Richard Domander

Agenttien uskottavuus ja tekotyperyys realistisessa autopelissä

Tietotekniikan
(ohjelmistotekniikka)
pro gradu -tutkielma
26. marraskuuta 2014

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Richard Domander
Yhteystiedot: richard.domander@gmail.com
Työn nimi: Agenttien uskottavuus ja tekotyperyys realistisessa autopelissä
Title in English: The Believability of Agents and Artificial Stupidity in a Simulated Racing Game
Työ: Tietotekniikan (ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 112
Tiivistelmä: "Tämä pro gradu -tutkielma tarkastelee agenttien uskottavuutta realistisessa autopelissä. Tällaisessa pelissä agentit ohjaavat pelaajan kanssa kilpailevia autoja. Uskottavuus on varsin subjektiivinen ilmiö, joka riippuu niin pelistä kuin pelaajasta. Se kuvaa, kokeeko pelaaja olevansa tekemisissä älykkään, itsenäisen toimijan kanssa, joka tarjoaa hänelle sekä sopivasti haastetta että riittävästi viihdettä. Opinnäytteeni käsittelee myös tekotyperyyttä. Se tarkoittaa virheiden ja puutteiden tietoista lisäämistä agentin toteutukseen. Tekotyperyyden tarkoituksena on parantaa uskottavuutta. Teoksen laaja tekninen osuus käsittelee *Desideriusta*, joka on TORCS-autopeliin ohjelmoimani agentti. TORCS on avoimen lähdekoodin peli, joka on saatavilla niin Windowsille, Linuxille kuin Macille. Sitä käytetään laajalti akateemisessa tutkimuksessa. Ohjelmoin *Desideriuksen* tutkiakseni, mitä uskottavuus tarkoittaa autopelissä, miten eri heuristiikat vaikuttavat siihen ja millaisia tekotyperiä tekniikoita on mahdollista toteuttaa. Lopputulos on epäonnistunut, sillä *Desiderius* on teknisesti liian yksinkertainen ollakseen uskottava. Se ei myöskään tarjoa riittävää alustaa uskottavien tai tekotyperien tekniikoiden kokeilemiselle."
English abstract: "This Master's thesis studies the believability of agents in a racing simulation game. In a racing game, agents control the vehicles not driven by the player. The believability of an agent is a highly subjective phenomenon which depends on the individual player and the game played. Generally, it describes whether the player feels she is facing an intelligent, independent actor, which presents her a fitting challenge and provides adequate entertainment. The thesis also presents the concept of artificial stupidity. Artificial stupidity denotes the intentional programming of mistakes and limitations into the behaviour of an agent. Its goal is to improve believability. The thesis also has an extensive technical portion, which describes the agent *Desiderius* I developed for TORCS. TORCS is an open source, cross-platform racing game which is widely used in academic research. The goal of project *Desiderius* was to explore how different heuristics affect the believability of an agent, what does believability mean in a racing game, and how can you imple-

ment artificial stupidity into one. As a whole, the project is a failure. The agent is too simplistic to be believable, or to provide a sufficient platform for testing out new techniques."

Avainsanat: Tekoäly, agentti, videopeli, tekotyperyys, uskottavuus, TORCS, autopeli, simulointi, kilpa-ajo, fysiikka, ohjelmointi, heuristiikka, haastavuus, viihdyttävyys

Keywords: Artificial intelligence, agent, video game, artificial stupidity, intelligent mistake, intentional mistake, believability, TORCS, racing game, simulation, racing, physics, programming, heuristic, challenge, enjoyability

Sisältö

Kuvat

1 Johdanto

"Ei siis pidä pyyhkiä pois matkamme mutkia, sillä mitään muutahan ei käsiimme jäänyt."
-Stanislaw Lem, Isännän ääni, 1. luku

Tämä pro gradu -työ on jatkoa omalle kandidaatintutkielmalleni [5], joka on kirjallisuuskatsaus agentin uskottavuuteen ja tekotyperyyteen peleissä. Tämä työ käsittelee aihetta syvemmin, sillä se keskittyy ihmisvastustajaa simuloiviin agentteihin eli *botteihin* ja autopeleihin. Pro graduni tutkii, mitä botin uskottavuus tarkoittaa autopelissä, ja miten sellaiseen voi toteuttaa uskottavan botin. Työn erityisnäkökulmana on uskottavuuden lisääminen erilaisten tekotyperien tekniikoiden avulla. Tutkielman empiirisessä osuudessa selostetaan ja arvioidaan ohjelmoimani tekoäly. Arvioinnissa tarkastellaan niin suorituskykyä kuin uskottavuutta. Lisäksi pohditaan, miten toteuttamani tekotyperät tekniikat vaikuttavat uskottavuuteen. Ohjelmoimaani bottia arvioidaan sekä löytämääni teoreettista kirjallisuutta vasten että vertaamalla sitä muihin TORCS-pelin botteihin. TORCS on lyhenne pelin koko nimestä: *The Open Race Car Simulator*. Päätin kirjoittaa opinnäytteeni suomeksi, koska tietääkseni aiheesta ei ole aikaisempia julkaisuja suomen kielellä.

Ohjaajani Tommi Kärkkäinen ja Jussi Rasku ehdottivat, että tekisin pro graduni samasta aiheesta kuin kandidaatintutkielmani. He ideoivat, että laajemmassa opinnäytteessäni tutkisin uskottavan ja tekotyperän agentin ohjelmoimista johonkin avoimen lähdekoodin peliin. Valitsin TORCSin, koska olen kiinnostunut autopeleistä, ja koska uskoin löytäneeni hyvät ohjeet, kuinka kehittää siihen tekoäly. Alunperin tarkoituksenani oli sisällyttää työhön kyselytutkimus, jossa vertaan ohjelmoimaani bottia Wymannin [51] oppaassa esiteltävään bottiin. Tutkimuksessa haastateltaville olisi näytetty video molempien bottien ajosta, ja sitten kysytty kumpaan niistä eri adjektiivit pätevät paremmin. Uskottavuus on varsin subjektiivinen aihe [19], ja kyselytutkimuksella siitä olisi saanut edes jonkinlaista mitattavaa dataa. Projektin osoittautuessa kuormittavaksi päätin yhdessä ohjaajieni kanssa jättää kyselytutkimuksen pois. Esitin kuitenkin Instanssi 2013 -tapahtumassa[1] videon botistani, ja kysyin yleisön mielipidettä siitä. Opinnäytteen kuormittavuutta lisäsivät muun muas-

[1] `http://instanssi.org/2013/ohjelma/`

sa mittavat tekniset vaikeudet sekä puutteliset valmiuteni tekoälyn ohjelmoimiseksi. Työstin opinnäytteeni valmiiksi reilun kahden vuoden aikana.

Luku 2 on yleinen johdatus opinnäytteen aihepiiriin. Siinä selvitetään käsitteiden *tekoäly*, *agentti*, *botti*, *uskottavuus* ja *tekotyperyys* merkitys tässä teoksessa, sekä rajataan tutkimus koskemaan digitaalisia pelejä. Luvussa kerrotaan, miten agentit eroavat arkkitehtuuriltaan, ja miten pelin tyyppi vaikuttaa niiden toteutukseen. Lisäksi esitellään minkälaisissa eri rooleissa ne voivat toimia, ja miten agentin perustehtävä vaihtelee pelistä toiseen. Koska uskottavuudelle ei ole olemassa täsmällistä, kattavaa määritelmää [42], käsitettä pyritään tarkastelemaan mahdollisimman monipuolisesti. Lisäksi tutkitaan, miten tekotyperyys ja uskottavuus liittyvät toisiinsa. Luku on kuin referaatti kandidaatintutkielmastani, mutta se painottaa botteja voimakkaammin, koska ohjelmoimani tekoälyn päämääränä on simuloida ihmisvastustajaa. Luku pohjustaa myös luvun 5 syvempää, aihekohtaista teoriaa.

Luvussa 3 käsitellään ajamisen fysiikkaa sekä auton tekniikkaa. Sen tarkoituksena on auttaa ymmärtämään, mihin luvuissa 5 ja 6 esitettävät yhtälöt perustuvat, ja mistä niiden muuttujat juontuvat. Luku on melko pitkä, mutta se pyrkii antamaan lukijalle käsityksen siitä, miksi ajolinjan noudattaminen lyhentää kierrosaikaa, miksi auto ei voi kääntyä mielivaltaisen jyrkästi ja miksi lyhin reitti radan ympäri ei ole nopein. Toisin sanoen luku selventää kokonaisvaltaisia kilpa-ajamisen ilmiöitä. Se havainnollistaa myös, kuinka monimutkaista autopelin agentin heuristiikkojen suunnittelu ja optimointi on. Toisaalta autopelit ja ajamisen ilmiöt voivat lukijalle olla entuudestaan tuttuja, mutta näihin liittyvä tietämys on useimmiten intuitiivista, implisiittistä ja ”hiljaista” [33]. Luku 3 pyrkii tekemään tästä tiedosta eksplisiittistä ja formaalia. Se kertoo lukijalle, mihin fysiikan ilmiöihin hänen intuitionsa ajamisesta perustuvat.

Luvussa 4 kerrotaan TORCS-pelin historiasta ja ominaisuuksista. Pelistä on kaksi rinnakkaista toteutusta: työpöytä- ja palvelinversio [45]. Edellinen on tarkoitettu pelaajille, ja jälkimmäinen akateemiseen tutkimukseen ja bottien kilpailuttamiseen [20]. Päätin käyttää työpöytäversiota, koska arvioin botin toteuttamisen olevan sillä helpompaa, koska botilla on siinä enemmän tietoja käytettävissään. Lisäksi uskoin työpöytäversion asentamisen ja suorittamisen olevan yksinkertaisempaa. Myös Wymannin [51] bottiopas on laadittu työpöytäversiota varten. Pelin kummankin version pääpiirteet on kuitenkin hyvä tuntea, sillä teoksessa käsitellään molemmille ohjelmoituja botteja. Luvussa kerrotaan, miten botin toteutus sijoittuu työpöytäversion arkkitehtuurissa, sekä esitellään pelin bottimoduulien rakenne. Lisäksi selos-

tetaan palvelinversion rajapinta, ja palvelinversion erot työpöytäsovellukseen nähden.

Luvussa 5 pohditaan, miten agentin uskottavuus ilmenee autopeleissä sekä kerrotaan lyhyesti, miten genren eri pelit eroavat toisistaan. TORCS on realistinen autopeli, joka pyrkii simuloimaan ajamista tarkasti [21]. Tutkiessani kirjallisuutta löysin eriäviä näkemyksiä siitä, mikä agentin tehtävä on autopeleissä. Esimerkiksi Muñozin et al. [28] mielestä agentin tulee pelata kuin ihminen, kun taas West [50] katsoo, että sen tulee ennen kaikkea tarjota viihdettä. Luvussa pyrin sovittamaan mielipiteet toisiinsa ja osoittamaan, etteivät ne välttämättä ole ristiriidassa. Toisaalta Livingstonen [19] mukaan agentin eri tehtävät eivät ole toisensa poissulkevia. Täten ainakin TORCSin kohdalla voidaan mielestäni puhua perustellusti boteista, vaikka kaikki lähteet eivät sen agenteistä tätä termiä käytäkään. Luvussa kartoitan myös eri bottien käyttämiä tekniikoita. Esimerkkeinä käytetään pääasiassa TORCSin palvelin- ja työpöytäversiolle ohjelmoituja toteutuksia, mutta luvussa esitellään myös kaupallisten pelien menetelmiä. Luvussa lähinnä ideoidaan, minkälaisia tekotyperiä menetelmiä autopeliin voi toteuttaa.

Luvussa 6 selostan TORCSiin ohjelmoimani botin toteutuksen yksityiskohtaisesti. Seikkaperäisyys antaa oikean kuvan botin ohjelmoinnin laajuudesta sekä pyrkii tarjoamaan hyvän lähtökohdan niille, jotka haluavat luoda oman toteutuksensa. Luvun 6 tarkoituksena on kuvata ohjelmoimani tekniikat, kun taas luku 7 sisältää niiden arvioinnin. Luvussa 6 arvioidaan kuitenkin lyhyesti niitä menetelmiä, jotka jouduin jättämään koodista pois sekä kerrottaan, miksi luovuin niistä. Botin menetelmät esitetään matemaattisina kaavoina ja pseudokoodina. Valitsin nämä esitystavat selkeyden vuoksi. Luku 6 sekä sitä tukevat liitteet B ja C sisältävät myös C++-kieltä. Kielen erityispiirteitä käsitellään liitteessä A. Halusin esittää asioita ohjelmakoodina toisaalta vakuuttaakseni lukijan siitä, että olen todella tehnyt toteutuksen itse ja toisaalta analysoidakseni Wymannin [51] oppaan koodia. Luvussa 6 kerrotaan lisäksi ohjelmointityön lähtökohdista, etenemisestä sekä projektin aikana kohdatuista ongelmista. Siinä kuvataan kehitystyön haasteita sekä pohditaan mahdollisuuksia helpottaa sitä.

Luvussa 7 arvioidaan niin toteuttamani botin suorituskykyä kuin uskottavuutta. Lisäksi siinä tutkitaan, onnistuivatko kokeilemani tekotyperät tekniikat parantamaan uskottavuutta. Suorituskyvyn mittarina käytetään botin kierrosaikaa valitsemallani testiradalla. Bottini aikoja verrataan niin muiden TORCSin bottien kuin ihmispelaajien aikoihin. Uskottavuutta arvioidaan sekä luvun 5 teorian avulla että

vertaamalla bottiani Wymannin [51] toteutukseen. Lisäksi kerron botista saamastani yleisöpalautteesta. Luvussa perustelen, miksi ohjelmoin valitsemani tekniikat sekä mitä vaihtoehtoja minulla olisi ollut. Alaluvussa 7.4 arvioin koko opinnäytteen onnistumista.

Luku 8 on lyhyt yhteenveto opinnäytteeni sisällöstä. Siinä kerrataan lyhyesti työn sisältö ja rakenne, sekä kerrotaan, miten teoksen eri osat tukevat toisiansa. Luku palauttaa mieleen tutkimuskysymyksen sekä työn teknisen osuuden ja tulokset.

2 Uskottavat agentit ja tekotyperyys peleissä

Tässä luvussa kerrotaan, mitä *tekoälyllä* tarkoitetaan pro gradu -työssäni. Luvussa selvitetään, mitä ovat itsenäiset tekoälyohjelmat eli *agentit*, sekä kerrotaan, minkälaisia eri agentteja on olemassa. Kuten koko teos, luku keskittyy videopelien agentteihin. Luvussa kerrotaan, miten pelin genre sekä agentin rooli ja tehtävä vaikuttavat sen toteutukseen. Lisäksi tarkastellaan agentin *uskottavuutta*. Pelaaja pitää uskottavaa agenttia älykkäänä [24], haastavana ja viihdyttävänä [42]. Uskottavuuteen liittyy myös *tekotyperyys*. Tekotyperyydellä tarkoitan agentin toimintaan suunnitellusti ohjelmoituja virheitä ja puutteita. Tekotyperien tekniikoiden on tarkoitus parantaa agentin uskottavuutta [15, 18, 50]. Lopuksi esitellään tämän työn varsinainen tutkimuskohde *botit*. Kutsun boteiksi ihmispelaajaa imitoivia agentteja. Luvun tarkoituksena on johdattaa lukija opinnäytteen tutkimusaiheeseen sekä pohjustaa luvun 5 syventävää teoriaa. Alaluvut 2.1 – 2.4 sekä osin 2.5 on lainattu omasta kandidaatin tutkielmastani [5].

2.1 Tekoäly ja agentit

Tekoälyllä tarkoitetaan tietokoneohjelmaa, joka toimii näennäisen älykkäällä tavalla. Se on kuitenkin haastava käsite, sillä Negnevitskyn [29] mukaan edes *älykkyyden* määritelmästä ei ole yksimielisyyttä. Jotkut filosofit epäilevät, onko älykkyys itsenäinen tekijä, joka voidaan erottaa muista inhimillisistä piirteistä, kuten luovuudesta, tunteista tai moraalista. Toiset taas uskovat, että ainakin periaatteellisella tasolla koneet pystyvät kaikkeen mihin ihminenkin. Negnevitsky määrittelee älykkyyden kyvyksi oppia, ymmärtää, ratkaista ongelmia ja tehdä päätöksiä.

Turing [47] kiersi määritelmien hankaluudet kokonaan muotoilemalla tekoälytutkimukselle konkreettisen tavoitteen. Hän julkaisi vuonna 1950 artikkelin, jossa esittelee tavan testata tekoälyjä. Negnevitskyn [29] tulkinnan mukaan Turing esitti artikkelillaan, että tekoälytutkimuksen tulee pyrkiä luomaan kone, joka läpäisee älykästä käyttäytymistä mittaavan testin. Alkuperäisessä Turingin testissä kuulustelija keskustelee tietokoneen välityksellä kahden eri huoneessa olevan kumppanin kanssa. Toinen kumppaneista on mies ja toinen nainen. Testin ensimmäisessä vai-

heessa kuulustelijan on pääteltävä kumppaneiden sukupuoli oikein. Sääntöjen mukaan miehen on yritettävä hämätä, ja naisen vakuutettava olevansa nainen. Toisessa vaiheessa miehen korvaa tekoäly, joka yrittää myös petkuttaa olevansa nainen. Jos kone onnistuu huiputtamaan kuulustelijaa yhtä hyvin kuin ensimmäisen kierroksen mies, sen katsotaan läpäisseen testin. Turing kutsui testiään myös imitaatiopeliksi. Testin nykyversiossa ei välitetä keskustelijoiden sukupuolesta, vaan ainoastaan siitä, erehtyykö kuulustelija luulemaan tekoälyä ihmiseksi. Livingstone [19] huomauttaa, että vaikka tekoäly läpäisisi Turingin testin, se ei tarkoita, että ohjelma todella ajattelisi.

Searle [36] pohtii tekoälyn mahdollisuuksia ja määritelmää Kiinalaisena huoneena tunnetun ajatuskokeen avulla. Siinä kiinaa osaamaton mies on kiinankielisessä kirjastossa. Kirjaston tekstit jakautuvat kahteen joukkoon: syötteisiin ja tulosteisiin. Miehellä on täydelliset säännöt syötteiden ja tulosteiden vastaavuuksista. Ne on kirjoitettu miehen ymmärtämällä kielellä. Kun mieheltä kysytään asioita kiinankielisillä syötteillä, kysyjä saa oikeat kiinankieliset vastaukset. Ulkopuolinen saa vaikutelman, että kirjastossa on älykäs toimija, vaikka mies vain sokeasti noudattaa ohjeita ymmärtämättä syötteitä tai tulosteita. Searlen luokittelussa heikko tekoäly on kuin ajatuskokeen mies kirjastossa: se toimii näennäisen älykkäästi. Heikko tekoäly luo illuusion älykkyydestä suorittamalla toimintoja, joihin ulkopuolinen tarkkailija voi luulla vaadittavan älyä. Vahva tekoäly puolestaan ymmärtää, mitä tekee. Esimerkissä vahva tekoäly olisi mies, joka osaa kiinan kieltä. Vahva tekoäly ei ole vain simulaatio mielestä — se on mieli. Vahva tekoäly on nykypäivänä kuitenkin vain tieteiskirjallisuutta, ja niinpä tässä tutkielmassa tarkoitetaan tekoälyllä nimenomaan heikkoa tekoälyä.

Tekoälyn ohjaamia itsenäisiä tietokoneohjelmia kutsutaan agenteiksi [24]. Ne eivät lähdekooditasolla eroa välttämättä lainkaan muista ohjelmista. Siksi agentit usein tunnistetaan ja määritellään toiminnallisten erityispiirteidensä avulla. Agentilla on sensoreita, joilla se havainnoi toimintaympäristöään, ja aktuaattoreita eli toimilaitteita, joilla se vaikuttaa ympäristöönsä. [6] Agentin ympäristö voi olla täysin synteettinen, kuten digitaalisen pelin maailma. Se voi myös toimia tosimaailmassa, esimerkiksi ohjatessaan robottia. Kuitenkaan pelkkä ympäristön havainnoiminen ja siihen vaikuttaminen ei vielä tee ohjelmasta agenttia [24]. Näiden lisäksi agenttia kuvaavat Woolridgen ja Jenningsin [24] mukaan muun muassa seuraavat erityispiirteet:

- Autonomisuus: agentit toimivat ilman muiden väliintuloa, ja ne hallinnoivat

toimintaansa ja sisäistä tilaansa.

- Sosiaaliset taidot: agentit kommunikoivat toisten agenttien ja mahdollisesti myös ihmisten kanssa.
- Reaktiivisuus: agentit reagoivat ympäristönsä tapahtumiin enemmän tai vähemmän reaaliaikaisesti.
- Ennakoivuus: agentit eivät pelkästään reagoi ympäristönsä tapahtumiin, vaan toimivat aloitteellisesti saavuttaakseen päämääriänsä.

Erään hyvän määritelmän agentille antavat Franklin ja Grasser [6]: *"Autonominen agentti on järjestelmä, joka sijaitsee tietyssä ympäristössä, ja on osa sitä. Se havainnoi ympäristöään ja toimii siinä tavoitellen päämääriään. Agentti pyrkii vaikuttamaan siihen, mitä se tietää tulevaisuudesta."*[1] Nämä kaksi luonnehdintaa antavat yhdessä riittävän tarkan, ja toisaalta tarpeeksi laajan rajauksen sille, mitä agentit ovat [24].

Tekoälytutkimus on tähän mennessä tunnistanut kolmenlaisia agentteja: reaktiivisia, suunnittelevia ja näiden piirteitä yhdisteleviä hybridiagentteja [24]. Reaktiiviset agentit ovat toteutustavaltaan agenteista yksinkertaisimpia. Niiden toiminta koostuu ennalta määrätyistä heräte-vaste pareista. Tietty syöte agentin sensoreihin aiheuttaa tietyt aktuaattoreiden toiminnat. Reaktiivisen agentin voi toteuttaa esimerkiksi sääntöpohjaisena järjestelmänä (engl. *rule-based system*) tai äärellistä tilakonetta apuna käyttäen. Seuraava kaava kiteyttää reaktiivisten agenttien logiikan:

$$\textit{ympäristön nykyinen tila} \rightarrow \textit{toiminta}$$

Reaktiivisten agenttien etuna on niiden helppo toteutettavuus sekä laskennallisten resurssien vähäinen kulutus. Niiden haittapuolena on käyttäytymisen kankeus sekä kyvyttömyys oppia tai laatia pitkän aikavälin suunnitelmia.

Suunnittelevat agentit muodostavat ympäristöstään sisäisiä malleja [24]. Ne laativat mallien avulla suunnitelmia saavuttaakseen tiettyjä tavoitetiloja. Seuraava kaava tiivistää suunnittelevan agentin toiminnanohjauksen:

$$\textit{ympäristön nykyinen tila} + \textit{tavoitetila} \rightarrow \textit{suunnitelma}$$

[1] *"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."*

Suunnittelevien agenttien suurin etu on niiden kyky harkita toimiaan etukäteen. Ne ovat kuitenkin melko monimutkaisia eivätkä pysty takaamaan reaaliaikaisuutta. Suunnittelevan agentin voi olla myös vaikea ylläpitää johdonmukaista mallia ympäristöstään, jos sen maailma on kovin nopeatempoinen ja dynaaminen.

Hybridiagentit yhdistelevät reaktiivisten ja suunnittelevien agenttien parhaita puolia [24]. Hybridiagentti voi esimerkiksi käyttäytyä reaktiivisen agentin tavoin aikakriittisissä tehtävissä, kuten törmäystarkistuksessa (engl. *collision detection*), ja suunnittelevan agentin tavoin harkita toimiaan ennalta. Hybridiagentti koostuu ikään kuin kahdesta eri agentista, ja niiden sujuva yhteistoiminta voi olla haastavaa toteuttaa.

2.2 Agentit peleissä

Ennen pelikonsoleita ja kaupallisia pelejä agentteja kehitettiin lähinnä lautapelien digitaalisiin vastineisiin, kuten tammeen, shakkiin ja ristinollaan [16]. Nykyaikaiset pelit tarjoavat agenteille mitä moninaisimpia ympäristöjä, ja tekniikan kehittyessä ympäristöt monipuolistuvat edelleen. Pelin ympäristö voi olla simulaatio tosimaailmasta tai oma fiktiivinen maailmansa. Agentin ohjelmoiminen tosimaailman simulaatioon voi olla haastavampaa kuin sen toteuttaminen fiktiiviseen ympäristöön. Tämä johtuu siitä, että pelaajalla voi olla simulaatiosta enemmän tietoja ja oletuksia kuin agentilla.

Ympäristön lisäksi pelin genre eli tyylilaji vaikuttaa agentin rooliin ja tekniikoihin [16]. Eri genrejä ovat muun muassa: autopelit, jumalpelit, kamppailupelit, roolipelit, seikkailupelit, strategiapelit, toimintapelit ja urheilupelit. Monet pelit yhdistelevät eri genrejä, ja jotkin niistä eivät sovi mihinkään yksittäiseen genreen. Toiminta-, urheilu- ja autopelit vaativat reaaliaikaista reagointia, havainnointia ja päätöksentekoa. Reaaliaikainen strategiapelissa agentin on mukauduttava ihmispelaajan toimintaan. Roolipeleissä agentit esittävät fantasiamaailman hahmoja, ja niiden tulee reagoida maailman tapahtumiin ja muistaa pelaajan teot [24].

Agentit toimivat peleissä erilaisissa rooleissa. Rooleja ovat esimerkiksi taktinen ja strateginen vastustaja, pelaajan kumppani, sivullinen, yksikkö ja selostaja [16]. Agentit voivat samassa pelissä toimia useassa eri roolissa. Esimerkiksi *NHL 14*-jääkiekkopelissä[2] agentit toimivat taktisina vastustajina, pelaajan joukkuetovereina, sivullisina ja selostajina. Agentin rooli vaikuttaa sen toteutukseen. Esimerkiksi tak-

[2] `http://www.ea.com/fi/nhl-14`

tisten vastustajan ja pelaajien kumppanin vaatimukset poikkeavat toisistaan. Taktinen vastustaja pyrkii tarjoamaan haastavan mittelön, kun taas kumppani yrittää mukautua pelaajan tavoitteisiin [24]. Vihollisen on järjestettävä väijytyksiä, hakeuduttava suojaan ja kutsuttava apujoukkoja. Kumppanin tulee ymmärtää joukkuetyöskentelyä, mallintaa ihmisen tavoitteita ja mukautua tämän pelityyliin.

Livingstone [19] sekä Spronck ja den Teuling [39] esittävät agenttien tehtäviksi:

1. ihmisvastustajan simuloiminen [19]
2. pelimaailman roolin suorittaminen [19]
3. viihteen tarjoaminen [39]
4. pelaajan taitojen harjoittaminen [39]

Nämä eivät ole toisensa poissulkevia. Esimerkiksi roolia esittävän agentin päämääränä on usein samalla tarjota viihdettä. Kun agentti simuloi ihmisvastustajaa, se pelaa samoilla säännöillä kuin ihminen. Agentin on toisaalta pelattava yhtä taitavasti kuin ihminen, mutta toisaalta sillä on oltava samat rajoitukset kuin pelaajalla [16]. Ihmisvastustajaa simuloiviin agentteihin palataan luvussa 2.5. Livingstone tarkoittaa pelimaailman roolilla jotakin agentille asetettua tehtävää, toimea, ammattia tai muuta päämäärää. Esimerkiksi toimintapeleissä agentit esittävät sotilaan roolia. Ihmisvastustajan simuloinnin ja roolin esittämisen välillä on hienosyinen mutta selvä ero. Esimerkiksi strategiapelin komentajaa voi ohjata sekä agentti että ihminen. Komentaja vastaa strategisen tason päätöksistä, kuten tuotannosta, resurssien keräämisestä ja hyökkäyssuunnitelmista. Ihmisen taas ei voi kuvitella ohjaavan jokaista pelin yksikköä, vaan ne ovat aina tekoälyn vastuulla. Kun agentti toimii komentajana, se simuloi ihmisvastustajaa. Kun se taas ohjaa pelin yksiköitä, se esittää roolia. Toisin sanoen agentti suorittaa roolia silloin, kun se ei ole korvattavissa ihmispelaajalla.

Kun agentti tarjoaa viihdettä, se pelaa periaatteessa eri peliä kuin ihminen. Viihdyttävä agentti tarjoaa haastetta, mutta pelaa lopulta hävitäkseen [18]. Westin [50] mukaan sopiva haaste riippuu peligenrestä. Kaksinpeleissä pelaaja odottaa agentilta varsin inhimillistä käytöstä. Kaksinpelejä ovat muun muassa shakki, pokeri ja biljardi. Kun pelaaja kohtaa yksin monta vastustajaa esimerkiksi toimintapelissä, hän olettaa niiden olevan kyvyiltään heikompia. Toimintapeleissä haasteen muodostaakin useimmiten vihollisten määrä. Näiden ääripäiden välissä ovat pelit, joissa pelaaja kilpailee useamman tasavertaisen vastustajan kanssa. Tasavertaisten agenttien

ei odoteta olevan yhtä suorituskykyisiä kuin kaksinpeleissä, mutta toisaalta niiden halutaan tarjoavan enemmän haastetta kuin toimintapelin vastustajien. West nostaa esimerkiksi kohtalaisen realistisista agenteista autopelien kanssakilpailijat. Harjoittavan agentin tehtävänä on auttaa pelaajaa kehittymään. Se voi esimerkiksi opettaa pelaajalle perustoimintoja pelin harjoitusjaksossa (engl. *tutorial*). Harjoittavia agentteja esiintyy myös niin sanotuissa hyötypeleissä [39].

2.3 Agentin uskottavuus

Uskottavuudella ei ole tarkkaa, yleisesti hyväksyttyä merkitystä [42]. Sen sisältö riippuu sekä agentin tehtävästä [19], roolista [24] että pelin genrestä. Lisäksi Umarovin ja Mozgovoyn [48] mukaan akatemian ja peliteollisuuden käsitykset uskottavuudesta poikkeavat toisistaan. Tässä luvussa pohditaan agentin uskottavuutta eri näkökulmista. Ensinnäkin se on subjektiivinen ilmiö, joka riippuu niin pelistä kuin pelaajasta [19]. Uskottavuus ei riipu agentin teknisestä toteutuksesta, kunhan agentti ei epäonnistu perustehtävissään [18]. Uskottava agentti pyrkii tarjoamaan sopivaa haastetta ja viihdyttämään pelaajaa [40].

Agentin uskottavuus on subjektiivistä. Agentti on uskottava, jos pelaaja kokee sen uskottavaksi. [19] Uskottavuuskokemuksen syntyminen riippuu niin pelaajan kulttuuritaustasta kuin pelin ja peligenren tuttuudesta [24]. Aloittelelija ei välttämätä erota eritasoisia agentteja toisistaan [19]. Kokenut pelaaja taas voi tuntea agentin niin hyvin, että osaa hyödyntää heikkouksia sen toteutuksessa ja tietää, milloin agentti huijaa. Myös pelaajan tietämys tekoälymenetelmistä vaikuttaa hänen tulkintoihinsa agentin toimista [18]. Lisäksi lukuisat hienovaraiset seikat vaikuttavat agentin uskottavuuteen. Esimerkiksi Mac Nameen [24] mukaan agentin ulkonäkö vaikuttaa siihen, kuinka älykästä käyttäytymistä siltä odotetaan. Pelaajat ovat paljon anteeksiantavampia tyylitellyille kuin realistisille hahmoille.

Pelaaja ei aina pysty arvioimaan agentin uskottavuutta kovin hyvin. Hän ei välttämättä ehdi tarkkailla tekoälyn käyttäytymistä suorittaessaan pelin tehtäviä [15]. Toisaalta hän kohtaa monet agenteista vain hetkittäisesti [42]. Pelaajat eivät aina huomaa agenttien monimutkaisuutta ja toisaalta yliarvioivat joskus niiden teknisen tason [18]. Pelaaja havaitsee agentin toiminnasta vain pienen osan. Hän ei tiedä, mitä agentti tekee, kun se ei ole näköpiirissä, tai mitä informaatiota sen saatavilla on. [15] Toisaalta monissa peleissä agentit eivät tee mitään, jos ne eivät ole pelaajan lähettyvillä, ja aloittavat toimintansa tyhjästä, kun pelaaja saapuu niiden alueelle [24].

Lewis et al. [17] mukaan pelaajan odotukset agentin käyttäymisen suhteen vaihtelevat sekä peligenren, agentin tehtävän että roolin mukaan. Ammuntapeleissä vastustajan oletetaan toimivan jossakin määrin varomattomasti, mutta ei kuitenkaan tyhmänrohkeasti. Pelaajat taas odottavat kumppaniagenteiltaan sopivaa malttia, jotta ne eivät menehtyisi heti alkuunsa. Kuitenkaan liittolaiset eivät saa olla niin varovaisia, ettei niistä ole pelaajalle mitään apua. Tasohyppelypelien agenteilta ei odoteta monimutkaista käytöstä. Ne voivat pelkästään kulkea lyhyttä reittiä edestakaisin. Genre vaikuttaa myös siihen, kuinka merkittäviä agentit ovat pelin kannalta [42]. Joidenkin pelien keskeisenä tavoitteena on agenttien päihittäminen, kun taas toisissa niitä ei ole lainkaan.

Uskottavuudella ei välttämättä ole mitään tekemistä sen kanssa, kuinka edistynyt agentti on teknisesti. Uskottavalle agentille ei ole tärkeintä valita aina tehokkainta vaihtoehtoa, vaan toimia järkevästi. Sen tulee myös reagoida valintojensa seurauksiin siten, että pelaaja voi kuvitella agentin omaavan tavoitteita ja uskomuksia [24]. Ohjelmoijat helposti yliarvioivat tekniikan merkityksen [18]. He kuvittelevat, että monimutkaisempi tekniikka tekee automaattisesti agentin näkyvästä käyttäytymisestä uskottavampaa. Voittamaton tammitekoäly [34] on hyvä esimerkki teknisesti edistyneestä, mutta epäuskottavasta tekoälystä. Digitaalisessa tammessa tekoälyn tehtävä on simuloida ihmisvastustajaa [39], ja ihmisvastustajat eivät ole voittamattomia. Toisaalta uskottavuutta voidaan joskus parantaa hyvin yksinkertaisilla teknisillä keinoilla. Esimerkiksi Lidén [18] kertoo, että *Half-Life*-pelin testeissä havaittiin ongelma agenttien polunetsintäalgoritmissa: ne eivät aina pystyneet pakenemaan kranaatin räjähdystä. Pelin kehittäjät eivät pyrkineet ratkaisemaan vikaa, vaan he animoivat agentit suojautumaan räjähdykseltä.

Agentti ei ole uskottava, jos se epäonnistuu ihmispelaajalle triviaaleissa tehtävissä [18]. Batesin [24] mielestä uskottavuudeksi riittää pelkästään se, että agentti ei ole ilmeisen tyhmä tai epärealistinen. Agentin epäonnistumiset johtuvat usein *bugeista* eli virheistä ohjelmakoodissa. Lewis et al. [17] mukaan tavallisia epäonnistumisia ovat seiniin törmäily tai muihin esteisiin juuttuminen, kulkuväylien tukkiminen ja reagoimattomuus tapahtumiin. Kuitenkin koska agentin toimintahäiriöt ovat poikkeuksia sen odotetussa käyttäytymisessä, ne ovat osin subjektiivisia.

Vaikka agentti toimisi häiriöttä, se ei välttämättä ole uskottava. Käyttäytymisen staattisuus, ennustettavuus ja reagoimattomuus riittävät rikkomaan illuusion älykkäästä toimijasta [19]. Strategiapelissä agentti voi toistaa samankaltaista hyökkäystä, vaikka pelaaja torjuu sen aina. Tämä tekee sen toiminnasta ennalta arvattavaa.

Tyypillinen esimerkki reagoimattomuudesta on agentin välinpitämätön suhtautuminen ympäristönsä tapahtumiin. Esimerkiksi *Fallout Tactics*-pelissä vartija-agentit jatkavat partiontia, vaikka kävelisivät surmatun toverinsa ylitse. Toisaalta MacNameen [24] mukaan harvassa pelissä agentit muuttavat käytöstään pysyvämmin. Esimerkiksi monien roolipelien hahmot reagoivat pelaajaan samalla tavalla riippumatta siitä, miten tämä on niitä kohtaan aikaisemmin käyttäytynyt.

Agentin uskottavuus kärsii, jos pelaaja huomaa sen huijaavan [24]. Tässä huijaamisella tarkoitetaan sitä, että agenttilla on kykyjä, etuja tai tietoja, joita pelaajalla ei. Kuitenkin huijaaminen haittaa vain, jos se on kovin ilmeistä tai epäreilua [17]. MacNamee [24] huomauttaa, että pelaajat helposti uskovat huijaavan agentin yksinkertaisesti pelaavan heitä paremmin. Lisäksi huijaaminen ei haittaa uskottavuutta, jos pelaaja odottaa pelin olevan epäreilu. Esimerkiksi *Civilization IV*-strategiapelissä agentit saavat vaikeimmilla tasoilla käyttöönsä enemmän pelin resursseja kuin pelaaja. Säätäessään vaikeustasoa haastavammaksi pelaaja tietää, että agentit tulevat huijaamaan — ne pelaavat siis täysin odotustenmukaisesti.

Pelin tärkein ominaisuus on sen viihdyttävyys [18]. Haaste on merkittävässä osassa viihdyttävän pelikokemuksen syntymisessä [40]. Haastava peli asettaa pelaajalle tavoitteita, joiden saavutaminen on epävarmaa [52]. Pelin agentin tulee osana muuta peliä tarjota sopivaa haastetta ja viihdyttää pelaajaa. Agentin uskottavuutta voidaan arvioida myös sen mukaan, miten se onnistuu näissä tehtävissä [42]. Ei kuitenkaan osata tarkkaan sanoa, minkälainen agentin käyttäytyminen viihdyttää pelaajaa [52]. Agentin haastavuus on kytköksissä sen suorituskyvyn kanssa. Mahdollisimman suorituskykyisen agentin ohjelmoiminen ei silti takaa viihdyttävää peliä, sillä pelaajat reagoivat haasteeseen eri tavoin. [43] Viihdyttävyyttä on mitattu niin kvalitatiivisesti kuin kvantitatiivisesti. Esimerkki edellisestä on Sweetserin ja Wyethin [40] *GameFlow*-menetelmä, joka tutkii, missä määrin pelaaja saavuttaa niin sanottuja virtauskokemuksia. Sweetserin ja Wyethin mukaan virtauskokemus syntyy silloin, kun viihde ja haaste yhdistyvät optimaalisesti. Kvantitatiivisesti viihdyttävyyttä on tutkittu esimerkiksi mittaamalla pelaajien fysiologisia vasteita, kuten pulssia ja hengitystiheyttä [43].

2.4 Tekotyperyys

Tässä luvussa kerrotaan tekotyperyydestä, eli tarkoituksellisten virheiden ja puutteiden lisäämisestä agentin toteutukseen. Lisäksi pohditaan, miten tekotyperyys ja

uskottavuus liittyvät toisiinsa. Luvussa pyritään esimerkein havainnollistamaan, mitä tekotyperyys on, ja miten se voi parantaa agentin uskottavuutta.

Agenteilla on monissa peleissä etulyöntiasema ihmiseen nähden [50]. Esimerkiksi toimintapelin agentti voi tietää, missä pelaajan hahmo on, vaikkei tämä olisikaan näköpiirissä. Tämä voi johtua muun muassa siitä, että agentin pääsyä pelin tietorakenteisiin ei ole rajoitettu. Lidénin [18] mukaan ylivertaisen agentin ohjelmoiminen on vieläpä suhteellisen helppoa. Hän kertoo, että pelaajat huomaavat ennen pitkää agenttien yli-inhimilliset kyvyt, vaikkeivät tarkkaan pystyisikään kertomaan, mikä niiden toiminnassa on outoa. Soni ja Hingston [38] esittävätkin, että agentin tarkoituksena pelissä ei ole päihittää pelaajaa, vaan tarjota tälle viihdettä. He kertovat: *"– videopelien tehtävänä on tarjota pelaajalle mielenkiintoisia, ei optimaalisia vastustajia."*[3] Niinpä agentin toimintaa tulee usein jollakin tavalla rajoittaa.

Yksinkertaisin tapa rajoittaa agenttia on alentaa sen algoritmien tehokkuutta tai rajoittaa niiden laskenta-aikaa. Kuitenkin tämä lähestymistapa tekee sen käyttäytymisestä helposti epärealistista. Agentti alkaa tehdä virheitä, joita ihmispelaaja ei koskaan tekisi. Tällöin illuusio haastavasta vastustajasta särkyy. Pelaajan tulisi kokea voittaneensa agentin omasta ansiostaan, ja että vastustaja todella kamppaili voitosta. [50]

Parempi tapa rajoittaa agenttia on tarkoituksellisesti ohjelmoida sen toimintaan virheitä ja puutteita [18]. Tällä tavalla rajoitettuja agentteja kutsutaan *tekotyperiksi* (engl. *artificial stupidity*). Tekotyperyys ei terminä kuitenkaan ole vakiintunut, vaan sitä käytetään eri merkityksissä. Esimerkiksi Lewis et al. [17] kutsuvat tekotyperäksi toimintahäiriöistä kärsivää tekoälyä. Tekotyperien agenttien virheistä käytetään englanniksi termejä *intentional* ja *intelligent mistake*. Tekotyperyyden tarkoituksena on parantaa agentin uskottavuutta ja viihdyttävyyttä, sekä tehdä sen tarjoamasta haasteesta sopivaa.

