Opinnäytetyö (AMK)

Tietotekniikka

NTIETS12P

2016

Sami Kangasmaa

GRAAFINEN PELIOHJELMOINTI

– OpenGL ja GLSL-varjostimet

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka

2016 | 47

yliopettaja & dosentti Mika Luimula

Sami Kangasmaa

GRAAFINEN PELIOHJELMOINTI

- OpenGL ja GLSL-varjostimet

Graafisella ohjelmoinnilla tarkoitetaan ohjelmoinnin osa-aluetta, jossa keskitytään käyttäjälle näkyviin asioihin. Graafista ohjelmaa suoritetaan yleensä kehyksissä, joita näytetään käyttäjälle tietyin aikavälein. Tällä hetkellä saatavilla olevia grafiikkarajapintoja ovat OpenGL, DirectX, Vulkan ja Metal. Erilaisia graafisen ohjelmoinnin tekniikoita ovat esimerkiksi HDR-renderöinti, normal mapping ja anti-aliasing. Lowglow'n pelimoottori toteutettiin C++:lla Cocos2d-x-viitekehyksen päälle, joka sisältää kaikki tarvittavat ominaisuudet pelin toteutukseen. GLSL-ohjelmat ovat OpenGL shader -ohjelmia, jotka suoritetaan näytönohjaimella. GLSL-ohjelmilla voidaan toteuttaa erilaisia tehosteita sekä jälkikäsittelyefektejä. Projektissa käytettiin suurimmaksi osaksi GLSL:n pikseliohjelmia. Grafiikan suorituksen nopeutta pystytään optimoimaan erilaisilla tavoilla. Lowglow'ssa oleva valosysteemi toteutettiin kahdella render-tekstuurilla ja GLSL-ohjelmalla.

ASIASANAT:

shader, GLSL, OpenGL, grafiikkarajapinta, optimointi, cocos2d-x

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Tehcnology

2016 | 47

senior teacher & professor Mika Luimula

Sami Kangasmaa

GRAPHICS PROGRAMMING IN GAMES

- OpenGL and GLSL-shaders

Graphics programming is a programming area which focuses on components that users see. A graphical program is usually executed in frames that are shown to the user in short timeframes. Currently there are a few of graphic APIs such as OpenGL, DirectX, Vulkan and Metal. Different graphic programming techniques include, for example, HDR-rendering, normal mapping and anti-aliasing. The game engine of Lowglow was made with C++ using the Cocos2d-x framework which includes all the main features for developing games. GLSL-programs are OpenGL shaders that are executed on graphical processing unit. GLSL-programs can be used to create different effect and post-processing effects. We used mostly GLSL-pixel programs in our project. Graphical performance can be optimized by using different techniques such as bathing and rectangle checks. The light system in Lowglow was created using two render-texture and the GLSL-program.

KEYWORDS:

shader, GLSL, OpenGL, graphic API, optimization, cocos2d-x

SISÄLTÖ

KÄYTETYT LYHENTEET TAI SANASTO 6

1 JOHDANTO 7

2 GRAAFINEN OHJELMOINTI 8

2.1 Shader-ohjelmat ja renderöinti yleisesti 9

3 GRAFIIKKARAJAPINNAT 11

3.1 Yleistä 11

3.2 OpenGL 11

3.3 DirectX 11

3.4 Vulkan 12

3.5 Metal 12

3.6 Grafiikkarajapinnan valinta Lowglowiin 12

4 GRAAFISEN OHJELMOINNIN SOVELLUTUKSIA 13

4.1 Anti-aliasing 13

4.1.1 Täyden skenen anti-aliasing 14

4.1.2 Moninäyte anti-aliasing 14

4.2 HDR-renderöinti 16

4.3 Normal mapping –tekniikka 17

5 LOWGLOW'N PELIMOOTTORI 20

5.1 Yleistä 20

5.2 Cocos2d-x 20

5.2.1 Solmut ja hierarkiasysteemi 21

5.2.2 Koordinaatisto 23

5.2.3 Toimintasysteemi 23

6 GLSL-OHJELMAT 26

6.1 Yleistä 26

6.2 Tietotyypit 26

6.3 Pisteohjelmat 27

6.4 Pikseliohjelmat 28

6.5 Uniform 29

6.6 GLSL-ohjelmat Lowglow’ssa 30
6.6.1 Lowglow’n pisteohjelma 30

7 GRAFIIKAN OPTIMOINTI LOWGLOWISSA 32
7.1 Yleistä 32
7.2 Grafiikan optimoinnin teoriaa 32
7.2.1 Pistetaulukon valmistelu ja niputtaminen 32
7.2.2 Shader-ohjelmien suoritus 33
7.2.3 Täyttönopeus 34

8 EFEKTIEN TOTEUTUS LOWGLOWISSA 36
8.1 Valosysteemi 36
8.1.1 RTT 36
8.1.2 GLSL-ohjelma valoille 37
8.2 Ympäristöpalikat 40
8.2.1 Tekstuurien pakkaus ja valinta-algoritmi 41
8.2.2 Palikoiden GLSL-ohjelma 41

9 YHTEENVETO 44

LÄHTEET 45

KUVAT

Kuva 1. Vertailu ilman anti-aliasingia ja sen kanssa. (NVIDIA 2016.) 13
Kuva 2. MSAA-tekniikan havainollistaminen 15
Kuva 3. HDR-renderöintivertailu. (Smalley, T. 2004.) 17
Kuva 4. Puunkuora kuvaava tekstuuri ja siitä generoitu normal map -tekstuuri. 18
Kuva 5. 3D-kuution normal mapping -vertailu. (Photobucket 2016.) 19
Kuva 6. Esimerkki vanhempi-lapsi hierarkiasta. (Cocos2d-x 2016.) 22
Kuva 7. Hierarkiassa suhtteelinen paikka. (Cocos2d-x 2016.) 23
Kuva 8. Esimerkki valomaskista 37
Kuva 9. Kuvasta ja valomaskista lopputuloksen saaminen. 38
Kuva 10. Kuva Lowglowista valojen GLSL-ohjelman suorituksen jälkeen. 40
Kuva 11. Lowglow’n ympäristöpalikka 40

KÄYTETYT LYHENTEET TAI SANASTO

Cocos2d-x	Avoimen lähdekoodin viitekehys, joka on suunniteltu pelien toteuttaamiseen. (Cocos2d-x 2016)
CPU	Tietokonelaitteiston keskusprosessori.
FSAA	Full scene anti-aliasing –tekniikka. (NVIDIA 2016)
GLSL	OpenGL shading language. (OpenGL Wiki 2016)
GPU	Näytönohjain.
HDR-renderöinti	Tekniikka, jossa värejä käsitellään suurella dynaamisella alueella. (Unity 2016)
MSAA	Multisample anti-aliasing –tekniikka. (Wikipedia 2016)
RTT	Tekniikka, jossa piirretään tekstuuriin. (Craitoiu, S. 2014)
Shader	Näytönohjaimella suoritettava ohjelma. (Wikipedia 2016)

1 JOHDANTO

Työn aiheena on graafinen ohjelmointi Lowglow nimiseen peliprojektiin. Työssä käsitellään eri grafiikkarajapintoja sekä miksi päädyttiin tiettyihin rajapintoihin. Aluksi käsittelen teoriaa graafisesta ohjelmoinnista sekä erilaisia graafisia sovellutuksia, jotka koskettavat suurta osaa videopeleistä.

Työssä käsitellään myös Cocos2d-x–viitekehystä, jota käytimme Lowglow'n pohjana. Kerron, miten Lowglow käyttää hyödyksi Cocos2d-x:n tarjoamia ominaisuuksia sekä miten Cocos2d-x:n grafiikkasysteemi toimii.

Suurin osa työstä keskittyy OpenGL-kielen GLSL-ohjelmiin, joita käytimme Lowglowissa paljon. Käsittelen aiheeseen liittyen shader-ohjelmien toimintaa sekä niiden erilaisia toteutustekniikoita. Käsittelen myös samassa osiossa grafiikan optimointia shader-ohjelmien avulla ja, miten tekniikoita käytettiin hyväksi Lowglow'n grafiikan optimoinnissa.

Lopuksi keskityn kertomaan tehosteista ja efekteistä, joita tein Lowglowiin shader-ohjelmien avulla ja miten saimme toteutettua ne mahdollisimman tehokkaasti.

2 GRAAFINEN OHJELMOINTI

Graafisella ohjelmoinnilla (engl. graphics programming) tarkoitetaan ohjelmoinnin osa-aluetta, jossa keskitytään käyttäjälle näkyviin asioihin. Esimerkiksi videopeleissä ja viihdesovelluksissa käytetään todella paljon graafista ohjelmointia hyväksi. Graafinen ohjelmointi koostuu suurimmaksi osaksi tekstuurien (engl. texture) piirtämisestä näyttölaitteelle sekä niiden muokkaamiseen ohjelman suorituksen aikana shader-ohjelmien avulla. (Computer graphics 2016.)

Yleensä videopeliä halutaan suorittaa reaaliajassa, mikä tarkoittaa sitä, että sen tilaa päivitetään useita kertoja lyhyessä ajassa. Yhtä päivitysten välistä tilaa kutsutaan kehykseksi (engl. frame), jonka aikana haluttu tila piirretään näyttölaitteelle. Reaaliaikaisissa graafisissa sovelluksissa halutaan piirtää uusi kehys tarpeeksi nopeasti niin, että käyttäjä ei huomaa kehyksen selvästi vaihtuvan. Tällä tavoin ohjelma näyttää ihmisen silmälle sulavalta ja pehmeältä. Päivitysnopeutta (engl. frame rate) mitataan yleensä yksikössä FPS, kehystä sekunnissa (engl. frames per second), joka kertoo, kuinka monta kehystä suoritetaan sekunnissa. Tutkimusten mukaan ihminen pystyy erottamaan 10–12 täysin yksittäistä kuvaa sekunnissa. Kuitenkin erilaisten kappaleiden liikettä ihmisen silmä pystyy tunnistamaan nopeammin. (Frame rate 2016)

Tutkin verkkokauppa.com-sivustolla myytäviä näyttölaitteita, joista selvisi, että yleisimmin käytetty päivtysnopeus on 60 Hz. Pelaamiseen tarkoitetut erikoisnäytöt pystyvät päivittämään jopa 144 kuvaa sekunnissa. Ohjelman kehysten piirtoväli yritetään yleensä sovittaa yhteen näytön virkistystaajuden kanssa, jotta näytölle piirtyy aina kokonainen kuva uusimmasta kehyksestä. Tähän käytetään pystytahdistus tekniikkaa (vsync), joka kertoo näytönohjeimelle, koska näyttöpuskuriin voidaan piirtää (Adaptive V-Sync 2016).

Jotta ohjelman päivitysnopeus pysyy halutussa taajuudessa, täytyy yhden kehyksen suorituksen kestää vähemmän kuin näytön virkistyksen välisen ajan. Muutoin päivitysnopeus (FPS) laskee. Esimerkiksi 60 Hz:n päivitystaajuudella yhden kehyksen suoritukseen saa kulua enintään 16 ms, jotta uusi kehys valmistuu ennen ajankohtaa, jolloin se halutaan piirtää. Kehyksen suoritusaikaan vaikuttaa ohjelmaa suorittava laitteisto. Siksi yhden kehyksen aikana suoritettavien toimintojen määrässä tulee ottaa huomioon kohdelaitteiston suorituskapasiteetti.

