Anssi Höykinpuro

OPENGL ES VARJOSTIMET KAJAK3D PELIMOOTTORISSA

Opinnäytetyö
Kajaanin ammattikorkeakoulu

Luonnontieteet

Tietojenkäsittely

Syksy 2010

Kajaanin ammattikorkeakoulu

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteet	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Anssi Höykinpuro	
Työn nimi OpenGL ES varjostimet Kajak3D pelimoottorissa	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Ohjaaja(t) Veli-Pekka Piirainen
	Toimeksiantaja Kajaanin ammattikorkeakoulu
Aika Syksy 2010	Sivumäärä ja liitteet 44+4

Älypuhelimien laskentateho kasvaa koko ajan, joka mahdollistaa grafiikan laadun kohenemisen ja uusien laskentatapojen hyödyntämisen. Teknologian kehittyessä myös mahdollisuus OpenGL ES piirtoprosessin ohjelmoimiseen avautui. Piirtoprosessin ohjelmointi käyttäen varjostinohjelmia mahdollistaa piirtojäljen muokkaamisen kehittäjän tarpeiden mukaiseksi. Varjostinohjelmien käyttäminen älypuhelimien piirtoprosessissa on yhä tuore asia, mutta muuttuu yhä yleisemmäksi ja tärkeämmäksi laitteiden kehittyessä.

Tämän opinnäytetyön tavoite on tutkia OpenGL ES varjostimien kehittämistä Kajak3D pelimoottorille. Teoria on hankittu varjostinohjelmien kehittämiseen keskittyneistä kirjoista ja OpenGL ES 2.0 varjostinkielen virallisesta määrittelystä. Kajak3D teoria on hankittu Kajak3D-projektin dokumentaatiosta, JSR-184 määrittelystä ja henkilökohtaisina tiedonantoina Kajak3D:n kehitystiimiltä. Tietoa varjostimien kehittämisestä on paljon, mutta vain osa tuosta tiedosta on hyödynnettävää OpenGL ES 2.0 varjostinohjelmien kehittämissä älypuhelinalustoille.

Opinnäytetyön ohella on kehitetty Fixed Function Pipeline-varjostin, joka tulee toimimaan oletusvarjostinohjelmana Kajak3D pelimoottorin OpenGL ES 2.x toteutuksessa. Varjostinohjelman tavoite on toteuttaa vastaava toiminnallisuus, jota OpenGL ES 1.x käyttää verteksi- ja fragmenttiprosessoinnissa. Varjostinohjelma noudattaa myös Kajak3D:n käyttämää JSR-184 määrittelyä.

Toteutettu varjostinohjelma on testattu osa-alue kerrallaan PC-ympäristössä käyttäen OpenGL ES 2.x emulointia. Testauksessa on luotu kaksi sovellusta käyttäen OpenGL ES 1.x ja OpenGL ES 2.x toteutusta, joiden piirtojäljen eroavuuksia on verrattu silmämääräisesti. Lisäksi varjostinohjelman toimivuus on testattu Maemo ja Bada käyttöjärjestelmillä varustetuilla älypuhelimilla. Toteutus osoittautui kiinteää piirtoprosessia hitaammaksi. Osa varjostinohjelman toiminnasta on todettu moitteettomaksi, mutta lopullinen piirtojälki kuitenkin poikkeaa tavoitteesta.

Kieli	Suomi
Asiasanat	opengl es, kajak3d, varjostimet, essl, verteksi, fragmentti, fixed function, älypuhelimet
Säilytyspaikka	☐ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

Kajaanin ammattikorkeakoulu
University of Applied Sciences

THESIS
ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Anssi Höykinpuro	
Title OpenGL ES shaders in Kajak3D game engine	
Optional Professional Studies Game programming	Instructor(s) Veli-Pekka Piirainen
	Commissioned by Kajaani University of Applied Sciences
Date Fall 2010	Total Number of Pages and Appendices 44+4

The processing power of smart phones is growing all the time, which allows the quality of graphics to improve and new calculating methods can be used. The advancement of technology has also opened the possibility to program graphics pipelines. Programming the graphics pipeline by using shader programs allows the developer to modify the result of rendering to what he or she wants it to be. The use of shader programs in smart phone graphics is still a fresh issue, but it is getting more common and important as devices advance.

The goal of this thesis is to study the development of OpenGL ES shader for the Kajak3D game engine. The theory is based on books that focus on shader program development and on the official OpenGL ES 2.0 shading language specification. The Kajak3D theory was acquired from the documentation of the Kajak3D game engine, JSR-184 specification and personal statements of the Kajak3D development team. There is a lot of information available about the development of shaders, but only a portion of that information is usable in the development OpenGL 2.0 shader programs for smart phones.

Along with the thesis a Fixed Function Pipeline-shader was developed, which will work as the default shader program in OpenGL ES 2.x implementation of the Kajak3D game engine. The goal of the shader program is to implement corresponding functionality to what OpenGL ES 1.x uses in vertex and fragment processing. The shader program also follows the JSR-184 specification that is also used by the Kajak3D.

The shader program was also tested a division at a time in PC environment using OpenGL ES 2.x emulation. Two applications using OpenGL ES 1.x and OpenGL ES 2.x implementations were created for testing. The rendering results of the applications were compared. In addition, the shader program functionality was tested on smart phones equipped with Maemo and Bada operating systems. The implementation turned out to be slower than the fixed graphics pipeline. A portion of the shader program functionality was found immaculate, but the final rendering result, however, differs from the goal.

Language of Thesis	Finnish
Keywords	opengl es, kajak3d, shaders, essl, vertex, fragment, fixed function, smart phones
Deposited at	☐ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

SISÄLLYS

SYMBOLILUETTELO

1 JOHDANTO 2

2 OPENGL ES VARJOSTIMIEN PERUSTEET 3
2.1 OpenGL ES 3
2.2 OpenGL ES varjostimet 6
2.2.1 Verteksivarjostimet 7
2.2.2 Fragmenttivarjostimet 8
2.3 ESSL varjostinkieli 9
2.4 Sovelluksen ja varjostimen välinen kommunikointi 11
2.5 Varjostimien käyttötarkoituksia 12
2.5.1 Normaalikartoitus 13
2.5.2 Kuvan jälkikäsittely 14
2.6 Varjostimien kehittäminen 15
2.6.1 Työkalut 16
2.6.2 Virheiden paikantaminen 16
2.6.3 Varjostimen nopeuden lisääminen 17

3 KAJAK3D PELIMOOTTORI 18
3.1 Kajak3D:n tukemat alustat 18
3.2 Koordinaatistot 19
3.3 Kajak3D:n varaamat muuttujat 22
3.4 Varjostimien käyttäminen Kajak3D:ssä 23
3.5 Varjostinohjelman toteuttaminen 24
3.5.1 Verteksivarjostin 25
3.5.2 Fragmenttivarjostin 25
3.5.3 Sovellus 26
3.6 Kuvan jälkiprosessoinnin toteuttaminen 27

4 KAJAK3D VARJOSTIMIEN TOTEUTUS 28
4.1 Fixed Function Pipeline-varjostimen kehitys 28
4.1.1 Muunnosvaihe 30
4.1.2 Valaistus 31

4.1.3 Tekstuuriympäristö 33
4.1.4 Värisumma 33
4.1.5 Sumu 34
4.1.6 Läpinäkyvyystestaus 35
4.2 FFP-varjostimen testaus 35
4.3 Jatkokehitysmahdollisuudet 38

5 POHDINTA 39

LÄHTEET 41

LIITTEET

SYMBOLILUETTELO

2D	Kaksiulotteinen koordinaatisto.
3D	Kolmiulotteinen koordinaatisto.
API	Application Programming Interface, ohjelmointirajapinta.
Binormaali	Pinnan myötäinen vektori, joka on samansuuntainen tangenttikoordinaatiston Y-akselin kanssa.
CPU	Central Processing Unit, tietokoneen prosessori.
Emulointi	Laitteen tai ohjelman toiminnan matkimista, eli jäljittelyä.
ESSL	Sulautetuille järjestelmille tarkoitettu GLSL:ään pohjautuva ohjelmointikieli.
Fragmentti	Yhden kuvapisteen kokoinen tietue, joka saattaa päästä kuvapisteeksi kuvapuskurille.
Fragmenttivarjostin	Varjostin joka toimii piirtoprosessin fragmenttiprosessoinnissa.
GLSL	OpenGL Shading Language on korkean tason ohjelmointikieli OpenGL varjostimille.
GPU	Graphics Processing Unit, tietokoneen näytönohjainpiiri.
Java	Alustariippumattomien sovellusten kehitysympäristö ja ohjelmointikieli.
JSR	Java Spesification Request, vaatimuslista Java määritykselle.
Kajak3D	Kajaanin ammattikorkeakoulussa kehitetty pelimoottori.
Kuvapuskuri	Piirtoprosessin tuottama sarja kuvapisteitä.
M3G	Mobile 3D Graphics API, määritys Java ohjelmien 3D grafiikan tuottamista varten.
Maailmakoordinaatisto	Koordinaatisto, johon 3D-mallit sijoitetaan.

Normaali	Pinnan kohtisuora vektori.
OpenGL	Open Graphics Library, avoin grafiikkakirjasto API.
OpenGL ES	OpenGL for Embedded Systems, sulautetuille järjestelmille suunnattu OpenGL API.
Sprite	Osittain läpinäkyvä animoitu kuvasarja, josta piirretään yksittäisiä kuvaruutuja kerrallaan.
Tangentti	Pinnan myötäinen vektori, joka on samansuuntainen tangenttikoordinaatiston X-akselin kanssa.
Tangenttiavaruus	Tangent Space on pinnan myötäinen koordinaatisto.
Tekseli	Tekstuurin kuvapiste.
Varjostin	Näytönohjaimessa ajettava pienohjelma.
VBO	Vertex Buffer Object, näytönohjaimen muistissa säilytettävä verteksitietue.
Verteksi	Kulmapiste joista 3D geometria muodostuu.
Verteksivarjostin	Varjostin joka toimii piirtoprosessin verteksiprosessoinnissa.

1 JOHDANTO

Älypuhelimien laskentateho on nousussa ja kykenee samaan kuin tietokoneet muutama vuosi sitten. Mooren lain mukaan mikropiirien laskentateho nousee vuotuisesti noin 40 - 60 %, joten yhä vaativampia toimenpiteitä on mahdollista toteuttaa. Älypuhelimet tarjoavat jo nyt tuen ohjelmoitavalle grafiikkaliukuhihnalle, joka mahdollistaa näyttävien visuaalisten tehosteiden luomisen. Näiden tehosteiden toteuttaminen on mahdollista varjostinohjelmilla, jotka korvaavat piirtoprosessin vaiheita. (Pulli 2008, 6)

Varjostinohjelmilla on mahdollista lisätä yksityiskohtien määrää piirrettävien mallien pinnoille, laskea kuvapisteentarkka valaistus ja muokata pintojen reagoimista valoon juuri halutunlaiseksi. Mikäli tavoiteltu tyyli ei yritä matkia todellisuutta, voidaan kappale piirtää esimerkiksi sarjakuvamaiseksi. Varjostinohjelmilla toteutettavat kuvan jälkiprosessointitehosteet voivat tarpeen tullen lisätä vaikkapa kuvan sumeutta. Varjostimia käyttäessä kehittäjä voi itse valita mitä hän haluaa piirtää. Vain taidot ovat rajana.

Tässä opinnäytetyössä käsitellään Kajak3D-pelimoottorille kehitettäviä varjostinohjelmia. Opinnäytetyö kertoo mitä tulee ottaa huomioon varjostimia kehittäessä, luo katsauksen ESSL varjostinkieleen sekä OpenGL ES grafiikkaohjelmointirajapintaan. Opinnäytetyön tavoite on perehdyttää lukija OpenGL ES varjostimien luomiseen Kajak3D käytössä. Opinnäytetyössä on toteutettu fixed function pipeline-varjostin, joka kiteyttää OpenGL ES 1.x toiminnallisuuden yhteen varjostimeen. Varjostinohjelman tavoite on toimia oletusvarjostimena Kajak3D ympäristössä ja näin ollen nopeuttaa sovellusten kehittämistä.

Opinnäytetyö on rajattu älypuhelimille kehitettäviin ESSL varjostimiin ja Kajak3D pelimoottorin käyttämiseen. Opinnäytetyö ei käsittele OpenGL ES toimintoja, vaan Kajak3D pelimoottorin toimintoja, joiden avulla varjostimien kehittäminen ja käyttäminen on mahdollista. Opinnäytetyö ei myöskään perehdy syvemmin mahdollisten varjostintyökalujen toimintaan, vaan luo katsauksen käytettävissä oleviin työkaluihin.

2 OPENGL ES VARJOSTIMIEN PERUSTEET

OpenGL ES on laajasti eri laitteilla käytössä oleva grafiikkarajapinta, jolla on myös mahdollista luoda näyttäviä varjostintehosteita käyttäen ESSL ohjelmointikieltä. Varjostinohjelmat mahdollistavat piirtoprosessin ohjelmoimisen ja kehittäjien ei tarvitse rajoittaa luovuuttaan valmiisiin laskentamalleihin.

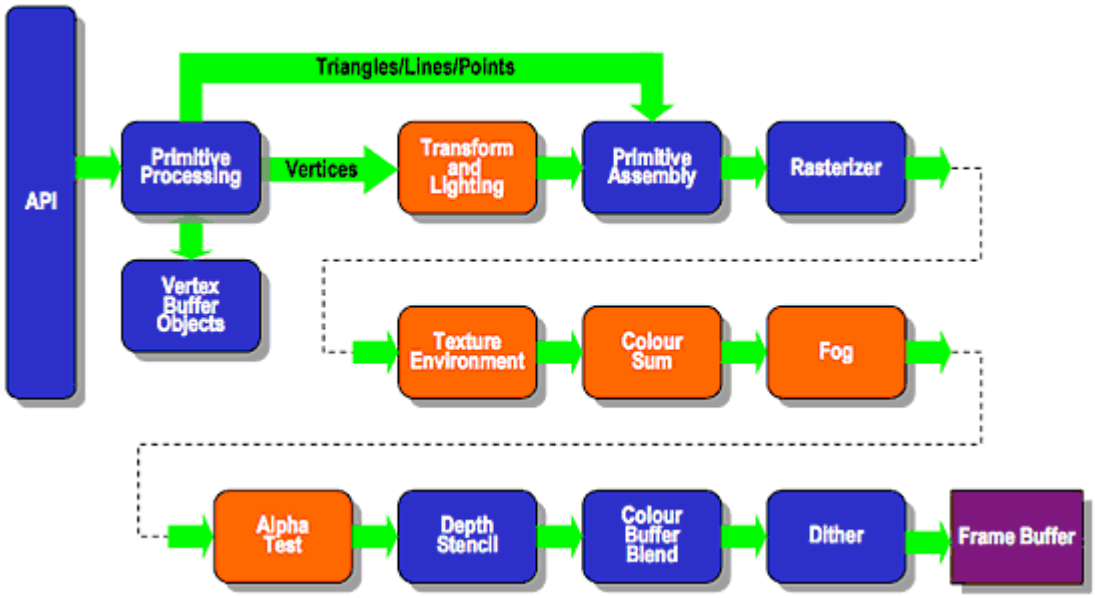
2.1 OpenGL ES

Open Graphics Library for Embedded Systems (OpenGL ES) on alustariippumaton ohjelmointirajapinta 2D- ja 3D-grafiikkasovelluksille. OpenGL ES tukemia alustoja ovat esimerkiksi älypuhelimet, navigointilaitteet, mediasoittimet sekä pelikonsolit. Suosituimmat älypuhelinkäyttöliittymät, kuten Symbian, iOS ja Android, ovat yksimielisesti omaksuneet OpenGL ES standardin käyttöönsä ja tukevat näin ollen kyseessä olevaa ohjelmointirajapintaa. (Khronos 2010 a)

OpenGL ES:n kehittäjä, Khronos Group, on vuonna 2000 perustettu yritysten yhteenliittymä, jonka tavoitteena on luoda avoimia alustariippumattomia ohjelmointirajapintoja ja mediaratkaisuja. Khronos Groupiin kuuluu yli 100 yritystä, joista mainittakoon AMD, Intel, nVidia, Nokia, Motorola ja Apple. (Khronos 2010 b)

OpenGL ES voidaan jakaa kahteen pääversioon: OpenGL ES 1.x ja OpenGL ES 2.x. OpenGL ES 1.x on kiinteän piirtoprosessin omaava ohjelmointirajapinta. Kiinteä piirtoprosessi tarkoittaa piirtoprosessia, jota voidaan muokata antamalla valmiille toiminnallisuudelle arvoja, mutta toiminnallisuutta ei ole mahdollista ohjelmoida. OpenGL ES 1.x on suunnattu laitteille, joiden laitetehot ovat vähäiset, kuten varhaiset älypuhelimet. OpenGL ES 2.x on edeltäjäänsä kehittyneempi ohjelmointirajapinta, jonka piirtoprosessi on ohjelmoitavissa. OpenGL ES 2.x tarjoaa siis mahdollisuuden käyttää varjostimia, mutta samalla poistaa mahdollisuuden käyttää OpenGL ES 1.x piirtoprosessia (Kuvio 1.). OpenGL ES 2.x on suunnattu kehitystä johtaville laitteille, jotka ovat yleensä myös varustettu OpenGL 1.x ajureilla. (Khronos 2010 a)

Piirtokäskyn lähdettyä sovellukselta on monta vaihetta, jotka suoritetaan ennen kuin piirrettävä kohde pääsee kuvapuskurille. Ohjelmointirajapinta (API) kutsuu piirtokäskyä, joka aloittaa piirtoprosessin. Tämän jälkeen kuvan piirtämiseen tarvittavat tiedot kulkevat piirtoprosessin läpi muokkautuen lopulliseksi kuvaksi.



Kuvio 1. OpenGL ES 1.x piirtoprosessia kuvaava kuvio. (Kuva: Khronos 2007 a, muokattu)

- Primitive Processing ottaa sovellukselta lähetetyt peli-kohtaiset mallin tiedot ja muuntaa ne vertekseiksi, joita verteksiprosessori voi käsitellä. (Penfold 2002)

- Verteksipuskuriobjektit (Vertex Buffer Object) (VBO) ovat tietueita, jotka säilyttävät verteksitietoa näytönohjaimen muistissa mahdollistaen nopeamman verteksien syötön objekteja piirrettäessä. (Pulli 2008, 180)

- Muunnos ja valaistus (Transform and Lighting) vaihe muuntaa sille saapuvat verteksit mallikoordinaatistosta silmäkoordinaatistoon. Verteksikohtainen valaistus lasketaan ja piirtoalueen ulkopuolelle jäävät verteksit hylätään. Valaistut verteksit muunnetaan kaksiulotteiseen Clip-koordinaatistoon. (Pulli 2008, 183)

- Primitive Assembly-vaihe koostaa saamiensa verteksien tiedoista geometriaa kuten pisteitä, viivoja, kolmioita tai monikulmioita. Kun vaihe on saanut täydennettyä geometrian vaatimat verteksit, lähetetään muoto seuraavalle vaiheelle. (Rost 2006, 14)

- Rasterointi (Rasterizer) vaihe muuntaa saamansa geometrian fragmenteiksi kuvapuskurissa, jolle kuva piirretään. Fragmentti on yhden kuvapisteen piirtämiseen vaadittava määrä tietoa. Lisäksi verteksikohtainen tieto interpoloidaan alueen fragmenteille.

