TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ANTTI HEINONEN

KAHDEN REAALIAIKAISESSA RENDEROINNISSA KÄYTETYN VARJOKARTTATEKNIIKAN TOTEUTUS JA VERTAILU

Kandidaatintyö

Tarkastaja:
projektitutkija Mikko Nurminen
Jätetty tarkastettavaksi 10.12.2017

TIIVISTELMÄ

ANTTI HEINONEN: Kahden reaaliaikaisessa renderoinnissa käytetyn varjokarttatekniikan toteutus ja vertailu
Tampereen teknillinen yliopisto
Kandidaatintyö, 27 sivua, 20 liitesivua
Joulukuu 2017
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: projektitutkija Mikko Nurminen

Avainsanat: PSSM, RTW, varjokartta, renderointi, OpenGL, virtuaaliympäristö

Tässä työssä käydään läpi rinnakkaisjaetun-varjokarttatekniikan ja suoralinjaisesti väännätetyn varjokarttatekniikan toteutus. Varjokartan avulla voidaan luoda virtuaaliympäristöön varjot kolmiulotteisille geometrisille objekteille. Työn tarkoitus on vertailla tekniikoiden tehokkuutta, muistinkulutusta, sekä toteutuksien eroja ja kompleksisuutta. Aluksi käsitellään reaaliaikaisen renderoinnin määritelmä, sekä esitellään grafiikan renderoinnissa käytetty OpenGL-rajapinta. Tämän jälkeen esitellään varjokartan toiminnallisuutta ja esitellään työssä käytetyt varjokarttatekniikat. Varjokarttatekniikoiden tehokkuuserojen ja toteutuksien kompleksisuuden mittaamista varten käytetään testiohjelmaa, jonka toteutusta käydään läpi. Lopuksi esitellään tulokset ja yhteenveto.

Tehokkuuden mittarina käytetään ruudun piirtoaikaa, eli kuinka kauan lopputuloksen renderointi näytölle kestää kokonaisuudessaan. Muistinkulutus mitataan staattisesti, koska varjokarttatekniikoiden vaatimat resurssit ovat etukäteen tiedossa. Kompleksisuus mitataan tiedostomäärinä, sekä ohjelmakoodirivien lukumääränä.

Tuloksista huomataan, että rinnakkaisjaetun-varjokarttatekniikan toteutus on yksinkertaisempi, mutta sen tehokkuus on huonompi ja muistinkulutus on suurempi verrattuna suoralinjaisesti väännätettyyn varjokarttatekniikkaan.

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. OPENGL 3
2.1 Toiminta 3
2.2 Liukuhihnamalli 4
3. VARJOKARTTA 7
3.1 Toiminta 7
3.2 Rinnakkaisjaettu-varjokartta 8
3.3 Suoralinjaisesti väännätetty varjokartta 11
4. TESTIOHJELMA JA MENETELMÄT 15
4.1 Testiohjelman toteutus 15
4.1.1 Virtuaaliympäristön luonti 15
4.1.2 Rinnakkaisjaetun-varjokartan toteutus 16
4.1.3 Suoralinjaisesti väännätetyn varjokartan toteutus 16
4.2 Tutkimusmenetelmät 18
4.3 Laitteisto 18
5. TULOKSET 19
5.1 Toteutuksien kompleksisuus 19
5.1.1 Muistinkulutus 20
5.2 Ruudun piirtoaika 21
5.3 Visuaalinen ero 22
6. YHTEENVETO 24
LÄHTEET 26

LIITE A: RINNAKKAISJAETUN-VARJOKARTTATEKNIIKAN C# OHJELMAKOODI

LIITE B: RINNAKKAISJAETUN-VARJOKARTTATEKNIIKASSA KÄYTETYT GLSL VARJOSTIMET

LIITE C: SUORALINJAISESTI VÄÄNNÄTETYSSÄ VARJOKARTTA-TEKNIIKASSA KÄYTETTY C# OHJELMAKOODI

LIITE D: SUORALINJAISESTI VÄÄNNÄTETYSSÄ VARJOKARTTA-TEKNIIKASSA KÄYTETYT GLSL VARJOSTIMET

1. JOHDANTO

Sovellukset, joissa luodaan kolmiulotteinen virtuaalinen ympäristö, käyttävät grafiikan renderointia ympäristön visualisointiin. Renderointi on prosessi, jossa luodaan kaksiulotteinen kuva yhdestä tai useammasta mallista. Malli on kaksi tai kolmiulotteinen pistejoukko, joka kuvaa geometrista kappaletta. Virtuaaliympäristö on kokoelma malleja. Interaktiivisissa sovelluksissa, kuten peleissä, käytetään tosiaikaista renderointia. Tosiaikaisessa renderoinnissa sovelluksen virtuaalinen ympäristö renderoidaan useita kertoja sekunnissa, jotta videokuva virtuaalisesta ympäristöstä olisi sulava.

Interaktiivisissa sovelluksissa tosimaailman ympäristöä jäljittelevä virtuaalinen ympäristö on usein muuttuva, jolloin tarvitaan tosiaikaista valaistusta todenmukaisuuden luomiseksi. Yksi virtuaaliympäristön tosiaikaisen valaistuksen osa on valolähteiden aiheuttamat geometrian varjot.

Virtuaalisen ympäristön varjot ovat usein toteutettu varjokarttojen avulla. Varjokartta on kaksiulotteinen bittikartta, joka sisältää syvyysarvoja. Syvyysarvo kertoo etäisyyden ensimmäiseen geometriseen kappaleeseen valonlähteen näkökulmasta. Varjokartan syvyysarvon perusteella voidaan laskea, mitkä kohdat ovat varjostuneet valonlähteen näkökulmasta. Koska varjokartta on diskreetti bittikartta, sen sisältämä tieto on rajattu sen kokoon. Suurempi virtuaalinen ympäristö aiheuttaa varjojen laadun heikkenemisen, koska varjokartan täytyy kattaa suurempi määrä geometriaa.

Varjokartan luomiseen on useita eri tekniikoita, joissa pyritään tarpeellisen tiedon maksimointiin parhaimman kuvanlaadun saamiseksi. Tässä työssä tutkitaan kahden eri varjokarttatekniikan tehoeroja, eli renderoimiseen kulunutta aikaa, sekä resurssien kulutusta ja toteutuksien kompleksisuutta. Tehokkuutta tutkitaan mittaamalla ruudun piirtoaikoja työtä varten luodussa testiohjelmassa, jossa molemmat varjokarttatekniikat ovat toteutettu. Testiohjelma renderoi saman ympäristön käyttäen molempia tekniikoita. Tutkittavat varjokarttatekniikat ovat rinnakkaisjaettu-varjokartta (engl. parallel-split shadow map, lyh. PSSM) ja suoralinjaisesti väännätetyn bittikartan hyödyntäminen varjokartan luomisessa (engl. rectilinear texture warping for shadow mapping, lyh. RTWSM) [1][2]. Luvussa 2 käsitellään testiohjelmassa virtuaaliympäristön renderoimiseen käytetyn OpenGL-grafiikkarajapinnan ominaisuuksia [3]. Luvussa 3 esitellään varjokartan toimintaperiaate, sekä tutkittavien varjokarttatekniikoiden toimintaa. Luvussa 4 esitellään testiohjelma, tarkastellaan sen toteutusta, sekä käytettyjä tutkimusmenetelmiä, joita käyttäen tulokset saadaan testiohjelmasta. Työn testiohjelmassa toteutetaan virtuaaliympäristön renderointi ja varjokartta käyttäen tutkittavia varjokarttatekniikoita. Testiohjelman avulla suoritetaan testejä, joissa mitataan

molempien varjokarttatekniikoiden piirtoaikaa sekä eroja varjojen visualisoinnissa. Luvussa 5 esitellään testiohjelman avulla saatuja ruudun piirtoaikoja, muistinkulutusta, varjokartta tekniikoiden toteutuksien kompleksisuutta, sekä niiden visuaalisia eroja. Tuloksien perusteella luvussa 6 tehdään yhteenveto eroista varjokarttatekniikoiden muistinkulutuksessa, tehokkuudessa ja kompleksisuudessa. Luvussa myös pohditaan testiohjelman luotettavuutta ja esitellään uusia näkökulmia jatkotutkimuksen kannalta.

2. OPENGL

Tässä työssä toteutettava testiohjelma käyttää OpenGL-grafiikkarajapintaa. Tässä luvussa käsitellään testiohjelmalle ja varjokartan toiminnallisuudelle oleellisia OpenGL-rajapinnan ominaisuuksia. Käyttämällä grafiikkarajapintaa, testiohjelma hyödyntää isäntälaitteen näytönohjainta geometrian renderoinnissa. Grafiikan renderointia kiihdytetään näytönohjaimella sijaitsevalla grafiikkasuorittimella. Jotta sovellus voi käyttää grafiikkarajapintaa renderoinnin kiihdyttämiseen, pitää näytönohjaimen laiteajurin toteuttaa käytetyn rajapinnan määrittelemät kutsut.

OpenGL on vuonna 1992 julkaistu grafiikkarajapinta [4]. Tällä hetkellä uusin versio on 4.6, joka on julkaistu vuonna 2017 [5]. OpenGL:sta on kaksi eri määrittelydokumenttia; ensimmäinen määrittelee ydinprofiilin ja toinen yhteensopivuusprofiilin. Yhteensopivuusprofiili on lisäys ydinprofiiliin, se määrittelee tuen vanhoille OpenGL kutsuille, jotka ovat tarkoitettu käytettäväksi vanhemman sukupolven näytönohjaimilla [3 s. 3]. Tässä luvussa käydään läpi OpenGL:n ydinprofiilin ominaisuuksia version 4.5 määrittelydokumentin perusteella.