Tekotyperyys ilmenee eri tavalla eri peligenreissä. Esimerkiksi shakkitekoäly *Fritz* luo pelissä tilanteita, joita kekseliäs pelaaja voi hyväksikäyttää [50]. Se ei kuitenkaan tee ilmeisen huonoja siirtoja. Jos pelaaja keksii hyödyntää *Fritzin* tarjoaman mahdollisuuden, se alkaa pelata jälleen voittaakseen. West [50] ehdottaa, että biljardipelissä agentin tulee joskus päättää lyövänsä huonosti. Huono lyönti on sellainen, joka ei pussita palloa, tai josta lyöjän on vaikea jatkaa. Tällainen virhe on illuusio, sillä Westin mukaan agentti voi laskea täydellisen lyönnin mistä tilanteesta tahansa. Kun West ohjelmoi biljardiagentin, hänelle selvisi, että tekotyperyydeksi ei pelkästään

[3] *"– in video games, the aim is to provide interesting opponents for human players, not optimal ones."*

riitä, jos ohjelmoi agentin lyömään joskus ohi. Hänen asiakkaidensa mielestä tällainen agentti pelasi liian taitavasti. He kokivat, agentti jätti valkoisen pallon itselleen liian hyvään asemaan. Tosiasiallisesti se ei suunnitellut lainkaan, mihin valkoinen pallo jää.

Lidén [18] esittelee joukon toimintapeleihin sopivia tekotyperiä menetelmiä, joiden tarkoituksena on pitää haaste sopivana. Agentit voivat esimerkiksi pitää ääntä pelaajan lähestyessä, jotteivät ne yllättä tätä. Ne voivat vetäytyä, jos pelaaja on pahasti alakynnessä. Agentit voivat ilmaista aikeensa pelaajalle antamalla toisilleen näennäisesti käskyjä. Lidén myös kehitti *Half Life*-peliin järjestelmän, joka huolehtii, että agenttijoukosta vain kaksi kerrallaan saa luvan tulittaa. Sillä aikaa, kun muut odottavat vuoroansa, ne lataavat aseitansa, siirtyvät suojaan tai pysyvät muuten toimeliaana. Lidénin mukaan pelaajat kokivat ylivoimatilanteen edelleen haastavaksi, vaikkeivät agentit täysin hyödyntäneetkään etulyöntiasemaansa.

West [50] suunnitteli pokeripeliin agentteja, jotka tekevät samantyyppisiä virheitä kuin ihmispelaajat. Ne määrittävät ensin, mikä seuraavista mahdollisista toimista on paras. Sitten ne jättävät tämän parhaan vaihtoehdon suorittamatta tietyllä todennäköisyydellä. Todennäköisyys riippuu valitusta vaikeustasosta. Pelin agentit tekevät melko hienovaraisia, mutta merkittäviä virheitä. Ne voivat esimerkiksi luovuttaa silloin, kun pelaaja korottaa panosta, vaikka niillä olisi kortit, jotka voittavat suurella todennäköisyydellä. Toisaalta ne voivat jatkaa peliä, vaikka niillä olisi huono käsi. Westin mukaan nämä ovat tyypillisiä kokemattoman ihmispelaajan virheitä. Hänen mielestään edellä kuvatun kaltainen tekotyperyys sopii pokeriin hyvin, koska pelin satunnaisen luonteen vuoksi pelaaja ei voi varma, milloin agentti tekee virheitä. Pelaaja voittaa useammin kuin todennäköisyyksien valossa pitäisi, mutta hän ei huomaa sen johtuvan siitä, että agentti pelaa huonommin kuin osaisi.

2.5 Bottien uskottavuus ja tekotyperyys

Tässä tutkielmassa *botilla* tarkoitetaan ihmisvastustajaa simuloivaa agenttia. Jos peli sisältää erilaisia hahmoja, niin botilla tarkoitetaan sellaista agenttia, jonka ohjaamaa hahmoa voisi pelata myös ihminen [19]. Botit ovat tämän teoksen varsinainen tutkimuskohde, sillä pyrin ohjelmoimaan TORCS-peliin juuri botin. Luvussa tarkastellaan, mitä ihmisvastustajan simulointi peleissä käsittää. Lisäksi pohditaan, mitä uskottavuus ja tekotyperyys tarkoittavat botin kohdalla. Esimerkkeinä käytetään pääosin toimintapelien botteja, koska toimintapelit ovat suosituin genre myynnillä

mitattuna[4]. Tässä toimintapeleiksi lasketaan myös ammuskelupelit (engl. *first person shooter*).

Botin voidaan ajatella yrittävän läpäistä pelin kontekstiin sovitettua Turingin testiä (ks. luku 2.1). Se yrittää vakuuttaa pelaajan siitä, että hän pelaa toista ihmistä vastaan, tai että botin hallitsemaa hahmoa ohjaa toinen ihminen. [19] Ihmisvastustajan simulointiin liittyy sekä tämän kykyjen että rajoitteiden simuloiminen [16]. Lisäksi on huomattava, ettei botin välttämättä tarvitse simuloida parasta ihmisvastustajaa. Shakkibotin ei täydy aina pelata suurmestarin tasolla. [48] Pelattava peli vaikuttaa siihen, millä kriteereillä botin inhimillisyyttä arvioidaan [52]. Esimerkiksi *Pongissa* botin tulee vain liikutella mailaa edestakaisin [19], kun taas toimintapelissä sen tulee suorittaa monia toimia samanaikaisesti, kuten juosta, tulittaa ja suojautua [42]. Arviointikriteereihin vaikuttaa myös botin rooli pelissä. Se voi olla esimerkiksi pelaajan vastustaja tai kumppani [16]. Ei välttämättä ole kuitenkaan mielekästä tuomita bottia täysin epäuskottavaksi, jos se ei läpäise pelin Turingin testiä, sillä joitakin inhimillisiä piirteitä on verrattain helpompi simuloida kuin toisia. On esimerkiksi yksinkertaista hidastaa liian nopeasti tulittavaa bottia, kun taas ihmisen kaltainen mukautumiskyky on vielä täysin tekniikan saavuttamattomissa [12]. Livingstone [19] esittääkin, että Turingin testi voidaan jakaa osatesteihin, jotka testaavat botin toteutuksen eri puolia.

Inhimillisten piirteiden ohjelmoimiseen käytetään eri tekniikoita. Botti voidaan saada käyttäytymään näennäisen inhimillisesti yksinkertaisilla *ad hoc* ratkaisuilla. *Ad hoc* tarkoittaa tässä tiettyä tarkoitusta varten räätälöityä ratkaisua. Toisaalta botin toteutuksessa voidaan käyttää edistyneitä menetelmiä, kuten kognitiivisiä järjestelmiä, jotka perustuvat psykologian käsityksiin mielestä. Eräs mahdollisuus on opettaa botti toimimaan ihmisten pelaamisesta kerättyjen esimerkkien avulla. [48] Näitä edistyneempiä menetelmiä käytetään kuitenkin pääasiassa akateemisessa tutkimuksessa [24]. Esimerkkinä kaupallisista peleistä *Unreal Tournament 2004*:än botit käyttävät sumeita tilakoneita [38]. Tilakone koostuu joukosta tiloja, sekä suunnatuista siirtymistä niiden välillä. Sumean tilakoneen siirtymät noudattavat sumeaa logiikkaa. Sumea logiikka on tavanomaisen logiikan laajennos, joka sallii *toden* ja *epätoden* lisäksi osittaiset totuusarvot. [24] *Unreal*-botti valitsee tilakoneen avulla käyttämänsä skriptin. Skriptit vastaavat jotakin korkean tason tavoitetta tai käyttäytymistä. Näitä ovat esimerkiksi vaeltelu, jäljittäminen ja rynnäköiminen. Vaeltava botti kerää ympäristöön siroteltuja varusteita, jäljittävä koittaa löytää näköpiiris-

[4] `www.theesa.com/facts/pdfs/esa_ef_2013.pdf`

tään kadonneen vastustajan ja rynnäköivä hyökkää vastustajaa kohti samalla tulittaen. [38]

Botti on sitä uskottavampi, mitä paremmin se simuloi ihmispelaajaa [48]. Taatgenin et al. [48] mukaan: *"Tietokonevastustaja tulee sitä mielenkiintoisemmaksi ja viihdyttävämmäksi, mitä enemmän se käyttäytyy kuin ihminen."* [5] Paremmin ihmistä simuloiva botti on myös haastavampi [16]. Sonin ja Hinstonin [38] mukaan erityisesti käyttäytymisen ennustamattomuus tekee botista haastavan. Umarov ja Mozgovoy [48] pitävät uskottavan botin piirteinä oppimista, virheiden tekemistä sekä kykyä muuttaa strategiaa pelaajan toimien perusteella. Jones et al. [48] mielestä botilla tulee olla samat tiedot sekä samankaltaiset sensorit ja motoriikka kuin pelaajalla. Lisäksi sen tulee käsitellä dataa samalla tavalla. Se, miten nämä piirteet toteutuvat riippuu pelistä [48]. Kuten muidenkin agenttien, myös bottien uskottavuus on subjektiivista [19]. Pelaajan kokemuksiin botista vaikuttaa esimerkiksi se, tietääkö hän pelaavansa konetta vastaan vai ei. Pelaajat nauttivat pelistä enemmän, jos he uskovat pelaavansa toisia ihmisiä vastaan [42].

Botin kohdalla tekotyperyytenä voidaan pitää inhimillisten heikkouksien ja rajoitteiden simuloimista. Kuten edellä todettiin, virheiden tekeminen on yksi uskottavan botin piirteistä [48]. Esimerkiksi Laird ja Duchi [15] ohjelmoivat *Quake II*-toimintapeliin botteja, joiden kykyjä he pyrkivät rajoittamaan. Ne eivät esimerkiksi näe kappaleita esteiden takana. Lisäksi niillä on rajallinen näkökenttä. Ne eivät myöskään havaitse ääniä kovin kaukaa. Botit eivät tunne pelin ympäristöä valmiiksi, vaan joutuvat opettelemaan sen muodostamalla sisäisiä malleja. Laird ja Duchi eivät erittele, miten nämä rajoitukset vaikuttivat pelaajien kokemuksiin boteista. He kuitenkin huomasivat, että pelaajat pitivät bottia inhimillisempänä, jos sen osumatarkkuutta alennettiin parhaasta mahdollisesta. Paras botti laski aina tarvittavan ennakon, ja osui hyvin tarkasti. Umarov ja Mozgovoy [48] huomauttavatkin, että botilla ei saa olla yli-inhimillisiä kykyjä. Näiden karsimista toteutuksesta voidaan myös pitää tekotyperyytenä [18].

Seuraavaksi käsitellään esimerkin vuoksi taktisen vastustajan roolissa toimivia toimintapelien botteja. Ensin luetellaan taktisten vastustajien vaatimuksia yleensä. Sitten listataan bottien piirteitä, joita pelaajat pitivät inhimillisinä tai epäinhimillisinä. Toisin sanoen sellaisia, jotka joko paransivat tai heikensivät botin uskottavuutta. Lairdin ja van Lentin [16] mukaan toimintapelin botin tulee muun muassa kyetä na-

[5] *'a computer opponent becomes more interesting and enjoyable to play against as it behaves more like a person"'*

vigoimaan, mukautua erilaisiin pelaajiin ja heidän toimintaansa sekä vuorovaikuttaa ympäristönsä ja toisten bottien kanssa. Lisäksi sillä on oltava inhimilliset reaktioajat, ja sen on liikuttava realistisesti. Laird ja van Lent jatkavat: jotta botti navigoisi ja vuorovaikuttaisi ympäristönsä kanssa realistisesti, sillä tulee olla ihmisen aistien kaltaiset sensorit. Se ei saa nähdä pelaajaa pimeässä eikä esteiden läpi. Laird ja van Lent esittävät, että botti voisi imitoida esimerkiksi turhautumisen tai suuttumisen tunteita muuttamalla ”tunnekuohun” aikana käyttäytymistään. Alla on listattu pelaajien inhimilliseksi kokemia piirteitä Lairdin ja Duchin [15] *Quake II*-pelin botista:

- Inhimilliset reaktio- ja päätöksentekoajat. Tässä tapauksessa 0,05 – 0,1 sekuntia.
- Sopivan tarkka tähtäys.
- Taktinen ja strateginen päättelykyky.

Laird ja Duchin tutkimuksessa pelaajat kokivat botin toiminnan sitä inhimillisemmäksi, mitä monimutkaisempia taktiikoita se käytti. Heidän mielestään taktiikat eivät kuitenkaan tehneet botista taitavampaa. Hingston [11, 12] puolestaan järjesti kilpailuja *Unreal Tournament 2004*-pelin boteille. Kilpailujen tuomarit pitivät niiden käyttäytymisessä muun muassa seuraavia piirteitä epäinhimillisenä:

- Matala aggressio [11].
- Suorissa linjoissa liikkuminen [11].
- Huonot taktiikat, kuten väärien aseiden käyttäminen [11].
- Suunnittelun puute [12].
- Epäjohdonmukaisuus, esimerkiksi vihollisten unohtaminen [12].
- Paikoilleen juuttuminen [12].
- Liian tarkka tähtääminen [12].
- Puutteet tarkkaavaisuudessa [12].
- Jääräpäisyys [12].

Hingston ei erittele, mitä esimerkiksi jääräpäisyys tai suunnittelun puute tarkoittavat tässä yhteydessä.

3 Kilpa-ajamisen fysiikka ja auton tekniikka

Tässä luvussa käsitellään yleisluontoisesti ajamisen fysiikkaa ja auton tekniikkaa. Sen tarkoituksena ei ole esitellä mahdollisimman todenmukaisia malleja tai tarkkoja laskuja, vaan antaa lukijalle aiheista riittävä käsitys myöhempiä lukuja varten. Tässä luvussa selvitetään esimerkiksi, miksi ja miten moottorin tehokkuus, auton paino tai ajolinjan valinta vaikuttavat kierrosaikoihin. Luku pyrkii auttamaan lukijaa ymmärtämään, mistä ajosimulaation matemaattiset mallit juontuvat. Sen pääasialliset lähteet ovat Beckmanin teos [1] ja Parkerin kirja [31].

3.1 Kiihdyttäminen ja jarruttaminen

Paikaltaan lähtevän auton kulkema matka d metreinä ajan t suhteen saadaan kaavalla:

$$d = \frac{1}{2}at^2,$$

missä a on ajoneuvon kiihtyvyys g-voimissa ja t kiihdytykseen kulunut aika sekunteina. Kiihtyvyys a oletetaan tässä tasaiseksi. Suorituskykyinen urheiluauto voi kiihtyä $0{,}5\,g$ voimalla, jolloin $a = 0{,}5\,g = 4{,}90\,\mathrm{m/s^2}$. Auto kulkee tällä kiihtyvyydellä ensimmäisen sekunnin aikana 2,45 m ja kahdessa sekunnissa 9,81 m. Taulukossa 3.1 on esimerkkejä eri kiihtyvyyksistä.

Taulukko 3.1: Ajoneuvon lähdöstä kulkema matka metreinä eri kiihtyvyyksillä.

Kiihtyvyys	Aika s		
g	2,50	5,00	7,50
0,20	6,13	24,53	55,18
0,35	10,73	42,92	96,57
0,50	15,32	61,31	137,95

Tosimaailmassa auton kiihtyvyys ei pysy tasaisena, eikä auto kiihdy ikuisesti. Seuraavassa tarkastelussa on poikkeuksellisesti lainattu Beckmanin [1, osa 6] laskuja suoraan. Ainoastaan mittayksiköt on muunnettu kansainväliseen yksikköjärjestelmään, ja tulokset pyöristetty. Olkoon auton huippunopeus 240 km/h ja sen moot-

torin teho 179 kW. Tällainen auto kiihtyy pysähdyksistä 100 km/h nopeuteen 6 sekunnissa, 160 km/h nopeuteen 15 sekunnissa ja huippunopeuteensa noin 60 sekunnissa. Suurin kiihtymistä hidastava tekijä on ilmanvastus. Toiseksi merkittävin on renkaiden ja tienpinnan välinen kitka eli vierintävastus. Lisäksi lukuisat vähäisemmät tekijät, kuten auton eri osien välinen kitka hidastavat kiihtymistä. Ilmanvastus on monimutkainen ilmiö, mutta yksinkertainenkin malli antaa kuvan sen merkittävyydestä. Beckman käyttää ilmanvastuksen laskemiseen Landaun ja Lifshitzin [1] likimääräistä yhtälöä:

$$F = \frac{1}{2} C_v A \rho v^2$$

missä F on vastusvoima, C_v muotovastus, A ajoneuvon pinta-ala ajosuuntaan nähden, ρ ilman tiheys ja v auton nopeus. Lyhyesti kerrottuna muotovastus C_v kuvaa sitä, kuinka esteettömästi ilma pääsee kulkemaan auton muotoja pitkin, ja minkälaisen ilmavirran auto jättää jälkeensä [31]. Pinta-alaa A voidaan ajatella auton profiilina suoraan edestäpäin katsottuna. Esimerkissä $A = 1{,}86\,\mathrm{m}^2$, $C_v = 0{,}30$ ja $\rho = 1{,}284\,\mathrm{kg/m}^3$.

Edellä esitetystä ilmanvastuksen kaavasta ilmenee, että se kasvaa auton nopeuden neliössä. Niinpä kun 50 km/h nopeudessa ilmanvastus työntää autoa 7 kg voimalla, vastuksen voima on 100 km/h nopeudessa 28 kg. 240 km/h vauhdissa esimerkin auto kohtaa 162 kg suuruisen voiman. Beckmanin mukaan noin 108 kW teho riittää tämänsuuruisen ilmanvastuksen kumoamiseen. Auton tehosta 71 kW kuluu siis vierintävastuksen ynnä muiden hidastavien seikkojen voittamiseen.

Se, kuinka nopeasti auto voi hidastaa vauhtiaan riippuu sen kineettisestä energiasta eli liike-energiasta. Kineettinen energia saadaan kaavalla:

$$E = \frac{1}{2} m v^2$$

missä E on energia, m kappaleen massa newtoneina ja v sen nopeus metreinä sekunnissa. Liike-energia kasvaa nopeuden neliössä, eli pysähtyminen esimerkiksi 150 km/h nopeudesta kestää neljä kertaa niin kauan kuin 75 km/h nopeudesta. Myös renkaiden pito vaikuttaa siihen, kuinka nopeasti auto voi pysähtyä. Jos ajoneuvo jarruttaa liian voimakkaasti, sen renkaat menettävät pitonsa. Tällöin puhutaan lukkojarrutuksesta: renkaat eivät pyöri, vaan luisuvat tienpinnalla.

Jarrutusta voidaan tarkastella negatiivisena kiihtymisenä. Beckmanin esimerkissä rengas pitää korkeintaan $-1{,}0\,g$ kiihtyvyydessä. Tämä tarkoittaa, että ajoneuvon vauhti ei voi hidastua enempää kuin $9{,}81\,\mathrm{m/s} = 35{,}31\,\mathrm{km/h}$ sekunnissa. Pysähty-

miseen kuluva aika saadaan kaavalla:

$$t = v/a$$

missä t on aika sekunteina, v nopeus metriä sekunnissa ja a negatiivinen kiihtyvyys g-voimissa. Pysähtymismatka lasketaan puolestaan:

$$d = vt - \frac{1}{2}at^2$$

missä d on matka metreinä, v nopeus metriä sekunnissa, t aika sekunteina ja a kiihtyvyys g-voimissa. Huomattavaa on, että jarrutusmatka kasvaa nopeuden neliössä, kun taas jarrutusaika kasvaa nopeuden suhteen lineaarisesti. Molemmissa kaavoissa oletetaan, että negatiivinen kiihtyvyys on tasaista. Jos halutaan laskea aika tai matka, joka kuluu kun jarrutetaan tietystä alkunopeudesta v_1 nopeuteen v_2, voidaan edellisten kaavojen termi v korvata nopeuksien erotuksella $\hat{v} = v_1 - v_2$. Taulukossa 3.2 on esitetty pysähtymiseen kuluva aika ja matka eri nopeuksilla, kun kiihtyvyys on $-1{,}0\,g$.

Taulukko 3.2: Pysähtymiseen kuluva aika ja matka $-1{,}0\,g$ kiihtyvyydellä.

Nopeus km/h	Aika s	Matka m
25	0,71	2,46
50	1,42	9,83
75	2,12	22,12
100	2,83	39,33
125	3,54	61,45

3.2 Auton painojakauma

Auton painojakauman hallitseminen on tärkeä osa ajamista niin kilparadalla kuin liikenteessä. Kuljettajan toimenpiteet, kuten kiihdyttäminen, jarruttaminen ja ohjaaminen vaikuttavat auton painon jakautumiseen sen eri pyörille. Auton kiihdyttäessä paino siirtyy takapyörille, kun taas jarruttaessa etupyörille. Auton kääntyessä paino siirtyy sisäkaarteen pyöriltä ulkokaarteen pyörille.

Pohjimmiltaan auton painon siirtymisessä on kyse inertiasta eli massan hitaudesta. Hitaus kohdistuu auton painopisteeseen. Painopiste on näennäinen tasapainokohta, jonka eri puolilla lepää yhtä suuri osuus auton painosta. Painoa siirtyy

renkaalta toiselle auton kiihdyttäessä, jarruttaessa ja kääntyessä sitä enemmän, mitä korkeammalla sen painopiste sijaitsee. Tästä syystä matala auto on usein helpommin hallittavissa kuin korkea. Newtonin liikelakien [31] avulla painon jakautumista voidaan tutkia tarkemmin. Ne ovat:

Mekaniikan I peruslaki eli jatkuvuuden laki

Suoraa linjaa tasaisella nopeudella kulkeva kappale jatkaa liikettänsä, jos siihen ei vaikuta jokin ulkoinen voima. Massan hitaus johtuu liikkeen jatkuvuudesta.

Mekaniikan II peruslaki eli dynamiikan peruslaki

Kappaleen kiihtyvyyteen vaikuttavat sen massa ja siihen kohdistuva voima. Lain kiteyttää yhtälö:

$$F = ma \Rightarrow a = F/m, \tag{3.1}$$

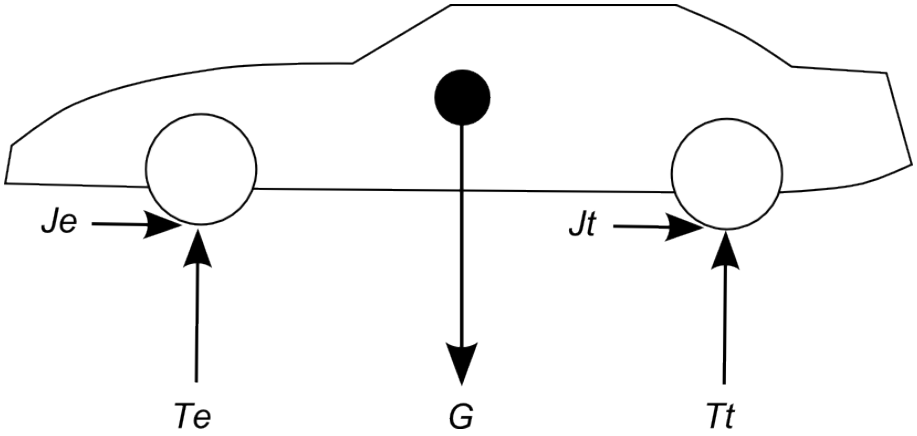
missä F on kokonaisvoima, m massa ja a kiihtyvyys. Kiihtyvyys voi olla myös negatiivista. Suurempi voima aiheuttaa nopeampia muutoksia auton liikkeessä. Painava auto taas kiihtyy hitaasti. Laki selittää, miksi kilpa-autot ovat usein kevyitä, ja niiden moottorit tehokkaita.

Mekaniikan III peruslaki eli voiman ja vastavoiman laki

Kun kappaleeseen kohdistuu jokin voima, siitä aiheutuu samansuuruinen vastakkainen voima. Esimerkiksi kun auto jarruttaa, sen renkaat työntävät maanpintaa vasten tietyllä voimalla, ja maanpinta työntää vastaavalla voimalla takaisin.

Jarruttaminen, kiihdyttäminen ja kääntyminen muistuttavat fysikaalisina ilmiöinä pitkälti toisiaan, joten riittää tarkastella niistä vain yhtä, jotta selviää, miksi ja miten kiihtyvyys vaikuttaa auton painon jakautumiseen. Seuraavassa käsittelyssä ei yksinkertaisuuden vuoksi huomioida esimerkiksi jousituksen vaikutusta painon siirtymiseen.

Kuva 3.1 havainnollistaa voimat, jotka kohdistuvat autoon $-1{,}0\,g$ kiihtyvyyden eli jarrutuksen aikana. Kun kiihtyvyys on $-1{,}0\,g$, jarruttava voima on Newtonin II lain perusteella yhtä suuri kuin auton massa, eli $-1{,}0 = -F/m \Rightarrow F = m$. Kuvassa G tarkoittaa auton painopisteeseen kohdistuvaa vetovoimaa, joka antaa auton massalle painon. Te ja Tt osoittavat maanpinnasta etu- ja takarenkaisiin kohdistuvia työntäviä voimia. Je ja Jr merkitsevät eteen ja taakse kohdistuvia jarruttavia voimia. Kuvasta on selkeyden vuoksi jätetty pois vastavoimat.



Kuva 3.1: Autoon kohdistuvat voimat sen jarruttaessa. Kuva mukailtu Beckmanin teoksesta [1].

Oletetaan auton painon jakautuvan lepotilassa tasaisesti etu- ja takapyörille. Lepotila tarkoittaa, että autoon ei kohdistu kiihtyvyyksiä, tai siihen kohdistuvat kiihtyvyydet kumoavat toisensa. Tosimaailmassa eri automallien painojakauma vaihtelee suuresti esimerkiksi moottorin sijoittumisen mukaan. Kun paino on tasaisesti jakautunut, työntävät voimat Te ja Tt ovat yhtä suuria (ks. kuva 3.1). Kumpikin kantaa puolet ajoneuvon painosta. Kun auto jarruttaa, eturenkaisiin kohdistuva voima Te kasvaa, ja takarenkaisiin kohdistuva Tt vähenee. Tällöin auton takaosa ikäänkuin kevenee. Jarruttavat voimat Je ja Jt työntävät renkaita taaksepäin ja lopulta vanteiden, jousituksen ynnä muiden osien kautta vaikuttavat koko auton liikkeeseen.

Jarruttavat voimat vaikuttavat maan tasalla, kun taas auto kokonaisuutena pyrkii inertian vuoksi jatkamaan liikettänsä painopisteen tasalla. Tämä saa ajoneuvon kiertymään oman vaaka-akselinsa ympäri. Autoon kohdistuu vääntövoima sen painopisteen suhteen. Jarruttaessa myös maanpinnasta eturenkaisiin kohdistuva voima Te kasvaa. Se vastustaa vääntövoimaa. Jarrutuksen vääntövoima riippuu jarruttavien voimien Je ja Jt suuruudesta sekä auton painopisteen korkeudesta. Vääntövoiman lisäksi voiman Te tulee vastustaa takapyöriä työntävää voimaa Tt, sillä myös se saa auton kallistumaan vaaka-akselinsa ympäri. Yleiset kaavat siirtyvän painon laskemiselle ovat:

$$Te = pG + Fh/a$$
$$Tt = (1-p)G - Fh/a,$$

missä p on etupyörille jakautuvan painon osa lepotilassa, F autoon kohdistuva kiihtyvyys g-voimissa, G auton paino kilogrammoissa ja h painopisteen korkeus sekä a

akseliväli senttimetreinä. Etupyörien osuudelle pätee $0{,}0 \leq p \leq 1{,}0$. Olkoon painopiste esimerkiksi 50 cm korkeudella, akseliväli 250 cm, paino 1600 kg ja kiihtyvyys $-1{,}0\,g$. Tällöin

$$\begin{aligned} Te + Tt &= 1600\,\text{kg} \\ Te \cdot 125\,\text{cm} - Tt \cdot 125\,\text{cm} &= 1600\,\text{kg} \cdot 50\,\text{cm} \end{aligned}$$

josta voidaan laskea

$$\begin{aligned} Te &= \frac{1600\,\text{kg} \cdot 50\,\text{cm}}{125\,\text{cm}} + Tt \\ Te &= \frac{1600\,\text{kg} \cdot 50\,\text{cm}}{125\,\text{cm}} + 1600\,\text{kg} - Te \\ 2Te &= 1600\,\text{kg} \cdot 0{,}4 + 1600\,\text{kg} \\ Te &= 1600\,\text{kg} \cdot 0{,}2 + 800\,\text{kg} = 1120\,\text{kg} \end{aligned}$$

Painoa siis siirtyy jarrutuksen aikana etupyörille 320 kg eli 20 % auton kokonaispainosta.

3.3 Renkaiden pito

Auton rengas menettää pitonsa ja alkaa luisua, jos siihen kohdistuu liian suuri voima, tai liikaa painoa siirtyy renkaalta pois. Samalla tavalla luisuvan renkaan voi saada taas pitäväksi, jos sille siirtyy painoa, tai autoon kohdistuvat voimat heikkenevät. Taitava kilpa-ajaja osaa arvioida autonsa painojakauman kullakin hetkellä, ja aavistaa miten kiihdyttäminen, jarruttaminen ja kääntäminen vaikuttavat sen pitoon.

Renkaan pitoa tarkastellaan sen kosketuspinnalla. Kosketuspinta on se renkaan osa, joka kullakin hetkellä on kiinni tienpinnassa. Kosketuspinnalla renkaaseen vaikuttaa sivuttaissuuntaisia voimia auton kääntyessä ja pitkittäissuuntaisia kiihdyttäessä ja jarruttaessa. Sen kokoon vaikuttaa renkaan halkaisija ja leveys. Kosketuspinta on yleensä soikionmallinen, ja se pysyy ajon aikana jotakuinkin samanmuotoisena. Beckmanin mukaan kosketuspinta on riittävän yksinkertaisen ja tarkka malli pidon tarkasteluun, eikä se poikkea merkittävästi todellisuudesta.

Renkaan luisuttamiseen tarvittava voima kasvaa suhteessa renkaan kannattelemaan painoon. Suhteen ilmaisee yhtälö:

$$F \leq \mu W,$$

missä F on voima, jolla rengas vastustaa luisumista, μ kitkakerroin ja W renkaan kosketuspinnalla lepäävä paino. Tässä kitkakerroin μ oletetaan vakioksi. Tosiasiallisti siihen vaikuttavat ajon aikana sekä tienpinnan että renkaan ominaisuudet. Renkaat myös kuluvat kilpa-ajossa merkittävästi, mikä alentaa niiden kitkaa. Kaikkien kitkaan vaikuttavien tekijöiden simulointi ei kuitenkaan ole usein peleissä mahdollista laskennallisista syistä.

Kullakin auton renkaalla lepää osa sen massasta. Kun kahteen saman kokoiseen mutta massaltaan erisuuruiseen autoon kohdistuu $-1{,}0\,g$ kiihtyvyys, etupyörille siirtyy yhtä suuri osuus massasta, mutta siirtynyt paino eroaa kilomääräisesti. Renkaalla lepäävän massan osuuden ja painon suhteen ilmaisee kaava $W = f(a)mg$, missä $f(a)$ on tämänhetkinen pyörän kannattelema osuus auton massasta, m auton massa ja g painovoiman autoon kohdistama kiihtyvyys. Kaavassa käytetään funktiota $f(a)$, koska renkaan kannattelema osuus massasta muuttuu jatkuvasti kiihtyvyyden a mukaan.

Kun renkaaseen kohdistuu voima F, saadaan kosketuspintaan vaikuttava kiihtyvyys a Newtonin II peruslain mukaisesti $a = F/f(a)m$. Yhdistämällä kaavat

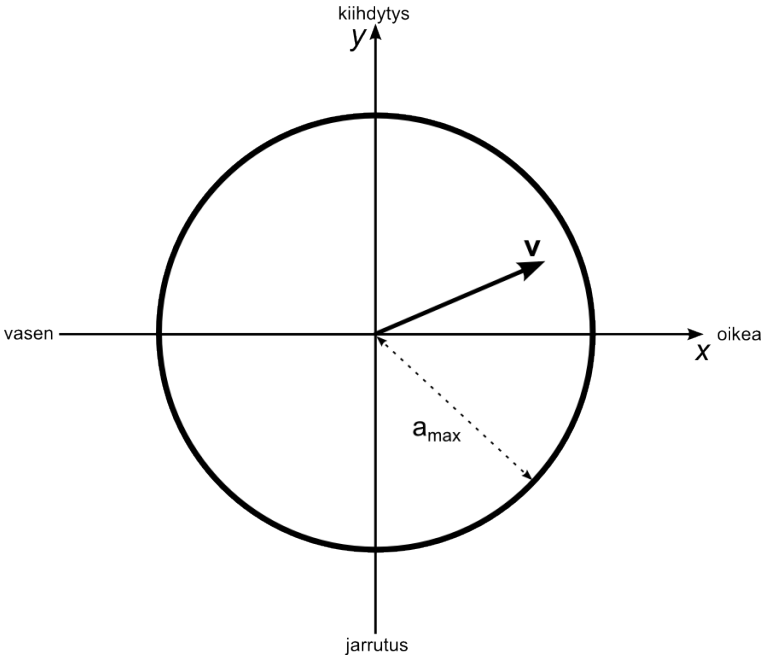
$$\begin{aligned} a &= F/f(a)m \\ F &\leq \mu W \\ W &= f(a)mg \end{aligned}$$

voidaan laskea

$$a_{max} = \frac{\mu W}{f(a)m} = \frac{\mu f(a)mg}{f(a)m} = \mu g \tag{3.2}$$

Tässä a_{max} on suurin kiihtyvyys, jolla rengas säilyttää pitonsa. Kaavasta on nähtävissä, että a_{max} on riippumaton auton massasta. Myös tässä kitkakerroin μ oletetaan vakioksi. Rengas luisuu, jos siihen kohdistuvien pitkittäis- ja sivuttaiskiihtyvyyksien yhteisvaikutus ylittää sen pidon eli rajan a_{max}. Tästä syystä esimerkiksi jarrutuksen on tapahduttava ennen kaarretta, sillä rengas ei pidä, kun siihen kohdistuu samanaikaisesti sekä suuri negatiivinen kiihtyvyys pitkittäissuunnassa että kääntymisen aiheuttama sivuttaiskiihtyvyys.

Pidon visualisoinnissa (ks. kuva 3.2) käytetään ympyrää, joka kuvaa renkaan tarjoamaa pitoa sekä vektoria, joka vastaa kiihtyvyyksien summaa. Ympyrän säde r on yhtä kuin renkaan suurin sietämä kiihtyvyys a_{max}. Kiihtyvyyksien summavektori $\boldsymbol{v}$ koostuu alkioista v_x ja v_y, jotka vastaavat sivuttais- ja pitkittäiskiihtyvyyksiä. Vektorin $\boldsymbol{v}$ pituus on $||\boldsymbol{v}|| = \sqrt{v_x^2 + v_y^2}$. Jos $||\boldsymbol{v}|| \leq a_{max}$ niin rengas pitää.



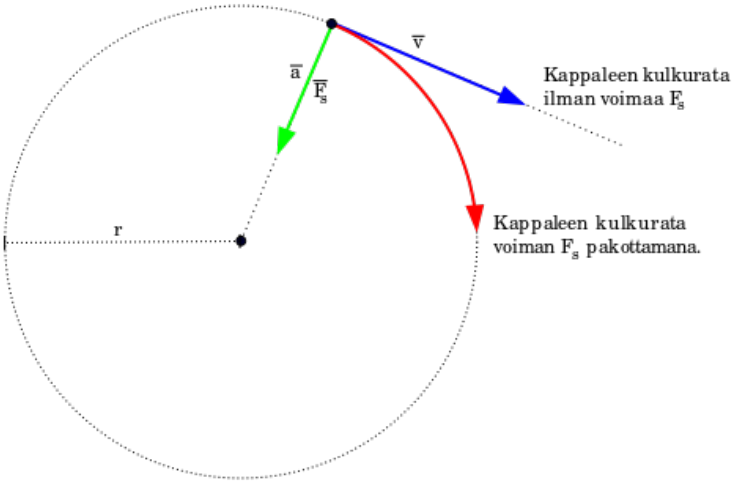
Kuva 3.2: Kuva havainnollistaa renkaan pitoa sekä siihen kohdistuvia kiihtyvyyksiä.

3.4 Kaarteet ja kääntyminen

Kun kuljettaja kääntää auton ohjauspyörää, renkaat kääntyvät ja alkavat työntää maanpintaa vasten sivusuunnassa. Newtonin III peruslain mukaisesti maanpinta työntää tällöin renkaita vastakkaiseen suuntaan, mistä aiheutuu autoon kohdistuva sivuttaiskiihtyvyys. Sivuttaiskiihtyvyys saa auton liikesuunnan muuttumaan. Jos kääntyvän ajoneuvon suunta ja nopeus pysyvät samana, se kulkee lopulta täyden ympyrän palaten lähtöpisteeseensä. Kääntyvän auton voidaan aina ajatella kulkevan pitkin kuvitteellisen ympyrän kehää.

Sentripetaali- eli keskihakuvoima $\bar{F}_a$ (ks. kuva 3.3) saa aikaan muutoksen suoraan kulkevan auton liikesuunnassa $\bar{v}$. Se suuntautuu kohti auton kulkeman kuvitteellisen kehän keskipistettä. Kääntyvien ja pyörivien kappaleiden yhteydessä puhutaan usein keskipakoisvoimasta, mutta se on vain näennäisvoima, joka havainnollistaa Newtonin I peruslain kuvaamaa inertiaa. Tosiasiallisesti autoon kohdistuvat ainoastaan voimat $\bar{v}$ ja $\bar{F}_a$, mutta inertia luo vaikutelman ikään kuin jokin vetäisi kappaleita käännöksestä poispäin. Sen voi havaita esimerkiksi siitä, että auton matkustajien on jyrkässä kaarteessa pinnisteltävä istuakseen suorassa.

Mitä nopeammin tai jyrkempää kaarretta auto kulkee, sitä enemmän keskihakuvoimaa tarvitaan sen kääntämiseen. Jos ajoneuvo kulkee liian suurella nopeudella



Kuva 3.3: Keskihakuvoiman vaikutus kappaleen kulkusuuntaan. Tekijä: Wikipedian käyttäjä *Tomia*. Kuvalla on CC-BY-2.5 lisenssi.

tai pyrkii kääntymään liian jyrkästi, niin sen pyörät eivät pysty välittämään suunnan muutokseen tarvittavaa voimaa. Tällöin pyöriin kohdistuva kiihtyvyys ylittää niiden pidon. Kun pyörät eivät pidä kaarteessa, auto aliohjautuu eli kääntyy liian loivasti.

Newtonin II lain avulla voidaan laskea, kuinka suuri sivuttaiskiihtyvyys tarvitaan kääntämään auto, joka kulkee nopeudella v kaarteessa, jonka säde on r. Jos lisäksi tiedätään renkaiden tarjoama pito a_{max} (yhtälö 3.2), voidaan laskea suurin mahdollinen nopeus, jolla auto voi kääntyä. Olkoon auton kulkema matka ajassa dt pitkittäissuunnassa dx ja sivuttaissuunnassa dy. Tällöin sen nopeudeksi v pitkittäissuunnassa saadaan $v = dx/dt$. Aikavälin dt alussa on nopeus sivusuunnassa nolla, ja sen lopussa dy/dt. Nopeuden muutos eli kiihtyvyys ajassa dt on siten dy/dt. Sivuttaiskiihtyvyyden a ilmaisee yhtälö

$$a = \frac{dy}{dt}\frac{1}{dt}$$

Kun auto liikkuu eteenpäin ympyrän säteen murto-osan $m = dx/r$, sen tulee kulkea sivusuunnassa samanmittainen matka, jotta se pysyy kulkemallansa kehällä. Tällöin pätee $dy = mdx = \frac{dx}{r}dx$. Nyt korvaamalla termi dy saadaan

$$a = \frac{dy}{dt}\frac{1}{dt} = \frac{\frac{dx}{r}dx}{dt^2} = \frac{dx^2}{dt^2}\frac{1}{r} = \frac{v^2}{r} \tag{3.3}$$

Edellisestä yhtälöstä ilmenee, että auton kääntymiseen vaadittava sivuttaiskiihtyvyys kasvaa nopeuden neliössä. Pienillä nopeuseroilla on suuri vaikutus tarvitta-

vaan keskihakuvoimaan. Toisaalta yhtälöstä ilmenee myös kaarteen säteen vaikutus vaadittuun kiihtyvyyteen. Mitä suurempi säde ja loivempi käännös, sitä pienempi kiihtyvyys tarvitaan kääntämään autoa. Taulukossa 3.3 on esimerkkejä tarvittavasta sivuttaiskiihtyvyydestä eri nopeuksilla erilaisissa kaarteissa. Vertailun vuoksi Beckmanin mukaan moottoriurheiluun soveltuva rengas pitää korkeintaan noin 1,1 g kiihtyvyydessä [1, s. 6].