2.1 Shader-ohjelmat ja renderöinti yleisesti

Shader-ohjelma on ohjelma, jota suoritetaan yleensä näytönohjaimella. Shader-ohjelmilla pystytään muokkaamaan ohjelman suorituksen aikana piirtyvää kuvaa sekä luomaan jälkikäsittelyefektejä. Shader-ohjelmat ovat laajalti käytettyjä videopeleissä sekä elokuvateollisuudessa. (Shader 2016)

Perinteisesti shader-ohjelmat on jaettu kahteen osa-alueeseen: pikseliohjelma (engl. pixel shader) sekä pisteohjelma (engl. vertex shader). Näiden lisäksi on myös olemassa geometrinen-shader (geometry shader) ja suoritus-shader (compute shader), joista jäkimmäistä käytetään useammin muuhun laskentaan kuin grafiikan piirtämiseen. (Shader 2016)

Pisteohjelmat (vertex shader) käsittelevät 3-ulotteisiä pisteitä, ja niiden tarkoitus on laskea pisteen paikka 2-ulotteisessa koordinaatistossa, joka yleensä on näyttölaitteen koordinaatisto. Ohjelma käy läpi jokaisen pisteen, joka sille ollaan syötetty ja syöttää vastauksen eteenpäin renderöintiputkistossa. Piste (vertex) voi sisältää tiedon paikasta, väristä sekä tekstuurin koordinaatista. Pisteiden tietoja voidaan tehokkaasti muokata pisteohjelmassa. Pisteohjelma ei kuitenkaan pysty enää suorituksen aikana luomaan uusia pisteitä. (Vertex Shader 2016)

Geometrinen-shader pystyy käsittelemään pisteitä, linjoja sekä kolmioita. Syöte tulee joko suoraan renderöintiputkiston alusta tai pisteohjelmalta (vertex shader). Geometrinen-shader (geometry shader) pystyy lisäämään uusiä pisteitä suorituksen aikana toisin kuin pisteohjelma (vertex shader). Tätä käytetään usein muokkaamaan 3-ulotteisen mallin tarkkuutta (level of detail) riippuen kameran etäisyydesta malliin poistamalla tai lisäämällä pisteitä. Geometria-ohjelma on jokseenkin uutta teknologiaa, joten vanhimmat laitteistot ja grafiikkarajapinnat eivät välttämättä tue sitä. (Geometry Shader 2016)

Pikseli-ohjelmaan syötetään pisteohjelmalta tai geometriaohjelmalta laskettujen pisteiden koordinaatit näytöllä, jotka on jaettu osiin (fragment). Yleensä yksi osa vastaa yhtä pikseliä näytöllä. Pikseliohjelma käy läpi jokaisen osan ja rasteroi sen. Laskettu ulostuloarvo on aina jokin väri. Pikseli-ohjelmia voidaan käyttää erilaisen värien yhdistämisten sekä jälkikäsittelyefektien toteuttamiseen. (Fragment Shader 2016)

Yleensä renderöintiputkisto etenee seuraavasti:

Prosessori syöttää käännetyt shader-ohjelmat, geometrian sekä käytettävät tekstuurit näytönohjaimelle. Tämän jälkeen suoritetaan piste-ohjelma jokaiselle geomerian sisältämälle pisteelle. Pisteohjelmalta ulostulleet pisteet muutetaan kolmioiksi, jonka jälkeen ne muutetaan osiksi (fragment), jotka edelleen syötetään pikseli-ohjelmalle. Pikseliohjelma käsittelee jokaisen osan ja syöttää tuloksen kuvapuskuriin, josta se piirretään seuraavalla päivityskerralla näyttölaitteelle. (Rendering pipeline overview 2016)

3 GRAFIIKKARAJAPINNAT

3.1 Yleistä

Grafiikkarajapinnat ovat sovellusohjelmointirajapintoja (engl. application programming interface), jotka ovat vuorovaikutuksessa suoraa näytönohjaimeen. Grafiikkarajapinnat suorittavat matalan tason toimintoja, joilla voidaan suorittaa erilaisia toimenpiteitä näytönohjaimella. Yleisimmin grafiikkarajapintoja käytetään piirtämään pikseleitä näytönohjaimen avulla näyttölaitteelle. Tällä hetkellä saatavilla on useita eri grafiikkarajapintoja eri näytönohjaimille. Kaikki näytönohjaimet eivät tue kaikkia grafiikkarajapintoja, joten joissakin tilanteissa sovelluksen pitää pystyä käyttämään useaa eri rajapintaa toimiakseen eri laitteilla. (Graphics library 2016)

3.2 OpenGL

OpenGL eli Open Graphics Library on avoin järjestelmäriippumaton grafiikkarajapinta, josta ensimmäinen versio julkaistiin vuonna 1992. OpenGL on myös ohjelmointikieliriippumaton, vaikka sen syntaksi muistuttaa hyvin paljon C-kieltä. OpenGL:lle on tehty lukuisia kielisidoksia esimerkiksi WebGL (JavaScript), iOS (C), Android (Java). OpenGL keskittyy ainoastaan grafiikan piirtämiseen eikä se sisällä toimintoja ikkunoiden hallintaan tai luomiseen. OpenGL:stä on myös olemassa karsittu versio: OpenGL ES, joka on laajalti käytössä nykyisissä mobiililaitteissa. Versiosta 2.0 asti OpenGL:ssä on ollut tuki GLSL-kielelle (engl. graphics library shading language), joka mahdollistaa shader-ohjelmien kirjoittamisen C-tyylisellä syntaksilla. (OpenGL 2016)

3.3 DirectX

DirectX sovellusohjelmointirajapinta sisältää grafiikkarajapinnan nimeltä Direct3D, joka on 3D-grafiikkarajapinta Microsoftin alustoille. Kuten OpenGL, DirectX tukee myös omaa shader-kieltänsä. DirectX:n shader-kielenä toimii HLSL (high level shading language), joka muistuttaa hyvin paljon GLSL-kieltä. HLSL-tuki julkaistiin DirectX 9.0:n mukana. (Microsoft 2016)

3.4 Vulkan

Vulkan on uuden sukupolven grafiikkarajapinta, joka julkaistiin vuonna 2016. Vulkanista sanotaan, että se on OpenGL:n seuraaja, vaikka se ei suoranaisesti ole uusi OpenGL-versio. Maailman surimmat näytönohjainten valmistajat ovat tukeneet Vulkanin kehitystä paljon. Esimerkiksi AMD, NVIDIA ja Intel tarjoaa osaan näytönohjainmalleista jo ajurit Vulkanille. (Khronos Group 2016)

3.5 Metal

Metal on Applen kehittämä grafiikkarajapinta iOS- ja MacOSX-alustoille. Metal julkaistiin vuonna 2015. Metal tarjoaa oman shader-kielensä, joka pohjautuu C++-kieleen. Apple lupaa Metallin käyttävän CPU-tehoja huomattavasti vähemmän kuin OpenGL. Lisäksi Metal tukee paremmin monisäieajoa, jolloin näytönohjain pystyy suorittamaan tehokkaammin useita asioita samanaikaisesti (Apple 2016). Vielä nähtäväksi jää miten Metallin kehityksen käy, kun Vulkanin käyttö yleistyy.

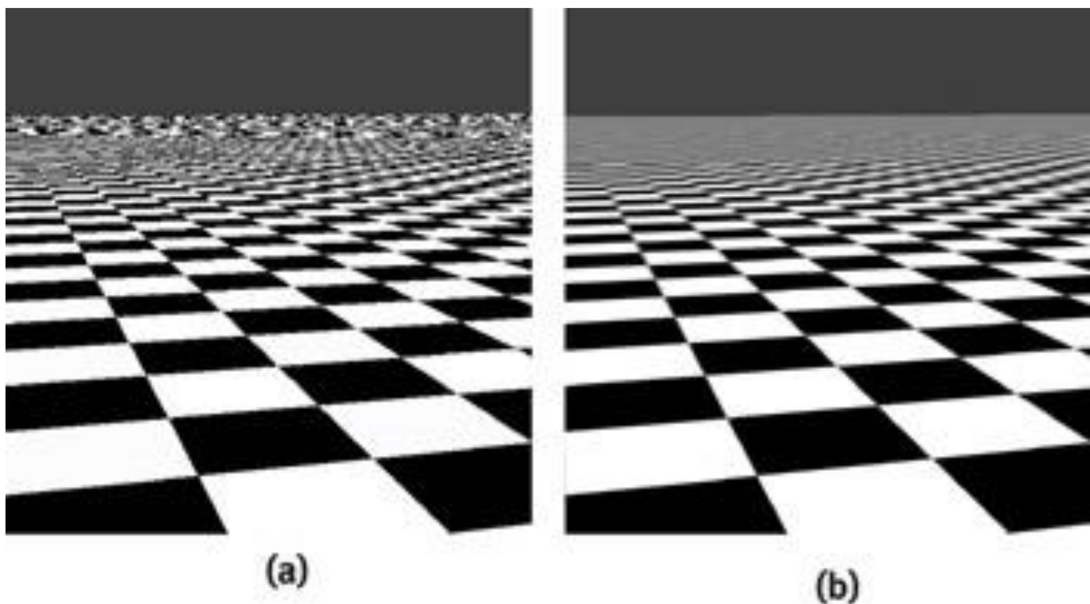
3.6 Grafiikkarajapinnan valinta Lowglowiin

Ennen projektin aloitusta tutkimme eri vaihtoehtoja. Peli tuli olla käännettävissä mahdollisimman helposti eri alustoille, joten halusimme käyttöömme teknologiaa, joka toimii suoraa eri alustoilla. Päädyimme käyttämään C++-kieltä käyttäen apuna Cocos2d-x –viitekehystä. Cocos2d-x on rakennettu pääasiassa OpenGL-grafiikkarajapinnan päälle, joten se ohjasi valintamme OpenGL:n käyttöön. Cocos2d-x sisältää myös DirectX-rajapinnan WindowsPhone-käyttöjärjestelmille, mutta puuttuva HLSL-tuki sai meidät luopumaan kokonaan DirectX:n käytöstä. OpenGL:n hyvät puolet projektiimme olivat, että se toimii iOS-, Android-, MacOSX- ja Windows –alustoilla. Vulkania emme harkinneet käytettäväksi, koska projektin aloitushetkellä Vulkanista ei oltu vielä julkaistu valmista versiota.

4 GRAAFISEN OHJELMOINNIN SOVELLUTUKSIA

4.1 Anti-aliasing

Anti-aliasing-tekniikkaa käytetään minimoimaan kuvan reunojen vääristymiä, kun korkearesoluutoista kuvaa näytetään pinemmällä resoluutiolla. Ilman anti-aliasing tekniikkaa kuvaan muodostuu rosoisia reunoja, kun sitä skaalataan pienemmäksi. Anti-aliasing pehmentää rosoisia reunoja suodattamalla kuvaa yhden tai useamman kerran. Kuvan pehmeys määräytyy suodatuskerroista (level of anti-aliasing). (MSDN 2016)



Kuva 1. Vertailu ilman anti-aliasingia ja sen kanssa. (NVIDIA 2016.)

Kuvassa 1 näkyy, miten anti-aliasing-tekniikka vähentää reunojen rosoisuutta. Vasemassa kuvassa (a) ei ole käytetty anti-aliasingia. Oikeassa kuvassa (b), jossa on käytetty anti-aliasingia reunat näyttävät huomattavasti pehmeämmiltä.