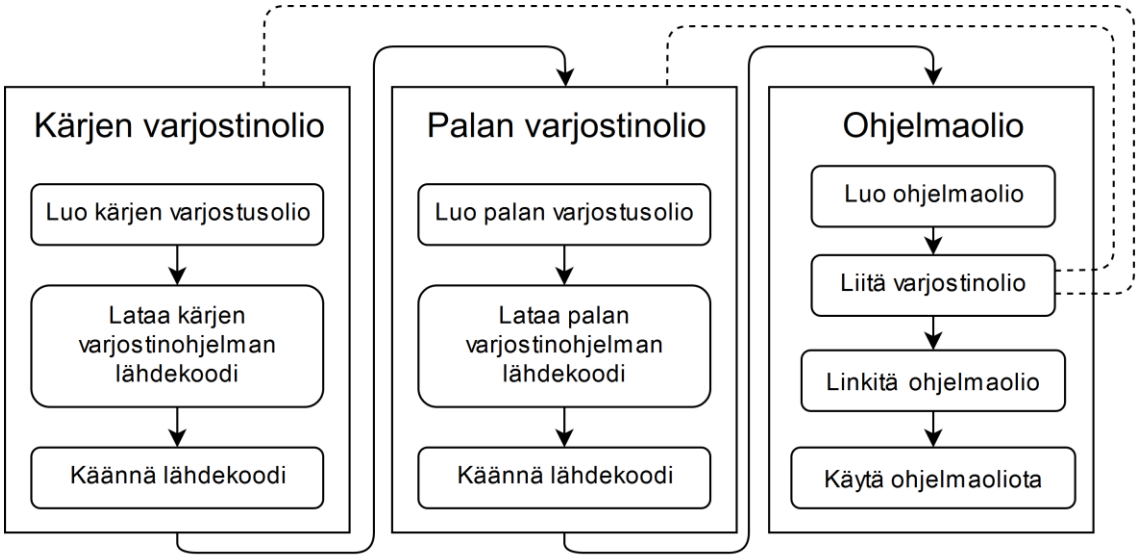
2.1 Toiminta

OpenGL perustuu asiakas-palvelin-malliin. Asiakas (ohjelma) tekee kutsuja palvelimelle (OpenGL), joka tulkkaa kutsut ja suorittaa ne. OpenGL-rajapintaa käyttävän ohjelman alustuksessa luodaan OpenGL *konteksti* ja se liitetään käyttöjärjestelmän ikkunaan. Konteksti pitää yllä OpenGL palvelimen tilan. Ohjelma voida luoda useita kontekseja, esimerkiksi jokaiselle ikkunalle eri kontekstin. Kontekstin luomisen jälkeen ohjelma voi määritellä mikä konteksti valitaan käytettäväksi, jonka jälkeen OpenGL kutsuja voidaan suorittaa. Kutsujen suorittaminen ilman käytettävissä olevaa kontekstia johtaa ohjelman määrittelemättömään käyttäytymiseen. [3, s. 2, 9]

OpenGL määrittelee monia eri tyyppisiä *olioita*, joita ohjelma voi rajapinnan kautta luoda, muokata, hakea ja tuhota. Olio on sidottu yhteen kontekstiin. Sidotut oliot määrittelevät varjostinohjelmat, puskurit, tekstuurit ja ruutupuskurimuistin, joita OpenGL-piirtokomennot käyttävät. Jokaisella oliotyypillä on oma määritelty joukko kutsuja, joiden avulla olion tilaa hallitaan. [3, s. 26-27]

Varjostinolio määrittää varjostinohjelman, joka suoritetaan tietyssä ohjelmoitavassa vaiheessa. Varjostinohjelman ohjelmointikielenä käytetään *OpenGL Shading Language*, lyhennettynä *GLSL*. Varjostinohjelman lähdekoodi ladataan varjostinoliolle, ja se käännetään käytössä olevan näytönohjaimen tukemalle konekielelle. Luotu varjostinolio voidaan liittää yhteen tai useampaan *ohjelmaolioon*. Ohjelmaolio yhdistää yhden tai useamman varjostinolion. Käännetyistä varjostinohjelmista ajettava ohjelmakoodi

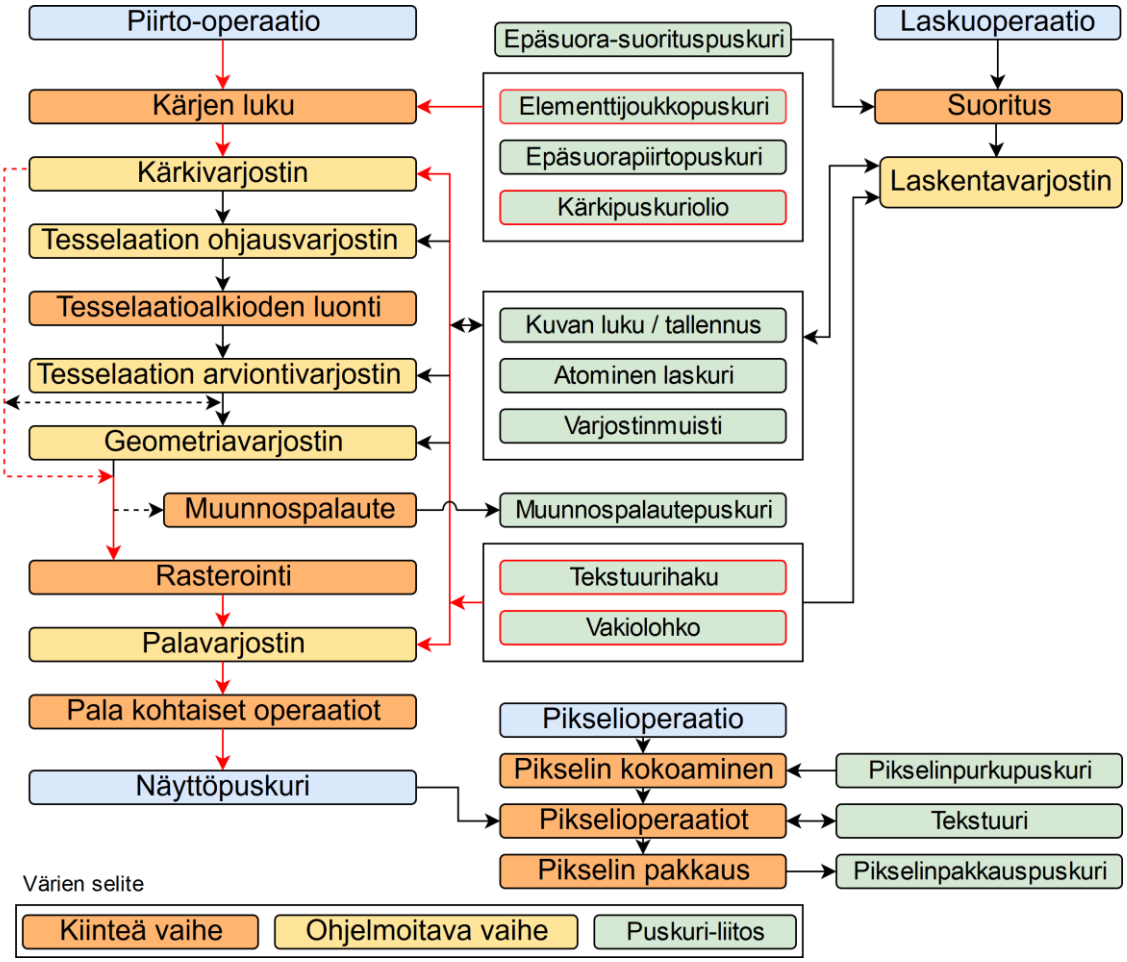
luodaan yhdistämällä (engl. link) ohjelmaolio. Lopulta ohjelmaolio asetetaan käytettäväksi. Kuvassa 1 on esitetty testiohjelman prosessi, miten ohjelmaolio luodaan käyttämällä varjostinolioita ja asetetaan käytettäväksi. [3, s. 84]



Kuva 1. *Prosessi, jota testiohjelman käyttää varjostinohjelmien latauksessa.*

2.2 Liukuhihnamalli

Kuvassa 2 on esitetty OpenGL:n määrittelemä liukuhihnamalli. Nuoli osoittaa tiedon kulkusuuntaa, eli mahdollisia luku- tai kirjoitusoperaatiota liukuhihnassa ja siihen liitetyissä puskureissa tai olioissa. Katkaistu viiva merkkaa liukuhihnan vaihtoehtoista kulkureittiä. Punainen väri merkkaa testiohjelmassa käytettyä kulkureittiä sekä siihen liitettyjä olioita. Mallissa on määritelty kolme eri operaatiota, piirto-, pikseli-, ja laskuoperaatio. Seuraavaksi esitellään piirto-operaatioon kuuluvien vaiheiden toimintaa. Muita operaatioita ei tässä työssä käytetä.



Kuva 2. *OpenGL määrittelydokumentin liukuhihnamalli [3, s. 35].*

Ensimmäisessä vaiheessa *kärki* (engl. vertex) luetaan vaiheeseen liitetyistä puskurista. Kärki on osa renderoitavan geometrian *primitiiviä* (engl. primitive), jonka rakenteen ohjelma määrittelee. Primitiivi voi olla piste, viiva tai monikulmio. Esimerkiksi testiohjelmassa geometria on määritelty joukkona monikolmioita, jonka kärjet koostuvat kolmiulotteisesta pisteestä ja normaalivektorista. Ensimmäiseen vaiheeseen liitettyjen puskurien tyyppejä voi olla *kärkipuskuriolio* (engl. vertex buffer object), *elementtijoukkopuskuri* (engl. element array buffer) ja *epäsuorapiirtopuskuri* (engl. draw indirect buffer). Kärkipuskuriolio sisältää joukon kärkielementtejä [3, s. 349]. Piirrossa käytetyn kärjen sijainti osoitetaan elementtijoukkopuskurilla, joka sisältää joukon indeksejä osoittamaan kärjen sijainti kärkipuskurissa [3, s. 350]. Testiohjelmassa ensimmäiseen vaiheeseen on liitetty kärki- ja elementtijoukkopuskuri oliot. Eli kärki saadaan kärkipuskuriolion kärkielementtijoukosta elementtijoukkopuskurin osoittaman indeksin avulla.

