eri päivitystaajuuksien taakse. Työn kannalta oli riittävää, että päivitystaajuudet olivat välillä 200–800 ms. Kyseiset päivitystaajuudet olivat sen verran hitaita, että säikeille jäi riittävästi aikaa tehtäviensä suorittamiseen ennen seuraavaa sykliä, ja voitiin luottaa toimintojen suorittuvan ajallaan. Ajastus tehdään luomalla QTimer-luokkaan perustuvia olioita, joille määritellään haluttu aikaväli ja suoritettava slotti (kuva 11).

```
m_cameraControlTimer = new QTimer(this);
connect(m_cameraControlTimer, &QTimer::timeout, this, &CameraControl::cameraControlCycle);

if(m_cameraControlTimer != NULL)
{
    if (!m_cameraControlTimer->isActive())
        m_cameraControlTimer->start(CAMERACONTROL_CYCLETIME);
}
```

KUVA 11. QTimer-luokan käyttäminen

Qt huolehtii säikeiden välisestä resurssien käytöstä ja tämän ansiosta siihen liittyviltä ongelmilta työssä vältyttiin. Yksi huomioitava asia useita säikeitä käytettäessä oli kuitenkin signaloinnin oikeanlainen kytkeminen. QThread-luokkaan sisältyy mm. signaali, joka lähetetään silloin kun kyseinen säie on käynnistynyt onnistuneesti. Tämän taakse kytkettiin olioiden alustustoimenpiteet, kuten signaloinnin yhdistäminen ja alempien kerrosten sekä olioiden luominen. Tämän avulla varmistettiin koodin oikea-aikainen toiminta ja luotujen olioiden siirtyminen niille kuuluvien säikeiden alaisuuteen. Signalointia käytettiin myös olioita poistettaessa muistivuotojen estämiseksi. Tämän varmistamiseksi apuna käytettiin Valgrindia, joka oli sen asentamisen jälkeen helposti integroitavissa Qt Creatoriin. Valgrind sisältää useita työkaluja erilaisten muistivirheiden paikallistamiseksi ja korjaamiseksi.

Käyttöliittymää laajennettiin myös huomattavasti testiajojen ja niistä saatavan tiedon tarkkailun helpottamiseksi. Siihen lisättiin navigaatioon liittyvien parametrien muokkaaminen ja useita ajonaikaisia debug-tietoja. (Kuva 12.)



KUVA 12. Testausta varten luotu käyttöliittymä

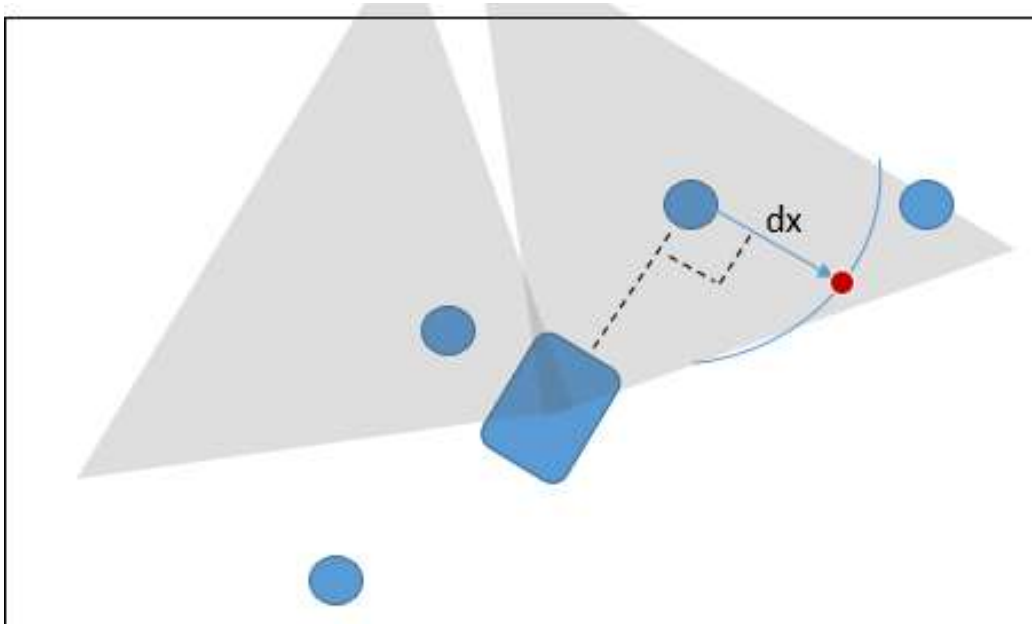
4.3 Navigointi

Lähtökohtana navigoinnin suunnittelulle oli, että robotista tulisi löytyä seinänseuranta ja sen pitäisi pystyä ajamaan mutkat ja mahdolliset U-käännökset kartioiden määrittämällä tavalla. Aluksi työssä suunniteltiin kartioiden perusteella tapahtuva ohjaus ja sen jälkeen seinäpintojen mukaan navigoiva osuus. Osat testattiin erikseen ja vasta sen jälkeen ohjelmoitiin niiden yhtäaikainen toiminta. Lopuksi ohjelmaan lisättiin vielä törmäyksenesto.

