Y2 Ohjelmointiprojekti, Loppudokumentti

Rasmus Siljander

665649

Bioinformaatioteknologia

23.4.2019

Yleiskuvaus

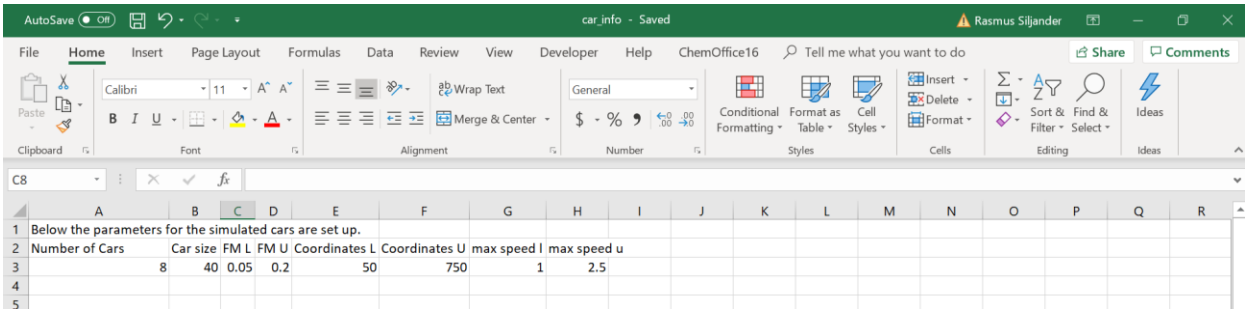
Ohjelma on liikennesimulaatio, jossa ikkunaan kuvitetaan autoja (punaisia kolmioita), jotka pyrkivät kulkemaan alkupaikastaan päätepisteeseen. Autot kuitenkin tarkkailevat ympäristöään ja havaitsevat sellaiset tilanteet, jossa autot ovat törmätä toisiinsa. Tällöin on autojen suoritettava väistöliikkeitä ja jarrutettava tai käännyttävä muista autoista poispäin. Lähestyessään päätepistettään, autot aloittavat hidastamisprosessin, jossa autojen vauhtia hiljennetään lineaarisesti. Kun kaikki autot ovat päässeet maaliinsa, ohjelma suljetaan poistumiskäskyllä.

Projektin vaikeusaste on keksivaikea, eli ikkunaan ei piirretä esteitä. Alun perin projektiin ei oltu suunniteltu poistumiskäskyä, joten sellainen lisättiin. Tämän lisäksi aiemmin suunniteltu polunseurausominaisuus poistettiin.

Käyttöohje

Ohjelma toimii ajamalla main.py moduuli. Ohjelma saa parametrikseen csv-tiedoston nimen, mikä sisältää kaikki simulaatioon tarvittavat tiedot.

Kaikki autojen tiedoista arvotaan – autojen määrä, koko, maksiminopeus, ohjautuvuuskyky, alkupiste, loppupiste. On kuitenkin tärkeää, että arvotut numerot ovat mielekkäitä. Tämän takia csv-tiedostoon on kirjoitettu luotavien autojen määrän lisäksi välit, jolla arvot ovat hyväksyttäviä. Esimerkkitiedosto:



Rivi 1: Header-tietoa tiedososta.

Rivi 2: Otsikot kolmannen rivin arvoille. Tunnisteet:

- L = alaraja (Lower Bound)
- U = yläraja (Upper Bound)
- FM = Force to mass ratio. Kertoo auton kyvyn vaihtaa suuntaa. Seuraa Newtonin toisesta liikelaista ($F = ma \rightarrow a = F/m$)
- Size = auton koko (pikseleinä)
- Coordinates = koordinaattirajat, jonka välille alku- ja loppukoordinaatit arvotaan.
- Max speed = maksiminopeus

Rivi 3: Arvot. Arvoilla on kuitenkin rajoja, minkä välille numeroiden on määrä tulla. Kaikki numerot ovat itse määritelty empiirisen kokeilun perusteella.

$$1 \leq number\ of\ cars \leq 15$$

$$30 \leq size \leq 50$$

$$0.05 \leq fm \leq 0.20$$

$$50 \leq coordinates \leq 750$$

$$0.75 \leq \max speed \leq 2.50$$

Käyttäjän ei siis tarvitse muuta kuin tallentaa csv-muotoiseen tiedostoon tiedot autoja varten ja antaa se parametrina ajettavalle funktiolle.