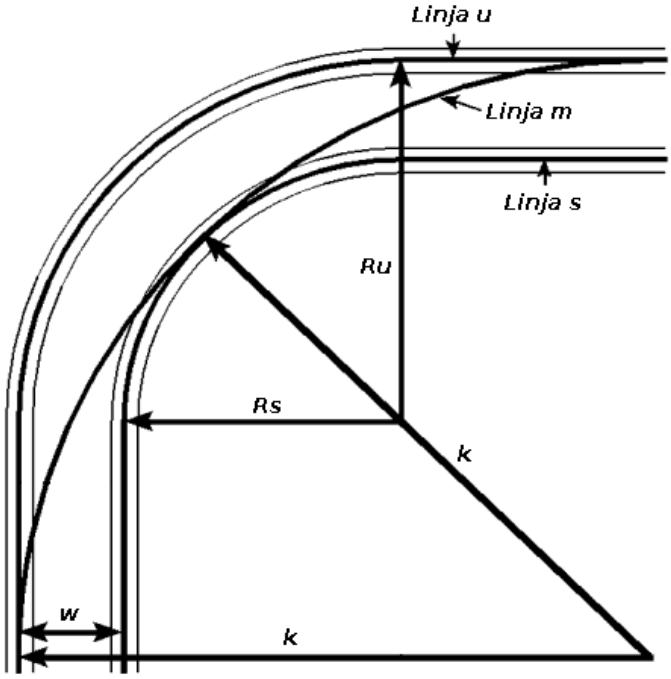
Taulukko 3.3: Tarvittava sivuttaiskiihtyvyys kaarteissa (g)

Nopeus	Säde m		
km/h	15	50	100
25	0,33	0,09	0,05
50	1,31	0,39	0,19
100	5,24	1,57	0,79

3.5 Ajolinjat

Auton sijoittuminen kaarteissa vaikuttaa merkittävästi kierrosaikoihin. Niin kutsuttu klassinen ajolinja (engl. *classic raceline*) lähestyy kaarretta radan ulkolaidalta, siirtyy sisälaidalle käännöksen taitekohdassa (engl. *apex*) ja kulkee taas ulkolaidalle tullessaan kaarteesta ulos. Taitekohdassa käännös on jyrkimmillään. Sen jälkeen auto kääntyy yhä loivemmin, kunnes kulkee jälleen suoraan. Klassinen ajolinja pyrkii sekä maksimoimaan ajoneuvon nopeuden pitämällä käännöksen mahdollisimman loivana että säilyttämään kuljetun matkan mahdollisimman lyhyenä. Koska klassinen ajolinja hyödyntää koko radan leveyden, se loiventaa kaarretta.

Seuraavaksi vertaillaan eri ajolinjojen nopeutta kaarteessa, joka kääntyy oikealle 90° kulmassa (ks. kuva 3.4). Tarkastelussa kaarretta tutkitaan yksinkertaisuuden vuoksi erillään sitä edeltävistä ja seuraavista rataosuuksista. Esimerkissä oletetaan lisäksi, että jarrutuksesta siirrytään välittömästi kääntymiseen, ja kääntymisestä kiihdyttämiseen, ja että nämä kolme vaihetta ovat erillisiä. Olkoon tarkastellun käännöksen sisäkaarteen säde $r_s = 20\,\text{m}$ ja ulkokaarteen $r_u = 30\,\text{m}$. Tällöin radan keskellä kulkevan linjan säde $r = 25\,\text{m}$. Olkoon radan leveys $w_r = 10\,\text{m}$ läpi käännökseen. Kun huomiodaan auton leveys $w_a = 2\,\text{m}$, kuljettajan käytettäväksi jää radan leveydestä on $w = w_r - w_a = 8\,\text{m}$. Samalla tavalla korjatut sisä- ja ulkokaarteen säteet ovat: $R_s = r_s + \frac{1}{2}w_a = 21\,\text{m}$ ja $R_u = r_u - \frac{1}{2}w_a = 29\,\text{m}$.



Kuva 3.4: Eri ajolinjoja 90° kaarteessa. Kuva Beckmania [1] mukaillen.

Yhtälön 3.3 perusteella suurin mahdollinen nopeus v_{max}, jolla kaarteen voi ajaa kasvaa suhteessa käännöksen säteen neliöjuureen $v_{max} = \sqrt{ra}$, kun sivuttaiskiihtyvyys a oletetaan vakioksi. Tämän perusteella klassinen ajolinja m (ks. kuva 3.4) sallii suurimman nopeuden, sillä sen säde on suurin. Linja m on yksinkertaistettu, sillä se saavuttaa taitekohtansa kaarteen geometrisessa keskipisteessä. Klassisen ajolinjan taitekohta tulee keskipisteen jälkeen, koska tällöin auto voi alkaa kiihdyttää aikaisemmin.

Kuvan 3.4 ajolinja s noudattaa käännöksen sisäkaarretta ja u ulkokaarretta. Linja s tarjoaa tiettyjä etuja linjaan m nähden: se kulkee lyhyemmän matkan ja mahdollistaa pidemmän kiihdytyksen ennen ja jälkeen kaarteen. Toisaalta linjaa s noudattavan auton tulee kääntyä huomattavasti pienemmällä säteellä kuin linjalla m. Linja u on puolestaan sekä matkaltaan pidempi että säteeltään pienempi kuin linja m. Taulukossa 3.4 on listattu näiden kolmen ajolinjan sallimat nopeudet v_s, v_m ja v_u, kun sivuttaiskiihtyvyys on $1{,}10\,g$. Linjan m säde k on noin 48 m.

Jotta linjojen vertailu on pätevä, tulee varsinaisen käännöksen lisäksi tarkastella myös linjojen s ja u suoria osuuksia. Beckmanin mukaan ajolinja m on kokonaisuutena useita sekunnin kymmenyksiä nopeampi kuin s, ja s hieman nopeampi kuin u. Tarkastelussa on oletettu, että auton kiihtyvyys on $0{,}5\,g$ ja jarrutus $-1{,}0\,g$. Auto voi

Taulukko 3.4: Käännösnopeudet kuvan 3.4 eri ajolinjoilla (km/h)

Linja s	Linja m	Linja u
54,19	81,93	63,68

linjalla s kiihdyttää ennen käännöstä, mutta sen on jyrkän käännöksen vuoksi jarrutettava niin voimakkaasti, että se kulkee kaarteen jälkeisellä suoralla linjaa m noudattavaa ajoneuvoa hitaammin. Beckmanin mukaan sekunnin kymmenyksien ero on kilpa-ajossa ratkaiseva jo kierroksen kokonaisajassa, saatika yksittäisessä kaarteessa.

Beckman jatkaa vertailua muuntelemalla esimerkin kaarteen sädettä ja radan leveyttä. Hänen mukaansa ajolinja m on nopeampi kuin s, ja s nopeampi kuin u riippumatta kaarteen säteestä. Mitä pienempi säde on, sitä nopeampi m on suhteessa muihin linjoihin. Se on myös nopein radan leveydestä riippumatta. Ero korostuu linjan m hyväksi, mitä leveämpi rata on. Linja u muuttuu nopeammaksi kuin s huomattavan leveillä radoilla.

Yksittäisen kaarteen ajolinjaa ei kuitenkaan voi tosiallisesti tarkastella erillään muusta radasta. Esimerkiksi käännöksen taitekohdan viivästäminen tai aikaistaminen vaikuttaa siihen, kuinka pitkään auto voi kiihdyttää ennen seuraavaa rataosuutta. Oletetaan, että kaarretta seuraa pitkä suora. Beckman tarkastelee teoksessaan millä tavalla ajoneuvon nopeus suoran alussa vaikuttaa sen nopeuteen suoran lopussa. Lisäksi hän tutkii, miten alkunopeus vaikuttaa suoralla kuluvaan aikaan. Beckman huomioi laskuissaan lukuisia kiihtyvyyteen vaikuttavia tekijöitä, kuten moottorin väännön ja kierrosluvun, ilmanvastuksen ja vierintävastuksen sekä valitun vaihteen välityssuhteen. Hän päätyy laskuissaan tulokseen, jonka mukaan 60 m suoralla 3,2 km/h ero alkunopeudessa johtaa 0,06 s eroon ajassa. 9,7 km/h ero taas 0,2 s eroon. Auton nopeus suoran lopussa ei muutu kovin merkittävästi eri alkunopeuksilla. Esimerkiksi kun ajoneuvo saapuu suoralle 80 km/h nopeudella, se kulkee suoran lopussa vain noin 13 km/h nopeammin kuin jos se oli aloittanut 40 km/h nopeudesta.

3.6 Auton tekniikka

Tässä luvussa esitellään auton tekniikkaa siinä määrin, mitä tarvitaan lukujen 5, 6 ja 7 seuraamiseksi. Luvussa selostetaan ABS- ja TCL-järjestelmien sekä vaihdelaatikon

ja kytkimen toiminta pääpiirteissään.

ABS-jarrut (saks. *Antiblockiersystem*) eli lukkiutumattomat jarrut on suunniteltu ehkäisemään jarrutuksessa tapahtuvaa luisumista. Nykyaikaisia ABS-jarruja ohjaa tietokone, joka tarkkailee renkaiden pyörimisnopeutta. Se päättelee pyörimisnopeuden perusteella, onko pyörä lukkiutunut. Jos pyörä on menettänyt pitonsa, tietokone vapauttaa jarrun. Se voi vapauttaa ja lukita pyörän jopa 15 kertaa sekunnissa [31]. Parkerin mukaan ABS-jarrut eivät tarjoa merkittävää etua kuivalla tiellä, mutta ne pysäyttävät auton märällä tai jäisellä pinnalla selvästi tavanomaisia jarruja nopeammin. Parker ei kerro tarkemmin, miten hän vertaa ABS-jarrujen suorituskykyä. Tavallisilla jarruilla voidaan suorittaa lukkojarrutus, tai niitä voidaan niin sanotusti pumpata. Pumppaus tarkoittaa sitä, että poljinta nostetaan ajoittain niin, että pyörät pääsevät pyörimään. Tämä tekniikka lyhentää jarrutusmatkaa lukkojarrutukseen verrattuna.

TCL- (engl. *traction control*) eli luistonestojärjestelmä vähentää pyörien sutimista auton kiihdyttäessä, ja siten parantaa pitoa [30]. Toisin kuin ABS-järjestelmä TCL tarkkailee vain vetävien pyörien liikettä. Jos järjestelmä havaitsee, että jokin renkaista menettää pitonsa, se siirtää kuormitusta muille pyörille [31]. TCL päättelee luisuuko rengas kaavan $\frac{r \cdot \omega - v}{r \cdot \omega}$ avulla. Kaavassa r on renkaan säde, ω sen kulmanopeus ja v ajoneuvon nopeus. Rengas sutii, jos $(r \cdot \omega - v) > 0$. TCL-järjestelmä voi toimia eri tavoin esimerkiksi katkaisemalla kaasutuksen tai käyttämällä jarruja.

Keskeisimpiä moottorin suorituskykyä mittaavia suureita ovat *teho* ja momentti eli *vääntö*. Siinä missä teho saa kappaleen kulkemaan suoraan, vääntö saa sen pyörimään. Vääntö mittaa moottorin kykyä pyörittää auton pyöriä. Sen mittayksikkö on *newtonmetri*, *Nm*. Vääntö kulkee moottorista auton pyöriin vaihdelaatikon välityksellä. Vaihdelaatikko muuntaa moottorin väännön pyörille sopivaksi. Vaihteet myös tasaavat moottorin kuormitusta.

Vaihdelaatikko koostuu vaihderattaista, sekä niitä siirtävästä koneistosta. Rattaista aina yksi kerrallaan on yhteydessä moottoriin ja voimaansiirtoakseliin. Voimaansiirtoakseli välittää väännön vetäville pyörille. Vaihdelaatikkoa tarvitaan, koska moottorin käyntinopeus vaikuttaa sen tuottamaan vääntöön. Parkerin mukaan moottori saavuttavat maksimivääntönsä silloin, kun sen käyntinopeus on 50 % – 60 % maksimista. Vaihtamalla moottorin väännön kerrointa vaihdelaatikon avulla, tätä väännön vaihtelua voidaan tasata. Vaihteen kerrointa kutsutaan välityssuhteeksi. Jos välityssuhde on esimerkiksi 3:1, moottori pyörii kolme kierrosta jokaista voimansiirtoakselin kierrosta kohden. Tämä tarkoittaa, että vaihde kolminkertaistaa

sille tulevan väännön.

Kytkin on auton osa moottorin ja vaihdelaatikon välissä [37]. Koska vaihderattaat ovat erikokoisia, ne pyörivät eri nopeudella [31]. Tästä johtuen moottorin ja voimansiirtoakselin pyörimisnopeuksien suhde muuttuu, kun vaihdetta vaihdetaan. Kytkimen avulla nämä pyörimisnopeudet voidaan sovittaa toisiinsa asteittain. Kytkin myös tasaa katkoksia kiihdytyksessä ja vähentää voimansiirron osien kulumista. Sitä voidaan ohjata automaattisesti tai manuaalisesti kytkinpolkimen avulla. Kytkimen oikea käyttö on eräs kilpa-ajajan taidoista, sillä mitä kauemmin vaihteen vaihtaminen kestää, sitä enemmän se kasvattaa kierrosaikaa.

4 The Open Race Car Simulator

Tässä luvussa esittellään *The Open Race Car Simulator* [45] eli TORCS, joka on avoimen lähdekoodin autopeli. Luvussa kerrotaan TORCS-projektin historiasta ja nykytilasta, sekä pelin ominaisuuksista. Lisäksi tarkastellaan TORCSin toteutusta yleisellä tasolla sekä selvitetään, miten agentit sijoittuvat pelin arkkitehtuurissa. Luvussa esitellään lisäksi TORCSin palvelinversio, ja kerrotaan sen eroista työpöytäversioon nähden.

4.1 TORCSin historia ja ominaisuudet

Tämän alaluvun pääasiallinen lähde on TORCS-projektin kotisivut [45]. TORCS on realistinen autopeli, joka mallintaa muun muassa renkaan pidon, ilmanvastuksen ja polttoaineen kulutuksen [21]. Pelin versio 1.0.0 julkaistiin GPLv2-lisenssillä joulukuussa vuonna 2001. Viimeisin versio 1.3.6 julkaistiin 24.4.2014. Peli on tarjolla Linux-, OS X- ja Windows-käyttöjärjestelmille. TORCS tukee useita eri syöttölaitteita, kuten peliohjaimia, näppäimistöä ja hiirtä. TORCS-projektin kotisivuilla on yli 3 000 rekisteröitynyttä käyttäjää, ja sillä on vierailtu yli 10 miljoonaa kertaa sitten vuoden 2001. Peli ladattiin *SourceForge*-sivustolta 117 000 kertaa vuoden 2013 aikana. Lataustilastojen perusteella Windows-versio on suosituin [46]. TORCSiin on mallinnettu lukuisia eri autoja ja kilparatoja. Osa pelin mukana tulevista radoista on fiktiivisiä, ja osa mallintaa tosimaailman ratoja. Realististen ratojen nimet on muutettu.

TORCS on ohjelmoitu C++-ohjelmointikielellä. Se ei noudata tiukasti C++-kielen tyyliä, vaan laajoja osia pelistä on toteutettu C-kielellä. Peli käyttää grafiikan hahmontamiseen (engl. *render*) OpenGL-rajapintaa [51]. Se hyödyntää toteutuksessaan lisäksi GLUT-, GLU- ja plib-kirjastoja. Pelin äänet on ohjelmoitu OpenAL-kirjaston avulla. TORCSin loivat alunperin Eric Espié ja Christophe Guionneau, mutta nykyisin projektia johtaa Bernhard Wymann. TORCSista on syntynyt myös sisarprojekti *Speed Dreams*[1].

TORCSin mukana tulee 11 eri bottia. Ne poikkeavat toisistaan merkittävästi niin

[1] `http://www.speed-dreams.org/`

toteutustekniikoiltaan kuin ajotaidoiltaankin. Esimerkiksi *berniw* laskee ajolinjansa reaaliaikaisesti. Se kellottaa varsin kilpailukykysiä kierrosaikoja [3]. *Bt*-botin toteutus perustuu pitkälti Wymanin oppaaseen [51]. Kuitenkin se päivittää ajon aikana tietojansa kaarteiden efektiivisestä säteestä. Toisin sanoen se opettelee, miten kaarteet voi ajaa kaikkein leveimmällä tavalla. Osa pelin agenteista vaikuttaa kokeellisilta. Esimerkiksi *bt* ajoi testeissäni säännönmukaisesti ulos radan *wheel 2* neulansilmässä. Neulansilmä on arkikielinen nimitys jyrkälle kaarteelle, joka kääntyy noin 180°. TORCSissa ihmispelaaja on ikäänkuin botin erikoistapaus. *Human*-moduuli sisältää pelaajan toteutuksen, ja se sijaitsee samassa hakemistossa kuin muutkin botit.

TORCSissa on harjoittelutila (engl. *practice mode*), jonka avulla voi testata botteja. Botin toteutuksen voi kääntää binääriksi harjoituskertojen välissä ilman, että peliä tarvitsee sulkea. Myös botin käyttämiä paremetreja voidaan suoritusten välillä muuttaa. Tämä johtuu siitä, että bottimoduuli ladataan aina uudelleen ennen ajoa. Toistuva lataaminen voi tosin aiheuttaa muistivuotoja ainakin pelin vanhemmissa versioissa [51]. Harjoitusajon jälkeen TORCS esittää yhteenvedon botin suoriutumisesta. Yhteenvedossa kerrotaan paras kierrosaika, huippunopeus ja ajoneuvon kärsimät vauriot. Peli voidaan käynnistää myös tekstitilaan. Tällöin botti harjoittelee ilman, että peli hahmontaa grafiikkaa ruudulle. Tämä nopeuttaa esimerkiksi koneoppimista hyödyntävien tekoälyjen harjoittamista. Toisaalta pelistä voi kaapata videota - myös tekstitilassa. Testeissäni videon kaappaaminen hidasti pelin suoritusnopeuden noin puoleen.

4.2 Agentit TORCSin arkkitehtuurissa

TORCSissa botin toteutus muodostaa erillisen moduulin, joka kommunikoi pelin kanssa rajapinnan läpi. Tässä luvussa esitellään moduulin ja rajapinnan toteutus, sekä kerrotaan minkälaisia tietorakenteita rajapinnan kautta kulkee. Lisäksi esitellään moduulin konfiguraatio- ja muut oheistiedostot. Selostusta luettaessa on huomattava, että se perustuu omaan ymmärrykseeni TORCSin toiminnasta. Siitä ei ole tarjolla kattavaa dokumentaatiota, ja toisaalta sen lähdekoodi on harvakseltaan kommentoitua. Luvun lähteenä käytetty Wymannin bottiopas [51] on kirjoitettu vuonna 2004 TORCSin versiota 1.2.4 varten, kun taas itse käytän versiota 1.3.4. Opas koostuu pitkälti esimerkkikoodista, jota ei selitetä kovin seikkaperäisesti.

TORCSin pelimoottori simuloi kilpa-ajamista 2 ms jaksoissa [45]. Bottien ohjelmakoodia kutsutaan 20 ms välein. Botin on joka kutsulla siis reagoitava kymmenen

jakson tapahtumiin. Kilpa-autot voivat kulkea jopa 300 km/h [49]. Tällä nopeudella botti etenee kutsujen välissä 1,6 m. Bottia kutsutaan tiedoston `raceengine.cpp` silmukasta. Tämä silmukka on pelin pääohjelma silloin, kun kilpa-ajo on käynnissä. Simulaatiomoottorin tai grafiikan hahmontamisen toteutusta TORCSissa ei tässä teoksessa käsitellä sen tarkemmin, sillä niiden yksityiskohtainen tunteminen ei ole tarpeen botin ohjelmoinnissa. Esimerkiksi bottioppaassa pelin toteutuksesta ei puhuta juuri lainkaan.

TORCS rekisteröi käynnistyessään käyttöönsä kaikki hakemistosta `/data/bots/` löytyvät bottimoduulit. Rekisteröintiä varten moduulin tulee sisältää lähdekooditiedosto `<moduulinNimi>.cpp`, joka toteuttaa rajapinnan funktiot `<moduulinNimi>` ja `InitFuncPt`. Aivan ensimmäiseksi peli kutsuu funktiota `<moduulinNimi>`, jossa luodaan moduulin ilmentymät `tModInfo[]`-taulukkoon. Moduulista luodaan yhdestä kymmeneen ilmentymää. Kukin ilmentymä voi ohjata pelissä omaa autoaan. Testeissäni versio 1.3.4 kaatui, jos botista ei luotu maksimimäärää ilmentymiä. Ilmentymien maksimimäärä on tallennettu globaaliin vakioon `MAX_MOD_ITF`. Kaikkia luotuja ilmentymiä ei kuitenkaan tarvitse käyttää. Funktio `InitFuncPt` kertoo pelille, mitä moduulin funktiota sen tulee eri tilanteissa kutsua. Se tekee tämän antamalla tietueen `tRobotItf` funktio-osoittimille moduulin funktioiden osoitteet. TORCS kaatuu, jos johonkin näistä ei ole sijoitettu kelvollista osoitetta. Tietue `tRobotItf` muodostaa osan rajapinnasta, sillä myös se määrää, minkämuotoisia funktioita moduulin on toteutettava. C++-kieli ei erikseen tarjoa tapaa esitellä rajapintaa, kuten esimerkiksi Javan avainsana `interface` [25].

Bottimoduulin liittyy XML-tiedosto, joka ilmoittaa pelille sen ilmentymien tiedot. Näitä ovat muun muassa ilmentymän ajoneuvo, joukkue sekä tunnusväri. Ilmentymien ominaisuuksia voi muuttaa vain XML-tiedostoa muokkaamalla. Versiossa 1.3.4 ei ole pelin sisäistä valikkoa tätä varten. Kuhunkin ilmentymään voi liittyä lisäksi XML-tiedosto, joka kuvaa sen käyttämän auton säädöt. Näitä ovat muun muassa renkaiden pystykallistuma, polttoaineen määrä ja jousituksen kireys. Myös botti voi lukea näitä XML-tiedostoja.

TORCSin työpöytäversiossa botilla ei ole ennalta määriteltyjä sensoreita. Sen sijaan ohjelmoija voi itse luoda erilaisia sensoreita, sillä botin saatavilla on paljon tietoja autosta ja radasta. Peli välittää botille osoittimet tietueisiin `tTrack`, `tCarElt` ja `tSituation`. Nämä ovat laajoja tietueita, jotka jakautuvat edelleen osatietueisiin. Niihin on tallennettu myös pelin grafiikkamoottorin tarvitsemia tietoja. Auton staattiset ja dynaamiset tiedot on tallennettu tietueeseen `tCarElt`. Staattisia tietoja

ovat esimerkiksi auton mitat, ja dynaamisia nopeudet eri akseleiden suhteen. Tietue `tSituation` sisältää tietoja kuluvasta kilpailusta, esimerkiksi kisan johtajan ajamat kierrokset. Tietue `tTrack` sisältää kaksisuuntaisen linkitetyn listan, johon on ladattu kilparadan tiedot. Lista on linkitetty renkaaksi, eli viimeisestä alkiosta on linkki ensimmäiseen ja päinvastoin. Listan alkiot ovat `tTrackSeg`-tietueita, ja linkit `tTrackSeg*`-osoittimia. Listan tiedot ladataan XML-tiedostosta. Rata on tiedostossa jaettu kaarteisiin, suoriin ja muihin osuuksiin. Kun rata ladataan linkitettyyn listaan, se jaetaan muutaman metrin mittaisiksi segmenteiksi. Yksi listan alkio vastaa yhtä segmenttiä. Yksittäinen radan kaarre tai suora jakaantuu useaksi segmentiksi. Segmenttejä on kolmea päätyyppiä: kaarre vasemmalle, kaarre oikealle ja suora. `TTrackSeg`-tietue sisältää tietoja segmentin geometrisista ominaisuuksista. Näitä ovat muun muassa sen pituus, leveys ja säde.

TORCSin työpöytäversiossa botti saa tietoja toisista agenteista tietueen `tSituation` kautta. Useampi toteutus, kuten *berniw*, *inferno* ja *Tutor* käyttää luokkia `Opponent` ja `Opponents` toisten kuljettajien seuraamiseen. Kustakin vastustajasta luodaan `Opponent`-olio. Tämä lisätään moduulien ilmentymien kesken jaettuun `Opponents`-olioon. `Opponent`-luokkaa ei kuitenkaan ole määritelty kovin täsmällisesti, sillä siinä on metodi, joka palauttaa `Opponent`-luokan yksityisen `tCarElt* car` attribuutin. Täten kunkin botin on mahdollista lukea toisten kuljettajien ajoneuvojen tietoja.

4.3 TORCSin palvelinversio

Tässä luvussa kerrotaan TORCSin palvelinversion toteutuksesta, ja vertaillaan sitä työpöytäversioon. Luvun lähteenä on käytetty Loiaconon et al. [20] teosta, joka kuvaa vuoden 2011 palvelinversion. TORCSia hyödyntävä tieteellinen tutkimus käyttää pääasiassa palvelinversiota, joten sen toteutus on hyvä tuntea pääpiirteissään.

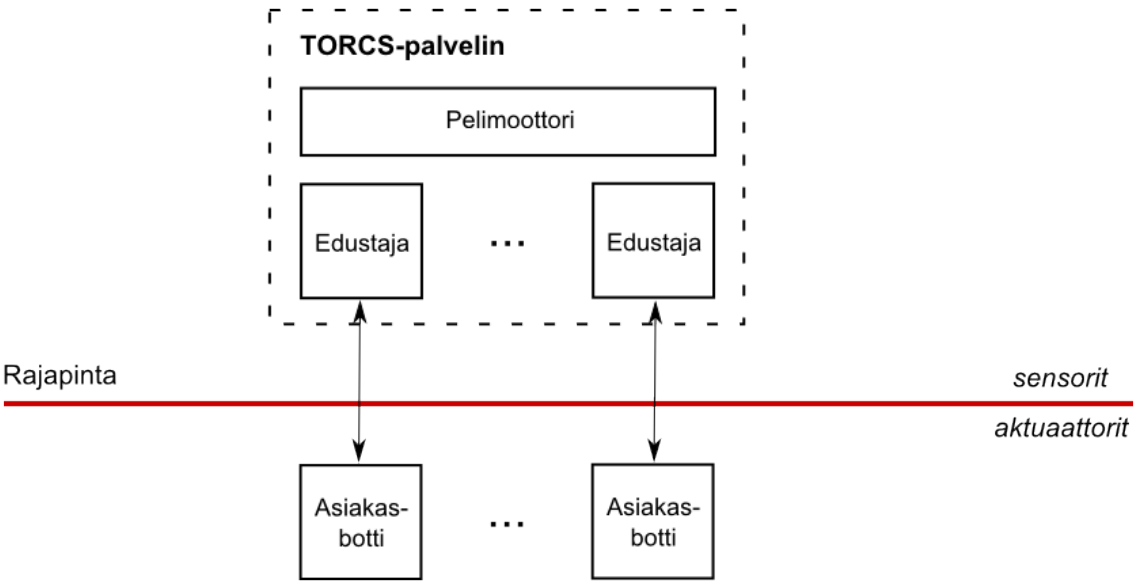
TORCSiin ohjelmoiduille boteille järjestetään vuosittain kilpailuja, joissa ne ajavat toisiaan vastaan. Botit voivat kilpailla eri tavoilla: ne voivat ajaa nopeimmasta kierroksesta aika-ajoissa tai kisan voitosta yhtäaikaa samalla radalla. Kilpailu voi sisältää myös lämmittelykierroksia. Lämmittelykierrosten aikana esimerkiksi koneoppimista hyödyntävä botti voi opetella ajamaan radalla. Kilpailuissa käytettävä TORCS noudattaa asiakas-palvelin -arkkitehtuuria. Loiacono et al. perustelevat ratkaisua seuraavasti:

- Koska pelimoottoria ei työpöytäversiossa ole erotettu botin toteutuksesta, bo-

tilla on pääsy kaikkeen kilpailua ja kilparataa koskevaan tietoon. Tällöin ei voida taata, että botit hyödyntävät saatavilla olevaa tietoa tasavertaisesti.

- Botin voi toteuttaa työpöytäversioon vain C- tai C++- ohjelmointikielellä. Palvelinversio mahdollistaa muitakin kieliä.
- Palvelinversioon voi ohjelmoida botin tuntematta lainkaan pelimoottorin toteutusta.
- Palvelinversio asettaa tiukemman aikarajan botin toimille.

Agentti toimii TORCSin kilpailuversiossa erillisinä prosessina, joka kommunikoi palvelimen kanssa UDP-yhteyden yli. Palvelin ajaa samaa pelimoottoria kuin TORCSin työpöytäversio. Ainoa poikkeus toteutuksessa ovat palvelimella suoritettavat bottien edustajat (engl. *proxy*), jotka hallinnoivat bottien ja palvelimen välisiä yhteyksiä. Kutakin asiakasbottia vastaa yksi edustaja. Ennen kilpailun alkamista asiakas muodostaa yhteyden edustajaansa. Kisan aikana edustaja lähettää asiakkaalle sen sensoreiden keräämät tiedot, ja odottaa vastausta 10 ms. Jos palvelin ei määräaikaan mennessä saa edustajalta vastausta, se olettaa botin jatkavan edellisen jakson toimia. Palvelin päivittää pelin tilannetta 20 ms väliajoin.



Kuva 4.1: TORCSin palvelinversion arkkitehtuuri.

Botti ja sen edustaja eivät kommunikoi suoraan keskenään, vaan rajapinnan kautta. Rajapinta tarjoaa asiakasbotille joukon sensoreita. Ne antavat tietoja muun muas-

sa auton etäisyydestä radan reunaan, vastustajien sijainnista, jäljellä olevasta polttoaineesta, valitusta vaihteesta sekä auton nopeudesta. Radan reunojen ja vastustajien havainnointi perustuu palvelinversiossa säteenseurantaan. Asiakasbotin aktuaattoreita ovat ohjauspyörä, kaasu- ja jarrupoljin sekä kytkin ja vaihteet. Lisäksi se voi suorittaa erilaisia metatoimintoja, kuten pyytää kisan aloittamista alusta. Päätettyään aktuaattoreidensa arvot botti lähettää ne edustajalleen.

5 Autopelien agenttien tekniikat, uskottavuus ja tekotyperyys

Tässä luvussa annetaan yleiskuva autopeleistä, ja niiden agenteista niin tekniikan kuin uskottavuuden osalta. Luvussa pohditaan myös, miten tekotyperyys voisi ilmetä niissä. Teoksen tutkimuskohteena on botit, mutta tässä luvussa käsitellään laajemmin erilaisia autopelien agentteja. Käsittely on laajempi, koska mielipiteet agenttien perustehtävästä autopeleissä eriävät [28, 50]. Pyrin luvussa 5.4 muodostamaan eri näkemyksistä jonkinlaisen synteesin. Luvussa 5.1 kerrotaan, millaisia menetelmiä agentti voi käyttää ohjaukseen, nopeuden säätelyyn ja vaihteiden vaihtamiseen. Luvussa 5.2 puolestaan selvitetään, miten agentti voi muodostaa ajolinjan ja noudattaa sitä. Luvussa 5.3 esitellään menetelmiä, joiden avulla agentti voi reagoida muihin kuljettajiin. Lopuksi luvussa 5.4 pohditaan, mitä agentin uskottavuus, viihdyttävyys ja tekotyperyys tarkoittavat autopelissä.

Autopelit ovat urheilupelien alagenre. Toisin kuin useimmissa urheilupeleissä, autopeleissä ei ohjata ihmishahmoa vaan autoa. Toisin sanoen pelaaja ei säätele kuljettajan liikkeitä, vaan ohjaa suoraan auton hallintalaitteita. [14] Autopelit poikkeavat toisistaan niin realismiltaan kuin tyyliltään. Esimerkiksi *Pelit*-lehti jaottelee autopelit *arcade-* ja *simulaattoripeleihin* [32]. *Arcade*-pelien ajomalli on yksinkertainen. Ne eivät pyri simuloimaan fysiikkaa mahdollisimman tarkasti, vaan helpottavat ajamista ja auton hallinta. *Arcade*-pelit tarjoavat kevyttä viihdettä, ja niissä näyttävä ajaminen on usein keskeisessä osassa. Esimerkiksi *Burnout 3: Takedown* ja suomalainen *FlatOut 2* ovat arcade-pelejä. Niissä molemmissa saa törmätä toisiin autoihin ja käyttää oikoreittejä. Realistiset simulaattoripelit pyrkivät mallintamaan ajamista tarkasti, ja niissä on ajettava sääntöjen mukaisesti. Niissä ei saa oikaista kaarteita tai törmätä toisiin autoihin. Muun muassa *Forza Motorsport 5*, *Gran Turismo 6* [14] ja TORCS [21] ovat simulaattoripelejä.

Autopelin agentti toimii monimutkaisessa ympäristössä, joka vaatii siltä reaaliaikaista päätöksentekoa. Sen tavoitteena on voittaa kilpailu tai ajaa nopein kierros. Näiden tavoitteiden saavuttaminen vaatii useiden osatavoitteiden hallitsemista. Agentin tulee muun muassa pysyä tiellä, tarkkailla muita kuljettajia ja säilyttää nopeutensa korkeana. [49] Nämä osatavoitteet ovat lisäksi keskenään riippuvaisia.

Esimerkiksi tiellä pysyminen on sitä haastavampaa, mitä nopeammin auto kulkee. Laird ja van Lent [16] katsovat, että agentit ovat autopeleissä taktisen vastustajan roolissa. Umarovin ja Mozgovoyn [48] mukaan näille: *"– strateginen päätöksenteko ei useimmiten ole niin tärkeää kuin välitön ja järkevä reagoiminen vastustajan tekemisiin."*[1] Luvussa 2.3 esitetään taktisen vastustajan teknisiä vaatimuksia. Esimerkiksi navigoimisen ja realistisen liikkumisen vaatimusten voi autopelissä tulkita tarkoittavan, että agentti noudattaa klassista ajolinjaa. Koska autopelin agentti toimii reaaliaikaisesti, sen käyttämät heuristiikat eivät voi olla laskennallisesti raskaita. Toisaalta kauppallisten pelien menetelmät eivät saa olla liian monimutkaisia ohjelmoida.

Tässä luvussa käytetään esimerkkeinä pääosin TORCSin botteja. Näitä ovat sekä pelin mukana tulevat että akateemisessa tutkimuksessa kehitetyt toteutukset. Vertailun vuoksi tarkastellaan myös kaupallisten pelien agentteja. TORCSin mukana tulevista boteista esitellään *Tutor*ia ja *berniw*iä. Ne ovat molemmat pelin pääsuunnittelijan Bernhard Wymannin ohjelmoimia [45, 51]. *Tutorilla* viittaan bottiin, joka ohjelmoidaan Wymannin vuonna 2004 kirjoittamassa oppaassa [51]. *Berniw* on hänen kehittämänsä edistyneempi botti, josta tulee TORCSin jakelun mukana kolme eri versiota. Versiot lähinnä variovat samoja tekniikoita.

5.1 Agenttien tekniset menetelmät

Tässä luvussa kerrotaan tekniikoista, joiden avulla agentti kuljettaa autoa, eli ohjaa, kiihdyttää, jarruttaa ja vaihtaa vaihteita. Luvussa esitellään pääasiassa akateemisessa tutkimuksessa kehitettyjä menetelmiä. Lisäksi esitellään lyhyesti erilaisia koneoppimista hyödyntäviä tekniikoita.

Agentti kerää sensoreillaan tietoja radasta ja muista autoista. Kerättyjen tietojen avulla sen on päätettävä toimenpiteistään. Mitä tarkemmat sensorit, sitä enemmän ne tarjoavat agentille informaatiota sen ympäristöstä [44]. Toisaalta mitä hienostuneempia heuristiikkoja agentti käyttää, sitä tarkemmin se voi ajaa. Esimerkiksi *berniw* ja *Tutor* laskevat jarrutusmatkan eri tavalla. Berniw laskee matkan d yhtälöllä:

$$d = (v_1^2 - v_2^2)(m/(2{,}0fgm + v_2^2(fA_xA_y)))$$
$$f = \mu\nu\kappa$$

missä v_1 on auton nopeus ja v_2 tavoitenopeus, m ajoneuvon ja polttoaineen yhteen-

[1] *"–strategic decision-making is usually less important than quick and smart reaction to immediate actions of the opponent."*

laskettu massa, μ tienpinnan ja ν ajoneuvon kitka, κ tien vaakakallistus sekä A_x ja A_y aerodynamiikkaan liittyviä tekijöitä, joihin vaikuttavat muun muassa auton maavara. Kertoimen ν tarkoitusta ei lähdekoodissa selitetä. *Tutor* taas käyttää yksinkertaista kaavaa: $(v_1^2 - v_2^2)/(2{,}0\mu g)$. Jälkimmäinen kaava arvioi jarrutusmatkan usein edellistä pitemmäksi. Se tekee botin ajamisesta varovaisempaa kuin tarvitsisi. Yksinkertaisempi kaava ei myöskään huomio ajoneuvon suorituskykyä.

Tomlinsonin [44] mukaan pelaajat eivät huomaa kuin erittäin tökerön liikkumisen. He huomaavat, jos agentti juuttuu ympäristön esteisiin, mutta eivät erota tarkkaa ohjausalgoritmia keskinkertaisesta. Tomlinsonin mukaan kovin yksinkertaiset tekniikat eivät kuitenkaan sovellu nykyaikaiseen autopeliin. Hän jakaa agenttien ohjausmenetelmät lokaaleihin ja globaaleihin. Lokaalit menetelmät soveltuvat nimensä mukaisesti kulkusuunnan etsimiseen lähiympäristöstä tai jostakin sen esiprosessoidusta mallista. Muun muassa säteenseuranta- ja voimakenttämenetelmä ovat lokaaleja. Globaalit menetelmät vaativat suunnittelua. Niiden avulla agentti kulkee kohti päämäärää, jota se ei tällä hetkellä havaitse. Esimerkiksi A*-algoritmi on globaali menetelmä.

TORCSin palvelinversio tarjoaa botille säteenseurantasensorit, joiden avulla se voi ohjata autoa [20]. Ne toimivat siten, että auton keskipisteestä piirretään 18 vektoria eri suuntiin. Ensimmäinen vektori suuntaan $-\frac{\pi}{2}$, ja viimeinen $\frac{\pi}{2}$. Kulmat ovat suhteessa ajoneuvon suuntaan: jos ajoneuvon suunta on α, niin ensimmäinen vektori kulkee suuntaan $\alpha - \frac{\pi}{2}$. Kutakin vektoria seurataan, kunnes se kohtaa radan reunan, toisen auton tai saavuttaa tietyn maksimietäisyyden. Maksimi vaihtelee pelin eri versioissa [3, 20]. Sensorit eivät kerro, kumman radan raunan, tai minkä auton säteet kohtaavat. Seurannan jälkeen botti valitsee auton suunnaksi säteen, joka kulki kauimmaksi. Kuva 6.3 esittää säteenseurantamenetelmän pääpiirteet. Eri tutkijaryhmät, kuten Butz ja Lönneker [3], Kinnaird-Heether ja Reynolds [22] sekä Quadflieg et al. [33] ovat käyttäneet menetelmää bottikilpailuissa (ks. luku 4.3). Kinnaird-Heether ja Reynolds pitävät sitä riittävän hyvänä, joskaan eivät optimaalisena. Tomlinson [44] huomauttaa, että säteenseuranta voi olla laskennallisesti raskas, jos agentin ympäristö on geometrialtaan monimutkainen, tai käytetään hyvin montaa sädettä.

Toinen Tomlinsonin [44] mainitsema lokaali ohjausmenetelmä on voimakentät. Ne ovat ikään kuin käänteinen tapa toteuttaa säteenseuranta. Sen sijaan, että seurattaisiin säteitä ajoneuvosta käsin, etsitään kenttiä, jotka vetävät autoa puoleensa tai hylkivät sitä. Uusitalo ja Johansson [49] käyttivät voimakenttiä TORCS-bottinsa

ohjaamiseen. Heidän toteutuksensa yhdistelee useita eri voimakenttämenetelmiä. Yhden kentät ohjaavat autoa kohti lyhintä reittiä, toisen pyrkivät säilyttämään sen ajolinjan mahdollisimman loivana ja kolmannen pitävät sen tiellä. Lyhimmän reitin menetelmässä kaukaisimpien kenttien puoleensa vetävät voimat asetetaan suurimmiksi. Agentti ohjaa auton suuntaan, jossa se havaitsee voimakkaimman vetävän voiman. Se kulkee siis kauimmaksi kantaneen sensorin suuntaan, aivan kuten säteenseurantamenetelmässä. Tästä voidaan päätellä, että myös säteenseuranta pyrkii ohjaamaan agenttia pitkin lyhintä reittiä.