4.3.1 Maamerkit

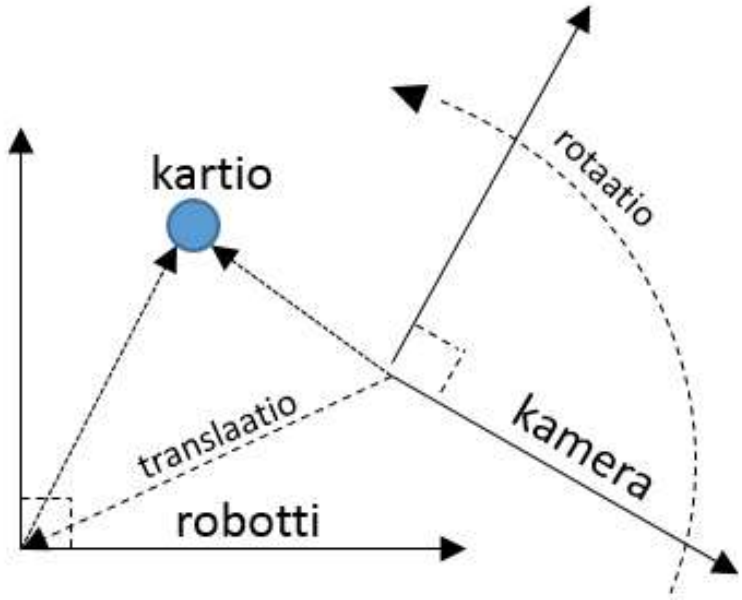
Maamerkkien navigoinnissa oletettiin kartioiden esiintyvän aina perättäin joko suoran tai kaarroksen muodossa. Tämän perusteella hahmoteltiin, että robotti valitsisi aina seuraavan vuorossa olevan kartion kohteekseen. Navigointia suunnitellessa vaihtoehtoina oli, käytettäisiinkö robotista jo löytyvää liikeradan laskemista vai tehtäisiinkö jonkinlainen säätöön perustava ohjaus. Liikeradan ajamisen kanssa ongelmia kuitenkin aiheutti sen hetkinen odometrian puuttuminen, minkä takia sen luotettavuus ei olisi ollut kovin hyvä. Toisekseen tarkoituksena oli kehittää mahdollisimman yksinkertainen menetelmä, jolla radan seuranta onnistuisi. Tämän pohjalta navigointia päätettiin lähteä lähestymään säätöpohjaisen ohjauksen kautta.

Säätöpohjaista ohjausta suunnitellessa ajateltiin edessä olevan maamerkin sijaitsevan robotin kulkusuuntaan nähden kohtisuoralla akselilla ja että kohdepiste määritettäisiin samalle akselille tietyn välimatkan *dx* etäisyydelle maamerkistä. Tällöin kohdepisteen tulisi liikkua puolikaaren muodossa maamerkin ympärillä, kun robotti kääntyy sitä kohti. (Kuva 13.) Ajateltiin että lisäämällä tähän suhdesäätö, jossa robotti kääntyy kohdepistettä kohti voimakkuudella, joka riippuu sen hetkisestä suuntauksesta ja parametrina annettavasta vahvistuksesta, robotin tulisi ohjautua luontevalla tavalla pistettä kohti. Ohjaukseen tuli myös määritellä etäisyys, jonka perusteella valittaisiin kohteeksi seuraava maamerkki.



KUVA 13. Kohdepisteen hakeminen maamerkistä

Maamerkkien navigointiin käytettiin robotin etukameraa. Maamerkkien keskipisteet saatiin kameran mukaisessa karteesisessa koordinaatistossa. Ohjausta varten konenäöltä saadut xyz-koordinaatit täytyi muuttaa robotin koordinaatistoon. Tämä tarkoitti aluksi vain translaatiota koordinaatistojen välillä. Translaatiossa kameran koordinaatiston origo (nollapiste) siirretään vastaamaan robotin origoa. Testeissä ongelmia kuitenkin ilmeni Kinectin näkökentän kapeuden takia. Vasemmalle tehtävissä käännöksissä robotin näkökentästä hävisivät herkästi sen oikealla puolella sijaitsevat maamerkit, mikä aiheutti ongelmia loivissa mutkissa tai suoria reittejä ajettaessa. Asian korjaamiseksi Kinectin kulmaa päätettiin muuttaa osoittamaan hieman etuviistoon, minkä takia koordinaattimuunnokseen tarvittiin avuksi myös rotaatio. (Kuva 14.)



KUVA 14. Koordinaatistomuunnos kameralta robotille

Muunnokset kirjoitettiin käyttämällä apuna C++-kielelle tehtyä Eigen-kirjastoa, joka sisältää mm. operaattoreita ja algoritmeja matriisien sekä vektoreiden käsittelyyn. Muunnoksien toteuttamiseen olisi riittänyt kaksiulotteiset matriisit ja vektorit, mutta laajennettavuuden kannalta käytettiin neljäulotteisia, joista jätettiin pystyakselin (y-akseli) suuntainen koordinaatti käyttämättä. Rotaatio tehdään y-akselin ympärillä käyttäen sille määriteltyä rotaatiomatriisia (kaava 1).

$$\mathrm{R_y}(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \qquad \textit{KAAVA 1}$$

