Arttu Salin

Liikennesimulaattorin integrointi pelimoottoriin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

17.11.2022

Tiivistelmä

Tekijä:	Arttu Salin
Otsikko:	Liikennesimulaattorin integrointi pelimoottoriin
Sivumäärä:	35 sivua
Aika:	17.11.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Insinöörityössä kehitettiin integraatio SUMO-liikennesimulaattorin ja Unity-pelimoottorin välille hyödyntäen liikennesimulaattorin TraCI-rajapintaa. Työ toteutettiin C#-ohjelmointikielen ja avoimen lähdekoodin TraCI.NET-ohjelmointikirjaston avulla.

Autonomisella liikenteellä on eri luokituksia, jotka kuvaavat yksittäisen ajoneuvon kykyä suoriutua ajosta ilman kuljettajan toimia. Täysin autonomisia ajoneuvoja simuloitiin insinöörityössä SUMO-liikennesimulaattorin avulla, ja niiden tiedot tuotiin Unity-pelimoottorilla toteutettuun liikenteenvalvontasovellukseen.

Projektissa keskityttiin kehittämään liikenteenvalvontasovelluksen kannalta kaikki riittävät toiminnot sisältävä integraatio, jonka avulla jatkokehitys olisi helpompaa ja nopeampaa. Integraatiossa päätavoitteena oli liikennesimulaattorin ja pelimoottorin samanaikainen käynnistäminen, yhteyden luominen Unityn ja SUMO:n välille TraCI-rajapinnan avulla sekä rajapinnan komentojen käytön mahdollistaminen. Ajoneuvojen lisäys simulaation määrittämällä aikaleimalla ja niiden tietojen haun automaatio helpottavat integraation käyttöä. Lisäksi osana insinöörityötä kehitettiin Unity-pelimoottoriin työkalu, jolla voidaan hakea simuloitavan alueen karttakuva.

Insinöörityön lopputuloksena saatiin käyttövalmis integraatio, jolla toteutettiin Metropolia Ammattikorkeakoulun SAM (Smart Autonomous Mobility) -hankkeen liikenteenvalvontasovellus. Insinöörityön ja sen avulla kehitetyn sovelluksen avulla hankkeessa kyetään selvittämään autonomisen liikenteen valvontasovelluksien käyttäjiin kohdistuvia vaatimuksia.

Avainsanat: itsenäinen liikenne, liikennesimulaattori, Unity

Abstract

Author:	Arttu Salin
Title:	Integration of a traffic simulator to a game engine
Number of Pages:	35 pages
Date:	17 November 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisor:	Miikka Mäki-Uuro, Senior Lecturer

This final year project is about the development of an integration between the SUMO traffic simulator and the Unity game engine, using the C# programming language and the open source TraCI.NET programming library.

Autonomous traffic has different levels of classification that describe the ability of a single vehicle to drive independently without driver intervention. Fully autonomous vehicles were simulated in the project using the SUMO traffic simulator and they were visualized in a traffic control application implemented using the Unity game engine.

The final year project focused on developing a complete integration with all the necessary functionalities for the application to function and to help with further development. The focus of the integration was starting the traffic simulator and game engine in parallel, to establish connection between SUMO and Unity through the TraCI interface, and to allow the use of TraCI commands from Unity. To ease development with the integration, adding vehicles at the correct time and the retrieval of their data from the simulation was automated. In addition, an extension for the Unity game engine was developed for fetching the map view of the simulated area.

The result of the final year project was a ready-to-use integration that was used to implement the traffic control application of the SAM (Smart Autonomous Mobility) project of the Metropolia University of Applied Sciences. The engineering work and the resulting application will help the project to identify the demands on the users of autonomous traffic monitoring applications.

Keywords: autonomous mobility, traffic simulator, Unity

Sisällys

Lyhenteet

1 Johdanto 1

2 Autonominen liikenne 2

3 SUMO-liikennesimulaattori 4

3.1 Kartta 5
3.2 Ajoneuvot 7
3.3 TraCI-rajapinta 8
3.4 TraCI.NET-kirjasto 9

4 Unity-pelimoottori 10

5 Liikennesimulaattorin integraatio 12

5.1 Työmenetelmät 12
5.2 Integraation rakenne 14
5.3 SUMO:n ja Unityn yhteys 18
5.4 Ajoneuvojen liikkuminen 22
5.5 Kartan luonti 27
5.6 Jatkokehitys 32
5.7 Loppuanalyysi 33

6 Yhteenveto 34

Lähteet 36

1 Johdanto

Insinöörityötä tehtiin osana Metropolia Ammattikorkeakoulun Smart Autonomus Mobility (SAM) -hanketta, jossa tavoitteena oli kehittää autonomisen liikenteen valvontasovellus. Työn tarkoituksena oli mahdollistaa liikenteenvalvontasovelluksen toteutus, jossa autonominen liikenne tuotetaan liikennesimulaattorilla ja pelimoottoria käytetään itse valvontasovelluksen tuottamiseen ja ajoneuvoille ilmaantuvien ongelmatilanteiden luomiseen. Näitä ongelmatilanteita käytettäisiin hankkeessa tutkimaan autonomisten ajoneuvojen hallinnan aiheuttamaa kuormitusta ja täten sitä, kuinka montaa ajoneuvoa yksittäinen henkilö voi hallita.

Insinöörityön tavoite, eli liikennesimulaattorin integrointi pelimoottoriin, toteutettaisiin käyttäen avoimen lähdekoodin SUMO-liikennesimulaattoria, kolmiulotteista Unity-pelimoottoria ja avoimen lähdekoodin ohjelmakirjastoa TraCI.NET, joka mahdollistaa liikennesimulaattorin hallintaan käytettävän rajapinnan käytön C#-ohjelmointikielellä, jota myös Unity-pelimoottori käyttää.

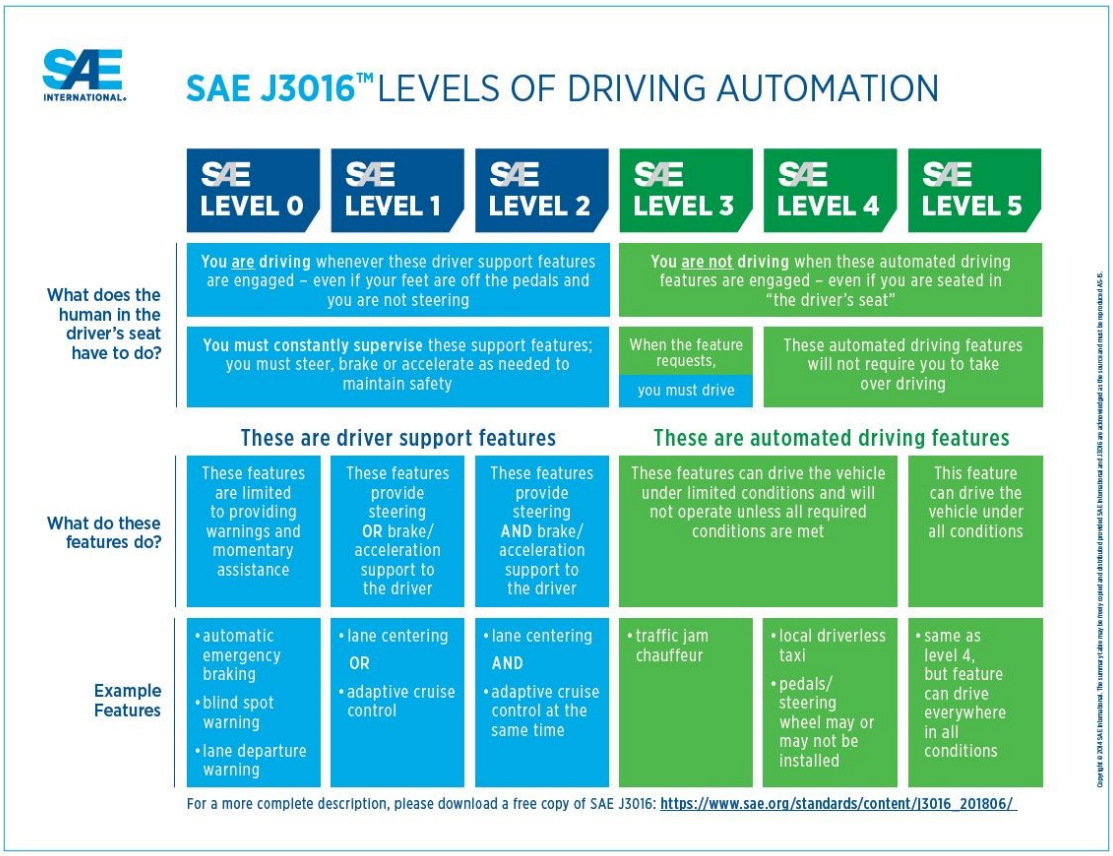
Metropolia Ammattikorkeakoulun SAM-hankkeesta on tehty myös muita insinööritöitä, jotka keskittyivät esimerkiksi liikenteenvalvontasovelluksen käyttöliittymään, sen aliohjelmien käyttöön ja visualisointiin sekä toteutuksessa käytettyyn tietokantaan.

Luvussa 2 käsitellään autonomisen liikenteen määrittelytasoja ja perehdytään projektin hankkeeseen. Luvussa 3 esitellään SUMO-liikennesimulaattori sekä sen käyttämää hallintarajapintaa. Luvussa 4 perehdytään projektissa käytettyyn Unity-pelimoottoriin, ja luvussa 5 käydään läpi kehitystyötä, sen käytäntöjä sekä integraation toteutusta.

2 Autonominen liikenne

Autonomisen ajoneuvon määritelmä on Traficomin mukaan seuraava: “Autonominen ajoneuvo (engl. autonomous vehicle) kykenee suoriutumaan ajotehtävästä ilman kuljettajaa ja ilman yhteyttä muihin ajoneuvoihin tai infrastruktuuriin” (1). Autonominen liikenne jaetaan yhdysvaltalaisen autoalan standardisointijärjestön SAE Internationalin luokituksen mukaan kuuteen eri luokkaan ajoneuvon ominaisuuksien ja vaatimusten perusteella (kuva 1) (2). Näistä ensimmäinen eli luokka nolla ei sisällä kuin yksinkertaisia avustavia järjestelmiä ja ajoneuvon ohjaaminen on täysin kuljettajan vastuulla. Tällaisia järjestelmiä ovat esimerkiksi automaattinen hätäjarrutusjärjestelmä ja katvealueen valvontajärjestelmä. Luokan yksi ajoneuvossa on ainakin yksi kuljettajaa auttava järjestelmä. Tästä luokasta SAE:n esimerkkinä on kaistallapitoavustin tai adaptiivinen vakionopeudensäädin. Luokan kaksi ajoneuvoissa järjestelmä kykenee hallitsemaan ajoneuvon ohjausta ja nopeutta, mutta kuten luokan yksi ja nolla ajoneuvoissa, täytyy kuljettajan olla valmiina hallitsemaan ajoneuvoa jokaisessa tilanteessa. Tämän tason ratkaisuja on jo saatavissa kaupallisissa ajoneuvoissa. (2.)

Luokissa kolme, neljä ja viisi ei henkilö enää vastaa ajoneuvon kuljettamisesta, kun näiden tasojen toiminnot ovat aktiivisina. Luokassa kolme ajoneuvo kykenee automaattiseen ohjaukseen, eli ajoneuvon järjestelmät kykenevät tekoälyä hyödyntämällä selviämään lähestulkoon kaikista ajon osa-alueista ilman kuljettajan osallistumista. Kuitenkin kuljettajan on oltava valmiina tarpeen vaatiessa ohjaamaan ajoneuvoa. Luokassa neljä on kyse jo korkean tason automaatiosta, jossa kuljettajan ei tarvitse puuttua ajoneuvon toimintaan, vaan sen järjestelmät kykenevät selviämään yleisessä ajossa tai jopa vikatilanteen sattuessa. Tämä toiminnallisuus voi kuitenkin olla rajoitettu vain tiettyihin olosuhteisiin. Luokka viisi on täysin itse ajava, eli ihmisen ei tarvitse huolehtia ajoneuvon hallinnasta missään tilanteessa. Ajoneuvossa ei myöskään siten tarvitse olla polkimia tai ohjauspyörää. Tällä hetkellä autonominen liikenne on nopeasti kehittyvä ala, johon riittää paljon kiinnostusta niin valtiollisilta kuin kaupallisiltakin toimijoilta. (2.)



Kuva 1. SAE Internationalin autonomisten ajoneuvojen luokittelu (3).

SAM-hanke (Smart Autonomus Mobility) eli älykäs autonominen liikenne on Metropolia Ammattikorkeakoulun kehittämä hanke, jonka tavoitteena on ”luoda uutta liiketoimintaa sekä kehittää jo olemassa olevaa liiketoimintaa autonomisten ajoneuvojen ja liikenteen valvonnan alalle” (4). Tämän lisäksi tavoitteisiin kuuluu kestävän liikenneinfrastruktuurin ja sen suunnittelun edistäminen sekä autonomisten ajoneuvojen käytön saavutettavuuden tarkasteleminen.