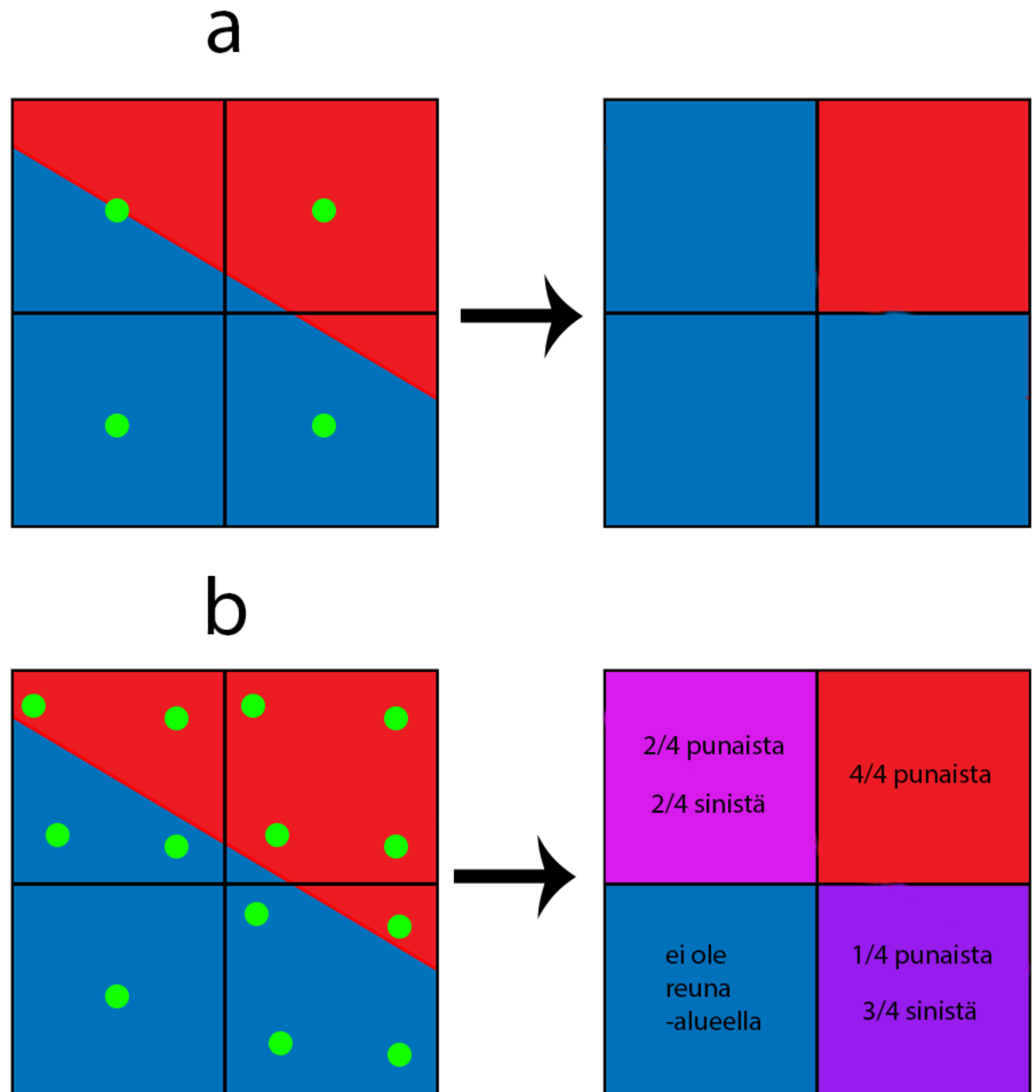
Videopeleissä käytetään hyvin usein anti-aliasing tekniikkaa. Kuva voidaan suodattaa esimerkiksi joko FSAA-tekniikalla tai MSAA-tekniikalla.

4.1.1 Täyden skenen anti-aliasing

FSAA-tekniikka eli full scene anti-aliasing toimii niin, että kuva piirretään suuremmalla resoluutiolla kuin lopullinen näytölle piirtyvä kuva. Tämä tarkoittaa sitä, että anti-aliasing suodatus tapahtuu vasta, kun koko kehyksen kuva on piirretty näyttöpuskuriin. Kuvan piirtoresoluutio riippuu anti-aliasing tasosta (level of anti-aliasing).Tätä kutsutaan super näytteenotoksi. Ennen näytölle piirtämistä kuva skaalataan näytön resoluutioon ja kuva suodatetaan anti-aliasing suodattimen läpi. FSAA-tekniikka vie hyvin paljon laitteiston suoritustehoa, koska jokainen pikseli pitää piirtää useamman kerran. Esimerkiksi 4x FSAA-tasolla, kuva piirretään nelinkertaisella resoluutiolla, jolloin piirrettäviä pikseleitä on 16-kertainen määrä alkuperäiseen nähden. (High-Resolution Antialiasing 2016)

4.1.2 Moninäyte anti-aliasing

MSAA-tekniikka, eli moninäyte anti-aliasing, on suoritustehollisesti parempi tapa suorittaa reunojen pehmennys. Tällöin kuva suodatetaan suoraa oikealla resoluutiolla, toisin kuin FSAA-tekniikassa. Moninäytetekniikassa jokaisesta monikulmion reunalla sijaitsevasta pikselistä otetaan, MSAA-tasosta riippuen, useampi näyte. Jokainen näyte otetaan hieman eri kohdasta pikselin keskipisteen ympäriltä, jonka jälkeen näytteet yhdistetään niiden keskiarvoksi. (Multisample anti-aliasing 2016)



Kuva 2. MSAA-tekniikan havainollistaminen

Kuvasta 2, jonka olen tehnyt havainollistamaan MSAA-tekniikkaa, näkee miten näytteiden ottaminen tapahtuu. Kuvassa yksi musta ruutu kuvaa yhtä pikseliä ja vihreät pallot näytteenoton paikkaa. Ylemmässä kuvassa (a) ei ole käytössä moninäyte anti-aliasing tekniikkaa, joten näyte otetaan aina pikselin keskeltä. Lopputulos näkyy kuvan (a) oikealla puolella, jossa pikselin väri on aina suoraa yhden näytteen väri. Tämä aiheuttaa reunojen rosoisuutta. Alemmassa kuvassa (b) on käytössä 4x MSAA, jossa otetaan neljä näytettä yhtä pikseliä kohden, jos pikseli sijaitsee monikulmion reuna-alueella. Vasemmalla alhaalla oleva pikseli ei ole sinisen monikulmion reuna-alueella, joten sille otetaan vain yksi näyte pikselin keskeltä. Muista pikseleistä huomaa, että osa

näytteistä osuu joko punaiselle alueelle tai siniselle alueelle. Näytteistä on laskettu keskiarvo alempaan oikeanpuoleiseen kuvaan, josta huomaa selvän eron lopputuloksessa, jos vertaa ilman anti-aliasing tekniikkaa piirrettyyn kuvaan.

4.2 HDR-renderöinti

HDR-renderöinti tarkoittaa sitä, että valojen laskenta tehdään suuremmalla dynaamisella alueella, kuin normaalisti. HDR-renderöinti mahdollistaa kuvan yksityiskohtien säilymisen ympäristössä, jossa on paljon kirkkaita- sekä paljon tummia kohtia.

Ongelma on, että tietokoneen väriarvot ovat väliltä nolla ja yksi (0-1), jossa nolla on täysin musta ja yksi täysin valkoinen. Ilman HRD-renderöintiä liian kirkkaat kohdat palavat täysin valkoisiksi alueiksi sekä tummat alueet täysin mustiksi alueiksi. HDR-tekniikassa valoja käsitellään suuremmalla alueella. Esimerkiksi kirkasta aurinkoa kuvaavan valon lähettämä valon väriarvo voi olla viisi (5). Normaalisti ilman HDR-renderöintiä, alueet johon valo osuu, käytetään yksinkertaisimmillaan vain värien kertolaskua. Tässä tapauksessa esimerkin auringon valaisemista kohdista voi tulla lopputulokseksi yli yksi. Koska suurin väriarvo yksi on täysin valkoinen, kaikki sen ylittävät arvot leikkautuvat valkoisiksi. Jos renderöinnissä ei käytetä HDR-tekniikkaa, mikään valojen väriarvoista ei saa ylittää yhtä (1). (Unity 2016)

HDR-tekniikalla laskennat suoritetaan pienimmän ja suurimman valon väriarvon välillä, jolloin leikkautumista ei tapahdu. Näytön esittämien värien arvot voivat olla vain välillä nolla ja yksi. Tämän takia lasketut laajan dynamiikan väriarvot tulee lopuksi interpoloida välille 0-1.

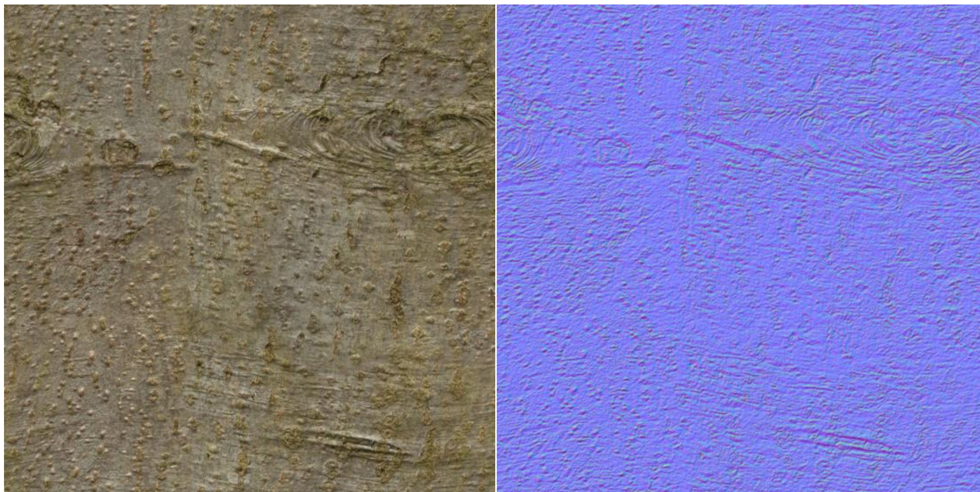
Kuva 3. HDR-renderöintivertailu. (Smalley, T. 2004.)

Ylläoleva kuva esittää HDR-renderöintitekniikkaa CryEnginessä. Vasemmalla puolella on kuva, joka on renderöity ilman HDR-tekniikkaa. Kuvasta huomaa, että se on erittäin tumma ja yksityiskohdat häviävät. Oikeanpuoleisessa kuvassa, jossa HDR-renderöinti on päällä, huomaa selvästi miten kirkkaat kohdat erottuvat paremmin sekä tummien kohtien yksityiskohdat tulevat paremmin esille.

4.3 Normal mapping –tekniikka

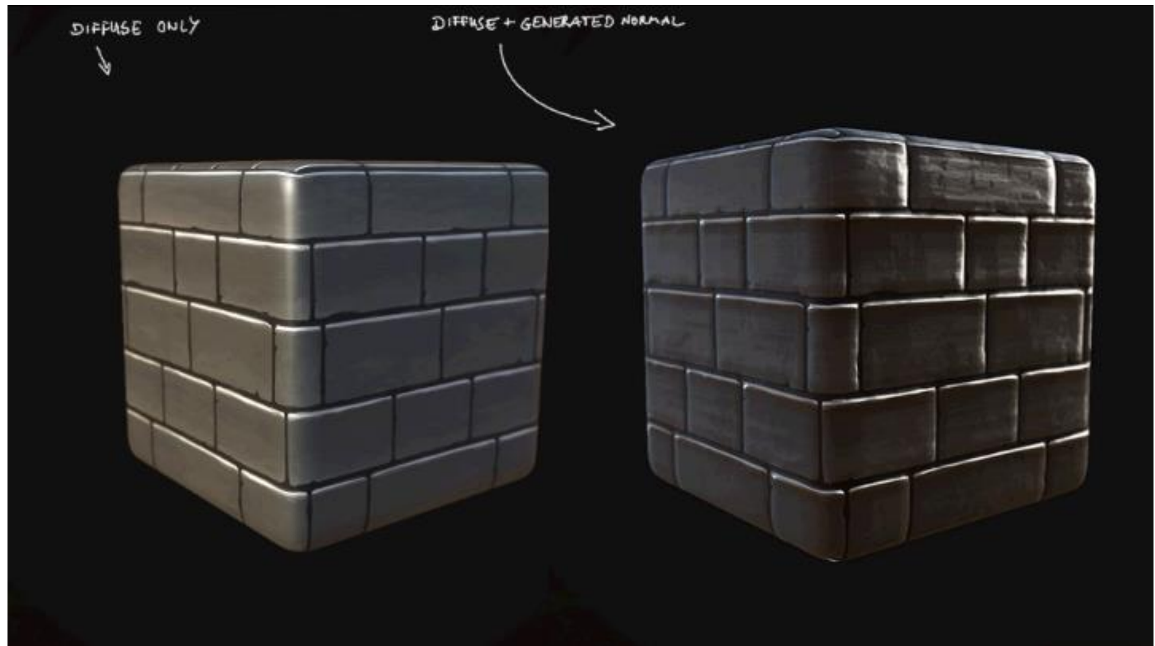
Normal mapping –tekniikka on hyvin yleisesti käytetty tekniikka videopeleissä. Sen ideana on ”huijata” kappaleen valaistus näyttämään siltä, että siinä olisi ylimääräisiä yksityiskohtia. Ilman normal mapping –tekniikkaa 3D-malliin pitää lisätä monikulmioita, jos siihen halutaan lisää yksityiskohtia. Tämä ei ole tehokas tapa, jos 3D-malleja pitää renderöidä useita samanaikaisesti reaaliajassa. Normal mapping –tekniikalla matalamonikulmion 3D-malli saadaan näyttämään yksityiskohtaiselta korkean monikulmion mallilta. (Learn OpenGL 2016)

Normal mapping tallennetaan usein RGB-tekstuuriin, jonka värikomponentit vastaavat x, y ja z –koordinaatteja pinnan normaaliin nähden. Pinnan normaalia muokataan näiden koordinaattien avulla. Pinnan lambertian-valaistus (yleensä terminä diffuse color) lasketaan ottamalla pistetulo yksikkövektorista varjostuspisteestä valonlähteeseen ja yksikkövektorista joka on varjostettavan pinnan normaali. Muuntamalla pinnan normaalia, valaistuksen intensiteetti saadaan joko suuremmaksi tai pinemmäksi, jonka avulla tasaiselle pinnalle saadaan yksityiskohtia.