- Tekstuuriympäristö (Texture Environment) lisää tekstuurin fragmentteihin, jos teksturointi on käytössä. (Pulli 2008, 195)

- Värisumma (Colour Sum) yhdistää fragmentin verteksi-, valaistus- ja tekstuurivärinarvon fragmentinväriksi.

- Sumu (Fog) vaihe häivyttää ja värjää kaukaisia fragmentteja, saaden näin aikaan ilmaperspektiivi-vaikutelma. (Pulli 2008, 210)

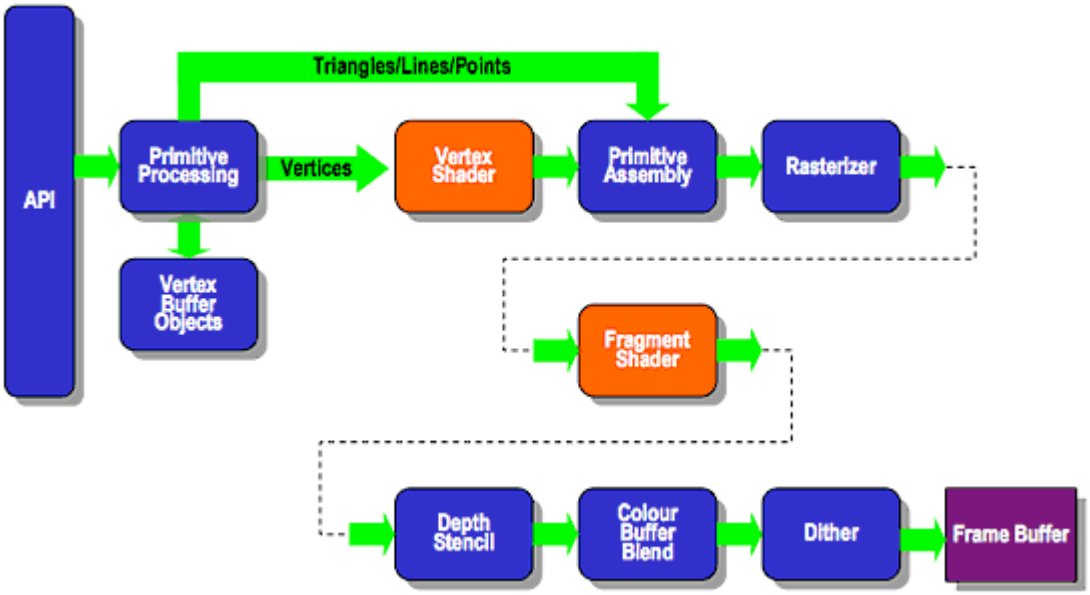
- Läpinäkyvyystestaus (Alpha Test) tarkastelee saapuvien fragmenttien alpha arvoa ja hylkää fragmentit jotka ovat liian läpinäkyviä. (Pulli 2008, 214)

- Syvyys sapluuna (Depth Stencil) tarkastelee fragmenttien syvyysarvoa ja päättää näin mikä fragmentti kuuluu päällimmäiseksi kuvapuskurissa. (Pulli 2008, 218)

- Väripuskuri sekoitus (Colour Buffer Blend) yhdistää saapuvan fragmentin väripuskuriin. (Pulli 2008, 219)

- Dither poistaa kuvaan mahdollisesti ilmentyneitä artifakteja lisäämällä kuvaan kerroksen joka toimii vastapainona virheväreille. (Pulli 2008, 220)

- Kuvapuskuri (FrameBuffer) on alue muistissa, jonne OpenGL:llä piirretty kuva säilötään. (Rost 2006, 6)



Kuvio 2. OpenGL ES 2.0 piirtoprosessia kuvaava kuvio. (Kuva: Khronos 2007 b, muokattu)

Osa OpenGL ES 2.0 piirtoprosessin vaiheista on poistettu ja ne ovat korvattavissa varjostimilla. Piirtoprosessin Transform and Lighting-vaihe on poistettu ja on korvattavissa verteksivarjostimella. Texture Environment, Colour Sum sekä Fog vaiheet ovat siirtyneet ohjelmoitavan fragmenttivarjotimen vastuulle (Kuvio 2.). Jotta piirto olisi mahdollista, tulee puuttuva toiminnallisuus korvata varjostimilla. (Ginsburg 2006; Khronos 2010 c; Pulli 2008, 18)

2.2 OpenGL ES varjostimet

OpenGL ES varjostimet ovat OpenGL ES piirtoprosessin ohjelmoitaviin vaiheisiin luotavia ohjelmia. Tällä hetkellä ohjelmoitavia vaiheita OpenGL ES piirtoprosessissa ovat verteksi- ja fragmenttiprosessit. Varjostinohjelmat poistavat kiinteän piirtoprosessin kahleet ja antavat kehittäjille vapautta ja mahdollisuuden luoda uusia piirtotehosteita. Varjostimien käytön pitäisi myös pienentää valmistuskustannuksia ja virrankulutusta pienlaitteilla. (Munshi 4, 7; Mäki 2007; Kessenich 2009, 11)

Varjostinohjelma, eli varjostin, koostuu verteksi- ja fragmenttivarjostimesta. Molemmissa varjostimista voi olla useampi varjostin yhdistettynä niihin, mutta vain yhdessä kunkin varjostintyypin lähdekoodissa saa olla main()-metodi. Kun lähdekoodit on yhdistetty verteksi- ja fragmenttivarjostimeen, ne käännetään ja niistä muodostuu objekteja, jotka voidaan liittää varjostinohjelmaan. Varjostinohjelmien lähdekoodit ovat tavallisia merkkijonoja. Yleensä

verteksi- ja fragmenttivarjostimet tulee säilyttää erillisissä tiedostoissa, joista ne luetaan sovellusta käynnistäessä. Varjostinohjelmat käännetään siis sovelluksen ajon aikana. (Munshi 2008, 28, 58; Rost 2007, 65)

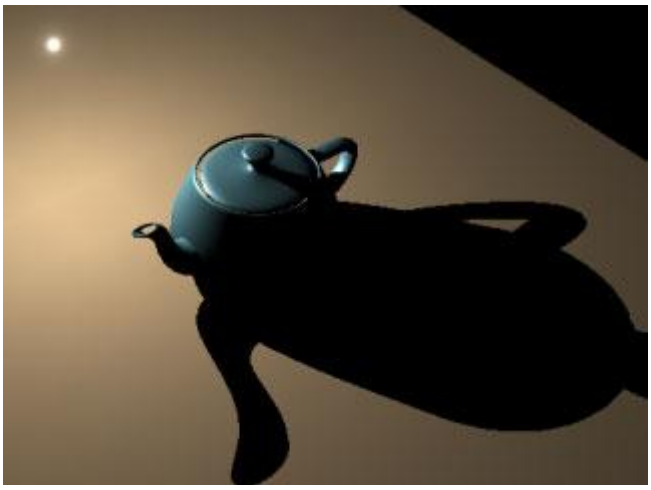
Esimerkiksi Zen Bound pelin jatko-osa Zen Bound 2 siirtyi käyttämään OpenGL ES 2.0:aa, joka mahdollisti pelin ulkoasun parantamisen varjostimien avulla (Kuvio 3.). Varjostimet luovat huomiotavan eron tarkkuudessa. Esimerkiksi punaiselle maalipinnalle saadaan näyttävämpi ja realistisempi vaikutelma, kun se reagoi valon kanssa.

Kuvio 3. Varjostimien tuoma etu on havaittavissa vertailtaessa piirtojälkeä kiinteän piirtoprosessin piirtojäljen kanssa. (Kuva: Touch Arcade 2010)

2.2.1 Verteksivarjostimet

Verteksivarjostimet (Vertex Shader) ovat verteksiprosessorissa jokaista piirrettävää verteksiä kohti ajettavia varjostimia. Verteksivarjostin muodostuu lähdekoodista, joka käännetään ja suoritetaan verteksiprosessorissa. Verteksiprosessori on ohjelmoitava prosessori, joka käsittelee saapuvia verteksejä ja niihin liitettyä tietoa. Verteksiprosessorissa käsitellään seuraavia

toimintoja: verteksimuunnos, normaalimuunnos, tekstuurikoordinaattien luominen, tekstuurikoordinaattien muunnos, valaistus sekä värimateriaalien asettaminen. Verteksivarjostin toimii verteksikohtaisesti, joten se ei voi toteuttaa toimintoja jotka vaativat muiden geometrian muodostavien verteksien tiedot. Verteksivarjostin soveltuu esimerkiksi verteksien muodostaman varjoalueen muodon laskemiseen. Esimerkiksi valonlähteen ja kahvipannun verteksien välisiä vektoreita pidentämällä voidaan laskea kahvipannun varjoa vastaavien verteksien sijainti (Kuvio 4.). (Kessenich 2009, 11; Rost 2007, 40; Simpson 2009, 8)



Kuvio 4. Verteksien muodostama varjoalue. (Kuva: Tsiombikas 2003)

2.2.2 Fragmenttivarjostimet

Fragmenttivarjostimet (Fragment Shader) ovat fragmenttiprosessorissa jokaista piirrettävää kuvapistettä kohti ajettavia varjostimia. Fragmenttivarjostin koostuu lähdekoodista, joka käännetään ja suoritetaan fragmenttiprosessorissa. Fragmenttiprosessori on ohjelmoitava prosessori, joka käsittelee fragmenttien arvoja ja niihin liittyvää tietoa. Fragmenttiprosessorin toimintoja ovat värisummien laskenta, tekstuuritiedon käsittely sekä sumun lisääminen. Fragmenttivarjostimen päätehtävä on määrittää fragmentin väriarvo. Fragmenttivarjostin ei voi muuntaa fragmentin sijaintia tai käsitellä viereisiä fragmentteja. Fragmenttivarjostin voi lukea myös sille lähetettyjen tekstuurien tietoa. Aaltoileva merenpinta voidaan piiretää fragmenttivarjostimessa käyttäen korkeuskarttana toimivaa tekstuuria (Kuvio 5.). (Kessenich 2009, 11; Polysplendor 2009; Rost 2007, 40; Simpson 2009, 8)

Kuvio 5. Yksinkertainen vedenpinta on mahdollista toteuttaa käyttäen korkeuskarttana toimivaa tekstuuria. (Kuva: Polysplendor 2009)

2.3 ESSL varjostinkieli

OpenGL ES Shading Language (ESSL, GLSL ES) on OpenGL Shading Languagen (GLSL) pohjalta kehitetty varjostimien ohjelmointikieli sulautetuille järjestelmille. ESSL koostuu kahdesta erittäin samankaltaisesta kielestä, joita yleisesti käsitellään samana kielenä. Nämä kielet ovat verteksi- ja fragmenttivarjostimien kehittämistä varten luotuja ohjelmointikieliä. Kielet ovat erillisiä, koska verteksi- ja fragmenttivarjostimen tavoitteet ja tietojen käsittelyoikeudet poikkeavat toisistaan. Niitä kuitenkin pidetään samana kielenä, koska poikkeama kielten välillä on vähäinen. ESSL kuten myös GLSL pohjautuvat C-ohjelmointikieleen, joten ne ovat syntaksiltaan hyvin samankaltaisia C-ohjelmointikielen kanssa. Muita varjostinkieliä ovat High Level Shading Language (HLSL) sekä C for Graphics (Cg), mutta nämä jäävät aiherajauksen ulkopuolelle. (Munshi 2008, 78; Simpson 2010, 8)

ESSL muistuttaa hyvin paljon C-ohjelmointikieltä, mutta joitakin eroavuuksia on olemassa. Esimerkiksi muuttujat vaativat tarkkuuden määrittelyn esittelynsä yhteydessä ja uusia muut-

tujatyyppejä on käytettävissä. Lisäksi koodin tulisi olla mahdollisimman yksinkertaista nopeuden vuoksi.

ESSL kielessä on myös mahdollista käyttää esikääntäjiä, joiden avulla on mahdollista muokata varjostinohjelman lähdekoodia käännön yhteydessä. Esikääntäjiä ovat esimerkiksi #define, #ifdef, `#else` ja #endif. Esikääntäjä #version määrittelee varjostinkielen version, jota tulee käyttää kääntäessä varjostinta. Mikäli vaadittu versio on 1.00, tulee esikääntäjän arvon olla 100. (Munshi 2008, 92)

Muuttujatyyppejä ESSL-kielessä ovat perinteiset void, bool, int ja float. Vektoreita voi käsitellä vec2, vec3 ja vec4 muuttujilla. Vektoreissa voi myös käyttää totuusarvomuuttujia, bvec2, bvec3 ja bvec4, sekä kokonaislukuja, ivec2, ivec3 sekä ivec4. Matriiseja voidaan käsitellä mat2, mat3 sekä mat4 muuttujina. (Simpson 2009, 18)

Rakenteiden ja taulukoiden käyttö on mahdollista, mutta molempien käytössä on rajoituksia. Rakenteiden sisällä ei saa esitellä toisia rakenteita. Taulukon alkiot täytyy asettaa yksitellen ja taulukkoon viitatessa indeksinä ei voi käyttää muuttujia, jotka eivät ole tiedossa varjostimen kääntöhetkellä. Jotkin alustat saattavat tukea erilaista toimintatapaa, mutta seurauksena saattaa seurata muistivuotoja ja epävakautta. Silmukoiden käyttö on sallittua, mutta rajoitettua. Iteraattoreita saa olla vain yksi ja sitä täytyy kasvattaa tai pienentää silmukan määrittelyssä, mutta sen arvoa ei saa muuttaa silmukan sisällä. Silmukan lopetusehdon täytyy olla vertailu iteraattorin ja muuttumattoman arvon välillä. (Munshi 2008, 82-84, 87; Simpson 2009, 23-24)

Metodien kirjoittamisessa on myös poikkeama C-ohjelmointikieleen verrattaessa. Metodit eivät voi olla rekursiivisia. Metodin parametrit voi määritellä in, inout tai out määritteillä. Metodi ei voi muokata in parametreja, mutta kykenee muokkaamaan inout ja out parametreja. Out määritteellä varustettu parametri ei ota muuttujan arvoa metodia kutsuttaessa, mutta palauttaa muokatun arvon. Kielessä on myös lukuisia sisäänrakennettuja metodeja, jotka käsittelevät kulmia, trigonometriaa, eksponentteja, geometriaa, matriiseja, vektoreita, tekstuureita sekä yleisiä toimintoja. (Munshi 2008, 85-86; Simpson 2009, 63-72)

Huomioitava eroavaisuus perinteisen GLSL:n ja ESSL:n välillä on se, että jälkimmäinen vaatii muuttujien tarkkuuden määrittelyn. Muuttujan tarkkuuden määritteleminen alhaiseksi mahdollistaa nopeampia matemaattisia operaatioita muuttujien vaatiessa epätarkempia arvoja. Tarkkuus voidaan määritellä liuku- ja kokonaisluvuille alhaisesta korkeaan käyttäen avainsanoja lowp, mediump tai highp. Huomioitavaa kuitenkin on, että OpenGL ES määrittely ei

vaadi näiden eri tarkkuustasojen tukemista. Tästä johtuen kaikki näytönohjaimet eivät välttämättä hyödy tarkkuuden määrittelystä, vaan käyttävät aina korkeaa määritystä. Muuttujille voidaan myös määritellä oletustarkkuus, jolloin muuttujien tarkkuutta ei ole pakko määritellä jokaisen muuttujan kohdalla (Kuvio 6.). Verteksivarjostimen oletustarkkuus on highp, mutta fragmenttivarjostimen oletustarkkuutta ei ole määritelty. (Munshi 2008, 96)

```
// Oletustarkkuuden määritteleminen liukuluvuille
precision mediump float;
```

Kuvio 6. Oletustarkkuuden määritteleminen liukuluvuille.

2.4 Sovelluksen ja varjostimen välinen kommunikointi

Kommunikointi sovelluksen ja varjostimen välillä on mahdollista, mutta vain sovellukselta yksisuuntaisesti varjostimelle. Tämä johtuu siitä, että varjostinohjelma voi ulostulona ainoastaan piirtää pinnoille, kuten väri- ja syvyyspuskurille. GLSL varjostimelle voi lähettää arvoja OpenGL:n tilaa konfiguroimalla. Tämä ei kuitenkaan ole mahdollista OpenGL ES:ssä, sillä suurin osa sisäänrakennetuista uniform-muuttujista on poistettu. (Ginsburg 2007; Lighthouse 3D 2010)

Käyttäjän määrittelemät muuttujat voidaan jakaa kahteen ryhmään; uniform- ja attribute-muuttujiin. Uniform-muuttujat ovat käytettävissä verteksi- ja fragmenttivarjostimessa. Uniform-muuttujan arvo on sama jokaista verteksiä ja kuvapistettä kohden. Attribute-muuttujan arvot taas ovat verteksikohtaisia ja niitä voi lukea ainoastaan verteksivarjostimessa. Verteksivarjostin voi myös lähettää varying-muuttujia fragmenttivarjostimelle. Varying-muuttujat ovat verteksivarjostimessa verteksikohtaisia muuttujia, jotka interpoloidaan fragmenttikohtaisiksi muuttujiksi fragmenttivarjostinta varten. (Simpson 2009, 31; Lighthouse 3D 2010; Munshi 2008, 90)

Uniform-muuttujat ovat OpenGL ES 2.0 ohjelmointirajapinnalta varjostinohjelmalle lähetettäviä muuttujia. Uniform-muuttujia voidaan lukea sekä verteksi- että fragmenttivarjostimesta käsin. Uniform-muuttujat jakavat saman nimiavaruuden, joten samannimiset uniform-muuttujat verteksi- ja fragmenttivarjostimessa viittaavat samaan arvoon. Näin ollen samannimiset uniform-muuttujat tulee olla määritelty samalla tarkkuudella ja samantyyppisiksi. Vaikka uniform-muuttujat ovat luettavissa ainoastaan varjostinohjelmassa, niiden sisältämiä arvoja voidaan muuttaa sovelluksesta käsin. Näin ollen uniform-muuttujat soveltuvat parhai-

ten arvoille, jotka ovat staattisia jokaista piirrettävän kappaleen verteksiä tai fragmenttia kohden. Koko kappaleeseen vaikuttavia muuttujia ovat esimerkiksi transformaatiomatriisi sekä valonlähteiden sijainnit. (Munshi 2008, 88; Simpson 2009, 31)

Attribute-muuttujat säilövät verteksikohtaisia arvoja ja soveltuvat hyvin verteksien sijaintien, värien, normaalien sekä tekstuurikoordinaattien säilyttämiseen. Attribute-muuttujat voivat kuitenkin sisältää mitä tahansa käyttäjän haluamaa verteksikohtaista dataa. Attribute muuttujat ovat käytettävissä ainoastaan verteksivarjostimessa. (Munshi 2008, 89)

Varying-muuttujat ovat verteksivarjostimen tulosteita ja fragmenttivarjostimen syötearvoja. Muuttujan arvo interpoloidaan rasterointiprosessissa jokaista fragmenttia kohden ja tästä syystä muuttujat soveltuvat esimerkiksi tekstuurikoordinaattien lähettämiseen sekä gouraud-varjostuksen luomiseen. (Munshi 2008, 90)