Rotaatiossa ja translaatiossa käytettävät suureet annetaan käyttäjän muutettavina parametreina, jotka määritetään kameran sijainnin ja asennon perusteella. Robotin kohteeksi ei kuitenkaan haluttu suoraan maamerkkiä, vaan piste maamerkin ympäriltä. Parametrilla *m_landmark_x_offset* määritellään kuinka paljon ja kummalle puolelle maamerkkiä robotin halutaan navigoivan. (Kuva 15.)

```
Matrix4d rotationYMat;
m_CFRotAngle = CF_ROTATION / 180.f * M_PI;
rotationYMat << qCos(m_CFRotAngle), 0, qSin(m_CFRotAngle), 0,
                0, 1, 0, 0,
                -qSin(m_CFRotAngle), 0, qCos(m_CFRotAngle), 0,
                0, 0, 0, 1;

Vector4d robotCoordinate(0, 0, 0, 1);
Vector4d landmarkLocation(landmark.x, 0, landmark.z, 0);
Vector4d displacementNxWp(m_landmark_x_offset, 0, 0, 0);
Vector4d cameraDisplacement(CF_OFFSETX, CF_OFFSETY, CF_OFFSETZ, 0);


robotCoordinate += landmarkLocation;
robotCoordinate = rotationYMat * robotCoordinate;
robotCoordinate -= cameraDisplacement;
robotCoordinate -= displacementNxWp;
```

KUVA 15. Koordinaattimuunnos kameralta robotille

Ohjausta varten koordinaatit muutetaan vielä xyz-koordinaateista xy-muotoon (kuva 16).

```
m_nextWpx = *(robotCoordinate.data());
m_nextWpy = *(robotCoordinate.data() + 2);
```

KUVA 16. Muunnos xyz-koordinaatistosta xy-koordinaatistoon

Tämän jälkeen xy-koordinaatit viedään seuraavaan ohjauskerrokseen, jossa niiden perusteella toteutetaan suhdesäätö, joka päivittää robotin kääntymisestä vastaavaa arvoa. Robotin kanssa keskustelevalle Python-koodille tieto välitetään 200 ms:n välein TCP/IP-yhteyden kautta, käyttäen lähetettävinä arvoina ajonopeutta ja renkaiden välistä nopeussuhdetta. Nämä arvot Python-koodi muuttaa renkaita vastaaviksi ohjausarvoiksi, jotka lähetetään vuorostaan eteenpäin CAN-muuntimelle.

4.3.2 Seinä

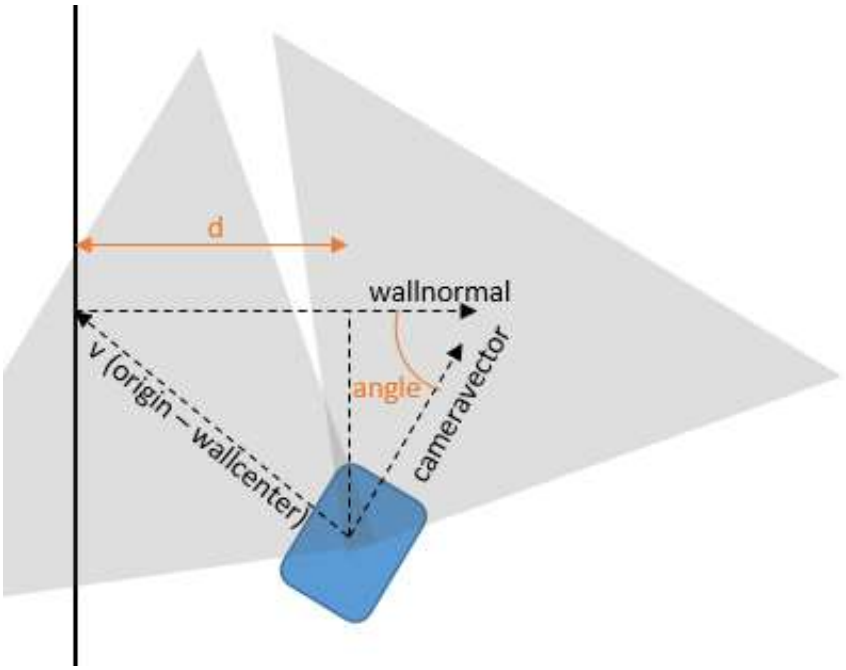
Seinänseurantaa varten saatavilla oli tietoa tasopintojen keskipisteistä, koosta ja pintanormaaleista. Näiden perusteella ensimmäinen mieleen tullut toteutus oli laskea trigonometrian avulla robotin etäisyys seinästä ja säädön avulla ohjata robottia pitämään etäisyys vakiona. Jatkon kannalta haluttiin kuitenkin seinistäkin saatavan koordinaattipisteitä, joita kohti robotti ajaisi. Tämä toteutettiin luomalla

metodi, jossa lasketaan vektoriprojektioiden avulla etäisyys sekä kulma kameran ja seinän välille. Tasopintojen kanssa oli myös keksittävä, miten seinien tasopinnat saadaan erotelluiksi muista ympäristöstä löytyvistä tasoista. Tässä kuitenkin päädyttiin aluksi vain rajaamaan käytettäville tasoille pienin koko, jonka ylittävät tasot luokiteltiin seiniksi. Yksi testiajoissa ilmennyt ongelma liittyi Kinectin ajureista johtuvaan ”valetasoon”, jonka ohjelma luokitteli seinäksi. Taso muodostui kameran syvyysakselin negatiiviselle puolelle, minkä ei pitäisi olla mahdollista. Tämä oli kuitenkin helppo suodattaa pois käyttämällä laskennassa vain positiivisia syvyysarvoja. (Kuva 17.)

```
while (icoarse.hasNext())
{
    if(icoarse.next().plane_size > m_wallsize)
    {
        if(icoarse.peekPrevious().z > 0)
        {
            wall = icoarse.peekPrevious();
            m_noWall = false;
        }
    }
}
```

KUVA 17. Tasopintojen erottelu

Etäisyys ja kulma lasketaan käyttämällä pistetuloa vektoreiden välillä. Vektoreiden väliset kulmat on helppo laskea käytettäessä normeerattuja vektoreita. Tällöin niiden välinen kulma saadaan suoraan ottamalla käänteinen kosini niiden pistetulosta. Tästä vähentämällä 90° (π/2) saadaan 0°:n kulma robotin ja seinän ollessa yhdensuuntaisia. (Kuva 18; kuva 19.)



