Esa Paavilainen

Sähkötekniikan kandidaattiohjelma | 2. vuosikurssi

8.5.2020

Liikennesimulaatio

1. Yleiskuvaus

Valitsin projektiaiheekseni liikennesimulaation, jossa samassa tilassa olevat ajoneuvot pyrkivät noudattamaan omaa reittiään kolaroimatta toisiinsa tai ympäristöönsä. Ajoneuvot pyrkivät navigoimaan tilan läpi kohti tavoitteena olevaa maalipistettä. Ajoneuvojen liikettä ohjaavat seuraavat periaatteet:

- Ajoneuvot hiljentävät vauhtiaan kun ne huomaavat lähellä edessään muita, jotka liikkuvat hitaammin kuin ne itse (Braking)
- Ajoneuvot pyrkivät noudattamaan omaa reittiään (Seek / Path following)
- Ajoneuvot välttävät törmäilyä toisiinsa muuttamalla suuntavektoriaan sopivasti. (Separation)
- Ajoneuvot välttävät seiniä (Avoidance / Wall following / Containment) (ajoneuvot eivät halua joutua seinää vasten eivätkä etenkään seinän sisään, koska se on mahdotonta)

Omassa toteutuksessani ajoneuvot kulkevat kaupungin keskustassa, jossa on paljon risteäviä reittejä. Kenttä on jaettu neliönmuotoisiin paloihin, jotka voivat olla joko suoria tienpätkiä, mutkia tai risteyksiä. Tavoittelen korkeinta arvosanaa, joten olen tehnyt vaativan tason toteutuksen. Vaativa toteutus sisältää seuraavat ominaisuudet:

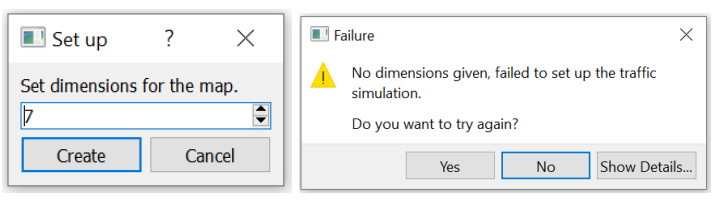
- Graafinen käyttöliittymä
- Tilassa on määriteltävää geometriaa (seiniä), jota ajoneuvot joutuvat väistämään
- Ajoneuvot seuraavat reittejä (joko ennalta määrättyjä tai hakualgoritmillä löydettyjä)

Toteutuksessani ei ole fyysisiä seiniä, vaan näiden vastineena toimivat kaistat, joita pitkin ajoneuvot kulkevat. Ajoneuvot pyrkivät pysymään kaistojen reunojen sisällä, joten kaistojen reunoja voi ajatella seininä. Kaistan sisäreunan määrittelee joko kaksinkertainen keltainen viiva tai yksinkertainen valkoinen katkoviiva. Ulkoreunan tunnistaa helposti kaistaa vierustavasta nurmikosta. Ohjelma hyödyntää Dijkstran algoritmiä minimaalisten polkujen luomiseen ajoneuvoille.

Ohjelmani ei täysin vastaa sitä, millaiseksi sen alun perin ajattelin muodostuvan. Ohjelmassa on enemmän tapoja vuorovaikuttaa sen kulkuun kuin alun perin suunnittelin, monet näistä tuli luotua testaamistarkoitusta varten, mutta osoittautuivatkin mukaviksi myös pitää osana lopullista versiota. Näitä ovat start/stop-, restart-, erase ja ''set vehicle count''-ominaisuudet, näistä lisää seuraavassa osiossa. Ohjelmassa on myös suunnitelmasta poiketen uusi luokka Radar. Radar-luokka luotiin muiden ajoneuvojen havaitsemista varten, koodin olisi voinut sisällyttää Vehicle-luokkaan, mutta näin oli mielestäni selkeämpi.

2. Käyttöohje

Ohjelma käynnistetään ajamalla gui-niminen moduuli, tämän ajettuaan näytölle ilmestyy seuraavanlainen laatikko:



Käyttäjä valitsee, kuinka iso kenttä luodaan. Tähän annettu kokonaisluku vastaa neliönmuotoisen kentän sivun mittaa. Sivu mitataan palikoiden (city blocks) lukumääränä. Annettu kokonaisluku pitää valita väliltä 3 ja 9 ja tämä vaikuttaa suoraan kentää kuvaavan olion rakenteeseen. Jos numeron valitseminen syystä tai toisesta epäonnistuu, tulee oikealla nähtävä viesti ruudulle. Viestilaatikko sallii käyttäjän yrittää uudestaan kentän luontia. Mikäli käyttäjä painaa ’’yes’’, tulee numerovalikko uudelleen näkyviin. Jos valinta epäonnistuu uudestaan, ohjelma sulkeutuu tällä kertaa.

CityCenter-olio vastaa siitä, miltä kenttä tulee näyttämään, mutta sillä on muitakin vastuualueita. Kun koko on valittu, aukeaa näytölle seuraavanlainen näkymä:

Simulaatiolla on aluksi perusasetukset ja simulaatio on pysäytetty, oikealle kuva käynnissä olevasta simulaatiosta. Perusasetuksilla tarkoitetaan simulaatiotyyppiä (casual/rush hour), ajoneuvojen toivottua lukumäärää ja erase-statusta. Ajoneuvojen lukumäärä ei kasva ennen kuin simulaatio on aloitettu. Käyttäjä pystyy vaikuttamaan ohjelman kulkuun nappeja painamalla, nappien nimet kuvaavat hyvin niiden toimintoja. Ylin nappuloista sallii käyttäjän pysäyttää/käynnistää simulaation koska tahansa. ’’rush hour’’ nappi muuttaa ajoneuvojen luonnetta ja nostaa niiden kokonaisylärajaa, ajoneuvot alkavat kiirehtiä eivätkä oikein piittaa liikennesäännöistä enää. Tässä yhteydessä liikennesäännöillä tarkoitetaan tasa-arvoisista risteyksistä tuttua oikean käden sääntöä.