Ohjelman saa myös pysäytettyä ennen simulaation valmistumista painamalla ”*press to exit*” napista.

Ulkoiset kirjastot

Tähän projektiin on käytetty pääosin Pythonin standardikirjastoa. Vektorilaskennan apuun on käytetty moduulia $math$ ja numeroiden arvontaan on käytetty $random$ – moduulia. Graafisen käyttöliittymän apuun on käytetty *PyQt5*:n kirjastosta moduuleja $QtWidgets$, $QtCore$ sekä $QtGui$.

Ohjelman rakenne

Ohjelman toteutus toimii seuraavanlaisella toimintajärjestyksellä:

1. Tietojen lataaminen

    a. Tiedoston avaaminen
    b. Tietojen tallentaminen tiedostosta
2. Autojen luonti tietojen perusteella
3. Graafinen käyttöliittymä
    a. GraphicsItemien luonti
    b. Itemien lisääminen GUI-ikkunaan.
4. Simulaation ajaminen (timerEvent).

Ohjelman luokat

- $CarInfoFileError$
- GUI
- $Path$
- $Vector$
- $Vehicle$
- $VehicleGraphicsItem$
- $VehicleWorld$

Ensimmäiseksi käytettävä luokka on $VehicleWorld$, mikä on maailma, jossa kaikki autot sijaitsevat. $VehicleWorldin$ luonti on tärkeää, koska se pitää sisässään kaikki autot ja tällöin kaikki tiedot kaikista autoista ja koko simulaatiosta. $VehicleWorldia$ kutsutaan main-moduulissa, eikä sille anneta parametrinään mitään.

$VehicleWorldin$ alustamisvaiheessa määritellään Worldin autojen lista. Lista on alussa tyhjä (Worldiin pitää siis lisää autoja sisäänrakennetulla metodilla $add\_vehicle(vehicle)$). $VehicleWorld$ pitää sisällään $Vehicle$- olioita.

$Vehicle(fm, size, \dots)$ luo $Vehicle$ (auto) – olion. Auto saa parametrina fm-luvun, koon, maksiminopeuden, polun (kaksiulotteinen lista, joka pitää sisällään alku- ja päätepisteet (x-, y-koordinaattien lista), suunnan (vektori alkupisteestä päätepisteeseen) sekä maailman ($type\ VehicleWorld$) mihin auto lisätään. $Vehicle$-luokassa sijaitsee myös metodit auton liike-algoritmeille.

$Vector(x, y)$ – luokka luo vektori-olioita (joilla on x- ja y-komponentti). Vektori-olioiden luonti tekee graafisen käyttöliittymän paikka-, etäisyys- ja suuntalaskelmien tekemisestä helpompaa. Luokassa on metodit kaikille lisäys-, erotus- ja kokolaskelmille.

$Path(start, finish)$- luokka luo polku-olion, mikä pitää tietoa auton polusta. Se saa parametrikseen alku- ja päätepisteen, ja tekee koordinaateista muun muassa vektoreita helpottaakseen polkulaskelmia.

$VehicleGraphicsItem(vehicle)$ luo autosta $vehicle$ $VehicleGraphicsItem$- olion. Olion avulla voidaan piirtää autoja GUI-ikkunaan. Luokka siis piirtää kolmioita ja sijoittaa ne ikkunaan auto-olioiden mukaan. Se myös päivittää autojen sijaintia ja orientaatiota $Vehicle$- olion liikealgoritmien mukaan. $VehicleGraphicsItem$ perii $QtWidgets$:in $QGraphicsPolygonItem$:in.