KUVA 18. Seinänseurannan laskuissa käytetyt muuttujat

```
qreal dotproduct = wallnormal.dot(cameravector);
qreal angle = qAcos(dotproduct);
m_wallAngleInRadians = angle - M_PI/2;

m_wallAngle = (180/M_PI*m_wallAngleInRadians) * -1;
```

KUVA 19. Seinän ja robotin välisen kulman laskeminen

Etäisyys seinästä lasketaan myös pistetulon avulla projisoimalla kameran ja tason keskipisteen välinen vektori tasosta lähtevälle pintanormaalille (kuva 20).

```
Vector3d v = origin - wallcenter;
qreal d = v.dot(wallnormal);

m_wallDistance = d;
```

KUVA 20. Seinän ja robotin välisen etäisyyden laskeminen

Kun kameran etäisyys seinästä tiedetään, lisätään parametrit, jotka määrittävät, kuinka lähellä seinää robotin halutaan ajavan. Tästä saadaan xy-koordinaatit seinän mukaisessa koordinaatistossa, minkä jälkeen tehdään koordinaatistomuunnos robotille käyttäen vektoriprojektioista saatua kulmaa. Riippuen tilanteesta ohjausohjelman logiikka päättää, käytetäänkö maamerkeiltä vai tasopinnoilta saatuja koordinaatteja ohjaukseen.

4.3.3 Törmäyksenesto

Konenäön tunnistusprosessi on navigointikokonaisuuden eniten laskenta-aikaa vievä osa, joten uutta tietoa maamerkeistä ja tasopinnoista ei saada niin nopeasti kuin navigoinnin alin ohjauskerros edellyttäisi. Ohjausprosessin koordinaattipisteiden laskenta tapahtuu kuitenkin 600 ms:n välein, joten tunnistusprosessilta saatava tieto kerkeää päivittymään. Törmäyksenestoa varten haluttiin kuitenkin tieto havaituista esteistä suoraan alimmalle ohjauskerrokselle. Tähän olisi mielellään käytetty Kinectiltä saatavia syvyysarvoja. Ajurit olivat kuitenkin varattuina konenäköprosessien käyttöön. Ratkaisuksi valikoitui lukea pistepilvitietoa Kinectia käyttävän prosessin julkaisemalta topicilta. Robotin ohjausprosessi siirretään ROS:n alaisuuteen rekisteröimällä se nodeksi, minkä jälkeen tilataan pistepilviviestin sisältävä topic (kuva 21).

```
int argc = 0;
char **argv;
ros::init(argc, argv, "collision_detection");
m_collisionWarning = false;

if(ros::master::check())
{
    m_rosRunning = true;
    ros::NodeHandle n;
    m_sub = n.subscribe("/ObjectDetect_front/kinect1", 1, &CollisionDetection::onMessageReceived, this);
}
else
{
    m_rosRunning = false;
}
```

KUVA 21. Topicin tilaaminen konenäköprosessilta

Pistepilven sisältävä viesti muunnetaan helpommin käsiteltävään muotoon ja etsitään siitä lähimpänä kameraa oleva piste (kuva 22).

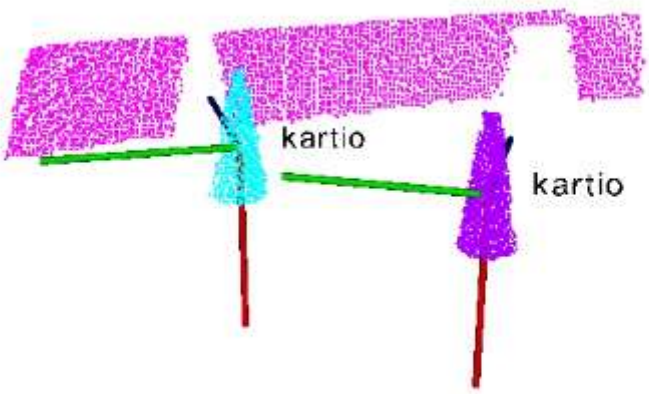
```
pcl::PCLPointCloud2 cloud;
pcl_conversions::toPCL(*msg_cloud, cloud);

pcl::PointCloud<pcl::PointXYZ>::Ptr cloudxyz(new pcl::PointCloud<pcl::PointXYZ>);
pcl::fromPCLPointCloud2(cloud, *cloudxyz);

qreal closestz = 0;
for (int i = 0; i < cloudxyz->points.size(); i++)
{

    if ((closestz == 0 || (closestz > cloudxyz->points[i].z)) && cloudxyz->points[i].z > 0)
    {
        closestz = cloudxyz->points[i].z;
    }
}
```

KUVA 22. Pistepilvitiedon muuntaminen ja lähimmän pisteen etsiminen

Törmäyksenestosta saatiin tällä tavalla nopeammin reagoiva kuin jos oltaisiin käytetty tunnistusprosessin jälkeistä tietoa. Ratkaisu on kuitenkin ajateltu väliaikaiseksi ja myöhemmin toteutettavissa muunlaisen anturitiedon avulla. Esimerkiksi Kinectin tunnistusalue alkaa vasta noin puolen metrin päästä, minkä takia törmäyksenesto ei toimi esteen sijaitessa tätä lähempänä kameraa. Vaikkakin reaktioaika on nyt nopeampi, kuin se olisi ollut tunnistusprosessilta saatavan tiedon pohjalta, sitä ei voida silti pitää tarpeeksi luotettavana turvatoiminnoksi ajateltuna.