Kuva 4. Puunkuora kuvaava tekstuuri ja siitä generoitu normal map -tekstuuri.

Generoin ylläolevasta vasemmanpuoleisesta kuvasta normal mapin, joka näkyy yllä olevista kuvista oikealla puolella. Tekstuurissa punainen värikomponentti kertoo x-vektorin, vihreä värikomponentti y-vektorin sekä sininen värikomponentti z-vektorin.



Kuva 5. 3D-kuution normal mapping -vertailu. (Photobucket 2016.)

Ylläoleva kuva havainollistaa, miten normal mapping –tekniikka vaikuttaa valaistuksen lopputulokseen. Vasemmalla puolella kuutio on teksturoitu pelkästään käyttämällä yhtä kuvaa. Valon intensiteetin laskemisessa oletetaan pinnan normaali aina saman suuruiseksi. Tämän takia pinta näyttää hyvin tasaiselta. Oikealla puolella on käytössä sama kuvatekstuuri sekä sen lisäksi generoitu normal map –tekstuuri. Pinta näyttää paljon epätasaisemmalta ja siinä on enemmän yksityiskohtia. Pinnan normaalia muokkaamalla on saatu aikaan kirkkaampia sekä tummempia kohtia.

5 LOWGLOW’N PELIMOOTTORI

5.1 Yleistä

Lowglow on fysiikaan perustuva ongelmanratkontapeli tietokone- ja mobiililaitteille. Halusimme tehdä audiovisuaalisesti uniikin näköisen pelin, jossa olisi rauhallinen tunnelma. Aloitimme kehittämään Lowglow:ta maaliskuussa 2015 ja saimme julkaistua sen Windows- ja MacOSX-alustoille 3.12.2015. Myöhemmin Lowglow tullaan julkaisemaan iOS- sekä Android-alustoilla. Pelin grafiikka pohjautuu hyvin paljon graafiseen ohjelmointiin ja tulen myöhemmin esittelemään aiheesta lisää. Minun päävastuullani oli pelimoottorin ohjelmoiminen Cocos2d-x -viitekehyksen ympärille sekä GLSL-ohjelmien tekeminen.

5.2 Cocos2d-x

Lowglow’n pelimoottori toteutettiin Cocos2d-x –viitekehyksen päälle. Cocos2d-x on järjestelmäriippumaton avoimen lähdekoodin C++ viitekehys, joka on suunniteltu käytettäväksi pelien tekemiseen. Cocos2d-x sisältää lähes kaikki tarvittavat toiminnot pelimoottorin tekemiseen. (Cocos2d-x 2016)

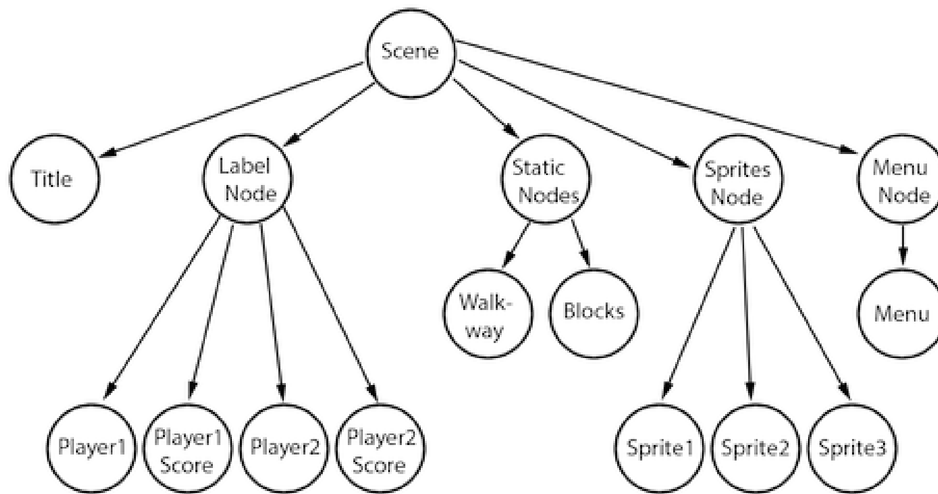
Cocos2d-x sisältää seuraavat ydinominaisuudet:

- grafiikan piirto
    - OpenGL ES 2.1 Androidilla ja iOS:llä
    - OpenGL 2 Windows:lla ja MacOSX:llä
    - DirectX Windows Phone 8:lla
- käyttöliittymät (GUI)
    - painikkeet
    - tekstilappu
    - tekstialue
    - listanäkymä
    - valintaruutu
    - rullanäkymä
- fysiikkasimulaatio ja törmäystunnistus
    - Box2D

  - chipmunk
- verkkoyhteydet
  - HTTP
  - SSL-salaus
- Lua- ja JavaScript-sidos
- automaattinen muistinhallinta
- syötteiden käsittely
  - kosketusnäytöt
  - näppäimistö
  - hiiri
- äänen- ja musiikin toisto
  - tuetut tiedostoformaatit: MP3, OGG, CAF ja WAV
- partikkelisysteemit
- GLSL-shader tuki
- järjestelmäriippumaton tiedostojen ja kansioiden käsittely
- scene-systeemi

5.2.1 Solmut ja hierarkiasysteemi

Cocos2d-x:n kaikki piirtyvät oliot on peritty Node-luokasta (solmu), joka sisältää atribuutit sen paikasta, kierrosta, skaalasta, väristä sekä läpinäkyvyydestä. Solmut toimivat vanhempi-lapsi hierarkiassa, joka tarkoittaa sitä, että kaikki lapsisolmut toimivat oman vanhempansa tilamatriisissa. Esimerkiksi skaalaamalla vanhempisolmua, kaikki sen lapsisolmut skaalautuvat myös suhteessa vanhempisolmun hierarkiatasossa oleviin solmuihin. (Cocos2d-x API reference 2016)



Kuva 6. Esimerkki vanhempi-lapsi-hierarkiasta. (Cocos2d-x 2016.)

Ylläolevassa kuvassa on esimerkki, miten hierarkia voisi toimia Cocos2d-x:ssä. Ylimpänä näkyy Scene, joka on hierarkiassa kaikista ylin. Tämä tarkoittaa sitä, että kaikki muut solmut piirretään suhteessa siihen. Oikealta katsottuna toinen Sprites Node on tyhjä solmu, joka sisältää vain sen perusatribuutit. Sen lapsisolmuina ovat Sprite1, Sprite2 ja Sprite3, jotka ovat kuvia. Kuvien lopulliset atribuutit määräytyvät siis Sprite Noden ja Scenen mukaa, koska ne ovat hierarkiassa ylempänä.

Esimerkiksi Sprite1 on kuva, jonka koko on 512 x 512 pikseliä. Scenen skaala-atribuutti on 1,0, Sprite Node:n skaala-atribuutti on 0,5 ja Sprite1:n skaala-atribuutti on 1,75.

Lopullinen kuvan pikselikoko lasketaan seuraavasti:

$$x_1 = scale_{sprite1} * \ scale_{spritenodes} * \ scale_{scene} * \ x_0$$

$$y_1 = scale_{sprite1} * \ scale_{spritenodes} * \ scale_{scene} * \ y_0$$

Jossa x_0 on kuvan alkuperäinen leveys pikseleinä ja y_0 kuvan alkuperäinen korkeus pikseleinä.

$$x_1 = 1{,}75 * 0{,}5 * 1{,}0 * 512 = 448$$

$$y_1 = 1{,}75 * 0{,}5 * 1{,}0 * 512 = 448$$

Näytölle piirtyvän kuvan koko on siis 448x448.

5.2.2 Koordinaatisto

Cocos2d-x käyttää oikean käden koordinaatistoa, joka tarkoittaa sitä, että piste (0, 0) on ruudun vasemmassa alareunassa. X-koordinaatti kasvaa oikealle ja Y-koordinaatti kasvaa ylös.

Kuva 7. Hierarkiassa suhtteelinen paikka. (Cocos2d-x 2016.)

Kuvassa 7 on esimerkki, miten solmujen hierarkia vaikuttaa kuvien paikkaan. Oletetaan, että sinisen- ja ruskean pallon koot ovat 100x100. Ruskea pallo on sinisen pallon lapsisolmu. Sininen pallo on paikassa (100, 100) sen vanhempisolmuun nähden. Ruskean pallon paikka atribuutti on myös pisteessä (100, 100). Koska sininen pallo on ruskean pallon vanhempisolmu, ruskean pallon paikka lasketaan sinisen pallon vasemmasta alakulmasta, joka on sinisen pallon paikka-avaruudessa piste (0, 0). Ruskean pallon paikka on siis aina suhteessa (engl. relative) sinisen pallon paikkaan. Jos sinista palloa liikutetaan, liikkuu myös ruskea pallo sen mukana.

5.2.3 Toimintasysteemi

Cocos2d-x sisältää myös erittäin hyödyllisen toimintasysteemin (engl. action system), jolla pystyy interpoloimaan solmujen atribuutteja halutun aikajakson välillä. Solmut pystyvät suorittamaan useita toimintoja samanaikaisesti poislukien tilannetta, jossa kaksi eri toimintoa muokkaa samoja solmun atribuutteja.

Yleisimmin käytettyjä solmujen perusatribuutteja muokkaavia toimintoja ovat

- ScaleTo

    - Muokkaa solmun skaalausatribuutin lineaarisesti annettuun arvoon tietyn ajan sisällä.
- ScaleBy
    - Muokkaa solmun skaalausatribuutin lineaarisesti arvoon tietyn ajan sisällä, joka on: skaala-arvo alussa * syötetty arvo.
- MoveTo
    - Muokkaa solmun paikka-atribuuttia lineaarisesti annettuun arvoon tietyn ajan sisällä.
- MoveBy
    - Muokkaa solmun paikka-atribuutin lineaarisesti arvoon tietyn ajan sisällä, joka on: paikka alussa + syötetty paikka-arvo.
- RotateTo
    - Muokkaa solmun kierreatribuuttia lineaarisesti annettuun arvoon tietyn ajan sisällä käyttäen lyhimpää reittiä.
- RotateBy
    - Muokkaa solmun kierreatribuuttia lineaarisesti arvoon tietyn ajan sisällä, joka on: kierre alussa + syötetty kierrearvo.
    - Toiminto ei ota huomioon lyhintä reittiä.
- FadeTo
    - Muokkaa solmun läpinäkyvyyttä lineaarisesti annettuun arvoon tietyn ajan sisällä.

Cocos2d-x sisältää myös muita somujen toimintoja, jotka eivät suoraa muokkaa solmun atribuutteja. Nämä toiminnot pystyvät ottamaan parametreiksi perusatribuutteja muokkaavia toimintoja sekä muita toimintoja tietyin poikkeuksin.

