Toni Kirjalainen

OPTIMOINTI JA DATAN VISUALISOINTI

Case: Sähköpostiverkoston visualisointi

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma

Huhtikuu 2017

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Toni Kirjalainen	Tradenomi (AMK)	Huhtikuu 2017

Opinnäytetyön nimi

Optimointi ja Datan Visualisointi
Case: Sähköpostiverkoston visualisointi

43 sivua
1 liitesivua

Toimeksiantaja

Digitalia

Ohjaaja

Jukka Selin

Tiivistelmä

Opinnäytetyön aiheena oli datan visualisointi, ja optimointi ohjelmoinnin näkökulmasta. Tavoitteena oli toteuttaa sähköpostiverkoston visualisointiin tarkoitettu ohjelma, joka luo annetuista sähköposteista voimasuunnatun verkon 3D-ympäristöön. Toimeksiantajana toimi Digitalia, digitaalisen tiedonhallinnan tutkimus- ja kehittämiskeskus.

Opinnäytetyön teoriaosuudessa perehdytään optimointiin ja visualisointiin. Optimointia käsitellään yleisellä tasolla ja Unity3D-pelimoottorin kannalta. Visualisoinnista selvitetään, mitä termillä tarkoitetaan, ja kuinka visualisointia voidaan toteuttaa voimasuunnattujen verkkojen avulla. Käytännön osuudessa ohjelma pilkotaan pieniin osiin, ja selvitetään niiden sisältö ja toiminta. Ohjelman toiminta käydään läpi vaihe vaiheelta, selvittäen kuinka se toimii. Lopuksi pohditaan ohjelman puutteita ja mahdollista jatkokehitystä.

Suurin ongelma ohjelmaa tehdessä oli algoritmin alhainen suorituskyky, joka saatiin ratkaistua käyttämällä näytönohjainta hyödyksi laskennassa. Toimeksiantaja oli tyytyväinen tehtyyn ohjelmaan ja sen toiminnallisuuteen, joten sen osalta opinnäytetyössä päästiin tavoitteisiin.

Asiasanat

visualisointi, optimointi, ohjelmointi

University of Applied Sciences

Author (authors)	Degree	Time
Toni Kirjalainen	Bachelor of Business Administration	April 2017

Thesis Title

Optimization and Data Visualization
Case: Email-network visualization

43 pages
1 pages of appendices

Commissioned by

Digitalia

Supervisor

Jukka Selin

Abstract

The subjects of this thesis were data visualization and software optimization. The objective of the thesis was to create an application for visualizing an email-network that would create a force-directed graph out of given emails in 3D-space. The study was commissioned by Digitalia, research center on digital information management.

Optimization and visualization were explained in the theory part of the thesis. Optimization was covered on a general level and from the Unity3D game engine's standpoint. Visualization was clarified as a term and how it could be implemented with the help of force-directed graphs. In the practical part the software was divided into smaller parts which were examined by what they consisted of and how they worked. The application's operations were introduced step by step. In the end the application's shortcomings and possible improvements were considered.

The largest problem when creating the program was the algorithm's low performance which was solved by using the graphics card for computing. Digitalia was satisfied with the program and its functionality, based on which the objective was reached.

Keywords

visualization, optimization, programming

SISÄLLYS

LIITTEET

1 JOHDANTO

Tämän opinnäytetyön aiheena on optimointi ohjelmoinnissa ja datan visualisointi. Optimoinnissa käydään läpi yleisellä tasolla, mitä sillä tarkoitetaan ja millaisilla tavoilla optimointia yleisesti toteutetaan. Optimointi on jaoteltu yleiseen ja Unity3D:ssä tehtyyn optimointiin, sillä se on yleisesti hyvin erilaista peliohjelmoinnissa, verrattuna perinteiseen ohjelmointiin. Datan visualisoinnissa käydään läpi hieman visualisoinnin teoriaa, mutta erityisesti katsotaan voimasuunnattujen verkkojen avulla toteutettavaa visualisointia.

Toimeksiantona tehtiin ohjelma sähköpostiverkoston visualisointiin 3D-ympäristöön, jossa ilmeni tarpeita optimoinnille ja josta tämän opinnäytetyön aiheet saivat alkunsa. Työssä käydään läpi miltä valmis ohjelma näyttää ja mitä sillä pystyy yleisesti ottaen tekemään. Toteutettu ohjelma käydään myös suhteellisen tarkasti läpi kohta kohdalta ja selvennetään, kuinka ohjelma toimii. Työssä katsotaan myös vastaan tulleita ongelmia ja kuinka ohjelmaa voisi vielä kehittää jatkossa eteenpäin.

2 OPTIMOINTI YLEISESTI

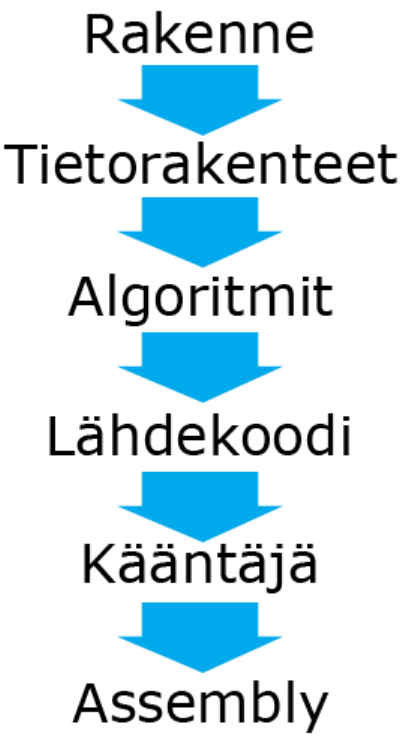
Ohjelmoinnissa optimoinnilla tarkoitetaan prosessia, jolla ohjelma saadaan toimimaan mahdollisimman tehokkaasti suorituskyvyn, virrankulutuksen tai jonkin resurssin (esimerkiksi keskusmuistin tai levytilan) kannalta. Optimointia miettiessä onkin tarkasteltava ohjelman käyttökohde huomioon ottaen, minkä kriteerin kannalta ohjelmaa aletaan optimoida. On esimerkiksi täysin mahdotonta, että ohjelmasta otetaan kaikki mahdollinen suorituskyky irti ja samalla ohjelma kuitenkin toimisi mahdollisimman pienellä virrankulutuksella ja veisi minimaalisen määrän keskusmuistia.

Optimointiin kuuluu useita erilaisia tekniikoita, joita hyödyntäen ohjelma saadaan toimimaan nopeammin ja tehokkaammin. Ohjelmaa koodatessa täytyy miettiä mitkä olisivat sopivat algoritmit kyseisen tehtävän suorittamiseen. Myös tietorakenteet täytyy valita käyttökohteen mukaan. Tietysti ohjelmasta pitää poistaa myös kaikki tarpeeton kuten ylimääräiset funktiokutsut, ehtolauseet ja referenssit. Edellä mainitut optimoinnit ovat sellaisia, joihin ei yleisesti

vaikuta millä kohdelaitteistolla ohjelmaa ajetaan. (Bryant & O'Hallaron 2001, 474–475.)

2.1 Optimoinnin tasot

Optimoinnin voi jaotella tasoihin, jotka menevät korkean tason koodista kuten C#, kohti alemman tason koodia eli konekieltä. Yleisesti ottaen korkean tason optimoinnin hoitaa ohjelmoija, kun taas alemman tason optimointi jää kääntäjälle. Alemman tason optimointia voi tehdä myös käsin, mutta se on ajallisesti kallista ja vaatii ohjelmoijalta paljon tietoa ja taitoa. (Code Optimization s.a.)



Kuva 1. Optimoinnin tasot karkeasti

Korkeimmalta löytyy ohjelmiston yleinen rakenne ja kehikko. Rakenteellisilla muutoksilla voi saada huomattavia parannuksia suorituskyvyn suhteen, mutta ne yleisesti vaativat myös reilusti aikaa ja muutoksia koodiin, mikäli niitä joutuu muuttamaan kehityksen myöhemmissä vaiheissa. Esimerkiksi ohjelmointikielen muutos kesken projektin voi vaatia paljon ylimääräistä työtä.

Tietorakenteet ovat yksi helpoimmista optimointikohteista, sillä niiden muutokset eivät vaadi yleisesti hirveän paljon muutoksia muuhun koodiin. Eri tietorakenteilla on omia vahvuuksia ja heikkouksia, jonka vuoksi oikean tietoraken-

teen käyttö voi vaikuttaa suorituskykyyn ratkaisevasti. Esimerkiksi taulukkoa ei kannatta käyttää, jos tietorakenteeseen kohdistuu paljon hakuja, koska huonoimmassa tapauksessa taulukosta tiedon etsiminen tarkoittaa koko taulukon läpikäymistä. (Mitchell 2003.)

”Ohjelmoinnissa algoritmi on kokoelma järjestettyjä, hyvin määriteltyjä käskyjä ohjelman suorittamiseen” (Algorithm in Programming s.a). Algoritmeja miettiessä kannattaa ensin tutkia, onko sen hetkiseen ongelmaan jo olemassa valmiiksi mietitty ja testattu algoritmi. Paremmalla algoritmilla saa myös huomattavasti enemmän nopeutta kuin itse koodin optimoinnilla. Jos algoritmi on tarpeeksi huono, ei ohjelmasta välttämättä saa tarpeeksi nopeaa millään alemman tason optimoinneilla.

Seuraavana on itse koodin optimointi. Tässä vaiheessa koodista pyritään ottamaan kaikki ylimääräinen pois. Laskentaoperaatioita yritetään vähentää tai vaihtaa niitä laskennallisesti halvempiin vaihtoehtoihin. Silmukat ja ehtolauseet optimoidaan. Prosessorin välimuistia pyritään käyttämään mahdollisimman tehokkaasti.

Kääntäjän optimoinnilla tarkoitetaan sitä, kun koodi kirjoitetaan niin, että kääntäjä osaa luoda siitä mahdollisimman optimoitua konekieltä (Bryant & O’Hallaron 2001, 475). Kääntäjä on ohjelma, joka kääntää kirjoitetun koodin muotoon, jota tietokone ymmärtää (Definition of: compiler 2016). Yksi iso osa optimoinnista onkin kirjoittaa korkean tason koodista (esimerkiksi C#) sellaista, jonka kääntäjä osaa optimoida mahdollisimman hyvin. Nykyään kääntäjät osaavat optimoida koodia jo erittäin hyvin monissa eri tilanteissa, muttei kääntäjäkään kaikkeen pysty. On esimerkiksi optimointeja, jotka toimivat vain tietyissä tilanteissa tietyn tyyppisillä arvoilla, ja kääntäjä ei voi etukäteen tietää minkälaisia arvoja ohjelma ottaa vastaan, joten se ei myöskään yritä tehdä tämän tyyppisiä optimointeja. Kääntäjät käyttävät oletusasetuksilla vain turvallisia optimointeja, jolloin ohjelma toimii samalla tavalla kaikissa tilanteissa (Bryant & O’Hallaron 2001, 477).

Assembly on matalan tason ohjelmointikieli, joka on yhden askeleen ylempänä konekieltä. Assembly kieli voi vaihdella hieman eri prosessori-tyyppien välillä, sillä assembly-kielessä on vain operaatiot, jotka on määritelty itse proses-

sorissa (Assembly Language s.a.). Kääntäjien kehityksen myötä optimointia suoritetaan melko harvoin näin matalalla tasolla varsinkaan isommissa projekteissa. Mitä pidemmälle kääntäjien kehitys etenee, sen vaikeampaa assemblykielellä on saada huomattavia suorituskykyeroja.

2.2 Milloin optimoida?

Yleinen ongelma on, että yksittäisten metodien suorituskykyä käydään miettimään jo ennen kuin niitä edes testataan. Tässähän ei sinällään ole mitään väärää, sillä kannattaa suorituskyky aina pitää mielessä. On kuitenkin myös hyvä muistaa, ettei muutaman millisekunnin hyödyn takia kannata haaskata hirveän paljon aikaa.

Optimointia kannattaa alkaa miettiä vasta siinä vaiheessa, kun ohjelman suorituskyvyn kanssa alkaa esiintyä ongelmia. Optimoitu koodi voi olla vaikealukuista, ja tämä vaikuttaa ohjelman kehitykseen negatiivisesti, mutta kannattaa huomioida, ettei näin aina ole (Mitchell 2014). Suunnitteluvaiheessa kannattaa käyttää aikaa ohjelman yleiseen rakenteeseen ja algoritmien mietintään välttääkseen tarvetta kirjoittaa isoja osia koodista uudestaan.

