Liikennevalosimulaattori

Antti Mäkipää
Opiskelijanumero: 350682
TkK, Tietotekniikka, 2013

8.5.2014

Yleiskuvaus

Liikennevalosimulaattori on peli, jossa pelaajan tarkoitus on liikennevaloja ohjaamalla saada tietty määrä autoja perille, eli pelikentän reunoilta ulos, mahdollisimman nopeasti. Pelaajan ei tarvitse huolehtia autojen liikkumisesta, sillä ne ohjaavat kukin itseään valiten reittinsä, väistellen toisiaan ja tietysti noudattaen liikennevaloja.

Autot liikkuvat graafisessa ympäristössä, jollainen on annettu valmiina, mutta joita voi myös määritellä uusia pelin asetustiedostoon.

Autot eivät törmää toisiinsa ja kulkevat täysin autonomisesti. Jokaiselle arvotaan sen luomishetkellä lähtöpaikka ja seuraava tie, jolle se pyrkii. Autot pyrkivät siis pääsemään perille seuraavaan kohteeseensa, välttämään törmäykset toisiinsa ja noudattamaan liikennevaloja. Pelaaja ei pysty aiheuttamaan kolareita, vaan valittavissa on kunkin risteyksen liikennevaloille jokin asento, jolla törmäyksiä ei synny ja näiden välillä on vaihdeltava sopivasti siten, että kaikki autot pääsevät mahdollisimman nopeasti perille.

Autoja syntyy tasaisesti, kunnes tavoitemäärä on saavutettu. Tämän jälkeen niitä syntyy harvemmin tai kun aikaisempia autoja pääsee perille.

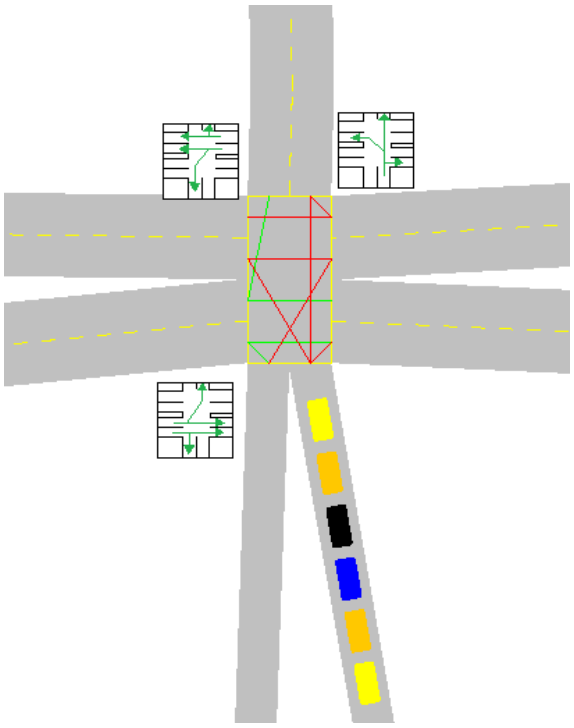
Pelaajan tulos tallentuu huipputuloksiin, jotka näytetään pelin jälkeen.

Käyttöohje

Ohjelma käynnistetään ajamalla MapGraph-niminen olio, joka perii Scalan SimpleSwingApplication:in. Esiin aukeaa kartta, joka on luotu asetustiedosto gamefile.txt:n perusteella. Vasemmassa ylänurkassa näkyy pelin läpäisyyn vaadittavien autojen määrä: näin monta autoa on ohjattava ulos kartalta mahdollisimman nopeasti.

Ylhäällä olevista nappuloista "Start Game" aloittaa pelin, jolloin autoja alkaa syntyä kentän reunoilta. "Give Up" lopettaa käynnissä olevan pelin, minkä jälkeen "Start Game" aloittaa jälleen uuden pelin. "Highscores" näyttää tulostaulukon kymmenen parasta tulosta.

Peliä pelataan painelemalla risteysten vieressä olevia ruutuja, joista kukin esittää yhtä kyseiseen risteykseen valittavissa olevaa liikennevaloyhdistelmää. Risteysten halki risteilevät punaiset ja vihreät viivat kertovat, mitkä mahdollisista reiteistä ovat juuri sillä hetkellä sallittuja autoille: punainen viiva kuvaa palavaa punaista valoa, vihreä vastaavasti vihreää valoa.



Kuva 1: Pelaajan kannattanee valita yhdistelmä oikealla ylhäällä

Kun pelaaja painaa jotakin nappulaa, valot vaihtuvat sallimaan ne reitit, joita pikkukuvake esittää. Näin pelaajan tulee auttaa autot mahdollisimman sujuvasti liikenteen halki.

Ohjelman rakenne

Ohjelma on jaettu kolmeen eri pakkaukseen, joista kustakin löytyy tietystä osa-alueesta huolehtivia luokkia. "trafficLogic" sisältää liikenteen taustalla pyörivään simulointiin tarvittavat luokat, kuten autojen ja teiden loogiset osat. "mapLogic" sisältää kokonaisuutta hallitsevia luokkia ja pakkauksesta "graphical" löytyy kunkin luokan graafinen vastine sikäli kun sellainen on.