’’new map’’-nappi pysäytttää simulaation ajakseen ja avaa samanlaisen numeronsyöttökentän kuin alussakin. Jos numeroa ei anneta, simulaatio jatkuu siitä mihin se jäi. Uuden kokonaisluvun antaminen puolestaan sallii käyttäjän koska tahansa luoda uuden kentän. Mikäli uusi kenttä luodaan, suurin piirtein kaikki oliot rakennetaan uusiksi, muutama poikkeusta lukuun ottamatta GUI-luokassa. Uuden kentän luominen vastaa käytännössä ohjelman ajamista alusta, perusasetukset palautetaan ja simulaatio on pysähtynyt. ’’restart’’-toiminto vastaa jossain määrin uuden kentän luomista, vaikka oikeastaan uutta kenttää ei luoda. Kun tätä nappulaa painaa, kenttä tyhjennetään ajoneuvoista ja oletusasetukset palautetaan, simulaatio vain on aluksi käynnissä. Vaikka uuden kentän loisi samankokoisena kuin tämänhetkinen, on se eri asia kuin uudelleen aloitus käyttäjänkin kannalta. Tämä johtuu siitä, että kentän luominen on osittain satunnaista. Samankokoisista kentistä ei toki ole loputtomasti eri variaatioita.

’’erase’’-nappula sallii nimensä mukaan ajoneuvojen vaivattoman poistamisen kentältä, riittää kun klikkaa nappulaa ja sen jälkeen poistettavaa kulkuneuvoa. Kumituksen voi asettaa pois päältä klikkaamalla nappulaa uudestaan. Jos ajoneuvoja klikkaa kumituksen ollessa pois päältä, tulee näkyviin pitkä jono pieniä mustia pisteitä ja läpinäkyvä punainen ympyrä. Mustat pisteet kuvaavat ajoneuvolle luotua reittiä ja punainen ympyrä (radar) aluetta, jonka ajoneuvo pystyy näkemään. Ajoneuvo seuraa polkua, kunnes se on saavuttanut maalinsa, punainen ympyrä taas seuraa ajoneuvoa. Ajoneuvojen poistaminen ei kuitenkaan laske ajoneuvojen lukumäärää, vaan tämä onnistuu valitsemalla uusi arvo numeronsyöttökentästä oikeassa alakulmassa. Kun ajoneuvoja katoaa maalipisteen saavutettuaan tai käyttäjän toimesta, niitä ilmestyy samaa tahtia kentäle. ’’exit’’-nappula keskeyttää ohjelman suorituksen.

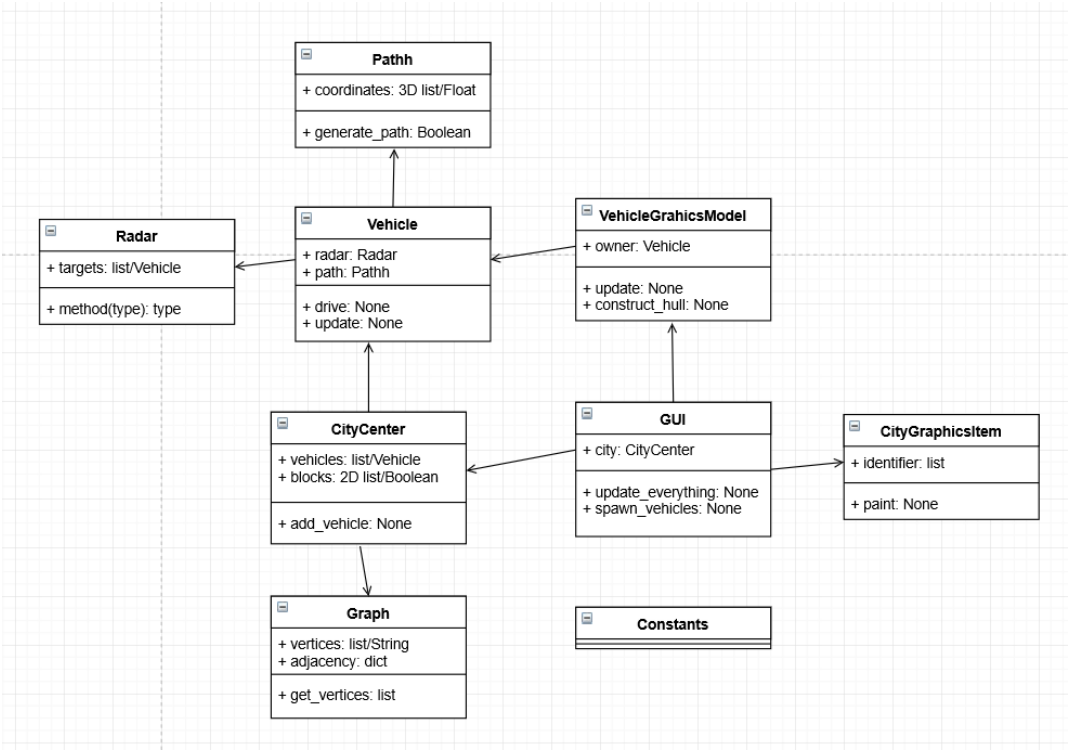
3. Ulkoiset kirjastot

Ohjelmani tarvitsee toimiakseen PyQt5:n kirjastot QtWidgets, QtCore, QtGui ja Qt. QtGui, QtCore ja Qt ovat välttämättömiä grafiikoiden luonnissa. Kentällä näkyvät ajoneuvot, niiden reitit ja koko kenttä on luotu m.m. luokkien QGraphicsItem, QGraphicsPolygonItem, QBrush, QColor ja QPointF avulla. QtWidgets on myös välttämätön m.m. applikaation ja ikkunan (QMainWindow) luomisessa. Kaikkia importattuja kirjastoja käytetään useissa luokissa. Luokkia, jotka tarvitsevat PyQt5 kirjastoja, ovat CityGraphicsItem, GUI ja VehicleGraphicsModel.