GUI perii graafisen käyttöliittymän $QMainWindow$:n. Se on vastuussa prosessien kuvittamisesta: se lisää $QGraphicsItem$:it ikkunaan. Luokassa on myös $timerEvent$- mikä mahdollistaa animaation. Se pyytää luokkaa piirtämään uuden ikkunan tasaisin väliajoin.

$CarInfoFileError$ on ohjelman yleinen virheluokka. Jos tiedostossa on jokin vika (väärin kirjoitettu), ohjelma nostattaa $CarInfoFileError$:in.

Algoritmit

Vektorialgoritmit

Vektoriyhteenlaskun tarkoitus on laskea yhteen vektorien komponentit → tulos on vektori, jonka komponentit ovat yksittäisten komponenttien summat:

$$a + b = [a.x + b.x, a.y + b.y]$$

, jossa $a.x$, jne. ovat vektorien yksittäiset komponentit ja hakasuluilla kuvataan vektorinotaatiota. Eli siis käsky $a.add(b)$ lisää komponentteja yhteen. Sama idea toimii erotusalgoritmilla ($a.sub(b)$), mutta b:n komponentit ovat negatiivisia.

Vektorikerronnassa ($a.mult(c)$) kerrotaan kaikki vektorin algoritmit annetulla vakiolla:

$$c * a = [c * a.x, c * a.y]$$

Asettaessa vektorille jokin suuruus ($a.set\_magn(c)$), vektorista tehdään ensin yksikköpituinen – jaetaan pituudella – ja kerrotaan annetulla arvolla:

$$a_{magn=c} = [c * \frac{a.x}{|a|}, c * \frac{a.y}{|a|}]$$

Pistetulo ($a.dot\_product(b)$) on komponenttien tulojen summa:

$$a \cdot b = a.x * b.x + a.y * b.y$$

Vektorin kulma saadaan $math$:in metodilla $atan2(y, x)$ eli kahden luvun arcustangentilla. Tämä perustuu siihen, että vektorin [x,y] kulma x-akselin suhteen saadaan arctanilla, jossa lukuna annetaan vektorin y-komponentin pituuden ja x-komponentin pituuden suhde.

$$\theta = \tan^{-1}(\frac{y}{x})$$

Liikealgoritmit

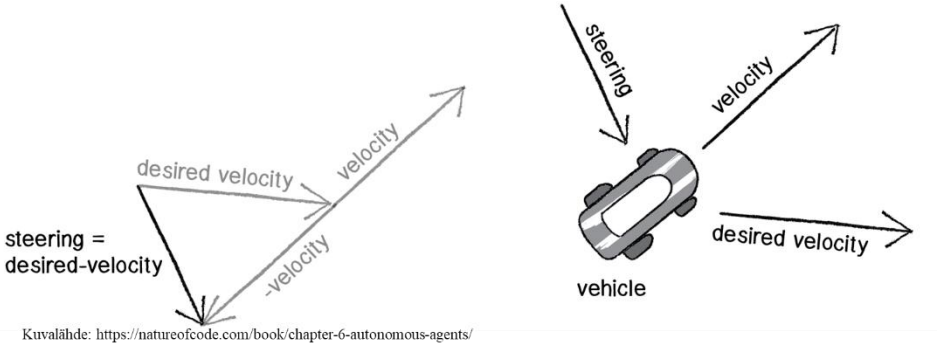
(Huom: annetut muuttujat ovat tyyppiä vektori)

Jokaiselle liikkeenmuutos algoritmille on sama periaate: autolle etsitään muutosvektori $steer$, mikä muuttaa nopeusvektoria, mikä taas lisätään auton sijaintiin. Tästä saatu vektori on auton uusi sijainti.