Kun kärki on luettu, suoritetaan kärjen käsittely. Ensimmäisenä suoritetaan kärjen varjostus, joka on ohjelmoitava vaihe. Siinä kärjenvarjostin määrää mitä operaatioita suoritetaan kärjelle [3, s. 386]. Esimerkiksi testiohjelmassa kärjenvarjostin muuntaa kärjen pisteen virtuaalikameran määrittelemän projektiokoordinaatistoon. Muuntaminen

tapahtuu käyttämällä virtuaalikameran projektiomatriisia. Sitä ennen kärkeä voidaan myös muuntaa kärjen ja kameran muunnosmatriisilla. Kärjen varjostamisen jälkeen suoritetaan valinnainen *tesselaatio* (engl. tessellation), joka luo uusia primitiivejä vaiheeseen syötetystä kolmiosta tai neliöistä, jakaen ne edelleen varjostinohjelman määritelmän mukaisesti [3, s. 391-392]. Tässä työssä tesselaatiovaihetta ei käytetä, joten sen käsittely jätetään pois. Seuraavaksi suoritetaan valinnainen geometriavarjostin, joka käsittelee primitiiviä ja voi päästää yhden tai useamman primitiivin ulostuloon [3, s. 412]. Viimeisenä kärjen käsittelyvaiheessa suoritetaan kiinteitä operaatioita. Esimerkiksi yhdessä operaatiossa kuvaportin ulkopuolelle jääneet primitiivit hylätään, tai leikataan, jos ne sijaitsevat kuvaportin reunalla [3, s. 423]. Vaihtoehtoinen *muunnospalautevaihe* (engl. transform feedback) kirjoittaa vaiheeseen liitettyyn puskuriin siinä hetkellä olevan primitiivien kärkien ominaisuudet [3, s. 424]. Geometriavarjostimen ja muunnospalautevaiheen toimintaa ei käsitellä laajemmin, koska niitä ei tässä työssä käytetä.

Kärjen käsittelyn jälkeen suoritetaan primitiivien rasterointi, jossa primitiivi muunnetaan kaksiulotteiselle bittikartalle. Rasteroinnissa määritetään primitiivin alue ikkunakoordinaatistossa [3, s. 443]. Rasteroinnissa muodostuneen yksittäisen bittikartan elementtiä kutsutaan nimellä *pala* (engl. fragment). Ennen palojen varjostusta, niille suoritetaan testejä, joissa pala voidaan hylätä tai muokata [3, s. 464]. Esimerkiksi testiohjelmassa palakohtainen syvyystesti hylkää palan, jos palan syvyys (z) on suurempi kuin palan koordinaatin (x, y) kohdalla olevan syvyyspuskurilla sijaitseva arvo [3, s. 484]. Syvyysarvo määrittää palan etäisyyden virtuaalikamerasta. Lopulta ohjelmoitava palavarjostin käsittelee palan, jonka jälkeen suoritetaan pala kohtaisia operaatioita ennen sen arvon kirjoittamista kuvapuskuriin [3, s. 469]. Palan syvyysarvo kirjoitetaan *syvyyspuskuriin* (engl. depth buffer, Z-buffer).

Piirto- ja laskuoperaation ohjelmoitaviin vaiheisiin on mahdollista liittää puskureita tai muistioliota. Kuvassa 2 on erikseen jaoteltu vain luettavissa olevat liitokset, ja luettavissa sekä kirjoitettavissa olevat liitokset. Testiohjelmassa kärki- ja palavarjostimet käytävät luettavissa olevaa vakiolohkoa (engl. uniform block) ja tekstuurihakua (engl. texture fetch). Vakiolohkoon on asetettu ruudun aikana käytetyn virtuaalikameran ja renderoitavan geometrian muunnosmatriisit. Tekstuurihakua käytetään bittikarttojen lukemiseen varjostimissa.

3. VARJOKARTTA

Varjokartta on kaksiulotteinen bittikartta, jonka tarkoitus on tässä työssä kertoa, onko alue varjostunut jonkin valonlähteen näkökulmasta. Interaktiivisissa visuaalisissa sovelluksissa, kuten peleissä, virtuaalimaailman valonlähteet sekä geometria ovat usein dynaamisia. Dynaamisissa virtuaalisissa ympäristöissä varjokartta joudutaan luomaan uudestaan, kun valonlähteen tai geometrian sijainti virtuaaliympäristössä muuttuu. Tässä luvussa tutkitaan varjokartan toimintaa ja esitellään kaksi testiohjelmassa käytettävää varjokartan luontitapaa.

3.1 Toiminta

Yksinkertainen algoritmi varjokartan luomiseen on esitelty Lance Williamssin julkaisussa [6]. Siinä hyödynnetään renderoinnissa syntyvää syvyyspuskuria. Ensimmäisenä ympäristön geometria renderoidaan valonlähteen näkökulmasta, jolloin geometrian syvyysarvot kirjoitetaan syvyyspuskuriin. Tässä vaiheessa geometrian rasteroinnissa syntyneiden palojen varjostusta ei tarvitse suorittaa, koska vain syvyysarvoja käytetään hyväksi. Seuraavaksi ympäristö renderoidaan havainnoitsijan näkökulmasta. Kun palan varjostus suoritetaan, sen sijainti muunnetaan valonlähteen koordinaattiavaruuteen, jossa arvot vaihtelevat joka akselilla välillä [-1, 1]. Koska bittikartan luku tapahtuu käyttämällä x- ja y koordinaatiston liukulukuja välillä [0, 1], normalisoidaan sijainti edelleen bittikartan avaruuteen. Tätä sijaintia käyttämällä luetaan valonlähteen näkökulmasta renderoidyn geometrian syvyyskartasta syvyysarvo. Seuraavassa esimerkki algoritmissa muunnetaan kärjen sijainti valonlähteen avaruuteen ja edelleen syvyyskartan avaruuteen. Saatua sijaintia käytetään lopulta palavarjostimessa varjokartan arvon lukemiseen.

```
kärkiValonAvaruudessa = valonlähteenMuunnosMatriisi * kärjenSijainti;
varjoKoordinaatti = (kärkiValonAvaruudessa.xyz * 0.5) + 0.5;
```

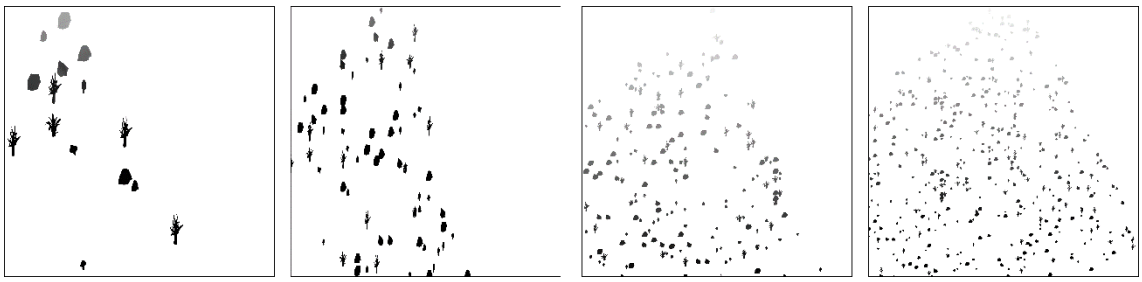
Syvyysarvo kertoo sijainnin kohdalla valonlähdettä lähimpänä olevan geometrian syvyysarvon. Vertaamalla valonlähteen koordinaatistoon muunnetun havainnoitsijan palan syvyysarvoa luettuun syvyysarvoon, saadaan tieto, onko pala varjossa. Seuraavaksi esimerkki palavarjostimessa suoritettavasta pseudoalgoritmista, jonka avulla päätellään, onko pala varjossa:

```
bool varjossa = false;
float lähinSyvyys = texture(syvyyskartta, varjoKoordinaatti.xy).z;
float syvyys = varjoKoordinaatti.z;
if (lähinSyvyys < syvyys.z)
{
  varjossa = true;
}
```

Kun virtuaalimaailman ja renderoitavan näköalueen koko kasvaa, täytyy varjokartan myös sisällyttää enemmän tietoa geometrian syvyysarvoista. Kun virtuaalimaailma on todella iso, ja koska varjokartta on diskreetti bittikartta, sen tarjoaman tiedon perusteella ei voida luoda visuaalisesti laadukkaita varjoja. Varjojen visuaalinen laatu riippuu siis varjokartan ja virtuaalimaailman koon suhteesta. Varjokartan sisältämän tarpeellisen syvyystiedon maksimointiin on olemassa useita eri tekniikoita. Tarpeellisella syvyystiedolla tarkoitetaan niiden geometrioiden syvyystietoja, joiden luomat varjot päätyvät virtuaalikameran näköalueeseen. Esimerkiksi yksinkertaisen varjokarttatekniikan bittikartta sisältää syvyystietoa geometrioista, joiden luomat varjot jäävät lopulta virtuaalikameran näköalueen ulkopuolelle. Seuraavaksi esitellään kaksi eri tekniikkaa optimoida tarpeellisen tiedon määrää varjokartassa.

3.2 Rinnakkaisjaettu-varjokartta

Zhang *et al.* esittävät julkaisussaan varjokarttatekniikan, joka on erityisesti tarkoitettu käytettäväksi laajoissa virtuaaliympäristöissä [1]. Tekniikassa kameran näköalue V jaetaan useisiin alueisiin syvyyssuunnassa. Virtuaaliympäristössä kameran näköalue on muodoltaan *katkaistu pyramidi* (engl. frustum). Jokaiselle jaetun näköalueen osalle V_i renderoidaan oma varjokartta T_i. Lopulta, kun ympäristö renderoidaan kameran näkökulmasta, luetaan renderoitavan palan kohdalta valonlähteen syvyysarvo näköalueen osan V_i varjokartasta T_i. Näköalueen osa V_i, jossa pala sijaitsee, saadaan vertaamalla palan syvyysarvoa ja osan V_i sijaintia syvyyssuunnassa. Kuvasta 3 huomataan, että varjokartan kattama alue laajenee riippuen näköalueen osan V_i suuruudesta.



Kuva 3. *Testiohjelman rendoiroimat varjokartat {T_1, T_2, T_3, T_4} vasemmalta oikealle.*

Yksi tekniikan haasteista on näköalueen osien sijaintien $\{C_i\}$ määrittely. Tavoitteena on, että varjojen visuaalinen laatu pysyisi samana osien välillä. Toisin sanoen pyritään välttämään yli- tai alinäytteistämistä näköalueen osan varjokartan renderoinnissa. Julkaisussa esitellään kolme eri menetelmää kameran näköalueen jakamiseen; tasainen, logaritminen ja käytännöllinen jako. Tasaisesti jaetussa näköalue jaetaan syvyyssuunnassa saman pituisiin osiin. Logaritmisessa jaossa jako tapahtuu logaritmisesti. Käytännöllisessä jaossa yhdistetään tasainen ja logaritminen jako käyttäen niiden keskiarvoa. Julkaisussa todetaan, että käytännöllinen jako on parhain. Tasaisessa jaossa ensimmäisestä osa-alueesta tulee liian suuri ja taaimmaisista liian pieni.