Ohjelma tarvitsee vielä kirjaston math ja funktion randint kirjastosta random. Muutamat toiminnot vaativat satunnaisuutta, näitä ovat ajoneuvojen ''spawnaaminen'' kentälle sekä muutama ajoneuvon attribuutti. Ennen ajonevon sijoittamista kentälle, ajoneuvon tyyppi ja väri arvotaan, näin saadaan mukavaa vaihtelua. Ajoneuvon sijoitus tehdään myös arpomalla, jotta ne leviäisivät tasaisesti. Math-kirjastoa tarvitaan, sillä Vehicle- ja Radar-luokat tekevät paljon laskentaa kulmien ja yksinkertaisten vektorilaskujen parissa. Tarvittavia funktioita ovat m.m. neliöjuuri, kosini ja sini.

4. Ohjelman rakenne

Ohjelma jakautuu seuraaviin luokkiin: GUI, CityCenter, Vehicle, Radar, Pathh, VehicleGraphicsModel, CityGraphicsItem, Graph ja Constants. Luokkien väliset riippuvuudet esitelty tarkemmin seuraavassa kuvassa:



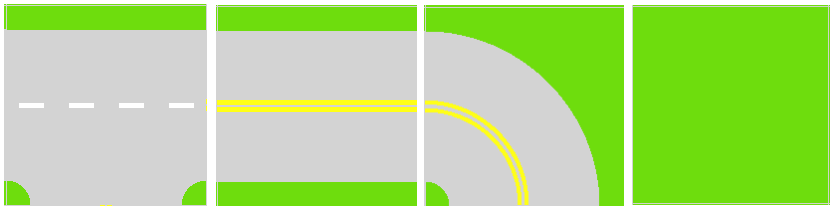
UML-luokkakaavio ohjelman rakenteesta. Kaikilla luokilla lukuun ottamatta luokkaa Graph, on riippuvuus luokasta Constants. Nämä riippuvuudet on jätetty merkitsemättä selkeyden vuoksi.

Vehicle-, Radar- ja Pathh-luokat ovat vastuussa ohjelman kannalta keskeisimpien ongelmien ratkaisusta. Yksinkertaistettuna Vehicle-luokka vastaa auton realistisesta liikkeestä, Pathh-luokka vastaa ajoneuvon kulkemasta reitistä ja Radar-luokka muiden ajoneuvojen havainnoimisesta. Vehicle on nimensä mukaan ajoneuvoa kuvaava luokka. Luokalla on lukuisia metodeja, joilla kuvataan ajoneuvoille tyypillistä liikettä. Vehicle-luokasta löytyvässä update-metodissa tehdään suurin työ, update-metodin sisällä kutsutaan yhteensä yhdeksää muuta metodia. Edellä mainituista tärkein ja mittavin on set_intersections, tässä luokassa alustetaan kaikki vaaralliset sijainnit

kyseiselle ajoneuvolle. Vaarallisia sijainteja ovat kohdat kentällä, joissa ajoneuvojen kulkemat reitit risteävät. Tämän metodin saamien tietojen avulla asetetaan lukuisia parametreja, jotka loppujen lopulta määräävät ajoneuvon nopeuden drive-metodissa. drive-metodi hyödyntää update-metodin avulla kerätyn tiedon; jos vaikkapa tulokseksi on saatu, että ajoneuvo on jumissa (blocked), tulee pysähtyä. drive-metodissa kutsutaan metodeita seek ja regain_course, nämä ovat tärkeitä, sillä ne korjaavat ajoneuvon kurssin, mikäli se poikkeaa toivotusta. Todellisuudessa seek-metodia ei oikein tarvita, tai jos tarvitaan, sen vaikutusta on vaikea huomata. regain_course pystyy pitämään ajoneuvon hyvin reitillään.

Radar-luokan kiistämättä tärkein metodi on intersects, tämä huomioi kaikki kantaman sisällä olevat ajoneuvot ja päättää jokaiselle yksi kerrallaan, tulisiko Radarin omistajan tehdä tietä (yield), seurata tai jättää kyseinen ajoneuvo huomiotta. Tätä metodia kutsutaan Vehicle-olion metodista set_intersections. Pathh-luokan selvästi tärkein metodi on generate_path, generate_path hyödyntää Dijkstran algoritmia minimaalisen polun virittämiseen. Ajoneuvolle luodaan reitti kentän yhdeltä laidalta toiselle, lähtöpisteen ja maalipisteen välille. Pathh-olio saa tiedon kentän rakenteesta ja toivotuista maali- sekä lähtöpisteistä suoraan CityCenter-oliolta. Reitti on mahdollisista vaihtoehdoista lyhin. Kun tämä on tehty, ajoneuvo voidaan tuoda kentälle. Ennen tätä ajoneuvolla ei ole määriteltyä sijaintia, nopeutta tai montaa muutakaan toiminnan kannalta keskeistä parametria alustettuna. Dijkstran algoritmi hyödyntää Graph-oliota, johon on tallennettu tiedot kentän (CityCenterin) risteyksistä, niiden välisistä etäisyyksistä sekä suunnasta, johon tulee kulkea päätyäkseen merkattuun risteykseen.