$$velocity_{new} = velocity + steer$$

$$location_{new} = location + velocty_{new}$$

Seek



Kuvalähde: https://natureofcode.com/book/chapter-6-autonomous-agents/

$Seek(point)$ pyrkii liikkumaan kiintopistettä $point$ kohti. Autolla on nopeutta (jolla on suunta) sekä haluttu suunta (vektori sijainnista kiintopisteeseen, $point - location$), ja kaikki auton ohjautuvuus ($steer$) käytetään ohjaamaan autoa tähän suuntaan. Seuraa siis yksinkertainen lasku:

$$steer = v_{desired} - v_{car}$$

, jossa saatua vektoria voidaan käyttää auton uuden sijainnin laskemiseen. Laskemisessa täytyy muistaa, että haluttu vauhti voi olla vain auton maksiminopeuden suuruinen, eli ennen laskua nopeudelle täytyy asettaa kattoarvo max$_speed$ ($v_{desired}.set\_limit(\max\_speed)$). Lopuksi täytyy tietenkin uudelleenasettaa auton suuntavektori, mikä on uuden nopeuden yksikkövektori.

Arrive

$Arrive(point)$ toimii muuten $seek$:in tavoin, mutta sen avulla auto pyrkii pysähtymään pisteeseen päästyään. Tämä tarkoitta sitä, että algoritmi etsii halutun pisteen ($desired$) ja ohjautuu sitä kohti ($steer$), mutta tällä kertaa auton etäisyyttä kiintopisteestä tarkkaillaan. Etäisyys lasketaan ottamalla vektori kiintopisteestä ja ottamalla sen erotus auton sijaintiin:

$$d = [target.x, target.y] - [location.x, location.y]$$

Laskemalla vektorin suuruus ($d.get\_magn()$) saadaan auton etäisyys kiintopisteestään. Jos tämä etäisyys on pienempi kuin jokin vertausluku (esim. 150), niin haluamme aloittaa hidastamisprosessin. Hidastamisalgoritmeja on monta (logaritminen, eksponentiaalinen), mutta tähän projektiin valittiin yksinkertainen lineaarinen hidastuminen, jossa auton nopeus on v kun etäisyys d on 150 , ja 0 kun d lähenee kohti nollaa:

$$y = ax + b$$

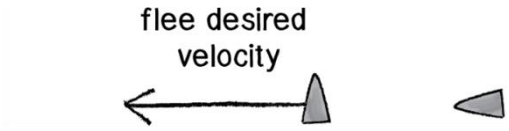
Ehtoja: $y(0) = 0, y(150) = v_{max}$. Tästä seuraa $b = 0$ ja että

$$a * (150) = v_{max} \rightarrow a = v_{max}/150$$

Eli yhtälö nopeudelle on $v = \frac{v_{max}}{150} * d$. Tätä nopeutta v käytetään auton uuden sijainnin laskemiseen kuten ylhäällä näytetty.

Escape

Jos auton on törmätä toiseen autoon, niin täytyy auton tehdä väistöliikkeitä. Tämä selvitetään kaksiosaisesti: a) selvitetään ovatko jotkin autot ”vaara-alueella” b) muuntamalla liikettä tarpeen mukaan. Vaaratilanteiden selvittäminen lasketaan vertailemalla auton ennustepistettä muihin autoihin. Auton ennustepiste on yksinkertaisesti piste, joka on auton sijainti siirrettynä jonkin pikselimäärän (esim. 20) auton nopeuden suuntaan:



Kuvalähde: https://natureofcode.com/book/chapter-6-autonomous-agents/

$vec_{prediction} = v.set\_magn(20)$, jossa $vec_{prediction}$ = auton skaalattu nopeusvektori.

$$\therefore location_{prediction} = location + vec_{prediction}$$

Tätä ennustepistettä sitten verrataan muiden autojen sijainteihin metodilla $check\_danger()$, se käy läpi $VehicleWorld$:in kaikki autot ja laskee niiden autojen etäisyyden vertailtavan auton ennustepisteeseen ($d = (location_{other} - location_{car}).get\_magn()$), ja jos etäisyys on pienempi kuin tietty tunnusluku (esim. 50), niin auto on vaarassa törmätä, ja toisen auton koordinaatit lisätään ”vaarakoordinaatit”-listaan.