Keskeisimpiin luokkiin kuuluu varmasti "trafficLogic":n "Car", joka jolla on päävastuu autojen tekoälystä eli niiden liikkumisesta liikenteessä. Tähän se käyttää muun muassa sisäisiä luokkia "SpeedVector" ja "AccelVector", jotka auttavat ohjauksen laskennassa. "Road" kuvaa teitä, ja sisältää paljon tarpeellisia metodeja etenkin kartan lukuvaiheessa, mutta myös itse pelin aikana. "Crossing" kuvaa pelissä olevia risteyksiä ja auttaa muun muassa "Road"-luokkaa määrittämään teiden pääte- ja alkupisteitä, mutta myös pelin aikana säilyttää tietoa muun muassa siitä, mikä on kulloinkin valittu valoyhdistelmä.

"Car"-luokka sisältää paljon metodeja sisäisten luokkamäärittelyjensä lisäksi. Se säilöö muun muassa mittansa ja tiedot nykyisestä ja seuraavasta kaistasta ja risteyksestä. Olennaisia metodeja ovat calc, joka laskee autolle uuden sijainnin annetun ajanhetken jälkeen ja jonka avuksi useimmat muut metodit ovat. passedPoint kertoo onko auto ohittanut annetun pisteen, yleensä annetun seuraavan etapin. scope antaa alueen, jota auto tarkkailee esteiden varalta, ja positionInScope kertoo auton sijainnin toisen auton alueella. Suunnistamisen tärkeimmät metodit ovat steerTowards ja arriveTowards, joista ensimmäistä käytetään kun halutaan ajaa tietyn pisteen kautta ja jälkimmäistä kun halutaan saapua johonkin pisteeseen. Molemmat palauttavat kiihtyvyysvektorin joka ohjaa haluttuun suuntaan. Yksi tärkeimmistä ja monisäikeisimmistä metodeista lukuisine sisäisine metodeineen on findNextLeg, joka tallentaa autolle seuraavan navigointipisteen sijainnin ja tyypin lukuisten eri muuttujien perusteella.

"mapLogic"-pakkauksesta on nostettava esiin luokka "Game", joka pyörittää peliä säilyttämällä tietoja pisteistä, käytetystä ajasta, tavoitteesta, pelissä olevista teistä, risteyksistä, nappuloista ja autoista sekä pitämällä autojen sijainnin laskennan ja kartan piirtämisen jatkuvasti käynnissä. "FileReader" puolestaan lukee kartan tiedostosta, ja on myös olennainen palanen.

"graphical" - kuten sanottua - sisältää graafiset vastineet kullekin tielle, risteykselle, risteyksen kaistalle, autolle ja nappulalle, jotka luokan "GamePanel" on yksinkertaista piirtää tarvitsematta paljoa for-silmukkaa monimutkaisempia rakenteita. Jokaisen laskentakierroksen - jokaiselle autolle lasketun uuden sijainnin - jälkeen "Game"-luokka kutsuu "GamePanel":in repaint-metodia. "GamePanel" on nimensä mukaisesti Scalan "Panel"-luokan perivä luokka. Graafisen käyttöliittymän isäntänä toimii "MapGraph"-olio, joka sisältää pelin nappulat, niiden määrittelyn, sekä "GamePanel":in ja käynnistää pelin omassa säikeessään kun käyttäjä painaa nappia.

Luokista ja niiden instansseista viitataan toisiinsa melko runsaasti, sillä niiden on tiedettävä toisistaan, voidakseen toimia itse oikein. Esimerkiksi teiden on tunnettava seuraavat ja edelliset tiet ja risteykset, jotta ne voivat laskea oman lähtöpisteensä. Samoin esimerkiksi auton on tiedettävä sekä nykyinen kaistansa että tie, jolle se seuraavaksi pyrkii.

Ylimpänä säilönä on "Game"-luokka, jossa on tallessa kaikki tiet, autot, nappulat ja risteykset. Game myös toteuttaa piirreluokan "Runnable", jotta siitä voidaan luoda oma säie. Säikeen käynnistys käynnistää metodin run, joka pyörittää ohjelman laskentaa while-silmukan sisällä, kunnes peli on päättynyt jostain syystä. Toinen vaihtoehto olisi ollut toteuttaa animaatio ja ohjelman suoritus ajastimella. Silmukalla vältetään tilanne, jossa laskenta kestääkin kauemmin kuin ajastimeen määritetty aika ja toisaalta sen kestäessä vähemmän aikaa saadaan laskennasta nopeampaa, jolloin animaatio ja autojen reagointi on sulavampaa.