Hankkeen tuottamille tutkimustuloksille on ollut suurta kysyntää muun muassa Traficomilta, joka on ollut mukana lukuisissa autonomisen liikenteen hankkeissa, joka aktiivisesti kannustaa tällaisten hankkeiden kehitystä. Myös tarvittavat luvat autonomisen liikenteen kokeiluihin ovat haettavissa Traficomilta (5). Hankkeen keskiössä on autonomisten ajoneuvojen etähallinnan simulointi liikenteenvalvontasovelluksen avulla, jotta kyetään luomaan ympäristö, jossa voidaan simuloida erilaisia ongelmatilanteita. Näiden ongelmatilanteiden avulla

testaan etähallintaa suorittavan työntekijän kykyä selvitä haasteista sekä kartoittamaan, kuinka montaa ajoneuvoa yksittäinen työntekijä voi samanaikaisesti valvoa. Tähän kokonaisuuteen sisältyvät myös fysiologiset ja kognitiiviset mittaukset, joilla arvioidaan testihenkilön kokemaa rasitusta. (4.) Tähän insinöörityöhön kuului kuitenkin vain liikenteenvalvontasovelluksen kehittäminen.

3 SUMO-liikennesimulaattori

Insinöörityössä valittu liikennesimulaattori on Eclipse SUMO. SUMO eli Simulation of Urban Mobility on avoimen lähdekoodin liikennesimulaattori, ja sen ensimmäinen versio on julkaistu vuonna 2001. Sen kehityksestä vastaa Saksan liikennejärjestelmien instituutti (saksaksi ”Institut für Verkehrssystemtechnik”), joka on Saksan ilmailu- ja avaruuskeskuksen alainen järjestö, jonka tehtävänä on liikenteen, auto- ja rautatiejärjestelmien tutkimus- ja kehitystyö. Vuodesta 2017 se on ollut Eclipse Foundationin alainen projekti. (6; 7.)

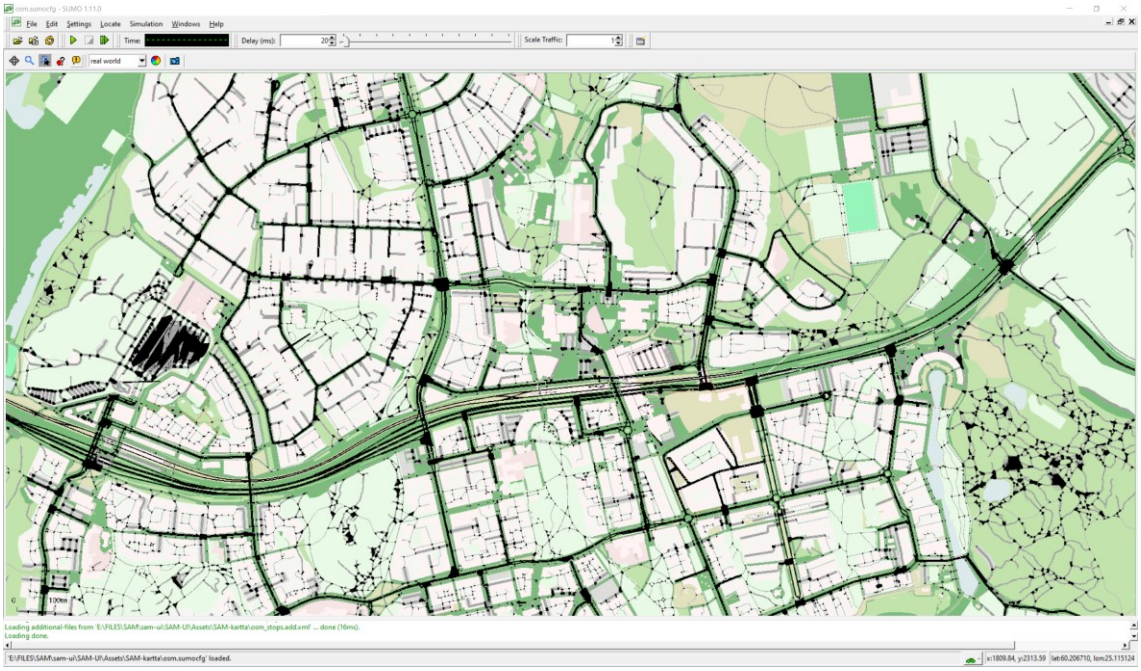
SUMO on mikroskooppinen liikennesimulaatio, joten siinä liikennettä käsitellään monena yksittäisenä ajoneuvona, jotka mallinnetaan liikkumaan määriteltyjen sääntöjen mukaisesti. Tämä poikkeaa makroskooppisista liikennesimulaatioista, joissa yksittäisten ajoneuvojen sijaan simuloidaan liikenteen virtaa ja suurempia ryhmiä ajoneuvoja. SUMO:n mikroskooppinen luonne oli tärkeää projektille, sillä tarkoituksena on seurata yksittäisiä ajoneuvoja simulaatiossa. SUMO:on on myös mahdollista lisätä liikennettä makroskooppisen määrittelyn pohjalta, käyttäen SUMO:n mukana tulevaa marouter-työkalua. Tämäkin liikenne kuitenkin simuloidaan yksittäisinä ajoneuvoina. (6; 8.)

SUMO on myös diskreetin ajan sijasta jatkuvan ajan simulaatio, eli toisin kuin diskreetin ajan simulaatioissa, joissa simulaatio etenee siinä olevien tapahtumien myötä, etenee SUMO tietyn aika-askeleen mukaisesti, vaikka aika-askeleiden välillä ei muutoksia ajoneuvojen tilassa tapahtuisikaan. Tämä on seurausta mikroskooppisesta simulaatiosta, jossa käytetään niin kutsuttua ”car-following model” -tekniikkaa mallintamaan ajoneuvojen liikettä. Tätä tekniikka hyö-

dyntäviä malleja on useita, mutta kaikille niille on yhteistä ajoneuvojen liikeyhtälön määrittely edellä ajavan ajoneuvon etäisyyden ja nopeuden sekä ajoneuvon oman nopeuden avulla differentiaaliyhtälöä hyödyntäen. (6; 9.)

3.1 Kartta

SUMO-liikennesimulaatiossa on merkittävässä osassa tiestö (kuva 2), jolla ajoneuvot liikkuvat. Se koostuu yksittäisistä tieosuuksista, joihin viitataan nimellä “edge” eli särmä. Yhdellä särmällä voi olla useampi kaista, mutta ne eivät voi kulkea eri suuntaan, sillä jokaisella särmällä on suuntansa. Näin ollen kaksisuuntainen tie koostuu vähintään kahdesta eri särmästä. Särmien lisäksi tiestö koostuu myös risteyksistä ja liittymistä, joihin viitataan termillä “junction”. Näiden risteyksien ja liittymien toiminnot ja säännöt, kuten mahdolliset liikennevalot ja niiden vaihtumisnopeudet tai mahdollinen väistämisvelvollisuus, ovat muokattavissa. (10.)



Kuva 2. SUMO:n graafinen käyttöliittymä. Näkyvissä osa projektissa simuloidusta tieverkostosta.

Simulaatiota varten kartan voi luoda käsin tai koodin avulla. Manuaalisesti karttaa luodessa tai valmista karttaa muokatessa voi käyttää SUMO:n mukana saatavaa netedit-ohjelmaa, joka mahdollistaa tiestön käsittelyn graafisen käyttöliittymän kautta. Projektissa käytettiin netedit-työkalua tieverkoston hienosäätöä varten. (11.)

Projektissa kartan luontiin käytettiin SUMO-liikennesimulaattorin mukana saatavaa ”OSM Web Wizard” -ohjelmaa (kuva 3). Tämä ohjelma on Python-ohjelmointikielellä tuotettu työkalu, jonka avulla voidaan luoda liikennesimulaatiota varten kartta käyttäen ”Open Street Map” -palvelun karttatietoja. Näin voitiin helposti ja nopeasti tuottaa laajojakin tieverkostoja, jotka perustuivat reaalimaailman tiestöihin. OSM Web Wizardin avulla haetut kartat voivat sisältää myös automaattista liikennettä, jota hyödynnettiin projektissa, jotta simuloidut autonomiset ajoneuvot eivät liikkuisi liikennesimulaatiossa tyhjillä tieosuuksilla, vaan tämä liikenne hidastaisi niiden kulkua realistisesti. Työkalun tuottamat kartat eivät tosin olleet täysin toimivia tai eivät aina onnistuneesti edustaneet reaalimaailman tiestöjä: jotkin risteykset eivät toimineet tarkoituksensa mukaisesti, vaan saattoivat esimerkiksi koostua neljästä pienemmästä risteyksestä. Nämä ongelmat syntyvät työkalun heuristiikasta, joka yrittää korjata puutteellista tai monitulkinnaista dataa. Ne saatiin kuitenkin korjattua manuaalisesti netedit-työkalun avulla. (12.)

Kuva 3. OSM Web Wizard -ohjelman käyttöliittymä, jossa osa simulaation käyttämästä alueesta valittuna.

3.2 Ajoneuvot

Simulaatiossa keskipisteessä ovat erilaiset ajoneuvot, joiden liikkumista tiestöllä simuloidaan. Ajoneuvoilla on eri luokkia, jotka määrittävät sen, millä reiteillä ne voivat simulaatiossa liikkua. Näitä luokkia on SUMO:ssa 26, joista kaksi on varattu tarvittaessa käyttäjän räätälöitäväksi. Oletusarvoisesti simulaatioon lisätyt ajoneuvot kuuluvat SUMO:ssa henkilöautoluokkaan. Useimmat ajoneuvoluokat kuvastavat erilaisia tiellä liikkuvia ajoneuvoja, kuten busseja, takseja, huoltoajoneuvoja ja pelastusajoneuvoja. Kaikki ajoneuvot eivät kuitenkaan liiku teillä, vaan erilaiset luokat liikkuvat eri reitillä. Niihin lukeutuvat esimerkiksi jalankulkijat, pyöräilijät, jotka voivat kulkea omilla reiteillään, sekä viisi eri raideliikenteen muotoa, minkä lisäksi myös laivoille on oma luokkansa. (13.)

Kaikki ajoneuvot liikkuvat niille määrätyllä reitillä kiihdyttäen realistisesti tavoitenopeuteen, joka on riippuvainen tieverkon nopeusrajoituksesta sekä edellä ajavan ajoneuvon nopeudesta ja etäisyydestä. Ajoneuvolla pitää olla määrätty reitti, joka voi olla joko yhden tieosuuden mittainen tai kiertää koko tieverkon useita kertoja. Reitin voi määrittää myös simulaation aikana uudelleen, ja tällöin

sen voi ilmoittaa joko normaalisti listana tieosuuksia, joille ajoneuvon on liikuttava, tai antamalla ajoneuvolle uudelleenreitityskomento sekä uusi määränpää. Tällöin simulaatio laskee optimaalisen reitin kohteeseen.

3.3 TraCI-rajapinta

TraCI, eli “Traffic Control Interface”, on rajapinta, jonka avulla voidaan ohjata SUMO-liikennesimulaattoria. Sen perusteena on TCP-tietoliikenneprotokollaa hyödyntävä palvelin-asiakaskonerakenne. Tällöin SUMO toimii palvelimena, joka kommunikoi TraCI-asiakkaan kanssa, jolloin SUMO:n graafinen käyttöliittymä ei ole käytössä. SUMO kommunikoi tällöin valitun portin kautta. Tämä mahdollistaa myös sen, että SUMO-instanssi voi olla joko paikallisella tietokoneella tai siihen voidaan muodostaa yhteys verkon kautta. TraCI:n käyttämistä varten pitää SUMO ensin käynnistää komentoriviltä antaen sille parametrinä “--remote-port”, minkä jälkeen ilmoitetaan haluttu porttinumero. Kun SUMO on käynnistetty näin, ensin ladataan valittu simulaatio, joka sisältää tiedon simuloitavasta alueesta, sen tiestöstä ja mahdollisesta liikenteestä. Tämän jälkeen SUMO jää odottamaan TCP-yhteyttä asiakaskoneelta. Kun yhteys on muodostettu, simulaatiota voidaan edistää ennalta määritetyn aika-askeleen verran tai määriteltyyn aika-askeleeseen asti “Simulation Step” -komennolla. (14; 15.)

TraCI:n kautta on mahdollista hallita liikennesimulaattorin toimintaa ja noutaa haluttujen kohteiden tietoja. Näin voidaan muun muassa lisätä simulaatioon uusia ajoneuvoja, muokata tiestöjen nopeusrajoituksia tai hakea esimerkiksi ajoneuvojen tietoja, kuten nopeus, sijainti tai hetkellinen päästöjen määrä. TraCI-rajapinnan avulla on myös mahdollista suorittaa “Variable Subscription”, jossa valitun olion muuttujat “tilataan”. Näin ollen jokaisella simulaatiota edistävällä komennolla saadaan paluuarvoksi kaikkien “tilattujen” muuttujien arvot. Tämä mahdollistaa esimerkiksi liikennevalojen tilan tai ajoneuvon kulutuksen seuraamisen. TraCI:n toiminnallisuus rajoittuu kuitenkin ennalta määritettyihin komentoihin, eikä uusien komentojen määrittely ole mahdollista.