Käyttöliittymää kuvataan GUI, VehicleGraphicsModel- ja CityGraphicsItem-olioilla. CityGraphicsItem-olioilla mallinnetaan CityCenter-olion rakennetta. Näitä luodaan yhtä monta kuin kentässä on neliöitä, jokainen vastaa yhtä palaa kentässä. Jos kenttä on vaikkapa 7x7, paloja luodaan 49 kappaletta, poikkeuksena tähän on 3x3 kenttä, jota varten luodaan 15 palaa. Pala voi olla risteys, mutka, suora tie tai pelkkää nurmikkoa. Se, millaiset palat luodaan, riippuu täysin CityCenter-olion attribuutista ''blocks''. Ainutlaatuisia vaihtoehtoja on yhteensä 12 kappaletta, attribuutti identifier määrää, mikä näistä piirretään paint-metodissa. Alla kuvat muutamasta erilaisesta CityGraphicsItem-oliosta:



VehicleGraphicsModel Vehiclelle on kuin CityGraphicsItem CityCenterille, erona vain on se, että yhtä ajoneuvoa kohden riittää yksi VehicleGraphicsModel-olio. VehicleGraphicsModel-oliolla on attribuuttinen kyseinen Vehicle-olio, josta saa kätevästi sijainnin ja halutun kääntökulman. Mikäli ajoneuvon väriä ei oteta huomioon, VehicleGraphicsModel-olioita on kolme erilaista.

GUI-luokka on komentoketjussa kaikista ylimpänä, GUI rakentaa koko näkymän kahden laatikkon muotoisen layoutin avulla. Ulompi layout (QBoxLayout) pitää sisällään sisemmän layoutin (QVBoxLayout). Ulommassa layoutissa on itse simulaatio ja sisemmässä widgetit päällekkäin asetettuna, joilla käyttäjä pystyy vaikuttamaan simulaation kulkuun. GUI:n tärkein metodi on update_everything. Tätä metodia kutsutaan, syklisesti QTimerin avulla. Mikäli simulaatio ei ole pysäytetty ja kentällä on tilaa, update_everything kutsuu toista tärkeää metodia, spawn_vehiclesia. Spawn vehicles hyödyntää satunnaisuutta uusien ajoenuvojen tuomisessa kentälle niin, että ajoneuvot eivät näytä ilmestyvän heti edellisen kadottua. Jos poikkeama tavoitteesta on yli 2 ajoneuvoa, tai ruuhka-aikana yli 1 ajoneuvo, näitä ilmestyy lisää ilman viivettä. Metodi Remove_items huolehtii, että kaikkien poistettavien ajoneuvojen grafiikat poistetaan grafiikkanäkymästä välittömästi. Tieto tarpeettomista ajoneuvoista saadaan myös update_everythingissä, kun kutsutaan CityCenterin metodia update.

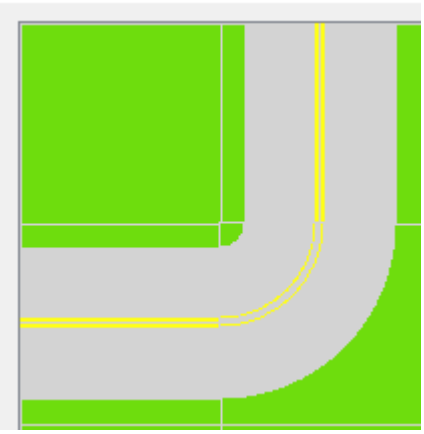
CityCenter toimii välikätenä GUI:lle, toteutus olisi onnistunut ilmankin CityCenteriä, mutta tein tämän jaon selkeyden vuoksi. Huomasin jo alkuvaiheessa, että GUI-luokka tulee tekemään paljon

kaikenlaista. Luokassa on nytkin yli 700 riviä koodia, josta valtaosaa osaa pystyy uudelleenkäytämään, joten jako oli ihan järkevä päätös. Toisaalta koin, että olisi kätevää, jos GUI pääsisi suoraan käsiksi VehicleGraphicsModel- ja CityGraphicsItem-olioihin. Tämän takia GUI:lla on suora assosiaatio näiden luokkien kanssa. Vehicle, Radar ja Pathh kuuluvat samaan kategoriaan, mutta kaikkea koodia ei tässäkään ole laitettu saman luokan sisälle. Selkeä työnjako on tehty, sillä Vehicle-luokassa on yli 1000 riviä koodia. Osan tästäkin olisi voinut vielä siirtää Pathh-luokkaan.

Muutamat valmiit python luokat osoittautuivat eritysen hyödyllisiksi, näitä ovat m.m. QGraphicsEllipseItem, QPolygonF, QPushButton ja QInputDialog. Monet asiat sattuvat olemaan ympyrän muotoisia, kuten ajoneuvojen polkujen koordinaatit ja ajoneuvojen Radarin kattama alue. QPolygonF mahdollisti vaivattoman ajoneuvojen luonnin, näihin eivät olisi riittäneet pelkät suorakulmiot ja ellipisit. QPushButtonit sallivat käyttäjän vaivattomasti vaikuttaa ohjelman kulkuun, mitään ei tarvitse edes kirjoittaa. Ohjelmassa on yhteensä 6 eri QPushButtonia itse simulaation oikealla puolella. QInputDialog-luokka sallii vaivattoman kokonaisluvun syöttämisen alkuperäistä tai uutta kenttää luodessa. QInputDialogin tekemiseen mennyt koodi oli myös helppo uudelleenkäyttää.

5. **Algoritmit**

Ohjelmassa käytetään laajasti tunnettuja Dijkstran- ja DFS-algoritmia. Dijkstran algoritmi luo ohjelmassa esityksen lyhimmästä reitistä maalisolmusta ja lähtösolmuun. Esitys on sanakirja prev, johon on tallennettu tieto kutakin solmua edeltävästä solmusta täyttäen minimaalisen etäisyyden ehdon. Näistä siis riittää, kun tarkastellaan valittua maalisolmua ja valitaan kaikki edeltävät solmut yksi kerrallaan. Kun solmut maalin ja lähdön välillä on valittu, tulee vielä järjestys kääntää, jotta saadaan reitti lähtöpisteestä maaliin. Joissain poikkeustapauksissa polkua valittujen pisteiden välille ei pystytä muodostamaan, tätä varten tulee tarkastella toista Dijkstran algoritmin antamaa tulosta, sanakirjaa dist. dist kertoo etäisyyden kuhunkin solmuun, alussa solmujen etäisyyksien arvot alustetaan äärettömäksi, paitsi lähtösolmu asetetaan nollaksi. Jos jokin solmu jää kokonaan käymättä läpi, jää siis tämän etäisyys äärettömäksi ja tätä on turha etsiä sanakirjasta prev. Jos siis tavoitesolmun etäisyys jää arvoon ääretön, tulee yrittää uudestaan uudella maalisolmu-lähtösolmu-parilla. Tämä on täysin mahdollista, sillä useassa eri kentässä on paloja, joita ei voi saavuttaa ulkopuolelta. Tästä kuva alapuolella:



