Markus Kivioja

Hyperspektrikuvantaminen fotometrisessä stereossa ja sen hyödyntäminen tietokonegrafiikassa

Tietotekniikan pro gradu -tutkielma

10. kesäkuuta 2017

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Markus Kivioja

Yhteystiedot: `mailkivi@jyu.fi`

Ohjaajat: Ilkka Pölönen, Matti Eskelinen ja Tuomo Rossi

Työn nimi: Hyperspektrikuvantaminen fotometrisessä stereossa ja sen hyödyntäminen tietokonegrafiikassa

Title in English: Hyperspectral imaging in photometric stereo and its utilization in computer graphics

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Tieteellinen laskenta

Sivumäärä: 80+38

Tiivistelmä: Tässä tutkielmassa tutkitaan hyperspektrikuvien avulla toteutetun fotometrisen stereon vaikutuksia tietokonegrafiikassa. Tutkimuksessa hyperspektrikuvien ja fotometrisen stereon avulla luotiin malleja, joista edelleen renderöitiin kuvia. Hyperspektrikuvantamisen havaittiin parantavan renderöinnin kuvanlaatua hieman, mutta heikentävän sen suorituskykyä kymmenkertaisesti. Lisäksi tutkittiin mallien luomisen nopeuttamista rinnakkaislaskennan avulla. Tuloksena pintojen muotojen määrittäminen pintojen orientaatioiden avulla saatiin grafiikkasuorittimella yli 13-kertaisesti nopeammaksi.

Avainsanat: hyperspektrikuvantaminen, fotometrinen stereo, tietokonegrafiikka, gpu, gpgpu

Abstract: In this thesis the effects of photometric stereo implemented using hyperspectral images in computer graphics are studied. In the study hyperspectral images and photometric stereo was used to create models that were further rendered into images. It was observed that hyperspectral imaging slightly improved the image quality of the rendering but decreased its performance by a factor of ten. Additionally it was studied how the model creation can be accelerated using parallel computing. As a result resolving surface shapes from surface orientations got more than 13 times as fast when a graphics processing unit was used.

Keywords: hyperspectral imaging, photometric stereo, computer graphics, gpu, gpgpu

Termiluettelo

Aallonpituusalue	Jokin rajattu ja yhtenäinen aallonpituuksien joukko.
RGB	Punaista, vihreää ja sinistä väriä vastaavat valon aallonpituusalueet (red, green, blue).
CPU	Tietokoneen keskussuoritin.
GPU	Tietokoneen grafiikkasuoritin.
CUDA	Ohjelmointirajapinta GPU:lla suoritettavan laskennan ohjelmointiin.
Matlab	Numeerisen laskennan tietokoneohjelmisto ja ohjelmointikieli.
$\boldsymbol{v}$	Vektori (pienet lihavoidut kirjaimet).
$\boldsymbol{M}$	Matriisi (isot lihavoidut kirjaimet).
$\boldsymbol{M}^T$	Matriisin transpoosi.
$\boldsymbol{M}^{-1}$	Käänteismatriisi.

Kuviot

Kuvio 1. Näkyvän valon spektri 4
Kuvio 2. Hyperspektrikuva 19
Kuvio 3. Eri kuvantamismenetelmien spektrien erot 20
Kuvio 4. Valojen sijainnit ja numerointi 33
Kuvio 5. Kuvausasetelma 34
Kuvio 6. Yhdistetyt kalibraatiokuvat 35
Kuvio 7. Maailman koordinaattien määrittäminen 49
Kuvio 8. Hyperspektrikuvien avulla renderöidyt kuvat verrattuna kontrollikuviin 57
Kuvio 9. Syvyyskarttojen avulla renderöidyt kuvat 58
Kuvio 10. Valonlähteen suunnan vaihtelu 59
Kuvio 11. Normaalivektoreiden elevaatiokulmat aallonpituuden suhteen 60
Kuvio 12. Normaalivektorin atsimuuttikulma aallonpituuden suhteen 60

Taulukot

Taulukko 1. Renderöityjen kuvien histogrammien keskiarvot $\bar{x}$ ja keskihajonnat σ_x, muodossa $\bar{x} \pm \sigma_x$ 58
Taulukko 2. Valojen suuntien elevaatio- ja atsimuuttikulmien keskivirheet radiaaneina. 59
Taulukko 3. Normaalivektoreiden elevaatio- ja atsimuuttikulmien keskineliövirheiden neliöjuuret. 61
Taulukko 4. Eri vaiheiden suoritukseen eri toteutuksien avulla käytetyt ajat sekunteina. 62

Sisältö

1 JOHDANTO ... 1

2 TEORIA ... 4
2.1 Valosta yleisesti ... 4
2.1.1 Sähkömagneettinen säteily ... 4
2.1.2 Radiometriikka ... 5
2.1.3 Fotometria ... 6
2.2 Valon heijastuminen ... 7
2.2.1 Albedo ... 7
2.2.2 Diffuusi heijastus ja Lambertin pinta ... 9
2.2.3 Kollimoidun valon heijastuminen Lambertin pinnalla ... 10
2.3 Fotometrinen stereo ... 12
2.3.1 Katsojasuuntautunut koordinaatisto ja reflektanssikartta ... 12
2.3.2 Albedon ja normaalin määrittäminen ... 13
2.3.3 Pinnan muodon määrittäminen ... 16
2.4 Spektroskopia ... 17
2.4.1 Spektrikuvantaminen ... 17
2.4.2 Hyperspektrikuvantaminen ... 18
2.4.3 Hyperspektrikameroista ... 20
2.5 Tietokonegrafiikka ... 22
2.5.1 Kolmiulotteinen tietokonegrafiikka ... 23
2.5.2 Varjostinohjelmat ... 25
2.5.3 Teksturointi ... 26
2.5.4 OpenGL ... 27
2.6 GPGPU ja numeerinen lineaarialgebra ... 28
2.6.1 GPGPU ... 28
2.6.2 OpenCL ... 29
2.6.3 Numeerinen lineaarialgebra ... 30
2.6.4 ViennaCL ... 31

3 HYPERSPEKTRIKUVANTAMINEN FOTOMETRISESSÄ STEREOSSA ... 32
3.1 Koejärjestely ... 32
3.1.1 Kuvausasetelma ... 32
3.1.2 Koelaitteisto ... 33
3.1.3 Koeasetukset ... 33
3.2 Valojen suunnat ... 35
3.2.1 Kalibrointi- ja maskikuvat ... 35
3.2.2 Suuntien määrittäminen ... 36
3.3 Albedo- ja normaalikarttojen luominen ... 38
3.3.1 Lähteen irradianssi ... 38
3.3.2 Albedojen ja normaalien määrittäminen ... 39
3.3.3 Matlab-toteutus ... 40

3.3.4 GPU-toteutus ... 41
3.3.5 Säikeistetty CPU-toteutus ... 42
3.4 Syvyyskartan luominen ... 42
3.4.1 Harvan yhtälöryhmän muodostaminen ... 42
3.4.2 Syvyyskarttojen muodostaminen Matlabilla ... 45
3.4.3 Syvyyskarttojen muodostaminen GPU:lla ... 46

4 HYÖDYNTÄMINEN TIETOKONEGRAFIIKASSA ... 48
4.1 Renderöijän toteutus ... 48
4.1.1 Simuloidut valonlähteet ... 51
4.1.2 Grafiikkaliukuhihnan sovellusvaihe ... 51
4.1.3 Varjostinohjelmien toteutus ... 53
4.1.4 Kontrollikuvat ... 54

5 TULOKSET ... 56
5.1 Hyperspektrikuvantamisen vaikutus kuvanlaatuun ... 56
5.2 Orientaatiot aallonpituuksien funktioina ... 59
5.3 Toteutuksien suoritusajat ... 61
5.4 Pohdinta ... 62
5.4.1 Fotometrinen stereo ja hyperspektrikuvantaminen tietokonegrafiikassa . 62
5.4.2 Hyperspektrikuvantaminen fotometrisessä stereossa ... 64
5.4.3 Fotometrisen stereon suorituskyky ... 65

6 YHTEENVETO ... 67

LÄHTEET ... 68

LIITTEET ... 75
A Valojen suuntien laskennan Matlab-lähdekoodi ... 75
B Fotometrisen stereon Matlab-toteutuksen lähdekoodi ... 76
C Syvyyskartan laskennan Matlab-toteutuksen lähdekoodi ... 77
D Fotometrisen stereon ViennalCL-toteutuksen lähdekoodi ... 79
E Syvyyskartan laskennan ViennaCL-toteutuksen lähdekoodi ... 86
F Fotometrisen stereon Armadillo-toteutuksen lähdekoodi ... 89
G Renderöijän CPU-osuuden lähdekoodi ... 101
H Renderöijän verteksivarjostinohjelma ... 111
I Renderöijän kuvapistevarjostinohjelma ... 111

1 Johdanto

Tässä tutkielmassa tarkastellaan fotometrisen stereon toteuttamista hyperspektrikuvien avulla sekä sen tuomia hyötyjä ja haittoja tietokonegrafiikassa. Lisäksi tutkitaan hyperspektrikuvien avulla toteutettuun fotometriseen stereoon liittyvän laskennan nopeuttamista moniydin- ja grafiikkasuorittimilla suoritettavaa rinnakkaislaskentaa käyttäen. Tutkimus pyrkii vastaamaan kysymyksiin siitä, millä tavoin ja kuinka paljon hyperspektrikuvantaminen vaikuttaa fotometrisellä stereolla luoduista malleista renderöityjen kuvien kuvanlaatuun, sekä kuinka paljon rinnakkaislaskenta nopeuttaa näiden mallien luontia.

Fotometrinen stereo on tietokonenäön alaan kuuluva menetelmä, jolla kaksiulotteisten kuvien pohjalta muodostetaan tietoa kuvattujen kohteiden kolmiulotteisista muodoista ja kyvyistä heijastaa valoa. Sitä sovelletaan usein käyttäen tavallisia valokuvia, joissa on kolme värikanavaa: punainen, vihreä ja sininen. Hyperspektrikuvat sen sijaan saattavat muodostua sadoista kapeista valon aallonpituusalueista, jotka yhdessä muodostavat jatkuvan spektrin. Käyttämällä fotometrisessä stereossa hyperspektrikuvia on mahdollista luoda tietokonegrafiikassa käytettäviä kolmiulotteisia malleja, jotka sisältävät mallinnettavasta kohteesta enemmän informaatiota kuin tavallisten valokuvien avulla luodut mallit. Lisäinformaation avulla voi olla mahdollista parantaa malleista renderöityjen kuvien kuvanlaatua. Datan määrän kasvaessa myös menetelmän vaatiman laskennan määrä kasvaa, joten on tärkeää tutkia tapoja sen nopeuttamiseksi. Menetelmässä suoritettava laskenta on hyvin rinnakaistuvaa, joten on mielekästä tutkia sen nopeuttamista moniydin- ja grafiikkasuorittimella suoritettavaa rinnakkaislaskentaa hyödyntämällä.

Spektrikuvantamisen hyödyntämistä fotometrisessä stereossa on tutkittu aiemmin eri näkökulmista. Nam ja Kim (2014) esittivät tutkimuksessaan menetelmän, jossa multispektrikuvantamisen avulla voidaan fotometrisestä stereosta poistaa tutkittavan kappaleen itse aiheuttamien heijastusten vaikutus. Menetelmän avulla fotometrisen stereon tarkkuutta pystyttiin kasvattamaan huomattavasti. Ozawa, Sato ja Yamaguchi (2017) tutkivat hyperspektrikuvantamisen hyödyntämistä fotometrisessä stereossa esittämällä menetelmän, jonka avulla fotometrinen stereo on mahdollista toteuttaa ainoastaan yhtä valonlähdettä ja hyperspektrikuvaa käyttäen. Eskelinen ym. (2016) pyrkivät tutkimuksessaan arvioimaan ihmisen ihon raken-

netta yhdistämällä fotometrisen stereon ja hyperspektrikuvantamisen. Fotometrisen stereon ja hyperspektrikuvantamisen yhteiskäytön hyödyntämisestä tietokonegrafiikassa ei ole aiempaa tutkimusta.

Fotometrinen stereo on laskennallisesti suhteellisen raskas menetelmä myös tavallisia valokuvia käyttäen, joten sen toteuttamista grafiikkasuorittimen avulla on tutkittu aiemminkin. Pevar ym. (2015) käsittelivät tutkimuksessaan reaaliaikaista fotometristä stereota. Tätä tarkoitusta varten he kehittivät optimoidun grafiikkasuoritintoteutuksen, joka oli 950-kertaisesti nopeampi kuin sarjallinen toteutus. Varnavas ym. (2010) kehittivät tutkimuksessaan algoritmin, joka poistaa fotometrisen stereon kanssa yleensä käytetyn oletuksen paikan suhteen vakiosta valon suunnasta. Algoritmi toteutettiin grafiikkasuorittimille, koska sen laskennallinen vaativuus kasvaa lineaarisesti kuvapisteiden lukumäärän suhteen. Molemmat tutkimukset toteutettiin käyttäen Nvidian CUDA-ohjelmointirajapintaa ja itse toteutettua, käyttötarkoitukseen mukautettua grafiikkasuorittimen lähdekoodia. Fotometrisen stereon toteuttamista valmista, grafiikkasuoritinta hyödyntävää lineaarialgebrakirjastoa käyttäen ei ole tutkittu aiemmin. Lisäksi rinnakkaislaskennan hyödyntämisestä erityisesti hyperspektrikuvantamalla toteutetussa fotometrisessä stereossa ei ole aiempaa tutkimusta.

Tutkimuksessa havaittiin, että hyperspektrikuvien käyttäminen fotometrisessä stereossa parantaa sen avulla luoduista malleista renderöityjen kuvien kuvanlaatua, mutta erot ovat pieniä. Tutkimuksessa kehitetyt moniydin- ja grafiikkasuorittimen rinnakkaislaskentaa hyödyntävät hyperspektraalin fotometrisen stereon toteutukset eivät olleet sarjallista toteutusta nopeampia. Kolmiulotteisen mallin luomisen viimeisessä vaiheessa – jossa pinnan muoto määritetään sen orientaatioiden avulla – grafiikkasuoritintoteutus oli yli 13 kertaa niin nopea kuin sarjallinen toteutus.

Luvussa 2 selvennetään tutkimuksessa käytettyjä käsitteitä ja teoriaa. Alaluvut 2.1 ja 2.2 esittelevät valon voimakkuuden, sen heijastumisen sekä pintojen heijastavuuden mittaamiseen käytettäviä suureita ja menetelmiä. Alaluku 2.4 pyrkii selittämään mitä hyperspektrikuvantaminen on, mikä on sen suhde muuhun kuvantamiseen ja kuinka sitä käytännössä toteutetaan. Alaluku 2.5 selventää tietokonegrafiikan yleisiä käsitteitä, toimintaperiaatteita ja työkaluja. Alaluvussa 2.6 selvitetään mitä numeerinen lineaarialgebra tarkoittaa ja kuinka sen laskennassa voidaan hyödyntää grafiikkasuorittimia.

Luku 3 kuvailee tutkimuksen osuuden, jossa tutkittavat kohteet hyperspektrikuvataan ja niistä muodostetaan kolmiulotteiset mallit. Luvun alussa esitellään tutkimuksessa käytetty asetelma ja laitteisto, sekä selvitetään kuinka valonlähteiden suunnat määritettiin. Tämän jälkeen käydään läpi fysikaalinen ja matemaattinen teoria mallien luomisen taustalla sekä sen eri toteutukset.

Luvussa 4 esitellään mallien renderöijän toteutus. Aluksi esitellään renderöinnissä käytetyt virtuaaliset valonlähteet, jonka jälkeen pyritään selittämään renderöijän lähdekoodin toimintaa ja sen toteuttamisessa käytettyjä ratkaisuja. Lopussa kerrotaan tuloksien vertailukohtina toimineiden kontrollikuvien luomisessa käytetystä teoriasta ja toteutuksesta.

Luku 5 listaa kaikkien tutkimuksessa suoritettujen mittauksien tulokset. Luvussa myös tarkastellaan tutkittavien pintojen muotojen vaihteluita aallonpituuden suhteen. Lisäksi luvussa pohditaan syitä saatuihin tuloksiin, tutkimuksessa käytettyjen oletuksien oikeutettavuutta ja kuinka tutkimusta voitaisiin tulevaisuudessa kehittää. Luvun lopussa ehdotetaan vielä lyhyesti aiheita mahdollisille tulevaisuuden tutkimuksille.

2 Teoria

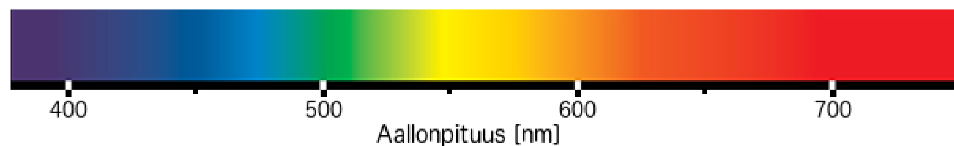
2.1 Valosta yleisesti

2.1.1 Sähkömagneettinen säteily

Sähkömagneettisen säteilyn kaikkia tunnettuja taajuuksia ja niitä vastaavia aallonpituuksia kutsutaan kollektiivisesti sähkömagneettisen säteilyn *spektriksi*, joka on jaoteltu aallonpituuden perusteella edelleen pienempiin luokkiin. Nämä luokat ulottuvat ELF (extremely low frequency) -säteilyn satojentuhansien kilometrien kokoluokan aallonpituuksista aina gammasäteilyn pikometrin kokoluokan aallonpituuksiin.

Sähkömagneettisen säteilyn vaikutus ympäristöön riippuu sen aallonpituudesta, koska säteilyn sisältämä energia on kääntäen verrannollinen sen aallonpituuteen. Sähkömagneettisen säteilyn lyhyiden aallonpituuksien päässä noin 380 nm ja 740 nm välissä olevaa aluetta kutsutaan näkyväksi valoksi. Tällä välillä olevan säteilyn sisältämä energiamäärä kykenee aiheuttamaan ihmissilmässä molekyylitason muutoksen, jonka ihminen kokee näköaistimuksena. Väli on myös lähes sama kuin aallonpituusalue, jonka ilmakehä päästää auringon säteilystä lävitseen.

Myös näköaistimuksen laatu riippuu silmän vastaanottaman säteilyn aallonpituudesta. Ihminen havaitsee näkyvän valon aallonpituusalueen sisällä olevat eri aallonpituudet eri väreinä. Tätä värijakaumaa kutsutaan näkyvän valon spektriksi, ja se kuvaa värien sekä aallonpituuksien välistä suhdetta (Tekatch 2009).



Kuvio 1. Näkyvän valon spektri.

2.1.2 Radiometriikka

Sähkömagneettisen säteilyn mittaamiseen käytettävien tekniikoiden ja suureiden kokoelmaa kutsutaan *radiometriikaksi*. Radiometriset suureet ja niiden yksiköt ovat

säteilyn vuo	Φ	$[W]$,	(2.1)
säteilyn intensiteetti	$I = \frac{d\Phi}{d\omega}$	$\left[\frac{W}{sr}\right]$,	(2.2)
irradianssi	$E = \frac{d\Phi}{dA}$	$\left[\frac{W}{m^2}\right]$,	(2.3)
eksitanssi	$M = \frac{d\Phi}{dA}$	$\left[\frac{W}{m^2}\right]$ ja	(2.4)
radianssi	$L = \frac{d^2\Phi}{dA_{\text{proj}}d\omega}$	$\left[\frac{W}{m^2 sr}\right]$,	(2.5)

missä

$$dA_{\text{proj}} = dA\ \cos(\theta), \tag{2.6}$$

ja θ on saapuvan säteilyn ja tarkasteltavan pinnan normaalivektorin välinen kulma (Nicodemus ym. 1977).

Säteilyn vuo mittaa emittoidun, saapuvan tai heijastuneen sähkömagneettisen säteilyn kokonaistehoa. Säteilyn vuo ilmaisee säteilyn sisältämän energian määrää aikayksikköä kohden, eli $\Phi = \frac{dQ}{dt}$, missä Q on säteilyn energia, ja t on aika. Esimerkiksi hehkulampun säteilyn vuo voidaan periaatteessa mitata summaamalla sen jokaisen yhden sekunnin aikana emittoiman fotonin energia yhteen.

Säteilyn intensiteetti on suunnan huomioon ottava suure. Se mittaa säteilyn vuon tiheyttä suhteessa avaruuskulmaan.

Irradianssi on jollekin pinnalle saapuvan säteilyn vuon tiheys suhteessa pinta-alaan. Irradianssi määritellään suhteessa johonkin pintaan, joka voi olla kuvitteellinen pinta tai useimmiten tutkittavan kohteen pinta. Kun puhutaan valon lähteen säteilemästä irradianssista, irradianssi mitataan säteilyn suuntaa vastaan kohtisuoralta kuvitteelliselta pinnalta.

Eksitanssi ilmaisee irradianssin tavoin säteilyn vuon tiheyttä suhteessa pinta-alaan. Eksitanssia käytetään kuitenkin ainoastaan mitatessa pinnalta poispäin heijastuvan säteilyn vuon

tiheyttä. Jos pinta on täydellisen heijastava, niin irradianssi ja eksitanssi ovat yhtä suuret.

Säteilyn intensiteetin tavoin myös radianssi on suunnan huomioon ottava suure. Se ilmaisee säteilyn vuon tiheyttä suhteessa sekä pinta-alaan että avaruuskulmaan. Sen voidaan ajatella olevan säteilyn suuntaa vastaan kohtisuoralta pinnalta mitatun irradianssin (tai eksitanssin) tiheys suhteessa avaruuskulmaan:

$$L = \frac{dE}{d\omega \cos(\theta)}. \tag{2.7}$$

Radianssi ilmaisee kuinka suuri osa emittoidusta tai heijastuneesta sähkömagneettisesta säteilystä saapuu sitä jostakin suunnasta mittaavaan sensoriin (silmään, kameraan tai muuhun vastaavan). Valon tapauksessa se näin ollen kuvaa sitä, kuinka kirkkaalta jokin kohde näyttää. Radianssin voidaan ajatella olevan yhden valon säteen voimakkuus.

Jos radianssin määritelmän pinta-ala mitataan pinnalta, johon säteily saapuu, niin avaruuskulma on säteilyn lähteen muodostama avaruuskulma tältä pinnalta katsottuna. Jos taas pinta-alalla viitataan säteilyn lähteen pinta-alaan niin avaruuskulma on säteilyn mittaavan sensorin muodostama avaruuskulma lähteestä katsottuna (Horn ja Sjoberg 1979).

Kaikki edellä kuvatut suureet ovat käytännössä aina riippuvaisia myös säteilyn aallonpituudesta. Aallonpituuden funktioina olevia suureita kutsutaan *spektraalisiksi* suureiksi, ja ne on määritelty suureiden osuutena infinitesimaalisen leveää aallonpituusaluetta kohden eli suureiden derivaattana aallonpituuden suhteen. Spektraalisia suureita merkitään yleisesti alaindeksillä λ (Palmer ja Carroll 1999). Esimerkiksi *spektraalinen radianssi*

$$L_\lambda = L_\lambda(\lambda) = \frac{d^3\Phi}{dA_{\text{proj}} d\omega d\lambda} \left[\frac{W}{m^2 sr\ nm}\right]. \tag{2.8}$$

2.1.3 Fotometria

Radiometriikka käsittelee puhtaasti fysikaalisia suureita eikä näin ollen ota huomioon inhimillisiä havaitsemiseen liittyviä tekijöitä. Fotometria taas on radiometriikkaan hyvin läheisesti liittyvä ala, jossa suureet on kiinnitetty ihmissilmän herkkyyteen. Fotometria mittaa valoa, eli sähkömagneettisen säteilyn näkyvää aallonpituusaluetta, jolloin se on rajoitettu suurin piirtein aallonpituuksien 380 nm ja 740 nm välille. Se mittaa valoa yksiköissä, jotka on painotettu ihmissilmän visuaalisella valonvasteella.

Ihmissilmän valonherkkyys riippuu valon aallonpituudesta, jolloin esimerkiksi kaksi saman suuruista mutta eri aallonpituusalueiden radianssia eivät näytä yhtä kirkkailta. Tästä syystä fotometriassa kaikki radiometriikan suureet on painotettu aallonpituudesta riippuvalla funktiolla, jota kutsutaan *CIE fotometriseksi käyräksi*. Se on tilastollisin menetelmin kehitetty, muodoltaan Gaussin jakaumaa muistuttava funktio, jonka maksimi on noin 555 nm kohdalla. Myös fysikaalisien säteilysensorien kuten kameroiden havaitsemisherkkyys riippuu aallonpituudesta, jolloin niilläkin on niille ominaiset spektraalisen herkkyyden painofunktiot.

Fotometria eroaakin radiometriikasta ainoastaan siinä, että sen yksiköt on painotettu aallonpituudesta riippuvalla funktiolla ja että tarkasteltava aallonpituusalue on kapeampi (Ashdown 2002).

2.2 Valon heijastuminen

2.2.1 Albedo

Radiometriikassa säteilyn heijastumista pinnalla kuvataan *kaksisuuntaisheijastusjakaumafunktiolla* BRDF (bidirectional reflectance distribution function). BRDF on pinnalta johonkin tiettyyn suuntaan heijastuneen differentiaalisen radianssin suhde pinnalle jostakin tietystä suunnasta saapuvaan differentiaaliseen irradianssiin. BRDF on siis sekä saapuvan (viitataan jatkossa alaindeksillä i) että heijastuvan (viitataan alaindeksillä r) suunnan funktio ja sitä merkitään symbolilla f_r.

$$\text{BRDF} = f_r(\theta_i, \phi_i; \theta_r, \phi_r) = \frac{dL_r(\theta_i, \phi_i; \theta_r, \phi_r)}{dE_i(\theta_i, \phi_i)} \qquad \left[\frac{1}{sr}\right], \tag{2.9}$$

missä θ_i ja ϕ_i ovat saapuvan irradianssin pallokoordinaateissa ilmaistun suunnan elevaatio- ja atsimuuttikulmat tässä järjestyksessä. θ_r ja ϕ_r ovat heijastuvan radianssin suunnan vastaavat kulmat (Nicodemus ym. 1977). Kulmat ovat ilmaistu *lokaalissa* koordinaatistossa, missä karteesisen koordinaatiston z-akseli on samansuuntainen pinnan normaalin kanssa sekä θ määritelty suhteessa siihen ja ϕ suhteessa x-akseliin (Horn ja Sjoberg 1979). Jatkossa on aina käytössä lokaali koordinaatisto kunnes toisin ilmoitetaan.

Kuten luvussa 2.1.2 esitettiin, irradianssi ja radianssi ovat aallonpituuden funktioita, joten

BRDF voidaan ilmaista myös näiden spektraalisten suureiden avulla. Näin saadaan *spektraalinen kaksisuuntaisheijastusjakaumafunktio* (Schaepman-Strub ym. 2006)

$$\mathrm{BRDF}_\lambda = f_{r_\lambda}(\theta_i, \phi_i; \theta_r, \phi_r; \lambda) = \frac{dL_{\lambda_r}(\theta_i, \phi_i; \theta_r, \phi_r; \lambda)}{dE_{\lambda_i}(\theta_i, \phi_i; \lambda)} \qquad \left[\frac{1}{sr}\right]. \tag{2.10}$$

BRDF:n arvot eivät ole mitattavissa, koska BRDF on differentiaalisten suureiden suhde ja suureet voidaan mitata käytännössä vain joidenkin nollaa suurempien intervallien yli (Nicodemus ym. 1977). BRDF:stä voidaan kuitenkin johtaa useita relevantteja mitattavia suureita ottamalla saapuvissa ja/tai heijastuvissa suunnissa huomioon nollaa suuremmat äärelliset avaruuskulmat ja integroimalla niiden yli (Schaepman-Strub ym. 2006). Osaa näistä suureista kutsutaan *reflektansseiksi*. Reflektanssit kuvaavat pinnalta heijastuvan säteilyn vuon suhdetta pinnalle saapuvaan säteilyn vuohon ja niitä merkitään symbolilla ρ (Nicodemus ym. 1977):

$$\rho = \frac{d\Phi_r}{d\Phi_i}. \tag{2.11}$$

Reflektanssit ovat yksiköttömiä ja saavat energiaperiaatteen nojalla arvoja vain väliltä $[0,1]$.

Yksi tällaisista reflektansseista on *kaksoispuolipalloinen reflektanssi* BHR (BiHemispherical Reflectance)(Nicodemus ym. 1977) eli *albedo* (Martonchik, Bruegge ja Strahler 2000). Se on pinnalta koko puolipallon kokoiseen avaruuskulmaan heijastuvan säteilyn vuon suhde pinnalle koko puolipallon kokoisesta avaaruuskulmasta saapuvaan säteilyn vuohon. Albedoa merkitään jatkossa symbolilla a, ja

$$a = \rho(2\pi; 2\pi) = \frac{d\Phi_r(2\pi; 2\pi)}{d\Phi_i(2\pi)}. \tag{2.12}$$

Sulkujen sisällä ensimmäinen 2π tarkoittaa, että saapuvaa säteilyä integroidaan puolipallon kokoisen avaruuskulman yli ja jälkimmäinen heijastuvaa.

Radianssin ja irradianssin suhteesta (2.7) seuraa, että

$$dM_r = L_r \cos(\theta_r)\, d\omega_r, \text{ ja} \tag{2.13}$$

$$dE_i = L_i \cos(\theta_i)\, d\omega_i. \tag{2.14}$$

Nyt radiometristen suureiden määritelmistä (2.3), (2.4) ja (2.5) sekä integroimalla yhtälön (2.13) heijastuvan suunnan suhteen ja yhtälön (2.14) saapuvan suunnan suhteen, molemmat

puolipallojen yli, saadaan seuraavaa:

$$\begin{aligned} a &= \frac{d\Phi_r(2\pi;2\pi)}{d\Phi_i(2\pi)} \\ &= \frac{dA\ M_r(2\pi;2\pi)}{dA\ E_i(2\pi)} \\ &= \frac{\iint_{\omega_r=2\pi} L_r(2\pi;\theta_r,\phi_r)\ \cos(\theta_r)\ d\omega_r}{\iint_{\omega_i=2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i}. \end{aligned} \tag{2.15}$$

Lisäksi BRDF:n määritelmästä (2.9) ja yhtälöstä (2.14) seuraa, että

$$dL_r = f_r\ dE_i = f_r L_i \cos(\theta_i)\ d\omega_i. \tag{2.16}$$

Integroimalla edellisen yhtälön puolittain saapuvan suunnan suhteen puolipallon yli ja sijoittamalla tulokseen (2.15) saadaan albedo kirjoitettua BRDF:n ja saapuvan radianssin avulla seuraavasti:

$$a = \frac{\iint_{2\pi} \iint_{2\pi} f_r(\theta_i,\phi_i;\theta_r,\phi_r)\ L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i\ \cos(\theta_r)\ d\omega_r}{\iint_{2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i}. \tag{2.17}$$

Spektraalisen BRDF:n ja spektraalisen saapuvan radianssin avulla saadaan spektraalinen albedo

$$a_\lambda = \frac{\iint_{2\pi} \iint_{2\pi} f_{\lambda_r} L_{\lambda_i} \cos(\theta_i)\ d\omega_i\ \cos(\theta_r)\ d\omega_r}{\iint_{2\pi} L_{\lambda_i} \cos(\theta_i)\ d\omega_i}. \tag{2.18}$$

2.2.2 Diffuusi heijastus ja Lambertin pinta

Kun valo heijastuu joltakin pinnalta täydellisesti, se heijastuu ainoastaan yhteen suuntaan siten, että heijastuneen valon suunnan ja pinnan välinen kulma on yhtä suuri kuin saapuvan valon suunnan ja pinnan välinen kulma. Käytännössä valo kuitenkin heijastuu lähes aina yhden suunnan sijaan useisiin eri suuntiin, jolloin sitä kutsutaan *diffuusiksi* heijastukseksi.

Vaikka yksisuuntainen heijastus edellyttääkin täysin sileää pintaa, ei diffuusi heijastus yleisesti johdu pinnan rosoisuudesta. Yleisin syy diffuusille heijastukselle on valon tunkeutuminen pinnan läpi ja siroaminen pinnanalaisissa sisärakenteissa täysin satunnaisesti ennen paluutaan takaisin pinnan ulkopuolelle. Tämän lisäksi myös pinnan mahdolliset mikroskooppiset epätasaisuudet aiheuttavat diffuusia heijastusta (Hanrahan ja Krueger 1993).

Jos pinnalta heijastuva radianssi on suunnan suhteen vakio, niin pinta heijastaa täydellisen diffuusisti ja pintaa kutsutaan *Lambertin pinnaksi*. Lambertin pinta on ideaalinen mattapinta. Radianssin ja irradianssin suhteesta 2.7 nähdään, että jos heijastuva radianssi on elevaatiokulman θ suhteen vakio, niin pinnalta heijastuneelle irradianssille eli eksitanssille täytyy päteä $M = M_0 \cos(\theta)$, missä M_0 on pintaa vastaan kohtisuoran suunnan eksitanssi. Toisin sanoen Lambertin pinnalta heijastunut eksitanssi noudattaa Lambertin *kosinilakia*. Koska Lambertin pinnalta heijastunut radianssi ei riipu suunnasta, myös pinnan BRDF on vakio (Akenine-Möller, Haines ja Hoffman 2008, 110-111).

Jos pinta ei absorboi valoa lainkaan, vaan heijastaa kaiken sille saapuneen valon ja heijastus on täydellisen diffuusia, niin pintaa kutsutaan *ideaaliseksi* Lambertin pinnaksi (Martonchik, Bruegge ja Strahler 2000).

Jos pinta ei heijasta Lambertin pinnan tavoin vaan heijastunut radianssi riippuu heijastussuunnasta, niin heijastus on *spekulaarista* (Akenine-Möller, Haines ja Hoffman 2008, sivu 105).

2.2.3 Kollimoidun valon heijastuminen Lambertin pinnalla

Koska Lambertin pinnalta heijastuva radianssi on vakio, ei se, eikä sen myötä myöskään BRDF riipu saapuvan radianssin suunnasta. Näin ollen kun albedo esitetään BRDF:n ja saapuvan radianssin avulla tuloksen (2.17) mukaisesti, niin BRDF voidaan siirtää saapuvan suunnan suhteen suoritettavan integraalin ulkopuolelle ja saadaan

$$\begin{aligned} a &= \frac{\iint_{2\pi} f_r(\theta_i,\phi_i;\theta_r,\phi_r) \iint_{2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i\ \cos(\theta_r)\ d\omega_r}{\iint_{2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i} \\ &= \frac{\iint_{2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i \iint_{2\pi} f_r(\theta_i,\phi_i;\theta_r,\phi_r)\ \cos(\theta_r)\ d\omega_r}{\iint_{2\pi} L_i(\theta_i,\phi_i)\ \cos(\theta_i)\ d\omega_i} \\ &= \iint_{2\pi} f_r(\theta_i,\phi_i;\theta_r,\phi_r)\ \cos(\theta_r)\ d\omega_r. \end{aligned} \tag{2.19}$$

Lambertin pinnalta heijastuva radianssi ja siten BRDF:kään ei riipu myöskään heijastumissuunnasta, joten integroitavaksi jää ainoastaan $\cos(\theta_r)$. Tämä integraali saadaan määritettyä esittämällä differentiaalinen avaruuskulma pallokoordinaattien avulla seuraavasti:

$$d\omega_r = \sin(\theta_r)\ d\theta_r\ d\phi_r. \tag{2.20}$$

Integroimalla sekä elevaatio- että atsimuuttikulmien suhteen saadaan

$$\begin{aligned} a &= f_r \iint_{2\pi} \cos(\theta_r)\, d\omega_r \\ &= f_r \int_0^{2\pi} \int_0^{\pi/2} \cos(\theta_r)\, \sin(\theta_r)\, d\theta_r\, d\phi_r \\ &= f_r\, \pi, \end{aligned} \tag{2.21}$$

josta edelleen

$$f_r = \frac{a}{\pi}. \tag{2.22}$$

Jos valonlähteen etäisyys tarkasteltavasta pinnasta on suuri verrattuna valonlähteen kokoon, voidaan valo olettaa pistemäiseksi eli *pistevaloksi* (point light), tai jos etäisyys on lisäksi suuri verrattuna tarkasteltavan pinnan kokoon, voidaan se olettaa *suuntaisvaloksi* (directional light). Näissä tapauksissa pinnalle saapuva valo on *kollimoitua* eli se saapuu yhteen pisteeseen ainoastaan yhdestä suunnasta. Lisäksi suuntaisvalon tapauksessa valonsuunta on vakio myös kaikissa tarkasteltavissa pisteissä (Gustavson 2017, 39-40). Tällaiset valonlähteet eivät ole reaalimaailmassa mahdollisia, koska koko säteilyn vuo saapuisi nollan kokoisesta avaruuskulmasta, jolloin sitä vastaava radianssi olisi ääretön (Horn ja Sjoberg 1979). Kollimoidun valon radianssia voidaan kuitenkin kuvata Diracin deltafunktioiden avulla, jolloin integraalit niiden yli ovat äärellisiä. Horn ja Sjoberg (1979) kutsuvat tätä esitystapaa kollimoidun lähteen radianssin "tupladelta"-esitykseksi ja se on

$$L_i = \frac{E_0}{\sin(\theta_0)} \delta(\theta_i - \theta_0)\delta(\phi_i - \phi_0), \tag{2.23}$$

missä θ_0 ja ϕ_0 ovat saapuvan valon suunnan elevaatio- ja atsimuuttikulmat tässä järjestyksessä, E_0 saapuva irradianssi tätä suuntaa vastaan kohtisuoralta pinnalta mitattuna ja δ on Diracin deltafunktio.

