Kimmo Parsama

Reaaliaikaiset jälkikäsittelyn algoritmit

Opinnäytetyö

Kevät 2013

Tekniikan yksikkö

Tietotekniikan koulutusohjelma

Seinäjoen ammattikorkeakoulu
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan Yksikkö

Koulutusohjelma: Tietotekniikan koulutusohjelma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Parsama, Kimmo

Työn nimi: Reaaliaikaiset jälkikäsittelyn algoritmit

Ohjaaja: Mäkelä, Petteri

Vuosi: 2013 Sivumäärä: 40 Liitteiden lukumäärä: 0

Työn tarkoituksena oli tutusta pelikehityksessä käytettäviin reaaliaikaisiin jälkikäsittelyalgoritmeihin. Näitä algoritmeja voidaan käyttää parantamaan videopelien tai reaaliaikaisten animaatioiden kuvanlaatua.

Työssä tutkittiin ensin jälkikäsittelyä ja digitaalista kuvankäsittelyä yleisesti. Tämän jälkeen käytiin eräiden jälkikäsittelyn algoritmien teoriaa läpi. Lopuksi kaikki esitellyt algoritmit toteutettiin ja testattiin.

Työssä käydyt algoritmit antavat grafiikkaohjelmoijille ja pelinkehittäjille hyvän pohjan, mistä voi lähteä kehittämään oman pelimoottorin jälkikäsittelyä.

Avainsanat: Jälkikäsittely, GLSL, OpenGL, varjostin

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Software Development

Author: Parsama, Kimmo

Title of thesis: Real time post processing algorithms

Supervisor: Mäkelä, Petteri

Year: 2013 Number of pages: 40 Number of appendices: 0

The aim of this thesis was to study the use of post processing algorithms in video game development. These algorithms can be used to enhance the image quality of a game or a real time animation.

First, the use of post processing algorithms in games was examined. Then a short theory of each of the presented post processing algorithm was described. Finally, all the presented algorithms were implemented and tested.

The results presented here, give game developers a general guideline on how to implement a selection of the most popular post processing algorithms. This thesis can be used as a starting base for creating good looking real time computer graphics.

Keywords: post processing, GLSL, OpenGL, shader

SISÄLTÖ

Opinnäytetyön tiivistelmä ... 2
Thesis abstract ... 3
SISÄLTÖ ... 4
Kuvaluettelo ... 6
Käytetyt termit ja lyhenteet ... 7
1 Johdanto ... 8
1.1 Työn tausta ... 8
1.2 Työn tavoite ja rajaus ... 8
1.3 Työn rakenne ... 8
2 Videopelien jälkikäsittely ... 9
2.1 Digitaalinen kuvankäsittely ... 9
2.2 Jälkikäsittely ... 9
2.3 Grafiikkasuoritin ... 10
3 Jälkikäsittelyalgoritmien teoria ... 11
3.1 Ympäristön varjostus (Ambient occlusion) ... 11
3.1.1 Yleiskatsaus ... 13
3.1.2 Alinäytteistys ... 13
3.1.3 Näytteistys ... 14
3.1.4 Näytteiden laskeminen ... 15
3.1.5 Bilateraalinen alipäästösuodatin ... 19
3.2 Sumennukset (Blurs) ... 19
3.2.1 Laatikkosumennus (box blur) ... 20
3.2.2 Gaussin sumennus (Gaussian blur) ... 21
3.3 Syväterävyysalue (Depth of Field) ... 23
3.4 Hehku (Bloom) ... 27
4 Jälkikäsittelyalgoritmien toteutus ... 30
4.1 Ympäristön varjostus (Ambient Occlusion) ... 30
4.1.1 Alinäytteistys ... 31
4.1.2 Näytteiden laskeminen ... 32
4.1.3 Bilateraalinen ylinäytteistys ... 33

4.1.4 Bilateraalinen alipäästösuodatin ja sumennus 34
4.2 Sumennukset (Blurs) 34
4.2.1 Laatikkosumennus (Box blur) 35
4.2.2 Gaussin sumennus (Gaussian blur) 35
4.3 Syväterävyysalue (Depth of field) 36
4.4 Hehku (Bloom) 37
5 Yhteenveto 39
LÄHTEET 40

Kuvaluettelo

Kuva 1. Ympäristön varjostus eräässä huoneessa

Kuva 2. Lomitettu näytteistyskuvio

Kuva 3. Vasemmalla lähimmän naapurin ylinäytteistys, keskellä bilineaarinen ylinäytteistys ja oikealla bilateraalinen ylinäytteistys

Kuva 4. Ristikko ja pallo sumennettu oikealla käyttäen Gaussin sumennusta

Kuva 5. 3x3-alue pikseleitä

Kuva 6. Laatikkosumennus

Kuva 7. Gaussin sumennus

Kuva 8. Kaksiulotteisen Gaussin kaavan graafinen esitys

Kuva 9. Pascalin kolmio, binomikertoimia voidaan käyttää Gaussin painoina

Kuva 10. Syväterävyysalue on tarkka alue kuvan puolivälissä

Kuva 11. Linssin toiminta

Kuva 12. Yksinkertainen syväterävyysalueen toteutus

Kuva 13. Esimerkkikuva yksikkömuunnoksesta

Kuva 14. Oikealla kuvan kirkkaat valkoiset osat ovat vuotaneet mustien viivojen ja ympyrän päälle

Käytetyt termit ja lyhenteet

GLSL	GLSL(OpenGL Shading Language) on korkean tason varjostinkieli, jolla OpenGL-varjostimia ohjelmoidaan.
OpenGL	OpenGL(Open Graphics Library) on laitteistoriippumaton ohjelmointirajapinta grafiikkaohjelmointia varten.
Varjostin	Varjostin on yksittäinen GLSL-kielellä tehty ohjelma.
Pikseli	Pienen mahdollinen kuvan osa, joka määrittää kuvan värin siinä pisteessä.
AO	AO(Ambient Occlusion) eli ympäristön varjostus.
SSAO	SSAO(Screen Space Ambient Occlusion) eli näyttökoordinaatistossa tehtävä ympäristön varjostuksen laskenta.
Tekstuuri	Näytönohjaimelle varattu muistialue, joka voi pitää sisällään kuvia, normaalivektoreita ja muuta tietoa.
G-puskuri	Tekstuurikokoelma, johon tallennetaan piirretyn geometrian tietoja esimerkiksi normaalivektorit ja värit.
Tekseli	Pienin mahdollinen tekstuurin osa, joka määrittää tekstuurin arvon siinä pisteessä.

1 Johdanto

1.1 Työn tausta

Modernit videopelit panostavat entistä enemmän näyttävään grafiikkaan. Iso osa pelissä näkyvästä grafiikasta tehdään ennen varsinaista pelin julkaisua graafikkojen toimesta. Jälkikäsittelyssä grafiikkaohjelmoija ohjelmoi koodinpätkän, joka ajetaan sen jälkeen kun kaikki graafikkojen mallintamat mallit on piirretty peliin. Jälkikäsittelyn avulla voidaan helposti muuttaa pelin graafista ulkoasua jälkeenpäin. Jälkikäsittelyllä voidaan muun muassa korostaa pelin fotorealistisuutta, ohjata pelaajan katsetta ruudulla tai muuttaa pelin värimaailmaa.

1.2 Työn tavoite ja rajaus

Työn tavoitteena on tutustua erilaisiin reaaliaikaisiin videopelien jälkikäsittelyalgoritmeihin ja toteuttaa ne. Esiteltyjen algoritmien avulla lukija voi toteuttaa yksinkertaisen jälkikäsittelyn omalle pelilleen.

Työ rajoittuu tutkimaan vain keskeisimpiä jälkikäsittelyalgoritmeja. Algoritmien toteutus on kirjoitettu GLSL-varjostinkielellä, jota käytetään OpenGL-ohjelmointirajapinnan kanssa.

1.3 Työn rakenne

Ensimmäisessä luvussa käydään läpi opinnäytetyön taustaa. Luvussa kuvataan myös työn tavoite ja aiheen rajaus. Lopuksi esitellään työn rakenne.

Toisessa luvussa kerrotaan yleisesti digitaalisesta kuvankäsittelystä ja reaaliaikaisesta jälkikäsittelystä.

Kolmannessa luvussa käydään läpi työssä esiteltävät algoritmit yleisesti ja niiden teoria. Neljännessä luvussa kerrotaan algoritmien käytännön toteutuksesta GLSL-varjostinkielellä.

2 Videopelien jälkikäsittely

Tässä luvussa käydään läpi lyhyesti digitaalisen kuvankäsittelyn periaatteita ja miten ne liittyvät jälkikäsittelyyn. Lisäksi tarkastellaan myös kuinka jälkikäsittely toteutetaan ja mikä sen rooli on tietokonegrafiikassa.

2.1 Digitaalinen kuvankäsittely

Digitaalinen kuvankäsittely tarkoittaa digitaalisen kuvan muokkaamista ja säätämistä tietokoneella. Digitaalinen kuva koostuu äärellisestä määrästä pikseleitä, jotka määrittelevät kuvan värikirjon. Digitaalinen kuva voi olla peräisin monesta eri lähteestä. Se voi olla esimerkiksi kameralla otettu valokuva, käsin piirretty ja skannattu piirros tai tietokoneella generoitu kuva. Kuvassa olevien pikseleiden lukumäärä kertoo kuvan ulottuvuudet. Värikuvissa jokainen pikseli pitää sisällään kolme tai neljä värikanavaa. Kolme ensimmäistä kanavaa ovat punainen, vihreä sekä sininen, neljäs kanava on yleensä alpha- eli läpinäkyvyysarvo. (Lonroth & Unger 2009.)

2.2 Jälkikäsittely

Jälkikäsittely on kuvankäsittelytekniikka, jossa kuvaa yritetään parantaa, yleensä lisäinformaation avulla, esimerkiksi ympäristön syvyysarvoilla. Jälkikäsittely vaikuttaa merkittävästi siihen, miten kuvaa tulkitaan ja miltä se näyttää. Jälkikäsittely on näin ollen tärkeä osa tietokonegrafiikkaa. (Lonroth & Unger 2009.)

Jälkikäsittely tapahtuu sen jälkeen kun kuva on saatu 3D-piirrosta tai jostain muusta lähteestä. Jälkikäsittelyyn kuuluu muun muassa yksinkertaiset erikoisefektit, kuten harmaasävytys, sumennus, värikyllästys ja kontrastin säätö sekä kehittyneemmät efektit kuten liikesumennus, syväterävyysalue ja erilaiset varjot. (Lonroth & Unger 2009.)

Tavallisesti jälkikäsittelyssä saatavilla oleva informaatio on vähäisempi kuin normaalin piirron aikana. Esimerkiksi valot, objektit ja geometria on korvattu yhdellä

tai useammalla kuvalla. Yksinkertainen jälkikäsittely ei vaikuta tavalliseen piirtoon mitenkään, vaan se suoritetaan piirron tuottamalle kuvalle erillisenä toimenpiteenä. Kehittyneet jälkikäsittelyalgoritmit vaativatkin yleensä enemmän kuin pelkän värikuvan käyttöönsä, esimerkiksi syvyyspuskurin tai normaalitekstuurin. (Lonroth & Unger 2009.)