Kuvassa näkyvät kolme palaa on otettu kentän vasemmasta yläkulmasta, palat ovat saavuttamattomissa, mikäli ajoneuvo tulee kentälle jostain muualta kuin kahdesta havaittavasta reunasta. Toisaalta, jos ajoneuvo saapuu kentälle jostain muusta sisääntulosta kuin kuvassa näkyvistä, ovat kuvan kolme palaa ajoneuvon ulottumattomissa. Dijkstran algoritmin pseudokoodi näyttää seuraavalta:

Dijkstras algorithm(Graph, target):

V = Graph.vertices

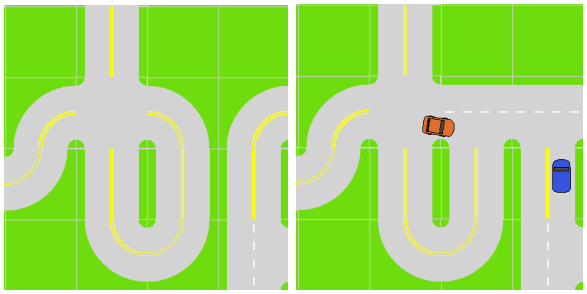
Adj = Graph.adjacency

for vertex in V:

```
        dist[v] = inf
        prev[v] = undef
dist[start] = 0
while verices not empty:
        u = vertex in V with mid dist[u]
        V.pop(u)
        for every adjacent vertex v for u and vertex v still in V:
                altdist = dist[u] + edgelength(u, v)
                if altdist < dist[v]:
                        dist[v] = altdist
                        prev[v] = u
return dist, prev
if dist[target] equals inf:
        fail
        get new goal
```

Oma versioni tästä poikkeaa melko paljon alkuperäisestä, sillä polkua tehdessä tulee ottaa huomioon myös risteysten väliet palat, risteysten välille ei voi vai vain merkata koordinaatteja suoraan. Oma versioni ottaa myös huomioon suunnan, johon verkossa edetään. Tämän avulla saadaan tieto siitä, tulisiko polkuun liittää suora vai kaareva pala. Dijkstran algoritmille annetussa Graph-oliossa on merkitty naapurisolmun nimi, etäisyys ja suunta. Kun Dijkstra on valmis ja polku luodaan pala kerrallaan, tarvitaan vielä tieto kaikista solmuja yhdistävistä paloista. Tätä varten Pathhin generate_path saa syötteeksi myös CityCenterin koko rakenteen, city_blocks:n.

Toinen mainitsemani algoritmi, DFS, löytyy luokasta CityCenter. Metodi check_for_dead_loops kutsuu rekursiivista algoritmia DFS ”kuolleiden” silmukoiden löytämiseksi. Poikkeavia silmukoita ovat seuraavissa kuvissa esiintyvät silmukat. Vasen kuva edustaa kuollutta silmukkaa ja oikea muuten vain hyödytöntä silmukkaa (kuollut silmukka on myös hyödytön):



Päätin jo alkuvaiheessa tehdä metodin, joka estää kuolleita silmukoita muodostumasta. Tein tämän päätöksen, koska oletin kuolleiden silmukoiden johtavan kaikenlaisiin ongelmiin m.m. Graph-olion ja polun luonnissa. Vasten kaikkia odotuksia, ohjelma toimii moitteetta, vaikka kentässä olisi kuolleita silmukoita. Tämä tuli minulle täytenä yllätyksenä, mutta on ihan mukavaa kuitenkin, ettei tällaisia silmukoita jää kenttään. Kuolleet silmukat tyypillisesti pirstaloivat kentän pieniin paloihin ja ovat muutenkin täysin hyödyttömiä.

Oikeassa kuvassa oleva silmukka on myös hyödytön, mutta CityCenter ei tee mitään niiden poistamiseksi. Ajoneuvot eivät koskaan valitse kahden risteyksen välin kulkemiseen pidempää reittiä, joten nämä todistavat Dijkstran algoritmin toimivan. Tällaisten silmukoiden havoinnointi olisi myös paljon työläämpää kuin kuolleiden silmukoiden havaitseminen.

DFS käy rekursiivisesti läpi CityCenterin rakennetta, rakenne on ilmaistu attribuutilla blocks. Algoritmissä jokainen tarkastamaton neliö käydään läpi kertaalleen, ennen aloittamista kaikki nurmikko- ja reunapalat merkitään tarkastetuiksi. Palojen läpikäyminen aloitetaan ensimmäisestä vierailemattomasta palasta vasemmassa yläkulmassa. Pala merkitään vierailluksi ja siirrytään seuraavaan palaan. Seuraavia vaihtoehtoja on nollasta kolmeen riippuen kyseisen palan muodosta ja siitä, onko naapuripala jo merkattu tarkastetuksi. Suunnasta (0,1,2,3), josta tähän palaan saavuttiin, huomioidaan myös, jotta ei liikuttaisi takaisin sinne, mistä tultiin. Ensimmäisenä kokeillaan liikkumista suuntaan 0, eli oikealle, tämän jälkeen suuntaan 1, eli ylös jne. Suuntiin, joista löytyy vain nurmikkoa, ei luonnollisesti liikuta.