3.4 TraCI.NET-kirjasto

TraCI.NET on avoimen lähdekoodin kirjasto, jonka tarkoituksena on ollut tuoda virallinen TraCI-kirjasto, joka on saatavilla vain Python-ohjelmointikielelle, toiminnallisuus C#-ohjelmointikielelle. Tämä on projektin kannalta erittäin merkittävää, sillä se mahdollistaa SUMO-liikennesimulaation ja Unity-pelimoottorin välisen yhteyden luomisen samalla ohjelmointikielellä, jota käytetään Unity-pelimoottorissa itse kirjoitetussa koodissa. Kirjaston käyttöä varten se käännettiin dynaamisesti linkitetyksi kirjastoksi eli dll-tiedostoksi. Näin Unity-pelimoottori kykenee muodostamaan tarvittavat viittaukset automaattisesti, kun ohjelma käännetään suorittamista varten. Kirjasto on saatavilla GitHub-sivustolla, ja sen yhteydessä on yksinkertainen esimerkki kirjaston käytöstä. Koodissa merkittävässä osassa on ”TraCIClient”-olio, jonka avulla yhteys liikennesimulaatioon luodaan ja jonka kautta komennot annetaan. (16.) Esimerkkikoodi 1 näyttää, miten olion luonti ja yhteyden muodostaminen voidaan toteuttaa.

```
var client = new TraCIClient();
var task = client.ConnectAsync("127.0.0.1", 4321);
while (!task.IsCompleted)
{
     /*  Wait for task to be completed before using traci commands */
}
```

Esimerkkikoodi 1. TraCI.NET-ohjelmakirjaston esimerkki ”TraCIClient”-olion luonnista ja yhdistämisestä palvelimena toimivaan SUMO-instanssiin (17).

TraCI.NET ei tosin ole kokonainen TraCI-toteutus, vaan joitain rajapinnan komentoja ei vielä ole valmiina. Tämä ei aiheuttanut kehitystyön aikana suurempia ongelmia, mutta lisäsi joidenkin ongelmatilanteiden kehittämisen haastavuutta. TraCI.NET-kirjastoa on kirjoitushetkellä viimeksi päivitetty vuoden 2020 elokuussa, joten puuttuvien TraCI-rajapinnan toimintojen täydentämistä ei kannattanut odottaa. Kuitenkin TraCI.NET mahdollisti Unity-pelimoottorin yhdistämisen SUMO-liikennesimulaattoriin huomattavan paljon helpommin kuin muut vaihtoehdot, joissa olisi ollut tarpeen muodostaa yhteys C#-kielen ja jonkin toisen ohjelmointikielen välille.

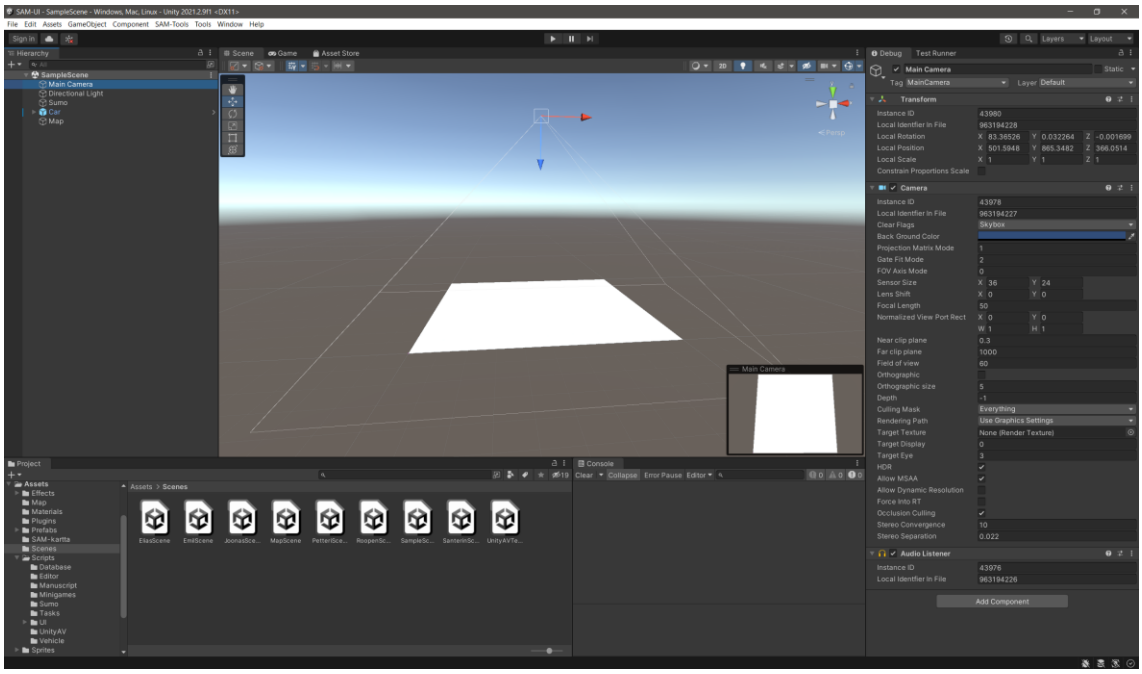
4 Unity-pelimoottori

Unity on ensimmäisen kerran vuonna 2005 julkaistu pelimoottori, jonka on kehittänyt alun perin Tanskassa perustettu Unity Technologies. Pelimoottorit ovat ohjelmia ja ohjelmakirjastoja, joiden avulla voidaan kehittää pelejä. Unity on suosittu pelimoottori monesta syystä, joista eräs on sen rahoitusmalli, joka antaa kehittäjien käyttää pelimoottorin “Personal”-tason lisenssiä ilmaiseksi, kunhan rahoitus tai tuotot ovat viimeisen kahdentoista kuukauden aikana alle 100 000 dollaria (18). Unity-pelimoottoria varten on myös suuri määrä opetusmateriaaleja ja aktiivinen kehittäjäyhteisö, jonka toimintaa Unity Technologies myös rahallisesti tukee. Tarjolla on myös suuri määrä valmiita lisätyökaluja ja ohjelmakirjastoja, jotka joko lisäävät Unityyn uusia toiminnallisuuksia tai helpottavat olemassa olevien toiminnallisuuksien käyttöä tai saatavuutta. Unity tarjoaa myös mahdollisuuden kehittää omia laajennuksia ja kirjastoja, joilla voi helpottaa ja nopeuttaa omaa kehitystyötä. Näitä työkaluja voi myös kaupata muille kehittäjille Unityn ylläpitämässä Unity Asset Storessa. (19.)

Unityssä käytetään oman koodin kirjoittamiseen C#-ohjelmointikieltä. Se on Microsoft Corporationin vuonna 2000 julkaisema korkean tason ohjelmointikieli, joka on keskittynyt olio-ohjelmointiin. Sen kehityksen lähtökohtana on ollut tuottaa helppokäyttöinen ohjelmointikieli, joka ei ole alustariippuvainen ja johon on helppo siirtyä vanhemmista C- ja C++-ohjelmointikielestä. Unityn avulla voi myös kehittää pelejä yli 25 alustalle, mukaan lukien PC-tietokoneet ja yleisimmät pelikonsolit, sekä virtuaalitodellisuuslaseille ja laajennetun todellisuuden laitteille. (19; 20.)

Unity-pelimoottorilla kehitys on suunniteltu GameObjectien eli peliolioiden ja niihin liitettävien komponenttien luomiseen ja yhdistelyyn. Pelimoottorissa pelioliot sijaitsevat aina näkymässä, johon viitataan termillä “scene”. Näkymää voi muokata Unityn graafisen käyttöliittymän avulla (kuva 4) ja lisätä sinne peliolioita, joita on valmiina saatavilla esimerkiksi yksinkertaisia muotoja, kuten kuutio ja pallo, valoja, joilla näkymää voi valaista, tai käyttöliittymän-komponentteja, joilla pelin sisäinen käyttöliittymä toteutetaan. Näiden peliolioiden komponentteja ja

niiden arvoja voi muuttaa Unityn “inspector”-ikkunan avulla. Näiden arvojen muokkaus on mahdollista myös koodin kautta, jolloin tarvitaan viittaus kyseiseen komponenttiin. Myös kaikki kehittäjän kirjoittama koodi, joka halutaan suorittaa ajon aikana, on kiinnitetty komponenttina johonkin peliolioon näkymässä. (21; 22; 23.)



Kuva 4. Unity-pelimoottorin Editor-näkymä. Keskellä nähtävissä aktiivisena oleva näkymä. Vasemmalla näkymän sisällä olevat oliot. Oikealla "inspector"-ikkuna, jossa näkyvät kamera-olion komponentit. Alareunassa projektin tiedostot-ikkuna ja loki. (24.)

Näkymään voi myös lisätä peliolioita ajon aikana, mutta tällöin niiden tulee olla valmiiksi määriteltyjä prefab-resursseja. Ne ovat kehittäjän luomia peliolioita, joilla voi olla lapsiolioita ja komponentteja, joiden tila on tallennettu. Toinen vaihtoehto on koodin kautta luoda yksinkertainen peliolio, johon sen jälkeen lisätään kaikki komponentit ja mahdolliset lapsioliot. Tätä lähestymistapaa on kuitenkin hyvä välttää, sillä se on työläämpää ja kaikkien koodikomponenttien muuttujien arvot eivät välttämättä ole alustettu valmiiksi. (25.)

5 Liikennesimulaattorin integraatio

Tässä luvussa käsitellään insinöörityön projektin toteutusta ja sen eri vaiheita. Ensimmäisessä aliluvussa käsitellään projektin työmenetelmiä ja työkaluja. Seuraavassa aliluvussa keskitytään pelimoottorin ja liikennesimulaattorin integraation toteutuksen rakenteeseen. Tämän jälkeen käsitellään yhteyden muodostamista Unityn ja SUMO:n välille. Aliluvussa neljä käsitellään ajoneuvojen liikkeen siirtoa simulaattorista pelimoottoriin. Tämän jälkeen käsitellään kartan luontia ja sitä varten kehitettyä työkalua. Aliluku kuusi käsittelee projektin jatkokehitystä ja aliluku seitsemän sisältää loppuanalyysin.

5.1 Työmenetelmät

SAM-hankkeen projektissa työskenneltiin kuuden opiskelijan ryhmässä. Ryhmä tapasi projektin ohjaajien kanssa viikoittain. Projektin alussa keskityttiin kartoittamaan mahdollisia teknologioita ja ratkaisuja projektia varten. Lopulta päädyttiin ratkaisuun, jossa liikennesimulaattori yhdistettäisiin pelimoottoriin, jolloin liikennesimulaattori vastaisi ajoneuvojen liikkumisesta reaalimaailman tieverkostossa ja pelimoottori mahdollistaisi ongelmatilanteiden toteuttamisen ja simuloidun liikenteenvalvontasovelluksen ulkonäön sekä hankkeessa keskeisessä osassa olevien mittaustulosten kirjaamisen. Liikennesimulaattoriksi valittiin Eclipse SUMO, koska sen lähdekoodi on avoin, kevyistä suoritusvaatimuksista ja kyvystä simuloida yksittäisiä ajoneuvoja ja suuria tieverkkoja. Pelimoottorina päädyttiin käyttämään Unityn versiota 2021.2.9f1, joka oli projektin aloitushetkellä uusin vakaa julkaisu. Unity-pelimoottori valittiin pääosin ryhmän aikaisemman kokemuksen takia ja projektin kannalta sopivan lisenssin seurauksena.

Kun projektin työkalut ja teknologiat olivat selvillä, aloitettiin työskentely perehtymällä SUMO:n toimintaan dokumentaatioiden ohella saatavilla oleviin opetusmateriaalien avulla. Nämä opetusmateriaalit olivat laajalti tieverkkojen rakentamisesta netedit-työkalulla ja erilaisten liikenteen muotojen luomista TraCI-rajapinnan kautta Python-ohjelmointikieltä hyödyntäen. Kun SUMO:n ja TraCI-raja-

pinnan toiminta oli selkeämpää, hyödynnettiin OSM Web Wizard -työkalua reaalimaailman kartan mallintamiseen SUMO:ssa. Näiden karttojen vaatimat korjaukset olivat mahdollisia aikaisemmin tehtyjen netedit-työkalun harjoitusten avulla. Tämän jälkeen aloitettiin kehitys Unity-pelimoottorin avulla. Avoimen lähdekoodin kirjastoon TraCI.NET päädyttiin, koska etsittiin ratkaisua TraCI-komentojen käyttämiseen Unityn avulla tuotetussa koodissa ja tämän ohjelmistokirjaston avulla toteutuksessa pystyttiin käyttämään vain yhtä ohjelmointikieltä eli Unityn käyttämää C#-kieltä. Projektin kannalta oli tärkeää, että kaikki tarvittavat SUMO:n toiminnallisuudet olivat käytettävissä.