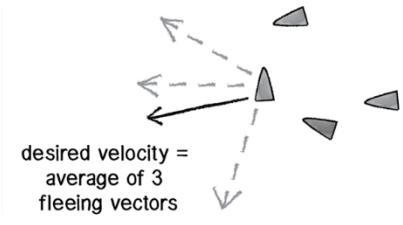
Yhtä autoa on helppo paeta: otetaan vektori vaarapisteestä poispäin ja seurataan sitä. Tämä toteutetaan ottamalla vektori vaarapisteestä autoon, skaalaamalla sitä 50 kokoiseksi ja menemällä sitä kohti ($seek$):

$$v_{escape} = [location_{danger} - location_{car}].set\_magn(50)$$

$$location_{escape} = location + v_{escape}$$

$$vehicle.seek(location_{escape})$$

Jos lähellä olevia autoja on enemmän kuin yksi, niin paras pakenemissuunta on vaarakoordinaattien keskiarvosta poispäin. Tällöin vaarakoordinaateista otetaan vain keksiarvo (summa/määrä) ja tätä pisteettä käytetään vaarakoordinaattina ($location_{danger}$) pakenemiseen.



Kuvalähde: https://natureofcode.com/book/chapter-6-autonomous-agents/

Tietorakenteet

Projektissa Pythonin sisäänrakennettuja tietorakenteita käytettiin yhdessä oliomuotoisten tietorakenteiden kanssa. Koska jokaisella autolla oli niin monta erilaista ominaisuutta, niin yksittäisen auton tietoja oli helpompi seurata luomalla siitä olio, joka ylläpitää tietyn autot ainutlaatuiset ominaisuudet.

Toinen tärkeä tietorakenne on vektori-oliot. Koska Pythonissa ei ollut sisäänrakennettua vektorilaskentakirjastoa (tai sen löytämiseen ei perehdytty), niin sellainen oli hyvä luoda. Vektorilaskenta helpotti autojen sijainti- ja liikelaskelmia. Toinen samanlainen luokka $Path$, mikä piti tietoa auton kuljettavasta polusta.

Muuten olioiden tukena on käytetty vain yksinkertaisia yksi- tai kaksiulotteisia listoja. Näitä käytettiin tallentamaan tietoja esimerkiksi koordinaateista tai autoille arvottavista ominaisuuksista. Ehkäpä ohjelmalle tärkein lista oli $VehicleWorld$:issa sijaitseva lista, mikä piti sisässään kaikki maailmaan siihen mennessä lisätyt auto-oliot.

Vaihtoehtoisia toteutuksia tälle olisi ollut. $Path$- luokkaa ei olisi tarvinnut luoda ollenkaan, vaan polun tiedot olisi voitu tallentaa suoraan auton ominaisuuksiin. Luokka haluttiin kuitenkin pitää, koska se selkeytti auton liikelaskentatoimia. Vastaavasti myös listoista olisi voitu luopua kokonaan ja käyttää jotain vastaavaa tietorakennetta (esimerkiksi puhelinluettelo). Listoihin kuitenkin päädyttiin niiden yksinkertaisuuden takia. Myös $VehicleWorld$:in olisi voinut jättää tekemättä ja autot-lista olisi voitu tallentaa jonnekin muualle (esimerkiksi $main$:iin), mutta tämä olisi vaikeuttanut listaan käsiksi pääsyä muissa moduuleissa.

Tiedostot

Ohjelman mukana tulee ylempänä alustettu csv-tiedosto *car_info.csv*.

