Ohjelmointistudio 2: Parvisimulaattori

Projektidokumentti

18.4.2018

Henkilötiedot

Nimi: Välimäki, Tuomas Heikki
Opiskelijanumero: 649 649
Koulutusohjelma: Tik
Vuosikurssi: 1.

Yleiskuvaus

Laadittiin sovellus, joka simuloi kaksiulotteisessa avaruudessa liikkuvien yksinkertaista parviälyä noudattavien kappaleiden (joita jatkossa kutsutaan kaloiksi) liikettä ja visualisoi kappaleiden liikettä reaaliaikaisesti animaation avulla. Sovellus sisältää graafisen käyttöliittymän jonka avulla simulaation etenemistä voi seurata ja samaan aikaan reaaliaikaisesti vaikuttaa tiettyihin simulaation parametreihin.

Simulaatiossa jokainen yksittäinen kala liikkuu autonomisesti. Yksittäinen kala pystyy havainnoimaan tietyn säteen sisällä olevat muut kalat, jolloin parviäly muodostuu seuraavasta kolmesta eri säännöstä, jota jokainen kala itsenäisesti noudattaa:

- Alignment: kala korjaa liikesuuntaansa siten että kalan liikesuunta olisi sama kuin ympärillä olevien kalojen liikesuunta, jotta parven jäsenet liikkuisivat samaan suuntaan.
- Cohesion: kala pyrkii kohti lähistöllä olevien muiden kalojen massakeskipistettä.
- Separation: kala pyrkii poispäin lähellä olevista kaloista.

Edellisten sääntöjen keskinäisten vaikutussuhteiden voimakkuutta voidaan säätää reaaliaikaisesti graafisesta käyttöliittymästä käsin.

Sovelluksen käynnistyessä simulaatio on ”tyhjä” ja siihen voidaan reaaliaikaisesti lisätä kaloja tai voidaan simulaation edetä lisätä esteitä, joihin törmäämistä kalat keinoin pyrkivät väistelemään. Esteet ovat yksinkertaisuuden vuoksi vakiosäteisiä ympyröitä.

Sovelluksessa voidaan tallentaa esteiden sijainnit. Sovelluksessa on vain yksi ”tallennuspaikka”, vakioniminen tekstitiedosto, joka ylikirjoitetaan tallennettaessa. Tiedoston korruptoituminen tai puuttuminen antaa virheilmoituksen yritettäessä ladata.

Projekti oli luokiteltu asteikolle keskivaikea-vaativa, ja mielestäni toteutukseni kallistuu vaativan puolelle, koska annettuihin ohjeisiin nähden lisäsin esteet ja tallennus/latausominaisuuden. Mainittakoon että parviälyn toteutus oli suhteellisen helppoa, esteiden välttely ja niihin törmäämisen ehkäiseminen oli huomattavasti hankalampaa siten, että liikkeestä sai jotenkin luonnollisen näköistä.

Käyttöohje

Sovellus käynnistetään suorittamalla *boids.ui* packagen ***boidsUI.scala***-tiedosto. Sovellus käynnistyy vakiokokoiseen ikkunaan, jonka kokoa ei voi säätää.

Kuvassa 1 on kuvakaappaus sovelluksesta. Seuraavassa on kuvattu käyttöliittymän eri toiminnot:

Randomize fish -painike: Lisää satunnaisiin koordinaatteihin 20 kalaa, joilla on ennalta määrätty nopeus ja sattumanvarainen liikesuunta. Kaloilla on maksimilukumäärä 500 kpl.

Clear fish -painike: Poistaa kaikki kalat.

Clear Walls -painike: Poistaa kaikki esteet.

Circles ON/OFF -painike: Simulaatiossa on määritelty säde, jonka sisällä olevat muut kalat yksittäinen kala havaitsee. Painiketta painamalla voidaan kytkeä päälle/pois se, piirtääkö simulaatio jokaisen ympärille sädettä vastaavan ympyrän. Oletusarvoisesti eli sovelluksen käynnistyessä pois päältä.

Separation, Alignment ja Cohesion-liukusäätimet: Säätää simulaation parametreja. Liukusäätimien vaikutus voidaan havaita muutoksena kalojen käyttäytymisesä.