Samplerit ovat ESSL:ssä osoittimia tekstuureihin ja niitä voidaan käsitellä ainoastaan fragmenttivarjostimesta käsin. Perinteinen kaksiulotteinen tekstuuri tunnetaan nimellä sampler2D. Samplereita voi esitellä ainoastaan metodin parametrina tai uniform-muuttujana. Samplereita voi käsitellä ESSL:n sisäänrakennetuilla funktioilla, kuten texture2D(). Metodi hakee määritellystä sampler2D tekstuurin kohdasta tekselin, eli tekstuurin kuvapisteen, arvon. (Simpson 2009, 22, 71)

Varjostimen suunnittelija päättää miten varjostin käsittelee tekstuurin sisältämää dataa, joten sitä voidaan myös käyttää muuhunkin kuin kuvadatan säilyttämiseen. Tekstuuria voidaan käyttää myös spekulaari- (Specular Map), emissio- (Emissive Map), normaali- (Normal Map) ja ympäristökarttana (Environment Map). (Lighthouse 3D 2010; Olsson 2008)

2.5 Varjostimien käyttötarkoituksia

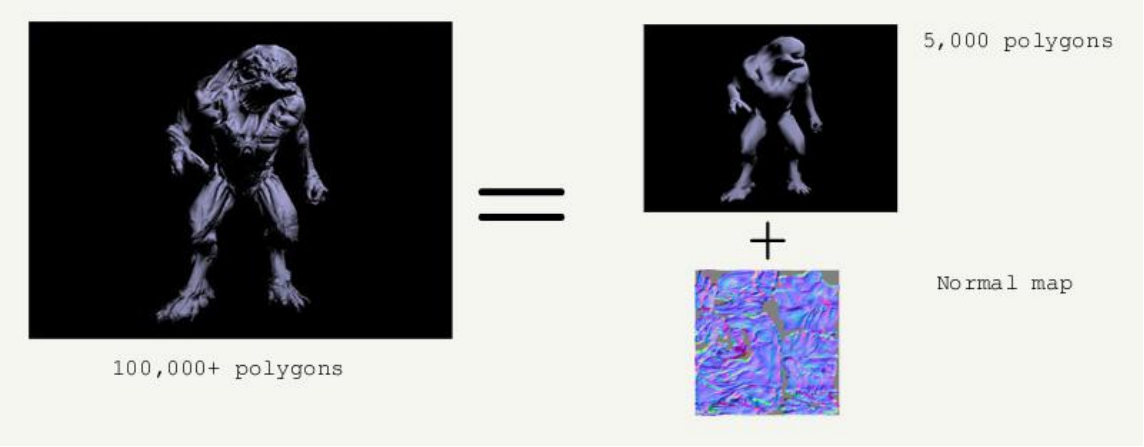
Varjostimilla on mahdollista luoda monenlaisia tehosteita. Käyttötarkoituksia ovat muun muassa kuvapistekohtaisen valaistuksen ja sarjakuvamaisen värityksen tuottaminen. Varjostimilla on mahdollista tehdä mallista yksityiskohtaisemman näköinen käyttäen normaalikartoitusta. Lisäksi varjostimilla voidaan käsitellä piirrettyä kuvaa, joka mahdollistaa koko piirtoalueen kattavien tehosteiden käytön.

2.5.1 Normaalikartoitus

Pinnan pisteen kirkkauden laskemista varten tarvitaan kaksi vektoria: pinnan normaali ja vektori pinnasta valonlähteeseen. Näiden kahden vektorin välinen kulma, eli pistetulo, määrittää pinnan pisteen valaistuksen. Kulman ollessa pieni, valo ja normaalivektori osoittavat samaan suuntaan ja näin ollen myös valo osoittaa kohti pintaa, jolloin piste on kirkkaasti valaistu. Kun taas kulma on suuri, pinta osoittaa poispäin valosta ja piste on heikosti valaistu. Näin kappale saadaan varjostettua, mutta vähäisillä polygonimäärillä varjostus saattaa näyttää karulta. Luomalla lisää yksityiskohtia kappaleeseen polygonien määrä nousee, joka ei taas ole suotavaa reaaliaika sovelluksissa kuten videopeleissä. (Cloward 2006)

Kappaleen pinnalle voidaan luoda lisää yksityiskohtia normaalikartoituksella kasvattamatta polygonimäärää (Kuvio 7.). Normaalikartta on tavallinen tekstuuri, joka säilyttää tiedot pinnan pisteiden normaaleista. Tekstuurin R, G ja B värikanavat säilövät normaalivektorin X, Y ja Z arvot ja niitä voidaan käyttää pisteen kohdalla pinnan normaalin sijaan. (Cloward 2006)

Pinnan pisteen valoisuus on vektoreiden N ja L pistetulo, jossa L on vektori pisteestä valonlähteeseen ja N on pisteen normaali. Pisteen normaali N lasketaan normaalikartasta poimitun normaalin ja pinnan normaalin avulla. (Cloward 2006)

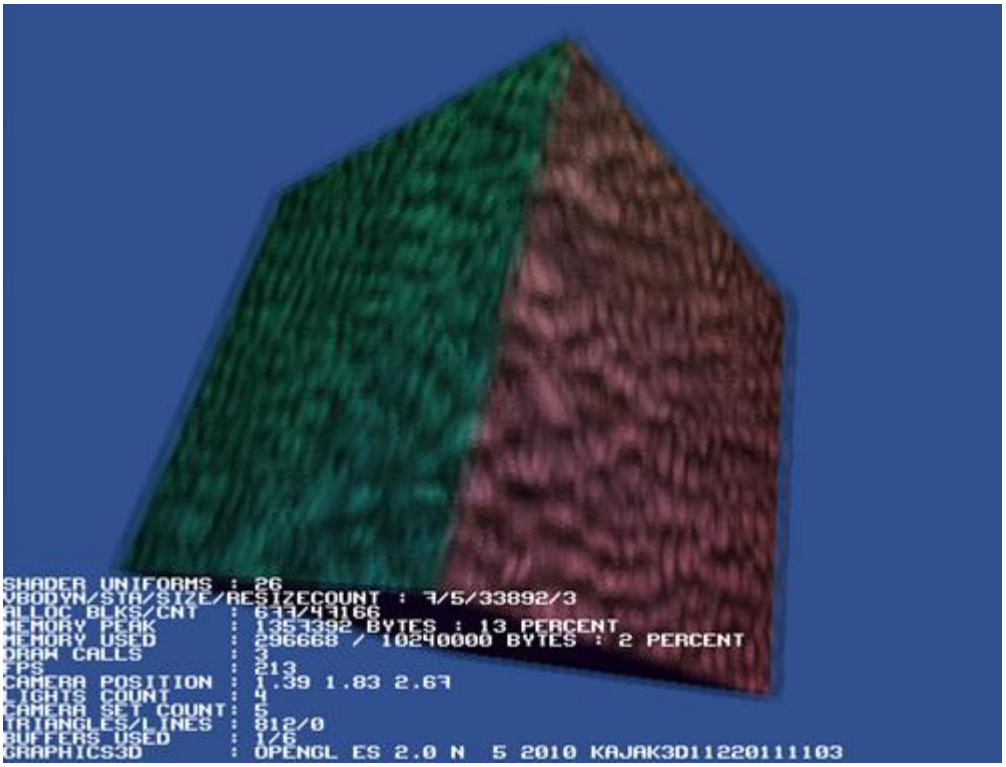


Kuvio 7. Korkean monikulmiomäärän 3D-malli voidaan piirtää reaaliajassa käyttämällä alhaisen monikulmiomäärän mallia ja laskemalla valaistus käyttäen normaalikarttaa pinnan muotojen kuvaamiseen. (Kuva: Sanglard 2004)

2.5.2 Kuvan jälkikäsittely

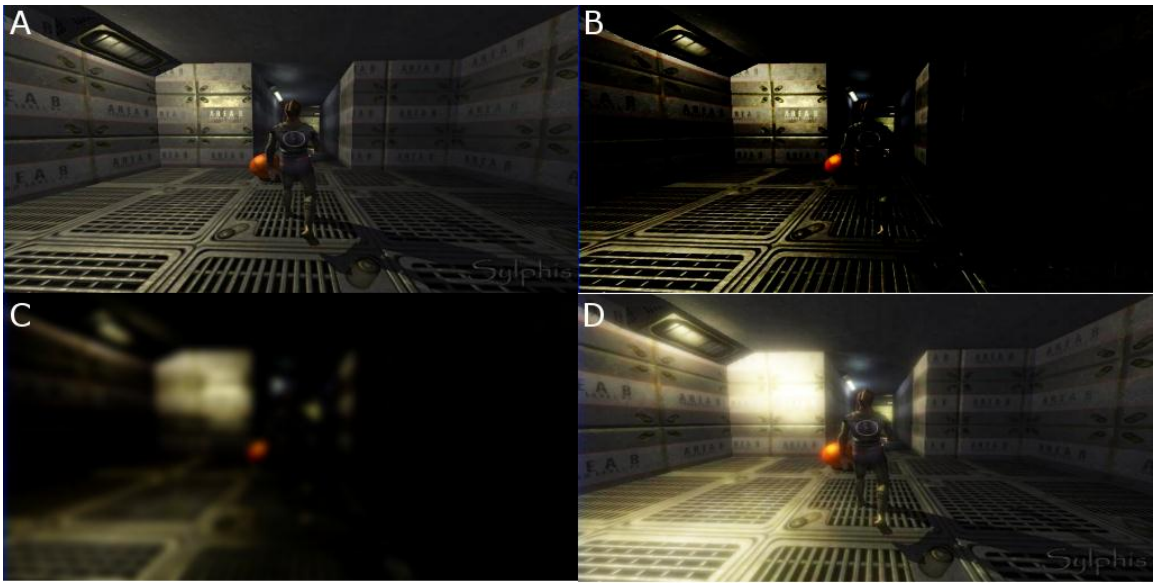
Kuvan jälkikäsittelyllä (Image Postprocessing) voidaan luoda koko näytön kattavia erikoistehosteita. Kuvan jälkikäsittely on mahdollista piirtämällä kohde toiselle kuvapuskurille. Tämän jälkeen kuvapuskuri piirretään näytön koko alueelle venytettynä. Piirtovaiheessa käytetään kuvan jälkiprosessointiin suunniteltua varjostinohjelmaa. (Munshi 2008, 296-297)

Blur-varjostimen tavoitteena on sumentaa piirrettävää pintaa. Yleisimmin tätä tehostetta käytetään kuvan jälkiprosessoinnissa. Varjostin ottaa piirrettävän tekselin ympäriltä näytteitä muista väreistä ja laskee näiden keskiarvon sumentaen samalla piirrettävää kuvaa. Yksinkertainen blur-varjostin voi olla toteutettavissa laskemalla neljän fragmenttia ympäröivän tekselin värin keskiarvo (Kuvio 8.). Blur-varjostimen ollessa fragmenttipainotteinen se vaatii myös monta laskutoimitusta, etenkin jos sitä käytetään kuvan jälkiprosessoinnissa. Jotta varjostinohjelma toimisi mahdollisimman nopeasti, on mahdollista piirtää sumennettu kuvapuskuri kuvaruudun oikeaa kokoa pienemmälle tekstuurille. Näin sumennuksen laatu paranee ja sen toiminnallisuus nopeutuu. Sumennuksen laatua on mahdollista myös parantaa lisäämällä tekstuurista otettavien näytteiden määrää, joka on kuitenkin puolestaan raskaampaa. (Kalogirou 2006; Munshi, A. 2008. 297)



Kuvio 8. Blur-varjostin toiminnassa.

Blur-varjostimen pohjalta on myös mahdollista kehittää esimerkiksi Bloom-varjostin. Bloom-varjostimella on mahdollista luoda erittäin kirkkaan näköisiä alueita. Koska näytöistä tulevan valon määrä on rajattua, täytyy käyttäjälle uskotella toisella tavalla että alue on muita kirkkaampi. Tämä voidaan tehdä sumentamalla valaistua aluetta, jolloin luodaan vaikutelma korkeasta valoisuudesta. Bloom-tehoste voidaan saavuttaa sekoittamalla normaalisti piirrettävän ympäristön kuvapuskuria ja sumennettua valoisuus-tekstuuria, johon ympäristön valaistut kohdat on merkitty (Kuvio 9.). (James 2004; Kalogirou 2006)



Kuvio 9. Bloom-tehosteen luominen vaiheittain. (Kuva: Kalogirou 2006, muokattu)

2.6 Varjostimien kehittäminen

Varjostimien kehittäminen vaatii tuotoksen ymmärtämistä ennen ohjelmakoodin kirjoittamista. Ennen varjostimen kirjoittamista tulee tietää mitä sen on tarkoitus tehdä ja kuinka se toteutetaan. Käytettävät matemaattiset metodit tulee ymmärtää ja aina tulee pyrkiä yksinkertaisuuteen. Varjostinohjelma tulee rakentaa asiakokonaisuus kerrallaan ja lisätä monimutkaisuutta vasta toimivuuden varmistuttua. Esimerkiksi ensin tulee varmistaa että malli piirtyy muodoltaan oikein kuvapuskurille, ennen kuin lisää tekstuurikoordinaattien käsittelyn ja valolaskelmat mukaan. Muutosten testaaminen lyhyin väliajoin auttaa myös varmistamaan että varjostin toimii oikein. (Rost 2007, 216-217)

2.6.1 Työkalut

Varjostinohjelmien ollessa pelkkiä merkkijonoja ne on mahdollista toteuttaa lähes millä tahansa tekstinkäsittelyohjelmalla. Esimerkiksi tämän opinnäytetyön ohella luodut varjostimet on kirjoitettu Notepad++ ohjelmalla, joka tarjoaa yksinkertaisia helpotuksia kuten avainsanojen värjäämisen ja rivinumeroinnin.

Mali GPU Shader Development Studio on OpenGL ES 2.0 varjostimien kehitysohjelma, jolla voi kehittää ja kokeilla varjostimien toimintaa eri arvoilla. Opinnäytetyö ei kuitenkaan käsittele ohjelman toimintaa. HLSL ja GLSL varjostinohjelmia voi kehittää NVIDIA FX Composer ja AMD RenderMonkey ohjelmilla. Nämä ohjelmat eivät tue ESSL varjostimia, mutta soveltuvat varjostimien suunnittelemiseen ja arvojen kokeilemiseen. Ohjelmat tarjoavat myös näkymän luodusta varjostinohjelmasta ja reaaliaikaisen uniform-arvojen muuttaminen. (ARM 2010, 2; AMD 2008; NVIDIA 2009)

Tärkein työkalu on kuitenkin OpenGL ES 2.0 emulointikirjastot, jotka mahdollistavat ESSL varjostimien kehittämisen PC ympäristössä.

2.6.2 Virheiden paikantaminen

OpenGL ES:n havaitsemat virheet varjostinohjelman lähdekoodissa tulostetaan Kajak3D konsoli-ikkunaan varjostinohjelman käännön yhteydessä. Nämä virheet ovat helppoja korjata, sillä virheilmoitus kuvailee virheen selkeästi ja mainitsee myös verteksi- tai fragmenttivarjostimen lähdekoodin rivin numeron, jolla virhe on tapahtunut.

Vaikeampaa sen sijaan on paikantaa virheitä varjostinohjelman toiminnallisuudesta. Kajak3D ei ilmoita, jos olemattomaan varjostimen muuttujaan yritetään lähettää arvoa. Arvoa ei vain tuolloin lähetetä. Virhettä ei myöskään ilmoiteta jos muuttujaan ei koskaan aseteta arvoa. Varjostinohjelma toimii, mutta arvot eivät ole oikeita. Varjostinohjelmasta on joskus hyvin vaikea sanoa saavuttaako se koskaan tiettyä kohtaa varjostimen toiminnallisuudessa. Varjostinohjelma ei voi tulostaa tekstiä konsoli-ikkunaan, vaan ainoa ulostulo varjostimesta on kuvapuskurille. Näin ollen kuvapuskurille voidaan tulostaa jotain poikkeavaa virheen sattuessa tai varjostinohjelman saavuttaessa tietyn pisteen. (Rost 2007, 220-221)

Virheellisten arvojen tarkistaminen voidaan toteuttaa if-vertailulla, jonka ollessa tosi ajetaan poikkeava toiminnallisuus. Verteksivarjostimessa tapahtuvia virheitä voidaan tarkkailla muuttamalla verteksien sijaintia tai lähettämällä fragmentin väriarvo fragmenttivarjostimelle. Fragmenttivarjostimessa määriteltyjen olosuhteiden ollessa tosia, voidaan fragmentti värittää poikkeavan väriseksi tai hylätä koko fragmentti. (Rost 2007, 221-222)

2.6.3 Varjostimen nopeuden lisääminen

Fragmenttivarjostimet soveltuvat tarkkojen yksityiskohtien tuottamiseen, mutta vaativat useita ajokertoja pintaa kohden ja ovat siten hitaampia kuin verteksivarjostimet. Näin ollen kuvanlaadun kannalta laskutoimitukset voi suorittaa fragmenttivarjostimessa, mutta nopeuden kannalta olisi hyvä jos mahdollisimman moni laskutoimitus suoritettaisiin verteksivarjostimessa. Paras vaihtoehto olisi kuitenkin jos arvo voitaisiin laskea jo ennen varjostinohjelman ajamista. (Rost 2007, 218)

Esimerkiksi kuvan jälkiprosessointi voidaan toteuttaa piirtämällä koko kuvaruudun peittävä neliö, jonka tekstuuria käsitellään fragmenttivarjostimella. Kun kuvaruudun koko on 320x200 kuvapistettä, niin fragmenttivarjostin ajetaan tuolloin 64 000 kertaa. Verteksivarjostin ajetaan tuolloin kuitenkin vain neljä kertaa. Mikäli joku toimenpide voidaan siirtää fragmenttivarjostimelta verteksivarjostimelle, vähenee laskutoimitusten määrä.