Optimointia aloittaessa on tärkeä tietää, mitä asioita koodissa lähtee optimoimaan. Tätä varten on erilaisia työkaluja koodin profilointiin. Näiden työkalujen avulla on yleensä suhteellisen helppoa nähdä, mitkä asiat vievät ohjelmassa eniten aikaa ja/tai resursseja. Tietysti vaikka profiloija näyttää mitkä asiat koodissa vievät paljon resursseja, voi välillä olla haastavaa ymmärtää miksi näin tapahtuu ja kuinka koodista saisi suorituskykyisempää. Profiloinnilla vältetään myös optimoinnit, joilla ei ole suorituskyvyn kannalta suurta merkitystä.

Yleisesti ottaen ohjelmoinnissa koodi pyritään pitämään mahdollisimman helppolukuisena ja modulaarisena. Optimointi, varsinkin jos ohjelmasta yritetään ottaa kaikki mahdollinen suorituskyky irti, voi johtaa koodin huonoon luettavuuteen ja jaotteluun. Optimointi kannattaakin jättää kehityksen viimemetreille, varsinkin jos optimoinnin kohteena on suurempi projekti. Aina kannattaa myös pitää mielessä, ettei optimointi ole välttämätön toimenpide, jos suorituskyvyn kanssa ei ole ongelmia.

3 SUORITUSKYVYN OPTIMOINTI UNITY3D:SSÄ

Unity3D on Unity Technologiesin kehittämä pelimoottori, joka tukee monia alustoja kuten Xbox One, Playstation 4 ja Windows ja jolla voi tehdä 2D-, 3D-, VR- ja AR -pelejä (Fast Facts 2016). Tässä osiossa käydään läpi optimointitekniikoita jotka soveltuvat käytettäväksi Unity3D:llä tehtyihin peleihin ja ohjelmiin. Suuri osa näistä tekniikoista toimii kuitenkin myös muissa pelimoottoreissa tai ohjelmoinnissa yleisesti.

Optimointi on erityisen yleistä pelinkehityksessä hyvin yksinkertaisesta syystä. Peleistä halutaan tehdä mahdollisimman näyttäviä ja hienoja uusimpia tekniikoita hyväksi käyttäen, koska jo pelkästään pelin hyvällä ulkoasulla voidaan saada paljon myyntiä. Kuvataajuus (Frame rate) eli toisin sanoen pelin suorituskyky täytyy kuitenkin pitää hyväksyttävällä tasolla. Pelintekijöiden täytyy myös huomioida, ettei kaikilla käyttäjillä ole uusimpia ja tehokkaimpia komponentteja. Jo pelkästään näiden asioiden pohjalta voidaan todeta, että suorituskyky on erittäin iso ja tärkeä osa peliohjelmointia. Peliohjelmoinnissa optimoinnilla voidaan myös tarkoittaa asioita kuten tekstuurien tai varjojen tarkkuuden pienennystä, joka ei ole oikeaa optimointia tietotekniikan näkökulmasta, sillä se muuttaa ohjelman/pelin lopputulosta.

3.1 Profiler

Unity3D:n mukana tulee Unity3D-projektin profilointiin tarkoitettu työkalu nimeltä Profiler, joka tarkkailee ja tallentaa pelin suorituskykyyn liittyvää dataa. Profiloinnin avulla voi selvittää, mitkä asiat pelissä vievät eniten aikaa ja resursseja. Sillä voi myös nopeasti testata, miten erilaiset muutokset koodiin vaikuttavat suorituskykyyn. Työkalua käyttäessä täytyy kuitenkin huomioida, että profiloinnin ollessa käytössä peli voi toimia hitaammin kuin normaalisti.

Profilointi-työkalun päänäkymässä näkyy kaikki lisätyt profiloijat, kuten esimerkiksi prosessorin käyttö, piirto (rendering) ja muistin käyttö. Pääsääntöisesti profilointi tapahtuu yksi kuva (frame) kerrallaan. Eri profiloijat näyttävät erilaista infoa kuvaajien muodossa jotka kertovat, paljonko jotakin resurssia käytetään. Profiler näyttää myös lisätietoja valitusta profiloijasta. CPU-

profiloija, joka on optimoinnin kannalta oleellisin, näyttää alemmassa osiossa sillä hetkellä valitun kuvan aikana suoritetut metodit.

Kuva 2. Profiler-työkalun päänäkymä

CPU-profiloija antaa tietoja suoritetuista metodeista. Suoritukseen käytettyä aikaa näytetään millisekunteina, sekä prosentuaalisesti verrattuna muihin metodeihin. Profiloija näyttää myös, montako kertaa kutakin metodia on kutsuttu kyseisen kuvan (frame) aikana. Muistinkäytön osalta näytetään paljonko roskankerääjälle (Grabage Collector) on asetettu muistia vapautettavaksi. Roskankeräys on näkymätön, prosessorille suhteellisen raskas operaatio (Understanding Automatic Memory Management 2016), ja tämän takia se näytetään CPU-profiloijassa.

Yleisesti profiloijan antamat tiedot oletusasetuksilla eivät riitä antamaan ideaa siitä, missä mahdolliset pullonkaulat ovat. Tätä varten työkalussa on mahdollisuus ottaa käyttöön syvä profilointi (Deep Profile), jolloin kaikki ajetut metodit profiloidaan. Tämä kuitenkin hidastaa profiloitavaa peliä huomattavissa määrin ja jos peli on tarpeeksi iso, syvää profilointia ei välttämättä edes pysty käyttämään. Vaihtoehtoisesti koodiin voi rajata halutut alueet Unity3D:n Profiler-luokan avulla, jolloin voi välttää turhaa profilointia (Profiler Class Documentation 2016).

3.2 Yleisiä optimointikohteita peliohjelmoinnissa

Peliä tehdessä tulee yleisesti vastaan suorituskykyongelmia, jotka eivät ole sidonnaisia kirjoitettuun koodiin. Vaikka pelin koodit olisi optimoitu kuinka hyvin, pelin suorituskyky voi jäädä liian alhaiseksi. Pelioptimoinnissa on myös yleistä, ettei laskentataakkaa pienennetä, vaan sitä siirretään esimerkiksi prosessorilta näytönohjaimelle. Suorituskyvyn parantaminen pelin ulkonäön kustannuksella on myös yleistä.

3.2.1 Piirtokutsut

Piirtokutsu (Draw Call) on prosessorin antama käsky näytönohjaimelle piirtää (Render) yksi verkko (Mesh). Ennen piirtokutsua prosessorin täytyy myös määrittää, kuinka verkot piirretään. Tätä operaatiota kutsutaan nimellä piirtotila (Render State). Näihin määrityksiin kuuluu esimerkiksi materiaalit, tekstuurit ja valot. Nämä käskyt eivät ole yksittäin raskaita, mutta objektien määrän noustessa alkaa se myös näkyä suorituskyvyssä. (Trümpler 2015.)

Piirtokutsuja on mahdollista suorittaa erissä (Draw Call Batching), jolloin piirtokutsujen kokonaismäärä pienenee. Unity3D käyttää kahta tekniikkaa tämän suorittamiseen: dynaaminen niputus (Dynamic Batching) ja staattinen niputus (Static Batching). Unity3D käyttää dynaamista niputusta automaattisesti silloin kun se on mahdollista, kun taas staattinen niputus tehdään manuaalisesti merkkaamalla halutut peliobjektit staattiseksi. Kumpikin tekniikkaa vaatii, että kyseessä olevilla peliobjekteilla on samat materiaalit, joten materiaalien uudelleenkäyttö on tärkeää. Dynaamisella niputuksella on tämän lisäksi myös monia muita ehtoja, joiden täytyy täyttyä. Näistä voi lukea lisää mainitusta lähteestä. Näiden lisäksi piirtokutsuja voi myös vähentää yhdistämällä toisiaan lähellä olevia verkkoja (Mesh) manuaalisesti. (Draw call batching 2016.)

3.2.2 Valot ja varjot

Valot ja varjot ovat yksi suorituskyvyn kannalta kalliimpia asioita. Niissä joudutaan tinkimään, jotta kuvataajuus saadaan pidettyä tarpeeksi korkealla. Unity3D asettaa valoja joko verteksi-valotuksella (Vertex Lighting) tai pikseli-

valotuksella (Pixel Lighting). Verteksi-valotus laskee valotuksen verteksien arvojen mukaan ja levittää sen perusteella valon objektin pinnalle. Pikseli-valotus laskee valotuksen joka pikselille erikseen. Unity3D:n oma algoritmi määrittelee, mitkä valot piirretään milläkin tavalla. Tähän voi vaikuttaa hieman asettamalla haluamansa valot korkeammalle prioriteetille, jolloin Unity3D priorisoi näitä valoja pikseli-valotuksen suhteen. Pikseli-valotuksella valotettujen valojen maksimi määrän voi määrittää Unity3D:n laatu-asetuksista. (Light troubleshooting and performance 2016.)

Reaaliajassa lasketut valot ja varjot vievät paljon laskentatehoa. Tästä syystä niitä pyritään välttämään, kun se on mahdollista. Valojen kartoituksella (Light-mapping) staattisille objekteille voidaan laskea valot ja varjot etukäteen. Useasti etukäteen lasketuista valoista ja varjoista saadaan myös paremman näköisiä esimerkiksi heijastuksien avulla ja ovat liian raskaita laskettaviksi reaaliajassa. Unity3D:n 5.0 versio lisäsi myös valojen kartoituksen ja reaaliaikaisen varjojen laskennan yhdistelmän (Precomputed Realtime GI), joka laskee kaikki mahdolliset heijastukset etukäteen ja käyttää tätä informaatiota ajon aikana. (Global Illumination 2016.)

3.2.3 Sekalaisia optimointeja

3D-malleissa kiinnitetään yleisesti huomiota verteksien määrään, sillä näytönohjain joutuu tekemään enemmän töitä, mitä isompi määrä niitä on malleissa (Optimizing graphics performance 2016). Yleisesti ottaen objekteihin, joita pelaaja ei välttämättä tule katsomaan hirveän läheltä, ei kannata laittaa hirveästi detaljeja, toisin kuin esimerkiksi pelaajahahmoihin. Sama sääntö pätee myös tekstuurien resoluution kanssa. Tekstuurien kanssa kannattaa myös käyttää pakkausta joka vähentää muistin käyttöä ja nopeuttaa niiden piirtämistä (Optimizing graphics performance 2016).

Piirtoetäisyydellä saadaan tyypillisesti vähennettyä laskentataakkaa huomattavasti ja tätä tekniikkaa käytetäänkin melkein pelissä kuin pelissä. Tällä tarkoitetaan objektien piilottamista, kun ne ovat jonkin tietyn välimatkan ulkopuolella pelaajaan nähden (Optimizing graphics performance 2016). Yleisesti pienemmillä objekteilla ei ole hirveän suurta piirtoetäisyyttä, kun taas esimerkiksi isoilla rakennuksilla ei välttämättä ole mitään rajoituksia. Toinen tähän liittyvä

tekniikka on rajoittaa objektien yksityiskohtien määrää (Level Of Detail) mitä kauempana niistä sijaitaan (Level of Detail (LOD) 2016). Rakennuksessa on turha olla hirveää määrää yksityiskohtia, jos sitä katsotaan kilometrin päästä eikä näitä detaljeja ole mahdollista nähdä. Piirtoetäisyyden lisäksi objekteja voidaan myös jättää piirtämättä kameran näkymän ulkopuolelta, vaikka ne olisivat pelaajan lähellä, tai jos objektit ovat vaikkapa jonkin toisen objektin takana (Occlusion Culling).

3.3 Multithreading

Monisäikeinen ohjelmointi (Multithreading) tarkoittaa sitä, kun yksi prosessi käyttää useaa säiettä (Thread) tehtävien suorittamiseen. Näitä säikeitä ei välttämättä suoriteta saman aikaisesti, sillä useita säikeitä on mahdollista olla myös yksi-ytimisissä järjestelmissä. Vaikka säikeitä ei suoritettaisi samanaikaisesti, käyttöjärjestelmän järjestelijän (Scheduler) ansiosta ohjelma näyttää tekevän useaa asiaa samanaikaisesti käyttäjän näkökulmasta. Yksinkertaisesti selitettynä, käyttöjärjestelmä käyttää ajastettua tarkistusta, joka tarkastaa tietyin väliajoin onko seuraavia säikeitä jonossa. Näitä säikeitä ajetaan tietyn asetetun ajan verran, jonka jälkeen tarkistus tehdään uudelleen. Tätä asetettua aikaa kutsutaan aikaviipaleeksi (Time Slice), ja se on yleisesti 10–20 millisekuntia riippuen käyttöjärjestelmästä. Jos esimerkiksi kahden säikeen suorittamista vaihdellaan 20 millisekunnin välein, käyttäjälle tämä näyttää kuin kahta asiaa suoritettaisiin samaan aikaan. (Scali 2012.)