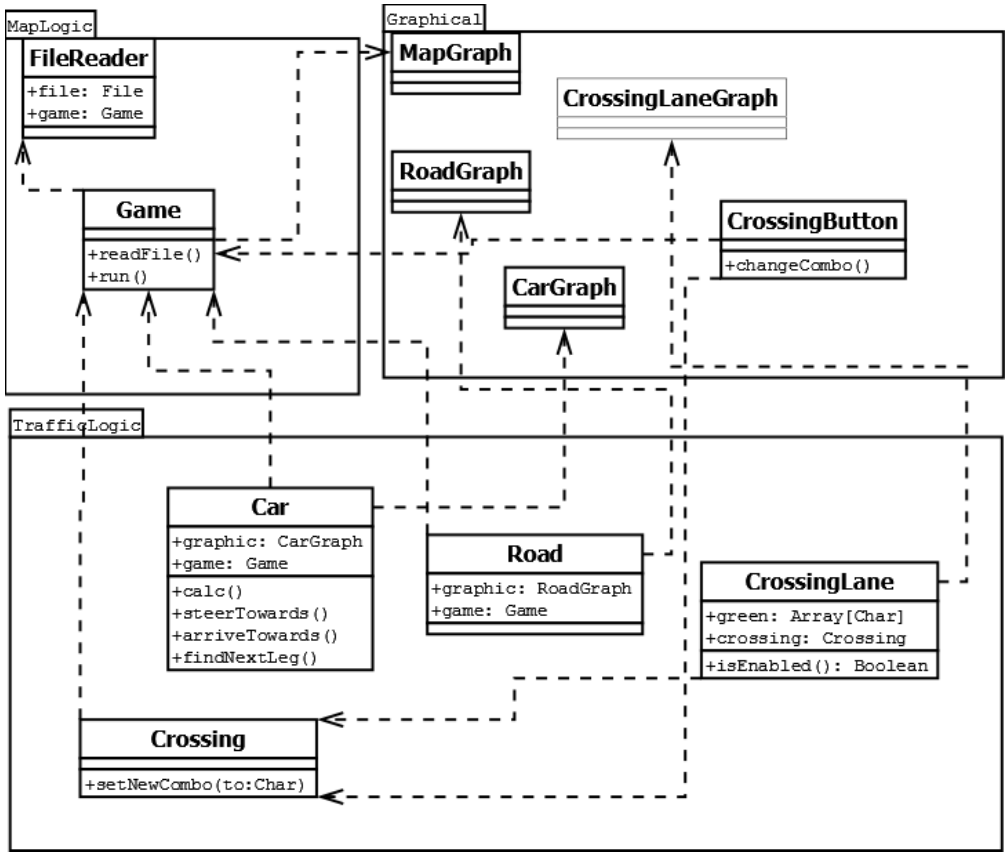
"Game" ei kuitenkaan suoranaisesti ihan kaikkea säilö, vaan tiet tuntevat kaistansa, ja risteykset omat mahdolliset reittinsä, ohjelmassa "CrossingLane":t. Kaiken kaikkiaan rakenne pyrkii olemaan

intuitiivinen ja looginen. Välillä luokista viitataan toisiinsa, mikä saattaa vaikeuttaa niiden uudelleenkäyttöä sellaisinaan, mutta lienee paikoin välttämätöntä. Esimerkiksi tien on tunnettava kaistansa, mutta samoin myös jokaisen kaistan on tunnettava tie, johon kuuluu, jotta se esimerkiksi voi kysyä vasemmalla puolellaan olevaa kaistaa tai seuraavaa kaistaa.

Tien keskeisimmät tiedot ovat sen lähtö- ja päätepisteet, sekä edellinen ja seuraava tie tai risteys. Näitä tarvitaan kartan piirtämiseen, suuntien laskemiseen, autojen reittien määrittämiseen ja paljon muuhun, mitä ohjelma tekee.

Tien lähtöpiste riippuu edellisestä tiestä tai risteyksestä, jos sellainen on olemassa. Tien sijainti määritellään vasemman lähtökulman perusteella. Jos kuitenkin oikea kulma on se, joka koskee edelliseen tiehen, on vasemman kulman sijainti laskettava sen perusteella. Tätä ei tietenkään voida tehdä heti tietä luodessa, sillä jos edellistä tietä ei ole vielä luotu, ei saada oikeaa tulosta. Tien päätepiste luetaan suoraan tiedostosta, paitsi jos seuraavana on risteys, jolloin sijainti täytyy kysyä risteykseltä. Risteyksen leveys, ja oikea piste riippuu myös muista siihen liittyvistä teistä, joten kokonaisuus kartan piirtämiseksi on varsin monimutkainen. Tätä varten "Crossing" ja "Road"-luokat sisältävät paljon apumetodeita, joissa usein toistuu jossain muodossa touchCorner, jolla tarkoitetaan sitä pistettä, jossa tie koskettaa edellistä tai seuraavaa. Nämä metodit, sekä ne jotka laskevat kumpi kulma koskettaa, ovat keskeisessä asemassa kartan piirtämisessä.

"Road"-luokan metodit findStart ja findEnd ovat siis ne, joilla evaluoidaan alku- ja päätepisteet, ja näitä auttaa olennaisesti touchCorner. Vielä yksi metodi, jota ei tarvita kartan luonnissa vaan jota autot myöhemmin käyttävät, on whichLaneFor, joka kertoo mikä tien kaistoista johtaa halutulle seuraavalle tielle.



Kuva 2: Tärkeimpiä luokkia ja niiden suhteita lopullisessa toteutuksessa

Algoritmit

Kartan luominen

Algoritmi oikean reitin löytämiseksi olisi ollut yksi haaste, etenkin kun kartasta ei haluta tehdä ruudukkomaista eli teiden pituudet eivät ole kokonaislukuja tai muita helppoja monikertoja. Tämän tehtävän kannalta eivät autojen järkevät reittivalinnat kuitenkaan ole olennaisia, joten riittää että ne pääsevät joskus perille eli kartan reunalle. Käyttäjälle ei myöskään näy mihin autot ovat menossa (se voisi olla hieno lisäominaisuus pelin myöhemmissä versioissa: autoa klikkaamalla kartalle piirtyisi sen reitti), joten reitinlaskennan voi "huijata" arpomalla vain seuraavan tien ennen jokaista risteystä. Eri teiden painottamiseen on mahdollisuus pelin asetustiedostossa, jos joistain teistä halutaan tehdä toisia ruuhkaisempia. Tämä tapahtuu laskemalla mahdollisten teiden painokertoimet yhteen, arpomalla kokonaisluku nollan ja tämän summan väliltä ja katsomalla minkä tien kohdalle luku osui.