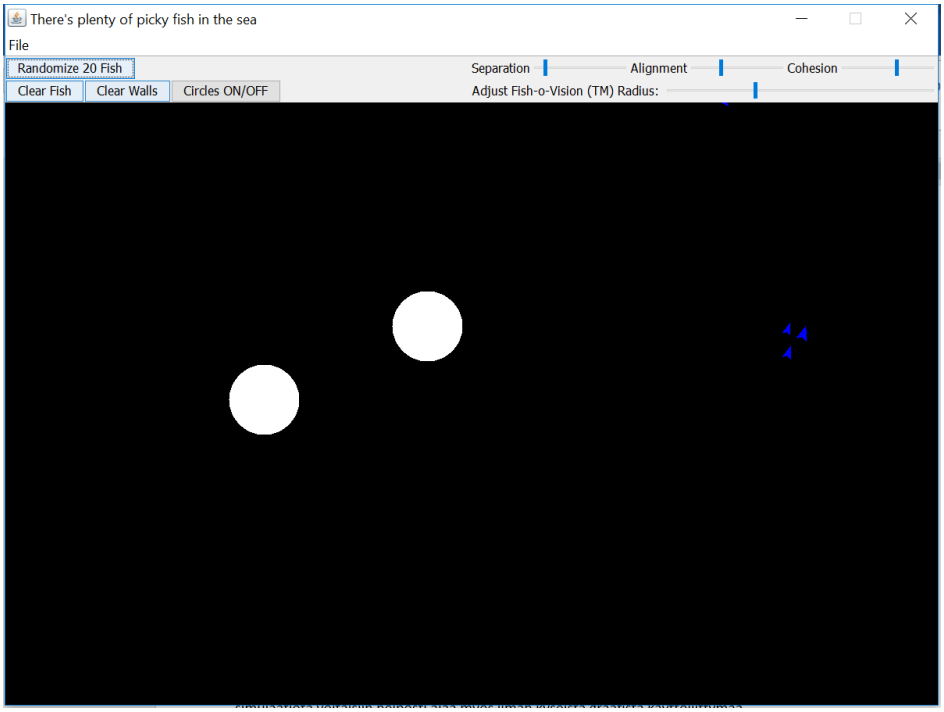
Adjust Fish-o-Vision(TM) Radius -liukusäädin: Säätää sädettä, jonka sisällä olevat muut kalat yksittäinen kala havaitsee. Säätimen vaikutus voidaan visuaalisesti havaita, jos **CIRCLES ON/OFF** -painikkeesta on kytketty sädettä vastaavien ympyröiden piirto päälle.

File-valikko:

- **Load**: Lataa tallennettut esteet.
- **Save**: Tallentaa esteet, eli niiden koordinaatit. Tallennuspaikkoja on vain yksi, joka ylikirjoitetaan aina talletettaessa.

Hiiren vasemmalla painikkeella voidaan lisätä yksittäisiä kaloja simulaatioon tiettyyn sijaintiin. Tällöin kalalle arvotaan satunnainen liikesuunta ja lisäksi massa. Siinä missä **randomize 20 fish** -painike lisää 20 vakiokokoista kalaa, yksittäisiä kaloja lisättäessä erimassaisia kaloja (kaloilla on myös graafisesti eri koko).

Hiiren oikealla painikkeella voidaan lisätä yksittäisiä esteitä. Esteet voivat mennä päällekkäin ja niiden lukumäärää ei ole rajoitettu.