Usean säikeen käyttö vaatii algoritmien rinnakkaistamisen, eli muokkauksen sellaisiksi, että niitä on mahdollista suorittaa monella säikeellä samanaikaisesti. Tämä tapahtuu yleisesti pilkkomalla algoritmi pienempiin osiin. Käytetään esimerkkinä yksinkertaista laskutoimitusta:
$vastaus = a \cdot b + c \cdot d$

Kyseisen laskutoimituksen voi laskea kahdessa osassa:
$säie1 = a \cdot b$
$säie2 = c \cdot d$
$vastaus = säie1 + säie2$

Tästä näemme, ettei koko algoritmia pysty suorittamaan rinnakkain ja se joutuu odottamaan, kunnes molemmat säikeet ovat tehneet työnsä loppuun. Tämä on yksi rinnakkaistamisen rajoituksista. Tyypillisesti kaikkea algoritmin laskennasta on mahdotonta laskea rinnakkain ja tämän takia säikeillä harvoin saavutetaan lineaarista nousua suorituskyvyssä. Tähän liittyen voi lukea lisää Amdahlin ja/tai Gustafsonin laista jotka käsitelevät asiaa (ks. esim. Multiprocessor Laws s.a).

Unity3D:n ohjelmointirajapinta (API eli Application Programming Interface) on suunniteltu käytettäväksi vain Unity3D:n pääsäikeessä (Main Thread), joten se ei ole turvallinen usean säikeen kanssa (Thread Safe). Tästä syystä Unity3D:n omia luokkia ja metodeja ei voi käyttää uusien säikeiden kanssa, muutamaa poikkeusta lukuun ottamatta. Tämä ei kuitenkaan tarkoita, ettei Unity3D:n kanssa ohjelmoidessa voisi käyttää säikeitä. Se olisi vain helpompaa, jos Unity3D itsessään tukisi usean säikeen käyttöä. Esimerkiksi peliobjekteilla (GameObject), jotka ovat keskeinen osa Unity3D-pelimoottoria, ei pysty tekemään mitään muualla kuin pääsäikeessä.

Säikeitä voi käyttää joko laskemaan jokin asia nopeammin yhden kuvan (frame) aikana tai laskea taustalla usean kuvan (frame) aikana keskeyttämättä pääsäiettä. Jälkimmäistä käytetään, jotta pelin käyttöliittymä ottaa käyttäjän käskyjä vastaan, eikä lukitu koko operaation ajaksi. Välillä kuitenkin jonkun raskaamman laskun tuloksia tarvitaan jo saman kuvan (frame) aikana, eikä tällöin laskennan levittäminen useamman kuvan ajaksi ole mahdollista. Tällöin säikeiden käytöllä rinnakkain voidaan saada lisää laskentatehoa.

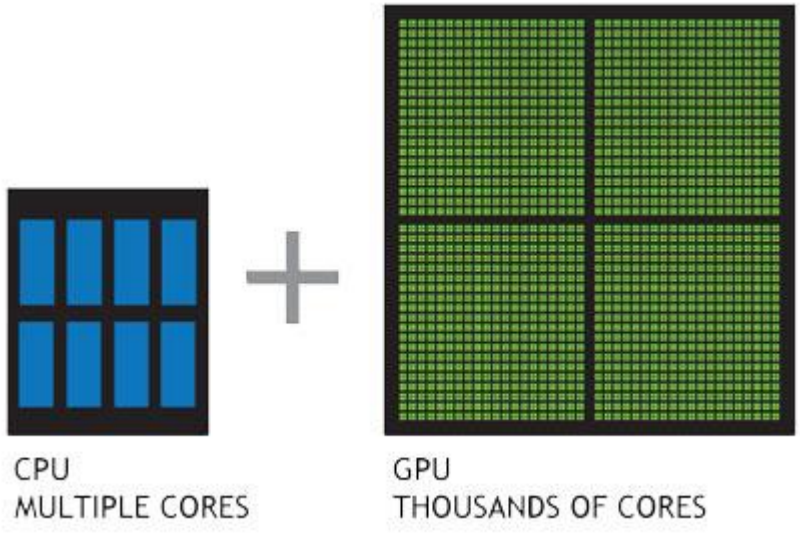
Käytännön toteutus onnistuu melko pitkälti samalla lailla, kun missä tahansa C#-projektissa, eli käyttämällä System.Threading-luokkaa. Unity3D:n tämän hetkinen versio käyttää kuitenkin .NET-kehikon vanhaa versiota, jossa ei ole tarjolla uusimpia säikeisiin liittyviä luokkia ja kirjastoja.

3.4 GPGPU

GPGPU (General-Purpose computation on Graphics Processing Units) termillä tarkoitetaan sitä, kun näytönohjaimella (GPU) suoritetaan yleistä laskentaa, joka on perinteisesti laskettu prosessorilla (Bawaskar ym. 2012, 106). Näy-

tönohjaimissa on huomattavasti enemmän laskentatehoa kuin prosessoreissa, joten sitä pyritään hyödyntämään mahdollisimman paljon. Vertaillessa prosessorin ja näytönohjaimen laskentatehoa kannattaa kuitenkin huomioida, että prosessorit on suunniteltu tekemään paljon erilaisia tehtäviä, kun taas näytönohjaimella on melko rajattu käyttökohde (Bawaskar ym. 2012, 107-108).

Nykyajan prosessoreissa on yleisesti 2–8 ydintä, kun taas näytönohjaimissa on useista sadoista jopa tuhansiin (Kuva 3). Eri näytönohjainvalmistajien näytönohjaimissa ytimet voivat olla hyvinkin erilaisia. Näytönohjain jossa on enemmän ytimiä, ei siis ole välttämättä parempi.



Kuva 3. Ytimien havainnollistus (Ytimien havainnollistus s.a.)

Näytönohjaimen ytimien ominaisuudet ovat rajallisia mutta koska niitä on todella paljon, näytönohjain on äärimmäisen nopea rinnakkaislaskennassa. Näytönohjain on suunniteltu data-rinnakkaista (Data-Parallelism) laskentaa varten joka on yksi rinnakkaislaskennan muodoista jossa samoja, tai hyvin samankaltaisia operaatioita suoritetaan datan eri elementeille (GPU Parallelizable Methods s.a). Data-rinnakkaisuuteen erikoistuneen suunnittelun takia näytönohjain on prosessoria huomattavasti nopeampi, mikäli data-rinnakkaisuutta päästään hyödyntämään (Bawaskar ym. 2012, 107-108).

3.4.1 Laskentavarjostimet

Unity3D:ssä GPGPU hoidetaan laskentavarjostimien (Compute Shader) avulla (Kuva 4), jotka ovat assetteina projektissa samaan tapaan muiden varjostimien kanssa. Laskentavarjostimet ovat ohjelmia, jotka suoritetaan näytönohjaimella, erillään muista näytönohjaimen tehtävistä.

```
1  #pragma kernel CSMain
2
3  RWTexture2D<float4> Result;
4
5  [numthreads(8,8,1)]
6  void CSMain (uint3 id : SV_DispatchThreadID)
7  {
8      Result[id.xy] = float4(1,0,0,1); // Värjää kyseinen pikseli punaiseksi
9  }
```

Kuva 4. Esimerkki laskentavarjostimesta

Laskentavarjostimet kirjoitetaan HLSL-kielellä, joka on korkean tason varjostinkieli DirectX-rajapinnalle. HLSL-kielen lisäksi laskentavarjostimissa käytetään #pragma-direktiivejä. Niiden avulla määritellään kernelit, joilla varjostinta kutsutaan. Laskentavarjostimet vaativat Shader Model 5.0 tuella varustetun näytönohjaimen toimiakseen. (Compute Shaders 2016.)

Esimerkissä (Kuva 4) hakasulkeissa oleva numthreads-määritys määrittää, montako säiettä yksi säikeiden joukko (Thread Group) luo kerrallaan. Säikeiden määrä annettaan kolmiulotteisena taulukkona (X * Y * Z), eli esimerkissä yhdellä joukolla on 64 säiettä. Säikeiden maksimimäärä yhden joukon sisällä Unity3D:ssä on tällä hetkellä 1024, ja lisäksi Z-akselilla maksimi syvyys on 64. (Numthreads s.a.)

Laskentavarjostimen ajo Unity3D:n skripteissä onnistuu suhteellisen helposti. Unity3D:n ComputeShader-luokka tarjoaa tarvittavat työkalut laskentavarjostimen alustukseen ja ajamiseen.

```
using UnityEngine;

public class runCS : MonoBehaviour {

    public ComputeShader shader;

    void Start () {
        int kernelIndeksi = shader.FindKernel("CSMain"); // Etsitään monesko kernel "CSMain" on kyseisessä shaderissä

        // Luodaan uusi tekstuuri ja sallitaan RandomWrite, jotta näytönohjain saa kirjoittaa tektuuriin
        RenderTexture tekstuuri = new RenderTexture(256, 256, 24);
        tekstuuri.enableRandomWrite = true;
        tekstuuri.Create();

        // peliobjektin (johon tämä skripti on kiinnitetty) renderer
        Renderer renderer = GetComponent<Renderer>();
        renderer.enabled = true;

        shader.SetTexture(kernelIndeksi, "Result", tekstuuri); // Asetetaan tekstuuri näytönohjaimelle
        shader.Dispatch(kernelIndeksi, 256 / 8, 256 / 8, 1); // Ajetaan shader

        renderer.material.SetTexture("_MainTex", tekstuuri); // Asetetaan tekstuuri
    }
}
```

Kuva 5. Esimerkki laskentavarjostimen alustuksesta ja ajosta

Esimerkin (Kuva 4) tapauksessa varjostimelle täytyy antaa tekstuuri, johon se tallentaa muutokset, jotta siinä olisi mitään järkeä. Tämä onnistuu ComputeShader-luokan SetTexture-metodilla. Kun laskentavarjostimelle on annettu tarvittavat resurssit, se voidaan ajaa Dispatch-metodilla. Ensimmäinen parametri tälle metodille ottaa vastaan kernelin indeksin. Indeksin avulla varjostin tietää, mikä sen määritellyistä kerneleistä ajetaan. Seuraavalla kolmella kokonaisluvulla määritellään kolmiulotteisen taulukon muodossa, montako säikeiden joukkoa ajetaan, eli montako kertaa kernel ajetaan.

Yleisesti laskentavarjostimet tarvitsevat dataa, jota käytetään jonkin määritetyn asian laskennassa. Kyseisen laskennan tulokset täytyy myös saada luettua muistista. Tämä onnistuu ComputeBuffer-luokan avulla (ComputeBuffer Class Documentation 2016), jolla dataa voidaan lukea ja kirjoittaa näytönohjaimen muistiin. ComputeBuffer luodaan ja täytetään Unity3D:n skripteissä jonka jälkeen sitä voidaan käyttää laskentavarjostimissa. Luokkaa käyttäessä kannattaa muistaa, että tiedon siirto näytönohjaimelle ja sieltä pois on suhteellisen hidasta.

3.4.2 Hyvät ja huonot käyttökohteet

GPGPU on erittäin tehokas tekniikka tilanteessa, jossa samoja operaatioita voidaan tehdä datan jokaiselle elementille aiemmin mainitun datarinnakkaisuuteen erikoistuneen suunnittelun takia. Yleisesti myös suorituskyky paranee, mitä isompi määrä dataa on käsiteltävänä. Tällöin näytönohjaimen suurta määrää ytimiä päästään hyödyntämään samanaikaisesti paremmin.

Suurella data-määrällä näytönohjaimen sisäisen muistin hyödyntäminen on myös todennäköisempää, johon näytönohjain pääsee todella nopeasti käsiksi. Tämä tietysti riippuu myös käytetystä algoritmista. Harris ym. (2007) totesivat tämän suorituskyvyn nousun isolla datamäärällä tehdessään optimointeja heidän kaikki parit (All-pairs) läpikäyvään algoritmiin, joka visualisoi verkon 3D-ympäristöön.