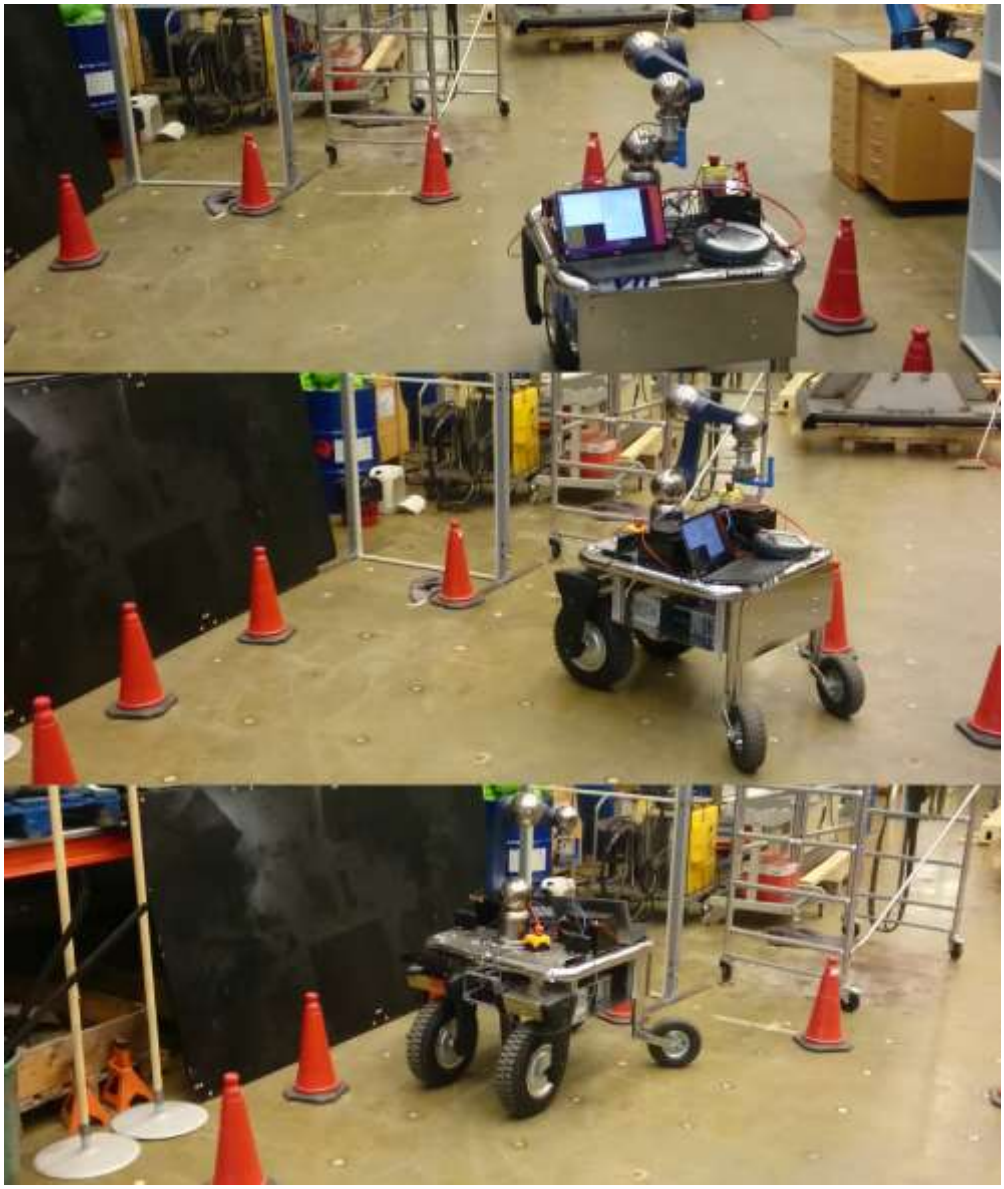
4.4 Testaus ja tulokset

Testejä tehtiin useita työn edetessä. Tällä varmistuttiin edellisten osien toiminnasta ennen uusien toimintojen lisäämistä. Konenäön parametreja jouduttiin virittämään aluksi muutamaan kertaan, jotta maamerkit saatiin tunnistettua hyvällä luotettavuudella. Kameroiden paikkaa muutettaessa tuli lattian tasopinta aina opettaa uudelleen konenäkösovellukselle. Maamerkkien navigoinnissa joitain ongelmia tuotti kameroiden kapea näkökenttä. Tätä korjattiin kääntämällä etukameraa kuvaamaan hieman etuviistoon, että robotti ei hukkaisi kartioita ajettaessa suorempia osuuksia. Tästä johtuen U-käännöksien kohdalla saatiin näkökentässä näkymään yleensä vain kaksi kartiota (kuva 23).