Kartan piirtäminen vaatii paljon trigonometriaa, muttei mitään monimutkaisempaa. Siihen ei syvennytä sen tarkemmin, sillä vaikka se oli tarkkaa, ja logiikka matematiikan takana jopa hienosäätöä niin perustuu se varsin yksinkertaisiin laskuihin, jotta tiet noudattavat seuraavia sääntöjä:

- Jos edellinen tie tai risteys on olemassa, lähtöpiste määräytyy sen mukaan, muuten tien oma lähtöpiste
- Jos seuraava risteys on määritelty, päätepiste riippuu siitä, muuten tien oma päätepiste
- Tien sijainti määritellään sen vasemman reunan pääte- ja aloituspisteiden avulla. Jos oikea kulma koskettaa edellistä tai seuraavaa tietä tai risteystä, on vasemman kulman sijainti laskettava

Mainitaan kuitenkin muutamalla sanalla tiedostosta luvun järjestyksestä, sillä se on olennainen osa kartan piirtämisen onnistumista. Ensin luodaan risteykset. Tämän jälkeen lisätään kaikki tiet, jonka jälkeen kaikille teille naapurit. Teiden alku- ja päätepisteet saattavat riippua muista teistä, joten niitä ei voida määrittää välittömästi, sillä muita teitä ei välttämättä ole olemassa. Niitä ei myöskään haluta laskea joka kerta uudelleen. Siksi tien alku ja loppu ovat laiskasti evaluoituvia vakioita. On kuitenkin pidettävä huolta, ettei niitä päästä tarvitsemaan ennen kuin niiden laskeminen on mahdollista, joten esimerkiksi "Crossing"-luokan joidenkin metodien on tärkeää kerätä tietoja teistä sellaisten ominaisuuksien avulla, jotka tunnetaan aina. Tämä heikentää myöhempää muokattavuutta, mutta on kenties välttämätön paha.

Itsenäiset autot

Autojen tekoälyn algoritmit ovat niinikään matematiikaltaan varsin yksinkertaisia, mutta huomattavasti mielenkiintoisempia. Piste, jota kohti auto suunnistaa, voi olla monenlaista tyyppiä, mistä riippuu kuinka sitä kohti ajetaan. Auto voi ohjautua kahdella tyylillä: **saapumalla** (arrive) ja **ohjaamalla** (steer). Saapuminen ohjaa autoa kohti haluttua pistettä jarruttaen siihen. Ohjaaminen ohjaa auton kulkemaan pisteen ohi, muttei pysähtymään. Lisäksi auton liikkumista ohjaa muiden autojen väistely näiden käyttäytymisten yläpuolella. Käytännössä auto asettaa tällöin itselleen väliaikaisen saapumispisteen tietä tukkivan auton eteen. Auton ohjaaminen noudattelee seuraavia sääntöjä:

- Jos seuraava piste on punainen valo, **saavutaan** tien loppua kohden.
- Jos seuraava piste on kaistanvaihto, vihreä valo, tien alku tai loppu tai kartan reuna, **ohjataan** kohti sitä.

Kun piste ohitetaan, valitaan uusi piste seuraavien sääntöjen mukaan:

- Kun ohitetaan vihreä valo tai tien loppu, on seuraava piste kohti seuraavan kaistan alkua
- Kun ohitetaan kaistanvaihto tai tien alku, on seuraava piste kohti uuden kaistan loppua
- Kun ohitetaan kartan reuna, päivitetään pistetilanne ja poistetaan auto

Kaikki ohjaaminen tapahtuu soveltamalla laskettua kiihtyvyysvektoria nykyiseen nopeuteen, jolloin saadaan uusi nopeus. Uusi sijainti saadaan lisäämällä vanhaan sijaintiin uusi nopeus kerrottuna ajalla,

joka on kulunut edellisestä kerrasta kun sijainti laskettiin. Laskenta eli auton metodikutsu "calc" kulkee jotakuinkin seuraavasti.

Ensin tarkistetaan, onko tarvetta laskea uusi saapumispiste. Näin tehdään luonnollisesti silloin, kun edellinen piste on ohitettu, mutta myös jos seuraava piste on vihreä tai punainen valo, jotta huomataan kun valo vaihtuu. Tämä vaatii toki runsaasti ylimääräisiä kierroksia findNextLeg-metodin kautta verrattuna ratkaisuun, jossa liikennevalot olisivat vaihtuessaan informoineet kaikkia niitä kohti kulkevia autoja muutoksestaan, mutta tapauksessa jossa valo ei ole vaihtunut metodikutsu vaatii vain muutaman tarkistuksen. Lähdettäessä tarkemman optimoinnin ja tehokkuuden tielle vaihtoehtoinen tapa olisi eittämättä lopulta ollut tehokkaampi.