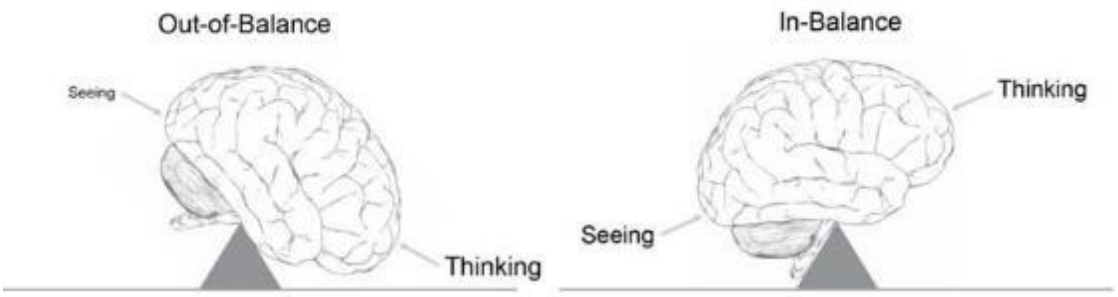
Tiedon siirto keskusmuistista näytönohjaimen muistiin tai toisinpäin on hidasta. Tämän vuoksi näytönohjaimella ei kannata laskea algoritmeja, jossa on vähän laskettavaa verrattain näytönohjaimelle siirrettävän tiedon määrään. Tällöin iso osa ajasta kuluu tiedon siirrossa, ei laskennassa. Kannattaakin käyttää Unity3D:n profilointi-työkaluja hyväksi ja tarkastaa, onko laskenta ylipäänsä kannattavaa näytönohjaimella vai kannattaako laskenta hoitaa perinteisesti prosessorilla.

Näytönohjaimet ovat huomattavasti hitaampia laskemaan double-tyyppisillä arvoilla verrattuna float-tyyppisiin arvoihin. Float on 32 bittinen ja double 64 bittinen liukuluku. Peleissä ei yleisesti tarvita niin tarkkoja lukuja, että 64-bittiset liukuluvut olisivat tarpeellisia ja sillä suuri osa näytönohjaimista on tehty pelaamiseen, iso osa niistä ei ole nopeita niiden laskennassa. Mikäli ylimääräistä tarkkuutta ei tarvi, kannattaa aina käyttää float-tyyppisiä muuttujia. (Explaining FP64 performance on GPUs 2015.)

4 VERKKOJEN VISUALISOINTI JA TEORIA

Abstraktin tiedon näyttäminen fyysisien ominaisuuksien (esimerkiksi koko, väri ja muoto) avulla on tiedon visualisointia. Visualisoinnilla pyritään tuomaan haluttuja asioita esille kohteena olevasta informaatiosta ja sillä pyritään edistämään datan analysointia ja/tai kommunikaatiota. Ihmisen havainnointikyky ja kognitio ovat olennainen osa visualisointia, joiden ymmärtämisen kautta saa suunniteltua tehokkaita visualisointeja. (Few 2014.)

Fewin (2014) mukaan tiedon visualisointi on tehokasta, sillä se tasapainottaa aivojen työskentelyä ajattelun ja havainnoinnin välillä.



Kuva 6. Ajattelun ja havainnoinnin tasapaino (Few 2014)

Visuaalinen havainnointi tapahtuu aivojen takaosassa, näköaivokuoressa, joka on todella nopeaa ja tehokasta. Verrattuna aivojen etuosa, jossa kognitio tapahtuu, on suhteellisen hidas (Few 2014). Tätä voisi rinnastaa tietokoneessa näytönohjaimeen ja prosessoriin, jossa näytönohjainta pyritään käyttämään nykyään yhä enemmän ja enemmän prosessorin rinnalla sen laskentatehon vuoksi.

4.1 Verkkoteorian oleellinen termistö

Verkko, toiselta termiltään graafi, koostuu solmuista ja näitä yhdistävistä kaarista. Solmuja voidaan myös kutsua nimellä piste. Kaaret voivat olla suunnattuja ja tällöin niitä kutsutaan nuoliksi. Verkkoja sovelletaan monella eri alalla, ja esimerkkinä voidaan mainita vaikkapa sähköverkko, erinäiset tietopankit ja Internet. (Pesonen 2013, 1.)

Verkkoja voidaan luokitella sen mukaan, minkälaisia ominaisuuksia sen solmuilla ja kaarilla on. Verkko on suunnattu, suuntaamaton tai näiden sekoitus, riippuen onko sen kaikki kaaret suunnattuja vai ei (Graafit 2007, 468). Verkko voi olla myös painotettu, jolloin sen solmuilla ja kaarilla on painokertoimet (Pesonen 2013, 1).

Solmuja kutsutaan vierekkäisiksi tai naapureiksi, jos niiden välissä on kaari. Solmulla voi siis olla useampia vierekkäisiä solmuja. Solmuun yhdistyneiden kaarien määrä kertoo solmun asteluvun. Mikäli solmun asteluku on 0, sitä kutsutaan erilliseksi tai eristetyksi. (Pesonen 2013, 3.)

4.2 Voimasuunnatut algoritmit

Voimasuunnatut (Force-directed Layout Algorithm), tai toiselta nimeltä jousi sijoitetut (Spring Layout Algorithm), algoritmit ovat algoritmien luokka, joita käytetään graafien solmujen sijoittelun laskemiseen kaksi- tai kolmiulotteiseen tilaan. Yleisimmin nämä algoritmit käyttävät jousen omaista voimaa laskemaan vetovoimaa yhdistyneiden solmujen välillä, ja sähköisesti varattujen kappaleiden omaista voimaa laskemaan työntövoimaa kaikkien solmujen välillä. Algoritmit voivat myös hyödyntää esimerkiksi barysentrisiä koordinaatteja hyödyntävää ratkaisua tai solmujen välisiä voimia voidaan laskea solmujen välisien reittien pituuksien mukaan. (Kobourov 2013, 383.)

Esimerkkinä sähkövarauksen omaista tekniikkaa hyödyntävästä algoritmista on esimerkiksi Fruchtermanin ja Reingoldin (1991) esittämä algoritmi, joka on myös käytössä opinnäytetyön käytännön työssä. Näissä metodeissa jokaisen solmun välillä on toisiaan hylkivä voima, ja yhdistettyjen solmujen välillä on vetovoima. Toisin sanottuna yksittäinen solmu liikkuu kauemmas kaikista muista solmuista, ja kohti kyseisen solmun naapureita laskettujen voimien perusteella. Algoritmin yhdessä iteraatiossa lasketaan solmun liike ja suunta. Näistä voimista lasketaan energia, jonka perusteella algoritmi tietää jatketaanko laskentaa. Energiaa voidaan laskea joko koko graafista, pienemmistä joukoista tai yksittäisitä solmuista. Algoritmi jatkaa laskentaa, kunnes se pääsee tasapainotettuun tilaan. Tasapainotetulla tilalla tarkoitetaan, kun kaikki voimat sulkevat toisensa pois, eikä liikettä enää tapahdu. Yleisesti tasapainotettuun tilaan ei kuitenkaan päästä järkevässä ajassa, ja algoritmi pyrkii pääsemään lähelle sitä.

Voimasuunnatulla algoritmilla saa aikaiseksi selkeitä verkkoja suhteellisen helposti, mutta se on kuitenkin laskennallisesti yksi raskaimmista vaihtoehdoista. Tämän tyyppisen algoritmin suosion vuoksi sitä on kuitenkin kehitetty ja tutkittu huomattavissa määrin ja suorituskykyongelmia on pyritty poistamaan. Voimasuunnattujen algoritmien tuloksia on myös vaikea ennustaa ja algoritmin käyttö samaan dataan useaan otteeseen luo erilaisia tuloksia. (Ebert ym. 2012, 153.)

5 CASE: SÄHKÖPOSTIVERKOSTON VISUALISOINTI

Tehtävänä oli luoda ohjelma, joka visualisoi sähköpostiverkoston 3D-ympäristöön. Opinnäytetyön toimeksiantajana toimi Digitalia. Digitalia on Kaakkois-Suomen ammattikorkeakoulun, Helsingin yliopiston ja Kansalliskirjaston yhteinen digitaalisen tiedonhallinnan tutkimuskeskus, joka perustettiin vuonna 2015. Tavoitteena ei ollut luoda valmista ohjelmaa, vaan versio, jossa on tarvittavat ominaisuudet ohjelman toiminnan kannalta. Mitään erityisiä vaatimuksia ohjelman toiminnallisuudelle ei asetettu, eli käytetyt työkalut ja toteutus olivat vapaasti päätettävissä. Virtuaalitodellisuuslasien tukea toivottiin, mutta se ei ollut vaatimuksena. Toimeksianto oli melko kokeellinen Digitalian kannalta, sillä suoranaista tarvetta tällaiselle ohjelmistolle ei ollut. Digitalialla oli samaan aikaan tekeillä sovellus, joka generoi csv-tiedostoja sähköpostiviesteistä. Tällä toimeksiannolla ehkä haluttiin katsoa, voisiko sähköpostiverkoston visualisoinnista olla hyötyä datan analysoinnissa.

Projektin pääasialliseksi alustaksi valikoitui Unity3D-pelimoottori. Visualisointiin tarkoitettu työkalu voi olla hyvinkin pelin omainen, joten pelimoottori soveltuu sen kehittämiseen luontevasti. Unity3D-pelimoottori tuntui hyvältä vaihtoehdolta, sillä sen käytöstä oli aiempaa kokemusta. Myös Unreal Engine pelimoottori olisi ollut hyvä valinta, mutta siitä ei kuitenkaan löytynyt aiempaa kokemusta ja myös sen käyttämä C++-ohjelmointikieli oli vieras. Tämän takia se tuntui turhan haastavalta ja olisi vaatinut huomattavat määrät uusien asioiden opiskelua.

5.1 Suunnitelma

Suunnitelmana toimi tekstidokumentti jossa oli suurpiirteisesti listattu ominaisuuksia ja asioita, joiden ajateltiin olevan tärkeitä ohjelman toiminnan kannalta. Tätä dokumenttia ei taidettu avata kertaakaan sen luomisen jälkeen, eli käytännössä suunnitelmaa ei ollut. Tämä niin sanotun suunnitelman kirjoittaminen toimi enemmän ideointityökaluna kuin varsinaisena suunnitelmana, jota noudatettaisiin ohjelmaa tehdessä.

Kattavan suunnitelman tekeminen olisi varmasti auttanut säästämään aikaa ohjelman tekovaiheessa. Useita osioita ohjelmasta jouduttiin kirjoittamaan

useaan otteeseen erinäisten muutosten takia ohjelman muihin osiin tai kokeilumielessä. Juuri tätä ongelmaa pyritään välttämään kirjoittamalla kattava suunnitelma ennen ohjelmoinnin aloittamista.

Suunnitelman kirjoittaminen on kuitenkin haastavaa, jos sillä hetkellä työn alla olevaa ohjelmaa, tai mitään sen kaltaista, ei ole aikaisemmin tehnyt. Suunnitellessa täytyy kuitenkin tietää ainakin osittain, minkälaisia tekniikoita tullaan käyttämään ja miten ohjelma toimii yleisellä tasolla. Tämän projektin kohdalla opiskeltiin sitä mukaa, kun implementoitiin uusia asioita. Tämän takia suunnitelman teko olisi vaatinut huomattavaa määrää työtä ja tutkintaa. Suunnittelu jäikin erittäin suurpiirteiseksi eikä siitä ollut hirveästi hyötyä itse ohjelman toteutuksen aikana.