2.3 Grafiikkasuoritin

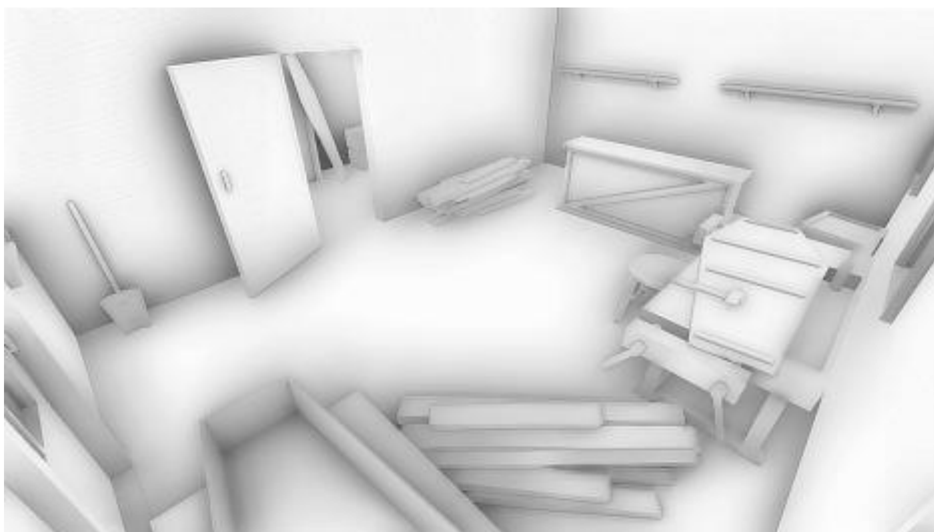
Grafiikkasuoritin on tietokoneessa erillinen prosessori, jonka tarkoituksena on nopeuttaa 2D- tai 3D-grafiikan laskentaa. Ennen kuin grafiikkasuorittimia oli olemassa, tietokoneen yleinen prosessori joutui laskemaan kaikki mahdolliset laskennat. Tämä oli kuitenkin hidasta, koska prosessori oli optimoitu laskemaan laskuja sarjassa eikä rinnakkain. Nykyaikaiset grafiikkasuorittimet ovat pitkälle erikoistuneita rinnakkaislaskentaan, jota tietokonegrafiikka vaatii. Lähes kaikki tietokonegrafiikan laskennat lasketaan nykyään erillisellä grafiikkasuorittimella. (Atkin [Viitattu 26.4.2013].)

Grafiikkasuorittimien kehittyessä ohjelmoijille annetaan enemmän ja enemmän vapauksia sen käytössä. Nykyään lähes kaikki modernit tietokonegrafiikkaohjelmat on tehty niin kutsuttua ”ohjelmoitavaa grafiikkaputkea” käyttäen. Tämä antaa ohjelmoijalle vapauden muuttaa esimerkiksi verteksivarjostinta, pikselivarjostinta ja geometriavarjostinta. Verteksivarjostin käsittelee kaikki grafiikkaputkelle lähetetyt geometrian kulmapisteet. Pikselivarjostinta kutsutaan jokaisen piirretyn geometrian pikselille, jotka voivat mahdollisesti näkyä näytöllä. Lähes kaikki nykyaikaiset jälkikäsittelyalgoritmit toimivat yksinomaan pikselivarjostimessa. Geometriavarjostin on yksi uusimmista varjostintyypeistä ja se voi lisätä tai poistaa grafiikkaputkelle lähetettyä geometriaa. (Lonroth & Unger 2009.)

3 Jälkikäsittelyalgoritmien teoria

3.1 Ympäristön varjostus (Ambient occlusion)

Ambient occlusion (AO) yrittää matkia valon vuorovaikutusta eri pintojen ja niiden materiaalien kanssa. Ambient occlusion ei ole tosielämän ilmiö, koska tosielämässä kappaleeseen tuleva valo on harvoin yhtä voimakasta joka, suunnasta ja valo heijastuu monista eri pinnoista ennen kuin saavuttaa kappaleen. Vaikka ambient occlusion on keinotekoinen, AO voi lisätä merkittävästi piirretyn kuvan realismia. AO luo kuvassa näkyville pinnoille ja kappaleille muotoa ja syvyyttä. Tämän takia AO-tekniikkaa käytetään usein laskennallisesti halpana vaihtoehtona kalliille globaalin valotuksen ratkaisuille. AO on yleisesti käytetty tekniikka, sekä animaatioelokuvissa että TV-ohjelmissa. Sitä käytetään kasvattamaan videon realistisuutta. Nykyään tietokonepelit pystyvät käyttämään reaaliaikaista AO-tekniikkaa, koska näytönohjaimet tulevat yhä tehokkaimmiksi. (Thai-Duong & Kok-Lim 2012.)



Kuva 1. Ympäristön varjostus eräässä huoneessa

Ambient occlusionin laskemiseen on olemassa erilaisia algoritmeja, jotka eroavat toisistaan suorituskyvyltään ja tarkkuudeltaan. Kaikista tarkimmat algoritmit käyttävät yleensä Monte Carlo -säteenjäljitystä, joka tekee niistä hitaita ja hyödyllisiä

vain esilasketun AO:n laskemiseen. Esilaskettua AO:ta voidaan käyttää animaatioissa tai reaaliaikaisten animaatioiden materiaalien tekstuureissa. Esilasketun AO:n käyttäminen reaaliaikaisissa animaatioissa pakottaa ympäristön geometrian ainakin osittain staattiseksi, koska jos geometria muuttuu, muuttuu myös AO. Reaaliaikaisesta AO:n laskennasta on vasta viimeaikoina tullut mahdollista ohjelmoitavan grafiikkaprosessorin nopeutumisen ansiosta. (Thai-Duong & Kok-Lim 2012.)

SSAO (Screen-Space Ambient Occlusion) on AO algoritmi, joka suoritetaan tavallisen geometrian piirron jälkeen piirretylle kuvalle. SSAO:n etuna muihin verrattuna on se, että se toimii kaikille mahdollisille ympäristöille ja on huomattavasti nopeampi toteuttaa ja integroida valmiiseen piirtojärjestelmään. Heikkoina puolina on SSAO:n epätarkkuus ja erilaiset laatuun vaikuttavat asiat. SSAO on kuitenkin niin nopea ja yksinkertainen, että siitä on tullut nopeasti suosittu ominaisuus 3D-peleissä ja muissa interaktiivisissa ohjelmissa, jotka suosivat nopeutta enemmän kuin tarkkuutta ja laatua. SSAO toteutetaan yleensä yhden ydinperiaatteen mukaan, mutta yksityiskohdat ja tulokset voivat erota merkittävästikin eri toteutuksien välillä. (Thai-Duong & Kok-Lim 2012.)

Nykyisillä SSAO algoritmeilla ei ole mitään ongelmaa laskea paikallisia AO-efektejä, kuten pienten rakojen tummumista. Laajemman AO:n laskennassa ne kuitenkin törmäävät nopeasti suorituskykyongelmiin, kun algoritmin pitäisi laskea suuria määriä näytteitä suurelta alueelta kuvasta. Jotkut algoritmit yrittävät kiertää tämän ongelman ottamalla satunnaisia näytteitä, mutta tämä tuottaa kohinaa tai sumuisuutta kun kohinaa yritetään vähentää. (Thai-Duong & Kok-Lim 2012.)

Tässä työssä käydään läpi SSAO:n laskennan perusperiaatteet, sekä MSSAO (Multi-Resolution Screen-Space Ambient Occlusion) algoritmi, joka laskee AO:n yhdistelemällä usean eri resoluution AO-tuloksia. Laskemalla AO monella eri resoluutiolla saadaan tarkat varjot sekä pienistä yksityiskohdista että suurista kappaleista. Tekniikka perustuu siihen, että kaukana olevien kappaleiden varjot ovat matalataajuisia, joten ne voidaan laskea karkeammalla resoluutiolla. Lähellä olevat varjot taas ovat korkeataajuisia ja ne pitää laskea tarkemmalla resoluutiolla. Tämä mahdollistaa sen, että pieniä näytemääriä voidaan käyttää joka resoluutiolla ja saavuttaa korkean suorituskyvyn ottamatta näytteitä satunnaisesti. Lisäksi säilyttämällä AO:n arvon resoluutioiden muuttuessa, pystymme kompensoimaan näky-

vyyden puutetta tiettyyn pisteeseen asti ja näin saadaan parempia tuloksia kuin muilla SSAO-algoritmeillä. (Thai-Duong & Kok-Lim 2012.)

3.1.1 Yleiskatsaus

SSAO:n laskeminen alkaa geometria-puskurin luonnilla. G-puskuriin tallennetaan piirretyn geometrian normaalivektorit ja syvyydet sillä resoluutiolla, jolla piirretty geometria halutaan käyttäjälle näyttää. Tämän jälkeen g-puskuri jaetaan moneen osaan monelle eri resoluutiolle. Sen jälkeen jokaisen g-puskurin jokaiselle pikselille lasketaan AO-arvo näytteistämällä sen pikselin viereisiä pikseleitä pienellä alueella. Lopullinen AO lasketaan suurimmalla g-puskurin resoluutiolla yhdistämällä pienempien resoluutioiden AO-arvoja. Edellä mainitun algoritmin toteutus tuottaa pikselöitymistä AO:ssa, koska alemman resoluution epätarkoilla AO arvoilla on yhtä suuri painoarvo kuin korkean resoluution tarkoilla AO-arvoilla. Parempi tekniikka onkin laskea AO-arvot monessa eri piirtokutsussa epätarkimmasta tarkimpaan yhdistämällä alemman resoluution arvo sillä hetkellä laskettavan pikselin AO-arvoon. Jokaisella piirtokutsulla siis lasketaan sen hetkisen resoluution AO ja yhdistetään se edellisen resoluution ylinäytteistettyyn AO-arvoon. Ylinäytteistykseen käytetään bilateraalista ylinäytteistystä, jolla vältetään ylinäytteistys isojen syvyys- ja normaalierojen välillä. Ennen kuin alemman resoluution AO-arvot ylinäytteistetään, ne ajetaan bilateraalisen alipäästösuodattimen läpi, jossa niitä myös sumennetaan pienellä alueella. (Thai-Duong & Kok-Lim 2012.)