Testaus

Suurin osa ohjelman testeistä tapahtuu tiedoston lukuvaiheessa, koska itse simulaatiossa ei ole muuta kuin laskutoimituksia ja ne voidaan olettaa toimivan moitteetta. Tiedoston lukuvaiheessa on luotu itse ohjelmaan kattava testaus tilanteille, jossa ohjelmalla on annettu vääränlainen tiedosto. Virhetilanteita oli monia:

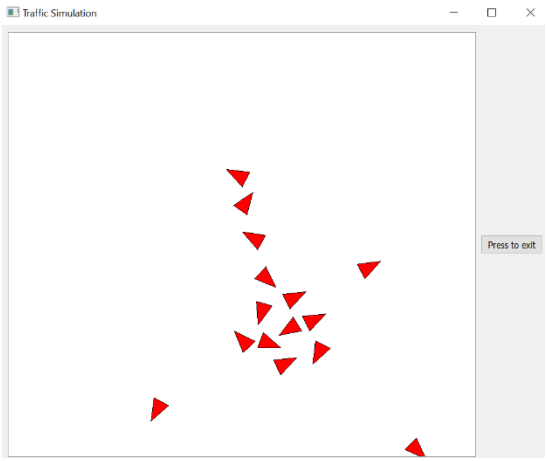
- Kaikkia ominaisuuksia ei annettu
- Annetut luvut olivat liian pieniä tai liian suuria
- Annetut ylä- ja alarajat olivat väärin päin
- Ominaisuuksiin ei annettu lukuja vaan kirjaimia tms.

Itse tiedoston lukemiseen käytettiin *try-except* – mallia jossa *exception*:inä toimi $ValueError$ (tämä sen takia, että jos ominaisuuksiin oli annettu jotain muuta kuin lukuja, niin siinä vaiheessa kun ohjelma haluaa lukea $float$-tyyppisiä lukuja listaan, niin aiheutuu $ValueError$.

Lukujen väärinantamista varten luotiin oma testiluokka $CarInfoFileError$, mikä nostettiin ohjelmassa aina, jos jokin tiedoston ominaisuuksista oli väärin. Tämä oli hyvä yleisluokka virheille, ja antoi tietoa käyttäjälle virheen tyypistä.

Yksikkötestausta varten luotiin testimoduuli $test.py$, jossa oli testiluokat $Vector$:in, $Path$:in ja $Main$:in testausta varten. $Vector$:issa piti testata, että kaikki laskualgoritmit toimivat, eli niihin käytettiin yksinkertaista $assertEqual$- testiä. $Path$:issa testattiin toiminta samoin metodein. $Main$:in testauksessa keskityttiin virhetilanteisiin, eli testattiin, nostattaako ohjelma Error-tilan, jos ohjelmalle antaa parametriksi väärän tiedoston. Tähän käytettiin siis $assertRaises$- testiä.

Ohjelman testikattavuus oli mielestäni hyvä, mutta yksikkötestauksessa jäi puutteita. $Vehiclen$ liikelaskujen sekä graafisen käyttöliittymän yksikkötestit jäivät puuttumaan kokonaan.



Ohjelman puutteet ja viat

Väistö

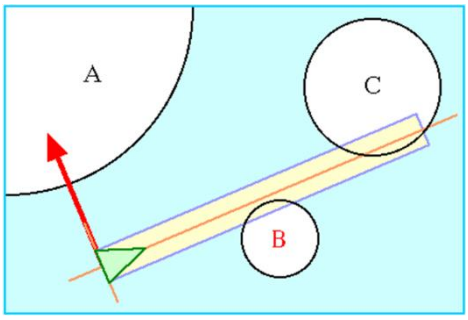
Ohjelman suurin puute on kenties väistöalgoritmissa. Jos autoja on monia ja menevät samojen pisteiden kautta, niin autoista muodostuu ikään kuin ”kasa” josta autot eivät pääse ulos, vaan jäävät pyörimään paikoilleen kassaan pitkäksikin aikaa. Voi kestää (kymmeniä) sekunteja, että iso määrä kasaantuneita autoja saadaan eroteltua toisistaan.