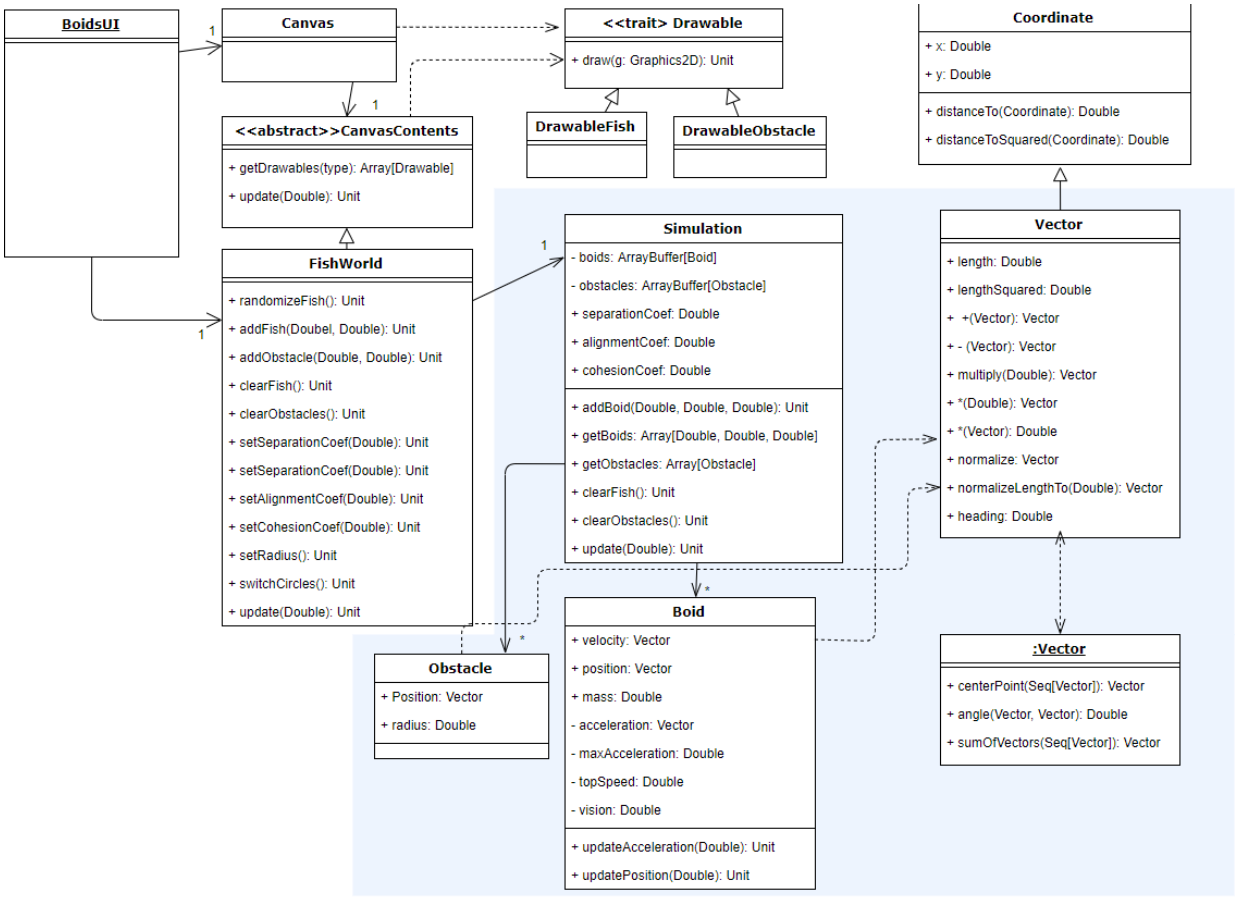


Kuva 1. Kuvakaappaus sovelluksesta.

Sovelluksen rakenne

Rakennekaavio

Sovelluksen rakenne on esitetty kuvassa 2. Kaaviosta on jätetty pois lataamiseen ja tallennukseen liittyvä luokka sekä jotkut yksinkertaiset kumppanioliot.



Kuva 2. Sovelluksen rakenne UML-kaaviona.

Sovelluksen toiminnan ja rakenteen kuvailu

Yleiskuvaus:

Koska kysymyksessä on simulaatio graafisen käyttöliittymän kera, suunnitelman lähtökohtana on ollut pitää graafinen käyttöliittymä ja varsinainen simulaatio toisistaan erillään. Tämä mahdollistaa sen, että simulaatiota voitaisiin helposti ajaa myös ilman kyseistä graafista käyttöliittymää.

Kuvassa 1 on vaaleansinisellä alueella olevat luokat liittyvät simulaatioon, valkoisella alueella olevat luokat graafiseen käyttöliittymään. Esimerkiksi siis simulaatiolla ei ole minkäänlaista kytkentää simulaation kappaleiden graafiseen ulkoasuun.

FishWorld-luokka on eräänlainen adapteriluokka, jolla graafinen käyttöliittymä ja simulaatio saadaan toimimaan yhteen.

Graafiikan piirtämisen ja simulaation päivityksen yleinen kuvailu:

Canvas on graafisen käyttöliittymän *Panel*-luokasta periytetty luokka, jonka tehdasmetodi *repaint*-hoitaa varsinaisen ruudunpäivityksen. Tätä metodia kutsutaan kymmenen millisekunnin välein.

Kun *Canvas*-luokasta tehdään instanssi, sille annetaan parametriksi viittaus abstraktiin luokkaan *CanvasContents*, jolla on määritelty metodi getDrawables. Kun Canvas luokan tehdasmetodia *repaint* kutsutaan, se vastaavasti kutsuu CanvasContents-luokan metodia getDrawables, joka saa listan piirrettävistä Drawables luokasta periytetyistä objekteista (joita sovelluksessa on kaksi: *DrawableFish* ja *DrawableObstacle*). Jokaisella tämän luokan objekteilla on metodi *draw*, joka loppujen lopuksi piirtää yksittäisen kappaleen.

FishWorld on *CanvasContents*-luokan toteuttava luokka, josta luotu instanssi varsinaisesti annetaan parametriksi *Canvas*-luokasta tehdylle instanssille, eli ”piirtoalustalle”. *FishWorld* on adapteri graafisen käyttöliittymän ja graafisen käyttöliittymän välillä. *FishWorld*-luokka pitää sisällään varsinaisen simulaatio-objektin (*Simulation*-luokka) ja kaikki kommunikaatio simulaation ja graafisen käyttöliittymän välillä tapahtuu tämän luokan välityksellä.