Kehityksessä käytettiin versionhallintana Metropolian GitLab-sivustoa, jonka käyttöä helpotti graafinen GitHubin työpöytäsovellus. Projektia varten luotiin GitLabin ”repository”, eli kuvauskanta, jonka avulla ryhmä pystyi jakamaan keskenään projektiin tekemät muutoksensa. Projekti jaettiin moneen eri haaraan, joiden sisällä kehitettiin tiettyjä projektin osia, esimerkiksi käyttöliittymää kehitettiin omassa haarassa. Näin ollen eri osa-alueiden muutokset eivät vaikuttaneet toisten osien kehitykseen. Päähaaroja projektiin luotiin kaksi, kehityshaara ja mainhaara. Muutokset laitettiin ensin kehityshaaraan ja siirrettiin main-haaraan, kun ongelmat ja virheet oli korjattu. Näin saatiin aikaan vakaa sovellus main-haaraan ja samaan aikaan kehityksen muutoksia voitiin testata kehityshaarassa ilman, että sovelluksen toimivuus oli uhattuna.

Projektissa haasteita aiheuttivat Unityn näkymätiedostot, sillä niihin tehtyjen muutosten yhdistäminen ei onnistu helposti Git-työkalujen avulla. Tämän seurauksena projektissa päätettiin luoda jokaiselle henkilölle oma näkymä, johon he tekivät muutoksensa. Sen jälkeen muutokset yhdistettiin manuaalisesti siirtämällä muutetut osiot eli pelioliot ja niiden komponentit yhteen yhteiseen näkymään. Näin vältyttiin Git-työkalun ”merge conflict” -virheiltä eli yhdistämisristiriidoilta, jotka syntyvät, kun Git-työkalu ei kykene automaattisesti selvittämään samaan tiedostoon eri lähteistä tehtyjä muutoksia, kun muutokset koskevat samoja tekstitiedoston rivejä. Näitä virheitä voi manuaalisesti korjata, kun kyse on tiedostoista, jotka on suunniteltu ihmisten luettaviksi. Unityn näkymätiedostot eivät kuitenkaan ole muodoltaan tarkoitettu luettaviksi, joten yhdistämisristiriitojen

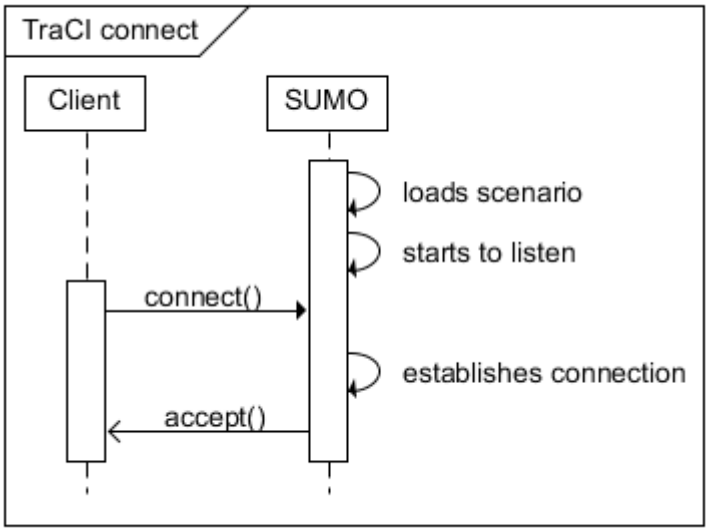
selvittäminen manuaalisesti on kovin hankalaa (26). Kuitenkin seurauksena siitä, että kehitystä tehdään eri näkymätiedostoissa, koodimuotoisten komponenttien muutokset välittyvät kaikkiin instansseihin siitä komponentista ja koodivirheitä ilmenee, kun näkymät eivät esimerkiksi sisältäneet jotain pelioliota, johon koodi viittasi.

Versionhallintatyökalujen lisäksi projektin työnjakoa ja työtehtävien seuraamista helpotti Microsoft Teams -ohjelma, jonka kautta toteutettiin palaverit ja jonne dokumentoitiin aktiivisesti ryhmän etenemistä jäsenkohtaisesti. Myös työtunnit sekä palaverien perusteella jaetut työtehtävät kirjattiin yhteisiin dokumentteihin. Kehitystyötä varten kirjoitettiin myös pelisuunnitteludokumentti, jota päivitettiin aktiivisesti työn ohella projektin edetessä. Metropolian GitLab-sivusto mahdollisti myös työtehtävien seuraamisen ja jakamisen ryhmän sisällä ”issues”-välilehden avulla. Sen avulla voitiin luoda listaan nimike jostain tarvittavasta ominaisuudesta ja kuvaus halutusta toteutuksesta, minkä jälkeen se voitiin osoittaa jollekin ryhmän henkilöistä ja tehtävän tehtyään hän pystyi merkitsemään sen ”issues”-välilehdellä valmiiksi. Näin saatiin seurattua ryhmän edistymistä ja paikannettua haastavat osiot, joissa kesti pidempään kuin alun perin oli suunniteltu, ja kyettiin lisäämään näihin tehtäviin resursseja.

5.2 Integraation rakenne

SUMO-liikennesimulaattorin integrointi Unity-pelimoottoriin vaati kommunikaation näiden kahden ohjelman välille. Projektin suunnitteluvaiheen aikana keskusteltuja vaihtoehtoja tämän ongelman ratkaisemiseksi oli kaksi, joista ensimmäinen oli liikennesimulaattorin yhdistäminen suoraan osaksi pelimoottoria hyödyntäen niin kutsuttua Libsumo-ratkaisua, jossa SUMO-liikennesimulaattorin lähdekoodi rakennetaan C++-ohjelmointikielen ohjelmistokirjastoksi (27). Tällä ratkaisulla on monta etua verrattuna toiseen ratkaisuun, jossa hyödynnetään SUMO:n kykyä toimia TCP-palvelimena ja vastaanottaa TraCI-rajapinnan kautta komentoja (kuva 5). Libsumo-ratkaisussa etuina ovat suorittamisen nopeus, kun liikennesimulaatio suoritetaan osana valmista ohjelmaa, sekä etu verrattuna TCP-tietoliikenneprotokollan aiheuttamaan suorituskyvyn laskuun, joka liittyy

palvelin-asiakaskonerakenteen lisäämään monimutkaisuuteen. Kuitenkin SUMO:n kyky toimia palvelimena ja hyödyntää tällöin TraCI-rajapintaa, kuten projektin alkuvaiheessa, helpotti tämän ratkaisun käyttöönottoa huomattavissa määrin.



Kuva 5. SUMO-dokumentaation havainnekuva yhteyden luomisesta SUMO-palvelimeen (15).

Valintaan vaikutti myös alkuperäinen tavoite ajaa valmis ohjelma verkkoselaimen kautta hyödyntäen Unityn tukemaa WebGL (lyhennys sanoista Web Graphics Library) -muotoa. WebGL on alun perin Mozilla Foundationin kehittämä ohjelmointikirjasto, joka on suunniteltu kaksi- ja kolmiulotteisen tietokonegrafiikan reaaliaikaiseen renderöintiin verkkoselaimessa. Sen kehityksestä vastaa nykyisin Khronos Group. WebGL-versiota varten olisi myös ollut yksinkertaisempaa käyttää SUMO:a palvelimena ohjelmakirjaston sijaan. Tällöin olisi myös ollut helppoa ajaa SUMO-instanssi ja Unitylla toteutettu ohjelma palvelimella loppukäyttäjän tietokoneen sijasta. Näistä syistä päädyttiin käyttämään SUMO-liikennesimulaattoria palvelimena.

SUMO:n integrointia varten Unityssä luotiin jokaiseen näkymään lisättävä peliolio, jonka komponenttina oli SumoConnect-olio. Tämä olio seuraa niin kutsuttua

”singleton”-suunnittelumallia, eli jokaisena hetkenä on olemassa vain yksi instanssi tästä oliosta ja siihen on globaalisti saatavilla oleva viite (28). Näin ollen muiden olioiden, joiden tarvitsee kommunikoida liikennesimulaattorin kanssa, ei tarvitse erikseen hakea viitettä tähän SumoConnect-olioon. Unityssä yleisin tapa saada ajonaikainen viittaus on Object-luokan metodien FindObjectOfType ja FindObjectsOfType avulla, joista ensimmäinen palauttaa ensimmäisen halutun tyyppisen olion näkymästä, ja toinen palauttaa kaikki haluttua tyyppiä vastaavat oliot näkymästä. Nämä metodit ovat huomattavasti hitaampia kuin singleton-malli, sillä ne käyvät läpi kaikki näkymän pelioliot yksi kerrallaan, tosin kuin singleton-mallin suora staattinen viite.

SumoConnect-luokka pitää sisällään tarvittavan koodin SUMO-liikennesimulaattorin käynnistämiseen ja TCP-yhteyden luomiseen SUMO:n ja liikenteenvalvontasovelluksen välille. Sovelluksen toimivuuden kannalta oli tärkeää, että kahta sovellusta ei tarvitsisi käynnistää erikseen eikä olisi tarvetta manuaalisesti valita käynnistettävää simulaatiotiedostoa tai yhdistää sovellusta TCP-portin kautta liikennesimulaattoriin. Tämän seurauksena päätettiin, että sovelluksen tulisi käynnistyessään käynnistää ensin SUMO-ohjelma ja ladata samalla ennalta määritelty simulaatiotiedosto ja avata SUMO:n portin kuuntelu. Tämän jälkeen sovelluksen tulee muodostaa yhteys SUMO:on ja valmistella tarvittavien ajoneuvojen muuttujien seuraaminen tilausmallia hyödyntäen. Tilausmalli (engl. Subscription model) tarkoittaa haluttujen kohteiden tietojen ”tilaamista”, jolloin jokainen simulaatiota edistävä SimulationStep-metodikutsu palauttaa kokoelman kaikkien tilattujen olioiden haluttujen muuttujien arvot. Tämä malli on suositeltu tapa hakea tietoa ajoneuvoista ja muista simulaation kohteista, sillä se on tehokkaampaa kuin näiden tietojen hakeminen jokaisen simulaatioaskeleen jälkeen TraCI.NET-kirjaston Vehicle-olion Get-metodien, kuten GetSpeed, avulla.

Ensimmäinen integraation osuus oli siis käynnistää SUMO-liikennesimulaattori samanaikaisesti liikennesimulaattorin kanssa. Tämä toteutettiin SumoConnect-luokassa hyödyntäen C#-ohjelmointikielen System.Diagnostics-nimiavaruutta. Tämä nimiavaruus Microsoft Corporationin dokumentaatioiden mukaan ”Tarjoaa

luokkia, joiden avulla voit olla vuorovaikutuksessa järjestelmäprosessien, tapahtumalokien ja suorituskykylaskureiden kanssa” (29). Nämä järjestelmäprosessien käytön mahdollistavat luokat olivat avainasemassa liikennesimulaattorin käynnistämisessä ohjelman kautta. Process-luokan Start-metodi mahdollistaa sovelluksen käynnistämisen ja sille tarvittavien komentoriviargumenttien antamisen. Tämä mahdollisti SUMO:n käynnistämisen TCP-palvelimena ja oikean simulaatiotiedoston lataamisen. Kuitenkin haasteena oli se, että WebGL-toteutuksessa ei ole mahdollista suorittaa käyttäjän laitteella olevia ohjelmia Process-luokan avulla. On ratkaisuja, joissa ohjelmien suorittaminen on mahdollista, mutta ne vaativat käyttäjän manuaalisen hyväksynnän. Tämän seurauksena ohjelma käynnistää esimerkkikoodin 2 mukaisesti SUMO-instanssin vain, kun ohjelma ajetaan joissain muussa muodossa kuin WebGL-toteutuksena.

```
#if !UNITY_WEBGL || UNITY_EDITOR
     System.Diagnostics.Process.Start("sumo", $"-c {SimulationPath} --
remote-port {Port}");
#endif
```

Esimerkkikoodi 2. C#-ohjelmointikielen esikäsittelijälle annetuilla komennoilla Start-metodia kutsutaan vain ehtojen toteutuessa.

WebGL-toteutusta varten tutustuttiin muihin tapoihin käynnistää ohjelmat samanaikaisesti. Koska valmis WebGL-toteutus on isännöitävä web-palvelimen avulla, päätettiin tämä isännöinti ja SUMO-liikennesimulaattorin palvelin käynnistää yksinkertaisen ”scriptin” avulla. Näitä ”scriptejä” tehtiin kaksi, esimerkkikoodien 3 ja 4 mukaisesti, jotta käynnistäminen olisi helppoa Windows-pohjaisilla ja useilla Linux-pohjaisilla käyttöjärjestelmillä. Näin mahdollistettiin liikenteenvalvontasovelluksen ja liikennesimulaation yhteinen käynnistäminen saumattomasti kaikilla tavoitealustoilla.

```
@echo off
start sumo -c E:\PROGRAMS\Sumo\tools\2022-01-24-14-32-31\osm.sumocfg -
-remote-port 4321
start npx http-server -o --gzip true
```

Esimerkkikoodi 3. Host.bat-tiedoston koodi, joka käynnistää ensin SUMO-palvelimen, minkä jälkeen se isännöi web-palvelimella kansion sisällön.