KUVA 23. Maamerkkien näkyvyys konenäkösovelluksessa

Kaarroksia tehtäessä ja kartioiden ollessa harvassa robotti saattoi huomata ne vasta viime hetkellä, jolloin korjausliikkeistä tuli voimakkaita. Kartiot voidaan tietysti asetella tiheämpään, mutta toisaalta ongelma ratkeaa myös laajemman näkökentän omaavalla kameralla. Robotti kuitenkin selvisi kaarroksista hyvin, kun niihin saavuttaessa tulokulma pidettiin loivana. Silmämääräisesti tarkasteltuna toteutuneen kaarroksen muoto vaikutti myös seuraavan aseteltujen kartioiden määrittämää puolikaarta. (Kuva 24.)



KUVA 24. U-käännös kartioiden mukaan

Seinänseurannassa ongelmia aiheutti Kinectilta saatu ”valetaso”. Ongelman paikallistamisen jälkeen se oli kuitenkin helppo suodattaa pois. Ohjelmaan eksyi sitä

kirjoittaessa virheitä, joita testien parissa jouduttiin etsimään ja korjaamaan käyttäen apuna Qt Creatorin debug-työkaluja. Yksi iso ongelma työn alusta asti oli robotin puoltaminen ja ongelmat sen kääntymisessä oikealle. Tämän pohjalta paljastui myöhemmin vika robotin elektroniikassa. Hieman toispuolinen ohjaus oli kuitenkin riittävä navigoinnin toteuttamiseksi ja testaamiseksi, kunhan sen vaikutukset otettiin ennalta huomioon. Ohjausta testattiin vielä kertaalleen, kun vika oli saatu korjattua, ja navigoinnin todettiin toimivan suunnitellusti.

Törmäykseneston todettiin myös toimivan ja reagoivan nopeasti eri tilanteissa, kun robotti havaitsi jotain olevan liian lähellä. Kinectin ominaisuuksien takia siitä ei kuitenkaan ollut hyötyä alle puolen metrin etäisyyksillä. Törmäyksenestoon tulisi tulevaisuudessa käyttää omaa anturia ja tehdä ohjelmasta mahdollisimman laiteläheinen. Kinectin tapauksessa paras ratkaisu olisi unohtaa pistepilvet ja lukea syvyysarvoja suoraan sille tehtyjen ajurikirjastojen avulla.

Testeissä esille tulleita asioita olivat ohjaukseen liittyvät ongelmat, kun robottia ajettiin hitailla ns. ryömintänopeuksilla ja kuorman muuttuessa. Hitailla nopeuksilla robotti saattoi vastustaa kääntymistä jo pelkästään kasterirenkaiden asennon aiheuttaman vastuksen takia. Yksittäisten kaarroksien välille saattoi myös muodostua huomattavia eroja, vaikka käytettiin samoja ohjausarvoja. Tämä oli osaltaan odotettavissa kun robottia testattiin ilman minkäänlaista odometriaa. Tässä työssä toteutettua reaktiivista menetelmää käytettäessä nämä ongelmat vaikuttivat kuitenkin liittyvän enemmän robotin hienosäätöön. Karkea ratapohjainen ohjaus tuntui toimivan ja robottia onnistuttiin ajamaan paikasta toiseen suuripiirteisellä tarkkuudella. Robottia ei suoranaisesti ole suunniteltu sisätiloissa tapahtuvaan automaattiajoon, joten tarkkuutta vaadittaessa, se miten moottoreita alemmalla tasolla ohjataan tulisi selvittää ja suunnitella käyttötarkoituksen mukaiset säätömenetelmät.

5 YHTEENVETO

Työn tavoitteena oli toteuttaa Probot Oy:n valmistamalle mobiilirobottialustalle navigointimenetelmä, jonka avulla robotti pystyy seuraamaan kartioiden ja seinien määrittämää rataa. Aikatauluna oli saada navigointiohjelma toimimaan ja dokumentoitua kevään 2017 aikana.

Työ aloitettiin rajaamalla navigointimenetelmältä vaaditut toiminnot. Tämän pohjalta aloitettiin suunnittelu siitä, miten yksinkertaisimmillaan menetelmä olisi mahdollista toteuttaa. Ohjelmointi jakaantui konenäön liittämisen, ohjausrakenteen ja navigointialgoritmien kesken. Lopuksi tehdystä työstä laadittiin signaalikaaviot, luokkakuvaukset ja toimintaohje.

Työ tarjosi sen tekijälle kiinnostavan aiheen ja mahdollisuuden perehtyä robotiikkaan ja siihen liittyviin kehitystyökaluihin sekä ideoihin. Työtä tehdessä ilmeni ongelmia niin konenäön kuin mobiilirobottialustan elektroniikan kanssa, joiden parissa tehty vianetsintä osaltaan auttoi aiheen laajempaa ymmärtämistä. Ohjelmistoista Qt ja ROS eivät myöskään olleet ennestään tuttuja, ja niiden opettelu tapahtui myös työtä tehdessä. Varsinkin työn alkumetrit herättivät paljon kysymyksiä, joihin kuitenkin alkoi löytyä vastauksia kirjallisuuden ja VTT:n työntekijöiden asiantuntijuuden avulla.

Lopputuloksena saatiin liite 1:n mukainen Qt:n ominaisuuksia hyödyntävä ohjelma, joka pystyy vastaanottamaan tietoa konenäkösovellukselta ja tämän perusteella laskemaan xy-koordinaatit sen tunnistamista maamerkeistä ja seinäpinnoista. Itse ohjaus koordinaatteja kohti tapahtuu suhdesäätöä käyttäen. Robotti saatiin navigoimaan seinien ja kartioiden avulla määriteltyjä ratoja pitkin. Testien perusteella robotti ajoi kaarrokset ilman suuria korjausliikkeitä, kunhan kartiot olivat aseteltuina riittävän tiheään. Ohjelmasta löytyy myös aiempana opinnäytetyönä tehdyn koordinaattiohjauksen sisältävä luokka, joka on mahdollista ottaa käyttöön myöhemmin. Tätä varten olisi kuitenkin oleellista muuttaa koordinaattiohjausta käyttämään odometriaa ajoaikoihin perustuvan laskennan sijasta. Ohjelmassa osaltaan jatkettiin aiempaa robotin parissa tehtyä työtä ja siihen lisättiin uutta toiminnallisuutta sekä luotiin pohjaa, jolle jatkokehitys voi tapahtua.

