Tampereen yliopisto

Santeri Kangas

ROBOTTIAJONEUVORYHMÄN REITITYKSEN SIMULOINTI

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja: TkT. Juhana Helovuo
Huhtikuu 2024

TIIVISTELMÄ

Santeri Kangas: Robottiajoneuvoryhmän reitityksen simulointi
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Huhtikuu 2024

Robottiajoneuvot ovat itseohjautuvia ajoneuvoja. Toimintaympäristössä liikkuu yleensä useita robottiajoneuvoja, jolloin puhutaan robottiajoneuvoryhmästä. Keskeiseksi ongelmaksi muodostuukin, miten kaikki ryhmän robottiajoneuvot saadaan reititettyä ajotehtävilleen samaan aikaan. Robottiajoneuvojen reitinsuunnittelussa tarvitaankin reititysalgoritmeja. Tässä työssä suunnitellaan ja toteutetaan simulointialusta, jolla robottiajoneuvoryhmän reititystä voidaan simuloida ja käyttää myöhemmin kehitettävien reititysalgoritmien testaamisessa.

Työ on luonteeltaan soveltuvuusselvitys ja työssä on käytetty konstruktiivista menetelmää. Tutkimusongelmana on, voidaanko simulointialusta toteuttaa käyttäen teknologioita Rust, ROS (engl. Robot Operating System) ja RViz-visualisointityökalu. Työn tavoitteena on selvittää, miten reititystä ja robottiajoneuvoja on mallinnettava simulointialustan toteuttamiseksi.

Robottiajoneuvojen tyypillisimpiä käyttökohteita ovat varasto- ja teollisuushallien sisäiset kuljetustehtävät. Robottiajoneuvot ovat keskeisessä asemassa osana tuotantojärjestelmää ja toimitusketjua, mistä aiheutuu tarve optimoida kuljetustehtävien suoritusta. Tätä varten on kehitetty yhä monimutkaisempia reititysalgoritmeja, jotka optimoivat jotakin tekijää kuten lyhintä reittiä. Optimointiongelmasta riippumatta reititysalgoritmien keskeisenä tehtävänä on konfliktittoman reitin tuottaminen kaikille robottiajoneuvoille samaan aikaan. Reitinsuunnittelu huolehtii siis, etteivät robottiajoneuvot törmää toisiinsa tai jää jumiin lukkiutumistilanteisiin.

Simulointi on osoittautunut hyväksi keinoksi testata reititysalgoritmien toimivuutta. Simuloinnin toteuttamisessa korostuvat erityisesti oikean mallin valinta, mallinnuksen tarkkuus ja visualisointi. Visualisoinnilla on erityinen rooli simuloidun mallin ja tulosten paremmassa kommunikoinnissa. Robottiajoneuvojen tapauksessa täytyy mallintaa sekä ajoneuvot että niiden toimintaympäristö. Reitityksen simuloinnissa ajoneuvojen ja toimintaympäristön yksityiskohtainen mallintaminen ei ole tarpeellista.

Simulointialusta toteutetaan Rust-ohjelmointikielellä ja visualisointiin käytetään ROSia ja RViziä. Ajokarttojen piirtämiseen käytetään Tiled-tasoeditoria. Keskeisenä vaatimuksena on mahdollistaa reititysalgoritmien kytkeminen simulointialustaan siten, että tuotetut reittisuunnitelmat voidaan tarkastaa mahdollisten ongelmien varalta ja hylätä tarvittaessa ennen simulointia. Simulointialustan täytyy myös tuottaa varoitukset reittisuunnitelmien ongelmatilanteista.

Robottiajoneuvoille toteutetaan yksinkertainen 3D-robottimalli, joka koostuu rungosta ja takapuskurista. Ajokarttojen piirtoa varten toteutetaan segmenttimuodot: suora, mutka ja S-mutka. Simuloinnin suoritus kuvataan skenaarioina, jotka koostuvat ajokartasta, ajoneuvojen määrittelystä ja ajotehtävälistasta. Simulointialustan ja reitinsuunnittelun väliin toteutetaan rajapinta, jonka kautta uusia reittisuunnitelmia pyydetään reititysalgoritmilta. Reittisuunnitelmien törmäystilanteisiin toteutetaan myös visuaaliset varoitukset. Liikkeen simulointia varten robottiajoneuvoille toteutetaan kiihdytys- ja hidastusvaiheet, etenemissuunnan vaihto ja paikallaan kääntyminen. Visualisointia varten toteutetaan komponentit, jotka kykenevät julkaisemaan visualisointiviestejä ROSin kautta RVizille.

Tutkimus osoittaa, että valituilla teknologioilla on mahdollista toteuttaa vaatimusten mukainen simulointialusta. Simulointialusta on kuitenkin osin yksinkertaistettu ja jatkokehitettäviä kohteita ovat nykyisen mallinnuksen hienontaminen, uusien parempien mallien toteuttaminen ja visualisoinnin parantaminen.

Avainsanat: robottiajoneuvot, reitinsuunnittelu, reititys, simulointi, ROS

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

ABSTRACT

Santeri Kangas: Routing Simulation of a Robot Vehicle Fleet
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
April 2024

Robot vehicles are self-driving vehicles. There are typically several robot vehicles in an operating environment, in which case the term robot vehicle fleet can be used. The main problem is how all the robot vehicles in a fleet can be routed at the same time. Thus, the route planning of the robot vehicles requires routing algorithms. In this thesis, a simulation software is designed and implemented for simulating the routing of a robot vehicle fleet and testing later developed routing algorithms.

The thesis is a proof of concept, and the constructive method is used. The research question is if technologies Rust, ROS (Robot Operating System), and RViz visualizer can be used to build the simulation software. The goal of the thesis is to study how the routing and robot vehicles must be modelled for the simulation.

The most typical operating environments for the robot vehicles are warehouses and factories where they perform internal transportation tasks. Robot vehicles play significant roles in manufacturing systems and supply chains, which causes the need to optimize the performance of the transportation tasks. This has led to the implementation of increasingly complex routing algorithms that optimize some specified factor like shortest path. Regardless of the optimization problem, the main task for the routing algorithms is to produce conflict-free paths for every robot vehicle in the fleet at the same time. Route planning is thus responsible for providing routes so that the robot vehicles do not hit each other or get involved in gridlocks.

Simulation has proven to be an effective way to evaluate how the routing algorithms perform. Implementing simulation is characterized by selecting a correct model, determining the accuracy of the modelling, and building visualization. Visualization is important especially in communicating the simulated model and the results of the simulation. Simulation of robot vehicles requires modelling both the vehicles and the operating environment. Detailed modelling of the robot vehicles and operating environment is not necessary when the routing is simulated.

The simulation software is implemented with Rust programming language, and for the visualization ROS and RViz are used. Tiled level editor is used to drawing the driving maps. The main goal is to provide a way for the routing algorithms to connect to the simulation software so that the produced route plans can be checked for issues and declined before simulating them. The simulation software must also produce warnings if there are issues with the route plans.

A simple 3D robot model is implemented for the robot vehicles consisting of a frame and rear bumper. For the driving maps the following segment types are implemented: straight, curve, and S-curve. The simulations are described as scenarios which consist of a driving map, defined vehicles, and driving tasks. An interface between the simulation software and route planning is developed through which new route plans can be requested during simulation. Visual warnings are implemented for collisions in the route plans. Basic motion simulation for the robot vehicles is implemented consisting of acceleration and deceleration phases, changing the progress direction, and turning in place. For visualization, software components that can publish visualization messages through ROS to RViz are implemented.

The thesis demonstrates that the simulation software can be implemented with the given technologies according to the requirements. However, the simulation software is partially simplified, and further development includes improving the modelling and visualization.

Keywords: robot vehicles, route planning, routing, simulation, ROS

The originality of this thesis has been checked using the Turnitin Originality Check service.

ALKUSANAT

Tämä diplomityö on tehty yritykselle Atostek Oy syksyn 2023 ja kevään 2024 aikana.

Haluan kiittää Hannu-Matti Järvistä ja Juhana Helovuota työni ohjaamisesta ja kattavasta palautteesta. Erityiskiitokset Atostek Oy:lle diplomityöpaikasta ja Juhanalle toteutuksen ohjaamisesta ennen varsinaista kirjoitustyötä. Lopuksi kiitän vielä ystäviäni ja perhettäni työhön kannustamisesta ja vertaispalautteesta.

Tampereella, 24.4.2024

Santeri Kangas

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. TAUSTAA 2
2.1 Historiaa 2
2.2 Robottiajoneuvot 4
2.3 Reitinsuunnittelu ja sen aikataulutus 7
2.4 Reititysongelma ja reitinsuunnittelumallit 8
2.5 Reitinsuunnittelun haasteet 9
3. SIMULOINTI 13
3.1 Simulointi ja mallinnus 13
3.2 Tietokonesimulointi 14
3.3 Visualisoinnin tärkeys 15
3.4 Robottiajoneuvojen simulointi 16
3.5 Ajokartat simuloinnissa 17
4. TYÖKALUT 18
4.1 Rust-ohjelmointikieli 18
4.2 Robot Operating System 21
4.3 RViz 24
4.4 Ajokarttojen piirtotyökalu 27
5. SIMULOINTIALUSTAN MÄÄRITTELY 29
5.1 Vaatimukset ja tavoitteet 29
5.2 Rajapinnat ja arkkitehtuuri 31
5.3 Yleinen toiminta 32
6. SIMULOINTIALUSTAN TOTEUTUS 34
6.1 Ajoneuvojen mallinnus 34
6.2 Ajokartat ja piirto 37
6.3 Simulointiskenaariot 41
6.4 Reittisuunnitelmat ja reitinsuunnittelu 43
6.5 Liikkeen simulointi 50
6.6 Visualisointi 62
6.7 Jatkokehitys 66
7. YHTEENVETO 68
LÄHTEET 70

KUVALUETTELO

Kuva 1. *1950-luvun Barrett-Cravensin rakentama "traktori" oli yksi ensimmäisiä robottiajoneuvoja. [1]* ... 3
Kuva 2. *Kontteja kuljettavia robottiajoneuvoja Rotterdamin satama-alueella.* ... 4
Kuva 3. *Starship Technologies -yrityksen valmistama pieni kuljetusrobotti.* ... 5
Kuva 4. *Reitinsuunnittelu keskeisimmät osat.* ... 7
Kuva 5. *Reititysongelmat voidaan esittää graafina.* ... 8
Kuva 6. *Esimerkki kolmen ajoneuvon lukkiutumistilanteesta risteyksessä.* ... 9
Kuva 7. *Havainnekuva ajosegmenttien primääri- ja sekundäärivarauksista kahden ajoneuvon tapauksessa.* ... 10
Kuva 8. *Havainnekuva varastohallin asettelusta ja ajokartasta [16].* ... 17
Kuva 9. *Julkaisija-tilaaja-mallin toiminta ROS-solmujen välillä.* ... 21
Kuva 10. *Havainnekuva DDS:n ja RTPS-protokollan toteutuksesta IP-verkon päällä.* ... 23
Kuva 11. *RViz-visualisointityökalun yleisilme.* ... 24
Kuva 12. *Esimerkkirobotti visualisoituna RVizissä.* ... 26
Kuva 13. *Robottimallit ja niiden linkit kytkettynä yhteen yleiseen maailmakoordinaatistoon.* ... 26
Kuva 14. *Tiled-piirtotyökalun yleisilme.* ... 27
Kuva 15. *Korkean tason arkkitehtuurikuvaus simulointialustan toiminnasta.* ... 31
Kuva 16. *Simulaattorin keskeisimmät osat.* ... 32
Kuva 17. *Simuloinnin yleinen kulku korkealla tasolla.* ... 33
Kuva 18. *Esimerkki yksittäisen ajoneuvon visualisoinnista.* ... 36
Kuva 19. *Esimerkki kahden segmentin kytkeytymisestä ja ajosuunnasta.* ... 37
Kuva 20. *Esimerkki suorakaiderajauksesta kahden segmentin tapauksessa.* ... 38
Kuva 21. *Tiled-projektiin toteutetut segmenttimuodot.* ... 38
Kuva 22. *Esimerkki malliobjektista piirtotyökalussa.* ... 39
Kuva 23. *Ajokartan lukeminen JSON-tiedostosta korkealla tasolla.* ... 40
Kuva 24. *Yksittäisen ajokartan segmentin visualisointi.* ... 40
Kuva 25. *Esimerkki reittisuunnitelmien visualisoinnista.* ... 45
Kuva 26. *Törmäysvaroituksen visualisointi.* ... 49
Kuva 27. *Ajoneuvon edettävä matka saadaan ajettavaksi päätetyistä legeistä.* ... 50
Kuva 28. *Havainnekuva nopeusprofiilin maksiminopeustapauksesta.* ... 50
Kuva 29. *Havainnekuva nopeusprofiilin kiihdytys ja hidastus -tapauksesta.* ... 51
Kuva 30. *Nopeusprofiilien molemmat tapaukset merkintöineen.* ... 53
Kuva 31. *Havainnekuva nopeusprofiilin uudelleen määrittämisestä.* ... 55
Kuva 32. *Esimerkki nopeusprofiilista tietyllä simulaatiohetkellä.* ... 56
Kuva 33. *Esimerkki etenemisestä suoran segmentin pistejonossa.* ... 57
Kuva 34. *Uuden pisteen määrittäminen kahden tunnetun välille.* ... 57
Kuva 35. *Esimerkki etenemissuunnan vaihdosta.* ... 58
Kuva 36. *Havainnekuva segmenttimuotojen välisestä kulmasta yhteyspisteessä.* ... 59
Kuva 37. *Esimerkki tekstin ja murtoviivan visualisoinnista.* ... 64
Kuva 38. *Havainnekuva S-mutkasegmentin murtoviivaesityksestä.* ... 65
Kuva 39. *Esimerkki ajoneuvojen tunnisteiden ja simulaatioajan tekstimuotoisista visualisoinneista Marker-viestityypillä.* ... 65
Kuva 40. *Esimerkki suorakaiderajauksesta ja tarkemmasta arviosta.* ... 67

LYHENTEET JA MERKINNÄT

AGV	engl. Automated Guided Vehicle, robottiajoneuvo
AGVS	engl. Automated Guided Vehicle System, robottiajoneuvojen hallinnointijärjestelmä
AMR	engl. Autonomous Mobile Robot, itseohjautuva robottiajoneuvo
BFS	engl. Breadth-First Search, graafihakualgoritmi jonkin ominaisuuden täyttävän solmun löytämiseksi leveys ensin -periaatteella
CSV	engl. Comma-Separated Values, pilkuilla eroteltujen arvojen kuvausformaatti taulukkomuotoisen tiedon tallennukseen
DDS	engl. Data Distribution Service, avoin rajapintastandardi julkaisija-tilaaja-mallin mukaisen tiedonsiirron toteuttamiseksi
IDL	engl. Interface Description Language, ohjelmointikielestä tai arkkitehtuurialustasta riippumaton kuvauskieli, jolla kuvataan rajapinnat ja tietotyypit
JSON	engl. JavaScript Object Notation, avoimen standardin kuvausformaatti sekä tiedonvälitykseen että tallennukseen
MCFP	engl. Minimum-Cost Flow Problem, graafien virtausmääriin liittyvä optimointiongelma
QoS	engl. Quality of Service, palvelun laatupolitiikka
ROS	engl. Robot Operating System, robotiikan väliohjelmisto ja sovellusalusta
RTPS	engl. Real-Time Publish-Subscribe Protocol, DDS:n julkaisija-tilaaja-mallin toteuttamiseksi käyttämä tiedonsiirtoprotokolla
TCP	engl. Transmission Control Protocol, eräs tietoliikenneprotokolla tietokoneiden väliseen luotettavaan tiedonsiirtoon
UDP	engl. User Datagram Protocol, eräs yhteydetön tietoliikenneprotokolla
URDF	engl. Unified Robot Description Format, ROSin robottimallien kuvausformaatti
WLAN	engl. Wireless Local Area Network, langaton lähiverkko
XML	engl. Extensible Markup Language, merkintäkielien standardi

1. JOHDANTO

Robottiajoneuvot ovat erilaisia automatisoituja eli itseohjautuvia ajoneuvoja, jotka tyypillisesti suorittavat rutiininomaisia kuljetustehtäviä esimerkiksi teollisuus- ja varastohalleissa. Toimintaympäristössä liikkuu yleensä useita robottiajoneuvoja samaan aikaan, jolloin voidaan puhua *robottiajoneuvoryhmästä*. Ryhmän lisäksi toimintaympäristössä voi liikkua muitakin toimijoita kuten ihmisiä tai ihmisten ajamia ajoneuvoja. Keskeiseksi ongelmaksi muodostuukin, miten kaikki ryhmän robottiajoneuvot saadaan reititettyä ajotehtävilleen tehokkaasti ja törmäämättä muihin toimijoihin. Robottiajoneuvojen reitinsuunnittelussa käytetäänkin reititysalgoritmeja, jotka optimoivat joitakin tiettyjä tekijöitä kuten esimerkiksi ajotehtävän suoritusnopeutta tai lyhintä reittiä. Reititysalgoritmien toimivuutta täytyy voida testata. Simulointi on eräs keino testata ja arvioida reitinsuunnittelun onnistumista.

Tässä diplomityössä suunnitellaan ja toteutetaan simulointialusta, jolla robottiajoneuvoryhmän reititystä voidaan simuloida ja käyttää myöhemmin kehitettävien reititysalgoritmien testauksen apuvälineenä. Työ on luonteeltaan soveltuvuusselvitys ja työssä käytetään konstruktiivista menetelmää. Tutkimusongelmana on, voidaanko simulointialusta toteuttaa käyttäen teknologioita Rust, ROS (engl. Robot Operating System) ja RViz-visualisointityökalu. Työn tavoitteena on selvittää, miten reititystä ja robottiajoneuvoja on mallinnettava simulointialustan toteuttamiseksi.

Luvussa 2 esitellään keskeisin toimialan historia, robottiajoneuvot yleisesti ja reitinsuunnittelu. Luvussa 3 kuvataan simuloinnin teoria, robottiajoneuvojen simulointi ja ajokartat simuloinnissa. Työssä käytetyt työkalut ja niiden keskeisin sisältö esitellään luvussa 4. Luvussa 5 esitellään rakennettavan simulointialustan vaatimukset ja tavoitteet yhdessä toteutettujen rajapintojen kanssa. Luvussa 6 käydään läpi työn varsinainen toteutus, ratkaisut ja jatkokehitysideat. Lopuksi luvussa 7 esitetään yhteenveto tämän diplomityön keskeisimmästä sisällöstä.

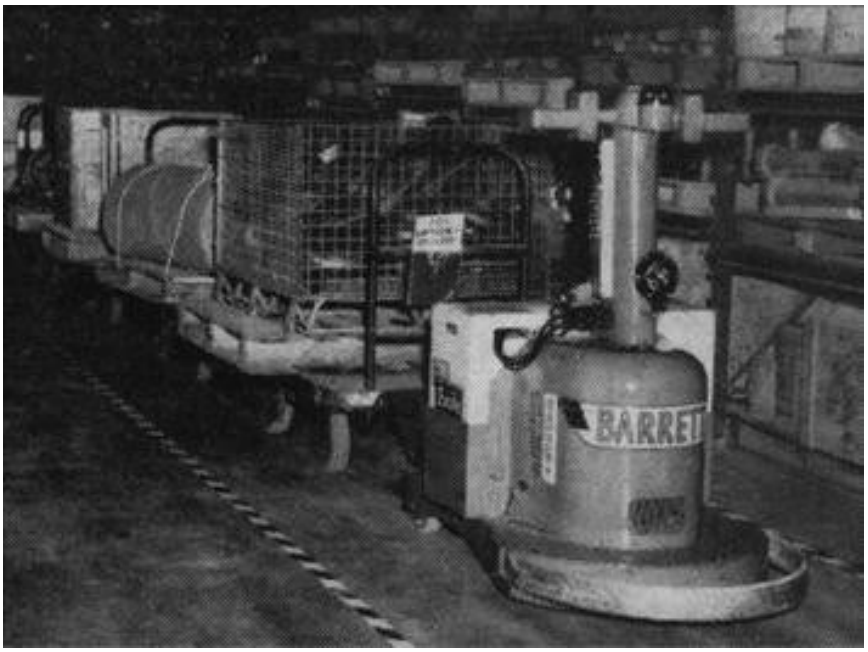
2. TAUSTAA

Robottiajoneuvojen kehitys on alkanut noin 1950-luvun alussa, kun ensimmäiset *AGV*-tyyppiset (engl. Automated Guided Vehicle) ajoneuvot esiteltiin kuljettamaan tavaroita teollisuus- ja varastohalleissa [1, s. 1–2]. Kehityksen taustalla oli pitkälti pyrkimys toistuvien työtehtävien automatisointiin. Ajoneuvojen automatisointi on kuitenkin nostanut esiin toisen keskeisen ongelmakohdan eli reitinsuunnittelun haasteet. Tässä luvussa esitellään keskeisin toimialan historia, robottiajoneuvot yleisesti sekä reitinsuunnittelu ja sen haasteet.

2.1 Historiaa

Robottiajoneuvojen kehitys voidaan jakaa kolmeen keskeisimpään aikakauteen. Ensimmäinen aikakausi käsitti varsinaisen idean kehittämisen ja ensimmäiset toteutukset. Toinen aikakausi keskittyi automaation ja robottiajoneuvojen idean jatkojalostamiseen eri teollisuuden aloille. Toisen aikakauden keskeinen saavutus oli robottiajoneuvojen yleinen kehitys, johon kuului esimerkiksi akkujen ja elektroniikan paraneminen, ajoneuvojen navigoinnin standardoituminen ja automaatioprosessien kehitys. Kolmas aikakausi puolestaan jatkoi toisen aikakauden aloittamaa kehitystä ja ajoneuvojen kontaktiton navigointi muodostui standardiksi. [1, s. 1–14]

Ensimmäinen aikakausi alkoi 1950-luvun alussa, kun amerikkalainen yritys Barrett-Cravens (nyk. Savant Automation Inc) kehitti ja valmisti ensimmäisenä perävaunujen vetämiseen ja tavaroiden kuljettamiseen tarkoitetun ajoneuvon. Kuvassa 1 on esitetty eräs ensimmäisistä Barrett-Cravensin rakentamista ajoneuvoista vetämässä perävaunuja. Kuvasta on havaittavissa ensimmäisille robottiajoneuvoille tyypillisiä ominaispiirteitä kuten lattiaan merkityt ajoradan reunaviivat ja ajoneuvon etupuskuri. Muita tyypillisiä piirteitä olivat erilaiset mekaaniset hätäkahvat. Ajoradan reunaviivoja käytettiin huomiotekijöinä, jotta ihmiset osasivat varoa toimintaympäristössä liikkuvia robottiajoneuvoja. Etupuskuri toimi samalla turvamekanismina, sillä törmäys tiellä olevaan esteeseen aiheutti ajoneuvon pysähtymisen. [1, s. 2, 4]



Kuva 1. *1950-luvun Barrett-Cravensin rakentama "traktori" oli yksi ensimmäisiä robottiajoneuvoja. [1]*

Ensimmäiset robottiajoneuvot olivat vielä melko kankeita ja vaativat fyysisiä muutoksia tehdas- ja varastohallien pohjaratkaisuihin. Teknologia ja tekniikka oli vasta kehittymässä ja ajoneuvot vaativat erikseen rakennetun ajoradan. Ensimmäiset ajoradat rakennettiin lattiakiskoista, joita pitkin ajoneuvot liikkuivat. Lattiakiskojen korvaajaksi tulivat myöhemmin sähköjohdot, jotka olivat kehitysaskel kohti kontaktitonta navigointia. Myöhemmin ajoneuvojärjestelmien kehittyessä ajoradan seuraamiseen rakennettiin myös optisia antureita, joiden avulla ajoneuvot pystyivät seuraamaan värillisiä lattianauhoja. 1960-luvulta lähtien robottiajoneuvoja alettiin käyttää yhä enemmän eri käyttötarkoituksiin ja sovelluskohteisiin. Uutena ajoneuvotyyppinä tulivat esimerkiksi automatisoidut trukit. [1, s. 2–6]

Toinen aikakausi alkoi noin 1960- ja 1970-lukujen taitteessa, jolloin kehitettiin ja parannettiin robottiajoneuvojen hallinnointiin erikoistuneita *AGVS*-järjestelmiä (engl. Automated Guided Vehicle System). Elektroniikan määrää lisättiin sekä ajoneuvoissa että toimintaympäristöjen ohjausjärjestelmissä. Kehitys mahdollisti entistä laajemmin erilaisten kuljetustehtävien automatisoinnin ja paransi samalla ajoneuvojen yleistä suorituskykyä. Ajoneuvot pystyivät yhä enemmän suoriutumaan itsenäisesti esimerkiksi lastaustehtävistä ilman ihmisen apua. Toinen aikakausi keskittyi myös joustavuuden ja ajoneuvojen luotettavuuden parantamiseen toimintaympäristöissä. Ajoneuvot pystyivät nyt myös lähettämään sijaintitietoaan hallintajärjestelmälle infrapuna- tai radiosignaaleilla pelkän kiinteän sähköjohdon sijaan. Toista aikakautta kuitenkin varjosti edelleen ajoneuvojen ja AGVS-järjestelmien kankeus. [1, s. 6–10]

Kolmas aikakausi 1990-luvulta alkaen keskittyikin edelleen järjestelmien kehittämiseen ja niiden joustavuuden parantamiseen. WLAN (engl. Wireless Local Area Network) eli langaton lähiverkko vakiintui ajoneuvojen ja hallintajärjestelmien välisessä tiedonsiirrossa ja navigointiin alettiin käyttää pitkälti laser- tai magneettipisteohjausta aiemmin mainitun sähköjohtopohjaisen navigoinnin lisäksi. Erilaisten antureiden ja tehokkaampien tietokonekomponenttien käyttäminen muussakin kuin navigoinnissa paransi robottiajoneuvojen tarkkuutta, tehokkuutta ja joustavuutta erilaisissa työtehtävissä. [1, s. 10–14]

2.2 Robottiajoneuvot

Nykypäivänä robottiajoneuvot hoitavat sisäisen logistiikan tehtäviä monessa eri paikassa. Perinteisten teollisuus- ja varastohallien lisäksi keskeisiä kohteita ovat esimerkiksi satamat ja sairaalat. Toimintaympäristöjen laajan kirjon takia robottiajoneuvoja löytyy useassa eri koossa pienistä kuljetusroboteista isompiin ja raskaampiin ajoneuvoihin. Robottiajoneuvot ovat tärkeä osa tuotantoketjua ja siten koko toimintaorganisaatiota. [1, s. 15–18] Kuvassa 2 on esitetty esimerkki kontteja kuljettavista robottiajoneuvoista Rotterdamin satama-alueella.

Kuva 2. *Kontteja kuljettavia robottiajoneuvoja Rotterdamin satama-alueella.*

Robottiajoneuvot kattavat perinteisemmän AGV:n lisäksi *AMR*-tyyppiset (engl. Autonomous Mobile Robot) autonomiset eli itseohjautuvat robotit. Raja näiden ajoneuvotyyppien välillä vedetään kirjallisuudessa siihen, että autonominen robotti kykenee esimerkiksi kiertämään erilaisia esteitä tai määrittämään ja optimoimaan itse omaa kulkureittiään, kun taas AGV vaatii ennalta määrättyä reititystä. Toisaalta itseohjautuvuus määritellään myös niin, ettei ajoneuvo välttämättä vaadi ennalta määrättyä erillistä rataa tai muuta tietoa liikkumisympäristöstään. Liikkumiseen tarvittava tieto saadaan käyttämällä monipuolisesti erilaisia ympäristön havainnointiin erikoistuneita antureita. Molemmissa robottiajoneuvotyypeissä on kuitenkin nykypäivänä samoja ominaisuuksia ja teknologiaa, minkä takia raja ajoneuvotyyppien välillä on häilyvä. [2, s. 1–3] Hyvä esimerkki itseohjautuvista roboteista ovat erityisesti viime aikoina yleistyneet kauppaostosten kuljetuksiin erikoistuneet pikkuajoneuvot. Kuvassa 3 on esitetty eräs tällainen Starship Technologies -yrityksen valmistama kuljetusrobotti.