Sisäänrakennettujen metodien käyttö on myös suotavaa. Ne ovat näytönohjaimen valmistajan optimoimia ja toimivat todennäköisesti nopeammin kuin varjostinohjelman kehittäjän vastaava metodi. Jotkin toimenpiteet vaativat kuitenkin enemmän laskentatehoa kuin toiset. Esimerkiksi trigonometriset toimenpiteet kuten sin ja cos, sekä potenssilaskut kuten pow ovat muita raskaampia. Jakolaskut, neliöjuuret, käänteisarvot, vertailuoperaatiot, exponentit ja taulukon indeksointi vaativat muihin toimenpiteisiin verrattuna keskitason laskentatehoa. Muut toimenpiteet vaativat vähän tai eivät käytännössä lainkaan laskentatehoa. Muita neuvoja nopeuden saavuttamiselle ovat matriisikertolaskujen välttäminen, varying-muuttujien määrän rajoittaminen ja ohjelman koon pitäminen pienenä. (ARM 2009, 14-15; Rost 2007, 219)

3 KAJAK3D PELIMOOTTORI

Kajaanin ammattikorkeakoulun kehittämä Kajak3D pelimoottori on pelien sovelluskehitysalusta älypuhelimille, mutta se soveltuu myös tietokone- ja konsoliympäristöön. Kajak3D sovellukset luodaan käyttäen C++ ohjelmointikieltä. Kajak3D on kehitetty käyttäen JSR-184 spesifikaatiota, joka on älypuhelimille suunnatun Mobile 3D Graphics API:n (M3G) käyttämä määrittely. M3G on korkeantason API, joka on suunniteltu toimimaan tehokkaasti OpenGL ES grafiikkaohjelmointikirjaston kanssa. Kajak3D-pelimoottorilla tehty sovellus toimii Windows, Linux, iPhone, Android, Bada sekä Nokia N900 ympäristöissä. Kajak3D:n ollessa usealle alustalle tarkoitettu ohjelmointi ympäristö, tarvitsee se myös laajasti eri alustoilla käytössä olevan grafiikkaohjelmointirajapinnan: OpenGL ES:n. Kajak3D käyttää OpenGL ES 1.x ja 2.x ohjelmointirajapintoja näytönohjaimen hyödyntämiseen, mutta ei vaadi käyttäjältä tietotaitoa näiden rajapintojen käyttämisestä. Varjostimien luominen Kajak3D:lle ei poikkea paljoa yleisestä varjostimien kehittämisestä, mutta Kajak3D tarjoaa joukon muuttujia jotka yksinkertaistavat varjostinohjelmien toteutusta. Kajak3D:n varaamia muuttujia, käyttäen sovelluksen kehittäjä pääsee nopeaan alkuun, eikä hänen tarvitse toteuttaa lähes joka kerta tarvittavien muuttujien lähettämistä varjostinohjelmalle. Varjostimien käyttöönotto Kajak3D ympäristössä tulee tietenkin toteuttaa käyttäen Kajak3D toimintoja. (Kajak3D 2010; Pulli 2008, 19)

3.1 Kajak3D:n tukemat alustat

Kajak3D:llä on mahdollista kehittää sovelluksia useille eri alustoille. Näitä alustoja ovat muun muassa Windows, Linux, iPhone, Android, Bada ja Maemo. PC:llä vaaditaan OpenGL rajapintaa ja mobiililaitteilla OpenGL ES rajapintaa. Yksi suurimmista ongelmista on alustojen erilaisuus. Eri puhelinmallit käyttävät eri käyttöjärjestelmiä ja eri prosessoreita. Osa prosessoreista ei tue liukulukuja ja osa tukee. (Kajak3D 2010; Munshi, A. 339-340)

Vaatimuksena varjostimien kehittämiselle PC ympäristössä on ainoastaan OpenGL 2.0 tuettu näytönohjain ja emuloinnin vaatimat kirjastot. Mikäli tarkoituksena on luoda sovelluksia PC käyttöön tulee varjostimet kirjoittaa käyttäen GLSL kieltä.

Android on Googlen kehittämä Linuxiin pohjautuva käyttöjärjestelmä älypuhelimille. Androidille luotavat sovellukset käyttävät Java-ohjelmointikieltä, mutta C/C++:llä luodut sovellukset on myös mahdollista saada toimimaan. Android 2.2 versio tukee OpenGL ES 2.0 grafiikkarajapintaa. Android 2.2 on saatavilla esimerkiksi HTC Desire puhelimeen, joka on varustettu 1Ghz prosessorilla. (Android Developers 2010 a; Android Developers 2010 b; HTC 2010)

Maemo on Nokian kehittämä Linux-pohjainen ohjelmistoalusta Nokian älypuhelimille, kuten Nokia N900:lle. Nokia N900 on varustettu ARM Cortex-A8 prosessorilla, joka toimii 600mhz nopeudella. 3D-kiihdytystä varten puhelin tukee OpenGL ES 2.0:aa. (Nokia 2010 a; Nokia 2010 b)

Linuxiin pohjautuvalle Maemolle kehitettävien ohjelmien tulee myös olla UNIX-pohjaisia. Ohjelmistokehitystä varten Maemo SDK tarjoaa virtuaalikoneen, joka matkii Maemo alustan ympäristöä. Sovellus on mahdollista kääntää virtuaalikoneen Scratchbox-ympäristössä. Scratchbox on ristikääntäjä ja työkalupakki Linux-pohjaisille sulautettujen järjestelmien sovelluksille. Virtuaalikone voidaan ajaa esimerkiksi VMware Player ohjelmalla. (Maemo 2009; Movial 2010)

OpenGL ES 2.0 vaatii libGLESv2.so-kirjaston kopioitavaksi samaan kansioon ohjelmatiedoston kanssa.

3.2 Koordinaatistot

Koordinaatistot mahdollistavat sijaintien määrittelyn. Koordinaatisto koostuu akselien suunnista sekä koordinaatiston sijainnista. Koordinaatiston sijainnin määrittelemiseksi tarvitaan siis toinen koordinaatisto, jonka osaksi koordinaatisto kuuluu. Poikkeus tapahtuu kuitenkin maailmakoordinaatiston osalta, jota voidaan pitää alustana muille koordinaatistoille. Kajak3D:n käyttämät oikeankätiset kolmiulotteisen koordinaatiston akselit on nimetty X, Y ja Z-akseleiksi, jotka ovat suorakulmassa toisiinsa nähden. (Pulli 2008, 27, 29)

- Maailmakoordinaatisto
- Objektikoordinaatisto

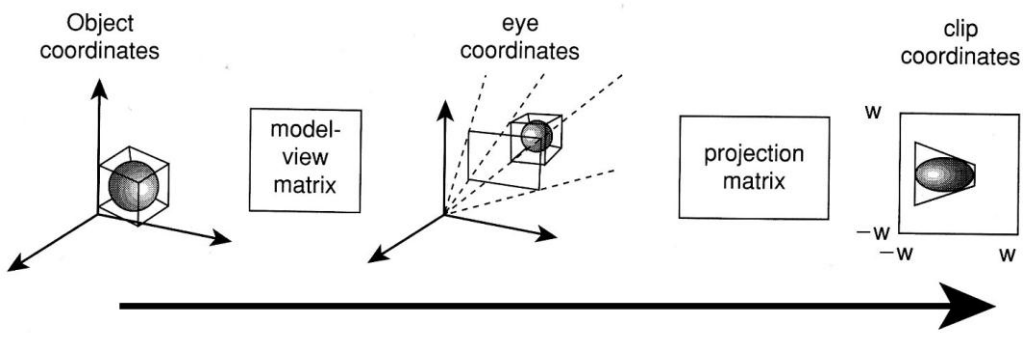
- Silmäkoordinaatisto
- Clip-koordinaatisto
- Tangenttikoordinaatisto

Maailmakoordinaatisto toimii pohjana muille koordinaatistoille. Maailmakoordinaatisto ilmaisee sijaintien suhteet toisiinsa ja toimii näin yhteisenä koordinaatistona kaikille sijainneille. (Rost 2007, 22)

Objektikoordinaatisto, eli mallikoordinaatisto, on mallikohtainen koordinaatisto, joka kuvastaa objektin sijaintia, suuntaa ja kokoa maailmakoordinaatistossa. Verteksien sijainnit on määritelty tässä koordinaatistossa. Objektikoordinaatistoa muuntamalla myös sen alaisuudessa olevat sijainnit muuntuvat. (Rost 2007, 22)

Silmäkoordinaatisto on kameraa tai silmää kuvastava koordinaatisto. Kaiken piirrettävän on oltava silmäkoordinaatistossa. Kamera on aina silmäkoordinaatiston origossa. Koordinaatiston Z-akseli osoittaa kameran kanssa samaan suuntaan, X-akseli osoittaa oikealle ja Y-akseli osoittaa ylöspäin. (Pulli 2008, 42)

Clip-koordinaatisto on tarkoitettu piirtoalueen rajaamista varten. Koordinaatistossa olevat sijainnit on mahdollista leikata pois, mikäli ne eivät ole piirtoalueen sisällä. Tämä on viimeinen vaihe josta verteksivarjostimia tehdessä tarvitsee huolehtia. (Rost 2007, 25)



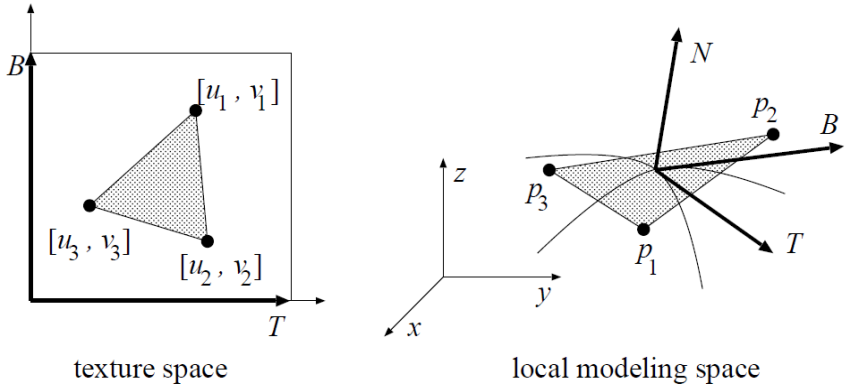
Kuvio 10. Muunnos koordinaatistosta toiseen tapahtuu kertomalla koordinaatit muunnosmatriisilla. (Kuva: Pulli 2008, 28)

Koordinaatistot voidaan kuvailla 4x4 matriiseina. Sijainteja voidaan muuntaa toiseen matriisiin kertomalla ne muunnosmatriisilla. Muunnosmatriisit voi kertoa yhdeksi muunnosmatriisiksi, joka vähentää tehtyjen kertolaskujen määrää. Tästä syystä malli-kamera- ja projek-

tiomatriisit on yhdistetty malli-kamera-projektio matriisiksi. Muunnosmatriisien käyttö on etenkin tarpeen muuntaessa sijainteja sovelluksen 3D-koordinaatistosta kuvapuskurin vaatimaan 2D-koordinaatistoon (Kuvio 10.). (Pulli 2008, 31, 33-34; Rost 2007, 21)

Kaikki mallit täytyy muuntaa objektikoordinaatistosta yhteiseen koordinaatistoon, eli maailmakoordinaatistoon jotta ne voitaisiin piirtää osana samaa ympäristöä. Piirtämistä varten sijainnit maailmakoordinaatistossa tulee muuntaa myös silmäkoordinaatistoon. Verteksivarjostimessa muunnos objekti- tai maailmakoordinaatistosta silmäkoordinaatistoon tapahtuu kertomalla sijainti m3g_meshAndCamera_Matrix -matriisilla. Muunnos clip-koordinaatistoon tapahtuu kertomalla sijainti m3g_modelViewProjectionMatrix -matriisilla. (Rost 2007, 23)

Tangenttikoordinaatisto, eli tangenttiavaruus (Tangent Space), on kappaleen pinnan myötäinen koordinaatisto (Kuvio 11.). Koordinaatiston muodostavat pintaan kuuluvat kulmapisteet. Tuon koordinaatiston avulla pinnan pisteet voidaan muuttaa maailmakoordinaatistoon (World Space). Näin ollen pinnan myötäiset avaruudet, kuten tekstuuriavaruus, voidaan muuttaa maailmakoordinaatistoon, joka mahdollistaa kehittyneiden varjostimien kuten normaalikartoituksen sekä kuvapistevarjostuksen käytön. Tangenttiavaruus määritellään tangentti-, binormaali- ja normaalivektoreilla, jossa tangentti ja binormaali ovat pinnanmyötäisiä vektoreita, kun taas normaali on pinnan kohtisuora vektori. Tangenttivektori (T) ja binormaalivektori (B) vastaavat tekstuurikoordinaatistossa u- ja v-vektoreita. T voidaan siis määritellä vektoriksi (1,0) ja B vektoriksi (0,1). (Jouvie 2010)



Kuvio 11. Tangenttikoordinaatisto mahdollistaa tekstuurikoordinaatistossa olevan pisteen käsittelyn mallikoordinaatistossa. (Kuva: Jouvie 2010)

3.3 Kajak3D:n varaamat muuttujat

Kajak3D tarjoaa joukon muuttujia, jotka siirtävät API:n käyttämiä arvoja automaattisesti varjostimelle. Nämä muuttujat käsittelevät valonlähteiden, sumun, materiaalien, verteksien ja matriisien tiedot (LIITE 1). Muuttujat esitellään jokaisen varjostimen alussa automaattisesti, joten ne ovat käytettävissä ilman erillistä esittelyä.

Verteksikohtaiset tiedot lähetetään varjostimelle attribute-muuttujina. Lähes jokaisessa varjostimessa tarvittavia arvoja ovat verteksin sijainti (m3g_position), verteksin normaali (m3g_normal) ja tekstuurikoordinaatit (m3g_texcoord0). Verteksin väriarvo (m3g_color) on varjostimelle saapuessaan kokonaisluku. Tämä johtuu siitä, että Kajak3D säilyttää väriarvot kokonaislukuina ja eri värikomponentit on talletettu tuon kokonaisluvun bitteihin. Uniform-muuttujia käsiteltäessä Kajak3D muuntaa väriarvot liukuluvuiksi ennen varjostimelle lähettämistä. Tätä ei kuitenkaan tehdä toimiessa attribute-muuttujien kanssa. Verteksin väriarvo tulee muuntaa verteksivarjostimessa liukuluvuksi jakamalla se arvolla 250 tai kertomalla arvolla 0.003921568627451. Kertolaskut ovat kuitenkin jakolaskuja nopeampia, joten niiden käyttö on suositeltavaa. Tangenttikoordinaatistoon liittyvät attribute-muuttujat (tangent ja binormal) lähetetään ainoastaan jos Kajak3D:lle annetaan erillinen käsky tehdä näin (Kuvio 12.).

```
VertexBuffer* vertexBuffer = m_mesh->getVertexBuffer();
IndexBuffer* indexBuffer = m_mesh->getIndexBuffer(0);
VertexArray::generateBinormalsAndTangents(StaticVertexArray,
vertexBuffer, indexBuffer);
```

Kuvio 12. Binormaalien ja tangenttien laskeminen m_mesh-oliolle.

Useimmat valoihin liittyvät muuttujat ovat yksiselitteisiä, mutta jotkin vaativat selitystä. Valoihin liittyvät arvot on taulukoitu valokohtaisesti ja valojen määrä on rajattu kahdeksaan. Saman indeksin omaavat muuttujat ovat siis osa samaa valonlähdettä. Valotyyppi (m3g_lightType) voi olla arvoltaan 128 (ympäröivä valo), 129 (suuntavalo), 130 (pistevalo) tai 131 (kohdevalo). Sumuun liittyvät muuttujat määrittelevät onko sumu päällä, kuinka tiheää se on, minkä väristä se on ja kuinka pitkällä matkalla se vaikuttaa. Useimpien muuttujien käyttö on yksinkertaista, mutta sumun laskentakaavan (m3g_fogMode) käyttö vaatii tarkan tiedon siitä mitä arvo voi olla. Sumun laskentakaava on siis kokonaisluku, jonka arvo kertoo mitä laskentakaavaa API haluaa käyttää. Muuttujan arvo voi olla 80 (eksponentiaalinen sumu) tai 81 (lineaarinen sumu).

Uniform-muuttuja määrän pitämiseksi pienenä, osa muuttujista lähetetään varjostinohjelmalle taulukkona. Nämä muuttujat ovat totuusarvoja, jotka kuvaavat onko jokin toiminto päällä vai ei. Arvot säilytetään liukulukuina. Muuttujat kertovat muun muassa ovatko valot päällä, käytetäänkö verteksikohtaisia värejä ja onko teksturointi päällä. Taulukon soluun voi viitata käyttäen Kajak3D:n muuttujien yhteydessä esiteltäviä esikääntäjän vakioita (LIITE 1). Esimerkiksi sumun päällä oleminen on mahdollista tarkistaa varjostimessa lähdekoodilla: "if (m3g_featuresFlags[SHADER_FOG_ENABLED] == 1.0)". Ehdon ollessa tosi sumu on käytössä.

3.4 Varjostimien käyttäminen Kajak3D:ssä

Varjostinohjelmat ovat käytettävissä ainoastaan OpenGL ES 2.x ohjelmointirajapinnan kanssa, joten kehitettävän sovelluksen tulee myös käyttää sitä. Kajak3D sovelluksen alustuksen yhteydessä tulee Kajak3DFramework:ille antaa osoitin OpenGL ES 2.x grafiikkakontekstiin käyttäen metodia void setGraphics3D(render::Graphics3D* g3d). OpenGL ES 2.x grafiikkakonteksti luodaan metodilla OGLES2_Graphics3D() (Kuvio 13.).

```
// Kajak3D:n OpenGL ES 2.x vaatima kirjasto
#include <render/OGLES/OGLES2_Graphics3D.h>
...
// Kajak3D:n asettaminen OpenGL ES 2.x tilaan
Kajak3DFramework.setGraphics3D(KAJAK3D_NEW OGLES2_Graphics3D());
```

Kuvio 13. Kajak3D:n asettaminen OpenGL ES 2.x tilaan

Kajak3D käyttää perinteistä OpenGL ohjelmointirajapintaa tietokoneille suunnatuissa sovelluksissa. Älypuhelimille kehitettävän sovelluksen kehitysvaiheessa on käytännöllistä testata sovellukset tietokoneympäristössä, mutta ongelmalliseksi testaamisen tekee se, etteivät pöytäkoneet tue OpenGL ES:ää. Tästä seurauksena varjostimessa olevat ESSL virheet ja varoitukset eivät myöskään ilmene käännön yhteydessä. Käyttämällä AMD:n OpenGL ES 2.0 emulointia virheet saa kuitenkin näkyviin ja sovellukset voidaan ajaa myös tietokoneella. Emulointi vaatii libEGL.lib ja libGLESv2.lib kirjastot lisättäväksi projektiin. Oletuksena Kajak3D käyttää pöytäkoneella OpenGL ES:n sijaan OpenGL:ää. Tämä oletus on mahdollista ottaa pois käytöstä kommentoimalla Kajak3D:n render kirjaston render_pp.h tiedoston rivi #define OPENGL_ES_USES_NORMAL_OPENGL. Näin sovellus käyttää OpenGL ES 2.x emulointia perinteisen OpenGL:n sijaan. (Opengles-book.com 2010)

Kaikki varjostimiin liittyvä toiminnallisuus on määritelty Kajak3D:n render-nimiavaruudessa. Tilan säästämiseksi opinnäytetyön esimerkkilähdekoodeissa funktioita käsitellään kuin toimittaisiin render-nimiavaruudessa.

Kajak3D:ssä varjostinohjelmia käsitellään ShaderProgram-olioina. Varjostinohjelman luominen vaatii verteksi- ja fragmenttivarjostimen, joita käsitellään VertexShader ja FragmentShader olioina. ShaderProgram-olion alustusmetodi vaatii parametreinaan VertexShader- ja FragmentShader-olioiden osoittimet. Fragmenttivarjostimen luominen vaatii parametrina fragmenttivarjostimen lähdekoodin. Verteksivarjostin vaatii verteksivarjostimen lähdekoodin lisäksi myös totuusarvon siitä käytetäänkö Fixed Function Pipeline-varjostinta osana verteksivarjostinta. Jos arvo on tosi, varjostimen lähdekoodin alkuun lisätään FFP-varjostimen lähdekoodi, joka tulee ajaa varjostimessa metodilla m3g_ffunction().