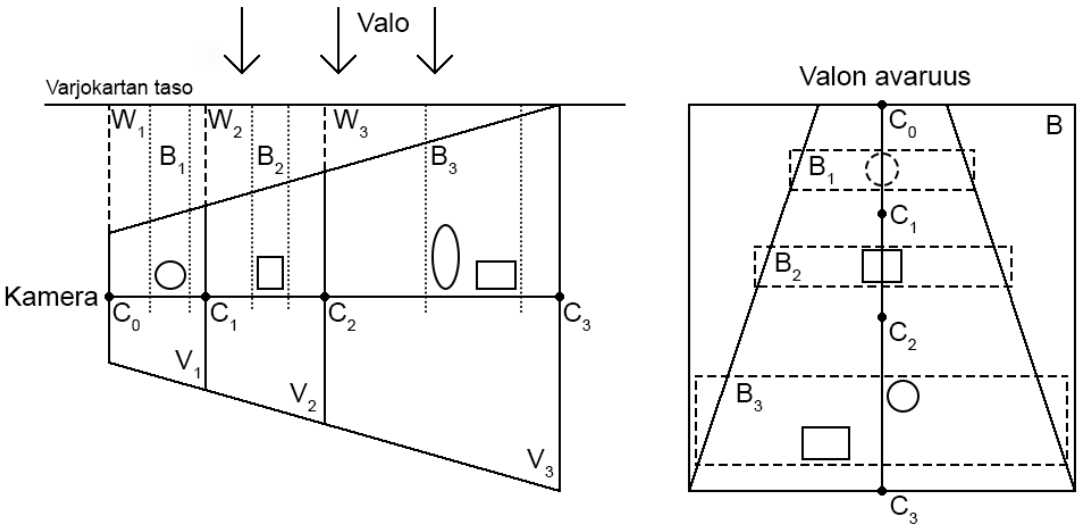
Logaritmisessa jaossa taas lähimmäisistä alueista tulee liian pieni ja taaimmaisesta liian suuri. Kun otetaan keskiarvo tasaisesta ja logaritmisesta jakomenetelmästä, saadaan tasainen yhdistelmä molemmista ääripäistä, käytännöllinen jako:

$$C_i = \frac{n\,(f/n)^{i/m} + n + (f-n)i/m}{2}, \tag{1}$$

jossa n on kameran lähitason syvyys, f kameran etätason syvyys ja m alueiden määrä. Kuvassa 4 havainnollistetaan kaavaa 1 käyttäen näköalueiden osien sijoittuminen kameran näkökulmasta.

Kuva 4. *Testiohjelman renderoima lopputulos, jossa havainnollistetaan näköalueen jako neljään osaan {V_1, V_2, V_3, V_4} värisävyä tummentamalla.*

Valonlähteen näköalue *W* jaetaan yhtä moneen osaan kuin kameran näköalue *V*. Jokainen osa W_i saadaan, kun lasketaan valonlähteen koordinaattien akselien suuntainen rajausalue V_i ja *W* alueiden leikkauksesta. Kun objektien koko ja sijainnit ovat tiedossa, voidaan edelleen laskea tiukemmat rajausalueet $\{B_i\}$ valonlähteen osien alueille. Tiukemman rajauksen myötä saadaan hyödynnettyä varjokartan resoluutiota paremmin. Kuvassa 5 on havainnollistettu näköalueen osien jakautumista, sekä rajausalueita kameran ja valonlähteen näkökulmasta.



Kuva 5. *Näköalueiden osien jakautumista havainnollistaminen kameran ja valon koordinaattiavaruudessa [1].*

Testiohjelmassa rinnakkaisjaettu-varjokartta on toteutettu seuraavien vaiheiden mukaisesti:

1. Etsitään kaikki kameran näköalueella V sijaitsevat geometriaobjektit. Näin vältytään lähettämästä turhia renderointikomentoja objekteille, jotka eivät sijaitse näköalueella.

2. Mukautetaan kameran näköalueen, eli katkaistun pyramidin, lähi- ja etätason syvyydet. Syvyydet saadaan, kun etsitään renderoitavien geometriaobjektien lähimmät ja kauimmat pisteet kameran sijainnista.

3. Jaetaan kameran näköalue V osiin $\{V_i\}$ käyttäen käytännöllistä jakoalgoritmia. Testiohjelmassa alueiden määräksi on valittu kolme. Tämä valinta on tehty sen perusteella, koska julkaisussa todettiin, että kolme aluetta tuotti parhaan tuloksen tehokkuuden ja visuaalisuuden kannalta.

4. Jokaiselle kameran näköalueen osan V_i ja valonlähteen näköalueen W leikkaukselle W_i etsitään geometriaobjektit, jotka sijaitsevat siinä. Nämä geometriaobjektit ovat niitä, jotka renderoidaan varjokartalle.

5. Jokaiselle alueelle W_i lasketaan leikkausmatriisi käyttäen vaiheissa 1 ja 4 löydettyjä geometriaobjekteja. Leikkausmatriisin avulla alue W_i tiukennetaan lopulliseen alueeseen B_i.

6. Jokaiselle alueelle V_i renderoidaan varjokartta T_i käyttäen valolähteen optimoitua näköaluetta B_i. Tässä vaiheessa varjokartalle renderoidaan vain vaiheessa 4 löydetyt geometriaobjektien syvyysarvot, eli alueelta W_i löydetyt varjostajat.

7. Lopuksi renderoidaan virtuaaliympäristö kameran näkökulmasta, jolloin hyödynnetään varjokarttoja varjojen luomiseen. Lopputulos näytetään näyttöpuskurissa.

3.3 Suoralinjaisesti väännätetty varjokartta

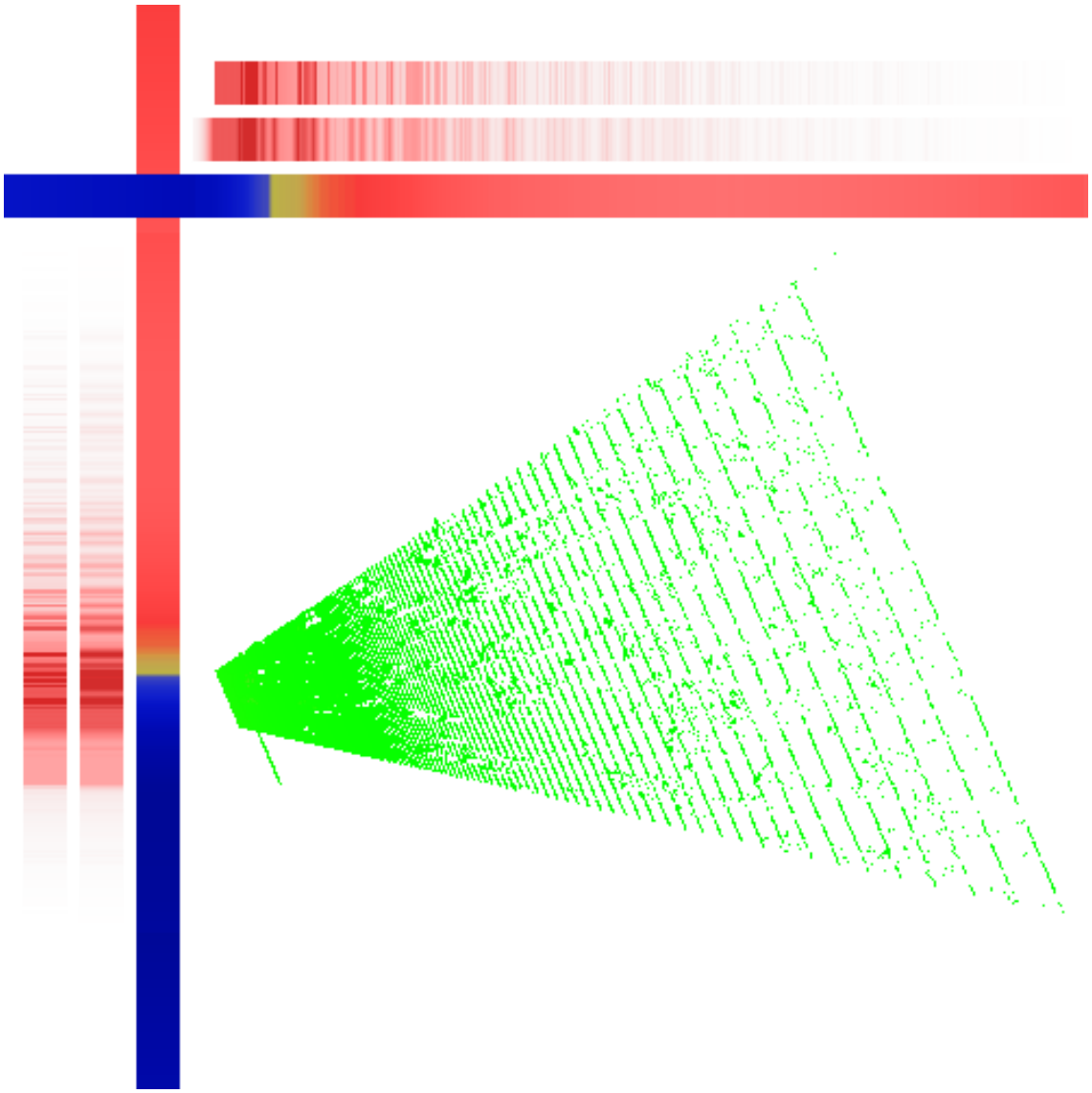
Rosen esittelee julkaisussaan varjokarttatekniikan, joka poistaa tarpeen kameran näköalueen osittamiselle [2]. Esitetyllä varjokarttatekniikalla päästään eroon osittamisesta johtuvien varjojen laadun tasoerojen suuresta vaihtelusta osien välillä. Tekniikka perustuu varjokartan väännättämiseen. Vääntäminen suoritetaan kahden yksiulotteisen *vääntymiskartan* (engl. warping map) avulla, yksi vertikaaliselle ja toinen horisontaaliselle akselille. Vääntymiskartat luodaan *tärkeyskartan* (engl. importance map) avulla, joka kertoo kameran näköalueella varjostamiselle tärkeät kohteet valonlähteen näkökulmasta. Tärkeyskartta muodostetaan *tärkeysfunktioiden* (engl. importance functions) avulla, joita ovat esimerkiksi geometrian etäisyys kamerasta ja geometrian pinnan normaali.