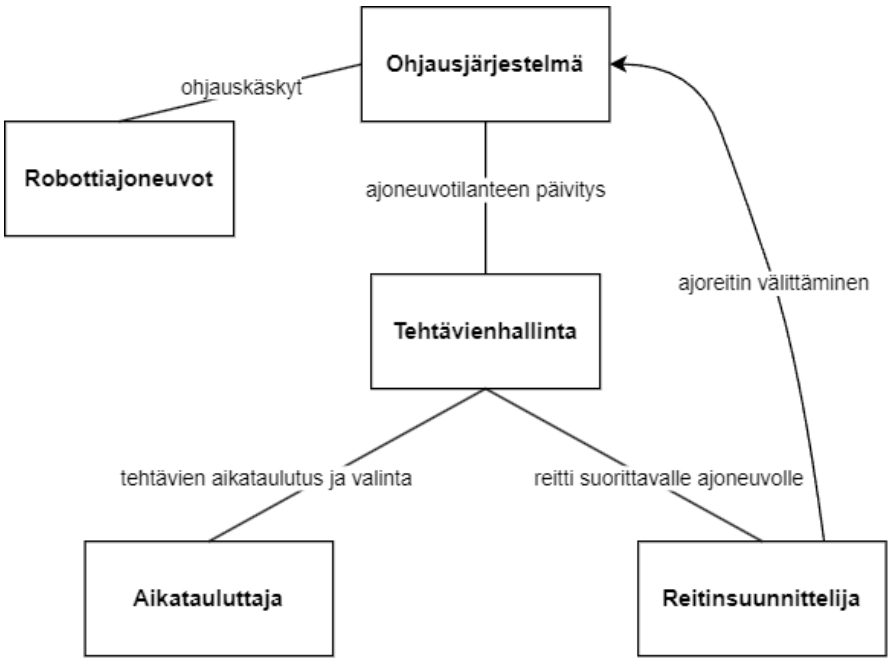
Kuva 3. *Starship Technologies -yrityksen valmistama pieni kuljetusrobotti.*

Robottiajoneuvojen ohjelmisto koostuu karkeasti seuraavista alijärjestelmistä: navigointi, liike-, kommunikointi- ja turvajärjestelmät. Näiden lisäksi ajoneuvoilla on tyypillisesti omaan erikoistumiseensa liittyen muita järjestelmiä, kuten esimerkiksi robottitrukilla haarukan liikuttaminen. Muita yleisiä järjestelmiä ovat akunhallinta ja virransyöttö, jotka toimivat yhteistyössä muiden järjestelmien kanssa. Myös navigointi-, liike- ja turvajärjestelmät toimivat vahvasti yhteistyössä toistensa kanssa. [1, s. 142–145]

Tässä diplomityössä robottiajoneuvot rajataan niin, että niiden kulkua toimintaympäristössä halutaan hallita ja ajoneuvojen reitit ovat etukäteen määritettyjä. Ajoneuvot eivät siis tee omia ratkaisuja tai optimointeja reitteihinsä, vaan ajettavat reitit tuotetaan keskitetysti reitinsuunnittelijan toimesta. Toteutetun simulointialustan kannalta ei kuitenkaan ole suoraa rajoitetta robottiajoneuvojen tyypille.

2.3 Reitinsuunnittelu ja sen aikataulutus

Reitinsuunnittelu on prosessi, jossa suoritettavat ajotehtävät aikataulutetaan ja reititetään. Prosessiin kuuluu kuitenkin olennaisena osana myös ajotehtäviä suorittavien ajoneuvojen valinta ja reittisuunnitelmien välittäminen ajoneuvojen ajettavaksi. Keskeisimmät osat reitinsuunnittelussa ovat *ohjausjärjestelmä*, *tehtävienhallinta*, *aikatauluttaja* ja *reitinsuunnittelija*, jotka on esitetty kuvassa 4. [1, s.125–128]

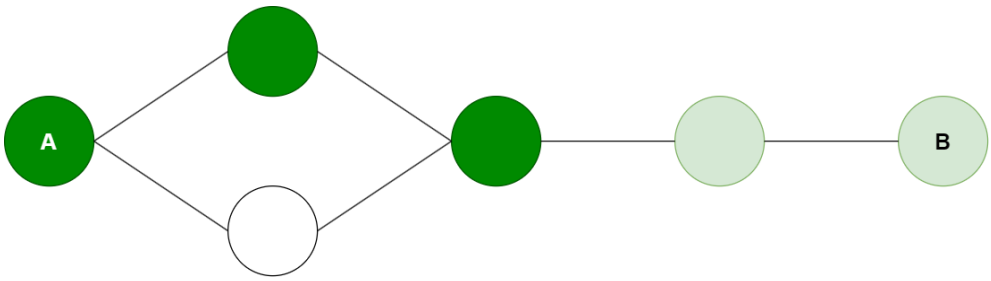


Kuva 4. *Reitinsuunnittelu keskeisimmät osat.*

Kuvassa ylimpänä on ohjausjärjestelmä, jonka tehtävänä on kommunikoida ja antaa ohjauskäskyjä yksittäisille robottiajoneuvoille. Ohjausjärjestelmällä on tiedossa kaikkien toimintaympäristön ajoneuvojen tilanne kuten sijainti ja akkujen varaustaso. Ohjausjärjestelmällä on myös tieto suoritettavista ajotehtävistä ja niiden tilanteesta, jotka se välittää eteenpäin tehtävienhallintaan. Ajotehtävät ovat joko nouto- tai toimitustehtäviä. Tehtävienhallinta huolehtii, että ajotehtävät tulevat kuitatuksi ja niiden suoritus etenee järjestelmässä. Aikatauluttaja puolestaan huolehtii siitä, mikä ajoneuvo lähtee suorittamaan tiettyä ajotehtävää ja missä järjestyksessä tehtävät ylipäänsä suoritetaan. Suorittavan ajoneuvon valinta voi perustua esimerkiksi lyhimpään etäisyyteen tai matka-ajan kestoon ajotehtävän aloituspisteelle. Kun ajotehtävä on valmis suoritettavaksi ja suorittava ajoneuvo valittu, päästään varsinaiseen reititykseen, jossa reitinsuunnittelija luo reittisuunnitelman pyydettyä ajotehtävää varten. Reitinsuunnitteluprosessiin kuuluu myös reittisuunnitelmasta tiettyjen osien ajettavaksi päättäminen. Reitinsuunnittelijan tuottama reittisuunnitelma välitetään takaisin ohjausjärjestelmälle, joka antaa yksittäisille ajoneuvoille suunnitelman mukaisesti ajokäskyjä. [1, s.123–128]

2.4 Reititysongelma ja reitinsuunnittelumallit

Reititysongelma ilmaistaan yleensä graafina, jota merkitään $G = (V, E)$, jossa V kuvaa joukon solmuja ja E kaaria eli yhteyksiä solmujen välillä. Esimerkiksi paikasta A paikkaan B reitittäminen voidaan kuvata graafina kuvan 5 tapaan. Kuvan graafin solmut edustavat yksittäisiä ajokartan palasia ja kaaret yhteyksiä niiden välillä. Kuvaan on merkitty vihreällä värillä suunniteltu reitti. Tummempi väri ilmaisee osaa, jonka ajoneuvo saa ajaa, ja vaaleampi sellaista suunnitelmaa, joka voi vielä myöhemmin muuttua.



Kuva 5. *Reititysongelmat voidaan esittää graafina.*

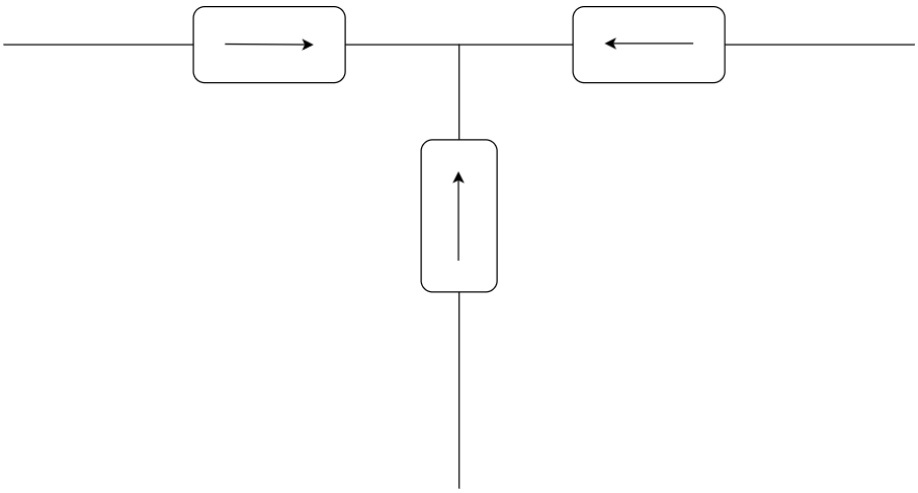
Jatkossa näitä solmuja nimitetään joko *ajosegmenteiksi* tai *legeiksi*. Termiä legi [3] käytetään yleisesti reitityksessä kuvaamaan etappipisteiden välisiä yksittäisiä reitille kuuluvia osia. Reitinsuunnittelussa erotellaan legityypit *ajettavaksi päätetylle* ja *suunnitellulle* ajo-ohjeelle. Legit, jotka reitinsuunnittelija on päättänyt ajettavaksi, sijoittuvat peräkkäisinä reittisuunnitelman alkuun ja niitä ei enää myöhemmin muuteta. Ajettavaksi päätetyt legit ovat myös niitä, jotka ohjausjärjestelmä kommunikoi robottiajoneuvolle ajettavaksi. Suunnitellut legit sen sijaan voivat vielä muuttua, jos reitinsuunnittelija pyrkii välttämään konfliktitilanteen syntymisen.

Kirjallisuudessa esitellään kaksi eri reitinsuunnittelumallia, jotka ovat *staattinen* ja *dynaaminen reititys*. Staattisessa reitityksessä tietyn kuljetustehtävän reitti on jo etukäteen määrätty ennen ajotehtävän suorittamista. Reitinsuunnittelija ei siis enää muokkaa suunniteltuja legejä, vaan antaa vain lisää ajettavaa sen hetkisen ajotilanteen mukaisesti. Esimerkiksi aiemmin esitetyn kuvan 5 reititystehtävän kohdalla käytettäisiin aina samaa ajoreittiä paikkojen A ja B välillä. Dynaaminen reititys puolestaan pyrkii ottamaan huomioon muuttuvat ajotilanteet ja reitti kahden solmupisteen välillä voi vaihdella. Staattinen reitti voi johtaa turhaan ajoneuvojen odotteluun, jolloin dynaaminen reititys pyrkii optimoimaan ajoneuvojen etenemisen valitsemalla toisen reitin ajotehtävää varten. [4, s. 4]

2.5 Reitinsuunnittelun haasteet

Reitinsuunnittelun haasteet johtuvat oleellisesti kolmesta tekijästä. Ensinnäkin kaikki toimintaympäristön robottiajoneuvot täytyy saada reititettyä ajotehtävilleen samaan aikaan siten, että ajotehtävien suoritus etenee tasapuolisesti ja tehokkaasti. Toiseksi robottiajoneuvojen reitinsuunnittelussa on kyse optimointiongelmasta, johon vaikuttaa samaan aikaan useita eri tekijöitä ja muuttujia. Reitinsuunnittelun tavoitteet määräävät esimerkiksi toimintaympäristö, ajotehtävät ja niiden tyyppi, käytettävissä olevat robottiajoneuvot ja ryhmän koko. Optimoitavat tekijät eivät ole triviaali asia ratkaistavaksi ja toisaalta ratkaisut eivät aina tuota parempia tuloksia. Kolmanneksi toimintaympäristöt ja reitinsuunnittelu ovat jatkuvassa kehityksessä ja muutoksessa. Esimerkiksi toimialat, robottiajoneuvojen määrä ja optimoitavat tekijät lisääntyvät.

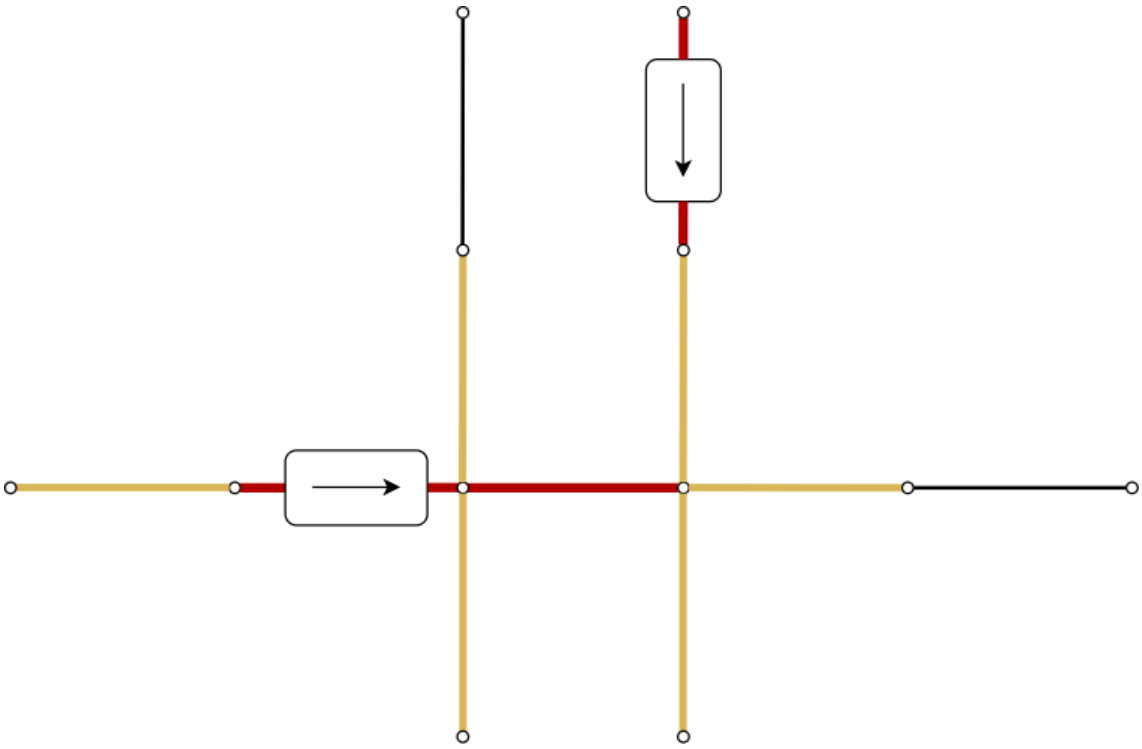
Reitinsuunnittelun tavoitteena on tuottaa reittisuunnitelmia, jotka ovat turvallisia ajaa. Toimintaympäristössä liikkuu mahdollisesti useita kymmeniä robottiajoneuvoja samaan aikaan. Robottiajoneuvojen lisäksi toimintaympäristössä voi liikkua muitakin toimijoita kuten ihmisiä tai ihmisten ajamia ajoneuvoja. Reitinsuunnittelussa korostuukin *konfliktiton reititys*, joka huolehtii, etteivät suunnitellut reitit risteä toistensa kanssa siten, että robottiajoneuvot voisivat törmätä toisiinsa. Myös mahdolliset muut kulkuesteet ajoreiteillä on otettava huomioon. Reitinsuunnittelun täytyy myös huolehtia, etteivät ajoneuvot jää jumiin lukkiutumistilanteisiin, joissa jokin ajoneuvo ei pääse enää etenemään reitillään. [4, s. 6] Kuvassa 6 on esitetty esimerkki lukkiutumistilanteesta, jossa kaikki kolme ajoneuvoa ovat saapumassa samaan aikaan risteykseen. Kuvasta havaitaan, että lukkiutumistilanteen purkamiseksi ajoneuvot täytyisi reitittää uudelleen pois toistensa tieltä ja vasta sitten hallitusti yksi kerrallaan risteyskohdan läpi.



Kuva 6. *Esimerkki kolmen ajoneuvon lukkiutumistilanteesta risteyksessä.*

Konfliktitilanteiden tarkastelua varten reitinsuunnittelussa pidetään kirjaa ajosegmenttien varauksista eri ajoneuvoille. Varauksista erotellaan erikseen niin kutsutut *primääri-* ja *sekundäärivaraukset*. Primäärivarauksia ovat kaikki ne ajosegmentit, jotka on päätetty ajettavaksi, ja sekundäärivarauksia puolestaan kaikki *leikkaavat segmentit*. Leikkaavia segmenttejä ovat ne, joilla oleviin ajoneuvoihin, ajoneuvo voisi primäärivarauksellaan osua. Törmäystilanteiden kannalta voidaan tarkastaa, ettei primääri-primääri tai primääri-sekundääri varauksia tule päällekkäin. Toisin sanoen, jos jokin ajoneuvo on tehnyt primäärivarauksen, niin samalle ajosegmentille ei saa tulla silloin mitään muita varauksia.

Kuvassa 7 on havainnollistettu ajosegmenttien primääri- ja sekundäärivarauksia kahden ajoneuvon tapauksessa. Kumpikin ajoneuvo on saapumassa risteysalueelle ja ne liikkuvat nuolen osoittamaan kulkusuuntaan. Segmenttien päät on korostettu kuvaan erillisinä pisteinä, jotka samalla kuvaavat yhteyksiä segmenttien välillä. Kuvaan on merkitty paksummalla punaisella värillä primääri- ja keltaisella ohuemmalla sekundäärivaraukset. Kuvan esimerkissä ei synny konfliktitilannetta, kunhan ylempi ajoneuvo ei etene merkittyyn kulkusuuntaansa ennen kuin alempi ajoneuvo on mennyt kokonaan ohi.



Kuva 7. *Havainnekuva ajosegmenttien primääri- ja sekundäärivarauksista kahden ajoneuvon tapauksessa.*

Konfliktittoman reitityksen keskeisenä ominaisuutena on parantaa robottiajoneuvojen turvallisuutta. Ajoneuvot liikkuvat määrättyä reittiään, kunnes reitti joko päättyy tai muu turva- tai varojärjestelmä aiheuttaa keskeytyksen. Keskeytyksen syynä voi olla esimerkiksi kulkueste tai ajoneuvon vikaantuminen. Reitinsuunnittelun tehtävänä on kuitenkin estää ja ennaltaehkäistä tällaisten tilanteiden syntymistä ennen ajoneuvon omia turva- tai varojärjestelmiä.

Haasteita reitinsuunnittelulle tuottavat myös robottiajoneuvojen toimintaympäristöt. Esimerkiksi Aziez et al. nostavat esiin tutkimuksessaan [5], että erityisesti sairaalat ovat haastava ympäristö robottiajoneuvojen reitinsuunnittelulle. Keskeiseksi tekijäksi mainitaan kuljetustehtävätyyppien laaja määrä erilaisista ruoka- ja tarvikekuljetuksista lääkkeisiin ja sairaalainstrumentteihin. Kuljetustehtävät hoidetaan erillisillä kärryillä, joita robottiajoneuvot kuljettavat paikasta toiseen. Optimointiongelmaksi tutkimus esittää robottiajoneuvoryhmän koon ja kuljetuskärryjen määrän. Reitinsuunnittelun täytyy tehdä päätös optimaalisen ajoneuvon valinnasta eli esimerkiksi mikä ajoneuvoista on lähimpänä tehtävään vaadittua kärryä. Oikea aikataulutus ja suunnittelun onnistuminen ovat erityisen tärkeitä, jos ajoneuvo kuljettaa sairaalainstrumentteja, joiden on tarkoitus pysyä steriileinä. Toisena käytännön esimerkkinä haastavasta toimintaympäristöstä ovat erityisen kapeakäytäväiset varastohallit. Polten ja Emde painottavat tutkimuksessaan [6] erityisesti kuljetustehtävien aikataulutuksen haastetta reitinsuunnittelussa. Tutkimuksessa nostetaankin keskeisesti esiin jo aiemmin mainitut lukkiutumistilanteet, joihin varautuminen ja niiden ratkaiseminen kapeissa tiloissa on erityisen tärkeää.

Toimintaympäristöt, joissa vaaditaan jokin tietty kapasiteetti- tai läpivientitaso, luovat reitinsuunnittelulle haasteellisen optimointiongelman. Tietyssä ajassa täytyy siis pystyä ajamaan tietty määrä kuljetustehtäviä ja kaikki ylimääräinen odotusaika pitää pyrkiä minimoimaan. Ongelma voidaan kuvata niin kutsuttuna MCFP-optimointiongelmana (engl. Minimum-Cost Flow Problem), jossa graafin kaariin on liitetty *kustannuksia* ja *virtausmääriä*. Reitinsuunnittelun tehtävänä voi olla esimerkiksi pienimmän kustannuksen etsiminen reititettävälle kuljetustehtävälle siten, että virtausmäärät pysyvät kapasiteetti- tai läpivientitasojen rajoissa. Satama-alueet ovat hyvä esimerkki tällaisesta toimintaympäristöstä. Satama-alueet ovat myös jatkuvassa kasvussa sekä ajoneuvojen määrän että kuljetustehtävien osalta. Tästä aiheutuu samalla jatkuva pyrkimys optimoida kuljetustehtävien suorituskykyä, mikä on samalla haaste reitinsuunnittelulle. [7, luvut 1, 2 ja 5]

Vaikka reitinsuunnittelu onnistuisikin, tapahtuu reaalimaailmassa usein tilannemuutoksia. Osa ajoneuvoista saattaa myöhästyä tai olla etuajassa määrätyltä reitiltänsä. Syynä voi olla esimerkiksi este tai jonkin toisen ajoneuvon vikaantuminen. Ylipäänsä tilannemuutokset ovat hyvin yleisiä joustavissa tuotantojärjestelmissä, joissa kuljetustehtävät ja ajoneuvot vaihtelevat ja reitinsuunnittelu elää reaaliaikaisen tilanteen mukaan. Saattaa käydä niin, että lukkiutumis- tai konfliktitilanteen välttämiseksi ja ratkaisemiseksi jotkin ajoneuvot täytyykin reitittää pidempää tai lyhyempää reittiä, jolloin alkuperäisiin suunnitelmiin tulee väistämättä muutoksia. Tästä aiheutuu vaikeus arvioida tarkkaa kuljetustehtävien suoritukseen kuluvaa aikaa. [4, s. 133–136]

Robottiajoneuvot ovat nykypäivänä tyypillisesti akkusähkökäyttöisiä, jolloin reitinsuunnittelun täytyy ottaa huomioon myös latausasemien sijainti sekä ajoneuvojen akun varaustaso. Kabir ja Suzuki painottavatkin tutkimuksessaan [8] akunhallinnan merkitystä osana reitinsuunnittelua ja siten koko tuotantojärjestelmän tehokkuutta. Reitinsuunnittelussa tulee pitää huoli siitä, etteivät latausasemat pääse ruuhkautumaan. Toisin sanoen liian montaa ajoneuvoa ei reititetä samalle latausasemalle samaan aikaan. Tutkimus nostaa myös esiin, että akunhallinta osana reitinsuunnittelun optimointia on vielä melko harvaan tutkittu ongelma. Tämä on hyvä esimerkki siitä, että erilaisia optimointinäkökulmia nousee edelleen esiin ja reitinsuunnittelu on jatkuvassa kehityksessä.

Kaikista edellä mainituista reitinsuunnittelulle nousee tärkeäksi ominaisuudeksi kyky sietää ongelmatilanteita ja muuttuvia olosuhteita. Sietämisen lisäksi ongelmatilanteet ja muuttuvat olosuhteet täytyy pystyä ratkaisemaan. Toimintaympäristössä on vain rajallinen määrä mahdollisia reittejä ja jokainen ajoneuvoista täytyisi saada etenemään ja suorittamaan omaa kuljetustehtäväänsä. Robottiajoneuvoryhmän konfliktiton reititys toistuu kaikissa dynaamisissa reitinsuunnittelumalleissa ja on edelleen keskeisin ratkaistava ongelma riippumatta muista mahdollisista optimoitavista tekijöistä.

3. SIMULOINTI

Robottiajoneuvojen reitinsuunnittelun haastavuudesta ja monimutkaisuudesta johtuen ratkaisujen toimivuutta täytyy voida testata. Pahimmillaan virhetilanteet voivat johtaa esimerkiksi koko teollisuus- tai varastohallin pysähtymiseen tai jopa henkilövahinkoihin. Siksi tarvitaan simulointialusta, jolla erilaisia robottiajoneuvojen kokoonpanoja ja reitityksiä voidaan testata. Tässä luvussa käsitellään simulointi ja sen mallinnus yleisesti sekä lisäksi robottiajoneuvojen simulointi.

3.1 Simulointi ja mallinnus

Simuloinnilla tarkoitetaan todellisuuden jäljittelyä, jota varten tarvitaan mallinnus simuloitavasta ilmiöstä tai systeemistä. Malli puolestaan kuvaa ja pyrkii selittämään ilmiön tai systeemin ominaisuuksia ja käyttäytymistä. Mallin ominaisuuksiin kuuluu myös jonkinasteinen parametrointi, joka mahdollistaa simuloinnin ajon erilaisilla kokoonpanoilla. Simulointia käytetään usein suunnittelun ja päätöksenteon tukena tilanteissa, joissa muut keinot eivät tuota riittävän kattavaa lopputulosta tai ovat muuten epäkäytännöllisiä. Uutta järjestelmää suunniteltaessa voidaan pyrkiä kartoittamaan esimerkiksi kannattavuutta, pullonkauloja tai muita odottamattomia ongelmatilanteita simuloimalla järjestelmää erilaisilla kokoonpanoilla. Simuloinnin tuloksena saadaan yleensä jotain verrattavaa dataa, jonka pohjalta sen onnistumista ja kehityskohteita voidaan arvioida. [9]

Mallinnuksella ei pyritä esittämään simuloitavaa kohdetta täydellisesti, vaan siinä tehdään aina jotain kompromisseja ja yleistyksiä. Mallinnuksen yhteydessä tehdään tiettyjä taustaoletuksia, joilla simulointia ja sen tarkkuutta rajataan. Pyrkimyksenä on pitää simuloitu malli mahdollisimman yksinkertaisena. On hyvä kuitenkin huomioida, että simuloinnin tulos on vain yhtä tarkka kuin sitä varten luotu malli. Toivottua tarkkuutta ei kuitenkaan saavuteta yleensä ensimmäisellä yrityksellä, vaan simulointi ja sen mallinnusprosessi ovat iteratiivisia prosesseja. [9]

Simulointia ja sen mallinnuksen luonnetta voidaan tarkastella muutamasta eri näkökulmasta. Simulointi voi olla *staattista* tai *dynaamista*. Staattisessa tapauksessa malli esittää systeemiä ajasta riippumattomana tai kiinnitettynä tiettyyn ajanhetkeen, kun taas dynaamisessa systeemiä kuvataan ajan suhteen muuttuvana kokonaisuutena. Simulointi voi myös olla *determinististä* tai *epädeterminististä*. Deterministinen simulointi tuottaa samoilla syötteillä aina saman lopputuloksen. Epädeterministisessä simuloinnissa puolestaan samoilla syötteillä voidaan saada eri lopputulos. Esimerkiksi jokin järjestelmä

ruuhkautuessaan voi tuottaa erilaisia tuloksia, vaikka syötteet olisivatkin samoja. Epädeterministiseen simulointiin kuuluu alalajina *stokastinen simulointi*, joka on luonteeltaan todennäköisyyksiin pohjautuvaa ja sitä voidaankin kuvata käyttäen todennäköisyysjakaumia ja muita tilastollisia menetelmiä. Viimeisenä simulointi voidaan vielä jakaa *diskreetiksi* tai *jatkuvaksi*. Diskreetissä simuloinnissa mallin muuttujat voivat muuttua vain tietyllä ajanhetkellä ja jatkuvassa simuloinnissa puolestaan koko simulaation aikana ajan edetessä. Edellä mainitut näkökulmat voivat jokseenkin mennä päällekkäin toistensa kanssa, jos simulointi ja sen mallinnus ovat monimutkaisia. Simuloitavaa systeemiä voidaan kuvata useilla eri malleilla, jotka hyödyntävät eri näkökulmia. [10, s. 1–3]

3.2 Tietokonesimulointi

Tietokonesimulointi on nykypäivänä merkittävä simulointitapa kaikkialla toimialasta riippumatta. Esimerkiksi elektroniikan ja kemian tekniikan parissa tietokonesimuloinnilla on merkittävä rooli analyysityökaluna ja suunnittelun apuvälineenä. Erilaiset videopelit ovat myös hyvä esimerkki tietokonesimuloinnista. [10, s. 178, 227–229] Simulaattorit ovat myös vakiintuneet erilaisissa koulutuksissa ja opetuksessa. Esimerkkeinä näistä ovat vaikkapa ajo- tai lentosimulaattorit, joita käytetään täydentämään koulutusta. Tieteessä ja tutkimuksessa tietokonesimuloinnilla on myös tärkeä rooli, ja sillä voidaankin selittää monimutkaisia järjestelmiä analyyttisiä ratkaisuja paremmin [11, s. 2].