- EaseAction
    - EaseAction-toimintoa voidaan käyttää pehmentämään persuatribuutteja muokkaavia toimintoja. Pehmennystoiminnot toimivat interpoloimalla perustoiminnoille syötettävää aikaa bezier-interpolaatiolla. Cocos2d-x sisältää useita valmiiksi määriteltyjä bezier-kokoonpanoja, jossa interpolaation tarvitsemat kontrollipisteet ovat ennalta määritelty.
- Sequence
    - Sequence-toiminto ei itsessään tee mitään, vaan ottaa parametreikseen muita toimintoja ja tekee niistä suoritettavan jonon. Jonon toiminnot

suoritetaan siinä järjestyksessä, missä ne on annettu Sequence-toiminnolle.

- Repeat
    - Repeat-toiminto ottaa parametrikseen toisen toiminnon. Syötettyä toimintoa suoritetaan haluttu määrä kertoja.
- RepeatForever
    - RepeatForever-toiminto ottaa parametrikseen minkä tahansa muun toiminnon. Tämä toimii samalla tavalla kuin Repeat-toiminto, mutta syötettyä toimintoa suoritetaan loputtomiin. RepeatForever-toimintoa ei voida antaa parametriksi millekään muulle toiminnolle.
- CallFunc
    - CallFunc-toiminto kutsuu parametsiki annettua funktiota. Toiminto on hyödyllinen esimerkiksi tilanteessa, jossa halutaan suorittaa pelilogiikkaa jonkin Sequence-toiminnon päätteeksi.

6 GLSL-OHJELMAT

6.1 Yleistä

Suuri osa Lowglow'n visuaalisuudesta toteutettiin GLSL-shader-ohjelmien avulla. GLSL (graphic library shading language) on OpenGL-grafiikkarajapinnalle tarkoitettu ohjelmointikieli, jolla voidaan ohjelmoida näytönohjaimella ajettavia ohjelmia (engl. shaders). GLSL tukee C:n tapaan silmukkarakenteita (for, while ja do-while), haaroituksia (if, elseif ja else) sekä käyttäjän itse määrittelemiä funktioita. (Kessenich ym. 2014, 27.) GLSL sisältää suuren määrän erilaisia valmiiksi määriteltyjä funktioita, joiden toiminta on yleensä sisäänrakennettu laitteistoon parhaimman suoritusnopeuden saavuttamiseksi.

6.2 Tietotyypit

GLSL tukee monia eri muuttujien tietotyyppejä. Muutuja pitää määritellä ennen, kuin sitä voidaan käyttää. Määrittelyssä muuttujalle on annettava vähintään muuttujan tietotyyppi ja halutessaan voidaan asettaa arvo määrittelyn yhteydessä. Muuttujan nimessä ei saa olla välilyöntejä eikä se saa alkaa numerolla. Muuttujat määritellään seuraavasti:

Ilman alustusta: *tietotyyppi muuttujan_nimi;*

Alustuksen kanssa: *tietotyyppi muuttujan_nimi = 0;*

GLSL:n tukemat tietotyypit ovat

- skalaariarvot
    - bool: arvo voi olla joko tosi (true) tai epätosi (false)
    - int: 32-bittinen etumerkillinen kokonaisluku
    - uint: 32-bittinen etumerkitön kokonaisluku
    - float: yksöistarkkuuden liukuluku
    - double: tuplatarkkuuden liukuluku
    - skalaariarvoille voidaan suorittaa kerto-, jako-, yhteen- ja vähennyslaskuja.
- sektorit

    - vektorit sisältävät joko 2, 3 tai 4 skalaarikomponenttia
    - vektorin määrittelysääntö noudattaa kaavaa:
        - bvecn: bool –vektori, jossa n on komponenttien määrä.
        - ivecn: int –vektori, jossa n on komponenttien määrä.
        - uvecn: uint –vektori, jossa n on komponenttien määrä.
        - vecn: float –vektori, jossa n on komponenttien määrä.
        - dvecn: double –vektori, jossa n on komponenttien määrä.
    - float-vektori on yleisimmin käytetty, jonka takia sillä ei ole erillistä etuliitettä.
    - vektoreilla voidaan suorittaa samat laskutoimitukset kuin skalaariluvuilla.
- matriisit
    - Matriisit ovat aina tyypiltaan liukulukuja.
    - Matriisin määrittely on matn tai matnxn, jossa n on matriisin koko.
- läpinäkymättömät tyypit
    - Eivät sisällä itsessään mitään arvoa, vaan viittauksen itse oikeaan dataan.

(Kessenich ym. 2014, 31)

6.3 Pisteohjelmat

GLSL-pisteohjelma laskee syötettyjen pisteiden paikan projektiomatriisin avulla. Lasketut pisteet syötetään eteenpäin pikesliohjelman käsittelyyn. Pisteet (vertex) voivat sisältää myös värin sekä tekstuurikoordinaatin, jota voidaan käyttää hyväksi pisteohjelman jälkeen suoritettavassa pikseliohjelmassa. Syötettävät arvot määritellään sanalla ”attribute”, joka kertoo pisteohjelmalle, että muuttujien arvot luetaan ohjelman ulkopuolelta. Eteenpäin välitettävät arvot määritellään sanalla ”varying”. Nämä arvot välitetään yleensä eteenpäin piskeliohjelmalle, joka esimerkiksi tarvitsee kuvan piirtämiseen tekstuurin koordinaateja. Alla on yksinkertainen GLSL-pisteohjelma, jonka kirjoitin.

```
attribute vec4 a_position;

attribute vec2 a_texCoord;

attribute vec4 a_color;
```

```
varying vec4 v_fragmentColor;
varying vec2 v_texCoord;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * a_position;
    v_fragmentColor = a_color;
    v_texCoord = a_texCoord;
}
```

Muuttuja a_position on määritelty atribuutiksi, johon syötetään pisteen paikka ohjelman ulkopuolelta. Muuttuja a_texCoord on määritelty atribuutiksi, johon syötetään pisteen tekstuurikoordinaatti ohjelman ulkopuolelta. Atribuutti a_color on pisteen RGBA-väriarvo, joka syötetään ohjelman ulkopuolelta. Muuttuja v_fragmentColor ja – v_texCoord on määritelty varying-sanalla, jolloin muuttujat välitetään eteenpäin pisteohjelmasta suorituksen jälkeen. Ohjelman suoritus aloitetaan sen pääfunktiosta (void main()), joka pitää olla aina määritelty. gl_Position on yleensä ennalta määritelty atribuutti, joka kuvaa pisteen lopullista paikkaa. Esimerkkiohjelma asettaa paikaksi projektiomatriisin ja syötetyn paikan tulon, joka kertoo syötetyn pisteen paikan näyttölaitteen koordinaatistossa. Lopuksi ohjelma asettaa varying-muuttujiin syötetyn väriarvon sekä syötetyn tekstuurikooridnaatin, jotta ne välittyisivät eteenpäin myöhempää käyttöä varten.

6.4 Pikseliohjelmat

Pikseliohjelma (fragment shader) suoritetaan jokaiselle pisteohjelmasta syötetylle osalle (fragment), jonka se rasteroi pikseleiksi. Loppuarvo on aina jokin väri. Yksinkertaisimmillaan piskeliohjelma tarvitsee vain pääfunktion, jossa asetetaan osan väriarvo.

```
void main (void)
{
```

```
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Ylläoleva esimerkkiohjelma värjää kaikki rasteroidut pikselit punaisiksi. gl_FragColor on OpenGL:ssä ennalta määritelty atribuutti, joka kertoo pikseliohjelmassa lasketun osan väriarvon. (Kessenich ym. 2014, 139.) Väri on vec4-muodossa, jonka komponentit vastaavat RGBA-väriä.

6.5 Uniform

Uniform on globaali muuttuja, jonka arvoa voidaan vaihtaa shader-ohjelman ulkopuolelta. Uniform-muuttujat toimivat eräänlaisina parametreina, jota voivat vaikuttaa ohjelman suoritukseen. Uniform-muuttujat ovat pysyviä, joka tarkoittaa sitä, että niiden arvo säilyy kahden shader-ohjelman suorituksen välissä. (Kessenich ym. 2014, 57)

Uniformit sijaitsevat tietyssä paikassa (engl. uniform location), joka ilmoitetaan kokonaislukuarvona (GLint). Jotta uniform-muuttujan arvoa voidaan vaihtaa, pitää sen paikka tietää. Paikan saa funktiolla *glGetUniformLocation(*glsl_ohjelma_olio, "uniformin nimi shader-ohjelmassa")*. Funktio palauttaa GLint-tyyppisen kokonaislukuarvon, joka yleensä otetaan talteen myöhempää käyttöä varten. Suurin sallittu määrä uniformeja yhdessä shader-ohjelmassa on vähintään 1024 (Uniform (GLSL) 2016), mutta jotkin laitteistot tukevat suurempaa määrää.

Uniform-muuttujan arvoa voidaan vaihtaa OpenGL uniform –funktioilla, kun uniformin paikka tiedetään. Jokaiselle GLSL-kielen tukemalle tietotyypille on oma uniform-funktionsa. Uniform-funktiot noudattavat kaavaa: glUniform[n][t](GLint paikka, arvo1, arvo2, ...), jossa [n] on kohdemuuttujan arvojen määrä ja [t] on kohdemuuttujan arvojen tietotyyppi. Esimerkiksi *float*-tyyppisen uniformin arvo asetetaan seuraavasti: glUniform1f(location, 0.5f).

6.6 GLSL-ohjelmat Lowglow'ssa

Lowglow'ssa käytettiin efektien tekemiseen erilaisia pikseliohjelmia ja ainoastaan yhtä pisteohjelmaa. Efektien toteuttamiseen tehdy pikseliohjelmat saavat parametrinsa uniformien kautta. Samaa pikseliohjelmaa käytetään usealla eri peliobjektilla, jolloin uniformin arvon muuttaminen vaikuttaisi kaikkiin samaa shader-ohjelmaa käyttäviin objekteihin. Halusimme kuitenkin jokaiselle peliobjektille omat parametrit. Tässä tapauksessa jokaiselle objektille tulee määritellä omat uniformin paikat. Käytimme apunamme Cocos2d-x:n sisäänrakennettua *GLProgramState* –systeemiä. Jokainen shader-ohjelmaa käyttävä peliobjekti sisältää oman *GLProgramState*-olion, johon objektin uniformien paikat säilötään. Tällöin uniformin arvo pystytään asettamaan ainoastaan halutulle peliobjektille.

6.6.1 Lowglow'n pisteohjelma

Lowglowissa käytetty pisteohjelma on yksinkertainen nelikulmion laskenta. Nelikulmion kulmapisteet vastaavat 2-uloitteisen kuvan kulmapisteitä, joiden paikka lasketaan projektiomatriisin ja peliobjektin paikan avulla. Alla on Lowglow'ssa käytetyn pisteohjelman GLSL-koodi kommentoituna:

```
attribute vec4 a_position; //Objektin paikka pelimaailmassa
attribute vec2 a_texCoord; //Tekstuurin koordinaatit
attribute vec4 a_color; //Pisteen RGBA-väriarvo

/*
Erotellaan esikäsittelijällä OpenGL ES ja OpenGL-täysi versio.
*/
#ifdef GL_ES
varying lowp vec4 v_fragmentColor;
varying mediump vec2 v_texCoord;
#else
varying vec4 v_fragmentColor;
varying vec2 v_texCoord;
#endif
```