Tärkeyskartta koostuu positiivisista liukuluku arvoista. Suurempi arvo merkitsee suurempaa tärkeyttä, nolla arvo merkitsee ei tarvittavaa tietoa. Julkaisu esittelee kolme eri tapaa tärkeyskartan luomiseen, suora-, käänteinen- ja risteymäanalyysin. Suora-analyysissä ympäristö renderoidaan valonlähteen näkökulmasta syvyyskarttaan, josta tärkeystiedot saadaan suoraan lukemalla. Käänteisanalyysissa renderoidaan ympäristö kameran näkökulmasta tallentaen syvyys- ja väritiedot. Tärkeystiedot saadaan projisoimalla renderoity ympäristö valonlähteen näkökulmaan. Risteymäanalyysissä suoritetaan molemmat analyysit ja yhdistetään niiden tulokset. Testiohjelmassa valittiin käänteisanalyysi tärkeyskartan luomiseksi, koska julkaisussa todettiin sen antavan parhaimman tehokkuuden.

Kun tärkeyskartta on luotu, voidaan rakentaa vääntymiskartat. Aluksi tiivistetään tärkeyskartta kahdeksi yksiulotteiseksi kartaksi, jossa ensimmäinen kuvaa horisontaalista ja toinen vertikaalista akselia. Tiivistys suoritetaan etsimällä tärkeyskartasta suurin tärkeysarvo jokaisesta rivistä ja sarakkeesta. Rivien maksimiarvot siis sijoitetaan vertikaaliseen karttaan ja sarakkeiden horisontaaliseen karttaan. Molemmille kartoille suoritetaan Gauss-sumennus. Sumennus auttaa pehmentämään näytteenottotaajuuden muutosta, sekä pienentämään väännättämisen aiheuttamia virheitä rasteroinnissa. Lopulliset vääntymiskartat saadaan, kun sumennettujen karttojen I_x ja I_y alkioille $\{k_n\}$ lasketaan painoarvo kaavalla:

$$Vääntymä(k) = \left(\sum_{j=1}^{k-1} I_j \Big/ \sum_{j=1}^{n} I_j\right) - \left(\frac{k}{n}\right). \quad (2)$$

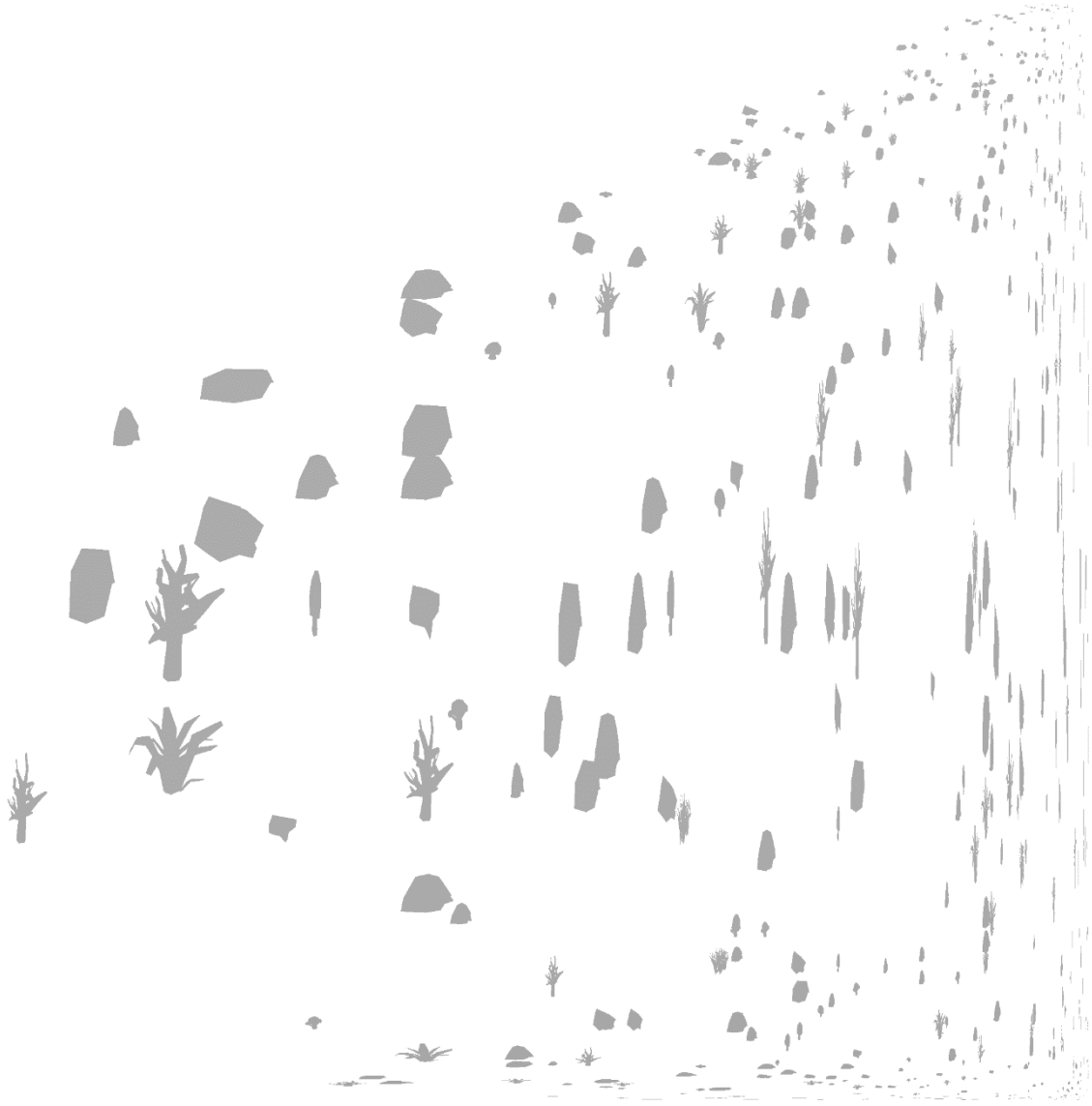
Kuvasta 6 huomataan kuinka tiivistetyt tärkeyskartat (reunimmaiset punaiset nauhat) painottavat kameran lähialuetta ja häivenevät kauemmaksi siirtyessä. Lopulliset vääntymiskartat (puna-sininen nauha) kertovat siis mihin suuntaan ja kuinka paljon kärkeä siirretään valonlähteen koordinaattiavaruudessa. Tässä kuvassa sininen tarkoittaa negatiiviseen suuntaan ja punainen positiiviseen. Kuvassa näkyy myös hyvin pistejoukon projisointi kamerasta virtuaaliympäristöön.



Kuva 6. *Testiohjelman renderoimien tärkeyskartan (vihreät pisteet) ja vääntökarttojen visualisointi (puna-sininen nauha).*

Nyt varjokartan renderointi voidaan suorittaa vääntymiskarttojen avulla. Kun kärkeä varjostetaan ja se on projisoitu valonlähteen avaruuteen, haetaan sen x- ja y-koordinaatin siirtymä vääntymiskartoista sen sijaintia käyttäen. Kuvasta 7 huomataan, kuinka

geometriaa on väännätetty priorisoiden niitä, jotka sijaitsevat lähellä kameraa. Myös kaukana tai kameran näköalueen ulkopuolella sijaitsevat geometriaobjektit ovat supistuneet vieden vähän tilaa varjokartalta. Hyödyntämällä paremmin varjokartan resoluutiota, saadaan parannettua huomattavasti lähellä olevien geometrioiden varjojen visuaalista laatua.



Kuva 7. *Testiohjelman renderoima varjokartta käyttäen kuvassa 6 esitettyä vääntymiskarttaa.*

Testiohjelmassa suoralinjaisesti väännätetty varjokartta on toteutettu seuraavien vaiheiden mukaisesti:

1. Virtuaaliympäristö renderoidaan kameran näkökulmasta, tallentaen syvyys-, normaali- ja väritiedon erillisille bittikartoille.

2. Tärkeyskartta luodaan renderoimalla kamerasta projisoituva pistejoukko valonlähteen näkökulmasta. Pisteet projisoidaan vaiheessa 1 luodun syvyyskartan

avulla. Pistejoukko siis havainnollistaa valonlähteen näkökulmasta mitä kamera näkee. Esimerkki tärkeyskartasta kuvassa 6.

3. Palavarjostimen avulla suoritetaan tärkeyskartan tiivistys, jossa tärkeyskartan I_{xy} rivien ja sarakkeiden maksimiarvot sijoitetaan yksiulotteisille bittikartoille I_x ja I_y.

4. Vaiheessa 3 luoduille bittikartoille suoritetaan Gauss-sumennus palavarjostinta käyttäen.

5. Lopulliset vääntymiskartat saadaan, kun palavarjostimessa lasketaan sumennettujen bittikarttojen alkioiden painoarvot kaavalla 2. Kuvassa 6 on visualisoitu esimerkki vääntymiskartoista.

6. Varjokartta renderoidaan ja kärkien sijaintia valonlähteen avaruudessa siirretään vääntymiskartan mukaan.

7. Lopuksi lasketaan varjostus ja yhdistetään se vaiheessa 1 luotuun värikarttaan. Varjokartan luku koordinaatteja siirretään vääntymiskartan mukaan. Tämä tulos näkyy näyttöpuskurissa.

4. TESTIOHJELMA JA MENETELMÄT

Testiohjelma toteuttaa luvussa 3 esitetyt varjokarttatekniikat. Testiohjelma on interaktiivinen sovellus, joka renderoi virtuaalisen ympäristön. Testiohjelman avulla mitataan esimerkiksi ruudunpiirtoaika, jonka avulla tehdään päätelmiä varjokarttatekniikoiden tehokkuuksista.

4.1 Testiohjelman toteutus

Testiohjelma on toteutettu C#-ohjelmointikielellä. Tämä ohjelmointikieli valittiin työn tekijän aikaisemman kokemuksen perusteella. Testiohjelma käyttää OpenTK kirjastoa, jonka avulla saadaan tuki OpenGL-rajapinnalle, käyttöjärjestelmän ikkunointirajapinnalle, sekä lineaarialgebralle [7]. Ohjelma on kehitetty Windows 10 ympäristössä käyttäen Visual Studio Community-ohjelmaa [8]. C#-ohjelmointikieli ja OpenTK kirjasto tukevat myös Mac ja Linux ympäristöjä, mutta testiohjelman toimivuutta niissä ei ole testattu. Ohjelmakoodin virheenkorjauksessa on käytetty Visual Studio Community-ohjelmaa ja grafiikkan renderoimisen virhekorjauksessa RenderDoc-ohjelmaa [9].