Tietokonesimulointi on poikkileikkaavaa eli siinä yhdistyy moni eri tieteenala samaan aikaan. Mallinnuksessa ja simuloinnissa tarvitaankin usein matematiikkaa, fysiikkaa ja tietotekniikkaa, mutta lisäksi voidaan tarvita biologiaa ja materiaalitekniikkaa. Tietokonesimuloinnin ympärille on rakennettu iso määrä siihen erikoistuneita alustoja ja ohjelmistoja. Tästä huolimatta simulointialustojen rakentamisessa korostuu yhä omienkin ratkaisujen toteuttaminen tilanteissa, joissa esimerkiksi tehokkuus on erityisen tärkeää. Visualisointiratkaisuja ei yleensä rakenneta ainakaan kokonaan itse, vaan niissä suositaan valmiiksi tehtyjä ja jokseenkin vakiintuneita työkaluja. Tietokonepohjaisessa simuloinnissa on otettava huomioon laskennallisen epätarkkuuden ja pyöristysvirheiden mahdollisuus. Erityisesti dynaamisessa simuloinnissa, jossa mallia esitetään ajan suhteen muuttuvana, voivat laskennalliset virheet kumuloitua simuloinnin edetessä. Laskennallisen tarkkuuden ja virheiden mahdollisuus täytyy siis tunnistaa ja ymmärtää, miten ne vaikuttavat mallin käyttäytymiseen. Simulointiohjelmistoissa arvostetaan usein suorituskykyä ja helppokäyttöisyyttä. Joissakin simulaatioissa tehdään raskasta laskentaa visualisoinnin lisäksi, jolloin valitun teknologian täytyy olla riittävän suorituskykyinen. Tällaisissa tapauksissa hyödynnetään yleensä käännettäviä ohjelmointikieliä kuten esimerkiksi C ja C++. Toisaalta simulointiohjelmistojen laaja määrä tekee oikean valitsemisesta

haastavaa ja usein joudutaankin puntaroimaan vähintään muutaman eri teknologian välillä. [11, s. 3–4]

3.3 Visualisoinnin tärkeys

Visualisoinnista on tullut tärkeä elementti ja etenkin tietokonesimuloinnin myötä sen merkitys on korostunut entisestään. Näkö on väitetysti ihmisen tärkein aisti, joten on selvää, että visuaalisen informaation käyttäminen on tärkeää monessa eri tilanteessa, jos ei jopa välttämätöntä.

Rohrer esittää julkaisussaan [12] keskeisimmiksi visualisoinnin hyödyiksi simuloinnin ja sen mallinnuksen kannalta seuraavat tekijät: vahvistus ja validointi, tulosten parempi ymmärtäminen ja kommunikointi, ja uskottavuuden parantuminen. Visualisoinnilla pyritään siis ennen kaikkea näyttämään, miten simuloitu malli toimii ja käyttäytyy. Visualisointi auttaa yleensä kommunikoimaan tuloksia paremmin tilanteissa, joissa simulointimallia tai sen tuloksia ei täysin ymmärretä. Esimerkiksi fysikaalinen malli voi visualisoinnin avulla hahmottua huomattavasti paljon helpommin ja kattavammin kuin pelkässä numeerisessa esityksessään. Uskottavuuden parantuminen rakentuu pitkälti edellä mainittujen kohtien päälle, sillä visualisoinnin tavoitteena on konkretisoida simuloitavaa mallia. Näin vähennetään mallin epävarmuutta, mikä korostuu erityisesti silloin, kun simulointia ja sen tuloksia tarkastelee henkilö, joka ei ymmärrä kaikkea mallin teknistä taustatietoa. Toisaalta visualisointi voi myös johtaa harhaan, jos malli toimiikin väärin tai sen toimintaa ei ymmärretä. Juuri tästä syystä mallin vahvistus, validointi ja kommunikointi ovat erityisen tärkeitä myös onnistuneen visualisoinnin kannalta.

Visualisointeja käyttäessä korostuvat niihin kuuluvat laatu- ja muut hyväksi todetut käytänteet. Rohrer painottaakin julkaisussaan [12], että tärkeimmät hyvän visualisoinnin tekijät ovat vuorovaikutteisuus, joustavuus, realismi, suorituskyky ja helppokäyttöisyys. Vuorovaikutteisuus voi näkyä esimerkiksi simulointimallin tarkasteluna eri näkökulmista tai mahdollisuutena valita yksittäisiä kohteita tarkempaan tarkasteluun. Jos visualisoituja yksityiskohtia tai yksittäisiä kohteita on paljon, on järkevää joustavuuden nimissä tarjota mahdollisuus kytkeä näitä päälle tai pois visualisoinnin aikana. Näin voidaan syventyä haluttuihin kohteisiin tarkemmin sekä ymmärtää mallia ja sen tuloksia paremmin. Visualisointien tulisi myös olla järkevissä määrin realistisia, mikä on vahvasti kytköksissä aiemmin mainittuun uskottavuuteen. Epärealistinen visualisointi herättää kyseenalaistusta ja heikentää mallin uskottavuutta. Suorituskyvyllä puolestaan pyritään siihen, että mallin visualisointi on mahdollisimman sulavaa ja suoraviivaista. Hidas ja takkuileva visualisointi aiheuttaa helposti ihmetystä ja vaikeuttaa tulosten tarkastelua. Viimeisenä tekijänä on helppokäyttöisyys, jossa korostuu visualisoinnin lisäämisen ja käyttämisen

helppous. Siis uusia visualisointeja pitäisi voida lisätä ja vanhoja muokata sekä parannella mahdollisimman helposti. Esimerkiksi ohjelmistokehityksessä visuaaliset elementit tulisi suunnitella ja toteuttaa niin, että niitä voidaan jatkossa lisätä ja laajentaa.

3.4 Robottiajoneuvojen simulointi

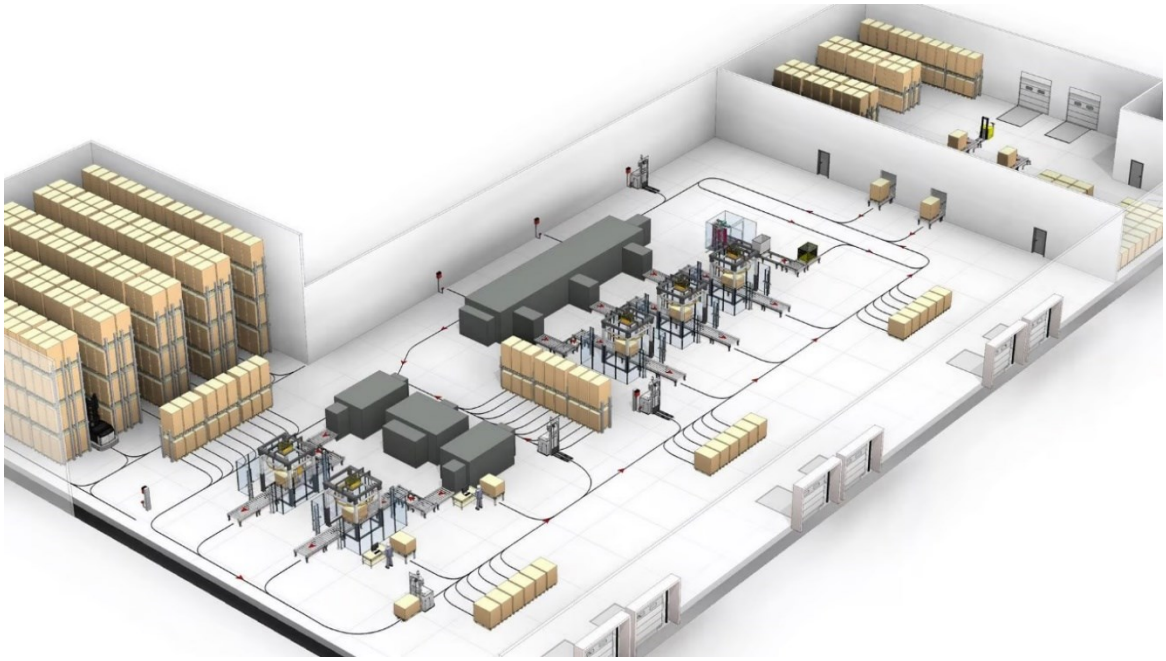
Robottiajoneuvojen simulointia on enimmäkseen käytetty tuotantojärjestelmien suunnittelun ja testaamisen apuvälineenä. Tavoitteena on ollut suunnitella ja optimoida tuotantoketju ajoneuvojen kannalta parhaimmaksi. Simuloinnin tuloksilla on voitu määrittää esimerkiksi optimaalinen ajoneuvojen määrä, järjestelmän yleinen tehokkuus ja suorituskyky. Lisäksi on voitu optimoida ja testata reitinsuunnittelua, mihin tämänkin diplomityön simulointialusta pyrkii. [1, s. 208–209]

Robottiajoneuvojen simuloinnissa korostuu kaksi selkeää tasoa, jotka painottavat eri yksityiskohtia. Matalalla tasolla voidaan simuloida ajoneuvojen omia sisäänrakennettuja järjestelmiä kuten navigointijärjestelmiä. Korkealla tasolla puolestaan voidaan simuloida ajoneuvojen hallinta- ja ohjausjärjestelmiä, jotka hoitavat reitinsuunnittelun ja ajotehtävien suoritusta osana tuotannonhallintaa. Toisin sanoen voidaan siis simuloida tarkemmin itse robottiajoneuvoja tai laajemmin koko tuotantoketjun ja prosessin tapahtumia sekä kulkua. Robottien simulointiin löytyy useita erilaisia työkaluja, jotka mahdollistavat sekä matalan että korkean tason toiminnan kuvaamisen. Esimerkiksi Gazebo-simulaattori [13] pyrkii tarjoamaan kattavat rajapinnat, joiden kautta todellista robottia voidaan simuloida mahdollisimman tarkasti. Vastaavasti FlexSim [14] keskittyy simuloimaan laajemmin koko tuotantojärjestelmää ja toimitusketjua mahdollistaen kuitenkin yksityiskohtaisemmankin kuvauksen. [15]

Korkean tason simuloinnissa ei kuitenkaan yleensä tarvita mainittujen esimerkkien kaltaisia hienostuneita ja yksityiskohtaisia työkaluja. López et al. nostavatkin tutkimuksessaan [15] esiin, että monimutkaisten järjestelmien kohdalla voi olla järkevämpää kehittää joko analyyttinen tai yksinkertaistettu simulointiratkaisu. Tutkimus esittää yhdeksi ratkaisuksi näiden kahden simulointitason hajauttamisen erilleen omiksi kokonaisuuksiksi. Näin kummankin simulointitason hyviä puolia voidaan hyödyntää yksilöllisesti ilman tarvetta mallintaa ja sitoutua yhteen isoon ja monimutkaiseen simulointimalliin.

3.5 Ajokartat simuloinnissa

Robottiajoneuvojen reitityksen simuloinnissa tarvitaan ajokarttoja. Ajokartat koostuvat yksittäisistä paloista, joista käytetään nimitystä *segmentti*. Ajokarttojen luontia varten täytyy mallintaa vähintään tyypillisimpiä *segmenttimuotoja*, joita ovat suorat ja erilaiset mutkat. Mutkasegmenttejä käytetään kääntymiseen, sillä kaikki robottiajoneuvot eivät voi kääntyä paikallaan. S-mutka on tyypillinen kaistanvaihtoon käytetty segmentti kahden suoran välillä. Kuvassa 8 on esitetty havainnekuva esimerkkinä varastohallin asettelusta ja ajokartasta.



Kuva 8. *Havainnekuva varastohallin asettelusta ja ajokartasta [16].*

Kuvasta havaitaan hyvin erilaisten segmenttimuotojen käyttäminen ajokartassa. Kuvassa on esitetty robottitrukkien tyypillinen toimintaympäristö, josta löytyy sekä nouto- että toimituspisteitä kuljetustehtäviä varten. Kuvasta havaitaan myös muutama akkusähkökäyttöisten robottitrukkien latausasema. Reitityksen simuloinnissa robottiajoneuvojen ja toimintaympäristön yksityiskohtainen mallintaminen ei ole tarpeellista. Ajokartasta riittääkin pohjapiirustusmainen kuvaus, josta yksittäiset segmentit ovat eroteltavissa ja toimintaympäristö hahmottuu yleistasolla.

4. TYÖKALUT

Tämän diplomityön simulointialustan toteutuksen kannalta olennaisimmat työkalut ovat Rust-ohjelmointikieli, robotiikan työkalu *ROS* (engl. Robot Operating System) ja siihen kuuluva visualisointityökalu *RViz*. Ajokarttojen piirtämiseen tarvitaan myös erillinen työkalu, joksi on valittu *Tiled*-niminen tasoeditori. Tässä luvussa esitellään edellä mainitut työkalut olennaisilta osiltaan.

4.1 Rust-ohjelmointikieli

Rust on vuonna 2010 julkaistu Mozilla Foundationin kehittämä ohjelmointikieli. Kieli pyrkii C- ja C++-ohjelmointikielien tehokkuuteen kuitenkaan tinkimättä turvallisesta ohjelmoinnista. Keskeisimpänä pyrkimyksenä Rustissa onkin välttää erityisesti muistinhallintaan liittyvät ohjelmointivirheet ja ongelmatilanteet. Kieltä on povattu tulevaisuudessa korvaamaan C-kielet kokonaan. Tehokkuuden lisäksi Rust markkinoi itseään luotettavana ja tuottavana kielenä [17]. [18]

Rust on vielä melko tuore ohjelmointikieli, joka kehittyy ja saa uusia ominaisuuksia aktiivisesti. Kielelle on rakennettu oma Cargo-paketinhallintajärjestelmä, joka tarjoaa muitakin hyödyllisiä ominaisuuksia kuten yksikkötestauksen ja dokumentaation tuottamisen lähdekoodeista. Cargo tarjoaa työkaluja myös Rust-projektien luontiin, kääntämiseen ja ohjelmistoriippuvuuksien hallintaan [19, luku 1]. Keskeisin hyöty Cargossa on, ettei käyttäjän tarvitse valita eri työkalujen ja teknologioiden välillä. Rust on herättänyt suurta kiinnostusta ja sen ympärille on rakentunut yhteisö, mikä näkyy esimerkiksi crates.io-palvelussa julkaistujen Rust-kirjastojen suurena määränä. [17]

Rustissa keskeisiä käsitteitä ja mekanismeja turvalliseen ohjelmointiin liittyen ovat *muutettavuus* (engl. mutability), *omistajuus* (engl. ownership) ja *lainaaminen* (engl. borrowing). Oletuksena kaikki arvot ovat *muuttumattomia* ja *muutettavuus* täytyy erikseen ilmaista *mut*-avainsanalla. [19, luvut 3 ja 4] Alla on esitetty esimerkki muutettavan kokonaislukumuuttujan luomisesta:

```
let mut luku: i32 = 42;
```

Omistajuus on seurausta siitä, miten Rust toteuttaa muistinhallintaa, jonka tarkastukset tehdään kääntämisen aikana. Omistajuus on määritelty kielessä arvo-omistaja-pareina ja yhtä arvoa kohden voi kerrallaan olla vain yksi omistaja. Esimerkiksi kun muuttuja alustetaan, niin sitä vastaava arvo on sidottu muuttujan nimeen, joka on omistajana kyseiseen arvoon. Kun omistajan *näkyvyysalue* päättyy (engl. out of scope), arvo puretaan.

Omistajuus voi myös siirtyä toiselle, jolloin yleistyksenä aiempaan arvo puretaan, kun sillä ei ole enää omistajaa. Lainaamisella tarkoitetaan viitteen luomista johonkin arvoon kuitenkin sillä rajoitteella, ettei viite omista kyseistä arvoa. Myös viitteillä on näkyvyysalueensa eli kuinka pitkään viite on voimassa. Omistajan näkyvyysalueen täytyy kuitenkin olla suurempi kuin viitteiden, jotta viitteet arvoon ovat ylipäänsä voimassa olevia. Samaan tapaan kuin arvojen kohdalla viitteet voivat olla joko *muuttamattomia* tai *muuttavia*. [19, luku 4] Alla on esitetty esimerkki muuttamattoman viitteen luomisesta aiemmin esitettyyn kokonaislukuun:

```
let viite_luku: &i32 = &luku;
```

Muuttavia viitteitä voidaan luoda vain, jos arvo on muutettava. Muuttamattomia viitteitä yhteen arvoon voi olla rajaton määrä, mutta muuttavia vain yksi kerrallaan saman näkyvyysalueen sisällä. Eri viitetyyppejä samaan arvoon ei voi myöskään olla olemassa samaan aikaan. Esimerkiksi saman arvon muuttamaton ja muuttava viite eivät voi olla olemassa samaan aikaan. Näillä mekanismeilla Rust takaa, ettei rinnakkaisuuden kannalta synny kilpailutilanteita. [19, luku 4]

Olio-ohjelmointi on mahdollista Rustilla, vaikka periminen ei ole suoraan mahdollista. Yhteiset ominaisuudet määritellään *trait*-tyyppinä, joka vastaa joistakin olio-ohjelmointikielistä löytyvää rajapinta-tietotyyppiä (engl. interface). Tietotyypit voidaan määritellä joko *struct*- tai *enum*-tyyppeinä, joille voidaan määrittää erillinen toteutuslohko eli metodit *impl*-avainsanalla. Jäsenten näkyvyys on lähtökohtaisesti aina yksityinen, ellei niitä erikseen merkitä julkiseksi *pub*-avainsanalla. Jäsenmetodit määritellään välittämällä funktioparametrina *self*, joka mahdollistaa pääsyn olion tilaan. [19, luku 17]

Rustissa ei ole poikkeuksia eikä siten poikkeuskäsittelyn *try*- ja *catch*-lohkoja. Virheenkäsittelyä varten on olemassa *Result*-luettelotyyppi [19, luku 9]:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Koodissa esiintyvät *T* ja *E* ovat tyyppiparametreja. *T* edustaa tietotyyppiä, jonka arvo asetetaan onnistuneessa tapauksessa. *E* puolestaan edustaa virhetilanteessa asetettavaa virhetietotyyppiä. Vakavissa virhetilanteissa voidaan keskeyttää säikeen suoritus kokonaan. Kielessä ei myöskään ole viitteen *null*-arvoa, vaan sille vastineena on *Option*-luettelotyyppi [19, luku 6]:

```
enum Option<T> {
    None,
    Some(T),
}
```

T edustaa *Option*-luettelotyypin yhteydessä tietotyyppiä, joka asetetaan, jos arvo on olemassa. Ylempänä esitellyt *Result*- ja *Option*-luettelotyypit voidaan käsitellä *match*-rakenteella, jolla Rustissa voidaan yleisesti käydä läpi luettelotyyppien variantteja [19, luku 6].

Ohjelmassa 1 on esitetty Rust-koodiesimerkki, josta nähdään keskeisimpiä yllä esitettyjä kielelle tyypillisiä ominaisuuksia. Ohjelman alussa määritellään oma *List*-tietotyyppi ja sille toteutuksena metodit listaan lisäämiselle ja alkion hakemiselle indeksillä. Jäsenmetodeista havaitaan kielen lainaamismekanismi, sillä olio itsessään välitetään viitteenä metodin kutsun yhteydessä. Pääohjelmasta *main* puolestaan havaitaan, että muuttuja *list* on määritetty muutettavaksi *mut*-avainsanalla, jotta lisäämismetodia voidaan ylipäänsä kutsua. Viimeisenä nähdään käytännön esimerkki *match*-rakenteen käytöstä.

```
1    pub struct List {
2        items: Vec<String>,
3    }
4
5    impl List {
6        pub fn add(&mut self, item: String) {
7            self.items.push(item);
8        }
9
10       pub fn get(&self, i: usize) -> Option<&String> {
11           self.items.get(i)
12       }
13   }
14
15   fn main() {
16       let mut list = List {
17           data: Vec::new(),
18       };
19       list.add("Hello world!".to_string());
20
21       match list.get(0) {
22           Some(item) => println!("{}", item),
23           None => println!("Invalid index"),
24       };
25   }
```

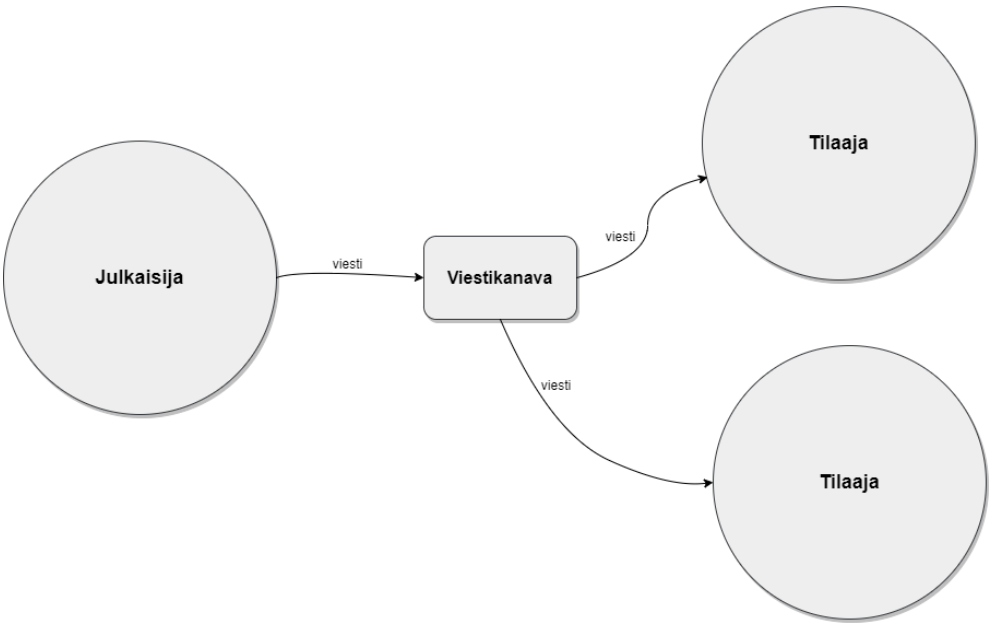
Ohjelma 1. *Yksinkertainen Rust-esimerkkiohjelma.*

Rust on valittu simulointialustan toteutukseen, koska se on moderni ja kattava ohjelmointikieli, joka tarjoaa samalla tehokkuutta tinkimättä turvallisesta ohjelmoinnista. Myöhemmin toteutettavat reitinsuunnittelijat ja reititysalgoritmit aiotaan myös toteuttaa Rustilla, jolloin niiden kytkemisestä simulointialustaan tulee suoraviivaista.

4.2 Robot Operating System

ROS on robotiikan sovelluskohteisiin tarkoitettu avoimen lähdekoodin väliohjelmisto, eikä se nimestään huolimatta ole käyttöjärjestelmä. Väliohjelmistolla tarkoitetaan käyttöjärjestelmän ja sovellusohjelmien väliin sijoittuvaa erillistä ohjelmistokerrosta, joka yksinkertaistaa tietyn alan sovelluskehitystä. Robotiikan väliohjelmisto tarjoaa esimerkiksi ajureita ja kirjastoja, mutta myös työkaluja sovelluskehitykseen, ohjelmistojen suoritukseen ja monitorointiin. ROSista erotellaan kaksi eri versiota: ROS ja ROS 2. Tässä työssä keskitytään nimenomaan versioon ROS 2, johon myöhemmin viitataan vain nimellä ROS. [20, s. 1] [21, s. 3–4]

ROS-sovellukset rakennetaan niin kutsutun *laskennallisen verkon* päälle. Siinä verkon *solmut* edustavat yksittäisiä laskentayksikköjä ja *kaaret* kommunikaatioyhteyksiä niiden välillä. Laskentayksiköt voivat myös sijaita hajautetusti eri tietokoneilla. Kommunikaatioon käytetään *julkaisija-tilaaja-mallia*, jossa joukko solmuja julkaisee viestejä *viestikanavaan* ja toinen joukko kuuntelee niitä. Näitä viestikanavia kutsutaan nimellä *topic*. Julkaisija-tilaaja-malli mahdollistaa solmujen välisen asynkronisen kommunikaation ja viestinvälitysarkkitehtuurille ominaiseen tapaan solmujen ei tarvitse tietää toisistaan. Kuvassa 9 on havainnollistettu julkaisija-tilaaja-mallin toimintaa ROS-solmujen välillä. Kuvassa on esitetty yksi julkaisija- ja kaksi tilaajasolmua, jotka kommunikoivat viestikanavaa pitkin. Viestikanavaan julkaistut viestit kulkeutuvat sieltä kaikille tilaajasolmuille. Kuvan esimerkki on yksinkertaistettu ja todellisuudessa sekä julkaisijoita että tilaajia voisi olla enemmänkin yhtä viestikanavaa kohden. [20, s. 3–4] [21, s. 5, 8]



Kuva 9. *Julkaisija-tilaaja-mallin toiminta ROS-solmujen välillä.*

Muita kommunikaatiotyyppejä ovat *palvelut* (engl. services) ja *toiminnot* (engl. actions), jotka mahdollistavat asiakas-palvelin-arkkitehtuurin toteuttamisen solmujen välille. Solmu voi siis tehdä kutsun suoritettavasta palvelusta ja saada toiselta siihen liittyvän tuloksen. Palvelukutsun tulos voi yksinkertaisuudessaan olla vain tieto siitä, onnistuiko pyydetty operaatio vai ei. Toiminnot puolestaan ovat kuin laajennettuja palveluita, mutta ne on tarkoitettu ajallisesti pitkäkestoisten operaatioiden suoritukseen. Tällöin normaalin palvelupyynnön ja -vastauksen lisäksi on erillinen viestikanava *palautteelle* (engl. feedback), johon palveluntuottaja julkaisee tilannetietoa ja palvelunkutsuja puolestaan kuuntelee sitä. [20, s. 3] [21, s. 8]

Kommunikaatioon käytetyt rajapinnat määrätään *viestityyppeinä*, jotka kuvataan omalla rajapintakielellään *IDL*:llä (engl. Interface Description Language). IDL on ohjelmointikielestä tai arkkitehtuurialustasta riippumaton avoimen standardin kuvauskieli, jolla määritellään tietotyypit ja rajapinnat [22]. ROS itse määrittelee viestien rajapinnat omassa ROS IDL -formaatissaan *msg*-tiedostomuotoina. [20, s. 4]

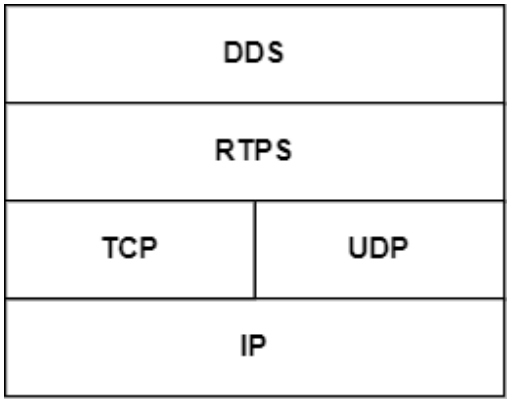
Esimerkiksi robotin kulkureitin navigaatioviesti *Path.msg* kuvataan seuraavanlaisena rajapintana:

```
# An array of poses that represents a Path for a robot to follow.

# Indicates the frame_id of the path.
std_msgs/Header header

# Array of poses to follow.
geometry_msgs/PoseStamped[] poses
```

Rajapintakuvauksesta havaitaan, että navigaatioviesti koostuu kahdesta tietotyypistä: *Header* ja *PoseStamped*-taulukko. *Header* on viestin otsake, joka sisältää esimerkiksi aikaleiman. Otsakkeet sisältävät myös tiedon koordinaatistosta (engl. frame_id), johon viesti liittyy. ROSin koordinaatistoista kerrotaan tarkemmin seuraavassa aliluvussa 4.3. *PoseStamped[]* puolestaan kuvaa listan etappipisteitä, jotka muodostavat robotin kulkureitin. *Header*- ja *PoseStamped*-tietotyypeillä on aiemmin esitetyn navigaatioviestin tapaan oma msg-tiedostomuotoinen rajapintakuvauksensa.

ROSin laskennallinen verkko on toteutettu käyttäen *DDS*:ää (engl. Data Distribution Service), joka on avoin rajapintastandardi. DDS:n pyrkimyksenä on tuottaa suorituskykyinen, skaalautuva, luotettava ja alhaisen viiveen takaava hajautettava julkaisija-tilaaja-mallin mukainen arkkitehtuuri [23]. DDS siis tarjoaa abstraktiot esimerkiksi edellä mainituista laskennallisen verkon solmuista, julkaisijoista, tilaajista, viestikanavista ja julkaisija-tilaaja-mallin mukaisen kommunikaation toteutuksesta. [24]

Tiedonsiirtoyhteyksien toteuttamiseksi DDS käyttää *RTPS-protokollaa* (engl. Real-Time Publish-Subscribe Protocol), joka on suunniteltu toimimaan erityisesti IP-verkkojen päällä [25, s. 8]. DDS:lle ja RTPS-protokollalle löytyy useampia eri toteutuksia. ROS tukee esimerkiksi eProsima Fast DDS:ää, joka mahdollistaa tiedonsiirrossa sekä UDP- (engl. User Diagram Protocol) että TCP-protokollan (engl. Transmission Control Protocol) käytön [26]. Kuvassa 10 on esitetty korkean tason havainnekuva DDS:n ja RTPS-protokollan toteutuksesta IP-verkon päällä.