Tarkistuksen jälkeen tarkistetaan muut autot "Game"-luokan metodilla, joka tarkastaa kaikille muille autoille paitsi autolle itselleen, onko näiden graafisen vastineen "CarGraphin" piirtämä ääriviiva tarkistavan auton havainnoinnin sisällä. Tämä "scope" on auton keulasta jarrutusmatkan verran eteenpäin ja viidesosaleveyden verran sivuille laskettu alue, jälleen trigonometriaa käyttäen. Jarrutusmatka lasketaan kaavalla

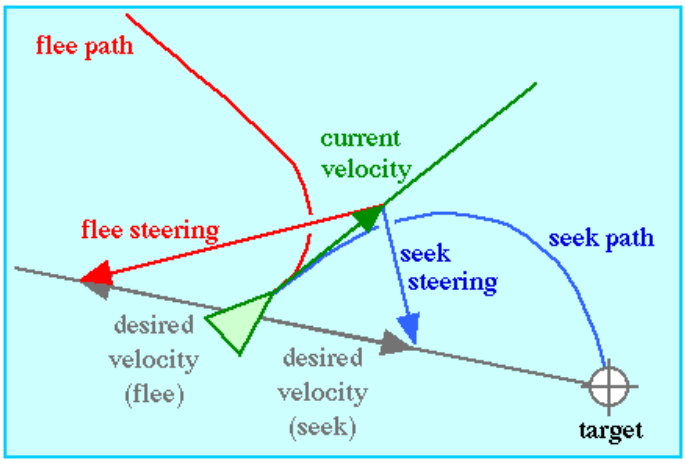
½ * nopeus^2 / jarrutusteho

todellista käyttäytymistä mukaillen.

Alueen sisällä olemisen tarkistus suoritetaan Javan "Area"-luokan valmiilla intersects-metodilla isInScope-nimisen "Car"-luokan metodin sisällä. Jos huomioitavia autoja löytyy, etsitään for-silmukalla niistä lähimpänä oleva soveltaen kuhunkin "Car"-luokan metodia positionInScope. Se kasvattaa kokeiltavaa testaavan auton osa-aluetta pikseli kerrallaan kunnes yllä mainittu isInScope kertoo auton olevan tämän pienimmän alueen sisällä. Lyhyemmin sanoen: Auto katsoo eteensä pikseli kerrallaan kunnes löytyy lähimpänä oleva toinen auto, jonka mukaan jarrutetaan.

Jos muuta reagoitavaa liikennettä löydetään, ohjataan niiden mukaan seuraavaksi kuvattavan ohjautumislogiikan mukaisesti **saapuen** kohti edessä olevan auton edustaa.

Muun liikenteen tarkkailun jälkeen auto ottaa tähtäimeensä sille asetetun navigointipisteen ja joko saapuu tai ohjautuu sitä kohden aiemmin mainitun logiikan mukaisesti riippuen pisteen tyypistä.



Kuva 3: Lähteen esittämä havainnollistava kuva ohjauslogiikasta

Saapumisella ja ohjautumisella on käyttäytymisen kannalta merkittävästä erosta huolimatta ohjelman sisäisen logiikan kannalta vain pieni ero, ja molemmat käyttävätkin lopulta samaa metodia. Ainoa ero on nopeudessa, johon ne tähtäävät. Palataan nopeuden määräytymiseen pian, mutta pureudutaan ensin siihen, mitä tapahtuu molemmissa tapauksissa kun se on laskettu.

Tällöin turvaudutaan "Car":n sisäluokan "SpeedVector"in metodiin steerFor, joka palauttaa kiihtyvyysvektorin jota käyttämällä nopeus muuttuu kohti haluttua. Ohjautumisessa ja saapumisessa on käytetty osoitteesta http://www.red3d.com/cwr/steer/gdc99/ [1] löytyviä ideoita niiden toteutuksesta. steerFor etsii kuvan 3 vektoria "seek steering" vastaavan kiihtyvyysvektorin kohti oikeaa suuntaa. On huomiotava, että samassa kuvassa on esitetty sekä kiihtyvyys- että nopeusvektoreita, eivätkä niiden pituudet ole aivan yksikäsitteisesti verrattavissa. Ohjelma käyttääkin yksinomaan pelin toiminnan kannalta valittua kerrointa kaksi (2) "seek steering" vektorin pituuteen, joka ensin on laskettu tavoitellun nopeuden (desired velocity) ja senhetkisen (current velocity) erotuksesta. Vielä kerran korostettakoon, että fysikaalisesti laskutoimitus ei noudata mitään autojen käyttäytymiselle ominaista kaavaa, ja kahdella kertominen tarkoittaa jotakuinkin sitä, että puolessa sekunnissa samalla kiihtyvyydellä jatkaen olisi suunta ja nopeus haluttu. Ottaen huomioon, että kiihtyvyys pienenee muutaman millisekunnin välein, ei edes näin ylimalkaista selitystä "seek steeringin"

suuruudelle voida antaa, joten se hyväksyttäköön vain mielivaltaisen pituisena kiihtyvyysvektorina nuolen osoittamaan **suuntaan**.

Kuten kuvastakin nähdään, määräytyy lasketun kiihtyvyysvektorin suunta ja suuruus kaikesta yllä olleesta vähättelystä huolimatta tavoitellun nopeuden (desired velocity) suunnasta ja suuruudesta. Tämän vektorin kulloinkin käytettävä metodi arriveTowards tai steerTowards antaa steerFor-metodin käsiteltäväksi. Tämä vektori on itse asiassa kahden vektorin summa, sillä kuvan 3 kohti kohdetta (target) osoittavan vektorin lisäksi autosimulaation tarkoituksiin on siihen lisätty vektori joka osoittaa kohti kaistan keskustaa auton silloisesta sijainnista. Tähän asti ohjautuminen ja saapuminen toimivat identtisesti.