4.1.1 Virtuaaliympäristön luonti

Ohjelmassa käytetty virtuaaliympäristö ladataan tekstitiedostosta. Se kertoo ladattavan geometriaobjektin tyypin ja sijainnin. Virtuaaliympäristössä käytettyjen geometriaobjektien 3d-mallit ovat ladattu BlendSwap palvelusta ja ovat lisensoitu Creative Commons Zero-lisenssillä [10]. Mallit ovat muunnettu *obj*-tiedostoformaattiin, jotta testiohjelmaan mallien luku on helpompaa [11]. Ohjelmassa *obj*-tiedostojen purkamiseen on käytetty *ObjLoader* kirjastoa [12]. Virtuaaliympäristössä käytetyt mallit kuvastavat puita, kiviä ja viherkasvustoa.

Tekstitietoon sijoitettujen geometriaobjektien sijainnit ovat luotu .NET kirjaston satunnaisgeneraattorilla [13]. Jotta varjot näkyisivät selvästi, ympäristön origon ympärille neliön muotoiseen alueeseen lisätään 400 kappaletta viisi yksikköä pitkiä ja leveitä neliöitä, jotka toimivat alustana muulle geometrialle. Alusta koostuu monesta osasta siksi, että turhat alueen osat voidaan jättää pois, jotta valonlähteen näköalue voidaan optimoida paremmin. Geometriaobjektien sijainnit ovat luotu neliön alueelle, jolloin sijaintien x- ja z-koordinaatit vaihtelevat alueella [-50, 50]. Yhteensä ympäristö koostuu kokonaisuudessaan 2 609 160 kärjestä.

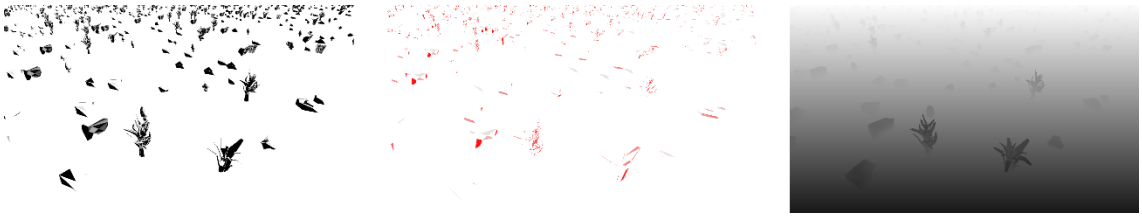
4.1.2 Rinnakkaisjaetun-varjokartan toteutus

Ohjelmassa toteutetun rinnakkaisjaettu-varjokarttatekniikan C# toteutus on ohjelmakoodina liitteessä A. Luokan *ParallelSplitShadowMapRenderer* sisältämä *Render*-funktio suorittaa virtuaaliympäristön renderoinnin sekä varjokartan luomisen. Funktion parametri *Scene* sisältää virtuaaliympäristön tilan. Funktio *UpdateSplitPositions* päivittää näköalueen osien sijainnit luvun 3 kaavan (1) mukaisesti listamuuttujaan *mSplitPositions*. Funktiota *RenderCasters* kutsutaan kameran näköalueen jokaiselle osalle. Funktio renderoi sen alueen ja valonlähteen näköalueen leikkauksen sisällä olevat geometriaobjektien syvyysarvot varjokartalle. Tässä testiohjelman toteutuksessa varjokartta on testistä riippuen 2048 * 2048 tai 4096 * 4096 pikselin kokoinen kaksikanavainen kaksiulotteinen bittikartta, jonka pikselin syvyyskanavalle on varattu 24-bittiä. Varjokartassa pikselin tyypiksi on määritelty merkitön kokonaisluku. Varjokarttoja on kolme kappaletta, jokaiselle kameran näköalueen osalle oma. Luokassa varjokarttoja kuvataan listamuuttujalla *mShadowMap*. *RenderCaster*-funktion käyttämä kärkivarjostin (*pssm_shadow.vert*) on liitteessä B.

Lopulta kun kaikille kameran näköalueen osille on luotu varjokartta, suoritetaan funktio *RenderReceivers*, joka renderoi lopullisen tuloksen näyttöpuskurille. Sen käyttämät kärki- ja palavarjostin (*pssm_scene.vert, pssm_scene.frag*) ovat liitteessä B.

4.1.3 Suoralinjaisesti väännätetyn varjokartan toteutus

Ohjelmassa toteutetun suoralinjaisesti väännätetyn varjokarttatekniikan C# toteutus on ohjelmakoodina liitteessä C. Ohjelmakoodi ja siinä käytettävät varjostimet perustuvat Rosenin julkiseen esimerkkiohjelmakoodiin, joka on edelleen työn tekijän toimesta sovitettu C# ja OpenGL ympäristöön [14]. Luokan *RTWShadowMapRenderer* funktio *Render* suorittaa virtuaaliympäristön renderoinnin sekä varjokartan luomisen. Funktiossa kutsutaan edelleen *GenerateSurface*-funktiota, jonka tarkoituksena on renderoida ympäristö kameran näkökulmasta tallentaen väri-, normaali- ja syvyystiedot bittikarttoihin. Näitä bittikarttoja kuvaa luokan muuttujat *mColorTexture*, *mCameraDepthMap* ja *mDotNormalTexture*. Funktiossa *GenerateSurface* käytetyt kärki- ja palavarjostimet (*rtw_depth_image.vert, rtw_depth_image.frag*) ovat liitteessä D. Kuvassa 6 on havainnollistettu tässä vaiheessa renderoidyt bittikartat, väri-, normaali- ja syvyyskartat, vasemmalta oikealle.



Kuva 8. *Testiohjelmassa kameran näkökulmasta renderoidyt väri-, normaali ja syvyyskartat.*

Tämän jälkeen kutsutaan funktiota *GenerateImportance,* joka luo tärkeyskartan. Funktiossa renderoidaan kaksiulotteinen pistejoukko, jonka koko on (ikkunanleveys / 4) * (ikkunankorkeus / 4) (testiohjelmassa 320 * 180) ja niiden sijainnit ovat tasaisesti normalisoitu alueelle [0, 1] akseleilla x ja y. Funktiossa käytetyt kärki- ja palavarjostimet (*rtw_importance.vert, rtw_importance.frag*) ovat liitteessä D. Kärkivarjostimessa piste muunnetaan takaisin kameran kuvaportin koordinaattiavaruudesta virtuaalimaailman koordinaattiavaruuteen käyttämällä kameran käänteismatriisia ja kameran syvyyskarttaa. Tämän jälkeen se muunnetaan valonlähteen kuvaporttiin. Palavarjostimessa lasketaan tärkeysarvot tärkeysfunktioiden avulla, joita ovat pisteen etäisyys kamerasta (syvyys) ja pinnan normaali. Tärkeyskarttaa kuvaa muuttuja *mImportanceTexture*. Tärkeyskartaksi on valittu yksikanavainen kaksiulotteinen bittikartta 512 * 512 pikseliä. Pikselin tyypiksi on määritelty 16-bittinen liukuluku.

Kun tärkeyskartta on luotu, luodaan sen perusteella vääntymiskartat. Vääntymiskarttoja kuvaa yksikanavainen kaksiulotteinen bittikartta, jonka koko on 512 * 2 pikseliä. Pikselin tyypiksi on määritelty 32-bittinen liukuluku. Tässä esitysmuodossa ensimmäinen rivi vastaa y-akselin vääntymiskarttaa ja toinen rivi x-akselin vääntymiskarttaa. Funktio *GenerateWarpMap* suorittaa kolme eri renderointioperaatiota, joiden palavarjostimet ovat liitteessä D. Kaikista operaatiosta tallennetaan tekstuuri listamuuttujaan *mWarpTexture*. Ensimmäisenä suoritetaan tärkeyskartan kutistus palavarjostimessa (liite D, *rtw_compact.frag*). Seuraavaksi kutistettu tärkeyskartta sumennetaan palavarjostimessa Gauss-sumennus algoritmilla (liite D, *rtw_blur.frag*). Viimeiseksi sumennetusta tärkeyskartasta luodaan vääntymiskartta palavarjostinta hyödyntäen (liite D, *rtw_buildwarp.frag*). Nyt lopulliset x- ja y-akselin vääntymiskartat ovat muuttujan *mWarpTexture*:n viimeisessä alkiossa.

Nyt funktiossa *GenerateShadowMap* varjokartta renderoidaan ja kärkivarjostimessa kärkeä siirretään kuvaportin avaruudessa vääntymiskarttojen mukaisesti. Ohjelmakoodissa varjokarttaa kuvaa muuttuja *mShadowMapTexture*. Ohjelmassa varjokartta on testistä riippuen 2048 * 2048 tai 4096 * 4096 pikselin kokoinen kaksikanavainen kaksiulotteinen bittikartta, jonka pikselin syvyyskanavalle on varattu 24-bittiä. Pikselin tyyppi on merkitön kokonaisluku. Funktion käyttämä kärkivarjostin on liitteessä D (*rtw_shadow.vert*).

Lopulta funktiossa *RenderComposite*, jo luodun värikartan päälle lasketaan varjostus varjokartan, vääntymiskartan ja kameran syvyyskartan avulla. Funktiossa käytetyt kärki- ja palavarjostimet (*rtw_scene.vert, rtw_scene.frag*) ovat liitteessä D.