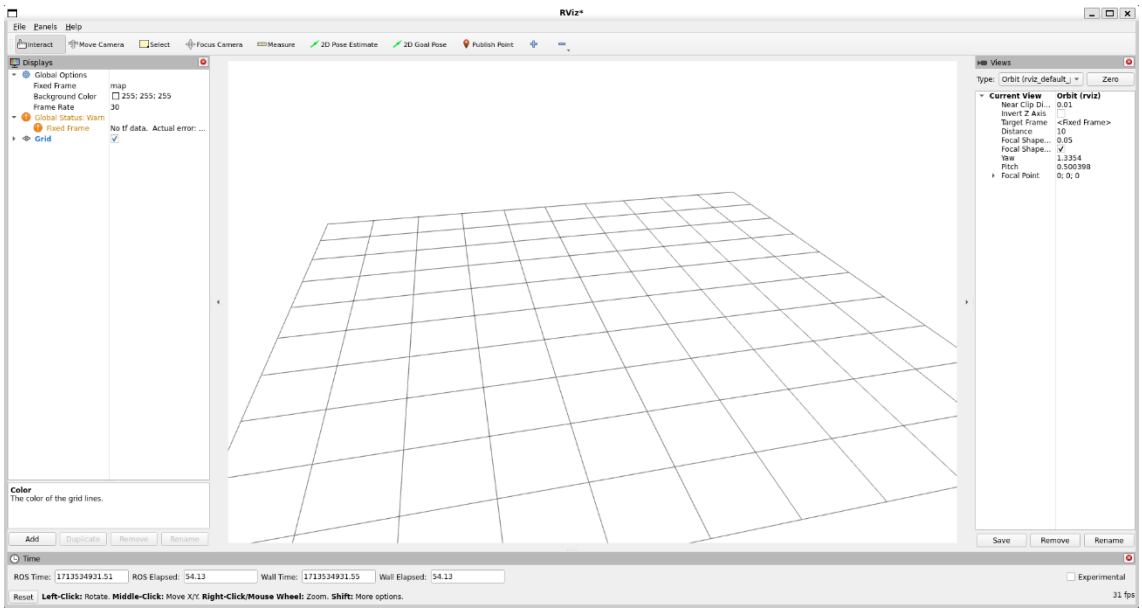
Kuva 10. *Havainnekuva DDS:n ja RTPS-protokollan toteutuksesta IP-verkon päällä.*

Keskeinen asia DDS:n arkkitehtuurissa on QoS (engl. Quality of Service) eli palveluiden laatupolitiikka ja sen määrääminen. Laatupolitiikalla DDS pystyy optimoimaan laskennallisen verkon suorituskykyä ja viivettä sekä määräämään palveluille luotettavuustason. Luotettavuudelle on kaksi politiikkaa, joista ensimmäinen yrittää lähettää viestin täsmälleen kerran ja jälkimmäinen niin kauan, kunnes vastaanottaja kuittaa viestin saaduksi. Viestien pysyvyydelle voidaan määrittää, että ne joko unohdetaan lähettämisen jälkeen tai niitä säilötään mahdollisia myöhässä liittyviä tilaajia varten. Historiapolitiikalla määrätään, mitä tehdään, kun verkko ei pysy perässä. Tätä varten voidaan joko pitää historiatieto kaikista viesteistä, kunnes ohjelma käyttää ne, tai vain tietty kiinteä jono viimeisimpiä, jotka voivat korvaantua uusilla jonon täyttyessä. Mainittujen laatupolitiikkojen lisäksi DDS tarjoaa vielä muitakin politiikkoja, joita tarvitaan erityisesti reaaliaikaisten järjestelmien kohdalla. [20, s. 5]

ROSin käyttämiseksi tämän diplomityön simulointialustassa hyödynnetään Atostek Oy:n kehittämiä Rust-pohjaisia toteutuksia *RustDDS* [27] ja *ros2-client* [28]. RustDDS on nimensä mukaisesti Rustilla tehty DDS-toteutus. ROS-asiakaskirjasto *ros2-client* puolestaan on rakennettu *RustDDS*-toteutuksen päälle ja mahdollistaa ROSin komponenttien käyttämisen. Simulointialustan kannalta ROS on viestinvälitysalusta, jonka avulla simuloinnin tilaa välitetään visualisoitavaksi.

4.3 RViz

RViz [29] on 3D-visualisointityökalu, joka pystyy kytkeytymään ROSin viestikanaviin ja visualisoimaan niihin tulleita viestejä. RViziin on määritelty viestirajapinnat ja niitä vastaavat visualisoinnit kuten *Path.msg*, jonka rajapintamäärittely on esitetty edellisessä aliluvussa. Kuvassa 11 on esitetty RVizin perusnäkymä ja sovelluksen yleisilme. Vasemmassa reunassa esitetään *näyttötyypit* (engl. displays) eli valitut visualisoinnit ja keskellä puolestaan 3D-näkymä, jolle on oikeassa reunassa omat kuvakulmien asetuksensa. Näiden paneelien yläpuolella on vielä työkalurivi, joka tarjoaa esimerkiksi 3D-näkymän kameran liikuttamiseen tai kohteiden valintaan liittyviä toimintoja. 3D-näkymässä näytetään oletusarvoisesti tasoruudukko (engl. grid), joka helpottaa visualisointien hahmottamista.



Kuva 11. *RViz-visualisointityökalun yleisilme.*

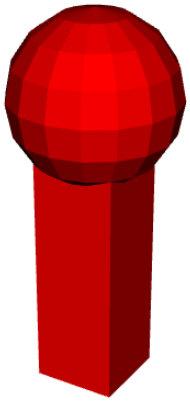
ROS tarjoaa robottien kuvaamiseen oman XML-pohjaisen (engl. Extensible Markup Language) merkintäkielen nimeltä URDF (engl. Unified Robot Description Format). Merkintäkielellä kuvataan robotin yleinen geometria, rakenne ja ulkonäkö. Yksittäiset robottimallin osat kuvataan *linkkeinä* ja niitä voidaan sitoa toisiinsa määrittämällä *liitoksia*. Liitoksella kuvataan, miten eri linkit suhteutuvat toisiinsa. Liitoksille voidaan määrittää erilaisia tyyppejä, joilla kuvataan, miten linkit voivat liikkua toistensa suhteen. Linkkien *geometria* kuvataan oman elementtinsä sisällä ja se tukee joitain yleisimpiä muotoja kuten suorakulmainen särmiö ja pallo. Linkkien visualisointielementeissä voidaan käyttää RGBA-värejä (engl. Red Green Blue Alpha) kuvaamalla ne *materiaaleina*. [30]

Ohjelmassa 2 on esitetty esimerkki robotin kuvaamisesta URDF-merkintäkielellä. Esimerkkiohjelmassa luodaan linkit robotin vartalolle ja päälle. Robotin pää sidotaan kiinni vartaloon kuvaamalla liitos näiden kahden linkin välille. Ohjelmassa luodaan myös punaisen värinen materiaali, jota käytetään vartalo- ja päälinkeissä.

```
1    <?xml version="1.0"?>
2    <robot name="robot">
3       <material name="main">
4           <color rgba="1.0 0 0 1.0"/>
5       </material>
6
7      <link name="body">
8        <visual>
9          <geometry>
10           <box size="0.5 0.5 1.5"/>
11         </geometry>
12         <material name="main"/>
13       </visual>
14     </link>
15
16     <link name="head">
17       <visual>
18         <geometry>
19           <sphere radius="0.5"/>
20         </geometry>
21         <material name="main"/>
22         <origin xyz="0 0 1"/>
23       </visual>
24     </link>
25
26      <joint name="body_to_head" type="fixed">
27       <parent link="body"/>
28       <child link="head"/>
29     </joint>
30   </robot>
```

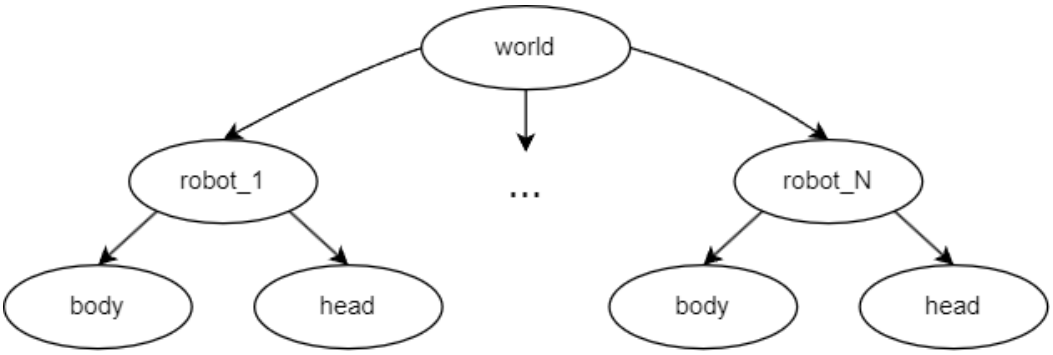
Ohjelma 2. *Esimerkki robotin kuvaamisesta URDF-merkintäkielellä.*

Kuvassa 12 on esitetty, miltä ohjelmassa 2 kuvattu robottimalli näyttää visualisoituna RVizissä. Kuvasta havaitaan robottimallin punainen väri sekä pää ja vartalo, jotka on mallinnettu pallona ja suorakulmaisena särmiönä.



Kuva 12. *Esimerkkirobotti visualisoituna RVizissä.*

Robotin URDF-kuvaus voidaan julkaista ROSin *robot_description* viestikanavaan, johon myös RViz voi kytkeytyä ja visualisoida robottimallin *RobotModel*-näyttötyypillä. Robotin mallin lisäksi täytyy sen sijainti ja suhde ympäristöönsä pystyä määrittämään. Tätä varten ROS tarjoaa *TF-komponenttikirjaston*, jolla pidetään kirjaa ja sidotaan yhteen kaikki sovelluksessa käytettävät koordinaatistot ja niiden esitykset. Kuvassa 13 on havainnollistettu, kuinka N kappaletta aiemmin esiteltyä robottimallia voidaan kytkeä yhteen yleiseen maailmakoordinaatistoon. Jokaisella robotin linkillä on myös oma koordinaatistoesityksensä. [31]



Kuva 13. *Robottimallit ja niiden linkit kytkettynä yhteen yleiseen maailmakoordinaatistoon.*

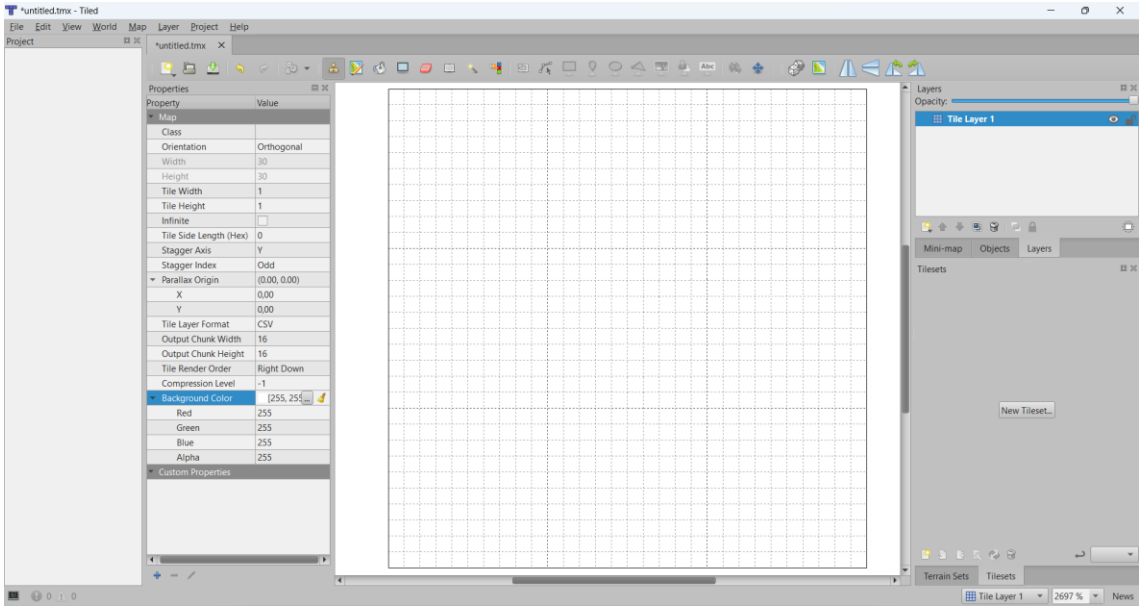
RVizissä on yksi yleinen koordinaatisto, jonka suhteen kaikki visualisoitavat elementit muunnetaan ja kuvataan. Visualisoitavan robottimallin muutoksia kuunnellaan siis *tf*-viestikanavasta, johon julkaistut viestit kuvaavat, mitkä liitokset muuttuvat ja miten.

4.4 Ajokarttojen piirtotyökalu

Ajokarttojen piirtämiseen käytetään Tiled-nimistä avoimen lähdekoodin 2D-tasoeditoria. Tiled on tarkoitettu videopelien karttalaattojen luomiseen, sävytykseen ja muokkaamiseen. Se tarjoaa myös kattavan dokumentaation ja tuen erilaisille alustoille. Tiledilla on takanaan melko aktiivinen yhteisö, jolle tarjotaan esimerkiksi oma keskustelufoorumi ja mahdollisuus osallistua kehitykseen projektin GitHub-säiliön kautta. [32]

Karttojen työstämiseen tarjotaan erilaisia *tasoja* (engl. layer), joihin voidaan tuoda laattoja, erilaisia objekteja ja kuvia. Objektitasoon voidaan luoda muotoja kuten suora- ja monikulmioita sekä ellipsejä. Näiden lisäksi objektitasoon voidaan piirtää omia muotoja murtoviivoina ja lisätä tekstejä sekä kuvia. Objekteja voidaan skaalata eri kokoon, siirtää ja käännellä tasossa. Käyttäjä voi halutessaan tallentaa aiemmin luotuja muotoja ja objekteja malleiksi (engl. template), jolloin niitä voidaan käyttää uudelleen myöhemmin. Piirto-objekteille voidaan antaa myös omia arvoja kuten väri, totuusarvo, numero, teksti tai toinen objekti. [33]

Kuvassa 14 on esitetty Tiled-piirtotyökalun yleisilmettä. Vasemmasta reunasta löytyvät avoinna olevat projektit ja sen vierestä joko avoinna olevan kartan tai valitun objektin tiedot. Kuvan tapauksessa nähdään avoinna olevan kartan tietoja kuten sen koko. Keskellä puolestaan nähdään itse kartta ja 2D-piirtotaso. Oikeasta reunasta löytyvät näkymät piirtotasoille ja kartalla oleville objekteille. Oikean reunan alapuolella ovat myös laattojen työstämistyökalut. Kartanpiirtoikkunoiden yläpuolella on vielä työkalunauha, jossa on sävytykseen sekä muotojen ja objektien luontiin liittyviä työkaluja.



Kuva 14. *Tiled-piirtotyökalun yleisilme.*

Tiled käyttää oletusarvoisesti kartan tallennuksessa omia XML-pohjaisia tiedostomuotojaan *tmx* tai *tsx*, mutta kaiken saa tallennettua myös *JSON*-muotoisena (engl. JavaScript Object Notation). Aiemmin mainittuja malliobjekteja voidaan myös tallentaa JSON-muodossa.

5. SIMULOINTIALUSTAN MÄÄRITTELY

Simulointialustan on tarkoitus toimia tulevien reititysalgoritmien suunnittelun ja testauksen apuvälineenä. Yhtenä keskeisenä tavoitteena on siis tarjota mahdollisimman kattavat eväät ongelmien havainnointiin ja niistä varoittamiseen. Tässä luvussa esitellään simulointialustalle asetetut vaatimukset ja tavoitteet sekä lisäksi keskeisimmät rajapinnat ja korkeantason kuvaus toiminnasta.

5.1 Vaatimukset ja tavoitteet

Päätavoitteena on tuottaa simulointialusta, joka kykenee simuloimaan yhtäaikaisesti usean eri robottiajoneuvon eli ryhmän liikkumista ja etenemistä reiteillään. Simulointialusta koostuu itse simulaattorista ja siihen kuuluvasta visualisoinnista, jolla simulointia ja sen tuloksia voidaan tarkastella. Alustaa täytyy kuitenkin pystyä ajamaan myös ilman visualisointia, joten näiden kahden komponentin pitää olla erillisiä kokonaisuuksia. Työssä on tarkoitus testata ja arvioida, voidaanko simulointialusta toteuttaa käyttäen valittuja työkaluja *Rust*, *ROS* ja *RViz*. Näiden lisäksi vaatimuksena on määrittää ja kehittää alkuun ajokarttojen piirtotyökalu, jolla voidaan piirtää yleisimpiä ajokartan muotoja. Ajokarttojen piirtotyökaluksi on esitetty aiemmassa aliluvussa 4.4 tasoeditori *Tiled*.

Simuloinnille tyypilliseen tapaan alustalla pitää pystyä simuloimaan erilaisia kokoonpanoja eli esimerkiksi eri tyyppisiä ajoneuvoja ja niiden määriä. Keskeisenä vaatimuksena on siis mahdollisuus luoda erilaisia simulointiskenaarioita, joihin kuuluvat ajokarttojen luonti, ajoneuvojen määrä ja alkusijainnit, ja skenaariossa suoritettavat ajotehtävät. Ajokarttojen piirtotyökalun täytyy mahdollistaa yleisimmät segmenttimuodot kuten suora, jonkinlainen mutka kääntymiseen ja S-mutka kaistanvaihtoon. Piirrettyjä ajokartan segmenttejä täytyy pystyä yhdistämään toisiinsa. Ajoneuvoja täytyy voida alustaa yksittäisille ajokartan segmenteille ja määrittää, mihin kulkusuuntaan ja miten päin ajoneuvo lähtee etenemään. Suoritettavat ajotehtävät pitää pystyä kuvaamaan ajoneuvokohtaisesti ja määrittämään ajotehtävän tavoitesegmentti sekä mahdollinen ajoneuvon kulkusuunta tavoitteessa.

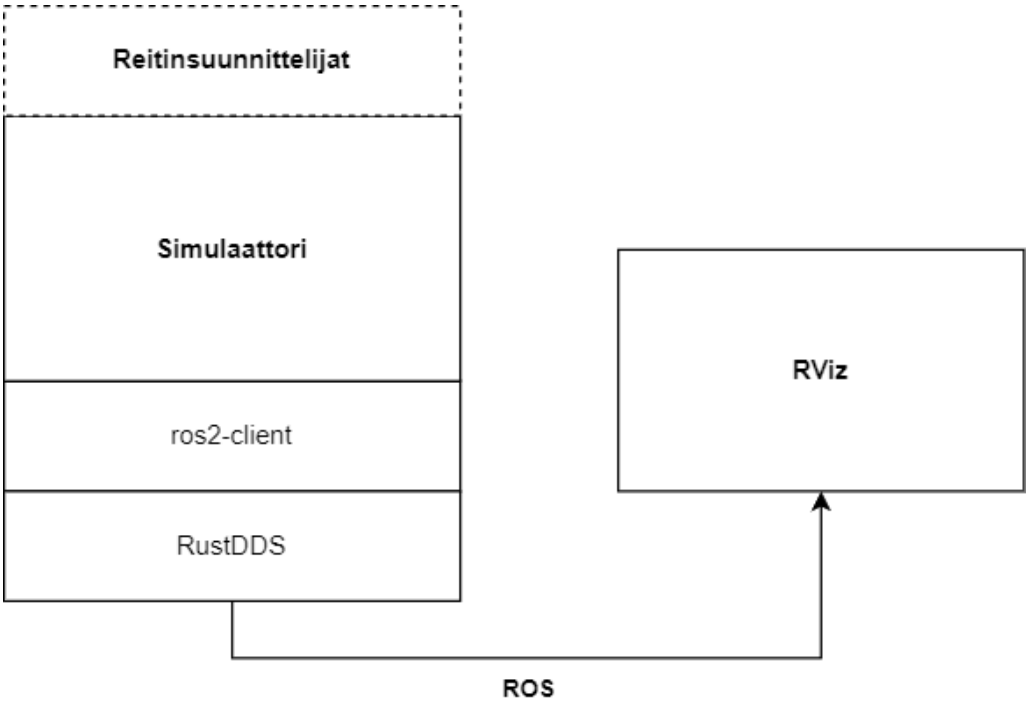
Simuloinnin ajankulkua pitää pystyä säätämään eli tarvittaessa ajamaan nopeammin, hitaammin tai pysäyttämään kokonaan. Ajankulkua täytyy voida myös seurata simulaation edetessä eli tarvitaan jokin keino, jolla simulaation aika näytetään käyttäjälle. Näiden lisäksi simulaattorin täytyy olla *deterministinen* eli samoilla syötteillä se tuottaa aina saman lopputuloksen. Lopputulos riippuu skenaarion kokoonpanosta ja simuloinnin aikana tuotetuista reittisuunnitelmista. Jos skenaarion kokoonpano pysyy samana, mutta simuloinnin lopputulos muuttuu, voidaan muutoksen lähde rajata reititysalgoritmiin. Näin taataan, että simulointialustalla voidaan luotettavasti testata ja arvioida reitinsuunnittelun onnistumista.

Ryhmän eri ajoneuvot täytyy pystyä erottamaan toisistaan eli ajoneuvoilla täytyy olla yksilöllinen tunniste simulaattorissa ja visualisoinnissa. Ajoneuvon visualisointia varten täytyy rakentaa *URDF*-robottimalli, jonka runko voi yksinkertaisimmillaan olla suorakulmainen särmiö. Robottimallista pitää kuitenkin pystyä päättelemään ajoneuvon etenemissuunta. Liikkeen simulointia varten toteutetaan ajan suhteen lineaariset kiihdytys- ja hidastusvaiheet, joita varten ajoneuvoilla tulee olla parametroitavat kiihdytys- ja maksiminopeudet. Myös ajoneuvojen koko ja mitat sekä kääntymiskyky täytyvät olla parametroitavissa. Kääntymiskykyyn riittää tieto, pystyvätkö ajoneuvot kääntymään paikallaan vai eivät.

Simulointialustan tulee luonnollisesti mahdollistaa reitinsuunnittelijoiden kytkeytyminen siihen. Tarvitaan siis rajapinta, jonka kautta reitinsuunnittelijoilta voidaan pyytää reittisuunnitelmia. Rajapinnassa täytyy erotella keskenään kaksi pyyntötyyppiä: uuden reittisuunnitelman luonti ajotehtävää varten ja siitä ajettavaksi määrääminen. Reitinsuunnittelijan täytyy saada rajapinnan kautta tieto simuloinnin tilanteesta eli kaikkien ajonevojen sen hetkiset sijainnit, reittisuunnitelmat ja mahdolliset ajotehtävät. Reitityksen simuloinnissa eri ajoneuvojen reittisuunnitelmat täytyy pystyä erottelemaan keskenään. Reittisuunnitelmasta pitää voida jollain tapaa erottaa *ajettavaksi päätetyt* ja vasta *suunnitellut legit*. Oleellisena vaatimuksena ovat reittisuunnitelmien järkevyystarkastukset ja mahdollisista ongelmista sekä törmäystilanteista varoittaminen. Järkevyystarkastuksia ovat esimerkiksi suunnitelman jokaisen ajosegmentin löytyminen tietorakenteista ja että ajoneuvo ei hyppää toiselle puolelle ajokarttaa tai vaihda ajosuuntaa väärin segmenttien välillä. Mahdolliset törmäystilanteet voidaan päätellä reittisuunnitelmista tutkimalla, ettei päällekkäisiä segmenttivarauksia synny, kuten aliluvussa 2.5 on esitetty.

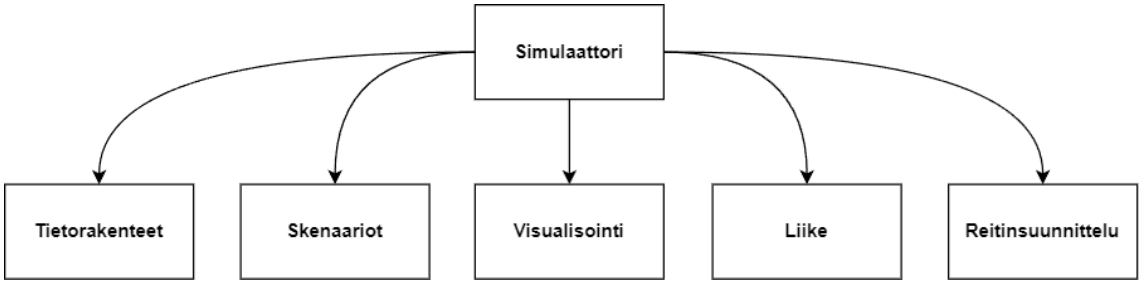
5.2 Rajapinnat ja arkkitehtuuri

Simulointialustan keskeisimmät komponentit ovat reitinsuunnittelijat, simulaattori ja ROSin hyödyntämiseen käytetyt Rust-kirjastot ros2-client ja RustDDS. Kuvassa 15 on esitetty simulointialustan korkean tason arkkitehtuurikuvaus, jossa keskeisimmät komponentit ovat esitetty tasoina ylhäältä alas. Ylimpänä tasona kuvassa on esitetty reitinsuunnittelijat, jotka kytkeytyvät simulaattoriin reitinsuunnittelun rajapinnan kautta. Reitinsuunnittelijoille välitetään simulaattorin ajo- ja tehtävätilanne sekä nykyiset reittisuunnitelmat. Näillä tiedoilla reitinsuunnittelijat tuottavat uusia reittisuunnitelmia simulaattorille, joka puolestaan simuloi ajettavia reittejä ja ajoneuvojen liikettä sekä välittää sen hetkistä tilannekuvaa visualisoitavaksi. Kirjastoja ros2-client ja RustDDS hyödynnetään ROSin käyttämiseksi ja sen kautta välitetään tarvittavat visualisointiviestit RVizille. ROS itsessään toimii tässä siis viestinvälitysalustana simuloinnin tilalta RVizille.



Kuva 15. *Korkean tason arkkitehtuurikuvaus simulointialustan toiminnasta.*

Varsinainen simulaattori koostuu useammasta eri osasta, joista keskeisimmät on esitetty kuvassa 16. Simuloinnin *perustietorakenteita* ovat ajokartta, ajoneuvot, ajotehtävät ja reittisuunnitelmat. *Skenaariot*-osaan kuuluvat simulointiskenaarioiden luonti, kuvaaminen ja lataaminen simulaattorin suoritettavaksi. *Visualisointi* pitää sisällään erikoistuneet komponentit, jotka ovat simuloinnin tilan julkaisijoita. Ne välittävät ROSin viestikanaviin simuloinnin tietorakenteiden sen hetkistä tilaa. Visualisointikomponentit ovat vastuussa ajokartan, ajoneuvojen, reittisuunnitelmien ja törmäysvaroitusten julkaisemisesta visualisoitavaksi. Liike-osuus kattaa ajoneuvojen liikuttamisen reittisuunnitelmillaan ja nopeusprofiilien laskemisen liikettä varten. Viimeisenä osana on reitinsuunnittelu, joka yhdistää samalla ajotehtävien ja reittisuunnitelmien hallinnan sekä mahdollistaa reitinsuunnittelijoiden kytkeytymisen simulaattoriin. Reitinsuunnittelun yhteydessä ovat myös reittisuunnitelmien ja törmäystilanteiden tarkastukset.

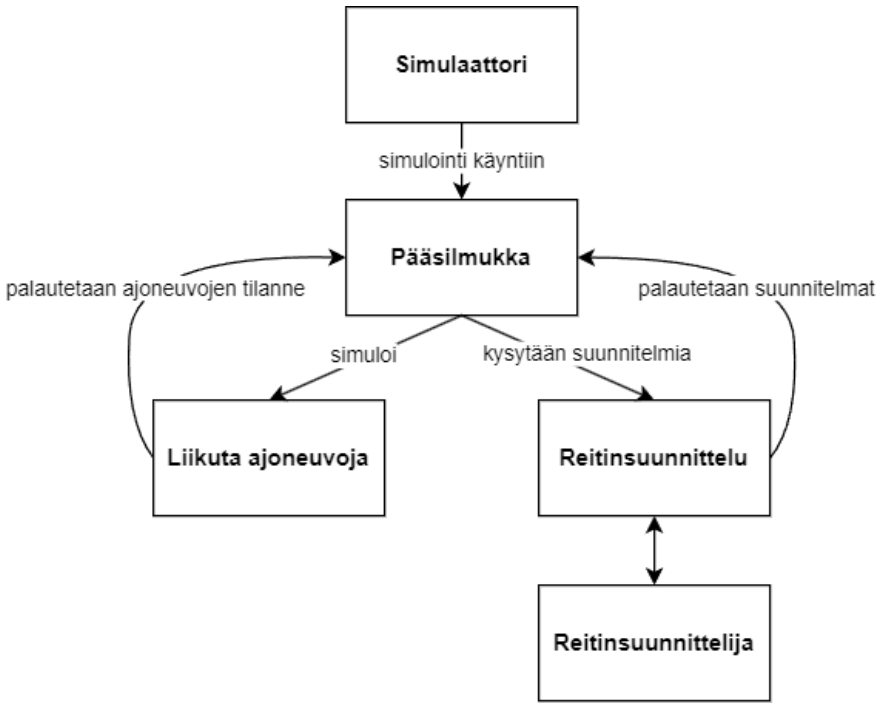


Kuva 16. *Simulaattorin keskeisimmät osat.*

5.3 Yleinen toiminta

Alustan käyttö alkaa kuvaamalla simulointiskenaario, jota halutaan simuloida. Skenaariota varten luodaan ajokartta piirtotyökalulla. Ajokartan lisäksi käyttäjä esittelee ajoneuvot ja ajotehtävät, joita skenaariossa on tarkoitus suorittaa. Ajotehtävät eivät sinänsä ole pakollisia, mutta silloin ajoneuvot eivät liiku aloitussegmenttejään pidemmälle. Simulointialustalle on toteutettu erillinen komentoriviohjelma, jolla simulaattoria käskytetään. Sen kautta voidaan esimerkiksi ladata skenaario, aloittaa ja pysäyttää simulointi sekä tyhjentää tietorakenteet. Komentoriviohjelman kautta on myös mahdollista käynnistää RViz-visualisointityökalu.