3.1.2 Alinäytteistys

Alinäytteistysprosessi alkaa korkeimman resoluution g-puskurista ja tuottaa aina yhden asteen pienemmän resoluution g-puskurin jokaisella piirtokerralla. Jokaisella piirtokerralla jokaiselle pikselille lasketaan syvyys ja normaalivektori sitä vastaavan ylemmän resoluution pikseleiden syvyyksistä ja normaalivektoreista. Alinäytteistyskertojen määrä riippuu siitä, kuinka laajan AO:n haluaa. Käytännössä neljän tai viiden resoluution g-puskurin käyttö tuottaa kohtuullisen laajan AO:n ja mitätöi virheitä, joita alhaisen resoluution g-puskurit aiheuttavat. Useampien g-

puskureiden käyttö hyödyttää vain vähän, koska AO:sta tulee sitä pienitaajuisempaa, mitä pienempi resoluutio on. (Thai-Duong & Kok-Lim 2012.)

Yksinkertaisin tapa yhdistää korkean resoluution pikseleitä on ottaa niiden keskiarvo. Tässä algoritmissa päätettiin käyttää mediaania, koska mediaani on stabiilimpi kuin keskiarvo. Yleisesti tiedetään, että mediaani on vakaampi kuin keskiarvo, koska arvojoukon muuttuessa mediaani vaihtelee vähemmän kuin keskiarvo. Tämän takia mediaani säilyttää paremman ajallisen koherenssin kuin keskiarvo. Mediaanissa on kuitenkin yksi huono puoli, joka keskiarvoa käyttäessä ei tule esille. Mediaani ei säilytä näköavaruuden pisteiden välisiä suhteellisia etäisyyksiä. Tämä johtaa siihen, että kauempana olevat pisteet tulevat lähemmäksi alhaisimmilla resoluutioilla. Tämä taas aiheuttaa sen, että pisteet joiden ei pitäisi varjostaa mitään, ovatkin yhtäkkiä siirtyneet lähemmäksi ja luovat varjon AO:n vaikutusalueen ulkopuolelta. (Thai-Duong & Kok-Lim 2012.)

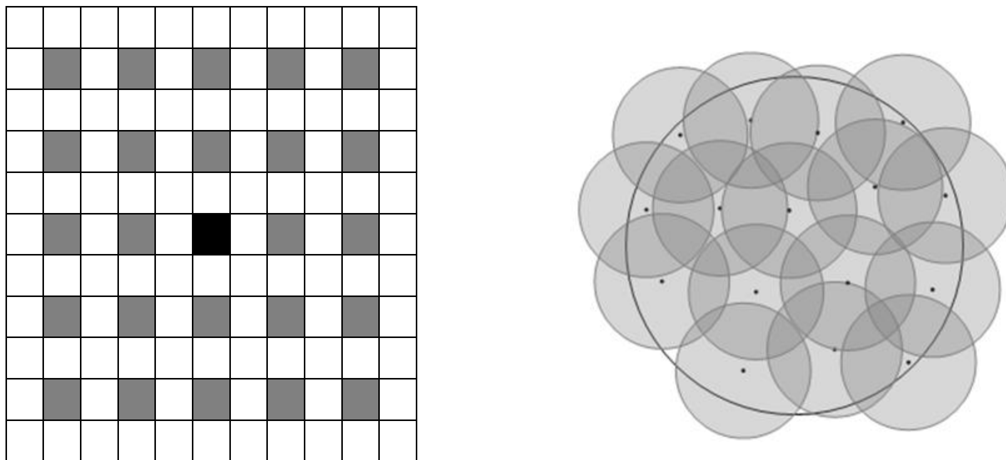
Mediaani lasketaan ottamalla neljä tarkemman resoluution syvyyttä ja laskemalla kahden keskimmäisen syvyyden keskiarvo. Jos neljä tarkemman resoluution syvyyttä ovat tietyn maksimietäisyyden päässä toisistaan, otetaankin toiseksi pienin syvyys mediaanin sijasta. Jos mediaania ei käytettäisi ollenkaan, vaan otettaisiin aina toiseksi pienin syvyys, aiheuttaisi animaatio ja kameran liike huomattavaa vilkkumista lopullisessa kuvassa. Normaalivektorit alinäytteistetään samalla tavalla kuin syvyydet. (Thai-Duong & Kok-Lim 2012.)

3.1.3 Näytteistys

AO:n laskemiseksi jokaisella resoluutiolla jokaisen pikselin *p* ympäröiviä pikseleitä näytteistetään pieneltä alueelta näyttökoordinaatistossa (screen space). Alue, jolta pikseleitä näytteistetään, määritellään seuraavasti: Ensin määritellään AO:n vaikutusalue d_{max} kamerakoordinaatistossa, joka kertoo kuinka kaukana olevat pisteet otetaan mukaan laskentaan. Seuraavaksi projisoidaan d_{max} näyttökoordinaatistoon, josta saadaan säde *r1*, pikseleissä. Tämä säde on karkea arvio siitä kuinka isolta alueelta näyttökoordinaatistossa otetaan näytteitä. Tämä toistetaan jokaiselle resoluutioille. Esimerkiksi toiseksi tarkin resoluutio käyttäisi sädettä *r2=r1/2* ja kolmanneksi suurin sädettä *r3=r1/4* ja niin edelleen. Perspektiiviprojektion aiheut-

taman vääristymän takia AO:n vaikutusalue riippuu pikseleiden syvyydestä. Tämä johtaa siihen, että AO:n vaikutusalue näyttökoordinaatistossa kasvaa pikseleiden lähestyessä kameraa. Näyttökoordinaatiston näytteistyssäteitä *r1*,*r2*,*r3* ja niin edelleen ei käytetä suoraan, vaan ne rajoitetaan maksimiiin r_{max}. Näytteistyssäteitä rajoitetaan, koska jos kamera on lähellä geometriaa, säteet kasvavat niin isoiksi, että suorituskyky alkaa kärsiä. (Thai-Duong & Kok-Lim 2012.) Tyypillinen r_{max} arvo tässä työssä oli viisi.

Näytteistysalueelta, jonka säde *r* määrittelee, ei näytteistetä kaikkia pikseleitä, vain joka toinen. Tämä vähentää näytteiden määrää tyypillisessä 11x11-alueessa 121:stä 36:een. Tällä noin 70%:n vähennyksellä saavutetaan huomattava parannus suorituskyvyssä ja lomitettu näytteistys toimii hyvin 3x3-sumennuksen kanssa, joka tehdään näytteistyksen jälkeen. Jos kuitenkin tarvitaan isoa näytteistysaluetta, voidaan käyttää esimerkiksi Poissonin levyä. (Thai-Duong & Kok-Lim 2012.)



Kuva 2. Lomitettu näytteistyskuvio ja poissonin levy (Thai-Duong & Kok-Lim 2012).

3.1.4 Näytteiden laskeminen

Jokaiselle pikselille *p* jokaisella resoluutiolla lasketaan kerätyistä näytteistä $AO_{lähi}$ arvo. $AO_{lähi}$ lasketaan kaavalla:

$$AO_{lähi}(\mathrm{p}) = \frac{1}{N}\sum_{i=1}^{N} \rho(\mathbf{p}, d_i)\,\overline{n \cdot \overline{q_i - p}} \tag{1}$$

jossa **p** laskettava piste,

N on näytteiden määrä,

n on pisteen normaali,

q_i on laskettava näyte,

$\rho(\mathbf{p}, d_i)$ on heikkenemisfunktio,

ja d_i on **p**n etäisyys näytteestä q_i kamerakoordinaatistossa.

Viiva pistetulon yläpuolella tarkoittaa, että sen arvo on rajattu välille nollasta yhteen. Funktio $\rho(\mathbf{p}, d_i)$ on heikkenemisfunktio, joka tasaisesti vähentää varjostajan vaikutusta sen etäisyyden kasvaessa. Heikkenemisfunktion pitää vähentyä tasaisesti yhdestä nollaan, kun etäisyys d_i kasvaa nollasta d_{max}. (Thai-Duong & Kok-Lim 2012.) Heikkenemisfunktio lasketaan seuraavasti:

$$\rho(\mathbf{p}, d_i) = 1 - \min\left(1, \left(\frac{d_i}{d_{max}}\right)^2\right) \quad (2)$$

jossa **p** on laskettava piste,

d_i on pisteen **p** etäisyys näytteestä q_i kamerakoordinaatistossa,

d_{max} on aiemmin valittu AOn vaikutusalueen säde.

Heikkenemisfunktio on yksinkertainen neliöllinen funktio, jonka arvo heikkenee sitä enemmän mitä kauempana näyte on lasketusta pisteestä.

Nyt kun $AO_{lähi}$ on laskettu, se pitää yhdistää $AO_{kaukana}$-arvoon, joka on ylinäytteistetty aikaisemmasta alhaisemman resoluution $AO_{lähi}$-arvosta. Arvon $AO_{kaukana}$ laskemiseen käytettään bilateraalista ylinäytteistystä, joka estää arvojen sumennuksen isojen syvyys- ja normaalierojen välillä. Ylinäytteistys yhdistää neljä alemman resoluution AO-arvoa, jotka ovat lähinnä pistettä *p*. Bilateraalisessa ylinäytteyksessä käytettävä painoitusarvo on syvyyksien ja normaalien erotuksen ja bilineaarisen painon yhdistelmä. Tarkalleen ottaen pikselille p_i, jolla on syvyys z_i sekä normaali $\boldsymbol{n}_i$, painot lasketaan seuraavilla kaavoilla

$$w_{i,n} = \left(\frac{\boldsymbol{n} \cdot \boldsymbol{n}_i + 1}{2}\right)^{t_n} \quad (3)$$

ja

$$w_{i,z} = \left(\frac{1}{1 + |z_i - z|}\right)^{t_z} \tag{4}$$

jossa $\boldsymbol{n}_i$ on pikselin p_i normaali,

z_i on pikselin p_i syvyys,

$\boldsymbol{n}$ on laskettavan pikselin normaali,

z on lasketavan pikselin syvyys.

Potenssit t_n ja t_z ovat toleranssiarvoja, jotka riippuvat ympäristön koosta. Jos ympäristö on pieni, lähellä olevat pikselit vastaavat lähellä olevia kamerakoordinaatiston ”pisteitä”. Tämä johtaa toleranssiarvojen kasvamiseen, koska isompia syvyyseroja täytyy ottaa huomioon vähemmän. Tarkoituksena on saavuttaa tasaisen pehmeä AO sekä säilyttää tarkat yksityiskohdat (Thai-Duong ja Kok-Lim 2012). Tässä työssä toleranssiarvot olivat $t_n = 8$ ja $t_z = 16$.

Kuva 3. Vasemmalla lähimmän naapurin ylinäytteistys, keskellä bilineaarinen ylinäytteistys ja oikealla bilateraalinen ylinäytteistys (Thai-Duong & Kok-Lim 2012).

Kuvasta 3 näkee selvästi bilateraalisen ylinäytteistyksen paremmuuden. Lähimmän naapurin ylinäytteistys on kulmikas ja bilineaarinen ylinäytteistys taas vuotaa varjoja yli geometriarajojen.