Ero syntyy halutun nopeuden (desired velocity) suuruudesta, joka on ohjautumisessa aina maksiminopeus ja saapumisessa tietyn etäisyyden jälkeen lineaarisesti riippuva etäisyydestä kohteeseen. Myös yllä mainittu lähde kuvaa tämän käyttäytymisen.

Näin saatu kiihtyvyysvektori kerrotaan laskentakierrokseen kuluneella ajalla jolloin saadaan nopeuden muutos, joka lisättynä aikaisempaan nopeuteen antaa uuden nopeuden, joka on siis tulos reagoinnista muihin autoihin, liikennevaloihin, mutkiin tai vaikka tarvittavaan kaistanvaihtoon. Kaavana esitettynä uusi nopeusvektori on siis

$$\mathbf{v} = \mathbf{v}_0 + \mathbf{a} * t$$

missä **v** on uusi nopeus, $\mathbf{v}_0$ on vanha nopeus, **a** on edellä kuvatusti laskettu kiihtyvyys ja on skalaarisuure, joka kertoo laskentakierrokseen käytetyn ajan. Nopeusvektoreiden yhteenlaskut ovat triviaaleja x- ja y-komponentteihin purkamisia, osien yhteenlaskua ja takaisin polaarimuotoon muuttamisia.

Autojen tekoäly toimii näin varsin hyvän näköisesti ja kohtalaisen luotettavasti. Muita vaihtoehtoja ei oikeastaan ole edes harkittu, sillä löydetty malli vaikutti sekä yksinkertaiselta, että myös realistiselta ja kaiken kaikkiaan algoritmina varsin kauniilta, minkä vaikutelman myös onnistunut toteutus osoittaa oikeaksi. Koko ohjautuminen tapahtuu vain muutaman lyhyehkön metodin avulla.

Puutteitakin on, esimerkiksi tismalleen rinnakkain vastakkain kaistoja vaihtavat kaksi autoa. Pelin kehityksen jatkuessa tekoälyn kehittäminen hienostuneemmaksi olisi ehdottomasti vaatimuslistalla, mutta leikkimielisen kisailun tarpeet jo autojen nykyinen käyttäytymisen varmuus täyttää oivallisesti.

Tietorakenteet

Muuttumattomia kokoelmia ja vakioita on pyritty käyttämään mahdollisuuksien mukaan, muttei liikaa selkeydestä tinkien. Silmään saattaa osua "Road"-luokan edelliset ja seuraavat tiet, jotka ovat var-muuttujia vakion sijaan. Tämä toteutus johtuu siitä, ettei edellistä tai seuraavaa tietä ole aina olemassa luontihetkellä. Monimutkaisen rekursion tai kutsupinon rakentamisen sijaan muuttujan käyttö palvelee tarkoitusta paremmin. Naapuritiet lisätään erikseen kaikkien teiden luonnin jälkeen saman metodin sisällä.

Enimmäkseen ohjelmassa on käytetty Vector- ja Array-tyypin kokoelmia. Muuttumattomat rakenteet on pyritty valitsemaan silloin kun mahdollista eikä niiden muuttamiselle ole tarvetta. Tällaisia ovat esimerkiksi tiet ja risteykset. Sen sijaan kartan rakennusvaiheessa risteyskaistoja (CrossingLane) joudutaan lisäämään niin moneen otteeseen, että ne on toteutettu yksinkertaisuuden vuoksi muutuvatilaisella Buffer-tyypillä. Samoin on autojen laita, joita luodaan ja poistetaan läpi pelin. Omia tietorakenteita ei tarvittu, vaan toteutus onnistui kätevästi yksinkertaisimmilla valmiilla kokoelmatyypeillä

Tiedostot

Tiedostojen luku on olennainen osa liikennevalosimulaattoria. Pelin kartta luodaan teksitiedostoon ja sellaiset asetukset kuin voittoon vaadittavat pisteet ja kerralla näkyvien autojen tavoitemäärä asetetaan samassa tiedostossa. Lisäksi erillisessä tiedostossa pidetään listaa huipputuloksista

Asetustiedoston eri käskyt luetaan riveittäin. Symboli # aloittaa kommentin, jonka jälkeisiä osia ei huomioida. Samoin välilyöntejä tai kirjainkokoa ei huomioida. Rivin aloittaa aina kolmimerkkinen tunniste sekä sitä seuraava määritysosa, jotka on selitetty alla. Huomaa, että kaikki esimerkit ovat pelin todellisen oletuskentän asetustiedostosta.