Agentti voi määrittää tavoitenopeutensa radan geometrian perusteella reaaliaikaisesti. Näin tekee esimerkiksi *Tutor* [51]. Tämä ei ole optimaalinen tapa, koska kaarteen geometrinen säde on pienempi kuin sitä kulkevan klassisen ajolinjan [1]. Toisaalta kaarteisiin sopivat nopeudet voi olla laskettu ennalta [26]. Quadfliegin ryhmän [33] TORCS-botti muodostaa radasta etukäteen mallin, jota se noudattaa ajaessaan. Malli jakaa radan erilaisiin osuuksiin, kuten suoriin, loiviin kaarteisiin ja neulansilmiin. Mallin muodostamisen jälkeen botti oppii evoluutiolaskennan avulla, minkä suuruinen nopeus sopii minkäkin tyyppiselle osuudelle. Quadflieg et al. havaitsivat bottinsa pystyvän kilpailemaan vuoden 2009 *Simulated Car Racing Championship* -kilpailun nopeimpien kuljettajien kanssa. Yksi näistä oli *Cobostar* (ks. alla). Kuitenkaan ryhmän botti ei pysynyt ihmispelaajan vauhdissa kuin hyvin yksinkertaisilla radoilla.

Butzin ja Lönnekerin [3] TORCSiin ohjelmoima *Cobostar*-botti määrittää säteenseurannan avulla myös nopeutensa. Se laskee tavoitenopeutensa v_t kaavalla

$$
\begin{aligned}
v_t &= max(p_7, v'_t) \\
v'_t &= \begin{cases} x & \text{jos } d < \theta_2 \\ 100 & \text{jos } d \geq \theta_2 \end{cases} \\
x &= p_1 + p_2 d + p_3 max \left\{ 0, \frac{d - \theta_1}{\theta_2 - \theta_1} \right\}^{p_4} - p_5 (\frac{\alpha - 9}{9})^{p_6},
\end{aligned}
$$

missä d on pisimmän säteen kantomatka ja α sen kulma. Parametrien p_i ja θ_i arvot on määritettävä etukäteen. Butz ja Lönneker optimoivat ne *CMA-ES*-menetelmällä (engl. *Covariance Matrix Adaptation Evolution Strategy*). He kertovat tavoitenopeuden kaavasta: *"Yhtälö koostuu vakiosta sekä lineaarisesta ja polynomisesta termistä, jotka riippuvat säteen kantomatkasta. Lisäksi siinä on vähentävä tekijä, joka riippuu säteen kulmasta."*[2] Parametri p_7 on minimitavoitenopeus. Butz ja Lönneker havaitsivat, että

[2]In essence, the equation determines the target speed out of a constant, linear, and polynomial

tietylle radalle optimaaliset parametrit eivät ole yleistettävissä. Tästä syystä he tallensivat *Cobostarille* parametrit, jotka antoivat keskimäärin parhaan kierrosajan eri radoilla.

Vaihteen oikea-aikainen vaihtaminen on monimutkainen optimointitehtävä, jolla ei saavuteta suuria etuja suorituskyvyssä [2]. Optimointi perustuu siihen, että vaihdetta vaihdetaan silloin, kun moottori tarjoaa suurempaa vääntöä seuraavalla tai edellisellä vaihteella. Moottorin suorituskyky ei ole tasainen, vaan vääntö vaihtelee käyntinopeuden mukaan [31]. Optimoinnissa tulisi huomioida myös latenssi eli aika, joka kuluu vaihteiden vaihtamiseen. Braghin et al. [2] käyttävät yksinkertaista heuristiikkaa. Se vaihtaa korkeamman vaihteen, kun moottorin käyntinopeus ylittää maksimiväännön $rpm_{i_{max}}$. Menetelmä vaihtaa alemman vaihteen, jos käyntinopeus laskee alle rajan $rpm_{i_{max}} \cdot \frac{\tau_{i-1}}{\tau_i}$. Kaavassa τ_{i-1} on seuraavaksi alemman, ja τ_i tämänhetkisen vaihteen välityssuhde. Braghin et al. mukaan kytkimen käyttöä ei tarvitse optimoida. Sitä tulee käyttää aina mahdollisimman nopeasti.

Agentti voi myös oppia ajamaan ihmispelaajaa seuraamalla. Tällöin pelin kehittäjä ei ohjelmoi agentille monimutkaisia sääntöjä, vaan se muodostaa ne koneoppimisella itse [21]. Vaikka agentti olisi oppinut ajamaan pelaajia seuraamalla, se ei välttämättä aja kuten ihminen [28]. Umarov ja Mozgovoy [48] uskovat, että agentin opettaminen pelaajadatan avulla tulee yleistymään kaupallisissa peleissä. Heidän mukaansa opettamalla on mahdollista toteuttaa uskottavia ja suorituskykyisiä agentteja. Esimerkiksi *Forza Motorsport 5*-pelin *Driveatar*-järjestelmä kerää pelaajista tietoja pilveen. Jokaisesta pelaajasta muodostetaan profiili, joka kertoo muun muassa kuinka usein tämä leikkaa kaarteita ja osuu toisiin autoihin [23]. Peli pyrkii lataamaan agentille pilvestä profiilin, jolla se on taidoiltaan samanveroinen kuin pelaaja. Toisaalta pelaaja voi eri asetuksilla säädellä agenttien tasoa. Myös jo vuonna 2000 ilmestyneen *Colin McRae Rally 2.0*-pelin agentit oppivat ajamaan pelaajaa seuraamalla [9]. Sen kohdalla agentit harjoitettiin ennen pelin julkaisua. Ne oppivat ohjaamaan autoa neuroverkon avulla, mutta muut niiden tekniikat, kuten ohittaminen olivat ohjelmoituja.

component, which depend on the distance measure and as an additional subtraction component, which takes the measured angle into account. (sic)

5.2 Ajolinjan laskeminen ja noudattaminen

Tässä luvussa kerrotaan, miten agentin ajolinjat muodostetaan ja esitetään autopelissä. Ajolinja voidaan muodostaa etukäteen asiantuntijoiden avulla [4] tai laskennallisesti optimoimalla [2]. Toisaalta agentti voi määrittää linjansa reaaliaikaisesti [26]. Luvussa kuvataan lisäksi joitakin tekniikoita, joilla agentti voi noudattaa ajolinjaa.

Autopelin ajolinja on pohjimmiltaan tietokoneen muistiin tallennettu spline-viiva, joka muodostaa suljetun silmukan [44]. Spline on sulava ja jatkuva viiva, joka lasketaan interpoloimalla [35]. Ajolinjaa kuvaavaa spline-viivaa rajaa kaksi tekijää: sen on pysyttävä radan sisällä ja mukailtava klassista ajolinjaa [44]. Ajolinja voi olla joko puhtaan geometrinen tai dynaaminen. Jälkimmäinen huomio radan muotojen lisäksi ajoneuvon ominaisuudet. Se on lähempänä optimaalista kuin geometrinen. [4] Optimaalisen ajolinjan muodostaminen on kuitenkin NP-täydellinen ongelma [2]. Muñoz et al. [28] kertovat, että agentilla voi olla useampi etukäteen laadittu ajolinja. Sillä voi esimerkiksi varsinaisen ajolinjan lisäksi olla erityinen ohittamiseen tarkoitettu linja.

Braghin et al. [2] mukaan ajolinjan muodostaminen on rajoitettu optimointiongelma, jossa etsitään parasta kompromissia linjan loivuuden ja pituuden välillä. Kuten edellä mainittiin, tämä optimointiongelma on rajoitettu. Casanova [4] kertoo, että ajolinja voidaan määrittää tarkemmin, jos huomioidaan radan geometrian lisäksi dynaamiset seikat, kuten auton jarrujen teho. Täydellinen tarkkuus ajolinjan määrittämisessä vaatisi auton tämänhetkisten säätöjen ja ominaisuuksien huomioimista, sillä esimerkiksi jousituksen kireys ja renkaiden kuluma vaikuttavat sen suorituskykyyn. Näin tarkasti ajolinjaa ei kuitenkaan peleissä voi laskea. Beckman [1] huomauttaa, että parasta ajolinjaa ei voi määrittää käännös kerrallaan, vaan on huomioitava sekä kaarretta edeltävien että seuraavien osuuksien vaikutus. Esimerkiksi suoralla auto on asemoitava niin, että seuraavaa kaarretta lähestytään radan ulkoreunalta. Toisaalta nopein tapa kulkea useampi peräkkäinen kaarre usein poikkeaa siitä, kuinka yksittäinen kaarre ajettaisiin.

Millington ja Funge [26] jakavat ajolinjaa noudattavat agentit ”kiskoilla kulkeviin” ja sellaisiin, jotka seuraavat linjaa aktuaattoreitansa säätelemällä, eli ajamalla. Kiskoilla kulkeva agentti noudattaa ajolinjaa sokeasti. Sen ei välttämättä tarvitse kuljettaa autoa lainkaan, jos ajolinjan yhteyteen on tallennettu myös sille sopivat nopeudet. Kiskot tekevät agentista kuitenkin mukautumiskyvyttömän: se ei osaa ohittaa eikä välttää yhteentörmäyksiä. Agentti voi kuitenkin erikoistilanteissa, ku-

ten kolareissa käyttää erilaisia menetelmiä, joilla se pääsee takaisin kiskoille. Kiskoja noudattavat agentit olivat tavallisia varhaisissa autopeleissä, mutta ne eivät enää sovellu nykyaikaisiin. Tänä päivänä uskottavan agentin on noudatettava ajolinjaansa riittävän joustavasti, jotta se pystyy reagoimaan toisiin kuljettajiin ja pelaajaan sekä toisaalta tekemään tekotyperiä virheitä [44].

Millingtonin ja Fungen [26] mukaan valtaosa nykypelien agenteista ajaa autoa samalla tavalla kuin pelaaja. Niiden aktuaattoreita ovat samat auton hallintalaitteet, joita pelaaja ohjaa syötteillään. Ajamisen tulee olla yhtä haastavaa niin pelaajalle kuin agentille. Kuitenkin agentti noudattaa usein hieman yksinkertaistettua ajomallia. Myös aktuaattoreitaan säätelevät agentit pyrkivät useimmiten noudattaman ennalta laadittua ajolinjaa. Ne eivät kuitenkaan vain kulje spline-viivaa pitkin, vaan seuraavat sitä autoa ohjaamalla ja nopeuttansa säätelemällä. Eräs tapa saada agentti seuramaan ajolinjaa on valita linjalta piste, jota kohti agenti pyrkii ajamaan. Kun auto liikkuu, pistettä siirretään. Luvussa 6.2 selostetaan *Tutor*-botin versio tästä menetelmästä. Myös *berniw* käyttää tätä tekniikkaa. Linjan seuraaminen ei kuitenkaan yksin riitä. Esimerkiksi ensimmäisessä *Gran Turismo*-pelissä agentti yritti jatkaa linjan noudattamista, vaikka toinen kuljettaja oli juuri tönäissyt sitä. Tämä johti usein jarrutuksen myöhästymiseen ja radalta suistumiseen. Linjan noudattamisen lisäksi agentti tarvitsee siis muita tekniikoita muun muassa erikoistilanteita varten.

Agentti voi myös suunnitella ajolinjansa samalla, kun se ajaa. Esimerkiksi *Berniw* sovittaa radalle spline-viivoja reaaliaikaisesti. Ensin se jakaa radan lukuisiin pieniin segmentteihin. Sitten kultakin segmentiltä etsitään piste, joka on toisaalta riittävän lähellä edellistä ja toisaalta riittävän pienessä kulmassa siihen nähden. Toisin sanoen pisteille p ja q tulee pääteä ehto $|\frac{p_y - q_y}{p_x - q_x}| \leq \alpha$, missä α on ohjelmoijan asettama raja kulman jyrkkyydelle. Mitä pienempiin segmentteihin rata jaetaan, sitä kaarevampi ja sulavampi ajolinjasta muodostuu. Toisaalta suurempi määrä segmenttejä vaatii enemmän laskentaa. *Berniw* muodostaa spline-viivoja, jotka noudattavat 3. asteen yhtälöä $P(x) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$. Radan pisteiden määrittämisen jälkeen on ratkaistava, millä parametreilla a, b, c ja d funktio $P(x)$ kulkee niiden kautta. [51] Spline-viivan piirtämisen jälkeen *Berniw* pyrkii ajamaan sitä pitkin. Omien testieni perusteella *Berniw* ei kuormita prosessoria sen enempää kuin muutkaan TORCSin botit. Yllä kuvattu reaaliaikainen menetelmä on siis laskennallisesti riittävän kevyt.

5.3 Agentti ja muut kuljettajat

Tässä luvussa kerrotaan, millä tavoin, ja millä tekniikoilla agentin on huomioitava pelin muut kuljettajat. Pääasiassa luvussa kerrotaan siitä, miten agentti voi ohittaa. Toisaalta kolaroinnin välttäminen liittyy kiinteästi ohittamiseen, sillä juuri ohittamisen aikana tilaa on vähän ja törmäämisen vaara suuri [21]. Tässä luvussa ei pohdita, miten agentin tulee reagoida pelaajaan, tai tulisiko sen reagoida tähän eri tavalla kuin toisiin agentteihin. Näitä kysymyksiä käsitellään luvussa 5.4.

Botin on kilpailussa mukauduttava toisten agenttien taktiikoihin, vältettävä yhteentörmäyksiä ja pyrittävä ohittamaan sopivissa kohdissa. Yhteentörmäyksiä kannattaa välttää, koska ne voivat vaurioittaa autoa, tai saada sen suistumaan radalta. Butz ja Lönneker [3] havaitsivat, että auto vaurioituu kisan aikana eniten silloin, kun se kolaroi. TORCSissa toiset agentit voivat kuulua samaan tai eri joukkueeseen [51], mikä vaatii erilaista reagointia. Botin kannattaa päästää oman joukkueen nopeampi kuljettaja ohitseen, mutta puolustaa asemiaan, mikäli toisen joukkueen agentti pyrkii edelle. Toisaalta puolustamisessa on riskinsä, sillä mikään ei takaa, että perässä ajava osaisi ohittaa siististi.

Loiaconon et al. [21] mukaan ohittaminen suorilla ja loivissa kaarteissa vaatii erilaisia heuristiikkoja kuin ohittaminen jyrkissä kaarteissa. Millingtonin ja Fungen [26] mukaan ohittaminen suoralla on melko yksinkertaista. Loiaconon et al. kehittämä botti pyrkii suorilla ajamaan mahdollisimman lähellä edellä kulkevaa. Tämä pienentää ilmanvastusta. Kun botin nopeus kasvaa riittävästi, se pyrkii ohittamaan. Suorilla botin on tärkeintä ohjata autoa oikeaan suuntaan. Ennen ohitusta sen on ajettava samaan suuntaan ohitettavan kanssa ja ohituksen aikana säilytettävä riittävä etäisyys sekä toiseen autoon että radan reunaan. Millingtonin ja Fungen mukaan agentti voi kaarteissa pyrkiä ohittamaan muuttamalla ajolinjaansa. Ohittamislinja voidaan muodostaa etukäteen, tai agentti voi laskea sen tosiaikaisesti. Millington ja Funge eivät kuitenkaan tunteneet yhtään vuoteen 2009 mennessä julkaistua agenttia, joka määrittäisi ohittamislinjansa reaaliaikaisesti.

Loiaconon et al. [21] mukaan agentin on helpointa ohittaa kaarteissa myöhäistämällä jarrutustaan siten, että se pääsee toisen kuljettajan edelle. Ryhmä toteutti botin, joka opetteli Q-oppimisen (ks. alla) avulla, miten jarrutusta tulisi myöhäistää. Heidän bottinsa ei käyttänyt erityistä ohittamisajolinjaa. Ryhmälle selvisi, että ohittaminen onnistuu sitä todennäköisemmin, mitä myöhemmin botti jarruttaa toiseen kuljettajaan nähden. Toisaalta jos jarrutusta myöhäistää liikaa, botti suistuu radalta. Loiacono et al. havaitsivat, että oikean myöhäistämisen arviointi on haas-

tavaa: täysin satunnaisella myöhäistämisellä botti onnistui vain 0,96 % ohituksista. Q-oppiminen (engl. *Q-learning*) on laskennallisen älykkyyden (engl. *computational intelligence*) menetelmä. Siinä agentti ikään kuin ehdollistuu toimimaan oikealla tavalla. Q-oppimisessa botti saa toiminnastaan jonkinlaisen palkkion. Ohituskäyttäytymisen palkkiota kuvaa funktio:

$$f(n) = \begin{cases} -1 & \text{jos auto suistuu radalta tai törmää ohitettavaan} \\ 1 & \text{jos ohitus onnistuu} \\ 0 & \text{muussa tapauksessa} \end{cases}$$

Agentti oppii palkkiofunktion avulla oikeat arvot aktuaattoreilleen, kun ohitustilannetta toistetaan riittävän monta kertaa. Botin pitää oppimisessaan toisaalta suosia toimia, jotka tuottavat positiivisen palkkion, ja toisaalta kokeilla uusia vaihtoehtoja löytääkseen vielä parempia. Loiaconon et al. botti etsii myöhäistämiselle arvoja, jotka johtavat onnistuneeseen ohitukseen mahdollisimman usein. Yleensä Q-oppiminen etenee siten, että aluksi painotetaan vahvasti uusien vaihtoehtojen kokeilemista, ja lopuksi tähän mennessä löydetyt arvot saavat suuremman painoarvon.

Loiacono et al. [21] vertasivat bottiansa *berniwin* suoriutumiseen. He pitivät sitä yhtenä parhaista vuoteen 2010 mennessä julkaistuista TORCS-boteista. Ryhmän kehittämä botti suoritui 10 000 toiston jälkeen ohituksista *berniw*iä paremmin sekä loivilla että jyrkillä rataosuuksilla. Se kuitenkin epäonnistui jyrkissä kaarteissa vielä 5,2 % todennäköisyydellä. Ohituskäyttäytymisen oppiminen vaatii paljon laskentaa. Toisaalta kun agentti on kerran oppinut ohittamisen, se osaa sen jatkossakin. Periaatteessa agentista on mahdollista jaella TORCSin yhteydessä versiota, joka on valmiiksi opetettu. Kuitenkin jos tulevassa versiossa vaikkapa pelin tapa mallintaa renkaan pitoa muuttuisi, botti olisi opetettava uudelleen.

Loiacono et al. [21] havaitsivat, että Q-oppimisella kehitettyyn käyttäytymiseen saattoi vielä soveltaa online-oppimista. He muuttivat bottiansa siten, että se mukauttaa jarrutustaan renkaan muuttuvan kitkakertoimen mukaan. Renkaiden kitkakerroin alenee kisan aikana, minkä vuoksi jarrutusta on aikaistettava. Onlinemenetelmä huomioi ainoastaan kulumisen vaikutuksen kertoimeen — ei muita tekijöitä. Ryhmän mukaan kitkakertoimen muutokseen mukautuva agentti suoriutuu paremmin kuin sellainen, joka noudattaa pelkästään staattisia, ennalta opittua käyttäytymistä. Artikkelista ei kuitenkaan selviä, kuinka vaativa tämä onlinemenetelmä on laskennallisesti.

Tutor pyrkii sekä välttämään yhteentörmäyksiä että ohittamaan hitaampia kul-

jettajia. Se ei kuitenkaan hallitse samantyyppisiä edistyneempiä ohitustaktiikoita kuin Loiaconon et al. [21] ohjelmoima botti. *Tutor* päättelee yhdellä ja samalla menetelmällä, mitä botteja sen tulee ohittaa, ja mitä väistää. Se tarkkailee vain niitä kuljettajia, jotka ovat vähemmän kuin 200 m edellä tai 50 m takana. Etäisyys lasketaan *Tutorin* auton keskipisteestä vastustajan keskipisteeseen. Tämä ei Wymannin [51] mukaan anna tarkkaa etäisyyttä. Hän huomauttaa, että etäisyyden laskeminen autojen kaikkien neljän kulman välillä antaisi paremman arvion. Tämä tosin vaatisi 16 laskua yhden sijaan. [51]

Tutorin menetelmä toimii kahdessa vaiheessa: Ensimmäisessä toiset kuljettajat merkitään joko lipulla `OPP_FRONT`, `OPP_BACK`, `OPP_SIDE` tai `OPP_COLL`. Merkinnät tarkoittavat: vastustaja on edellä ja hitaampi, takana ja nopeampi, lähellä sivusuunnassa sekä välitöntä kolarivaaraa. Seuraavaksi *Tutor* muokkaa aktuaattoreidensa arvoja eri funktiolla sen mukaan, millä lipulla vastustajia on merkitty. Funktio `filterBColl` saa botin jarruttamaan, jos `OPP_COLL`-lipulla merkitty kilpailiija kulkee liian lähellä. Funktio `filterSColl` ohjaa *Tutoria* pois päin vastustajasta, jos se on vaarassa törmätä siihen. Funktio `getOvertakeOffset()` puolestaan ohjaa autoa sivuun sen verran, että se mahtuu edellä ajavan ohi. Se lisää kiintopisteheuristiikan antamaan pisteeseen (ks. luku 6.2) poikkeaman (engl. *offset*), joka muuttaa hieman aiottua kulkusuuntaa. [51]

5.4 Uskottavuus ja tekotyperyys autopeleissä

Autopelin agentin uskottavuus riippuu eniten siitä, kuinka realistisesti se liikkuu eli kuljettaa autoa [26]. Muñoz et al. [28] mukaan uskottava agentti ajaa kuin ihmispelaaja. Tämä ei ole helppo määritelmä, koska pelaajien ajotaidot vaihtelevat. Jos esimerkiksi aloittelija näkee taitavaa pelaajaa simuloivan agentin suorittavan manööverejä, joihin hän ei itse pysty, hän voi pitää sen kykyjä yli-inhimillisinä. Toisaalta taitava pelaaja voi erehtyä pitämään aloittelijaa huonosti ohjelmoituna agenttina. Muñoz et al. esittävät, että uskottavan agentin on joko pyrittävä ajamaan kuin keskivertopelaaja, tai mukauduttava eri pelaajien taitoihin.

Muñoz et al. [28] toteavat, että agentti ei ole uskottava, jos se ei hallitse autopelin perustaitoja. Heidän mielestään näitä ovat:

- Optimaalisen ajolinjan noudattaminen.
- Yhteentörmäysten välttäminen.

- Kyky ohittaa hitaampia vastustajia.
- Kyky estää toisia ohittamasta.
- Radalta suistumisesta ja muista erikoistilanteista selviäminen.

Muñoz et al. huomauttavat, että nämä kriteerit pätevät realistiseen autopeliin, jossa ajetaan päällystetyillä kilparadoilla. Togelius et al. [41] puolestaan kertovat seuraavien tekijöiden vaikuttavan autopelien viihdyttävyyteen:

- Vauhdin hurma: pelissä pitää saada ajaa kovaa.
- Riittävä haaste: vauhti ei yksin riitä, vaan radoilla tulee olla myös haastavia kaarteita.
- Sopiva haaste: pelaajat eivät pidä radalta suistumisesta tai yhteentörmäyksistä.
- Haasteen vaihtelevuus: haaste ei saa pysyä samantasoisena kaiken aikaa.

Tarkemmin ryhmä tutki, mikä tekee autopelin radoista hauskoja. Kosterin [41] mukaan viihdyttävä peli on sellainen, jossa pelaaja oppii jatkuvasti. Togelius et al. tulkitsevat tämän tarkoittavan, että paras rata on sellainen, joka on aluksi haastava, mutta jolla pelaajan suoritus paranee jatkuvasti.

Mielestäni Togeliuksen et al. [41] kriteerejä voidaan soveltaa myös agentin viihdyttävyyden arviointiin. On vaikea sanoa, miten agentit voivat vaikuttaa siihen, että pelaaja saa ajaa radalla lujaa. Ainakin ne voivat väistyä nopeamman pelaajan edestä, jotteivät hidastaisi tätä. Toisaalta agentin uskottavuus voi kärsiä, jos se ei osaa puolustaa sijoitustaan kilpailussa [28]. Kompromissinä pelaaja selvästi hitaammat kilpailijat voisivat ”herrasmiesmäisesti” väistää, kun taas muut pyrkisivät säilyttämään asemansa. Sen lisäksi, että pelaajat eivät pidä yhteentörmäyksistä, ne haittaavat vauhdikasta ajamista. Pelaajaa nopeampien agenttien tuleekin ohittaa siten, että ne eivät osu pelaajan autoon tai haittaa tämän etenemistä. Toisaalta pienet tönäisyt saattaavat viestiä aggressiivisuudesta ja tehdä kisasta jännittävän. Ainakin toimintapelin kohdalla pelaajat pitävät aggressiivisuutta merkkinä inhimillisyydestä [11]. Tulkitsen Muñozin et al. [28] tarkoittavan, että uskottava agentti osaa ohittaa tilaanteesta riippumatta. Ei riitä, että se pääsee pelaajan edelle vain suorilla. Kuitenkin jyrkissä kaarteissa ohittaminen on haastavaa [21]. Loiaconon et al. [21] taktiikka kaarteissa ohittamiseen sisältää aina yhteentörmäyksen riskin (ks. luku 5.3). Toisaalta

tällainen ohitus voi onnistuessaan antaa vaikuttavan kuvan agentin taidoista. Kompromissina tekotyperä agentti voisi yrittää kaarteissa ohittamista harvemmin kuin on taktisesti edullista, jotta se törmäisi pelaajaan mahdollisimman harvoin.

Asemien puolustaminen ja pelaajan väistäminen oikeissa tilanteissa liittyvät sopivan haasteen tarjoamiseen. Kuten luvussa 2.3 todetaan, oikean tasoinen haaste on olennainen osa pelin viihdyttävyyttä ja agentin uskottavuutta. Westin [50] mukaan pelaaja ei odota autopelin agenteilta ihmisen veroisia taitoja, koska hänellä on vastassaan useampi kilpailija samanaikaisesti. Toisaalta Muñozin ryhmän [28] mielestä agenttien tulee pelata kuin ihminen. Umarov ja Mozgovoy [48] esittävät, ettei ihmispelaajan simuloiminen välttämättä tarkoita, että agentilla on parhaan pelaajan taidot. Tosin heikkotasoista pelaajaa simuloiva agentti ei välttämättä sovi kokeneelle pelaajalle ja päinvastoin. Näiden eriävien mielipiteiden kompromissina saman kisan agentit voisivat olla taidoiltaan eritasoisia. Esimerkiksi vain yksi tai kaksi kilpailijaa voisi ajaa erittäin taitavasti. Tällöin keskinkertainenkin pelaaja pääsisi mukaan kärkkikamppailuun, mutta vaatisi todellista taitoa voittaa kilpailu. Toisaalta autopelin haaste ei muodostu yksin sen agenttien taidoista, vaan myös ajomallin monimutkaisuudesta [14, 28, 32]. Lisäksi pelaajan tulee mukautua esimerkiksi erilaisten autojen tuntumaan ja muuttuviin sääolosuhteisiin.

Haasteen variointi autopelissä voi tarkoittaa sitä, että agentit mukautuvat pelaajaan tämän taitojen karttuessa. Toisaalta pelaajat eivät pidä siitä, jos he eivät kehittyessään ala menestyä kilpailuissa paremmin [28]. Agenttien ei kuitenkaan välttämättä tarvitse mukautua pelaajan taitoihin kovin nopeasti. Monissa autopeleissä simuloidaan kokonaisia kilpailukausia ja -sarjoja [14]. Kasvavan haasteen voisi kytkeä näihin siten, että agenttien taidot paranevat, kun pelaaja siirtyy sarjasta tai kaudesta toiseen. Tällöin pelaaja kokisi taitojensa kasvavan, koska hän sijoittuu paremmin kauden lopussa kuin alussa. Toisaalta peli seuraisi Kosterin [41] jatkuvan oppimisen periaatetta, koska pelaaja joutuisi kohtaamaan uuden haasteen seuraavan kauden alkaessa.

Autopelin agenttien haastavuutta voisi varioida myös samaan tapaan kuin West [50] muunteli pokeriagenttejaan (ks. luku 2.4). Toisin sanoen variaatio voisi tulla siitä, kuinka usein agentti tekee tekotyperiä ajovirheitä. Tällainen olisi esimerkiksi huonon ajolinjan valitseminen kaarteessa. Huono ajolinja yhdessäkin kaarteessa voi kasvattaa kierrosaikaa useita kymmenesosasekunteja [1]. Autopelin agentin ei kuitenkaan tulisi tehdä virheitä kiinteällä todennäköisyydellä, vaan mukauttaa sekä niiden lukumäärää että merkittävyyttä sen mukaan, kuinka pelaaja ajaa. Teko-

typerien virheiden tulee olla hienovarainen tapa säilyttää haaste sopivana, kuten Westin toteutuksessa. Pelaaja ei saa huomata, että agentti toheloi jatkuvasti, jotta pelaaja pysyisi sen kannoilla. Haasteen variointi voi olla myös eksplisiittistä. Tällä tarkoitan sitä, että pelaajalle ilmoitetaan tavalla tai toisella, minkälaista haastetta on odotettavissa. Pelit tarjoavat esimerkiksi eri vaikeusasteita, jotka vaikuttavat siihen, kuinka taitavia vastustajat ovat. Näiden avulla pelaaja voi säätää itselleen sopivan haasteen. Toisaalta pelaajan on mahdollista tasoittaa haastetta kytkemällä päälle erilaisia ajoavusteita, kuten automaattivaihteet, luistonesto ja ABS-jarrut [14].

Kuten edellä pohditaan, hyvänä kompromissina autopelin agenteista voi tehdä eritaitoisia. Jimenezin [13] mukaan agenttien taitojen varioiminen on kuitenkin haastavaa. Riittämätön variaatio aiheuttaa tilanteita, jossa pelaaja joutuu ajamaan yksin kaukana muista. Tämä ei ole lainkaan viihdyttävää. Tilanne, jossa pelaaja ajaa yksin voi syntyä jos:

- Pelaaja on selvästi taitavampi kuin agentit. Hän ajaa kisan muiden edellä.
- Agentit ovat selvästi taitavampia kuin pelaaja. Pelaaja kulkee muiden jäljessä.
- Pelaaja jää yksin hitaasti ja nopeasti ajavien agenttien väliin.

Kaksi ensimmäistä kohtaa ovat esimerkkejä tilanteesta, jossa variaatiota ei ole riittävästi. Viimeisessä variaatio ei ole oikein jakautunut.

Jimenez [13] esittää agenttien taitojen varioinnin ratkaisuksi eräänlaista muunnelmaa niin kutsutusta kuminauhatekniikasta. Kuminauhatekniikan tavanomainen toteutus nopeuttaa agentteja, kun ne ajavat pelaajan jäljessä, ja hidastaa, kun ne ajavat edellä. Vaikutus on sitä suurempi, mitä kauempana agentit ovat pelaajasta. [28] Toisaalta menetelmä voi pelkästään nopeuttaa perässä ajavia agentteja [14]. Pelaajat kuitenkin kokevat kuminauhatekniikan helposti turhauttavaksi [28] tai epäreiluksi [13]. Pahimmillaan pelaaja voi kokea agenttien huijaavan, koska ne pysyvät hänen kannoillaan, vaikka tekisivät virheitä [28]. Toisaalta uskottavuus kärsii myös, jos pelaaja huomaa agenttien selvästi hidastelevan hänen vuokseen [13]. Tällöin ajaminen ei ole hauskaa, koska kilpailussa ei ole jännitystä tai mahdollisuutta epäonnistua.

Jimenezin [13] kuminauhatekniikassa agentit mukautuvat pelaajan taitoihin noudattamalla eräänlaista ”kisan käsikirjoitusta”. Tällaisen käsikirjoituksen on oltava kattava. Sen on käsiteltävät tilanteet, jossa pelaaja on huonompi kuin hitain agentti ja nopeampi kuin paras. Käsikirjoituksen on myös oltava riittävän joustava, sillä kisan tapahtumia on vaikea täysin ennakoida. Jimenez antaa esimerkiksi seuraavan

käsikirjoituksen: Oletetaan, että pelaaja aloittaa kisan viimeiseltä sijalta, ja että hän on yhtä taitava kuin parhaat agenteista. Tällöin pelaaja ohittaa hitaimmat kuljettajat pian lähdön jälkeen. Hänen sijoituksensa kohoaa tasaisesti ensimmäiset 70–80 % kisan kokonaiskestosta. Kilpailun lopussa, eli viimeiset 20–30 % kisan kestosta pelaajan tulee kamppailla ensimmäisestä sijasta. Kärkikamppailussa agenttien tulee pelata hävitäkseen. Toisin sanoen niiden tulee varmistaa, että pelaaja voittaa. Niiden on esimerkiksi hidastettava, kun pelaaja tekee virheen, sillä on turhauttavaa hävitä kilpailu viimeisissä kaarteissa tehdyn erheen vuoksi. Kilpailun agenttien päämääränä on tarjota pelaajalle kokemus, että hän on taistellut voitostaan reilusti, ja että hänen tekemistään virheistä ei rankaistu kohtuuttomasti.

Teknisesti yllä kuvattu käsikirjoitus toimii siten, että agentit jaetaan kolmeen ryhmään: hitaaseen, keskinkertaiseen ja nopeaan. Kukin näiden ryhmien agenteista pyrkii seuraamaan tiettyä kiintopistetta, jonka sijainti riippuu sekä pelaajasta että ryhmästä. Mikäli agentit ovat tavoitteestaan jäljessä, ne nopeuttavat ajoaan. Jos ne ovat edellä, ne hidastavat. Tekniikassa on kuitenkin selkeä ero tavanomaiseen kuminauhatekniikkaan nähden: agentit eivät hidasta heti, kun pääsevät pelaajan edelle, vaan voivat jatkaa karkumatkaansa, jos ne eivät ole vielä saavuttaneet kiintopistettään. Vastaavasti ne voivat jatkaa hidastamista, vaikka olisivat jääneet jo jälkeen. Lisäksi agentit eivät hidastu ja nopeudu mielivaltaisesti, vaan muutoksilla on etukäteen asetetetut rajat. On siis mahdollista, että agentti jää pysyvästi kiintopisteestään jälkeen tai päinvastoin. Kiintopisteet myös siirtyvät kilpailun aikana. Aluksi nopeimman ryhmän ajajat pyrkivät 500 m pelaajan edelle, kun taas kisan lopussa ne pysyttelevät mahdollisimman lähellä pelaajaa. Tämä saa aikaan sen, että käsikirjoituksen mukaisesti pelaaja joutuu taistelemaan tiensä kärkikamppailuun, mutta saavutettuaan nopeimman ryhmän hänellä on hyvät mahdollisuudet voittaa kilpailu. Lisäksi muutama agentti ei kuulu kiinteästi mihinkään ryhmään. Ne aloittavat joukon hänniltä kilpailun alussa, ja seuraavat pelaajaa tämän edetessä ryhmästä toiseen. Näiden agenttien tarkoituksena on ehkäistä tilanteita, joissa pelaaja joutuu ajamaan yksin. [13]

Kuten luvun 5 alussa todetaan, agentit ovat autopeleissä taktisen vastustajan roolissa [16]. Laird ja van Lent [16] ehdottavat, että taktiset vastustajat olisivat uskottavampia, jos ne ilmentäisivät toimissaan jonkinlaista persoonallisuutta. Jimenez [13] nimittää persoonallisuudeksi sellaisia agentin varioitavia ominaisuuksia, jotka eivät vaikuta sen suorituskykyyn. Hänen toteutuksessaan eräs näistä oli agentin taipumus kiihdyttää kaarteissa niin voimakkaasti, että auto joko ali- tai yliohjau-

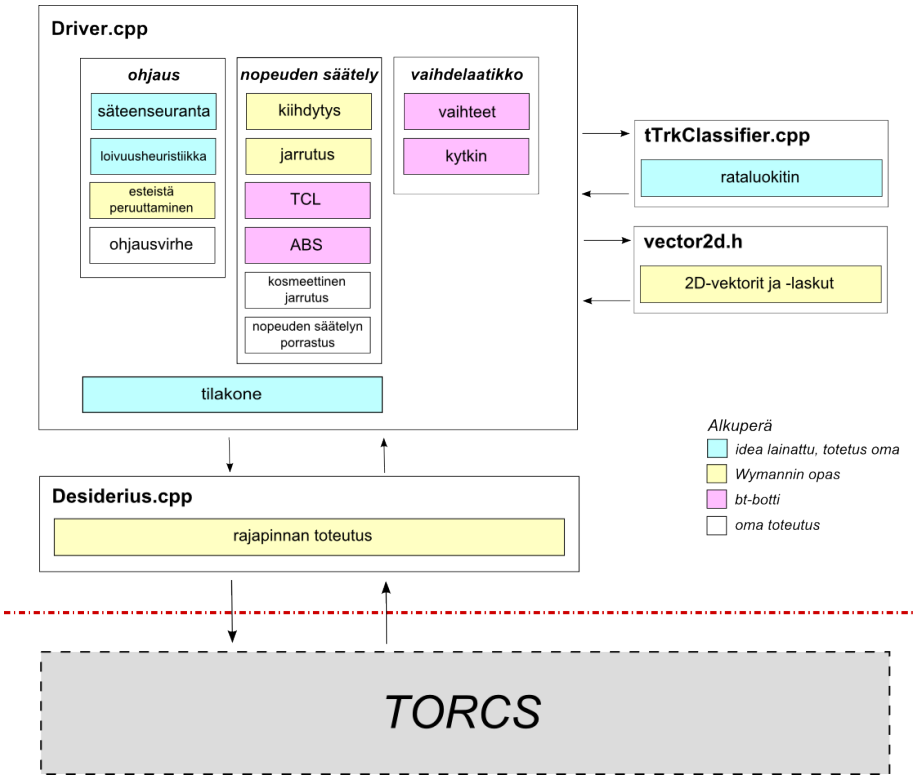
tuu. Tätä Jimenez nimitti aggressiviisuudeksi. Agentin voi kuitenkin autopelissä olla vaikea ilmaista persoonallisuuttaan. Pelaaja ei esimerkiksi voi kuulla sen puhetta, tai nähdä sen eleitä kuten toimintapelissä [18].

Lairdin ja van Lentin [16] mielestä taktisella vastustajalla tulee persoonallisuuden lisäksi olla muun muassa inhimilliset reaktioajat ja sensorit. Autopelissä reaktionopeuden hidastaminen voi tarkoittaa esimerkiksi sitä, että agentti tekee hallitsemattoman väistöliikkeen tai törmää pelaajaan, jos tämä ajaa sen tielle odottamatta. Reaktioajat voisivat ilmetä myös siten, että agentti ei käsittele auton hallintalaitteita mielivaltaisen nopeasti. Esimerkiksi jarruvalot eivät saa vilkkua jatkuvasti päälle ja pois, sillä tosielämässä polkimen painallus vie kuljettajalta vähintään satoja millisekunteja [2]. Realistiset sensorit voivat tarkoittaa esimerkiksi sitä, että agentti ei havaitse toisia kuljettajia kuin tietyn matkan päästä. Tämä voisi ilmetä esimerkiksi siten, että agentti siirtyy ohittamaan edellä ajavaa vasta, kun havaitsee tämän. Toisaalta agentin ei tule nähdä autoja esteiden takana.

Tässä luvussa on jo sivuttu tekotyperyyttä autopeleissä. Esimerkiksi agenttien hidastaminen, kun ne pääsevät pelaajan edelle on eräs tekotyperyyden muoto. Edellä pohdittiin myös mahdollisuutta soveltaa autopeleihin Westin [50] ajatusta, jossa agentin tekevät tarkoituksellisesti virheitä tietyllä todennäköisyydellä. Autopelissä agentilla on useimmiten vastassaan sekä pelaaja että muita agentteja. Tekotyperä agentti voisi hienovaraisesti suosia pelaajaa monella eri tapaa. Se voisi päästää pelaajan ohi useammin kuin muut kuljettajat, ohittaa tätä harvemmin ja varoa yhteentörmäyksiä hänen kanssaan enemmän. Tekotyperyys voisi ilmetä myös siten, että agentti ei noudata ajolinjaansa täydellisesti, vaan sen kulussa esiintyy variaatiota. Se voisi aika ajoin tehdä myös selkeitä virheitä ja suistua radalta. Virheiden tulee olla myös monipuolisia: agentti voi esimerkiksi valita väärän ajolinjan, ohjata liian jyrkästi, jarruttaa liian myöhään tai kiihdyttää niin, että renkaista katoaa pito.