Varjostinohjelma otetaan käyttöön Graphics3D-luokan metodilla void setActiveShaderProgram(ShaderProgram* shader_program) (Kuvio 14.). Parametrinä metodille annetaan ShaderProgram-olio, jota halutaan käyttää. Aktiivinen varjostin poistetaan käytöstä antamalla metodin parametriksi arvo 0. Kajak3D sovelluksen void render(Graphics3D* g3d)-metodi tarjoaa osoittimen aktiivisena olevaan Graphics3D-olioon.

```
// Varjostinohjelman vaatimat kirjastot
#include <render/Shader/VertexShader.h>
#include <render/Shader/FragmentShader.h>
#include <render/Shader/ShaderUniforms.h>
...
// Varjostinohjelman luominen
VertexShader* vShader = VertexShader(vSource, false);
FragmentShader* fShader = FragmentShader (fSource);
ShaderProgram* shader = ShaderProgram(vShader, fShader, false);
...
// Varjostinohjelman aktivoiminen
g3d->setActiveShaderProgram(shader);
```

Kuvio 14. Varjostinohjelman luominen ja aktivoiminen.

3.5 Varjostinohjelman toteuttaminen

Varjostinohjelma piirtää kappaleen kameran sijainnista katsottuna. Lisäksi kappaleen tekstuurista saatu väriarvo ja käyttäjän määrittelemä väriarvo kerrotaan keskenään, jolloin kappale värittyy mutta säilyttää yhä tekstuurista saadun kuvion. Varjostinohjelman toteuttamiseksi täytyy kehittäjän kirjoittaa verteksi- ja fragmenttivarjostimen lähdekoodi.

3.5.1 Verteksivarjostin

Verteksivarjostimen tärkein tehtävä on ilmoittaa verteksin sijainti verteksivarjostimen sisäänrakennetulle gl_Position-muuttujalle, joka määrittelee verteksin sijainnin clip-koordinaatistossa. (Simpson 2009, 59)

Esimerkissä texCoord nimiseen varying-muuttujaan sijoitetaan kappaleeseen sidotut tekstuurikoordinaatit, mutta ennen tätä tekstuurikoordinaatit kerrotaan tekstuurimatriisilla. Näin otetaan huomioon myös mahdollisuus, että tekstuuria on kappaleen pinnalla venytetty, pyöritetty tai siirretty. Verteksin sijainti voidaan muuntaa mallikoordinaatistosta clip-koordinaatistoon kertomalla se m3g_modelViewProjectionMatrix-matriisilla (Kuvio 15.).

```
varying mediump vec2 texCoord;

void main()
{
      texCoord = m3g_textureMatrix0 * vec4(m3g_texcoord0, 0.0, 1.0);
      gl_Position = m3g_modelViewProjectionMatrix * vec4(m3g_position,
      1.0);
}
```

Kuvio 15. Verteksivarjostimen lähdekoodi muuntaa verteksin sijainnin objektikoordinaatistosta clip-koordinaatistoon.

3.5.2 Fragmenttivarjostin

Fragmenttivarjostin voi asettaa fragmentin värin sijoittamalla väriarvon muuttujaan gl_FragColor. Muuttuja gl_fragColor on fragmenttivarjostimen sisäänrakennettu muuttuja, joka määrittelee fragmentin väriarvon. Väriarvo voidaan myös sijoittaa muuttujaan gl_FragData[0]. Ero muuttujien välillä on se että gl_FragData voi viitata myös muihin puskureihin kuin piirrettävälle väripuskurille. Laitteiston ei kuitenkaan ole pakko tukea kuin yhtä piirtopuskuria, joten gl_FragColor muuttujan käyttö on suositeltavaa. Arvo voitaisiin myös antaa gl_FrontFacing muuttujalle. Tällöin fragmentti piirretään ainoastaan kun pinta on kohti katsojaa, mutta JSR-184 ei erottele pinnan etu- ja takapuolta. Fragmenttivarjostimen ei ole myöskään pakko määritellä fragmentin väriä tai se voi hylätä fragmentin kokonaan discard käskyllä. (Munshi 2008, 8; Simpson 2009, 60-61)

Esimerkissä tekstuurista saatu väriarvo kerrotaan käyttäjän lähettämän uniform-arvon kanssa (Kuvio 16.). Tekstuurin väriarvo haetaan tekstuurikartan kohdassa texCoord fragmenttivarjostimen metodilla texture2D() ja se sijoitetaan muuttujaan texel. Fragmentin väriarvoksi sijoitetaan texel ja myColor muuttujien tulo.

```
uniform mediump vec4 myColor;
varying mediump vec2 texCoord;

void main()
{
      vec4 mediump texel = texture2D(m3g_texture0, texCoord);
      gl_FragColor = texel * myColor;
}
```

Kuvio 16. Fragmenttivarjostimen lähdekoodi, joka asettaa fragmentin väriksi uniform-muuttujan ja tekstuurista haetun väriarvon tulon.

3.5.3 Sovellus

Uniform-muuttujien arvon määritteleminen tapahtuu Kajak3D sovelluksessa. Kajak3D käsittelee varjostinkohtaisia muuttujia ShaderUniforms-luokalla. Varjostinohjelmassa esitellyt uniform-muuttujat haetaan varjostimelta luokan metodilla getUniforms(). Yksittäisiä muuttujia etsitään tuosta uniform-muuttujien joukosta metodilla findUniform(). Metodi palauttaa kokonaisluvun, joka toimii osoittimena uniform-muuttujaan. Metodi ottaa parametrinaan muuttujan nimen varjostimen lähdekoodissa tai kokonaisluvun joka on sidottu muuttujaan.

Käyttäen setUniformValue() metodia ja sen ylikirjoitettuja metodeja on mahdollista lähettää varjostimelle erityyppisiä arvoja. Ensimmäisenä parametrina metodille täytyy antaa uniform-muuttujaan viittaava kokonaisluku, jonka findUniform() metodi palauttaa. Toisena parametrina metodille annetaan muuttujaan sijoitettava arvo. Tekstuuria lähettäessä metodi vaatii myös tiedon, monesko aktiivinen tekstuuri kuva on. Esimerkissä sovellus lähettää varjostinohjelmalle punaisen väriarvon (Kuvio 17.).

```
ShaderUniforms* uniforms = shaderProgram->getUniforms();
int uniformId = uniforms->findUniform(”myColor”);
uniforms->setUniformValue(uniformId, slmath::vec4(1.0f, 0.0f, 0.0f,
1.0f));
```

Kuvio 17. Kajak3D sovelluksen lähdekoodin osa lähettää shaderProgram-varjostinohjelman myColor uniform-muuttujalle arvon (1.0, 0.0, 0.0, 1.0), joka vastaa punaista väriä.

3.6 Kuvan jälkiprosessoinnin toteuttaminen

Kuvan jälkiprosessointi mahdollistaa varjostimien käyttämisen koko ruudun kokoisella alueella. Kuvan jälkiprosessointi on mahdollista toteuttaa piirtämällä kuva tekstuuriin tai kopioimalla piirretty kuvapuskuri tekstuurille. Tämän jälkeen tekstuuri piirretään koko ruudun alueelle venytettynä. Tekstuurin, jota käytetään kuvan jälkiprosessoinnin toteuttamiseen, tulisi olla koko piirtoalueen kokoinen. Kajak3D:ssä tekstuurista luodaan piirtokohde luomalla RenderTarget-olio, joka ottaa tekstuurin parametrina (Kuvio 18.). Ennen ympäristön piirtämistä tulee tuo piirtokohde asettaa aktiiviseksi Graphics3D-luokan metodilla bindTarget(). Nyt piirto tapahtuu tavallisen kuvapuskurin sijaan tekstuurille. Ympäristön piirtämisen jälkeen tekstuuri voidaan piirtää koko ruudun alueelle venytettynä käyttäen Graphics3D-luokan metodia rect(). Tämän piirron yhteydessä tulee käyttää kuvan jälkiprosessointiin suunniteltua varjostinohjelmaa.

```
// 2D-tekstuurin luominen
Texture2D* texture = g3d->createNewTexture();
// Tee tekstuurista piirtämisen kohde
RenderTarget* renderTarget = KAJAK3D_NEW RenderTarget(texture);
...
// Piirrä ympäristö tekstuurille
g3d->bindTarget(renderTarget);
g3d->setActiveShaderProgram(0);
g3d->clear(0);
g3d->render(m_world);
g3d->releaseTarget();

// Piirrä tekstuuri kuvapuskurille
g3d->bindTarget(0);
g3d->clear(0);
g3d->setActiveShaderProgram(BlurShader);
g3d->rect(slmath::vec2(0.0f, 0.0f), slmath::vec2(1.0f, 1.0f), texture);
g3d->releaseTarget();
```

Kuvio 18. Kajak3D BlurShader jälkiprosessointitehosteen käyttäminen.

4 KAJAK3D VARJOSTIMIEN TOTEUTUS

OpenGL ES 2.x tarjoama mahdollisuus käyttää varjostimia tuo vapauden lisäksi myös vastuuta. OpenGL ES 2.x piirtoprosessin ohjelmoitavat vaiheet ovat tyhjiä, eivätkä ne tee mitään ellei niihin luoda varjostimilla toiminnallisuutta. OpenGL ES 2.x ei näin ollen kykene piirtämään mitään ilman varjostinohjelmia.

Tavoitteena on kehittää varjostinohjelma, joka toimii ohjelmoitavissa prosessoreissa samalla tavalla kuin vastaavat OpenGL ES 1.x piirtoprosessin vaiheet. Varjostinohjelman tarkoitus on toimia oletusvarjostimena Kajak3D pelimoottorissa, jotta siirtyminen OpenGL ES 2.x:n käyttämiseen sujuu vaivattomasti.

Varjostinohjelman tulee piirtää malli samanlaisena kuin OpenGL ES 1.x. Varjostimen tulee siis korvata OpenGL ES 1.x piirtoprosessin muunnos-, valaistus, teksturointi-, värisumma-, sumu- ja läpinäkyvyystestausvaihe. Jokaisen vaiheen tulee tuottaa samanlainen lopputulos kuin vastaava vaihe tuottaa OpenGL ES 1.x piirtoprosessissa.

4.1 Fixed Function Pipeline-varjostimen kehitys

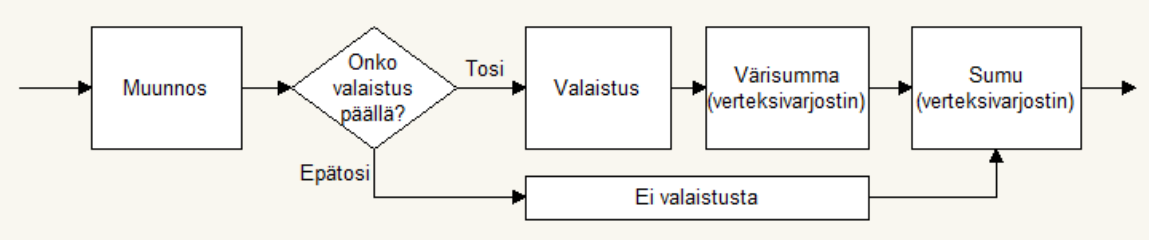
Fixed Function Pipeline-varjostin (FFP) on kiinteän piirtoprosessin toiminnallisuutta matkiva varjostinohjelma. Tarve varjostimelle syntyy, kun alustalle on tarjolla tuki ohjelmoitavalle piirtoprosessille, mutta sovellus on toteutettu kiinteää piirtoprosessia käyttäen. Kehittäjä voi tuolloin haluta siirtää sovelluksen kehittyneemmälle ohjelmoitavalle piirtoprosessille, mutta tämä vaatisi sovelluksen uudelleen toteuttamista käyttäen varjostimia. Tällöin on mahdollista käyttää FFP-varjostinta, joka piirtää sovelluksen identtisesti kiinteän piirtoprosessin kanssa.

Opinnäytetyön tavoite on siis luoda varjostinohjelma, joka toimii samalla tavalla kuin OpenGL ES 1.3 piirtoprosessi, mutta noudattaa myös JSR-184-määritelmää. Määritelmä on toiminut Kajak3D:n OpenGL ES 1.3 implementaation ohjenuorana ja sen määrityksiä käyttäen pitäisi piirtojäljestä tulla lähes identtistä kiinteän piirtoprosessin kanssa. Varjostimen tulisi myös toimia lisäämättä kehittäjän työn määrää. Kehittäjän tulee vain valita grafiikkarajapinta ja ohjelma toimii yhä ilman suurempia muutoksia.

Seuraavat vaiheet OpenGL ES piirtoprosessissa on korvattu varjostimilla:

- Muunnos ja valaistus
- Tekstuuriympäristö
- Värisumma
- Sumu
- Läpinäkyvyystestaus

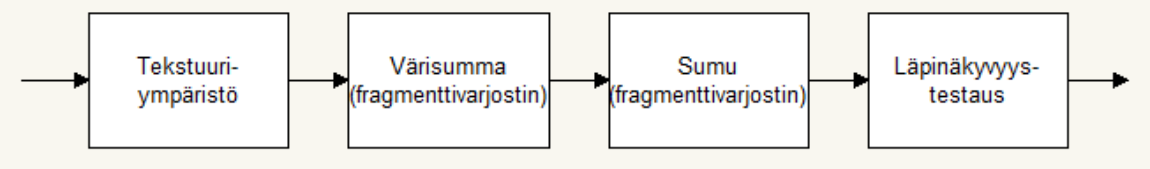
Nämä asiakokonaisuudet voi käsitellä verteksi- tai fragmenttivarjostimessa. Piirron nopeuttamiseksi mahdollisimman moni vaihe käsitellään verteksivarjostimessa. Muunnos ja valaistus toteutetaan verteksivarjostimessa. Muunnos ja valaistus käsitellään erillisinä asiakokonaisuuksina niiden laajuuden vuoksi. Nopeuden saavuttamiseksi myös sumu ja osa värisumman laskemisesta siirrettään osittain verteksivarjostimelle. Fragmenttivarjostimelle jää siis käsiteltäväksi tekstuuriympäristö, läpinäkyvyystestaus ja loput värisumman laskemisesta.



Kuvio 19. Verteksivarjostimen m3g_ffunction()-metodin yksinkertaistettu toimintamalli.

Verteksivarjostimessa toiminnallisuus on toteutettu m3g_ffunction()-metodissa (LIITE 2). Metodin toimintamalli on melko yksinkertainen (Kuvio 19.). Metodi käy läpi sen vastuulle kuuluvat piirtoprosessin vaiheet. Ensimmäisenä suoritetaan muunnosvaihe, jossa toteutetaan esimerkiksi verteksimuunnokset. Mikäli valaistus on käytössä, suoritetaan valaistus- ja värisummavaihe. Valaistusvaiheessa käydään läpi valojen vaikutus verteksiin ja värisummavaiheessa tuo vaikutus lisätään verteksin väriarvoon. Näin ollen värisumman laskeminen jää tarpeettomaksi jos valaistus ei ole käytössä. Valaistuksen ollessa pois päältä käytetään verteksin väriarvona sen alkuperäistä väriä. Sumun vaikutus lasketaan myös verteksikohtaisesti, mutta se lisätään fragmenttikohtaisesti fragmenttivarjostimessa. Myös sumu voi olla pois päältä, mutta sillä ei ole vaikutusta toimintamallin etenemiseen. Verteksivarjostin lähettää

fragmenttivarjostimelle verteksin väriarvon, tekstuurikoordinaatit, spekulaarivalojen summan ja sumun kertoimen.



Kuvio 20. Fragmenttivarjostimen m3g_ffunction()-metodin yksinkertaistettu toimintamalli.

Samoin kuin verteksivarjostimessa, myös fragmenttivarjostimen toiminnallisuus on toteutettu m3g_ffunction()-metodissa (LIITE 3). Fragmenttivarjostin käy läpi sen vastuulle jääneet piirtoprosessin vaiheet (Kuvio 20.). Fragmentin pohjavärinä käytetään verteksivarjostimesta lähetettyä verteksin väriarvoa. Tämän jälkeen tekstuurien vaikutus fragmenttiin lasketaan. Myös teksturointi voi olla kytkettynä pois päältä, mutta se ei vaikuta toimintamallin seuraaviin vaiheisiin. Fragmenttivarjostimessa värisumman laskennassa ei tapahdu muuta kuin peiliheijastuksen lisääminen. Fragmenttivarjostimen sumuvaiheessa fragmentin väriä muokataan lisäämällä siihen sumun väriä ja läpinäkyvyyttä. Läpinäkyvyydentestauksessa tarkastellaan onko fragmentti tarpeeksi näkyvä piirrettäväksi. Vaikka tämä toimenpide tehtäisiin viimeisenä, se saattaa nopeuttaa fragmenttiprosessorin jälkeisen piirtoprosessin toimintaa.

4.1.1 Muunnosvaihe

Muunnosvaihe (Transform) käsittelee verteksien, tekstuurikoordinaattien sekä normaalien laskemiseen liittyvän toiminnallisuuden. (Pulli 2008, 183)

Verteksin sijainti tulee muuttaa objektikoordinaatistosta Clip-koordinaatistoon. Tämä tapahtuu kertomalla verteksin sijainti malli-, näkymä- ja projektiomatriiseilla, jotka on yhdistetty yhdeksi matriisiksi: m3g_modelViewProjectionMatrix. Koska valaistuksen laskeminen tapahtuu silmäkoordinaatistossa, täytyy verteksin sijainti laskea myös tuossa koordinaatistossa. Tämä tapahtuu kertomalla verteksin sijainti objektikoordinaatistossa malli- ja näkymämatriiseilla, jotka on yhdistetty m3g_meshAndCamera_Matrix-matriisiksi. Normaalit tulee myös muuntaa silmäkoordinaatistoon, mutta tämä tapahtuu kertomalla verteksin normaali m3g_normalMatrix-matriisilla. Normaalien laskenta ei ole tarpeen mikäli valaistus ei ole käytössä. Tästä johtuen normaalien laskenta on sijoitettu valaistuksen yhteyteen lähdekoodissa.

Tekstuurimuunnokset vaativat tekstuurikoordinaattien kertomisen tekstuurimatriisilla. Tämä mahdollistaa tekstuurin siirtämisen, skaalaamisen ja pyörittämisen. Tekstuurikoordinaatit täytyy lähettää fragmenttivarjostimelle varying-muuttujina.

```
// Transform vertex to clip space
M3Gposition = m3g_modelViewProjectionMatrix * vec4(m3g_position, 1.0);

// Transform vertex position to eye space
vec3 ecPosition = vec3(m3g_meshAndCamera_Matrix * vec4(m3g_position, 1.0));

// Texture coordinates
m3g_out_texcoord[0] = vec2(m3g_textureMatrix0 * vec4(m3g_texcoord0, 0.0, 1.0));
m3g_out_texcoord[1] = vec2(m3g_textureMatrix1 * vec4(m3g_texcoord1, 0.0, 1.0));
m3g_out_texcoord[2] = vec2(m3g_textureMatrix2 * vec4(m3g_texcoord2, 0.0, 1.0));
m3g_out_texcoord[3] = vec2(m3g_textureMatrix3 * vec4(m3g_texcoord3, 0.0, 1.0));
```

Kuvio 21. Muunnosvaiheen toteutus verteksivarjostimen lähdekoodissa.