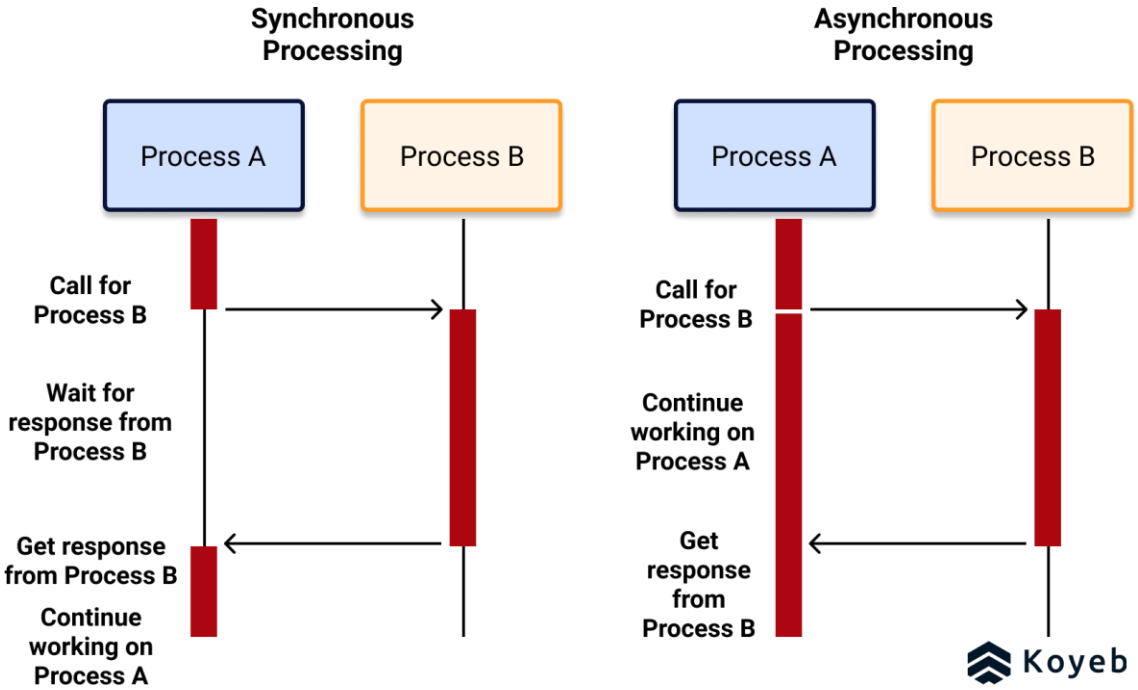
```
#!/usr/bin/env bash
sumo -c E:/PROGRAMS/Sumo/tools/2022-01-24-14-32-31/osm.sumocfg --re-
mote-port 4321
npx http-server -o --gzip true
```

Esimerkkikoodi 4. Esimerkkikoodia 3 vastaava toteutus Bash-ohjelmointikielellä Linux-palvelimia varten.

5.3 SUMO:n ja Unityn yhteys

Unity-pelimoottorin ja SUMO-liikennesimulaattorin kommunikaatio tapahtuu SumoConnect-luokan instanssin kautta. SUMO-liikennesimulaattorin käynnistyttyä yhteys siihen muodostetaan luomalla ensin instanssi TraCI.NET-ohjelmakirjaston TraCIClient-luokan oliosta. TraCIClient-luokan metodi ConnectAsync luo TCP-tietoliikenneprotokollaa hyödyntäen yhteyden SUMO:n palvelimeen. Tämä on ASYNC-metodi eli asynkroninen metodi (kuva 6). Se tarkoittaa, että tavallisesta poiketen tätä koodia suorittaessa ei siirrytä riviltä seuraavalle heti, kun koodi on suoritettu, vaan muuta koodia voidaan suorittaa samalla, kun metodi odottaa tietoa. Tämä metodi palauttaa Task-olion, joka on osa System.Threading-nimiavaruutta.

Kun metodia on kutsuttu, aloitetaan Corutine, joka mahdollistaa uuden säikeen luomisen. Sen sisällä odotetaan, kunnes Task-olion totuusarvoisen muuttujan IsCompleted arvo on tosi. Se kertoo tämän Task-olion prosessin olevan valmis. Tämän jälkeen on tarkastettava, että yhdistäminen onnistui. Se voidaan tarkistaa Task-olennon kahden totuusarvoisen muuttujan avulla: IsCanceled, joka kertoo, onko prosessin suorittaminen keskeytetty, ja IsFaulted, joka taas kertoo, onko prosessi keskeytynyt törmättyään virheeseen. Jos yhteyden muodostaminen ei näistä syistä ole onnistunut, kirjataan tämä Unityn lokiin virheenä, jolloin ongelman huomaaminen helpottuu.



Kuva 6. Asynkronisen ja synkronisen prosessin toiminta (30).

Kun ohjelma on onnistuneesti muodostanut yhteyden, kirjataan onnistunut yhteys Unityn lokiin, minkä jälkeen suoritetaan SumoConnect-luokan OnConnectionFormed-metodi, jonka avulla toimet, jotka vaativat yhteyden Unityn ja SUMO-liikennesimulaattorin välille, voidaan suorittaa vasta, kun yhteys on valmis. Näistä tärkein on määrittää, kumpaa TraCI.NET-kirjaston toteutuksista tilausmallista käytetään. Kirjaston kehittäjien suositus on käyttää Dictionary- eli sanakirja-tietorakennetta hyödyntävää uudempaa toteutusta. Tämä valinta tehdään C#-ohjelmointikielen Eventien eli tapahtumien avulla. Esimerkkikoodin 5 mukaisesti TraCIClient-olion, johon esimerkkikoodissa viitataan nimellä Client, VehicleSubscription-tapahtuma tilataan, jolloin kutsutaan funktiota, jonka nimi on ”tilausmerkinnän” oikealla puolella.

```
// Sets what happens when a subscription is updated
Client.VehicleSubscription += Client_VehicleSubscriptionUsingDiction-
ary;
```

Esimerkkikoodi 5. C#-kielen tapahtuman tilaus.

Tämän jälkeen SumoConnect-luokan totuusarvoisen IsConnected-muuttujan arvo asetetaan olemaan tosi. Tämän muuttujan arvon voi lukea muista luokista,

mutta sitä voi muuttaa vain SumoConnect-luokan sisältä. Näin ollen voidaan tämän muuttujan arvoa seurata muissa luokissa, joissa jonkin toiminnallisuuden pitää odottaa yhteyden muodostumista tai tulla käytettäväksi vasta, kun yhteys on muodostettu. Kun tämä vaihe on valmis ja yhteys on luotu, jää sovellus odottamaan simulaation aloitusta, eli simulaatio ei etene, ennen kuin SumoConnect-luokan SimulationPaused-muuttujan arvo on epätosi. Tämän muuttujan arvoa muuttamalla voidaan keskeyttää ja jatkaa simulaation etenemistä haluttuna ajankohtana.

Jotta Unity-pelimoottorin ja SUMO-liikennesimulaation integraatiossa ajoneuvot eivät liikkuisi joko liian nopeasti tai hitaasti kartalla verrattuna niiden nopeuteen simulaatiossa, tuli varmistaa, että simulaation aika-askel vastaisi sitä, kuinka usein Unityssä kutsutaan TraCI.NET-kirjaston SimStep-komentoa. Tämän mahdollistaisi silmukka, jossa odotettaisiin simulaation aika-askelta vastaava aika ja sen jälkeen edistettäisiin simulaatiota yksi askel eteenpäin. Unity-pelimoottorissa on kaikille MonoBehaviour-luokan periville olioille kutsuttava Update-niminen metodi, joka suoritetaan jokaisella kerralla, kun sovelluksen kuva päivitetään. Tämän seurauksena Update-metodien välinen aika on riippuvainen sovelluksen virkistystaajuudesta ja sen takia vaihtelee riippuen muun muassa tietokoneen tehosta ja käytettävien resurssien määrästä sekä sovelluksen prosessoinnin vaativuuden vaihtelun seurauksena. Aika edellisestä Update-kutsusta on kuitenkin saatavilla Unityn Time-luokan deltaTime-muuttujan avulla, joka näyttää ajan edellisestä piirretystä kuvasta sekunneissa. Time.deltaTime-muuttujan avulla on mahdollista kompensoida Update-metodikutsujen vaihtelevaa aikaväliä. (31; 32.)

Kuitenkin insinöörityössä päädyttiin käyttämään Unityn FixedUpdate-metodia, jota sitäkin kutsutaan kaikille MonoBehaviour-luokan periville olioille, tosin Update-metodista poiketen tasaisella aikavälillä. Tämä aikaväli on määriteltävissä projektikohtaisesti, mutta vakiona se on yhteydessä pelimoottorin fysiikkamallintamisen laskuihin ja sen on määritelty olevan 20 millisekuntia, eli metodia kutsu-

taan viisikymmentä kertaa sekunnissa. Tämän seurauksena oli tarpeen määrittää simulaation aika-askel 20 millisekunnin mittaiseksi, jolloin simulaatio etenee reaaliajassa eikä ole tarpeen kompensoida aikavälin vaihtelua. (31.)

Jotta toteutus sallisi myös simulaation toistamisen tarvittaessa reaaliaikaa nopeammin, lisättiin SumoConnect-luokkaan kokonaislukumuuttuja StepsPerCycle, joka kertoo, kuinka monta simulaation askelta edetään jokaisella pelimoottorin FixedUpdate-metodikutsulla, kun simulaatio ei ole pysäytetty SimulationPaused-muuttujan avulla. Tämä toteutus edellytti, että TraCI.NET-ohjelmakirjaston SimStep-komentoa kutsuttaisiin ylimääräisen parametrin kanssa, toisin kuin aikaisemmassa kehitysvaiheessa, jossa komennon kanssa ei ollut tarpeen käyttää parametreja. Kun SimStep-komentoa kutsutaan ilman parametreja, etenee simulaatio yhden askeleen, ja kutsuttaessa SimStep-komentoa yhdellä double-luvulla, joka kuvastaa aika-askelta, etenee simulaatio haluttuun aika-askeleeseen saakka. Tätä toteutusta varten tarvittiin myös kaksi muuta muuttujaa, jotka nekin ovat double-tyyppisiä. Toinen on juokseva luku _currentStep, joka kuvaa tämänhetkistä aika-askelta, ja toinen, stepLength-muuttuja kuvaa simulaation aika-askeleen pituutta sekunteina, joten sen arvo on kiinteä 0,02. SimStep komennon parametriksi annetaan esimerkkikoodi 6 mukaisesti laskutoimitus, jossa nykyiseen aika-askeleeseen lisätään aika-askeleen pituuden ja StepsPerCycle-muuttujan arvon tulo. Näin simulaatio etenee tarvittaessa nopeammin kasvattamalla StepsPerCycle-muuttujan arvoa.

```
private void FixedUpdate()
{
     if (!IsConnected || SimulationPaused) return;

     var response = Client.Control.SimStep(_currentStep + (stepLength *
StepsPerCycle));
     if (response.Result != ResultCode.Success)
          Debug.LogError($"Response: Result = {response.Result} | Iden-
tifier {response.Identifier} | Error {response.ErrorMessage}");

     _currentStep += stepLength;

}
```

Esimerkkikoodi 6. SumoConnect-luokan FixedUpdate-metodi.

TraCI.NET-kirjaston SimStep-komento palauttaa TraCIResponce-olion, joka sisältää tietoa simulaation askeleesta. Jotta mahdolliset ongelmat simulaatiossa huomattaisiin sovelluksen kehityksen tai ajon aikana, tarkastetaan tämän TraCIResponce-olion Result-muuttuja. Tämä muuttuja on ResultCode-tyyppinen, eli se on tavutyyppinen Enum eli arvojoukko. Tämän arvojoukon mahdollisia arvoja ovat Success eli onnistuminen, NotImplemented eli puutteellinen implementaatio, joka on seurausta TraCI.NET-kirjaston joidenkin TraCI-rajapinnan toimintojen puutteellisuudesta, sekä Failed eli epäonnistunut. Jos SimStep-komennon palauttaman vastausolion Result-muuttuja on arvoltaan jokin muu kuin onnistuminen, kirjataan Unityn lokiin virhe. Sen tulostukseen lisätään Result-muuttujan arvon lisäksi Identifier-tavumuuttujan ja ErrorMessage- eli virheilmoitusmerkkijonomuuttujan arvot.

5.4 Ajoneuvojen liikkuminen

Tärkeänä osana insinöörityön liikenteenvalvontasovellusta oli valittujen simuloitujen ajoneuvojen seuraaminen karttanäkymässä Unity-pelimoottorin sovelluksessa. Tämä edellytti ajoneuvojen saumatonta luomista, niiden tilan päivittämistä simulaation edetessä ja ajoneuvon poistamista sen poistuttua simulaatiosta. SUMO-liikennesimulaattorissa ajoneuvot lisätään joko simulaatiotiedostojen osoittamalla aika-askeleella tai manuaalisesti TraCI-rajapinnan komentojen avulla. Projektissa seurattavien ajoneuvojen reitit luotiin SUMO-liikennesimulaattorin mukana tulevan netedit-työkalun avulla. Näin kaikki ajoneuvot tulevat simulaatioon ennalta määritettyinä aikoina ja aloittavat kulkunsa omia reittejä pitkin. Nämä ajoneuvot myös lisätään Unityyn samalla aikaleimalla, kun ne tulevat simulaatioon.