```
void main() //Pääfunktio
{
   //lasketaan pisteen paikka näytöllä projektiomatriisin ja pisteen pelimaailman paikan avulla
   gl_Position = CC_PMatrix * a_position;
   v_fragmentColor = a_color; // syötetään eteenpäin pisteen väri
   v_texCoord = a_texCoord; //syötetään eteenpäin tekstuurin koordinaatti
}
```

Shader-ohjelmaa on optimoitu käyttämällä eri tarkkuudella toimivia muuttujia. OpenGL ES-versiolla (mobiililaitteilla) suorittaessa käytetään matalaa tarkkuutta pisteen värissä sekä keskitason tarkkuutta tekstuurin koordinaatissa paremman suoritustehon saavuttamiseksi. Muuttujat voidaan määrittää joko matalalle tarkkuudelle (lowp), keskitason tarkkuudelle (mediump) tai korkealle tarkkuudelle (highp). Lowglowissa kaikki muuttujat shader-ohjelmissa ovat oletuksella korkealla tarkkuudella.

7 GRAFIIKAN OPTIMOINTI LOWGLOWISSA

7.1 Yleistä

Suurin osa Lowglow'n efekteistä on tehty muokkaamalla valmiiksi tehtyjä kuvia shader-ohjelmilla. Esittelen tässä osiossa muutamia efektien toteutustapoja sekä tutkimustuloksia niiden suoritustehon optimoinnista. Grafiikan piirron optimointi tuli erittäin tärkeäksi Lowglowissa, koska sen tulee toimia mobiililaitteilla ja suurin osa Lowglow'n suoritusajasta kuluu näytönohjaimen puolella.

7.2 Grafiikan optimoinnin teoriaa

Grafiikan piirron suoritusnopeutta voidaan optimoida monella tapaa. Tavoitteena olisi, että haluttu lopputulos saadaan aikaiseksi mahdollisimman pienellä suoritusajalla. Suoritusaikaan vaikuttaa suurimmaksi osaksi laitteiston prosessori (CPU) ja – näytönohjain (GPU). Grafiikan piirto suoritetaan Lowglowissa vaiheittain jonossa noudattaen OpenGL:n renderöintiputkistoa.

7.2.1 Pistetaulukon valmistelu ja niputtaminen

Aluksi piirettävän kohteen pistetaulukot valmistellaan renderöintiä varten. Tämän vaiheen suorittaa laitteiston CPU, joka välittää valmistellut pisteet näytönohjaimelle. Jokainen funktiokutsu näytönohjaimelle kuluttaa ylimääräistä aikaa (engl. overhead), jonka takia jokaista piirrettävää kohdetta ei kannata kutsua piirrettäväksi yksitellen. Jotta ylimääräinen hukka-aika saataisiin mahdollisimman pieneksi, useat eri piirrettävät kohteet niputetaan (engl. batching) ja piirretään samalla funktiokutsulla. Nippuun valmistellaan kaikkien siihen sisällytettävien kohteiden pisteet ja muut atribuutit. Niputtamisessa kuitenkin on rajoitteena, että kaikkien samaan nippuun kuuluvien kohteiden tulee olla identtisiä. (Krzeminski, M. 2014)

Esimerkiksi Lowglowissa käytetyt partikkeliefektit voivat sisältää jopa 500 yksittäistä partikkelia ja efektejä voi olla samaan aikaan näytöllä useita. Tässä tapauksessa näytönohjaimelle tehtäisiin jopa yli 5 000 piirtokutsua kehyksessä pelkästään partikeliefekteille, joka veisi erittäin paljon suoritusaikaa prosessorilta. Koska partikkelit

käyttävät samaa tekstuuria ja samaa piirtotapaa, ovat ne identisiä, jolloin ne voidaan piirtää nipussa. Niputettuna partikkeliefektit tarvitsevat vain yhden piirtokutsun, jolloin niiden piirtämiseen kuluu huomattavasti lyhyempi aika.

7.2.2 Shader-ohjelmien suoritus

Kun pistetaulukko on valmisteltu, kutsutaan näytönohjainta piirtämään se. Tässä vaiheessa laiteiston CPU pystyy jatkamaan omaa suoritustaan ja laskenta siirtyy näytönohjaimelle (Rendering Pipeline Overview 2016). Näytönohjaimet ovat suunniteltu niin, että ne pystyvät laskemaan erittäin nopeasti samaa kaavaa noudattavia laskutoimituksia suuria määriä (Wikipedia 2016). Vaikka GLSL-kieli tukee haaroitusrakenteita, tulisi niiden käyttöä välttää viimeiseen asti, koska näytönohjaimet käsittelevät loogisia operaatioita huomattavasti hitaammin kuin CPU, koska näytönohjaimet on rakenettu suorittamaan laskuja rinnakkain (Understanding the parallelism of GPUs 2014). Koska piirron aikana shader-ohjelman laskutoimitukset voidaan suorittaa useita miljoonia kertoja, kannattaa sen suunnittelussa ottaa huomioon laskutoimitusten määrä. Joissakin tapauksissa sama lopputulos voidaan saavuttaa huomattavasti pienemmällä määrällä laskutoimituksia.

```
uniform sampler2D dst_texture;
uniform sampler2D src_texture;
uniform float alpha;
varying vec2 v_texCoord;

void main (void)
{
  vec3 dst_color = texture2D(dst_texture, v_texCoord);
  vec3 src_color = texture2D(src_texture, v_texCoord);
  if(alpha != 0)
  {
    gl_FragColor = vec3(src_color.r  - ((1-alpha)*dst_color.r)/alpha, src_color.g  - ((1-
alpha)*dst_color.g)/alpha, src_color.b  - ((1-alpha)*dst_color.b)/alpha);
  }
  else
  {
```

```
    gl_FragColor = dst_color;
  }
}
```

Ylläoleva tekemäni esimerkkipikseliohjelma tekee *alpha blend*-operaation kahdelle tekstuurille. Ohjelmassa on käytetty jakolaskua selvittämään lopullinen värikanavan arvo. Jos alpha-muuttujan arvo on 0, suoritetaan matemaattisesti mahdoton laskutoimitus, jonka takia ohjelman tulee tarkistaa loogisella operaatiolla onko arvo sallittu. Tämä hidastaa piirto-operaatiota huomattavasti. Esimerkkiohjelma voidaan optimoida, jolloin lopullinen ohjelma näyttää tältä:

```
uniform sampler2D dst_texture;
uniform sampler2D src_texture;
uniform float alpha;
varying vec2 v_texCoord;