Kun graafinen käyttöliittymä käynnistetään, käynnistyy samalla säie, joka kutsuu silmukassa *FishWorld*-luokan instanssin *update*-metodia, joka vastaavasti kutsuu simulaation *update*-metodia. Simulaatio toimii siis graafisesta käyttöliittymästä irrallaan. Graafinen käyttöliittymä *FishWorld*-luokan välityksellä pyytää säännöllisen väliajoin listan piirrettävistä objekteista.

Samaten myös, jos esimerkiksi graafisesta käyttöliittymässä säädetään simulaation parametreja, tämä pyyntö välittyy simulaatiolle *FishWorld*-luokan välityksellä.

Keskeiset luokat ja metodit:

FishWorld-luokka

```
randomize(): Unit
```

Luo 20 satunnaisiin suuntiin liikkuvaa kalaa satunnaisiin koordinaatteihin ja lisää ne simulaatioon.

```
update(dt: Double): Unit
```

Välittää simulaatiolle päivityspyynnön.

```
getDrawables: Array[Drawable]
```

Saa simulaatiolta listan piirrettävistä *Drawable*-piirreluokan toteuttavista objekteista (kaloista ja esteistä). Kutsutaan ruudunpäivityksen yhteydessä.

```
getObstacleCoordinates(): Array[(Int, Int)]
```

Pyytää simulaatiolta listan esteiden koodinaateista tiedostoon tallennusta varten.

```
setObstacles(coords: Array[(Int, Int)])
```

Tiedoston latauksen yhteydessä välittää simulaatiolle tiedon ladattujen esteiden koordinaateista ja lisää ne simulaatioon.

```
addFish(x: Double, y: Double): Unit
```

Lisää simulaatioon uuden kalan. Kutsutaan graafisestä käyttöliittymästä käsin, kun hiirellä lisätään yksittäisiä kaloja.

```
addObstacle(x: Double, y: Double): Unit
```

Lisää simulaatioon uuden esteen. Kutsutaan graafisesta käyttöliittymästä kun hiirellä lisätään uusi este.

Simulation-luokka

```
update(dt: Double): Unit
```

Simulation luokan keskeinen metodi, joka ottaa parametrikseen ajan, joka on kulunut edellisestä päivityksestä. *Simulation*-luokka pitää sisällään taulukon kaloista (Boid-luokan objekteista) ja esteistä (Obstacle-luokan objekteista). Metodi kutsuu ensin jokaisen kalan updateAcceleration-metodia ja sitten jokaisen kalan updatePosition-metodia.

```
getBoids: Array[(Double, Double, Double, Double)]
```

Metodi, jota kutsumalla simulaatiolta saa taulukon kaloista, niiden sijanneista sekä suunnasta. Metodin palauttaman taulukon alkio on Tuple, jonka arvot ovat (x-koordinaatti, y-koordinaatti, kulma/suunta, massa)

```
getObstacles: Array[(Double, Double, Double)]
```

Metodi, jota kutsumalla saa taulukon esteistä. Metodin palauttaman taulukon alkio on Tuple, jonka arvot ovat (x-koordinaatti, y-koordinaatti, säde)

Boid-luokka

```
updateAcceleration(dt: Double, otherBoids: Array[Boid], obstacles: Array[Obstacle],S: Double, C: Double, A: Double, radius: Double): Unit
```

Metodi muodostaa koko simulaation ytimen eli yksittäisen kalan älyn. Metodi saa parametrikseen aikavälin eli ajan joka on kulunut edellisestä päivityksestä (dt), listan kaloista (otherBoids), listan esteistä (obstacles), sekä listan simulaation parametreista. Nimestään huolimatta laskee ensin kalalle uuden kiihtyvyyden ja myös päivittää nopeuden.

```
updatePosition(dt: Double): Unit
```

Ottaa parametrikseen aikavälin ja päivittää jokaisen kalan uuden sijainnin edellisen metodin avulla laskettujen nopeuksien perusteella.

Boid-luokan kumppaniolio

```
alignmentSteerVector(boid: Boid, otherBoids: Array[Boid]): Vector
```

Laskee kalalle voimavektorin (jonka avulla saadaan kalan kiihtyvyys), joka saa aikaan parviälyn alignment-ominaisuuden (tarkempi kuvaus myöhemmin).

```
cohesionSteerVector(boid: Boid, otherBoids: Array[Boid]): Vector
```

Laskee kalalle voimavektorin, joka saa aikaan parviälyn cohesion-ominaisuuden.

```
separationSteerVector(boid: Boid, otherBoids: Array[Boid]): Vector
```

Laskee kalalle voimavektorin, joka saa aikaan parviälyn separation-ominaisuuden.

```
steerAwayFromObstacles(boid: Boid, obstacles: Array[Obstacle]): (Vector, Boolean)
```

Laskee kalalle voimavektorin, joka saa aikaan sen, että kala korjaa suuntaansa, jos kalan liikerata leikkaa esteen edessäpäin, tietyn etäisyyden päässä.

```
obstacleCollisionAvoidanceVector(boid: Boid, obstacles: Array[Obstacle]): (Vector, Boolean)
```

Estää viimekädessä törmäyksen esteisiin, eli laskee voimavektorin, joka aiheuttaa kalassa ison kiihtyvyyden jolloin kala korjaa äkillisesti liikettään.