5.2 Käyttöliittymä ja ulkoasu

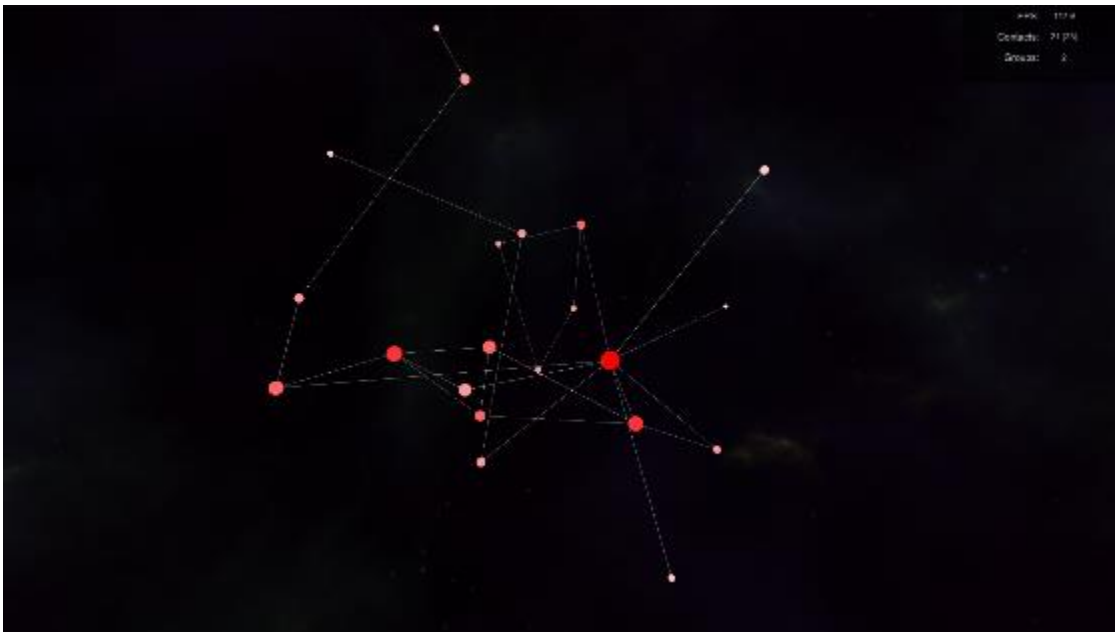
Ohjelmassa on kaksi skeneä, joista ensimmäinen toimii alkuvalikkona ja toisessa näytetään itse visualisointi. Alkuvalikossa määritetään polku csv-tiedostolle, josta haetaan visualisoitava data. Näkymässä on myös joukko asetuksia, joita voi muuttaa haluamallaan tavalla. Tietysti alkuvalikosta löytyy myös nappi, jota painamalla käynnistetään visualisointi.

Kuva 7. Alkuvalikko

Asetuksista voi säätää rajallisesti visualisoinnin ulkoasua värien ja koon mukaan. Solmujen väliset yhteydet voi ottaa pois käytöstä visualisoinnin ajaksi,

jolloin viivat piirretään vasta visualisoinnin ollessa valmis, tai jos se pysäytetään. Viivojen kokoa on myös mahdollista muuttaa. Hylkyvoiman (Repulsion) ja vetovoiman (Attraction) asetuksilla pystyy vaikuttamaan ohjelman laskemiin voimiin solmujen välillä, joka vaikuttaa paljonko solmujen välillä on välimatkaa visualisoinnin lopussa. VR-tuen käyttöönotto ja kitkan määrän säätö eivät tee tällä hetkellä mitään.

Painamalla visualisoi-nappia ohjelma siirtyy toiseen skeneen, jossa itse visualisointi suoritetaan. Käyttäjä voi halutessaan pysäyttää ja jatkaa visualisointia missä vaiheessa tahansa. Käyttäjällä on myös mahdollisuus palata takaisin alkuvalikkoon.



Kuva 8. Vaiheessa oleva visualisointi

Graafisesti visualisointi on hyvin yksinkertainen. Solmut kuvataan primitiivisillä objekteilla palloina, ja solmujen yhteyksiä kuvataan viivoilla. Solmuissa olevalla värillä kuvataan yhteyksien määrää; voimakkaampi väri tarkoittaa enemmän yhteyksiä verrattain muihin verkon solmuihin. Oikeassa yläkulmassa näkyy sen hetkinen kuvataajuus, montako solmua visualisoinnissa on mukana (sulkeissa näytetään solmujen kokonaismäärä luetusta datasta) ja montako joukkoa luetusta datasta luotiin. Joukkojen ja solmujen kokonaismäärät näytetään, sillä ohjelma visualisoi vain luetusta datasta luoduista verkoista suurimman, jättäen loput datasta huomioimatta.

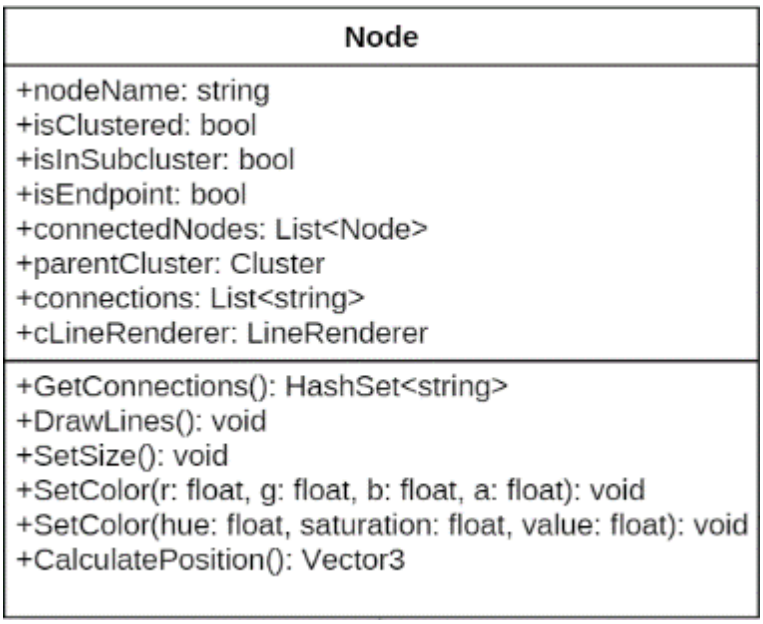
5.3 Ohjelman tärkeimmät osat

Tässä osiossa käydään läpi ohjelman oleellisimmat osat. Näistä osista kerrotaan mihin niitä tarvitaan ja niiden tärkeimmät ominaisuudet käydään läpi. Tässä mainitut asiat isolta osin sivuutetaan luvussa 5.4, jossa kerrotaan ohjelman toiminta vaihe kerrallaan. Tätä siis kannattaa referoida mikäli jokin ohjelman osa tuntuu epäselvältä.

5.3.1 Rakenteelliset luokat

Node-, Cluster- ja Subcluster-nimiset luokat toimivat luokkina, jotka pitävät visualisoitavan datan tiedot sisällään ja joiden avulla se kategorioidaan eri osiin. Luokissa on myös sisällään toiminnallisuutta, eli niitä ei ole tehty pelkästään datan säilytykseen. Nämä luokat ovat myös sidoksissa pelissä oleviin peliobjekteihin. Niitä ei siis voi luoda manuaalisesti, vaan luokan instanssi luodaan sille tarkoitetun peliobjektin luonnin yhteydessä automaattisesti. Instanssi myös tuhoutuu automaattisesti, mikäli siihen liitetty peliobjekti tuhotaan.

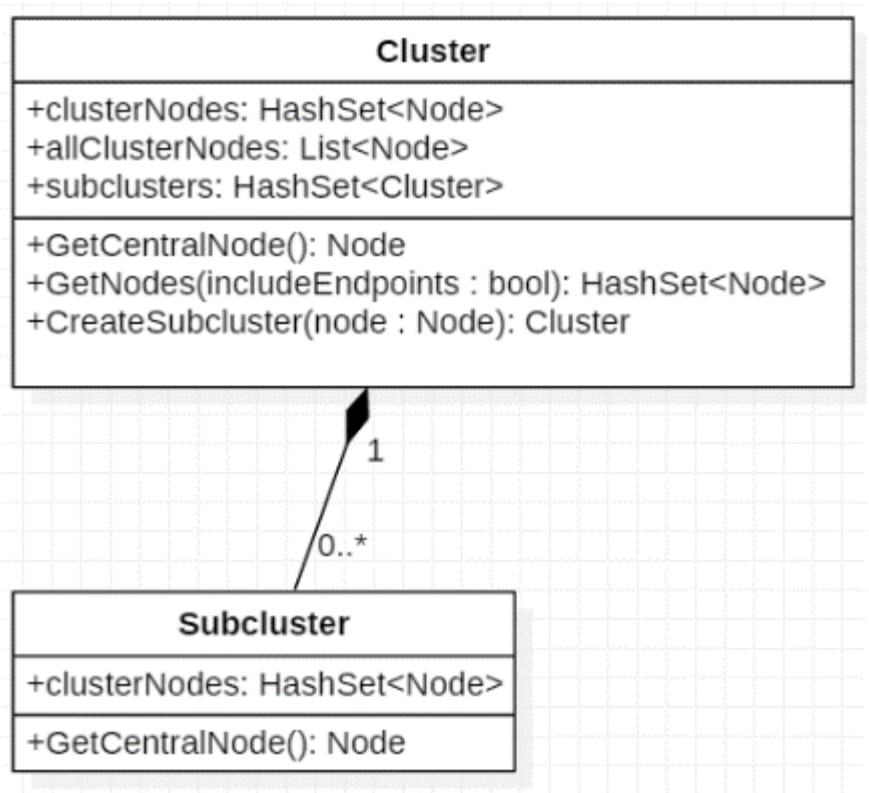
Node-luokalla kuvataan verkon yhtä solmua. Luokka pitää sisällään pääasiassa tietoja siitä, millainen solmu on kyseessä ja mitkä muut solmut ovat sen naapureita.



Kuva 9. Node-luokan rakenne

Solmun graafiseen toiminnallisuuteen kuuluu värin ja koon muutos, sekä näiden lisäksi viivojen piirto yhdistyneihin solmuihin. GetConnections-metodia käytetään solmujen luonnin jälkeen, ja sen avulla yhdistyneiden solmujen nimet saadaan varmasti ilman duplikaatteja. CalculatePosition-metodilla lasketaan kyseisen solmun päivitetty positio voimien avulla muihin solmuihin nähden. CalculatePosition-metodia käytetään ainoastaan silloin, kun GPGPU ei ole käytössä.

Cluster-luokalla kuvataan solmujen joukkoa, jotka ovat yhdistyneet toisiinsa. Toisin sanoen yksi Cluster pitää sisällään kokonaisen verkon, jossa jokainen solmu on yhdistynyt kaikkiin muihin verkon solmuihin suorasti tai epäsuorasti. Luokka pitää kirjaa siihen kuuluvista solmuista ja sen sisällä olevista alajoukoista. Alajoukolla on myös oma luokkansa nimeltä Subcluster. Tällaiseen alajoukkoon kuuluu aina yksi normaali solmu ja tähän solmuun yhdistyneet lehdet. Lehdellä tarkoitetaan solmua, jonka asteluku on 1.



Kuva 10. Cluster- ja Subcluster-luokan rakenne

Kuvasta 10 huomaa, että luokat ovat hyvin samankaltaisia keskenään. Molemmilla luokilla on tieto mitä solmuja ne pitävät sisällään. Cluster-luokalla on myös lista, jossa säilytetään omien solmujen lisäksi myös alajoukoissa olevia

solmuja. Molemmat luokat osaavat kertoa keskeisimmän solmun, eli solmun jolla on suurin asteluku, käyttämällä GetCentralNode-metodia. Cluster-luokalla on myös toiminnallisuus luoda uusia alajoukkoja tarvittaessa.

Edellä mainittujen luokkien hallinnointia hoitaa ClusterManager- ja Node-Manager-luokat. Nämä pitävät sisällään toiminnallisuuden tehdä operaatioita kaikille, tai suurelle osalle Nodeista ja Clustereista. Manager-luokat pitävät tallessa referenssejä Nodeille ja Clustereille josta ne ovat helposti saatavilla esimerkiksi Visualize-luokalle.

5.3.2 Parseri

Parseri hoitaa visualisoitavan tiedon luvun sille annetusta csv-tiedostosta. Tiedoston täytyy olla tietyssä muodossa, jotta parseri osaa lukea siitä oikeat tiedot. Parseri tehdään sen mukaan, millaisessa muodossa tiedot annettaan Digitalian puolelta.

```
1  Id;Lähettäjä;Vastaanottaja(t);Cc;Otsikko;Aika
2  Id;Lähettäjä;Vastaanottaja(t);Cc;Otsikko;Aika
3  Id;Lähettäjä;Vastaanottaja(t);Cc;Otsikko;Aika
4  Ja niin edelleen...
```

Kuva 11. Havainnollistava kuva csv-tiedon formaatista

Ohjelman kannalta oleelliset tiedot ovat sähköpostin lähettäjä ja sen vastaanottajat. Muita kuvassa 11 näkyviä tietoja ei oteta talteen, sillä niitä ei käytetä missään eikä ne ole ohjelman toiminnan kannalta tarpeellisia. Tämä voidaan kuitenkin helposti muuttaa tulevaisuudessa, mikäli siihen on tarvetta.