Näin ollen kollimoidun valonlähteen Lambertin pinnalta heijastunut radianssin saadaan sijoittamalla "tupladelta"-esitys (2.23) sekä tulos (2.22) BRDF:n yhtälöön (2.16) ja integroimalla saapuvan suunnan suhteen puolipallon yli:

$$\begin{aligned} L_r &= \iint_{2\pi} \frac{a}{\pi} \frac{E_0}{\sin(\theta_0)} \delta(\theta_i - \theta_0)\delta(\phi_i - \phi_0)\, \cos(\theta_i)\, d\omega_i \\ &= \frac{a}{\pi}\, E_0 \int_0^{2\pi} \int_0^{\pi/2} \delta(\theta_i - \theta_0)\delta(\phi_i - \phi_0)\, \frac{\sin(\theta_i)}{\sin(\theta_0)}\, \cos(\theta_i)\, d\theta_i\, d\phi_i \\ &= \frac{a}{\pi}\, E_0\, \cos(\theta_0). \end{aligned} \tag{2.24}$$

Spektraalisten suureiden avulla ilmaistuna

$$L_{\lambda_r} = \frac{a_\lambda}{\pi} E_{\lambda_0} \cos(\theta_0). \tag{2.25}$$

2.3 Fotometrinen stereo

2.3.1 Katsojasuuntautunut koordinaatisto ja reflektanssikartta

Kuten luvussa 2.2.1 esitettiin, on tähän asti ollut käytössä ainoastaan lokaali koordinaatisto. Näin sen vuoksi, että BRDF, albedo ja muut reflektanssit ovat määritelty tässä koordinaatistossa, joten yksinkertaisuuden ja esittämisen selkeyden vuoksi myös saapuvan valon suunta on esitetty samassa koordinaatistossa. Todellisuudessa lokaali koordinaatisto on kuitenkin epäkäytännöllinen, koska yleensä tarkasteltavat pinnat ovat epätasaisia, jolloin lokaalin koordinaatiston orientaatio muuttuu paikan funktiona, mikä tekee globaalien jakaumien kuten valonlähteen radianssin määrittelemisestä hankalaa. Tästä syystä sopivampaa on käyttää paikan suhteen kiinteää koordinaatistoa. Yksi tällainen koordinaatisto on *katsojasuuntautunut* (viewer-oriented) koordinaatisto, jossa z-akseli on samansuuntainen katsomissuunnan kanssa. Katsojasuuntautuneessa koordinaatistossa elevaatiokulma θ on määritelty suhteessa katsomissuuntaan ja atsimuuttikulma ϕ katsomissuuntaa vastaan kohtisuoralla pinnalla suhteessa x-akseliin (Horn ja Sjoberg 1979).

Jos pinnan normaalin suunnan vastaavat kulmat ovat θ_n ja ϕ_n, niin kollimoidun valon irradianssin ja sen Lambertin pinnalta heijastuneen radianssin suhteelle (2.24) saadaan katsojasuuntatuneessa koordinaatistossa seuraavaa (Horn ja Sjoberg 1979):

$$L_r = \frac{a}{\pi} E_0 \; \cos(\theta_n)\cos(\theta_0) + \sin(\theta_n)\sin(\theta_0)\cos(\phi_0 - \phi_n). \tag{2.26}$$

Jos oletetaan, että valonlähde on tasainen suuntaisvalo, niin irradianssi E_0 ja kulmat θ_0 sekä ϕ_0 ovat paikan suhteen vakioita. Jos lisäksi oletetaan, että albedo a on tunnettu, niin edellisen yhtälön mukaisesti, heijastunut radianssi L_r riippuu ainoastaan pinnan normaalin kulmista θ_n ja ϕ_n. Toisin sanoen pinnan kirkkaus sisältää tiedon pinnan lokaalista muodosta. Tämän tiedon esittämiseksi B. K. P. Horn (1977) sekä Horn ja Sjoberg (1979) määrittelevät *reflektanssikartan* R, joka kuvaa pinnalta heijastuvan radianssin pinnan gradientin funktiona

katsojasuuntautuneessa koordinaatistossa:

$$R = R(p,q) \left[\frac{W}{m^2 sr} \right], \text{ missä } \quad p = \frac{\delta z}{\delta x} \quad \text{ja} \quad q = \frac{\delta z}{\delta y}. \tag{2.27}$$

z on pinnan korkeus suhteessa katsomissuuntaa vastaan kohtisuoraan vertailutasoon ja x sekä y etäisyydet tällä tasolla vastaavien ortogonaalisten koordinaattiakselien suuntaisesti mitattuna. Horn ja Sjoberg (1979) esittävät, että jos pinnan gradientit p ja q ilmaistaan pinnan normaalin suuntakulmien θ_n ja ϕ_n avulla sekä valon suunnan vastaavat arvot p_0 ja q_0 sen suuntakulmien θ_0 ja ϕ_0 avulla, saadaan

$$\begin{aligned} p &= -\cos(\phi_n)\tan(\theta_n), \quad q = -\sin(\phi_n)\tan(\theta_n), \\ p_0 &= -\cos(\phi_0)\tan(\theta_0), \quad q_0 = -\sin(\phi_0)\tan(\theta_0), \end{aligned} \tag{2.28}$$

jolloin näistä ja yhtälöstä (2.26) saadaan kollimoidun valon ja Lambertin pinnan reflektanssikartalle (Horn ja Sjoberg 1979)

$$R(p,q) = \frac{a}{\pi} E_0 \frac{p_0 p + q_0 q}{\sqrt{1 + p^2 + q^2}\sqrt{1 + p_0^2 + q_0^2}}. \tag{2.29}$$

Näin ollen pinnan normaali ja valon suunta voidaan lisäksi esittää normalisoitujen sarakevektorien $\boldsymbol{n} = [-p, -q, 1]^T / \sqrt{1 + p^2 + q^2}$ ja $\boldsymbol{l} = [-p_0, -q_0, 1]^T / \sqrt{1 + p_0^2 + q_0^2}$ avulla tässä järjestyksessä, jolloin reflektanssikartta voidaan kirjoittaa lyhyesti käyttäen niiden pistetuloa (Woodham 1980):

$$R(p,q) = \frac{a}{\pi} E_0 \, \boldsymbol{n} \bullet \boldsymbol{l}. \tag{2.30}$$

2.3.2 Albedon ja normaalin määrittäminen

Jos jokin kohde kuvataan kameralla, niin reflektanssikartta on käytännöllinen kohteen geometrian, sen heijastavuuden ja käytetyn valonlähteen yhdistävä malli. Mutta vaikka se mahdollistaa kuvan radianssiarvon ilmaisemisen kuvattavan kohteen pinnan orientaation funktiona, se ei kuitenkaan skalaariarvoisena kahden muuttujan funktiona ole kääntyvä. Näin ollen pinnan lokaalia asentoa ei pystytä määrittämään mitatusta radianssista ilman lisätietoa. Tähän perustuen Woodham (1980) esitti *fotometriseksi stereoksi* kutsutun menetelmän, jossa kohteesta otetaan vähintään kaksi kuvaa siten, että valonlähteestä saapuvaa valon suuntaa muutetaan kuvien välillä, pitäen kuitenkin kuvaussuunnan ja kohteen paikan sekä asennon

vakiona. Osoittautuu, että näin saadaan tarpeeksi informaatiota kohteen pinnan muotojen määrittämiseksi.

Jatkossa oletetaan, että kuvattava kohde heijastaa Lambertin pinnan mukaisesti ja että kohdetta valaistaan suuntaisvalolla, jonka irradianssi $E_0 = \pi \frac{W}{m^2}$, jolloin yhtälöstä (2.30) saadaan reflektanssikartaksi

$$R(p,q) = a\,\boldsymbol{n} \bullet \boldsymbol{l}. \tag{2.31}$$

Näin ollen, jos on olemassa kaksi kuvaa, joiden radianssit pisteessä (x,y) ovat $L_1(x,y)$ sekä $L_2(x,y)$, ja ne on kuvattu muuttamalla valonlähteen suuntaa siten, että ensimmäisessä kuvassa valon suunta on $\boldsymbol{l}_1$ ja toisessa $\boldsymbol{l}_2$, niin niitä vastaaville reflektanssikartoille $R_1(p,q)$ sekä $R_2(p,q)$ pätevät riippumattomat lineaariset yhtälöt

$$\begin{aligned} L_1(x,y) = R_1(p,q) = a\,\boldsymbol{n} \bullet \boldsymbol{l}_1, \\ L_2(x,y) = R_2(p,q) = a\,\boldsymbol{n} \bullet \boldsymbol{l}_2, \end{aligned} \tag{2.32}$$

joissa $\boldsymbol{n}$ on pinnan gradienteista p ja q riippuva vektori. Jos valon suuntien $\boldsymbol{l}_1$ ja $\boldsymbol{l}_2$ elevaatiokulmat θ_1 ja θ_2 ovat eri suuret ja aiemmin esitetty oletus tunnetusta albedosta pätee, niin tällä yhtälöparilla on yksikäsitteinen ratkaisu (Woodham 1980).

Jos myös albedo on tuntematon, niin voidaan lisäksi suunnasta $\boldsymbol{l}_3$ valaisemalla ottaa kolmas kuva, jonka radianssit ovat $L_3(x,y)$ ja vastaava reflektanssikartta $R_3(p,q)$, jolloin saadaan kolmas lineaarinen ja riippumaton yhtälö

$$L_3(x,y) = R_3(p,q) = a\,\boldsymbol{n} \bullet \boldsymbol{l}_3. \tag{2.33}$$

Muodostuneesta kolmen yhtälön lineaarisesta yhtälöryhmästä voidaan ratkaista sekä pinnan gradientit p ja q että albedo a.

Jos kolmen kuvan radianssiarvoista muodostetaan sarakevektori $\boldsymbol{i} = [L_1, L_2, L_3]^T$ ja rivivektoreiksi muutetuista valojen suunnista $\boldsymbol{l}^T$ matriisi $\boldsymbol{L}$, siten että

$$\boldsymbol{L}_{3\times 3} = \begin{bmatrix} l_{1_x} & l_{1_y} & l_{1_z} \\ l_{2_x} & l_{2_y} & l_{2_z} \\ l_{3_x} & l_{3_y} & l_{3_z} \end{bmatrix}, \tag{2.34}$$

niin edellä mainittu lineaarinen yhtälöryhmä voidaan esittää matriisimuodossa

$$\boldsymbol{L}\, a\, \boldsymbol{n} = \boldsymbol{i}. \tag{2.35}$$

Matriisi $\boldsymbol{L}$ on kääntyvä, jos ja vain jos sen rivivektorit ovat lineaarisesti riippumattomia eli valojen suunnat eivät ole samassa tasossa. Tässä tapauksessa yhtälöryhmälle saadaan yksikäsitteinen ratkaisu

$$a\, \boldsymbol{n} = \boldsymbol{L}^{-1}\boldsymbol{i}. \tag{2.36}$$

Koska normaali $\boldsymbol{n}$ on normalisoitu, eli $|\boldsymbol{n}| = 1$, niin albedo a saadaan ratkaisuvektorin pituutena

$$a = |a\, \boldsymbol{n}|, \tag{2.37}$$

josta edelleen normaali

$$\boldsymbol{n} = \frac{a\, \boldsymbol{n}}{a}. \tag{2.38}$$

Albedon tavoin myös tällä menetelmällä määritetty normaalivektori voi vaihdella aallonpituuden funktiona.

Käytännössä käy usein niin, että jokin kuvattava kohta on varjossa ainakin yhteen valonlähteeseen nähden, jolloin sekä normaalin että albedon määrittämiseen ei ole tarpeeksi informaatiota. Tätä ja muita mittauksesta johtuvia epätarkkuuksia voidaan korjata ottamalla useampia kuin kolme kuvaa, jolloin saadaan ylimääritelty yhtälöryhmä, joka voidaan ratkaista pienimmän neliösumman menetelmällä (Barsky ja Petrou 2003):

$$a\, \boldsymbol{n} = (\boldsymbol{L}^T\boldsymbol{L})^{-1}\boldsymbol{L}^T\boldsymbol{i}. \tag{2.39}$$

Edellä otettiin huomioon ainoastaan Lambertin pinnan heijastus, mutta fotometrinen stereo soveltuu myös monimutkaisempien BRDF:ien omaaville pinnoille käyttämällä niille ominaisia reflektanssikarttoja (Woodham 1980). Tällaisten pintojen BRDF:t omaavat katsomissuunnasta riippuvan eli spekulaarisen komponentin.

Fotometristä stereota on sovellettu muuan muassa kolmiulotteisessa kasvojen tunnistuksessa (Gao 2016), paperin ja kartongin valmistuksen laadunvalvonnassa (Löyttyniemi ym. 2017) sekä materiaalien tunnistamisessa (Kampouris ym. 2016). Fotometriseen stereoon liittyvä

uudempi tutkimus on käsitellyt muuan muassa pyrkimyksiä parantaa sen toimivuutta varjojen (Chandraker, Agarwal ja Kriegman 2007) ja spekulaaristen heijastuksien kanssa (Chung ja Jia 2008).

2.3.3 Pinnan muodon määrittäminen

Tähän mennessä on saatu määritettyä kuvatun kohteen pinnan lokaali asento kaikissa kuvan pisteissä. Näiden avulla myös pinnan kolmiulotteinen muoto eli kunkin pisteen kolmiulotteinen karteesinen koordinaatti (x,y,z) saadaan määritettyä.

Luvussa 2.3.1 esitetyn normaalivektorin $\boldsymbol{n}$ määritelmän mukaisesti

$$\boldsymbol{n}_x = -\frac{p}{|\boldsymbol{n}|}, \qquad \boldsymbol{n}_y = -\frac{q}{|\boldsymbol{n}|}, \qquad \boldsymbol{n}_z = \frac{1}{|\boldsymbol{n}|}, \tag{2.40}$$

mistä seuraa, että

$$p = -\boldsymbol{n}_x|\boldsymbol{n}| = -\frac{\boldsymbol{n}_x}{\boldsymbol{n}_z}, \qquad q = -\boldsymbol{n}_y|\boldsymbol{n}| = -\frac{\boldsymbol{n}_y}{\boldsymbol{n}_z}, \tag{2.41}$$

eli osittaisderivaattojen avulla ilmaistuna

$$\frac{\delta z}{\delta x} = -\frac{\boldsymbol{n}_x}{\boldsymbol{n}_z}, \qquad \frac{\delta z}{\delta y} = -\frac{\boldsymbol{n}_y}{\boldsymbol{n}_z}. \tag{2.42}$$

Kun tämä osittaisdifferentiaaliyhtälöpari ratkaistaan joillakin reunaehdoilla, saadaan ratkaisuna pinnan korkeus paikan funktiona $z(x,y)$.

Koska kuvat muodostuvat diskreeteistä pisteistä, voidaan osittaisderivaatat kirjoittaa mitattavan suuruisten differenssien osamäärinä seuraavasti:

$$\frac{\Delta z_x}{\Delta x} = -\frac{\boldsymbol{n}_x}{\boldsymbol{n}_z}, \qquad \frac{\Delta z_y}{\Delta y} = -\frac{\boldsymbol{n}_y}{\boldsymbol{n}_z}. \tag{2.43}$$

Valitsemalla koordinaatisto siten, että kahden vierekkäisen kuvapisteen välinen etäisyys sekä x- että y-suunnassa on yksi ($\Delta x = \Delta y = 1$), saadaan

$$\Delta z_x = z_{x+1,y} - z_{x,y} = -\frac{\boldsymbol{n}_x}{\boldsymbol{n}_z}, \qquad \Delta z_y = z_{x,y+1} - z_{x,y} = -\frac{\boldsymbol{n}_y}{\boldsymbol{n}_z}. \tag{2.44}$$

Soveltamalla näitä yhtälöitä jokaiseen kuvapisteeseen syntyy ylimääritelty lineaarinen yhtälöryhmä, jossa on $2N$ yhtälöä ja N tuntematonta, joissa N on kuvapisteiden lukumäärä. Yhtälöryhmä saadaan ratkaistua pienimmän neliösumman menetelmällä, ja ratkaisuna on N-pituinen vektori, joka muodostuu kuvapisteiden syvyysarvoista z.

On huomioitava, että tuloksena saadut pisteet (x, y, z) ovat kuvakoordinaatistossa, jossa ykkösen pituinen siirtymä joka akselilla vastaa vierekkäisten kuvapisteiden välistä etäisyyttä. Jos kameran linssin vaaka- sekä pystykuvakulmien suuruudet ja kameran etäisyys kohteesta tunnetaan, voidaan pisteet muuntaa reaalimaailmaa vastaavaan koordinaatistoon (Wu 1978).

Jälleen on syytä huomioida, että normaalivektorin aallonpituusriippuvuuden myötä myös syvyysarvot z ovat aallonpituuden funktioita.

2.4 Spektroskopia

Sähkömagneettisen säteilyn ja materian välisen vuorovaikutuksen tutkimista kutsutaan *spektroskopiaksi*. Spektroskopiassa tutkittavan kohteen säteilemää, heijastamaa tai absorboimaa sähkömagneettista säteilyä tarkastellaan säteilyn aallonpituuden tai taajuuden funktiona. Termillä viitataan yleensä kokeellisiin spektroskopisiin mittausmenetelmiin, joissa sähkömagneettisesta säteilystä eritellään sen eri aallonpituudet tai taajuudet eli tuotetaan säteilyn spektri.

Spektrin tuottamiseen ja mittaamiseen käytettävät laitteet ovat *spektrometrejä*. *Optisilla* spektrometreillä mitataan erityisesti valon voimakkuutta aallonpituuden tai taajuuden funktiona. Optisissa spektrometreissä valon eri aallonpituudet tai taajuudet voidaan eritellä käyttäen prismaa ja valon taittumista tai hilaa ja diffraktiota. Optiset spektrometrit voidaan toteuttaa myös käyttäen *interferometriä*, joka hyödyntää usean valoaallon superpositioperiaatetta erottaen eri aallonpituudet konstruktiivisten ja destruktiivisten interferenssien avulla (Harvey 2012).

2.4.1 Spektrikuvantaminen

Spektrikuvantaminen on spektroskopian ala, jossa spektroskopia yhdistetään valokuvaukseen. Spektrikuvantamisessa kuvan jokaiseen kuvapisteeseen kerätään sähkömagneettisen säteilyn spektri, joka on normaalin kameran keräämää spektriä laajempi ja huomattavasti erottelukykyisempi (Garini, Young ja McNamara 2006). Spektrikuvantamisella on paljon eri sovelluskohteita, joista esimerkkeinä tähtitiede (Pasquini ym. 2002), kaukokartoitus (Teke ym. 2013) ja lääketiede (Lu ja Fei 2014).

Spektrikuvantaminen jaetaan kerättävän spektrin leveyden ja jatkuvuuden sekä spektraalisen erottelykyvyn perusteella alaluokkiin, joita ovat mm. *multispektrikuvantaminen* ja *hyperspektrikuvantaminen*.

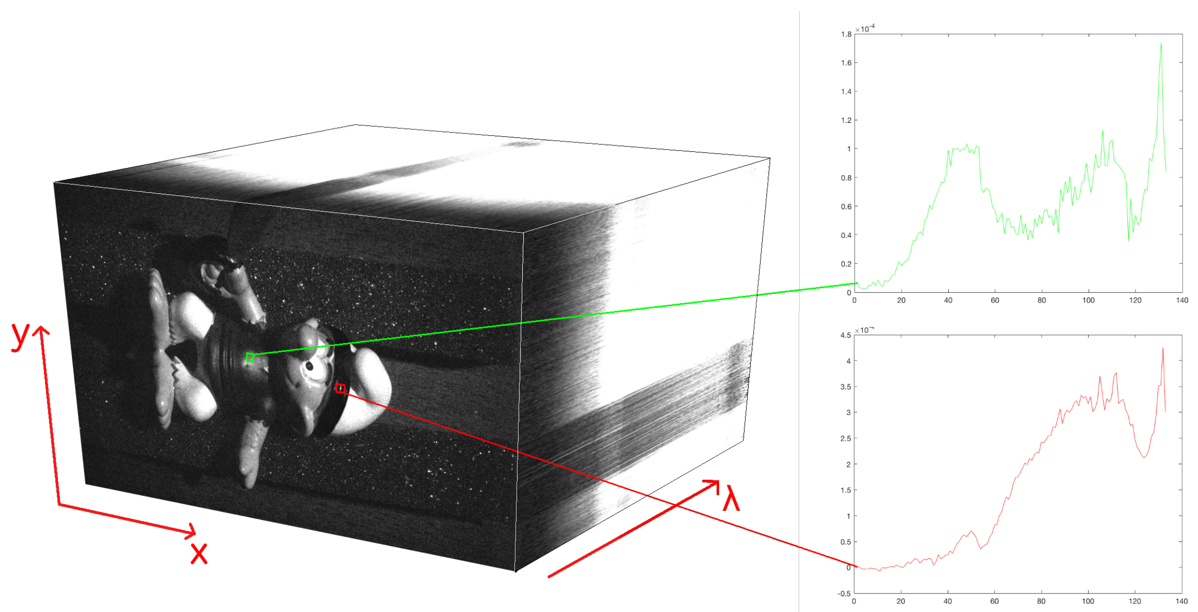
Multispektrikuvantamisessa sähkömagneettisen säteilyn spektri jaetaan kapeisiin aallonpituusalueisiin ja kuvapisteisiin kerätään useamman kuin kolmen eri aallonpituusalueen säteilyn voimakkuus. Multispektrikuvantamisessa kerättävät aallonpituusalueet voivat olla erillään toisistaan jättäen väliinsä keräämättömiä aallonpituusalueita, jolloin kerättävä spektri on epäjatkuva. Aallonpituusalueet voivat olla myös näkyvän valon alueen ulkopuolella, ja multispektrikuvat usein sisältävätkin infrapuna-alueen aallonpituuksia (Coffey 2012).

2.4.2 Hyperspektrikuvantaminen

Hyperspektrikuvantamisessa kuvan jokaiseen kuvapisteeseen kerätään joltain aallonpituusväliltä koko valon jatkuva spektri suurella spektraalisella erottelukyvyllä.

Tavallisen kameran kerätessä kolme leveää aallonpituusaluetta (sininen, vihreä ja punainen) hypespektrikameran kuvat voivat muodostua sadoista kapeista aallonpituusalueista. Hyperspektrikuvien voidaan ajatella olevan kolmiulotteisia kuutioita, jotka muodostuvat useista kaksiulotteisista, eri aallonpituusalueita vastaavista kuvista. Tällaisen kuution kuvapisteisiin viitataan kolmiulotteisilla koordinaateilla (x, y, λ), joissa x ja y ovat kuvan avaruudelliset koordinaatit ja λ aallonpituuskoordinaatti, jonka yksi arvo käsittää jonkin kapean aallonpituusalueen (Smith 2012). Hyperspektrikuvien voidaan ajatella olevan myös kaksiulotteisia kuvia, joiden kuvapisteiden arvot ovat m-pituisia vektoreita, missä m on kerättyjen aallonpituusalueiden lukumäärä. Kuvio 2 pyrkii havainnollistamaan molempia lähestymistapoja.

Hyperspektrikuvantaminen eroaa multispektrikuvantamisesta kapeampien aallonpituusalueiden ja aallonpituusalueiden suuremman lukumäärän vuoksi. Näin ollen hyperspektrikuvantamisen spektraalinen erottelukyky on suurempi. Tämän lisäksi hyperspektrikuvantaminen eroaa merkittävästi multispektrikuvantamisesta siten, että hyperspektrikuvantamalla kerätyt spektrit ovat aina jatkuvia. Toisin sanoen kerätyt aallonpituusalueet ovat yhtenäisiä (todellisuudessa lisäksi osittain päällekkäisiä), jolloin niiden väliin ei jää keräämättömiä aallonpituusalueita (Govender ym. 2008). Kuviossa 3 on esitettynä tavallisen valokuvan, multispekt-

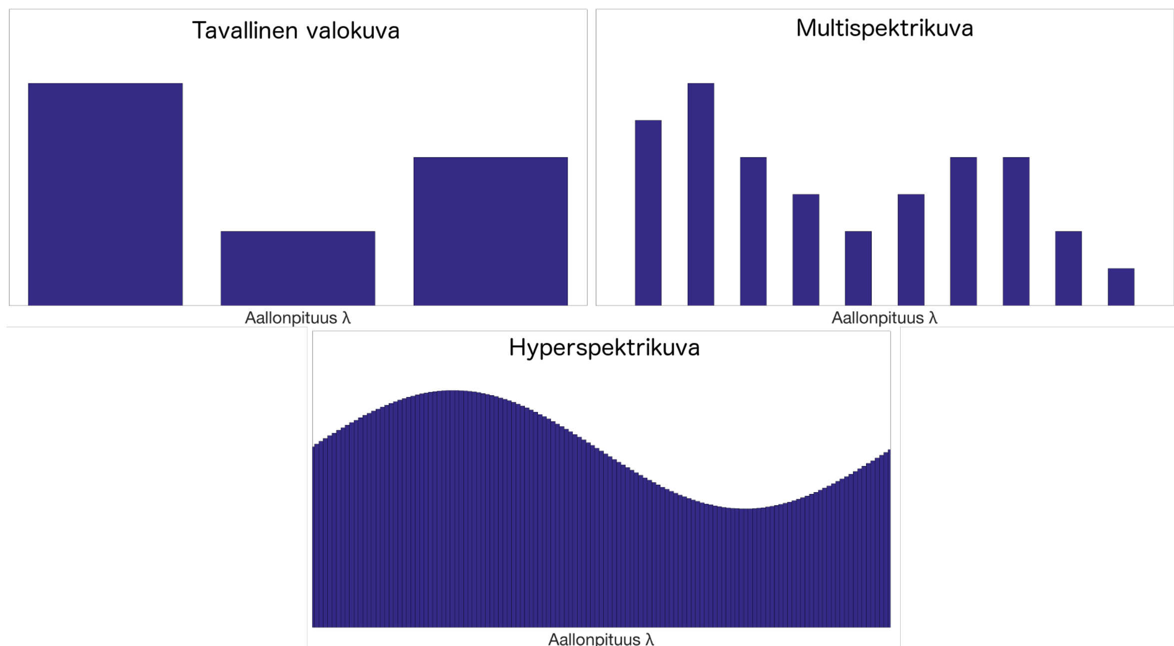


Kuvio 2. Hyperspektrikuvaa vastaava kuutio ja kahden kuvapisteen arvot aallonpituuden funktiona.

rikuvan ja hyperspektrikuvan yhteen kuvapisteeseen kerättyjen aallonpituusalueiden erot.

Eri materiaaleilla on erilaiset niille ominaiset spektrit eli niin sanotut *spektraaliset sormenjäljet*. Näin ollen hyperspektrikuvien avulla on mahdollista tunnistaa kuvatun kohteen materiaali vertaamalla hyperspektrikameran keräämää spektriä eri materiaalien spektraalisiin sormenjälkiin. Hyperspektrikuvantaminen kehitettiinkin alunperin kaivosteollisuuden ja geologian tarpeisiin (Yuen ja Richardson 2010).

Hyperspektrikameroiden viimeaikainen kehittyminen on mahdollistanut hyperspektraalisen kaukokartoituksen tekemisen myös miehittämättömien ilma-alusten avulla (Saari ym. 2013). Tekniikan kehitys on lisäksi mahdollistanut hyperspektrikuvantamisen leviämisen laajalti myös muihin sovelluskohteisiin kuten rikospaikkatutkintaan (Kuula ym. 2012), taidemaalausten aitouden todentamiseen (Polak ym. 2017) ja ihosyövän diagnosointiin (Zheludev ym. 2015).



Kuvio 3. Eri kuvantamismenetelmien spektrien erot.

2.4.3 Hyperspektrikameroista

Hyperspektrikameroiden toiminta perustuu yleensä useiden kaksiulotteisten osakuvien ottamiseen jollakin menetelmällä ja osakuvien yhdistämiseen lopulliseksi kolmiulotteiseksi hyperspektrikuvaksi. Hyperspektrikamerat jakautuvat toimintaperiaatteeltaan neljään eri kategoriaan sen perusteella mistä datasta niiden kaksiulotteiset osakuvat muodostuvat. Kategoriat ovat avaruudellisspektraalisesti (spatiospectral) skannaavat, avaruudellisesti skannaavat, skannaamattomat ja spektraalisesti skannaavat hyperspektrikamerat.

Avaruudellisspektraalisesti skannaavat hyperspektrikamerat kuvaavat useita kaksiulotteisia kuvia ja yhdistävät ne jälkikäteen. Yhden tällaisen kaksiulotteisen osakuvan jokainen vaakarivi vastaa eri aallonpituusaluetta, jolloin osakuvan aallonpituus on avaruudellisen y-koordinaatin funktio ($\lambda = \lambda(y)$). Koko kolmiulotteisen hyperspektrikuvan muodostaminen vaatii usean osakuvan ottamista, kameran siirtämistä niiden välillä ja osakuvien yhdistämistä ohjelmallisesti. Avaruudellisspektraalisesti skannaavalla hyperspektikameralla kuvattuna kuvion 2 kuutio muodostuu $y\lambda$-suuntaisen kyljen diagonaalisesti leikkaavista osittain päällekkäin olevista suorakulmioista.

Avaruudellisesti skannaavat kamerat kuvaavat yhden vaakarivin kuvapisteet vuorollaan keräten kyseisten kuvapisteiden koko spektrin kerralla. Yhden tällaisen kaksiulotteisen osakuvan kuvapisteiden koordinaatit ovat siis (x, λ). Koko kolmiulotteinen hyperspektrikuva muodostetaan kuvaamalla useita vaakarivejä ja yhdistämällä ne. Avaruudellisesti skannaavalla hyperspektrikameralla kuvattuna kuvion 2 kuutio muodostuu y-akselin suunnassa päällekkäin olevista kaksiulotteisista suorakulmioista. Eri vaakarivit kuvataan eri aikaan ja niiden välillä kameraa siirretään. Avaruudellisesti skannaavat hyperspektrikamerat ovat yleisiä kaukokartoituksessa, koska kaukokartoituksessa kameran liikuttaminen on mielekästä.

Skannaamattomat hyperspektrikamerat keräävät kaksiulotteiselle valoherkälle kennolle kaiken kolmiulotteisen (x, y, λ) hyperspektrikuvan muodostamiseen tarvittavan datan kerralla. Yksi tällainen kaksiulotteinen datajoukko voidaan tulkita hyperspektrikuvaa vastaavan kuution perspektiiviprojektiona, josta kolmiulotteinen rakenne voidaan rekonstruoida tietokoneen avulla.

Spektraalisesti skannaavat hyperspektrikamerat kuvaavat useita kaksiulotteisia monokromaattisia kuvia, jotka yhdistetään kolmiulotteiseksi hyperspektrikuvaksi. Yksi kaksiulotteinen osakuva sisältää molemmat avaruudelliset vapausasteet x ja y aallonpituuskoordinaatin λ ollessa vakio. Spektraalisesti skannaavalla hyperspektrikameralla kuvattuna kuvion 2 kuutio muodostuu λ-akselin suunnassa vierekkäin olevista suorakulmioista. Spektraalisesti skannaavat kamerat pidetään koko spektrin keräämisen ajan paikoillaan. Eri osakuvien ottamisen välillä aallonpituussuodatinta säädetään tai se vaihdetaan kokonaan (Garini, Young ja McNamara 2006).

Teknologian tutkimuskeskus VTT Oy on kehittänyt spektraalisesti skannaavan hyperspektrikameran, jonka toiminta perustuu pietsosähköisellä aktuaattorilla säädettävään Fabry-Perot-interferometriin. Fabry-Perot-interferometrit ovat säädettäviä aallonpituussuodattimia, joita voidaan valmistaa erittäin pienikokoisina ja kevyinä. Näin ollen ne ovat mahdollistaneet hyperspektrikameran miniatyrisoinnin käsikäyttöiseksi. Teknologia on myös kustannustehokas, mikä mahdollistaa hyperspektrikameroiden sarjavalmistuksen.

Fabry-Perot-interferometri muodostuu kahdesta samansuuntaisesta, valoa osittain läpäisevästä peilistä. Peilien heijastavat pinnat ovat vastakkain ja niiden välissä on kapea ilmarako.

Kun peilejä valaistaan ulkopuolelta, osa valosta pääsee kulkeutumaan suoraan peilien läpi ja osa jää heijastelemaan peilien väliin kunnes se jonkin heijastuskertojen määrän jälkeen läpäisee jälkimmäisen peilin. Peilien välillä heijastelleet valoaallot interferoivat peilien välistä aiemmin poistuneiden valoaaltojen kanssa siten, että tiettyjen aallonpituuksien aallot vahvistavat toisiaan, kun taas muiden aallonpituuksien aallot vaimentavat. Näin ollen peilien läpi tulevan valon spektriin muodostuu vahvistavien interferenssien aallonpituuksien kohdalle piikkejä ja Fabry-Perot-interferometri toimii aallonpituussuodattimena. Suodattimen läpi pääsevien piikkien aallonpituudet riippuvat peilien välisen ilmaraon suuruudesta.

VTT:n kehittämässä kamerassa peilien välisen raon suuruutta säädetään erittäin nopeasti ja tarkasti pietsosähköisen aktuaattorin avulla (Rissanen ja Saari 2014). Säätäminen voidaan tehdä niin nopeasti, että kaksiulotteiset osakuvat voidaan ottaa lähes välittömästi toistensa jälkeen. Nopeutta rajoittavina tekijöinä onkin kameran valoherkän kennon nopeus sekä valotusaika. Raon suuruutta voidaan säätää siten, että suodattimen läpi tulevan spektrin maksimikohdat pystytään määräämään alle yhden nanometrin tarkkuudella.

Fabry-Perot-interferometrin läpi pääsevien piikkien maksimikohtien aallonpituuksille pätee likimääräisesti

$$\lambda_n = \frac{2d}{n}, \tag{2.45}$$

missä d on peilien välisen raon suuruus ja $n \in \mathbb{N} = \{1, 2, 3, \ldots\}$. Raon suuruus voidaan säätää siten, että tarkasteltavalle aallonpituusvälille osuu aina vain yhdestä kolmeen suodattimen läpi pääsevän spektrin piikkiä. Kamera kerää suodattimen läpi tulevan spektrin Bayer-suodattimella varustetulla RGB-valoherkällä kennolla. Näin ollen spektrin kaikkien piikkien (1-3 kpl) intensiteetit voidaan määrittää, koska RGB-kennon punaista, vihreää ja sinistä vastaavilla kuvapisteillä on eri spektraalinen herkkyysjakauma. Yhdellä Fabry-Perot-interferometrin raon suuruudella saadaan siis kuvattua yhdestä kolmeen aallonpituusaluetta eli hyperspektrikuvan osakuvaa kerralla (Saari ym. 2013).