Simuloinnin yleinen kulku korkealla tasolla on esitetty kuvassa 17. Varsinainen *simulaattori* pyörii aiemmin mainitun komentoriviohjelman alaisena. Kun käyttäjä on ladannut simulointiskenaarion ja käynnistänyt simuloinnin, siirrytään simulaattorin *pääsilmukkaan*, joka suorittaa varsinaista simulointitapahtumaa. Simulointitapahtumassa kysytään uusia reittisuunnitelmia, jotka voivat muuttua tai pysyä samoina. *Reitinsuunnittelu* välittää alustaan kytketylle *reitinsuunnittelijalle* pyynnön tuottaa uusia reittisuunnitelmia sen hetkisen simulointitilanteen mukaisesti. Reittisuunnitelmat palautetaan lopulta *pääsilmukkaan*, jossa ne tallennetaan simulaattorin tietorakenteisiin. *Pääsilmukassa* kutsutaan myös varsinaista simulointia, joka liikuttaa ajoneuvoja yhden *simulointiaskeleen* reittisuunnitelmillaan ja palauttaa uuden tilanteen. Uusi ajoneuvojen tilanne päivitetään simulaattorin tietorakenteisiin. Tietorakenteisiin päivitetään myös simulaatioaika eli kasvatetaan sitä simulointiaskeleen verran. Lopuksi lähdetään suorittamaan seuraavaa simulointitapahtumaa *pääsilmukassa*.



Kuva 17. *Simuloinnin yleinen kulku korkealla tasolla.*

Pääsilmukan suoritus voidaan keskeyttää, jos käyttäjä pysäyttää simuloinnin tai reittisuunnitelmissa havaitaan mahdollinen törmäystilanne. Yllä kuvattu simuloinnin kulku on tarkoituksella yksinkertaistettu, jotta alustan toimintaperiaate hahmottuisi korkealla tasolla. Toteutus on käsitelty tarkemmin seuraavassa luvussa 6.

6. SIMULOINTIALUSTAN TOTEUTUS

Simulointialustan toteutuksen kannalta olennaisimmat osat ovat ajoneuvot, ajokartat ja simulointiskenaariot, reitinsuunnittelun toiminta, liikkeen simulointi ja visualisointi. Tässä luvussa kuvataan työn varsinainen toteutus ja ratkaisut. Lopuksi nostetaan esiin simulointialustan jatkokehitettävät ominaisuudet.

6.1 Ajoneuvojen mallinnus

Robottiajoneuvojen osalta toteutukseen kuuluu kaksi eri mallinnusta, jotka ovat simulointialustan sisäisesti käytetty tietotyyppi ja sen visualisointi. Yksittäinen ajoneuvo on kuvattu seuraavana tietotyyppinä:

```
pub struct Vehicle {
  vehicle_id: VehicleId,
  color: ColorRGBA,
  dimensions: VehicleDimensions,
  max_speed: VehicleSpeed,
  acceleration_rate: AccelerationRate,
}
```

Ajoneuvon tietotyyppi koostuu siis tunnisteesta, väristä, mitoista sekä nopeus- ja kiihdytyskyvyistä. Tunniste *VehicleId* on yksilöllinen kokonaislukuarvo, joka asetetaan ajoneuvon luonnin yhteydessä. Väri toimii myös tunnisteena ja samaa väriä käytetään kaikissa saman ajoneuvon visualisoinneissa. Mittatietotyyppi *VehicleDimensions* sisältää ajoneuvon pituuden, korkeuden ja leveyden. Ajoneuvon mittoja hyödynnetään visualisoinnissa. Maksiminopeus ja kiihdytyskyky ovat liukulukuarvoja, joita käytetään ajoneuvojen liikkeen simuloinnissa.

Ajoneuvojen visualisointia varten on toteutettu yksinkertainen URDF-robottimalli, joka on melko tynkä ja runko sisältääkin ainoastaan päälinkin *vehicles*. Tämä johtuu siitä, että yksittäiset ajoneuvot kuvataan erikseen simulointiskenaarion latauksen yhteydessä, jolloin saadaan samalla tieto, montako ajoneuvoa luodaan. Robottimallin runko on esitetty alla:

```
<?xml version="1.0"?>
<robot name="Simulator Vehicles">
  <link name="vehicles"></link>
</robot>
```

Ohjelmassa 3 on esitetty ajoneuvojen mallintamiseen käytetty URDF-kuvaus, joka luodaan jokaista simulointiskenaarion ajoneuvoa kohden ohjelmallisesti. Ajoneuvon yksilölliset tiedot kuten nimi, mitat ja väri ovat muuttujina, jotka esitetään ohjelmassa kursivoituina aaltosulkeiden sisällä. Ajoneuvoille luodaan oma *materiaali*, jolla kuvataan käytettävä yksilöllinen väri. Geometria kuvataan *linkkinä*, jonka nimeksi tulee ajoneuvon tunnistetieto. Geometria koostuu suorakulmaisesta särmiöstä ja ympyrälieriöstä. Lopuksi luodaan *liitos*, jolla kuvattu ajoneuvo liitetään aiemmin mainittuun robottimallin päälinkkiin *vehicles*. Robottimallien luominen ja URDF-kuvauskieli on esitetty tarkemmin alaluvussa 4.3.

```
1    <material name="material_{vehicle_name}">
2     <color rgba="{color}"/>
3    </material>
4    <link name="{vehicle_name}">
5      <visual>
6        <geometry>
7          <box size="{dim_as_string}"/>
8            <origin xyz="0 0 0"/>
9        </geometry>
10       <material name="material_{vehicle_name}" />
11     </visual>
12     <visual>
13
14       <geometry>
15         <cylinder length="{cylinder_length}"
16                   radius="{cylinder_radius}"/>
17       </geometry>
18       <origin xyz="-{cylinder_offset} 0 0"/>
19       <material name="material_{vehicle_name}" />
20     </visual>
21    </link>
22
23   <joint name="{vehicle_name}_to_vehicles" type="fixed">
24     <parent link="vehicles"/>
25     <child link="{vehicle_name}"/>
26     <origin xyz="0 0 0"/>
27   </joint>
```

Ohjelma 3. *Yksittäisen ajoneuvon mallintamiseen käytetty URDF-kuvaus.*

Kuvassa 18 on esitetty esimerkki, miltä ylempänä kuvattu URDF-robottimalli näyttää yhden ajoneuvon tapauksessa. Kuvasta havaitaan laatikkomainen runko, johon on liitetty osittain rungon sisään menevä ympyrälieriö. Ympyrälieriö mallintaa ajoneuvon takapuskuria ja siis samalla kertoo ajoneuvon etenemissuunnasta. Robottimallin lisäksi kuvasta havaitaan ajoneuvon tunnisteteksti, joka on oma visualisointiobjektinsa, eikä siis liity robottimalliin.

Kuva 18. *Esimerkki yksittäisen ajoneuvon visualisoinnista.*

Ajoneuvoista on tallennettu kolme eri hakutietorakennetta, joissa avaimena on ajoneuvon tunniste ja arvona sitä vastaava tietotyyppi. Hakutietorakenteiden hallinnointia varten on toteutettu *Vehicles*-tietotyyppi, joka on esitetty alla:

```
pub struct Vehicles {
  vehicles: VehicleData,
  positions: VehiclePositions,
  speed_profiles: SpeedProfiles,
}
```

VehicleData pitää sisällään tiedon yksittäisistä ajoneuvoista eli *Vehicle*-tietotyypin olioista. *VehiclePositions* sisältää ajoneuvojen sijainnit, jotka koostuvat koordinaateista, rotaatiosta ja ajoneuvon sen hetkisestä nopeudesta. Koordinaatit ja rotaatio kuvaavat, missä tason pisteessä ajoneuvo sijaitsee ja mihin suuntaan sen etupää osoittaa. *SpeedProfiles* sisältää ajoneuvojen nopeusprofiilit, joita käytetään liikkeen simuloinnissa määrittämään seuraava nopeus ja matka, jonka ajoneuvo etenee simuloinnin aikana.

6.2 Ajokartat ja piirto

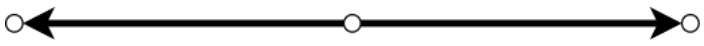
Ajokartat koostuvat yksittäisistä segmenteistä, joille on toteutettu segmenttimuodot: suora, mutka ja S-mutka. Yksittäinen segmentti on kuvattu simulointialustassa seuraavana tietotyyppinä:

```
pub struct Segment {
  id: SegmentId,
  waypoints: Vec<Point>
  segment_type: SegmentType,
  direction: SegmentDirection,
  bounding_box: BoundingBox,
}
```

Segmentti koostuu tunnisteesta, *pistejonosta*, segmenttityypistä eli muodosta, suunnasta ja *suorakaiderajauksesta*. *SegmentId* on kokonaislukuarvo, joka toimii yksilöllisenä tunnisteena segmentille. Pistejono *waypoints* on lista tason pisteitä, jotka kuvaavat segmentin geometrian ja muodon. *SegmentDirection* on erillinen tietotyyppi, jolla kuvataan segmentin ajosuunta. Ajosuunta voi olla joko yksi- tai kaksisuuntainen, joista yksisuuntainen voi lisäksi olla pistejonon alusta loppuun tai lopusta alkuun. Viimeisenä on tietotyyppi *BoundingBox*, joka edustaa suorakaiderajausta segmentille.

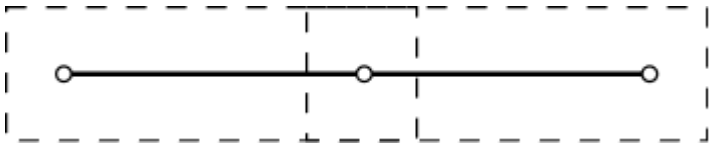
Simulointialustassa ajokartan tietorakenteet koostuvat segmenteistä ja *leikkaavista segmenteistä*. Kumpikin tietorakenteista ovat hakurakenteita, joissa avaimena toimii *SegmentId*. Tietorakenteista voidaan siis hakea kokonaislukutunnisteella segmentin tiedot ja sitä leikkaavat segmentit. Leikkaavilla segmenteillä tarkoitetaan sekä jostain segmentistä lähteviä *yhteyksiä* että sellaisia, joilla oleviin ajoneuvoihin, ajoneuvo nykyisellä segmentillä liikkuessaan voisi osua. Leikkaavien segmenttien tietoa käytetään reitinsuunnittelussa määrittämään, päästäänkö joltain segmentiltä toiselle ja huomioimaan *osuvat segmentit*, jotta ei ajauduta konfliktitilanteeseen.

Yhteydet ja osuvat segmentit tutkitaan erikseen. Yhteys segmenttien välillä voi syntyä vain pistejonojen alku- tai päätepisteistä, mutta sen lisäksi on otettava huomioon mahdollinen ajosuunta. Esimerkiksi yhden segmentin loppupiste voi olla toisen alkupiste mutta molempien ajosuunta päinvastainen, jolloin kummaltakaan segmentiltä ei synny yhteyttä toiselle. Mainitun esimerkin mukaista tilannetta on havainnollistettu kuvassa 19. Segmenttien päät on korostettu erillisinä pisteinä ja nuolet kuvaavat ajosuuntaa. Selkeästi segmentit kytkeytyvät toisiinsa, mutta yhteyttä ei synny ajosuuntien ollessa päinvastaiset. Tällaisessa tapauksessa segmentit käsitellään osuvina.



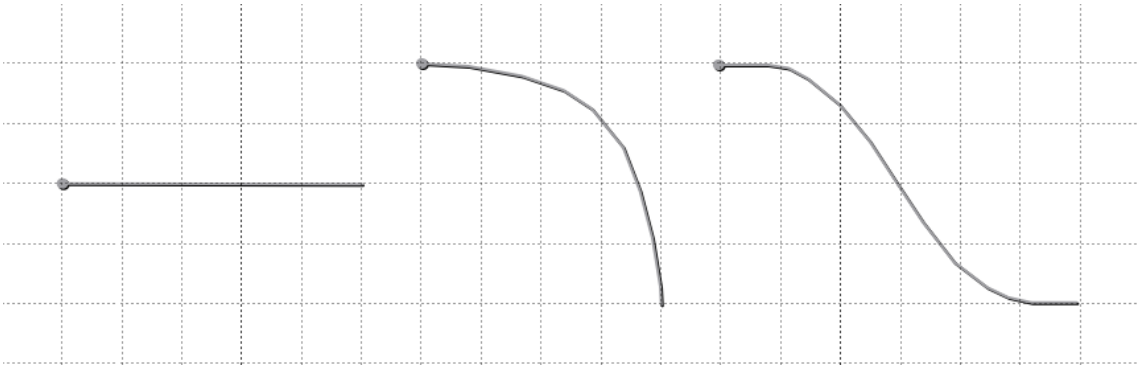
Kuva 19. *Esimerkki kahden segmentin kytkeytymisestä ja ajosuunnasta.*

Suorakaiderajausta käytetään apuna määrittämään osuvat segmentit. Jokaista segmenttiä kohden lasketaan suorakaiderajaus ajoneuvon mittojen perusteella. Ajoneuvo oletetaan ympyräksi, jonka halkaisija on maksimiarvo ajoneuvon leveydestä tai pituudesta. Näin saadaan arvioitua, kuinka paljon tilaa ajoneuvo segmentillä liikkuessaan vie ja siten minimi- ja maksimikoordinaatit suorakaiderajaukselle. Toteutettu algoritmi käy suorakaiderajaukset läpi jokaisen segmentin osalta ja vertaa menevätkö rajaukset päällekkäin. Kuvassa 20 on esitetty esimerkki suorakaiderajauksesta aiemman kuvan kahden segmentin tapauksessa. Suorakaiderajaukset on esitetty kuvassa katkoviivoin segmenttien ympärillä. Kuvasta havaitaan, että rajaukset menevät päällekkäin ja segmentit ovat siis toisiinsa osuvia.



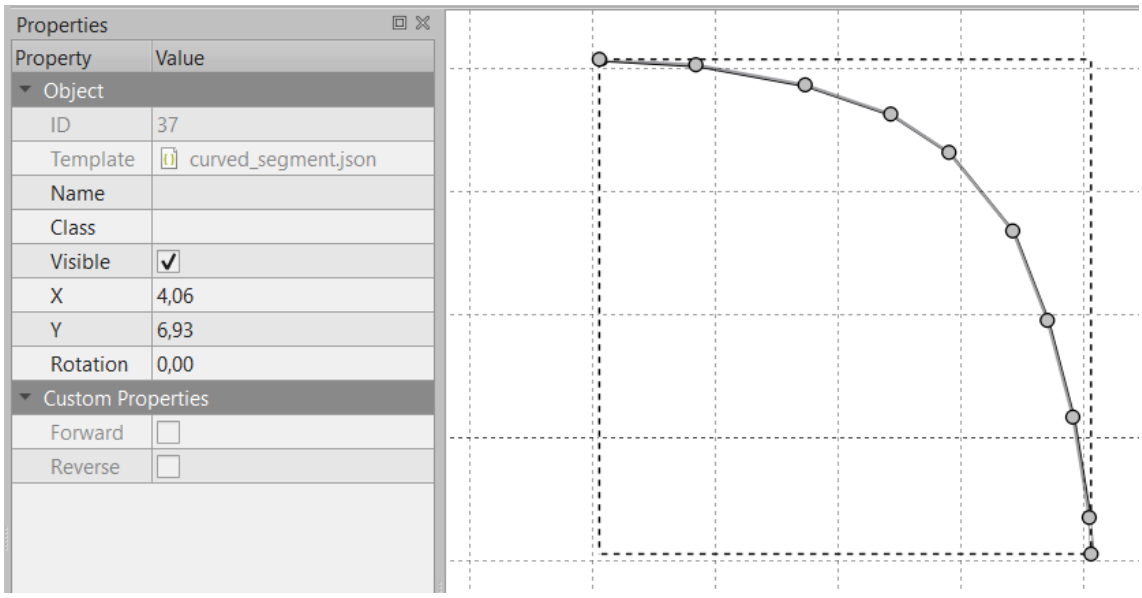
Kuva 20. *Esimerkki suorakaiderajauksesta kahden segmentin tapauksessa.*

Ajokarttojen piirtotyökaluna on käytetty Tiled-tasoeditoria, joka on esitelty tarkemmin aliluvussa 4.4. Tilediin on toteutettu omat mallit segmenttiobjekteille. Näitä malleja voidaan raahata Tiledin piirtotasolle, jossa niitä voidaan liikuttaa ja kääntää. Kuvassa 21 on esitetty toteutetut segmenttimuodot piirtotasolla. Kuvassa on esitetty järjestyksessä vasemmalta oikealle suora, mutka ja S-mutka. Segmenttiobjektien päässä oleva erillinen piste kuvastaa kohtaa, josta muodon piirto aloitetaan. Aiemmin mainittu segmentin pistejono alkaa siis kyseisestä pisteestä.



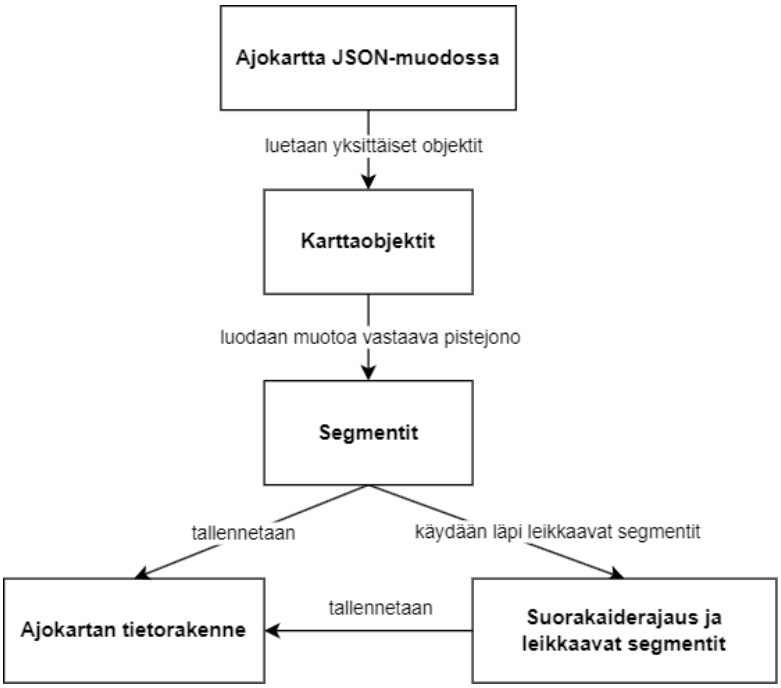
Kuva 21. *Tiled-projektiin toteutetut segmenttimuodot.*

Segmenttiobjektit on luotu piirtämällä halutun muodon mukainen murtoviiva ja tallentamalla se mallina. Malleille on lisätty segmentin ajosuunnan valinta, jonka arvot ovat etu- tai takaperin. Kuvassa 22 on esitetty esimerkki valitusta mutkasegmentistä piirtotyökalussa. Kuvassa vasemmalla on esitetty malliobjektin ominaisuudet ja koordinaatit piirtotasolla. Oikealla puolestaan nähdään mallin muoto ja murtoviiva, joka samalla havainnollistaa segmentin pistejonoa.



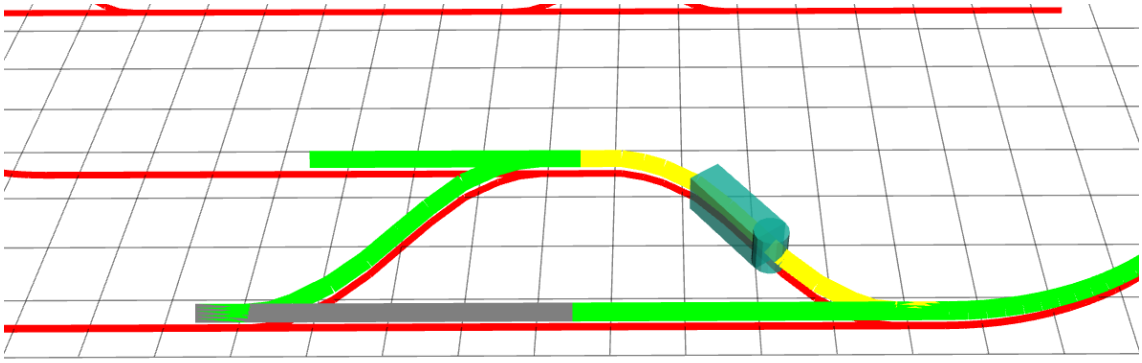
Kuva 22. *Esimerkki malliobjektista piirtotyökalussa.*

Tiled-piirtotyökalulla luotu ajokartta tallennetaan JSON-muodossa, joka luetaan ja tallennetaan simulointialustan omiin tietorakenteisiin. Ajokartan lukeminen ja tallennus on kuvattu korkealla tasolla kuvassa 23. Tiled tuottaa JSON-muodossa listauksen piirtotasoon luoduista objekteista, jotka luetaan väliaikaiseen *karttaobjekti* tietotyyppiin. Karttaobjekteista luodaan *pistejono*, joka kuvaa segmentin muodon ja geometrian. Koska Tiled sijoittaa piirtotason origon vasempaan yläreunaan ja RViz-visualisointityökalu puolestaan tason keskelle, täytyy karttaobjektien pistejonoille suorittaa myös translaatio koordinaatistojen välillä. Näin RVizin tasoruudukkoon visualisoidut segmentit näkyvät täsmälleen kuin ne Tiledissa on piirretty. Karttaobjekteista ja luoduista pistejonoista rakennetaan *konkreettiset segmentit,* jotka ovat aiemmin kuvattuja simulointialustan käyttämiä sisäisiä tietotyyppejä. Lopuksi käydään läpi leikkaavat segmentit *suorakaiderajausta* käyttäen ja tallennetaan tietorakenteisiin sekä yksittäiset segmentit että niitä vastaavat leikkaavat segmentit.



Kuva 23. *Ajokartan lukeminen JSON-tiedostosta korkealla tasolla.*

Yksittäinen ajokartta voi helposti koostua useista kymmenistä segmenteistä, joten niiden tunnistaminen pelkän yksilöivän kokonaisluvun perusteella on hankalaa. Siksi on toteutettu ominaisuus, jolla ajokartan yksittäisiä segmenttejä voidaan korostaa. Korostusvisualisoinnissa näytetään pyydetyn segmentin lisäksi sitä leikkaavat segmentit. Kuvassa 24 on esitetty yksittäisen ajokartan segmentin korostusvisualisointi ja ajoneuvo, joka ajaa kyseisellä segmentillä. Kuvassa on esitetty keltaisella värillä korostettavaksi pyydetty segmentti ja vihreällä yhteydet eli ne segmentit, joihin siitä päästään. Harmaalla värillä esitetyt segmentit ovat niitä, joihin ei ole suoraa yhteyttä, mutta joilla oleviin toisiin ajoneuvoihin keltaisella segmentillä liikkuva ajoneuvo voisi osua.



Kuva 24. *Yksittäisen ajokartan segmentin visualisointi.*

Kuvan 24 tapaus on samalla hyvä esimerkki siitä, kuinka toteutettu suorakaiderajaus tuottaa välillä vääriä tuloksia. Selkeästi kuvan ajoneuvo ei keltaisella segmentillä liikkuessaan osuisi harmaalla segmentillä liikkuvaan toiseen samankokoiseen ajoneuvoon tai toisin päin. Suorakaiderajauksen korvaaminen tarkemmalla arviolla on yksi jatkokehitettävistä ominaisuuksista.

6.3 Simulointiskenaariot

Simulointiskenaariot kuvataan luomalla ajokartta, määrittämällä ajoneuvot ja lopuksi lisäämällä ajotehtävät, joita skenaariossa on tarkoitus suorittaa. Ajokartta tallennetaan JSON-muotoisena ja sen luonti on kuvattu edellisessä aliluvussa. Ajoneuvot ja ajotehtävät puolestaan kuvataan omina CSV-tiedostoinaan (engl. Comma-Separated Values). Jokaista skenaariota varten luodaan oma kansio, jonka nimeksi tulee skenaarion nimi. Skenaario ladataan samalla nimellä komentoriviohjelmassa. Kansion alle tallennetaan piirtotyökalulla luotu kartta nimellä *map.json* sekä tiedostot *tasks.csv* ja *vehicles.csv* ajotehtäviä ja ajoneuvoja varten. Skenaarioiden kansiorakenne näyttää siis seuraavalta:

```
<Skenaarion nimi>
├── map.json
├── tasks.csv
└── vehicles.csv
```

Ajotehtävät kuvataan suorittavan ajoneuvon ja tavoitesegmentin tunnistetiedoilla:

```
vehicle_id,goal_segment_id,goal_progress_direction
```

Kenttä *goal_progress_direction* ei ole vaadittu, mutta sillä voidaan välittää reitinsuunnittelijalle tieto, miten päin ajoneuvon tulisi olla tavoitesegmentillä. Mahdolliset vaihtoehdot ovat *eteenpäin* tai *peruuttaen*. Tyhjä kentän arvo tarkoittaa, ettei etenemissuunnalla ole merkitystä ja ajoneuvo voi saapua tavoitesegmentille kummin päin tahansa.

Ajotehtävät kuvataan lukujärjestyksessä ylhäältä alas, eikä eri ajoneuvojen keskinäisellä järjestyksellä ole väliä. Jokaiselle ajoneuvolle on oma tehtäväjono, johon ajotehtäviä lisätään lukujärjestyksessä. Ajotehtäviin ei ole kuvattu erillistä aloitussegmenttiä, vaan suoritus alkaa ajoneuvon nykyiseltä segmentiltä. Simulointialustan tietorakenteissa ajotehtävät kuvataan seuraavana tietotyyppinä:

```
pub struct Task {
  pub id: TaskId,
  pub goal: SegmentId,
  pub assignee: VehicleId,
  pub goal_direction: GoalDirection,
  pub status: TaskStatus,
}
```

Ajotehtävillä on oma kokonaislukutunnisteensa *TaskId*, joka määräytyy CSV-tiedoston rivinumeron perusteella. Muita tietoja ovat tavoitesegmentti, suorittava ajoneuvo ja etenemissuunta tavoitesegmentillä. Tietotyyppiä *TaskStatus* käytetään reitinsuunnittelussa kuvaamaan, odottaako tehtävä reititystä vai onko se jo reititetty.

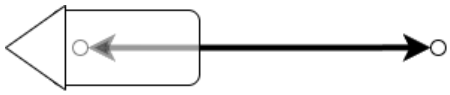
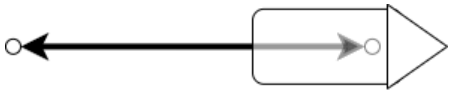
Ajoneuvot kuvataan antamalla yksilöllinen tunnistetieto ja aloitussegmentti:

```
vehicle_id,start_segment_id,segment_orientation,progress_direction
```

Kentällä *segment_orientation* määritellään *ajosuunta*, joka kuvaa, lähteekö ajoneuvo ajamaan segmentin pistejonon *etuperin* alusta loppuun vai *takaperin* lopusta alkuun. Kentällä *progress_direction* voidaan määritellä ajoneuvon *etenemissuunta*. Etenemissuunta kuvaa, ajaako ajoneuvo *eteenpäin* vai *peruuttaen*. Etenemissuunta on valinnainen ja oletusarvoisesti ajoneuvot lähtevät ajamaan eteenpäin.

Taulukossa 1 on esitetty mahdolliset ajo- ja etenemissuunnat kaksisuuntaisella aloitussegmentillä. Taulukon segmentin pistejono on alustettu niin, että vasemmalla on aloitus- ja oikealla loppupiste, jolloin etuperin edetään vasemmalta oikealle ja takaperin päinvastoin. Ajoneuvon etupää on esitetty erillisenä kolmiona ja auttaa hahmottamaan etenemissuuntaa.

Taulukko 1. *Ajoneuvojen mahdolliset ajo- ja etenemissuunnat aloitussegmentillä.*

Alustustilanne	**Segmentin ajosuunta**	**Ajoneuvon etenemissuunta**
	Etuperin	Eteenpäin
	Etuperin	Peruuttaen
	Takaperin	Peruuttaen
	Takaperin	Eteenpäin

Simulointiskenaario ladataan tietorakenteisiin järjestyksessä ajokartta, ajoneuvot ja ajotehtävät. Lataamisen yhteyteen on toteutettu tarkastuksia, jotka varmistavat esimerkiksi, että tarvittavat skenaarion tiedostot löytyvät ja niiden sisältö on oikein. Jos skenaarion alta puuttuu yksikin määrittelytiedosto, alustus epäonnistuu ja tuotetaan virheilmoitus. Ajokartta ei saa olla tyhjä, jotta skenaario voidaan alustaa. Sen sijaan ajotehtävät tai ajoneuvot eivät ole välttämättömiä, mutta silloin skenaariossa ei myöskään tapahdu mitään. CSV-tiedostojen otsakkeet täytyy kuitenkin löytyä ajotehtävien ja ajoneuvojen osalta, vaikka varsinaiset arvot voivatkin olla tyhjät.