Ylinäytteytyksen jälkeen pitää arvot $AO_{lähi}$ ja $AO_{kaukana}$ yhdistää. Tähän asti molempia AO arvoja on käsitelty yksittäisinä arvoina, mutta itse asiassa $AO_{lähi}$ pitää sisällään kaksi arvoa ja $AO_{kaukana}$ kolme arvoa. Määritellään $AO_{lähi}^{[j]}$ tarkoittamaan $AO_{lähi}$ muuttujan j:ttä elementtiä. $AO_{lähi}^{[1]}$ on lasketun AO:n arvo ja $AO_{lähi}^{[2]}$ on laskentaan käytettyjen näytteiden määrä. $AO_{yhdistetty}$ arvot lasketaan seuraavasti

$$AO_{yhdistetty}^{1} = \max \frac{AO_{lähi}^{1}}{AO_{lähi}^{2}}, AO_{kaukana}^{1} \tag{5}$$

$$AO_{yhdistetty}^{2} = AO_{lähi}^{1} + AO_{kaukana}^{2} \tag{6}$$

$$AO_{yhdistetty}^{3} = AO_{lähi}^{2} + AO_{kaukana}^{3} \tag{7}$$

$AO_{kaukana}$ on siis ylinäytteistetty $AO_{yhdistetty}$. $AO_{kaukana}$ ensimmäinen elementti on sen hetkinen maksimi AO:n arvo, toinen elementti on normalisoimaton AO arvo ja kolmas on kaikkien ero resoluutioiden yhteenlaskettujen näytteiden määrä. Normalisoitumaton AO:n arvo ja näytteiden määrä tallennetaan, koska jos jollain resoluutiolla ei tule yhtään näytettä, niin AO:n keskiarvo pystytään silti laskemaan. Jos AO-arvo olisi normalisoitu, niin AO:n keskiarvo olisi mahdotonta laskea myöhemmillä resoluutioilla. Korkeimmalla ja viimeisellä resoluutiolla lasketut AO-arvot yhdistetään ja lasketaan lopullinen AO-arvo:

$$AO_{maksimi} = \max \frac{AO_{lähi}^{1}}{AO_{lähi}^{2}}, AO_{kaukana}^{1} \tag{8}$$

$$AO_{keskiarvo} = \frac{AO_{lähi}^{1} + AO_{kaukana}^{2}}{AO_{lähi}^{[2]} + AO_{kaukana}^{[3]}} \tag{9}$$

$$AO_{lopullinen} = (1 - AO_{keskiarvo})(1 - AO_{keskiarvo}) \tag{10}$$

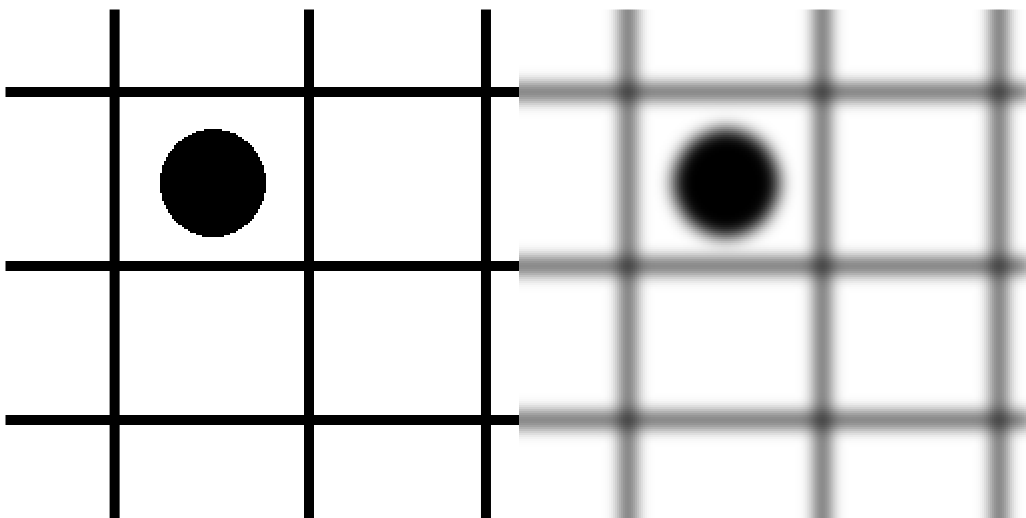
Maksimifunktion käyttö estää varjostajien moninkertaisen vaikutuksen tulokseen. Jos jollain resoluutiolla varjostajaa ei oteta huomioon, niin se voidaan ottaa huomioon vielä myöhemmillä resoluutioilla. Ainoastaan maksimiarvon käyttö ei ole riittävää, koska laskettavaa pistettä varjostetaan yleensä usealla resoluutiolla. Tämä johtaa siihen että lopullinen AO-arvo on usein isompi kuin $AO_{maksimi}$. Tämän välttämiseksi muokataan $AO_{maksimi}$ niin, että ehto $0 \leq AO_{maksimi} \leq AO_{lopullinen} \leq 1$ täyttyy. Lopullista AO arvoa voidaan käyttää suoraan pikseleiden valotukseen tai sitten perinteisen valotuksen apuna. (Thai-Duong ja Kok-Lim 2012.)

3.1.5 Bilateraalinen alipäästösuodatin

Lasketut AO-arvot ajettiin bilateraalisen alipäästösuodattimen ja sumennuksen läpi eri resoluutioiden välissä, koska käytettiin lomitettua näytteistystä. Käytetty bilateraalinen suodatin on samanlainen kuin ylinäytteytyksessä käytetty, mutta bilateraalisten painojen sijasta käytettiin Gaussin painoja. Toinen ero ylinäytteytykseen on kasvanut näytteistysalue, 2x2-pikselistä 3x3-pikseliin. (Thai-Duong & Kok-Lim 2012.)

3.2 Sumennukset (Blurs)

Monissa eri jälkikäsittelyalgoritmeissa käytetään sumennusta. Tehokkaiden sumennusalgoritmien avulla voidaan tehdä tehokkaita jälkikäsittelyalgoritmeja. Sumennusta käytetään esimerkiksi liikesumennuksessa, syväterävyysalueen laskennassa sekä ympäristön varjostuksessa. Liikesumennusta ja syväterävyysaluetta voidaan käyttää tekemään videopeleistä entistä realistisemman näköisiä. Liikesumennusta ja syväterävyysaluetta käytetäänkin yhä useammin uusissa videopeleissä ja animaatiossa. (Sherrod 2008, 466.)



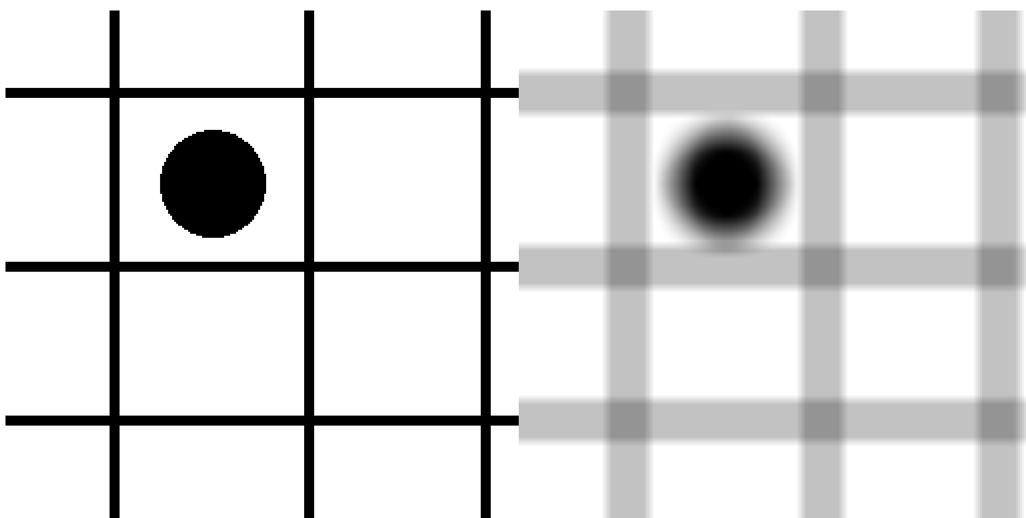
Kuva 4. Ristikko ja pallo sumennettu oikealla käyttäen Gaussin sumennusta

On olemassa erilaisia sumennusalgoritmeja, kuten Gaussin sumennus, sädesumennus ja keskiarvosumennus. Kaikki sumennusalgoritmit toteutetaan yleensä niin että otetaan tietty määrä pikseleitä ja lasketaan niiden keskiarvo jossakin suhteessa. Esimerkiksi jos kuvasta 5 haluttaisiin sumentaa pikseli 0,0 , laskettaisiin kaikkien pikseleiden arvot yhteen 3x3-alueelta ja jaettaisiin tulos yhdeksällä. Lopputuloksena pikseli 0,0 on sumennettu ottamalla keskiarvo viereisistä pikseleistä. Kun tämä tehdään kaikille kuvan pikseleille, kuva on sumennettu. Tätä kutsutaan laatikkosumennukseksi ja tällainen sumennus on hyvin yksinkertainen toteuttaa pikselivarjostimessa. (Sherrod 2008, 466.)

-1,1	0,1	1,1
-1,0	0,0	1,0
-1,-1	0,-1	1,-1

Kuva 5. 3x3-alue pikseleitä

3.2.1 Laatikkosumennus (box blur)

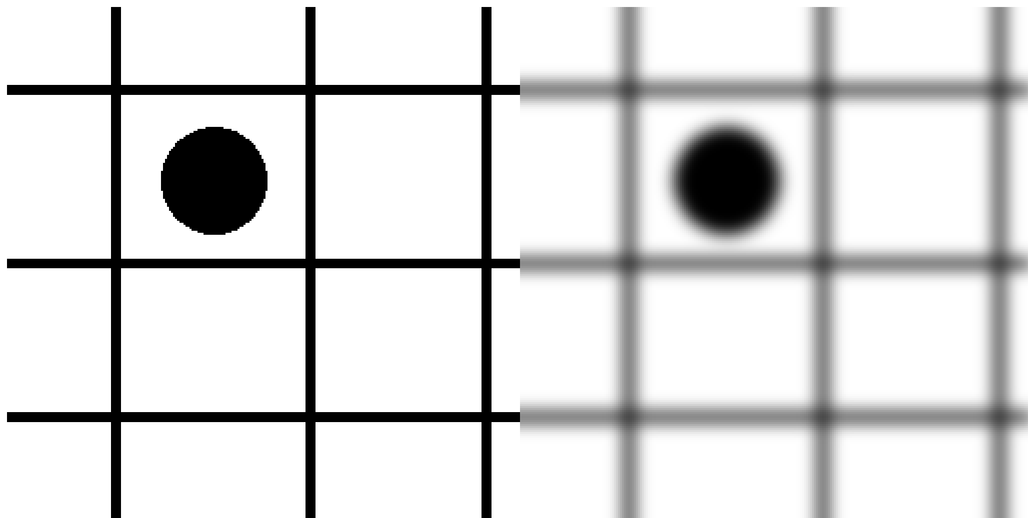


Kuva 6. Laatikkosumennus

Laatikkosumennuksessa kaikilla pikseleillä on yhtä suuri paino sumennuksessa. Laatikkosumennuksen etuna on nopea suorituskyky ja algoritmin helppous. Huonona puolena on kuitenkin heikko laatu verrattuna Gaussin sumennukseen. (Huxtable [Viitattu 26.4.2013].)