SIZ1880,1000	kentän koko (vaaka, pysty)
GOA50	pelin voittamiseen vaadittava pistemäärä
NUM35	autojen samanaikainen tavoitemäärä
CRO 02, 1410.333, ABC	Risteyksen numero, sijainti (x.y), mahdolliset valokombinaatiot (vain ensimmäinen huomioidaan aloitusvalintana)
ROA 0004,1 , 0.0, 1175.350, 01,0005,10 ROA0023, 2, 0.0, 0.0, 0019,05, 10	Tien numero, kaistojen lukumäärä, alun sijainti (x.y), lopun sijainti (x.y), edellinen tie tai risteys, seuraava tie tai risteys, painoarvo
CON026, 02, 0015.1->0008.0, C	Risteyskaistan numero, risteys johon kuuluu, tie ja kaista jonka yhdistää (tien numero.kaista) -> tie ja kaista johon yhdistää, valokombinaatio jolloin vihreä
THU04,B, 974.751, src/CRO04_B.png	Pienkuvakkeen risteys johon kuuluu, valoyhdistelmä jonka aktivoi, sijainti kartalla (x.y), suhteellinen tiedostosijainti

Tien määrittelystä tulee huomata muutama asia. Edellisen ja seuraavan tien tai risteyksen määrittelyssä **kaksimerkkinen numerosarja tulkitaan risteykseksi ja nelimerkkinen numerosarja tieksi**. Jos edellinen tie **tai** risteys on määritelty, on lähtöpiste merkityksetön. Jos seuraava **risteys** on määritelty, on päätepiste merkityksetön. Samalla tiellä **ei voi olla sekä edellistä risteystä että seuraavaa risteystä**. Tie, joka alkaa tai loppuu muualle kuin toiseen elementtiin, luetaan kartan reunaksi. Painoarvo kertoo todennäköisyyden suhteessa muihin teihin, että auto syntyy tai kääntyy risteyksestä juuri kyseiselle tielle kaikista vaihtoehdoista.

Testaus

Ohjelman testaus suoritettiin pitkälti suunnitelmassa aiotulla tavalla. Kehittäminen aloitettiin kartasta, jonka piirtämisen onnistumisesta sai palautetta varsin luonnollisesti visuaalisesti. Haasteet syntyivät odotetuissa paikoissa, eli autojen tekoälyssä sekä kartan piirtämisessä. Kehitys eteni pitkälti järjestelmätestaussuunnitelman mukaisesti, ja lopullinen ohjelma läpäiseekin kirkkaasti kaikki aiotut testit, kuten oli ohjelman toimivuudelle välttämätöntä.

Kun graafinen puoli oli olemassa, oli myös autojen lisääminen helppo tehdä ja liikkuminen todeta. Kun kaikki toimi yksinkertaisessa pelikentässä, rakensin lopullisen kentän, jolloin erilaiset erikoistapaukset tulivat ilmi ja oli mahdollista havaita ja korjata.

Yksikkötestejä ei ohjelman kehityksessä kirjoitettu, vaan bugien havaitsemisessa turvauduttiin sovelluskehittimen debug-työkaluun, visuaaliseen palautteeseen ja ajon aikana konsoliin kirjoittamiseen. Toisaalta animaation aikainen toteuttaminen mahdollisti hyvin intuitiivisen testaamisen, toisaalta kaikkia erikoistapauksia ei pystynyt niin hallitusti luomaan ja tarkastelemaan. Testausta voisi kuvailla hallitun huolimattomaksi, sillä ohjelman toteutusjärjestys oli huolella ajateltu sellaiseksi, että eri osat rakentuvat luonnollisesti olemassa olevan päälle niin, että niiden toiminnan havaitsee heti lopullisessa ympäristössään.

Ohjelman tunnetut puutteet ja viat

Autojen käyttäytymisessä on silloin tällöin esiintyvä porsaanreikä, jossa kaksi täsmälleen rinnakkain kulkevaa autoa vaihtavat kaistaa vastakkaisiin suuntiin. Tällöin autot jumittuvat toisiinsa ja tukkivat kyseisen tien liikenteen. Autojen tekoälyä voi parantaa tunnistamalla tällaiset tilanteet ja laskemalla esimerkiksi tällaisissa tilanteissa kumpi on oikealla ja antaa tälle luvan kiertää este toisen odottaessa.

Kartan järkevyyttä ei testata millään tavalla etukäteen. Kartan kehittäjän vastuulla on varmistua, että se toimii oikein.

Erilaisiin virhetilanteisiin varautuminen on jäänyt vähäiselle. try- ja catch lohkoilla on "MapGraph"-luokassa ratkaistu pari ominaisuutta, mutta varsinaista virheisiin varautumista tämä ei ole. Käyttäjälle ei tarjota esimerkiksi ikkunaa suoraan graafiseen käyttöliittymään, joka kuvaisi virhetilanteen sellaisen sattuessa.

Risteyskaistojen nuolista olisi voinut tehdä havainnollisempia, samoin mahdollisuus lisätä karttaan kuvia, joilla ei ole mitään toiminnallisuutta.

3 parasta ja 3 heikointa kohtaa

+ Autojen itsenäinen liikkuminen ja pääosin näppärä toistensa väistely on onnistuttu toteuttamaan verrattain yksinkertaisilla metodeilla ja ainakin kehittäjälle hyvin hallittavissa olevan kokoisena. Etenkin sisäisten vektoriluokkien käyttöön olen tyytyväinen, vaikkei koodi niiden sisällä ole mitenkään erityisen häikäisevää. Ja ennen kaikkea ne toimivat.

+ Laskennan ja grafiikan pyörittäminen omassa säikeessään onnistui ensi yrittämällä, mikä oli yllättävää sillä ennakkokäsitys aiheesta oli ehkä liiankin kunnioittava.

+ Toteutus, jossa -Graph -luokat toimivat välikappaleena itse työt tekevän luokan ja näytölle piirtämisen välillä toimi hyvin alusta asti ja tuki hyvin kehitystyötä. Se oli yksinkertaisuudessaan luotettava, eikä koko graafiseen osa-alueeseen tarvinnut juurikaan koskea, vaan siitä saattoi nopeasti katsoa miten jokin muutos oikeasti vaikutti.