Ajotehtäviä ei voi esitellä, jos yhtäkään ajoneuvoa ei ole. Ensiksi siis varmistetaan, että ajoneuvon tunniste on kelvollinen ja sitä vastaava ajoneuvo olemassa. Tavoitesegmentin osalta varmistetaan, että se on määritelty ajokartan tietorakenteissa. Ajoneuvon etenemissuunnalle tavoitesegmentillä ei ole rajoituksia alustusvaiheessa, vaan oikeellisuus tarkastetaan reittisuunnitelmista, jotka esitellään tarkemmin seuraavassa aliluvussa. Ajoneuvojen osalta tarkastetaan, että annettu tunnistetieto on yksilöllinen, eikä siis toista ajoneuvoa ole jo esitelty samalla tunnisteella. Aloitussegmentin täytyy löytyä ajokartan tietorakenteista ja määrätyn ajosuunnan olla segmentin mukainen eli yksisuuntaista segmenttiä ei voida ajaa väärään suuntaan. Vastaavasti kuin edellä määrittelytiedoston puuttuessa, jos yhdenkin kentän arvo on väärin, simulointiskenaarion alustus epäonnistuu ja tuotetaan virheilmoitus.

6.4 Reittisuunnitelmat ja reitinsuunnittelu

Reittisuunnitelmien hallinnointia varten on toteutettu tietotyyppi *RouteController*, joka samalla omistaa reittisuunnitelmien tietorakenteen. *RouteController* tarjoaa rajapinnan, jonka kautta voidaan hakea nykyiset reittisuunnitelmat ja asettaa uusia. Uusien reittisuunnitelmien asettamisen yhteydessä tehdään myös *järkevyys-* ja *törmäystarkastukset*, joita käsitellään tarkemmin reitinsuunnittelun yhteydessä. *RouteController* on toteutettu seuraavana tietotyyppinä:

```
pub struct RouteController {
  route_plans: RoutePlans,
  collision_controller: CollisionController,
}
```

Reittisuunnitelmat *RoutePlans* on hakutietorakenne, jossa avaimena on ajoneuvon tunniste ja arvona sitä vastaava reittisuunnitelma. Tietotyyppi *CollisionController* pitää sisällään reittisuunnitelmien törmäystarkastukset ja niiden visualisoinnit. Yksittäiset reittisuunnitelmat koostuvat legeistä, reittipisteestä ja suoritetuista ajotehtävistä.

Simulointialustassa yksittäiset reittisuunnitelmat on mallinnettu *RoutePlan*-tietotyyppinä:

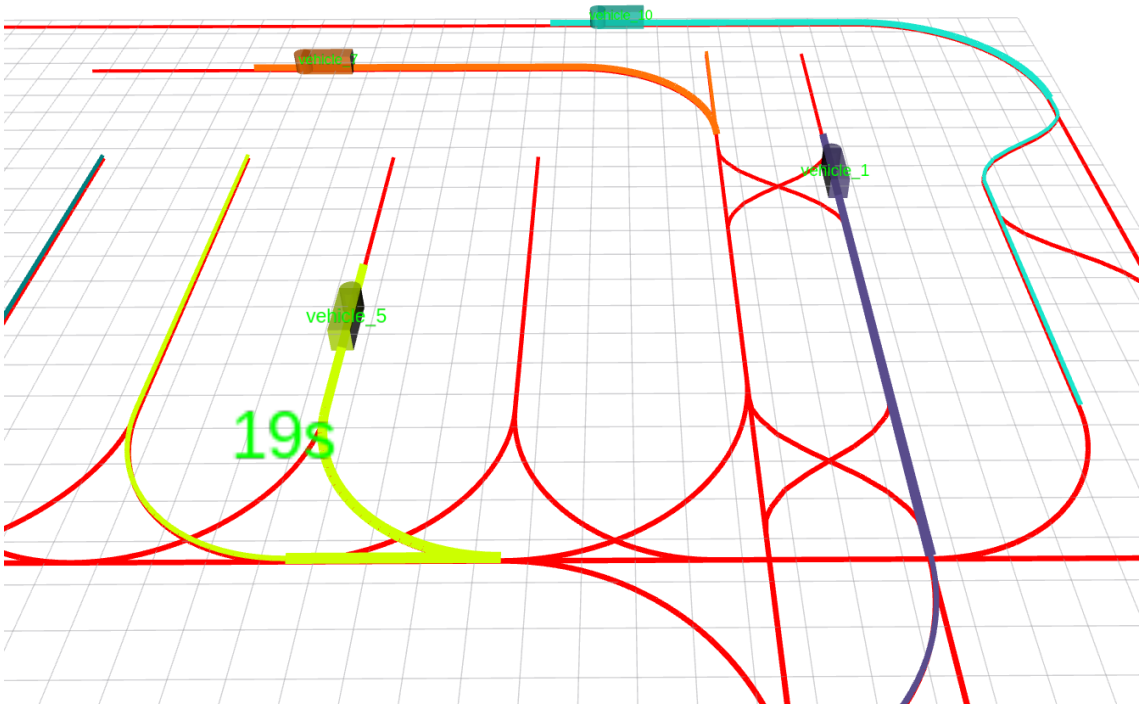
```
pub struct RoutePlan {
  legs: VecDeque<Leg>,
  route_point: RoutePoint,
  passed_goals: Vec<TaskId>,
}
```

Legien kirjanpito on toteutettu jonona, jossa ensimmäinen alkio kuvaa, millä segmentillä ajoneuvo kyseisellä hetkellä ajaa. Reittipistettä varten on oma tietotyyppinsä *RoutePoint*, joka kuvaa ajoneuvon sijaintia ja etenemistä *nykyisellä legillään*. Etenemistietoa käytetään ajoneuvon liikkeen simuloinnissa, joka on esitetty tarkemmin seuraavassa aliluvussa. Kun ajoneuvo liikkuessaan vaihtaa segmenttiä, poistetaan legijonosta ensimmäinen, jolloin seuraavasta legistä tulee uusi nykyinen. Ehtona on, että reittisuunnitelman legijono ei ole koskaan tyhjä, vaan ajettavan loppuessa jää vähintään nykyinen legi. Suoritetut ajotehtävät kerätään talteen etenemisen yhteydessä tutkimalla, onko saavuttu legille, joka on tavoitesegmentti jollekin ajotehtävälle. Legit itsessään voi ajatella yksittäisen segmentin ajo-ohjeina. Yksittäinen legi puolestaan on kuvattu omana tietotyyppinään ja esitetty alla:

```
pub struct Leg {
  pub progress_dir: ProgressDirection,
  pub leg_traverse: LegTraverse,
  pub plan_type: PlanType,
  pub segment: Rc<Segment>,
  pub goal_for_task: Option<TaskId>,
}
```

Legi pitää sisällään tiedon ajoneuvon *etenemissuunnasta*, segmentin *ajosuunnasta*, *suunnitelman tyypistä*, segmentistä ja siitä, onko kyseinen legi jonkin ajotehtävän tavoite. Etenemis- ja ajosuunnat on esitetty tarkemmin edellisessä aliluvussa ja toimintaperiaate on samanlainen reittisuunnitelmien legien tapauksessa. Etenemissuunta kertoo, miten päin ajoneuvo etenee, kun taas ajosuunta kumpaan suuntaan segmenttiä ajetaan. Suunnitelman tyyppi kuvaa, onko legi päätetty *ajettavaksi* vai vasta *suunnitelluksi*. Ajoneuvo voi edetä vain ajettavaksi päätettyjä legejä pitkin. Legiin on myös liitetty viite varsinaiseen ajokartan segmenttiin, jonka tietotyyppi on esitetty tarkemmin aliluvussa 6.2. Ajotehtävän tavoite on mallinnettu *Option*-luettelotyypillä, johon voidaan tallentaa arvo, jos reitinsuunnittelija on merkinnyt legin tavoitteeksi jollekin ajotehtävälle.

Reittisuunnitelmille on toteutettu oma visualisointinsa, joka näytetään samalla värillä kuin sitä vastaava ajoneuvo. Visualisoinnissa on eritelty ajettavaksi päätetyt ja suunnitellut legit. Kuvassa 25 on esitetty esimerkki reittisuunnitelmien visualisoinnista useamman ajoneuvon tapauksessa. Kuvasta havaitaan, että jokainen reittisuunnitelma on visualisoitu samalla värillä kuin sitä vastaava ajoneuvo. Lisäksi nähdään, että suunnitelmissa on selkeästi ohuempia ja paksumpia osia. Paksummat ovat reittisuunnitelman ajettavaksi päätettyjä ja ohuemmat suunniteltuja eli mahdollisesti vielä muuttuvia legejä.



Kuva 25. *Esimerkki reittisuunnitelmien visualisoinnista.*

Simulointialustan reitinsuunnitteluosaa varten on toteutettu tietotyyppi *FleetController*, joka omistaa ajotehtävät ja hallinnoi niiden suoritusta sekä mahdollistaa konkreettisten reitinsuunnittelijoiden kytkeytymisen simulointialustaan. *FleetController* tarjoaa rajapinnan, jonka kautta voidaan hakea ajotehtävät tai pyytää uusia reittisuunnitelmia. *FleetController* on toteutettu seuraavana tietotyyppinä:

```
pub struct FleetController {
  route_planner: Box<dyn RoutePlannerInterface>
  tasks: Tasks,
}
```

Konkreettinen reitinsuunnittelija toteuttaa rajapinnan *RoutePlannerInterface* ja reitinsuunnittelija välitetään *FleetControllerille* rakentajassa. Ajotehtävät *Tasks* on hakutietorakenne, jossa avaimena on ajoneuvon tunniste ja arvona sitä vastaava tehtäväjono. Reitinsuunnittelua varten on toteutettu omat reittisuunnitelman tietotyypit *FleetPlan* ja

FleetLeg. Simulointialustan käyttämään tietotyyppiin *RoutePlan* verrattuna *FleetPlan* sisältää vain ajoneuvon tunnistetiedon ja jonon legejä. Tietotyypit *Leg* ja *FleetLeg* pitävät sisällään lähes saman tiedon, mutta *FleetLeg* ei sisällä suoraa viitettä segmentin tietotyyppiin, vaan sen sijaan käytetään segmentin tunnistetietoa. Tietotyyppien eriyttäminen selkeyttää, milloin on kyse reitinsuunnittelun tuottamista ja simulointialustan ajettavista suunnitelmista. Eriyttäminen mahdollistaa samalla uusien kenttien tai arvojen lisäämisen reitinsuunnittelua varten vaikuttamatta *RoutePlan*-tietotyyppiin. Muunnokset tietotyyppien välillä on toteutettu niin, että *FleetController* muuntaa reittisuunnitelmien pyynnön yhteydessä nykyiset ajettavat suunnitelmat *FleetPlan*-muotoon ja aiemmin esitelty *RouteController* puolestaan uudet tuotetut suunnitelmat takaisin *RoutePlan*-muotoon.

Varsinainen reitinsuunnittelun rajapinta *RoutePlannerInterface* on toteutettu Rustin *trait*-tyyppinä ja esitetty ohjelmassa 4. Rajapinta erottelee kaksi eri kutsutyyppiä, joista ensimmäisellä pyydetään reitti jotakin ajotehtävää varten ja toisella lisää ajettavaa jo reititetylle ajotehtävälle. Tietotyyppi *FleetData* sisältää kaikkien ajoneuvojen reittisuunnitelmat ja ajotehtävät kutsuhetkellä. Ajotehtävät on sisällytetty erilliseen *FleetTasks*-tietotyyppiin, jonka kautta reitinsuunnittelija voi kysyä palveluina seuraavaa mahdollista ajotehtävää ja jo reititettyjä ajotehtäviä jollekin ajoneuvolle. Tietotyyppi *Layout* tarjoaa ajokartan, joka sisältää segmenttien ja leikkaavien segmenttien tietorakenteet. Ajokartan kautta reitinsuunnittelija voi esimerkiksi selvittää pääseekö joltain segmentiltä toiselle.

```
1    pub trait RoutePlannerInterface {
2
3      fn on_route_plans_requested(
4        &self,
5        task: &Task,
6        fleet_data: &FleetData,
7        layout: &Layout,
8      ) -> Result<Vec<Result<FleetPlan, RouteError>>, RouteError>;
9
10     fn on_fixed_legs_requested(
11       &self,
12       task: &Task,
13       fleet_data: &FleetData,
14       layout: &Layout,
15     ) -> Result<Vec<Result<FleetPlan, RouteError>>, RouteError>;
16   }
```

Ohjelma 4. *Reitinsuunnittelun rajapinta määriteltynä Rustin trait-tyyppinä.*

Rajapintakutsujen paluuarvona reitinsuunnittelija palauttaa *Result*-luettelotyypin, jonka sisällä on joko lista reittisuunnitelmia tai virhetyyppi *RouteError*. Myös yksittäinen reittisuunnitelma käyttää samaa *Result*-rakennetta eli tuloksena on joko onnistunut reittisuunnitelma tai virhetyyppi. *RouteError* on luettelotyyppi, jolla reitinsuunnittelija voi esimerkiksi palauttaa, ettei reittiä aloituspisteestä ajotehtävän tavoitteeseen löydy. Monimutkaiselta näyttävä paluuarvo johtuu siitä, että reitinsuunnittelija saattaa joutua muokkaamaan useita reittisuunnitelmia samaan aikaan taatakseen konfliktittoman reitityksen. Voi siis käydä niin, että koko reitinsuunnittelu tai vain yksittäiset reittisuunnitelmat epäonnistuvat.

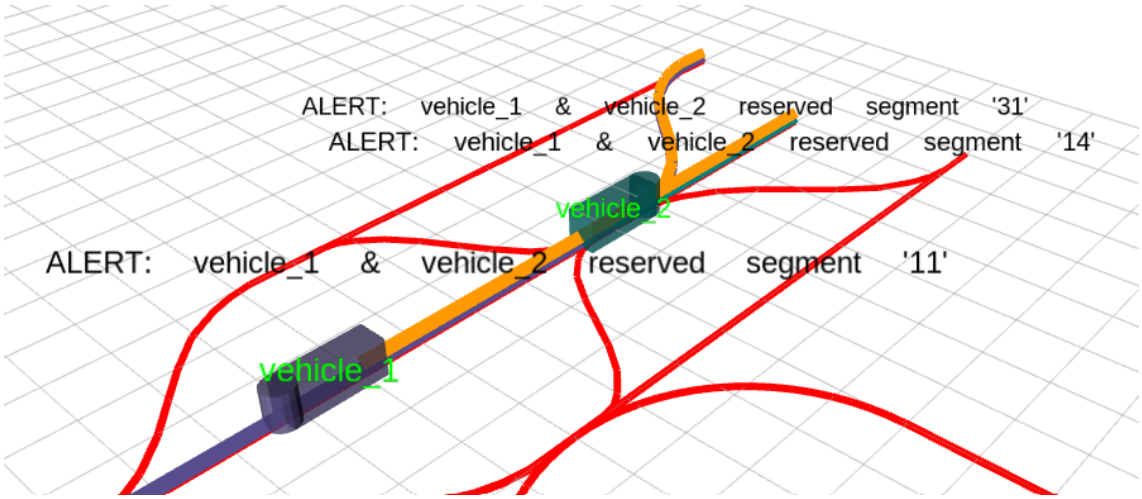
Simulointialustan ja reitinsuunnittelun toiminnan testaamiseksi on toteutettu alustava ensimmäinen reitinsuunnittelija *InitialRoutePlanner*, joka toteuttaa reitinsuunnittelijan rajapinnan, mutta ei takaa konfliktitonta reititystä. Reittisuunnitelmia kysyttäessä *InitialRoutePlanner* pyrkii etsimään reitin ajotehtävälle käyttäen *BFS* (engl. Breadth-First Search) eli leveyteen ensin -graafihakualgoritmia. Kun reitinsuunnittelulta pyydetään lisää ajettavaa, päätetään ajettavaksi enintään kolme ensimmäistä legiä reittisuunnitelmassa. *InitialRoutePlanner* reitittää vain pyydetyn ajotehtävän, eikä edes pyri muokkaamaan muita reittisuunnitelmia samaan aikaan. Alustava reitinsuunnittelija ei myöskään tue etenemissuuntaa ajotehtävän tavoitesegmentillä.

Ajotehtävien suoritus ja hallinnointi on toteutettu *FleetControllerissa* reittisuunnitelmien pyynnön yhteydessä. Simulointialustan ajettavista suunnitelmista käydään ensiksi läpi, mitkä ajotehtävät on suoritettu ja poistetaan jo suoritetut ajotehtävät jonosta. Tehtäväjonosta otetaan ensimmäinen alkio ja aiemmin mainitun ajotehtävän *TaskStatus*-luettelotyypin perusteella määrittyy, pyydetäänkö reitinsuunnittelijalta kokonaan uusi reittisuunnitelma vai lisää ajettavaa. Jos ajoneuvon tehtäväjono on tyhjä, ei reititystä pyydetä ja ilmoitetaan komentoriviohjelmassa, ettei ajoneuvolla ole enää uusia ajotehtäviä. *FleetController* huolehtii myös ajotehtävien tarkastuksista ennen kuin uusia reittisuunnitelmia viedään simulointialustan ajettavaksi. Jokaisesta reittisuunnitelmasta käydään ensiksi läpi, onko legejä merkitty jonkin ajotehtävän tavoitteeksi. Jos tavoitelegejä löytyy, niistä tarkastetaan, että järjestys vastaa ajoneuvon tehtäväjonon suoritusjärjestystä ja ajotehtävä on ylipäänsä merkitty ajoneuvon suoritettavaksi. Jos ajotehtävällä on tietty etenemissuunta tavoitesegmentillä, tarkastetaan legistä, että ajoneuvon etenemissuunta vastaa tavoitetta. Etenemissuunnan lisäksi tarkastetaan, että legi vastaa tavoitesegmenttiä. Uusi reittisuunnitelma hylätään ja tuotetaan virheilmoitus komentoriviohjelmaan, jos yksikin yllä mainituista tarkastuksista epäonnistuu.

Reittisuunnitelmien järkevyystarkastukset on toteutettu tietotyypissä *RouteController* uusien reittisuunnitelmien asettamisen yhteydessä. *RouteController* käy läpi jokaisen reittisuunnitelman yksilöllisesti ja yrittää muuntaa tietotyypin *FleetPlan* simulointialustan *RoutePlaniksi*. Uusi reittisuunnitelma ei saa olla tyhjä ja jokaista legiä vastaava segmentin tunniste täytyy löytyä ajokartan tietorakenteista. Uuden reittisuunnitelman alusta on löydyttävä vähintään kaikki vanhan reittisuunnitelman ajettavaksi päätetyt legit. Siis tarkastetaan, ettei reitinsuunnittelija vaihda jo ajettavaksi päätettyjä legejä takaisin suunnitelluiksi. Ajettavista legeistä tarkastetaan, että ne on sijoitettu *jatkuvasti* reittisuunnitelman alkuun eli esimerkiksi keskeltä ei saa löytyä yksittäisiä ajettavaksi päätettyjä legejä. Reittisuunnitelmasta tarkastetaan myös, että peräkkäisten legien välillä on yhteys ja segmenttejä ajetaan oikeaan ajosuuntaan. Ajosuunnasta tarkastetaan, ettei yksisuuntaista segmenttiä ajeta väärään suuntaan. Legeistä tarkastetaan myös ajoneuvon etenemissuunta segmentillä. Poikkeuksena etenemissuunnan tarkastukselle on, jos ajoneuvot kykenevät kääntymään paikallaan, jolloin etenemissuunta voi vaihtua legien välillä rajattomasti. Liikkeen simulointi huolehtii, että ajoneuvo pysähtyy ennen etenemissuunnan vaihtoa. Muussa tapauksessa etenemissuunta voi vaihtua vain, jos segmenttien yhteystietoon on merkitty, että suunta vaihtuu. Ajotehtävien tarkastusten tapaan *RouteController* hylkää uuden reittisuunnitelman ja tuottaa virheilmoituksen komentoriviohjelmaan, jos yksikin tarkastuksista epäonnistuu.

Reittisuunnitelmien järkevyystarkastusten lisäksi on toteutettu erilliset törmäystarkastukset, joista vastaa tietotyyppi *CollisionController*. Poikkeuksena järkevyystarkastuksiin nähden törmäystarkastukset eivät estä uusien reittisuunnitelmien ajoon hyväksymistä, vaan ongelmista tuotetaan erillinen varoitus. Jos reittisuunnitelma menee läpi järkevyystarkastuksista, *RouteController* välittää sen *CollisionControllerille* törmäystarkastuksia varten. Törmäystarkastukset on toteutettu pitämällä kirjaa primääri- ja sekundäärivarauksista segmenteille kuten aliluvussa 2.5 on esitetty. *CollisionController* käy reittisuunnitelmista läpi kaikki ajettavaksi päätetyt legit ja lisää kirjanpitoon niitä vastaavat primäärisegmenttivaraukset. Ajettavaksi päätettyjen legien leikkaavat segmentit puolestaan lisätään sekundäärisegmenttivarauksina. Kun kaikki reittisuunnitelmien varaukset on tallennettu, voidaan tarkastaa, ettei primääri-primääri tai primääri-sekundääri varauksia tule päällekkäin.

Törmäysvaroituksia varten on toteutettu sekä visualisointi että varoitus komentoriviohjelmaan. Kuvassa 26 on esitetty esimerkki törmäysvaroituksen visualisoinnista. Kuvan tilanteessa ajoneuvo 1 on etenemässä kohti ajoneuvoa 2, joka on pysähtyneenä. Päällekkäiset segmenttivaraukset on korostettu oranssilla ja lisäksi segmenttien päällä on esitetty varoitusteksti.



Kuva 26. *Törmäysvaroituksen visualisointi.*

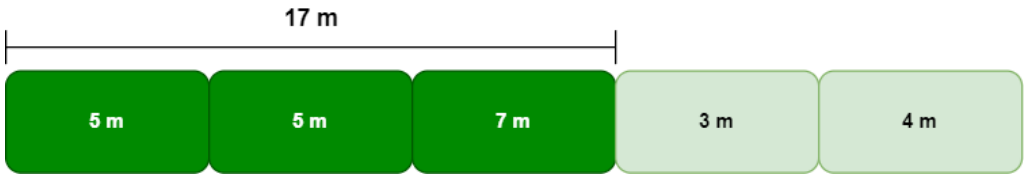
Visualisoinnin lisäksi törmäysvaroituksista tuotetaan komentoriviohjelmaan tarkempi tilannekuvaus. Kuvauksessa mainitaan segmentin tunniste sekä erotellaan primääri- ja sekundäärivaraukset kyseisellä segmentillä. Varausten alle puolestaan tuotetaan listaus ajoneuvoista, joita törmäysvaroitus koskee. Esimerkiksi aiemman kuvan törmäysvaroitus näyttää segmentin 11 tapauksessa komentoriviohjelmassa seuraavalta:

```
COLLISION ALERT: Possible collision detected on segment '11'

Primary reservations
 - vehicle_1
Secondary reservations
 -vehicle_2
```

Vastaavanlainen tilannekuvaus tuotetaan jokaisen segmentin osalta törmäystilanteessa. Siis kuvan esimerkkitapauksessa tuotettaisiin vastaavat tilannekuvaukset myös segmenttien 14 ja 31 osalta.

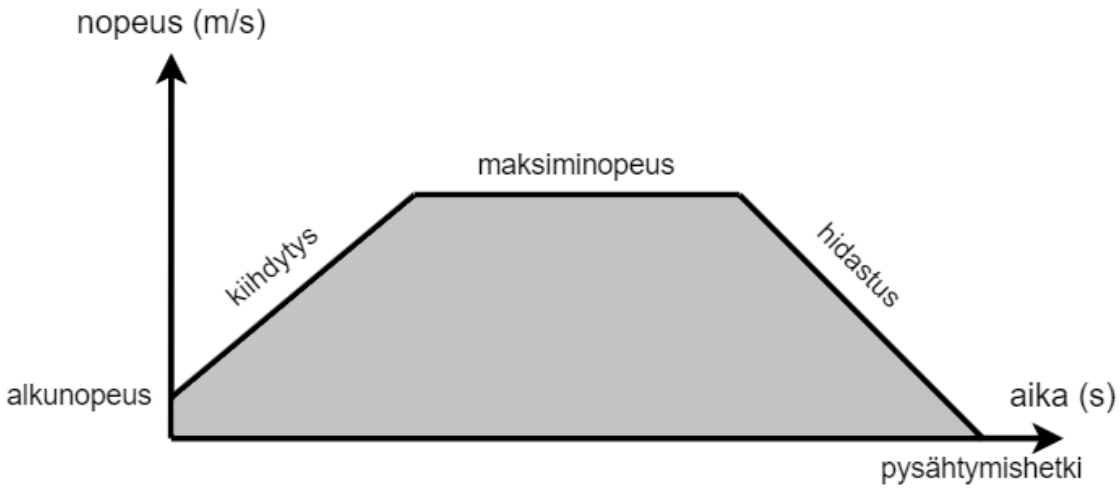
6.5 Liikkeen simulointi

Simulointialustan ajoneuvojen liike on yksinkertaista, sillä ne etenevät reittisuunnitelmiensa ajettavaksi päätettyjä legejä pitkin tai pysähtyvät, jos ajettava loppuu. Liikkeen simulointia varten edettävä matka voidaan siis laskea ajettavaksi päätetyistä legeistä. Kuvassa 27 on havainnollistettu edettävän matkan laskemista reittisuunnitelmasta. Kuvassa on esitetty tummemmalla vihreällä reittisuunnitelman ajettavaksi päätetyt ja vaaleammalla suunnitellut legit. Kuvassa on myös esitetty jokaista legiä vastaava matka. Kuvasta havaitaan, että edettävä kokonaismatka koostuu kolmen ajettavaksi päätetyn legin matkojen summasta.



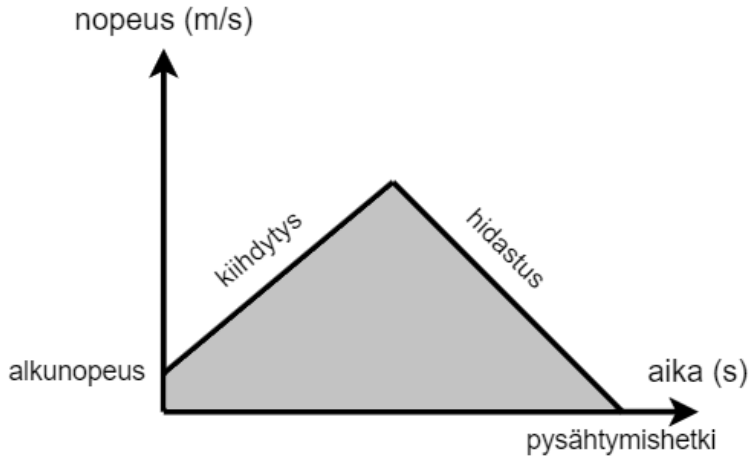
Kuva 27. *Ajoneuvon edettävä matka saadaan ajettavaksi päätetyistä legeistä.*

Jos edettävää matkaa on riittävästi, ajoneuvo kiihdyttää maksiminopeuteen ja ajaa sitä jonkin aikaa ennen hidastusta. Muussa tapauksessa ajoneuvo kiihdyttää nopeuteen, josta on heti hidastettava. Liikkeelle saadaan siten johdettua *kaksi tapausta*. Liike voidaan mallintaa lineaarisilla *nopeusprofiileilla*, jotka kuvaavat ajoneuvon nopeuden suhteessa aikaan. Nopeusprofiilien alle jäävä pinta-ala vastaa ajoneuvon edettävää matkaa. Kuvassa 28 on esitetty esimerkki nopeusprofiilin *maksiminopeustapauksesta*, joka koostuu *kiihdytys*-, *maksiminopeus*- ja *hidastusvaiheista*. Kuvassa on myös esitetty ajoneuvon alkunopeus ja pysähtymishetki, johon samalla nopeusprofiili päättyy.



Kuva 28. *Havainnekuva nopeusprofiilin maksiminopeustapauksesta.*

Kuvassa 29 puolestaan on esitetty esimerkki nopeusprofiilin *kiihdytys ja hidastus -tapauksesta*. Ajoneuvon alkunopeus ja pysähtymishetki on esitetty samaan tapaan kuin aiemmassa kuvassa 28.



Kuva 29. *Havainnekuva nopeusprofiilin kiihdytys ja hidastus -tapauksesta.*

Ajoneuvoille on mallinnettu kiihdytys-, hidastus- ja maksiminopeuskyvyt, jotka on esitetty tarkemmin aliluvussa 6.1. Simulointialustan tietorakenteissa pidetään jatkuvasti kirjaa ajoneuvojen sijainnista ja sen hetkisestä nopeudesta. Nopeusprofiileja laskettaessa tiedetään siis ajoneuvon tiedot, alkunopeus ja edettävä matka. Näillä tiedoilla voidaan määrittää laskukaavat esimerkiksi pysähtymishetkelle, jonka kautta muut nopeusprofiilin vaiheet voidaan määrittää.