3.2.2 Gaussin sumennus (Gaussian blur)

Gaussin sumennusta käytetään laajalti erilaisissa kuvankäsittelyalgoritmeissa ja tietokonegrafiikassa luomaan fotorealistisen näköinen sumennus. Gaussin sumennusta käytetään sekä erilliskäsittelynä valokuvissa ja videoissa että reaaliaikaisena jälkikäsittelynä pelimoottoreissa. (Rákos 9.2010.)



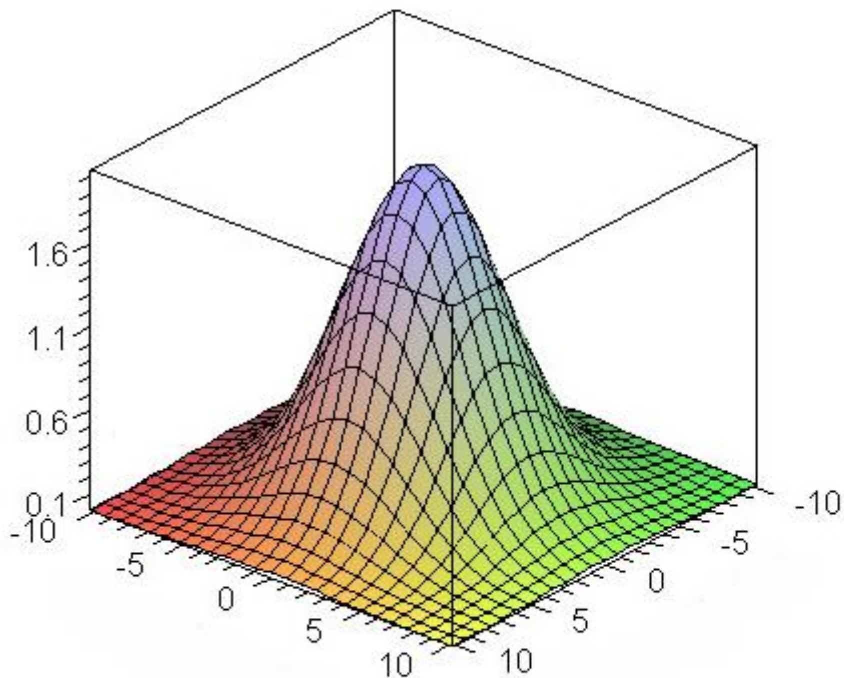
Kuva 7. Gaussin sumennus

Gaussin sumennuksessa pikseliä ympäröivät pikselit lasketaan yhteen painottamalla pikseleitä Gaussin kaavan perusteella:

$$G\ x,y\ = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{11}$$

$$G\ x\ = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{12}$$

Kaava 11 on yksiulotteinen gaussin kaava ja kaava 12 on kaksiulotteinen. Gaussin kaavan käyttäminen on suoraviivaisin ja geneerisin tapa saada tarvittavat painot sumennukseen. (Rákos 9.2010.)



Kuva 8. Kaksiulotteisen Gaussin kaavan graafinen esitys (Rákos 9.2010).

Yksinkertaisempi tapa painojen laskemiseen on Pascalin kolmion binomikertoimien käyttö (Rákos 9.2010).

Index N	Coefficients	Sum = 2^N
0	1	1
1	1 1	2
2	1 2 1	4
3	1 3 3 1	8
4	1 4 6 4 1	16
5	1 5 10 10 5 1	32
6	1 6 15 20 15 6 1	64
7	1 7 21 35 35 21 7 1	128
8	1 8 28 56 70 56 28 8 1	256
9	1 9 36 84 126 126 84 36 9 1	512
10	1 10 45 120 210 252 210 120 45 10 1	1024
11	1 11 55 165 330 462 462 330 165 55 11 1	2048
12	1 12 66 220 495 792 924 792 495 220 66 12 1	4096

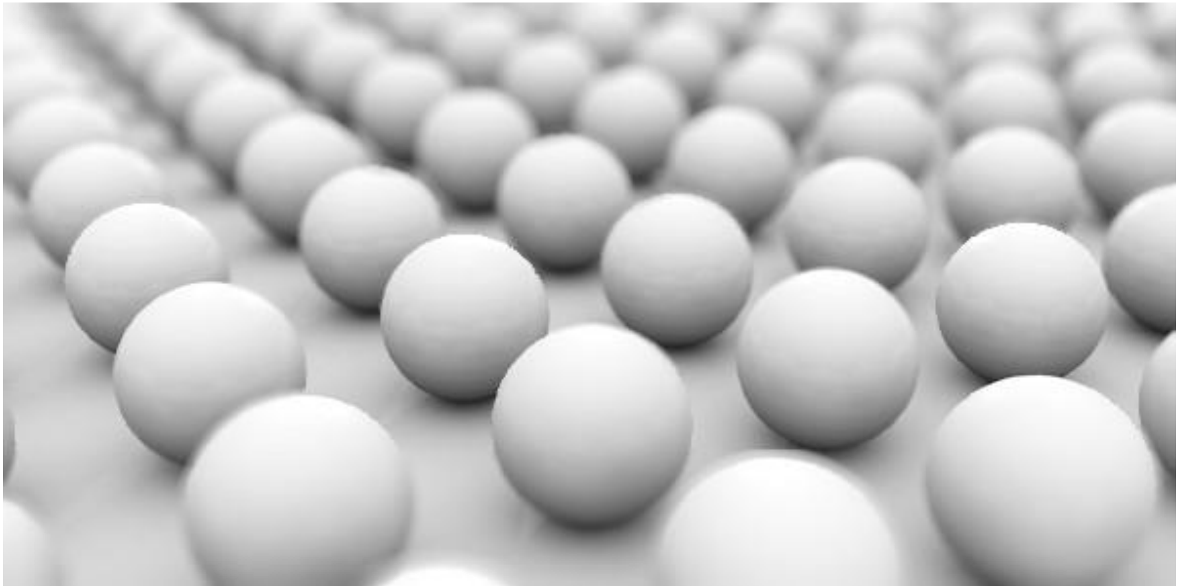
Kuva 9. Pascalin kolmio, binomikertoimia voidaan käyttää Gaussin painoina (Rákos 9.2010).

Kuten muissa sumennuksissa, Gaussin sumennus tehdään yleensä neliön alueelle. Kun sumennuksen laajuus on esimerkiksi 9x9-alue pikselin ympärillä, täytyy varjostimen lukea 81 kertaa tekstuurista. Kaksiulotteisen Gaussin funktion voi kuitenkin laskea kahden yksiulotteisen Gaussin funktion tulona. Gaussin funktion tulo 2σ-alueella on sama kuin kahden Gaussin funktion tulo σ-alueella. Tämä tarkoittaa sitä, että sumennus voidaan jakaa kahteen osaa. Ensimmäisellä kerralla sumennetaan horisontaalisesti ja luetaan tekstuuria yhdeksän kertaa. Toisella kerralla sumennetaan vertikaalisesti ja luetaan toiset yhdeksän kertaa. Näin ollen jos sumennuksen jakaa kahteen eri osaan, voidaan tekstuurikutsujen määrä vähentää 81:stä 18:aan. Tämä kahden osan tekniikka tuottaa tarkan sumennuksen sekä parantaa huomattavasti suorituskykyä. (Rákos 9.2010.)

3.3 Syväterävyysalue (Depth of Field)

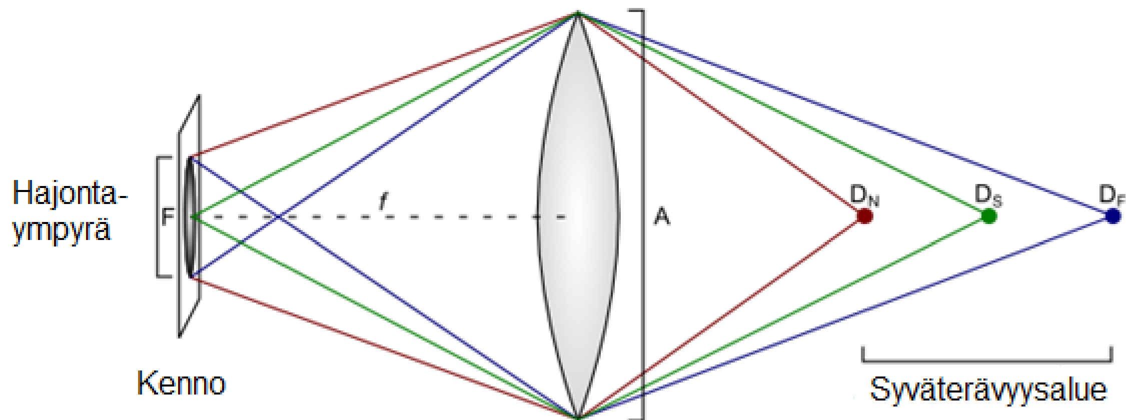
Syväterävyysalue on alue, missä kuvattava kohde näkyy tarkasti. Kun kappale liikkuu kauemmas tarkasta alueesta, se alkaa sumentua. Sumentuminen johtuu valon ja linssien fysikaalisista ominaisuuksista. Tietokonegrafiikassa ei tarvitse ottaa huomioon tällaisiä fysikaalisia ominaisuuksia, sillä tietokone piirtää kaiken äärimmäisen tarkasti käyttäen matematiikkaa. Joissain tapauksissa, kuten teknisessä piirtämisessä, täydellinen tarkkuus on toivottua. Luonnollisia asioita piirtäes-

sä taas epätarkkuus lisää kuvan aitoutta ja saa kuvan näyttämään elävämmältä. Syväterävyysaluetta käytetään paljon valokuvauksessa, koska ihmissilmä huomaa tarkasti kuvatut asiat ennen sumennettuja. Videopeleissä syväterävyysaluetta käytetään juuri samasta syystä. Asettamalla vain pienen osan piirretystä kuvasta teräväksi, voidaan pelaajan huomio siirtää kohti ohjaajan haluamaa tapahtumaa. Syväterävyysaluetta pitää kuitenkin käyttää varovasti, sillä liiallinen käyttö pilaa helposti pelikokemuksen. (Meyer 10.7.2012.)



Kuva 10. Syväterävyysalue on tarkka alue kuvan puolivälissä (Meyer 10.7.2012).