Kaikki SUMO-liikennesimulaattorin kautta simuloitavat ajoneuvot ovat insinöörityön kirjoittamisen aikana autoja, joten ne käyttävät autoja varten luotua Car-luokkaa, mutta liikennesimulaattorin ja pelimoottorin yhdistäminen voi sisältää myös muita ajoneuvotyyppejä, joiden simulaatio tapahtuu SUMO:n kautta. Tämän vuoksi kehitettiin luokka SumoObject, joka perii Unityn MonoBehaviour-

luokan ja siten toimii kantaluokkana kaikille SUMO:ssa simuloitavien ajoneuvojen omille luokille (esimerkkikoodi 7). Luokka on merkitty abstract-avainsanan avulla abstraktiksi luokaksi, eli pelkästään SumoObject-luokan olioita ei voi luoda, vaan sen tulee olla peritty luokka. (31.)

Tullessaan aktiivisiksi SumoObject-luokan oliot lisäävät viitteen itsestään SumoConnect-luokan sanakirja-tietorakenteeseen, jossa avaimena käytetään olion yksilöivää tunnistetta ja arvona on viite SumoObject-olioon. Tämän mahdollistaa MonoBehaviour-luokan OnEnable-metodi, jota kutsutaan oliolle, kun se tulee aktiiviseksi. Vastaavasti MonoBehaviour-luokan OnDisable-metodi mahdollistaa sen, että SumoObject-olio voi poistaa viittauksen itseensä SumoConnect-luokan sanakirjasta, kun olio poistetaan käytöstä tai tuhotaan. (31.)

```
public abstract class SumoObject : MonoBehaviour
{
     // As the object is enabled it's added to the list in SumoConnect
and removed when it's disabled
     private void OnEnable() => SumoConnect.SumoObjects.Add(id, this);
     private void OnDisable() => SumoConnect.SumoObjects.Remove(id);
     [Tooltip("Set ID to mach a vehicle id in SUMO")] public string id
= "Test";
     public VehicleType type;
     public abstract List<byte> GetVariablesToSubscribeTo();
     public abstract void OnSumoUpdate(VariableSubscriptionEventArgs
args);
}
```

Esimerkkikoodi 7. Abstrakti luokka SumoObject.

Ajoneuvojen lisäystä varten luotiin SumoConnect-luokkaan Action-tyyppinen tapahtuma OnCarCreate. Tämä tapahtuma sisältää merkkijonon, jota käytetään kertomaan ajoneuvon tunniste, ja VehicleType-arvojoukon arvon, joka kuvaa luodun ajoneuvon tyyppiä. Tätä tapahtumaa käytetään ajoneuvon tarvittavien osien luontiin, sillä ajoneuvon olio on lisättävä kartalle ja siitä on lisättävä ajoneuvolistaan olio, jotta ajoneuvon hallintaa varten ei tarvitse valita kartalla olevaa kuvaketta. OnCarCreate-tapahtuma tilataan ohjelman käynnistyessä SumoConnect-luokan AddCarToMap-metodiin ja VehicleList- eli ajoneuvolista-luokan AddVehicle-metodiin. AddCarToMap-metodi lisää ajoneuvon peliolion, esimerkkikoodin 8 mukaisesti, Unity-pelimoottorin Instantiate-funktiolla kartan lapsiolioksi.

Ajoneuvo-oliot on toteutettu Unityssä prefab-resursseina, eli ajoneuvot ovat valmiita pelioliota, joiden komponentit ja lapsioliot on määritelty kehitysvaiheessa. Ennen ajoneuvon lisäämistä näkymään muutetaan ajoneuvo-olion SumoObject-komponentin tunniste- ja ajoneuvotyyppi-muuttujat vastaamaan metodin kutsussa annettuja arvoja. Kun ajoneuvo lisätään näkymään, se aktivoi muuttujien tilauksen hyödyntäen ennalta määritettyjä variablesToSubscribeTo-muuttujan sisältämiä arvoja. Näin tilaus tapahtuu automaattisesti ajoneuvo luotaessa.

```
public void AddCarToMap(string carId, VehicleType vehicleType)
{
     var car = carPrefab.GetComponent<SumoObject>();
     car.id = carId;
     car.type = vehicleType;
     Instantiate(carPrefab, map.MapTransform);
}
```

Esimerkkikoodi 8. SumoConnect-luokan AddCarToMap-metodi.

Ajoneuvojen sijainnin siirtäminen SUMO-liikennesimulaattorista Unity-pelimoottoriin vaati simuloitavan alueen kartan toteutuksen sekä SUMO:n ja Unityn eriävien koordinaattijärjestelmien seurauksena kolmiulotteisten koordinaattien uudelleenkartoittamisen. Tätä varten kehitettiin SumoMap-luokka, joka sisältää tarvittavan toiminnallisuuden, jota käsitellään insinöörityön luvussa 5.5. Ajoneuvojen tiedot saadaan simulaatiosta jokaisella aika-askeleella tilausmallin avulla. Koska ohjelman alkaessa määritellään tilaukseen SumoConnect-luokan metodi, kutsutaan sitä kerran jokaista ajoneuvoa kohti. Tällöin esimerkkikoodin 9 mukaisesti tilaustapahtuman lähettämät tiedot osoitetaan oikealle oliolle käyttäen tietojen ObjectId- eli oliotunniste-muuttujaa oikean viitteen hakemiseksi sanakirjarakenteesta. Tämä on nopeampi tapa kuin esimerkiksi listan tai taulukon läpikäynti oikean olion etsimiseksi, mikä johtuu sanakirjarakenteen absoluuttisesta aikavaatimuksesta. Tämä tieto lähetetään oikealle oliolle antamalla se olion OnSumoUpdate-metodikutsulle parametrinä.

```
private void Client_VehicleSubscriptionUsingDictionary(object sender,
SubscriptionEventArgs e) => SumoObjects[e.ObjectId].OnSumoUpdate(e as
VariableSubscriptionEventArgs);
```

Esimerkkikoodi 9. SumoConnect-luokan metodi, joka ohjaa tilausmallin avulla saadut tiedot niitä vastaaville SumoObject-olioille.

SumoObject-luokka sisältää OnSumoUpdate-abstraktimetodin, eli tämä metodi on toteutettava kaikissa luokan perivissä luokissa. Tätä metodia kutsutaan kaikille SumoObject-olioille, joiden tunnisteella saadaan tietoa liikennesimulaatiosta. Metodia kutsutaan siis aina, kun SUMO:n simulaatio etenee askeleen, näin ollen metodin nimen haluttiin kuvastavan Unity-pelimoottorin sisäänrakennettuja Update- ja FixedUpdate-metodeja. OnSumoUpdate saa parametrinä olion tiedot liikennesimulaatiosta ja näin simulaation lähettämästä tiedosta saadaan haluttujen muuttujien arvot, on käytettävä esimerkkikoodi P:n mukaisesti TraCI.NET-kirjaston ResponceByVariableCode-sanakirjarakennetta, jonka avaimena käytetään TraCIConstants-luokan sisältämiä tyyppejä. Esimerkkikoodissa 10 haetaan ajoneuvon sijainti VAR_POSITION3D-tyypin avulla, minkä jälkeen se käännetään Unity-pelimoottorin koordinaatistoon ja annetaan Move- eli liikuta-metodille, joka vastaa ajoneuvo-olion liikkumisesta kartalla.

```
public override void OnSumoUpdate(VariableSubscriptionEventArgs args)
{
     var position = SumoMap.SumoToUnityPos(args.ResponseByVariable-
Code[TraCIConstants.VAR_POSITION3D].GetContentAs<Position3D>());
     Move(position);
     Rotate()
}
```

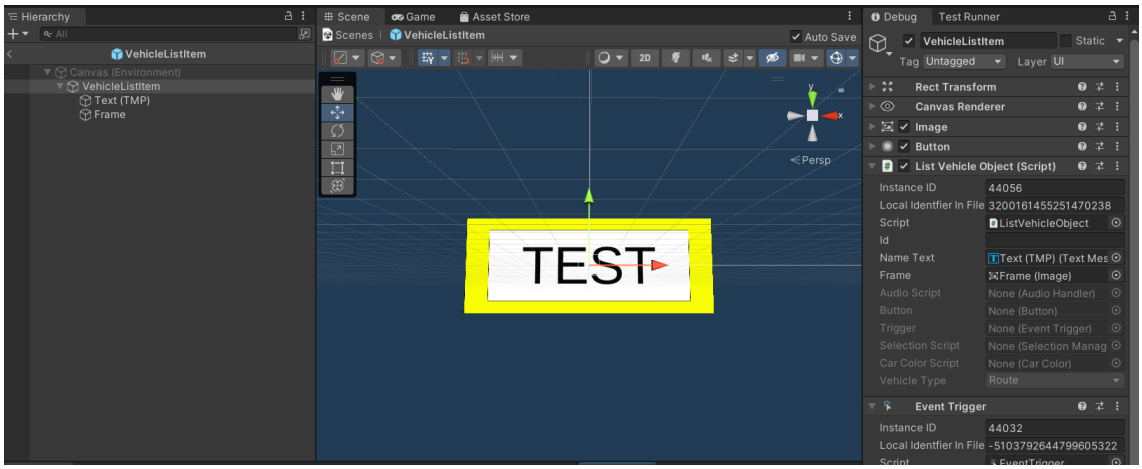
Esimerkkikoodi 10. Car-luokan toteutus OnSumoUpdate-metodista.

Ajoneuvojen liikkumisessa kartalla on myös tärkeää, että niiden kuvakkeet osoittavat ajoneuvon suunnan. Insinöörityössä törmättiin haasteisiin, kun SUMO-liikennesimulaattorista saatu angle, eli ajoneuvon kulmaa kuvaava muuttuja, ei riittänytkään ajoneuvojen rotaation kuvaamiseen oikein. Ongelmaa selvitettiin ja monia ratkaisuja implementoitiin, mutta osan ajoneuvoista kulkiessa oikein päin osa oli kääntynyt sivuttain eivätkä ne enää kulkeneet tien kulkusuuntaan. Jotta löydettäisiin ratkaisu mahdollisimman nopeasti, päätettiin jättää simulaattorin kautta saatu arvo huomioimatta ja esimerkkikoodin 11 mukaisesti toteutettiin ratkaisu, jossa ajoneuvo-olion rotaatio määräytyy sen liikkeen suunnan mukaisesti. Jotta ajoneuvon kuvake ei kääntyilisi liian paljon tai nopeasti, lisättiin esimerkkikoodin mukainen minimivaatimus liikutusta matkasta.

```
public void Rotate()
{
     Vector3 dir = transform.localPosition - lastPos;
     if(dir.magnitude < minMoveForRotation) return;
     float angle = Mathf.Atan2(dir.y,dir.x) * Mathf.Rad2Deg;
     transform.rotation = Quaternion.AngleAxis(angle, Vector3.forward);
}
```

Esimerkkikoodi 11. Car-luokan Rotate-metodi.

Lisättäessä ajoneuvo-olio näkymään pitää sitä vastaava elementti lisätä ajoneuvolistaan. Tähän käytetään prefab-resurssia, joka koostuu kuvakkeesta ja interaktiivisesta napista, jonka avulla ajoneuvo valitaan (kuva 7). Kun ajoneuvo on valittu näin tai klikkaamalla ajoneuvo-oliota kartassa, voidaan sille antaa komentoja käyttöliittymän avulla. Nämä komennot voivat käyttää TraCI-rajapintaa staattisen SumoConnect-instanssin kautta käytettävissä olevan TraCIClient-olion kautta ja tarvittava ajoneuvon tunniste saadaan VehicleList-olion kautta, sillä se pitää kirjaa valitusta ajoneuvosta.



Kuva 7. VehicleListItem-peliolio. Oikealla "inspector"-ikkunassa lista olion komponenteista.

Ajoneuvot liikkuvat näin ollen kartalla SUMO-liikennesimulaatiosta saadun sijaintitiedon avulla ja niiden lisääminen ja poistamien tehtiin mahdollisimman jouhevaksi. Ajoneuvojen lisääminen Manuscript-luokasta on täten suoraviivaista, mikä mahdollistaa hankkeen tarpeiden muuttuessa minimaaliset muutokset ajoneuvojen funktionaalisuuteen. Kuvassa 8 on ajoneuvo-prefab, joka sisältää ajo-

neuvon kuvakkeen ja CarName- eli ajoneuvon nimi -olion, jonka avulla ajoneuvot ovat helpommin tunnistettavissa karttanäkymästä. Ajoneuvo-olio sisältää myös komponentteja, jotka esimerkiksi määrittävät sen värin näkymässä.

Kuva 8. Ajoneuvo-peliolio.