Nopeusprofiilien kiihdytys- maksiminopeus- ja hidastusvaiheet voidaan määritellä suoran yhtälön tapaan seuraavasti

$$v_a(t) = at + v_0 \tag{1}$$

$$v_d(t) = -at + b \tag{2}$$

$$v_c(t) = v_{max} \tag{3}$$

Symboli a on samaan aikaan sekä ajoneuvon kiihdytys- että hidastuskyvyn positiivinen vakioarvo, t on aikamuuttuja, v_0 on ajoneuvon alkunopeus ja b on hidastusvaiheen suoran vakiotermi. Maksiminopeus puolestaan on esitetty symbolilla v_{max}. Hidastusvaiheen vakiotermi b on tuntematon ja se määrätään myöhemmin pysähtymishetken kautta.

Toteutuksessa tarvitaan tieto, milloin nopeusprofiilin eri vaiheet alkavat ja päättyvät. Kiihdytysvaihe aloitetaan aina hetkellä nolla ja se päättyy joko leikkauskohtaan hidastusvaiheen tai maksiminopeuden kanssa. Merkitään kiihdytyksen päättymishetkeä hidastusvaiheen suhteen symbolilla t_i, jolloin pätee $at_i + v_0 = -at_i + b$. Tällöin kiihdytyksen päättymishetkelle hidastusvaiheen suhteen saadaan kaava

$$t_i = \frac{b - v_0}{2a} \tag{4}$$

Merkitään kiihdytyksen päättymishetkeä maksiminopeuden suhteen symbolilla t_a. Määritelmän mukaan t_a toteuttaa siis yhtälön $at_a + v_0 = v_{max}$, jolloin saadaan kaava

$$t_a = \frac{v_{max} - v_0}{a} \tag{5}$$

Merkitään hidastusvaiheen alkamishetkeä maksiminopeuden suhteen symbolilla t_d, jolloin pätee $-at_d + b = v_{max}$, josta saadaan kaava

$$t_d = \frac{b - v_{max}}{a} \tag{6}$$

Hidastusvaiheen alkamishetki on nopeusprofiilin eri tapauksissa siis joko t_i tai t_d. Hidastusvaihe puolestaan päättyy pysähtymishetkeen, jota merkitään symbolilla t_z. Pysähtymishetkellä hidastusvaiheen nopeus on nolla eli t_z toteuttaa yhtälön $-at_z + b = 0$. Hidastusvaiheen vakiotermi voidaan siis esittää pysähtymishetken suhteen muodossa

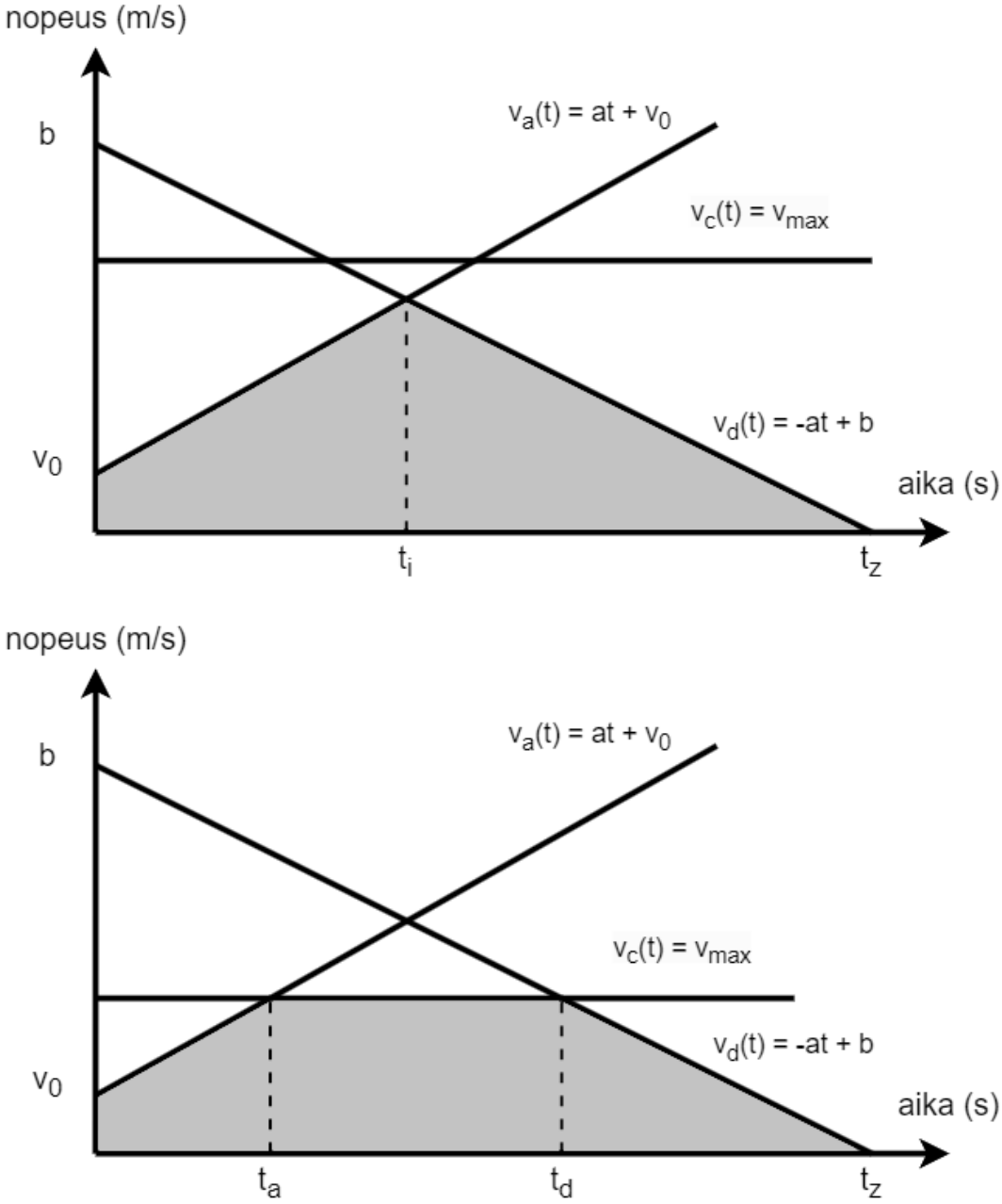
$$b = at_z \tag{7}$$

Taulukossa 2 on esitetty yhteenveto käytetyistä symboleista ja niiden määritelmistä. Vasemmassa sarakkeessa on esitetty symbolit ja oikeassa niitä vastaavat määritelmät.

Taulukko 2. *Yhteenveto käytetyistä symboleista ja niiden määritelmistä.*

Symboli	**Määritelmä**
a	Ajoneuvon kiihdytys- ja hidastuskyvyn positiivinen vakioarvo
b	Hidastusvaiheen suoran vakiotermi
v_0	Ajoneuvon alkunopeus nopeusprofiilia laskettaessa
v_{max}	Ajoneuvon maksiminopeus
t_i	Kiihdytysvaiheen päättymishetki hidastusvaiheen suhteen
t_a	Kiihdytysvaiheen päättymishetki maksiminopeuden suhteen
t_d	Hidastusvaiheen alkamishetki maksiminopeuden suhteen
t_z	Pysähtymishetki eli hidastusvaiheen päättymishetki

Kuvassa 30 on esitetty esimerkki nopeusprofiilien kahdesta tapauksesta merkintöineen. Ylempi nopeusprofiili esittää kiihdytys ja hidastus -tapauksen ja alempi maksiminopeus-tapauksen. Kuvassa on esitetty kaavojen (1) ja (2) mukaiset kiihdytys- ja hidastusvaiheet sekä kaavan (3) mukainen maksiminopeus. Kuvaan on myös havainnollistettu nopeus-profiilien eri vaiheiden alkamis- ja päättymishetket. Ylemmästä nopeusprofiilista havaitaan erikoistapaus, jossa kiihdytyksen päättymis- ja hidastuksen alkamishetki (t_i) ovat samat. Harmaalla värillä on korostettu suorien alle jäävä pinta-ala eli matka, jonka ajoneuvo kuvan mukaisissa nopeusprofiileissa ajaisi.



Kuva 30. *Nopeusprofiilien molemmat tapaukset merkintöineen.*

Edettävän matkan täytyy vastata nopeusprofiilin alle jäävää pinta-alaa, joka koostuu eri vaiheiden pinta-alojen summasta. Määritellään nopeusprofiilien kiihdytys-, maksiminopeus- ja hidastusvaiheissa edettäviksi matkoiksi samassa järjestyksessä: s_a, s_c ja s_d. Määritellään koko nopeusprofiilin edettäväksi matkaksi s_{tot}.

Kiihdytysvaiheen matkalle saadaan kinematiikan perusyhtälöistä kaava

$$s_a(t) = \frac{1}{2}at^2 + v_0 t, \tag{8}$$

jossa muuttujan t paikalle voidaan sijoittaa kiihdytysvaiheen päättymishetki. Maksiminopeusvaiheessa puolestaan on kyse tasaisesta liikkeestä, jolloin matka voidaan ilmaista

$$s_c(t) = v_{max} t, \tag{9}$$

jossa muuttujan t paikalle voidaan sijoittaa hidastuksen alkamis- ja kiihdytyksen päättymishetkien erotus maksiminopeuden suhteen eli $t_d - t_a$. Hidastusvaiheen matka voidaan laskea kolmion pinta-alana

$$s_d(t) = \frac{1}{2}vt, \tag{10}$$

jossa muuttujien v ja t paikoille voidaan sijoittaa hidastusvaiheen alkamisnopeus sekä päättymis- ja alkamishetkien erotus.

Nopeusprofiilin kiihdytys ja hidastus -tapauksessa kokonaismatka määräytyy kiihdytys- ja hidastusvaiheiden matkojen summana, jolloin saadaan $s_a + s_d = s_{tot}$. Kiihdytyksen päättymis- ja hidastuksen alkamishetki on nyt t_i, jolloin kokonaismatka voidaan ilmaista muodossa

$$s_a(t_i) + \frac{1}{2}v_a(t_i) \cdot (t_z - t_i) = s_{tot}, \tag{11}$$

johon sijoittamalla t_i ja b kaavojen (4) ja (7) mukaisesti saadaan yhtälö, joka voidaan ratkaista pysähtymishetken t_z suhteen. Näin saadaan kiihdytys ja hidastus -tapaukselle ei-negatiivisena ratkaisuna

$$t_z = \frac{2\sqrt{as_{tot} + \frac{{v_0}^2}{2}} - v_0}{a} \tag{12}$$

Nopeusprofiilin maksiminopeustapauksessa kokonaismatka määräytyy summana $s_a + s_c + s_d = s_{tot}$. Kiihdytysvaiheen päättymishetki on t_a, josta samalla maksiminopeusvaihe alkaa. Hidastusvaihe alkaa hetkellä t_d, jolloin kokonaismatka voidaan ilmaista muodossa

$$s_a(t_a) + v_{max} \cdot (t_d - t_a) + \frac{1}{2}v_{max} \cdot (t_z - t_d) = s_{tot} \tag{13}$$

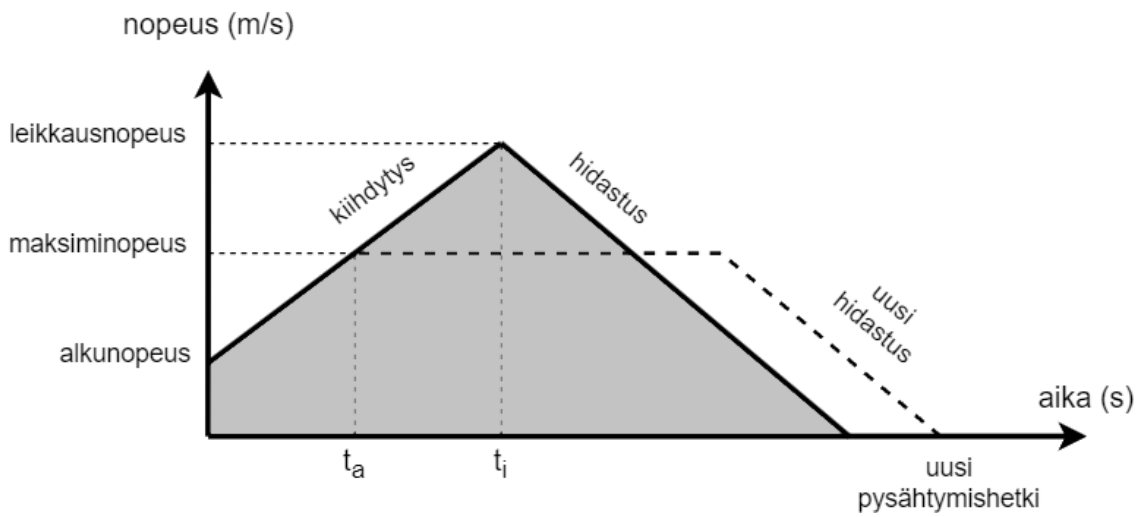
Kaavaan (13) sijoittamalla t_a, t_d ja b kaavojen (5), (6) ja (7) mukaisesti saadaan yhtälö, joka voidaan ratkaista pysähtymishetken t_z suhteen. Maksiminopeustapaukselle saadaan tällöin ei-negatiivisena ratkaisuna

$$t_z = \frac{{v_0}^2 - 2v_0 v_{max} + 2{v_{max}}^2 + 2as_{tot}}{2av_{max}} \tag{14}$$

Sijoittamalla pysähtymishetki t_z kaavaan (7) saadaan ratkaistua hidastusvaiheen vakiotermi b, jolla puolestaan voidaan määrittää kaavojen (4) ja (6) mukaiset hidastusvaiheen alkamishetket t_i ja t_d.

Nopeusprofiilien laskemiseksi toteutettu algoritmi yrittää ensi sijaisesti käyttää ajoneuvoille kiihdytys ja hidastus -tapausta. Kiihdytys- ja hidastusvaiheiden *leikkausnopeus* voi kuitenkin olla suurempi kuin ajoneuvon maksiminopeus. Leikkaus- ja maksiminopeudet saavutetaan toisaalta määritelmiensä mukaan hetkinä t_i ja t_a. Leikkaus- ja maksiminopeuksien vertailu on siis analoginen hetkien t_i ja t_a vertailulle. Leikkausnopeuden ollessa suurempi kuin maksiminopeus pätee $t_i > t_a$, jolloin täytyy käyttää maksiminopeustapausta. Hidastusvaihe täytyy määrittää uudelleen, jotta ajoneuvon edettävä matka pysyy samana molemmissa tapauksissa.

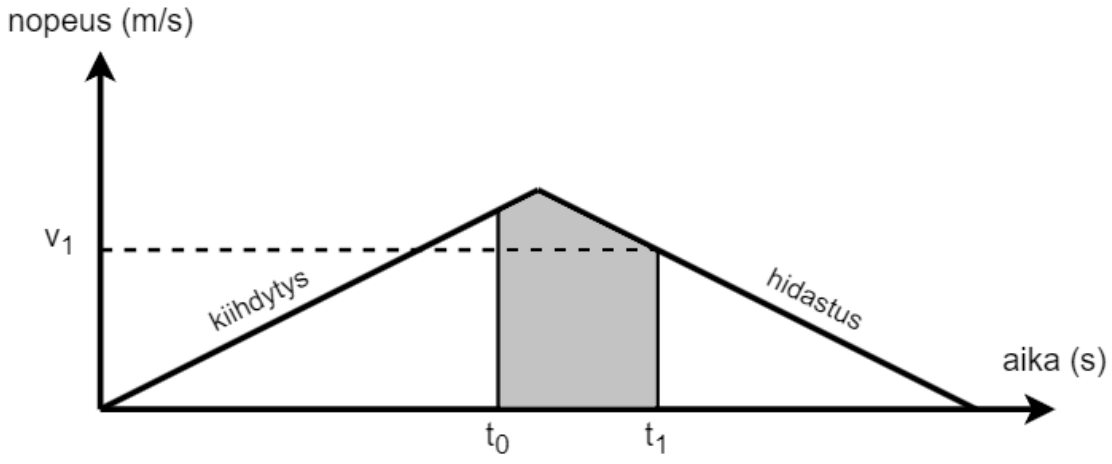
Kuvassa 31 on havainnollistettu nopeusprofiilin uudelleen määrittämistä. Kuvan kiihdytys- ja hidastusvaiheiden leikkausnopeus on suurempi kuin ajoneuvon maksiminopeus. Kuvaan on havainnollistettu myös hetket t_i ja t_a. Uudelleen määritetty maksiminopeustapaus on hahmoteltu kuvaan katoviivoin. Kuvasta havaitaan, että uusi pysähtymishetki siirtyy jonkin verran myöhemmäksi kuin alkuperäinen.



Kuva 31. *Havainnekuva nopeusprofiilin uudelleen määrittämisestä.*

Ajan suhteen mallinnetut nopeusprofiilit on valittu, koska niistä saadaan nopeus ajoneuvolle kullakin simulaation ajanhetkellä. Jos ollaan nopeusprofiilin ulkopuolella, nopeus on luonnollisesti nolla. Nopeusprofiilista voidaan siis määrittää seuraavaa simulaatioaikaa vastaava nopeus ja edettävä matka, kun tiedetään nykyinen simulaatioaika ja on määritetty simulointiaskeleen ajallinen kesto. Simulointiaskeleen aikana edetty matka saadaan nopeusprofiilista vastaavan aikavälin alle jäävänä pinta-alana.

Kuvassa 32 on esitetty havainnollistava esimerkki nopeusprofiilin käyttämisestä liikkeen simuloinnissa. Kuvassa on esitetty ajanhetket t_0 ja t_1, jotka kuvaavat nykyistä ja seuraavaa simulaatioaikaa. Harmaalla värillä on esitetty simulointiaskeleen aikana edettävä matka eli t_0 ja t_1 aikavälillä nopeusprofiilin alle jäävä pinta-ala. Kuvaan on myös havainnollistettu katkoviivalla seuraavaa simulaatioaikaa t_1 vastaava nopeus v_1. Simulointiaskel puolestaan voidaan ilmoittaa vakiomuutoksena $\Delta t = t_1 - t_0$. Toteutuksessa simulointiaskeleena on käytetty sekunnin kymmenesosaa. Simulointiaskel voi ajoittua samaan aikaan useampaan nopeusprofiilin vaiheeseen. Kuvan esimerkissä simulointiaskel ajoittuu samaan aikaan sekä kiihdytys- että hidastusvaiheeseen.

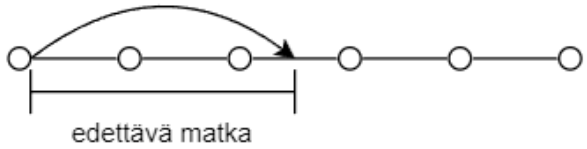


Kuva 32. *Esimerkki nopeusprofiilista tietyllä simulaatiohetkellä.*

Ajoneuvojen eteneminen segmentillä on toteutettu käyttäen reittipisteitä *RoutePoint*, jotka on esitetty lyhyesti edellisessä aliluvussa reittisuunnitelmien yhteydessä. Reittipiste on toteutettu seuraavana tietotyyppinä:

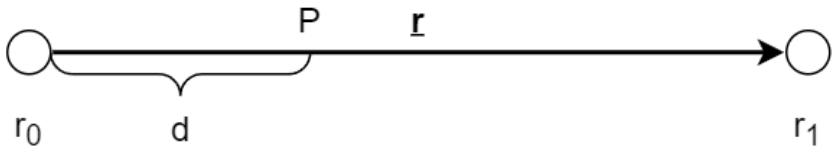
```
pub struct RoutePoint {
  pub index: usize,
  pub distance: f64,
}
```

Reittipisteen *index* kuvaa, missä segmentin pistejonon pisteessä ajoneuvo sijaitsee ja *distance* puolestaan etäisyyden tästä pisteestä seuraavaan nähden. Pistejonon väliin voidaan siis lineaarisesti interpoloida lisää pisteitä. Kuvassa 33 on esitetty esimerkki etenemisestä suoran segmentin pistejonossa. Kuvassa on havainnollistettu segmentin pistejonon yksittäiset pisteet ja edettävä matka segmentillä. Kuvan esimerkissä lähdetään ensimmäisestä pisteestä ja edetään kaksi eteenpäin, mutta jäädään kolmannen ja neljännen väliin. Väliin jäävä piste on ratkaistava, jotta ajoneuvon uusi sijainti tiedetään.



Kuva 33. *Esimerkki etenemisestä suoran segmentin pistejonossa.*

Pistejonon väliin interpoloitavat uudet pisteet voidaan ratkaista vektoreilla. Kuvassa 34 on havainnollistettu uuden pisteen P määrittämistä kahden tunnetun välille. Kuvassa on esitetty kaksi segmentin pistettä r_0 ja r_1 sekä niiden välinen vektori $\underline{r}$. Symboli d kuvaa etäisyyttä pisteestä r_0 uuteen pisteeseen P.



Kuva 34. *Uuden pisteen määrittäminen kahden tunnetun välille.*

Vektorilla $\underline{r}$ voidaan toisaalta lausua seuraavat $r_0 = r_0 + 0 \cdot \underline{r}$ ja $\underline{r_1} = r_0 + 1 \cdot \underline{r}$. Piste P voidaan siis esittää olevan etäisyyden d verran vektorin $\underline{r}$ suuntaan alkupisteestä r_0, jolloin saadaan seuraava kaava

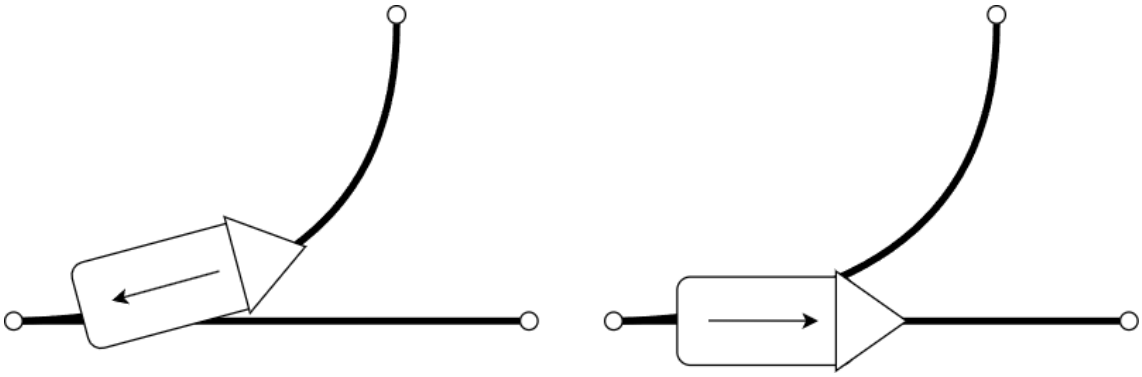
$$\underline{P} = r_0 + \frac{d}{\|\underline{r}\|}\underline{r} \tag{15}$$

Liikkeen simulointia varten on toteutettu myös ajoneuvojen paikallaan kääntyminen ja etenemissuunnan vaihto. Kääntymiskyky on toteutettu ajoneuvojen mallinnuksesta erillisenä asetuksena, joka määrittää, pystyvätkö simulointialustan ajoneuvot kääntymään paikallaan vai eivät. Kääntymiskyky on toteutettu seuraavana luettelotyyppinä:

```
pub enum TurnCapability {
  Curve,
  InPlace,
}
```

Luettelotyypin variantti *Curve* tarkoittaa, että ajoneuvot tarvitsevat mutkasegmentin kääntyäkseen ja *InPlace* puolestaan mahdollistaa paikallaan kääntymisen. Kääntymiskyvystä riippumatta ajoneuvot voivat saada reitinsuunnittelijalta käskyn vaihtaa etenemissuuntaa legien välillä, joskin mutkasegmentin kääntyäkseen tarvitsevat joutuvatkin näihin tilanteisiin useammin.

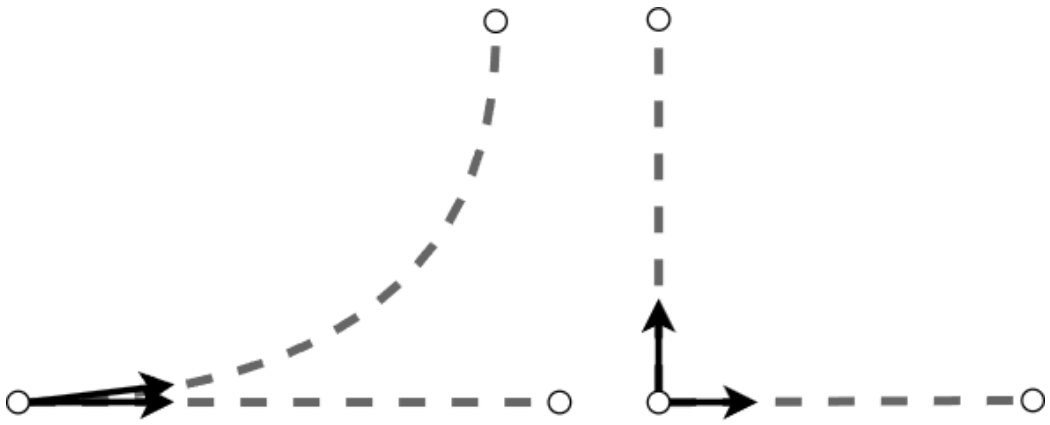
Kuvassa 35 on esitetty esimerkki etenemissuunnan vaihtotilanteesta, jossa ajoneuvo on vaihtamassa mutkasegmentiltä suoralle. Kuvassa on esitetty vasemmalla ajoneuvon peruutus mutkasegmentiltä ja oikealla eteenpäin jatkaminen suoralla. Nuoli kuvaa, mihin suuntaan ajoneuvo ajaa segmentillä. Ajoneuvon etupää on korostettu kuvaan erillisenä kolmiona, jotta etenemissuunnan vaihto hahmottuu. Kuvan tilanteesta voidaan samalla havaita, että segmenttien yhteyspisteessä täytyisi kääntyä paikallaan, jos ajoneuvo jatkaisikin peruuttaen suoralla segmentillä.



Kuva 35. *Esimerkki etenemissuunnan vaihdosta.*

Paikallaan kääntymistä ja etenemissuunnan vaihtoa varten on toteutettu *suunnan jatkuvuustarkastukset*, joilla segmenttien välisistä yhteyksistä tutkitaan, miten ajoneuvon etenemissuunta käyttäytyy segmentiltä toiselle vaihdettaessa. Suunnan jatkuvuustarkastukset on toteutettu laskemalla segmenttien väliset kulmat yhteyspisteissä ja määrittämällä kulmille arvot, joilla etenemissuunta pysyy samana, muuttuu päinvastaiseksi tai ajoneuvon täytyy kääntyä paikallaan. Segmenttien välinen kulma on määritetty vektorien välisenä kulmana. Vektori puolestaan on määritetty yhteyspisteestä lähimpään seuraavaan segmentin pistejonon pisteeseen nähden. Segmenttien alustuksesta ja yhteyspisteen geometriasta riippuen vektorin päätepiste on joko pistejonon toinen tai toiseksi viimeinen piste.

Kuvassa 36 on havainnollistettu segmenttien välisen kulman määrittämistä. Vasemmalla on esitetty suora – mutka ja oikealla suora – suora väliset tapaukset. Segmenttien välisistä yhteyspisteistä lähtevät vektorit on havainnollistettu nuolena. Segmenttien päät on esitetty kuvassa erillisinä pisteinä ja segmentin muoto on havainnollistettu harmaalla katkoviivalla. Vasemman puoleinen tapaus on sama kuin ylempänä esitetyssä kuvassa 35, jossa etenemissuunta vaihtuu päinvastaiseksi. Oikean puoleisessa tapauksessa puolestaan ajoneuvon on käännyttävä paikallaan riippumatta nykyisestä etenemissuunnasta.



Kuva 36. *Havainnekuva segmenttimuotojen välisestä kulmasta yhteyspisteessä.*

Suunnan jatkuvuustarkastukset on toteutettu ajokartan lataamisen yhteyteen. Tarkastukset suoritetaan samalla, kun segmenttien välillä todetaan yhteyksiä. Vaikka geometrisesti katsottuna suunnan jatkuvuustarkastuksissa on kyse segmenttien välisen kulman määrittämisestä, on toteutuksessa tarkastukset suoritettu erikseen jokaista simulointialustan segmenttimuotoa kohden. Toteutettuja segmenttimuotoja on kolme: suora, mutka ja S-mutka. Näin saadaan yhteensä kuusi segmenttimuotoyhdistelmää, kun kaksoiskappaleet poistetaan. Tarkastettavat segmenttimuotoyhdistelmät on listattu taulukossa 3.

Taulukko 3. *Suunnan jatkuvuustarkastusten segmenttimuotoyhdistelmät.*

Segmenttimuoto	**Segmenttimuoto**
Suora	Suora
Suora	Mutka
Suora	S-mutka
Mutka	Mutka
Mutka	S-mutka
S-mutka	S-mutka

Suunnan jatkuvuustarkastusten tuloksena saadaan segmenttien välisiin yhteyksiin tieto, miten ajoneuvon etenemissuunta käyttäytyy vaihdettaessa segmentiltä toiselle. Etenemissuunnan vaihtoa kuvaava tieto on toteutettu seuraavana luettelotyyppinä:

```
pub enum NextProgressDirection {
  Same,
  Opposite,
  TurnInPlace,
}
```

Same kuvaa tilannetta, jossa etenemissuunta voi pysyä samana ja *Opposite* puolestaan tarkoittaa, että etenemissuunta on vaihdettava päinvastaiseksi eli esimerkiksi eteenpäin ajava ajoneuvo lähteekin peruuttamaan. Viimeisenä on variantti *TurnInPlace*, jolloin ajoneuvon on käännyttävä paikallaan segmentiltä toiselle vaihdettaessa.