Uskoisin, että tässä vika on liikealgoritmissani. Ohjelmassani on ehtorakenne, joka määrää käytettävän liikealgoritmin seuraavasti:

```
if dis.get_magn() <= 150:
    self.arrive()
else:
    coors = self.check_danger()
    if len(coors) > 0:
        self.escape(coors)
    else:
        self.seek([self.target[0], self.target[1]])
```

Eli jos auto on lähellä päätepistettä, hidasta. Jos on vaarassa, väistä. Muuten kulje kohti päätepistettä. Tämä tarkoittaa sitä että auto haluaa melkein aina kulkea suoraa tietä alusta loppuun. Väistötilanteessa (kun ennustepiste on liian lähellä muita) auto kääntyy päinvastaiseen suuntaan, mutta liikkuessaan pienenkin matkan takaisin päin auto poistuu vaara-alueelta, ja tällöin aktivoituu $seek(päätepiste)$ algoritmi, kääntäen auton suunnan takaisin kohti kasaantumista. Jos auton polulla on siis este, niin auto haluaa silti aina kulkea sen läpi, ja ainoa asia mikä sitä estää, on $escape$- algoritmi. Auto jää siis logiikkaloukkuun: Juuri ja juuri vaara-alueella → väistä → kulje muutama pikseli poispäin → turvassa, kulje kohti päätepistettä! → käännös takaisin estettä kohti. Tämä näyttää käyttäjälle siltä, että autot pyörivät paikoillaan.

Tämän olisi voinut korjata luomalla Craig Reynoldsin esittelemä *Obstacle avoidance*- algoritmi, mikä tunnistaisi esteet (tässä tapauksessa pysähdyksissä olevat/hitaasti liikkuvat autot) ja väistäisi niitä tarvittaessa.



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

Toinen (helpompi) vaihtoehto olisi ollut virittää auto pidemmäksi aikaa väistötilaan: Kun auto kerran menee vaara-alueelle ja on osua muihin autoihin, niin tämä pysyy *escape*- tilassa niin kauan, kunnes sen sijainti on paljon pidemmällä (+100 pikseliä) kuin vaara-alueen raja-arvo. Tällöin vaaratilanteissa olevat autot saisivat enemmän aikaa ja tilaa välilleen, ja kenties voisivat päästä lomittain toistensa välistä. Tämä olisi ollut paljon helpompaa, sillä sen olisi voinut aiheuttaa yksinkertaisesti asettamalla jokin vaara-tunnusluku autolle ja tarkistaa ehtojen kautta onko tämä tunnusluku päällä. Jos on, niin *escape*- algoritmia ylläpidetään.

Tätä ei oltu kuitenkaan testattu, ja on hyvin mahdollista, ettei tämä toimi. Voi olla, että kasa vain suurenee ja pienenee jojoillen, kun autot menevät kauemmas ja taas toisiaan kohti.

Kuormitus

Toinen suuri puute, mikä tässä ohjelmassa oli, on sen suoritustehokkuus. Ohjelma on rakennettu siten, että siinä on paljon sisäkkäisiä kuormittavia osia (for-loopit, listan tarkistukset) sekä paljon yksittäisiä yksityiskohtia, mitä *yksittäisestä* autosta pitää tarkistaa ja ylläpitää. Tämä tarkoittaa sitä, että yksinkertaisen liikkeen (saatikka yksinkertaisen 'framen') eteen on tehty järjetön määrä prosessointia, tarkoittaen sitä, että jos autoja on monta, niin laskentakuormitus kasvaa nopeasti simulointianimaation ajokyvyn yli, ja simulaatio tökkii. Tämän (sekä kasaantumisongelman) takia on jouduttu rajoittamaan autojen maksimimäärää viiteentoista.

Tähän minulla ei oikeastaan ole kunnon ratkaisua, enkä keksinyt omien tietojeni tai ohjelmointitaustani perusteella valita tehokkaampaa suoritusmenetelmää, johon minulla olisi riittänyt taidot sekä kyvyt.

3 parasta sekä kolme heikointa kohtaa

Kolme parasta kohtaa ovat varmaankin *arrive*-algoritmi ja väistöliikeet pienellä automäärällä. Ne toimivat graafisessa ikkunassa tehokkaasti ja näyttävät erittäin mielekkäältä. Viimeiseksi pidän myös siitä, miten kaikkien autojen ominaisuudet on arvottu. Tämä lisää vaihtelua eri autojen välille, luoden simulaatiosta mielenkiintoisemman, kun kaikilla autoilla on omanlaisensa leimat. Jotkin autot liikkuvat nopeasti päätepisteeseensä, jotkin hitaammin, jotkut ovat hyviä mutkittelemaan. Tämä lisää mielestäni simulaation arvoa.