6 Botin toteuttaminen TORCS-peliin

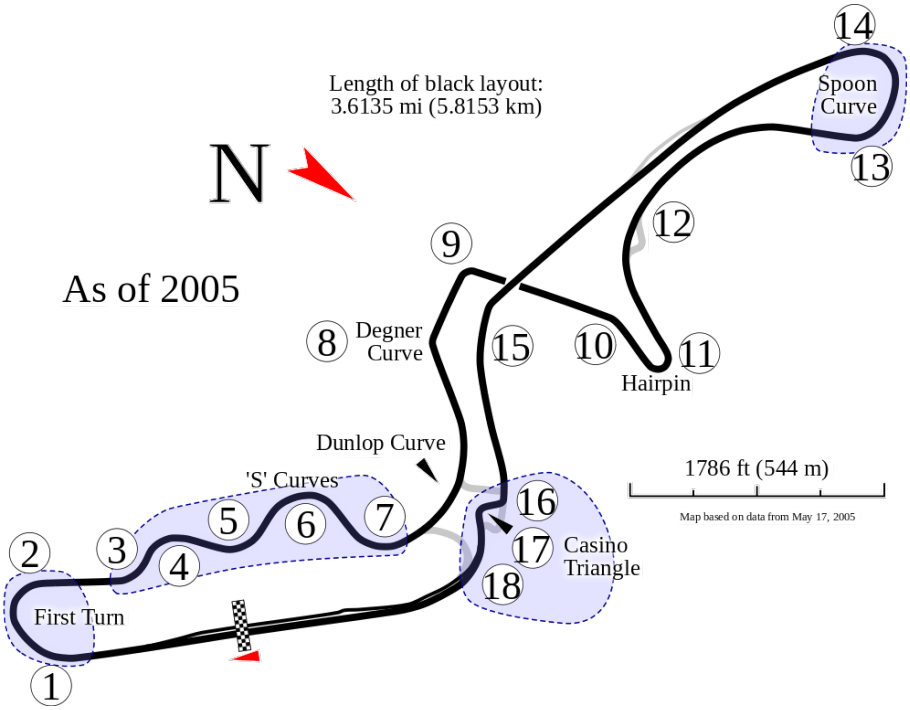
Tutkielmaa varten ohjelmoimani botti on nimeltään *Desiderius*. Se on nimetty Erasmus Desiderius Rotterdamilaisen mukaan, joka kirjoitti satiirisen kirjan *Tyhmyyden ylistys*. Tässä luvussa tarkastellaan *Desideriuksen* teknistä toteutusta, sen heuristiikkoja ja kehitysprosessia. Luvussa pyritään käsittelemään eri aktuaattoreihin liittyviä menetelmiä erikseen. TORCSissa botin aktuaattoreita ovat ohjauspyörä, kaasu- ja jarrupoljin, kytkin sekä vaihdelaatikko. Lisäksi botti voi kytkeä auton valot ja pyytää varikkopysähdystä. Aktuaattoreiden vaikutukset riippuvat valitusta ajoneuvosta. Esimerkiksi kaasupolkimen aiheuttama kiihtyvyys riippuu moottorin tehosta.



Kuva 6.1: Desideriuksen toteutuksen arkkitehtuuri.

Desideriuksen toteutus esitellään kuvassa 6.1. Se noudattaa luvussa 4.2 selostettua bottimoduulin rakennetta. Tiedosto `Desiderius.cpp` toteuttaa moduulin rajapinnan, mutta botin varsinainen toteutus on tiedostossa `Driver.cpp`. Tiedostot `tTrkClassifier.cpp` ja `vector2d.h` sisältävät botin käyttämiä apuluokkia.

Niitä käsitellään tarkemmin liitteissä B ja C. Aluksi toteutin Wymannin oppaan [51] menetelmät pääasiassa siinä järjestyksessä kuin ne esitellään. Toteutin kuitenkin erikoistilanteiden menetelmät, kuten esteistä peruuttamisen aikaisemmin kuin oppaassa, koska täten niitä oli helpompi testata. Oppaan läpikäynnin jälkeen ohjelmoin säteenseurantaheuristiikan, tilakoneen ja rataluokittimen. Sitten lainasin ABS-, TCL-, kytkin- ja vaihdeheuristiikat *bt* botin toteutuksesta. Lopuksi lisäsin kosmeettisen jarrutuksen, nopeuden säätelyn porrastuksen ja ohjausvirheen. Viimeiseksi sain loivuusheuristiikan yhdistettyä säteenseurantaan tilakoneen avulla.



Kuva 6.2: Suzaka Circuit, joka on radan `wheel2` tosimaailman vastine. Kuva on Will Pittengerin piirtämä, ja sillä on CC BY-SA lisenssi.

Kuvassa 6.2 on *Suzuka Circuit*, joka on *Desideriuksen* testauksessa käytetyn `wheel2`-radan tosimaailman vastine. Valitsin radan, koska se sisältää monenlaisia osuuksia. Sillä on niin pitkiä suoria kuin loivia ja jyrkkiä kaarteita. Lisäksi kaarre 11 on niin kutsuttu neulansilmä. Myös Quadflieg et al. [33] pitävät rataa monipuolisena. *Suzuka Circuit* on kokonaan päällystetty, joten päätin, että *Desideriuksen* tulee kyetä ajamaan vain asfalttilla. Päällystämättömillä teilla ajaminen vaatii omia tekniikoitaan [28]. Wymann testaa oppaansa [51] bottia radoilla `mixed-2`, `e-track-2`, `e-track-4` ja `g-track-3`. Näistä `mixed-2` on osin päällystämätön. Kun opas julkaistiin vuonna 2004, TORCS ei vielä sisältänyt `wheel2`-rataa.

6.1 Kehitystyö, -ympäristö ja -työkalut

Tässä luvussa kerrotaan *Desiderius*-projektin kulusta, sekä botin kehityksessä käytetyistä työkaluista. Luvussa pohditaan myös kehitystyön haasteita ja esitetään joitakin ajatuksia siitä, miten sitä voisi helpottaa. Lisäksi analysoidaan Wymannin opasta [51], jonka avulla ohjelmoin bottini. Oppaan esittelemää bottia kutsutaan tässä *Tutoriksi*.

Ohjelmoin *Desideriuksen* C++-kielillä, jolla muutkin TORCSin boteista on toteutettu. Jotkin osat niistä on ohjelmoitu enemmän C- kuin C++-kielen tyylillä. En käyttänyt toteutuksessani uudempien C++-standardien kuten C++11:en ominaisuuksia. En myöskään testannut niiden yhteensopivuutta TORCSin perinnekoodin (engl. *legacy code*) kanssa. Käytin botin kehityksessä pelin Linux-versiota 1.3.4, joka julkaistiin 19.10.2012 [45]. Sittemmin pelistä on julkaistu uudempia versioita. Uusin on 1.3.6, joka julkaistiin 24.4.2014. En testannut *Desideriusta* näissä uudemmissa versioissa.

Päätin alkaa kehittää bottia TORCSin Linux-versiolle, koska kaikki ohjeet pelin asentamiseksi ja botin ohjelmoimiseksi on laadittu sitä varten. Vaikka en ole aikaisemmin asentanut Linuxia tai ylläpitänyt Linux-ympäristöä, botin kehittäminen vaikutti silti yksinkertaisemmalta siinä kuin Windowsissa. Esimerkkinä ympäristöjen eroista Windowsille ei ole olemassa skriptiä, joka luo bottimoduulin tarvitsemat tiedostot. Siinä tiedostot pitää kopioida jostakin valmiista toteutuksesta, ja muokata käsin sopiviksi.

Kesti pitkään, ennen kuin sain sekä pelin että oman bottimoduulini toimimaan. Jouduin kokeilemaan useita eri kokoonpanoja. Sopivan yhdistelmän löytäminen vaati kaiken kaikkiaan 13 yritystä, joissa vaihtelin sekä käyttöjärjestelmää että pelin versiota. Päädyin Linux-käyttöjärjestelmän Lubuntu-jakeluun ja TORCSin versioon 1.3.4. Jouduin siinäkin vielä muokkaamaan pelin `make`-tiedostoa, koska se linkitti moduulit väärässä järjestyksessä. Valitsemani kehitysympäristö ei ole täysin ongelmaton, sillä käyttämäni tietokone ajaa peliä varsin hitaasti sen nimellisiin laitteistovaatimuksiin nähden. Se hahmontaa vain 20 – 30 ruutua sekunnissa resoluutiolla 640x480. Videota kaapattaessa peli hidastuu entisestään: ruutu päivittyy noin 10 kertaa sekunnissa. Grafiikkaongelmat saattavat johtua epäyhteensopivasta näytönohjaimen ajurista. Toisaalta samankaltaisia ongelmia esiintyi myös Windows-asennuksissa, jotka käyttävät eri ajuria. Windows 7:ssä grafiikka piirtyi vielä hitaammin kuin Lubuntussa, eikä peli ollut pelattavissa kuin resoluutiolla 320x200. Pelin Windows-versio myös kaatui, jos sitä ei käynnistänyt parametrilla `-s`, mikä

estää multiteksturoinnin. Taulukossa 6.1 on listattu TORCSin minimivaatimukset, ja kehityksessä käytetyn tietokoneen ominaisuudet:

Taulukko 6.1: TORCSin minimivaatimukset ja testikoneen ominaisuudet.

Ominaisuus	Minimi	Testikone
Prosessori	550 MHz	2000 MHz (2 ydintä)
Muisti	256 Mt	2560 Mt
Näytönohjain	64 Mt	256 Mt
Open GL	1.3	1.4

Botin kehittäminen TORCSiin oli hidasta, koska heuristiikkoihin tehtyjä muutoksia oli vaikea arvioida muuten kuin seuraamalla agentin suoriutumista pelissä. Tämä vaati moduulin kääntämistä ja uuden kisan käynnistämistä pelissä. Yksi kierros testiradalla vei hieman yli kaksi minuuttia. Tämä ei sinänsä ole pitkä aika, mutta koska prosessin joutui toistamaan useita kymmeniä kertoja päivässä, ohjelmointi kesti kauan. Koin ohjelmoinnin myös haastavaksi. Muutoksilla oli odottamattomia vaikutuksia botin käyttäytymiseen. Lisäksi havaitsin heuristiikkojen olevan keskenään riippuvaisia. Opin myös, että kun käyttää vain yhtä rataa botin testaamiseen, saattaa tahattomasti luoda *ad hoc*-ratkaisuja. Toisin sanoen sama botti voi ajaa testiradan puhtaasti, mutta tehdä muilla radoilla virheitä.

Kuten luvussa 4.2 kerrotaan, TORCS on mahdollista ajaa tekstitilassa.Tekstitilan tarjoaman yhteenvedon lisäksi botin voi ohjelmoida tulostamaan muita tietoja. *Desideriuksen* toteutuksessa `printUtils`-moduulin proseduurit (ks. liite B) tulostavat tietoja autosta ja radasta tekstitiedostoihin. Projektin aikana testasin yksittäisiä heuristiikkoja ohjelmoimalla ne tulostamaan laskemiaan arvoja konsoliin tai tekstitiedostoon. Vasta todettuani arvot oikeellisiksi, lisäsin heuristiikat osaksi toteutusta.

Tekstitilan avulla voi jossakin määrin arvioida botin suorituskykyä, mutta sen perusteella on vaikea tehdä johtopäätöksiä botin ajamisesta. Tilassa ei voi nähdä, miltä auton kulku näyttää. Esimerkiksi eräs *Tutorin* varhaisimmista versioista yrittää vain pysyä keskellä rataa. Tällä yksinkertaisella ohjausheuristiikalla botti ajaa kierroksen puhtaasti. Tekstitilan perusteella voisi siis luulla, että toteutus on onnistunut, vaikka *Tutorin* ajo näyttää täysin epärealistiselta. Koska pyrkimyksenäni oli toteuttaa uskottava botti, päätin testata bottia yksinomaan seuraamalla sen ajamista pelissä. En uskonut, että pelkkä suorituskyvyn tarkkailu tekstitilassa kertoo riittä-

västi uskottavuudesta. Pelistä voi kaapata videon myös tekstitilassa, mutta videon työstäminen esityskuntoon kestää huomattavasti kauemmin kuin pelin seuraaminen reaaliajassa.

Eräs mahdollisuus kehitystyön helpottamiseksi on toteuttaa apuohjelma, joka visualisoi botin ajosta kerättyä dataa. Ohjelma voisi esimerkiksi piirtää botin kulkemat ajolinjat, merkitä jarrutuskohdat ja näytää tilanteet, joissa auto menetti pitonsa. Kuten edellä kerrotaan, botin voi ohjelmoida tulostamaan tiedostoja pelin suorituksen aikana. Visualisointiin käytettävät tiedot voisi tulostaa esimerkiksi XML-muodossa. Toisaalta graafien tuottamiseen voisi yrittää soveltaa jotakin valmista ohjelmaa, kuten *gnuplotia*[1]. Tällöin pelin data pitäisi saada tulostumaan valitun ohjelman ymmärtämässä muodossa. Telemetriatietojen visualisoinnista saisi luultavasti paremman kuvan botin suorituskyvystä kuin tutkimalla tekstitilan antamaa yhteenvetoa. En kuitenkaan usko, että sen avulla pystyy helposti arvioimaan botin uskottavuutta. Silti räikeät epärealistisuudet, kuten edellä kuvattu *Tutorin* ajotapa voisivat erottua kuvista. Oman visualisointityökalun toteuttamiseen olisi kulunut merkittävästi aikaa. Sen ohjelmoinnissa olisi ratkaistava, mitä suureita visualisoidaan ja millä tarkkuudella, sekä mitä tietoja esitetään yhtä aikaa. Ohjelmasta olisi parasta tehdä sellainen, että nämä seikat ovat siinä säädeltävissä. Toisaalta mitä enemmän säädeltävyyttä visualisointiohjelma tarjoaa, sitä monimutkaisempi se on toteuttaa. Minulla ei ole kokemusta *gnuplotin* käyttämisestä, joten oma aikansa kuluisi myös sen toimintojen opettelemisessa. Kaiken kaikkiaan arvioin, että visualisoinnista saavutettava etu ei korvaa sen toteuttamiseen kuluvaa aikaa ainakaan tässä projektissa.

Wymann esittää oppaassaan joitakin suosituksia koodauskäytänteistä. Hän huomauttaa, että ensinnäkin botin koodin tulee olla mahdollisimman kevyttä. Siinä tulee muun muassa välttää kutsuja raskasta laskentaa vaativiin funktioihin. Wymann nostaa esimerkiksi tällaisesta funktion `RtTrackSideTgAngleL`, joka määrittää radan tangentin kulman ajoneuvon kohdalla. Wymann kehottaa lisäksi tarkistamaan, tarvitseeko kaikkia laskuja suorittaa joka simulaatiojaksolla. Botin koodi tulee myös kirjoittaa siten, että se toimii samalla tavalla eri kääntäjillä ja käyttöjärjestelmillä. Wymann huomauttaa erityisesti yhteensopivuusongelmista Visual C++ 6.0 kääntäjän kanssa, mutta tässä kohtaa on huomattava, että se julkaistiin vuonna 1998.

Wymannin opas on koostettu huolimattomasti. Siinä esimerkiksi neuvotaan lisäämään kutsu funktioon `filterABS`, jonka toteutusta ei esitellä missään kohdassa. Funktiossa `getBColl` käytetään vakioita, joita teoksessa ei esitellä. Myöskään

[1] `http://gnuplot.info/`

siinä suoritettavia laskuja ei selitetä. Ilman fysiikan tuntemusta on haastavaa selvittää, mihin ne perustuvat. Oppaan koodi on teknisesti pätevää, mutta sen selkeyttä, ylläpidettävyyttä ja kommentteja voi vielä parantaa. Wymann esimerkiksi lisää toteutukseen luokan `Driver`, mutta ei hyödynnä mahdollisuutta siirtää muuttujia sen attribuuteiksi. Jos muuttujat olisivat luokan jäseniä, niitä ei tarvitsisi esitellä joka metodin parametreissä uudelleen. Lisäksi jos botin osoittimista tehtäisiin jäseniä, ne voitaisiin suojata asianmukaisesti. Esimerkiksi osoitin `tCarElt* car` voitaisiin sijoittaa jäseneen `const tCarElt* const car`. `TCarElt`-tietueesta tarvitsee muuttaa vain osatietueen `tCarCtrl` arvoja. Sille voisi lisätä osoittimen `const tCarCtrl* actuators`. Attribuutteja lisäämällä voisi myös välttää samojen johdettujen suureiden laskemista uudelleen. Lisäksi esitettyjen asioiden järjestystä oppaassa voi vielä parantaa. Esimerkiksi suositukset koodauskäytänteistä esitetään vasta luvussa 3.

Moni asia on työlästä selvittää yksin TORCSin lähdekoodin perusteella, koska se on niukasti kommentoitua ja dokumentoitua. Myös Wymannin opas jättää kysymyksiä vastaamatta. Esimerkiksi monen muuttujan yksikkö jäi minulle epäselväksi. Yllättäen moottorin käyntinopeus on pelissä tallennettu liukulukuna, joka ilmaisee sen kymmeninä kierroksina. Toisin sanoen 3500 kierrosta minuutissa on siinä 350,0. En myöskään onnistunut oppaan perusteella selvittämään, miten bottimoduulin keskeiset tietorakenteet ja tietueet on toteutettu. Toteutusta hämärtää osin se, että koodissa käytetään paljon C-kielen makroja viitatessa tietueiden alkioihin. Esimerkiksi `tCarElt`-tietueen alkioon `car.priv.wheel[0].radius` viitataan oikopolulla `car._wheelRadius(0)`. Toisaalta ohjelmointi on vaivattovampaa näiden makrojen avulla.

Wymannin oppaan mukana tulee skripti, joka luo bottimoduulin tarvitsemat tiedostot. Sama skripti toimitetaan myös TORCSin version 1.3.4 mukana. Se luo virheellisiä lähdekooditiedostoja. Niistä käännetty moduuli saa pelin kaatumaan. Tämä johtuu siitä, että moduuli ei alustuksessa täytä riittävän montaa `tModInfo`-tietuetta (ks. luku 4.2) Korjasin skriptiä siten, että se kirjoittaa eheätä koodia. Tarjosin korjaustani Wymannille, mutta hän ei ollut siitä kiinnostunut, koska TORCS-projektin versionhallinnassa oli jo paranneltu skripti. En kuitenkaan löytänyt pelin sivuilta minkäänlaista mainintaa skriptin ongelmista saati korjauksesta. Kävin kirjeenvaihdon Wymannin kanssa 28.1.2013.

6.2 Bottioppaan ohjausheuristiikat

Botin tulee kyetä ohjaamaan autoa taitavasti, jotta se pysyisi radalla kovassakin vauhdissa. Tämä luku esittelee Wymannin oppaassa [51] kuvatut ohjausheuristiikat. Niistä ensimmäinen pyrkii yksinkertaisesti pitämään auton keskellä rataa. Sitä kutsutaan jatkossa *keskihakuheuristiikaksi*. Se laskee ohjaussuunnan kaavalla:

$$\alpha = \theta - \beta - d_x / w$$

missä α on auton uusi suunta, θ radan tangentin kulma, β auton tämänhetkinen suunta, d_x auton etäisyys radan keskilinjasta ja w radan leveys metreinä. Kaikki kulmat ovat radiaaneissa. TORCSissa suunta $\alpha < 0$, kun auton keula osoittaa radan tangentista vasemmalle ja $\alpha > 0$, kun oikealle. Etäisyys $d_x < 0$, kun auto on radan keskilinjan vasemmalla puolella, ja $d_x > 0$, kun oikealla. Tekijän d_x / w vuoksi botti kääntyy sitä enemmän, mitä kauempana se on radan keskeltä. Menetelmä pystyy myös ohjamaan suistuneen auton takaisin radalle, kunhan ajoneuvo ei ole juuttunut reunavalliin tai muuhun esteeseen.

Wymannin opas esittelee myös toisen, edistyneemmän ohjausheuristiikan. Siinä auton edelle asetetaan joka simulaatiojaksolla kiintopiste, jota kohden botti pyrkii kulkemaan. Pisteen etäisyys d riippuu auton nopeudesta. Se lasketaan kaavalla $d = m + v_x \cdot w$, missä m on minimietäisyys, v_x auton nopeus ja w nopeuden painokerroin. Oppaassa $m = 17{,}0\,\text{m}$ ja $w = \frac{1}{3}$. Kiintopisteheuristiikka vaatii toimiakseen luokan `V2d`, joka toteuttaa kaksiulotteisen vektorin ja yleisimmät vektorilaskut. Luokan kuvaus on liitteessä B.

Kiintopisteheuristiikka aloittaa etsimällä ratasegmentin, jonka etäisyys autosta on vähintään d m. Segmentiltä etsitään alkupiste $\boldsymbol{s} = (TR\_SL_x + TR\_SR_x, TR\_SL_y + TR\_SR_y)/2$, missä TR_SL ja TR_SR ovat segmentin etureunan vasen ja oikea verteksi. Suorilla kiintopiste $\boldsymbol{t}$ saadaan kaavalla $\boldsymbol{t} = \boldsymbol{s} + \boldsymbol{u} \cdot l$, missä $\boldsymbol{u}$ on suuntavektori ja l segmentin pituus. Suuntavektori $\boldsymbol{u} = (TR\_EL_x - TR\_SL_x, TR\_EL_y - TR\_SL_y)/l$, missä `TR_EL` ja `TR_SL` ovat segmentin vasemman reunan verteksit. En tutkinut, voiko suuntavektorista tulla erilainen, jos käytetään saman segmentin oikean reunan verteksejä vasemman sijaan.

Kaarteissa kiintopiste $\boldsymbol{t}$ lasketaan kaavalla:

$$\boldsymbol{t} = \boldsymbol{c} + \boldsymbol{u}, \quad \boldsymbol{u} = R(\delta)(\boldsymbol{s} - \boldsymbol{c}), \quad R(\delta) = \left[\begin{smallmatrix} \cos\delta & -\sin\delta \\ \sin\delta & \cos\delta \end{smallmatrix}\right]$$

missä $\boldsymbol{c}$ on segmentin keskipiste, $\boldsymbol{u}$ suuntavektori, R rotaatiomatriisi, δ käännöksen kulma radiaaneissa ja $\boldsymbol{s}$ segmentin alkupiste. Alkupiste $\boldsymbol{s}$ lasketaan samalla taval-

la kuin suoralla. Suuntavektori $\boldsymbol{u}$ muodostetaan kiertämällä vektoria $\boldsymbol{s}$ segmentin keskipisteen $\boldsymbol{c}$ ympäri kulman δ verran.

Kun kiintopiste $\boldsymbol{t}$ on etsitty, lasketaan sen suunta autosta. Suunta saadaan kaavalla $\beta = \arctan(\frac{t_y - p_y}{t_x - p_x})$, missä $\boldsymbol{p}$ on auton sijainti. Kulma lasketaan arkustangentin avulla, koska se on määritelty koko reaaliavaruudessa $\mathbb{R}$, ja siten myös välillä $[-\pi,\pi]$. Kun pisteen suunta on laskettu, botti kääntää autoa kulman $\beta - \alpha$ verran, missä α on auton tämänhetkinen suunta.

6.3 Säteenseurantaan perustuva ohjaus

Halusin Wymannin oppaan [51] heuristiikkojen lisäksi kokeilla muita ohjausmenetelmiä. Päätin toteuttaa säteenseurantaan perustuvan ohjauksen sen yksinkertaisuuden vuoksi. TORCSin palvelinversion säteenseurantasensorit on kuvattu luvussa 5.1. Työpöytäversiossa niitä ei ole valmiina, joten toteutin sellaiset itse Kinnaird-Heetherin ja Reynoldsin [22] antaman sanallisen kuvauksen perusteella. Etuna tässä oli se, että saatoin valita säteiden paremetrit: etäisyyden, välin ja lukumäärän. Botilla on työpöytäversiossa myös enemmän tietoa käytettävissään kuin palvelinversiossa. Se voi esimerkiksi lukea radan muodot suoraan pelin tietorakenteista. Sen ei tarvitse päätellä niitä sensoreidensa avulla, kuten Quadfliegin ryhmän [33] botti tekee.

Desideriuksen toteutuksessa säteiden välinen kulma on $\frac{\pi}{36}$, ja ne kulkevat välillä $\left[-\frac{\pi}{3}, \frac{\pi}{3}\right]$. Väli on kapeampi kuin TORCSin palvelinversiossa, koska säteet lähellä ääriasentoja $\pm\frac{\pi}{2}$ saivat botin suistumaan radalta. Tämä johtui luultavasti siitä, että poikittaiset säteet eivät leikannaat rataa lainkaan, kun auto kulki tien reunalla. Tällöin botti päätteli virheellisesti, että sen tulee kääntyä sivulle. Käytin toteutuksessani enemmän säteitä kuin palvelinversiossa, koska toivoin botin ohjaavan näin autoa sulavammin. Käytin säteille ensin kiinteää kantomatkaa kuten palvelinversiossa. Myöhemmin muutin kantomatkan d nopeudesta riippuvaksi kaavalla:

$$d = d_{min} + v_x \cdot w,$$

missä d_{min} on minimikantomatka, v_x on auton nopeus ja w nopeuden painokerroin. Käytin arvoja $d_{min} = 50{,}0\,\text{m}$ ja $w = 2{,}5$. Toteuttamani säteet eivät tarkkaile, kohtaavatko ne toisia autoja.

Algoritmit 1 ja 2 esittävät *Desideriuksen* säteenseurantaheuristiikan pseudokoodina. Heuristiikka jakautuu kahteen vaiheeseen: ennen säteiden leikkauspisteiden

Algoritmi 1 Alusta sädevektorit

$\gamma_0 \Leftarrow -\frac{\pi}{3}$
$\delta \Leftarrow \frac{\pi}{36}$
$d \Leftarrow d_{min} + v_x \cdot w$
$p \Leftarrow autonSijainti$
$\alpha \Leftarrow autonSuunta$
for $i = 0$ to $n - 1$ **do**
  $kulma[i] \Leftarrow \alpha + \gamma_0 + i \cdot \delta$
  $sädevektori \Leftarrow 1{,}0$ //yksikkövektori
  $sädevektori \Leftarrow sädevektori[i].kierrä((0{,}0), kulma[i])$ //kierto
  $sädevektori \Leftarrow sädevektori[i] \cdot d$ //skaalaus
  $sädevektori \Leftarrow sädevektori[i] + p$ //siirto
  $leikkausEtäisyydet[i] \Leftarrow d$
end for

etsimistä muodostetaan säteitä kuvaavat vektorit. Ensimmäistä vaihetta kuvaa algoritmi 1. Siinä säteiden kulmat lasketaan kaavalla $\gamma_i = \gamma_0 + i \cdot \delta + \alpha, \quad \forall i \in [0, n]$, missä γ_0 ensimmäisen säteen kulma, δ säteiden välinen kulma, α auton tämänhetkinen suunta ja n säteiden lukumäärä. Tässä $\gamma_0 = -\frac{\pi}{3}$ ja $\delta = \frac{\pi}{36}$. Kun kulmat γ_i on laskettu, saadaan säteiden vektorit:

$$\boldsymbol{v_i} = R(\gamma_i)\boldsymbol{u} \cdot d + \boldsymbol{p}, \quad R(\gamma_i) = \begin{bmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{bmatrix}$$

missä R on rotaatiomatriisi, $\boldsymbol{u} = (1{,}0)$ yksikkövektori, d säteen kantomatka ja $\boldsymbol{p}$ ajoneuvon sijainti radan koordinaatistossa.

Kun säteiden vektorit on laskettu, selvitetään missä kohdassa kukin säde leikkaa radan reunan (ks. algoritmi 2). Säteenseurantaheuristiikka tutkii rataa segmentti kerrallaan. Se aloittaa auton tämänhetkisestä sijainnista ja pysähtyy, kun segmentin etäisyys ylittää säteen kantomatkan d. Etäisyys lasketaan summaamalla segmenttien pituuksia. Yksinkertaisuuden vuoksi segmentin etäisyys arvioidaan tässä hieman liian suureksi. Tosiallisesti sitä laskettaessa pitää huomioida, että auto on kulkenut jo osan tämänhetkisestä ratasegmentistä. Kunkin segmentin kohdalla tarkistetaan leikkaavatko säteet sen reunoja. Tarkistuksessa käydään läpi vain ne säteet, joiden leikkauspiste ei vielä ole löytynyt. Toistaiseksi toteutus tarkistaa jokaisen säteen kohdalla sekä radan oikean että vasemman reunan. En tuntenut tapaa, jolla jommankumman reunoista voisi rajata tarkastelusta pois. Säteen ja ratasegmentin reunan leikkauspiste löytyy tutkimalla risteävätkö janat $L = \{\boldsymbol{p} + t\boldsymbol{r_i} \mid t \in [0, 1]\}$

Algoritmi 2 Päätä ohjaussuunta

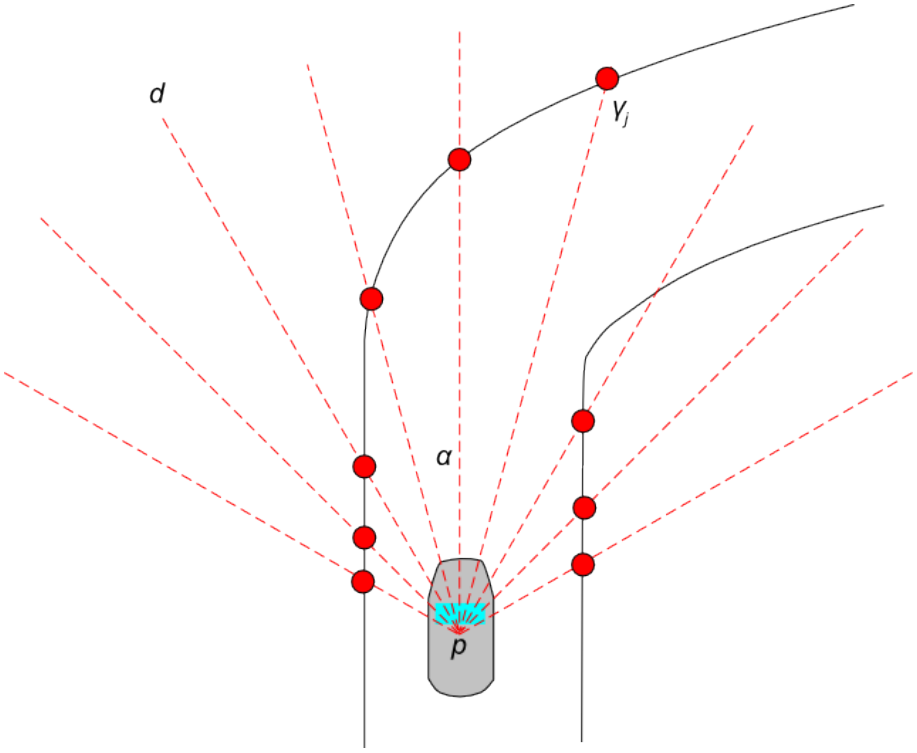
```
segmentti ⇐ nykyinenSegmentti
matka ⇐ 0
while matka < d do
  for i = 0 to n − 1 do
    if leikkausEtäisyydet[i] < d then
      continue //leikkauspiste löydetty
    end if
    l_v ⇐ leikkaaReuna(sädeVektori[i], kulma[i], vasen)
    l_o ⇐ leikkaaReuna(sädeVektori[i], kulma[i], oikea)
    if ||l_v − p|| < leikkausEtäisyydet[i] then
      leikkausEtäisyydet[i] ⇐ ||l_v − p||
    end if
    if ||l_o − p|| < leikkausEtäisyydet[i] then
      leikkausEtäisyydet[i] ⇐ ||l_o − p||
    end if
  end for
  matka ⇐ matka + segmentti.pituus
  segmentti ⇐ segmentti.seuraava
end while
ohjaussuunta ⇐ pisinSäde(leikkausEtäisyydet)
return ohjaussuunta − autonSuunta
```

ja E. Janan L päätepisteet ovat auton sijainti p ja sädevektori $\boldsymbol{r_i}$. Janan E päätepisteet ovat joko segmentin vasemman reunan verteksit `TR_SL` ja `TR_EL` tai oikean `TR_SR` ja `TR_ER`. Janojen leikkauspisteen löytämiseen käytetään Darel Rex Finleyn avoimen lähdekoodin algoritmia[2].

Kun leikkauspisteet on etsitty, niistä valitaan se, jonka etäisyys autosta on suurin. Sitten valitaan sitä vastaavan säteen suunta γ_j. Tämän jälkeen auton suuntaa muutetaan kulman $\gamma_j - \alpha$ verran. Tässä α on auton tämänhetkinen suunta. Säteenseurantamenetelmä ei tee eroa niiden säteiden välillä, joiden leikkauspisteet ovat yhtä kaukana. Yhtä pitkälle kulkeneista säteistä valitaan aina järjestyksessä ensimmäinen. Jos säde ei leikkaa rataa lainkaan, sen leikkauspisteen etäisyys on d. Kokeilin menetelmästä myös versiota, jossa yhtä pitkälle tai lähes yhtä pitkälle kulkeneista

[2] `http://alienryderflex.com/intersect/`

säteistä valittiin se, joka vähiten muutti auton suuntaa. Tämä muutos sai kuitenkin aikaan sen, että botti alkoi tehdä äkillisiä, jyrkkiä ohjausliikkeitä. Pienimmän muutoksen valitseminen viivästytti kääntymistä, mikä vaati lopulta suuria korjauksia kulkusuuntaan.



Kuva 6.3: Säteenseurantaheuristiikka ohjaa autoa pisimmälle kulkeneen säteen γ_j suuntaan.

Butzin ja Lönnekerin [3] *Cobostar*-botti käyttää useamman säteen tietoja määrittäessään ajosuuntansa. Kun se on löytänyt kulman γ_j, se ohjaa auton suuntaan $\beta = \gamma_j - 0{,}5 + \frac{d_j - d_{j-1}}{2d_j - d_{j-1} - d_{j+1}}$. Tässä d_j on kauimmaksi kulkeneen säteen leikkauspisteen etäisyys, ja d_{j-1} sekä d_{j+1} naapuroivien säteiden leikkauspisteiden etäisyydet. Butzin ja Lönnekerin mukaan tämä lisäys heuristiikkaan saa botin kulkemaan tarkemmin suuntaan, jossa on eniten esteetöntä tietä. Heidän testeissään se sai kuitenkin auton heittelehtimään puolelta toisella erityisen leveillä radoilla. Auton kulku pysyi suorana, kun menetelmää muokattiin siten, että termit d_{j-1} ja d_{j+1} kerrotaan painolla $w < 1{,}0$, kun radan leveys on yli 28 m.

Muutin *Desideriusta* siten, että se käyttää useampaa sädettä kuten *Cobostar* [3]. Tällöin se kuitenkin ohjasi autoa jatkuvasti puolelta toiselle. `Wheel2`-radan leveys on 12 m, joten sen ei pitäisi aiheuttaa aaltoilua. Ilmiö saattoi johtua siitä, että vuoron-

perää joko $d_{j-1} > d_{j+1}$ tai päinvastoin. Tämä sai auton kääntymään hieman vasemmalle, sitten taas oikealla, sitten taas vasemmalle ja niin edespäin. Kuitenkin kisan alussa auto on keskellä rataa, jolloin $d_{j-1} = d_{j+1}$. Tästä huolimatta auto alkoi heittelehtiä pian lähdön jälkeen. En onnistunut selvittämään, mistä ongelma johtui, joten palautin *Desideriuksen* säteenseurannan ennalleen.

6.4 Loivuusheuristiikka

Desiderius käyttää yhdessä säteenseurannan kanssa heuristiikkaa, joka ohjaa auton radan ulkolaidalle ennen seuraavaa kaarretta. Menetelmää nimitetään teoksessa *loivuusheuristiikaksi*. Lisäsin loivuusheuristiikan toteutukseen, koska uskoin sen tekevän botin ajamisesta uskottavamman näköistä.

Algoritmi 3 Loivuusheuristiikka

$m \Leftarrow \frac{w_t}{2} - \frac{3}{4}w_c$
$d \Leftarrow etäisyysKeskeltä$
if $|d| \geq m$ **then**
  return 0,0
end if
if seuraava kaarre vasemmalle **then**
  $x \Leftarrow -m - d$
else
  $x \Leftarrow m - d$
end if
$\beta \Leftarrow \frac{\pi}{120}$
$\delta \Leftarrow \beta \cdot \frac{x}{w_t - 2m}$
return δ

Algoritmi 3 esittää loivuusheuristiikan pseudokoodina. Se palauttaa suunnan δ, joka ohjaa autoa radan ulkolaidalle. Auton suunnaksi asetetaan $\theta + \delta$, missä θ on radan tangentti. Pseudokoodissa m on marginaali, joka estää autoa ajautumasta liian reunalle. Kulma β vaikuttaa siihen, kuinka jyrkästi auto siirtyy ulkolaidalle. Vakio w_t on radan ja w_c auton leveys. Muuttuja x ilmaisee auton etäisyyden radan ulkolaidalta. Halusin jättää radan reunalle $\frac{3}{4}w_c$ marginaalin, koska pienemmällä varalla auton renkaat kävivät liian usein tien ulkopuolella. Toisaalta jos marginaalia kasvattaa tästä edelleen, voi auto jäädä liian keskelle tietä.

Havaitsin, että loivuusheuristiikkaa voi soveltaa vain riittävän pitkillä suorilla. Tämä johtuu siitä, että botin on ehdittävä vaihtamaan ohjausmenetelmä säteenseurantaan ennen seuraavaa kaarretta. Vaihtamiseen kuluu aikansa, ja lyhyillä suorilla se myöhäistää kääntymistä liikaa. Nyt loivuusheuristiikkaa käytetään, kun suora on vähintään 60 metriä pitkä. Toinen syy vähimmäispituuden asettamiselle oli se, että erittäin lyhyet suorat saivat botin muuttamaan suuntaansa hyvin äkillisesti. Tämä johtui siitä, että botti vaihtoi nopeasti säteenseurannasta loivuusheuristiikkaan ja takaisin. Näin tapahtui esimerkiksi testiradan ensimmäisen ja toisen kaarteen välissä.

Loivuusheuristiikka ei vaikuta siihen, kuinka nopeasti *Desiderius* ajaa kaarteissa. Botti ei osaa hyödyntää menetelmän mahdollistamaa loivempaa ajolinjaa ja korkeampaa kaarrenopeutta. *Desiderius* määrittää vauhtinsa edelleen kaarteen geometristen ominaisuuksien perusteella. Loivuusheuristiikan tarkoituksena onkin ainoastaan parantaa botin uskottavuutta — ei sen suorituskykyä. Ulkolaidalle hakeutuminen luo illuusion siitä, että botti osaisi ajaa klassisen ajolinjan mukaan. Silti menetelmä vaikuttaa testien perusteella hieman kierrosaikoihin. Tämä johtunee siitä, että eräs *Desideriuksen* nopeutta säätelevistä heuristiikoista katkaisee kiihdytyksen, mikäli auto kulkee liiaksi radan ulkolaidalla (ks. luku 6.6).

6.5 Muut ohjausheuristiikat, heuristiikkojen yhdistäminen ja ohjaaminen poikkeustilanteissa

Tässä luvussa käsitellään loput *Desideriuksen* ohjausheuristiikoista, kuten erikoistilanteiden menetelmät. Tässä erikoistilanteilla tarkoitetaan muun muassa radalta suistumista ja esteisiin juuttumista. Lisäksi luvussa käsitellään eri ohjausheuristiikkojen yhdistämisen haasteita ja vaikutuksia.

Ohjelmoin *Desideriukselle* heuristiikan, jonka tarkoituksena oli varmistaa, että auton kaikki pyörät pysyvät radalla. Jos renkaat käyvät radan ulkopuolella, botti voi menettää auton hallinnan. Tämä johtuu siitä renkaat pitävät paremmin radalla kuin nurmikolla tai hiekalla. Lisäksi botti antaa huonon kuvan ajotaidoistaan, jos se poikkeaa tieltä taajaan. Edellä mainittua menetelmää kutsutaan jatkossa *palautusheuristiikaksi*.

Ennen palautusheuristiikan toteuttamista yritin muuttaa kiintopisteheuristiikkaa siten, että se pitää kaikki auton renkaat tiellä. Korjaus toimi siten, että kiintopistemenetelmän aloituspistettä $\boldsymbol{s}$ (ks. luku 6.2) siirrettiin sivuun marginaalilla $|m| = \frac{w}{2}$, jossa w on ajoneuvon leveys. Termin m merkki riippuu siitä, kumpaan suuntaan

aloituspistettä tulee siirtää. Toisin sanoen siitä, onko auto vaarassa kulkea radan vasemman vai oikean reunan yli. Aloituspisteen siirtäminen ei korjannut reunoilla ajamista, vaan paikoin jopa pahensi sitä. Yritin yhdistää kiintopisteheuristiikkaan myös loivuusheuristiikan alkeellista versiota. Se lisäsi suorilla auton suuntaan kulman $\delta = \pm\frac{\pi}{180}$. Menetelmää ei käytetty kaarteissa. Lisäys sai auton kääntymään äkillisesti kohtisuoraan rataan nähden. En onnistunut selvittämään, mistä virhe johtui. Päätin vaihtaa säteenseurantaan osittain juuri siksi, että en osannut yhdistää kiintopisteheuristiikkaa muihin menetelmiin.