void main (void)
{
  vec3 dst_color = texture2D(dst_texture, v_texCoord);
  vec3 src_color = texture2D(src_texture, v_texCoord);
   gl_FragColor = vec3( src_color.rgb * alpha + dst_color.rgb * (1.0 - alpha));
}
```

Kuten huomataan, sama lopputulos saadaan myös tehtyä ilman jakolaskua, jolloin alpha-muuttujan arvoa ei tarvitse erikseen tarkistaa. Optimoitu versio samasta ohjelmasta osoittautui testeissämme huomattavasti nopeammaksi.

7.2.3 Täyttönopeus

Näytönohjaimen täyttönopeus (engl. fillrate) mittaa, kuinka nopeasti näytönohjain pystyy kirjoittamaan pikseleitä näyttölaitteelle tai videomuistiin. 2-uloitteisissa videopeleissä yleensä pullonkaulaksi muodostuu täyttönopeuden riittämättömyys, koska niissä piirretään paljon kuvia päällekkäin. Koska pikseleiden määrä riippuu

resoluutiosta, joka on yleensä korkea tablet-laitteilla, Lowglowissa ongelmaksi tuli täyttönopeuden loppuminen kyseisillä laitteilla. Ongelmaan ei ole muuta ratkaisua, kuin pirtää vähemmän pikseleitä. Pikseleiden piirtämisen määrää voidaan rajoittaa usealla eri tavalla, joista esittelen meidän käyttämät tavat.

Yleensä 2-uloitteisissa peleissä piirretään monia kuvia päällekkäin. Joissakin tapauksissa päälimmäinen kuva voi peittää kokonaan sen alla olevan kuvan. Tässä tapauksessa ei ole järkevää piirtää ollenkaan alle jäävää kuvaa. Jos kuvat ovat suorakulmion muotoisia, voidaan helposti laskea jääkö alempi kuva ylemmän kuvan peittämäksi testaamalla kaikki sen kulmapisteet. Jos kaikki kulmapisteet ovat ylemmän kuvan alueen sisällä, voidaan alempi kuva jättää piirtämättä. Tässä kuitenkin on hyvä ottaa huomioon, että jokainen tarkistus vie suoritusaikaa laitteiston prosessorilta, joten aina se ei ole kannattavaa. Huomasimme testeissämme, että toimenpide kannattaa suorittaa vain hyvin suurille kuville, jotka peittävät pääasiassa koko näytön kattaman alueen.

Suurin täyttönopeuden kuluttaja Lowglowissa oli pelin tausta, joka oli alunperin kolmesta 2048x2048 pikselin kokoisesta RGBA-kuvasta muodostuva kokonaisuus. Kuvat piirrettiin yksi kerrallaan päällekkäin ja niitä liikutettiin eri nopeutta pientä ympyrärataa saadaksemme syvyysvaikutelman aikaiseksi. Optimointivaiheessa yhdistimme kaksi alinta kuvatasoa samaan tekstuuriin. Lisäksi kirjoitin shader-ohjelman, joka osaa piirtää kaksi tekstuuria samalla piirtokerralla sekä liikuttaa niitä uniform-muuttujan avulla haluttua ympyräliikettä. Käytimme kyseistä shader-ohjelmaa taustan piirtämiseen. Tämä paransi ohjelman suoritusnopeutta huomattavasti.

8 EFEKTIEN TOTEUTUS LOWGLOWISSA

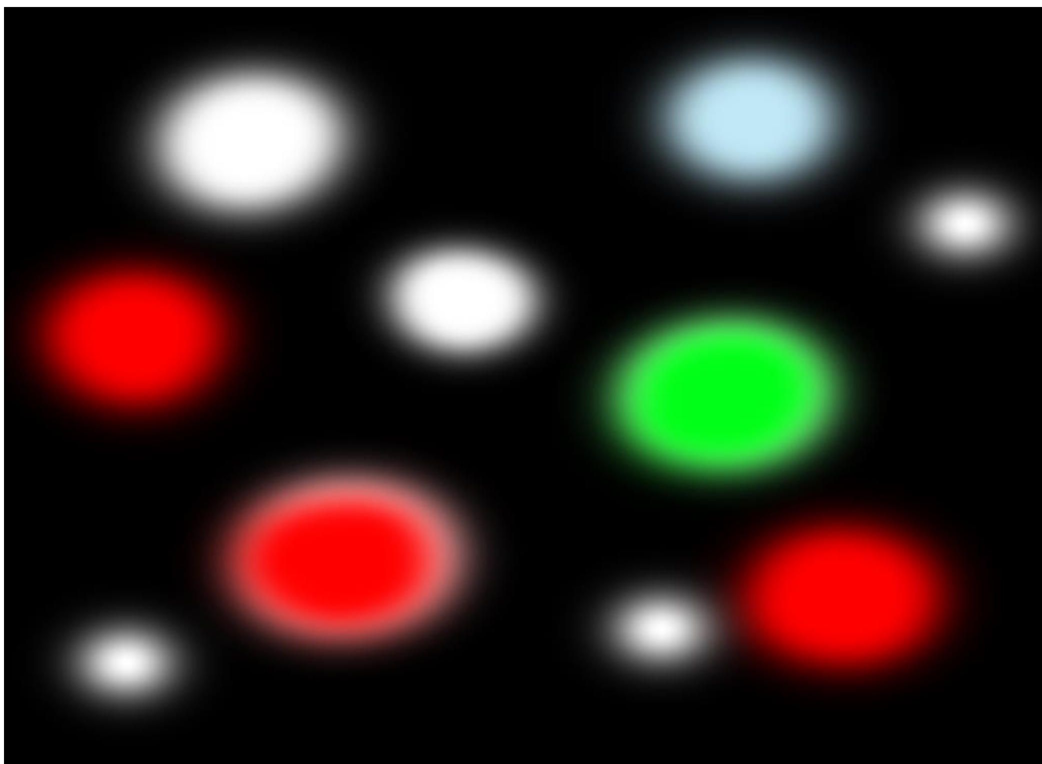
8.1 Valosysteemi

Lowglow'n valosysteemi toteutettiin kahden (2) render-tekstuurin (render texture) ja GLSL-ohjelman avulla. Valosysteemi toimii niin, että jokaisella peliobjektilla on erillinen valotekstuuri, joka piirretään valoille tarkoitettuun erilliseen render-tekstuuriin. Renderöityä valotekstuuria käytetään laskemaan lopulliset väriarvot shader-ohjelmassa valaistusefektin saamiseksi.

8.1.1 RTT

RTT (render to texture) on tekniikka, jossa piirtäminen tapahtuu näyttölaitteen sijasta näytönohjaimen muistissa olevaan tekstuuriin (Craitoiu, S. 2014). Tekniikaa voidaan käyttää hyödyksi sellaisissa tilanteissa, joissa efektin lopputulosta ei saada aikaiseksi piirtämällä suoraan näyttölaitteelle, vaan se täytyy suorittaa vaiheittain. Esimerkiksi suurin osa monivaiheisista jälkikäsittelysuodattimista vaatii RTT-tekniikan käyttöä, koska edellisen suodattimen lopputulos täytyy syöttää seuraavaan suodattimeen. Lowglow'n valosysteemi käyttää kahta (2) render-tekstuuria, joista toiseen piirretään kaikki peliobjektit ja toiseen kaikki valot.

Jokaisen kehyksen alussa kumpikin tekstuuri alustetaan täysin mustaksi (RGB 0,0,0) ja valo-objektit piilotetaan, jotta ne eivät piirry. Tämän jälkeen pelimaailma piiretään ensimmäiseen tekstuuriin. Ensimmäinen tekstuuri esittää tilannetta, jossa kaikki kohdat ovat täysin valaistuja. Seuraavaksi kaikki peliobjektit piilotetaan ja valo-objektit piirretään toiseen tekstuuriin, josta muodostuu valomaski. Valomaski esittää kohdat joissa esiintyy valoa. Täysin mustissa (RGB 0,0,0) kohdissa ei ole valoa ja täysin valkoisissa kohdissa (RGB 255,255,255) valo on vomakkaimmillaan.

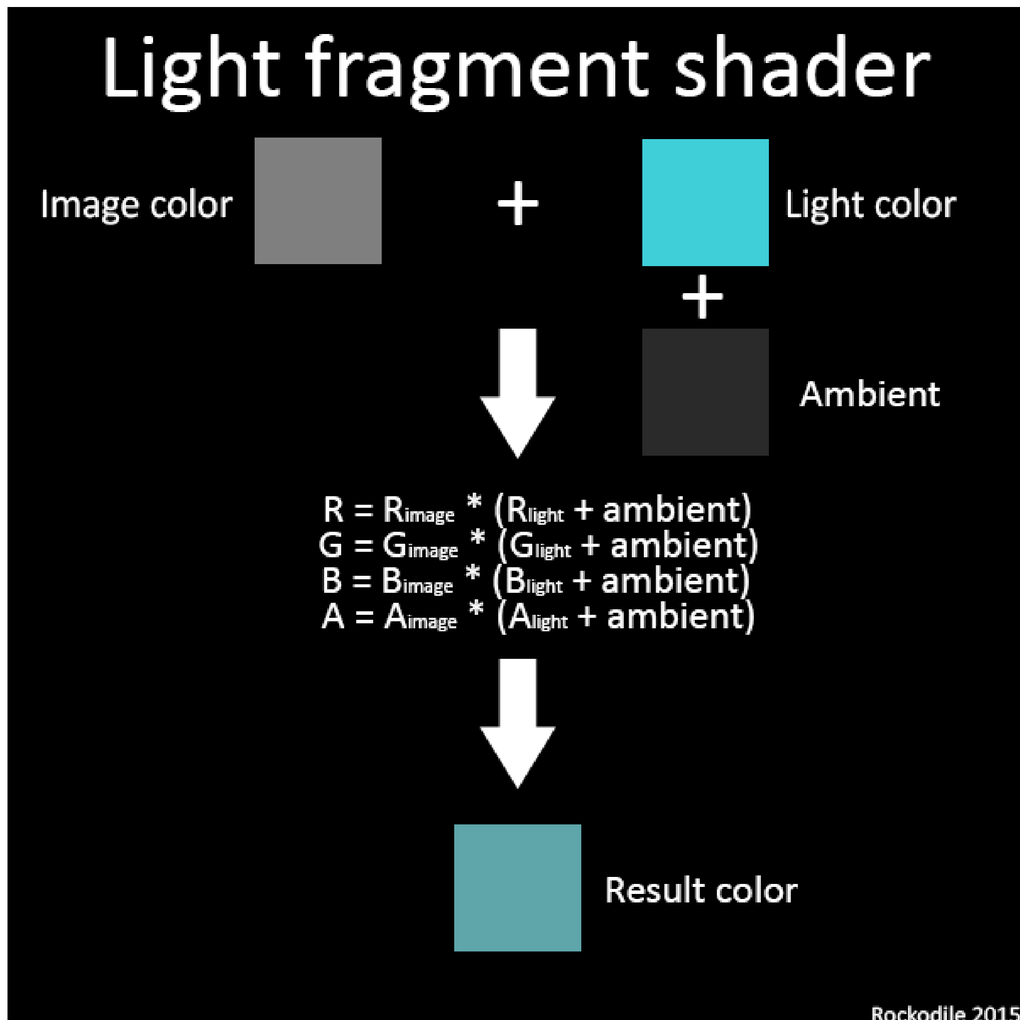


Kuva 8. Esimerkki valomaskista.

Ylläolevasta kuvasta näkee, miltä lopullinen valomaski voisi näyttää. Jokainen ellipsin muotoinen värialue kuvaa peliobjektin valoa, joka on piirretty tekstuuriin. Valot Lowglowissa on ennalta piirrettyjä mustavalkokuvia, jotka piirtovaiheessa asetetaan peliobjektin kanssa samaan paikkaan. Piirtyvän valon väriä voidaan vaihtaa muuttamalla pisteohjelmalle välitettävää pisteen väriä määrittävän muuttujan arvoa. Kun kaikki valot on piirretty tekstuuriin, siirrytään seuraavaan vaiheeseen.

8.1.2 GLSL-ohjelma valoille

Piirrettyjä render-tekstuureja ei itsessään piirretä mihinkään, vaan niitä käytetään apuna lopullisen pelikuvan piirtämisessä. Haluttu lopputulos saadaan aikaiseksi GLSL-pikseliohjelmalla. Pikseliohjelmaan syötetään uniform-muuttujina kumpikin render-tekstuuri, jotka on edellisessä vaiheessa valmisteltu shader-ohjelmaa varten.



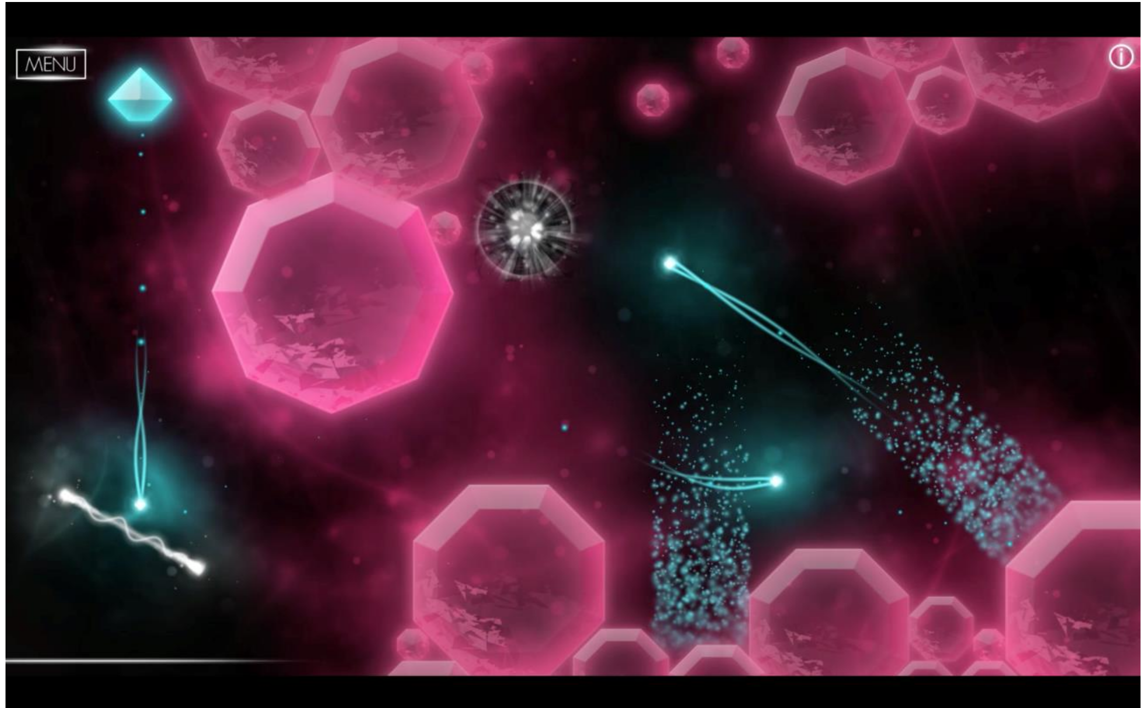
Kuva 9. Kuvasta ja valomaskista lopputuloksen saaminen.

Idea perustuu väriarvojen kertolaskuun. Ylläolevassa kuvassa on havainnollistettu yksinkertainen lopputuloksen laskeminen kertolaskulla. Jokainen värineliö kuvaa yhtä pikseliä tekstuurissa. Vasemmalla ylhäällä oleva pikseli on harmaa ja se on luettu ensimmäisestä tekstuurista, johon on piirretty pelimaailma. Pikseli halutaan yhdistää oikealla ylhäällä olevan kanssa, joka on luettu valomaskista. Lisäksi mukaan halutaan ambient-arvo, joka kuvaa valaistuksen tilaa, jossa ei ole yhtään valoa. Ambient-arvo on mukana siksi, että täysin valaisemattomat kohdat eivät näyttäisi täysin mustalta. Ambient-arvon takia myös täysin valaisemattomatkin kohdat näkyvät hiukan. Keskellä kuvaa näkyy kaava, jolla lopullisten värikanavien arvot lasketaan. Lopputuloksena on halutun värinen pikseli, joka piirretään näytölle.

```
varying vec2 v_texCoord;
```

```
uniform sampler2D u_texture1;

#define AMB_LIGHT 0.15

void main()
{
		vec4 color = texture2D(CC_Texture0, v_texCoord);
		vec4 mask = texture2D(u_texture1, v_texCoord);

   float AMBIENT = AMB_LIGHT-abs( (v_texCoord.x - 0.5))*AMB_LIGHT*0.5;
   AMBIENT =+ AMBIENT-abs( v_texCoord.y - 0.5)*AMB_LIGHT*0.5;