Parseri avaa tiedoston annetusta polusta ja lukee sitä yksi rivi kerrallaan. Nämä toiminnot suoritetaan FileStream- ja StreamReader-luokkia apuna käyttäen.

```
17      public static IEnumerable<List<string>> ReadCsv(string path)
18      {
19          List<string> csvLine = new List<string>();
20          string mailFrom = "";
21          List<string> mailsTo = new List<string>();
22
23          using (var fileStream = File.OpenRead(path))
24          using (var streamReader = new StreamReader(fileStream, Encoding.UTF8, true, 1024))
25          {
26              // One loop per line from csv
27              string line;
28              while ((line = streamReader.ReadLine()) != null)
29              {
30                  string[] values = line.Split(';');
31                  mailFrom = values[1]; // Skip index 0 since that info (id) isn't needed
32                  mailsTo = values[2].Split(',').ToList();
33                  csvLine.Clear(); // Clear previous line from the list before adding current line
34                  csvLine.Add(mailFrom);
35                  csvLine = csvLine.Concat(mailsTo).ToList();
36                  yield return csvLine;
37              }
38          }
39      }
```

Kuva 12. Parser-luokan ReadCsv metodi

Luetun tiedoston rivit käydään läpi while-silmukassa yksi rivi kerrallaan, jossa rivi pilkotaan pienempiin osiin csv-tiedoston erottimen mukaan, joka on tässä tapauksessa puolipiste. Erottelun jälkeen tarvittavat tiedot otetaan talteen listaan, joka palautetaan ”yield return” lausekkeella. Tästä lausekkeesta, sekä FileStream- ja StreamReader-luokista voi lukea tarkempaa tietoa Microsoftin .NET- ja C#-dokumentaatiosta.

5.3.3 Laskentavarjostin

Laskentavarjostimella (Compute Shader) hoidetaan ohjelman raskain laskenta, eli repulsiovoimien laskeminen. Ohjelman algoritmissa jokaiselle solmulle täytyy laskea repulsiovoima jokaisen muun solmun välillä. Laskentavarjostimessa määritetään muuttujat repulsiolle ja paljonko aikaa kului edellisen kuvan (frame) valmistumiseen. Varjostimessa määritetään myös Struct-tyyppinen rakenne, jota varjostimen puskuri käyttää. Puskurin avulla tietoa siirretään näytönohjaimen ja prosessorin välillä.

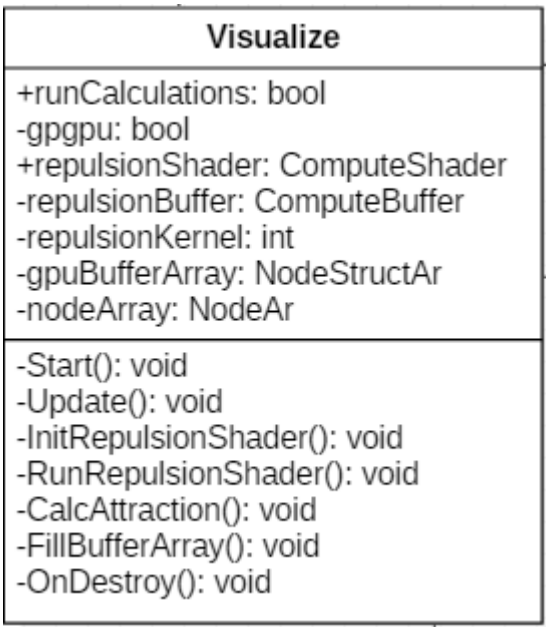
```
2     #pragma kernel CSRepulsion
3
4     float deltaTime;
5     float repulsion;
6
7     struct ContactStruct
8     {
9         float3 position;
10        float3 velocity;
11    };
12
13    RWStructuredBuffer<ContactStruct> bufContacts;
14
15    [numthreads(128, 1, 1)]
16    void CSRepulsion(uint3 id : SV_DispatchThreadID) { ... }
```

Kuva 13. Laskentavarjostin

Laskentavarjostin koostuu yhdestä kernelistä, joka määrittää säiejoukon koostumaan 128 säikeestä yksiulotteisen taulukon muodossa. Kernel pitää sisällään laskennan jota ei käydä sen tarkemmin läpi. Tiivistetysti kerrottuna kernel laskee repulsiovoimat ja yhdistää ne puskurin kautta sille annettujen vetovoimien kanssa joiden avulla se laskee solmuille uudet positiot ja tallentaa ne puskuriin.

5.3.4 Visualisoinnin hoitava luokka

Ohjelman aivoina toimii Visualize-luokka, joka ajetaan siirryttäessä alkuvalikosta visualisointi sceneen. Luokka hallinnoi visualisointiin liittyviä laskutoimituksia ja siihen liittyviä alustuksia. Luokka käyttää suurta osaa ohjelman muista luokista hyödykseen, kuten liitteessä 1 olevasta uml-kaaviosta näkee. Luokan pääasiallisena tehtävänä on siis hallinnoida ohjelman toimintaa.



Kuva 14. Visualize-luokan rakenne

Luokassa oleva staattinen runCalculations-muuttuja määrittää milloin visualisointia suoritetaan ja milloin ei. Tämän muuttujan avulla käyttäjä voi halutessaan pysäyttää ja jatkaa visualisointia saumattomasti. Kaikki luokan tekemä työ tapahtuu Start-, Update- ja OnDestroy-metodeissa, jotka ovat Unity3D:n määrittämiä. Start-metodi hoitaa tarvittavat alustukset visualisoinnin aloittamiseksi. Update-metodissa suoritetaan vaadittava laskenta visualisoinnille, eli solmujen positioiden laskenta. OnDestroy-metodissa hoidetaan näytönohjaimen käytössä olleiden resurssien vapauttaminen. Muut määritetyt metodit toimivat apumetodeina, jotka auttavat pilkkomaan koodia osiin sen selkeyttämiseksi.

5.4 Ohjelman toiminta

Tässä luvussa käydään läpi ohjelman toiminta askel kerrallaan alusta loppuun. Muutamia ohjelman osia ei käydä tarkemmin läpi, jotka eivät ole toiminnan kannalta hirveän tärkeitä, jotta osiosta ei tule äärettömän pitkä. Osiossa puhutaan myös Nodeista, Clustereista ja Subclustereista joista voi lukea tarkemmin luvusta 5.3.1.

5.4.1 Toiminta ennen visualisointia

Kun ohjelma käynnistetään, se lataa ensimmäisen scenen, joka koostuu alkuvalikosta (Kuva 7). Tämä scene koostuu pääasiassa kamerasta ja piirtoalu-

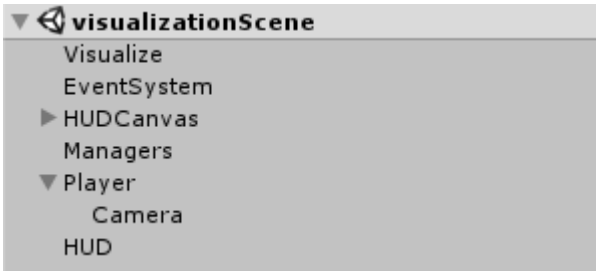
eesta, jolle käyttöliittymä piirretään. Alkuvalikossa käyttäjä määrittää polun visualisoitavalle datalle ja halutessaan muuttaa ohjelman asetuksia. ”Visualize”-nappia painettaessa ohjelma lukee käyttäjän syöttämän polun ja asetukset talteen Settings-luokkaan.

```
26      /// <summary> Reads all the information from start-menu and loads visualization scene.
29      public void OnVisualizeButton()
30      {
31          Settings.CsvPath = csvField.text;
32          Settings.EnableVr = enableVrToggle.isOn;
33          Settings.RelativeColoring = relativeColorToggle.isOn;
34          Settings.Hue = hueSlider.value; // hue value between 0...1
35          Settings.MaxSize = float.Parse(maxSizeField.text);
36          Settings.MinSize = float.Parse(minSizeField.text);
37          Settings.Increment = float.Parse(incrementField.text); // only used if relative sizing is false
38          Settings.RelativeSizing = relativeSizeToggle.isOn;
39          Settings.ConnectionLineWidth = float.Parse(lineWidthField.text);
40          Settings.DrawLines = drawLinesToggle.isOn;
41          Settings.Attraction = float.Parse(attractionField.text);
42          Settings.Repulsion = float.Parse(repulsionField.text);
43          Settings.Friction = float.Parse(frictionField.text);
44
45          SceneManager.LoadScene("visualizationScene");
46      }
```

Kuva 15. ”Visualize” nappia painettaessa ajettava metodi

Tämän jälkeen ladataan visualisointi-scene, jossa asetuksia käytetään. Settings-luokka tallentaa asetukset staattisiin muuttujiin, joita voidaan käyttää ohjelman missä tahansa luokassa helposti ilman uusien instanssien luomista.

Visualisointi-scene koostuu peliobjekteihin liitetyistä luokista, pelaajasta jonka sisällä on kamera ja piirtoalueesta jossa näytetään tietoja visualisoinnista. Pelaaja on vain tyhjä peliobjekti, jolla ei ole fyysistä muotoa.



Kuva 16. Scenen rakenne

Näiden lisäksi sceneen luodaan myös peliobjektit Nodeille, Clustereille ja Subclustereille ajon aikana. Nämä peliobjektit luodaan Prefab-objektien avulla, joissa on määritelty mitä komponentteja ne pitää sisällään.

Scenen latauksen jälkeen ohjelma siirtyy suorittamaan scenessä olevien luokkien Awake-metodeja. Tässä tapauksessa ainoat tällaiset metodit löytyvät

NodeManager- ja ClusterManager-luokista, jossa suoritetaan niiden kannalta tarpeelliset alustukset. Näiden jälkeen suoritetaan Start-metodit, joita löytyy käyttäjän kontrolleja ohjaavasta luokasta ja visualisoinnin hoitavasta Visualize-luokasta. Mainituista ensimmäinen ei ole oleellinen, mutta jälkimmäisessä hoidetaan datan konversio tekstistä luokiksi, joka käydään läpi seuraavaksi.

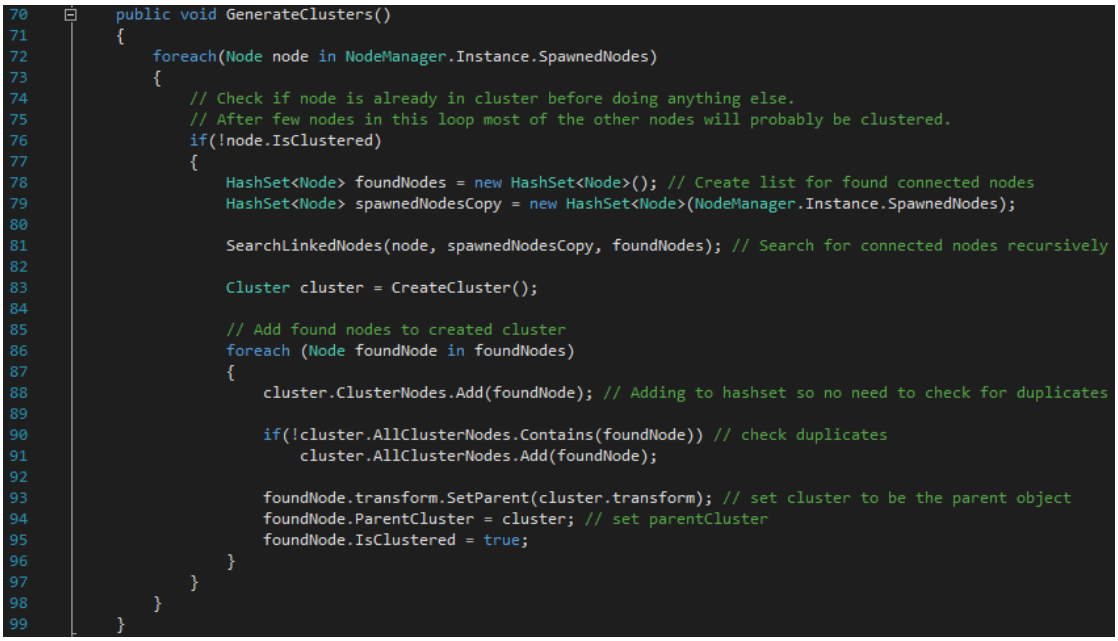
Ensimmäisenä luodaan Nodet eli solmut. Käyttäjän antama csv-tiedosto luetaan parserin avulla ja sen antamat tiedot annetaan rivi kerrallaan NodeManager-luokan SpawnNodes-metodille. Tämä metodi luo peliobjektit luetusta datasta CreateNode-metodin avulla ja tallentaa kaaret solmujen välillä nimen mukaan string-tyyppisenä.

```
192     public void SpawnNodes(List<string> connections)
193     {
194         for(int i = 0; i < connections.Count; i++)
195         {
196             Node spawnedNode = CreateNode(connections[i]);
197             if (spawnedNode != null)
198             {
199                 if (i == 0)
200                 {
201                     for (int j = 0; j < connections.Count; j++)
202                     {
203                         // Add receivers to senders list of connections
204                         if (i != j)
205                             spawnedNode.Connections.Add(connections[j]);
206                     }
207                 }
208                 else
209                 {
210                     // Add sender to receivers list of connections
211                     spawnedNode.Connections.Add(connections[0]);
212                 }
213             }
214         }
215     }
216 }
```

Kuva 17. SpawnNodes-metodi

CreateNode-metodi ottaa parametrina string-tyyppisen arvon, joka on Noden nimi. Mikäli tällä nimellä on jo luotu Node, niin metodi palauttaa sen tekemättä duplikaattia. Kuten kuvasta 17 näkyy, tässä vaiheessa kullekin Nodelle ei anneta referenssejä muihin Nodeihin joihin ne ovat yhteydessä kaarella. Tämä hoidetaan Nodejen luonnin jälkeen NodeManager-luokan SetConnectedNodes-metodilla, joka käy kaikki Nodet läpi ja tallentaa referenssit tässä (Kuva 12) käytetyn Connections-listan avulla.