5.5 Kartan luonti

SUMO-liikennesimulaattorissa käytettävä kartta oli tuotava Unity-pelimoottoriin, jotta liikenteenhallintasovellukseen saatiin haluttu karttanäkymä. Toteutuksia pohdittiin projektin suunnitteluvaiheessa, ja mahdollisia toteutuksia päätettiin olevan kaksi, joista ensimmäinen oli toteuttaa oma versio SUMO-liikennesimulaattorin sumo-gui-sovelluksen karttanäkymästä. Tämä siis olisi tarkoittanut ensin SUMO:n karttatiedoston lukemista ja sen särmien sekä risteyksien kartoittamista Unityn koordinaatistoon, minkä jälkeen olisi toteutettava graafinen esitys tästä tieverkostosta, esimerkiksi muodostamalla siitä kuva tai luomalla joko prefab-resurssien tai dynaamisesti muodostettujen olioiden avulla olioista koostuva tiestö. Tämän toteutuksen etuna on tarkka ajoneuvojen sijainnin kääntäminen liikennesimulaatiosta pelisovelluksen koordinaateiksi. Toteutettu kartta voi myös

olla visuaalisesti juuri sennäköinen kuin halutaan. Kuitenkin haasteena tässä toteutuksessa on sen monimutkaisuus ja siitä seuraten sen kehityksen vaatima aika. Koska hankkeessa pyrittiin liikennesimulaation integroinnin lisäksi tuottamaan simuloituja ongelmatilanteita, oli tarve saada integraation minimivaatimukset eli toimiva kartta ja sillä liikkuvat ajoneuvot valmiiksi mahdollisimman nopeasti.

Seurauksena kartan itse luomisen haasteista päädyttiin vaihtoehtoon, jossa kartta toteutettiin SUMO-liikennesimulaattorin simuloiman reaalimaailman alueen karttakuvana. Karttakuva piti saada määriteltyä vastaamaan samaa aluetta, jota SUMO simuloi, tai seurauksena olisi ajoneuvojen liikkuminen sattumanvaraiselta näyttäviä linjoja pitkin, jotka eivät välttämättä olisi edes osa tiestöä. Tämä toiminnallisuus saatiin hyödyntämällä MapBox-yrityksen tarjoamien palvelujen rajapintoja. Tässä toteutuksessa käytettiin MapBoxin StaticImages-rajapintaa, joka palauttaa karttakuvan koordinaateilla spesifioidusta alueesta. Jotta saatiin valittua juuri sama alue kuin liikennesimulaatiossa, tuli kirjata muistiin koordinaatit, joista simulaation kartta oli luotu OSM Web Wizard -työkalun avulla. Kehityksen nopeuden kannalta oli hyödyllistä, että nämä koordinaatit oli automaattisesti merkitty OSM Web Wizard -työkalun luomiin karttatiedostoihin. Näin saatiin haettua karttakuva simulaation alueesta. (33.)

Koska kartta tuotiin Unity-pelimoottoriin kuvana, piti ajoneuvojen sijainti uudelleen kartoittaa SUMO-liikennesimulaattorin kolmiulotteisesta koordinaatistosta kaksiulotteiseksi sijainniksi kuvalla. Ajoneuvo-oliot lisättiin kuvan lapsiolioiksi, jotta niiden paikallista sijaintia, eli sijaintia suhteessa vanhempiolion keskipisteeseen, voitaisiin käyttää sijainnin määrittämiseen kartalla. Tätä toiminnallisuutta varten kehitettiin SumoMap-luokka. SumoMap-luokka sisältää muuttujat, jotka kuvaavat SUMO-liikennesimulaattorin kartan kulmien koordinaatteja, eli pienimmän ja suurimman pituusasteen sekä pienimmän ja suurimman leveysasteen arvot. Näiden lisäksi SumoMap-luokka sisältää samat tiedot Unityn karttakuvasta, ja näiden muuttujien arvot lasketaan automaattisesti kuvan leveydestä ja korkeudesta esimerkkikoodin 12 mukaisesti.

```
private void Awake()
{
     var map = MapTransform.rect;
     UnityWidthMin = (map.width / 2d) * -1d;
     UnityWidthMax = map.width / 2d;

     UnityHeightMin = (map.height / 2d) * -1d;
     UnityHeightMax = map.height / 2d;
}
```

Esimerkkikoodi 12. SumoMap-luokan Awake-metodi, joka suoritetaan ohjelman käynnistyessä.

Jotta sijaintitieto liikennesimulaattorista saadaan pelimoottoriin yhteensopivaksi, käytetään SumoMap-luokan metodia SumoToUnityPos, joka esitellään esimerkkikoodissa 13. Tämä metodi ottaa parametrina SUMO:ssa käytettävän kolmiulotteisen sijaintivektorin ja palauttaa Unityn käyttämän kolmiulotteisen vektorin, jota käytetään ajoneuvon paikallisen sijainnin määrittämiseen. Uudelleenkartoittamista varten kehitettiin SumoMap-luokkaan MapTo-metodi, joka esitellään esimerkkikoodissa 14. Siinä arvo kahden luvun välillä muunnetaan vastaamaan arvoa kahden toisen luvun välillä. Näin saadaan muutettua koordinaatit SUMO:n ja Unityn välillä, tosin on myös huomioitava Z- ja Y-akselien vaihtuminen päittäin sovellusten välillä.

```
public static Vector3 SumoToUnityPos(Position3D positionInSumo)
{
     Vector3 result = new Vector3(
          (float)MapTo(SumoLonMin, SumoLonMax, UnityWidthMin, Uni-
tyWidthMax, positionInSumo.X),
          (float)MapTo(SumoLatMin, SumoLatMax, UnityHeightMin, Uni-
tyHeightMax, positionInSumo.Y),
          0f
     );
     return result;
}
```

Esimerkkikoodi 13. SumoMap-luokan SumoToUnityPos-metodi, jolla koordinaatit muutetaan SUMO:sta Unityyn.

```
public static double MapTo(double fromRangeMin, double fromRangeMax,
double toRangeMin, double toRangeMax, double valueToMap)
{
     return Mathf.Lerp((float)toRangeMin, (float)toRangeMax, Mathf.In-
verseLerp((float)fromRangeMin, (float)fromRangeMax, (float)valueTo-
Map));
}
```

Esimerkkikoodi 14. SumoMap-luokan staattinen MapTo-metodi.

Projektin alussa tämä karttakuva haettiin manuaalisesti komentorivin cURL-ohjelmalla. Jotta projektin osana tuotettu integraatio ei vaatisi tätä manuaalista karttakuvan hakemista, kehitettiin Unity-pelimoottoriin laajennus. Tämä laajennus lisää työkalupalkin kautta avattavan työkalun, jolla karttakuvan hakeminen ja oikeiden koordinaattien hakeminen karttatiedostosta voidaan tehdä graafisen käyttöliittymän kautta. Tämä MapCreator-työkalu kehitettiin Unityn sisältämien laajennustyökalujen avulla. Projektin edetessä päätettiin, että simulaation kartta-alueen lisäksi olisi hyvä saada karttakuvat myös ympäröivistä alueista, jotta liikenteenvalvontajärjestelmä vaikuttaisi käyttäjistä realistisemmalta. Tämän päätöksen seurauksena rajapinnasta haetaan yhdeksän samankokoista kuvaa, joista yksi on varsinainen simulaation kattama alue ja kahdeksan muuta ovat sitä ympäröiviä samankokoisia alueita. Yksi työkalun merkittävimmistä haasteista oli Static Images -rajapinnan maksimiresoluutio, 1280 kuvapistettä korkea ja 1280 kuvapistettä leveä (33). Näin ollen ei-neliömäisen simulaation karttakuvaa hakiessa tulisi suhteuttaa resoluutio maksimaalisen korkeaksi, mutta kuitenkin säilyttää oikea kuvasuhde.

Toinen isoista haasteista työkalun kehityksessä oli koordinaattien lukeminen SUMO-liikennesimulaattorin karttatiedostosta. Ensimmäisenä vaiheena oli luotava käyttöliittymään tapa valita haluttu karttatiedosto. Tämä toteutettiin hyödyntäen Unity-pelimoottorin työkalukehitykseen luotua apuluokkaa EditorUtility ja sen sisältä löytyvää OpenFilePanel-metodia. Se avaa kutsuttaessa käyttöjärjestelmän tiedostojen valintaan tarkoitetun ohjelman uudessa ikkunassa ja palauttaa polun valittuun tiedostoon. Kun tiedoston polku, eli sijainti on tiedossa, se voidaan esimerkkikoodin 15 mukaisesti avata ja käydä rivi kerrallaan läpi. Jotta työkalun ei tarvitse pitää tiedostoa turhaan auki tai lukea ylimääräisiä rivejä tiedostusta, tallennetaan haluttu rivi muistiin, minkä jälkeen tiedosto suljetaan. Seuraavaksi on eroteltava esimerkkikoodin mukaisesti rivin muodostavat osat toisistaan ja luettava tieto kohdasta ”origBoundary” eli alkuperäinen raja. Nämä tiedot tallennetaan coords-liukulukutaulukkomuuttujaan, jotta niitä voidaan käyttää kuvan hakemiseen rajapinnasta. Esimerkkikoodi 16 näyttää halutun rivin rakenteen SUMO:n karttatiedostossa.

```
private double[] ReadCoordinates(string path)
{
     var coords = new double[4];
     string[]? locationInfo = null;

     // read the map file line by line and get the line that has the
location information
     var lines = File.ReadLines(path);
     foreach (var line in lines)
     {
          if (!line.Trim().StartsWith("<location")) continue;
          locationInfo = line.Split(' ', StringSplitOptions.RemoveEmp-
tyEntries);
          break;
     }

     // Check that the location information was found in the file, re-
turn null if it's not found
     if (locationInfo == null) { return null; }

     foreach (var info in locationInfo)
     {
          // split the location info in to parts, only one of these
contains the coordinates needed
          var x = info.Split('=');

          // Replace all quote marks with spaces so they can be trimmed
out
          if (x[1].Contains('\"')) { x[1] = x[1].Trim().Replace('\"', '
'); }

          // Skip if the data is not what we are looking for
          if (x[0] != "origBoundary") continue;

          // Separate the coordinate numbers and Convert them from
strings to doubles
          var coordStrings = x[1].Split(',');
          for (var i = 0; i < coordStrings.Length; i++)
          {
               var tmpString = coordStrings[i].Trim().Replace('.',
',');
               coords[i] = Convert.ToDouble(tmpString);
          }
          break;
     }

     return coords;
}
```

Esimerkkikoodi 15. MapCreator-luokan ReadCoordinates-metodi, joka lukee tiedostosta halutut koordinaatit.

```
<location netOffset="-393722.21,-6673610.19" convBound-
ary="0.00,0.00,6718.19,4999.59" origBound-
ary="10.860532,53.941410,25.346675,60.231426" projParameter="+proj=utm
+zone=35 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"/>
```

Esimerkkikoodi 16. SUMO:n karttatiedoston rivi, joka sisältää koordinaatit.

Itse kuvien hakeminen toteutettiin kuitenkin työkalussa edelleen cURL-ohjelman avulla. Tämä yksinkertaisti kehitystä, ja sen seurauksena työkalun kehittäminen vei huomattavasti vähemmän aikaa. Työkalun toteutus jollain toisella tavalla olisi ollut myös mahdollista, mutta tavoitteena oli kehittää jatkokehitystä ja korjauksia nopeuttava työkalu, joten ylimääräinen työ olisi yksinkertaisesti ollut turhaa ja vienyt kehitysaikaa liikenteenvalvontajärjestelmän ydintoiminnoilta. Kuvan hakemiseen käytettävä cURL-ohjelma käynnistettiin SUMO:a vastaavalla tavalla hyödyntäen System.Diagnostics-nimiavaruuden Process-luokan Start-metodia.

5.6 Jatkokehitys

Liikennesimulaattorin ja pelimoottorin integraatiossa keskityttiin insinöörityössä pitkälti toteuttamaan niin kutsuttu pienin toimiva tuote, eli sellainen tuotteen versio, joka sisältää pelkästään välttämättömät toiminnalliset osat. Tämä oli seurausta projektin aikatauluista, sillä hankkeen toiset osuudet, jotka hyödynsivät toteutettua liikenteenvalvontajärjestelmää, tarvitsivat omaan työskentelyynsä käyttökelpoisen toteutuksen mahdollisimman nopeasti. Kuitenkin tavoitteena oli alusta asti jatkaa ensimmäisen version jatkokehitystä. Tavoitteena SUMO-liikennesimulaattorin ja Unity-pelimoottorin kannalta on tuoda useampi toiminto käytettäväksi SumoConnect-luokan tai SumoObject-luokan kautta. Näin ei olisi tarpeen aina hakea viittausta SumoConnect-luokan TraCIClient-olioon, jolloin integraatio tuntuisi vahvemmalta.

Muita jatkokehityksen kohteita integraation kannalta olisivat esimerkiksi SumoMap-luokan tiedot kartan koosta, jotka voitaisiin hakea automaattisesti SUMO:n simulaatiotiedostoista, toisin kuin tällä hetkellä, kun tiedot täytyy manuaalisesti etsiä karttatiedostosta. Myös SUMO:n käyttämien tietorakenteiden muuttaminen Unityn tietorakenteiksi C#-ohjelmointikielen laajennusmetodeja hyödyntäen olisi hyvä tavoite. Tämän toteutuksen avulla SUMO:n käyttämillä tietorakenteilla olisi ylimääräinen metodi, jonka avulla muuntaminen onnistuisi.