SaveLoad-objekti

```
save(fileName: String, data: Array[(Int,Int)]): Unit
```

Tallentaa esteiden koordinaatit tiedostoon.

```
load(filename: String): Array[(Int, Int)]
```

Lataa esteiden koordinaatit tiedostosta ja palauttaa koordinaatit taulukkona.

```
parser(input: BufferedReader): Array[(Int, Int)]
```

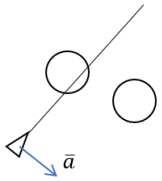
Parseroi tiedostonluvun yhteydessä luettavan data ja palauttaa esteiden koordinaatit taulukkona.

Parviälyn toteutus

Koko sovelluksen ytimen eli simulaation toteutuksessa oleellisinta parviälyn toteuttaminen on se miten yksittäisen kalan (eli *Boid*-luokan objektin) kiihtyvyys lasketaan. Kuten edellä kuvattiin, tämä oleellisesti tapahtuu, kun kutsutaan Boid-luokan updateAcceleration-metodia.

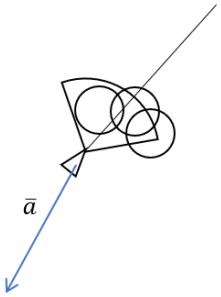
Tässä kuvataan metodin toiminta pääpiirteissään. Tarkempi kuvaus algebrasta löytyy liitteestä 1.

1. Metodi saa parametrikseen kaikki kalat taulukkona (itsensä mukaan lukien)
2. Taulukosta suodatetaan vain ne kalat jotka ovat tietyn säteen sisällä (samalla taulukosta suodatetaan pois myös kala itse ts. kaikki kalat joihin etäisyys on 0)
3. Kala ”näkee suoraan eteenpäin” tietyn pituisen matkan (ks. kuva alla). Lasketaan, leikkaako kalan liikerata tiettyyn etäisyyteen saakka esteen. Jos näin on, lasketaan kuvan esittämän suunnan mukainen kiihtyvyys (joka on kohtisuoraan liikesuuntaan nähden) ja asetetaan *avoidanceMode* päälle.



4. Edellinen ei kuitenkaan estä välttämättä esteisiin törmäämistä tilanteessa jossa esteitä on rykelmänä edessä. Kalalla onkin kapean sektorin muotoinen lyhyenkantaman näkökenttä. Mikäli tässä näkökentässä on esteitä kala muuttaa äkillisesti suuntaansa. Kalalle lasketaan hyvin iso

kiihtyvyysvektori, joka kuvan mukaisesti suuntautuu esteistä poispäin (tämän parametrien tuunaamisessa meni tovi, että käytös näyttäisi jokseenkin järkevältä). Jos esteitä oli tässä näkökentässä, asetetaan *avoidanceMode* päälle.



5. Mikäli *avoidanceMode* ei ole päällä, kala käyttäytyy normaalin parviälyn mukaan, eli kalalle lasketaan alignment, separation ja cohesion -kiihtyvyysvektorit lähistöllä olevien kalojen perusteella, joista lasketaan summa, joka asetetaan kalan kiihtyvyydeksi.
6. Lopullinen kiihtyvyys skaalataan kalan massalla ja lasketaan kalalle uusi nopeus.

Tietorakenteet

Sovelluksen toteuttamisessa ei tarvita kovin ihmeellisiä tietorakenteita. Pääasiassa sovelluksessa käytetään tietorakenteena muuttuvatilaista ArrayBuffer-kokoelmaa, simulaatiossa olevien ”kalojen” ja ”esteiden” tallettamiseksi.

Tiedostot

Sovelluksessa on mahdollisuus tallentaa tiedostoon (”data.txt”) esteiden sijainnit ja ladata ne. Tiedoston rakenne on hyvin yksinkertainen: Jokaisen esteen x- ja y-koordinaatti puolipisteellä erotettuna omalla rivillään, esim.

```
344;188
572;319
709;356
```

Sovelluksen selviää kunnialla tiedoston lukemisesta vaikka tiedostossa olisi ylimääräisiä välilyöntejä tai rivivaihtoja. Jos tiedosto on lukukelvoton (vääränlaisessa formaatissa) sovellus herjaa siitä alert-ikkunalla.