Muunnosvaiheen toteutus verteksivarjostimessa laskee verteksin sijainnin clip-koordinaatistossa, mutta arvoa ei sijoiteta suoraan gl_Position muuttujaan vaan verteksivarjostimen globaaliin M3Gposition muuttujaan. Näin käyttäjä voi muuntaa arvoa lisää ennen sijoitusta. Muuttuja ecPosition ilmaisee verteksin sijainnin silmäkoordinaatistossa. Sijaintia silmäkoordinaatistossa tarvitaan laskettaessa valojen ja sumun vaikutusta verteksiin. Tekstuurikoordinaatit lasketaan ja lähetetään varying-muuttujina fragmenttivarjostimelle (Kuvio 21.).

4.1.2 Valaistus

Verteksikohtainen valaistus toteutetaan laskemalla jokaisen valonlähteen vaikutus käsiteltävään verteksiin. On kuitenkin otettava huomioon, että valot eivät ole aina päällä ja että erityyppiset valonlähteet käyttävät erilaisia laskukaavoja. Myös materiaali vaikuttaa valon käyttäytymiseen, joten valojen väriarvoja ei voi vain lisätä verteksin väriin. Valoista tulevat erityyppiset väriarvot tulee summata omiin muuttujiinsa ja värisumman laskennan yhteydessä yhdistää ne materiaalin vastaaviin muuttujiin.

Valotyyppejä on JSR-184 määritelmässä neljä:

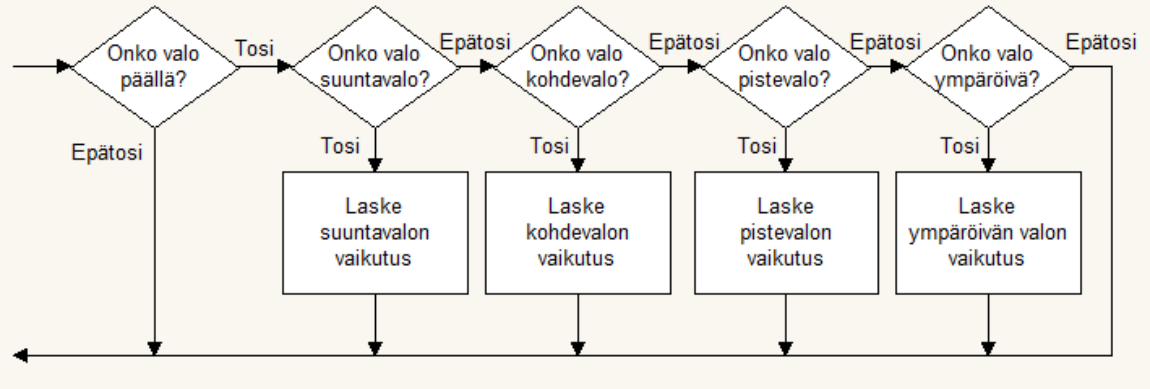
- Ympäröivä valo
- Suuntavalo
- Pistevalo

- Kohdevalo

Ympäröivä valo (Ambient Light) valaisee piirrettävän alueen tasapuolisesti joka suunnasta, näin ollen ympäröivän valon sijainnilla tai suunnalla ei ole merkitystä. Ympäröivällä valolla ei ole myöskään tarvetta hajavalolle tai peiliheijastukselle. Näin ollen ympäröivän valon toteutus on erittäin yksinkertainen. Verteksiin kohdistuvan ympäröivän valon määrään siis vain lisätään valolähteestä tulevan ympäröivän valon väri. Lisäksi mainittakoon että ympäröiviä valoja voi olla ympäristössä vain yksi. Muiden valotyyppien ympäröivän valon määrä on nolla. Eli muiden valotyyppien ympäröivän valon väriä ei tarvitse huomioida. (Pulli 2008, 68)

Suuntavalo (Directional Light) on valo jolla ei ole sijaintia. Se valaisee kaikki kohteet, jotka ovat kulmassa sitä vasten etäisyydestä riippumatta. Pistevalo (Omni Light) valaisee tasaisesti ympärilleen jokaiseen suuntaan. Pistevalon teho voi heiketä etäisyyden kasvaessa. Valon suunnalla ei siis ole merkitystä, mutta sijainnilla on. Kohdevalo (Spot Light) valaisee kartion muotoisen alueen sijaintinsa ja suuntansa mukaan. Kohdevalon voimakkuus voi myös heikentyä. (Pulli 2008, 68-69)

Kohteen ollessa kaukana valosta, valon valaisuteho heikkenee. Tätä ilmiötä, eli vaimentumista (Attenuation), kuvaillaan kolmella muuttujalla: tasainen-, lineaarinen- ja kvadraattinen-vaimennus. Tasainen vaimennus (Constant Attenuation) heikentää valon tehoa tasaisesti. Lineaarinen vaimennus (Linear Attenuation) heikentää valon tehoa lineaarisesti. Kvadraattinen vaimennus (Quadratic Attenuation) heikentää valon tehoa eksponentiaalisesti. (Pulli 2008, 69-70)



Kuvio 22. Yksittäisen valon käsittely noudattaa toimintamallia, jossa valotyypin mukaan kutsutaan metodi.

Toteutuksessa valojen eri komponentteja varten on luotu omat muuttujansa joihin summataan kunkin valon vaikutus. Verteksivarjostimen värisumman laskennan yhteydessä nuo komponentit yhdistetään verteksin väriin. Jokaista valoa kohti kutsutaan metodia, joka tarkistaa onko valo päällä ja minkä tyyppinen valonlähde on (Kuvio 22.). Jokaista valontyyppiä varten on toteutettu metodi, joka laskee valotyypin vaikutuksen verteksiin (LIITE 2).

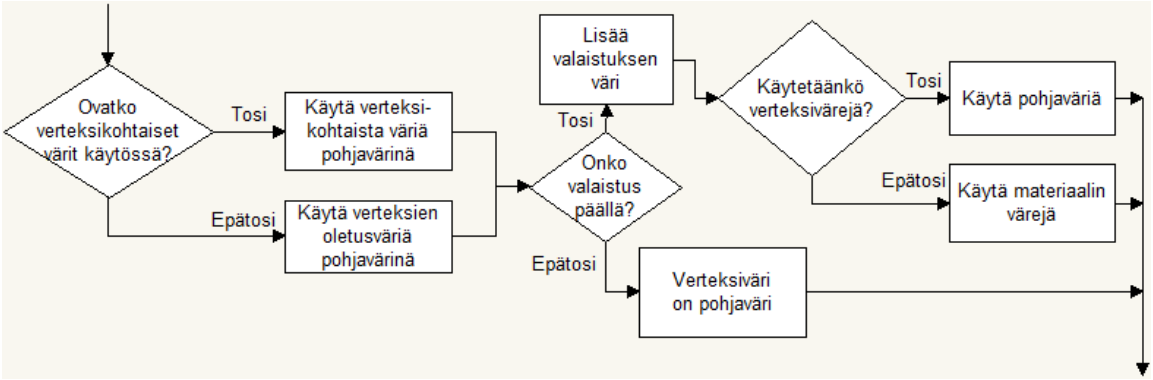
4.1.3 Tekstuuriympäristö

Teksturointi tapahtuu aina fragmenttivarjostimessa, sillä tekstuureita voi käsitellä ainoastaan sieltä käsin. Tekstuurin tekselin väriarvo ja fragmentin nykyinen väriarvo yhdistetään käyttäen jotain seuraavista laskutavoista: FUNC_ADD, FUNC_BLEND, FUNC_DECAL, FUNC_MODULATE tai FUNC_REPLACE. Tekstuuria lisätessä tulee siis tarkistaa mitä laskukaavaa se käyttää ja toimia sen mukaan. (Nokia 2005) Kajak3D varjostinmuuttujat tukevat piirtokohteen neljää ensimmäistä tekstuuria.

Taulukoiden selaaminen ei toimi jos indeksinä käytetään muuttujaa. Tästä johtuen tekstuureita ei voi esimerkiksi asettaa silmukkaan joka kävisi ne indekseittäin läpi. Ongelma on mahdollista kiertää tekemällä metodi, joka ottaa parametrina taulukon arvon ja toteuttaa toiminnallisuuden tuon arvon mukaan. Metodia kutsuttaessa taulukon indeksiin viitataan muuttumattomalla muuttujalla tai numerolla. (Nokia 2005)

4.1.4 Värisumma

Värisumman laskenta on toteutettu suurimmaksi osin verteksivarjostimessa, jossa se yhdistää verteksin alkuperäisen värin valaistuksesta saadun värin kanssa. Verteksin alkuperäinen väri voi olla peräisin verteksikohtaisesta väripuskurista, materiaalista tai yleisesti käytössä olevasta piirtoväristä. Jos valaistus ei ole päällä, asetetaan verteksin väriksi alkuperäinen verteksin väri. Fragmenttivarjostimessa värisumman osuudeksi jää valaistuksen peiliheijastus, joka tulee lisätä vasta teksturoinnin jälkeen. Näin fragmentin väriksi saadaan ennemmin valon väri kuin pinnan väri. (Rost 2007, 242)



Kuvio 23. Värisumman laskentaan liittyvä toimintamalli.

Verteksin pohjaväri voi olla peräisin verteksikohtaisesta väripuskurista tai piirron yhteydessä käytettävästä verteksin oletusväristä. Valaistuksen ollessa pois päältä verteksin väriarvoksi voidaan asettaa suoraan pohjaväri, mutta valaistuksen ollessa päällä täytyy valaistuksen väriarvo lisätä verteksin väriin. Pohjaväriä voidaan myös käyttää värisumman laskemisessa materiaalin väriarvon sijaan (Kuvio 23.).

4.1.5 Sumu

Sumu lisää fragmentin väriin omaa väriään ja häivyttää etäisiä fragmentteja vähentämällä niiden alfa-arvoa. JSR-184 määritys tukee kahta eri tapaa sumun laskemiselle: eksponentiaalinen ja lineaarinen. Lineaarinen sumu häivyttää fragmentit tasaisesti etäisyyden mukaan täysin näkyvästä täysin sumun peittämäksi. Eksponentiaalinen sumu taas häivyttää fragmentit kiihtyvällä tahdilla. (Nokia 2005; Pulli 2008, 332-333)

Sumu kuuluu osaksi fragmenttien käsittelyä piirtoprosessissa, mutta sumun kerroin voidaan laskea myös verteksikohtaisesti ja interpoloida arvot fragmentteja varten. Verteksivarjostimessa tarkistetaan onko sumu päällä. Jos on, niin sumun kerroin lasketaan sumutyypin mukaan. Sumun kerroin lähetetään fragmenttivarjostimelle, jossa sitä käytetään interpolointi arvona sumun ja fragmentin värin välillä. Tämä tarkoittaa myös sitä että fragmentti varjostimessa täytyy myös tarkistaa onko sumu päällä ennen kuin muutoksia tehdään. (Pulli 2008, 210; Rost 2007, 244-246)

4.1.6 Läpinäkyvyystestaus

Läpinäkyvyyden testaaminen tarkistaa onko fragmentin alfa-arvo alla sallitun rajan. Jos arvo on alle tuon rajan, fragmentti hylätään. Jos arvo taas on yli rajan, fragmentin käsittely jatkuu. Fragmentin hylkääminen nopeuttaa piirtoprosessin seuraavia vaiheita ja samalla estää sen etteivät läpinäkyvät fragmentit sekoita syvyyspuskuria. Läpinäkyvyyden testausta yksinkertaistaa se että testaus metodina käytetään aina GL_GEQUAL tarkastelua. Jos fragmentin alfa-arvo on enemmän tai yhtä suuri kuin vertailuarvo, fragmenttia ei hylätä. (Pulli 2008, 333)

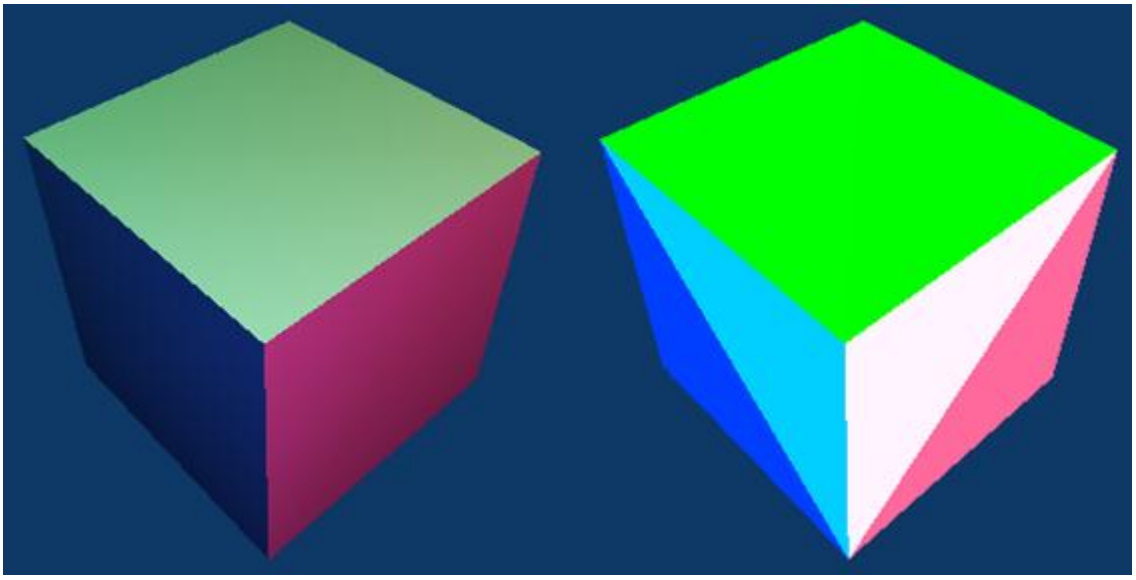
Fragmentti tulee hyväksyä jos se läpäisee tarkastuksen, eli se voidaan hylätä jos se ei läpäise tarkastusta (Kuvio 24.).

```
if (color.a < m3g_alphaTreshold)
{
      discard;
}
```

Kuvio 24. Alfa-testauksen toteutus.

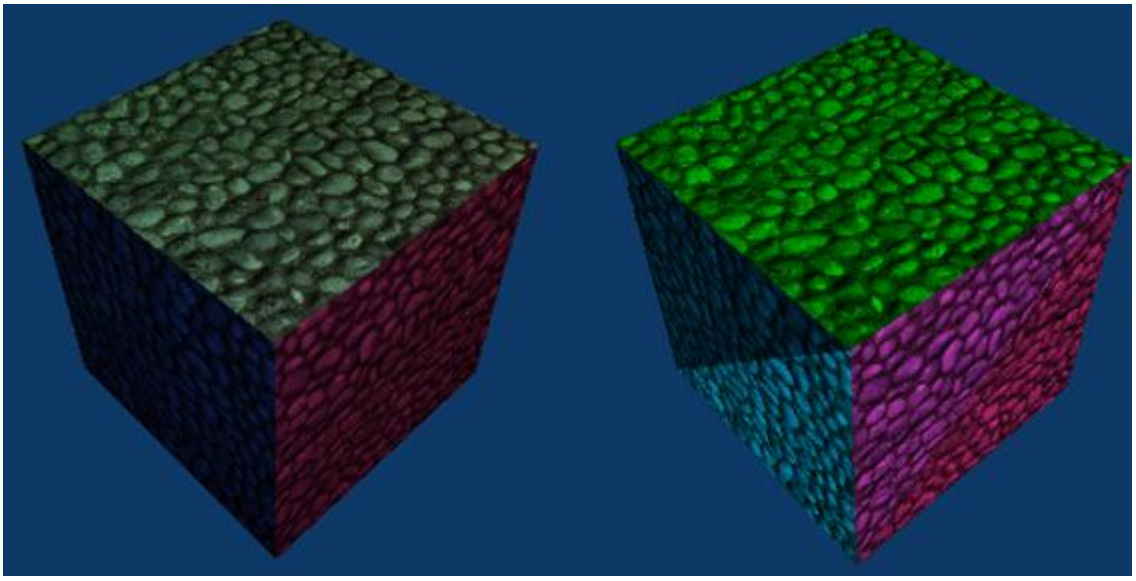
4.2 FFP-varjostimen testaus

Varjostinohjelman toimivuuden selvittämiseksi on sen piirtojälkeä vertailtava OpenGL ES 1.x piirtojälkeen. Vertailussa on luotu kaksi ohjelmaa jotka piirtävät saman ympäristön samoissa olosuhteissa. Molemmissa ympäristöissä on teksturoitu kuutio, jonka ympärillä on kolme pistevaloa valonlähteenä. Valonlähteet ovat sijoitettuna ympäristöön X, Y ja Z-akseleiden suuntaisesti ja ovat väriltään punainen, vihreä ja sininen. Ohjelmat käyttävät erillisiä kirjastoja toimiakseen PC ympäristössä. Tällä saattaa olla vaikutusta lopputulokseen, sillä kirjastot saattavat toimia eri nopeudella. Toinen ohjelma piirtää kuution käyttäen FFP-varjostinta ja toinen käyttäen OpenGL ES 1.x piirtoprosessia.



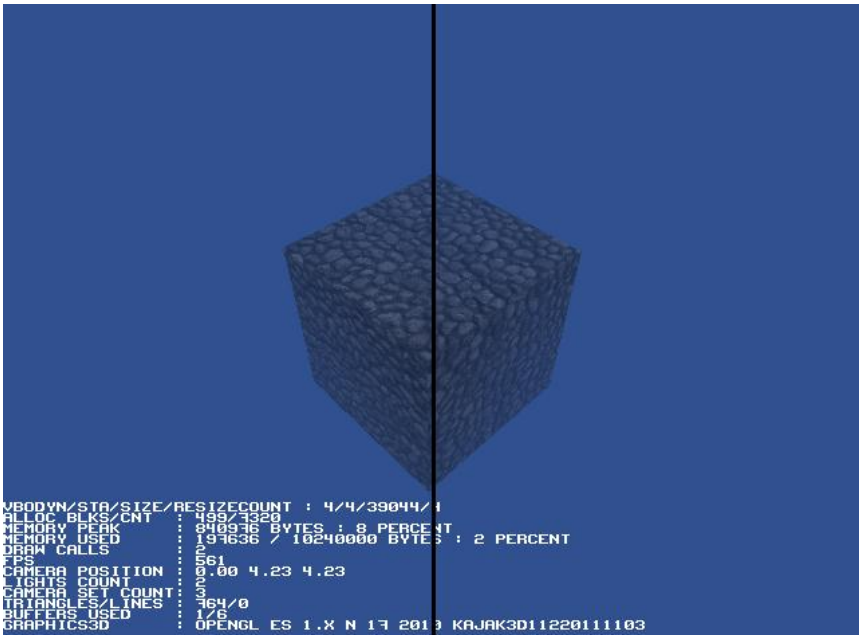
Kuvio 25. Kuutio ilman tekstuureita, vasemmalla OpenGL ES 1.x ja oikealla FFP-varjostin.