Kolme heikointa kohtaa ovat jo puutteissa mainittujen piirteiden lisäksi se tapaa, millä tiedostosta luetaan autojen tietoja. Tapa ei ole millään tapaa kovin elegantti, ja tiedoston kirjoittaminen ja muokkaaminen on kömpelöä ja käyttäjälle hämmentävää. Sen ainoa hyvä puoli on helppolukuisuus (csv-tiedostoja on nopea lukea), mutta tekee *main*-tiedostosta epätehokkaan

näköisen. Tämän olisi voinut kenties korjata *HumanWriteableIO* (kierros 5) – tyylisellä tiedostolla, mutta se olisi lisännyt omat haasteensa tiedoston lukuun, ja koska ohjelman painopiste ei ole kuitenkaan tiedoston luku vaan autojen simulointi, päädyttiin tämänlaiseen ratkaisuun. Graafiseen käyttöliittymään olisi voinut myös lisätä lomake, jolla käyttäjä itse syöttää autojen tiedot, mutta tämä olisi myös huomattavasti lisännyt projektin haastavuutta antamatta merkittävästi simulaatiolle lisäarvoa.

Poikkeamat suunnitelmasta

Projekti eteni muuten täysin suunnitelmien mukaan lukuun ottamatta $path\ following$ – algoritmin pois jättämistä (algoritmi ei toiminut eikä se parantanut simulaation laatua). Ajankäytöllisesti aikaa meni tietysti enemmän kuin suunnittelin, josta huonoiten suunnitteli oman ajankäytön (nimimerkillä kirjoitan tätä 4h. ennen dedistä…). Toteutus järjestys oli myös selkeä ja toimi hyvin, eli tein ensin kaikki yksittäiset autoanimaatiot ja aloin myöhemmin yhdistelemään niitä kokonaisuudeksi.

Toteutunut työjärjestys ja aikataulu

1. Graafisen käyttöliittymän, main-moduulin luonti (5.-6.3)
    a. Yksinkertainen ikkuna jota ajetaan mainissa
2. Loput luokat: vector, vehicle, jne. (11.3-16.3)
3. Simulaatio, csv-tiedoston tekeminen (19.3- 3.4)
    a. Autojen liikkuminen, toiminta standardi-inputilla
4. Korjaukset, päätöksenteko (18.4-19.4)
    a. Vaihtelu seekin, escapein, arriven välillä.
5. Yksikkötestit, viimeiset viilaukset (20.4 – 24.4)

Arvio lopputuloksesta

Mielestäni ohjelma on yleishyvä tulkinta Greg Reynoldsin itsenäisistä kappaleista. Se toimii lähes kaikella annetulla inputilla, animointi ei ole tökkivää tai muutenkaan epäselvää, ja graafinen käyttöliittymä on yksinkertainen sekä selkeä. Ohjelmassa ei ole suuria puutteita joita ei voisi helposti korjata.

Luokkajako toimi mielestäni erinomaisesti, ja esimerkiksi $vehicle$- luokkaan on helppo lisätä parannuksia/implementointeja – luokassa on valmiiksi implementoitu metodi eri käyttäytymistapojen vaihteluun.

Ohjelma olisi voinut olla tehokkaampi, koodi selkeämpää, sekä väistöalgoritmit korjata. Muuten koodi on erittäin toimivaa!

Viitteet

- ”6. Autonomous Agents”, *The Nature of Code*, Daniel Shiffman, https://natureofcode.com/book/chapter-6-autonomous-agents/
- *Steering Behaviors For Autonomous Characters*, Greg Reynolds, www.red3d.com/cwr/steer/gdc99/
- Pythonin standardimanuaali, https://docs.python.org/3/reference/index.html#the-python-language-reference
- Kurssimateriaalit

Liitteet