   gl_FragColor = vec4( color.r*(clamp(mask.r+AMBIENT, 0.0, 1.0)),
color.g*(clamp(mask.g+AMBIENT, 0.0, 1.0)), color.b*(clamp(mask.b+AMBIENT, 0.0,
1.0)), color.a*(clamp(mask.a+AMBIENT, 0.0, 1.0)) );
}
```

Ylläoleva GLSL-koodi on otettu Lowglow'n valosysteemistä. Ambient-arvo on määritelty arvoon 0.15. Color-muuttuja sisältää pikselin värin ensinmäisestä tekstuurista ja mask-muuttuja sisältää värin toisesta tekstuurista eli valomaskista. Laskuun mukaan otettava ambient-arvo lasketaan ennen sen käyttämistä värien laskuosiossa. Käytetyllä laskukaavalla ambient-arvo on suurempi ruudun keskiosassa ja pienempi ruudun reuna-alueilla. Halusimme luoda tällä tavoin ns. vingette-efektin, jolloin lopullinen kuva on kirkkaampi keskeltä kuin reunoilta. Lopuksi shader-ohjelma laskee piirrettävän pikselin väriarvon kertolaskukaavalla. Koska emme käyttäneet Lowglowissa HDR-renderöintitekniikkaa, väriarvot eivät saa ylittää arvoa 1, tulee lopputulos rajoittaa välille 0-1. Tämä tapahtuu GLSL-kielen clamp-funktiolla. Ilman ambient-arvon käyttöä rajoitusta ei tarvitse, koska kertolaskusta ei voi ikinä tulla arvoa, joka on yli 1.



Kuva 10. Kuva Lowglowista valojen GLSL-ohjelman suorituksen jälkeen.

Ylläolevasta kuvasta näkee miten valomaski vaikuttaa kuvan eri kohtiin. Paikat joissa valomaskissa on mustaa, ovat tummia lopullisessa kuvassa. Jokaisen valopisaran (siniset pisaran näköiset objektit) sekä ympäristöpalikan (punaiset 8-kulmion muotoiset objektit) ympärille on piirretty valomaskiin oikean värinen ja kokoinen kuva valosta.

8.2 Ympäristöpalikat

Kuva 11. Lowglow'n ympäristöpalikka.

Lowglow'n pelimekaniikat perustuvat pitkälti ympäristöpalikoihin, joihin valopisarat voivat törmäillä. Lowglowissa on 4 erilaista ympäristöpalikkatyyppiä, jotka kaikki käyttäytyvät eri tavalla. Koska palikat ovat pelin keskeisimpiä asioita, halusimme luoda niille muutamia tehosteita, jotta ne eivät tunnu liian staattisilta.

8.2.1 Tekstuurien pakkaus ja valinta-algoritmi

Kaikki ympäristöpalikat ovat ennalta piirrettyjä kuvia, jotka on pakattu neljään (4) eri tekstuuriin palikkatyypin mukaan. Pakattu tekstuuri sisältää kaikki muotovariaatiot palikoista.

Tekstuuria, joka sisältää useita eri kuvia kutsutaan kuva-arkiksi. Kuva-arkin koko on yleensä 2^n, koska jotkin näytönohjaimet voivat ladata muistiin vain kyseisen kokoisia tekstuureja (NPOT Texture 2016). Jos halutaan ladata kuva, jonka koko ei noudata kokoa 2^n, joutuu näytönohjain varaamaan kuvalle muistia seuraavaksi suurimman koon mukaan. Esimerkiksi, kun ladataan kuva jonka koko on 514 x 514, näytönohjain joutuu varaamaan muistia 1024 x 1024 kokoiselle tekstuurille, koska se on seuraava koko mihin kyseinen kuva mahtuu. Kuva-arkin ideana on, että monta eri kuvaa mahdutetaan yhteen tekstuuriin, jolloin varataan mahdollisimman vähän turhaa muistia. Käytimme Lowglowissa TexturePacker-nimistä sovellusta, joka osaa pakata useita yksittäisiä kuvia samaan kuvaan.

Pelinkentän luomisprosessissa käydään läpi kaikki ennalta kenttäeditorissa asetetut palikat. Kenttäeditorissa on asetettu palikan muoto, tyyppi, koko, paikka sekä suunta. Algoritmi, jonka suunnittelin, valitsee pakatusta tekstuurista lähimmän oikean kokoisen ja muotioisen kuva-alueen ja muodostaa siitä Cocos2d-x:n sprite-objektin. Tämän jälkeen sprite-objekti skaaltaan vastaamaan oikeaa kokoa, joka on asetettu kenttäeditorissa.

8.2.2 Palikoiden GLSL-ohjelma

Palikoiden shader-ohjelma käsittelee kaksi tehostetta. Ensimmäinen tehoste nostaa palikan värikylläisyyttä, riippuen kuinka lähellä valopisarat ovat sitä. Toinen tehoste aiheuttaa palikkaan aaltoiluefektin valopisaran törmättyä siihen. Alla on GLSL-koodi, jolla tehosteet toteutetaan.

```
#ifdef GL_ES
precision highp float;
#endif
```

```
varying vec2 v_texCoord;

uniform float u_amp;
uniform float u_opacity;
uniform float u_fade;
uniform float u_time;
uniform float u_saturation;

vec3 darken(vec3 color, float saturation)
{
   vec3 gray = vec3(dot(vec3(0.2126,0.7152,0.0722), color));
   return vec3(mix(color, gray, 1.0 - saturation));
}

void main()
{
   vec2 pos;

  //Aaltoilutehosteen tekstuurikoordinaatit
   pos.x = v_texCoord.x + sin((v_texCoord.y+u_time*0.25)*-100.0)*-u_amp;
   pos.y = v_texCoord.y + sin((v_texCoord.x+u_time*0.25)*-100.0)*-u_amp*2.0;

   vec4 color = texture2D(CC_Texture0, fract(pos));

   color.rgb = darken(color.rgb, u_saturation);

                gl_FragColor = color*u_opacity*u_fade;
}
```

Värikylläisyyden muuttaminen tapahtuu darken-funktiolla. Funktio laskee syötetystä väristä mustavalkoarvot vektorin pistekertoimen avulla. Mustavalkoarvot vastaavat syötettyä väriä, jonka värikylläisyys on 0. Parametri *saturation* kertoo mix-funktiolle alkuperäisen värin sekä harmaan värin sekoitussuhteen. *Saturation*-arvolla 0, lopputulokseksi saadaan väri, jossa ei ole yhtään värikylläisyyttä. Arvolla 1, saadaan

alkuperäinen väri. Käytimme tehosteen luomisessa negatiivisia *saturation*-arvoja, jolloin alkuperäinen väri saa lisää kylläisyyttä. Tehoste saa näyttämään kirkkaammilta ja värikkäämmiltä palikat, joiden läheisyydessä on valopisaroita.

Aaltoiluefekti perustuu tekstruurin koordinaattien muokkaamiseen sin-funktiota, aikaa ja amplitudia hyväksi käyttäen. Sin-funktion jakso rajoittuu välille 0-2 π, jonka avulla saamme helposti tehosteen toistamaan itseään pehmeästi tietyin aikavälein (Sine 2016). Aika syötetään shader-ohjelmaan u_time-muuttujan avulla. Arvo 100 on funktion taajuus. Taajuus vaikuttaa kuinka tiheästi sin-funktion jaksot sijaitsevat. Suuremmilla taajuusarvoilla tehoste luo tiheämmin aaltoja kohdekuvaan. Uniform-muuttuja u_amp on aallon amplitudi, eli aallon korkeus. Muuttujaa kasvatetaan aina kun valopisara törmää palikkaan, jolloin aaltoliike tulee näkyviin. Muutoin arvoa pienennetään ajan kuluessa, jolloin se saavuttaa lopulta arvon 0. Arvossa 0 aaltoliikettä ei enää näy, koska tekstuurin koordinaatit vastaavat alkuperäisiä.

9 YHTEENVETO

Päätöksemme valita OpenGL Lowglow'n grafiikkarajapinnaksi oli hyvä, sillä se tarjosi kaikista monipuolisimmat vaihtoehdot julkaisualustoille. Lisäksi Cocos2d-x-viitekehys auttoi huomattavan paljon pelimoottorin ohjelmoimisessa ja lyhensi kehitysaikaa. Haittapuolena oli se, että Cocos2d-x ei tue kunnolla HLSL-ohjelmia, joten jouduimme jättämään WindowsPhone-alustat pois Lowglow'n kohdealustoista. Harkinnassamme oli käyttää HDR-renderöintitekniikkaa sekä normal mapping -tekniikkaa, mutta päädyimme jättämään ne lopulta pois kokonaan, koska ne eivät sopineet pelin tyyliin.

GLSL-ohjelmien käyttö oli erittäin onnistunutta. Saimme tehtyä niillä paljon tehosteita, joita emme olisi saaneet toteutettua ilman shader-ohjelmia. Vaikka aluksi tehosteet olivat liian raskaita mobiililaitteille, saimme optimoitua niitä tarpeeksi, jotta ne toimivat myös kyseisillä alustoilla. Tärkeimmäksi optimoinnin kohteeksi osoittautui täyttönopeuden optimointi, koska lähes kaikki tehosteemme pohjautuivat pikselien piirtämiseen.

Valosysteemiä olisi voinut optimoida paremmin piirtämällä valot suoraa kuvapuskuriin tekstuurin sijasta, mutta tällä tavoin emme olisi saaneet ambient-arvon vingette-efektiä toteutettua. Vaihtoehtoinen ratkaisu olisi ollut, että käyttäisimme mobiilialustoilla optimoidumpaa tapaa ilman vingette-efektiä.

LÄHTEET

Metal Framework Reference 2016. Apple documentation. Viitattu 18.4.2016 https://developer.apple.com/library/tvos/documentation/Metal/Reference/MetalFrameworkReference/index.html

Cocos2d-x 2016. Cocos2d-x. Viitattu 18.4.2016 http://www.cocos2d-x.org/cocos2dx

Node Class Reference 2016. Cocos2d-x API reference. Viitattu 18.4.2016 http://www.cocos2d-x.org/docs/api-ref/cplusplus/v3x/d3/d82/classcocos2d_1_1_node.html#details

Craitoiu, S. 2014 Render To Texture in OpenGL. Viitattu 18.4.2016 http://in2gpu.com/2014/09/24/render-to-texture-in-opengl/

Kessenich, J.; Baldwin. D. & Rost, R. 2014. The OpenGL Shading Language. Viitattu 18.4.2016 https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

Vulkan 2016. Khronos group. Viitattu 18.4.2016 https://www.khronos.org/vulkan/

OpenGL Batch Rendering 2014. Krzeminski, M. Gamedev. Viitattu 18.4.2016 http://www.gamedev.net/page/resources/_/technical/opengl/opengl-batch-rendering-r3900

Normal Mapping 2016. Learn OpenGL. Viitattu 18.4.2016 http://learnopengl.com/#!Advanced-Lighting/Normal-Mapping

HLSL 2016. Microsoft. Viitattu 18.4.2016 https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

Enabling Antialiasing (Multisampling) 2016. MSDN. Viitattu 18.4.2016 https://msdn.microsoft.com/en-us/library/bb975403.aspx

Adaptive V-Sync 2016. NVIDIA. Viitattu 18.4.2016 http://www.geforce.com/hardware/technology/adaptive-vsync/technology

High-Resolution Antialiasing 2016. NVIDIA. Viitattu 18.4.2016 http://www.nvidia.com/object/feature_hraa.html

OpenGL Overview 2016. OpenGL. Viitattu 18.4.2016 https://www.opengl.org/about/

Shader 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Shader

Vertex Shader 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Vertex_Shader

Geometry Shader 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Geometry_Shader

Fragment Shader 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Fragment_Shader

Rendering Pipeline Overview 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Rendering_Pipeline_Overview

Uniform (GLSL) 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Uniform_(GLSL)

Performance 2016. OpenGL Wiki. Viitattu 18.4.2016 https://www.opengl.org/wiki/Performance

NPOT Texture 2016. OpenGL Wiki. Viitattu 18.4.2016
https://www.opengl.org/wiki/NPOT_Texture

Robert. 2012. Understanding the parallelism of GPUs. Viitattu 18.4.2016
http://renderingpipeline.com/2012/11/understanding-the-parallelism-of-gpus/

Sine 2016. Wikipedia. Viitattu 18.4.2016 https://en.wikipedia.org/wiki/Sine

Smalley, T. 2004. HDR Rendering with the NVIDIA GeForce 6800Ultra. Viitattu 22.4.2016.
http://www.bit-tech.net/gaming/pc/2004/11/03/farcry_patch13_eval/5

High Dynamic Range Rendering 2016. Unity3D Documentation. Viitattu 18.4.2016
http://docs.unity3d.com/Manual/HDR.html

Computer graphics 2016. Wikipedia. Viitattu 18.4.2016
https://en.wikipedia.org/wiki/Computer_graphics_(computer_science)

Graphics library 2016. Wikipedia. Viitattu 18.4.2016
https://en.wikipedia.org/wiki/Graphics_library

Frame rate 2016. Wikipedia. Viitattu 18.4.2016 https://en.wikipedia.org/wiki/Frame_rate

Multisample anti-aliasing 2016. Wikipedia. Viitattu 18.4.2016
https://en.wikipedia.org/wiki/Multisample_anti-aliasing

Shader 2016. Wikipedia. Viitattu 18.4.2016 https://en.wikipedia.org/wiki/Shader