Muunnokset toimivat oikein. Verteksit ovat molemmissa sovelluksissa samoissa kohdissa. Kappale on samankokoinen ja muotoinen. Kappaleen valaistuessa oikein ovat myös normaalit oikeassa asennossa. Myös tekstuurimuunnos toimii oikein tekstuurien ollessa oikein päin, eikä tekstuureissa ole havaittavissa venymistä tai poikkeamaa toisiinsa nähden. Valaistus toimii, mutta liian himmeästi (Kuvio 25.). Syy tälle ilmiölle ei ole selvinnyt. Valon vaimentuminen on toteutettu samaa kaavaa käyttäen kuin JSR-184 määrittely ehdottaa, joten ilmiön ei pitäisi sen seurausta.



Kuvio 26. Vertailussa vasemmalla oleva FFP-varjostin ja oikealla oleva OpenGL ES 1.x.

Varjostinohjelman ja aidon OpenGL ES 1.x:n piirtojälki poikkeaa selkeästi (Kuvio 26.). Varjostinohjelman lopputulos on synkempi, mutta varjostukseltaan tasaisempi. ES 1.x piirtoprosessin värinvaihtelut ovat teräväreunaisia, kun taas varjostinohjelman värit vaihtuvat sulavasti. Tekstuurit toimivat oikein, sillä jokainen tekstuurin lisäyskaava tuottaa saman tuloksen kuin OpenGL ES 1.x (LIITE 4). Sumu toimii identtisesti OpenGL ES 1.x:n kanssa (Kuvio 27.). Läpinäkyvyyden testaus toimii ja jos fragmentin alfa-arvo jää alle asetetun rajan, sitä ei piirretä. Vaikka varjostinohjelma ei tuo täysin samaa lopputulosta kuin OpenGL ES 1.x, se toimii lähes oikein ja voi toimia oletusvarjostimena. Huolestuttavinta varjostinohjelman toiminnassa on kuitenkin sen kuvataajuus, joka on noin 40 % alhaisempi kuin OpenGL ES 1.x:llä. Suurin jatkokehityksen haaste on nopeuden saavuttaminen.



Kuvio 27. OpenGL ES 1.x:n lineaarinen sumu vasemmalla ja FFP-varjostimen lineaarinen sumu oikealla.

Varjostinohjelman toimivuus on myös testatattu Maemoa käyttävällä Nokia N900 puhelimella, jossa varjostimen toiminnallisuus oli hidasta. Maemon kuvataajuus FFP-varjostimen ajohetkellä oli noin 25, kun taas OpenGL ES 1.x kuvataajuus oli 50. Tämä on huolestuttvan vähän, sillä testisovelluksessa ruudulle ei piirretä kuin yksi kuutio ja statistiikkatekstiä. Tuntemattomasta syystä varjostimen kehityksen loppupuolella, myös varjostimen visuaalinen toiminta alkoi toimia virheellisesti. Kuutio piirtyy ruudulle mustana. Tämä saattaa olla seurausta tehdyistä muutoksista varjostinohjelmaan tai Kajak3D kehitysympäristöön.

4.3 Jatkokehitysmahdollisuudet

Varjostinohjelma toteuttaa OpenGL ES 2.x:ssä puuttuvan toiminnallisuuden, joten sen toiminnoissa ei ole puutteita. Nopeudessa tosin on parantamisen varaa. Varjostinohjelma on yhä kaukana ihanteellisesta kuvataajuudesta ja lähdekoodi on kaukana optimoidusta. Jatkokehityksessä tulisi keskittyä eniten varjostinohjelman nopeuden korottamiseen. Saattaa myös olla, että tämänhetkisten laitteiden tehot eivät riitä näin laajan varjostinohjelman toteuttamiseen. Tehokkaampien laitteiden saapumista odotellessa voi siis varjostinohjelman toiminnallisuutta koittaa karsia vähäisimmälle tarvittavalle tasolle.

Lisäksi varjostinohjelman toimivuutta eri puhelinmalleilla voisi testata. Varjostinohjelman toiminta on todettu Maemo ja Bada-alustoilla, mutta Kajak3D:n tukemien alustojen kaarti on laajempi. Varjostinohjelman toimintaa ei ole vielä testattu esimerkiksi iOS tai Android ympäristöissä.

5 POHDINTA

Opinnäytetyötä voi pitää onnistuneena sillä se sisältää olennaiset asiakokonaisuudet varjostimen kehittämisestä Kajak3D:lle. Tuoreen Kajak3D:n pelimoottorin varjostintoiminnallisuus oli vielä testaamatonta, joka loi ongelmakohtia varjostimien kehittämisessä myös sovelluksen puolella. Tämä on toki yksi painavimpia syitä miksi opinnäytetyö oli tärkeää tehdä. Kajak3D:n varjostintoiminnallisuus on nyt entistä kehittyneempää.

Fixed function pipeline-varjostimen teossa joitakin asioita olisi ollut mahdollista tehdä toisin. Esimerkiksi varjostinohjelma olisi ollut mahdollista toteuttaa varjostimien kehittämiseen tarkoitetulla ohjelmalla ja kehittää tuon varjostimen pohjalta lopullinen varjostin. Vertailuoperaatioiden määrää olisi voinut pyrkiä vähentämään ja toteuttaa varjostin käyttäen aina oletustoiminnallisuutta, mutta tuolloin varjostin ei olisi toiminut niin kuin sen on tarkoitettu toimivan. Nykyinen varjostin toimii ohjeiden mukaisesti, eikä sen nopeutta voi parantaa vähentämättä toiminnallisuutta. Nopeuteen vaikuttaa luonnollisesti myös vähäiset laitteiden laskentatehot, joten tällä hetkellä varjostinohjelmaa ei olisi voinut toteuttaa paljoa paremmin.

FFP-varjostinohjelmaa voi pitää onnistuneena, koska suurin osa sen toiminnallisuudesta toimii moitteettomasti, huolimatta pienistä eroavuuksista tavoitteeseen nähden. Varjostinohjelman piirtojälki näyttää OpenGL ES 1.x piirtojälkeä tasokkaammalta, vaikka tavoite oli pyrkiä samaan lopputulokseen. Koska kyseessä on kuitenkin grafiikan tason nousu, ei tätä voi pitää kovinkaan suurena virheenä. Suurempana ongelmana voisi kuitenkin pitää valaistuksen synkkyyttä.

FFP-varjostinohjelman hitaus verrattuna OpenGL ES 1.x:ään tuli pettymyksenä, mutta odotettuna sellaisena. Varjostin on aivan liian laaja toimiakseen joka tilanteessa. Varjostinohjelmalla on muuttujat jokaista tilannetta varten vaikka se ei niitä käyttäisikään. Esimerkiksi varying muuttujat varataan jokaiselle tekstuurikoordinaatille, vaikka samoja tekstuurikoordinaatteja voisi käyttää päällekkäisten tekstuurien kanssa. Paras ratkaisu nopeuden lisäämiselle olisi tarpeisiin sopivan varjostinohjelman räätälöinti. Kaikki mitä varjostimessa ei tarvita poistettaisiin vertailuoperaatioiden myötä.

Varjostimen kokoa voitaisiin myös kutistaa luomalla pienempikokoisia varjostimia, jotka linkitettäisiin tarpeen mukaan yhdeksi kokonaisuudeksi. Esimerkiksi sumu voi olla exponentaalista tai lineaarista, mutta sumun tyyppiä ei tulisi tarkistaa jokaista verteksiä kohti.

Luomalla sumua varten kaksi varjostinta, jotka käsittelevät kumpikin vain toisen sumun laskentakaavan, voitaisiin käyttöön haluttava versio liittää varjostinohjelmaan. Sama keino toimisi myös valojen ja tekstuurien laskentaan. Lisäksi hajavaloja on ympäristössä vain yksi, joten sen voisi poistaa valo tyypin vertailusta ja korvata se aina lisättävällä muuttujalla. Myös valojen jotkin ominaisuudet voisi lähettää eri tavoin, kuten vaimenemiseen liittyvät muuttujat. Vaimeneminen määritellään kolmella float muuttujalla, mutta ne voisi lähettää myös yhtenä vec3:na.

Opinnäytetyön tekeminen on opettanut minulle kuinka OpenGL ES varjostimet poikkeavat GLSL varjostimista, joiden toteuttaminen oli minulle jo entuudestaan tuttua. GLSL varjostimet tulevat olemaan GLSL ES varjostimien kaltaisia, joten opinnäytetyön tekemisestä olen saanut hyötyä myös pöytäkoneille suunnattujen sovelluksien kehityksessä. Opinnäytetyö on prosessina opettanut työskentelyä pitkäkestoisen projektin parissa ja lisännyt lähdekriittisyyttäni.

LÄHTEET

KIRJALLISUUS

Kessenich, J. 2009. The OpenGL® Shading Language. PDF saatavilla: http://www.opengl.org/registry/doc/GLSLangSpec.1.50.11.pdf (luettu 7.11.2010)

Munshi, A., Ginsburg, D. & Shreiner, D. 2008. OpenGL ES 2.0 Programming Guide. Boston: Pearson Education, Inc.

Olsson, E. 2008. Opinnäytetyö: Mobile Phone 3D Games with OpenGL ES 2.0. Royal Institute of Technology. PDF saatavilla: http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2008/rapporter08/olsson_erik_08025.pdf (luettu 31.10.2010)

Pulli, K., Aarnio, T., Miettinen, V., Roimela, K. & Vaarala, J. 2008. Mobile 3D Graphics with OpenGL ES and M3G. Burlington: Morgan Kaufmann.

Rost, R. 2007. OpenGL Shading Language Second Edition. Boston: Pearson Education.

Simpson, R. 2009. The OpenGL® ES Shading Language. PDF saatavilla: http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf (luettu 8.11.2010)

INTERNET LÄHTEET

AMD 2008. RenderMonkey Toolsuite. http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx (luettu 10.11.2010)

Android Developers 2010 a. What is Android? http://developer.android.com/guide/basics/what-is-android.html (luettu 16.11.2010)

Android Developers 2010 b. Android 2.2 Platform. http://developer.android.com/sdk/android-2.2.html (luettu 16.11.2010)

ARM 2009. Mali GPU OpenGL ES – Application Development Guide. http://www.malideveloper.com/files/DUI0363D_opengl_es_app_dev_guide.pdf (luettu 10.11.2010)

ARM 2010. Mali GPU Shader Development Studio. http://infocenter.arm.com/help/topic/com.arm.doc.dui0504b/DUI0504B_mali_gpu_shader_development_studio_ug.pdf (luettu 10.11.2010)

Cloward, B. 2006. Creating and using normal maps http://www.bencloward.com/tutorials_normal_maps1.shtml (luettu 28.9.2010).

Ginsburg, D. 2006. OpenGL ES 2.0: Shaders Go Mobile. http://developer.amd.com/gpu_assets/GDCMobile2006-Ginsburg-OpenGLES2.0.pdf (luettu 17.10.2010)

Ginsburg, D. 2007. OpenGL ES 2.0: Start Developing Now. http://developer.amd.com/assets/GDC2007_Ginsbug_Dan_OpenGL_ES_20.pdf (luettu 17.11.2010)

HTC 2010. HTC Desire - Specification. http://www.htc.com/www/product/desire/specification.html (luettu 18.11.2010)

James, G. & O'Rorke, J. 2004. Real-Time Glow. http://www.gamasutra.com/view/feature/2107/realtime_glow.php (luettu 5.11.2010)

Jouvie, J. 2010. Lesson 8 : Tangent Space. http://jerome.jouvie.free.fr/opengl-tutorials/Lesson8.php (luettu 8.10.2010)

Kajak3D 2010. Kajak3D Pelimoottori. http://kajak3d.com/index.php?page=pelimoottori (luettu 2.3.2010).

Kalogirou, C. 2006. How to do a good bloom for HDR rendering. http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/ (luettu 5.11.2010)

Khronos 2010 a. OpenGL ES Overview. http://www.khronos.org/opengles/ (luettu 2.3.2010).

Khronos 2010 b. About the Khronos Group. http://www.khronos.org/about/ (luettu 2.3.2010).

Khronos 2010 c. OpenGL ES 2.X and the OpenGL ES Shading Language. http://www.khronos.org/opengles/2_X/ (luettu 28.9.2010)

Lighthouse 3D 2010. GLSL Tutorial. http://www.lighthouse3d.com/opengl/glsl/index.php?oglvariables (luettu 17.10.2010)

Maemo 2009. Software Platform. http://maemo.org/intro/platform/ (luettu 28.10.2010)

Movial 2010. Scratchbox. http://www.scratchbox.org/ (luettu 18.16.2010)

Mäki, M. 2007. OpenGL ES 2.0 ratifioitiin yksimielisesti. http://www.digitoday.fi/data/2007/03/08/opengl-es-20-ratifioitiin-yksimielisesti/20075881/66 (luettu 10.10.2010)

Nokia 2005. Mobile 3D Graphics API. http://jcp.org/aboutJava/communityprocess/mrel/jsr184/index.html (luettu 2.3.2010)

Nokia 2010 a. Get to know Maemo. http://maemo.nokia.com/maemo/ (luettu 28.10.2010)

Nokia 2010 b. Nokia N900 mobile computer. http://maemo.nokia.com/n900/ (luettu 28.10.2010)

NVIDIA 2009. NVIDIA FX Composer 2.5 GPU Shader Authorin Environment. http://developer.nvidia.com/object/fx_composer_home.html (luettu 10.11.2010)

Opengles-book.com 2010. OpenGL ES 2.0 Programming Guide – Downloads. http://opengles-book.com/downloads.html

Penfold, D. 2002. Primitive Processing : Vertex Shaders and Pixel Shaders. http://www.tomshardware.com/reviews/vertex-shaders-pixel-shaders,411-2.html (luettu 10.11.2010)

Polysplendor 2009. Water Shader Made Easy! http://polysplendor.com/?p=107 (luettu 29.10.2010)

KUVAT

Jouvie, J. 2010. http://jerome.jouvie.free.fr/images/OpenGl/Lessons/Lesson8-AxisSystems.png (luettu 7.11.2010)

Kalogirou, J. 2006. http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/ (luettu 29.10.2010)

Khronos 2007 a. http://www.khronos.org/opengles/2_X/img/opengles_1x_pipeline.gif (luettu 29.10.2010)

Khronos 2007 b. http://www.khronos.org/opengles/2_X/img/opengles_20_pipeline.gif (luettu 29.10.2010)

Polysplendor 2009. http://polysplendor.com/wp-content/uploads/2009/06/horizon001.jpg (luettu 29.10.2010)

Sanglard, F. 2004. http://www.fabiensanglard.net/bumpMapping/maxTomd5.jpg (luettu 5.11.2010)

Touch Arcade 2010. http://toucharcade.com/wp-content/uploads/2010/01/ZB1vs2.jpg (luettu 30.10.2010)

Tsiombikas, J. 2003. http://downloads.gamedev.net/pdf/VolumeShadowsTutorial-2036.pdf (luettu 29.10.2010)

JULKAISEMATTOMAT LÄHTEET

Kajak3D 2010. Kajak3D pelimoottorin lähdekoodi. Kajaanin ammattikorkeakoulu.

LIITTEET

LIITE 1 Kajak3D:n varaamat ESSL muuttujat

LIITE 2 FFP-varjostimen verteksivarjostimen lähdekoodi

LIITE 3 FFP-varjostimen fragmenttivarjostimen lähdekoodi

LIITE 4 FFP-varjostimen ja OpenGL ES 1.x teksturoinnin vertailu

Verteksi- ja fragmenttivarjostimen oletustarkkuudet
`precision mediump float;`
`precision lowp int;`

Verteksivarjostimen alkuun lisättävät muuttujat ja vakiot	
Verteksivarjostimen Kajak3D vakiot	
Vakion esittely	Vakion arvo
`#define SHADER_FOG_ENABLED`	`0`
`#define SHADER_TEXTURING_ENABLED`	`1`
`#define SHADER_TEXTURE0_ENABLED`	`2`
`#define SHADER_TEXTURE1_ENABLED`	`3`
`#define SHADER_TEXTURE2_ENABLED`	`4`
`#define SHADER_TEXTURE3_ENABLED`	`5`
`#define SHADER_LIGHTS_ENABLED`	`6`
`#define SHADER_VERTEXARRAY_COLOR_EXISTS`	`7`
`#define SHADER_MATERIAL_VERTEXCOLOR_TRACKING`	`8`
`#define SHADER_TABLE_SIZE`	`9`
`#define MAX_LIGHTS`	`8`
`#define AMBIENT_LIGHT`	`128.0`
`#define DIRECTIONAL_LIGHT`	`129.0`
`#define OMNI_LIGHT`	`130.0`
`#define SPOT_LIGHT`	`131.0`
`#define EXPONENTIAL_FOG`	`80.0`
`#define LINEAR_FOG`	`81.0`
Verteksivarjostimen Kajak3D muuttujat	
Muuttujan esittely	Muuttujan kuvaus
`attribute vec2 m3g_texcoord0,1,2,3`	`Verteksin tekstuurikoordinaatit.`
`attribute vec3 m3g_position`	`Verteksin sijainti mallikoordinaatistossa.`
`attribute vec3 m3g_normal`	`Verteksin normaali.`
`attribute vec4 m3g_color`	`Verteksikohtainen väriarvo.`
`uniform float m3g_featuresFlags[SHADER_TABLE_SIZE]`	`Taulukko Graphics3D:n asetuksista.`
`uniform mat4 m3g_modelViewProjectionMatrix`	`Malli-näkymä-projektio-matriisi.`
`uniform mat4 m3g_modelViewMatrix`	`Malli-näkymä matriisi.`
`uniform mat4 m3g_textureMatrix0,1,2,3`	`Tekstuurimatriisit.`
`uniform mat4 m3g_normalMatrix`	`Normaalimatriisi.`
`uniform mat4 m3g_meshAndCamera_Matrix`	`Malli ja kamera matriisi.`
`uniform mat4 m3g_vertexColor`	`Piirtoväri.`
`uniform int m3g_lightType[8]`	`Valotyyppi.`
`uniform mat4 m3g_lightMatrix[8]`	`Valomatriisi valolle (1-8).`
`uniform vec3 m3g_lightDirection[8]`	`Valon suunta.`
`uniform vec4 m3g_lightDiffuse[8]`	`Valon diffuse-väriarvo.`
`uniform vec4 m3g_lightSpecular[8]`	`Valon specular-väri.`
`uniform vec4 m3g_lightAmbient[8]`	`Valon ambient-väriarvo.`
`uniform vec3 m3g_lightSpotDirection[8]`	`Kohdevalon suunta.`
`uniform float m3g_lightConstantAttenuation[8]`	`Valon tasainen vaimennus arvo.`
`uniform float m3g_lightLinearAttenuation[8]`	`Valon lineaarinen vaimennus arvo.`
`uniform float m3g_lightQuadraticAttenuation[8]`	`Valon kvadraattinen vaimennus arvo.`
`uniform float m3g_lightSpotCutoff[8]`	`Kohdevalon leveyskulma.`
`uniform float m3g_lightSpotExponent[8]`	`Kohdevalon eksponentti.`
`uniform float m3g_materialShininess`	`Materiaalin kiiltävyys.`
`uniform vec4 m3g_materialAmbient`	`Materiaalin ympäröivä väri.`
`uniform vec4 m3g_materialDiffuse`	`Materiaalin hajaväri.`
`uniform vec4 m3g_materialSpecular`	`Materiaalin peiliheijastus väri.`
`uniform vec4 m3g_materialEmissive`	`Materiaalin emissioväri.`
`uniform float m3g_fogMode`	`Sumun laskentakaava.`
`uniform float m3g_fogDensity`	`Sumun tiheys.`
`uniform float m3g_fogNearDistance`	`Sumun lähietäisyys.`
`uniform float m3g_fogFarDistance`	`Sumun kaukoetäisyys.`

Verteksivarjostimen Kajak3D muuttujat, jotka vaativat esittelyn varjostimen lähdekoodissa	
Muuttujan esittely	Muuttujan kuvaus
`attribute vec3 binormal`	Binormaali tangenttiavaruutta varten.
`attribute vec3 tangent`	Tangentti tangenttiavaruutta varten.