Kokeilin yhdistää palautusheuristiikkaa säteenseurantaan, koska myös sillä auton renkaat käyvät ajoittain radan ulkopuolella. Näin tapahtuu esimerkiksi testiradan neulansilmässä eli 11. kaarteessa. Säteenseurannan kanssa palautusheuristiikka laskee kulman β kaavalla:

$$\beta = f(d) = \begin{cases} m/w_t & \text{jos } d \leq w_c \\ 0 & \text{jos } d > w_c, \end{cases}$$

missä $d = |m| - \frac{w_t}{2}$, m auton etäisyys radan keskipisteestä, sekä w_t radan ja w_c auton leveys. Tämän jälkeen auton suuntaa korjataan kulman β verran. Palautusheuristiikan yhdistäminen säteenseurantaan ei aiheuttanut silminnähtävää muutosta botin ajossa, mutta kierrosaika lyheni 12 sadasosasekuntia. Koska menetelmällä ei ollut merkittävää vaikutusta suuntaan taikka toiseen, poistin sen *Desideriuksen* toteutuksesta. Se ei pitänyt auton renkaita paremmin tiellä. Lisäksi kävi ilmi, että palautusheuristiikkaa ei välttämättä tarvita: botti ajaa kaarteet oikomatta ilmankin sitä.

Loivuusheuristiikan yhdistäminen säteenseurantaan aiheutti ongelmia: Auto nyki puolelta toiselle, kun menetelmät pyrkivät ohjaamaan sitä eri suuntiin. Ilmiö oli voimakkaimmillaan ennen kaarteita, kun loivuusheuristiikka ohjasi autoa radan ulkolaidalle ja säteenseuranta pyrki kääntämään sitä kaarteeseen. Tästä syystä päätin, että heuristiikkoja on käytettävä eriaikaisesti. Bottia ohjaa suorilla loivuusheuristiikka ja kaarteissa säteenseuranta. Tässä kohtaa muodostui ongelmaksi selvittää, miten botin tulee vaihtaa heuristiikasta toiseen. Teisteissä selvisi, että heuristiikkaa tulee vaihtaa selvästi ennen kaarretta. Muuten kääntyminen myöhästyy niin, että botti voi suistua radalta. Näin tapahtui heti testiradan ensimmäisessä kaarteessa. Heuristiikan vaihtaminen kaarteen päättyessä osoittautui ongelmattomaksi.

Desideriuksen ajo näytti epärealistiselta, kun se siirtyi loivuusheuristiikasta suoraan käyttämään säteenseurantaan. Botin kulusta saattoi nähdä, milloin se vaihtoi menetelmää. Ilmiöön kiinnitti huomiota myös ohjaajani Rasku. Tästä syystä lisäsin toteutukseen välivaiheen, jossa botti siirtyy asteittain heuristiikasta toiseen. Välivai-

heeseen siirrytään, kun auton etäisyys seuraavaan kaarteeseen on d_a, ja säteenseurantaan, kun etäisyys on d_b. Valitsin rajojen arvoiksi $d_a = 60\,\text{m}$ ja $d_b = 20\,\text{m}$. Aluksi yhdistin heuristiikat käyttämällä niiden painotettua keskiarvoa:

$$w \cdot \beta + (1 - w) \cdot \alpha,$$

missä β on loivuusheuristiikan antama suunta, ja α säteenseurannan. Painokerroin laskettiin kaavalla $w = \frac{d - d_b}{d_a - d_b}$, missä d on auton etäisyys seuraavasta kaarteesta. Kuitenkin jokin tässä välivaiheen toteutuksessa sai auton kääntymään yhtäkkiä jyrkästi tieltä. Näin tapahtui joka kerta testiradan 9. kaarteen jälkeen.

Koska en saanut ratkaistua painotetun keskiarvon ongelmia, päädyin toteuttamaan heuristiikkojen välivaiheen toisella tavalla. Ohjaajani Rasku ehdotti, että välivaiheessa voisi käyttää trigonometristä funktioita. Muutin toteutusta siten, että välivaiheessa käytetään ainoastaan säteenseurannan suuntaa α kaavalla

$$w \cdot \alpha, \quad w = \cos\left(\frac{d - d_b}{d_a - d_b} \cdot \frac{\pi}{2}\right)$$

Tämä tekee heuristiikkojen vaihtamisesta sulavaa. Tälla painokertomeilla ei myöskään esiinny sellaisia ongelmia kuin painotetulla keskiarvolla. Vakiot d_a ja d_b ovat samat kuin edellä.

Botti tarvitsee myös ohjausheuristiikkoja, joilla se selviää erikoistilanteista. *Desideriuksen* menetelmät toimivat oikein vain silloin, kun auto on tiellä. Esimerkiksi säteenseurantaa ei toimi radan ulkopuolella, koska se ei tarkista kummalta puolelta säteet leikkaavat tien reunan. En kuitenkaan yrittänyt yhdistää säteenseurantaan menetelmää, joka ohjaa botin takaisin tielle, koska *Desiderius* ei suistu testiradalta. Aikaisemmin kuvattu palautusheuristiikka esti autoa suistumasta tieltä, mutta se ei kykene palauttamaan jo suistunutta bottia tielle. *Desideriuksen* aikaisemmat versiot noudattavat Wymannin opasta [51], joten toteutin myös siinä esitellyt erikoistilanteiden heuristiikat. Oppaan ohjausheuristiikat toimivat myös radan ulkopuolella. Ne eivät kuitenkaan selviä tilanteesta, jossa botti on törmännyt johonkin esteeseen.

Esteitä varten oppaassa esitellään algoritmi 4. Jos sen ehdot täyttyvät, niin laskurin c arvoa kasvatetaan yhdellä. Mikäli ei, niin c nollataan. Ehdon ensimmäinen osa tarkistaa, että botti on riittävän kaukana radasta. Toinen, että sen vauhti on tarpeeksi alhainen. Kolmas, että auton keula osoittaa radasta poispäin. Neljäs ja viimeinen ehto tarkistaa, että keula osoittaa "riittävästi"eri suuntaan kuin radan tangentti. Wymannin mukaan viimeinen ehto tarvitaan siksi, että harvoin syntyy tilanne, jossa auton asento on radan suuntainen, ja se on juuttunut esteeseen. Kun laskurin c arvo

Algoritmi 4 tutkii onko botti juuttunut esteeseen

$m_{min} \Leftarrow 5$
$v_{max} \Leftarrow 2$
$\delta \Leftarrow \frac{\pi}{6}$
$\gamma \Leftarrow \theta - \alpha$ //auton suunnan ja radan tangentin erotus
if $(|m| < m_{min}) \vee (v_x > v\_max) \vee (m \cdot \gamma \geq 0) \vee (|\gamma| < \delta)$ **then**
  $c \Leftarrow 0$
else
  $c \Leftarrow c + 1$
end if
return $c \geq t$

ylittää rajan t, botti alkaa peruuttaa. Oppaassa $t = 100$, mikä vastaa ajallisesti noin kahta sekuntia. Mitä suurempi t, sitä kauemmin botti odottaa ennen kuin se alkaa peruuttaa. Tämä hidastaa botin paluuta radalle. Toisaalta jos aikaraja on liian lyhyt, botti voi alkaa peruuttaa turhaan. Peruuttamista jatketaan niin kauan kuin ehto $c \geq t$ pätee. Kun auto peruuttaa, sen ohjaamiseen käytetään keskihakuheuristiikkaa (ks. luku 6.2). Sen antama suunta kerrotaan tällöin kertoimella $k = -1$.

Lisäsin *Desideriuksen* ohjaukseen virhettä, jotta sen ajo näyttäisi vähemmän kaavamaiselta. Virhe lisätään ohjaukseen siten, että joka simulaatiojaksolla botin suuntaa muokataan kulmalla:

$$\epsilon = (1 - w) \cdot \alpha + w \cdot \beta$$
$$w = (t \bmod p) \cdot \frac{1}{p},$$

missä $\alpha, \beta \in \left[-\frac{\pi}{180}, \frac{\pi}{180}\right]$, w painokerroin, t simulaatiojaksojen lukumäärä kisan alusta ja p jakson pituus. Joka kerta, kun botin koodia kutsutaan muuttujan t arvoa kasvatetaan yhdellä. Jakson p pituus vaikuttaa siihen, kuinka nopeasti virhe vaikuttaa auton suuntaan. Jos p on varsin lyhyt, auto nykii nopeasti puolelta toiselle. Kisan alussa arvotaan kulmat α ja β. Tämän jälkeen aina, kun $t \bmod p = 0$ asetetaan $\alpha := \beta$, ja kulma β arvotaan uudelleen. Ohjausvirhe ei vaikuta *Desideriuksen* suorituskykyyn, eikä sen yhdistäminen säteenseuranta- tai loivuusheuristiikkaan aiheuttanut ongelmia.

6.6 Kiihdytys ja jarrutus

Tässä luvussa käsitellään *Desideriuksen* nopeuden säätelyyn eli kiihdytykseen ja jarrutukseen liittyviä heuristiikkoja. Näitä ovat Wymannin oppaassa [51] esitellyt menetelmät ja eri lähteistä lainaamani heuristiikat. Lisäksi luvussa kerrotaan, miten muuntelin Wymannin menetelmiä. Tietylle rataosuudelle sopiva nopeus voidaan päätellä monella eri tavalla. Wymannin oppaan ja *Desideriuksen* menetelmissä se lasketaan radan geometrian perusteella.

Tutorin ensimmäinen versio osaa ainoastaan kiihdyttää. Se valitsee ajon alkaessa ensimmäisen vaihteen ja asettaa kiihdytysaktuaattorin arvoksi 0,3. Koska botti ei vaihda 1. vaihdetta korkeammalle, pysyy nopeus koko ajon niin alhaisena, ettei jarrutusta tarvita. *Tutorin* seuraava versio vaihtaa vaihteita ja asettaa aktuaattorin arvoksi aina 1,0, kunnes auto saavuttaa nopeuden v_{max}. Suorilla $v_{max} = \infty$, ja kaarteissa:

$$v_{max} = \sqrt{g\mu r}, \tag{6.1}$$

missä μ on kitkakerroin ja r kaarteen säde (ks. luku 3.4). Tässä μ oletetaan vakioksi. Kun v_{max} on saavutettu, aktuaattorin arvo lasketaan kaavalla $x = \frac{v_{max}}{w_r} \cdot \frac{\phi}{\mathrm{RPM}_{max}}$. Kaavassa w_r on vetävän pyörän säde, ϕ vaihteen välityssuhde ja RPM_{max} moottorin maksimikäyntinopeus. Kaava pyrkii pitämään auton nopeuden vakiona. Edellä kuvattu heuristiikka vaatii jonkinlaisen kynnyksen ϵ, sillä muuten aktuaattori voi jäädä sahaamaan maksimikiihdytyksen 1,0 ja tasaisen kiihdytyksen x välille. Kynnyksen kanssa heuristiikka noudattaa funktiota:

$$f(v) = \left\{ \begin{array}{ll} 1{,}0 & \text{jos } v + \epsilon < v_{max} \\ x & \text{jos } v + \epsilon \geq v_{max} \end{array} \right\}$$

Missä v on auton nopeus ja $f(v)$ aktuaattorin arvo. Wymann käyttää kynnyksenä arvoa $\epsilon = 1{,}0\,\frac{\mathrm{m}}{\mathrm{s}}$.

Wymannin oppaassa esitellään jarrutusheuristiikka, joka seuraa, onko edessäpäin kaarteita, joihin nähden botin nopeus on liian korkea. Se tutkii ratasegmenttejä etäisyyden d päähän. Etäisyys d on matka, joka tarvitaan auton pysäyttämiseen. Se lasketaan kaavalla $d = \frac{v_1^2}{2\mu g}$, missä v_1 on auton nopeus (ks. luku 3.1). Jokaisen segmentin kohdalla tutkitaan toteutuuko $v_1 > v_2$, missä v_2 on kaavalla 6.1 laskettu segmentin nopeusrajoitus. Jos ehto toteutuu, lasketaan jarrutusmatka $\hat{d} = \frac{v_1^2 - v_2^2}{2\mu g}$. Mikäli jarrutusmatka $\hat{d}$ on pitempi kuin tutkittavan segmentin s etäisyys d_s, heuristiikka asettaa jarruaktuaattorin arvoksi 1,0. Mikäli yhdellekään segmentille matkalla

d ei toteudu ehto $v_1 > v_2$, aktuaattorin arvoksi jää 0,0. Tässä kuvattu menetelmä on melko alkeellinen, sillä se toimii "kaikki tai ei mitään"–periaatteella. Auto joko lukkojarruttaa tai ei jarruta lainkaan. Kuten luvussa 3.1 kerrotaan, lukkojarrutus ei ole tehokkain tapa pysäyttää autoa.

Edellä kuvattu menetelmä tekee botin jarruttamisesta varsin hetkittäistä. Se saa auton jarruvalot vilkkumaan jatkuvasti. Tämä tekee botin ajosta epärealistisen näköistä. Tästä syystä yritin muokata menetelmää. Ensin muutin sitä siten, että kun on löytynyt segmentti, jolle pätee $d_s < \hat{d}$, botti jarruttaa kunnes saavuttaa tämän segmentin. Muutos sai botin jarruttamaan niin pitkään, että se suistui joskus radalta. Seuraavaksi muokkasin menetelmää siten, että botti jarruttaa edelleen segmentille s asti, mutta pienentää jarrutusvoimaa joka simulaatiojaksolla. Tämä pidensi jarrutusmatkaa entisestään. Poistin muutokset, koska ne eivät ratkaisseet Wymannin [51] menetelmän ongelmia.

Sain kohennettua *Desideriuksen* jarrutuksen uskottavuutta lisäämällä toteutukseen itse kehittämäni *kosmeettisen jarrutuksen*. Se toimii siten, että aina kun *Desiderius* päättää jarruttaa, niin jarrutuksen kestoon lisätään viisi simulaatiojaksoa. Näiden ylimääräisten jaksojen aikana aktuaattorin arvo on 0,01. Botti siis näyttää jarruttavan, vaikka tosiasiallisesti sen vauhti ei juurikaan hidastu. Kosmeettisen jarrutuksen aikana varsinainen jarrutusheuristiikka toimii normaalisti. Toisin sanoen jos Wymannin menetelmä löytää segmentin $d_s < \hat{d}$, botti jarruttaa normaalisti. Tällöin tähän uuteen päätökseen lisätään jälleen viisi jaksoa ylimääräistä. Kosmeettisen jarrutus kasvattaa kierrosaikaa muutamalla sekunnin sadasosalla.

Tutorin ja *Desideriuksen* myöhemmät versiot käyttävät edellä kuvattujen heuristiikkojen monimutkaisempia versiota. Wymann muokkaa esimerkiksi kaavaa, jolla botti määrittelee nopeutensa (ks. yhtälö 6.1). Muokattu versio toimii suorilla samalla tavoin, mutta saa auton kiihdyttämään voimakkaammin kaarteissa. Uusi kaava ei käytä vain yhden vaan useamman samaan kaarteeseen kuuluvan ratasegmentin tietoja. Ensin lasketaan kulma α, joka on segmenttien kulmien summa. Segmenttejä summataan, kunnes $\alpha \geq \frac{\pi}{2}$ tai kaarre päättyy. Tämän jälkeen α skaalataan $\bar{\alpha} = \alpha / \frac{\pi}{2}$. Nyt kaavan 6.1 säteenä r käytetään termiä $r = (r_s + \frac{w_s}{2}) / \sqrt{\bar{\alpha}}$. Siinä r_s on segmentin keskilinjan säde ja w_s sen leveys. Tarkemmin r_s on kaarteen ulko- ja sisälaidan säteiden keskiarvo. En tiedä, miksi Wymann käyttää laskuissa arvoa $r_s + \frac{w_s}{2}$, eikä ulkokaarteen sädettä suoraan. Wymann perustelee kaavan muuttamista sillä, että se lyhentää botin kierrosaikaa. Tämä on myös oma havaintoni. Kaavan uusi versio vaatii kuitenkin seurakseen heuristiikan, joka katkaisee kiihdytyksen, jos auto on

vaarassa suistua radalta. Tämä esitellään seuraavaksi.

Mitä aggressiivisemmin *Tutor* kiihdyttää, sitä suurempi vaara on, että se suistuu radalta. Niinpä Wymannin opas [51] neuvoo apuheuristiikan, joka säätelee kiihdytystä sen mukaan, kuinka reunalla auto kulkee. Jos suorilla toteutuvat ehdot $|d| > m$ ja $v_x \geq v_{min}$, niin heuristiikka asettaa kiihdytysaktuaattorin arvoksi 0,0. Ehdossa d on auton etäisyys radan keskeltä. Marginaali $m = w_s/C$ säätelee kuinka reunalle botti saa ajautua. Siinä w_s on ratasegmentin leveys ja C ohjelmoijan valitsema vakio. Wymann käyttää arvoa $C = 4{,}0\,\mathrm{m}$. Mitä suurempi C, sitä varmemmin auto pysyy tiellä, mutta sitä hitaammin se kulkee kierroksen. Kiihdytys katkeaa sitä useammin, mitä suurempi vakion arvo valitaan. Ehto $v_x \geq v_{min}$ varmistaa, että kiihdytystä ei katkaista, jos auton vauhti on jo valmiiksi alhainen. Oppaassa käytetään arvoa $v_{min} = 5{,}0\,\frac{m}{s}$. Lisäksi kaarteissa kiihdytys katkaistaan vain, jos auton on radan ulkolaidalla. Muutin yllä kuvattua heuristiikkaa edelleen siten, että kiihdytys lopetetaan vain, jos auton todellinen liikesuunta osoittaa sisäkaarteesta poispäin. Liikesuunta $\boldsymbol{u}$ saadaan kaavalla $\boldsymbol{u} = \theta - \arctan(\frac{v_y}{v_x})$, missä θ on radan tangentti ja $\boldsymbol{v}$ on nopeus. Löysin muutoksen *bt*-botin lähdekoodista.

Lisäsin *Desideriuksen* toteutukseen TCL- eli luistonestoheuristiikan (ks. luku 3.6). Löysin sen *bt*-botin koodista. Lisäsin menetelmän, koska pidin Wymannin oppaan [51] kiihdytystä liian voimakkaana. Halusin kokeilla, lyheneekö botin kierrosaika, jos auton renkaat sutivat vähemmän kiihdyttäessä. Menetelmä laskee ensin vetävien renkaiden keskimääräisen nopeuden kaavalla $L = \frac{\omega_i + \omega_j}{2} * r_i$, missä ω_i ja ω_j ovat renkaiden pyörimäsmäärät ja r_i niiden säde. Tässä kohtaa on huomattava, että menetelmä toimii oikein vain takavetoisilla autoilla. Keskiarvon laskemisen jälkeen tutkitaan päteekö ehto $L - v_x > S$, missä v_x on auton nopeus ja S ohjelmoijan asettama raja. Oppaassa $S = 2\,\frac{\mathrm{m}}{\mathrm{s}}$. Jos ehto toteutuu, pyörien katsotaan sutivan. Tällöin menetelmä muokkaa kiihdytysaktuaattorin a arvoa kaavalla $a := a - \min(a, (L - S)/R)$. Termi R vaikuttaa siihen, kuinka tehokkaasti luistonesto vaikuttaa. Tässä käytetään arvoa $R = 10\,\frac{\mathrm{m}}{\mathrm{s}}$. Luistonesto kasvattaa *Desideriuksen* kierrosaikaa muutamalla kymmenesosasekunnilla. En tutkinut, miten se vaikuttaa botin ajoon liukkailla pinnoilla, kuten nurmikolla.

Lainasin *Desideriuksen* koodiin myös ABS-jarrutusheuristiikan *bt*-botin koodista. Menetelmä muistuttaa monessa suhteessa edellä kuvattua TCL-heuristiikkaa. ABS-heuristiikka laskee ensin pyörien nopeuden keskiarvon kaavalla $L = \sum_{i=1}^{n=4} \frac{\omega_i}{n} \cdot r_i$. Siinä ω_i on renkaan pyörimismäärä ja r_i sen säde. Tämän jälkeen tarkastetaan toteutuuko ehto $L - v_x > S$, missä v_x on auton nopeus ja S ohjelmoijan asettama raja. Raja

S määrää, milloin ABS-jarrutus käynnistyy. Tässä $S = 2\,\frac{\mathrm{m}}{\mathrm{s}}$. Mikäli ehto toteutuu, muutetaan jarruaktuaattorin b arvoa kaavalla $b := b - \min(b, (L - S)/R)$. Tässä $R = 5\frac{\mathrm{m}}{\mathrm{s}}$. ABS-jarrut pidentävät hieman jarrutusmatkaa ja lyhentävät kierrosaikaa muutamalla kymmenesosasekunnilla.

Yritin tehdä *Desideriuksen* kiihdytyksestä ja jarrutuksesta vähemmän aggressiivista jo ennen kuin löysin ABS- ja TCL-heuristiikat. Toteutin menetelmän, joka tutkii `tWheelInfo.wheelSlip` muuttujien arvoja. Nämä ilmoittavat, missä määrin pyörät sutivat. Botti yritti muuttujien perusteella päätellä, kiihdyttääkö tai jarruttaako se liian voimakkaasti. Kun muuttujien keskiarvo $\bar{X}$ ylitti rajan S, botti vähensi asianmukaisen aktuaattorin arvoa 0,05 askeleella. Heti kuin $\bar{X}$ laski takaisin alle rajan S, aktuaattorin arvoksi palautettiin 1,0. Tämä menetelmä aiheutti merkittävän viiveen botin toimiin. Se ei myöskään ehkäissyt esimerkiksi lukkojarrutusta. Menetelmä epäonnistui osittain siksi, että `wheelSlip`-muuttujien arvot heilahtelevat voimakkaasti. Niiden avulla ei voi tarkasti päätellä, sutivatko renkaat.

Lisäsin nopeuden säätelyyn portaita, koska pidin botin ajamista liian ehdottomana. Se osasi käytännössä vain kiihdyttää tai jarruttaa. Ohjelmoin *Desideriuksen* käyttäytymiseen vaiheet, jossa se yrittää säilyttää nopeuden vakiona, ja jossa se lakkaa kiihdyttämästä, mutta ei vielä jarruta. Wymannin oppaassa [51] esitellään ehto, jolla botti pitää nopeutensa vakiona, mutta se toimii liian kapealla alueella. *Desiderius* säilyttää nopeuden vakiona, jos ehto $0 < v_{max} - v_x \leq N$ toteutuu. Siinä v_{max} lasketaan kaavalla 6.1. Termi v_x on auton nopeus ja $N = \frac{10{,}0}{3{,}6}\,\frac{\mathrm{m}}{\mathrm{s}}$. Toinen lisäämäni porras noudattaa ehtoa $0 < v_x - v_{max} \leq B$, missä $B = \frac{15{,}0}{3{,}6}\,\frac{\mathrm{m}}{\mathrm{s}}$. Jos ehto toteutuu, niin kiihdytysaktuaattorin arvoksi asetetaan 0,0. Botti saa siis ajaa $B\,\frac{\mathrm{m}}{\mathrm{s}}$ ylinopeutta, ennen kuin se alkaa jarruttaa. Näiden lisäysten jälkeen *Desideriuksen* nopeuden säätely on neliportainen: se voi kiihdyttää, säilyttää nopeutensa vakiona, lakata kiihdyttämästä tai jarruttaa. Muutin nopeuden säätelyä edelleen siten, että tavoitenopeudeksi v_{max} asetetaan aina jyrkimmän matkalta d löytyneen ratasegmentin sallima nopeus. Jyrkin tarkoittaa tässä segmenttiä, jolle yhtälö 6.1 palauttaa pienimmän arvon. Matka d on sama kuin Wymannin jarrutusheuristiikassa. Tämä varovaisempi ajotapa kasvattaa kierrosaikaa yli kaksi sekuntia, mutta päätin käyttää sitä, koska se saa botin ajon näyttämään realistisemmalta.

Kokeilin ohjelmoida *Desideriukselle* menetelmän, joka käynnistää hätäjarrutuksen, kun auto suistuu tieltä. Sen tavoitteena oli, että botti palaa nopeammin radalle ja toisaalta välttyy törmäämistä reunavalleihin. Hätäjarrutuksessa jarruaktuaattorin arvoksi asetetaan 1,0. Algoritmi 5 kuvaa hätäjarrutuksen pseudokoodina. Siinä eh-

Algoritmi 5 hätäjarrutus

$\alpha \Leftarrow autonKulkusuunta$
$v_{min} \Leftarrow 10$ //nopeus m/s
return $(|m| > \frac{w}{2}) \wedge (m \cdot \alpha < 0) \wedge (v_x > v_{min})$

don ensimmäinen osa tutkii onko auto radan ulkopuolella. Muuttuja m on auton etäisyys radan keskeltä, ja w radan leveys. Seuraava osa tarkistaa kulkeeko botti poispäin radasta. Luultavasti auton varsinaisen liikkeen suunta $\arctan(\frac{v_y}{v_x})$ toimisi tässä paremmin kuin keulan suunta α. Ehdon viimeinen osa tarkistaa, kulkeeko auto niin nopeasti, että jarruttaminen on tarpeen. Menetelmä toimi vain osittain. Tiukka raja $v_x > v_{min}$ aiheutti sen, että auto jäi vuoronperää kiihdyttämään ja jarruttamaan. Auton nopeus sahasi arvon v_{min} molemmin puolin, kunnes jokin muu ehdoista ei enää täyttynyt. Lisäksi hätäjarrutus ei selviytynyt tilanteesta, joissa botti suistui radalta ennen jyrkkää kaarretta, koska se tulkitsi auton kulkevan edelleen radan suuntaisesti. Näistä syistä poistin menetelmän toteutuksesta.

6.7 Vaihteet ja kytkin

Botti tarvitsee TORCSissa jonkinlaisen heuristiikan vaihteiden vaihtamiseksi. Vaihdeheuristiikan valinnalla ei kuitenkaan saavuteta suuria eroja kierrosajassa [2]. Botti voi TORCSissa käyttää jotakin kytkinheuristiikkaa, mutta se ei ole välttämätöntä. Peli ei tarjoa tarkkoja tietoja siitä, miten moottori tuottaa vääntöä tai tehoa eri kierrosluvuilla, mistä syystä vaihdeheuristiikan optimointi on haastavaa [31]. Naiivi heuristiikka vaihtaisi korkeamman vaihteen, kun moottori ylittää tietyn kierrosluvun, ja alemman kun kierrokset laskevat alle toisen. Tässä on kuitenkin se ongelma, että moottorin kierrosluku väistämättä putoaa, kun vaihdetaan korkeampi vaihde, ja päinvastoin. Kun vaihde vaihdetaan esimerkiksi kolmannelta neljännälle, kierrosluku voi pudota niin alhaiseksi, että naiivi menetelmä päättää vaihtaa takaisin kolmannelle. Vaihtamiseen tarvitaan siis jonkinlainen kynnys.

Käytin *Desideriuksen* toteutuksessa *bt*-botin vaihdeheuristiikkaa. Algoritmi 6 esittää sen pseudokoodina. Menetelmä tutkii ensin käyttääkö botti peruutusvaihdetta. Sitten tulisiko sen vaihtaa korkeampi tai alempi vaihde. Pseudokoodissa RPM_{max} on moottorin maksimikäyntinopeus, ϕ_i tämänhetkisen ja ϕ_{i-1} edellisen vaihteen välityssuhde, r vetävän renkaan säde ja v_x auton nopeus. Vakio C määrittää moottorin suurimman sallitun käyntinopeuden. Kynnys K estää bottia vaihtamasta vaihdetta

Algoritmi 6 heuristiikka vaihteiden vaihtamiseksi

if $vaihde < 0$ **then**
  $vaihde \Leftarrow 1$
  return
end if
$\omega \Leftarrow \frac{\text{RPM}_{max}}{\phi_i}$
$r \Leftarrow renkaanSäde$
$C \Leftarrow 0{,}9$
if $\omega \cdot r \cdot C < v_x$ **then**
  $vaihde \Leftarrow vaihde + 1$
  return
end if
$\omega \Leftarrow \frac{\text{RPM}_{max}}{\phi_{i-1}}$
$K \Leftarrow 4{,}0$
if $(vaihde > 1) \wedge ((\omega \cdot r \cdot C) > (v_x + K))$ **then**
  $vaihde \Leftarrow vaihde - 1$
end if

edestakaisin.

Ennen *bt*-botin vaihdeheuristiikan löytämistä toteutin menetelmän, joka vaihtaa vaihteita moottorin tehoalueen (engl. *power band*) perusteella. Oletin tehoalueen sijaitsevan muuttujien `enginerpmMaxTq` ja `enginerpmMaxPw` välissä. Ensimmäinen ilmoittaa moottorin käyntinopeuden maksimiväännöllä, ja jälkimmäinen maksimiteholla . Nämä muuttujat ovat tietueessa `tCarElt`. Menetelmä toimi siten, että kun käyntinopeus laski alle maksimiväännön, vaihdettiin alempi vaihde. Kun käyntinopeus nousi yli maksimitehon, vaihdettiin ylempi vaihde. Menetelmä jäi aluksi vaihtamaan vaihteita edestakaisin, kunnes lisäsin siihen kynnyksen. Heuristiikkani ei lyhentänyt kierrosaikaa, joten ainakaan näin yksinkertaisesti tietoja moottorin suorituskyvystä ei voi hyödyntää.

TORCSissa voi vaihtaa vaihdetta käyttämättä kytkinaktuaattoria. En havainnut tämän vaurioittavan autoa. Wymannin oppaassa [51] kytkimen käytöstä ei puhuta lainkaan, mutta *bt*-botin toteutus sisältää heuristiikan sitä varten. Tätä heuristiikka tosin käytetään vain, kun valittuna on 1. vaihde tai peruutusvaihde, tai vaihde on vapaalla. Testeissäni menetelmä sai auton lähtökiihdytyksen näyttämään realistisemmalta, joten lisäsin sen *Desideriuksen* koodiin. Ilman kytkinheuristiikkaa au-

ton renkaat sutivat lähdössä voimakkaasti, ja tielle jää tummat jäljet. Menetelmä ei vaikuta merkittävästi suorituskykyyn. Se pidentää ensimmäisen kierroksen aikaa muutamalla sadasosasekunnilla.

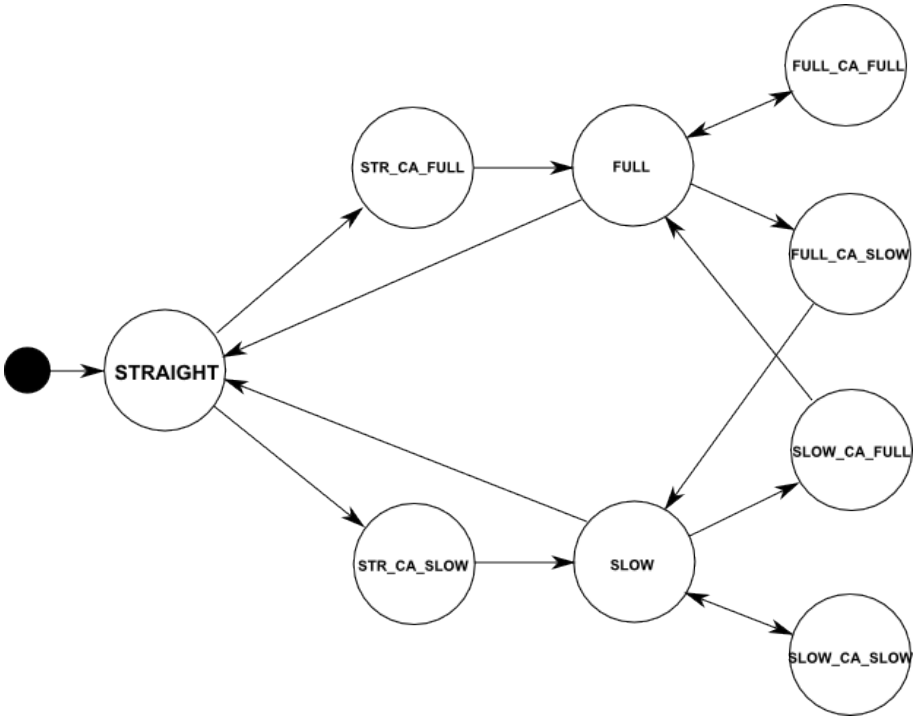
6.8 Kaarteiden tyypin tunnistus ja tilakone

Luin Quadfliegin et al. [33] artikkelin, jossa he kehittävät botillensa kyvyn luokitella erityyppisiä rataosuuksia. Botti jakaa radan suoriin ja kaarteisiin. Kaarteet luokitellaan edelleen loiviin, melko jyrkkiin, jyrkkiin ja neulansilmiin niiden kulman perusteella. Quadfliegin et al. mukaan kilpa-ajaja jakaa mielessään radan tämänkaltaisiin osuuksin. He pitävät siten kehittämäänsä luokittelua realistisena. Kiinnostuin erityisesti ryhmän havainnosta, jonka mukaan botin ei tarvitse alentaa nopeuttaan, jos kaarteen kulma $|\alpha| < \frac{\pi}{9}$. Päätin toteuttaa *Desideriukselle* samankaltaisen luokittelun, koska uskoin sen parantavan kierrosaikoja.

Desideriuksen sensorit havaitsevat luokittelun avulla enemmän. Ne eivät käsittele enää pelkästään yksittäisiä ratasegmenttejä, vaan kokonaisia osuuksia. Tietyssä mielessä botti saa nyt enemmän tietoja kuin mitä sen sensorit havaitsevat, sillä rataosuus voi jatkua niiden kantomatkan tuolle puolen. Luokittelun kanssa säteenseurantaheuristiikka on edelleen lokaali ohjausmenetelmä [44]. Nyt vain botin sensorit eivät enää havainnoi sen välitöntä ympäristöä, vaan ympäristön esiprosessoitua mallia.

Quadflieg et al. [33] toteuttivat menetelmän, jolla botti tunnistaa erilaiset kaarteet TORCSin palvelinversion säteenseurantasensorien avulla (ks. luku 5.1). Sitä ei voi käyttää reaaliaikaisesti, vaan botin on opeteltava radan geometria etukäteen. *Desideriukselle* tämä menetelmä ei ole tarpellinen, sillä se voi lukea radan tiedot suoraan joko XML-tiedostosta tai TORCSin tietorakenteesta. Päätin lukea tietorakenteita, sillä oletin sen olevan nopeampaa kuin tiedoston läpikäyminen. Tietorakenteessa rata on pilkottu pieniin segmentteihin. Tämän vuoksi lisäsin toteutukseeni menetelmän, joka kokoaa ratasegmentit laajemmiksi osuuksiksi. Päätin toteuttaa segmenttien koostamisen ja rataosuuksien luokittelun omassa luokassaan `TTrackClassifier`.

Rataluokittimen lisäksi toteutin sitä hyödyntävän tilakoneen. Molempien toteutusta esitellään liitteessä C. Tilakoneen tilat ja siirtymät on esitetty kuvassa 6.4. *Desiderius* voi tilakoneen avulla päättää ajamisestaan sekä tämänhetkisen että seuraavan rataosuuden perusteella. Toisin kuin Quadfliegin ryhmän [33] toteutus, luokittimeni jakaa rataosuudet vain kolmeen eri tyyppiin: `STRAIGHT`, `FULL` ja `SLOW`.



Kuva 6.4: Desideriuksen ajoa ohjaava tilakone.

Näistä tulee tilakoneeseen yhdeksan eri ajotilannetta: `STRAIGHT`, `STR_CA_FULL`, `STR_CA_SLOW`, `FULL`, `SLOW`, `F_CA_F`, `F_CA_S`, `S_CA_F` ja `S_CA_S`. `STRAIGHT` tarkoittaa, että botti ajaa suoralla. `FULL` tarkoittaa, että se ajaa loivaa kaarretta, ja `SLOW` hidasta. Tässä loivalla kaarteella tarkoitetaan sellaista, jonka botti voi Quadfliegin et al. mukaan ajaa hidastamatta. Yhdistelmät muotoa `*_CA_*` tarkoittavat tilannetta, jossa auto on lähestymässä seuraavaa kaarretta. Alkuosa merkitsee tämänhetkisen osuuden tyyppiä ja loppuosa seuraavan. Desiderius siirtyy `*_CA_*`-tilaan silloin, kun säteenseuranta havaitsee seuraavan kaarteen. Quadflieg et al. käyttivät säteille kiinteää kantomatkaa 90 m. Testeissäni tämä vaikutti liian lyhyeltä, joten kuten luvussa 6.3 kerrotaan, muutin säteenseurannan kantomatkan dynaamiseksi. *Desiderius* siirtyy `*_CA_*`-tilasta seuraavaan, kun se saavuttaa kaarteen ensimmäisen segmentin. Se siirtyy takaisin `STRAIGHT` tilaan, kun se saavuttaa suoran ensimmäisen segmentin. Tilakone sisälsi alunperin myös tilat `F_A_STR` ja `S_A_STR`, jotka tarkoittivat, että botti on tulossa kaarteesta suoralle. Poistin nämä tilat, koska en keksinyt, miten botti voisi niitä hyödyntää.

Kun olin ohjelmoinut rataluokittimen ja tilakoneen, testasin toteutusta radalla `wheel2`. Koska Quadfliegin ryhmän [33] mukaan botin ei tarvitse hidastaa `FULL`-kaarteissa, oletin, että sen ei tarvitse hidastaa myöskään tiloissa `STR_CA_FULL`,

F_CA_F tai S_CA_F. Tämä oletus päti testiradalla. Kuitenkin sama versio *Desideriuksesta* suistui tieltä *Forza*-radalla. Se johtui siitä, että *Forza* sisältää niin pitkän suoran, että vauhti ehtii kiihtyä liian korkeaksi jopa loivaan kaarteeseen. Kun palautin toteutuksen sellaiseksi, että *Desiderius* jarruttaa tiloissa STR_CA_FULL, F_CA_F ja S_CA_F, sen kierrosaika piteni selvästi. Lisäksi tämä varovaisempi versio ajaa edelleen ulos Aalborg-radalla. *Desideriuksen* viimeisin versio ei jarrutta ennen loivia kaarteita, mutta se käyttää porrastetun nopeuden säätelyn varovaisempaa versiota (ks. luku 6.6). Ratkaisu on kompromissi toteutuksen suorituskyvyn ja joustavuuden välillä.

Desiderius valitsee tilakoneen avulla myös käyttämänsä ohjausheuristiikan. Se käyttää loivuusheuristiikkaa STRAIGHT-tilassa. STR_CA_*-tiloissa se siirtyy loivuusheuristiikasta säteenseurantaan. Muissa tiloissa botti käyttää säteenseurantaa. Yritin yhdistellä loivuusheuristiikkaa ja säteenseurantaa myös tiloissa, joissa botti on siirtymässä kaarteesta toiseen. Tässä kohtaa oli lisäehtona, että kaarteet kulkevat eri suuntaan. Kuitenkin heuristiikkojan yhteensovittamisen ongelmat korostuivat näissä tilanteissa (ks. luku 6.5). Lisäksi auto teki liian jyrkkiä liikkeitä. Päädyin takaisin siihen, että botti käyttää kaarteissa yksin säteenseurantaa.

Koitin muokata rataluokitinta samanlaiseksi kuin Quadfliegin et al. [33] toteutuksessa. Siinä samansuuntaiset kaarteet kerätään yhdeksi osuudeksi, mikäli ne seuraavat toisiaan, tai niiden välissä on vain lyhyt suora. Osuuden tyyppi määräytyy jyrkimmän osakaarteen mukaan. Tämä muutos sai kuitenkin *Desideriuksen* suistumaan testiradan ensimmäisen ja toisen kaarteen välissä. Luokitin oli yhdistänyt nämä yhdeksi SLOW-osuudeksi. Toinen kaarre on kuitenkin selvästi jyrkempi kuin ensimmäinen, ja botti ei jarruttanut riittävästi ennen sitä. Ongelma olisi saattanut poistua, jos SLOW-kaarteet olisi edelleen jaettu tyyppeihin MEDIUM, SLOW ja HAIRPIN kuten Quadflieg et al. tekevät. Muutin luokitinta edelleen siten, että pelkästään peräkkäiset kaarteet yhdistetään suuremmiksi osuuksiksi. Toisin sanoen lyhytkin suora katkaisi osuuden. Tämän jälkeen auto ei enää suistunut radalta, mutta muutos kasvatti kierrosaikaa liikaa. Totesin, että luokittimeni tulee käyttää useampaa kuin kolmea luokkaa, jotta se voisi yhdistellä osuuksia. En kuitenkaan halunnut lisätä luokkia, koska arvion uusien tilojen tekevän tilakoneesta liian monimutkaisen. Täten luovuin yrityksistä muuttaa luokitinta.