2.5 Tietokonegrafiikka

Tietokonegrafiikka on laaja tietotekniikan ala, joka käsittelee digitaalisten kuvien tuottamista ja muokkaamista ohjelmallisesti tietokoneen avulla. Se käsittää useita aihepiirejä graafisista

käyttöliittymistä ja kuvankäsittelystä aina tietokoneanimaatioon sekä konenäköön saakka.

Tietokonegrafiikka voidaan jakaa karkeasti kahteen eri osa-alueeseen: kaksi- ja kolmiulotteiseen tietokonegrafiikkaan. Kolmiulotteinen tietokonegrafiikka on yleistynyt viime aikoina huomattavasti teknologian kehittymisen myötä (Hearn ja Baker 2003, 3-33).

2.5.1 Kolmiulotteinen tietokonegrafiikka

Kolmiulotteisessa tietokonegrafiikassa kolmiulotteisesta geometriadatasta eli *mallista* luodaan kaksiulotteinen kuva käyttäen kolmiulotteiseksi *renderöinniksi* kutsuttua prosessia. Luotuja kuvia voidaan näyttää reaaliajassa tai tallentaa myöhempää esittämistä varten. Kolmiulotteista grafiikkaa käytetään laajalti eri sovelluskohteissa, joita ovat muuan muassa tietokoneavusteinen suunnittelu, tietokonepelit ja elokuvat.

Kolmiulotteisessa tietokonegrafiikassa kuvan luominen muodostuu pääosin kolmesta eri vaiheesta: mallintamisesta, sijoittelusta ja renderöinnistä. Mallintamisessa jostakin kohteesta luodaan malli, joka koostuu kolmiulotteisesta geometriadatasta ja materiaalia mallintavista tekstuureista. Mallin voi luoda artisti, tai se voidaan luoda automaattisesti käyttäen esimerkiksi kolmiulotteista skanneria tai proseduaalisesti mallintavaa tietokoneohjelmaa. Sijoittelussa mallit sijoitetaan renderöitävään näkymään halutussa koossa ja haluttuihin paikkoihin sekä asentoihin. Renderöinnissä kolmiulotteisista malleista luodaan tietokoneen avulla kaksiulotteinen kuva. Renderöintimenetelmiä on useita erilaisia, ja ne voidaan jakaa niiden suoritusaikojen perusteella kahteen pääluokkaan: reaaliaikaisiin ja ei-reaaliaikaisiin renderöintimenetelmiin.

Ei-reaaliaikaisia menetelmiä käyttäen voi yhden kuvan renderöintiin kulua useita päiviä. Ei-reaaliaikaisten menetelmien tuottama kuvanlaatu on yleisesti ottaen kuitenkin huomattavasti korkeampi kuin reaaliaikaisten menetelmien. Ei-reaaliaikaisiin menetelmiin lukeutuu muun muassa *säteenseuranta*. Ei-reaaliaikaisten menetelmien renderöimät kuvat tallennetaan muistiin, josta ne voidaan myöhemmin ladata ja näyttää nopeasti.

Interaktiivisissa kolmiulotteisen tietokonegrafiikan sovelluskohteissa kuten tietokonepeleissä ja simulaatioissa on välttämätöntä käyttää reaaliaikaisia renderöintimenetelmiä. Niissä kuvaa saatetaan päivittää 60 kertaa sekunnissa, jolloin yhden kuvan renderöintiin on käy-

tettävissä korkeintaan noin 16,7 millisekuntia. Vaikka reaaliaikaiset renderöintimenetelmät eivät kuvanlaadullisesti yllä ei-reaaliaikaisten menetelmien tasolle, on niiden tuottamien kuvien fotorealistisuus kasvanut viime aikoina huomattavasti. Tähän on itse menetelmien kehittymisen lisäksi syynä niille osoitetun laitteistotuen kehittyminen. Yleisin nykyisin käytetty reaaliaikainen renderöintimenetelmä on *rasterointi*. Menetelmän eri vaiheita ja niiden järjestystä kuvaa *grafiikkaliukuhihnaksi* (graphics pipeline) kutsuttu malli, joka on nykyaikaisissa grafiikkasuorittimissa toteutettu laitteistotasolla. Grafiikkaliukuhihnan kolme päävaihetta ovat: sovellus-, geometria- ja rasterointivaihe.

Grafiikkasovellus, esimerkiksi tietokonepeli, suorittaa grafiikkaliukuhihnan sovellusvaiheen. Se suoritetaan keskussuorittimella, ja kehittäjällä on sen toteutuksen suhteen täysi kontrolli. Sovellusvaiheen vastuuna on suorittaa kolmiulotteisten mallien sijoittelu ja päättää mitä renderöidään. Sovellusvaiheen viimeinen ja tärkein tehtävä on syöttää renderöitävät geometriat geometriavaiheelle. Syötettävät geometriat muodostuvat *renderöinti primitiiveistä*, jotka ovat pisteitä, viivoja tai yleensä kolmioita.

Geometriavaihe suorittaa syötteenä saatujen primitiivien kulmapisteille eli *vertekseille* erilaisia operaatioita. Geometriavaihe jakaantuu itsessään viiteen pienempään alivaiheiseen: malli- ja näkymämuunnos-, verteksivarjostus-, projektio-, leikkaus- sekä näyttökuvaus (screen mapping) -vaiheeseen. Malli- ja näkymämuunnosvaiheessa mallit sijoitellaan haluttuihin paikkoihin ja asentoihin muuntamalla niiden verteksien koordinaatit mallimuunnoksella mallin lokaalista koordinaatistosta maailman koordinaatistoon. Tämän jälkeen verteksit muunnetaan näkymämuunnoksella maailman koordinaatistosta näkymäkoordinaatistoon, joka vastaa virtuaalisen kameran, sen hetkisen sijainnin ja orientaation mukaista näkymää. Verteksivarjostusvaiheessa on mahdollista muokata verplekseihin liittyviä tietoja kuten värejä, tekstuurikoordinaatteja ja normaalivektoreita, jonka jälkeen ne lähetetään rasterointivaiheelle. Projektiovaiheessa verteksien koordinaatit projisoidaan kolmesta ulottuvuudesta kahteen. Projektio on yleensä suora ortogonaaliprojektio tai perspektiivin huomioon ottava perspektiiviprojektio. Leikkausvaiheessa kuvan ulkopuolelle jäävät primitiivit ja verteksit poistetaan. Näyttökuvausvaiheessa primitiivien koordinaatit skaalataan renderöitävän kuvan kuvakoordinaatteihin.

Rasterointivaihe määrittää kuvakoordinaateissa olevien geometrioiden peittämät kuvapisteet

ja niiden värit. Vaihe jakaantuu neljään alivaiheeseen: kolmiojärjestely, kolmioiden läpikäynti, kuvapistevarjostus ja yhdistäminen. Kolmiojärjestelyvaiheessa kolmiopinnoille lasketaan gradientit ja muita tietoja, joita käytetään myöhemmin vertekseihin liitettyjen tietojen kuvapistekohtaisessa interpoloinnissa. Kolmioiden läpikäyntivaiheessa kaikki kuvapisteet käydään läpi ja tarkistetaan, onko niiden keskipiste jonkin kolmion sisällä. Kolmion kolmeen verteksiin liitetyt tiedot interpoloidaan jokaiselle kolmion sisään jäävälle kuvapisteelle. Kuvapistevarjostusvaiheessa jokaisen renderöitävän kuvapisteen väri määritetään interpoloituja tietoja käyttäen. Kuvapisteiden värit kirjoitetaan *väripuskuriin*, ja yhdistämisvaiheen tehtävänä on yhdistää kuvapistevarjostusvaiheen tuloksena syntyneet värit väripuskurissa jo valmiina olevien värien kanssa. Yhdistämisvaiheen tehtävänä on myös verrata sillä hetkellä renderöitävän ja väripuskurissa jo valmiina olevan kuvapisteen syvyysarvoja ja sen perusteella määrittää renderöitävän kuvapisteen näkyvyys (Akenine-Möller, Haines ja Hoffman 2008, 11-25).

2.5.2 Varjostinohjelmat

Osa edellä esitellyistä grafiikkaliukuhihnan vaiheista on toteutettu laitteistotasolla siten, että kehittäjällä ei ole niihin ollenkaan kontrollia tai kehittäjä pystyy kontrolloimaan niitä ainoastaan ennalta määrättyjen konfiguraatiovaihtoehtojen avulla. Moderneissa grafiikkasuorittimissa malli- ja näkymämuunnos-, verteksivarjostus- sekä kuvapistevarjostusvaiheet ovat kuitenkin täysin kehittäjän vapaasti ohjelmoitavissa.

Malli- ja näkymämuunnos- sekä verteksivarjostusvaiheita ohjelmoidaan grafiikkasuorittimella suoritettavan *verteksivarjostinohjelman* avulla. Muiden kun ortogonaaliprojektioiden tapauksessa myös osa projektiovaiheesta suoritetaan verteksivarjostinohjelmassa. Verteksivarjostinohjelma suoritetaan jokaiselle renderöitävälle verteksille ja yleensä usealle verteksille yhtäaikaisesti. Se saa syötteenä yhden verteksin paikkakoordinaatit ja muut verteksiin mahdollisesti liitetyt tiedot. Yksi verteksivarjostinohjelman suoritusinstanssi ei voi saada tietoja muista verteksistä kuin sen itsensä käsittelemästä. Sovellusvaihe voi kuitenkin syöttää verteksivarjostusohjelmille vertekseistä riippumattomia vakiotietoja, jotka on jaettu kaikkien eri suoritusinstanssien kesken. Tällaisia vakiotietoja voivat olla esimerkiksi koordinaatistomuunnosmatriisit. Verteksivarjostinohjelman on aina palautettava vähintään verteksin paik-

kakoordinaatit, mutta se voi palauttaa myös muita tietoja. Palautetut tiedot interpoloidaan rasterointivaiheessa jokaiselle kuvapisteelle.

Modernien grafiikkasuorittimien ohjelmoitava kuvapistevarjostusvaihe toteutetaan *kuvapistevarjostinohjelman* avulla. Kuvapistevarjostinohjelma suoritetaan jokaiselle kuvapisteelle ja yleensä usealle kuvapisteelle yhtäaikaisesti. Kuvapistevarjostinohjelma saa syötteenään verteksivarjostinohjelman palauttamat ja rasterointivaiheen kuvapisteelle interpoloimat tiedot. Kuvapistevarjostinohjelma ei voi saada tietoja muista kuvapisteistä kun sen itse käsittelemästä. Kuvapistevarjostinohjelmilla on kuitenkin mahdollisuus lukea sovellusvaiheen niille syöttämiä kuvapisteistä riippumattomia jaettuja vakiotietoja. Tällaisia vakiotietoja voivat olla esimerkiksi valojen sijainnit ja luvussa 2.5.3 esiteltävät *tekstuurit*. Kuvapistevarjostinohjelma palauttaa yleensä kuvapisteen värin, jonka yhdistämisvaihe voi kirjoittaa väripuskuriin (Bailey ja Cunningham 2009, 39-49).

Kuvapistevarjostinohjelman on usein tarkoituksena simuloida virtuaalisten valonlähteiden säteilemien irradianssien heijastuminen renderöitävän kohteen pinnalta ja laskea kuvapisteeseen heijastuneen radianssin arvo. Esimerkiksi jos kuvapistevarjostinohjelma saa syötteenä normaalivektorin, albedon, valonlähteen sijainnin sekä irradianssi ja renderöitävän kohteen materiaali heijastaa Lambertin pinnan mukaisesti, niin kuvapisteeseen saapuva radianssi voidaan laskea yhtälöstä 2.25. Koska irradianssi, albedo ja heijastunut radianssi ovat aallonpituuden funktioita, ne ovat tavallisesti ilmaistu kolmekomponenttisina vektoreina, joissa yksi komponenteista vastaa punaista, yksi vihreää ja yksi sinistä aallonpituusaluetta (Akenine-Möller, Haines ja Hoffman 2008, 110-116).

2.5.3 Teksturointi

Tekstuurit ovat pohjimmiltaan kaksiulotteisia datataulukkoja. Ne voivat olla väridataa sisältäviä kuvia tai muuta lokaalia dataa sisältäviä *karttoja*. Ne muodostavat renderöitävien mallien materiaalit. Tekstuurit liitetään renderöitäviin geometrioihin yleensä tekstuurikoordinaattien avulla siten, että mallinnusvaiheessa jokaiselle verteksille annetaan omat kaksiulotteiset tekstuurikoordinaatit. Rasterointivaihe interpoloi tekstuurikoordinaatit kuvapisteille, jonka jälkeen kuvapistevarjostinvaihe voi niiden avulla lukea tekstuurista juuri sillä hetkellä

prosessoitavaan kuvapisteeseen liittyvän elementin.

Yksi yleisimmin käytetyistä tekstuureista on *albedokartta*, jonka jokainen elementti sisältää mallin vastaavan kohdan lokaalin albedon, yleensä kolmekomponenttisena, kolmea eri aallonpituusaluetta vastaavana vektorina. Albedokartta on käytännössä siis mallin värit sisältävä kuva.

Kun mallin lokaaleja pinnan muotoja halutaan valaistuksen yhteydessä kuvata tarkemmalla tasolla kuin mitä vertekseihin liitetyt sijainnit ja normaalivektorit mahdollistavat, malliin voidaan liittää *normaalikartaksi* kutsuttu tekstuuri. Normaalikartan jokainen elementti sisältää mallin vastaavan kohdan kolmekomponenttisen normaalivektorin.

Toinen mallin lokaalien muotojen tarkentamisen mahdollistava tekstuuri on *syvyyskartta*. Syvyyskartan elementit sisältävät tiedon mallin pinnan lokaalista syvyydestä suhteessa renderöintiprimitiivin, esimerkiksi kolmion määräämään tasoon. Elementtien arvot ovat näin ollen skalaareja. Syvyysarvoja voidaan hyödyntää pisteen sijainnin ja näin ollen myös esimerkiksi valon suunnan tarkemmassa arvioinnissa (Akenine-Möller, Haines ja Hoffman 2008, 147-183).

2.5.4 OpenGL

OpenGL (Open Graphics Library) on ohjelmointikieli- ja alustariippumaton grafiikkaohjelmointirajapinta, jonka avulla kolmiulotteisista malleista voidaan renderöidä kaksiulotteisia kuvia käyttäen grafiikkasuorittimen laitteistotason tukea. Se mahdollistaa grafiikkaliukuhihnan sovellusvaiheen ja muiden ohjelmoitavien vaiheiden toteuttamisen sekä laitteistotasolla toteutettujen vaiheiden konfiguroimisen (Khronos Group 2016a). OpenGL on tilakone, joka muodostuu joukosta funktioita, joilla voidaan muuttaa sen tilaa. OpenGL tarjoaa ainoastaan rajapinnan määrittelyn. Toteutuksen toimittaa grafiikkasuoritinvalmistaja, käyttöjärjestelmä tai jokin muu kolmas osapuoli. OpenGL on käytössä laajalti eri aloilla kuten tieteellisessä visualisoinnissa, tietokoneavusteisessa suunnittelussa ja tietokonepeleissä (Puhakka 2008, 355-357). Rajapinnan ensimmäinen versio, versio 1.0 julkaistiin vuonna 1992 (Khronos Group 2016a) ja viimeisin, versio 4.5, vuonna 2014 (Khronos Group 2016b).

OpenGL:n ensimmäisissä versioissa grafiikkaliukuhihnan geometria- ja rasterointivaiheiden

ohjelmoitavat alivaiheet oli toteutettava matalan tason ARB assembly -ohjelmointikielellä. Kuitenkin vuonna 2004 julkaistusta versiosta 2.0 lähtien OpenGL on mahdollistanut verteksi- ja kuvapistevarjostinohjelmien toteuttamisen C-ohjelmointikieleen perustuvalla korkean tason ohjelmointikielellä GLSL (OpenGL Shading Language). Se tukee lähes kaikkia C-kielen kontrollirakenteita ja operaattoreita paitsi osoittimia ja rekursiota. GLSL-ohjelmat käännetään konekielelle OpenGL-rajapinnan ja grafiikkasuoritinvalmistajien kehittämien ajureiden avulla (Bailey ja Cunningham 2009, 25-31). GLSL:n viimeisin versio on vuonna 2014 julkaistu 4.50 (Khronos Group 2016b).

2.6 GPGPU ja numeerinen lineaarialgebra

2.6.1 GPGPU

Koska varjostinohjelmia on tarve suorittaa useille vertekseille ja kuvapisteille yhtäaikaisesti, on nykyaikaisten grafiikkasuorittimien kehitys keskittynyt laskennan rinnakkaistamiseen. Verteksi- ja kuvapistevarjostinohjelmille oli aiemmin käytössä omat erilliset suoritinyksikkönsä mutta nykyaikaisissa grafiikkasuorittimissa molemmat suoritetaan käyttäen samoja suoritinytimiä. Uusimmissa grafiikkasuorittimissa tällaisia ohjelmoitavia suoritinytimiä voi olla satoja tai jopa tuhansia.

GPGPU (General-purpose computing on graphics processing units) on ohjelmistokehitysmenetelmä, jossa grafiikkasuorittimen erittäin suurta suoritinytimien lukumäärää hyödynnetään grafiikan renderöinnin sijaan yleiseen, normaalisti keskussuorittimella suoritettavaan laskentaan. Vaikka keskussuorittimien kellotaajuus, sarjallisten operaatioiden suorituskyky ja toimintojen määrä on yleensä grafiikkasuorittimia korkeampi, muodostuvat keskussuorittimet vain muutamista peräkkäisiin operaatioihin optimoiduista suoritinytimistä (Ghorpade ym. 2012). Näin ollen tietynlaisten, hyvin rinnakkaistuvien ongelmien ratkaiseminen grafiikkasuorittimella voi keskussuorittimeen verrattuna olla kymmeniä tai jopa satoja kertoja nopeampaa (Pevar ym. 2015).

GPGPU:n rinnakkaislaskenta toteutetaan käyttäen *virtausprosessointi* (stream processing) -ohjelmointiparadigmaa, joka mahdollistaa useiden suoritinytimien yhtäaikaisen käytön, ilman eksplisiittistä allokoinnin, synkronoinnin ja suoritinytimien välisen kommunikoinnin

hallintaa. Virtausprosessoinnissa suoritettavan rinnakkaislaskennan mahdollisuudet on rajoitettu siten, että se yksinkertaistaa siihen tarvittavaa ohjelmistoa ja laitteistoa. Tyypillisesti virtausprosessoinnissa suoritetaan jonkin datajoukon jokaiselle elementille sama joukko operaatioita. Tämä on mahdollista moderneilla grafiikkasuorittimilla, koska ne kuuluvat rinnakkaistietokoneiden *SIMD* (single instruction, multiple data) -luokkaan. SIMD-luokan tietokoneet suorittavat samat operaatiot useille eri data-alkioille samanaikaisesti. Näin ollen ne toteuttavat datatason rinnakkaisuuden, jossa yhdellä ajanhetkellä käsitellään useita data-alkioita, mutta suoritetaan vain yhtä käskyä (Kapasi ym. 2003).

GPGPU:ta sovelletaan useilla eri aloilla, joihin lukeutuu muun muassa fysikaalinen simulointi (Unnc, Inoue ja Asar 2009) ja konenäkö (Varnavas ym. 2010).

2.6.2 OpenCL

GPGPU-menetelmää sovellettiin alussa käyttäen grafiikkarenderöintiin tarkoitettuja ohjelmointirajapintoja ja tapoja kuten verteksi- sekä kuvapistevarjostinohjelmia. Menetelmän avulla ratkaistavat ongelmat oli muunnettava ensin muotoon, joka oli esitettävissä grafiikkarenderöinnissä käytettävien käsitteiden avulla. Näin ollen GPGPU on ollut alusta lähtien yleinen erityisesti matriiseihin ja vektoreihin liittyvissä ongelmissa, joita grafiikkasuorittimet tukevat natiivisti.

GPGPU:n yleistyttyä ja paremman käyttörajapinnan tarpeen kasvettua menetelmälle on kehitetty useita omia ohjelmistokehyksiä. Yksi yleisimmistä on laitteistoriippumaton OpenCL. Se tukee grafiikkasuorittimen lisäksi myös muita kohdelaitteita kuten keskussuorittimia ja digitaalisia signaaliprosessoreja. OpenCL muodostuu isäntälaitteella suoritettavan osuuden toteuttamiseen tarkoitetusta ohjelmointirajapinnasta sekä kohdelaitteella suoritettavan osuuden toteuttamiseen tarkoitetusta ohjelmointikielestä. Tällä ohjelmointikielellä toteutetaan niin sanottuja *ydinfunktioita*, joita suoritetaan useille eri datajoukoille yhtäaikaisesti (Du ym. 2012).

OpenCL:n ensimmäinen versio julkaistiin vuonna 2008. Viimeisin versio on vuonna 2017 julkaistu 2.2 (Khronos Group 2017).

2.6.3 Numeerinen lineaarialgebra

Numeerinen lineaarialgebra on matematiikan ja tietotekniikan ala, joka käsittelee lineaarialgebran laskentaan liittyviä algoritmeja, ja erityisesti suurten lineaaristen yhtälöryhmien ratkaisemista tietokoneella. Se on tärkeä osa useita laskennallisten tieteiden aloja, ja sitä sovelletaan laajalti myös käytännössä.

Monissa laskennallisten tieteiden ongelmissa syntyy lineaarisia yhtälöryhmiä, joissa voi olla useita miljoonia yhtälöitä ja muuttujia. Tällöin on erityisen tärkeää, että yhtälöryhmän ratkaisemisessa käytetty menetelmä on mahdollisimman nopea. Tällaisia menetelmiä on kehitetty useita ja ne jakautuvat kahteen luokkaan: *suoriin* ja *iteratiivisiin* menetelmiin. Suorat menetelmät antavat yhtälöryhmälle suoraan tarkan ratkaisun, kun taas iteratiiviset menetelmät lähtevät liikkelle jostakin ratkaisun alkuarvauksesta ja parantavat likimääräistä ratkaisua iteratiivisesti, kunnes sille ollaan saavutettu haluttu tarkkuus (Layton ja Sussman 2014).

Suorat menetelmät ovat iteratiivisia menetelmiä luotettavampia, ja niiden suoritusajat ovat helpommin ennakoitavissa. Niiden vaatimat muistin määrät ja suoritusajat saattavat kuitenkin kasvaa kohtuuttomat suuriksi, kun ratkaistavien yhtälöryhmien koko kasvaa (Hartikainen ja Kouhia 2010). Eräs tunnettu suora menetelmä on *QR-ratkaisija*, joka hyödyntää yhtälöryhmän kerroinmatriisista muodostettua QR-hajotelmaa, ja joka soveltuu myös *ylimääriteltyjen* yhtälöryhmien ratkaisemiseen (MathWorks 2017). Yhtälöryhmä on ylimääritelty kun yhtälöiden lukumäärä on suurempi kuin muuttujien.

Iteratiiviset menetelmät voivat erittäin suurten yhtälöryhmien tapauksissa olla suoria menetelmiä nopeampia ja käyttää niitä vähemmän muistia. Yksi erityisesti *positiivesti definiittien* matriisien kanssa yleisesti käytetty iteratiivinen menetelmä on *konjugaattigradienttimenetelmä* (Hartikainen ja Kouhia 2010). Reaalinen matriisi $\boldsymbol{M}$ on positiivisesti definiitti, jos $\boldsymbol{z}^{\boldsymbol{T}}\boldsymbol{M}\boldsymbol{z} > 0$ kaikilla nollasta poikkeavilla sarakevektoreilla $\boldsymbol{z}$.

Iteratiiviset menetelmät soveltuvat erityisen hyvin *harvojen* kerroinmatriisien omaavien yhtälöryhmien ratkaisemiseen. Matriisi on harva, jos suuri osa sen alkioista on nollia. Tarkalleen ottaen kun nollasta poikkeavien alkioiden lukumäärä riippuu lineaarisesti yhtälöiden lukumäärästä (Hartikainen ja Kouhia 2010). Harvoille matriiseille voidaan käyttää niille erityisesti mukautettuja tietorakenteita, jotka tallentavat ainoastaan nollasta poikkeavat alkiot ja

voivat näin ollen säästää muistia huomattavan paljon.

Iteratiivisten menetelmien tuottamat likiarvoiset ratkaisut saattavat joidenkin yhtälöryhmien tapauksessa konvergoida erittäin hitaasti. Suppenemista voidaan nopeuttaa käyttämällä *pohjustinoperaatiota*, jossa alkuperäisen yhtälöryhmän $\boldsymbol{Ax} = \boldsymbol{b}$ sijaan ratkaistaan sen kanssa ekvivalentti, pohjustettu yhtälöryhmä $\boldsymbol{M}_1^{-1}\boldsymbol{A}\boldsymbol{M}_2^{-1}\boldsymbol{x} = \boldsymbol{M}_1^{-1}\boldsymbol{b}$, missä $\boldsymbol{M}_1^{-1}$ ja $\boldsymbol{M}_2^{-1}$ ovat vasemman- ja oikeanpuoleiset *pohjustinmatriisit* tässä järjestyksessä (Hartikainen ja Kouhia 2010). Pohjustinmatriisien muodostamiseen on useita menetelmiä ja yksi positiivisesti definiittien kerroinmatriisien omaavien yhtälöryhmien kanssa yleisesti käytetty on *Chow-Patel-IChol0* (TUWien 2016b).

2.6.4 ViennaCL

ViennaCL on avoimeen lähdekoodiin perustuva numeerisen lineaarialgebraan erikoistunut ohjelmointikirjasto. Se on mahdollista konfiguroida käyttämään sisäisessä toteutuksessaan OpenCL-kirjastoa, jolloin se voi hyödyntää laskennassaan kaikkia OpenCL:n tukemia kohdelaitteita, joista erityisesti mainittakoon grafiikkasuorittimet. ViennaCL-kirjasto muodostuu ohjelmointirajapinnasta, joka tukee C++-, Python- ja Matlab-ohjelmointikieliä. Kirjaston uusin versio on vuonna 2016 julkaistu 1.7.1.

ViennaCL-kirjaston toteuttamat lineaaristen yhtälöryhmien ratkaisualgoritmit ovat keskittyneet erityisesti iteratiivisiin menetelmiin. Siitä löytyy toteutus muun muassa edellisessä luvussa mainitulle konjugaattigradienttimenetelmälle. Lisäksi se tukee pohjustinoperaatioita, ja sen avulla voidaan muodostaa Chow-Patel-IChol0-pohjustinmatriiseja. Kirjastossa on toteutettu myös harvan matriisin tietorakenne ja harvojen matriisien muodostamien yhtälöryhmien ratkaiseminen.

ViennaCL tarjoaa valmiiden toteutuksien lisäksi mahdollisuuden toteuttaa myös omia, mukautettuja ydinohjelmia (TUWien 2016a).

3 Hyperspektrikuvantaminen fotometrisessä stereossa

Tässä tutkimuksessa tarkasteltiin fotometrisen stereon toteuttamista hyperspektrikameralla otettujen kuvien avulla. Fotometristä stereota sovellettiin erikseen jokaiseen hyperspektrikuvan aallonpituusalueeseen. Jokaiselle aallonpituusalueelle muodostettiin sitä vastaava albedokartta sekä pinnan muodon määrittävät normaali- ja syvyyskartat.

Tutkimuksessa hyperspektrikuvattiin kolme eri kohdetta: Smurffi-hahmo, jääkaappimagneetti ja Lego-rakennelma. Kohteet valittiin niiden diffuusin heijastavuuden sekä vaihtelevien muotojen ja värien perusteella. Valinnat tehtiin kuitenkin niin, että muotojen vaihtelut olivat tarpeeksi maltilliset, jotta kohteet eivät varjosta liikaa itseään eivätkä omaa liikaa syvyydellisiä epäjatkuvuuskohtia.

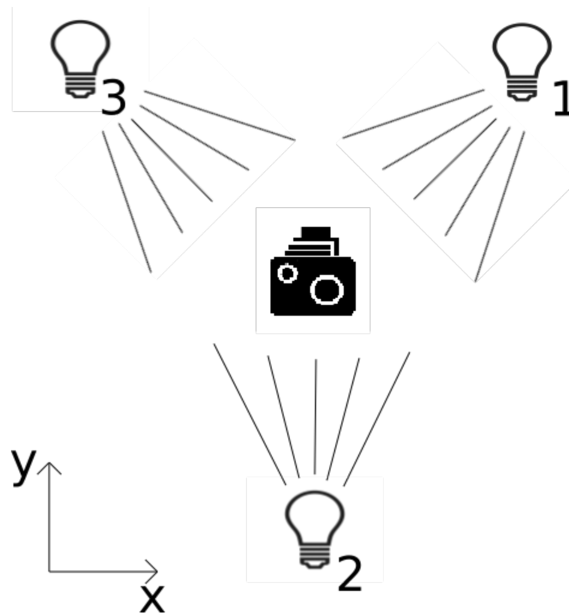
3.1 Koejärjestely

3.1.1 Kuvausasetelma

Tutkimuksessa kohteet kuvattiin käyttäen kolmea valonlähdettä. Jokaisesta kohteesta otettiin kolme hyperspektrikuvaa valaisten kohdetta kunkin kuvan kohdalla yhdellä valonlähteellä kerrallaan, luvussa 2.3.2 esitetyn menetelmän mukaisesti. Kuvat otettiin näiden valonlähteiden säteilemää valoa lukuunottamatta täysin pimeässä huoneessa varmistaen näin, että kohteen pinnalle saapuva irradianssi on peräisin ainoastaan halutusta valonlähteestä.

Kuviossa 4 on merkitty valonlähteiden suurpiirteiset sijainnit suhteessa kameraan ja näin ollen myös kuvion samassa kohdassa sijaitsevaan kuvattavaan kohteeseen. Sijannit on esitetty tutkimuksessa käytetyn, luvussa 3.2.2 esiteltävän koordinaatiston *xy*-suuntaiselle tasolle projisoituna. Kuvio esittää myös valonlähteiden järjestysnumerot, joihin tullaan viittaamaan jatkossa.

Kameran etäisyys kuvattavasta kohteesta oli noin 29 cm ja valonlähteiden noin 150 cm. Kuviossa 5 on tutkimuksessa käytetty kuvausasetelma valonlähteen 1 ollessa kytkettynä päälle.



Kuvio 4. Valojen sijainnit ja järjestysnumerot.

3.1.2 Koelaitteisto

Tutkimuksessa käytettiin luvussa 2.4.3 esitelytä Teknologian tutkimuskeskus VTT Oy:n kehittämää hyperspektrikameraa, jonka toiminta perustuu pietsosähköisellä aktuaattorilla säädettävään Fabry–Perot-interferometriin. Kameran valoherkkä kenno on tavallinen Bayer-suodattimella varustettu CMOS-tyyppinen RGB-kenno. Tällöin yhdellä valotuksella ja interferometrin raon suuruudella saadaan kuvattua maksimissaan kolme aallonpituusaluetta kerrallaan. Kennon koko on 1920×1200 kuvapistettä. Kameran spektraalista erottelukykyä kuvaavat aallonpituusalueiden puoliarvoleveydet ovat väliltä $7,00 - 25,00$ nm. Kamera kykenee havaitsemaan aallonpituusalueet väliltä $400,00 - 1000,00$ nm. Tutkimuksessa käytetty hyperspektrikamera on nähtävissä kuviossa 5.

Hyperspektrikamerassa käytetyn linssin polttoväli on 35 mm ja aukkosuhde $f/1,4$. Linssin vaakasuuntaisen kuvakulman suuruus on 20,9°ja pystysuuntaisen 15,8°.

3.1.3 Koeasetukset

Tutkimuksen hyperspektrikuvat otettiin käyttäen kameran suurinta mahdollista resoluutiota, jolloin kuvissa on 1920×1200 kuvapistettä jokaista aallonpituusaluetta kohden. Jatkossa

Kuvio 5. Kuvausasetelma.

yhtä tällaista aallonpituusaluetta vastaavien kuvapisteiden joukkoa kutsutaan hyperspektrikuvan *siivuksi*.

Tutkimuksessa otetuissa hyperspektrikuvissa on 133 siivua väliltä $456,00 - 840,00$ nm. Aallonpituusalueiden puoliarvoleveydet ovat väliltä $11,79 - 16,64$ nm ja niiden keskiarvo on $14,07$ nm. 133 aallonpituusaluetta kerättiin käyttäen 81 eri interferometrin raon suuruutta. Yhdellä raon suuruudella tehdyn valotuksen valotusaika oli 150 ms, jolloin yhden hyperspektrikuvan ottamiseen kulunut kokonaisaika oli 12,15 s.

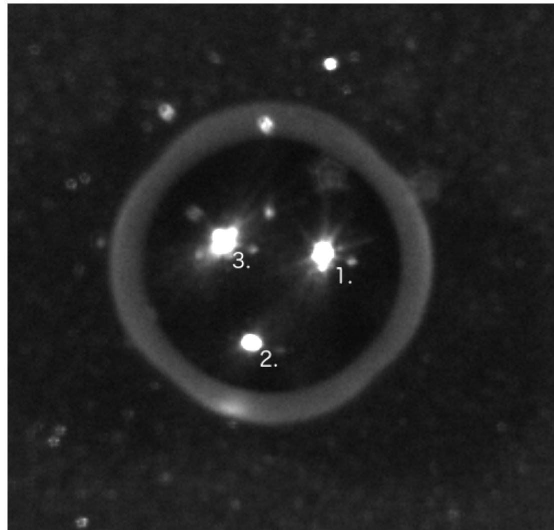
Hyperspektrikameran raakakuvista interpoloidut tutkimuksen toteutuksessa käytetyt kuvat tallennettiin ENVI-formaatissa. Kuvapisteet kirjoitettiin ENVI-tiedostoihin BSQ (Band Sequential Format) -muodossa, jossa tiedostot alkavat ensimmäisen siivun ylimmän vaakarivin kuvapisteiden arvoilla. Niitä seuraa saman siivun seuraavan vaakarivin kuvapisteiden arvot ja näin jatkuen alimpaan vaakariviin saakka, jonka jälkeen alkaa seuraava siivu ylimmästä vaakarivistä. Kuvapisteiden arvot on kirjoitettu kaskinkertaisen tarkkuuden liukulukuina siten, että liukuluvun eniten merkitsevät tavut on tallennettu liukuvun alimpiin muistiosoitteisiin.

3.2 Valojen suunnat

3.2.1 Kalibrointi- ja maskikuvat

Jotta luvussa 2.3.2 esitetty yhtälöryhmä (2.35) voitaisiin ratkaista, on valojen suuntavektorit $\boldsymbol{l}$ tunnettava. B. K. Horn (1989) ehdotti niiden määrittämiseksi menetelmää, jossa spekulaarisesti heijastava pallo kuvataan samoissa asetelmissa kuin tutkittava kohde, jolloin valojen spekulaaristen heijastuksien huippukohdista saadaan ratkaistua valojen suunnat. Kyseistä menetelmää käytettiin tässä tutkimuksessa.

Valojen suuntien määrittämistä varten otetuissa kalibrointikuvissa käytettiin kromipalloa, joka heijastaa valoa lähes ainoastaan spekulaarisesti. Pallosta otettiin kolme kuvaa valaisten sitä samoista etäisyyksistä ja suunnista kuin tutkittavaa kohdetta, yhdestä suunnasta kerrallaan. Kuvat otettiin samalta etäisyydeltä kuin tutkittavasta kohteesta ja samoilla kameran asetuksilla.



Kuvio 6. Kaikkien kolmen kalibraatiohyperspektrikuvan 67. aallonpituusalue yhdistettynä.

Kuviossa 6 on yhdistettynä kolmen hyperspektrikuvan 67. aallonpituusalue, jonka spektraalisen herkkyyden huipun aallonpituus on 615,05 nm ja puoliarvoleveys 15,69 nm. Näin ollen kaikkien kolmen valon spekulaaristen heijastuksien huippukohdat ovat näkyvissä samassa kuviossa. Kuviossa huippukohdat on merkitty niitä vastaavien valonlähteiden numeroilla, luvussa 3.1.1 esitetyllä numeroimistavalla.