- Autojen käyttäytymisessä on myös todella paljon hiottavaa. "- length / 2" ja " + Constants.preferredGap" toistuvat luokassa useasti ja paikoin niiden merkitys on itse kehittäjällekin hämärä. Autot pysähtyvät toistensa perään töksähtäen paljon kovemmasta vauhdista kuin risteyksiin ja paikoillaan kääntyminen on luonnottoman terävää. Paljon pientä hiomista löytyisi

- Koodi on paikoin huonosti kommentoitua ja jäsenneltyä, on myönnettävä. Metodien nimet toivottavasti auttavat ajatuksen seuraamisessa.

- Joitain ominaisuuksia joita suunnitelmassa oli kaavailtu jäi puuttumaan, kuten jalankulkijat tai bussipysäkit.

+- Piirre- tai abstrakteja luokkia ei ohjelmaan sisälly laisinkaan. "CrossingLane":a yritin periyttää "Road":sta, "Lane":sta ja yhteisestä yliluokasta, mutta mitään järkevää, mielekästä ratkaisua ei tuntunut löytyvän niiden lopulta varsin erilaisten ominaisuuksien takia. Kenties edellisiin ja seuraaviin kaistoihin liittyvää logiikkaa olisi voinut jalostaa, jolloin myös nextRoad ja nextCrossingLane muttujat olisi voinut korvata yhdellä muuttujalla.

+- Crossing-luokan sisäinen Side-luokka näyttää oikealta toiston hirviöltä, mutta loppujen lopuksi hirveästi parempaa toteutustapaa on vaikea äkkiseltään keksiä

Poikkeamat suunnitelmasta ja aikataulusta

Autojen tekoälystä huolehtiva algoritmi pysyi yllättävänkin hyvin suunnitelmassa kaavaillun laisena siltä osin, kuinka se kysyy ympärillään olevaa tilaa ja mukauttaa nopeutensa siihen sopivaksi.

Samoin oli ohjelman rakenteen ja luokkajaon laita MapGraph olio ei lopulta suoraan tiennyt graafisista sisarolioista, vaan "Game"-luokka tarjosi logiikkaluokkien kautta tiedon niistä erilliselle "GamePanelille", joka sitten sisällytettiin MapGraphiin.

Ajankäyttöarvio sekä toteutusjärjestys niinikään olivat hyvin kohdillaan. Sen sijaan aikataulutus oli poskellaan, ei niinkään suunnitelmassa kuin toteutuneessa elon kaaressa. Pahoittelen tässä myös myöhästynyttä palautusta. Projekti puskettiin lopulta valmiiksi lähes kokonaan noin puolessatoista viikossa aivan viime tingassa.

Ajanpuutteen takia jalankulkijat ja bussipysäkit jäivät uupumaan.

Arvio lopputuloksesta

Yleisesti ottaen olen hyvin tyytyväinen ohjelmaan. Autot toimivat hyvin, ja pelin kaatumiseen johtavia virheitä ei tapahtunut oman testaukseni aikana lainkaan. Silloin tällöin autot jumittuvat toisiinsa, mutta tämäkin on harvinaista. Pelin tunnelma on mielestäni kokonaisuutena hyvä, ainakin jos pitää nopeatempoisista peleistä joissa pitää hallita samanaikaisesti useita alueita. Kartta piirtyy oikein ja bugeitta, jos se vain on oikein rakennettu asetustiedostossa.

Paremman uudelleenkäytettävyyden saavuttamiseksi luokkien olisi kenties täytynyt toimia itsenäisemmin ja enemmän toisistaan irrallaan, mutten pidä toteutunuttakaan lopputulosta lainkaan hullumpana. Ainakin ohjelman sisällä laajennusta on mahdollista ja helppoakin tehdä monipuolisten ja mielestäni yleiskäyttöisten metodien ansiosta. Esimerkiksi auton tekoälyä voi kehittää halutessaan ja tiettyyn rajaan saakka irrallaan esimerkiksi teistä tai risteyksistä. Irrallaan ei siis tarkoita, etteikö esimerkiksi niihin liittyvää käyttäytymistä voisi kehittää, mutta jo tällaisenaan ne tarjoavat monipuolisesti metodeita esimerkiksi juuri auto-luokan käyttöön eikä juuri teiden ja risteysten koodiin tarvitse koskea.

Lähteet

[1] http://www.red3d.com/cwr/steer/gdc99/, Craig W. Reynolds 1999

http://docs.oracle.com/javase/7/docs/api/

http://www.scala-lang.org/api/2.10.3/index.html#package

Kurssin ICS-A1120 materiaali:

https://noppa.aalto.fi/noppa/kurssi/ics-a1120/luennot

Kurssin CSE-C2120 materiaali:

https://noppa.aalto.fi/noppa/kurssi/cse-c2120/luennot/CSE-C2120_luentokalvot_5.pdf, Otto Seppälä 2014

Suoraan tai lähes suoraan lainatut ohjelmakoodin pätkät on mainittu lähdekoodin kommenteissa:

http://stackoverflow.com/a/14819549/3280244, nimimerkki samthebest 5.5.2014

http://stackoverflow.com/questions/10570345/java-getaudioinputstream-symbol-not-found, nimimerkki Chin 4.5.2014

Liitteet

Lähdekoodi