LÄHTEET

1. Qt History. 2016. Qt wiki. Saatavissa: https://wiki.qt.io/Qt_History. Hakupäivä 22.4.2017.

2. NASDAQ Helsinki Welcomes Qt Group Plc. 2016. The Qt Company. Saatavissa: https://www.qt.io/qt-news/nasdaq-helsinki-welcomes-qt-group-plc. Hakupäivä 22.4.2017.

3. Language Bindings. 2017. Qt Documentation. Saatavissa: http://wiki.qt.io/Language_Bindings. Hakupäivä 23.4.2017.

4. The Meta-Object System. 2017. Qt Documentation. Saatavissa: https://doc.qt.io/qt-5/metaobjects.html. Hakupäivä 23.4.2017.

5. Signals & Slots. 2017. Qt Documentation. Saatavissa: https://doc.qt.io/qt-5/signalsandslots.html. Hakupäivä 23.4.2017.

6. Threads and QObjects. 2017. Qt Documentation. Saatavissa: http://doc.qt.io/qt-4.8/threads-qobject.html. Hakupäivä 23.4.2017.

7. Chetto, Maryline 2014. Real-Time Systems Scheduling 1: Fundamentals. Networks And Telecommunications series. London – Hoboken, New Jersey: ISTE Ltd – John Wiley & Sons, Inc.. Saatavissa: https://ebookcentral.proquest.com (Oulun ammattikorkeakoulun opiskelijatunnuksilla).

8. Williams, Rob 2006. Real-Time Systems Development. Oxford: Butterworth-Heinemann. Saatavissa: https://ebookcentral.proquest.com (Oulun ammattikorkeakoulun opiskelijatunnuksilla).

9. Beal, David 2005. Linux As a Real-Time Operating System. Freescale Semiconductor, Inc. Saatavissa: http://www.nxp.com/assets/documents/data/en/white-papers/CWLNXRTOSWP.pdf. Hakupäivä 24.4.2017.

10. History. 2017. ROS.org. Saatavissa: http://www.ros.org/history. Hakupäivä: 25.4.2017.

11. Gerkey, Brian 2017. Why ROS 2.0? Open Source Robotics Foundation, Inc. Saatavissa: http://design.ros2.org/articles/why_ros2.html. Hakupäivä 25.4.2017.

12. Supported Platforms. 2017. Qt Documentation. Saatavissa: http://doc.qt.io/qt-5/supported-platforms.html. Hakupäivä: 25.4.2017.

13. Getting Started: Concepts. 2017. ROS.org. Saatavissa: http://wiki.ros.org/ROS/Concepts. Hakupäivä 25.4.2017.

14. ROS Filesystem Concepts: Packages. 2017. ROS.org. Saatavissa: http://wiki.ros.org/Packages. Hakupäivä 25.4.2017.

15. OpenKinect/libfreenect/README.md. 2017. GitHub. Saatavissa: https://github.com/OpenKinect/libfreenect/blob/master/README.md. Hakupäivä 26.4.2017.

16. Kinect for Windows Sensor Components and Specifications. 2017. Microsoft: Developer Network. Saatavissa: https://msdn.microsoft.com/en-us/library/jj131033.aspx. Hakupäivä 26.4.2017.

17. How Does ESP work? Bosch Mobility Solutions. Saatavissa: http://products.bosch-mobility-solutions.com/en/de/specials/specials_safety/bosch_esp_3/esp__facts_4/esp_technik_2/esp_questions_and_answers_16.html. Hakupäivä 26.4.2017.

18. Point Cloud Library. 2015. Willow Garage. Saatavissa: https://www.willowgarage.com/pages/software/pcl. Hakupäivä 26.4.2017.

19. Holland, John 2004. Designing Autonomous Mobile Robots. Inside the Mind of an Intelligent Machine. Oxford: Newnes. Saatavissa: https://ebookcentral.proquest.com (Oulun ammattikorkeakoulun opiskelijatunnuksilla).

20. DeSouza, Guilherme – Kak, Avinash 2002. Vision for Mobile Robot Navigation: A Survey. IEEE Transactions on Pattern Analysis and Machine Intelligence 24 (2002) 237–267. Saatavissa: IEEE Xplore (vaatii käyttöoikeuden).

21. Scholz, Jonathan – Jindal, Nehchal – Levihn, Martin – Isbell, Charles – Christensen, Henrik 2016. Navigation Among Movable Obstacles With Learned Dynamic Constraints. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Saatavissa: IEEE Xplore (vaatii käyttöoikeuden).