Jotta huippukohtien sijainnit suhteessa kromipalloon saatiin määritettyä automaattisesti, oli ohjelman pystyttävä erottamaan kromipallo kuvasta. Tätä varten luotiin kuvankäsittelyohjelmalla manuaalisesti maskikuva, joka on saman kokoinen kuin hyperspektrikuvien yksi siivu ja jossa kuvapisteiden arvot ovat kaikkialla muualla nolla paitsi pallon kohdalla yksi. Kromipallon erottaminen kuvasta olisi mahdollista myös automaattisesti, jos kuvatessa huolehdittaisiin, että pallo erottuu tarpeeksi selkeästi taustastaan.

3.2.2 Suuntien määrittäminen

Kalibrointikuvia otettaessa kameran etäisyys suhteessa kromipallon halkaisijaan oli suuri, joten kameran projektio oletettiin ortogonaaliseksi. Lisäksi tutkimuksen jokaisessa vaiheessa käytetään *oikeakätistä* koordinaatistoa, jossa x-akseli osoittaa vasemmalta oikealle, y-akseli alhaalta ylös ja z-akseli kohti kameraa.

Valojen suuntien määrittäminen toteutettiin Matlab-ohjelmointikielellä ja sen lähdekoodi on liitteenä A.

Toteutuksessa ensimmäisenä ladataan edellisessä luvussa esitelty maskikuva, josta valitaan niiden kuvapisteiden joukko, joiden arvo on nollasta poikkeava. Etsimällä tämän joukon kuvakoordinaattien maksimit ja minimit sekä vaaka- että pystysuunnassa määritetään kromipallon keskipiste ja säde kuvassa. Vaakasuunnan mittayksikkönä on kuvapisteen leveys ja pystysuunnassa vastaavasti kuvapisteen korkeus.

Tämän jälkeen jokaisen valon suunta määritetään yksi kerrallaan käyttämällä aiemmin määritettyä pallon kuvan keskipistettä ja sädettä.

Jokaisen valonlähteen kohdalla sen vastaavasta hyperspektrikuvasta tarkastellaan yhtä siivua kerrallaan, ja siitä etsitään kaikista suurimman radianssiarvon omaava kuvapiste sekä sen vastaavat kuvakoordinaatit. Seuraavaksi määritetään kolmiulotteinen suuntavektori pallon keskipisteestä pallon pinnan kirkkaimpaan pisteeseen laskemalla suurimman radianssin omaavan kuvapisteen etäisyys pallon kuvan keskipisteestä ja asettamalla tämän etäisyyden vaakasuuntainen komponentti suuntavektorin x-komponentiksi ja pystysuuntainen komponentti suuntavektorin y-komponentiksi. Koska tämän suuntavektorin pituus tiedetään olevan yhtä suuri kuin pallon säde, niin jos suuntavektori on $\hat{\boldsymbol{n}}$, määritetään sen z-komponentti seu-

raavasti:

$$\hat{\boldsymbol{n}}_z = \sqrt{r^2 - \hat{\boldsymbol{n}}_x^2 - \hat{\boldsymbol{n}}_y^2}, \tag{3.1}$$

missä r on pallon säde.

Vektori $\hat{\boldsymbol{n}}$ on samansuuntainen pallon kirkkaimman kohdan normaalivektorin kanssa, joten normaali $\boldsymbol{n}_p$ määritetään normalisoimalla $\hat{\boldsymbol{n}}$.

$$\boldsymbol{n}_p = \frac{\hat{\boldsymbol{n}}}{|\hat{\boldsymbol{n}}|}, \tag{3.2}$$

missä alaindeksi p viittaa palloon.

Koska kameran projektio oletettiin ortogonaaliseksi, on kaikkien pisteiden kameran suunta oikeakätisessä koordinaatistossa $\boldsymbol{v} = [0, 0, 1]^T$. Valon suunta määritetään asettamalla pallon kirkkaimmasta kohdasta täydellisesti heijastuneen säteen suunnaksi kameran suunta ja ratkaisemalla heijastuvan suunnan yhtälöstä valon lähteen suunta. Jos hyperspektrikuvan m. aallonpituusalueesta määritettyä pallon kirkkaimman kohdan normaalivektoria merkitään symbolilla $\boldsymbol{n}_{p_m}$, niin sitä vastaava valon suunta

$$\boldsymbol{l}_m = 2\, \boldsymbol{v} \bullet \boldsymbol{n}_{p_m}\, \boldsymbol{n}_{p_m} - \boldsymbol{v}. \tag{3.3}$$

Lopullinen valon suunta $\mathbf{l}$ määritetään laskemalla kaikkia 133 aallonpituusaluetta vastaavat valon suunnat yhteen ja normalisoimalla näin saatu summavektori:

$$\boldsymbol{l} = \frac{\sum_{m=1}^{133} \boldsymbol{l}_m}{|\sum_{m=1}^{133} \boldsymbol{l}_m|}. \tag{3.4}$$

Valonlähteet oletetaan tutkimuksessa suuntaisvaloiksi, jolloin edellä määritetyt suunnat ovat riippumattomia käsiteltävästä kuvapisteestä.

Valojen suuntien määrittäminen on suhteellisen yksinkertaista ja nopeaa, joten sen toteuttamista muulla kuin Matlab-ohjelmointikielellä ei nähty tarpeelliseksi.

3.3 Albedo- ja normaalikarttojen luominen

3.3.1 Lähteen irradianssi

Luvussa 2.3.2 pinnalle saapuva irradianssi oletettiin yhtälöiden selkeyttämiseksi tietyn suuruiseksi, mutta tutkimuksessa lähteen irradianssi on määritettävä.

Irradianssin määrittämiseksi hyperspektrikameralla otettiin kolme *valkoreferenssikuvaa*, tasaisesta kameran suuntaa vastaan kohtisuorasta valkoisesta mattapintaisesta tasosta. Tähän tarkoitukseen käytettiin tavallista tulostuspaperia. Kuvausasetelma oli sama kuin kalibraatiokuvissa ja jokaisessa kuvassa pintaa valaistiin ainoastaan sillä valolla, jonka irradianssi sen kuvan avulla oli tarkoitus selvittää. Pinnan oletettiin olevan ideaalinen Lambertin pinta, toisin sanoen se heijastaa valoa ainoastaan diffuusisti ja sen albedo on kaikilla aallonpituuksilla yksi.

Koska kuvattava taso oli kohtisuorassa kameran suuntaa vastaan ja käytetään ortogonaaliprojektiota sekä oikeakätistä koordinaatistoa, niin pinnan normaali on kaikissa kuvapisteissä $\boldsymbol{n}_v = [0,0,1]^T$ (alaindeksi v viittauksena valkoreferenssiin). Nyt kun pinnan orientaatio, pinnan albedo sekä valon suunta tunnetaan ja jos lisäksi pinnalta heijastunut spektraalinen radianssi oletetaan tunnetuksi, niin spektraalinen irradianssi voidaan ratkaista yhtälöstä (2.25):

$$E_{\lambda_0} = \frac{\pi}{a_\lambda \cos(\theta_0)} L_{\lambda_r} = \frac{\pi}{\boldsymbol{l} \bullet \boldsymbol{n}_v} L_{\lambda_r}. \tag{3.5}$$

Koska spektraaliset suureet ovat osamääriä differentiaalisen aallonpituusalueen suhteen, niitä ei voida mitata tarkasti käytännössä. Valkoreferenssihyperspektrikuvan kuvapisteen arvo onkin pinnalta heijastunut spektraalinen radianssi painotettuna kameran ominaisella spektraalisella havaitsemisherkkyydellä, integroituna kuvan siivua vastaavan aallonpituualueen Λ yli. Merkitään valkoreferenssihyperspektrikuvan m. aallonpituusaluetta symbolilla Λ_m ja sitä vastaavan siivun kuvapisteen arvoa seuraavasti:

$$\hat{L}_{v_m} := \int_{\Lambda_m} H(\lambda) L_{\lambda_v} d\lambda, \tag{3.6}$$

missä $H(\lambda)$ on kameran spektraalinen havaitsemisherkkyys aallonpituuden funktiona, L_{λ_v} valkoreferenssitasolta heijastunut spektraalinen radianssi ja $m = 1,2,\ldots,133$. Näin ollen aallonpituusalueen yli integroitu spektraalisella herkkyydellä painotettu lähteen irradianssi

saadaan yhtälöstä (3.5):

$$\int_{\Lambda_m} HE_{\lambda_0} d\lambda = \frac{\pi}{\boldsymbol{l} \bullet \boldsymbol{n}_v} \int_{\Lambda_m} HL_{\lambda_v} d\lambda = \frac{\pi}{\boldsymbol{l} \bullet \boldsymbol{n}_v} \hat{L}_{v_m}. \tag{3.7}$$

Tutkimuksen toteutuksessa tämä arvo määritetään luvulla $\frac{1}{\pi}$ skaalattuna, koska se on käytännöllinen valinta luvun 3.3.2 albedojen ja normaalien määrittämiseen. Jos tätä skaalattua arvoa merkitään symbolilla $\hat{E}_m$, niin

$$\hat{E}_m := \frac{\int_{\Lambda_m} HE_{\lambda_0} d\lambda}{\pi} = \frac{\hat{L}_{v_m}}{\boldsymbol{l} \bullet \boldsymbol{n}_v}. \tag{3.8}$$

3.3.2 Albedojen ja normaalien määrittäminen

Kuten valkoreferenssikuvien tapauksessa, myös tutkittavan kohteen hyperspektrikuvien kuvapisteiden arvot ovat samalla painofunktiolla $H(\lambda)$ kerrottujen heijastuneiden spektraalisten radianssien integraaleja. Merkitään niitä $\hat{L}_m := \int_{\Lambda_m} HL_{\lambda_r} d\lambda$. Tutkimuksessa oletettiin, että albedo ja normaalivektori ovat vakioita hyperspektrikuvan pisteessä, minkä tahansa aallonpituusalueen Λ_m yli, ja niitä merkitään a_m ja $\boldsymbol{n}_m$ tässä järjestyksessä. Näin ollen yhtälöstä (2.25) ja määritelmästä (3.8) saadaan

$$\int_{\Lambda_m} HL_{\lambda_r} d\lambda = \int_{\Lambda_m} \frac{a_\lambda \, \boldsymbol{l} \bullet \boldsymbol{n}}{\pi} HE_{\lambda_0} d\lambda = \frac{a_m \boldsymbol{l} \bullet \boldsymbol{n}_m}{\pi} \int_{\Lambda_m} HE_{\lambda_0} d\lambda = a_m \boldsymbol{l} \bullet \boldsymbol{n}_m \hat{E}_m, \tag{3.9}$$

josta edelleen

$$\frac{\hat{L}_m}{\hat{E}_m} = a_m \boldsymbol{l} \bullet \boldsymbol{n}_m. \tag{3.10}$$

Tätä osamäärää merkitään $i_m := \frac{\hat{L}_m}{\hat{E}_m}$. Se määritetään jokaiselle kolmelle valolle, ja niistä muodostetaan yhtälöryhmän (2.35) vektori $\boldsymbol{i}_m = [i_m^1,\ i_m^2,\ i_m^3]^T$, missä yläindeksien numerointi vastaa luvun 3.1.1 valojen numerointia.

Yhtälöryhmä (2.35) ratkaistaan jokaiselle hyperspektrikuvan aallonpituusalueelle Λ_m muodostamalla sarakevektoreista $\boldsymbol{i}_m$ matriisi

$$\boldsymbol{I}_{3\times 133} = \begin{bmatrix} i_1^1 & \dots & i_{133}^1 \\ i_1^2 & \dots & i_{133}^2 \\ i_1^3 & \dots & i_{133}^3 \end{bmatrix} \tag{3.11}$$

ja tuloista $a_m \boldsymbol{n}_m$ matriisi

$$\boldsymbol{N}_{3\times133} = \begin{bmatrix} a_1 n_{1_x} & \dots & a_{133} n_{133_x} \\ a_1 n_{1_y} & \dots & a_{133} n_{133_y} \\ a_1 n_{1_z} & \dots & a_{133} n_{133_z} \end{bmatrix}, \tag{3.12}$$

jolloin

$$\boldsymbol{L}_{3\times3} \boldsymbol{N}_{3\times133} = \boldsymbol{I}_{3\times133}. \tag{3.13}$$

Tämä yhtälöryhmä ratkaistaan jokaiselle tutkittavan kohteen hyperspektrikuvan kuvapisteelle, jolloin ratkaisumatriisin m. sarakevektorin pituus on kyseisen kuvapisteen aallonpituusalueen Λ_m vastaava albedo a_m. Tätä aallonpituusaluetta vastaava normaalivektori $\boldsymbol{n}_m$ taas määritetään normalisoimalla m. sarakevektori.

3.3.3 Matlab-toteutus

Matlab-toteutus noudattaa hyvin suoraviivaisesti edellisessä luvussa esitettyä menetelmää ja sen lähdekoodi on liitteenä B.

Valkoreferenssien ja tutkittavan kohteen hyperspektrikuvat ladataan kolmiulotteisiin $1200 \times 1920 \times 133$ -kokoisiin *kuutiomatriiseihin*, joiden alkiot ovat yksinkertaisen tarkkuuden liukulukuja, jolloin yksi kuutiomatriisi varaa muistia noin 1,14 Gt. Hyperspektrikuvien suuren muistintarpeen vuoksi joka hetkellä pyritään muistissa säilyttämään vain minimimäärä kuutiomatriiseja. Tutkittavan kohteen kaikki kolme kuutiomatriisia on pidettävä samaan aikaan muistissa, mutta valkoreferenssikuutiomatriiseja tarvitaan vai yhtä kerrallaan, jolloin kaikki valkoreferenssikuvat ladataan vuorotellen samaan muistialueeseen. Tämä muistialue vapautetaan, kun viimeistäkään valkoreferenssikuutiomatriisia ei enää tarvita.

Valkoreferenssikuutiomatriisit jaetaan niitä vastaavilla luvuilla $\boldsymbol{l} \bullet \boldsymbol{n}_v$, jolloin kuutiomatriisien alkiot ovat suureita $\hat{E}_m$. Tämän jälkeen tutkittavan kohteen kuutiomatriisit jaetaan alkioittain $\hat{E}_m$-kuutiomatriiseilla, jolloin tuloksena saadun kuutiomatriisin alkiot vastaavat lukuja i_m.

Ennen fotometrisen stereon yhtälöryhmien ratkaisua normaalikartoille varataan muistista $1200 \times 1920 \times 3 \times 133$ -kokoinen neliulotteinen matriisi ja albedo-kartoille $1200 \times 1920 \times 133$ -kokoinen kuutiomatriisi. Tämän jälkeen 1200×1920 kuvapistettä käydään yksi ker-

rallaan läpi muodostaen tarvittavat matriisit ja ratkaisten yhtälöryhmät käyttäen Matlabin \ -operaattoria. Lisäksi jokaisen kuvapisteen kohdalla erotetaan ratkaisumatriisien sarakevektoreista albedot sekä normaalivektorit ja sijoitetaan ne aiemmin luotuihin matriiseihin.

Normaalien alkiot normalisoidaan välille $[0, 1]$ ja kirjoitetaan yksinkertaisen tarkkuuden liukulukuina 133 eri tiedostoon – jokainen eri aallonpituusaluetta vastaava normaalikartta omaansa. Albedot kirjoitetaan skaalaamattomina vastaavasti omiin albedokarttatiedostoihinsa käyttäen yksinkertaisen tarkkuuden liukulukuja.

3.3.4 GPU-toteutus

GPU:lla suoritettava versio albedo- ja normaalikarttojen luomisesta toteutettiin C++-ohjelmointikielellä ja avoimen lähdekoodin lineaarialgebrakirjastolla ViennaCL. Käytetty ViennaCL-kirjaston versio oli 1.7.1, ja se asetettiin käyttämään sisäisessä GPU-toteutuksessaan OpenCL-kirjastoa sekä sen versiota 1.2. Tässä luvussa esitettävän toteutuksen lähdekoodit ovat liitteenä D.

ViennaCL-kirjasto ei tue kuutiomatriiseja, joten hyperspektrikuvat kopioidaan näytönohjaimen muistiin käyttäen kaksiulotteisia 11520×13300 -kokoisia matriiseja. Yksi hyperspektrikuva varaa kaksi tällaista matriisia. Kun kuvattavan kohteen sekä sen vastaavan valkoreferenssin hyperspektrikuvat on ladattu näytönohjaimen muistiin, lukujen i_m määrittämiseen vaadittavat jakolaskut suoritetaan GPU:lla.

Ennen yhtälöryhmien ratkaisemista matriisista $\boldsymbol{L}$ muodostetaan GPU:lla LU-hajotelma, joka ylikirjoittaa näytönohjaimen muistissa alkuperäisen matriisin. Tämä tehdään vain kerran, koska valot oletettiin suuntaisvaloiksi, joten matriisi $\boldsymbol{L}$ on kuvapisteiden suhteen vakio. Lisäksi albedojen ja normaalien tallentamiseen varataan keskusmuistista yksiulotteinen taulukko, jonka alkiot ovat rakenteita, jotka sisältävät neljä yksinkertaisen tarkkuuden liukulukua.

Matlab-toteutuksen tavoin hyperspektrikuvan yhtä siivua vastaava määrä kuvapisteitä käydään läpi. Jokaisen pisteen kohdalla muodostetaan matriisi I ja kopioidaan se näytönohjaimen muistiin, jonka jälkeen yhtälöryhmä ratkaistaan GPU:lla käyttäen aiemmin luotua LU-hajotelmaa. Ratkaisumatriisin sarakevektoreista saadut albedot ja normaalivektorit kirjoitetaan aiemmin varattuun yksiulotteiseen taulukkoon. Normaalivektori kirjoitetaan taulukon

alkion rakenteen kolmeen ensimmäiseen jäsenmuuttujaan ja albedo viimeiseen.

3.3.5 Säikeistetty CPU-toteutus

Fotometrisen stereon säikeistetty CPU-toteutus luotiin käyttäen C++-ohjelmointikieltä ja Armadillo-lineaarialgebrakirjastoa. Armadillo linkitettiin edelleen OpenBLAS-kirjastoon, jolloin se hyödyntää sisäisessä totetuksessaan säikeistystä. Lähdekoodi on liitteenä F.

Armadillo lukee ja kirjoittaa matriisit sarakkeittain, joten hyperspektrikuvat oli muunnettava ensin vastaavaan muotoon. Muuten Armadillon toiminnallisuus vastaa hyvin tarkasti Matlabia, joten kielten välisiä syntaktisia eroavaisuuksia lukuunottamatta fotometrisen stereon Armadillo- ja Matlab-totetukset ovat lähes identtiset. Armadillo ei kuitenkaan tue neli- tai suurempiulotteisia matriiseja, joten normaalikartat kirjoitetaan 133 erilliseen muistialueeseen. Nämä muistialueet ovat yksiulotteisia taulukoita, joiden alkiot ovat kolmen yksinkertaisen tarkkuuden liukuluvun muodostamia rakenteita.

3.4 Syvyyskartan luominen

3.4.1 Harvan yhtälöryhmän muodostaminen

Tutkimuksen menetelmällä muodostetut syvyysarvot riippuvat valon aallonpituudesta ollen kuitenkin vakioita jokaisen hyperspektrikuvan aallonpitusalueen Λ_m yli. Aiemmin käytetyn merkitsemistavan mukaisesti, myös syvyysarvoissa m. aallonpituusväliin viitataan alaindeksillä m.

Yhtälöitä (2.44) sovelletaan jokaiseen hyperspektrikuvan kuvapisteeseen siten, että Δz_{m_x} mitataan suhteessa tarkasteltavana olevan kuvapisteen oikealla puolella ensimmäisenä sijaitsevaan kuvapisteeseen ja Δz_{m_y} suhteessa alapuolella ensimmäisenä sijaitsevaan kuvapisteeseen. Tällöin oikeakätisessä koordinaatistossa

$$\Delta z_{m_x} = z_m(x,y) - z_m(x+1,y) = \frac{\boldsymbol{n}_{m_x}(x,y)}{\boldsymbol{n}_{m_z}(x,y)}, \qquad \Delta z_{m_y} = z_m(x,y) - z_m(x,y-1) = -\frac{\boldsymbol{n}_{m_y}(x,y)}{\boldsymbol{n}_{m_z}(x,y)}. \tag{3.14}$$

Jos tämä ei ole mahdollista sen takia, että kuvapisteen oikealla tai alapuolella ei ole ku-

vapistettä, syvyyserot mitataan suhteessa vasemmalla ja yläpuolella ensimmäisenä oleviin kuvapisteisiin tässä järjestyksessä, jolloin

$$\Delta z_{m_x} = z_m(x,y) - z_m(x-1,y) = -\frac{\boldsymbol{n}_{m_x}(x,y)}{\boldsymbol{n}_{m_z}(x,y)}, \qquad \Delta z_{m_y} = z_m(x,y) - z_m(x,y+1) = \frac{\boldsymbol{n}_{m_y}(x,y)}{\boldsymbol{n}_{m_z}(x,y)}. \tag{3.15}$$

Hyperspektrikuvan aallonpituusaluetta Λ_m vastaavan siivun kuvapisteiden vastaavista syvyysarvoista $z_m(x,y)$ muodostetaan vektori $\boldsymbol{z}_m$ seuraavasti:

$$\boldsymbol{z}_m = \begin{bmatrix} z_m(1,1) \\ \vdots \\ z_m(w,1) \\ z_m(1,2) \\ \vdots \\ z_m(w,2) \\ \vdots \\ \vdots \\ z_m(1,h) \\ \vdots \\ z_m(w,h) \end{bmatrix}_N, \tag{3.16}$$

missä w on yhden siivun leveys kuvapisteinä, h korkeus ja $N = wh$ kuvapisteiden lukumäärä.

Näin ollen kun yhtälöitä (3.14) ja (3.15) sovelletaan jokaiseen yhden siivun kuvapisteeseen, muodostuneen yhtälöryhmän kerroinmatriisiksi saadaan

$$\hat{\boldsymbol{M}}_{2N\times N} = \begin{bmatrix} 1 & -1 & & & & & & & & \\ 1 & & & -1 & & & & & & \\ & 1 & -1 & & & & & & & \\ & 1 & & & -1 & & & & & \\ & & & \ddots & & & & & & \\ & & -1 & 1 & & & & & & \\ & & & 1 & & & -1 & & & \\ & & & & & \ddots & & & & \\ & & & & & 1 & -1 & & & \\ & & & & & 1 & & & & -1 \\ & & & & & & & \ddots & & \\ & & & & & & 1 & -1 & & \\ & & & -1 & & & 1 & & & \\ & & & & & & & & \ddots & \\ & & & & & & & & -1 & 1 \\ & & & & & & -1 & & & 1 \end{bmatrix}. \tag{3.17}$$

Matriisin jokainen merkitsemätön alkio on nolla, joten jokaisella rivillä on vain kaksi nollasta poikkeavaa alkiota, eli matriisi on harva. Parittomat rivit vastaavat syvyyseroja vaakasuunnassa vierekkäisten kuvapisteiden välillä ja parilliset pystysuunnassa. Jos parittoman rivin

negatiivisen alkion sarakeindeksi on suurempi kuin positiivisen alkion, rivi vastaa syvyyseroa oikealla puolella olevaan kuvapisteeseen verrattuna, muuten vasemmalla. Vastaavasti jos parillisen rivin negatiivisen alkion sarakeindeksi on suurempi kuin positiivisen alkion, rivi vastaa syvyyseroa alapuolella olevaan kuvapisteeseen verrattuna, muuten yläpuolella. Parillisien rivien nollasta poikkeavien alkioiden välissä on aina $w-1$ nollaa. Parittoman rivin negatiivisen alkion sarakeindeksi on pienempi kuin positiivisen alkion kun rivi-indeksi on $k2w-1$, missä $k=1,2,\ldots,h$. Pienin parillinen rivi-indeksi, jonka vastaavalla rivillä negatiivisen alkion sarakeindeksi on pienempi kuin positiivisen alkion, on $2(N-w+1)$.

Näin muodostetun matriisin aste $rank(\hat{\boldsymbol{M}})=N-1$, eli se ei ole täysiasteinen, jolloin matriisi $\hat{\boldsymbol{M}}^T\hat{\boldsymbol{M}}$ ei ole kääntyvä. Tämä ongelma ratkaistaan asettamalla reunaehto siten, että oikeassa alakulmassa olevan kuvapisteen alapuolella olevaa kuvitteellista kuvapistettä vastaava syvyysarvo asetetaan nollaksi. Näin ollen saadaan täysiasteinen matriisi $\boldsymbol{M}$, eli jolle $rank(\boldsymbol{M})=N$. Tämä eroaa matriisista $\hat{\boldsymbol{M}}$ ainoastaan siten, että sen alimmalla rivillä ei ole lainkaan negatiivista alkiota, vaan se on korvattu nollalla.

Muodostetaan myös vektori $\boldsymbol{\delta}_m$ siten, että

$$\boldsymbol{\delta}_m=\begin{bmatrix}\boldsymbol{n}_{m_x}(1,1)/\boldsymbol{n}_{m_z}(1,1)\\ \boldsymbol{n}_{m_y}(1,1)/\boldsymbol{n}_{m_z}(1,1)\\ \vdots\\ \boldsymbol{n}_{m_x}(w,1)/\boldsymbol{n}_{m_z}(w,1)\\ \boldsymbol{n}_{m_y}(w,1)/\boldsymbol{n}_{m_z}(w,1)\\ \vdots\\ \vdots\\ \boldsymbol{n}_{m_x}(1,h)/\boldsymbol{n}_{m_z}(1,h)\\ \boldsymbol{n}_{m_y}(1,h)/\boldsymbol{n}_{m_z}(1,h)\\ \vdots\\ \boldsymbol{n}_{m_x}(w,h)/\boldsymbol{n}_{m_z}(w,h)\\ \boldsymbol{n}_{m_y}(w,h)/\boldsymbol{n}_{m_z}(w,h)\end{bmatrix}_{2N}. \tag{3.18}$$

Lisäksi muodostetaan $2N$-pituinen sarakevektori $\boldsymbol{s}$, jonka jokainen parittoman rivin alkio on 1, lukuun ottamatta rivejä, joiden rivi-indeksi on $k2w-1$, missä $k=1,2,\ldots,h$, jolloin alkio on -1. Jokainen vektorin $\boldsymbol{s}$ parillisen rivin alkio on -1, lukuun ottamatta rivejä, joiden rivi-indeksi on suurempi tai yhtä suuri kuin $2(N-w+1)$, jolloin alkio on 1.

Tällöin aallonpituusaluetta Λ_m vastaava ylimääritelty lineaarinen yhtälöryhmä on muodos-

tettujen vektorien ja matriisin avulla ilmaistuna

$$\boldsymbol{M}\boldsymbol{z}_m = \boldsymbol{s} \circ \boldsymbol{\delta}_m, \tag{3.19}$$

missä $\circ$ on vektoreiden alkiokohtainen tulo.

Lisäksi jos vektoreista $\boldsymbol{z}_m$ muodostetaan matriisi $\boldsymbol{Z} = [\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_{133}]$ ja alkiokohtaisista tuloista $\boldsymbol{s} \circ \boldsymbol{\delta}_m$ matriisi $\boldsymbol{\Delta} = [\boldsymbol{s} \circ \boldsymbol{\delta}_1, \boldsymbol{s} \circ \boldsymbol{\delta}_2, \ldots, \boldsymbol{s} \circ \boldsymbol{\delta}_{133}]$, niin kaikkia eri allonpituusalueita Λ_m vastaavat lineaariset yhtälöryhmät voidaan esittää yhtenä ylimääriteltynä yhtälöryhmänä:

$$\boldsymbol{M}_{2N,N}\boldsymbol{Z}_{N,133} = \boldsymbol{\Delta}_{2N,133}. \tag{3.20}$$

3.4.2 Syvyyskarttojen muodostaminen Matlabilla

Syvyyskarttojen muodostamisen sarjallinen versio toteutettiin Matlab R2016b:llä ja sen lähdekoodi on liitteenä C.

Ensimmäisenä toteutuksessa muodostetaan vektori $\boldsymbol{s}$ ja matriisi $\boldsymbol{M}$. Vektori $\boldsymbol{s}$ ja matriisi $\boldsymbol{M}$ muodostetaan vain kerran, koska ne ovat aallonpituudesta riippumattomia vakioita. Matriisi $\boldsymbol{M}$ luodaan Matlabin sparse-funktiolla, jolloin matriisi tallennetaan muistiin käyttäen harvan matriisin pakattua tietorakennetta. Tämä on erityisen tärkeää, koska tiheän matriisin muodossa tallennettuna $\boldsymbol{M}$ varaisi muistia yli puoli miljoonaa -kertaisen määrän harvan matriisin tallennusmuotoon verrattuna. Matlabin harva matriisi muodostetaan kolmesta vektorista, joista ensimmäinen koostuu matriisin nollasta poikkeavien alkioiden rivi-indekseistä, toinen sarakeindekseistä ja kolmas itse alkioista.

Tämän jälkeen kaikki aalloonpituusalueet Λ_m käydään läpi ja jokaisen kohdalla muodestaan vektori $\boldsymbol{\delta}_m$ sekä alkioittainen tulo $\boldsymbol{s} \circ \boldsymbol{\delta}_m$. Lisäksi jokaisen siivun kohdalla ratkaistaan ylimääritelty yhtälöryhmä (3.19) käyttäen pienimmän neliösumman menetelmää, normalisoidaan näin saadut syvyysarvot ja kirjoitetaan ne yksinkertaisen tarkkuuden liukulukuina syvyyskarttatiedostoon. Tutkimuksessa käytetty Matlabin versio käyttää näiden yhtälöryhmien ratkaisemiseen sisäisessä toteutuksessaan QR-ratkaisijaa.

Tutkimuksessa yritettiin ratkaista myös kaikki yhtälöryhmät kerralla käyttäen yhtälöä (3.20), mutta jokaisella yrityksellä Matlabin versio R2016b sulkeutui ilman virheilmoitusta kesken operaation.

3.4.3 Syvyyskarttojen muodostaminen GPU:lla

GPU:lla suoritettavaan syvyyskarttojen muodostamiseen käytettiin fotometrisen stereon GPU-toteutuksen tavoin C++-ohjelmointikieltä ja ViennaCL- sekä OpenCL-kirjastojen yhdistelmää. Lisäksi käytettiin OpenBLAS-kirjastoon linkitettyä Armadillo-kirjastoa. Syvyyskarttojen muodostamisen GPU-toteutuksen lähdekoodi on liitteenä E.

Armadillon ja Matlabin ohjelmointirajapintojen samankaltaisuuden vuoksi GPU-toteutuksen vektorin $\boldsymbol{s}$ sekä matriisin $\boldsymbol{M}$ muodostamiseen käytettiin Armadillo-kirjastoa. Tämä eroaa Matlab-toteutuksesta kielen syntaksin lisäksi ainoastaan kahdella tavalla: Armadillo käyttää nollasta alkavaa indeksöintiä, Matlabin yhdestä alkavan sijaan, Armadillon harva matriisi luodaan yhdestä tiheästä matriisista ja yhdestä vektorista, Matlabin kolmen vektorin sijaan. Harvan matriisin luontiin käytettävä tiheä matriisi muodostuu kahdesta sarakevektorista, joista ensimmäinen koostuu harvan matriisin nollasta poikkeavien alkioiden riviindekseistä ja toinen sarakeindekseistä. Harvan matriisin luontiin käytettävä vektori muodostuu harvan matriisin nollasta poikkeavista alkioista.

ViennaCL-kirjasto tukee ainoastaan kääntyvien kerroinmatriisien omaavien yhtälöryhmien ratkaisemista, joten se ei sellaisenaan sovellu tutkimuksen ylimääritellyille yhtälöryhmille (3.19). Tästä syystä määritetään transpoosi $\boldsymbol{M}^T$ ja matriisitulo $\boldsymbol{M}^T\boldsymbol{M}$, joista jälkimmäinen on asetetun reunaehdon vuoksi kääntyvä. $\boldsymbol{M}^T\boldsymbol{M}$ on lisäksi symmetrinen ja positiivisesti definiitti matriisi. $\boldsymbol{M}^T$ ja $\boldsymbol{M}^T\boldsymbol{M}$ käyttävät molemmat harvan matriisin tallennusmuotoa, ja ne luodaan vain kerran. $\boldsymbol{M}^T\boldsymbol{M}$ kopioidaan näytönohjaimen muistiin, ja koska matriisia $\boldsymbol{M}$ ei sellaisenaan enää tarvita, sen varaama muisti vapautetaan.

Yhtälöryhmät (3.19) ratkaistaan GPU:lla käyttäen iteratiivista menetelmää, joten iteraatioiden suppenevuuden parantamiseksi käytetään pohjustinoperaatiota. Pohjustinoperaatioon käytettävä pohjustinmatriisi muodostetaan matriisista $\boldsymbol{M}^T\boldsymbol{M}$, joten myös tämä suoritetaan vain kerran. Empiirisesti havaittiin, että iteraatioiden nopein suppenevuus taataan käyttämällä ViennaCL-kirjaston vaihtoehdoista symmetrisille, positiivisesti definiiteille matriiseille soveltuvaa Chow-Patel-IChol0-pohjustinmatriisia.

Seuraavaksi totetuksessa käydään läpi kaikki aallonpituusalueet Λ_m muodostaen keskussuoritinta käyttäen vektorin $\boldsymbol{\delta}_m$ ja määrittäen vektorin $\boldsymbol{s} \circ \boldsymbol{\delta}_m$. Jokaiselle aallonpituusalueelle

määritetään myös tulo $\boldsymbol{M}^T \boldsymbol{s} \circ \boldsymbol{\delta}_m$, joka kopioidaan näytönohjaimen muistiin. Määritettyjen matriisien ja vektorien avulla jokaiselle aallonpituusalueelle ratkaistaan GPU:ta käyttäen pienimmän neliösumman menetelmän yhtälöryhmä $\boldsymbol{M}^T \boldsymbol{M}\, \boldsymbol{z}_m = \boldsymbol{M}^T \boldsymbol{s} \circ \boldsymbol{\delta}_m$. Yhtälöryhmät ratkaistaan käyttäen iteratiivista konjugaattigradienttimenetelmää, joka soveltuu symmetrisille, positiivisesti definiiteille kerroinmatriiseille. Empiirisillä kokeilla havaittiin 100 iteraation riittävän tarpeeksi tarkkaan ratkaisuun, jonka jälkeen tulos ei enää silmin nähden muutu.

Kuten Matlab-toteutuksessa, syvyysarvot lisäksi normalisoidaan, ja kirjoitetaan yksinkertaisen tarkkuuden liukulukuja käyttäen eri aallonpituusalueita vastaaviin syvyyskarttatiedostoihin.

4 Hyödyntäminen tietokonegrafiikassa

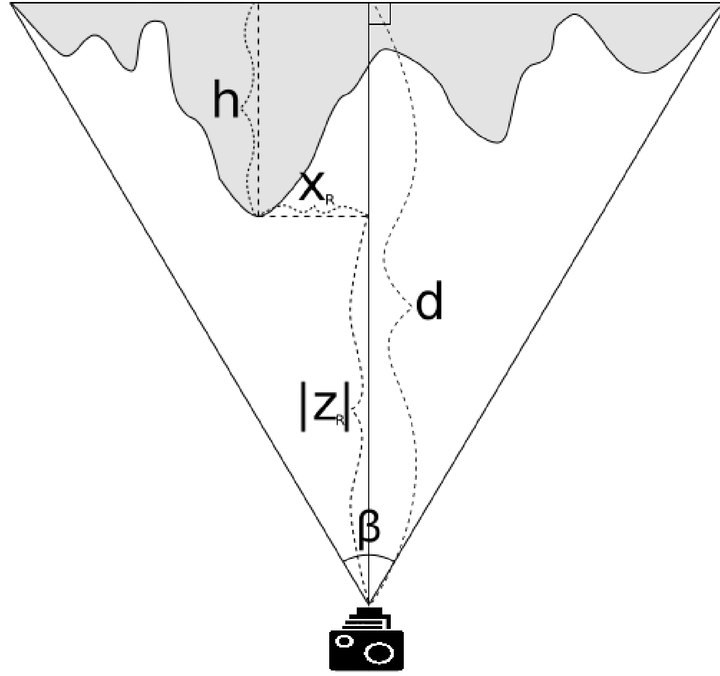
Tässä tutkimuksessa tarkasteltiin myös hyperspektrikuvien avulla toteutetun fotometrisen stereon hyödyntämistä tietokonegrafiikassa ja hyperspektrikuvantamisen tuomaa kuvanlaadullista lisäarvoa. Hyperspektrikuvista muodostettujen albedo-, normaali- ja syvyyskarttojen avulla kuvattavasta kohteesta muodostettiin malli, josta edelleen renderöitiin kuvia reaaliaikaisesti. Renderöinnin yhteydessä mallin valaisua simuloitiin käyttäen erilaisia virtuaalisia valonlähteitä. Nämä valonlähteet erosivat toisistaan niiden säteilemien spektrien osalta.