4.2 Tutkimusmenetelmät

Tutkitusmenetelmissä tehokkuuden mittariksi on asetettu ruudunpiirtoaika, joka mitataan millisekunneissa. Ruudunpiirtoaika on aika, joka mitataan siitä hetkestä, milloin ensimmäinen renderointikutsu suoritetaan ja viimeinen lopetetaan. Testiohjelmassa mitataan 500 ruudun piirtoon kulunut aika millisekunneissa käyttäen eri varjokarttatekniikoita ja varjokarttaresoluutioita. Testissä varjokarttaresoluutioksi on valittu 2048 * 2048 ja 4096 * 4096. Kaikissa testeissä kameran ja valon näköalue on sama. Näköalue on valittu käsin ja se pyrkii sisältämään geometriaobjekteja, jotka sijaitsevat kaukana ja lähellä kameraa. Näköalue ja geometriat eivät liiku ruudun piirtojen aikana, eli ympäristö on staattinen mittauksen aikana. Piirtoaika on mitattu käyttämällä .NET Frameworkin sisältämää *Stopwatch*-luokkaa [15]. Luokan avulla saa selville, kuinka monta askelta (engl. tick) prosessori etenee sekunnissa (esimerkissä *Frequency*) ja tietyn mittauksen aikana (esimerkissä *ElapsedTicks*). Testilaitteistolla *Stopwatch*-luokka ilmoitti sekunnissa tapahtuvan 3 222 661 askelta, joten lopullinen teoreettinen mittaustarkkuus on noin 310 nanosekunttia. Seuraavaksi esimerkki, miten mittaus on suoritettu testiohjelmassa:

```
stopwatch.Start();
shadowRenderer.Render(scene);
stopwatch.Stop();
double ms = (stopwatch.ElapsedTicks / stopwatch.Frequency) * 1000;
```

Toteutuksien kompleksisuutta tutkitaan ohjelmakoodirivien määränä, sekä ajonaikaisena muistinkulutuksena. Ohjelmakoodirivien määrän laskussa käytetään *cloc*-ohjelmaa, joka ei laske mukaan esimerkiksi kommentteja tai tyhjiä rivejä [16]. Ohjelmakoodiriveihin lasketaan toteutuksessa käytettävät varjostimet eli *GLSL*-tiedostot ja asiakasohjelmistossa varjokarttatekniikan toteuttava C#-tiedosto. Muistinkulutus lasketaan teoreettisesti etukäteen, koska varjokarttaa edustavan bittikartan koko ja pikselin tyyppi ovat tiedossa ennen ohjelman ajoa. Varjokarttoja ei myöskään muuteta ohjelman ajonaikana.

Visuaalisessa tutkimuksessa pyritään tutkimaan toteutuksien eroja laskostumisessa. Laskostuminen kertoo oleellisesti, onko varjokartan näytteenottotaajuuksissa eroja.

4.3 Laitteisto

Kaikki kokeet suoritettiin PC:llä, jossa on Intel i7-5820K prosessori, 32 gigatavua DDR4 keskusmuistia ja nVidia Geforce GTX 1080 näytönohjain. Testiohjelmaa ajettiin Windows 10 käyttöjärjestelmällä. Testiohjelman ikkunaresoluutio oli 1280x720. Kyseinen kokoonpano valittiin siksi, koska se on helposti työn tekijän käytettävissä.

5. TULOKSET

Tässä luvussa esitellään testiohjelman toteutuksesta ja sen suorittamasta mittauksesta saadut tulokset. Testiohjelmaa, tuloksien mittaustapaa ja testilaitteistoa on käsitelty tarkemmin luvussa 4.

5.1 Toteutuksien kompleksisuus

Kompleksisuutta mitataan muistinkulutuksella ja ohjelmakoodirivien määrällä. Testiohjelman luomat bittikartat ovat etukäteen tiedossa, joten niiden muistin kulutus voidaan laskea teoreettisesti.

Taulukossa 1 on esitetty rinnakkaisjaetun-varjokarttatekniikassa käytettyjen varjostimien ja asiakasohjelman toteutuksien ohjelmarivien määrä.

Taulukko 1. *Rinnakkaisjaetun-varjokarttatekniikan ohjelmakoodirivien ja tiedostojen lukumäärät.*

Ohjelmointikieli	Tiedostojen lkm.	Ohjelmakoodirivien lkm.
GLSL	3	62
C#	1	189

Taulukosta 2 huomaamme, että suoralinjaisesti väännätetyn varjokarttatekniikassa ohjelmakoodirivien määrä on melkein puolet suurempi. Myös varjostimien lukumäärä on melkein nelinkertainen.

Taulukko 2. *Suoralinjaisesti väännätetyn varjokarttatekniikan ohjelmakoodirivien ja tiedostojen lukumäärät.*

Ohjelmointikieli	Tiedostojen lkm.	Ohjelmakoodirivien lkm.
GLSL	11	211
C#	1	213

5.1.1 Muistinkulutus

Alla olevissa taulukoissa kanavien (R, G, B, A, Syvyys) koot ovat esitetty bitteinä. Pikselin koko on kanavien kokojen yhteenlaskettu summa. Bittikartan leveys on L sarakkeessa ja korkeus K sarakkeessa. Lopullinen koko saadaan, kun kerrotaan L, K ja pikselin koko sarakkeiden arvot.

Taulukossa 3 on esitelty rinnakkaisjaetun-varjokarttatekniikan muistinkulutus, kun varjokarttojen resoluutioksi on asetettu 2048 * 2048. Taulukosta huomataan, että varjokarttojen muistinkulutus megatavuina on yhteensä 3 * 100663296 bittiä / 8 bittiä / 1048576 tavua = 36 megatavua.

Taulukko 3. *Rinnakkaisjaetun-varjokarttatekniikan muistinkulutus 2048 * 2048 koon varjokartoilla.*

Nimi	R (b)	G (b)	B (b)	A (b)	Syvyyskanava (b)	Pikselin koko (b)	L (px)	K (px)	Koko (b)
Varjokartta T_1	0	0	0	0	24	24	2048	2048	100663296
Varjokartta T_2	0	0	0	0	24	24	2048	2048	100663296
Varjokartta T_3	0	0	0	0	24	24	2048	2048	100663296

Taulukossa 4 on esitelty rinnakkaisjaetun-varjokarttatekniikan muistinkulutus, kun varjokarttojen resoluutioksi on asetettu 4096 * 4096. Taulukosta huomataan, että varjokarttojen muistinkulutus megatavuina on yhteensä 144 megatavua. Nyt muistinkulutus on neljä kertaa suurempi kuin 2048 * 2048 koon varjokartalla.

Taulukko 4. *Rinnakkaisjaetun-varjokarttatekniikan muistinkulutus 4096 * 4096 koon varjokartoilla.*

Nimi	R (b)	G (b)	B (b)	A (b)	Syvyyskanava (b)	Pikselin koko (b)	L (px)	K (px)	Koko (b)
Varjokartta T_1	0	0	0	0	24	24	4096	4096	402653184
Varjokartta T_2	0	0	0	0	24	24	4096	4096	402653184
Varjokartta T_3	0	0	0	0	24	24	4096	4096	402653184

Taulukosta 5 huomataan, että vaikka suoralinjaisesti väännätetty varjokarttatekniikka käyttää enemmän bittikarttoja on yhteenlaskettu muistinkulutus pienempi. Nyt yhteenlaskettu muistinkulutus on noin 21 megatavua.

Taulukko 5. *Suoralinjaisesti väännätetyn varjokarttatekniikan muistinkulutus 2048 * 2048 koon varjokartalla.*

Nimi	R (b)	G (b)	B (b)	A (b)	Syvyyskanava (b)	Pikselin koko (b)	L (px)	K (px)	Koko (b)
Värikartta	8	8	8	8	0	32	1280	720	29491200
Normaalikartta	16	0	0	0	0	16	1280	720	14745600
Syvyyskartta	0	0	0	0	24	24	1280	720	22118400
Tärkeyskartta	32	32	0	0	0	64	512	512	16777216
Supistettu tärk.	32	0	0	0	0	32	512	2	32768
Sumennettu tärk.	32	0	0	0	0	32	512	2	32768
Vääntymiskartta	32	0	0	0	0	32	512	2	32768
Varjokartta	0	0	0	0	24	24	2048	2048	100663296

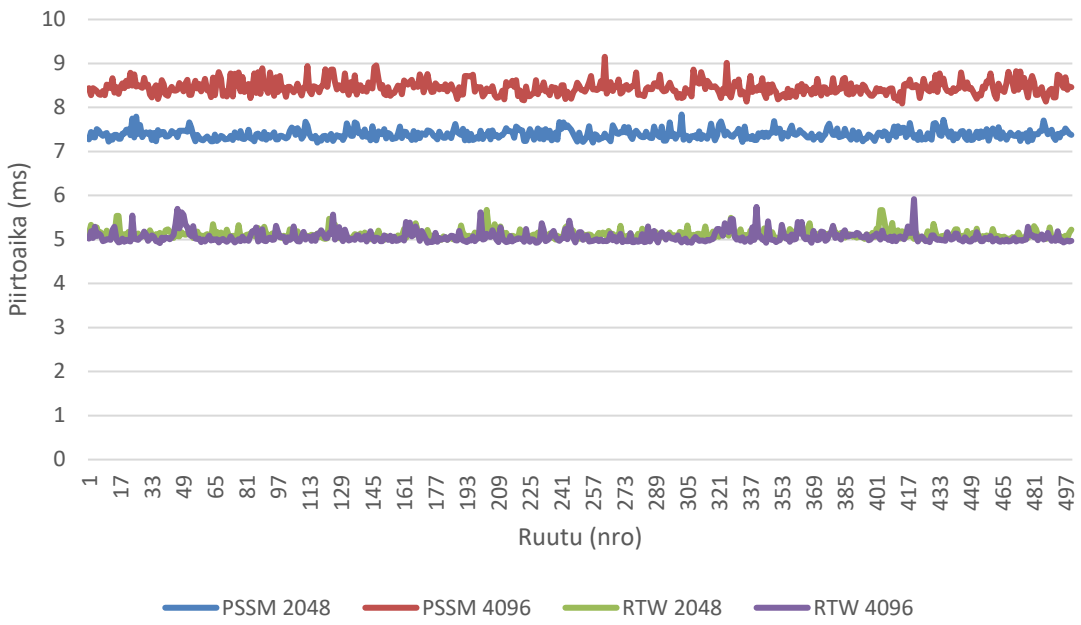
Kun varjokartan resoluutiota nostetaan 4096 * 4096 huomataan taulukosta 6, että muistinkulutus ei nouse niin paljon kuin rinnakkaisjaetun-varjokarttatekniikassa. Nyt suoralinjaisesti väännätetyn varjokarttatekniikan yhteenlaskettu muistinkulutus on noin 58 megatavua eli yli puolet vähemmän kuin rinnakkaisjaetulla varjokarttatekniikalla saman kokoisella varjokartalla.