Liikkuminen tapahtuu syvyyssuunnassa, eli kaikki oikeasta suunnasta löytyvä (vaikka koko kenttä) käydään läpi ennen kuin kokeillaan seuraavaa suuntaa. Algoritmi pitää kirjaa siitä, mitkä olivat edeltävän risteyspalan vaaka- ja pystyindeksi. Kun saavutaan seuraavaan risteyspalaan, tarkastetaan ovatko edellisen risteyksen indeksit samat kuin tämän risteyksen. Jos edellä mainittu ehto toteutuu, tarkastelu on siis aloitettu samasta risteyksestä, kuin mihin päädyttiin niin, että välissä ei ollut muita risteyksiä. Tämä viittaa kuolleeseen silmukkaan. Jos taas koko kenttä merkitään vierailluksi löytämättä kuolleita silmukoita, ei uutta kenttää tarvitse luoda. Metodia check_for_dead_loops voi joutua kutsumaan DFS-algoritmia useammin kuin kerran, mikäli kentästä löytyy aiemmin mainittuja irrallisia tiepaloja. Tässä pseudokoodi esitys algoritmista, indeksit i ja j kasvavat oikealle ja alaspäin edetessä

DFS(Visited, i, j, i_previous_crossing, j_previous_crossing, previous_direction):

Visted[i][j] = true

if intersection_piece:

    i_previous_crossing = i

    j_previous_crossing = j

if not lawn on the right and previous_direction not from_the_right:

    if to the right visited:

        if previous_crossing on the right:

            dead = true

    else: DFS(Visited, i+1, j, i_previous_crossing, j_previous_crossing, from_the_left):

if not lawn on above and previous_direction not from_above:

    if above visited:

        if previous_crossing above:

            dead = true

    else: DFS(Visited, i, j-1, i_previous_crossing, j_previous_crossing, from_below):

if not lawn on the left and previous_direction not from_the_left:

    if to the left visited:

        if previous_crossing on the left:

            dead = true

    else: DFS(Visited, i-1, j, i_previous_crossing, j_previous_crossing, from_the_right):

if not lawn on below and previous_direction not from_below:

    if below visited:

```
            if previous_crossing below:
                dead = true
        else: DFS(Visited, i, j+1, i_previous_crossing, j_previous_crossing, from_ above):
```

Helpommalla olisi päässyt, jos polut vain arpoisi kentän halki ja kuolleita silmukoita ei tarkastettaisi. Halusin kuitenkin tehdä simulaatiosta mahdollsimman todentuntuisen, joten päädyin tähän ratkaisuun. Muut toteutukset voisivat sisältää vapaampia liikkumisasteita ajoneuvoille, mutta näillä ei varmaankaan voisi tavoitella korkeinta arvosanaa.

Tietorakenteet

Ohjelmoin ohjelman käyttöä varten verkko-tietorakenteen luokassa Graph. Tällä mallinnetaan CityCenter-olion risteyksiä ja mahdollistetaan Dijkstran algoritmin käyttö myöhemmin ohjelmassa. Graph tallentaa tietoa verkon särmien pituuksien ja nimien lisäksi myös suunnista, josta viereiset solmut löytyvät. Tätä tietoa ei tarvita Dijkstran algoritmissa, vaan myöhemmin, kun polku luodaan palikka kerrallaan. Graphilla on kaksi attribuuttia: vertices ja adjacency. Vertices on lista solmuista, jokainen solmu on nimetty sen sijainnin mukaan kentällä. Palikassa indekseillä 4, 5 oleva risteys muodostaa solmun ’45’ jne. Jokaista kenttää reunustavaa suoraa tiepalaa ajatellaan myös solmuina. Muista poiketen, näillä on vain yksi naapurisolmu. Solmusta saadaan myöhemmin ajoneuvon sijainti nopeasti muuttamalla solmun arvot kokonaisluvuiksi ja kertomalla palikoiden koolla.

Adjacency on sanakirja, jossa avaimina toimivat listasta vertices löytyvät solmut. Sanakirjan arvoja ovat kolmialkioiset monikot. Monikon ensimmäinen termi on naapurisolmu, toinen etäisyys solmuun ja kolmas suunta tähän. Etäisyys määritellään kuljettavien palikoiden lukumääränä, ei absoluuttisena etäistyytenä (vektorina), esimerkiksi saman sivun jakavien palikoiden välinen etäisyys on 1 ja saman kulman jakavien 2. Suunta on kokonaisluku nollasta kolmeen, nolla tarkoittaa oikealle, yksi ylöspäin, kaksi vasemmalle ja kolme alaspäin.

Ohjelmassa muutamat luokat varastoivat tietoa rakenteestaan moniulotteisina listoina, näitä ovat luokat Pathh ja CityCenter. CityCenterin attribuutti blocks on kolmiuloitteinen lista, jossa ensimmäiset kaksi indeksiä kertovat palikan sijainnin kentällä. Sisäkkäisin lista, jonka olisi voinut myös ilmaista myös monikolla, on kyseinen palikka (block). Palikassa on neljä binääristä arvoa ja näitä on yhteensä 12 erilaista. Ainoa kielletty yhdistelmä on palikka, jossa on vain yksi yksi ja kolme nollaa. Indeksi nolla vastaa oikeaa sivua, indeksi yksi yläsivua jne. Jos sivua vastaava arvo on tosi, on palikkaan kulkuyhteys tältä sivulta. Kielletty palikka vastaisi umpikujaa, mikäli sellainen olisi olemassa.

Pathh-luokan attribuutti coordinates on myös 3-uloitteinen lista, ensimmäinen indeksi vastaa ensimääiselle palikalle mahtuvia koordinaatteja. Koordinaattipareja on yhdelle palikalle tyypillisesti 20, mutta mikäli polku tekee vain lyhyen käännöksen tällä palikalla, on näitä vain 12. Koordinaateissa sisimmäisin lista on fyysinen sijainti, x- ja y-koordinaatti. Fyysistä sijaintiakin olisi toki voinut ilmaista kaksialkioisella monikolla, sillä se ei tule muuttumaan ohjelman kulun aikana. Attribuuttia coordinates ei voi sellaisenaan käyttää, vaan Vehicle tekee tästä itselleen kaksiuloitteisen listan metodissa set_relevant_coordinates. Relevantteihin koordinaatteihin lisätään vain lähimmästä neljästä palikasta saadut koordinaattilistat, joista niistäkin hylätään tarpeeksi kauas taakse jääneet koordinaattiparit.