4.1 Renderöijän toteutus

Kuvattavasta kohteesta muodostetaan malli luomalla kahdesta kolmiosta nelikulmiota vastaava geometria ja asettamalla se kohtisuorasti kameran suuntaa vastaan siten, että se ortogonaalisella projektiomatriisilla renderöitynä, täyttää 1920×1200 -kokoisen kuva-alueen. Geometria teksturoidaan hyperspektrikuvista muodostetuilla albedo-, normaali- ja syvyyskartoilla siten, että tekstuurien vasemman alakulman kuvapiste sijoittuu renderöitävän kuva-alueen vasempaan ala-kulmaan. Vastaavasti tekstuurien oikean yläkulman kuvapiste sijoittuu renderöitävän kuva-alueen oikeaan yläkulmaan.

Jokaiselle hyperspektrikuvan aallonpituusalueelle on omat albedo-, normaali- ja syyvyykarttansa. Yhtä aallonpituusaluetta vastaava albedokartta eroaa tavanomaisesta siinä, että sen elementit ovat skalaareja, eivätkä kolmekomponenttisia RGB-vektoreita. Kaikkien aallonpituusalueiden kartat voidaan ajatella myös yhdistettynä niin, että albedo-ja syvyyskarttojen elementit ovat 133-komponenttisia vektoreita ja normaalikartan elementit 133×3 -kokoisia matriiseja.

Kohteen pinnan kuvakoordinaatit muutetaan renderöinnin yhteydessä reaalimaailmaa vastaavaan koordinaatistoon ottaen samalla huomioon kameran perspektiivin. Reaalimaailmaa vastaavassa koordinaatistossa kamera on sijoitettu origoon, ja koordinaattiakselien suunnat vastaavat edelleen oikeakätistä koordinaatistoa. Kuvakoordinaatit skaalataan ensin välille $[-1, 1]$ sekä vaaka- että pystysuunnassa, mikä yksinkertaistaa niiden suhdetta reaalimaailman koordinaatteihin.



Kuvio 7. Kohteen pinnan kuvakoordinaattien muuntaminen reaalimaailmaa vastaavaan koordinaatistoon x- ja z-suunnissa.

Jos kuvapisteen vastaavaa kohteen pinnan paikkaa reaalimaailman koordinaatistossa merkitään koordinaateilla (x_R, y_R, z_R), niin kuviosta 7 nähdään, että kuvion merkinnöillä

$$z_R = -d + h, \qquad x_R = |z_R| \tan(\frac{\beta}{2})\, x_s, \tag{4.1}$$

missä x_s on edellä mainittu välille $[-1, 1]$ skaalattu vaakasuuntainen kuvakoordinaatti ja h edellisessä luvussa muodostetun syvyyskartan kuvapisteen arvo skaalattuna reaalimaailman koordinaatistoon. Reaalimaailman y-suuntaiselle koordinaatille saadaan täysin vastaavasti

$$y_R = |z_R| \tan(\frac{\gamma}{2})\, y_s, \tag{4.2}$$

missä γ on kameran pystysuuntaisen kuvakulman suuruus. Syvyyskartan kuvapisteen arvo ja siten myös h riippuvat tarkasteltavasta aallonpituusalueesta Λ_m. Tällöin myös reaalimaailman koordinaatit riippuvat siitä, jolloin koordinaatteja merkitään $(x_{R_m}, y_{R_m}, z_{R_m})$. Reaalimaailman koordinaatiston etäisyyden yksiköksi on valittu cm.

Mallin valaisun simulointia varten virtuaalisten valonlähteiden spektrit diskretoitiin jakamalla koko hyperspektrikameran kattama aallonpituusalue 133 yhtä suureen osaan. Saapuva

spektraalinen irradianssi asetettiin vakioksi jokaisella näin luodulla osavälillä. Spektraalisen irradianssin arvo yhdellä osavälillä on yhtä suuri kuin spektraalinen irradianssi sen osavälin alussa, eli m. osavälin spektraaliselle irradianssille pätee

$$E_{\lambda_0}(\lambda) = E_{\lambda_0}(\lambda_m) =: E_{0_m}, \quad \text{kun } \lambda \in \Lambda_m = [\lambda_m, \lambda_m + \Delta\lambda_m], \tag{4.3}$$

missä $\Delta\lambda_m$ on aallonpituusalueen Λ_m leveys. Tutkimuksessa on toteutettu kaksi eri spektrin omaavaa pistemäistä valonlähdettä.

Tutkimuksen toteutuksessa simuloidaan tyypillistä kameraa, joka omaa kolme värikanavaa: sinisen, vihreän ja punaisen. Jokaisen värikanavan spektraalista herkkyyttä mallinnetaan normaalijakaumalla. Eri värikanavien normaalijakaumilla on eri odotusarvot ja keskihajonnat, jotka valittiin suurpiirteisesti vertaamalla oikeiden kameroiden spektraalisiin herkkyyksiin. Värikanavien spektraaliset herkkyydet diskretoitiin vastaavasti kuin valonlähteiden irradianssit

$$H_k(\lambda) = H_k(\lambda_m) =: H_{k_m}, \quad \text{kun } \lambda \in \Lambda_m = [\lambda_m, \lambda_m + \Delta\lambda_m], \tag{4.4}$$

missä $k = 1,2,3$. $k = 1$ vastaa sinistä värikanavaa, jolle $m = 1,\dots,38$, $k = 2$ vihreää värikanavaa, jolle $m = 18,\dots,53$ ja $k = 3$ punaista värikanavaa, jolle $m = 48,\dots,93$. Värikanavat ovat siis aallonpituuden suhteen osittain päällekkäin.

Simuloituun kameraan saapuva, spektraalisella herkkyydellä painotettu värikanavan irradianssi $\hat{L}_k := \int_{\Lambda_k} HL_{\lambda_r} d\lambda$ saadaan yhtälöstä 2.25 kertomalla molemmat puolet kameran spektraalisella herkkyydellä ja integroimalla puolittain:

$$\hat{L}_k = \int_{\Lambda_k} H \frac{a_\lambda}{\pi} E_{\lambda_0} \boldsymbol{l} \bullet \boldsymbol{n} \, d\lambda. \tag{4.5}$$

Tämä määritetään numeerisesti käyttämällä edellä määritettyä diskretoitua spektraalista herkkyyttä sekä diskretoitua valonlähteen spektraalista irradianssia ja muuntamalla integraali summaksi

$$\hat{L}_k = \sum_{m=m_{\min}(k)}^{m_{\max}(k)} H_{k_m} \frac{a_m}{\pi} E_{0_m} \boldsymbol{l}_m \bullet \boldsymbol{n}_m \, \Delta\lambda_m, \tag{4.6}$$

missä

$$\boldsymbol{l}_m = \frac{\boldsymbol{l}_{\text{pos}} - [x_{R_m}, y_{R_m}, z_{R_m}]^T}{|\boldsymbol{l}_{\text{pos}} - [x_{R_m}, y_{R_m}, z_{R_m}]^T|}. \tag{4.7}$$

$\boldsymbol{l}_{\text{pos}}$ on pistemäisen valonlähteen paikka aiemmin määritellyssä reaalimaailman koordinaatistossa.

4.1.1 Simuloidut valonlähteet

Tutkimuksessa on toteutettu kaksi eri spektrin omaavaa, joka suuntaan yhtä paljon säteilevää pistemäistä valonlähdettä. Valonlähteistä ensimmäinen simuloi auringon ja toinen hehkulampun säteilemää spektriä.

Simuloidut valonlähteet säteilevät mustan kappaleen tavoin noudattaen Planckin lakia. Auringon valoa simuloivaa valonlähdettä vastaavan mustan kappaleen absoluuttinen lämpötila on $T_a = 5500$ K. Simuloidun hehkulampun vastaava lämpötila on $T_h = 2700$ K.

Valonlähteiden kaikki – eri aallonpituusalueita Λ_m vastaavat – 133 spektraalista irradianssia lasketaan käyttäen skaalattua Planckin lain lauseketta

$$E_{0_m}(\lambda_m) = \frac{8\pi hc}{\lambda_m^5} \frac{1}{e^{hc/\lambda_m k_B T} - 1} k_s, \tag{4.8}$$

missä k_s on tutkimuksen toteutuksessa käytetty skaalauskerroin, h Planckin vakio, c valonnopeus, k_B Boltzmannin vakio ja T mustan kappaleen absoluuttinen lämpötila, joka saa arvoksi T_a tai T_h.

4.1.2 Grafiikkaliukuhihnan sovellusvaihe

Hyperspektrikuvista muodostettujen karttojen renderöijän keskussuorittimella suoritettavan sovellusvaiheen toteutukseen käytettiin C++-ohjelmointikieltä ja OpenGL-grafiikkaohjelmointikirjaston versiota 4.1. Toteutuksen lähdekoodi on liitteenä G.

Tarkoituksena oli toteuttaa reaaliaikainen renderöijä, ja empiiristen kokeiden avulla havaittiin, että se ei suorituskykyongelmien vuoksi ole 32-bittisiä tekstuurien värikanavia käyttäen mahdollista. 8-bittiset värikanavat sen sijaan hävittävät niin paljon tietoa, että tutkimuksen mielekkyys kyseenalaistuu. Näin ollen parhaimman kuvanlaadun ja suorituskyvyn tasapainon takaamiseksi OpenGL-tekstuurit luodaan käyttäen 16-bittisiä värikanavia. Muistin ja muistikaistan tarpeen vähentämiseksi, yhtä hyperspektrikuvan siivua vastaava albedo-, normaali- ja syvyyskartta pakataan kaikki samaan tekstuuriin. Tekstuurissa on neljä kanavaa

yhtä kuvapistettä kohden, jolloin yksi kuvapiste varaa 64 bittiä muistia. Tekstuurin kahteen ensimmäiseen kanavaan tallennetaan normaalivektorin x- ja y-komponentit, kolmanteen kanavaan albedo ja viimeiseen syvyys. Kanavat on tallennettu puolitarkkuuden liukulukuina.

Kaikki 133 edellä kuvatun kaltaista tekstuuria varaavat yhteensä yli 2,25 gigatavua muistia. Renderöijän toteuttamiseen ja testaamiseen käytetyn tietokoneen näytönohjaimessa oli muistia vain kaksi gigatavua. Tästä syystä renderöijä toteutettiin siten, että vain yksi kolmesta värikanavasta renderöidään kerrallaan. Yhtä värikanavaa kohden käytetään maksimissaan 45 tekstuuria, ja niistä luodaan tekstuuritaulukko. Jokaista eri värikanavaa renderöitäessä käytetään samaa kohdetekstuuria, mutta sen eri kanavaa, jolloin kolmen renderöintikerran lopputuloksena on kaikki värikanavat omaava kuva.

Ennen renderöinnin aloittamista jokaisen kolmen värikanavan vastaavat kameran spektraaliset herkkyydet alustetaan omiin taulukoihinsa. Herkkyydet lasketaan käyttäen normaalijakauman lauseketta. Odotusarvona käytetään värikanavalle valittujen aallonpituusalueiden keskimmäisen alueen spektraalisen herkkyyden huipun aallonpituutta. Normaalijakaumien keskihajonnat valittiin siten, että integroitaessa spektraalisia herkkyyksiä värikanaville valittujen aallonpituusalueiden yli numeerisesti tulokset ovat mahdollisimman lähelle yksi. Taulukoihin tallennettavat spektraalisen herkkyyden arvot kerrotaan lisäksi niitä vastaavien aallonpituusalueiden leveyksillä. Tällöin yhtälön (4.6) kerroin $\Delta\lambda$ on jo sisällytetty spektraaliseen herkkyyteen, eikä tuloa tarvitse laskea enää varjostinohjelmassa.

Myös simuloitujen valonlähteiden eri aallonpituusalueita vastaavat irradianssit alustetaan taulukohin siten, että jokaisella kolmella värikanavalla on oma taulukkonsa. Lähteiden irradianssit skaalataan empiirisesti sopiviksi havaituilla kertoimilla niin, että renderöinnin tuloksena syntyvät väriarvot ovat välillä $[0,1]$.

Alustusten jälkeen ohjelma renderöi jatkuvasti uusia kuvia suurimmalla nopeudella, johon ohjelmaa suorittava tietokone kykenee. Jokaisella renderöintikierroksella lasketaan simuloidun valonlähteen uusi paikka, jonka jälkeen laskettu paikka kopioidaan näytönohjaimen muistiin. Liitteenä olevassa versiossa valonlähde on asetettu kiertämään xy-tasolla ympyrää, jonka säde on 20 cm ja keskipiste $(0,0,-10)$, reaalimaailman koordinaateissa ilmaistuna. Lisäksi renderöintikierroksen aikana punaisen värikanavan spektraaliset herkkyydet, saapu-

vat irradianssit ja tekstuuritaulukko kopioidaan näytönohjaimen muistiin sekä annetaan näytönohjaimelle käskyt renderöidä punainen värikanava. Nämä vaiheet toistetaan myös ensin vihreälle ja sen jälkeen siniselle värikanavalle.

4.1.3 Varjostinohjelmien toteutus

Renderöijän grafiikkasuorittimella suoritettava osuus toteutettiin käyttäen OpenGL-kirjaston varjostinohjelmakieltä GLSL ja sen versiota 4.10. Toteutukset ovat liitteinä I ja H.

Renderöijä renderöi aina ainoastaan yhden mallin eikä kameraa ole mahdollista liikuttaa. Tästä syystä toteutettu verteksivarjostinohjelma on erittäin yksinkertainen, eikä se suorita grafiikkaliukuhihnan malli- ja näkymämuunnosvaihetta ollenkaan. Verteksivarjostinohjelma toteuttaa grafiikkaliukuhihnan verteksivarjostusvaiheen saamalla syötteenä yhden mallin verteksin kaksiulotteisissa koordinaateissa ilmaistuna ja laskemalla sen avulla sitä vastaavat tekstuurikoordinaatit. Tämän jälkeen ohjelma palauttaa lasketut tekstuurikoordinaatit ja syötteenä saadun verteksin sellaisenaan sekä muunnettuna nelialkioiseksi, homogeenisissa koordinaateissa ilmaistuksi vektoriksi. Verteksivarjostinohjelma ei suorita myöskään projektiovaihetta vaan asettaa homogeenisten koordinaattien w-komponentin ykköseksi, jolloin grafiikkasuoritin suorittaa käytännössä ortogonaaliprojektion. Verteksivarjostinohjelma suoritetaan yhtäaikaisesti mallin kuudelle eri verteksille.

Grafiikkaliukuhihnan kuvapistevarjostusvaihe on toteutettu kuvapistevarjostinohjelmassa, joka saa syötteenä verteksivarjostinohjelman palauttamat interpoloidut verteksin kaksiulotteiset paikkakoordinaatit ja tekstuurikoordinaatit. Näiden lisäksi se pystyy lukemaan näytönohjaimen muistiin kopioidut tekstuurit, valonlähteen paikkakoordinaatit ja irradianssit sekä kameran spektraalisen herkkyyden arvot.

Ohjelma käy läpi kaikki kyseisellä suorituskerralla renderöitävän värikanavan vastaavat aallonpituusalueet. Jokaisella iteraatiolla tekstuurista luetaan tekstuurikoordinaattia ja aallonpituusaluetta vastaava nelialkioinen kuvapisteen arvo. Arvon kaksi ensimmäistä alkiota ovat luvussa 3.3.3 normalisoidut normaalivektorin x- ja y-komponentit, jotka skaalataan ja siirretään kuvapistevarjostinohjelmassa takaisin välille $[-1, 1]$. Normaalivektorin z-komponentti lasketaan x- ja y-komponenteista käyttäen tietoja, että vektorin pituus on yksi sekä z-komponentti

positiivinen. Kolmas alkio eli albedo luetaan sellaisenaan, mutta viimeisen alkion normalisoitu syvyysarvo skaalataan takaisin kuvakoordinaatistoon. Syvyysarvoa, syötteenä saatuja paikkakoordinaatteja ja yhtälöitä (4.1) sekä (4.2) käytten lasketaan kohteen pinnan vastaavan pisteen reaalimaailman koordinaatit. Näistä koordinaateista ja valon sijainnista määritetään yhtälön (4.7) avulla valonlähteen suuntavektori. Iteraatiota vastaavan aallonpituusalueen irradianssi luetaan näytönohjaimen muistista. Tämän jälkeen lasketaan aallonpituusaluetta vastaava spektraalisella herkkyydellä painotettu radianssi eli summan (4.6) yksi termi. Silmukan lopussa termi lisätään kokonaissummaan, joka palautetaan koko kuvapistevarjostinohjelman suorituksen päätteeksi. Kuvapistevarjostinohjelma suoritetaan kaikille renderöitävän kuvan kuvapisteille yhtäaikaisesti.

Koska renderöitävä geometria muodostuu kahdesta aina kokonaan näkvyissä olevasta kolmiosta, grafiikkaliukuhihnan yhdistämisvaihe on konfiguroitu siten, että kaikkien kuvapisteiden värit kirjoitetaan ilman näkyvyystarkistusta aina väripuskuriin. Lisäksi värit ylikirjoittavat aina väripuskurissa jo valmiina olevat värit eikä niitä sekoiteta keskenään. Kuten luvussa 4.1.2 esitettiin, yksi renderöintikerta ylikirjoittaa kuitenkin vain yhden värikomponentin (punaisen, vihreän tai sinisen) jättäen väripuskurissa jo olevat muut komponentit koskemattomiksi.

4.1.4 Kontrollikuvat

Jotta hyperspektrikuvista muodostettujen albedo-, normaali- ja syvyyskarttojen avulla renderöityjen kuvien laatua ja näin ollen hyperspektrikuvantamisen tuomaa lisäarvoa pystyttiin arvioimaan, oli oltava vertailukohtia. Tätä tarkoitusta varten renderöitiin niin kutsuttuja *kontrollikuvia*.

Kontrollikuvien renderöintiä varten hyperspektrikuvat muunnettiin tavallisia kuvia vastaaviksi kolmen värikanavan kuviksi. Muuntaminen suoritettiin käyttäen luvussa 4.1 esitettyä virtuaalista kameraa. Hyperspektrikuvien arvot tulkittiin pinnalta heijastuneina spektraalisina radiansseina, ja ne painotettiin kameran värikanavien spektraalisilla herkkyyksillä. Painotetut spektraaliset radianssit integroitiin numeerisesti värikanavien aallonpituusalueiden Λ_k yli, kertomalla ne hyperspektrikuvien aallonpituusalueiden Λ_m leveyksillä $\Delta\lambda_m$ ja summaa-

malla yhteen:

$$L_{\mathrm{ctrl}_k} = \sum_{m=m_{\min}(k)}^{m_{\max}(k)} H_{k_m} \hat{L}_m \, \Delta\lambda_m. \tag{4.9}$$

Näin muodostettujen - kolmen värikanavan kameraa vastaavien - kuvien avulla luotiin lukujen 3.3 ja 3.4 menetelmillä albedo-, normaali- ja syvyyskartat. Jokaista yksi kappale kutakin värikanavaa kohden. Kartat luotiin käyttäen luvun 3.3.5 fotometrisen stereon säikeistettyä CPU-toteutusta ja luvun 3.4.3 syvyyskarttojen muodostamisen GPU-toteutusta, käyttämällä allonpituusalueiden lukumääränä luvun 133 sijasta lukua kolme.

Kontrollikuvien renderöijän alustuksessa myös valonlähteen spektraalinen irradianssi integroidaan numeerisesti. Kaikki yhtä värikanavaa vastaavat spektraaliset irradianssit summataan aallonpituusalueiden leveyksillä ja spektraalisilla herkkyyksillä painotettuna yhteen:

$$E_{0_k} = \sum_{m=m_{\min}(k)}^{m_{\max}(k)} H_{k_m} E_{0_m} \, \Delta\lambda_m. \tag{4.10}$$

Muodostetuista kartoista renderöidään kuvia luvussa 4.1 kuvatulla tavalla käyttäen aallonpituusalueiden lukumääränä lukua kolme, jokaista värikanavaa vastaavana spektraalisena herkkyytenä lukua yksi ja värikanavien vastaavina irradiansseina yhtälön (4.10) tuloksia. Toteutuksen lähdekoodi on yhdistettynä hyperspektraalisen renderöijän lähdekoodiin ja on siis liitteinä G ja I.

5 Tulokset

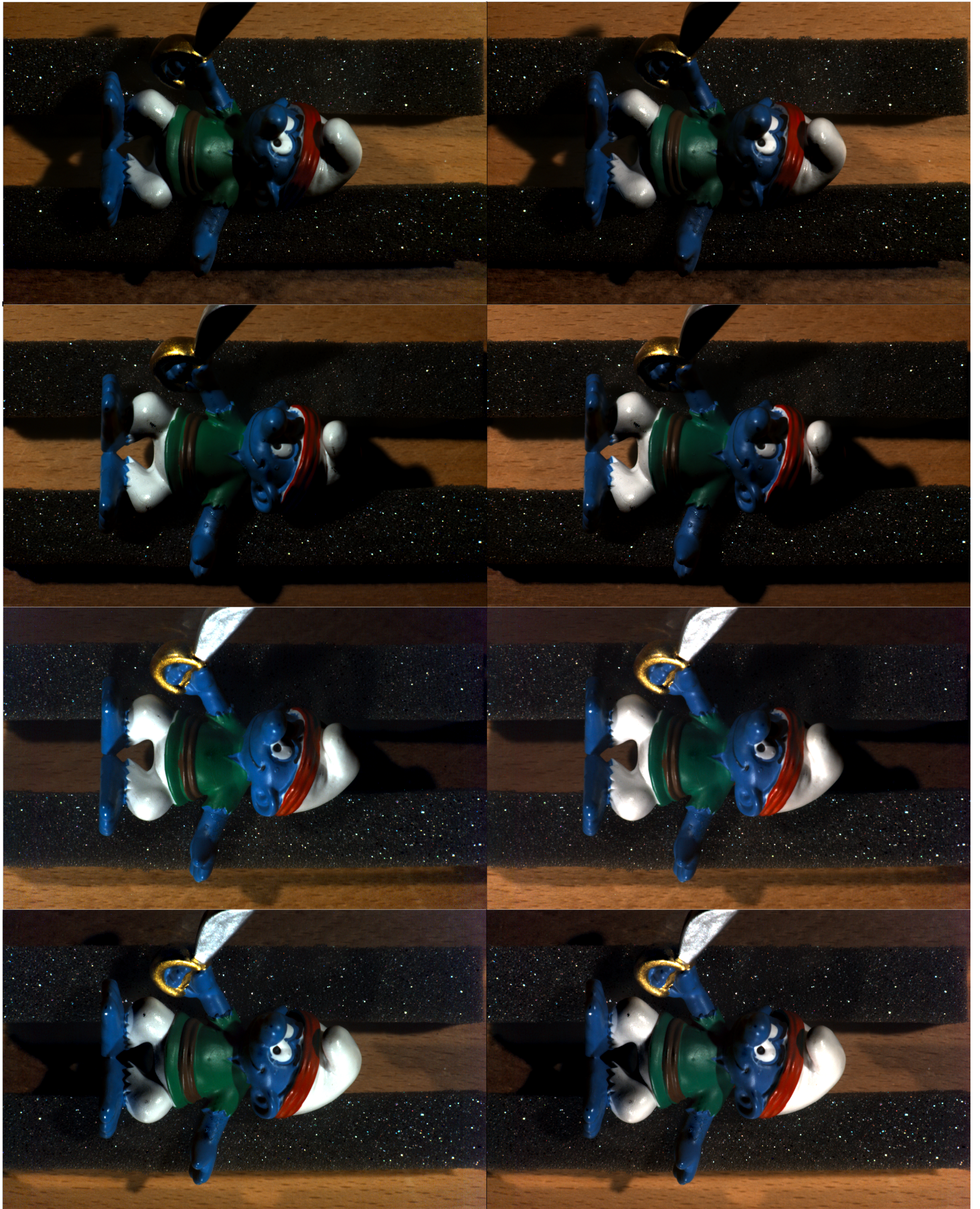
5.1 Hyperspektrikuvantamisen vaikutus kuvanlaatuun

Hyperspektrikuvantamisen tietokonegrafiikkaan tuomaa lisäarvoa tutkittiin arvioimalla subjektiivisesti sen vaikutusta kuvanlaatuun sekä mittaamalla eri värikanavien histogrammien tilastollisia tunnuslukuja. Arviointi ja mittaukset suoritettiin vertaamalla hyperspektrikuvien avulla renderöityjä kuvia luvun 4.1.4 menetelmällä renderöityihin kontrollikuviin.

Kuvion 8 vasemmanpuoleisessa sarakkeessa on näkyvissä hyperspektrikuvien avulla renderöidyt kuvat ja oikeanpuoleisessa kontrollikuvat. Kuvia renderöitäessä on simuloitu auringon säteilemän spektrin omaavaa pistemäistä valonlähdettä, joka kiertää 20 cm:n säteistä ympyrää, yhtälön $z = -10$ määrittämällä xy-suuntaisella tasolla. Kuvion ensimmäisellä rivillä valonlähteen atsimuuttikulma ϕ on positiivisesta x-akselista positiivisen y-akselin suuntaan mitattuna $\frac{\pi}{4}$, toisella $\frac{3\pi}{4}$, kolmannella $\frac{5\pi}{4}$ ja viimeisellä $\frac{7\pi}{4}$ radiaania.

Kuviosta 8 nähdään, että ero hyperspektrikuvien avulla renderöityjen kuvien ja kontrollikuvien välillä on hyvin pieni. Kuitenkin, jos kuvia verrataan alkuperäiseen kohteeseen, ovat hyperspektrikuvien avulla renderöityjen kuvien värit luonnollisemmat ja vastaavat tarkemmin alkuperäistä. Tämä on havaittavissa erityisesti alustana olevan pöydän, hahmon päällä olevan paidan ja lakkia reunustavan nauhan sävyeroina. Taulukossa 1 on nähtävissä renderöityjen kuvien eri värikanavien histogrammien keskiarvot $\bar{x}$ ja keskihajonnat σ_x, kun värikanavien arvot on ilmaistu välillä [0, 255]. Kontrollikuvien punaisen värikanavan keskiarvo ja keskihajonta ovat hyperspektraalisien kuvien vastaavia arvoja huomattavasti suuremmat. Siniseen päin siirryttäessä arvot kuitenkin lähenevät toisiaan ollen lopulta lähes samat. Tästä nähdään, että kontrollikuvat ovat hyperspektraalisia kuvia keskimäärin punasävyisempiä. Samat erot on nähtävissä myös virtuaalista hehkulamppua käyttäen renderöidyissä kuvissa.

Kuviossa 8 on havaittavissa sarakkeiden välillä pieniä eroavaisuuksia myös simuloidun valonlähteen aiheuttamissa varjostuksissa. Eroavaisuudet johtuvat siitä, että hyperspektraaliset normaali- ja syvyyskartat eroavat kontrollikuvien vastaavista kartoista. Hyperspektraalisten karttojen avulla renderöityjen kuvien varjostukset vastaavat tarkemmin reaalimaailmassa



Kuvio 8. Hyperspektrikuvien avulla renderöidyt kuvat verrattuna kontrollikuviin.

	Hyperspektraalinen	Kontrolli
Punainen	38,1±45,8	43,2±51,7
Vihreä	29,6±35,2	31,0±36,6
Sininen	18,7±29,8	20,2±29,8

Taulukko 1. Renderöityjen kuvien histogrammien keskiarvot $\bar{x}$ ja keskihajonnat σ_x, muodossa $\bar{x} \pm \sigma_x$.

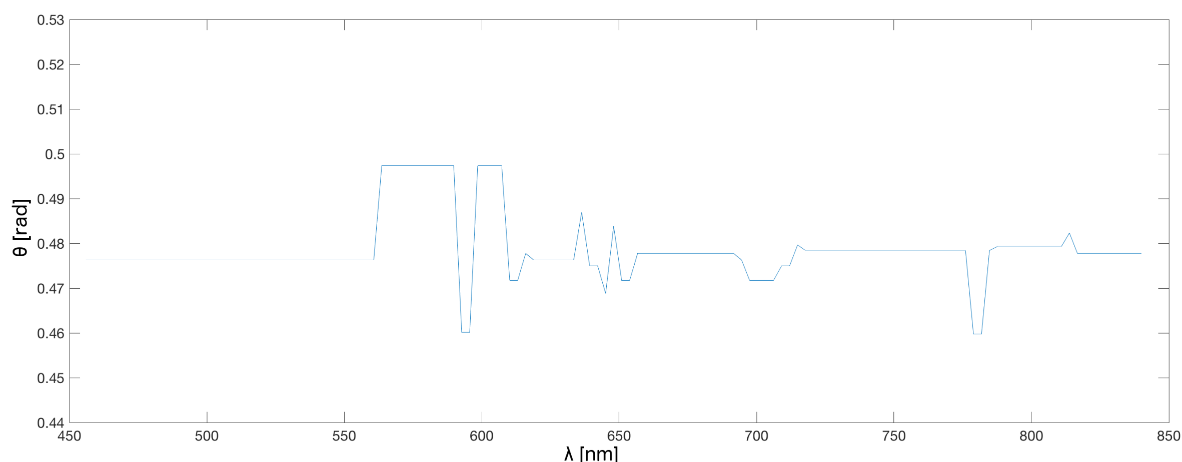
syntyviä varjostuksia. Nämä erot ovat kuitenkin niin pieniä, että ne voi havaita käytännössä ainoastaan vertailemalla yksittäisten kuvapisteiden numeerisia väriarvoja.

Kuviossa 9 on esitetty Smurffi-hahmon ja jääkaappimagneetin hyperspektrikuvien avulla tuotettujen syvyyskarttojen 70. aallonpituusalueesta renderöidyt kuvat.

Kuvio 9. Syvyyskarttojen 70. aallonpituusalueesta renderöidyt kuvat.

5.2 Orientaatiot aallonpituuksien funktioina

Spekulaarisesti heijastavien kohteiden tapauksessa valo heijastuu kaikilla aallonpituuksilla jo kohteen uloimmalta pinnalta, jolloin valo ei pääse siroamaan satunnaisesti kohteen sisärakenteissa. Näin ollen jos kohteen pinta on lisäksi sileä, eivät heijastumissuunnat riipu aallonpituudesta. Tästä syystä spekulaarisesti heijastavan kromipallon avulla määritettyjen valonlähteiden suuntien tulisi olla vakioita koko hypespektrikuvien kattaman kokonaisaallonpituusalueen yli. Tutkimuksessa valonlähteiden suunnissa kuitenkin havaittiin pientä vaihtelua eri aallonpituusalueiden Λ_m välillä. Tämä on nähtävissä esimerkiksi kuviossa 10 esitetyn valonlähteen 2 elevaatiokulmassa.



Kuvio 10. Valonlähteen 2 elevaatiokulma aallonpituuden suhteen.

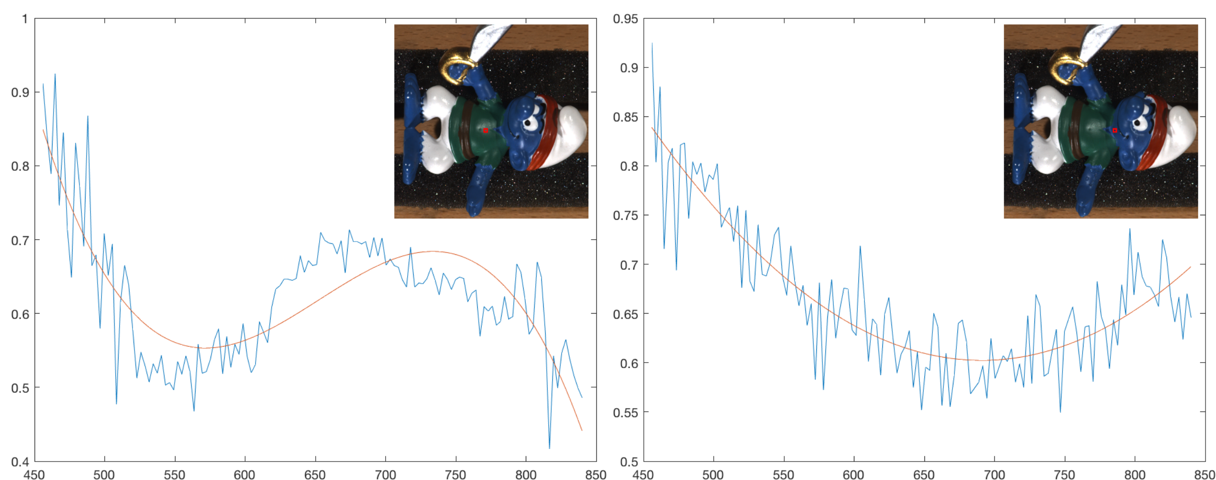
Kuviosta 10 on kuitenkin havaittavissa, että vaihtelu on satunnaista eikä noudata mitään trendiä. Näin ollen vaihtelut tulkittiin tilastollisina mittausvirheinä ja valojen suuntien elevaatio- ja atsimuuttikulmille laskettiin keskivirheet $s_{\bar{x}}$, jotka ovat nähtävissä taulukossa 2.

	Valo 1	Valo 2	Valo 3
θ-kulman $s_{\bar{x}}$	0,0014	0,0006	0,0013
ϕ-kulman $s_{\bar{x}}$	0,0036	0,0019	0,0039

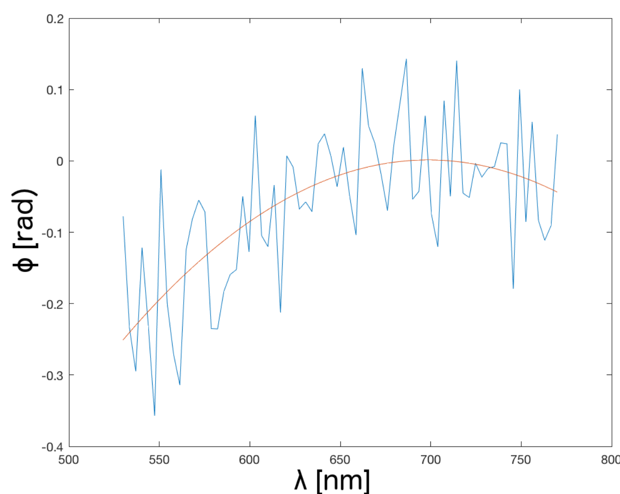
Taulukko 2. Valojen suuntien elevaatio- ja atsimuuttikulmien keskivirheet radiaaneina.

Myös tutkittavien kohteiden normaalivektoreiden aallonpituusriippuvuuksia tarkasteltiin. Kuvio 11 esittää Smurffi-hahmon avulla määritettyjen, kahden eri kuvapisteen normaalivekto-

reiden elevaatiokulmat aallonpituuden suhteen ja niihin sovitetut polynomifunktiot. Vasemmalla puolella oleva kuvaaja vastaa kuvapistettä, jonka kuvakoordinaatit ovat (850, 600) ja joka sijaitsee hahmon paidassa ollen näin väriltään vihreä. Oikean puolen kuvaaja vastaa kuvapistettä, jonka kuvakoordinaatit ovat (960, 600). Kuvapiste sijaitsee hahmon kasvoissa, joten se on väriltään sininen. Kuviossa 12 on esitetty myös kuvakoordinaatteja (850, 600) vastaavan kuvapisteen atsimuuttikulman riippuvuus aallonpituudesta.