7 Toteutuksen suorituskyky ja uskottavuus, sekä johtopäätökset

Tässä luvussa arvioidaan TORCSiin ohjelmoimani botin *Desideriuksen* suorituskykyä ja uskottavuutta. Aluksi kerron, mitä tekniikoita lisäsin toteutukseen, mitä jätin ohjelmoimatta sekä pyrin perustelmaan ratkaisuni. Kerron myös punnitsemistani vaihtoehtoisista menetelmistä. Luvuissa 7.1 ja 7.2 tarkastellaan toteutuksen tämänhetkisiä tekniikoita ja mahdollisuuksia muunnella niitä. Edellisessä suorituskyvyn ja jälkimmäisessä uskottavuuden osalta. Luvussa 7.2 selvitetään lisäksi, miten onnistuin toteuttamaan tekotyperiä tekniikoita, ja vaikuttivatko ne botin uskottavuuteen. Luvussa 7.3 käsitellään *Desideriuksen* toteutuksen täydentämistä sekä vaihtoehtoisten menetelmien ohjelmoimista. Siinä esitetään ajatuksia myös siitä, minkälaisia tekotyperiä tekniikoita *Desiderius* voisi tulevaisuudessa hyödyntää. Luvussa 7.4 arvioin koko pro gradun onnistumista sekä pohdin, kuinka hyvin se vastaa tutkimuskysymykseensä.

Tarkoituksenani oli ohjelmoida tätä pro gradu -tutkielmaa varten teknisesti yksinkertainen botti, jonka avulla kokeilla erilaisten tekotyperien ja uskottavuutta lisäävien menetelmien toteuttamista. Lopputulos on epäonnistunut. *Desideriuksen* toteutus ei onnistunut siitäkään huolimatta, että päätin, että sen tulee kyetä ajamaan ainoastaan yksin ja pelkästään asfalttiradoilla. Helpotin vaatimuksia, koska en ole koskaan aikaisemmin ohjelmoinut tekoälyä, saatika autopelin tekoälyä. Tehty rajaus jättää *Desideriuksen* vaillinaiseksi, sillä se ei osaa nyt lainkaan huomioida pelaajaa tai muita kuljettajia. Muiden ohittaminen, väistäminen ja oman sijoituksen puolustaminen ovat autopelin perustaitoja [28]. Toisaalta esimerkiksi radalla pysyminen, auton ohjaaminen ja riittävän nopea ajaminen ovat mielestäni vielä olennaisempia taitoja kuin edelliset, joten niihin liittyvät haasteet tulee ratkaista ensin. *Desiderius* ei selviä kovin suorituskykyisesti tai uskottavasti edes näistä, kuten luvuissa 7.1 ja 7.2 kerrotaan.

Uskottavien ja tekotyperien tekniikoiden tutkiminen olisi kenties onnistunut paremmin, jos olisin käyttänyt toteutukseni pohjana jotakin toista bottia kuin *Tutor*. Koen kuitenkin toisen kirjoittaman koodin ymmärtämisen haastavammaksi kuin ohjelmoinnin. Kuten luvussa 6 kerrottaan, TORCSin lähdekoodi on harvakseltaan

kommentoitua. Tämä pätee myös bottimoduuleihin. Pyysin Andrew Sumnerilta hänen bottinsa lähdekoodia, koska Sumnerin mukaan hänen toteutustansa on helppo seurata[1]. Sumnerin botti ei sisälly TORCSin latauspakettiin. Lähdekoodi ei kuitenkaan ollut enää saatavilla. Kävin kirjeenvaihdon Sumnerin kanssa 28.1.2013. Vielä eräs vaihtoehto olisi ollut ohjelmoida botti *Speed Dreamsiin*, joka on TORCSin sisarprojekti[2]. Kuitenkin tammikuussa 2013, kun aloitin bottini ohjelmoimisen, löysin vähemmän ohjeita *Speed Dreamsille* kuin TORCSille. Koitin kysyä neuvoa *Speed Dreamsin* IRC-kanavalta (`Freenode #speed-dreams`), mutta luovuin yrityksestä, koska en useaan päivään saanut minkäänlaista vastausta.

Wymannin oppaan [51] noudattamisessa on se hyvä puoli, että se opettaa botin ohjelmoimisen ”kädestä pitäen”. *Tutorin* toteutus on myös melko yksinkertainen ja siten helppo seurata. Vaikka *Desideriuksen* toteuttaminen olikin alkuun vain oppaan koodin kopioimista, uskon oppineeni enemmän kuin jos olisin pelkästään perehtynyt valmiin botin koodiin. Jouduin luopumaan osasta oppaan tekniikoita, kun lisäsin toteutukseeni uusia menetelmiä. Esimerkiksi kiintopisteheuristiikasta tuli epäyhteensopiva, kun ohjelmoin säteenseurantaheuristiikan. Haittapuolena oppaan seuraaminen saattoi rajoittaa näkemystäni siitä, miten botti voidaan ohjelmoida. Tulin esimerkiksi käyttäneeksi ainoastaan lokaaleja ohjausmenetelmiä [44].

Jätin *Desideriuksen* toteutuksesta pois seuraavat *Tutorin* osat: varikkopysähdys, ohittaminen, toisten väistäminen sekä auton ja moduulin ilmentymien tietojen lukeminen XML-tiedostosta. Jätin varikkopysähdykset pois, koska tavoitteenani ei ollut ohjelmoida bottia, joka ajaa pitkiä, niin kutsuttuja kestävyyskisoja. Toisaalta pidin parempana yrittää luoda botille menetelmiä, joilla se ei vaurioita autoa lainkaan kuin opettaa sitä käymään varikolla korjattavana. Lisäksi *Tutorin* menetelmä varikolle ajamiseksi on melko monimutkainen. Se tarvitsee siihen muun muassa splineviivoja (ks. luku 5.2). Ohittaminen, väistäminen ja moduulin ilmentymien tietojen lukeminen liittyvät toisten kuljettajien kanssa kilpailemiseen. Kerron edellä, miksi *Desiderius* osaa ajaa vain yksin. Wymannin [51] oppaassa ei esitellä menetelmiä, jotka hyödyntäisivät auton tietoja. En myöskään löytänyt tällaisia muista lähteistä kuin *berniw*-botin lähdekoodista. Sen jarrutusheuristiikka hyödyntää tietoja ajoneuvon aerodynaamista ominaisuuksista (ks. luku 5.1). En kuitenkaan lainannut menetelmää *Desideriuksen* toteutukseen, koska en uskonut sillä olevan merkittävää vaikutusta suorituskykyyn.

[1] `https://sites.google.com/site/torcscreations/drivers`
[2] `http://www.speed-dreams.org/`

Kuten kerron luvussa 6.3, valitsin säteenseurantaheuristiikan koska se on suhteellisen helppotajuinen ja mutkaton toteuttaa. Lisäksi pidän sitä uskottavampana kuin Wymannin [51] oppaan ohjausmenetelmiä. Säteenseuranta saa kuitenkin botin ajamaan suorilla keskellä rataa, mistä syystä lisäsin toteutukseen loivuusheuristiikan. Se siirtää botin radan ulkolaidalle ennen kaarteita. Harkitsin myös Uusitalon ja Johanssonin [49] loivuusheuristiikan käyttämistä, mutta epäilin että sitä ei voi irroittaa muusta toteutuksesta. Lisäsin *Desideriuksen* menetelmiin ohjausvirheen, jotta botin ajaminen näyttäisi vähemmän kaavamaiselta, ja siten uskottavammalta. Säteenseuranta on *Desideriuksen* varsinainen ohjausmenetelmä. Muut lähinnä täydentävät sitä. En löytänyt säteenseurannalle kovin hyviä vaihtoehtoja. Esimerkiksi Uusitalon ja Johanssonin voimakenttämenetelmät ovat mielestäni liian monimutkaisia. Eräs mahdollisuus on käyttää kiintopisteheuristiikkaa valmiin ajolinjan kanssa. Tämä tosin muuttaisi *Desideriuksen* toteutusta merkittävästi. Vaihtoehtoa pohditaan tarkemmin luvussa 7.3.

Lisäsin *bt*-botista lainaamani ABS-, TCL-, vaihde- ja kytkinheuristiikat *Desideriuksen* koodiin, koska ne saattoi yhdistää vaivattomasti Wymannin [51] oppaasta kopioituihin tekniikoihin. Ajattelin myös, että tekniikat kohentaisivat hieman bottini suorituskykyä tai uskottavuutta. En etsinyt menetelmille vaihtoehtoja, koska en usko niiden ratkaisevasti vaikuttavan lopputulokseen. Pidän esimerkiksi sopivan ohjausheuristiikan valitsemista merkittävämpänä. Luvussa 6.6 kerron, miksi lisäsin nopeuden säätelyyn lisää portaita. Ohjelmoin kosmeettisen jarrutuksen, koska halusin viivästää *Desideriuksen* yli-inhimillisen välitöntä jarruttamista. Tein Quadfliegien et al. [33] artikkelin innoittamana tilakoneen ja rataluokittimen, koska niiden avulla uskoin helposti pystyväni parantamaan botin suorituskykyä.

Desideriuksen lähdekoodi sisältää *Tutorin* erikoistilanneheuristiikkoja, mutta en ole koittanut saada niitä toimimaan säteenseurannan, tilakoneen tai muiden lisäämieni menetelmien kanssa. Toisin sanoen nämä menetelmät eivät tällä hetkellä ole käytössä. Tässä tarkoitan erikoistilanteilla radalta suistumista ja esteisiin juuttumista. En yrittänyt yhdistää erikoistilanteiden menetelmiä muuhun toteutukseen, koska *Desiderius* pysyy testiradalla. Se ajaa kyllä ulos muilla radoilla, mutta mielestäni on hedelmällisempää pyrkiä välttämään suistumista kuin yrittää selvitä erikoistilanteita. Toisaalta jos *Desideriuksen* halutaan ajavan kilpailuita, on erikoistilanteista suoriuduttava, sillä radalta suistuminen on silloin todennäköisempää. Toinen kuljettaja voi esimerkiksi työntää botin tieltä.

7.1 Suorituskyky

Tässä luvussa tarkastellaan *Desideriuksen* suorituskykyä. Suorituskyvyllä tarkoitetaan sitä, kuinka nopeasti botti ajaa puhtaan kierroksen. TORCSissa puhdas kierros on sellainen, jolla auto ei leikkaa eli oio kaarteita eikä törmää reunavalleihin. Radalta suistuminen ei yksin johda kierroksen hylkäämiseen. Kaikki tässä luvussa esitettävät kierrosajat ovat tilanteesta, jossa botti ajaa yksin ilman kanssakilpailijoita, ellei toisin mainita. Luvussa arvioidaan siis vain botin nopeutta, ei sen kykyä ohittaa muita kuljettajia tai estää muita ohittamasta sitä.

Desideriuksen suorituskykyä arvioidaan sekä tarkastelemalla eri tekniikoiden vaikutusta sen kierrosaikojen kehitykseen että vertaamalla sitä muihin botteihin. Sitä verrataan myös eri pelaajien suoriutumiseen. Vertaaminen on haastavaa, sillä muut kuljettajat eivät välttämättä aja samalla autolla kuin *Desiderius*. Toisaalta ne voivat käyttää samaa ajoneuvoa, mutta eri asetuksilla. *Desiderius* käyttää autoa `stock1`, jonka säätöjä ei ole muutettu. En verrannut *Desideriuksen* suoriutumista omaan ajamiseeni, koska en osaa pelata TORCSia. En omista peliohjainta tietokoneelle, ja näppäimistöllä ja hiirellä ajaminen on turhan haastavaa. En myöskään usko muissa peleissä kellottamieni kierrosaikojen olevan vertailukelpoisia. Ensinnäkin eri peleissä ajetaan eri autoilla, ja toiseksi niiden ajomallit poikkeavat TORCSista.

Desideriuksen kehitystyön aikana tutkin suorituskyvyn kehittymistä vertaamalla botin eri versioiden ensimmäisen kierroksen aikaa. Ensimmäinen kierros on hitain, koska siinä auto lähtee paikoiltaan. Muut kierrokset botti aloittaa vauhdista. Nopein versio *Desideriuksesta* ajaa ensimmäisen kierroksen ajassa 2:09.05. Toisen ajassa 2:04.71. Tämän jälkeen kierrosaika vaihtelee vain muutamalla sadasosasekunnilla. Nopein versio ei kuitenkaan ole viimeisin. Viimeisin versio ajaa ensimmäisen kierroksen ajassa 2:11.21. Sen suurin ero nopeimpaan versioon nähden on varovaisempi tapa määrittää tavoitenopeus (ks. luku 6.6). Viimeisin versio ajaa kierroksen yli kaksi sekuntia hitaammin, mutta toisaalta se tekee *Desideriuksen* ajosta uskottavampaa. Edellä mainitut versiot soveltuvat vain testiradalle, muilla ne ajavat ulos. `Wheel2:en` lisäksi kokeilin ratoja `Aalborg`, `CG Speedway 1` ja `Forza`. Koska en kykene ratkaisemaan, mikä *Desideriuksen* versioista on paras, käytän tämän luvun vertailussa sekä nopeinta että viimeisintä.

Vuonna 2007 pidettiin ensimmäinen bottikilpailu, jossa käytettiin `wheel2`-rataa [45], eli *Desideriuksen* testirataa. Kilpailu oli *TORCS Endurance World Championship 2007*. Siinä ei käytetty TORCSin palvelin- vaan työpöytäversiota. *Polimi_2007_2* ajoi kilpailun hitaimman kierroksen ajassa 2:03.41. *Desideriuksen* nopeinkin versio olisi

jäänyt siitä yli sekunnin jälkeen. Kilpailussa *berniw* ajoi ajan 1:47.611. Vuoden 2007 jälkeen kilpailun kierrosajat ovat entisestään lyhentyneet. `Polimi_2007_2`-botin toteutus pohjautuu pitkälti *bt*-bottiin. Sen merkittävin ero on, että se käyttää eri heuristiikkaa vaihteen vaihtamiseen. Ymmärtääkseni heuristiikka on muodostettu koneoppimisen avulla. En kuitenkaan löytänyt `Polimi_2007_2`-botin toteutuksesta muuta dokumentaatiota kuin sen lähdekoodin. Sekin on italiaksi kommentoitu.

Kun olin ohjelmoinut *Desideriukseen* kaikki *Tutorin* menetelmät, pois lukien luvun 7 johdannossa mainitut, sen ensimmäisen kierroksen aika oli 2:13.74. Wymannin [51] mukaan *Tutor* on kilpailukykyinen. Toisaalta hän ei testannut sitä `wheel2`-radalla. *Tutoriin* pohjautuva *Bt*-botti puolestaan ajoi säännönmukaisesti ulos testiradan neulansilmästä, eli kaarteesta 11. En siis voinut verrata *Desideriusta* siihen. Lisäämällä bottini toteutukseen menetelmiä muista lähteistä sekä toteuttamalla joitakin itse (ks. kuva 6.1) sain kierrosaikaa lyhennettyä puolitoista sekuntia. Suurimmat vaikutukset suorituskykyyn oli säteenseurannalla, rataluokittimella ja tilakoneella sekä porrastetulla nopeuden säätelyllä. Luvuissa 6.6 ja 6.8 selostetaan tilakoneen ja nopeuden säätelyn eri versioiden vaikutuksia kierrosaikaan.

Muñoz et al. [27] kehittivät TORCSiin botin, joka opetteli ajamaan takaisinkytketyn neuroverkon avulla. Neuroverkko harjoitettiin toisten bottien ja pelaajien ajosta kerätyn datan avulla. Sen syötteitä olivat auton nopeus, etäisyys radan reunoihin, asento radan tangenttiin nähden, säteenseurantasensorit, moottorin käynti- ja renkaiden pyörimisnopeus sekä valittu vaihde. Sen tulosteita olivat TORCSin aktuaattoreiden arvot. Muñoz et al. botin nopein versio kulki `wheel2`-radan ajassa 02:37.85. Se opetettiin Simmersonin [22] botin datalla. Simmersonin botti voitti WCCI-konferenssin kilpailun vuonna 2008. Kilpailussa ei ajettu `wheel2`-rataa. Muñoz et al. kokeissa Simmersonin botti suistui tältä radalta. Ryhmä kuitenkin arvioi sen kierrosajaksi 2:41.87. Muñoz et al. vertasivat opetettua bottiaan myös ohjelmoituun. Ohjelmoitu botti määritti tavoitenopeutensa säteenseurannan avulla samaan tyyliin kuin *Cobostar* (ks. luku 5.1). Tavoitenopeuden ja tämänhetkisen nopeuden perusteella se päätteli, tulisiko sen kiihdyttää tai jarruttaa. Ohjelmoitu botti ajoi nopeimman kierroksensa aikaan 3:11.52.

Muñoz et al. [27] vertasivat opetettua bottiaan myös ihmispelaajaan. Hän läpäisi radan parhaimmillaan ajassa 2:16.92. Pelaaja tosin pyrki ajamaan keskellä tietä sekä kiihdyttämään ja jarruttamaan kevyesti — toisin sanoen hitaammin kuin mitä osaisi. Muñoz et al. käyttivät tätä ”tasoitusta”, koska heidän mukaansa ihminen havainnoi peliympäristöä huomattavasti monipuolisemmin kuin botti. Tulosten pe-

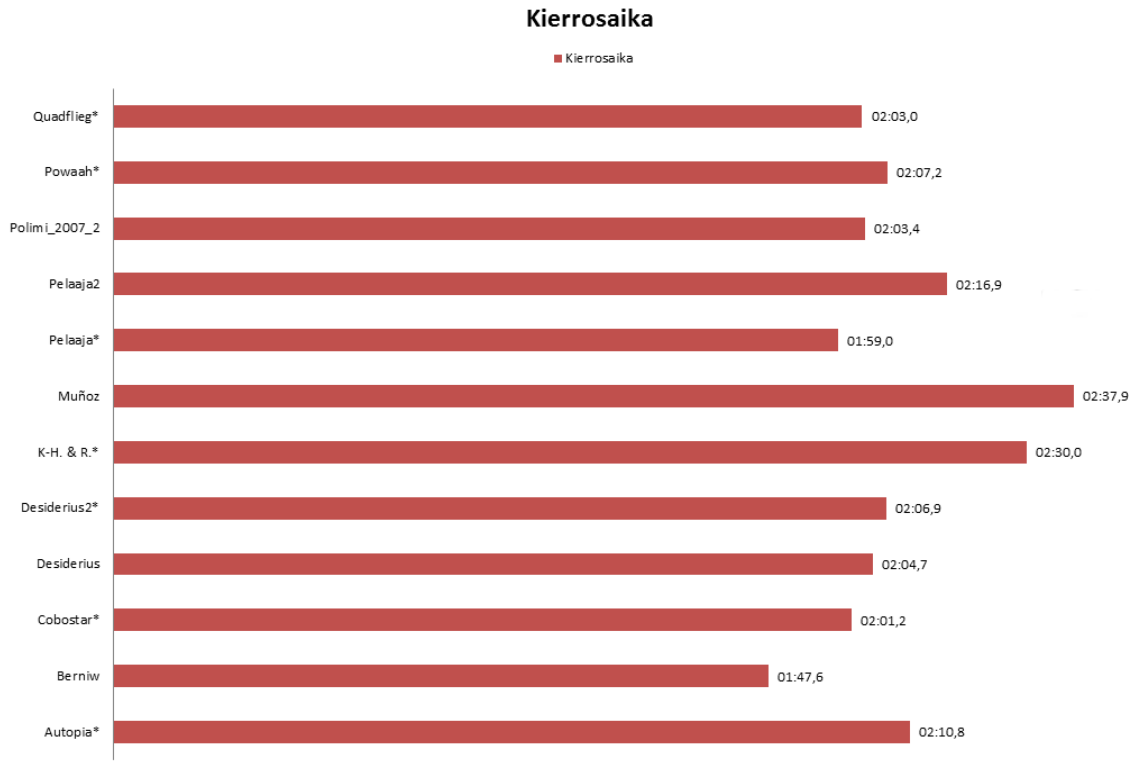
rusteella *Desiderius* on `wheel2`-radalla nopeampi kuin Muñozin et al. opetettu ja ohjelmoitu botti. Ihmispelaajan ero *Desideriukseen* on kuitenkin mielestäni pieni, kun otetaan huomioon, että tämä ajoi tahallisen hitaasti.

Luvussa 6.8 kerrotaan, kuinka lainasin rataluokittimen ja tilakoneen idean Quadfliegin et al. [33] toteutuksesta. Toisin kuin *Desiderius*, heidän bottinsa opetteli ajamaan radasta muodostetun mallin avulla. He eivät tuloksissaan ilmoita kuin kokonaisajan, joka botilta kului viiden kierroksen ajamiseen radalla `wheel2`. Paras versio Quadflieg et al. botista ajoi kokonaisajaksi 10:15. Sen kierrosajaksi on täten karkeasti arvioiden 2:03.00. Quadflieg et al. vertasivat bottiaan myös *Cobostariin* [3], Kinnaird-Heetherin ja Reynoldsin [22] bottiin sekä ihmispelaajaan. Kaikkein nopein oli pelaaja kokonaisajalla 9:55. Quadfliegin et al. mukaan hän oli taidoiltaan keskinkertainen. *Cobostarin* kokonaisaika oli vain muutamia sekunteja lyhyempi kuin Quadflieg et al. botin. Se oli vertailun nopein tekoälykuljettaja. Kinnaird-Heetherin ja Reynoldsin botti oli selvästi hitain ajalla 12:30. Se käytti verrattain yksinkertaisia, ohjelmoituja heuristiikkoja. Quadflieg et al. vertailussa *Desiderius* olisi ollut toiseksi hitain. Sen nopeimman version kokonaisaika olisi ollut hieman alle 10:30. Ero pelaajaan olisi kasvanut joka kierroksella noin seitsemän sekuntia.

Luvussa 5.1 esiteltiin Uusitalon ja Johanssonin [49] botti, joka käyttää auton ohjaamiseen erilaisia voimakenttiä. Se päättelee myös sopivan nopeuden niiden avulla. Uusitalo ja Johansson vertaavat *Powaah*-bottiaan *Cobostariin* ja *Autopiaan*, joista jälkimmäinen voitti IEEE Computational Intelligence in Games -konferenssin kilpailun vuonna 2010. Heidän mukaansa *Cobostar* on varsin kilpailukykyinen nopeilla radoilla, mutta ajaa haastavammilla melko varovaisesti. Uusitalon ja Johanssonin mielestä `wheel2` on haastava rata. He pitävät *Autopiaa* erittäin nopeana kuljettajana ja yhtenä bottinsa varteenotettavimmista kilpailijoista. Uusitalo ja Johansson ilmoittavat tuloksissaan vain kolmen kierroksen kokonaisajan. *Powaahin* aika `wheel2`-radalla oli 6:21,50, *Autopian* 6:32.50 ja *Cobostarin* 7:36.60. *Powaahin* kierrosaika oli arviolta 2:07.17, *Autopian* 2:10.08 ja *Cobostarin* noin 2:32.30. *Cobostarin* ajassa on huomattava ero Quadfliegin et al. [33] tuloksiin verrattuna (ks. yllä). Eräs eroa selittävä tekijä on se, että Uusitalon ja Johanssonin testeissä botit ajoivat viiden kierroksen sijasta kolme. Lyhyemmässä kisassa ensimmäisen kierroksen aiheuttama lisä kokonaisaikaan voi korostua. Uusitalon ja Johanssonin vertailussa *Desiderius* olisi ilmoitettujen aikojen perusteella ollut kilpailukykyinen.

Taulukko 7.1 listaa tässä luvussa esiteltyjen bottien kierrosaikoja. Siinä *-merkki nimen perässä tarkoittaa arviota. On huomattava, että *berniw*, *Desiderius*, *Desiderius2*

ja *polimi_2007_2* ajoivat aikansa TORCSin työpöytäversiolla, kun taas muut käyttivät palvelinversiota. Taulukossa *Desiderius* vastaa bottini nopeinta ja *Desiderius2* sen viimeisintä versiota. *Pelaaja* on lainattu Quadfliegin et al. [33] ja *Pelaaja2* Muñozin et al. [27] tuloksista. *Cobostar* on Quadfliegin et al. artikkelista. Vertailun boteista vain *berniw* käyttää ajolinjoja. Kaiken kaikkiaan *Desideriuksen* ajat ovat alempaa keskitasoa, mutta työpöytäversion boteista se on hitain. Koska se ei taulukon perusteella menesty kovin hyvin edes kokeellisten bottien joukossa, siitä tuskin on riittävää vastusta pelaajille. Toisaalta näiden kokeellistenkin bottien on tarkoitus ollut olla kilpailukykyisiä, sillä ne ovat joko osallistuneet kilpailuihin, tai niitä on artikkeleissa verrattu kilpailuissa menestyneisiin botteihin. Eräs syy *Desideriuksen* heikkoon suorituskykyyn voi olla ratavalinta. Quadflieg et al. mukaan valittu testirata `wheel2` on haastava, ja heidän kokeissaan yksikään botti ei pysynyt sillä pelaajan vauhdissa. Myös Butzin ja Lönnekerin [3] mielestä rata on haastava. Toisaalta valittu rata testaa bottia kattavasti, sillä se sisältää monipuolisesti erilaisia osuuksia. Sillä ei kuitenkaan ole merkittäviä korkeuseroja.



Kuva 7.1: Yhteenveto luvussa 7.1 esiteltyjen bottien kierrosajoista. *-merkki nimen perässä tarkoittaa arviota.

Desideriusta hidastaa eniten se, että se määrittää nopeutensa radan geometrian perusteella. Sen suorituskykyä voidaan kuitenkin edelleen parantaa, vaikka se ei käyttäisi ajolinjoja. Eräs vaihtoehto on lisätä nopeuden säätelyyn portaita, sillä botin ajaminen on vielä melko ehdotonta. Toisaalta kaikki *Desideriuksen* heuristiikat käyttävät varsin yksinkertaisia laskennallisia malleja, joten nopeuden säätelyä on mahdollista tarkentaa. *Desideriuksen* tulee myös paremmin huomioida autonsa ominaisuudet ja säädöt. Tällä hetkellä se ei osaa hyödyntää ajoneuvonsa suorituskykyä. Lisäksi rataluokitinta ja tilakonetta voi vielä parantaa. Kuten luvussa 6.8 kerrotaan, luokittimeni tuntee vain kolme erilaista osuutta.

Vaikka *Desiderius* ei käytä Wymannin [51] erikoistilanneheuristiikkoja, käsittelen tässä niiden suorituskykyä, koska arvion niiden olevan sovitettavissa bottini toteutukseen. Lisäksi testasin näitä menetelmiä kehitystyön aikana. Niiden yhteensovittamisessa on aluksi ratkaistava, milloin *Desiderius* käyttää erikoistilanteiden heuristiikkoja, ja milloin tavanomaisia. Normaaliajamiseen on turvallisinta siirtyä vasta, kun auto on varmasti kokonaan radalla. Kuten luvussa 6.3 kerrotaan, radan reunan päällä ajaminen aiheuttaa ongelmia säteenseurantaheuristiikalle. Toisaalta mitä varovaisemmin erikoistilanteissa toimitaan, sitä hitaammin botti palaa normaaliajoon.

Wymannin [51] heuristiikat antavat melko hyvät valmiudet selvitä erikoistilanteista. Niissä on kuitenkin puutteensa. Testatessani näitä menetelmiä havaitsin, että botti ei aina pääse esteistä irti. *Desiderius* saattoi esimerkiksi juuttua reunavalliin suistuttuaan kaarteesta 10. Toisaalta se ohjasi joskus takaisin radalle oudon jyrkässä kulmassa. Pahimmillaan botti ajoi jopa hieman tulosuuntaan. Arvelen ongelman johtuvan siitä, että botti tulkitsee radan ulkopuolella sijainnikseen sen segmentin, jolta suistui ulos. Se ei siis yritä palata geometrisesti lähimmälle ratasegmentille. Lisäksi *Desiderius* peruutti joskus turhan vauhdikkaasti irti esteistä. Se ehti valua pitkän matkaa taaksepäin, ennen kuin alkoi jälleen kulkea eteenpäin. Wymannin menetelmät eivät myöskään huomioi, että kitka radan ulkopuolella on pienempi. Tästä syystä *Desiderius* pyörähteli joskus nurmikolla kiihdyttäessään liian voimakkaasti.

7.2 Uskottavuus ja tekotyperyys

Tässä luvussa arvioidaan sekä yksittäisten tekniikoiden että *Desideriuksen* toteutuksen uskottavuutta kokonaisuutena. Mielestäni agenttien tehtävä on TORCSissa simuloida ihmisvastustajaa. Toisin sanoen ne ovat botteja [15, 19]. Perustelen näkemykseni sillä, että pelin yhteisö puhuu ”boteista” [20, 45, 51]. Lisäksi Quadflieg et

al. [33] ja Muñoz et al. [27] ovat pyrkineet ohjelmoimaan TORCSiin botin. *Desideriuksen* uskottavuutta on siis arvioitava sen mukaan, kuinka hyvin se simuloi pelaajaa. Sen ei kuitenkaan tarvitse imitoida taitavaa pelaajaa ollakseen uskottava [48]. Tässä luvussa arvioin *Desideriuksen* uskottavuutta luvussa 5 esitettävien kriteerien [28, 41] avulla. Kerron myös botista saamastani palautteesta, sekä vertaan sitä *Tutoriin*. *Tutor* ei ole erityisen uskottava botti, mutta tarjoaa hyvää vertailukohdan, koska voin täsmällisesti arvioida *Desideriukseen* lisäämieni menetelmien vaikutuksia. Selostan lisäksi, miten kokeilemani tekotyperät tekniikat vaikuttavat bottini uskottavuuteen.

Luvussa 5.4 esitellään Muñoz et al. [28] minimikriteerit uskottavalle autopelin agentille. *Desiderius* ei täytä niistä yhtäkään. Se ei noudata minkäänlaista ajolinjaa, eikä väistä toisia autoja. Se ei kykene ohittamaan suorilla eikä kaarteissa, eikä myöskään pyri estämään toisia ohittamasta sitä. Se ei edes havainnoi sensoreillaan toisia autoja. Tällä hetkellä botti ei myöskään selviä erikoistilanteista. Muñozin et al. listan perusteella *Desiderius* ei siis ole lainkaan uskottava botti. Monet *Desideriuksen* menetelmistä tekevät sen ajosta uskottavamman näköistä, mutta eivät paranna sen suorituskykyä samassa suhteessa. Säteenseuranta- ja loivuusheuristiikka yhdessä tilakoneen avulla saavat *Desideriuksen* kulun muistuttamaan klassista ajolinjaa. Ne eivät kuitenkaan nopeuta sitä yhtä paljoa kuin mitä ajolinjan seuraamisen pitäisi. Menetelmien hyöty on kyseenalainen, sillä näyttävästä ajamisesta ei ole iloa, jos botti ei pysty kilpailemaan pelaajan kanssa. Uskottavuus ei ilmene pelaajalle, jos botti ajaa kaukana jäljessä.

Koska *Desiderius* ei täytä Muñozin et al. [28] kriteerejä, Togeliuksen et al. [41] esittämät korkeamman tason vaatimukset eivät kovin hyvin sovellu sen arviointiin. Täydennän kuitenkin tässä bottini arvostelua niiden avulla. Taulukon 7.1 perusteella *Desiderius* ei tarjoa pelaajalle riittävää haastetta. Tosin on haastavaa arvioida, kuinka taitavia siinä esitetyt pelaajat ovat. Quadfliegin et al. [33] mukaan *Pelaaja* on taidoiltaan keskinkertainen. Muñozin et al. [28] *Pelaaja2* taas ajoi tahallaan hidastellen. *Desideriuksen* nopein versio sijoittuu näiden väliin. Se on 12,2 s nopeampi kuin *Pelaaja2*, mutta 5,7 s hitaampi kuin *Pelaaja*. *Desiderius* voi kierrosaikojensa puolesta kelvata yhdeksi kisan häntäpään ajajista. Se ei kuitenkaan varioi tarjoamaansa haastetta, vaan ajaa kierroksensa miltei identtisesti. *Desideriuksen* taidot eivät myöskään muutu kilpailusta toiseen.

Esitin videon *Desideriuksen* viimeisimmän version ajosta Instanssi 2013-tapahtumassa[3], ja pyysin yleisöltä palautetta siitä. Videolla botti ajaa yhden kokonai-

[3]http://instanssi.org/2013/ohjelma/

sen kierroksen testiradalla. Esityksen jälkeen yleisö sai kommentoida videota sekä paikan päällä että IRC-keskustelun välityksellä. Lisäksi video on ollut tapahtumasta lähtien katsottavissa YouTube-palvelussa[4]. Siellä sitä on katsottu 44 kertaa 19.11.2014 mennessä. Kaiken kaikkiaan en saanut paljoakaan palautetta *Desideriuksesta*. Erään yleisön jäsenen mielestä sen ajo näytti ”hyvältä”, mutta hän ei eritellyt sen tarkemmin, mikä siinä miellytti. Toista taas häiritsi botin ajossa esiintyvä nykiminen. Tällä hän saattoi viitata joko säteenseurannan aiheuttamaan nykimiseen tai ohjausvirheen aiheuttamiin pieniin korjausliikkeisiin. Joidenkin mielestä *Desiderius* ajoi liiaksi radan reunalla.

Desideriuksen toteutuksen taso on niin heikko, että se ei ole hyvä alusta tekotyperien tekniikoiden testaamiselle. Ei ole mielekästä yrittää parantaa uskottavuutta erilaisin silmänkääntötempuin, kun botin minimivaatimuksetkaan eivät täyty. Vaikka niiden täyttyminen ei välttämättä vielä tarkoita, että botti olisi uskottava, se ei ainakaan voi olla sitä, jos ne eivät [18]. Kokeilin kuitenkin ohjelmoida joitakin tekotyperiä menetelmiä. Parhaita näistä ovat ohjausvirhe ja kosmeettinen jarrutus. Ohjausvirhe lisää botin kulkuun hitusen vaihtelua. Sen aiheuttama variaatio ei kuitenkaan välttämättä paranna uskottavuutta. Kosmeettinen jarrutus on tekotyperistä tekniikoistani onnistunein. Mielestäni se parantaa uskotavuutta. Muilla kokeiluillani ei ollut minkäänlaista vaikutusta, tai ne joko hidastivat kierrosaikaa liikaa tai heikensivät uskottavuutta.

Eräs epäonnistunut tekotyperä tekniikka oli virheen lisääminen kiihdytys- ja jarrutusaktuaattoreihin. Joka kerta, kun *Desiderius* muutti jommankumman aktuaattorinsa arvoa, siihen lisättiin virhe $\epsilon \in [-0{,}2, 0{,}2]$. Virheen summaamisen jälkeen arvo normalisoitiin välille $[0, 1]$. Tämä tekniikka ei muuttanut sitä, miltä botin ajo näyttää, ja se hidasti kierrosaikaa monta sekuntia. Virheiden arvonnassa käytettiin lineaarista jakaumaa. Toisin sanoen suurella virheellä oli yhtä suuri todennäköisyys ilmaantua kuin pienellä. Toinen epäonnistuminen oli viiveen lisääminen botin toimiin. Tällöin se sääti aktuaattoreitansa vain joka viides simulaatiojakso, eli noin 100 ms välein. Tämä sai *Desideriuksen* suistumaan ulos heti testiradan ensimmäisestä kaarteesta, koska se jarrutti liian myöhään. En löytänyt valmiita tekotyperiä tekniikoita, joita olisin voinut käyttää toteutuksessani.

Säteenseurantaheuristiikka saa botin ajamaan lyhintä reittiä radan ympäri (ks. luku 5.1). Lyhin reitti muistuttaa klassista ajolinjaa siinä, että auto kulkee kaarteiden taitekohtien kautta. Toisaalta suorilla botti kulkee keskellä tietä. Se ei myöskään

[4]http://www.youtube.com/watch?v=FvSC-SRZn1M

pyri siirtymään radan ulkolaidalle ennen kaarretta tai kaarteen taitekohdan jälkeen. Säteenseuranta on mielestäni realistisimmillaan testiradan kaarteissa 16 ja 17, jotka ovat peräkkäisiä, erisuuntaisia kaarteita. Menetelmää hyödyntävän botin reitti erooa kuitenkin vielä silminnähtävästi klassisesta ajolinjasta. Säteenseuranta saa *Desideriuksen* suunnan myös aika ajoin nykimään. Toisin sanoen botti tekee pieniä korjausliikkeitä puolelta toiselle. Ilmiö korostuu pitkissä ja loivissa käännöksissä, kuten kaarteissa 12 ja 14. *Desideriuksen* pyörät myös aika ajoin ylittävät radan reunan. Kuten luvussa 6.5 kerroin, en onnistunut ratkaisemaan tätä ongelmaa. Ilmiö ei kuitenkaan aiheuta kaarteiden oikomista. Eräs mahdollisuus muuttaa säteenseurantaa on siirtää radan reunoja marginaalin m verran keskemmälle leikkauspisteitä etsittäessä. Marginaaliksi m sopii esimerkiksi puolet auton leveydestä $\frac{w}{2}$. Muutos voi vähentää reunalla ajamista.

Säteenseurantaheuristiikka parantaa *Desideriuksen* uskottavuutta Wymannin [51] keskihaku- ja kiintopisteheuristiikkoihin nähden. Edellinen antaa vaikutelman ikäänkuin auto kulkisi kiskoilla tai jonkinlaisessa kourussa. Se ei sisällä lainkaan ennakointia, vaan botti ohjaa autoa vain radan tämänhetkisen muodon perusteella. Tämä tekee liikkumisesta kulmikasta ja nykivää. Keskellä ajaminen ei myöskään muistuta lainkaan klassista ajolinjaa. Keskihakuheuristiikka on siinä määrin uskottava, että botti pysyy radalla ja ajaa puhtaan kierroksen. Renkaat eivät myöskään käy radan ulkopuolella. Kiintopisteheuristiikka pyrkii myös pitämään auton keskellä tietä. Se kuitenkin sisältää ennakointia, sillä botti ohjaa kohti edelle asetettua kiintopistettä. Menetelmä on uskottavampi kuin keskihakuheuristiikka. Radan muotojen ennakointi tekee botin kulusta jatkuvampaa ja vähemmän kulmikasta. Auton liikkeet vaikuttavat vähemmän tien geometriaan pakotetuilta, eikä vaikutelma kiskoilla kulkemisesta ole yhtä vahva. Botti kulkee kuitenkin yhä enimmäkseen keskellä tietä. Lisäksi renkaat käyvät joskus ruohikolla.

Loivuusheuristiikka tekee *Desideriuksesta* uskottavamman, sillä menetelmä saa botin kulun muistuttamaan enemmän klassista ajolinjaa. Kuten luvuissa 6.4 ja 6.5 kerrotaan, menetelmä toimii vain riittävän pitkillä suorilla. *Desiderius* käyttää siis kaarteissa säteenseurantaa. Tämä haittaa uskottavuutta, sillä nyt botti ajaa esimerkiksi kaarteet 1 ja 2 koko matkalta pitkin radan sisäreunaa. Ajolinjaa noudattava botti siirtyisi kaarteiden välissä lähemmäs radan ulkolaitaa. Säteenseurannan ja loivuusheuristiikan eriaikaisuus ei sinänsä haittaa, sillä sain *Desideriuksen* vaihtamaan menetelmästä toiseen melko sulavasti. *Desiderius* siirtyy säteenseurannasta loivuusheuristiikkaan heti kun se saapuu kaarteesta suoralle. Mielestäni se, että botti siir-

tyy radan ulkolaidalle hyvissä ajoin ei haittaa sen uskottavuutta. Toisaalta voisi olla uskottavampaa, jos *Desiderius* varautuisi seuraavaan kaarteeseen vasta, kun se on näkyvissä.

Desideriuksen nopeuden säätelyn tekniikoista lähinnä porrastus ja kosmeettinen jarrutus parantavat sen uskottavuutta *Tutoriin* nähden. Myös varovaisempi tapa määrittää tavoitenopeus kohentaa sitä. Näiden menetelmien tavoitteena on saada *Desideriuksen* toimet näyttämään tietoisilta, harkituilta päätöksiltä. Ne eivät kuitenkaan täysin ratkaise nopeuden säätelyn ehdottomuutta. Ehdottomuus ilmenee pelaajalle esimerkiksi siten, että botti vilkuttaa jarruvalojaan taajaan ja jättää tiehen renkaan jälkiä. Auto myös liirtää ajoittain lukkojarrutuksessa.

Nopeuden säätelyn porrastus tekee ajamisesta harkitumpaa, koska sen ansiosta *Desiderius* ei heti ala jarruttaa, kun tavoitenopeus ylittyy, vaan lakkaa ensin vain kiihdyttämästä. Toisaalta tavoitenopeus ei ylity niin helposti, koska porrastus saa *Desideriuksen* lopettamaan maksimikiihdytyksen aikaisemmin kuin *Tutor*. Kosmeettinen jarrutus vähentää jarruvalojen välkkymistä, mutta se pidentää jarrutusmatkaa. *Desideriuksen* ABS- ja TCL-tekniikat eivät hieman yllättäen vaikuttaneet uskottavuuteen. Vaihde- ja kytkinheuristiikat eivät myöskään aiheuttaneet silminnähtävää muutosta. Jälkimmäinen tosin paransi liikkeelle lähdön sulavuutta.