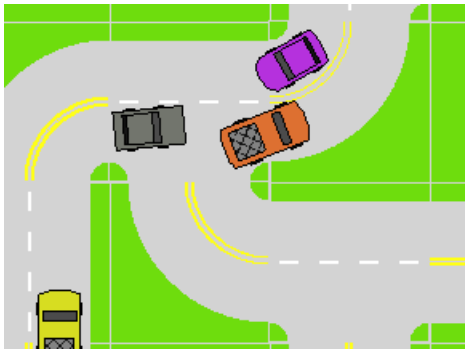
6. Tiedostot

Ohjelma ei käsittele tiedostoja, kaikki tarpeellinen löytyy lähdekoodista.

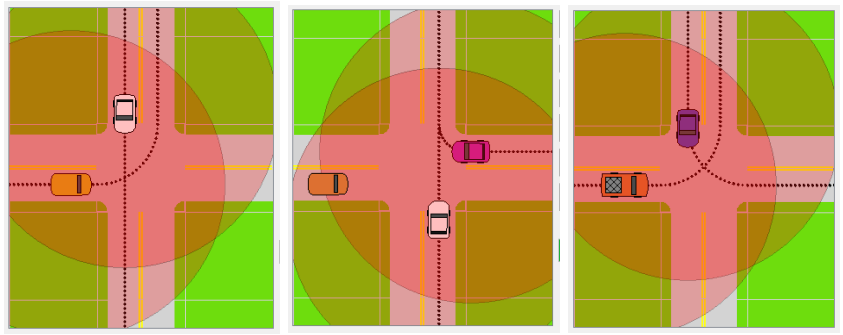
7. Testaus

Ohjelma läpäisee suunnitteluveiheen testit helposti, ajoneuvot pysyvät reiteillään vaivattomasti kovemmissakin nopeuksissa. Tätä testattiin aluksi yksinkertaisilla valmiiksi rakennetuilla poluilla. Paljon haastavammaksi osoittautui muiden autojen liikkeeseen reagoiminen. Ajoneuvoilla on monta attribuuttia (slows, yields, blocked, is_blocking) juuri tätä varten. Kentällä on kuitenkin lukuisia autoja kerrallaa, joten näiden printtaaminen osoittautui toivottomaksi. Tätä varten kehitin väliaikaisn metodin cybertruck VehicleGraphicsModel-luokassa. Tällä tarkasteltiin vastaavan

ajoneuvon yhtä attribuuttia kerrallaan, attribuutin muuttessa todeksi, VehicleGraphicsModel muuttui kulmikkaaksi. Tämä salli nähdä vaivattomasti, milloin ajoneuvo päätti tehdä esimerkiksi päätöksen väistää toista. Alla olevassa kuvassa harmaan ajoneuvon attribuutti blocked muuttui todeksi oranssin ajoneuvon tultua tämän eteen, printtaamalla kaikkien ajoneuvojen vastaavaa attribuuttia terminaaliin tämä ei olisi ollut niin helppoa.



Toinen suosimani tapa tehdä yksikkötestauksia oli luoda 3x3-kenttä ja katsoa, kuinka ajonevot reagoivat toisiinsa. 3x3-kenttä on ainoastaan yksi nelihaarainen risteys, johon saapuu monta ajoneuvoa eri suunnista. Tämän avulla näki nopeasti, kuinka kaukana kulkuneuvojen tulisi pysyä toisistaan kussakin tilanteessa, jossa niiden kulkemat reitit risteävät. Etäisyyksien määrittely olisi ollut tuskaista isommalla kentällä, jossa on tilaa vaihteluun. Alla esimerkki muutamasta risteystilanteesta:



8. **Ohjelman tunnetut puutteet ja viat**

Ohjelmassa liikkuvat ajoneuvot saattavat raapaista toisiaan tai poikkeustilanteissä päätyä päällekkäin. Näiden ongelmien todennäköisyys kasvaa sitä suuremmaksi, mitä enemmän ajoneuvoja tuodaan kentälle. Ajoneuvojen pitämät etäisyydet toisistaan perustuvat hyvin pitkälti niiden polkujen leikkaamiskohtaan ja leikkausten väliseen kulmaan. Olen parhaani mukaan pyrkinyt määrittelemään sopivia etäisyyksiä, joita ajoneuvojen tulee pitää ollakseen koskematta toisiinsa, mutta tähän asti nähty vaiva ei vielä täysin riitä. Isompi vika löytyy kuitenkin Radar-luokan intersects metodin sisältä. Metodin sisälle on määritelty funktio identical_paths, joka ei aina pysty tunnistamaan oman reitin yhtymistä toisen kanssa. Virhe johtuu siitä, että eri polkujen osilla saattaa olla eri etäisyydet peräkkäisten koordinaattipisteen välillä, joka jää huomiotta vikatilanteessa. Vikatilanne huonolla tuurilla johtaa yhden ajoneuvon liukumisen toisen päälle tästä täysin piittaamatta.