Testaus

Suunnitelmassa esitin että testaus tulee keskittymään Vektori-luokan testeihin, jotta voidaan parviälyä laatiessa luottaa siihen että vektorialgebran eri operaatiot tuottavat oikeellisia tuloksia.

Lopullisessa versiossa lukumäärällisesti suurin osa testeistä testaakin Vektori-luokan metodeja, jotka ovat hyvin suoraviivaisia.

Simulaation testaaminen tuotti aluksi hankaluuksia, koska yrityksestä huolimatta en onnistunut saamaan Boid-luokan privaattimetodin testejä toimimaan tähän tarkoitetuilla työkaluilla. Tein rakenteellisen muutoksen sovellukseen siten että siirsin suuren osan privaattimetodeista Boid-luokan kumppaniolion metodeiksi, jotta pääsin niihin testeistä käsiksi.

Simulaatiota on hyvin vaikea testata, mutta kalan kiihtyvyysvektoreita laskevia Boid-luokan kumppaniolion metodeita pystyi testaamaan tietyllä tasolla, esimerkiksi onko metodin tuottama vektori kohtisuorassa kalan nopeusvektoria vastaan. Jos rehellisiä ollaan nämä testit tuli kirjoitettua urheilun vuoksi vasta jälkikäteen.

Vektoriluokkaa rakentaessani käytin testejä varmistuakseni että vektoriluokka toimii oikein, mutta huomasin että tein testeistä liian puutteellisia (testit esimerksi testaavat summaamista vain yhdellä lukuparilla), jolloin vektoriluokkaan oli kuin olikin eksynyt virhe. Tämä luonnollisesti aiheutti aika paljon päänvaivaa kun simulaatio toimi aivan virheellisesti ja etsin vikaa Boid-luokasta vaikka vika olikin siinä, että Vektori-luokan eräs metodi (en enää muista mikä) toimi virheellisesti.

Ehkäpä yksikkötestauksessa järkevin ja onnistunein, josta oli merkittävästi apua, oli (hölmösti nimetyn) SaveLoad-objektin testaaminen (IOTests.scala) yksikkötestien avulla. Näiden testien avulla testattiin seuraavaa:

- Luetaanko oikeassa formaatissa oleva tiedosto oikein.
- Luetaanko tiedosto oikein ylimääräisten välilyöntien tapauksessa.
- Luetaanko tiedosto oikein ylimääräisten rivivaihtojen tapauksessa.
- Heitetäänkö oikeanlainen poikkeus, jos väärässä formaatissa olevaa tiedostoa yritetään lukea.
- Heitetäänkö oikeanlainen poikkeus, jos tiedostoa ei ole olemassa.

Tunnetut viat ja puutteet

- Alun perin tallennettujen esteiden latauksessa kaikkia kaloja ei tyhjennetty. Tämä on viime hetken purkkaviritelmä, sillä joskus harvoin sattui niin että jos este latautui (ilmeisesi) täsmälleen kalan päälle sovellus kaatui.
- En ole oikein varma mitä tapahtuu, kun manuaalisesti esteen lisää kalan päälle ja miksi edellistä kaatumista ei tapahdu silloin, vai tapahtuuko?
- En ole täysin tyytyväinen kalojen liikkeeseen, kun ne ajautuvat esteiden lähelle: tällöin esiintyy ihmeellistä ”väpätystä”. Luulen että tämä aiheutuu siitä, että kala siirtyy nopeassa tahdissa avoidance moodista ”normaaliin moodiin” ja takaisin, eli ei tiedä väistääkö estettä vai seurata kavereita.
- Esteiden väistelyssä on hieman huijattu: Este todellisuudessa piirretään siten että sen säde on 5 pikseliä pienempi mitä se on ”todellisuudessa”, koska kala on pistemäinen objekti.

3 parasta ja 3 heikointa kohtaa.

Parhaat kohdat

- Yleinen sovelluksen rakenne on järkevä.
- Kalat liikkuvat kivasti.
- Koodi on siististi kirjoitettu ja käyttäen funktionaalista tyyliä, käyttäen hyväksi map, filter, forAll jne kokoealmametodeja. Niitä on scalassa kiva käyttää, toisin kuin Javassa, jossa niitä ei ymmärrä kukaan.

Heikoimmat kohdat

- Graafisen käyttöliittymän ulkoasu on toteutettu vähän erikoisesti/kummallisesti/ purkkaviritelmätyylisesti.

- Tietyt luokat (esim. Simulation ja FishWorld) voisivat olla objekteja, koska niistä luodaan vain yksi instanssi joka tapauksessa.
- Kommentteja on paikoitellen laiskasti kirjoitettu.