Tietokonegrafiikassa ei ole linssejä perinteisessä mielessä. Käytännössä kaikki piirto tapahtuu tarkkojen matemaattisten mallien mukaan. Piirrettävä 3D-kappale projisoidaan 2D-kuvaksi ja kuva näytetään käyttäjälle. Tietokonegrafiikassa kameran linssi on äärettömän pieni piste, jolla on ääretön syväterävyysalue. Tosielämässä kameran linssi on isompi kuin piste ja sen täytyy pystyä tarkentamaan tuleva valo tietylle kohtaa kameran kennoa. Mitä isompi linssi ja apertuuri, sitä isompi polttoväli. Mitä isompi polttoväli, sitä pienempi syväterävyysalue. (Meyer 10.7.2012.)

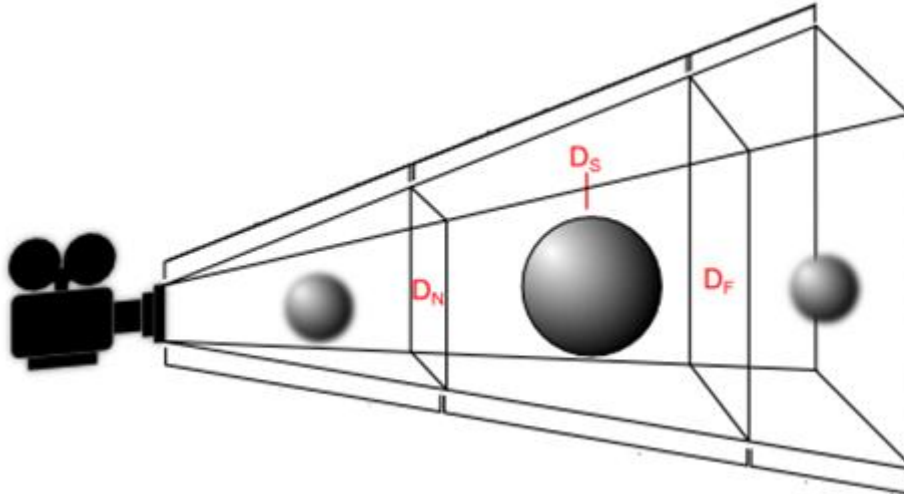


Kuva 11. Linssin toiminta (Meyer 10.7.2012).

Kuvassa 11 D_N on lähin etäisyys, missä kappaleet näkyvät terävinä,
D_S on etäisyys, missä kappaleet näkyvät kaikista tarkimmin,
D_F on pisin etäisyys missä kappaleet näkyvät terävinä,
F on polttopiste johon linssi kohdistaa valon,
f on polttoväli eli polttopisteen ja linssin etäisyys,
A on linssi halkaisija, halkaisija ilmoitetaan yleensä aukkosuhteena eli f-lukuna. (Meyer 2012.)

Kuvassa 12 linssi kohdistaa oikealta tulevan valon kohti vasemmalla olevaa kennoa. Kuvattava kappale on terävä kun valonsäteet leikkaavat kennossa. Vihreä viiva pisteestä D_S esittää juuri tätä. Kappaleet pisteiden D_N ja D_F välissä näyttävät silti tarkoilta, vaikka niistä tulevat säteet eivät leikkaakaan kennossa. Säteet kuitenkin osuvat hajontaympyrään ja koska ihmissilmä ei ole täydellinen, ne näyttävät tarkoilta. (Meyer 2012.)

Syväterävyysalue voidaan toteuttaa kahdella tavalla. Toinen on hyvin yksinkertainen ja toinen vaatii vähän matematiikkaa. Yksinkertaisin tapa on antaa varjostimelle D_N ja D_F syvyydet ja sumentaa tämän välin ulkopuolella olevat pisteet. (Meyer 2012.)



Kuva 12. Yksinkertainen syväterävyysalueen toteutus (Meyer 10.7.2012).

Toinen tekniikka kuvaa huomattavasti paremmin tosimaailmaa. Siinä käytetään sumennussäteen laskentaan oikeita kameran ominaisuuksia, kuten polttoväliä ja aukkosuhdetta. (Meyer 2012.) Tämä vaatii hieman enemmän laskentaa, mutta toteutuksen säätäminen vastaa paremmin tosielämän kameran säätöä. Sumennussäde lasketaan seuraavan kaavan avulla:

$$b = \frac{fm_s}{N}\frac{x_d}{D_S \pm x_d} \qquad (13)$$

jossa f on polttoväli,

m_s on kappaleen suurennus $m_s = \frac{f}{D_S - f}$,

N on aukkosuhde, eli f-luku,

x_d on tarkimman pisteen etäisyys laskettavasta pisteestä $x_d = |D - D_S|$,

D_s on etäisyys tarkimpaan pisteeseen,

D on laskettavan pisteen etäisyys kamerasta,

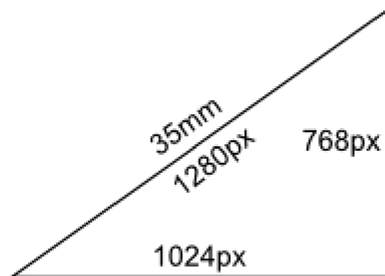
b on laskettu sumennussäde. (Meyer 2012.)

Etualalla olevat kappaleet lasketaan käyttämällä $D_S - x_d$ ja taustalla olevat kappaleet $D_S + x_d$. Kaavasta saatu sumennussäde on metreissä ja se pitää muuntaa pikseleihin. (Meyer 2012.) Sumennussäteen muuntosuhde saadaan jakamalla hypotenuusa linssin halkaisijalla:

$$b = \frac{\overline{a^2 + b^2}}{A} \qquad (14)$$

jossa a ja b ovat kuvan resoluutio leveys ja korkeussuunnassa
A on linssin halkaisija.

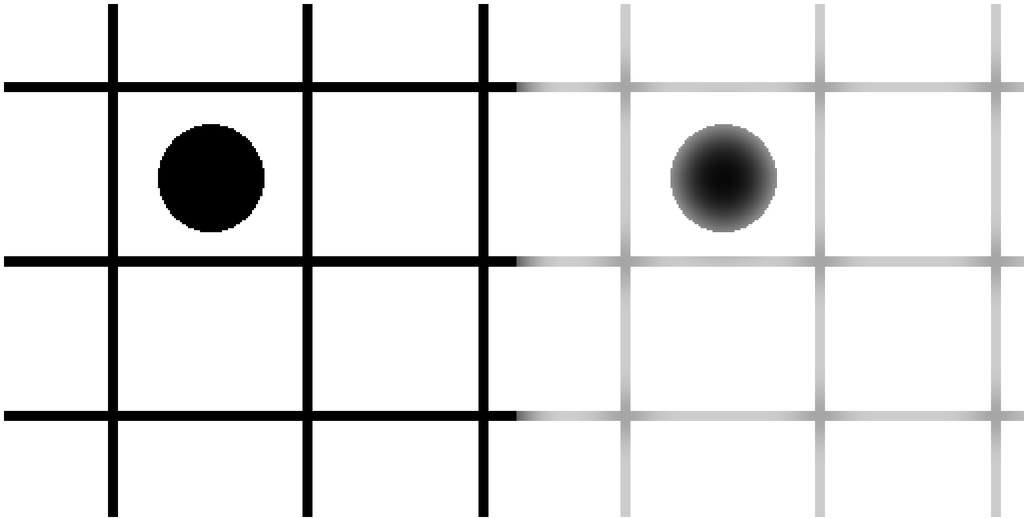
Esimerkiksi sumennussäde kameralla, jossa on 35 mm:n linssi ja resoluutio 1024*768-pikseliä, on muunnosluku 1280 pikseliä/35 mm = 36,57 pikseliä/mm.



Kuva 13. Esimerkkikuva yksikkömuunnoksesta (Meyer 10.7.2012).

3.4 Hehku (Bloom)

Kappaleiden hohtaminen johtuu yleensä siitä, että ne ovat huomattavasti kirkkaampia kuin muu ympäristö. Esimerkiksi aurinko hohtaa niin kirkkaasti taivaalla, että sen tarkkoja ääriviivoja on todella hankala erottaa. Taivaalla ja ympäristössä näkyvät hohdot ja haloilmiöt johtuvan valon siroutumisesta ilmakehässä ja silmässä. Tietokonegrafiikassa hehku on visuaalinen efekti, jossa kuvan kirkkaat kohdat vuotavat kirkkautta viereisiin pikseleihin. Tämä efekti on helppo laskea muutamalla pikselivarjostimella. Hehkun avulla saadaan tietokonegrafiikkaan tosielämää vastaavia kirkkaita kappaleita, jotka luovat valoa ympärilleen. (James & O’Rorke 26.52004.)



Kuva 14. Oikealla kuvan kirkkaat valkoiset osat ovat vuotaneet mustien viivojen ja ympyrän päälle

Hehkun toteutus on hyvin lähellä syväterävyysalueen toteutusta. Ensin lasketaan niin sanottu hehkutekstuuri. Hehkutekstuuri on alhaisen resoluution tekstuuri, jossa on kaikki ympäristön kappaleet, joita halutaan korostaa hehkulla. Hehkutekstuuri voidaan luoda erikseen piirtämällä siihen kaikki kirkkaat kappaleet tavallisen piirron jälkeen tai suodattamalla g-puskurin väritekstuurista. Sen jälkeen, kun hehkutekstuurissa on kaikki tarvittavat kappaleet, se pitää sumentaa. Sumennuksen voi tehdä melkein millä tahansa sumennustekniikalla tahansa. Kun hehkutekstuuri on sumennettu, yhdistetään se alkuperäisen g-puskurin väritekstuuriin. (Kalogirou 20.5.2006.)

G-puskurin värisuodatus voidaan tehdä seuraavalla kaavalla:

$$F\ \mathbf{p} = \begin{matrix} \mathbf{c}, & l < t \\ 0, & \end{matrix} \qquad (15)$$

missä $\mathbf{p}$ on laskettava pikseli,

$\mathbf{c}$ on väritekstuurin väri,

l on värin $\mathbf{c}$ luminanssi

ja t on raja-arvo.

Kaava on eräänlainen päästösuodatin, joka suodattaa pois värit, joiden luminanssi on alle raja-arvon.

Hehkutekstuurin ja väritekstuurin lopullinen yhdistäminen tapahtuu seuraavan kaavan avulla:

$$F\ \boldsymbol{p}\ = \boldsymbol{src} + \boldsymbol{dst} \tag{16}$$

missä $\boldsymbol{p}$ on laskettava pikseli,
$\boldsymbol{src}$ on hehkutekstuurista otettu väri
ja $\boldsymbol{dst}$ on G-puskurin väritekstuuri.