Kuvio 11. Kahden kuvapisteen normaalivektoreiden elevaatiokulmat aallonpituuden suhteen sekä kuvapisteiden sijainnit.



Kuvio 12. Kuvapisteen normaalivektorin atsimuuttikulma aallonpituuden suhteen.

Kuten kuvioista 11 ja 12 on nähtävissä, noudattavat normaalivektoreiden elevaatio- sekä atsimuuttikulmien muutokset selvästi tiettyjä säännönmukaisuuksia. Vasemmanpuoleiseen

elevaatiokulmien pistejoukkoon on sovitettu kolmannen asteen ja oikeanpuoleiseen toisen asteen polynomifunktio. Myös atsimuuttikulmien pistejoukkoon on sovitettu toisen asteen polynomifunktio. Näiden kolmen sovitetun mallin laatua mitattiin laskemalla niiden keskineliövirheiden neliöjuuret eli RMSE:t (root-mean-square error), jotka on esitetty taulukossa 3.

	θ	ϕ
Kuvapiste (850, 600)	0,0525	0,0807
Kuvapiste (960, 600)	0,0353	

Taulukko 3. Normaalivektoreiden elevaatio- ja atsimuuttikulmien keskineliövirheiden neliöjuuret.

5.3 Toteutuksien suoritusajat

Suoritusaikamittaukset suoritettiin Apple iMac (Retina 5K, 27-inch, Late 2015) -tietokoneella, jossa on 3,3GHz Intel Core i5 -keskussuoritin, 8 Gt 1867 MHz DDR3 -muistia ja AMD Radeon R9 M395 -grafiikkasuoritin. Keskussuorittimessa on neljä ja grafiikkasuorittimessa 1792 ydintä. Grafiikkasuorittimella on käytössä 2048 Mt grafiikkamuistia. Mittaukset suoritettiin jokaiselle kohteelle kolme kertaa, jolloin mittauspisteitä muodostui yhdeksän kappaletta jokaista mitattavaa arvoa kohden. Mittauksille laskettiin keskiarvot ja keskivirheet.

Taulukossa 4 on esitetty albedo-, normaali- ja syvyyskarttojen tuottamisen eri vaiheiden ja toteutuksien suoritusaikamittauksien tulokset. Syvyyskartan luomisen suoritusajat vastaavat hyperspektraalisen syvyyskarttakuution, yhden siivun määrittämiseen käytettyä aikaa. Näin ollen kun jokaisella mittauskerralla määritettiin 133 siivua, kertyi syvyyskarttojen luomisen suoritusajoille $133 * 9 = 1197$ mittauspistettä. Tällöin myös vaiheen kokonaissuoritusaikaa määritettäessä tulee taulukossa esitetty tulos kertoa luvulla 133. Syvyyskartan luominen ei ollut mahdollista Armardillo-kirjaston avulla, joten sen tulos puuttuu. Tulokset on esitetty muodossa keskiarvo $\pm$ keskivirhe.

Hyperspektrikuvien jakaminen luvulla $\hat{E}_m$ oli nopeinta Matlab-toteutuksen avulla. Erot eri toteutuksien välillä olivat kuitenkin hyvin pieniä ja keskivirheet huomioon ottaen Matlab- ja Armadillo-toteutuksien suoritusajat olivat lähes samat. Fotometrisen stereon suoritusajat

	Matlab (CPU, 1 ydin)	Armadillo (CPU, 4 ydintä)	ViennaCL (GPU)
Jako luvuilla $\hat{E}_m$	0,42±0,08	0,59±0,08	0,71±0,04
Fotometr. stereo	321,74±11,01	329,09±9,89	3553,66±112,20
Syvyyskartta	11,77±0,92		0,87±0,01

Taulukko 4. Eri vaiheiden suoritukseen eri toteutuksien avulla käytetyt ajat sekunteina.

olivat keskivirheet huomioon ottaen Matlab- ja Armadillo-toteutuksien tapauksessa käytännössä samat, mutta ViennaCL-toteutuksen suoritusaika oli yli kymmenkertainen niihin verrattuna. Syvyyskartan luomisessa ViennaCL-toteutus oli kuitenkin yli 13 kertaa niin nopea kuin Matlab-toteutus. Tämä on merkittävä ero kun otetaan huomioon koko hyperspektraalisen syvyyskarttakuution luominen, jonka suoritusaika Matlab:lla on noin 26 minuuttia ja ViennaCL:llä alle kaksi minuuttia.

Renderöijän suorituskykyä tutkittiin mittaamalla 1000 kuvan renderöintiajat ja laskemalla niiden keskiarvo ja keskivirhe. Näin mitattuna renderöijä pystyi hyperspektraalisia karttoja käyttäen renderöimään yhden kuvan keskimäärin $0,16 \pm 0,01$ sekunnissa, jolloin reaaliaikaisen renderöijän kuvataajuus oli noin 6,25 Hz. Kontrollikuvia käyttäen renderöijän kuvataajuus oli vakio 60 Hz, johon se oli ohjelmallisesti ylhäältäpäin rajoitettu.

5.4 Pohdinta

5.4.1 Fotometrinen stereo ja hyperspektrikuvantaminen tietokonegrafiikassa

Fotometrisen stereon havaittiin olevan hyvin toteutettuna erittäin käytännöllinen menetelmä tietokonegrafiikassa käytettävien mallien automaattiseen luomiseen reaalimaailman kohteista. Tässä tutkimuksessa käytetyssä perusmuodossaan se soveltuu kuitenkin ainoastaan tietynlaisten kohteiden mallintamiseen. Kuten luvun 5.1 kuviosta 9 on nähtävissä, aiheuttavat varjot ja kameran kuvaussuunnassa olevat muotojen epäjatkuvuuskohdat huomattavia virheitä normaali- ja syyvyyskarttoihin. Esimerkiksi Smurffi-hahmon lakin langettama varjo aiheuttaa hahmon kasvoihin kuopan ja jääkaappimagneetin jyrkkä alareuna on menetetty mallissa kokonaan. Kuvion 8 alimmalla rivillä taas on nähtävissä, kuinka tietystä suunnasta valaisemalla varjot ovat kääntyneet vastakkaisiksi. Alueet, jotka ylemmillä riveillä eivät ole

varjossa, ovat sitä alimmalla rivillä ja päinvastoin.

Jotta menetelmää voitaisiin käyttää menestyksekkäästi myös monimutkaisten muotojen omaavien kohteiden mallintamiseen, tulisi valonlähteitä ja otettuja kuvia olla useampia kuin kolme tai käytössä jokin muu kehittyneempi fotometrisen stereon versio, joka pyrkii ratkaisemaan nämä ongelmat. Myös vaatimus Lambertin pintaa vastaavasta heijastavuudesta on kierrettävissä monimutkaisempia reflektanssikarttoja ja kehittyneempiä fotometrisen stereon versioita käyttäen. Hyperspektrikuvantamisen avulla voitaisiin lisäksi huomioida kohteen sisäiset heijastukset, joissa pinnalle saapuu valoa muualtakin kuin suoraan valonlähteestä.

Hyperspektrikuvantamisen havaittiin tuovan tietokonegrafiikkaan kuvanlaadullista lisäarvoa, mutta erot olivat hyvin pieniä suhteessa menetelmän vaatimiin resursseihin. Näin ollen se ei ainakaan tämän tutkimuksen mittakaavassa ja nykyteknologialla toteutettuna ole mielekäs lisä reaaliaikaiseen renderöintiin. Hyperspektraalisia albedo-, normaali- ja syvyyskarttoja käyttäen renderöijä kykeni ainoastaan kymmenesosaan siitä kuvataajuudesta, jota suurin osa nykyaikaisista tietokonepeleistä ja animaatioista käyttää. Näin ollen se ei kyennyt luomaan uskottavaa liikkeen vaikutelmaa. 8-bittisiä värikanavia käyttämällä kuvataajuus saatiin kasvatettua lähes 60 Hz:iin, mutta se pienensi kuvanlaadussa saatuja hyötyjä entisestään.

Suorituskykyä rajoittavana tekijänä on hyperspektraalisten karttojen vaatima todella korkea muistikaistan käyttö. Grafiikkasuorittimen täytyy jokaista kuvaa renderöidessä lukea yli 40-kertainen määrän tekstuuridataa tavallisien karttojen avulla suoritettuun renderöintiin verrattuna. Suorituskykyä voisi näin ollen parantaa vähentämällä käytettyjen aallonpituusalueiden määrää, esimerkiksi jättämällä simuloidun kameran matalimpia spektraalisia herkkyyksiä vastaavat aallonpituusalueet kokonaan pois. Aallonpituusalueiden määrää voisi vähentää myös yhdistämällä vierekkäisiä aallonpituusalueita ennen renderöintiä suoritettavan esiprosessoinin avulla ja näin ollen pienentää spektraalista erottelukykyä. Muistikaistan käyttöä olisi mahdollista vähentää myös pienentämällä hyperspektraalista dataa omaavien kuvapisteiden lukumäärää. Näin voitaisiin tehdä esimerkiksi pienentämällä käytettyjen karttatekstuurien avaruudellista erottelukykyä tai rajoittamalla hyperspektraalisen tiedon soveltaminen ainoastaan niihin kuvapisteisiin, joissa siitä saadaan eniten hyötyä.

Hypersektrikuvantaminen tuo tietokonegrafiikkaan eniten hyötyjä tapauksissa, joissa mal-

linnettujen valonlähteiden säteilemät spektrit tai renderöitävien kohteiden spektraaliset sormenjäljet ovat yksityiskohtaisia ja monimutkaisia, jolloin ilman korkeaa spektraalista erottelukykyä hukataan tarvittavaa tietoa. Hyperspektraalisten karttojen tuoma hyöty on nähtävissä myös tapauksissa, joissa renderöitävien kohteiden materiaalit päästävät valoa osittain lävitseen ja läpäisemiskyky riippuu valon aallonpituudesta. Näiden lisäksi hyperspektrikuvantamista voi olla hyödyllistä käyttää tietokonegrafiikassa myös yleisemmissä tapauksissa jos käytettävän renderöijän ei tarvitse olla reaaliaikainen, jolloin suuret renderöintiajat eivät ole ongelma ja pienetkin kuvanlaadulliset parannukset toivottuja.

5.4.2 Hyperspektrikuvantaminen fotometrisessä stereossa

Kromipallon avulla määritetyissä valonlähteiden suunnissa olleet heilahtelut jäivät merkityksettömän pieniksi. Suuntien elevaatio- sekä atsimuuttikulmien keskivirheet olivat radiaanin tuhannesosan luokkaa, joten vaihtelut jätettiin huomiotta ja tutkimuksessa käytettiin ainoastaan keskiarvoja. Koska tutkimuksessa käytetty hyperspektrikamera kuvaa eri aallonpituusalueet eri ajanhetkillä, saattoivat suuntien vaihtelut johtua valonlähteiden, kameran ja/tai kromipallon pienistä heilahduksista hyperspektrikuvien ottamisen aikana.

Tutkittavien kohteiden hyperspektrikuvien avulla määritettyjen normaalivektoreiden heilahtelut taas noudattavat tiettyjä säännönmukaisuuksia. Aallonpituuden muuttuessa normaalivektori kääntyy jollakin välillä pääsääntöisesti samaan suuntaan. Tämän voidaan arvioida johtuvan siitä, että valo läpäisee kohteen materiaalin uloimman pinnan ja heijastuu takaisin vasta sisemmistä rakenteista. Täten jos valo tunkeutuu eri aallonpituuksilla eri syvyyksille, voi myös heijastuksen aiheuttaman syvyyden orientaatio muuttua aallonpituuden suhteen. Tulokset vastaavat jossain määrin myös oletuksia, koska tutkittavien kohteiden pinnat ovat suhteellisen sileitä, mutta heijastavat valoa silti lähes Lambertin pinnan tavoin, joten diffuusin heijastumisen on johduttava valon siroamisesta materiaalin sisärakenteissa. Normaalivektoreiden elevaatio- ja atsimuuttikulmissa on säännönmukaisten vaihteluiden lisäksi myös korkeataajuista kohinaa. Kohinan voidaan arvioida johtuvan ainakin osittain samoista syistä kuin valonlähteiden suuntien tapauksessa, mutta lisäksi materiaalin sisärakenteissa tapahtuvan valon siroamisen satunnaisuudesta.

Tutkittavien kohteiden hyperspektrikuvissa on valon diffuusin heijastumisen lisäksi havaittavissa myös spekulaarisia huippukohtia. Näin ollen tutkimuksessa käytetty oletus Lambertin pinnasta ei ole täysin oikeutettu ja se aiheuttaa luotuihin karttoihin sekä renderöityihin kuviin virheitä. Nämä ovat kuitenkin varjojen aiheuttamiin virheisiin verrattuna erittäin pieniä. Valonlähteiden käsitteleminen suuntaisvaloina on suhteellisen hyvä approksimaatio, koska valonlähteiden etäisyys tutkittavasta kohteesta oli monikymmenkertainen kohteiden kokoon verrattuna. Myös kameran etäisyys kromipallosta oli yli 18-kertainen suhteessa kromipallon halkaisijaan, joten valonlähteiden suuntia määritettäessä käytetty oletus ortogonaaliprojektiosta voidaan nähdä oikeutetuksi. Valkoreferenssikuvissa käytetyn tulostuspaperin albedon oletettiin olevan yksi. Tämän oletuksen oikeellisuus ei ole kovin merkittävä, koska se vaikuttaa ainoastaan lopputuloksen kuvapisteestä ja aallonpituudesta riippumattomaan kokonaiskirkkauteen. Tärkeämpää olisi arvioida, oliko tulostuspaperin albedo vakio koko kerätyn spektrin yli, mutta se jätettiin tämän tutkimuksen ulkopuolelle. Tutkittavien kohteiden albedojen, normaalivektoreiden ja syvyysarvojen oletettiin olevan vakioita yksittäisten aallonpituusalueiden yli. Jos näiden suureiden korkeataajuuksinen kohina jätetään huomiotta, niin havaitaan, että muutokset yhden aallonpituusalueen yli ovat todella pieniä, mikä tukee oletuksia.

5.4.3 Fotometrisen stereon suorituskyky

Vaihe, jossa hyperspektrikuvat jaetaan luvuilla $\hat{E}_m$, oli Matlab:lla toteutettuna jopa hieman nopeampi kuin Armadillo-kirjastolla. Tämä johtuu luultavasti siitä, että tutkimuksessa käytetty Matlabin versio osaa hyödyntää keskussuorittimen AVX-käskyjä, joiden avulla vektori- ja matriisioperaatiot käyttävät useita ytimiä yhtäaikaisesti SIMD-luokan tietokoneiden tavoin. Tästä syystä Armadillo-kirjaston kyky suorittaa rinnakkaislaskentaa ei tuo enää lisähyötyä. ViennaCL-toteutus osoittautui pienellä erolla kaikista hitaimmaksi. Tämä johtuu oletettavasti siitä, että suoritusajassa on mukana todella suuren datamäärän siirtäminen keskusmuistin ja grafiikkamuistin välillä itse grafiikkasuorittimella suoritetun aritmeettisen operaation kuitenkin ollessa nopeampi kuin keskussuorittimella suoritettuna. Tämän mittaaminen ei kuitenkaan ollut mahdollista ViennaCL-kirjaston rajoittuneen ohjelmointirajapinnan ansiosta.

Fotometrinen stereo oli lähes yhtä nopea Matlab:lla ja Armadillo-kirjastolla toteutettuina. Tämä johtuu luultavasti osittain samasta syystä kuin edellisen vaiheen tapauksessa. Lisäksi fotometrisen stereon toteutuksessa ratkaistaan silmukkarakenteen sisässä useita pieniä yhtälöryhmiä, jolloin Armadillo-kirjasto ei pysty hyödyntämään rinnakkaislaskentaa yhtä hyvin kuin ison yhtälöryhmän tapauksessa. Tästä samasta syystä myös ViennaCL-toteutus jäi todella hitaaksi. Silmukkarakenteen jokaisella kierroksella on kopioitava data keskusmuistista grafiikkamuistiin ja pysäytettävä keskussuoritin siksi aikaa kun grafiikkasuoritin ratkaisee pienikokoisen yhtälöryhmän sekä kopioi tuloksen takaisin keskusmuistiin. Grafiikkasuorittimet on suunniteltu juuri päinvastaiseen tarkoitukseen, jossa käsitellään suuria määriä alkioita kerralla ja keskussuorittimen kanssa asynkronisesti. Vaikka kaikkien hyperspektrikuvan siivujen käsitteleminen keralla pienentää tätä ongelmaa, se ei nähtävästi ole vielä tarpeeksi luomaan eroa sarjallisen ja rinnakkaisen laskennan välille. Sekä Armadillo- että ViennaCL-toteutuksen suorituskykyä voisi olla mahdollista parantaa asettamalla ongelma uuteen muotoon siten, että silmukkarakenteesta päästäisiin eroon ja ratkaistavan yhtälöryhmän koko kasvaisi. ViennaCL-toteutusta voisi parantaa myös toteuttamalla oman mukautetun ydinohjelman, joka kykenisi ratkaisemaan useita pienikokoisia yhtälöryhmiä rinnakkaisesti.

Syvyyskartan luontivaiheen suoritusajat tukevat arvioita edellisen vaiheen ViennaCL-toteutuksen hitaudesta. Hyperspektraalisen syvyyskartan siivu määritetään ratkaisemalla vain yksi erittäin suuri lineaarinen yhtälöryhmä, mikä voidaan suorittaa grafikkasuorittimella hyvin nopeasti. Lisäksi kun ViennaCL-kirjasto mahdollistaa pohjustinmatriisin käytön, on suoritusaika yli 13 kertaa niin nopea kun Matlab-toteutuksella. Tämän logiikan mukaisesti voidaan arvioida, että syvyyskartan luontivaiheessa myös Armadillo-toteutus olisi voinut olla Matlab-toteutusta nopeampi. Tämä rinnakkaislaskennan tuoma etu on merkittävä juuri hyperspektraalisten syvyyskarttojen tapauksessa, jossa ratkaistavia yhtälöryhmiä voi olla satoja.

Tulevaisuudessa olisi kannattavaa tutkia millä materiaaleilla ja virtuaalisilla valonlähteillä hyperspektrikuvantamisen hyöty tietokonegrafiikassa maksimoituu, jolloin käytännön sovellutuksissa voitaisiin valita missä tilanteissa sitä kannattaa käyttää. Lisäksi olisi hyödyllistä tutkia itse toteutettujen, hyperspektraaliin fotometriseen stereoon mukautettujen, grafiikkasuorittimella suoritettavien ydinohjelmien vaikutusta mallien luomisen suorituskyvyssä.

6 Yhteenveto

Tässä tutkielmassa tutkittiin, kuinka paljon hyperspektrikuvantaminen parantaa fotometrisellä stereolla luoduista malleista renderöityjen kuvien kuvanlaatua. Tutkimuksessa kuvattiin hyperspektrikameralla kolme eri kohdetta, ja niistä luotiin kolmiulotteiset mallit otettujen kuvien sekä fotometrisen stereon avulla. Malleista renderöitiin kahta eri virtuaalista valonlähdettä käyttäen kuvia, joita verrattiin kontrollikuviin, jotka oli luotu ilman hyperspektraalista dataa.

Subjektiivisen vertailun tuloksena hyperspektrikuvantamisen havaittiin tekevän värisävyistä luonnollisempia ja näin ollen parantavan kuvanlaatua. Lisäksi luotujen mallien muotojen havaittiin riippuvan valon aallonpituudesta, jota ei tavallisten valokuvien avulla luoduissa malleissa oteta yhtä tarkasti huomioon. Näin ollen myös hyperspektrikuvia käyttäen renderöidyt varjostukset vastaavat tarkemmin reaalimaailmaa. Vaikutukset jäivät kuitenkin hyvin pieniksi ja vaikeasti havaittaviksi. Hyperspektrikuvien avulla luotujen mallien renderöimisen havaittiin lisäksi olevan moninkerroin hitaampaa kuin kontrollikuvien avulla luotujen mallien. Tämä rajoittaa tutkimuksessa toteutetun menetelmän käyttämisen ei-reaaliaikaisiin renderöijiin.

Tutkielmassa tutkittiin lisäksi hyperspektrikuvantamisen avulla toteutetun fotometrisen stereon nopeuttamista moniydin- ja grafiikkasuorittimilla suoritettavalla rinnakkaislaskennalla. Tutkimuksen rinnakkaislaskentaa hyödyntävät toteutukset luotiin käyttäen valmiita numeerisen lineaarialgebran kirjastoja. Näiden toteutuksien suoritusaikoja verrattiin sarjallisien toteutuksien suoritusaikoihin.

Tutkimuksen rinnakkaislaskentatoteutukset rinnakkaistivat laskentaa vain osittain ja siten, että rinnakkaistetut tehtävät olivat pieniä verrattuna koko tehtävään. Näin ollen tutkimuksen toteutukset eivät hyödyntäneet rinnakkaislaskentaa optimaalisesti. Lisäksi kun sarjalliset toteutukset käyttivät suorittimen AVX-vektorikäskyjä, ei rinnakkaislaskennalla saatu pintojen orientaatioiden määrittämisen suoritusaikoja lainkaan pienemmiksi. Kuitenkin pintojen muotojen määrittäminen orientaatioiden avulla nopeutui grafiikkasuorittimella suoritettavalla rinnakkaislaskennalla yli 13-kertaisesti sarjalliseen toteutukseen verrattuna.

Lähteet

Akenine-Möller, Tomas, Eric Haines ja Naty Hoffman. 2008. *Real-Time Rendering 3rd Edition.* 3. painos. 1045. Natick, MA, USA: A. K. Peters, Ltd. ISBN: 987-1-56881-424-7.

Ashdown, Ian. 2002. "Photometry and Radiometry, A Tour Guide for Computer Graphics Enthusiasts" (lokakuu). Viitattu 23. toukokuuta 2017. http://www.helios32.com/Measuring%20Light.pdf.

Bailey, Mike, ja Steve Cunningham. 2009. *Graphics Shaders: Theory and Practice.* Natick, MA, USA: A. K. Peters, Ltd. ISBN: 1568813341, 9781568813349.

Barsky, Svetlana, ja Maria Petrou. 2003. "The 4-source photometric stereo technique for three-dimensional surfaces in the presence of highlights and shadows". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25 (10): 1239–1252. Viitattu 1. kesäkuuta 2017. http://www2.ee.ic.ac.uk/publications/p4689.pdf.

Chandraker, M., S. Agarwal ja D. Kriegman. 2007. "ShadowCuts: Photometric Stereo with Shadows". Teoksessa *2007 IEEE Conference on Computer Vision and Pattern Recognition,* 1–8. Kesäkuu. doi:10.1109/CVPR.2007.383288.

Chung, Hin-Shun, ja Jiaya Jia. 2008. "Efficient photometric stereo on glossy surfaces with wide specular lobes". Teoksessa *2008 IEEE Conference on Computer Vision and Pattern Recognition,* 1–8. Kesäkuu. doi:10.1109/CVPR.2008.4587771.

Coffey, Valerie C. 2012. "Multispectral Imaging Moves into the Mainstream". *Opt. Photon. News* 23, numero 4 (huhtikuu): 18–24. doi:10.1364/OPN.23.4.000018. http://www.osa-opn.org/abstract.cfm?URI=opn-23-4-18.

Du, Peng, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson ja Jack Dongarra. 2012. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". *Parallel Computing* 38 (8): 391–407. ISSN: 0167-8191. doi:https://doi.org/10.1016/j.parco.2011.10.002. http://www.sciencedirect.com/science/article/pii/S0167819111001335.

Eskelinen, Matti A., Ilkka Pölönen, Billy Braithwaite, Anna-Leena Erkkilä ja Tero Tuovinen. 2016. "Photometric stereo imaging of skin with a frame based spectral imager". Maaliskuu. doi:10.13140/RG.2.1.1076.9527.

Gao, Jiangning. 2016. "3D Face Recognition Using Multicomponent Feature Extraction from the Nasal Region and its Environs". Tohtorinväitöskirja, The Department of Electronic ja Electrical Engineering University of Bath. Viitattu 1. kesäkuuta 2017. http://opus.bath.ac.uk/55046/1/Thesis_Jiangning.pdf.

Garini, Yuval, Ian T. Young ja George McNamara. 2006. "Spectral imaging: Principles and applications". *Cytometry Part A* 69A (8): 735–747. ISSN: 1552-4930. doi:10.1002/cyto.a.20311. http://dx.doi.org/10.1002/cyto.a.20311.

Ghorpade, Jayshree, Jitendra Parande, Madhura Kulkarni ja Amit Bawaskar. 2012. "GPGPU Processing in CUDA Architecture". *CoRR* abs/1202.4347. http://arxiv.org/abs/1202.4347.

Govender, M, K Chetty, V Naiken ja H Bulcock. 2008. "A comparison of satellite hyperspectral and multispectral remote sensing imagery for improved classification and mapping of vegetation". *Water SA* 34 (helmikuu): 147–154. ISSN: 1816-7950. http://www.scielo.org.za/scielo.php?script=sci_arttext&pid=S1816-795020080002000 nrm=iso.

Gustavson, Stefan. 2017. *Lighting.* Viitattu 23. toukokuuta 2017. http://weber.itn.liu.se/~stegu/TNM061-2017/04_lights.pdf.

Hanrahan, Pat, ja Wolfgang Krueger. 1993. "Reflection from Layered Surfaces due to Subsurface Scattering". Viitattu 23. toukokuuta 2017. https://cseweb.ucsd.edu/~ravir/6998/papers/p165-hanrahan.pdf.

Hartikainen, Juha, ja Reijo Kouhia. 2010. "Elementtimenetelmän lineaarisen yhtälösysteemin iteratiivisesta ratkaisusta". *Rakenteiden Mekaniikka* 43 (2): 94–126. Viitattu 22. toukokuuta 2017. http://rmseura.tkk.fi/rmlehti/2010/nro2/RakMek_43_2_2010_2.pdf.

Harvey, David. 2012. *Chapter 10, Spectroscopic Methods.* Heinäkuu. Viitattu 23. toukokuuta 2017. https://www.saylor.org/site/wp-content/uploads/2012/07/Chapter1011.pdf.

Hearn, Donald D., ja M. Pauline Baker. 2003. *Computer Graphics with OpenGL.* 3. painos. Prentice Hall Professional Technical Reference. ISBN: 0130153907.

Horn, Berthold K. P. 1977. *Understanding Image Intensities.* Viitattu 28. helmikuuta 2017. https://www.cs.bgu.ac.il/~ben-shahar/Teaching/Computational-Vision/Readings/1977-Horn-Understanding_Image_Intensities.pdf.

Horn, Berthold KP. 1989. "Obtaining shape from shading information". Teoksessa *Shape from shading,* 123–171. MIT press. Viitattu 28. helmikuuta 2017. https://www.cs.bgu.ac.il/~ben-shahar/Teaching/Computational-Vision/Readings/1977-Horn-Understanding_Image_Intensities.pdf.

Horn, Berthold KP, ja Robert W Sjoberg. 1979. "Calculating the reflectance map". *Applied optics* 18 (11): 1770–1779. Viitattu 8. maaliskuuta 2017. http://people.csail.mit.edu/bkph/AIM/AIM-498-OPT.pdf.

Kampouris, Christos, Stefanos Zafeiriou, Abhijeet Ghosh ja Sotiris Malassiotis. 2016. "Fine-Grained Material Classification Using Micro-geometry and Reflectance". Teoksessa *Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part V,* toimittanut Bastian Leibe, Jiri Matas, Nicu Sebe ja Max Welling, 778–792. Cham: Springer International Publishing. ISBN: 978-3-319-46454-1. doi:10.1007/978-3-319-46454-1_47. http://dx.doi.org/10.1007/978-3-319-46454-1_47.

Kapasi, Ujval J., Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson ja John D. Owens. 2003. "Programmable Stream Processors". *Computer* (Los Alamitos, CA, USA) 36, numero 8 (elokuu): 54–62. ISSN: 0018-9162. doi:10.1109/MC.2003.1220582. http://dx.doi.org/10.1109/MC.2003.1220582.

Khronos Group. 2016a. "OpenGL Overview". Viitattu 24. toukokuuta 2017. https://www.opengl.org/about/.

Khronos Group. 2016b. “OpenGL Overview”. Viitattu 24. toukokuuta 2017. https://www.opengl.org/documentation/current_version/.

———. 2017. “Khronos OpenCL Registry”. Viitattu 24. toukokuuta 2017. https://www.khronos.org/registry/OpenCL/.

Using VIS/NIR and IR spectral cameras for detecting and separating crime scene details. 2012. Nide 8359. doi:10.1117/12.918555. http://dx.doi.org/10.1117/12.918555.

Layton, William, ja Myron Sussman. 2014. *Numerical Linear Algebra.* ISBN: 978-1-312-32985-0.

Lu, Guolan, ja Baowei Fei. 2014. “Medical hyperspectral imaging: a review”. *Journal of Biomedical Optics* 19, numero 1 (tammikuu): 10901–10901. doi:10.1117/1.JBO.19.1.010901. http://dx.doi.org/10.1117/1.JBO.19.1.010901.

Löyttyniemi, Tommi, Seyhan Nuyan, Marko Toskala ja Jari Alm. 2017. “On-line Topography Analysis By Photometric Stereo Method – New Tools For Tissue, Paper And Board Makers”. *O Papel* 78, numero 3 (maaliskuu): 76–81. Viitattu 1. kesäkuuta 2017. http://www.revistaopapel.org.br/noticia-anexos/1490151315_799a87f0e9688f51eb656e531146345.pdf.

Martonchik, John V, Carol J Bruegge ja Alan H Strahler. 2000. “A review of reflectance nomenclature used in remote sensing”. *Remote Sensing Reviews* 19 (1-4): 9–20. doi:10.1080/02757250009532407.

MathWorks. 2017. “Matlab Documentation, QR Solver”. Viitattu 24. toukokuuta 2017. https://se.mathworks.com/help/dsp/ref/qrsolver.html.

Nam, G., ja M. H. Kim. 2014. “Multispectral Photometric Stereo for Acquiring High-Fidelity Surface Normals”. *IEEE Computer Graphics and Applications* 34, numero 6 (marraskuu): 57–68. ISSN: 0272-1716. doi:10.1109/MCG.2014.108.

Nicodemus, F.E., NBS., Stati Uniti d'America. National bureau of standards, J.C. Richmond, J.J. Hsia, I.W. Ginsberg, T. Limperis ja United States. National Bureau of Standards. 1977. *Geometrical Considerations and Nomenclature for Reflectance.* NBS monograph. U.S. Government Printing Office. Viitattu 28. helmikuuta 2017. https://graphics.stanford.edu/courses/cs448-05-winter/papers/nicodemus-brdf-nist.pdf.

Ozawa, Keisuke, Imari Sato ja Masahiro Yamaguchi. 2017. "Hyperspectral photometric stereo for a single capture". *J. Opt. Soc. Am. A* 34, numero 3 (maaliskuu): 384–394. doi:10.1364/JOSAA.34.000384.

Palmer, James M, ja Lewis Carroll. 1999. "Radiometry and photometry FAQ". *URL: http://www. optics. arizona. edu/Palmer/rpfag/rpfag. htm# motivation.* Viitattu 28. helmikuuta 2017. https://employeepages.scad.edu/~kwitte/documents/Photometry_FAQ.PDF.

Pasquini, L., G. Avila, A. Blecha, C. Cacciari, V. Cayatte, M. Colless, F. Damiani ym. 2002. "Installation and commissioning of FLAMES, the VLT Multifibre Facility". *The Messenger* 110 (joulukuu): 1–9. Viitattu 23. toukokuuta 2017. http://esoads.eso.org/abs/2002Msngr.110....1P.

Pevar, Andrew, Lieven Verswyvel, Stamatios Georgoulis, Nico Cornelis, Marc Proesmans, Luc Van Gool ja Leuven. 2015. "Real-time Photometric Stereo". Viitattu 19. toukokuuta 2017. http://www.ifp.uni-stuttgart.de/publications/phowo15/190VanGool.pdf.

Polak, Adam, Timothy Kelman, Paul Murray, Stephen Marshall, David J.M. Stothard, Nicholas Eastaugh ja Francis Eastaugh. 2017. "Hyperspectral imaging combined with data classification techniques as an aid for artwork authentication". *Journal of Cultural Heritage.* ISSN: 1296-2074. doi:https://doi.org/10.1016/j.culher.2017.01.013. http://www.sciencedirect.com/science/article/pii/S1296207417301218.

Puhakka, Antti. 2008. *3D-grafiikka.* Talentum Media / Antti Puhakka. ISBN: 978-952-14-1192-2.

Rissanen, Anna, ja Heikki Saari. 2014. “Fabry-Perot Interferometer technologies”. Toukokuu. Viitattu 24. toukokuuta 2017. http://www.vttresearch.com/Impulse/Pages/Fabry-Perot-Interferometer-technologies.aspx.

Miniaturized hyperspectral imager calibration and UAV flight campaigns. 2013. Nide 8889. doi:10.1117/12.2028972. http://dx.doi.org/10.1117/12.2028972.

Schaepman-Strub, Gabriela, ME Schaepman, TH Painter, S Dangel ja JV Martonchik. 2006. “Reflectance quantities in optical remote sensing—Definitions and case studies”. *Remote sensing of environment* 103 (1): 27–42. doi:10.1016/j.rse.2006.03.002.

Smith, Randall B. 2012. *Introduction to Hyperspectral Imaging.* Verkkosivulla, tammikuu. Viitattu 23. toukokuuta 2017. http://www.microimages.com/documentation/Tutorials/hyprspec.pdf.

Tekatch, Anthony. 2009. “Electromagnetic Radiation Spectrum poster”. Helmikuu. Viitattu 23. toukokuuta 2017. http://unihedron.com/projects/spectrum/downloads/spectrum_20090210.pdf.

Teke, M., H. S. Deveci, O. Haliloğlu, S. Z. Gürbüz ja U. Sakarya. 2013. “A short survey of hyperspectral remote sensing applications in agriculture”. Teoksessa *2013 6th International Conference on Recent Advances in Space Technologies (RAST),* 171–176. Kesäkuu. doi:10.1109/RAST.2013.6581194.

TUWien. 2016a. “A manual and reference documentation of ViennaCL”. Viitattu 22. toukokuuta 2017. http://viennacl.sourceforge.net/doc/.

———. 2016b. “ViennaCL - The Vienna Computing Library”. Viitattu 24. toukokuuta 2017. http://viennacl.sourceforge.net/doc/manual-algorithms.html.

Unnc, M., Y. Inoue ja H. Asar. 2009. “GPGPU-FDTD method for 2-dimensional electromagnetic field simulation and its estimation”. Teoksessa *2009 IEEE 18th Conference on Electrical Performance of Electronic Packaging and Systems,* 239–242. Lokakuu. doi:10.1109/EPEPS.2009.5338432.

Varnavas, A., V. Argyriou, J. Ng ja A. A. Bharath. 2010. “Dense photometric stereo reconstruction on many core GPUs”. Teoksessa *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops,* 59–65. Kesäkuu. doi:10.1109/CVPRW.2010.5543152.

Woodham, Robert J. 1980. “Photometric method for determining surface orientation from multiple images”. *Optical engineering* 19 (1): 191139–191139. doi:10.1016/j.rse.2006.03.002.

Wu, Ying. 1978. “Radiometry, BRDF and Photometric Stereo”. Viitattu 8. maaliskuuta 2017. http://people.csail.mit.edu/bkph/AIM/AIM-498-OPT.pdf.

Yuen, P WT, ja M Richardson. 2010. “An introduction to hyperspectral imaging and its application for security, surveillance and target acquisition”. *The Imaging Science Journal* 58 (5): 241–253. doi:10.1179/174313110X12771950995716. http://dx.doi.org/10.1179/174313110X12771950995716.

Zheludev, Valery, Ilkka Pölönen, Noora Neittaanmäki-Perttu, Amir Averbuch, Pekka Neittaanmäki, Mari Grönroos ja Heikki Saari. 2015. “Delineation of malignant skin tumors by hyperspectral imaging using diffusion maps dimensionality reduction”. *Biomedical Signal Processing and Control* 16:48–60. ISSN: 1746-8094. doi:https://doi.org/10.1016/j.bspc.2014.10.010. http://www.sciencedirect.com/science/article/pii/S1746809414001608.