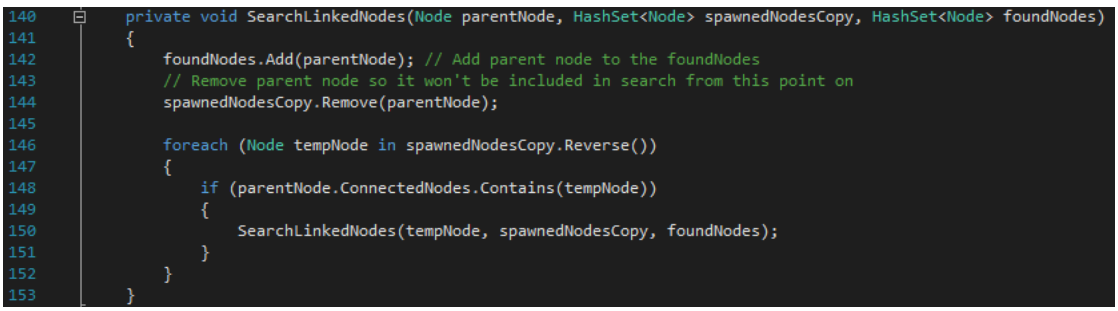
Tässä vaiheessa kaikki Nodet on luotu, mutta niitä ei ole vielä jaoteltu miten-kään. Seuraavaksi luodaan Clusterit eli verkot. Clustereiden luontiin käytetään ClusterManager-luokan GenerateClusters-metodia, joka etsii ja luo datasta yhtenäiset verkot, eli verkot joissa kaikkien solmujen asteluku on suurempi kuin nolla.



```
70    public void GenerateClusters()
71    {
72        foreach(Node node in NodeManager.Instance.SpawnedNodes)
73        {
74            // Check if node is already in cluster before doing anything else.
75            // After few nodes in this loop most of the other nodes will probably be clustered.
76            if(!node.IsClustered)
77            {
78                HashSet<Node> foundNodes = new HashSet<Node>(); // Create list for found connected nodes
79                HashSet<Node> spawnedNodesCopy = new HashSet<Node>(NodeManager.Instance.SpawnedNodes);
80
81                SearchLinkedNodes(node, spawnedNodesCopy, foundNodes); // Search for connected nodes recursively
82
83                Cluster cluster = CreateCluster();
84
85                // Add found nodes to created cluster
86                foreach (Node foundNode in foundNodes)
87                {
88                    cluster.ClusterNodes.Add(foundNode); // Adding to hashset so no need to check for duplicates
89
90                    if(!cluster.AllClusterNodes.Contains(foundNode)) // check duplicates
91                        cluster.AllClusterNodes.Add(foundNode);
92
93                    foundNode.transform.SetParent(cluster.transform); // set cluster to be the parent object
94                    foundNode.ParentCluster = cluster; // set parentCluster
95                    foundNode.IsClustered = true;
96                }
97            }
98        }
99    }
```

Kuva 18. GenerateClusters-metodi

Metodi käy kaikki Nodet läpi silmukan avulla. Ensimmäisenä se tarkastaa on-ko Node jo laitettu johonkin Clusteriin, sillä jo ensimmäisen Noden jälkeen iso osa tai kaikki silmukassa läpi käytävät Nodet saattavat olla laitettu Clusteriin. Seuraavaksi metodi luo tarvittavat kokoelmat SearchLinkedNodes-metodia varten ja ajaa sen. SearchLinkedNodes-metodi etsii rekursiivisesti kaikki sille annettun Noden naapurit ja laittaa ne foundNodes-kokoelmaan.



```
140   private void SearchLinkedNodes(Node parentNode, HashSet<Node> spawnedNodesCopy, HashSet<Node> foundNodes)
141   {
142       foundNodes.Add(parentNode); // Add parent node to the foundNodes
143       // Remove parent node so it won't be included in search from this point on
144       spawnedNodesCopy.Remove(parentNode);
145
146       foreach (Node tempNode in spawnedNodesCopy.Reverse())
147       {
148           if (parentNode.ConnectedNodes.Contains(tempNode))
149           {
150               SearchLinkedNodes(tempNode, spawnedNodesCopy, foundNodes);
151           }
152       }
153   }
```

Kuva 19. SearchLinkedNodes-metodi

SearchLinkedNodes-metodi käy läpi sille parametrina annetun ”spawnedNo-deCopy”-kokoelman, joka sisältää kaikki Nodet, joille ei ole vielä löydetty Clus-

teria. Kokoelmassa olevia Nodeja verrataan parametrina annetun ”parentNode”-Noden naapureihin. Mikäli jokin Nodeista täsmää, ajetaan SearchLinkedNodes-metodi uudestaan antamalla löydetty Node ensimmäisenä parametrina, jolloin rekursio tapahtuu. Kun metodi on kokonaisuudessaan saatu suoritettua, ”foundNodes”-kokoelmassa on yhden Clusterin kaikki Nodet.

Rekursiivisen metodin jälkeen GenerateClusters-metodi (Kuva 18) luo tyhjän Clusterin CreateCluster-metodin avulla. Tämän jälkeen Clusteriin löydetyt Nodet käydään läpi silmukassa, jossa ne lisätään Clusteriin ja niiden tiedot päivitetään Clusteriin liittymisen osalta.

Seuraavaksi ohjelma etsii, mitkä Nodeista ovat lehtiä, eli solmuja joilla on vain yksi kaari. Tätä tietoa tarvitaan pääasiassa Subclustereiden luomiseen. Ohjelman tämän hetkisessä versiossa Subclustereita ei kuitenkaan käytetä hyödyksi, joten niiden luomista ei käydä tarkemmin läpi. Lyhyesti selitettynä, ohjelma käy kaikki Nodet läpi, jotka eivät ole lehtiä. Näiden Nodejen kohdalla ohjelma tarkastaa, onko kyseisen Noden naapureina lehtiä ja jos on, tästä Nodesta ja siihen yhdistyneistä lehdistä muodostuu Subcluster.

Kun Nodet ovat jaoteltu Clustereihin ja Subclustereihin, määritellään niiden ulkonäkö ja tehdään muutamia valmistelevia toimenpiteitä visualisointia varten. Nodejen väri ja koko määrittyy niiden kaarien lukumäärän mukaan. Mitä enemmän kaaria, sen suurempi ja väriltään vahvempi Node. Tämän jälkeen etsitään Cluster, jossa on eniten Nodeja. Kaikki muut Clusterit (ja niiden sisällä olevat Nodet) otetaan pois käytöstä, eli niitä ei visualisoida.

Viimeiseksi hoidetaan muuttujien alustuksia ja suurimman Clusterin kaikille Nodeille luodaan satunnainen positio, jottei Nodet ole päällekkäin visualisoinnin alkaessa. Myös laskentavarjostimen tarvittavat alustukset hoidetaan. Kun nämä ovat tehty, Start-metodi on suoritettu loppuun ja ohjelma on valmis aloittamaan visualisoinnin.

5.4.2 Itse visualisointi

Kun Start-metodit ovat suoritettu, Unity3D alkaa suorittaa Update-metodeja. Visualisoinnin olennaisin osa, eli verkon solmujen paikkojen laskeminen, suo-

ritetaan Visualize-luokan Update-metodissa. Update-metodi ajetaan joka kuvan (frame) aikana ja tämän avulla visualisointiin saadaan sulava liike siirtämällä solmuja Update-metodin lopussa.

```
154         if (runCalculations && gpgpu) {
155             CalcAttraction(); // Calculate attraction and save results to gpuBufferArray
156 
157             repulsionBuffer.SetData(gpuBufferArray); // Update data for the buffer (gpu)
158 
159             RunRepulsionShader(); // Run the shader
160 
161             repulsionBuffer.GetData(gpuBufferArray); // Get data from the buffer (gpu)
162 
163             for (int i = 0; i < nodeArray.Length; i++)
164             {
165                 nodeArray[i].transform.position += gpuBufferArray[i].position; // set the position
166 
167                 if (Settings.DrawLines)
168                     nodeArray[i].DrawLines();
169             }
170         }
```

Kuva 20. Visualize-luokan Update-metodin oleellinen osa

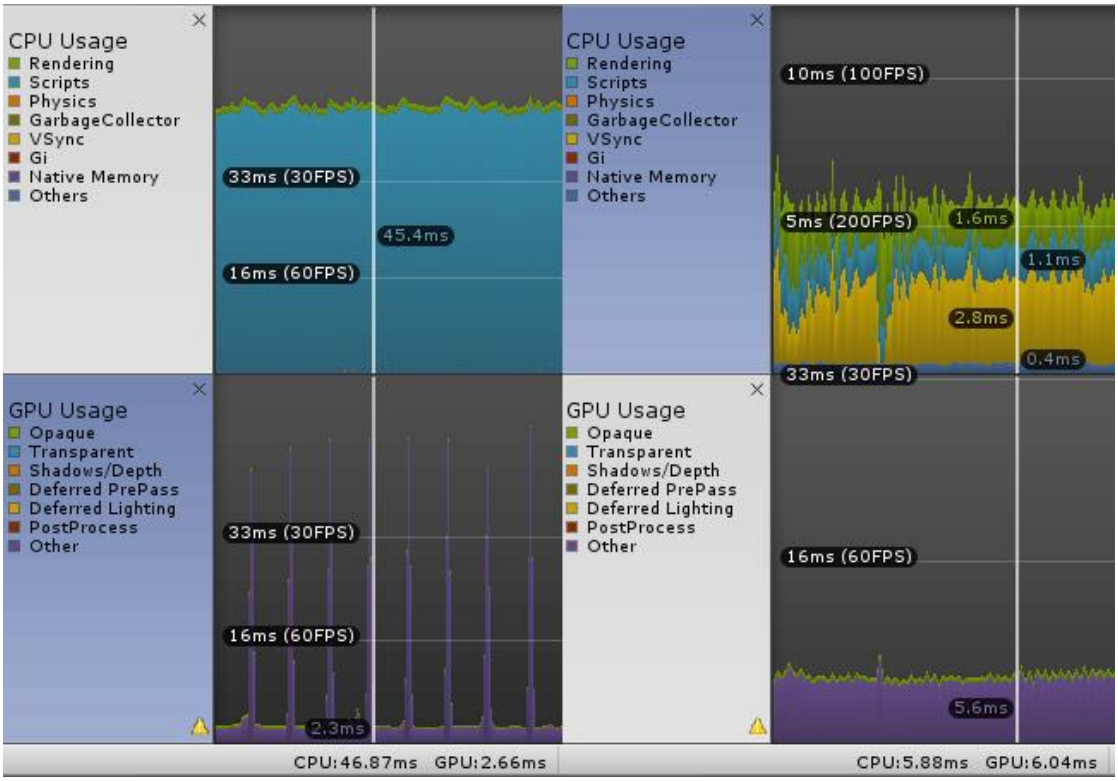
Kuvassa 20 näytetään Update-metodin oleellinen osa, jossa lasketaan solmujen positiot silloin, kun GPGPU on käytössä. Tätä osiota ajetaan niin kauan, kun käyttäjä sammuttaa ohjelman tai pysäyttää visualisoinnin. Ensimmäisenä kutsutaan CalcAttraction-metodia, joka laskee solmujen väliset vetovoimat jokaiselle solmulle ja tallentaa ne taulukkoon NodeStruct-tyyppisinä objekteina. Vetovoiman laskentaan käytetään pääasiassa solmujen sen hetkistä sijaintia ja käyttäjän asettamaa vetovoima-arvoa.