Poikkeamat suunnitelmasta

Suunnitelma piti aika hyvin. Tähän saattaa vaikuttaa se, että kirjoitin suunnitelman, kun olin tehnyt 90 % sovelluksesta valmiiksi. Muutama ominaisuus, jota en toteuttanut oli satunnaisuuden lisääminen liikkeeseen ja erilaiset presetit tai mahdollisuudet tallentaa useampaan save-slottiin.

Arvio lopputuloksesta

Sovellus täyttää sille asetetut kriteerit ja kalojen liikkeestä onnistuin saamaan suhteellisen luonnollisen näköistä pieniä klitsejä lukuun ottamatta. Sovellus on laadittu mielestäni rakenteellisesti järkevästi ja hyvien ohjelmointitapojen mukaisesti. Rakenteellisesti muutoksia olisi voinut tehdä ainakin siten että muutamat luokat (joista luodaan vain yksi instanssi olisi voinut korvata objekteilla.

Välillä ohjelmakoodia laatiessa tuntui siltä, että FishWorld luokka aiheutti ohjelmakoodissa turhaa redundanssia, johtuen sen luonteesta, jossa se lähinnä toimi välikätenä graafisen käyttöliittymän ja simulaation välillä. Toisaalta tämä erillinen adapteriluokka oli laadittu sitä silmällä pitäen, että sovellusta voidaan muokata/laajentaa. Esimerkiksi sovellus voitaisiin melko pienellä vaivalla muokata esimerkiksi sellaiseksi, että FishWorld luokkaan lisätään vaikkapa toisen kalaparven simulaatio rinnalle tai lisätä muunlaisia animoituja kappaleita.

Lisäksi piirreluokka Drawable mahdollistaa sen, että lisättäville graafisille kappaleille voidaan helposti kullekin laatia tietynlainen ”ulkonäkö”.

Toisaalta jos kalaparven haluttaisiin reagoivan muihin animoituihin objekteihin vaatisi tämä aika paljon uudelleenkirjoittamista.

Nyt jos laatisin sovelluksen uudestaan, saattaisin harkita sellaista rakennetta, että abstraktissa luokassa CanvasContent (jonka FishWorld -luokka nyt toteuttaa) olisi määritelty abstraktit metodit käyttöliittymän eri toiminnoille. Tällöin sovelluksen suunnittelun filosofia olisi, se että käyttöliittymä voidaan pitää täysin muuttamattomana ja CanvasContent-luokan toteuttava luokka toimisi paremmin rajapintana nimenomaan käyttöliittymän komponenttien ja sovelluslogiikan välillä.

Kirjallisuusviitteet ja linkit

”Steering behaviors for autonomous characters”:
http://www.red3d.com/cwr/steer/gdc99/

”Boids: Background and update”:
https://www.red3d.com/cwr/boids/

”A Concise Introduction to Scala GUIs”:
https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Scala%20GUI.html

”Some First Steps in Scala Animation”
https://blogscot.wordpress.com/2014/10/12/some-first-steps-in-scala-animation/

Testaamista:
http://www.scalatest.org/
http://www.scalatest.org/user_guide/other_goodies#privateMethodTester

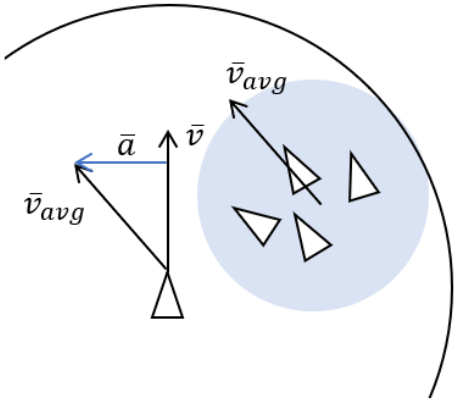
Säikeistä (kurssimateriaali):
https://plus.cs.hut.fi/studio_2/k2018/threads/

Liite: Kolmen Boid-periaatteen implementointi

Alignment

Aligment-ominaisuus tarkoittaa sitä, että kala korjaa liikesuuntaansa siten että kalan liikesuunta olisi sama kuin ympärillä olevien kalojen liikesuunta, jotta parven jäsenet liikkuisivat samaan suuntaan.

Tämän saavuttamiseksi lasketaan kalan lähiympäristössä olevien kalojen normalisoitujen nopeusvektorien summa, joka normalisoidaan.



Kuva L1. Kalan normalisoidun nopeusvektorin $\bar{v}$ ja lähiympäristössä olevien kalojen keskimääräisen (normalisoidun) nopeusvektorin $\bar{v}_{avg}$ avulla voidaan laskea kiihtyvyysvektori.