Liitteet

A Valojen suuntien laskennan Matlab-lähdekoodi

```
1  function [L, theta, phi] = CalculateLightDirections()
2      mask = imread('../../programming/data/calib/calib_mask.bmp');
3
4      % Find the center and radius of the mask ball
5      [maskRows, maskCols] = find(mask);
6      left = min(maskCols);
7      right = max(maskCols);
8      top = min(maskRows);
9      radius = (right - left) / 2;
10     center = [left + radius, top + radius];
11
12     disp('Started calculating light directions');
13     [L1, theta1, phi1] = LightDirection(1, center, radius);
14     [L2, theta2, phi2] = LightDirection(2, center, radius);
15     [L3, theta3, phi3] = LightDirection(3, center, radius);
16
17     L = [L1;L2;L3];
18
19     theta = [theta1; theta2; theta3];
20     phi = [phi1; phi2; phi3];
21
22     save L.mat L;
23 end
24
25 function [L, theta, phi] = LightDirection(index, center, radius)
26     % Find the brightest spot on the chrome ball
27     calibFiles = {'../../programming/data/calib/NE-kuula_RAD', ...
28                   '../../programming/data/calib/S-kuula_RAD', ...
29                   '../../programming/data/calib/NW-kuula_RAD'};
30     [ballBands,numBands] = ReadENVI(calibFiles{index});
31
32     theta = zeros(1, numBands);
33     phi = zeros(1, numBands);
34
35     % The direction to the camera
36     camera = [0, 0, 1];
37
38     L = [0,0,0];
39     for i = 1:numBands
40         ball = squeeze(ballBands(:,:,i));
41         [~,brightestX] = max(max(ball));
42         [~,brightestY] = max(ball(:,brightestX));
```

```
43
44          % Calculate the normal vector at the brightest spot
45          n = [brightestX - center(1),center(2) - brightestY ,0];
46          n(3) = sqrt(radius^2 - n(1)^2 - n(2)^2);
47          n = n / radius;
48
49          % Calculate the light direction when the reflection vector
50          % is the same as the camera vector
51          bandL = 2 * dot(camera, n) * n - camera;
52          bandL = bandL / norm(bandL);
53          L = L + bandL;
54
55          theta(i) = atan(sqrt((n(1)^2 + n(2)^2) / n(3)));
56          phi(i) = atan(n(2)/n(1));
57      end
58      L = L / norm(L);
59  end
```

B Fotometrisen stereon Matlab-toteutuksen lähdekoodi

```
1  function [NormalMap, AlbedoMap] = CalculateAlbedosAndNormals(L, imageFiles, wrFiles)
2      % Calculate reflectances I by dividing the radiances
3      % by the white reference radiances
4      [image, bands, width, height] = ReadENVI(imageFiles{1});
5      hatE = ReadENVI(wrFiles{1}) / dot(L(1,:), [0, 0, 1]);
6      I1 = image./hatE;
7      image = ReadENVI(imageFiles{2});
8      hatE = ReadENVI(wrFiles{2}) / dot(L(2,:), [0, 0, 1]);
9      I2 = image./hatE;
10     image = ReadENVI(imageFiles{3});
11     hatE = ReadENVI(wrFiles{3}) / dot(L(3,:), [0, 0, 1]);
12     I3 = image./hatE;
13
14     clearvars image hatE;
15
16     % Calculate normals and albedos using photometric stero
17     disp('Photometric stereo starts now!');
18     tic
19     NormalMap = single(zeros(height, width, 3, bands));
20     AlbedoMap = single(zeros(height, width, bands));
21     for x = 1:width
22         for y = 1:height
23             I = squeeze([I1(y,x,:);I2(y,x,:);I3(y,x,:)]);
24
25             G = L\I;
26
27             normals = single(zeros(3, bands));
```

```
28            albedos = single(zeros(1, bands));
29            for band = 1:bands
30                albedos(band) = norm(G(:,band));
31                normals(:,band) = G(:,band) / albedos(band);
32            end
33
34            NormalMap(y,x,:,:) = normals;
35            AlbedoMap(y,x,:) = albedos;
36        end
37    end
38    toc
39
40    save NormalMap.mat -v7.3 NormalMap;
41    save AlbedoMap.mat -v7.3 AlbedoMap;
42 end
```

C Syvyyskartan laskennan Matlab-toteutuksen lähdekoodi

```
1  function [] = CalculateAndSaveDepths(NormalMap, Folder)
2      dimensions = size(NormalMap);
3      width = dimensions(2);
4      height = dimensions(1);
5      bands = dimensions(4);
6      pixels = width * height;
7
8      signs = ones(2 * pixels, 1);
9      signs(2:2:length(signs)) = -1; % Positive y points up but we move down
10     % Left steps
11     for i = 1:height
12         signs(i * 2 * width - 1) = -1;
13     end
14     % Up steps
15     for i = 0:width - 1
16         signs((pixels - width + 1) * 2 + 2 * i) = 1;
17     end
18
19     % Two non-zero elements per row
20     Mrows = zeros(4 * pixels - 1, 1);
21     Mcolumns = zeros(4 * pixels - 1, 1);
22     Mvalues = ones(4 * pixels - 1, 1);
23     Mvalues(2 * pixels + 1:4 * pixels - 1) = -1;
24     % Positives
25     for i = 1:pixels
26         % x
27         Mrows(2 * i - 1) = 2 * i - 1;
28         Mcolumns(2 * i - 1) = i;
29         % y
```

```
30          Mrows(2 * i) = 2 * i;
31          Mcolumns(2 * i) = i;
32      end
33      % Negatives
34      for i = 1:pixels
35          % x
36          Mrows(2 * pixels + 2 * i - 1) = 2 * i - 1;
37          if signs(2 * i - 1) == 1
38              Mcolumns(2 * pixels + 2 * i - 1) = i + 1; % Right step
39          else
40              Mcolumns(2 * pixels + 2 * i - 1) = i - 1; % Left step
41          end
42          % y
43          if i ~= pixels % Down from the bottom right corner z = 0
44              Mrows(2 * pixels + 2 * i) = 2 * i;
45              if signs(2 * i) == -1
46                  Mcolumns(2 * pixels + 2 * i) = i + width; % Down step
47              else
48                  Mcolumns(2 * pixels + 2 * i) = i - width; % Up step
49              end
50          end
51      end
52      M = sparse(Mrows, Mcolumns, Mvalues);
53
54      for band = 1:bands
55          tic
56          n = zeros(2 * pixels, 1); %n_x and n_y divided by n_z
57          for i = 1:pixels
58              normal = squeeze(NormalMap(ceil(i / width), rem(i, width) + 1,:,band));
59              n(i * 2 - 1) = normal(1) / normal(3);
60              n(i * 2) = normal(2) / normal(3);
61          end
62          n = signs.*n;
63
64          z = M\n;
65
66          % Save depth maps to files
67          name = ['./results/',Folder,'/matlab/depth_map_',num2str(band),'.float'];
68          minDepth = min(z);
69          maxDepth = max(z);
70          z = (z - minDepth) / (maxDepth - minDepth); % Scale to [0,1]
71          fileID = fopen(name,'w');
72          fwrite(fileID,z,'float32');
73          fclose(fileID);
74          toc
75          disp(name);
76
```

```
77          Z = reshape(z, [width, height]);
78          Z = Z';
79          save Z63.mat Z;
80      end
81  end
```

D Fotometrisen stereon ViennalCL-toteutuksen lähdekoodi

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4
5  #include <boost/numeric/ublas/matrix.hpp>
6
7  #define VIENNACL_WITH_OPENCL
8  #include "viennacl/scalar.hpp"
9  #include "viennacl/vector.hpp"
10 #include "viennacl/matrix.hpp"
11 #include "viennacl/linalg/lu.hpp"
12 #include "viennacl/linalg/prod.hpp"
13 #include "viennacl/linalg/inner_prod.hpp"
14 #include "viennacl/linalg/matrix_operations.hpp"
15 #include "viennacl/tools/timer.hpp"
16
17 struct Vector4
18 {
19     float x = 0, y = 0, z = 0, w = 0;
20 };
21
22 static const int WIDTH = 1920;
23 static const int HEIGHT = 1200;
24 static const int BAND_COUNT = 133;
25 static const int BAND_SIZE = WIDTH * HEIGHT;
26 static const int CUBE_SIZE = BAND_SIZE * BAND_COUNT;
27 static const int LIGHT_COUNT = 3;
28
29 static const int MATRIX_COLUMNS = 13300;
30 static const int MATRIX_ROWS = 11520;
31
32 static const float gLightDirections[] =
33                         { 0.5737252f,  0.1018354f, 0.8126924f,   // Light 1
34                          -0.1972557f, -0.8252646f, 0.5291772f,   // Light 2
35                          -0.5274715f,  0.2341063f, 0.8166811f }; // Light 3
36
37 static viennacl::vector<float> L1(3);
38 static viennacl::vector<float> L2(3);
39 static viennacl::vector<float> L3(3);
```

```
40
41 static viennacl::matrix<float> L(LIGHT_COUNT, 3); // 3 dimensional space
42
43 viennacl::tools::timer gTimer;
44
45 typedef unsigned char byte;
46
47 using namespace boost::numeric;
48
49 float* readENVI(const char* filename)
50 {
51     std::ifstream file(filename, std::ios::binary);
52     if (!file.good())
53     {
54         std::cout << "Failed to open " << filename << std::endl;
55         std::terminate();
56     }
57
58     unsigned int size = WIDTH * HEIGHT * BAND_COUNT * sizeof(float);
59     char* data = new char[size];
60
61     file.seekg(0, std::ios::beg);
62     printf("start reading file\n");
63     gTimer.start();
64     file.read(data, size);
65     double exec_time = gTimer.get();
66     std::cout << "end reading file: " << exec_time << std::endl;
67     file.close();
68
69     return (float*)data;
70 }
71
72 void writeBandTo8BitFile(float* pSrc, const char* filename)
73 {
74     std::ofstream file(filename, std::ios::out | std::ios::app | std::ios::binary);
75     if (!file.good())
76     {
77         std::cout << "Failed to open " << filename << std::endl;
78         std::terminate();
79     }
80
81     byte* pDst = new byte[BAND_SIZE];
82
83     for (int i = 0; i < BAND_SIZE; ++i)
84     {
85         pDst[i] = fmin(pSrc[i], 1.0f) * 255;
86     }
```

```
87
88      file.write((char*)pDst, BAND_SIZE);
89      file.close();
90  }
91
92  void writeBandTo32BitFile(Vector4* pSrc, const char* filename)
93  {
94      std::ofstream file(filename, std::ios::out | std::ios::app | std::ios::binary);
95      if (!file.good())
96      {
97          std::cout << "Failed to open " << filename << std::endl;
98          std::terminate();
99      }
100
101     byte* pDst = new byte[BAND_SIZE * 4];
102
103     for (int i = 0; i < BAND_SIZE; ++i)
104     {
105         Vector4 albedoAndNormal = pSrc[i];
106         pDst[i * 4 + 0] = fmin(albedoAndNormal.x, 1.0f) * 255;
107         pDst[i * 4 + 1] = fmin(albedoAndNormal.y, 1.0f) * 255;
108         pDst[i * 4 + 2] = fmin(albedoAndNormal.z, 1.0f) * 255;
109         pDst[i * 4 + 3] = fmin(albedoAndNormal.w, 1.0f) * 255;
110     }
111
112     file.write((char*)pDst, BAND_SIZE);
113     file.close();
114 }
115
116 class GpuCube
117 {
118 public:
119     GpuCube(char* filename)
120     {
121         float* data = readENVI(filename);
122         printf("start uploading to gpu\n");
123         gTimer.start();
124         for (int i = 0; i < 2; ++i)
125         {
126             ublas::matrix<float> cpuData(MATRIX_ROWS, MATRIX_COLUMNS);
127             float* dst = &cpuData.data()[0];
128             memcpy(dst,
129                    data + i * MATRIX_ROWS * MATRIX_COLUMNS,
130                    MATRIX_ROWS * MATRIX_COLUMNS * sizeof(float));
131
132             gpuData[i] = viennacl::matrix<float>(MATRIX_ROWS, MATRIX_COLUMNS);
133             viennacl::copy(cpuData, gpuData[i]);
```

```
134         }
135         double exec_time = gTimer.get();
136         std::cout << "end␣uploading␣to␣gpu:␣" << exec_time << std::endl;
137         delete[] data;
138 
139         cpuData = ublas::matrix<float>(MATRIX_ROWS, MATRIX_COLUMNS);
140     }
141 
142     void update(char* filename)
143     {
144         float* data = readENVI(filename);
145         for (int i = 0; i < 2; ++i)
146         {
147             ublas::matrix<float> cpuData(MATRIX_ROWS, MATRIX_COLUMNS);
148             float* dst = &cpuData.data()[0];
149             memcpy(dst,
150                     data + i * MATRIX_ROWS * MATRIX_COLUMNS,
151                     MATRIX_ROWS * MATRIX_COLUMNS * sizeof(float));
152             viennacl::copy(cpuData, gpuData[i]);
153         }
154         delete[] data;
155         cpuDataDirty = true;
156     }
157 
158     void devideBy(const GpuCube& cube)
159     {
160         printf("division␣starts\n");
161         gTimer.start();
162         for (int i = 0; i < 2; ++i)
163         {
164             gpuData[i] = viennacl::linalg::element_div(gpuData[i], cube.gpuData[i]);
165         }
166         //viennacl::backend::finish();
167         double exec_time = gTimer.get();
168         std::cout << "division␣ends:␣" << exec_time << std::endl;
169         cpuDataDirty = true;
170     }
171 
172     void devideBy(float value)
173     {
174         for (int i = 0; i < 2; ++i)
175         {
176             gpuData[i] = gpuData[i] / value;
177         }
178     }
179 
180     void testWriteToFile(int band)
```

```
181     {
182         ublas::matrix<float> cpuData0(MATRIX_ROWS, MATRIX_COLUMNS);
183         viennacl::copy(gpuData[0], cpuData0);
184         ublas::matrix<float> cpuData1(MATRIX_ROWS, MATRIX_COLUMNS);
185         viennacl::copy(gpuData[1], cpuData1);
186         float* data0 = &cpuData0.data()[0];
187         float* data1 = &cpuData1.data()[0];
188         float* sliceData = new float[WIDTH * HEIGHT];
189         for (int i = 0; i < WIDTH * HEIGHT / 2; ++i)
190         {
191             float value = data0[band + BAND_COUNT * i];
192             sliceData[i] = value;
193         }
194         for (int i = 0; i < WIDTH * HEIGHT / 2; ++i)
195         {
196             float value = data1[band + BAND_COUNT * i];
197             sliceData[WIDTH * HEIGHT / 2 + i] = value;
198         }
199         writeBandTo8BitFile(sliceData, "/Users/markus/testiImage.jee");
200     }
201 
202     const float* getCpuData()
203     {
204         if (cpuDataDirty)
205         {
206             viennacl::copy(gpuData[0], cpuData);
207             cpuDataDirty = false;
208         }
209         return &cpuData.data()[0];
210     }
211 
212 private:
213     viennacl::matrix<float> gpuData[2];
214     ublas::matrix<float> cpuData; //(MATRIX_ROWS, MATRIX_COLUMNS);
215     bool cpuDataDirty = true;
216 };
217 
218 Vector4* calculateAlbedosAndNormals()
219 {
220     std::cout << " - Device Name: " << viennacl::ocl::current_device().name() << std::endl;
221 
222     std::vector<float> cpuZaxis;
223     cpuZaxis.push_back(0);
224     cpuZaxis.push_back(0);
225     cpuZaxis.push_back(1);
226     viennacl::vector<float> zAxis;
227     viennacl::copy(cpuZaxis, zAxis);
```

```
228
229     GpuCube gpuWr("/Users/markus/gradu/programming/data/wr/converted/NE-wr_RAD.dat");
230     gpuWr.devideBy(viennacl::linalg::inner_prod(L1, zAxis));
231     GpuCube gpuReflectance1("/Users/markus/gradu/programming/data/smurf/converted/NE-
        Smurf_RAD.dat");
232     gpuReflectance1.devideBy(gpuWr);
233     const float* reflectanceSlices1 = gpuReflectance1.getCpuData();
234
235     gpuWr.update("/Users/markus/gradu/programming/data/wr/converted/S-wr_RAD.dat");
236     gpuWr.devideBy(viennacl::linalg::inner_prod(L2, zAxis));
237     GpuCube gpuReflectance2("/Users/markus/gradu/programming/data/smurf/converted/S-
        Smurf_RAD.dat");
238     gpuReflectance2.devideBy(gpuWr);
239     const float* reflectanceSlices2 = gpuReflectance2.getCpuData();
240
241     gpuWr.update("/Users/markus/gradu/programming/data/wr/converted/NW-wr_RAD.dat");
242     gpuWr.devideBy(viennacl::linalg::inner_prod(L3, zAxis));
243     GpuCube gpuReflectance3("/Users/markus/gradu/programming/data/smurf/converted/NW-
        Smurf_RAD.dat");
244     gpuReflectance3.devideBy(gpuWr);
245     const float* reflectanceSlices3 = gpuReflectance3.getCpuData();
246
247     Vector4* albedosAndNormals = new Vector4[CUBE_SIZE];
248
249     ublas::matrix<float> cpuI(LIGHT_COUNT, BAND_COUNT);
250     viennacl::matrix<float> I(LIGHT_COUNT, BAND_COUNT);
251     for (int y = 0; y < HEIGHT / 2; ++y)
252     {
253         printf("Start calculating albedos and normals for row %d", y);
254         gTimer.start();
255         for (int x = 0; x < WIDTH; ++x)
256         {
257             float* dst = &cpuI.data()[0];
258             memcpy(dst + BAND_COUNT * 0,
259                    reflectanceSlices1 + (y * WIDTH + x) * BAND_COUNT,
260                   BAND_COUNT * sizeof(float));
261             memcpy(dst + BAND_COUNT * 1,
262                    reflectanceSlices2 + (y * WIDTH + x) * BAND_COUNT,
263                   BAND_COUNT * sizeof(float));
264             memcpy(dst + BAND_COUNT * 2,
265                    reflectanceSlices3 + (y * WIDTH + x) * BAND_COUNT,
266                   BAND_COUNT * sizeof(float));
267             viennacl::copy(cpuI, I);
268
269             viennacl::linalg::lu_substitute(L, I);
270
271             viennacl::copy(I, cpuI);
```

```
272                 float* Gdata = &cpuI.data()[0];
273                 for (int band = 0; band < BAND_COUNT; ++band)
274                 {
275                     float normalX = Gdata[band + BAND_COUNT * 0];
276                     float normalY = Gdata[band + BAND_COUNT * 1];
277                     float normalZ = Gdata[band + BAND_COUNT * 2];
278                     float albedo = sqrt(normalX*normalX + normalY*normalY + normalZ*normalX);
279                     normalX /= albedo;
280                     normalY /= albedo;
281                     normalZ /= albedo;
282                     albedosAndNormals[WIDTH * y + x].x = normalX;
283                     albedosAndNormals[WIDTH * y + x].y = normalY;
284                     albedosAndNormals[WIDTH * y + x].z = normalZ;
285                     albedosAndNormals[WIDTH * y + x].w = albedo;
286                 }
287             }
288             double exec_time = gTimer.get();
289             std::cout << "One␣row␣of␣albedos␣and␣normals␣calculated:␣" << exec_time << std::
        endl;
290         }
291 
292         return albedosAndNormals;
293 }
294 
295 void initL()
296 {
297     ublas::matrix<float> cpuL(LIGHT_COUNT, 3);
298     float* dst = &cpuL.data()[0];
299     memcpy(dst, gLightDirections, LIGHT_COUNT * 3 * sizeof(float));
300     viennacl::copy(cpuL, L);
301     viennacl::linalg::lu_factorize(L);
302 
303     ublas::vector<float> cpuLightDir(3);
304     float* dst2 = &cpuLightDir.data()[0];
305     memcpy(dst2, &gLightDirections[0], 3 * sizeof(float));
306     viennacl::copy(cpuLightDir, L1);
307     memcpy(dst2, &gLightDirections[3], 3 * sizeof(float));
308     viennacl::copy(cpuLightDir, L2);
309     memcpy(dst2, &gLightDirections[6], 3 * sizeof(float));
310     viennacl::copy(cpuLightDir, L3);
311 }
312 
313 int main()
314 {
315     initL();
316     Vector4* albedosAndNormals = calculateAlbedosAndNormals();
317     writeBandTo32BitFile(albedosAndNormals + BAND_SIZE * 63, "/Users/markus/
```

```
        testiAlbedoNormal.jee");
318     delete[] albedosAndNormals;
319
320     return EXIT_SUCCESS;
321 }
```

E Syvyyskartan laskennan ViennaCL-toteutuksen lähdekoodi

```
1  void calculateAndWriteDepthMaps()
2  {
3      // Calculate depths
4      fcolvec signs = ones<fcolvec>(2 * BAND_SIZE);
5      for (int i = 1; i < 2 * BAND_SIZE; i += 2)
6      {
7          signs(i) = -1; // Positive y points up but we move down
8      }
9      // Left steps
10     for (int i = 1; i <= HEIGHT; ++i)
11     {
12         signs(i * 2 * WIDTH - 2) = -1;
13     }
14     // Up steps
15     for (int i = 0; i < WIDTH - 1; ++i)
16     {
17         signs((BAND_SIZE - WIDTH + 1) * 2 + 2 * i - 1) = 1;
18     }
19
20     sp_fmat Mt; // Transposed M
21     viennacl::compressed_matrix<float> vclMtM; // Mt * M;
22     {
23         int rows = 2 * BAND_SIZE;
24         int cols = BAND_SIZE;
25         int value_count = 2 * rows - 1;
26         fcolvec values = ones<fcolvec>(value_count);
27         values.rows(2 * BAND_SIZE, value_count - 1) *= -1;
28         umat locations(2, value_count);
29
30         // Positives
31         for (int i = 0; i < BAND_SIZE; ++i)
32         {
33             // x
34             locations(0, 2 * i) = 2 * i;
35             locations(1, 2 * i) = i;
36             // y
37             locations(0, 2 * i + 1) = 2 * i + 1;
38             locations(1, 2 * i + 1) = i;
39         }
```

```
40          // Negatives
41          for (int i = 0; i < BAND_SIZE; ++i)
42          {
43              // x
44              locations(0, 2 * BAND_SIZE + 2 * i) = 2 * i;
45              if (signs(2 * i) == 1)
46              {
47                  locations(1, 2 * BAND_SIZE + 2 * i) = i + 1; // Right step
48              }
49              else
50              {
51                  locations(1, 2 * BAND_SIZE + 2 * i) = i - 1; // Left step
52              }
53              // y
54              if (i != BAND_SIZE - 1) // Down from the bottom right corner z = 0
55              {
56                  locations(0, 2 * BAND_SIZE + 2 * i + 1) = 2 * i + 1;
57                  if (signs(2 * i + 1) == -1)
58                  {
59                      locations(1, 2 * BAND_SIZE + 2 * i + 1) = i + WIDTH; // Down step
60                  }
61                  else
62                  {
63                      locations(1, 2 * BAND_SIZE + 2 * i + 1) = i - WIDTH; // Up step
64                  }
65              }
66          }
67
68          sp_fmat M(locations, values, rows, cols, true, false);
69          Mt = trans(M);
70          sp_fmat MtM = Mt * M;
71
72          viennacl::copy(MtM, vclMtM);
73      }
74
75      // configuration of preconditioner:
76      viennacl::linalg::chow_patel_tag chow_patel_ichol_config;
77      chow_patel_ichol_config.sweeps(3);        // three nonlinear sweeps
78      chow_patel_ichol_config.jacobi_iters(2); // two Jacobi iterations per triangular 'solve
        ' Rx=r
79      // create and compute preconditioner:
80      viennacl::linalg::chow_patel_icc_precond< viennacl::compressed_matrix<float> >
        chow_patel_ichol(vclMtM, chow_patel_ichol_config);
81
82      for (int band = 0; band < BAND_COUNT; ++band)
83      {
84          printf("Started calculating depth map %d\n", band + 1);
```

```
85          gTimer.start();
86          fcolvec n(2 * BAND_SIZE); // n_x and n_y divided by n_z
87          for (int i = 0; i < BAND_SIZE; ++i)
88          {
89              Vec3 normal = normalMaps[band][i];
90              n(i * 2) = normal.x / normal.z;
91              n(i * 2 + 1) = normal.y / normal.z;
92          }
93          n = signs % n;
94          n = Mt*n;
95          printf("It took %f seconds to generate the n and Mt*n for band %d\n", gTimer.
    elapsed(), band + 1);
96
97          viennacl::vector<float> vclMtn; // Mt*n on GPU
98          viennacl::copy(n, vclMtn);
99
100         gTimer.start();
101         viennacl::vector<float> vclZ = viennacl::linalg::solve(vclMtM, vclMtn, viennacl::
    linalg::cg_tag(1e-10, 100), chow_patel_ichol); // GPU
102         printf("It took %f seconds to solve the depth system for band %d\n", gTimer.elapsed
    (), band + 1);
103
104         fcolvec z(BAND_SIZE);
105         viennacl::copy(vclZ, z);
106
107         float minDepth = z.min();
108         float maxDepth = z.max();
109         printf("minz = %f, maxZ = %f", minDepth, maxDepth);
110         z = (z - minDepth) / (maxDepth - minDepth); // Scale to [0,1]
111
112         char depthFile[200];
113 #ifdef WRITE_FLOAT_TEXTURES
114         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
    depth_map_%d.float", band + 1);
115         sprintf(depthFile, "/Users/markus/gradu/programming/results/cyprus/cpp_float/
    depth_map_%d.float", band + 1);
116         //sprintf(depthFile, "/Users/markus/gradu/programming/results/lego/cpp_float/
    depth_map_%d.float", band + 1);
117         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/control/
    depth_map_%d.float", band + 1);
118         writeBandToFloatTexture(z.memptr(), depthFile, 1);
119 #else
120         sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
    depth_map_%d.bmp", band + 1);
121         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/control/
    depth_map_%d.bmp", band + 1);
122         writeBandTo8BitBMP(z.memptr(), depthFile);
```

```
123 #endif
124     }
125 }
```

F Fotometrisen stereon Armadillo-toteutuksen lähdekoodi

```
1  #include <iostream>
2  #include <armadillo>
3  #include <chrono>
4
5  #define VIENNACL_WITH_OPENCL
6  #define VIENNACL_WITH_ARMADILLO 1
7  #include "viennacl/scalar.hpp"
8  #include "viennacl/vector.hpp"
9  #include "viennacl/matrix.hpp"
10 #include "viennacl/linalg/lu.hpp"
11 #include "viennacl/linalg/prod.hpp"
12 #include "viennacl/linalg/matrix_operations.hpp"
13 #include "viennacl/tools/timer.hpp"
14 #include "viennacl/linalg/cg.hpp"
15 #include "viennacl/linalg/gmres.hpp"
16
17 using namespace std;
18 using namespace arma;
19
20 typedef unsigned char byte;
21
22 const int WIDTH = 1920;
23 const int HEIGHT = 1200;
24 const int BAND_COUNT = 133; // Use 3 for control
25 const int BAND_SIZE = WIDTH * HEIGHT;
26 const int CUBE_SIZE = BAND_SIZE * BAND_COUNT;
27
28 const int LIGHT_COUNT = 3;
29 const int DIMENSION_COUNT = 3;
30
31 const std::string normalMapsFile =
32         //"/Users/markus/gradu/programming/results/smurf/raw/normal_maps.raw";
33         "/Users/markus/gradu/programming/results/cyprus/raw/normal_maps.raw";
34         //"/Users/markus/gradu/programming/results/lego/raw/normal_maps.raw";
35 const std::string albedoMapsFile =
36         //"/Users/markus/gradu/programming/results/smurf/raw/albedo_maps.raw";
37         "/Users/markus/gradu/programming/results/cyprus/raw/albedo_maps.raw";
38         //"/Users/markus/gradu/programming/results/lego/raw/albedo_maps.raw";
39
40 struct Vec3 {
41     float x = 0;
```

```
42      float y = 0;
43      float z = 0;
44  };
45
46  fcube albedoMaps(HEIGHT, WIDTH, BAND_COUNT); // Column-major
47  Vec3* normalMaps[BAND_COUNT]; // Row-major
48
49                                  // Light1          Light2          Light3
50  float gLightDirections[] = { 0.5737252f,   -0.1972557f,   -0.5274715f,    // x
51                               0.1018354f,   -0.8252646f,    0.2341063f,    // y
52                               0.8126924f,    0.5291772f,    0.8166811f };  // z
53
54  float gL1[] = {  0.5737252f,  0.1018354f, 0.8126924f };
55  float gL2[] = { -0.1972557f, -0.8252646f, 0.5291772f };
56  float gL3[] = { -0.5274715f,  0.2341063f, 0.8166811f };
57  float gZaxis[] = { 0, 0, 1};
58
59  //#define LOAD_NORMALS_FROM_DISK
60  #define WRITE_FLOAT_TEXTURES
61
62  class Timer
63  {
64  public:
65      Timer() : beg_(clock_::now()) {}
66      void start() { beg_ = clock_::now(); }
67      double elapsed() const {
68          return std::chrono::duration_cast<second_>
69          (clock_::now() - beg_).count(); }
70
71  private:
72      typedef std::chrono::high_resolution_clock clock_;
73      typedef std::chrono::duration<double, std::ratio<1> > second_;
74      std::chrono::time_point<clock_> beg_;
75  };
76
77  Timer gTimer;
78
79  // Armadillo documentation is available at:
80  // http://arma.sourceforge.net/docs.html
81
82  float* readENVI(const char* filename, float* pData = nullptr)
83  {
84      std::ifstream file(filename, std::ios::binary);
85      if (!file.good())
86      {
87          std::cout << "Failed to open " << filename << std::endl;
88          std::terminate();
```

```
    }

    float* data;
    if (pData)
    {
        data = pData;
    }
    else
    {
        data = new float[CUBE_SIZE];
    }
    file.seekg(0, std::ios::beg);
    file.read((char*)data, CUBE_SIZE * sizeof(float));
    file.close();

    return data;
}

char* gBMP24Header;
int gBMP24HeaderSize;
char* gBMP8Header;
int gBMP8HeaderSize;

void initBMPHeaders()
{
    const char* filename24 =
        "/Users/markus/gradu/programming/results/smurf/normal_map_64.bmp";
    std::ifstream file24(filename24, std::ios::binary);

    if (!file24.good())
    {
        std::cout << "Failed to open " << filename24 << std::endl;
        std::terminate();
    }

    file24.seekg(10);
    file24.read((char*)&gBMP24HeaderSize, 4);
    gBMP24Header = new char[gBMP24HeaderSize];
    file24.seekg(0, std::ios::beg);
    file24.read(gBMP24Header, gBMP24HeaderSize);
    file24.close();

    const char* filename8 =
        "/Users/markus/gradu/programming/results/smurf/albedo_map_64.bmp";
    std::ifstream file8(filename8, std::ios::binary);

    if (!file8.good())
```

```
136     {
137         std::cout << "Failed to open " << filename8 << std::endl;
138         std::terminate();
139     }
140
141     file8.seekg(10);
142     file8.read((char*)&gBMP8HeaderSize, 4);
143     gBMP8Header = new char[gBMP8HeaderSize];
144     file8.seekg(0, std::ios::beg);
145     file8.read(gBMP8Header, gBMP8HeaderSize);
146     file8.close();
147
148 }
149
150 void writeBandTo8BitBMP(float* pSrc, const char* filename)
151 {
152     std::ofstream file(filename, std::ios::out | std::ios::app | std::ios::binary);
153     if (!file.good())
154     {
155         std::cout << "Failed to open " << filename << std::endl;
156         std::terminate();
157     }
158
159     byte* pDst = new byte[BAND_SIZE];
160
161     for (int x = 0; x < WIDTH; ++x)
162     {
163         for (int y = 0; y < HEIGHT; ++y)
164         {
165             int srcIdx = (HEIGHT - 1 - y) * WIDTH + x;
166             int dstIdx = y * WIDTH + x;
167             pDst[dstIdx] = fmin(pSrc[srcIdx], 1.0f) * 255;
168         }
169     }
170     file.write(gBMP8Header, gBMP8HeaderSize);
171     file.write((char*)pDst, BAND_SIZE);
172     file.close();
173 }
174
175 void writeBandTo24BitBMP(Vec3* pSrc, const char* filename)
176 {
177     std::ofstream file(filename, std::ios::out | std::ios::app | std::ios::binary);
178     if (!file.good())
179     {
180         std::cout << "Failed to open " << filename << std::endl;
181         std::terminate();
182     }
```

```
183
184     byte* pDst = new byte[BAND_SIZE * 3];
185
186     for (int x = 0; x < WIDTH; ++x)
187     {
188         for (int y = 0; y < HEIGHT; ++y)
189         {
190             int srcIdx = (HEIGHT - 1 - y) * WIDTH + x;
191             int dstIdx = y * WIDTH + x;
192             pDst[dstIdx * 3 + 0] = fmin(pSrc[srcIdx].z, 1.0f) * 255;
193             pDst[dstIdx * 3 + 1] = fmin(pSrc[srcIdx].y, 1.0f) * 255;
194             pDst[dstIdx * 3 + 2] = fmin(pSrc[srcIdx].x, 1.0f) * 255;
195         }
196     }
197     file.write(gBMP24Header, gBMP24HeaderSize);
198     file.write((char*)pDst, BAND_SIZE * 3);
199     file.close();
200 }
201
202 void writeBandToFloatTexture(void* pSrc, const char* filename, int componentCount)
203 {
204     std::ofstream file(filename, std::ios::out | std::ios::app | std::ios::binary);
205     if (!file.good())
206     {
207         std::cout << "Failed to open " << filename << std::endl;
208         std::terminate();
209     }
210
211     file.write((char*)pSrc, BAND_SIZE * componentCount * sizeof(float));
212     file.close();
213 }
214
215 void saveRawNormalMaps()
216 {
217     std::ofstream file(normalMapsFile, std::ios::out | std::ios::app | std::ios::binary);
218     if (!file.good())
219     {
220         std::cout << "Failed to open " << normalMapsFile << std::endl;
221         std::terminate();
222     }
223
224     for (int band = 0; band < BAND_COUNT; ++band)
225     {
226         file.write((char*)normalMaps[band], BAND_SIZE * sizeof(Vec3));
227     }
228     file.close();
229 }
```

```
230
231 void saveRawAlbedoMaps()
232 {
233     std::ofstream file(albedoMapsFile, std::ios::out | std::ios::app | std::ios::binary);
234     if (!file.good())
235     {
236         std::cout << "Failed to open " << albedoMapsFile << std::endl;
237         std::terminate();
238     }
239
240     for (int band = 0; band < BAND_COUNT; ++band)
241     {
242         file.write((char*)albedoMaps.slice(band).memptr(), BAND_SIZE * sizeof(float));
243     }
244     file.close();
245 }
246
247 void loadRawNormalMaps()
248 {
249     std::ifstream file(normalMapsFile, std::ios::binary);
250
251     if (!file.good())
252     {
253         std::cout << "Failed to open " << normalMapsFile << std::endl;
254         std::terminate();
255     }
256
257     file.seekg(0, std::ios::beg);
258
259     for (int band = 0; band < BAND_COUNT; ++band)
260     {
261         normalMaps[band] = new Vec3[BAND_SIZE];
262         file.read((char*)normalMaps[band], BAND_SIZE * sizeof(Vec3));
263     }
264     file.close();
265 }
266
267 void loadRawAlbedoMaps()
268 {
269     std::ifstream file(albedoMapsFile, std::ios::binary);
270
271     if (!file.good())
272     {
273         std::cout << "Failed to open " << albedoMapsFile << std::endl;
274         std::terminate();
275     }
276
```