Fragmenttivarjostimen alkuun lisättävät muuttujat ja vakiot	
Fragmenttivarjostimen Kajak3D vakiot	
Vakion esittely	Vakion arvo
`#define SHADER_FOG_ENABLED`	`0`
`#define SHADER_TEXTURING_ENABLED`	`1`
`#define SHADER_TEXTURE0_ENABLED`	`2`
`#define SHADER_TEXTURE1_ENABLED`	`3`
`#define SHADER_TEXTURE2_ENABLED`	`4`
`#define SHADER_TEXTURE3_ENABLED`	`5`
`#define SHADER_LIGHTS_ENABLED`	`6`
`#define SHADER_VERTEXARRAY_COLOR_EXISTS`	`7`
`#define SHADER_MATERIAL_VERTEXCOLOR_TRACKING`	`8`
`#define SHADER_TABLE_SIZE`	`9`
`#define FUNC_ADD`	`224.0`
`#define FUNC_BLEND`	`225.0`
`#define FUNC_DECAL`	`226.0`
`#define FUNC_MODULATE`	`227.0`
`#define FUNC_REPLACE`	`228.0`
Fragmenttivarjostimen Kajak3D muuttujat	
Muuttujan esittely	Muuttujan kuvaus
`uniform float m3g_featuresFlags[SHADER_TABLE_SIZE]`	Taulukko Graphics3D:n asetuksista.
`uniform sampler2D m3g_texture0,1,2,3`	Mallissa käytetyt tekstuurit.
`uniform float m3g_textureBlendingMode0,1,2,3`	Tekstuurin liittämisessä käytetty kaava.
`uniform vec4 m3g_textureBlendColor0,1,2,3`	Blend-väri FUNC_BLEND laskentakaavaa varten.
`uniform float m3g_alphaTreshold`	Läpinäkyvyystestauksen kynnysarvo.
`uniform vec4 m3g_fogColor`	Sumun väriarvo.

```
// global variables in vertex shader
vec4 AmbientLight;
vec4 DiffuseLight;
vec4 SpecularLight;
vec4 M3Gposition;
vec3 M3Gnormal;

// passed variables
varying vec2        m3g_out_texcoord[4];
varying vec4        m3g_out_vColor;
varying float       m3g_out_fogFactor;
varying vec4        m3g_out_specularIntensity;

// function prototypes
void m3g_ffunction();
void HandleLight(in int i, in vec3 ePos);
void DirectionalLight(in int i, in vec3 normal, in vec3 ePos);
void SpotLight(in int i, in vec3 normal, in vec3 ePos);
void PointLight(in int i, in vec3 normal, in vec3 ePos);

// LIGHTING RELEATED FUNCTIONS
void HandleLight(in int i, in vec3 ePos)
{
	if (m3g_lightType[i] != 0)
	{
		// DIRECTIONAL LIGHT
		if (m3g_lightType[i] == DIRECTIONAL_LIGHT)
		{
			DirectionalLight(i, M3Gnormal, ePos);
		}

		// SPOT LIGHT
		else if (m3g_lightType[i] == SPOT_LIGHT)
		{
			SpotLight(i, M3Gnormal, ePos);
		}

		// POINT LIGHT
		else if (m3g_lightType[i] == OMNI_LIGHT)
		{
			PointLight(i, M3Gnormal, ePos);
		}

		// AMBIENT LIGHT
		else if (m3g_lightType[i] == AMBIENT_LIGHT)
		{
			AmbientLight += m3g_lightAmbient[i];
		}
	}
}

void DirectionalLight(in int i, in vec3 normal, in vec3 ePos)
{
	// Compute the light direction vector
	vec3 lightPos = vec3(m3g_lightDirection[i]);
	vec3 lightDir = normalize(lightPos);

	// Compute the half vector between eye position and light position
	vec3 halfV = normalize(-ePos + lightPos);

	// Compute the diffuse light intensity
	float dVP = max(dot(normal, lightDir), 0.0);
```

```
	// Compute the approximated specular light base intensity
	float dHV = max(dot(normal, halfV), 0.0);

	// Power Factor for specular intensity
	float pf;

	if (dVP > 0.0)
		pf = pow(dHV, m3g_materialShininess);
	else
		pf = 0.0;

	// Add the lighting values from this light to total vertex lighting
	DiffuseLight  += m3g_lightDiffuse[i] * dVP;
	SpecularLight += m3g_lightSpecular[i] * pf;
}

void SpotLight(in int i, in vec3 normal, in vec3 ePos)
{
	// Compute light position
	vec3 lightPosition = vec3(m3g_lightMatrix[i] * vec4(0,0,0,1.0));

	// Compute the distance vector between light and vertex position
	float d = length(lightPosition - ePos);

	// Compute the vector from vertex to light
	vec3 VP = normalize(lightPosition - ePos);

	// Compute parallelism between the spot light direction and the vector
from vertex to light.
	// The more parallel the two vectors are, the larger the spotDot will be.
	// vec3(0,0,-1) is the spot light direction, which change as you like.
	mat4 spotMat = m3g_lightMatrix[i];
	spotMat[3].x = 0.0;
	spotMat[3].y = 0.0;
	spotMat[3].z = 0.0;
	vec4 spotDir = normalize(spotMat * m3g_lightSpotDirection[i]);

	float spotDot = dot(-VP, vec3(spotDir));

	// Decide is the vertex inside the spot light
	// If vertex is inside, calculate the decay caused by the distance from
center spot
	float att;
	if (acos(spotDot) > (0.01744*m3g_lightSpotCutoff[i]))
	{
		att = 0.0;
	}
	else
	{
		att = 1.0 / (m3g_lightConstantAttenuation[i] +
				m3g_lightLinearAttenuation[i] * d +
				m3g_lightQuadraticAttenuation[i] * d * d);
	}

	// Compute the half vector between eye position and light position
	vec3 halfV = normalize(-ePos + lightPosition);

	// Compute the diffuse light intensity
	float dVP = max(dot(normal, VP), 0.0);

	// Compute the approximated specular light base intensity
```

```
	float dHV = max(dot(normal, halfV), 0.0);

	// Power factor for speculat lighting
	float pf;

	if (dVP > 0.0)
	{
		pf = pow(dHV, m3g_materialShininess);
	}
	else
	{
		pf = 0.0;
	}

	// Add the lighting values from this light to total vertex lighting
	DiffuseLight  += m3g_lightDiffuse[i] * dVP * att;
	SpecularLight += m3g_lightSpecular[i] * pf * att;
}

void PointLight(in int i, in vec3 normal, in vec3 ePos)
{
	// Compute light position
	vec3 lightPosition = vec3(m3g_lightMatrix[i] * vec4(0,0,0,1.0));

	// Compute distance between light and and vertex position
    float d = length(lightPosition - ePos);

	// Compute attenuation
	float att = 1.0 / (m3g_lightConstantAttenuation[i] +
				m3g_lightLinearAttenuation[i] * d +
				m3g_lightQuadraticAttenuation[i] * d * d);

    // calculate the vector from vertex to light.
    vec3 VP = normalize(lightPosition - ePos);

    // calculate the half vector between eye position and light position.
    vec3 halfV = normalize(-ePos + lightPosition);

    // calculate the diffuse light intensity.
    float dVP = max(dot(normal,VP), 0.0);

    // calculate approximated specular light base intensity.
    float dHV = max(dot(normal,halfV), 0.0);

	// Power factor for specular light
	float pf;

	if (dVP == 0.0)
		pf = 0.0;
	else
		pf = pow(dHV, m3g_materialShininess);

	// Add the lighting values from this light to total vertex lighting
	DiffuseLight  +=  m3g_lightDiffuse[i] * dVP * att;
	SpecularLight +=  m3g_lightSpecular[i] * pf * att;
}

void m3g_ffunction()
{
	vec4 baseColor;

	if (m3g_featuresFlags[SHADER_VERTEXARRAY_COLOR_EXISTS] == 1.0)
```

```
	{
		// Vertex color needs to be converted from 0-255 scale to 0-1f
scale
		baseColor = m3g_color * 0.003921568627451;
	}
	else
	{
		// Use default vertex color value
		baseColor = m3g_vertexColor;
	}

	// The color sum of lighting, material, vertex etc.
	vec4 color = vec4(0.0);

	// HANDLE TRANSFORM
	// Transform vertex to clip space
	M3Gposition = m3g_modelViewProjectionMatrix * vec4(m3g_position, 1.0);

	// Transform vertex position to eye space
	vec3 ecPosition = vec3(m3g_meshAndCamera_Matrix * vec4(m3g_position,
1.0));

	// Texture coordinates
	m3g_out_texcoord[0] = vec2(m3g_textureMatrix0 * vec4(m3g_texcoord0, 0.0,
1.0));
	m3g_out_texcoord[1] = vec2(m3g_textureMatrix1 * vec4(m3g_texcoord1, 0.0,
1.0));
	m3g_out_texcoord[2] = vec2(m3g_textureMatrix2 * vec4(m3g_texcoord2, 0.0,
1.0));
	m3g_out_texcoord[3] = vec2(m3g_textureMatrix3 * vec4(m3g_texcoord3, 0.0,
1.0));

	// HANDLE LIGHTING
	if (m3g_featuresFlags[SHADER_LIGHTS_ENABLED] == 0.0)
	{
		color = baseColor;
		m3g_out_specularIntensity = vec4(0.0);
	}
	else
	{
		// clear the light intensity accumulators
		AmbientLight  = vec4(0.0);
		DiffuseLight  = vec4(0.0);
		SpecularLight = vec4(0.0);

		// Transform normal to eye space
		M3Gnormal = normalize(vec3(m3g_normalMatrix * vec4(m3g_normal,
1.0)));

		HandleLight(0, ecPosition);
		HandleLight(1, ecPosition);
		HandleLight(2, ecPosition);
		HandleLight(3, ecPosition);
		HandleLight(4, ecPosition);
		HandleLight(5, ecPosition);
		HandleLight(6, ecPosition);
		HandleLight(7, ecPosition);

		// HANDLE COLOUR SUM
		color += m3g_materialEmissive;

		if (m3g_featuresFlags[SHADER_MATERIAL_VERTEXCOLOR_TRACKING] ==
1.0)
```

```
		{
			color += AmbientLight * baseColor;
			color += DiffuseLight * baseColor;
		}
		else
		{
			color += AmbientLight * m3g_materialAmbient;
			color += DiffuseLight * m3g_materialDiffuse;
		}

		// Handle specularity in fragment shader
		m3g_out_specularIntensity = SpecularLight * m3g_materialSpecular;
	}

	// HANDLE FOG
	if (m3g_featuresFlags[SHADER_FOG_ENABLED] == 1.0)
	{
		float fogFactor = 0.0;
		float z = length(ecPosition);

		if (m3g_fogMode == LINEAR_FOG)
		{
			float distance = m3g_fogFarDistance - m3g_fogNearDistance;
			if (distance != 0.0)
			{
				fogFactor = (m3g_fogFarDistance - z) / distance;
			}
			else
			{
				fogFactor = exp(-m3g_fogDensity * z);
			}
		}
		else if (m3g_fogMode == EXPONENTIAL_FOG)
		{
			fogFactor = exp(-m3g_fogDensity * z);
		}

		fogFactor = clamp(fogFactor, 0.0, 1.0);
		m3g_out_fogFactor = fogFactor;
	}

	m3g_out_vColor = color;
}

void main()
{
	m3g_ffunction();
	gl_Position = M3Gposition;
}
```

```
// passed variables from vertex shader
varying vec2        m3g_out_texcoord[4];
varying vec4        m3g_out_vColor;
varying float       m3g_out_fogFactor;
varying vec4        m3g_out_specularIntensity;

// function prototypes
void m3g_ffunction();
void HandleTexture(in sampler2D texture, in float enabled, in int blendingMode,
in vec4 blendColor, in vec2 texcoord, inout vec4 color);

void HandleTexture(in sampler2D texture, in float enabled, in int blendingMode,
in vec4 blendColor, in vec2 texcoord, inout vec4 color)
{
       if (enabled == 1.0)
       {
              vec4 texel = texture2D(texture, texcoord);

              if (blendingMode == FUNC_MODULATE)
              {
                     color *= texel;
              }

              else if (blendingMode == FUNC_ADD)
              {
                     color.rgb += texel.rgb;
                     color.a        *= texel.a;
                     color     = clamp(color, 0.0, 1.0);
              }

              else if (blendingMode == FUNC_BLEND)
              {
                     vec3 col = mix(color.rgb, blendColor.rgb, texel.rgb);
                     color = vec4(col, color.a * texel.a);
              }

              else if (blendingMode == FUNC_DECAL)
              {
                     vec3 col = mix(color.rgb, texel.rgb, texel.a);
                     color = vec4(col, color.a);
              }

              else if (blendingMode == FUNC_REPLACE)
              {
                     color = texel;
              }
       }
}

void m3g_ffunction(out vec4 color)
{
       // Get the base color from the vertex color
       color = m3g_out_vColor;

       // TEXTURE ENVIRONMENT
       if (m3g_featuresFlags[SHADER_TEXTURING_ENABLED] == 1.0)
       {
              HandleTexture(m3g_texture0,
m3g_featuresFlags[SHADER_TEXTURE0_ENABLED], m3g_textureBlendingMode0,
m3g_textureBlendColor0, m3g_out_texcoord[0], color);
```

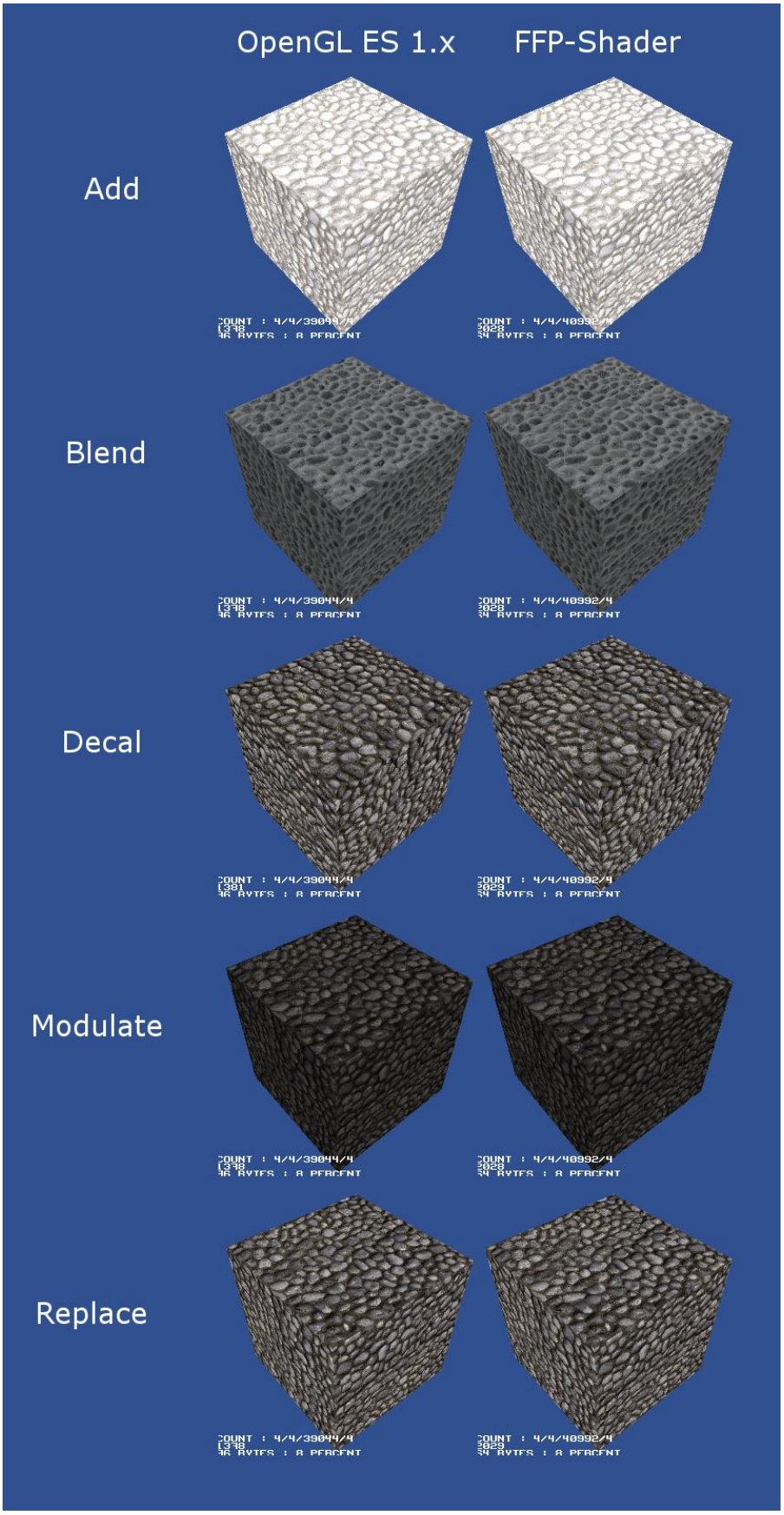
```
            HandleTexture(m3g_texture1,
m3g_featuresFlags[SHADER_TEXTURE1_ENABLED], m3g_textureBlendingMode1,
m3g_textureBlendColor1, m3g_out_texcoord[1], color);
            HandleTexture(m3g_texture2,
m3g_featuresFlags[SHADER_TEXTURE2_ENABLED], m3g_textureBlendingMode2,
m3g_textureBlendColor2, m3g_out_texcoord[2], color);
            HandleTexture(m3g_texture3,
m3g_featuresFlags[SHADER_TEXTURE3_ENABLED], m3g_textureBlendingMode3,
m3g_textureBlendColor3, m3g_out_texcoord[3], color);
      }

      // COLOUR SUM
      color += m3g_out_specularIntensity;

      // HANDLE FOG
      if (m3g_featuresFlags[SHADER_FOG_ENABLED] == 1.0)
      {
            color = mix(m3g_fogColor, color, m3g_out_fogFactor);
            color.a = m3g_out_fogFactor;
      }

      // HANDLE ALPHA TESTING
      if (color.a < m3g_alphaTreshold)
      {
            discard;
      }
}

void main()
{
      vec4 color;
      m3g_ffunction(color);
      gl_FragColor = color;
}
```


OpenGL ES 1.x
FFP-Shader
Add
Blend
Decal
Modulate
Replace