Tämän jälkeen yllä mainittu taulukko annetaan laskentavarjostimen puskurille, jonka jälkeen ajetaan RunRepulsionShader-metodi, joka käynnistää laskentavarjostimen. Laskentavarjostin laskee repulsiovoiman kaikkien solmujen välillä ja tämän jälkeen yhdistää tulokset sille annetun taulukon vetovoimien kanssa. Näiden tuloksien avulla se laskee solmujen uudet positiot ja tallentaa ne annettuun taulukkoon.

Seuraavana tiedot haetaan näytönohjaimen puskurista. Kaikki solmut käydään läpi ja niille annetaan päivitetyt tiedot sijainnista johon Unity3D siirtää ne pelimaailmassa. Mikäli viivojen piirto on käytössä, sijaintien päivittämisen yhteydessä myös viivat piirretään yhdistyneiden solmujen välille DrawLines-metodia käyttäen.

5.5 Ongelmat ja jatkokehitys

Ensiksi oli vaikeuksia saada visualisointia edes ylipäänsä toimimaan, mutta suurin ongelma tuli ohjelman suorituskyvyn suhteen. Alkuun ohjelma laski visualisointia vain prosessorin avulla ja ainoastaan yhdellä säikeellä. Tämän lisäksi ohjelma ei käyttänyt mitään hienoa algoritmia, jolla laskennan määrää olisi saanut pienennettyä. Tämä johti siihen, että visualisointi toimi hyväksyttävällä kuvataajuudella maksimissaan sadan solmun kokoisella verkolla. Asian tutkinnan jälkeen, löytyi mahdollinen ratkaisu, GPGPU. Alkuun ei ollut kuitenkaan ollenkaan varmaa toimisiko löydetty ratkaisu, mutta kokeilujen jälkeen kävi selväksi, että tekniikka toimii varsin hyvin verkon visualisoinnissa.



Kuva 21. Suorituskykyvertailu ilman GPGPU-tekniikkaa (vasen) ja sen ollessa päällä (oikea)

Kuten kuvasta 21 näkyy, suorituskyky nousi huimasti ottamalla GPGPU-tekniikan käyttöön. Kuvassa näkyvä testi tehtiin 268 solmun kokoisella verkolla. Keskimääräinen aika yhden kuvan (frame) piirtoaika oli pelkällä prosessorilla noin 47 millisekuntia kun taas näytönohjaimen avustamana lukema oli noin 6 millisekuntia. Kuvataajuuden muodossa nämä lukemat ovat noin 20 (47 ms) ja noin 160 (6 ms) kuvaa sekunnissa. Testissä kummallakin tekniikalla käytettiin samaa algoritmia.

Tietysti yllä oleva vertailu ei ole reilu koska prosessorilla pystyisi hyödyntämään erinäisiä monimutkaisia algoritmeja, joilla suorituskykyä saisi nostettua. Myöskään GPGPU-tekniikkaa varten kirjoitettu laskentavarjostin ei ole täysin optimoitu ja siitä saisi vielä enemmän tehoa irti. Testillä lähinnä pyritään näyttämään, kuinka tässä tapauksessa näytönohjainta hyödyntämällä päästiin tyydyttäviin tuloksiin suhteellisen helposti.

Ohjelman visuaalisen puolen kehitys jäi melko vähälle ja siinä olisi huomattavasti kehityksen varaa. Visualisoinnista saisi huomattavasti hienomman näköisen, lisäämällä esimerkiksi valoefektejä ja varjoja. Myös koko visualisoinnin voisi näyttää kokonaan eri tavalla, sillä nykyinen atomimallin omainen ratkaisu on melko tylsä, vaikkakin selkeä ja toimiva. Aloitusvalikko on myös vain nopeasti tehty viritys, joka vaatisi vielä mietintää. Käyttäjälle voisi myös näyttää enemmän tietoa solmuista ja tällä hetkellä ylimääräistä tietoa voisi käyttää visualisoinnissa. Esimerkiksi solmujen nimet olisi hyvä näyttää tavalla tai toisella ja sähköpostien aikaleimoja voisi hyödyntää.

Tekniseltä puolelta löytyy monia asioita joita voisi vielä hioa tai joita ei keretty toteuttaa ollenkaan. Suorituskyvyn suhteen parannuksia voisi tehdä laskentavarjostimen optimointiin ja prosessorin tehoja voisi hyödyntää paremmin käyttämällä useaa säiettä laskennassa. Myös ohjelman algoritmia solmujen visualisointiin voisi parantaa, varsinkin prosessorin laskennan osalta.

6 PÄÄTÄNTÖ

Opinnäytetyöllä ei ollut selkeitä tavoitteita johon olisi tähdätty, sillä toimeksiantaja ei asettanut ohjelmalle sen kummempia vaatimuksia eikä ohjelma ollut menossa suoraan käyttöön, vaan se tehtiin kokeilumielessä. Tämän takia ohjelma oli valmis silloin, kun se tuntui itselleni tarpeeksi valmiilta. Tietysti Digitalian puolelta täytyi saada vihreä valo ohjelman toiminnan suhteen, mutta luulisin että omat asettamani vaatimukset ohjelmalle olivat suuremmat kuin heidän. Pääsin omiin (ja samalla Digitalian) tavoitteisiin tekemällä toimivan ohjelman, jossa on tarvittava toiminnallisuus visualisoinnin kannalta, mutta kuitenkin vielä reilusti tilaa kehitykselle. Tämän perusteella voisin todeta, että tavoitteisiin päästiin.

Itse opinnäytetyön kirjoitus prosessina ei ole ollut itselleni hirveän mieleinen, sillä omat kirjoitustaidot ovat parhaimmillaan keskinkertaista, mutta kirjoittamisen aikana kuitenkin opin hyödyllisiä asioita optimoinnista ja visualisoinnista. Toimeksianto opetti paljon uusia asioita ohjelmoinnin kannalta, sillä en ollut tehnyt mitään tämän tyyppistä aiemmin. Tämän takia myös työn teko vaati hirveästi uusien asioiden opiskelua. Työn teko oli myös yllättävän haasteellista ilman selkeitä tavoitteita, koska ohjelman joutui suunnittelemaan alusta loppuun itse.

Ohjelma on tarkoitus julkaista GitHub-palvelussa avoimena lähdekoodina. Tätä en kerennyt vielä tekemään, sillä haluan vielä viimeistellä joitakin asioita, joita mainitsin luvussa 5.5. Myös koodien kommentoinnin osalta on vielä hieman tekemistä, joten julkaisu saa vielä odottaa.

LÄHTEET

Bryant, R. E. & O'Hallaron, D. R. 2001. Computer Systems: A Programmer's Perspective. PDF-dokumentti. Saatavissa: https://doc.lagout.org/programmation/Computer%20Systems%20-%20A%20Programmers%20Perspective.pdf [viitattu 7.3.2017].

Definition of: compiler. s.a. PCMag. WWW-dokumentti. Saatavissa: http://www.pcmag.com/encyclopedia/term/40105/compiler [viitattu 7.3.2017]

Code Optimization. s.a. PVS-Studio. WWW-dokumentti. Saatavissa: https://www.viva64.com/en/t/0084 [viitattu 7.3.2017]

Mitchell, S. 2003. An Extensive Examination of Data Structures. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/aa289148(VS.71).aspx [viitattu 7.3.2017]

Algorithm in Programming. s.a. Programiz. WWW-dokumentti. Saatavissa: https://www.programiz.com/article/algorithm-programming [viitattu 7.3.2017]

Assembly Language. s.a. Techopedia. WWW-dokumentti. Saatavilla: https://www.techopedia.com/definition/3903/assembly-language [viitattu 7.3.2017]

Mitchell, D. 2014. Premature Optimization. WWW-dokumentti. Saatavilla: http://wiki.c2.com/?PrematureOptimization [viitattu 7.3.2017]

Fast Facts. 2016. Unity Technologies. WWW-dokumentti. Saatavilla: https://unity3d.com/public-relations [viitattu 7.3.2017]

Understanding Automatic Memory Management. 2016. Unity Technologies. WWW-dokumentti. Saatavilla: https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html [viitattu 7.3.2017]

Profiler Class Documentation. 2016. Unity Technologies. WWW-dokumentti. Saatavilla: https://docs.unity3d.com/ScriptReference/Profiling.Profiler.html [viitattu 7.3.2017]

Scali. 2012. Multi-core and multi-threading performance (the multi-core myth?). WWW-dokumentti. Saatavilla: https://scalibq.wordpress.com/2012/06/01/multi-core-and-multi-threading [viitattu 7.3.2017]

Bawaskar, A., Ghorpade, J., Kulkarni, M. & Parande, J. 2012. GPGPU Processing in CUDA Architecture. PDF-dokumentti. Saatavissa: https://arxiv.org/ftp/arxiv/papers/1202/1202.4347.pdf [viitattu 7.3.2017]

GPU Parallelizable Methods. s.a. GPUSS. WWW-dokumentti. Saatavissa: http://www.oxford-man.ox.ac.uk/gpuss/simd.html [viitattu 7.3.2017]

Compute Shaders. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/ComputeShaders.html [viitattu 7.3.2017]

Numthreads. s.a. Microsoft Corporation. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ff471442(v=vs.85).aspx [viitattu 24.3.2017]

ComputeBuffer Class Documentation. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/ScriptReference/ComputeBuffer.html [viitattu 7.3.2017]

Trümpler, S. 2015. Render Hell – Book I. WWW-dokumentti. Saatavissa: https://simonschreibt.de/gat/renderhell-book1 [viitattu 17.3.2017]

Draw call batching. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/DrawCallBatching.html [viitattu 17.3.2017]

Light troubleshooting and performance. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/LightPerformance.html [viitattu 17.3.2017]

Global Illumination. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/GIIntro.html [viitattu 17.3.2017]

Optimizing graphics performance. 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html [viitattu 17.3.2017]

Level of Detail (LOD). 2016. Unity Technologies. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/Manual/LevelOfDetail.html [viitattu 17.3.2017]

Harris, M., Nyland, L. & Prins, J. 2007. GPU Gems 3 – Chapter 31. Fast N-Body Simulation with CUDA. WWW-dokumentti. Saatavissa: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch31.html [viitattu 17.3.2017]

Few, S. 2014. Data Visualization for Human Perception. WWW-dokumentti. Saatavilla: https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/data-visualization-for-human-perception [viitattu 19.3.2017]

Kobourov, S. 2013. Handbook of Graph Drawing and Visualization. PDF-dokumentti. Saatavilla: https://cs.brown.edu/~rt/gdhandbook/chapters/force-directed.pdf [viitattu 19.3.2017]

Pesonen, M. 2013. Verkkoteorian Alkeita. PDF-dokumenti. Saatavilla: http://cs.uef.fi/matematiikka/kurssit/MathematicsVisualizationMedia/CourseMaterial/VerkkoteoriaaSciFestiin2013.pdf [viitattu.1.4.2017]

Graafit. 2007. Tampereen Yliopisto. PDF-dokumentti. Saatavilla: http://www.sis.uta.fi/~tira/lectures/2007/luentomoniste/tira10.pdf [viitattu 1.4.2017]

Vallivaara, V. 2014. Tietoturvallisten verkkojen suunnittelu graafiteorian avulla. Opinnäytetyö. Saatavilla: http://jultika.oulu.fi/files/nbnfioulu-201411121982.pdf [viitattu 1.4.2017]

Multiprocessor Laws. s.a. University of Minnesota Duluth. PDF-dokumentti. Saatavilla: http://www.d.umn.edu/~tkwon/course/5315/HW/MultiprocessorLaws.pdf [viitattu 9.4.2017]

Fruchterman, T. & Reingold, E. 1991. Graph Drawing by Force-directed Placement. PDF-dokumentti. Saatavilla: http://reingold.co/force-directed.pdf [viitattu 9.4.2017]

Explaining FP64 performance on GPUs. 2015. ArrayFire. WWW-dokumentti. Saatavilla: http://arrayfire.com/explaining-fp64-performance-on-gpus/ [viitattu 11.4.2017]

Ebert, A., Keller, P. & Tarawneh, R. 2012. A General Introduction To Graph Visualization Techniques. PDF-dokumentti. Saatavilla: http://drops.dagstuhl.de/opus/volltexte/2012/3748/pdf/13.pdf [viitattu 11.4.2017]

KUVALUETTELO

Liite 1
UML-kaavio ohjelmasta