En ohjelmoinut rataluokitinta tai tilakonetta *Desideriuksen* uskottavuutta silmälläpitäen, vaan pyrin parantamaan niillä botin suorituskykyä. Ne kuitenkin lisäsivät uskottavuutta välillisesti, sillä niiden avulla sain esimerkiksi yhteensovitettua säteenseuranta- ja loivuusheuristiikan. Kuten luvussa 6.8 kerrotaan, päättelin Quadfliegin et al. [33] artikkelista, että botin ei tarvitse hidastaa ennen loivia kaarteita. Tämän päätelmän vaikutus *Desideriuksen* uskottavuuteen on ristiriitainen, koska osoittautui, ettei se ole yleispätevä. Joillakin radoilla muutos parantaa botin suorituskykyä merkittävästi, kun taas toisilla se saa auton suistumaan. Luokittimen avulla *Desiderius* myös kiihdyttää loivissa kaarteissa voimakkaammin, mikä saa sen ajautumaan useammin radan reunalla. Vaikutus korostuu tilakoneen `F_CA_F`- ja `FULL`-tiloissa. Toisaalta *Desiderius* ei nyt tiloissa `STR_CA_FULL` ja `F_CA_F` jarruta lainkaan, mikä vähentää jarruvalojen vilkkumista.

7.3 Vaihtoehtoiset tekniikat ja toteutuksen täydentäminen

Tässä luvussa pohditaan, mitä vaihtoehtoisia tekniikoita *Desideriuksen* toteutuksessa voisi käyttää nykyisten sijaan. Lisäksi esitetään, miten toteutusta tulee täydentää,

jotta *Desideriuksesta* tulisi uskottavampi. Luvussa tutkitaan esimerkiksi, mitä menetelmiä lähdekoodiin tulee lisätä, jotta botti täyttäisi Muñozin et al. [28] kriteereitä.

Mielestäni *Desideriuksen* tämänhetkistä toteutusta ei kannata jatkokehittää. Arvelen, että lokaaleja menetelmiä hiomalla on vaikea saavuttaa yhtä suorituskykyistä bottia kuin käyttämällä ajolinjoja. Helpoiten *Desideriuksen* saa seuraamaan ajolinjaa muuttamalla kiintopisteheuristiikkaa. Millingtonin ja Fungen [26] mukaan menetelmää voi käyttää myös siten, että kiintopiste haetaan ajolinjalta. Samalla pitää muuntaa tapa, jolla *Desiderius* määrittää tavoitenopeutensa. Nopeus tulee laskea ajolinjan, ei kaarteen säteen perusteella. Ajolinjojen käyttäminen muuttaisi osan toteutuksesta epäyhteensopivaksi. Ainakin säteenseurannasta ja loivuusheuristiikasta jouduttaisiin luopumaan. Rataluokitin ja tilakone voivat muunneltuina olla käyttökelpoisia. Ohjausvirhettä, vaihde- ja kytkinheuristiikkaa tai nopeuden säätelyn menetelmiä muutos ei haittaisi. Erikoistilanteiden heuristiikkojen kohdalla olisi ratkaistava, milloin botti käyttää niitä, ja milloin se palaa seuraamaan ajolinjaa.

Ajolinjoja käytettäessä tärkeimmäksi ratkaistavaksi kysymykseksi jäisi, miten ne muodostetaan. Tuleeko ne laatia etukäteen, vai laskeeko *Desiderius* ne reaaliaikaisesti, kuten *berniw*? Jos ne määritetään ennalta, *Desiderius* lakkaa olemasta yhteensopiva, kun TORCSiin lisätään uusia ratoja. Mikäli linjat laaditaan käsin, tarvitaan jonkinlainen työkalu niiden piirtämiseksi. Tällainen voisi olla esimerkiksi räätälöity *plug-in* eli lisäosa johonkin vektorigrafiikkaohjelmaan. Lisäosa lukisi TORCSin ratojen tiedot XML-tiedostoista ja piirtäisi ne pohjaksi grafiikkaohjelmaan. Kun ajolinjan spline-viiva olisi laadittu, lisäosa tallentaisi sen botin ymmärtämään muotoon. Ajolinjan laatiminen koneellisesti optimoimalla vaatisi myös työkalun, joka voi lukea ja kirjoittaa pelin tiedostoja. Valmiit ajolinjat *Desiderius* voi lukea samalla kuin TORCS lataa bottimoduuleita. Ajolinjan reaaliaikainen laskeminen monimutkaistaa botin toteutusta, mutta tekee siitä joustavamman. *Berniw* latii ajolinjansa reaaliaikaisesti, ja se on varsin nopea botti (ks. kuva 7.1). Sen ajaminen näyttää melko uskottavalta, mutta ajoittain se ohjautuu turhan jäykästi.

Botti ei kuitenkaan saa noudattaa ajolinjaa liian orjallisesti, sillä virheettömyys tekee sen ajamisesta epäuskottavaa [48]. Nykyisellään ohjausvirhe ei lisää *Desideriuksen* kulkuun riittävästi variaatiota, joten ajolinjoja varten on kehitettävä uusi menetelmä, joka saa auton reitin ”elämään”. Eräs mahdollisuus on laatia valmiiksi tekotyperiä eli tarkoituksellisesti epäoptimaalisia ajolinjoja. Näiden lisäksi tarvitaan heuristiikka, joka yhdistelee näitä linjoja. Tekotyperän botin tulee myös tehdä ajoittain suoranaisia ajovirheitä ja suistua radalta [44]. Pelin viihdyttävyyttä voi paran-

taa, jos botti tekee virheitä sitä todennäköisemmin, mitä kauemmin pelaaja ajaa sen kintereillä. Tällöin peli palkitsisi ajotaidoista, kylmähermoisuudesta ja sinnikkyydestä.

Desideriuksen suurimpia puutteita on se, ettei se reagoi toisiin autoihin. Muñoz et al. [28] pitävät väistämistä, ohittamista ja sijoituksen puolustamista uskottavan agentin perustaitoina. *Desideriuksen* säteenseurantaheuristiikan voi melko helposti muuttaa sellaiseksi, että radan reunojen lisäksi se tarkkailee toisia autoja. Säteet voi TORCSin työpöytäversiossa myös ohjelmoida ilmoittamaan, minkä auton ne kohtaavat. Täten botti voi esimerkiksi reagoida eri tavoin pelaajaan kuin toisiin agentteihin. Toisaalta arvelen, että ohituskäyttäytyminen on helpompaa yhdistää kiintopisteheuristiikkaan. Sen kanssa voi käyttää *Tutorin* valmiita menetelmiä ohittamiseen ja väistämiseen. Kiintopisteheuristiikan avulla valitun ohituskäyttäytymisen voi myös yhdistää ajolinjan noudattamiseen. *Tutorin* toteutuksessa ei kuitenkaan ole menetelmää, jolla botti estäisi toisia ohittamasta sitä. Tätä varten *Desideriuksen* toteutukseen on kehitettävä kokonaan uusi menetelmä, tai etsittävä valmis jonkin muun botin koodista.

Tällä hetkellä *Desiderius* ei käytä mitään erikoistilanteiden heuristiikkoja. Luvussa 7.1 pohditaan, miten sen koodissa olevat *Tutorin* menetelmät voidaan ottaa käyttöön. Koska nämä tekniikat ovat melko toimivia, niitä kannattaa pyrkiä hyödyntämään, ennen kuin etsii vaihtoehtoisia heuristiikkoja. Wymann [51] tosin itsekin huomauttaa, että *Tutorin* menetelmissä on puutteensa.

7.4 Johtopäätökset

Tämän pro gradu -työn päämääränä oli tutkia agentin uskottavuuden ja tekotyperyyden ilmenemismuotoja sekä niiden toteuttamista realistisessa autopelissä. Se epäonnistuu tehtävissään. Ensinnäkin *Desideriuksen* tekninen taso on niin heikko, ettei se sovellu alustaksi uskottavien tai tekotyperien menetelmien kehittämiseen. Toteutus ei myöskään esittele mitään uutta. Sen käyttämät tekniikat ovat triviaaleja ja epävarmoja viritelmiä, jotka eivät ole luotettavia edes TORCSin sisällä. Teos ei tarjoa parempaa lähtökohtaa jatkokehittämiselle kuin Wymannin [51] opas, koska jälkimmäinen esittelee sekä monipuolisempia että helpommin muunneltavia menetelmiä. Toisaalta tämä opinnäyte varoittaa TORCSin keskeneräisyydestä. Tutkielman ydinaihetta pohjustava teoreettinen osuus luvussa 5 on kohtalaisen onnistunut. Se kartoittaa hyvin olemassa olevaa kirjallisuutta. Luvun 5.4 oma pohdinta on melko

vapaamuotoista.

Koska pro gradun teknisen osuuden lopputulos on niin heikko, työ ei voi antaa tyydyttävää vastausta tutkimuskysymykseensä. Luku 7 keskittyykin lähinnä yksityiskohtien kaluamiseen ja ”tekee tikusta asiaa”. Opinnäyte olisi voinut onnistua paremmin, jos tutkimuskysymys olisi tarkemmin rajattu. Tutkimus olisi voinut selvemmin valita tarkasteltavaksi joko akateemisen tutkimuksen tai kaupallisten pelien botit. Nyt teos käsittelee hieman molempia, mikä tekee siitä epämääräisen, sillä tutkijoiden ja pelinkehittäjien käsitykset uskottavasta agentista poikkeavat toisistaan [48]. Lisäksi heillä on agentteja kehittäessään usein eri päämäärät. Tajusin vasta verrattain myöhään *botin* käsitteen epämääräisyyden sekä näkemyserot agentin tehtävästä autopelissä. Myös tämä haittasi rajausta.

Luvussa 7.2 esitetyt Muñozin et al. [28] kriteerit olisi pitänyt ottaa vahvemmin *Desideriuksen* toteutuksen lähtökohdaksi. Kehitystyön johtavana kysymykseni olisi pitänyt olla: ”edistääkö tämä tekniikka kriteereiden täyttymistä?” Vasta lopuksi olisi voinut pohtia uskottavuuden lisäämistä tekotyperyyden avulla tai muuten. Botin toteutus eteni aina helpoimman kautta. Tämä ilmenee esimerkiksi siitä, että pelkästään arvailen eri tekniikoiden ongelmien syitä, enkä pyri selvittämään tai ratkaisemaan niitä perinpohjaisesti.

Koska tämä pro gradu ei tarjoa uutta teoriaa eikä uusia tuloksia tai tekniikoita, se ei tarjoa aiheita jatkotutkimukselle. Autopeleissä ja niiden agenteissa on vielä tutkittavaa, mutta tämä teos ei anna hyvää lähtökohtaa uusille opinnäytteille. Joistakin esittämistäni ideoista voi olla esimerkiksi ohjelmointityön aiheeksi. Tällainen voisi olla työkalu botin ajosta kerätyn datan visualisoinniksi, jonka kuvaan luvussa 6.1. Toinen mahdollinen aihe on ohjelma ratojen ajolinjojen piirtämiseksi, josta kerron luvussa 7.3. TORCSin työpöytäversion teknisten ongelmien vuoksi en kuitenkaan voi varauksemattomasti suositellä sitä alustaksi muille töille.

8 Yhteenveto

Tämä opinnäyte tutkii agenttien uskottavuutta realistisessa autopelissä. Työ käsittelee lisäksi tekotyperyyttä, sekä sen yhteyttä uskottavuuteen. Tutkimuksen lähtökohdat ja tutkimuskysymys esitellään luvussa 1. Pro graduni aihe on jatkoa omalla kandidaatintutkielmalleni [5], joka käsittelee agenttien uskottavuutta ja tekotyperyyttä yleisesti. Ohjaajani ehdottivat, että tekisin laajemman opinnäytteeni samasta aiheesta, mutta syventyen johonkin tiettyyn osa-alueeseen. Valitsin tutkimuksen alustaksi TORCS-autopelin, koska löysin melko hyviä ohjeita, jotka selittävät, kuinka siihen voi ohjelmoida agentin. Lisäksi peliä on laajalti käytetty akateemisessa tutkimuksessa.

Luku 2 on yleinen johdatus opinnäytteen aiheeseen, ja siinä esitellään työn keskeiset käsitteet. Digitaalisissa peleissä tekoäly ohjaa itsenäisiä hahmoja, joita kutsutaan agenteiksi. Uskottavuus on subjektiivinen ilmiö, joka riippuu muun muassa pelistä, pelaajasta sekä agentin tehtävästä ja roolista. Uskottavuus kuvaa, kokeeko pelaaja olevansa tekemisissä älykkään toimijan kanssa. Toisaalta se kertoo, tarjoaako agentti riittävästi haastetta ja viihdettä. Yksi agenttien tehtävistä peleissä on simuloida ihmispelaajaa. Tällaisia agentteja kutsutaan boteiksi. Tekotyperät tekniikat lisäävät agenttiin tarkoituksellisesti virheitä ja rajoitteita. Niiden tarkoituksena on lisätä uskottavuutta ja pitää pelin haaste sopivana.

Lukujen 3 ja 4 tarkoituksena on antaa lukijalle valmiudet seurata ohjelmoimani botin toteutusta luvussa 6. Lisäksi ne pohjustavat lukua 5, joka esittelee erilaisia autopelien agenttien tekniikoita. Luku 3 selostaa kilpa-ajamisen fysiikkaa ja auton tekniikkaa. Siinä käsiteltyjä yhtälöitä esiintyy myöhemmin bottien heuristiikkojen laskuissa. Luku 4 käsittelee sekä TORCSin työpöytä- että palvelinversion rakennetta. Vaikka ohjelmoin oman bottini työpöytäversiolle, molempien toteutus on hyvä tuntea, sillä teoksessa käsitellään kummankin agentteja.

Luku 5 käsittelee autopelejä ja niiden agentteja. Autopeleissä agentit ohjaavat pelaajan kilpailijoiden autoja. Luvussa kerrotaan, miten eri autopelit eroavat toisistaan, ja minkälaisia ovat realistiset autopelit. Se tarkastelee sekä agenttien tekniikkaa että uskottavuutta. Lisäksi siinä pohditaan, miten tekotyperyys voi ilmetä autopeleissä. Luvussa käytetään esimerkkeinä pääasiassa TORCSin botteja. Luku syventyy

tutkimuskysymyksen teoriaan sekä tukee ohjelmoimani botin kuvausta ja arviointia.

Luku 6 kuvaa yksityiskohtaisesti *Desideriuksen* toteutuksen. Se on TORCSiin ohjelmoimani botti. Luku kertoo projektin kulusta, kohtaamistani haasteista sekä kokeilemistani heuristiikoista. Se kertoo myös epäonnistuneista tekniikoista sekä niistä, joista jouduin muista syistä luopumaan. *Desideriuksen* koodi noudatti aluksi Wymannin opasta [51]. Myöhemmin lisäsin siihen tekniikoita myös muista lähteistä. Luvussa arvioidaan hieman yksittäisten menetelmien vaikutuksia bottiin. Se pyrkii käsittelemään botin eri aktuaattoreihin liittyviä heuristiikkoja erikseen. Autopelissä aktuaattoreita ovat hallintalaitteet, kuten ohjauspyörä ja kaasupoljin.

Luku 7 arvioi *Desideriuksen* suorituskykyä ja uskottavuutta. Suorituskyvyllä tarkoitetaan botin parasta kierrosaikaa valitulla testiradalla. TORCS hylkää ajan, jos botti oikoo kaarteita tai törmää esteisiin. Sen on siis ajettava kierroksensa puhtaasti. *Desiderius* ei ole erityisen suorituskykyinen edes muihin botteihin verrattuna, joten siitä ei ole haastavaksi vastustajaksi pelaajille [33]. *Desiderius* ei myöskään ole uskottava, sillä se ei täytä yhtäkään autopelin agentin vaatimuksista [28]. Sen toteutus on liian yksinkertainen, jotta sen avulla voisi etsiä vastauksia opinnäytteen tutkimuskysymykseen. Uskottavuuden lisääminen tai tekotyperyyden kokeileminen ei ole mielekästä, kun botti ei hallitse autopelin perustaitojakaan. Tutkimus siis epäonnistui siksi.

Lähteet

[1] Brian Beckman, *The Physics of Racing*, saatavilla WWW-muodossa: <URL: `http://phors.locost7.info/contents.htm`>, viitattu 12.11.2014.

[2] F. Braghin, F. Cheli, S. Melzi ja E. Sabbioni, *Race driver model*, Computers and Structures, 86 (2008), s. 1503–1516.

[3] Martin V. Butz ja Thies D. Lönneker, *Optimized Sensory-motor Couplings plus Strategy Extensions for the TORCS Car Racing Challenge*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2009, s. 317–324.

[4] Luigi Cardamone, Daniele Loiacono, Pier Luca Lanzi ja Alessandro Pietro Bardelli, *Searching for the Optimal Racing Line Using Genetic Algorithms*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2010, s. 388–394.

[5] Richard Domander, *Agenttien uskottavuus ja tekotyperyys digitaalisissa peleissä*, kandidaatintutkielma, tietotekniikan laitos, Jyväskylän yliopisto, 2012.

[6] Stan Franklin ja Art Graesser, *Is It an agent, or just a program?: A taxonomy for autonomous agents*, Intelligent Agents III Agent Theories, Architectures, and Languages (toim. Jörg Müller), kirjasarjassa Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, Saksa, 1997.

[7] Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, Springer Berlin Heidelberg, Berliini, Saksa, 1993.

[8] Debasish Ghosh, *Generics in Java and C++ - A Comparative Model*, ACM SIGPLAN Notices, 39 (2004), s. 40–47.

[9] Jeff Hannan, kirjeenvaihto, saatavilla WWW-muodossa: <URL: `http://www.ai-junkie.com/misc/hannan/hannan.html`>, viitattu 20.08.2014.

[10] Päivi Hietanen, *C++ ja olio-ohjelmointi*, Docendo, Porvoo, 2004.

[11] Philip Hingston, *A Turing Test for Computer Game Bots*, IEEE Transactions on Computational Intelligence and AI in Games, 1 (2009), s. 169–186.

[12] Philip Hingston, *A New Design for a Turing Test for Bots*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2010, s. 345–350.

[13] Eduardo Jimenez, *The Pure Advantage: Advanced Racing Game AI*, saatavilla WWW-muodossa <URL: `http://www.gamasutra.com/view/feature/132313/the_pure_advantage_advanced_.php?print=1`>, viitattu 04.09.2014.

[14] Fares Kayali ja Peter Purgathofer, *Two Halves of Play - Simulation versus Abstraction and Transformation in Sports Videogames Design*, Eludamos Journal for Computer Game Culture, 2 (2008), s. 105–127.

[15] John E. Laird, John C. Duchi, *Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot*, konferenssijulkaisu, AAAI Spring Symposium on Artificial Intelligence and Computer Games, 2001, s. 54–58.

[16] John E. Laird, Michael van Lent, *Human-Level AI's Killer Application: Interactive Computer Games*, AI magazine, 22 (2001), s. 15–25.

[17] Chris Lewis, Jim Whitehead, Noah Wardrip-Fruin, *What Went Wrong: A Taxonomy of Video Game Bugs*, konferenssijulkaisu, Proceedings of the Fifth International Conference on the Foundations of Digital Games, 2010, s. 108–115.

[18] Lars Lidén, *Artificial Stupidity: The Art of Intentional Makes*, kirjassa AI Game Programming Wisdom 2, Charles River Media, Lontoo, Yhdistynyt kuningaskunta, 2003, s. 41–48.

[19] Daniel Livingstone, *Turing's Test and Believable AI in Games*, Computers in Entertainment — Theoretical and Practical Computer Applications in Entertainment, 4 (2006), s. 1–13.

[20] Daniele Loiacono, Luigi Cardamone ja Pier Luca Lanzi, *Simulated Car Racing Championship Competition Software Manual*, tekninen raportti, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italia, 2011.

[21] Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi ja Luigi Cardamone, *Learning to Overtake in TORCS Using Simple Reinforcement Learning*, konferenssijulkaisu, IEEE Congress on Evolutionary Computation, 2010, s. 1–8.

[22] Daniele Loiacono, Julian Togelius, Pier Luca Lanzi, Leonard Kinnaird-Heether, Simon M. Lucas, Matt Simmerson, Diego Perez, Robert G. Reynolds ja Yago Saez, *The WCCI 2008 Simulated Car Racing Competition*, konferenssijulkaisu, IEEE Symposium On Computational Intelligence and Games, 2008, s. 119–126.

[23] Shaun McInnis, *How Forza 5 Is Crowd-Sourcing Artificial Intelligence*, saatavilla WWW-muodossa <URL: `http://www.gamespot.com/articles/how-forza-5-is-crowd-sourcing-artificial-intelligence/1100-6409975/`>, viitattu 12.09.2014.

[24] Brian Mac Namee, *Proactive Persistent Agents — Using Situational Intelligence to Create Support Characters in Character-Centric Computer Games*, väitöskirja, University of Dublin, Trinity College, Dublin, Irlanti, 2004.

[25] Robert C. Martin, *Java and C++: a critical comparison*, Java Gems, Cambridge University Press, New York, New York, Yhdysvallat, 1998, s. 51–68.

[26] Ian Millington ja John Funge, *Artificial Intelligence for Games*, 2. painos, Morgan Kaufmann Publishers, Yhdysvallat, 2009, s. 820–822.

[27] Jorge Muñoz, German Gutierrez ja Araceli Sanchis, *Controller for TORCS created by imitation*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2009, s. 271–278.

[28] Jorge Muñoz, German Gutierrez ja Araceli Sanchis, *Towards Imitation of Human Driving Style in Car Racing Games*, kirjassa Believable Bots - Can Computers Play Like People? (toim. Philip Hingston), Springer-Verlag Berlin Heidelberg, Berliini, Saksa, 2012, s. 289–313.

[29] Michael Negnevitsky, *Artificial Intelligence: a guide to intelligent systems*, Pearson Education, Upper Saddle River, New Jersey, Yhdysvallat, 2005.

[30] Jong Hyeon Park ja Chan Young Kim, *Wheel Slip Control in Traction Control System for Vehicle Stability*, Vehicle System Dynamics, 31 (1999), s. 263–278.

[31] Barry Parker, *The Isaac Newton School of Driving: Physics and Your Car*, John Hopkins University Press, Baltimore, Maryland, Yhdysvallat, 2003.

[32] Pelit, 9/2004, 8/2006, 11/2011, 11/2012, 2/2013 ja 1/2014, Sanoma Magazines Finland Oy, Turku, 2004 – 2014.

[33] Jan Quadflieg, Mike Preuss, Oliver Kramer ja Günter Rudolph, *Learning the Track and Planning Ahead in a Car Racing Controller*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2010, s. 395–402.

[34] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu ja Steve Sutphen, *Checkers Is Solved*, Science, 317 (2007), s. 1518-1522.

[35] Larry Schumaker, *Spline Functions: Basic Theory*, 3. painos, Cambridge University Press, Cambridge, Yhdistynyt kuningaskunta, 2007.

[36] John R. Searle, *Minds, Brains, and Programs*, Behavioral and Brain Sciences, 3 (1980), s. 417–457.

[37] Alex Serrarens, Marc Dassen, Maarten Steinbuch, *Simulation and Control of an Automotive Dry Clutch*, Proceedings of the 2004 American Control Conference, 5 (2004), s. 4078–4083.

[38] Bhuman Soni ja Philip Hingston, *Bots trained to play like a human are more fun*, konferenssijulkaisu, International Joint Conference on Neural Networks, 2008, s. 363–369.

[39] Pieter Spronck ja Freek den Teuling, *Player Modeling in Civilisation IV*, kirjassa Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (toim. G. Michael Youngblood ja Vadim Bulitko), The AAAI Press, Stanford, Kalifornia, Yhdysvallat, 2010.

[40] Penelope Sweetser ja Peta Wyeth, *GameFlow: A Model for Evaluating Player Enjoyment in Games*, Computers in Entertainment — Theoretical and Practical Computer Applications in Entertainment, 3 (2005), s. 1–24.

[41] Julian Togelius, Renzo De Nardi ja Simon M. Lucas, *Making Racing Fun Through Player Modeling and Track Evolution*, konferenssijulkaisu, Proceedings of the SAB Workshop on on Adaptive Approaches to Optimizing Player Satisfaction, 2006, s. 1–10.

[42] Julian Togelius, Georgios N. Yannakakis, Sergey Karakovskiy ja Noor Shaker, *Assessing Believability*, Springer-Verlag Berlin Heidelberg, Saksa, 2012, s. 215–230.

[43] Simone Tognetti, Maurizio Garbarino, Andrea Bonarini ja Matteo Matteucci *Modeling enjoyment preference from physiological responses in a car racing game*, konferenssijulkaisu, IEEE Symposium on Computational Intelligence and Games, 2010, s. 321–328.

[44] Simon L. Tomlinson, *The Long and Short of Steering in Computer Games*, International Journal of Simulation Systems, 1–2 (2004), s. 33–46.

[45] TORCS-projektin kotisivu, <URL: `http://torcs.sourceforge.net/`>, viitattu 05.08.2014.

[46] TORCS-projektin sivusto SourceForge-palvelussa, <URL: `http://sourceforge.net/projects/torcs/`>, viitattu 05.08.2014.

[47] Alan Turing, *Computing machinery and Intelligence*, Mind, LIX (1950), s. 433–460.

[48] Iskander Umarov ja Maxim Mozgovoy, *Creating Believable and Effective AI Agents for Games and Simulations: Reviews and Case Study*, perustuu artikkeliin *Believable and Effective AI Agents in Virtual Worlds: Current State and Future Perspectives*, International Journal of Gaming and Computer Mediated Simulations, 4 (2012), s. 37–59.

[49] Tim Uusitalo ja Stefan J. Johansson, *A Reactive Mutli-agent Approach to Car Driving using Artificial Potential Fields (sic)*, konferenssijulkaisu, IEEE Conference on Computational Intelligence and Games, 2011, s. 203–210.

[50] Mick West, *Intelligent Mistakes: How to Incorporate Stupidity Into Your AI Code*, saatavilla WWW-muodossa <URL: `http://www.gamasutra.com/view/feature/3947/intelligent_mistakes_how_to_.php?print=1`>, viitattu 31.10.2014.

[51] Bernhard Wymann, *T.O.R.C.S. Manual installation and Robot tutorial*, saatavilla WWW-muodossa <URL: `http://www.berniw.org/tutorials/robot/`>, viitattu 12.11.2014.

[52] Georgios N. Yannakakis ja John Hallam, *Towards Capturing and Enhancing Entertainment in Computer Games*, Advances in Artificial Intelligence, 3955 (2006), s. 432–442.

A Huomioita C++-kielestä

Tässä liitteessä käsitellään C++-kielen syntaksia ja semantiikkaa siinä määrin, mitä vaaditaan teoksen koodiesimerkkien ymmärtämiseen. Lukijan oletaan tuntevan ohjelmoinnin perusteet, joten luvussa perehdytään vain C++:an erityispiirteisiin. Luvun pääasiallinen lähde on Hietasen kirja [10].

C++ käsittää kahdenlaisia viitemuuttujia: siinä on sekä C-kielestä perityt asteriskilla (`*`) merkittävät osoittimet (engl. *pointer*) että &-merkillä merkittävät viitteet (engl. *reference*). Molemmat voivat viitata sekä primitiivityyppeihin että olioihin. Esimerkiksi kokonaislukuosoitin esitellään `int* luku`. Esittelyn jälkeen itse osoitinta käsitellään syntaksilla `luku`, ja sen osoittamaa muuttujaa `*luku`. Muuttujan `luku` tulostaminen esittää sen osoittaman muistiosoitteen, kun taas `*luku` tulostaa sen osoittaman luvun arvon. Kun viitteitä tai osoittimia käytetään funktion parametrien esittelyssä, välitetään muuttujan osoite, eikä sen arvoa. Tapahtuu siis *copy-by-reference* ei *copy-by-value*. &-merkki parametrin edessä tekee viitteen välittymisestä implisiittistä.

```
int autoja = 0;
kasvata( autoja );
cout << autoja; //tulostuu "1"
vahenna( &autoja );
cout << autoja; //tulostuu "0"
...

void kasvata( int &laskuri )
{
    laskuri = laskuri + 1;
}
void vahenna( int *laskuri )
{
    *laskuri = *laskuri - 1;
}
```

Mikäli viitattavaa muuttujaa tarvitsee vain lukea, se voidaan parametrin esitte-

lyssä suojata avainsanalla `const`. Esimerkiksi `const int &luku`. Avainsana estää muuttamasta primitiivityypin arvoa. Se toimii eri tavalla osoittimien kohdalla. Jos parametri esitellään `const int* luku`, määre suojaa osoitinta, ei sen osoittamaa muuttujaa. Toisin sanoen osoittimen osoitetta ei voi vaihtaa, mutta sen osoittaman muuttujan arvoa voidaan muuttaa. Osoitettu muuttuja suojataan `int* const luku`, ja sekä osoitin että muuttuja `const int* const luku`.

Kun olioon viittaava parametri on suojattu `const`-määrellä, aliohjelma voi pelkästään lukea olion julkisia attribuutteja ja kutsua sen metodeista niitä, joilla ei ole sivuvaikutuksia. `Const`-määre luokan metodin esittelyn perässä tarkoittaa, että sen koodi ei muuta olion tilaa. Tällaisella metodilla ei siis ole sivuvaikutuksia. `Const`-määrettä rikkova koodi aiheuttaa käännösaikaisen virheen. C++:ssa `const`-määreellä on myös muita merkityksiä, joita ei tässä esitellä. Alla olevassa esimerkissä on koottu edellä esiteltyjä `const`-määreen vaikutuksia:

```
class Laskuri
{
private:
    int lkm;
    int luettu;
public:
    int getLkm() const
    {
        luettu = luettu + 1; //virhe
        return lkm;
    }
    void kasvata()
    {
        lkm = lkm + 1;
    }
    void lisaa( const int &luku )
    {
        lkm = lkm + luku;
        luku = 0; //virhe
    }
    void vahenna( const int *luku )
    {
```

```
        lkm = lkm - *luku;
        *luku = 0;
        luku = 0; //virhe
    }
    void summaa( const Laskuri &lask )
    {
        lkm = lkm + lask.getLkm();
        lask.kasvata(); //virhe
    }
}
```

C++-kielen mallit (engl. *template*) muistuttavat monessa suhteessa Javan geneerisiä tyyppejä (engl. *generics*) [8]. Mallien esittelyssä käytetään muodollisia parametreja, jotka kääntäjä korvaa automaattisesti todellisilla parametreilla. Näin yhtä toteutusta voidaan hyödyntää usealla eri tietotyypillä. Javassa muodolliset parametrit voivat korvata vain olioita, kun taas C++:ssa niiden tilalle voi sijoittaa myös primitiivityyppejä. Esimerkiksi malleilla toteutettu funktio kahden luvun summan neliön laskemiseksi näyttää C++:ssa tältä:

```
T summanNelio( T a, T b )
{
    T summa = a + b;
    return summa * summa;
}
...
int a=2, b=2;
double c=3.0, d=3.0;
summaaNelio(a, b);
summaaNelio(c, d);
```

Esimerkissä `T` on muodollinen parametri. Jotkin kääntäjät tukevat malleja heikosti. Esimerkiksi *g++:aa* käytettäessä ei ole mahdollista erottaa malleja käyttävien metodien toteutusta niiden esittelystä.

B Desiderius-botin apuluokat

Desideriuksen kiintopiste- ja säteenseurantaheuristiikka käyttävät laskuissaan vektoreita. Tämän vuoksi bottimoduuli tarvitsee luokan, joka toteuttaa kaksiulotteisen vektorin ja vektoreiden yleisimmät laskutoimitukset. Wymannin oppaassa [51] esitellään vektoriluokka `v2d`. En saanut sitä toimimaan linkitysongelmien vuoksi, joten päätin toteuttaa oman vektoriluokkani `V2d<T>`. Sen toteutus noudattaa pitkälti Wymannin luokkaa, mutta se käyttää C++:an malleja. Päätin käyttää malleja, koska ne tekevät luokasta yleiskäyttöisemmän. En etsinyt valmista kirjastoa, koska halusin vektoreideni toimivan samalla tavalla kuin Wymannilla.

Wymannin [51] `v2d`-luokka operoi `float`-liukuluvuilla, kun taas TORCSin versio 1.3.4 käyttää pääasiassa pelin nimeämää tyyppiä `tdble`. Toistaiseksi `tdble` vastaa `float`-tyyppiä. Loin vektorioliot `V2d<tdble>`-tyyppisinä, koska ne toimivat, vaikka `tdble`-tyypin määritelmää muuttuisi pelin tulevissa versioissa.

Luokan `v2d` kaikki muut metodit paitsi `normalize` on suunniteltu siten, että ne eivät muuta olion attribuutteja. Luokan olioista on siis pyritty tekemään muuttumattomia (engl. *immutable*). Alla on luokan metodeista esimerkkinä vektorin kierto:

```
V2d<T> rotate ( const V2d<T> &center, T rads ) const
{
    V2d<T> v = *this - center;
    T sina = sin( rads );
    T cosa = cos( rads );
    T dirX = v.getX() * cosa - v.getY() * sina;
    T dirY = v.getX() * sina + v.getY() * cosa;
    V2d<T> dir( dirX, dirY );
    return center + dir;
};
...
V2d<tdble> x(1.0, 0.0);
V2d<tdble> origo(0.0, 0.0);
V2d<tdble> y = x.rotate(origo, -PI/2.0);
```

Muuttumattomuutta haittasi se, että Wymannin [51] toteutuksessa luokan jäsenet

`x` ja `y` ovat julkisia. Omassa `V2d<T>`-luokassani ne ovat yksityisiä, ja niiden arvot luetaan metodeilla `T getX() const` ja `T getY() const`.

Toteutin joukon yksinkertaisia, staattisia proseduureja, jotka tulostavat pelin tietueiden ja tietorakenteiden arvoja tekstitiedostoihin. Proseduurit on esitelty otsikkotiedostossa `printUtils.h`, ja niiden toteutus on tiedostossa `printUtils.cpp`. Näiden proseduurien avulla pyrin selvittämään sekä pelin muuttujien yksiköitä, että tietorakenteiden koostumusta. Niiden avulla sain myös tietoja botin käyttämästä autosta, ja sen ajamasta radasta.

Radasta tulostettiin `tTrack`-tietueeseen tallennetut tiedot, kuten radan nimi ja pituus. Lisäksi tulostettiin tämän tietueen sisältämä linkitetty lista, joka sisältää radan segmentit. Segmenteistä tulostettiin vain ne, joiden tyyppi on `TR_MAIN`. Nämä kuvaavat itse radan tiedot. Muun tyyppiset segmentit sisältävät tietoja radan ympäristöstä. Niitä tarvitaan lähinnä ulkoasun hahmontamisessa. Vasta radan tietojen tulostamisen jälkeen sain selvyyden siitä, millä tavalla rata on pelin tietorakenteisiin tallennettu. XML-tiedostossa rata on jaettu suurempiin kokonaisuuksiin, kuten kaarteisiin ja suoriin, kun taas linkitetyn listan segmentit käsittävät vain muutaman metrin matkan. Yksi XML-tiedostoon tallennettu osio siis jaetaan pelissä useaan osaan. Ennen tietorakenteen lukemista pyrin virheellisesti lukemaan seuraavan kaarteen kulman sen ensimmäisestä segmentistä. Koska segmentti on niin lyhyt, sen käännös on hyvin pieni. Tällöin botti tulkitsi kaarteen varsin loivaksi, mikä johti usein ulosajoon. Linkitetyn listan rakennetta kyllä selvitetään Wymannin oppaassa [51], mutta ymmärsin sen väärin.

Botin käyttämästä ajoneuvosta tulostettiin sekä staattisia että dynaamisia tietoja. Nämä on pelissä tallennettu tietueeseen `tCarElt`. Auton staattisia tietoja ovat muun muassa sen mitat, moottorin suurin vääntö sekä vaihteiden välityssuhteet. Sen dynaamisia ominaisuuksia ovat muun muassa nopeus, kiihtyvyys ja sijainti. Dynaamisia tietoja tulostettiin yhden kierroksen ajan radalla *wheel 2* joka kymmenennellä simulaatiojaksolla, eli noin 200 ms välein. Auton nopeudesta, kiihtyvyydestä, suunnasta ja paikasta tallennettiin erikseen tiedot x-, y- ja z-akselien suhteen. `TCarElt` sisältää lisäksi yhden `tWheelState`-tietueen kutakin auton rengasta kohti. Myös näistä tallennettiin sekä dynaamisia että staattisia tietoja. Jostakin syystä renkaisiin kohdistuvia voimia kuvaavien muuttujien `Fx`, `Fy` ja `Fz` arvot pysyivät aina nollassa.

C Rataluokitin ja tilakone

Tässä liitteessä kerrotaan *Desiderius*-botin käyttämästä rataluokittimesta. Liitteessä kerrottaan, miten toteutin sen, ja miten sitä voisi jatkossa kehittää. Lisäksi rataluokittimeen liittyvä tilakone esitellään lyhyesti.

Rataluokittimen luokka `TTrackClassifier` sisältää osoittimet `curShape` ja `nextShape`, jotka viittaavat tämänhetkiseen ja seuraavaan rataosuuteen. Ne ovat tyyppiä `TTrkShape*`. Luokan `TTrkShape` oliot sisältävät yhden rataosuuden tiedot. Muun muassa sen, mistä segmentistä osuus alkaa ja mihin se päättyy. Luokitin jakaa osuudet suoriin (`STRAIGHT`) sekä loiviin (`FULL`) ja jyrkkiin (`SLOW`) kaarteisiin. Suoriksi lasketaan osuudet, joiden kulma $|\alpha| < \frac{\pi}{60}$. `FULL`-tyyppisten kaarteiden käännös on alle $|\frac{\pi}{9}|$. Tätä jyrkemmät kaarteet saavat `SLOW`-tyypin. Kun TORCS kutsuu botin `newRace`-proseduuria, botti kutsuu rataluokittimen `launch`-proseduuria. Rataluokitin koostaa proseduurissa segmenteistä kaksi seuraavaa rataosuutta. Mahdolliset peräkkäiset suorat yhdistetään yhdeksi osuudeksi. Kisan aikana botti ilmoittaa joka simulaatiojaksolla luokittelijalle, millä segmentillä auto sijaitsee. Jos botti on edennyt rataosuudelta toiselle, luokittelija asettaa ensin `curShape`-osoittimen osoittamaan samaan olioon kuin `nextShape`. Sitten se tutkii rataa eteenpäin ja muodostaa uuden osuuden talteen `nextShape` osoittimeen. Luokitin päättelee botin sijainnin segmentin järjestysnumeron `tTrackSeg.id` perusteella.

Jatkossa `TTrackClassifier`-luokasta voisi muokata sellaisen, että saman bottimoduulin eri ilmentymät jakavat sen. Se tulisi muuttaa *Singleton*-suunnittelumallin [7] mukaiseksi. Singleton-luokka on sellainen, josta luodaan yksi ainoa ilmentymä koko ohjelman suorituksen aikana. Tämän jälkeen jokaista ilmentymää kohden luotaisiin jonkinlainen kevyt, Edustaja-suunnittelumallin (engl. *proxy*) mukainen olio. Edustajat pitäisivät kirjaa kunkin kuljettajan tämänhetkisestä ja seuraavasta rataosuudesta. Nyt edustajaluokka sisältäisi osoittimet `curShape` ja `nextShape`. Muutosten jälkeen luokitin koostaisi rataosuudet vain silloin, kun aloitetaan kisa uudella radalla. Se tallentaisi osuudet johonkin tietorakenteeseen talteen, esimerkiksi linkitettyyn listaan. Tällä hetkellä rataluokitin kerää osuuksia kilpailun aikana jatkuvasti, eikä sen kanssa ole mahdollista käyttää useampaa kuljettajaa.

Luokittimen lisäksi toteutin yksinkertaisen tilakoneen ohjaamaan siirtymiä ajo-

tilanteesta toiseen. Sen tilat perustuvat rataluokittimen luokkiin. Se on toteutettu `Driver`-luokan `decideDriveStyle`-metodissa. Käytännössä tilakone koostuu vain peräkkäisistä `if–else`–rakenteista. Jatkossa olisi luultavasti järkevämpää siirtää tilakone omaksi luokakseen. Tällöin tulisi pyrkiä erottamaan itse tilakone, ja sen botissa aiheuttamat toimenpiteet toisistaan. Tämä voi toisaalta tehdä koodista monimutkaisempaa. Eriyttämistä voi lisäksi haitata se, että tilakoneen siirtymät riippuvat tällä hetkellä auton sijainnista, ja botin havainnoista.