Tämä kaava on yksinkertaisin tapa yhdistää hehkutekstuuri alkuperäiseen väritekstuuriin. Huonona puolena on se, että $\boldsymbol{src} + \boldsymbol{dst}$ on helposti yli yhden ja yli yhden arvot eivät sovi tavallisiin tekstuureihin. Tämän voi korjata sävykorjauksella tai käyttämällä seuraavaa kaavaa hehkutekstuurin ja väritekstuurin yhdistämiseen

$$F\ \boldsymbol{p}\ =\ \boldsymbol{src} + \boldsymbol{dst}\ - (\boldsymbol{src} * \boldsymbol{dst}) \tag{17}$$

4 Jälkikäsittelyalgoritmien toteutus

Kaikki jälkikäsittelyalgoritmit vaativat g-puskurin, joka luodaan ennen algoritmin ajamista. Kaikki jälkikäsittelyalgoritmit tarvitsevat myös omia piirtotekstuureita, joihin niiden tulokset tallennetaan. Ympäristönvarjostus käyttää huomattavasti enemmän tekstuureita kuin muut algoritmit, resoluutioiden määrästä riippuen jopa toista kymmentä.

4.1 Ympäristön varjostus (Ambient Occlusion)

Ympäristön varjostuksen toteutukseen tarvitaan g-puskuri, joka pitää sisällään normaalivektorit ja pikseleiden kamera-avaruuden koordinaatit. Normaalivektorit mahtuvat hyvin tavallisen RGB-tekstuurin sisään. Pitää ottaa huomioon se, että tavalliset tekstuurit on rajattu nollan ja yhden välille, joten normaalivektorit jotka ovat -1 ja yhden välillä pitää viedä oikealle välille. Tämä onnistuu helposti seuraavalla kaavalla

$$F(\boldsymbol{n}) = \boldsymbol{n} * 0.5 + 0.5 \tag{18}$$

jossa $\boldsymbol{n}$ on rajattava normaali.

ja purku tekstuurista onnistuu seuraavan kaavan avulla

$$F(\boldsymbol{n}) = \boldsymbol{n} * 2 - 1 \tag{19}$$

Pikselikoordinaattien tallennuksen voi tehdä vähintään kahdella tavalla. Ensimmäinen vaihtoehto on normalisoimattomat tekstuurit. Esimerkiksi RGBA16F tekstuuriformaattiin voi luvut tallentaa sellaisenaan ilman että pitää huolehtia rajaamisesti nollan ja yhden välille. Toinen vaihtoehto on syvyyspuskurin käyttö ja näyttökoordinaattien projisointi takaisin kamerakoordinaatistoon. Tässä työssä käytettiin syvyyspuskuria, koska testien jälkeen se todettiin nopeammaksi kuin RGBA*F-tekstuurit. Näyttökoordinaatiston pisteen projisointi takaisin kamerakoordinaatistoon onnistuu seuraavan koodin avulla:

```
vec3 reconstruct_position(float depth, vec2 tex_coord)
{
            vec4 position = vec4(tex_coord, depth, 1);
            position.xyz = position.xyz*2-1;
            position = inverse_MVP*position;
            position.xyz /= position.w;
            return position.xyz;
}
```

4.1.1 Alinäytteistys

Alinäytteytyksessä siis otetaan tarkan resoluution g-puskuri ja pienennetään se niin monta kertaa kuin on tarve. Tässä työssä alkuperäinen g-puskuri pienennettiin puoleen, neljäsosaan, kahdeksasosaan ja kuudenteentoistaosaan.

```
float maxZ = max(max(pos[0].z, pos[1].z), max(pos[2].z, pos[3].z));
float minZ = min(min(pos[0].z, pos[1].z), min(pos[2].z, pos[3].z));

int minPos, maxPos;

for (int i = 0; i < 4; ++i)
{
    if (pos[i].z == minZ)
      minPos = i;
    if (pos[i].z == maxZ)
      maxPos = i;
}

float d = distance(pos[minPos].xyz, pos[maxPos].xyz);

ivec2 median = ivec2(0, 0);
int index = 0;

for(int i = 0; i < 4 && index < 2; ++i)
    if (i != minPos && i != maxPos)
        median[index++] = i;

if(d < 1.0){
    Pos = (pos[median.x] + pos[median.y]) / 2.0;
    Norm = (norm[median.x] + norm[median.y]) / 2.0;
}
else{
    Pos = pos[median.x];
    Norm = norm[median.x];
}
```

4.1.2 Näytteiden laskeminen

Tämä koodi toteuttaa kappaleessa 3.1.4 esitellyn kaavan, jolla käydään läpi alue pikseleitä ja lasketaan niiden yhteinen AO-arvo.

```
void computeOcclusion(vec3 position, vec4 normal, vec2 xy)
{
    float depth = texture(depth_texture, xy).r;
    vec3 samplePos = reconstruct_position(depth, xy);
    float d = distance(position.xyz, samplePos.xyz);
    float attenuation = 1.0 - (min(1, d*d/(dMax*dMax)));

    vec3 v = normalize(samplePos.xyz-position.xyz);
    float angle = max(dot(normal.xyz, v),0.0);
    occlusion += angle * attenuation;
    sampleCount +=1.0;
}
```

Huomionarvoinen kohta on main-funktion toistolauseet. Alla olevassa koodissa toistolauseet käyvät muistia läpi lineaarisesti. Alkuperäisessä koodissa, joka tuli SSAO-paperin mukana, muistia käytiin läpi todella epälineaarisesti ja se aiheutti huomattavia menetyksiä suorituskyvyssä. Toteutuksessa käytettiin GL_RGB-tekstuuria AO:n välitulosten tallentamiseen. GL_RGB-tekstuurissa jokaiselle värikomponentille on varattu kahdeksan bittiä muistia ja siihen kirjoitetut arvot leikataan välille nollasta yhteen. Algoritmin piti kuljettaa lukuja, jotka eivät olleet välillä nollasta yhteen, tämän takia occlusion- ja sampleCount-arvot vietiin manuaalisesti sille välille, jotta niitä ei leikattaisi.

Kahdeksan bittiä per värikomponentti aiheuttaa myös toisen rajoituksen algoritmille. Jos tekstuuriin tallennettu sampleCount arvo ylittää 255, se pyörähtää takaisin nollaan. Tämä aiheuttaa huomattavia ongelma laskennassa. Esimerkiksi viittä resoluutiota käyttämällä jää jokaiselle resoluutiolle 63 tekstuurilukua per pikseli lukuun ottamatta tarkinta resoluutiota. Nämä haitat pystyttäisiin välttämään käyttämällä ei-normalisoituja tekstuureita, esimerkiksi GL_RGB-formaattiin verrattuna neljä kertaa isompaa GL_RGB32F-formaattia. Tällaisen tekstuurin käyttö aiheuttaisi kuitenkin isoja menetyksiä suorituskyvyssä ja kasvattaisi muistin käyttöä huomattavasti.

```
    for(float y=-rangeMax;y <= rangeMax; y+=2.0){
        for(float x=-rangeMax; x <= rangeMax; x+=2.0){
            vec2 sub_tex_coord = tex_coord + vec2(width_step*x, height_step*y);
            computeOcclusion(position, normal, sub_tex_coord);
        }
```

```
    }
    ambient_occlusion = vec4(occlusion / sampleCount,
                             occlusion/255.0,
                             sampleCount/255.0, 0.0);
```

Viimeisessä AO laskennassa yhdistettiin viimeisen resoluution ja aikasempien resoluutioiden AO-tulokset.

```
vec3 upsample = Upsample(position, normal);
float aoMax = max(upsample.x, occlusion / sampleCount);
float aoAverage = (upsample.y + occlusion) / (upsample.z + sampleCount);
float final_ao = (1.0 - aoMax) * (1.0 - aoAverage);
ambient_occlusion = vec4(final_ao);
```

4.1.3 Bilateraalinen ylinäytteistys

Bilateraalinen ylinäytteistys suoritettiin aina toiseksi pienimmästä resoluutiosta lähtien.

```
float normal_weight[4];
for(int i=0; i < 4; ++i)
{
    normal_weight[i] = (dot(low_res_normals[i].xyz, normal.xyz) + 1) / 2;
    normal_weight[i] = pow(normal_weight[i], 8.0);
}

float depth_weight[4];
for(int i=0; i < 4; ++i)
{
    depth_weight[i] = 1.0 / (1.0 + abs(position.z - low_res_depths[i]));
    depth_weight[i] = pow(depth_weight[i], 16.0);
}

float total_weight = 0.0;
vec3 AO = vec3(0);
for(int i=0; i < 4; ++i)
{
    float weight = normal_weight[i] * depth_weight[i] * (9.0 / 16.0) /
             (abs((tex_coord.x - low_res_text_coord[i].x * 2.0) * (tex_coord.y -
             low_res_text_coord[i].y * 2.0)) * 4.0);
    total_weight += weight;
    AO += low_res_AOs[i] * weight;
}
return AO / total_weight;
```

4.1.4 Bilateraalinen alipäästösuodatin ja sumennus

Tämä koodi toteuttaa kappaleessa 3.1.5 esitellyn bilateraalisen alipäästösuodattimen ja sumennuksen. Tämä koodi ajetaan siis aina jokaisen resoluution välissä tuoreille AO-arvoille. Arvoja sumennetaan lomittaisen näytteytyksen takia ja alipäästösuodatuksella saadaan hieman puhtaampi tulos.

```
for (float y = -1.0; y <= 1.0; y += 1.0){
    for (float x = -1.0; x <= 1.0; x += 1.0){
        vec2 xy = vec2(x, y) * inverse_texture_size;
        vec3 ao = texture(ao_texture, tex_coord+xy).xyz;
        vec3 sub_normal = texture(normal_texture, tex_coord+xy).xyz*2-1;
        float sub_depth = texture(depth_texture, tex_coord+xy).r;

        float normal_weight = (dot(sub_normal.xyz, normal.xyz) + 1.2) / 2.2;
        normal_weight = pow(normal_weight, 8.0);

        float depth_weight = 1.0 / (1.0 + abs(depth - sub_depth) * 0.2);
        depth_weight = pow(depth_weight, 16.0);

        float gaussian_weight = 1.0 / ((abs(y) + 1.0) * (abs(x) + 1.0));

        weight += normal_weight * depth_weight * gaussian_weight;
        total_ao += ao * normal_weight * depth_weight * gaussian_weight;
    }
}

ambient_occlusion = vec4(total_ao / weight, 1);
```

4.2 Sumennukset (Blurs)

Varjostimessa toteutetuissa sumennuksissa on eroja riippuen GLSL-versiosta. GLSL 1.40-versiosta lähtien ovat ohjelmoijat voineet käyttää texture kutsujen sijasta textureOffset-kutsua. Se on hieman helpompi käyttää, koska poikkeaman luettavaan tekseliin saa erillisenä parametrina, eikä tarvitse säätää tekstuurikoordinaatin ja tekstuurin koon kanssa. textureOffsetin käytössä on kuitenkin rajana offsetin koko, joka on toteutusriippuvainen. (Kessenich, Baldwin & Ros 2012.)