Paikallaan kääntyminen tai etenemissuunnan vaihto on otettava huomioon nopeusprofiilien laskemisessa, sillä näissä tilanteissa ajoneuvon on ensiksi pysähdyttävä, vaikka ajettavia legejä olisikin enemmän. Nopeusprofiileille on toteutettu tätä varten profiilin päättymistä kuvaava luettelotyyppi:

```
pub enum ProfileType {
  TillEnd,
  TillStop,
}
```

Variantti *TillEnd* tarkoittaa, että ajoneuvon nykyinen nopeusprofiili on laskettu päättymään samalla, kun reittisuunnitelman ajettavaksi päätetyt legit loppuvat. *TillStop* puolestaan päättyy pysähdykseen ennen kuin ajettavaksi päätetyt legit loppuvat. Luettelotyyppiä *ProfileType* käytetään siis määrittämään, voidaanko nykyinen nopeusprofiili korvata uudella, jos uudessa reittisuunnitelmassa onkin lisää ajettavaksi päätettyjä legejä. Ajoneuvon pysähtyminen paikallaan kääntymisen tai etenemissuunnan vaihdon yhteydessä voidaan tällöin toteuttaa käyttämällä varianttia *TillStop*, jolloin uusi nopeusprofiili lasketaan vasta pysähtymisen jälkeen.

6.6 Visualisointi

Simulointialustaan on toteutettu omat *visualisointi-* ja *julkaisijakomponentit*. Visualisointikomponentit ovat vastuussa jonkin tietyn simuloinnin tilan visualisoimisesta. Esimerkiksi reittisuunnitelmia varten on oma komponenttinsa, joka huolehtii kaikkien ajoneuvojen reittisuunnitelmien visualisoinnista. Vastaavasti ajokartan segmenttien ja törmäysvaroitusten visualisointiin on toteutettu omat komponenttinsa. Julkaisijakomponentit puolestaan ovat vastuussa *visualisointiviestien* julkaisusta ROSin viestikanaviin.

Jotta RViz-visualisointityökalulle voidaan ROSin viestikanavien kautta välittää simulointialustan tilaa, on täytynyt määritellä tarvittavat Rust-tietotyypit, jotka toteuttavat viestikanavissa käytettyjen viestityyppien rajapinnat. Taulukossa 4 on esitetty kaikki simulointialustaan toteutetut visualisoinnit sekä niitä vastaavat ROSin viestityypit ja julkaisijakomponentit.

Taulukko 4. *Toteutetut visualisoinnit sekä niitä vastaavat ROSin viestityypit ja julkaisijakomponentit.*

Toteutettu visualisointi	**ROSin viestityyppi**	**Julkaisijakomponentti**
Simulaation aika	Marker	MarkerPublisher
URDF-robottimalli	String	ModelPublisher
Ajoneuvojen sijainnit	TransformStamped	PositionPublisher
Ajoneuvojen tunnisteet	MarkerArray	MarkerArrayPublisher
Ajokartta ja segmentit	MarkerArray	MarkerArrayPublisher
Reittisuunnitelmat	MarkerArray	MarkerArrayPublisher
Törmäysvaroitukset	MarkerArray	MarkerArrayPublisher

Jokaista viestityyppiä kohden on toteutettu vastaava Rust-tietotyyppi, jota simulointialusta käyttää visualisointiviestien julkaisemiseksi. Esimerkiksi ajoneuvojen sijaintien julkaisemiseen on käytetty ROSin *TransformStamped*-viestityyppiä, jonka rajapintakuvaus on esitetty alla:

```
# The frame id in the header is used as the reference frame of this transform.
std_msgs/Header header

# The frame id of the child frame to which this transform points.
string child_frame_id

# Translation and rotation in 3-dimensions of child_frame_id from
# header.frame_id.
Transform transform
```

Viestityyppi koostuu otsakkeesta *header* ja koordinaatistosta *child_frame_id* sekä sen muunnoksesta *transform*. Muunnos kuvaa samaan aikaan ajoneuvon sijainnin eli tason pisteen ja mihin suuntaan ajoneuvon etupää osoittaa. ROSin koordinaatistojen ja muunnosten hallintaa on esitelty tarkemmin RVizin yhteydessä aliluvussa 4.3.

Otsakkeelle ja muunnokselle on ROSissa *TransformStampedin* tapaan määritelty omat viestityyppinsä, jotka on yksinkertaistamisen vuoksi jätetty määrittelemättä tässä yhteydessä. Useamman ajoneuvon sijainti voidaan lähettää viestityyppinä *TransformStampedSequence*, joka on yksinkertaisesti taulukko yksittäisiä *TransformStamped*-viestityyppejä. Ajoneuvojen sijaintien julkaisemiseksi on toteutettu seuraavat Rust-tietotyypit:

```
pub struct TransformStamped {
  pub header: MessageHeader,
  pub child_frame_id: String,
  pub transform: Transform,
}

pub type TransformStampedSequence = Vec<TransformStamped>
```

TransformStamped sisältää myös tietotyypit *MessageHeader* ja *Transform*, jotka on toteutettu vastaavien ROSin viestityyppien määrittelyjen mukaisesti. Tietotyypit on jätetty yksinkertaistamisen vuoksi määrittelemättä tässä yhteydessä.

Visualisointiviestien julkaisemiseksi tarvitaan julkaisija ja viestikanava ROSissa. ROS ja sen käyttäminen simulointialustassa on esitelty tarkemmin aliluvussa 4.2. Julkaisijan ja viestikanavan luomiseksi tarvitaan ROSin laskennallisen verkon solmu, jonka alaisena ne toimivat. Kirjasto ros2-client tarjoaa tietotyypit *Node*, *Publisher* ja *Topic* sekä metodit näiden luomiselle. *Node* vastaa solmua, *Publisher* julkaisijaa ja *Topic* viestikanavaa. Toteutetut julkaisijakomponentit huolehtivat edellä mainittujen tietotyyppien olioiden luomisesta. Tietotyypeistä keskeisin on *Publisher*, jonka kautta visualisointiviestit todellisuudessa julkaistaan. Esimerkiksi yllä esitettyä ajoneuvojen sijaintia varten on toteutettu julkaisijakomponentti *PositionPublisher*, joka luo seuraavan julkaisijan:

```
Publisher<TransformStampedSequence>
```

Luotu *Publisher* voi siis julkaista vain tietyn tyyppisiä visualisointiviestejä. Julkaisijakomponentit tarjoavat visualisointikomponenteille rajapinnan visualisointiviestien julkaisemiseksi. Toisin sanottuna julkaisijakomponentit toteuttavat julkisivu-mallin (engl. wrapper) ROSin käytön yksinkertaistamiseksi simulointialustassa.

Visualisointikomponentit voivat hyödyntää useita eri julkaisijakomponentteja visualisointiensa toteuttamiseksi. Esimerkiksi kaikki ajoneuvojen visualisoinnit on toteutettu yhdessä tietotyypissä *VehicleVisualizer*:

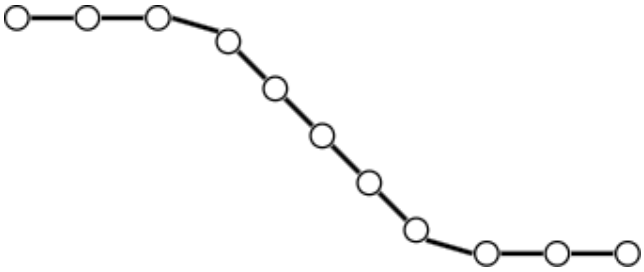
```
pub struct VehicleVisualizer {
  id_publisher: MarkerArrayPublisher,
  position_publisher: PositionPublisher,
  model_publisher: ModelPublisher,
  urdf: String,
}
```

VehicleVisualizer toteuttaa siis ajoneuvojen tunnisteiden, sijaintien ja robottimallin visualisoinnit. Kenttään *urdf* on tallennettu robottimallin merkkijonomuotoinen URDF-kuvaus, joka voidaan antaa robottimallin julkaisijakomponentille *model_publisher*. Ajoneuvojen URDF-robottimalli on esitetty tarkemmin aliluvussa 6.1.

Simulointialustassa käytetyin ROSin viestityyppi on *Marker*, jolla voidaan RVizissä visualisoida useita perusmuotoja kuten esimerkiksi kuutioita, palloja, lieriöitä ja suoria. Perusmuotojen lisäksi *Markerilla* voidaan visualisoida tekstejä, jotka RVizissä näkyvät suunnattuna käyttäjälle kameran kuvakulmasta riippumatta. Mainituista perusmuodoista on käytetty suoria segmenttien, reittisuunnitelmien ja törmäysvaroitusten visualisoinnissa, jotka hyödyntävät listamuotoa (engl. line list). Listamuodossa annetaan joukko pistepareja, joiden välille halutaan muodostaa murtoviiva. Kuvassa 37 on esitetty esimerkki tekstin ja murtoviivan visualisoinnista. Teksti on esitetty mustalla värillä ja murtoviiva punaisella.

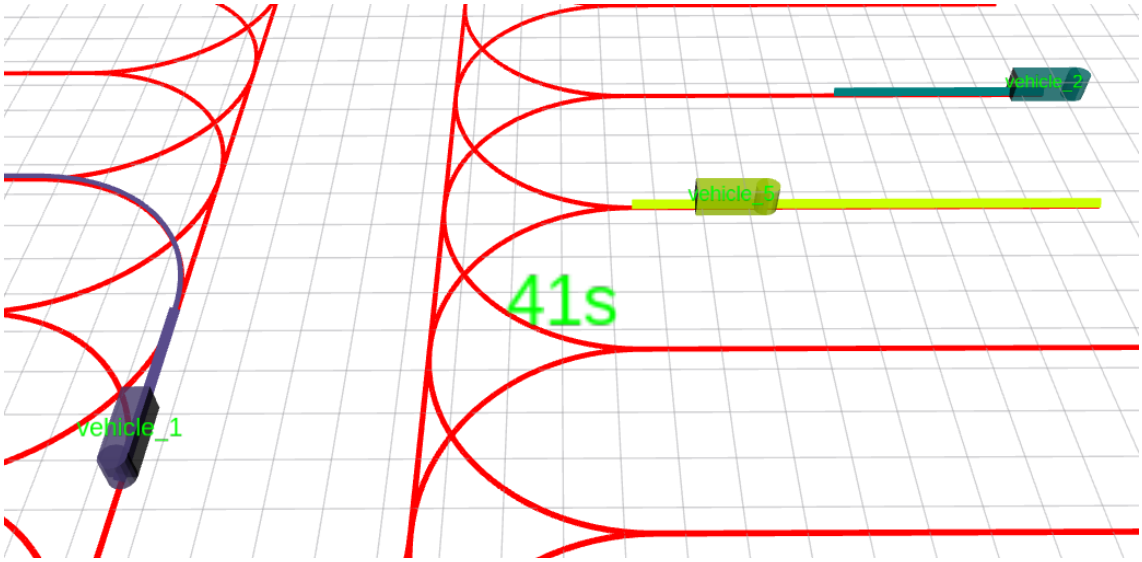
Kuva 37. *Esimerkki tekstin ja murtoviivan visualisoinnista.*

Listamuoto sopii segmenttien visualisointiin, kunhan segmenttien pisteet on kerätty pistepareiksi. Kerätyt pisteparit muodostavat kokonaisuutena segmentin visualisoinnin. Kuvassa 38 on havainnollistettu murtoviivan muodostamista S-mutkasegmentin tapauksessa. Kuvaan on havainnollistettu segmentin pistejonon pisteet ja janat niiden välillä.



Kuva 38. *Havainnekuva S-mutkasegmentin murtoviivaesityksestä.*

Samaa pisteiden keräys -menetelmää hyödynnetään myös reittisuunnitelmien tapauksessa. Tällöin kerätään erikseen ajettavaksi päätettyjen ja suunniteltujen legien pisteet, jotka julkaistaan omina visualisointiviesteinään. *Marker*-viestityypillä on toisaalta visualisoitu myös ajoneuvojen tunnisteet, törmäysvaroitustekstit ja simulaatioaika. Nämä visualisoinnit on toteutettu aiemmin mainittuina teksteinä. Kuvassa 39 on esitetty esimerkki ajoneuvojen tunnisteiden ja simulaatioajan tekstimuotoisesta visualisoinnista. Kuvassa on esitetty kolme ajoneuvoa tunnisteteksteineen. Simulaatioaika esitetään RVizin tasoruudukon keskellä.



Kuva 39. *Esimerkki ajoneuvojen tunnisteiden ja simulaatioajan tekstimuotoisista visualisoinneista Marker-viestityypillä.*

6.7 Jatkokehitys

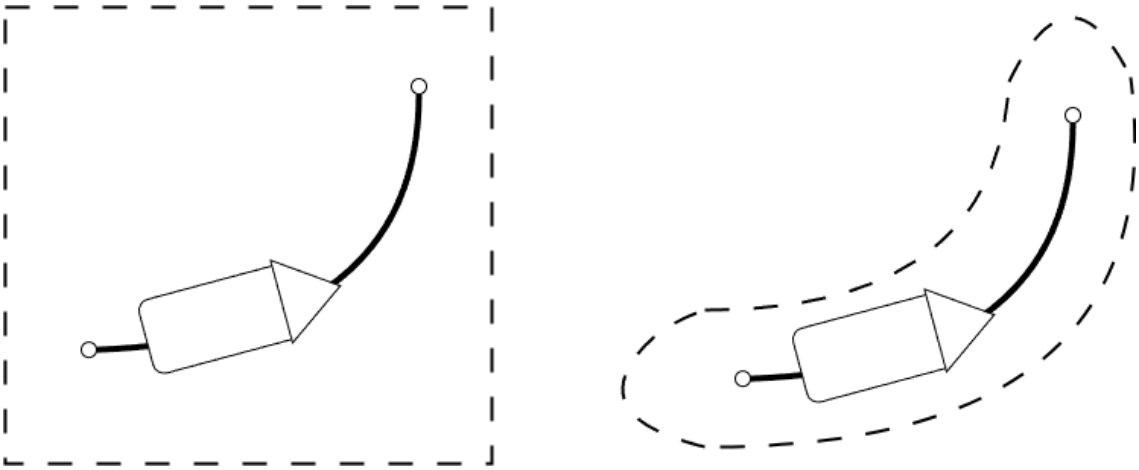
Simulointialustan tukea erilaisille kokoonpanoille voisi parantaa. Toteutetussa simulointialustassa voi olla kerrallaan vain yhdentyyppisiä ajoneuvoja, joilla kaikilla on samat kääntymis- ja kiihdytyskyvyt, maksiminopeudet ja mittasuhteet. Nykyiset asetukset on määritetty vakiotiedostossa, joten asetusten muuttaminen vaatii ohjelman kääntämisen uudelleen. Simuloinnin kokoonpanon voisi mahdollisesti siirtää omaan asetustiedostoonsa, joka luetaan esimerkiksi simulointiskenaarion lataamisen yhteydessä.

Nykyisessä toteutuksessa simuloinnin ja visualisoinnin päivitystaajuudet ovat samat. Toisin sanottuna simuloinnin tilaa julkaistaan visualisoitavaksi jokaisen simulointiaskeleen jälkeen. Jatkokehityksenä simuloinnin ja visualisoinnin voisi erottaa toisistaan, jolloin simuloinnin tarkkuutta voitaisiin kasvattaa ja samalla visualisoinnin vähentää. Toisaalta nykyisessä toteutuksessa visualisointikomponentit ja niiden toiminta on liitetty suoraan simuloinnin sekaan sopiviin kohtiin. Visualisoinnin voisi siis erottaa simulaattorista kokonaan erilliseksi kokonaisuudeksi, jolloin simuloinnin tilaa voitaisiin välittää visualisoitavaksi keskitetyn rajapinnan yli.

Segmenttien visualisoinnit on toteutettu erillisinä kokonaisuuksina, jolloin jokaiselle visualisoinnille on oma ROS-solmunsa ja viestikanavansa. Segmenteillä voi toisaalta olla samaan aikaan useita eri visualisointeja kuten ajokartta, päällekkäiset ajoneuvojen reittisuunnitelmat, segmenttien korostukset ja törmäysvaroitukset. Näistä visualisoinneista, erityisesti jatkuvasti muuttuvista reittisuunnitelmista, aiheutuu kuormitusta, kun useita visualisointiviestejä julkaistaan ROSin yli. Varsinkin simuloinnin kulun nopeutuessa voi RViz jäädä jälkeen julkaistuissa visualisointiviesteissä. Keskeisenä parannuksena voisi harkita segmenttivisualisointien yhdistämistä yhdeksi kokonaisuudeksi, jolloin vältyttäisiin ylimääräisten ROS-solmujen sekä viestikanavien luomiselta, ja siten vähennettäisiin samalla julkaistavien visualisointiviestien määrää.

Visualisointia voisi jatkokehittää myös ajoneuvojen URDF-robottimallin ja kartanpiirron osalta. URDF-robottimallia voisi parantaa ja hienontaa sekä mahdollisesti toteuttaa 3D-mallin esimerkiksi robottitrukista. Tiled-kartanpiirtotyökalu on toiminut simulointialustan kehitysvaiheessa melko hyvin pohjan luomiseksi, mutta se ei todennäköisesti ole paras työkalu varsinkaan monimutkaisten ajokarttojen luontiin. Siksi tulisikin löytää parempi piirtotyökalu, jolla erilaisia ajokarttoja ja segmenttimuotoja voitaisiin luoda helpommin. Samalla nykyistä kartanpiirtologiikkaa tulisi muokata yhteensopivaksi uuden työkalun osalta ja mahdollisesti poistaa nykyiset toteutetut segmenttimuodot kokonaan.

Reitinsuunnittelun testaamisen kannalta tulisi parantaa nykyistä suorakaiderajausmenetelmää, jota on käytetty leikkaavien segmenttien löytämiseksi ajokartasta. Suorakaiderajaus tuottaa helposti erityisesti mutkasegmenttien kohdalla vääriä tuloksia, mikä aiheuttaa turhia keskeytyksiä simuloinnin kulussa, kun käyttäjälle näytetään törmäysvaroituksia. Suorakaiderajaus tulisikin korvata jollakin tarkemmalla arviolla ajoneuvon segmentillä viemästä tilasta. Kuvassa 40 on esitetty esimerkki suorakaiderajauksesta ja tarkemmasta arviosta. Kuvassa on esitetty ajoneuvo ja mutkasegmentti. Vasemmalla on esitetty suorakaiderajaus ja oikealla tarkempi arvio ajoneuvon viemästä tilasta segmentillä.



Kuva 40. *Esimerkki suorakaiderajauksesta ja tarkemmasta arviosta.*

7. YHTEENVETO

Robottiajoneuvoryhmän reititys on luonteeltaan optimointiongelma, johon on kehitetty yhä monimutkaisempia reititysalgoritmeja. Optimointiongelmasta riippumatta reitinsuunnittelun keskeisenä tavoitteena on konfliktiton reititys kaikille ryhmän robottiajoneuvoille. Simulointi on osoittautunut hyväksi keinoksi testata ja arvioida reitinsuunnittelun onnistumista. Simuloinnissa keskeinen tekijä on mallinnuksen tarkkuus, sillä simulointi on vain yhtä tarkka kuin sitä varten luotu malli. Visualisointi on osoittautunut välttämättömäksi simuloinnissa, sillä se auttaa paremmin hahmottamaan monimutkaisia kokonaisuuksia.

Tässä diplomityössä suunniteltiin ja toteutettiin Rust-ohjelmointikielellä simulointialusta, joka hyödyntää visualisoinnin toteuttamiseksi ROSia ja RViz-visualisointityökalua. Ajokarttojen piirron toteuttamiseen käytettiin Tiled-nimistä 2D-tasoeditoria. ROSin käyttämiseksi simulointialustassa hyödynnettiin toimeksiantoyrityksen itse toteuttamia Rust-kirjastoja ros2-client ja RustDDS.

Simulointialustan robottiajoneuvoille mallinnettiin tunnisteet, maksiminopeudet sekä kiihdytys- ja hidastuskyvyt. Ajoneuvoille luotiin yksinkertaistettu 3D-robottimalli, jossa ajoneuvon runkona oli suorakulmainen särmiö ja takapuskurina ympyrälieriö. Ajokartan piirtoon toteutettiin kolme segmenttimuotoa: suora, mutka ja S-mutka. Segmentit mallinnettiin pistejonoina ja lisäksi niille toteutettiin valinta ajosuunnalle joko yksi- tai kaksisuuntaisena.

Simulointialustaan toteutettiin mahdollisuus luoda yksilöllisiä skenaarioita, jotka koostuvat ajokartasta, ajoneuvoista ja ajotehtävistä. Reitinsuunnittelua varten toteutettiin rajapinta, jonka kautta reitinsuunnittelijat voivat kiinnittäytyä simulointialustaan. Keskeisenä vaatimuksena oli toteuttaa reittisuunnitelmien järkevyystarkastukset ja varoittaa erikseen mahdollisista törmäystilanteista ajoneuvojen välillä.

Liikkeen simulointia varten toteutettiin ajan suhteen lineaariset nopeusprofiilit, joita varten täytyi johtaa laskukaavat. Liikkeen simulointiin toteutettiin myös ajoneuvojen etenemissuunnan vaihto ja paikallaan kääntyminen. Visualisoinnit toteutettiin välittämällä simulointialustan tilaa ROSin viestikanavien kautta RVizille. Visualisointiviestien julkaisemiseksi toteutettiin julkaisijakomponentit ja Rust-tietotyyppivastineet ROSin viestityyppimäärittelyille. Varsinaisia visualisointeja varten toteutettiin erilliset visualisointikomponentit, jotka hyödyntävät julkaisijakomponentteja visualisointiensa toteuttamiseksi.

Toteutetun simulointialustan tavoitteena oli soveltuvuusselvitys ja pohjan rakentaminen myöhempää käyttöä varten, joten tiettyjen osien toteutus pidettiin mahdollisimman yksinkertaisena. Jatkokehityksen kannalta nousikin keskeisesti esiin nykyisen mallinnuksen hienontaminen, uusien parempien mallien toteuttaminen ja visualisoinnin parantaminen. Työssä kuitenkin näytettiin, että valituilla teknologioilla oli mahdollista rakentaa ja toteuttaa vaatimusten mukainen simulointialusta.

LÄHTEET

[1] Ullrich, Günter, and Paul a Kachur. “The History of Automated Guided Vehicle Systems.” Automated Guided Vehicle Systems, Springer Berlin / Heidelberg, 2015, pp. 1–14, https://doi.org/10.1007/978-3-662-44814-4_1.

[2] Zhang, Jianqi, et al. “Automated Guided Vehicles and Autonomous Mobile Robots for Recognition and Tracking in Civil Engineering.” Automation in Construction, vol. 146, 2023, p. 104699–, https://doi.org/10.1016/j.autcon.2022.104699.

[3] TermWiki > leg (route). Saatavissa: https://en.termwiki.com/EN/leg_(route) (viitattu 10.11.2023)

[4] Fazlollahtabar, Hamed., and Mohammad. Saidi-Mehrabad. *Autonomous Guided Vehicles Methods and Models for Optimal Path Planning*. 1st ed. 2015., Springer International Publishing, 2015, https://doi.org/10.1007/978-3-319-14747-5.

[5] Aziez, Imadeddine, et al. “Fleet Sizing and Routing of Healthcare Automated Guided Vehicles.” *Transportation Research. Part E, Logistics and Transportation Review*, vol. 161, 2022, pp. 102679-, https://doi.org/10.1016/j.tre.2022.102679.

[6] Polten, Lukas, and Simon Emde. “Scheduling Automated Guided Vehicles in Very Narrow Aisle Warehouses.” *Omega (Oxford)*, vol. 99, 2021, pp. 102204-, https://doi.org/10.1016/j.omega.2020.102204.

[7] Rashidi, Hassan, and Edward P. K. Tsang. *Port Automation and Vehicle Scheduling: Advanced Algorithms for Scheduling Problems of AGVs*. 3rd ed., CRC Press, Taylor & Francis Group, 2022.

[8] Kabir, Qazi Shaheen, and Yoshinori Suzuki. “Comparative Analysis of Different Routing Heuristics for the Battery Management of Automated Guided Vehicles.” *International Journal of Production Research*, vol. 57, no. 2, 2019, pp. 624–41, https://doi.org/10.1080/00207543.2018.1475761.

[9] Maria, Anu. “Introduction to Modeling and Simulation.” *Proceedings of the 29th Conference on Winter Simulation*, IEEE Computer Society, 1997, pp. 7–13, https://doi.org/10.1145/268437.268440.

[10] Bandyopadhyay, Susmita, and Ranjan Bhattacharya. *Discrete and Continuous Simulation: Theory and Practice*. Taylor & Francis, 2014, https://doi.org/10.1201/b17127.

[11] Steinhauser, M. O. *Computer Simulation in Physics and Engineering*. Walter de Gruyter GmbH & Co. KG, 2013, https://doi.org/10.1515/9783110256062.

[12] Rohrer, M. W. “Seeing Is Believing: The Importance of Visualization in Manufacturing Simulation.” *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*, vol. 2, IEEE, 2000, pp. 1211–16 vol.2, https://doi.org/10.1109/WSC.2000.899087.

[13] Gazebo simulator. Saatavissa: https://gazebosim.org/home (viitattu 16.11.2023)

[14] FlexSim: 3D Simulation Modeling and Analysis Software. Saatavissa: https://www.flexsim.com/flexsim/ (viitattu 16.11.2023)

[15] López, Joaquín, et al. “A Simulation and Control Framework for AGV Based Transport Systems.” *Simulation Modelling Practice and Theory*, vol. 116, 2022, pp. 102430-, https://doi.org/10.1016/j.simpat.2021.102430.

[16] Swisslog: Automated guided vehicles AGV: The mobile solution of the future. Saatavissa: https://www.swisslog.com/fi-fi/products-systems-solutions/transport/agv-automated-guided-vehicles (viitattu 21.11.2023)

[17] Rust Programming Language. Saatavissa: https://www.rust-lang.org/ (viitattu 22.11.2023)

[18] Bugden, William, and Ayman Alahmar. “Rust: The Programming Language for Safety and Performance.” *ArXiv.Org*, 2022, https://doi.org/10.48550/arxiv.2206.05503.

[19] The Rust Programming Language: Rust Book. Saatavissa: https://doc.rust-lang.org/book/ (viitattu 22.11.2023)

[20] Macenski, Steven, et al. “Robot Operating System 2: Design, Architecture, and Uses in the Wild.” *Science Robotics*, vol. 7, no. 66, 2022, pp. eabm6074–eabm6074, https://doi.org/10.1126/scirobotics.abm6074.

[21] Rico, Francisco Martín. *A Concise Introduction to Robot Programming with ROS2*. 1st ed., vol. 1, CRC Press, 2023, https://doi.org/10.1201/9781003289623.

[22] About The Interface Definition Language Specification Version 4.2. Saatavissa: https://www.omg.org/spec/IDL/4.2/About-IDL (viitattu 24.11.2023)

[23] DDS Foundation: What is DDS? Saatavissa: https://www.dds-foundation.org/what-is-dds-3/ (viitattu 24.11.2023)

[24] ROS2 Documentation: Iron > Concepts > Intermediate Concepts > Different ROS 2 middleware vendors. Saatavissa: https://docs.ros.org/en/iron/Concepts/Intermediate/About-Different-Middleware-Vendors.html (viitattu 14.4.2024)

[25] The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPSTM) Specification. Saatavissa: https://www.omg.org/spec/DDSI-RTPS/2.5/PDF (viitattu 14.4.2024)

[26] eProsima Fast DDS Documentation. Saatavissa: https://fastdds.docs.eprosima.com/en/latest/ (viitattu 14.4.2024)

[27] RustDDS. Saatavissa: https://github.com/jhelovuo/RustDDS (viitattu 23.11.2023)

[28] Ros2-client: ROS2 client library based on RustDDS. Saatavissa: https://github.com/jhelovuo/ros2-client/ (viitattu 23.11.2023)

[29] ROS 3D Robot Visualizer. Saatavissa: https://github.com/ros-visualization/rviz (viitattu 23.4.2024)

[30] ROS2 Documentation: Iron > Tutorials > Intermediate > URDF. Saatavissa: https://docs.ros.org/en/iron/Tutorials/Intermediate/URDF/URDF-Main.html (viitattu 27.11.2023)

[31] ROS2 Documentation: Iron > Tutorials > Intermediate > tf2. Saatavissa: https://docs.ros.org/en/iron/Tutorials/Intermediate/Tf2/Tf2-Main.html (viitattu 27.11.2023)

[32] Tiled: Mapeditor. Saatavissa: https://www.mapeditor.org/ (viitattu 22.11.2023)

[33] Tiled Documentation. Saatavissa: https://doc.mapeditor.org/en/stable/ (viitattu 28.11.2023)