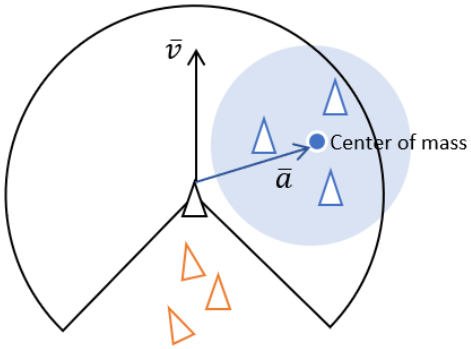
Saadun vektorin ja kalan oman normalisoidun nopeusvektorin avulla voidaan laskea kalan kiihtyvyysvektori seuraavasti:

$$\bar{a}_{alignment} = \bar{v}_{avg} - (\bar{v} \cdot \bar{v}_{avg})\bar{v}\,,$$

missä $\bar{v}$ on kalan normalisoitu nopeusvektori ja $\bar{v}_{avg}$ kalan ympäristössä olevien kalojen normalisoitu keskimääräinen nopeusvektori. Saatu vektori on aina kalan liikettä vastaan kohtisuorassa, ts. se ei vaikuta kalan nopeuteen.

Cohesion

Cohesion on ominaisuus, jossa kala pyrkii kohti lähistöllä olevan parven massakeskipistettä. Jotta kala pyrkisi liikkumaan parven kanssa samaa nopeutta lähiympäristön kaloista suodatetaan pois kalat, jotka sijaitsevat kalan takana, ks. kuva L2.



Kuva L2. Kalan pyrkiessä kohti parven keskipistettä takana olevia kaloja ei huomioida, jotta ryhmän edessä uiva kala ei hidastaisi.

Kiihtyvyysvektori suuruus riippuu etäisyydestä parven massakeskipisteeseen. Kiihtyvyysvektori lasketaan seuraavasti:

$$\bar{a}_{cohesion} = -\bar{r} + \frac{1}{n}\sum \bar{r}_i$$

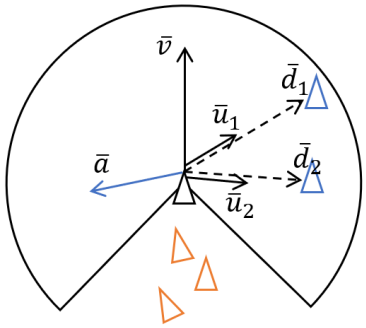
missä $\bar{r}$ on kalan sijainti, n lähistöllä olevien kalojen lukumäärä ja $\bar{r}_i$ lähistöllä olevan kalan sijainti.

Separation

Separation-ominaisuus on vastavoima koheesiolle, joka estää parven liiallista tiivistymistä. Separation-kiihtyvyysvektori saadaan seuraavasti

$$\bar{a}_{separation} = -\sum \frac{\bar{u}_i}{|\bar{d}_i|}$$

missä $\bar{d}_i$ on etäisyys lähellä olevaan kalaan ja $\bar{u}_i$ tämän vektorin normalisoitu vektori. Yksittäisen lähistöllä olevan kalan ”vaikutus” on siis kääntäen verrannollinen etäisyyteen.



Kuva L3. Separation-kiihtyvyys ohjaa kalaa pois muiden kalojen läheisyydestä.

Kokonaiskiihtyvyys

Kolmen ominaisuuden - alignment, cohesion ja separation - avulla saadaan laskettua kalan kiihtyvyysvektori summaamalla saadut kiihtyvyysvektorit yhteen sopivilla vakioilla painottamalla:

$$\bar{a}_{kok} = A \cdot \bar{a}_{alignment} + C \cdot \bar{a}_{cohesion} + S \cdot \bar{a}_{separation}$$

missä *A*, *C* ja *B* ovat vakioita.

Nopeuden ja sijainnin päivitys

Saadun kokonaiskiihtyvyyden avulla jokaiselle kalalle lasketaan uusi nopeusvektori

$$\bar{v} + \bar{a}_{kok} \cdot dt,$$

missä dt on simulaation aika-askeleen pituus.

Johtuen simulaatiotekniikan ominaisuuksista, kalan nopeus saattaa kasvaa mielivaltaisen suureksi. Käytännössä simulaatiossa asetetaan raja-arvo maksiminopeudelle. Jos raja-arvo ylitetään, kalan nopeus skaalataan seuraavasti:

$$\frac{\bar{v}}{|\bar{v}|} \cdot v_{max}$$

Kun kaikkien kalojen nopeudet on päivitetty, lasketaan jokaiselle kalalle aika-askeleen avulla uusi sijainti:

$$\bar{r} + \bar{v} \cdot dt$$