Karttakuvien luonti tehtiin työkalun avulla jatkokehityksen kannalta helpoksi. Näin esimerkiksi eri alueiden mallintaminen ja tuonti osaksi liikenteenvalvontajärjestelmää olisi huomattavasti nopeampaa ja helpompaa. Karttatyökalua hyödyntämällä vältytään myös mahdollisilta huolimattomuusvirheiltä rajapintakutsussa.

Jatkokehityksen mahdollisuuksiin kuuluu myös suunnitteluvaiheessa pohdittu tietokanta, johon tallennettaisiin hankkeen toteuttaman liikenteenhallintajärjestelmän käyttäjän kuormituksen testituloksia. Tätä toteutusta varten suunniteltiin käytettävän CSC:n, eli suomalaisen Tieteen tietotekniikan keskuksen, turvallisia tietokantatoteutuksia (34). Näin tutkimusdata olisi helposti hankkeen muiden toimihenkilöiden saatavilla. Myös ohjelman käyttöjärjestelmää voisi jatkokehittää, koska se sisältää joitain ominaisuuksia, joiden toiminnallisuus on vain osittain valmis. Tästä on esimerkkinä sovelluksen sisältämä säätietoelementti, joka insinöörityön kirjoittamisen aikana ei ole riippuvainen reaalimaailman säästä, kuten alun perin suunniteltu, vaan määräytyy satunnaisfunktion arvon mukaan.

Yleisesti jatkokehityksen mahdollisuudet huomioitiin ja sen edellytyksiä parannettiin kirjaamalla projektin koodiin suuri määrä kommentteja, joilla avataan toteutuksien käyttötarkoituksia ja jopa mahdollisia vajavuuksia. Näin projektin jatkaminen ja ryhmätyöskentely helpottuu ja ryhmään mahdollisesti liittyvät kehittäjät kykenevät helpommin ymmärtämään koodin toiminnallisuuksia ja rajoitteita sekä mahdollisia virheilmoitusten lähteitä. Tätä dokumentaatiota ja projektin GitLab-versionhallintasivuston tehtävälistaa päivitettiin aktiivisesti työn edetessä.

5.7 Loppuanalyysi

Tavoitteena oli kehittää liikenteenvalvontasovellus, jolla voidaan testata eri ajoneuvomäärien valvonnan ja niille tapahtuvien ongelmatilanteiden aiheuttamaa kuormitusta. Tämän tavoitteen mukaisen ohjelman toteuttaminen insinöörityössä onnistui. Myös insinöörityön tavoite liikennesimulaattorin ja pelimoottorin integroinnista onnistui osana projektia. Tämän onnistumisen kannalta olivat

merkityksellisiä kummankin valitun ohjelman, SUMO-liikennesimulaattorin ja Unity-pelimoottorin, laaja dokumentaatio ja näiden ohjelmien ominaisuudet.

SUMO-liikennesimulaattori oli erinomainen valinta sovelluksen kehityksen kannalta. Sen kyky simuloida suuriakin tieverkostoja ja monia yksittäisiä ajoneuvoja mahdollisti sen käytön projektissa. Se tarjoaa myös kaksi tapaa toteuttaa liikennesimulaattorin integraatio, libsumo-toteutus ja TraCI-rajapinta. TraCI-rajapinnan avulla oli mahdollista toteuttaa toiminnallisuudeltaan tarpeeksi kattava integraatio projektin aikana. Myös SUMO:n vähäinen resurssien tarve helpotti suuresti sekä Unity-toteutuksen että SUMO:n liikennesimulaation samanaikaista suorittamista.

TraCI.NET-kirjasto mahdollisti myös laajalta osin toteutuksen valmistumisen projektin aikana. Sen puutteellisesta TraCI-rajapinnan toteutuksesta huolimatta se mahdollisti kaikkien projektissa tarvittavien ominaisuuksien implementoinnin, ilman että oli tarvetta käsitellä pelkästään tavuista koostuvia TraCI-rajapinnan viestejä. Koska Unity-pelimoottori oli kehitystiimille entuudestaan tuttu, ei sen tai sen kautta toteutetuissa ominaisuuksissa ilmennyt suurempia haasteita.

Projektin suunnitteluvaiheessa päätetyistä ideoista osa ei toteutunut ollenkaan, tai niitä ei toteutettu vielä insinöörityön kirjoittamisen aikana, sillä tärkeintä oli saada kaikki perustoiminnallisuudet valmiiksi. Kuitenkin osaan ominaisuuksista, kuten joihinkin liikennevalvontasovelluksen ongelmatilanteista, oli jo käytetty huomattavan paljon aikaa. Näiden ominaisuuksien poisjättämisen seurauksena aikaa meni hukkaan. Turhan työn tekemiseltä tosin pääosin vältyttiin ja osa toteutuksista voidaan implementoida myöhemmin.

6 Yhteenveto

Autonominen liikenne kehittyy jatkuvasti. Täysin autonomiset ajoneuvot kykenevät ohjaamaan itseään kaikissa olosuhteissa ja tilanteissa. Automaation tasoja on kuitenkin monia, ja uudet markkinoille tulevat vaihtoehdot tarjoavat koko ajan

kehittyneempää automaatiota, vaikka täysin autonomisia ajoneuvoja ei vielä olekaan laajamittaisesti liikenteessä.

Liikenteen simulointiin on olemassa erilaisia vaihtoehtoja, jotka eriävät toisistaan valittujen tekniikkojen ja teknologioiden osalta. Insinöörityössä käytetty SUMO-liikennesimulaattori oli toimintaperiaatteeltaan jatkuvan ajan mikroskooppinen simulaatio. Tämä siis tarkoitti, että simuloitiin reaaliajassa kahdenkymmenen millisekunnin jaksoja liikenteestä, jossa jokainen ajoneuvo oli simuloitu yksittäin. Liikennesimulaattorin mukana tulleet ja sitä tukevat työkalut, kuten netedit-verkostonmuokkaustyökalu ja OSM Web Wizard -niminen työkalu, jolla OpenStreetMap-karttapalvelun kartat voidaan muuntaa SUMO-liikennesimulaattorin kartoiksi, tukivat itse simulaation kehitystä merkittävästi.

Liikenteenvalvontasovellus tuotettiin Unity-pelimoottorilla, joka kehitysryhmän aikaisemman kokemuksen avulla nopeutti ja helpotti kehitystyötä suuresti. Liikennesimulaattorin integrointi oli käytännöllistä sekä ulkoisen avoimen lähdekoodin ohjelmakirjaston TraCI.NET:n avulla että SUMO-liikennesimulaattorin implementoiman TraCI-rajapinnan ansiosta. Ilman näitä toteutuksen kehittäminen SUMO:n ja Unityn yhdistämiseksi olisi kestänyt huomattavasti pidemmän aikaa.

Projekti kehitettiin osana laajempaa Metropolia Ammattikorkeakoulun SAM-hanketta, ja sen jatkokehitysmahdollisuudet ovat hyvät, eikä niitä rajoita esimerkiksi liian jäykkä toteutus simulaattorin integraatiosta. Parannettavaa esimerkiksi työn dokumentoinnissa olisi, sillä jotkin projektin osat ovat monimutkaisempia, mutta sisältävät silti vähemmän kommentteja, kuin toiset osat.

Liikennesimulaattorin integrointi pelimoottoriin onnistui hyvin, ja kaikki tarvittavat toiminnallisuudet toteutettiin onnistuneesti. Vaikka toiminnallisuudet olivat kattavat, ei niissä kaikilta osin saatu hyödynnettyä kaikkea uuden integraation mahdollistamaa automatisointia. Insinöörityön tavoite, liikennesimulaattorin integrointi pelimoottoriin, saavutettiin onnistuneesti aikarajan sisällä, ja lopullinen tuote soveltui käyttötarkoitukseensa hyvin.

Lähteet

1 Verkottunut ja automatisoituva tieliikenne. 2021. Verkkoaineisto. Traficom. <https://www.traficom.fi/fi/liikenne/liikennejarjestelma/verkottunut-ja-automatisoituva-tieliikenne>. Luettu 15.7.2022.

2 Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. 2021. Verkkoaineisto. SAE International. <https://doi.org/10.4271/J3016_202104>. Luettu 15.7.2022.

3 Shuttleworth, Jennifer. 2019. Levels of driving automation. Verkkoaineisto. SAE International. <https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic>. Luettu 15.7.2022.

4 Lindblom, Pekko. Älykäs autonominen liikenne - Smart Autonomous Mobility. Verkkoaineisto. Metropolia Ammattikorkeakoulu. <https://www.metropolia.fi/fi/tutkimus-kehitys-ja-innovaatiot/hankkeet/sam>. Luettu 1.6.2022.

5 Tieliikenteen automaatiokokeilut. 2018. Verkkoaineisto. Traficom. <https://www.traficom.fi/fi/liikenne/tieliikenne/tieliikenteen-automaatiokokeilut>. Luettu 23.8.2022.

6 Lopez, Pablo Alvarez; Behrisch, Michael; Bieker-Walz, Laura; Erdmann, Jakob; Flötteröd, Yun-Pang; Hilbrich, Robert; Lücken, Leonhard; Rummel, Johannes; Wagner, Peter & Wießner, Evamarie. 2018. Microscopic Traffic Simulation using SUMO. Verkkoaineisto. IEEE Intelligent Transportation Systems Conference. <https://elib.dlr.de/127994/>. Luettu 5.9.2022.

7 Institute of Transportation Systems. Verkkoaineisto. DLR. <https://www.dlr.de/ts/en/desktopdefault.aspx/tabid-1221/1665_read-3070/>. Luettu 5.9.2022.

8 marouter. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/TraCI.html>. Luettu 29.8.2022.

9 Brügmann, Johannes. 2015. Modelling and Implementation of a Microscopic Traffic Simulation System. Berlin: Logos Verlag Berlin.

10 SUMO Road Networks. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/Networks/SUMO_Road_Networks.html>. Luettu 1.6.2022.

11 netedit. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/Netedit/index.html>. Luettu 1.6.2022.

12 OSM. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/Tools/Import/OSM.html>. Luettu 29.8.2022.

13 Definition of Vehicles, Vehicle Types, and Routes. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/Definition_of_Vehicles%2C_Vehicle_Types%2C_and_Routes.html>. Luettu 2.6.2022.

14 TraCI. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/TraCI.html>. Luettu 1.6.2022.

15 Protocol. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/TraCI/Protocol.html>. Luettu 1.6.2022.

16 TraCI.NET. Verkkoaineisto. CodingConnected. <https://github.com/CodingConnected/CodingConnected.Traci/tree/master/CodingConnected.TraCI.NET>. Luettu 1.6.2022.

17 UsageExample. Verkkoaineisto. CodingConnected. <https://github.com/CodingConnected/CodingConnected.Traci/blob/master/TracCI.NET-Usage-example/UsageExample.cs>. Luettu 4.6.2022.

18 Unity Personal. Verkkoaineisto. Unity Technologies. <https://store.unity.com/products/unity-personal>. Luettu 14.9.2022.

19 Unity. Verkkoaineisto. Unity Technologies. <https://unity.com>. Luettu 14.9.2022.

20 A tour of the C# language. Verkkoaineisto. Microsoft Corporation. <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>. Luettu 14.9.2022.

21 GameObject. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/GameObject.html>. 29.10.2022. Luettu 14.9.2022.

22 Scenes. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/CreatingScenes.html>. 28.10.2022. Luettu 14.9.2022.

23 The Scene view. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/UsingTheSceneView.html>. 28.10.2022. Luettu 15.9.2022.

24 Unity’s interface. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/UsingTheEditor.html>. 11.11.2022. Luettu 13.11.2022.

25 Prefabs. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/Prefabs.html>. 28.10.2022. Luettu 15.9.2022.

26 Format of Text Serialized files. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/FormatDescription.html>. 28.10.2022. Luettu 29.10.2022.

27 Libsumo. Verkkoaineisto. SUMO. <https://sumo.dlr.de/docs/Libsumo.html>. Luettu 1.6.2022.

28 Murray, J. W. 2014. C# game programming cookbook for Unity 3D. Boca Raton, FL: CRC Press.

29 System.Diagnostics Namespace. Verkkoaineisto. Microsoft Corporation. <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics>. Luettu 13.11.2022.

30 Broshar, Alisdair. 2021. Introduction to Synchronous and Asynchronous Processing. Verkkoaineisto. Koyeb. <https://www.koyeb.com/blog/introduction-to-synchronous-and-asynchronous-processing>. Luettu 30.9.2022.

31 Important Classes - MonoBehaviour. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-MonoBehaviour.html>. 28.10.2022. Luettu 16.9.2022.

32 Time. 2022. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-TimeManager.html>. 28.10.2022. Luettu 16.9.2022.

33 Static Images. Verkkoaineisto. Mapbox. <https://docs.mapbox.com/api/maps/static-images/>. Luettu 1.6.2022.

34 Services for Sensitive Data. Verkkoaineisto. Tieteen tietotekniikan keskus Oy. <https://research.csc.fi/sensitive-data>. Luettu 16.8.2022.