```
277     file.seekg(0, std::ios::beg);
278
279     for (int band = 0; band < BAND_COUNT; ++band)
280     {
281         file.read((char*)albedoMaps.slice(band).memptr(), BAND_SIZE * sizeof(float));
282     }
283     file.close();
284 }
285
286 void calculateAlbedosAndNormals()
287 {
288     fvec L1(gL1, 3, false, true);
289     fvec L2(gL2, 3, false, true);
290     fvec L3(gL3, 3, false, true);
291     fvec zAxis(gZaxis, 3, false, true);
292
293     // North east
294     float* radData1 =
295         //readENVI("/Users/markus/gradu/programming/data/smurf/float/column_major/NE-
        Smurf_RAD.dat");
296         readENVI("/Users/markus/gradu/programming/data/cyprus/float/column_major/NE-
        Cyprus_RAD.dat");
297         //readENVI("/Users/markus/gradu/programming/data/lego/float/column_major/NE-
        Lego_RAD.dat");
298     //float* radData1 =
299         //readENVI("/Users/markus/gradu/programming/data/smurf/control/NE-Smurf_RAD.dat");
300     fcube reflectance1(radData1, HEIGHT, WIDTH, BAND_COUNT, false, true);
301     float* wrData =
302         readENVI("/Users/markus/gradu/programming/data/wr/float/column_major/NE-wr_RAD.dat"
        );
303     //float* wrData =
304         //readENVI("/Users/markus/gradu/programming/data/wr/control/NE-wr_RAD.dat");
305     fcube wr(wrData, HEIGHT, WIDTH, BAND_COUNT, false, true);
306     wr /= dot(L1, zAxis);
307     gTimer.start();
308     reflectance1 /= wr;
309     printf("It took %f seconds to divide by E\n", gTimer.elapsed());
310
311     // South
312     float* radData2 =
313         //readENVI("/Users/markus/gradu/programming/data/smurf/float/column_major/S-
        Smurf_RAD.dat");
314         readENVI("/Users/markus/gradu/programming/data/cyprus/float/column_major/S-
        Cyprus_RAD.dat");
315         //readENVI("/Users/markus/gradu/programming/data/lego/float/column_major/S-Lego_RAD
        .dat");
316     //float* radData2 =
```

```
317         //readENVI("/Users/markus/gradu/programming/data/smurf/control/S-Smurf_RAD.dat");
318     fcube reflectance2(radData2, HEIGHT, WIDTH, BAND_COUNT, false, true);
319     readENVI("/Users/markus/gradu/programming/data/wr/float/column_major/S-wr_RAD.dat",
        wrData);
320     //readENVI("/Users/markus/gradu/programming/data/wr/control/S-wr_RAD.dat", wrData);
321     wr /= dot(L2, zAxis);
322     reflectance2 /= wr;
323 
324     // North west
325     float* radData3 =
326         //readENVI("/Users/markus/gradu/programming/data/smurf/float/column_major/NW-
        Smurf_RAD.dat");
327         readENVI("/Users/markus/gradu/programming/data/cyprus/float/column_major/NW-
        Cyprus_RAD.dat");
328         //readENVI("/Users/markus/gradu/programming/data/lego/float/column_major/NW-
        Lego_RAD.dat");
329     //float* radData3 =
330         //readENVI("/Users/markus/gradu/programming/data/smurf/control/NW-Smurf_RAD.dat");
331     fcube reflectance3(radData3, HEIGHT, WIDTH, BAND_COUNT, false, true);
332     readENVI("/Users/markus/gradu/programming/data/wr/float/column_major/NW-wr_RAD.dat",
        wrData);
333     //readENVI("/Users/markus/gradu/programming/data/wr/control/NW-wr_RAD.dat", wrData);
334     wr /= dot(L3, zAxis);
335     reflectance3 /= wr;
336     delete[] wrData;
337 
338     for (int band = 0; band < BAND_COUNT; ++band)
339     {
340         normalMaps[band] = new Vec3[BAND_SIZE];
341     }
342 
343     fmat L(gLightDirections, LIGHT_COUNT, DIMENSION_COUNT, false, true);
344     fmat I(LIGHT_COUNT, BAND_COUNT);
345     printf("Start␣photometric␣stereo!\n");
346     gTimer.start();
347     for (int x = 0; x < WIDTH; ++x)
348     {
349         for (int y = 0; y < HEIGHT; ++y)
350         {
351             I.row(0) = (frowvec)reflectance1.tube(y, x);
352             I.row(1) = (frowvec)reflectance2.tube(y, x);
353             I.row(2) = (frowvec)reflectance3.tube(y, x);
354 
355             fmat G = solve(L, I);
356 
357             int band = 0;
358             G.each_col([&](fcolvec& columnVec) {
```

```
359                     float albedo = norm(columnVec);
360                     columnVec /= albedo;
361                     columnVec += 1.0;
362                     columnVec *= 0.5;
363                     normalMaps[band][WIDTH * y + x] = *(Vec3*)columnVec.memptr();
364                     albedoMaps(y, x, band) = albedo;
365                     band++;
366                 });
367             }
368         }
369         printf("It took %f seconds to complete the photometric stereo\n", gTimer.elapsed());
370 
371         delete[] radData1;
372         delete[] radData2;
373         delete[] radData3;
374 
375         saveRawNormalMaps();
376         saveRawAlbedoMaps();
377 }
378 
379 void writeAlbedosToFiles()
380 {
381         for (int band = 0; band < BAND_COUNT; ++band)
382         {
383             char albedoFile[200];
384             fmat albedoMap = trans(albedoMaps.slice(band));
385 #ifdef WRITE_FLOAT_TEXTURES
386             //sprintf(albedoFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
        albedo_map_%d.float", band + 1);
387             sprintf(albedoFile, "/Users/markus/gradu/programming/results/cyprus/cpp_float/
        albedo_map_%d.float", band + 1);
388             //sprintf(albedoFile, "/Users/markus/gradu/programming/results/lego/cpp_float/
        albedo_map_%d.float", band + 1);
389             //sprintf(albedoFile, "/Users/markus/gradu/programming/results/smurf/control/
        albedo_map_%d.float", band + 1);
390             writeBandToFloatTexture(albedoMap.memptr(), albedoFile, 1);
391 #else
392             sprintf(albedoFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
        albedo_map_%d.bmp", band + 1);
393             //sprintf(albedoFile, "/Users/markus/gradu/programming/results/smurf/control/
        albedo_map_%d.bmp", band + 1);
394             writeBandTo8BitBMP(albedoMap.memptr(), albedoFile);
395 #endif
396         }
397 }
398 
399 void writeNormalsToFiles()
```

```
400 {
401     for (int band = 0; band < BAND_COUNT; ++band)
402     {
403         char normalFile[200];
404 #ifdef WRITE_FLOAT_TEXTURES
405         //sprintf(normalFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
    normal_map_%d.float", band + 1);
406         sprintf(normalFile, "/Users/markus/gradu/programming/results/cyprus/cpp_float/
    normal_map_%d.float", band + 1);
407         //sprintf(normalFile, "/Users/markus/gradu/programming/results/lego/cpp_float/
    normal_map_%d.float", band + 1);
408         //sprintf(normalFile, "/Users/markus/gradu/programming/results/smurf/control/
    normal_map_%d.float", band + 1);
409         writeBandToFloatTexture(normalMaps[band], normalFile, 3);
410 #else
411         sprintf(normalFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
    normal_map_%d.bmp", band + 1);
412         //sprintf(normalFile, "/Users/markus/gradu/programming/results/smurf/control/
    normal_map_%d.bmp", band + 1);
413         writeBandTo24BitBMP(normalMaps[band], normalFile);
414 #endif
415     }
416 }
417
418 void calculateAndWriteDepthMaps()
419 {
420     // Calculate depths
421     fcolvec signs = ones<fcolvec>(2 * BAND_SIZE);
422     for (int i = 1; i < 2 * BAND_SIZE; i += 2)
423     {
424         signs(i) = -1; // Positive y points up but we move down
425     }
426     // Left steps
427     for (int i = 1; i <= HEIGHT; ++i)
428     {
429         signs(i * 2 * WIDTH - 2) = -1;
430     }
431     // Up steps
432     for (int i = 0; i < WIDTH - 1; ++i)
433     {
434         signs((BAND_SIZE - WIDTH + 1) * 2 + 2 * i - 1) = 1;
435     }
436
437     sp_fmat Mt; // Transposed M
438     viennacl::compressed_matrix<float> vclMtM; // Mt * M;
439     {
440         int rows = 2 * BAND_SIZE;
```

```
441         int cols = BAND_SIZE;
442         int value_count = 2 * rows - 1;
443         fcolvec values = ones<fcolvec>(value_count);
444         values.rows(2 * BAND_SIZE, value_count - 1) *= -1;
445         umat locations(2, value_count);
446 
447         // Positives
448         for (int i = 0; i < BAND_SIZE; ++i)
449         {
450             // x
451             locations(0, 2 * i) = 2 * i;
452             locations(1, 2 * i) = i;
453             // y
454             locations(0, 2 * i + 1) = 2 * i + 1;
455             locations(1, 2 * i + 1) = i;
456         }
457         // Negatives
458         for (int i = 0; i < BAND_SIZE; ++i)
459         {
460             // x
461             locations(0, 2 * BAND_SIZE + 2 * i) = 2 * i;
462             if (signs(2 * i) == 1)
463             {
464                 locations(1, 2 * BAND_SIZE + 2 * i) = i + 1; // Right step
465             }
466             else
467             {
468                 locations(1, 2 * BAND_SIZE + 2 * i) = i - 1; // Left step
469             }
470             // y
471             if (i != BAND_SIZE - 1) // Down from the bottom right corner z = 0
472             {
473                 locations(0, 2 * BAND_SIZE + 2 * i + 1) = 2 * i + 1;
474                 if (signs(2 * i + 1) == -1)
475                 {
476                     locations(1, 2 * BAND_SIZE + 2 * i + 1) = i + WIDTH; // Down step
477                 }
478                 else
479                 {
480                     locations(1, 2 * BAND_SIZE + 2 * i + 1) = i - WIDTH; // Up step
481                 }
482             }
483         }
484 
485         sp_fmat M(locations, values, rows, cols, true, false);
486         Mt = trans(M);
487         sp_fmat MtM = Mt * M;
```

```
488
489         viennacl::copy(MtM, vclMtM);
490     }
491
492     // configuration of preconditioner:
493     viennacl::linalg::chow_patel_tag chow_patel_ichol_config;
494     chow_patel_ichol_config.sweeps(3);        // three nonlinear sweeps
495     chow_patel_ichol_config.jacobi_iters(2); // two Jacobi iterations per triangular 'solve
    ' Rx=r
496     // create and compute preconditioner:
497     viennacl::linalg::chow_patel_icc_precond< viennacl::compressed_matrix<float> >
    chow_patel_ichol(vclMtM, chow_patel_ichol_config);
498
499     for (int band = 0; band < BAND_COUNT; ++band)
500     {
501         printf("Started␣calculating␣depth␣map␣%d\n", band + 1);
502         gTimer.start();
503         fcolvec n(2 * BAND_SIZE); // n_x and n_y divided by n_z
504         for (int i = 0; i < BAND_SIZE; ++i)
505         {
506             Vec3 normal = normalMaps[band][i];
507             n(i * 2) = normal.x / normal.z;
508             n(i * 2 + 1) = normal.y / normal.z;
509         }
510         n = signs % n;
511         n = Mt*n;
512         printf("It␣took␣%f␣seconds␣to␣generate␣the␣n␣and␣Mt*n␣for␣band␣%d\n", gTimer.
    elapsed(), band + 1);
513
514         viennacl::vector<float> vclMtn; // Mt*n on GPU
515         viennacl::copy(n, vclMtn);
516
517         gTimer.start();
518         viennacl::vector<float> vclZ = viennacl::linalg::solve(vclMtM, vclMtn, viennacl::
    linalg::cg_tag(1e-10, 100), chow_patel_ichol); // GPU
519         printf("It␣took␣%f␣seconds␣to␣solve␣the␣depth␣system␣for␣band␣%d\n", gTimer.elapsed
    (), band + 1);
520
521         fcolvec z(BAND_SIZE);
522         viennacl::copy(vclZ, z);
523
524         float minDepth = z.min();
525         float maxDepth = z.max();
526         printf("minz␣=␣%f,␣maxZ␣=␣%f", minDepth, maxDepth);
527         z = (z - minDepth) / (maxDepth - minDepth); // Scale to [0,1]
528
529         char depthFile[200];
```

```
530 #ifdef WRITE_FLOAT_TEXTURES
531         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
        depth_map_%d.float", band + 1);
532         sprintf(depthFile, "/Users/markus/gradu/programming/results/cyprus/cpp_float/
        depth_map_%d.float", band + 1);
533         //sprintf(depthFile, "/Users/markus/gradu/programming/results/lego/cpp_float/
        depth_map_%d.float", band + 1);
534         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/control/
        depth_map_%d.float", band + 1);
535         writeBandToFloatTexture(z.memptr(), depthFile, 1);
536 #else
537         sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/cpp_float/
        depth_map_%d.bmp", band + 1);
538         //sprintf(depthFile, "/Users/markus/gradu/programming/results/smurf/control/
        depth_map_%d.bmp", band + 1);
539         writeBandTo8BitBMP(z.memptr(), depthFile);
540 #endif
541     }
542 }
543 
544 int main(int argc, char** argv)
545 {
546     cout << "Armadillo version: " << arma_version::as_string() << endl;
547 
548     initBMPHeaders();
549 
550 #ifdef LOAD_NORMALS_FROM_DISK
551     printf("Start loading normals\n");
552     loadRawNormalMaps();
553 #else
554     calculateAlbedosAndNormals();
555     printf("Start writing normal and albedo maps\n");
556     writeNormalsToFiles();
557     writeAlbedosToFiles();
558 #endif
559 
560     delete[] albedoMaps.memptr();
561 
562     printf("Start calculating depth maps\n");
563     calculateAndWriteDepthMaps();
564 
565     return 0;
566 }
```

G Renderöijän CPU-osuuden lähdekoodi

```
1 #include <stdlib.h>
```

```
#include <iostream>
#include <fstream>
#include <cmath>

#include <GLUT/glut.h>
#include <OpenGL/gl3.h>

#include "Shaders.h"
#include "BMPReader.h"
#include "SpectrumData.h"

#define USE_FLOAT_TEXTURES
#define USE_SUN_SPECTRUM
//#define USE_CONTROL

#ifdef USE_CONTROL
static const int RED_START_BAND = 3;
static const int GREEN_START_BAND = 2;
static const int BLUE_START_BAND = 1;

static const int RED_BAND_COUNT = 1;
static const int GREEN_BAND_COUNT = 1;
static const int BLUE_BAND_COUNT = 1;
#else
static const int RED_START_BAND = 48;
static const int GREEN_START_BAND = 18;
static const int BLUE_START_BAND = 1;

static const int RED_BAND_COUNT = 45;
static const int GREEN_BAND_COUNT = 45;
static const int BLUE_BAND_COUNT = 38;
#endif

static const char* object = "smurf";
//static const char* object = "cyprus";
//static const char* object = "lego";

int gProgramHandle;
unsigned int gVaoHandle;
unsigned int gVboHandle;

GLuint gRedTextures;
GLuint gGreenTextures;
GLuint gBlueTextures;

float gSpectralSensitivityRed[RED_BAND_COUNT];
float gSpectralSensitivityGreen[GREEN_BAND_COUNT];
```

```
49 float gSpectralSensitivityBlue[BLUE_BAND_COUNT];
50
51 float gLightSpectrumRed[RED_BAND_COUNT];
52 float gLightSpectrumGreen[GREEN_BAND_COUNT];
53 float gLightSpectrumBlue[BLUE_BAND_COUNT];
54
55 float calcGaussianWeight(float dist, float sigma)
56 {
57     float weight = 1.0f;
58     if (sigma > 0)
59     {
60         float g = 1.0f / sqrt(2.0f * 3.14159 * sigma * sigma);
61         weight = (g * exp(-(dist * dist) / (2 * sigma * sigma)));
62     }
63     return weight;
64 }
65
66 class Timer
67 {
68 public:
69     Timer() : beg_(clock_::now()) {}
70     void start() { beg_ = clock_::now(); }
71     double elapsed() const {
72         return std::chrono::duration_cast<second_>
73         (clock_::now() - beg_).count(); }
74
75 private:
76     typedef std::chrono::high_resolution_clock clock_;
77     typedef std::chrono::duration<double, std::ratio<1> > second_;
78     std::chrono::time_point<clock_> beg_;
79 };
80
81 Timer gTimer;
82
83 void render()
84 {
85     glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
86     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
87
88     GLuint lightPosHandle = glGetUniformLocation(gProgramHandle, "lightPos");
89     static float angle = 0.0f;
90     angle += 3.1415927f / 60.0f;
91     if (angle >= 2 * 3.1415927f) angle = 0.0f;
92     static const float r = 20.0f;
93     float x = cos(angle) * r;
94     float y = sin(angle) * r;
95     //glUniform3f(lightPosHandle, 20, 20, -20.0f*sin(angle));
```

```
96      glUniform3f(lightPosHandle, x, y, -10.0f);
97
98      GLuint spectralSensitivityHandle = glGetUniformLocation(gProgramHandle, "
        spectralSensitivity");
99      GLuint lightIrradianceHandle = glGetUniformLocation(gProgramHandle, "lightIrradiance");
100
101     glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE);
102     glBindTexture(GL_TEXTURE_2D_ARRAY, gRedTextures);
103     glUniform1fv(spectralSensitivityHandle, RED_BAND_COUNT, gSpectralSensitivityRed);
104     glUniform1fv(lightIrradianceHandle, RED_BAND_COUNT, gLightSpectrumRed);
105     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
106
107     glColorMask(GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE);
108     glBindTexture(GL_TEXTURE_2D_ARRAY, gGreenTextures);
109     glUniform1fv(spectralSensitivityHandle, GREEN_BAND_COUNT, gSpectralSensitivityGreen);
110     glUniform1fv(lightIrradianceHandle, GREEN_BAND_COUNT, gLightSpectrumGreen);
111     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
112
113     glColorMask(GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE);
114     glBindTexture(GL_TEXTURE_2D_ARRAY, gBlueTextures);
115     glUniform1fv(spectralSensitivityHandle, BLUE_BAND_COUNT, gSpectralSensitivityBlue);
116     glUniform1fv(lightIrradianceHandle, BLUE_BAND_COUNT, gLightSpectrumBlue);
117     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
118
119     glutSwapBuffers();
120
121     printf("It took %f seconds to render a frame\n", gTimer.elapsed());
122     gTimer.start();
123 }
124
125 void initOpenGL(int argc, char **argv)
126 {
127     glutInit(&argc, argv);
128     glutInitDisplayMode(GLUT_3_2_CORE_PROFILE | GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
129     glutInitWindowPosition(0, 0);
130     glutInitWindowSize(WIDTH, HEIGHT);
131     glutCreateWindow("Hyperspectral Renderer");
132     glDisable(GL_DEPTH_TEST);
133     glEnable(GL_CULL_FACE);
134     glFrontFace(GL_CW);
135     glClearColor(0, 0, 0, 0);
136
137     // Always use the same shaders
138     gProgramHandle = CreateProgram("VertexShader.glsl", "PixelShader.glsl");
139     glUseProgram(gProgramHandle);
140
141     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

```
142     glutIdleFunc((void (__cdecl*)(void))glutPostRedisplay);
143     glutDisplayFunc(render);
144     glReadBuffer(GL_LEFT);
145 }
146 
147 void createVertexBuffer()
148 {
149     glGenVertexArrays(1, &gVaoHandle);
150     glBindVertexArray(gVaoHandle);
151 
152     glGenBuffers(1, &gVboHandle);
153     glBindBuffer(GL_ARRAY_BUFFER, gVboHandle);
154     GLfloat vertexData[] = {
155         //  X      Y
156         -1.0f, -1.0f,
157         -1.0f,  1.0f,
158          1.0f, -1.0f,
159          1.0f,  1.0f
160     };
161     glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
162     glEnableVertexAttribArray(0);
163     glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);
164 }
165 
166 void createTexturesForChannel(GLuint* textureHandle, int startBand, int bandCount)
167 {
168     glGenTextures(1, textureHandle);
169     glBindTexture(GL_TEXTURE_2D_ARRAY, *textureHandle);
170 
171 #ifdef USE_FLOAT_TEXTURES
172     glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA16F,
173                  WIDTH, HEIGHT, bandCount, 0, GL_RGBA, GL_FLOAT, nullptr);
174 #else
175     glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA,
176                  WIDTH, HEIGHT, bandCount, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);
177 #endif
178 
179     char normalFile[200];
180     char albedoFile[200];
181     char depthFile[200];
182 
183     for (int band = startBand; band < startBand + bandCount; ++band)
184     {
185         memset(normalFile, 0, sizeof(normalFile));
186         memset(albedoFile, 0, sizeof(albedoFile));
187         memset(depthFile, 0, sizeof(depthFile));
188 #ifdef USE_FLOAT_TEXTURES
```

```
189 #ifdef USE_CONTROL
190         sprintf(normalFile,
191                 "/Users/markus/gradu/programming/results/%s/control/normal_map_%d.float",
192                 object, band);
193         sprintf(albedoFile,
194                 "/Users/markus/gradu/programming/results/%s/control/albedo_map_%d.float",
195                 object, band);
196         sprintf(depthFile,
197                 "/Users/markus/gradu/programming/results/%s/control/depth_map_%d.float",
198                 object, band);
199 #else
200         sprintf(normalFile,
201                 "/Users/markus/gradu/programming/results/%s/cpp_float/normal_map_%d.float",
202                 object, band);
203         sprintf(albedoFile,
204                 "/Users/markus/gradu/programming/results/%s/cpp_float/albedo_map_%d.float",
205                 object, band);
206         sprintf(depthFile,
207                 "/Users/markus/gradu/programming/results/%s/cpp_float/depth_map_%d.float",
208                 object, band);
209 #endif
210 #else
211         sprintf(normalFile,
212                 "/Users/markus/gradu/programming/results/%s/cpp_float/normal_map_%d.bmp",
213                 object, band);
214         sprintf(albedoFile,
215                 "/Users/markus/gradu/programming/results/%s/cpp_float/albedo_map_%d.bmp",
216                 object, band);
217         sprintf(depthFile,
218                 "/Users/markus/gradu/programming/results/%s/depth_map_%d.bmp",
219                 object, band);
220 #endif
221
222 #ifdef USE_FLOAT_TEXTURES
223         float* normalData = readFloat(normalFile);
224         float* albedoData = readFloat(albedoFile);
225         float* depthData  = readFloat(depthFile);
226 #else
227         byte* normalData = readBMP(normalFile);
228         byte* albedoData = readBMP(albedoFile);
229         byte* depthData  = readBMP(depthFile);
230 #endif
231
232 #ifdef USE_FLOAT_TEXTURES
233         float* textureData = new float[WIDTH*HEIGHT*4];
234 #else
235         byte* textureData = new byte[WIDTH*HEIGHT*4];
```

```
236 #endif
237
238         for (int normalSrcIdx = 0, albedoSrcIdx = 0, dstIdx = 0;
239              albedoSrcIdx < WIDTH*HEIGHT;
240              normalSrcIdx += 3, albedoSrcIdx++, dstIdx += 4)
241         {
242 #ifdef USE_FLOAT_TEXTURES
243             textureData[dstIdx + 0] = normalData[normalSrcIdx + 0];
244 #else
245             textureData[dstIdx + 0] = normalData[normalSrcIdx + 2];
246 #endif
247             textureData[dstIdx + 1] = normalData[normalSrcIdx + 1];
248             textureData[dstIdx + 2] = albedoData[albedoSrcIdx + 0];
249             textureData[dstIdx + 3] =  depthData[albedoSrcIdx + 0];
250         }
251
252 #ifdef USE_FLOAT_TEXTURES
253         glTexSubImage3D(GL_TEXTURE_2D_ARRAY, 0, 0, 0, band - startBand,
254                         WIDTH, HEIGHT, 1, GL_RGBA, GL_FLOAT, textureData);
255 #else
256         glTexSubImage3D(GL_TEXTURE_2D_ARRAY, 0, 0, 0, band - startBand,
257                         WIDTH, HEIGHT, 1, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
258 #endif
259
260         delete[] normalData;
261         delete[] albedoData;
262         delete[] depthData;
263     }
264
265     glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
266     glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
267     glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
268     glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
269 }
270
271 void createTextures()
272 {
273     createTexturesForChannel(&gRedTextures, RED_START_BAND, RED_BAND_COUNT);
274     createTexturesForChannel(&gGreenTextures, GREEN_START_BAND, GREEN_BAND_COUNT);
275     createTexturesForChannel(&gBlueTextures, BLUE_START_BAND, BLUE_BAND_COUNT);
276 }
277
278 void initializeSpectralSensitivity()
279 {
280 #ifdef USE_CONTROL
281     gSpectralSensitivityRed[0] = 1.0f;
282     gSpectralSensitivityGreen[0] = 1.0f;
```

```
283         gSpectralSensitivityBlue[0] = 1.0f;
284 #else
285         float sum = 0;
286         for (int i = 0; i < RED_BAND_COUNT; ++i)
287         {
288             float peakWavelength = wavelengths[RED_START_BAND + RED_BAND_COUNT / 2 - 1];
289             float wavelength = wavelengths[i + RED_START_BAND - 1];
290             float nextWavelength = wavelengths[i + RED_START_BAND];
291             float distToNext = nextWavelength - wavelength;
292             float distToPeak = fabsf(wavelength - peakWavelength);
293             gSpectralSensitivityRed[i] = calcGaussianWeight(distToPeak, 20.0f) * distToNext;
294             sum += gSpectralSensitivityRed[i];
295             printf("%d. peak = %f, wavelength = %f, dist = %f, sensitivity = %f\n",
296                    i, peakWavelength, wavelength, distToPeak, gSpectralSensitivityRed[i]);
297         }
298         printf("sum = %f\n", sum);
299         sum = 0;
300         for (int i = 0; i < GREEN_BAND_COUNT; ++i)
301         {
302             float peakWavelength = wavelengths[GREEN_START_BAND + GREEN_BAND_COUNT / 2 - 1];
303             float wavelength = wavelengths[i + GREEN_START_BAND - 1];
304             float nextWavelength = wavelengths[i + GREEN_START_BAND];
305             float distToNext = nextWavelength - wavelength;
306             float distToPeak = fabsf(wavelength - peakWavelength);
307             gSpectralSensitivityGreen[i] = calcGaussianWeight(distToPeak, 20.0f) * distToNext;
308             sum += gSpectralSensitivityGreen[i];
309             printf("%d. peak = %f, wavelength = %f, dist = %f, sensitivity = %f\n",
310                    i, peakWavelength, wavelength, distToPeak, gSpectralSensitivityGreen[i]);
311         }
312         printf("sum = %f\n", sum);
313         sum = 0;
314         for (int i = 0; i < BLUE_BAND_COUNT; ++i)
315         {
316             float peakWavelength = wavelengths[BLUE_START_BAND + BLUE_BAND_COUNT / 2 - 1];
317             float wavelength = wavelengths[i + BLUE_START_BAND - 1];
318             float nextWavelength = wavelengths[i + BLUE_START_BAND];
319             float distToNext = nextWavelength - wavelength;
320             float distToPeak = fabsf(wavelength - peakWavelength);
321             gSpectralSensitivityBlue[i] = calcGaussianWeight(distToPeak, 18.0f) * distToNext;
322             sum += gSpectralSensitivityBlue[i];
323             printf("%d. peak = %f, wavelength = %f, dist = %f, sensitivity = %f\n",
324                    i, peakWavelength, wavelength, distToPeak, gSpectralSensitivityBlue[i]);
325         }
326         printf("sum = %f\n", sum);
327 #endif
328 }
329
```

```
330 void initializeLightSpectrum()
331 {
332 #ifdef USE_SUN_SPECTRUM
333     const float* spectrum = solarSpectrum;
334     float scale = 1.2e-7;
335 #else
336     const float* spectrum = lightBulbSpectrum;
337     float scale = 3.0f / 1.0870e+5;
338 #endif
339 
340 #ifdef USE_CONTROL
341     static const int HYPERSPECTRAL_RED_START_BAND = 48;
342     static const int HYPERSPECTRAL_GREEN_START_BAND = 18;
343     static const int HYPERSPECTRAL_BLUE_START_BAND = 1;
344     static const int HYPERSPECTRAL_RED_BAND_COUNT = 45;
345     static const int HYPERSPECTRAL_GREEN_BAND_COUNT = 45;
346     static const int HYPERSPECTRAL_BLUE_BAND_COUNT = 38;
347 
348     float spectralSensitivityRed[HYPERSPECTRAL_RED_BAND_COUNT];
349     float spectralSensitivityGreen[HYPERSPECTRAL_GREEN_BAND_COUNT];
350     float spectralSensitivityBlue[HYPERSPECTRAL_BLUE_BAND_COUNT];
351 
352     float sum = 0;
353     for (int i = 0; i < HYPERSPECTRAL_RED_BAND_COUNT; ++i)
354     {
355         float peakWavelength = wavelengths[HYPERSPECTRAL_RED_START_BAND +
    HYPERSPECTRAL_RED_BAND_COUNT / 2 - 1];
356         float wavelength = wavelengths[i + HYPERSPECTRAL_RED_START_BAND - 1];
357         float nextWavelength = wavelengths[i + HYPERSPECTRAL_RED_START_BAND];
358         float distToNext = nextWavelength - wavelength;
359         float distToPeak = fabsf(wavelength - peakWavelength);
360         spectralSensitivityRed[i] = calcGaussianWeight(distToPeak, 20.0f) * distToNext;
361         sum += spectralSensitivityRed[i];
362         printf("control: %d. peak = %f, wavelength = %f, dist = %f, sensitivity = %f\n",
363                i, peakWavelength, wavelength, distToPeak, spectralSensitivityRed[i]);
364     }
365     printf("sum = %f\n", sum);
366     sum = 0;
367     for (int i = 0; i < HYPERSPECTRAL_GREEN_BAND_COUNT; ++i)
368     {
369         float peakWavelength = wavelengths[HYPERSPECTRAL_GREEN_START_BAND +
    HYPERSPECTRAL_GREEN_BAND_COUNT / 2 - 1];
370         float wavelength = wavelengths[i + HYPERSPECTRAL_GREEN_START_BAND - 1];
371         float nextWavelength = wavelengths[i + HYPERSPECTRAL_GREEN_START_BAND];
372         float distToNext = nextWavelength - wavelength;
373         float distToPeak = fabsf(wavelength - peakWavelength);
374         spectralSensitivityGreen[i] = calcGaussianWeight(distToPeak, 20.0f) * distToNext;
```

```
375         sum += spectralSensitivityGreen[i];
376         printf("control:␣%d.␣peak␣=␣%f,␣wavelength␣=␣%f,␣dist␣=␣%f,␣sensitivity␣=␣%f\n",
377                 i, peakWavelength, wavelength, distToPeak, spectralSensitivityGreen[i]);
378     }
379     printf("sum␣=␣%f\n", sum);
380     sum = 0;
381     for (int i = 0; i < HYPERSPECTRAL_BLUE_BAND_COUNT; ++i)
382     {
383         float peakWavelength = wavelengths[HYPERSPECTRAL_BLUE_START_BAND +
    HYPERSPECTRAL_BLUE_BAND_COUNT / 2 - 1];
384         float wavelength = wavelengths[i + HYPERSPECTRAL_BLUE_START_BAND - 1];
385         float nextWavelength = wavelengths[i + HYPERSPECTRAL_BLUE_START_BAND];
386         float distToNext = nextWavelength - wavelength;
387         float distToPeak = fabsf(wavelength - peakWavelength);
388         spectralSensitivityBlue[i] = calcGaussianWeight(distToPeak, 18.0f) * distToNext;
389         sum += spectralSensitivityBlue[i];
390         printf("control:␣%d.␣peak␣=␣%f,␣wavelength␣=␣%f,␣dist␣=␣%f,␣sensitivity␣=␣%f\n",
391                 i, peakWavelength, wavelength, distToPeak, spectralSensitivityBlue[i]);
392     }
393     printf("sum␣=␣%f\n", sum);
394 
395     gLightSpectrumRed[0] = 0;
396     for (int i = 0; i < HYPERSPECTRAL_RED_BAND_COUNT; ++i)
397         gLightSpectrumRed[0] += spectrum[i] * scale * spectralSensitivityRed[i];
398     gLightSpectrumGreen[0] = 0;
399     for (int i = 0; i < HYPERSPECTRAL_GREEN_BAND_COUNT; ++i)
400         gLightSpectrumGreen[0] += spectrum[i] * scale * spectralSensitivityGreen[i];
401     gLightSpectrumBlue[0] = 0;
402     for (int i = 0; i < HYPERSPECTRAL_BLUE_BAND_COUNT; ++i)
403         gLightSpectrumBlue[0] += spectrum[i] * scale * spectralSensitivityBlue[i];
404 #else
405     for (int i = 0; i < RED_BAND_COUNT; ++i)
406     {
407         gLightSpectrumRed[i] = spectrum[i + RED_START_BAND - 1] * scale;
408     }
409     for (int i = 0; i < GREEN_BAND_COUNT; ++i)
410     {
411         gLightSpectrumGreen[i] = spectrum[i + GREEN_START_BAND - 1] * scale;
412     }
413     for (int i = 0; i < BLUE_BAND_COUNT; ++i)
414     {
415         gLightSpectrumBlue[i] = spectrum[i + BLUE_START_BAND - 1] * scale;
416     }
417 #endif
418 }
419 
420 int main (int argc, char **argv)
```

```
421 {
422     initOpenGL(argc, argv);
423 
424     createVertexBuffer();
425 
426     createTextures();
427 
428     initializeSpectralSensitivity();
429 
430     initializeLightSpectrum();
431 
432     glutMainLoop();
433 
434     return 0;
435 }
```

H Renderöijän verteksivarjostinohjelma

```
1 #version 410 core
2 
3 #define USE_FLOAT_TEXTURES
4 
5 in vec2 pos;
6 
7 out vec2 pixelPos;
8 out vec2 texCoord;
9 
10 void main()
11 {
12     texCoord = (pos + 1.0f) * 0.5f;
13 #ifdef USE_FLOAT_TEXTURES
14     texCoord.y = -texCoord.y + 1.0f;
15 #endif
16     pixelPos = pos;
17     gl_Position = vec4(pos, 0.0, 1.0f);
18 }
```

I Renderöijän kuvapistevarjostinohjelma

```
1 #version 410 core
2 
3 uniform sampler2DArray sampler;
4 uniform vec3 lightPos;
5 uniform float lightIrradiance[45];
6 uniform float spectralSensitivity[45];
7 
8 in vec2 texCoord;
```

```
in vec2 pixelPos;

out vec4 color;

const float WIDTH = 1920.0f;
const float HEIGHT = 1200.0f;
const float DIST_TO_CAM = 29.0f; // in centimeters
const float PI = 3.1415927f;
const float DEGS_TO_RADS = 0.0174533f;
const float VERTICAL_FOV = 15.8f * DEGS_TO_RADS;
const float HORIZONTAL_FOV = 20.9f * DEGS_TO_RADS;
const vec2 FOV = vec2(HORIZONTAL_FOV, VERTICAL_FOV);
const vec2 TAN_FOV_PER_2 = tan(FOV / 2.0f);
const float PIXEL_IN_CENTIMETERS = DIST_TO_CAM * TAN_FOV_PER_2.x * 2.0f / WIDTH;
const float DEPTH_SCALE = 200.0f;

void main(void)
{
    color = vec4(0, 0, 0, 0);
    int bandCount = textureSize(sampler, 0).z;

    for (int band = 0; band < bandCount; ++band)
    {
        vec4 data = texture(sampler, vec3(texCoord, band));

        vec3 N = vec3(0, 0 ,0);
        N.xy = data.rg * 2.0f - 1.0f;
        N.z = sqrt(1 - dot(N.xy, N.xy));

        float albedo = data.b;
        float depth = data.a * DEPTH_SCALE;

        float wsZ = -DIST_TO_CAM + depth * PIXEL_IN_CENTIMETERS;
        vec3 wsPos = vec3(pixelPos * TAN_FOV_PER_2 * abs(wsZ), wsZ);

        vec3 L = normalize(lightPos - wsPos);

        float E_0 = lightIrradiance[band];

        float radiance = albedo / PI * E_0 * dot(L, N) * spectralSensitivity[band];

        color += vec4(radiance, radiance, radiance, 1.0f);
    }
}
```