Funktiokutsujen eron näkee selvästi seuraavasta koodiesimerkistä, jossa molemmat kutsut tekevät saman asian.

```
value = textureOffset(color_texture, tex_coord, ivec2(-2,0));

value = texture(color_texture, tex_coord + vec2(-2*inverse_texture_width,0);
```

Kolmantena vaihtoehtona GLSL 4.00-versiosta lähtien ovat olleet joissakin tapauksissa textureGather ja textureGatherOffset-funktio. Nämä funktiot lukevat neljän tekselin arvon ja palauttavat niistä jokaisesta yhden komponentin, riippuen aikaisemmin asetetuista asetukista. (Kessenich, Baldwin & Ros 3.8.2012.) Näillä funktioilla ei tietenkään ole erityistä hyötyä textureOffset ja texture funktiohin verrattuna sumennettaessa RGBA-kuvaa, mutta jos tarkoituksena on sumentaa yksikanavaista kuvaa tai vain kuvan yhtä kanavaa, ovat textureGather-funktiot erittäin hyviä tähän.

4.2.1 Laatikkosumennus (Box blur)

Seuraavassa koodissa tehdään laatikkosumennus vaakasuuntaan käyttäen textureOffset kutsuja.

```
vec4 total=vec4(0);
total += textureOffset(color_texture, tex_coord, ivec2(-2,0));
total += textureOffset(color_texture, tex_coord, ivec2(-1,0));
total += textureOffset(color_texture, tex_coord, ivec2(0,0));
total += textureOffset(color_texture, tex_coord, ivec2(1,0));
total += textureOffset(color_texture, tex_coord, ivec2(2,0));

color_out = total/5.0;
```

4.2.2 Gaussin sumennus (Gaussian blur)

Gaussin sumennus on jaettu kahteen osaan, vaaka- ja pystysuuntaiseen sumennukseen. Tällä saadaan noin 70 % vähennys tekstuurilukuihin. Käytetyt Gaussin painot on laskettu Pascalin binomikerroinkolmiosta riviltä neljä. Alla oleva Gaussin sumennuksen osa on vaakasuuntaan sumennus, sen näkee tekstuuriluvussa tekstuurikoordinaatin muuttuessa vaakasuunnassa.

```
sum+=texture(color_texture, vec2(tex_coord.x -2.0*blurSize, tex_coord.y))*(1/16.0);
sum+=texture(color_texture, vec2(tex_coord.x -1.0*blurSize, tex_coord.y))*(4/16.0);
sum+=texture(color_texture, vec2(tex_coord.x              , tex_coord.y))*(6/16.0);
sum+=texture(color_texture, vec2(tex_coord.x +1.0*blurSize, tex_coord.y))*(4/16.0);
sum+=texture(color_texture, vec2(tex_coord.x +2.0*blurSize, tex_coord.y))*(1/16.0);
```

Alapuolella on sama Gaussin sumennus samoilla painoilla, mutta sumennus on tehty pystysuunnassa.

```
sum+=texture(color_texture, vec2(tex_coord.x, tex_coord.y -2.0*blurSize))*(1/16.0);
sum+=texture(color_texture, vec2(tex_coord.x, tex_coord.y -1.0*blurSize))*(4/16.0);
sum+=texture(color_texture, vec2(tex_coord.x, tex_coord.y              ))*(6/16.0);
sum+=texture(color_texture, vec2(tex_coord.x, tex_coord.y +1.0*blurSize))*(4/16.0);
sum+=texture(color_texture, vec2(tex_coord.x, tex_coord.y +2.0*blurSize))*(1/16.0);
```

Sumennus on tässä tehty viiden pikselin alueelta, kaksi pikseliä sumennettavan pikselin ympäriltä sekä pysty- että vaakasuuntaan.

4.3 Syväterävyysalue (Depth of field)

Käytännössä syväterävyysalue toteutetaan kahdessa tai kolmessa piirtokutsussa. Kuva, jolle syväterävyysalue lasketaan, pienennetään ensin esimerkiksi neljäsosaan tai kahdeksasosaan. Tämän jälkeen pienennetty kuva sumennetaan käyttämällä esimerkiksi Gaussin sumennusta. Lopuksi sumennettu kuva ja alkuperäinen terävä kuva yhdistetään sumennussädettä käyttäen. Pikselit, jotka ovat täysin syväterävyysalueen ulkopuolella, otetaan sumennetusta kuvasta ja pikselit, jotka ovat syväterävyysalueen sisällä, otetaan tarkasta kuvasta. Loput pikselit, jotka on siinä välillä, interpoloidaan lineaarisesti sumennetun ja tarkan kuvan välillä. Mitä lähempänä piste on sumennussädettä sitä isommaksi sumennetun kuvan osuus kasvaa. (Meyer 10.7.2012.)

Syväterävyysalueessa laskettiin ensin laatikkosumennus pienennetylle g-puskurille. Eroa tavalliseen laatikkosumennukseen on se että sumennus tehtiin vain tietylle linssiyhtälön määräämälle alueelle. GetBlurDiameter toteuttaa kaavan 13 sivulta 28. Blur_coefficient on pikselistä riippumaton vakio, joten se laskettiin varjostimen ulkopuolella pääohjelmassa. Tässäkin tapauksessa olisi voitu käyttää textureOffset-kutsua texture-kutsun sijasta.

```
float GetBlurDiameter (float depth)
{
    float D = depth;
    float Ds = perfect_focus_distance;
    float Xd = abs(D - Ds);

    float b=0;
    if(depth < perfect_focus_distance)
        b = blur_coefficient * (Xd / ( Ds - Xd));
    else
        b = blur_coefficient * (Xd / ( Ds + Xd));

    return b * PPM;
```

```
}

const float MAX_BLUR_RADIUS = 10.0;

float blur_radius = GetBlurDiameter(depth);
blur_radius = min(floor(blur_radius), MAX_BLUR_RADIUS);


for(float i = -blur_radius/2.0; i < blur_radius/2.0; ++i)
    color += texture(color_texture, tex_coord + texel_offset * i);

color_out = color / blur_radius;
```

Lopuksi kun sumennettu tekstuuri on tehty, yhdistetään sumennettu tekstuuri tarkan tekstuurin kanssa lineaarisesti interpoloiden.

```
const float MAX_BLUR_RADIUS = 10.0;

vec4 color = texture(color_texture, tex_coord);
vec4 blurred_color = texture(blurred_color_texture, tex_coord);

float blur_radius = GetBlurDiameter(depth);
float blur_ratio = min(blur_radius / MAX_BLUR_RADIUS, 1.0);

color_out = mix(color, blurred_color, blur_ratio);
```

4.4 Hehku (Bloom)

Hehkutekstuuri laskettiin g-puskurista käyttämällä sivulla 29 olevaa kaavaa. Hehkutekstuurin koko oli kahdeksasosa alkuperäisestä g-puskurin koosta. Raja-arvoksi valittiin pitkän testaamisen jälkeen 0.9.

```
const float luma_thresshold = 0.9;
float luma = dot(color.rgb, vec3(0.2126,0.7152,0.0722));

if(luma > luma_thresshold)
    color_out =  color;
else
    color_out = vec4(0);
```

Sumennuksessa käytettiin kappaleessa 3.2.2 esiteltyä Gaussin sumennusta 7x7-alueella. Sumennus tehtiin kahdessa erässä pysty- ja vaakasuuntaan.

Lopuksi hehkutekstuuri ja väritekstuuri yhdistettiin väritekstuurin resoluutiolla. Kaavaksi valittiin kaava 17.

```
color_out= bloom_color+color_out-bloom_color*color_out;
```

5 Yhteenveto

Opinnäytetyössä tarkasteltiin yleisimpiä reaaliaikaisen jälkikäsittelyn algoritmeja, joita käytetään videopelien ja reaaliaikaisten animaatioiden grafiikassa. Tarkoituksena oli toteuttaa yksinkertainen, mutta kattava kokoelma keskeisimpiä jälkikäsittelyalgoritmeja. Tämän työn ohjeilla voi videopelien ja reaaliaikaisten animaatioiden ohjelmoija toteuttaa työlleen hyvännäköisen jälkikäsittelyn.

Työ oli osittain haastavaa sillä kirjallisuuslähteet eri jälkikäsittely algoritmeista on puuttelliset. Työssä käytettiin paljon verkkolähteitä joissa oli huonona puolena samojen algoritmien eriversiot. Jotkin versiot olivat todella hitaita ja toiset yksinkertaisesti eivät toimineet. Työhön valittiin algoritmit joissa yhdistyi sekä hyvä suorituskyky että laadukas jälki.

Tulevaisuudessa jälkikäsittelyalgoritmien käyttö tulee pitämään pintansa ainakin niin kauan kunnes reaaliaikaisesta säteenjäljityksestä tulee niin nopea, että sitä voidaan käyttää kaikissa tilanteissa.

LÄHTEET

Meyer, N. 10.7.2012. Shader Effects: Depth of Field. [Verkkosivu].[Viitattu 24.4.2013]. Saatavana http://devmaster.net/posts/3021/shader-effects-depth-of-field

Thai-Duong, H. & Kok-Lim, L. 2012. Efficient Screen-Space Approach to High-Quality Multi-Scale Ambient Occlusion. [Verkkojulkaisu]. [Viitattu 20.4.2013]. Saatavana http://www.comp.nus.edu.sg/~duong/tvc_ao.pdf

Lonroth, P. & Unger. M 11.5.2009. Advanced Real-time Post-Processing using GPGPU techniques. [Verkkojulkaisu]. [Viitattu 21.4.2013]. Saatavana http://dice.se/wp-content/uploads/Thesis-DoF.Per_.Lonroth.Mattias.Unger_.VT08.Final_.pdf

Rákos, D. 9.2010. Efficient Gaussian blur with linear sampling. [Verkkosivu]. [Viitattu 25.4.2013]. Saatavana http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

Sherrod, A. 4.2008. Game Graphics Programming. [Verkkokirja]. Boston: Course Technology. [Viitattu 24.4.2013]. Saatavana Ebrary-tietokannasta. Vaatii käyttöoikeuden.

Huxtable, J. Java Image Processing – Blurring for Beginners. [Verkkosivu]. [Viitattu 26.4.2013]. Saatavana http://www.jhlabs.com/ip/blurring.html

Atkin, D. Ei päiväystä.The Right CPU for You. [Verkkosivu]. [Viitattu 26.4.2013]. Saatavana http://www.computershopper.com/feature/the-right-gpu-for-you

Kessenich, J., Baldwin, D. & Ros, R. 3.8.2012. The OpenGL Shading Language, revision 6. [Verkkojulkaisu]. [Viitattu 29.4.2013]. Saatavana http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf

James, G & O'Rorke, J. 26.5.2004. Real-Time Glow. [Verkkosivu]. [Viitattu 3.5.2013]. Saatavana http://www.gamasutra.com/view/feature/2107/realtime_glow.php

Kalogirou, H. 20.5.2006. How to do good bloom for HDR rendering. [Verkkosivu]. [Viitattu 5.5.2013]. Saatavana http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/