Ajoneuvot saattavat joutua sumppuun, josta ne eivät pääsisi pois muuten kuin muuttamalla reittiään tai peruuttamalla. Näitä ominaisuuksia ei kuitenkaan tueta, joten tilanteelle ei ole ”älykästä” ratkaisua, vaan simulaatio tulee joko aloittaa alusta (restart tai new map) tai pahasti sumpussa olevat ajoneuvot tulee manuaalsiesti poistaa. Vehicle-luokasta löytyvä solve_standstill pyrkii ratkaisemaan tällaisia tilanteita, mutta ei pysty kaikkeen. Jumitilanteiden todennäköisyys kasvaa ruuhkaa simuloidessa ajonevojen määrän kasvaessa. Ruuhkasimulaatiossa ajoneuvot käyttäytyvät itsekkäämmin ja saattavat näin ollen ahtaa itsensä toivottomaan rakoon. Jumitilanteet

pystyisivät ehkäisemään, mikäli ajoneuvot pystyisivät havainnoimaan ympäristöään vielä tarkemmin. Tämä vaatisi havainnointialueen kasvattamista, kokonaan uusia metodeita tai jopa molempia. Uudet metodit voisi toki kehittää, jos aikaa olisi, mutta Radar-luokan kantaman tuplaaminen suurin piirtein nelinketaistaa tämän tekemän työmäärän. Radar-luokan metodi intersects käy läpi kahta sisäkkäistä for-silmukkaa ja on tämän takia aiheuttanut ohjelman hidastumista aiemmin.

9. 3 parasta ja 3 heikointa kohtaa

Mielestäni projektin 3 parasta kohtaa on sen helppokäyttöisyys, monipuolisuus ja visuaalinen miellyttävyys. Ohjelmaa käyttäessä ei tarvitse edes kirjoittaa mitää, kunhan vain valitsee kokonaisluvun kolmesta yhdeksään, loput tapahtuvat nappeja painamalla. Monipuolisuuden tarjoaa useat toiminnot, näistä ennen kaikkea uuden kentän luominen kesken simulaation. Kenttien luominen on osittain satunnaista, joten näitä riittää. Moniväriset ja muotoiset ajoneuvot sekä värikäs kenttä tekevät simulaatiosta mieluisen katsella. Olen katsonut ohjelmaani pidempään kuin kukaan muu, enkä ole itse edes kyllästnyt.

Jos edellisen kappaleen virheitä ei lasketa mukaan, sanoisin ohjelman heikkoudeksi sen, kun ikkunan laittaa fullscreen-muotoon. Puolikkaan näytön mittaisiksi venyneet nappulat eivät näytä kovin kivoilta. Ohjelma voi saada koneen hyrräämään, mutta tämä tapahtuu vasta kovemmilla asetuksilla. Oma tietokoneeni on melko tehoton, eikä sekään ole käytössä hidastunut. Olen nähnyt suhteellisen ison vaivan hidastumisen ehkäisemiseksi Vehicle-luokan metodeissa set_relevant_coordinates ja set_intersections.

Poikkeamat suunnitelmasta

Lopullinen versio on monimutkaisempi kuin alun perin suunniteltu, alun perin en suunnitellut tarvitsevani Radar-luokka tai niin montaa tapaa vaikuttaa ohjelman kulkuun kuin mitä nyt löytyy. Moni ominaisuus tuli tehtyä testaamista varten, mutta osoittautui pienellä lisävaivannäöllä toimivaksi ominaisuudeksi. Näitä ovat start/stop-, erase- ja restart-nappula. Minulla ei ollut alkuperäistä aika-arviota, mutta olin oikeassa siinä, että aikaa tulee kulumaan paljon yli 100 tunnin. Tiesin, että ohjelma tulee olemaan työläs. Sanoisin, että tehokasta työskentelyä on ollut 150 tuntia. Toteutusjärjestys oli juuri sellainen kuin ajattelin sen alunperinkin olevan, tein ensin komentoketjussa pohjalla olevat luokat ja metodit ja tämän jälkeen vasta näiden avulla toimivat luokat ja metodit. Järjestys osoittautui toimivaksi ja luotettavaksi, kun ei tarvinnut arvailla usean eri asian joukosta, mikä oikein on vialla ja mikä ei.

10. Toteutunut työjärjestys ja aikataulu

Kuten jo mainitsin aiemmassa kappaleessa, tein pohjimmaiset luokat valmiiksi ennen päällimmäisiä. Pohjalla olevia luokkia ovat Vehicle, Radar ja Pathh. Kaikista ylimpänä on GUI. Näiden välissä toimii CityCenter, VehicleGraphicsModel ja CityGraphicsItem. Pohjimmaiset ja keskimmäiset luokat olivat pääpiirteittäin valmiita huhtikuun puoleen väliin mennessä, GUI on valmistunut lopulliseen muotoonsa toukokuun ensimmäisellä viikolla. Vehicle- luokkan koodimäärä kasvoi tasaisesti huhtikuun puolenvälinkin jälkeen uusien ominaisuusten myötä, Vehicle on luokista koodimäärältään selvästi suurin. Projekti rakennettiin kuin pyramidi, maasta pyramidin kärkeen.

11. Arvio lopputuloksesta

Mielestäni projekti on onnistunut, se on helppokäyttöinen, selkeä ja sitä on mukava katsella. Ohjelman kanssa ei tylsisty nopeasti, sillä se sallii käyttäjän vaikuttaa simulaation kulkuun monella tavalla. Vikatilanteessakin käyttäjän on helppo poistaa ongelmia aiheuttava kulkuneuvo pois kentältä ja vikatilanteet ovat suhteellisen harvinaisia. Vikatilainteita ei okein esiinny, ennen kuin käyttäjä laittaa ruuhkasimulaation päälle ja nostaa ajonevojen lukumäärärän maksimiarvoonsa. Koodi on kauttaaltaan kommentoitua ja tehty niin, että sitä pystyvät muutkin kuin minä ylläpitämään.

12. **Viitteet**

A+:ssa linkattu Craig Reynoldsin sivu:

http://www.red3d.com/cwr/steer/gdc99/

Hyödylliseksi osoittautunut Youtube-kanava:

https://www.youtube.com/watch?v=JIz2L4tn5kM&list=PLRqwX-V7Uu6YHt0dtyf4uiw8tKOxQLvlW

https://doc.qt.io/archives/qtjambi-4.5.2_01/overview-summary.html

Qt-dokumentaatio

https://doc.qt.io/

PyQt-dokumentaatio

https://doc.qt.io/qtforpython/