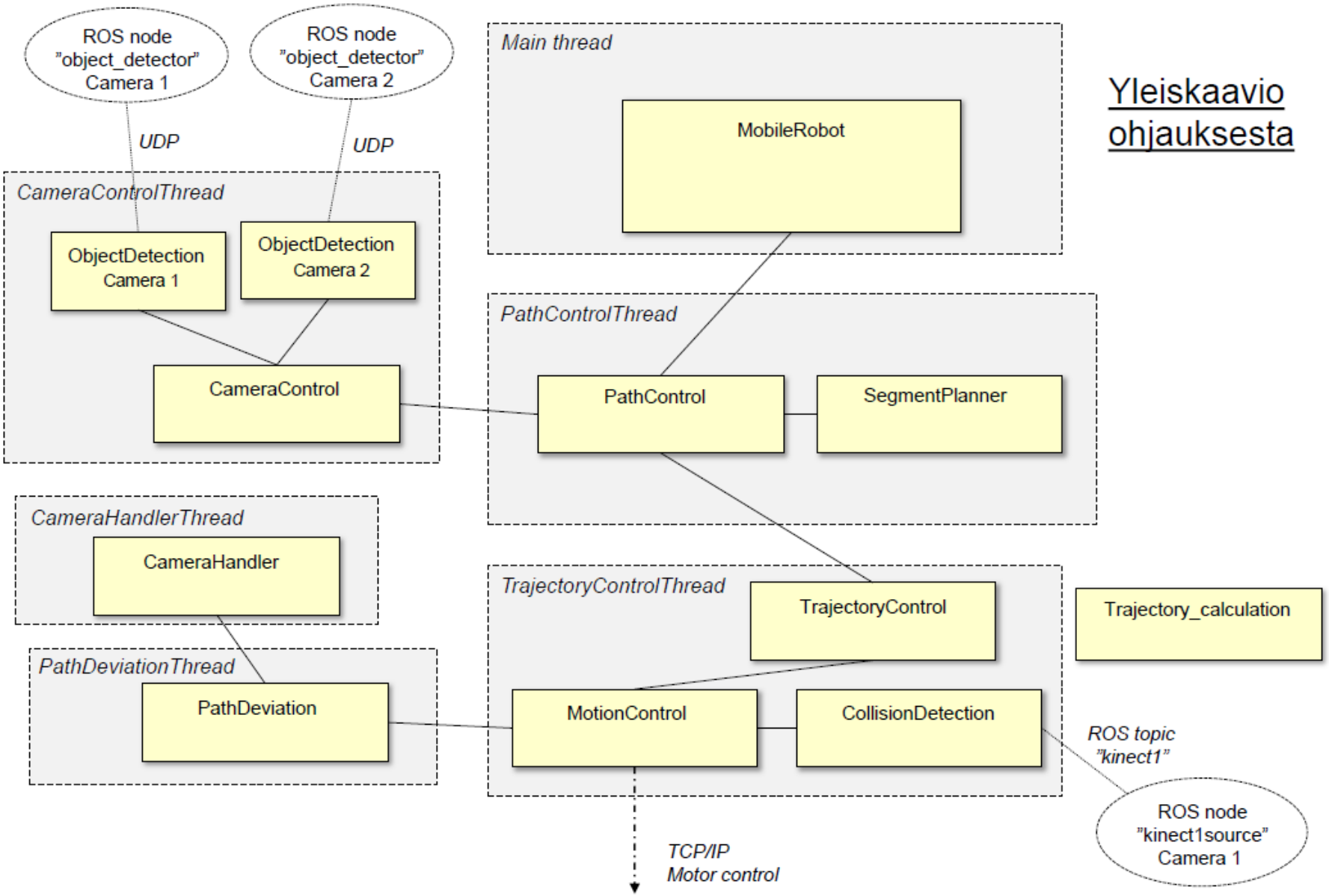
22. Huang, Wesley – Rajen, Brett – Fink, Jonathan – Warren, William 2006. Visual Navigation and Obstacle Avoidance Using a Steering Potential Function. Robotics and Autonomous Systems 54 (2006) 288–299. Saatavissa: http://www.sciencedirect.com/science/article/pii/S0921889005001995. Hakupäivä 15.5.2017.

23. Berkvens, Rafael – Vandermeulen, Dries – Vercauteren, Charles – Peremans, Herbert – Weyn, Maarten 2014. Feasibility of Geomagnetic Localization and Geomagnetic RatSLAM. International Journal on Advances in Systems and Measurements 7 nro 1 & 2 (2014) 44–56. Saatavissa: http://www.iariajournals.org/systems_and_measurements. Hakupäivä 15.5.2017.

24. Life with Kuri. 2017. Mayfield Robotics. Saatavissa: https://www.heykuri.com/living-with-a-personal-robot. Hakupäivä 29.4.2017.

25. The Roomba Now Sees and Maps a Home. 2015. MIT Technology Review. Saatavissa: https://www.technologyreview.com/s/541326/the-roomba-now-sees-and-maps-a-home. Hakupäivä 29.4.2017.

26. Corke, Peter 2011. Robotics, Vision and Control: Fundamental Algorithms in MATLAB. Springer Tracts in Advanced Robotics. Berlin Heidelberg: Springer-Verlag.


ROS node
"object_detector"
Camera 1
ROS node
"object_detector"
Camera 2
UDP
UDP
CameraControlThread
ObjectDetection
Camera 1
ObjectDetection
Camera 2
CameraControl
Main thread
MobileRobot
PathControlThread
PathControl
SegmentPlanner
CameraHandlerThread
CameraHandler
PathDeviationThread
PathDeviation
TrajectoryControlThread
TrajectoryControl
Trajectory_calculation
MotionControl
CollisionDetection
ROS topic
"kinect1"
ROS node
"kinect1source"
Camera 1
TCP/IP
Motor control
Yleiskaavio
ohjauksesta