Taulukko 6. *Suoralinjaisesti väännätetyn varjokarttatekniikan muistinkulutus 4096 * 4096 koon varjokartalla.*

Nimi	R (b)	G (b)	B (b)	A (b)	Syvyyskanava (b)	Pikselin koko (b)	L (px)	K (px)	Koko (b)
Värikartta	8	8	8	8	0	32	1280	720	29491200
Normaalikartta	16	0	0	0	0	16	1280	720	14745600
Syvyyskartta	0	0	0	0	24	24	1280	720	22118400
Tärkeyskartta	32	32	0	0	0	64	512	512	16777216
Supistettu tärk.	32	0	0	0	0	32	512	2	32768
Sumennettu tärk.	32	0	0	0	0	32	512	2	32768
Vääntymiskartta	32	0	0	0	0	32	512	2	32768
Varjokartta	0	0	0	0	24	24	4096	4096	402653184

5.2 Ruudun piirtoaika

Kuvassa 9 esitellään piirtoajat eri varjokarttatekniikoiden välillä. Siinä *PSSM* tarkoittaa rinnakkaisjaettua-varjokarttatekniikkaa ja *RTW* suoralinjaisesti väännätettyä varjokarttatekniikkaa. Numero vastaa käytettyä varjokartta resoluutiota. Kuvasta huomataan, että suoralinjaisesti väännätetyssä varjokartassa, piirtoaika ei muutu, vaikka varjokarttaresoluutiota suurennetaan. Kun taas rinnakkaisjaetussa varjokartassa piirtoaika nousee noin 14% verran.



Kuva 9. *Eri varjokarttatekniikoiden piirtoajat ruudun funktiona.*

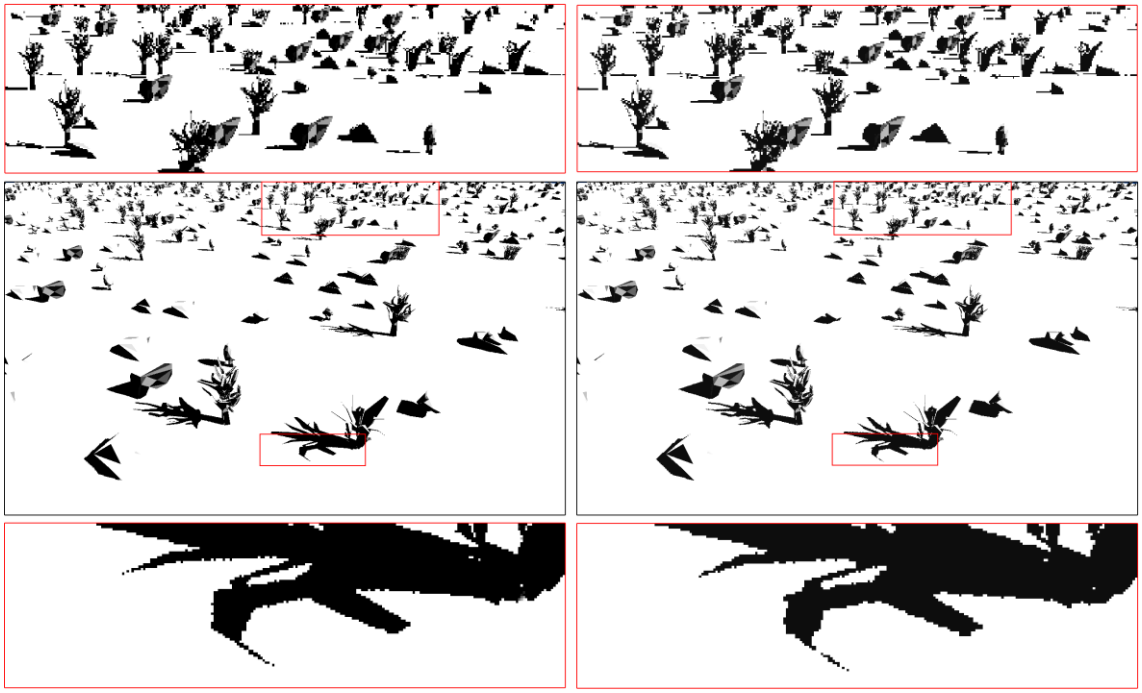
Taulukko 7 esittelee kuva 9 sisältämien piirtoaikojen keskiarvot, mediaanit ja minimit. Minimiaika on tärkein, koska se on optimaalisin aika, joka on saavutettu. Piirtoaikaan vaikuttaa myös testiohjelman ulkopuoliset ohjelmat, jotka vievät prosessorilta ja näytönohjaimelta resursseja.

Taulukko 7. *Piirtoaikojen keski- ja mediaaniarvot eri varjokarttatekniikoilla.*

Tekniikka	Resoluutio	Piirtoajan keskiarvo (ms)	Piirtoajan mediaani (ms)	Piirtoajan minimi (ms)
PSSM	2048 * 2048	7,39	7,38	7,20
PSSM	4096 * 4096	8,45	8,43	8,09
RTW	2048 * 2048	5,11	5,08	4,99
RTW	4096 * 4096	5,06	5,02	4,92

5.3 Visuaalinen ero

Kuvasta 10 huomataan, että kauimpana sijaitsevien geometristen objektien varjoissa on enemmän laskostumista suoralinjaisesti väännätetyllä varjokartalla, kun taas rinnakkaisjaettua-varjokarttaa käyttämällä varjojen laskostuminen on vähäisempi. Lähellä olevan geometrian varjot ovat laadultaan hyvin samanlaisia.



Kuva 10. *Suoralinjaisesti väännätetyllä (vasemmalla) ja rinnakkaisjaetulla (oikealla) varjokarttatekniikalla renderoidyt virtuaaliympäristöt.*

Visuaaliseen eroon vaikuttaa huomattavasti myös ikkunaportin resoluutio. Suuremmalla ikkunaresoluutiolla erot olisivat selvempiä, koska tällöin tehdään enemmän palojen näytteidenottoja rasteroidulle geometrialle. Kuvan 10 renderoinnissa ikkunaresoluutiona käytetettiin 1280 * 720 pikseliä.

6. YHTEENVETO

Tässä työssä käsiteltiin kaksi eri tapaa luoda varjot kolmiulotteiseen virtuaaliseen maailmaan varjokarttojen avulla. Tutkitut varjokarttatekniikat olivat rinnakkaisjaettu-varjokartta (lyhenne PSSM) ja suoralinjaisesti väännätetty varjokartta (lyhenne RTW). Tavoitteena oli tutkia niiden eroavaisuuksia suorituskyvyssä, kompleksisuudessa ja visuaalisuudessa. Tutkimisen avuksi toteutettiin testiohjelma, joka toteutti molemmat varjokarttatekniikat sekä renderoi virtuaalimaailman käyttäen niitä tekniikoita.

Testiohjelman avulla saaduista tuloksista huomataan, että suoralinjaisesti väännätetyn varjokarttatekniikan toteuttaminen vaatii huomattavasti enemmän työtä. Työssä toteutettu RTW-tekniikka käyttää 11 eri GLSL-varjostinta, kun taas rinnakkaisjaettu-varjokarttatekniikka käyttää vain kolmea GLSL-varjostinta. RTW-tekniikan suuri varjostimien lukumäärä johtuu sen käyttämän tärkeyskartan ja vääntämiskartan luonnin kiihdyttämisestä näytönohjaimella. PSSM-tekniikassa nojaudutaan enemmän tärkeiden geometristen objektien valintaan tietyn näköalueen sisällä, joka voidaan suorittaa helposti keskusprosessorilla C# ohjelmakoodissa.

Muistinkulutuksen eroissa huomataan, että vaikka RTW-tekniikka nojautuu useaan eri bittikarttaan, vie se lopulta vähemmän muistia kuin PSSM-tekniikka, vaikka varjokartan koko on sama. Tuloksista myös huomataan se, että RTW-tekniikka skaalautuu paremmin, kun varjokartan kokoa suurennetaan. Toisaalta RTW-tekniikan muistinkulutus on yhteydessä ikkunaresoluutioon, jolloin isolla ikkunaresoluutiolla voi RTW-tekniikka viedä enemmän muistia. Tuloksien muistinkulutuksen ero johtuu siitä syystä, että PSSM-tekniikka käyttää useaa samankokoista varjokarttaa, kun taas RTW-tekniikka pohjautuu yhteen varjokarttaan.

Varjokarttatekniikan tehokkuus mitattiin piirtoajan kestolla, eli kuinka kauan yhden ruudun renderointi kestää millisekunnin sadasosan tarkkuudella. Tuloksista huomataan, että RTW-tekniikka on nopein testissä käytetyistä varjokarttaresoluutioista riippumatta. Myöskään varjokartan koon suurentaminen ei vaikuttanut RTW-tekniikan suorituskykyyn. Tämä hyvin vahvasti viittaa siihen, että RTW-tekniikassa suurin määrä piirtoajasta kuluu tärkeyskartan ja vääntämiskartan luomisessa. PSSM-tekniikassa huomattiin, että varjokartan koon nosto aiheutti noin 15 prosentin nousun piirtoaikaan.

Renderoinnin lopputuloksen eroissa keskityttiin varjojen laskostumiseen lähellä ja kaukana kamerasta. Kuvaa 10 tarkastelemalla huomataan, että erot ovat hyvin pieniä lähellä, mutta kaukana olevat varjot ovat huomattavasti paremmin esillä PSSM-tekniikalla. Tämä ero tulisi korostumaan varsinkin isoilla testiohjelman resoluutiolla. Varjojen visuaalinen ero oletettavasti johtuu siitä, koska PSSM-tekniikassa käytettävissä olevaa varjokarttaresoluutiota on yhteensä enemmän. Eli kaukana olevien geometristen

objektien syvyystiedot ovat tarkemmin tallessa, kuin RTW-tekniikan yksittäisellä varjokartalla, johon kauimmaisten objektien geometrian syvyystiedot ovat ”tiivistetty” mukaan.

Tutkimuksen luotettavuuteen vaikuttavat muun muassa testiohjelmassa käytettyjen varjostimien tehokkuus eri varjotekniikoiden välillä. Varsinkin RTW-tekniikassa, jossa varjostimissa on paljon toiminnallisuutta. Myös erot virtuaalimaailman geometriassa ja virtuaalikameran sekä valonlähteen sijainti aiheuttavat eroja piirtoajoissa. Tämän työn puitteissa tehdyssä testissä ei voida varmasti taata kuin se, että RTW-tekniikka on nopeampi juuri tällä kokoonpanolla, virtuaaliympäristöllä ja toteutuksella. Tutkimuksen jatkokehityksen kannalta olisi hyvä testata myös useammalla eri kokoonpanolla, ikkunaresoluutioilla, virtuaaliympäristöillä ja eri toteutuksilla, esimerkiksi DirectX-grafiikkarajapinnalla, jolloin mahdollisia teho-eroja tulisi eriin niiden välillä.

Varjojen laskostumiseen vaikuttaa paljon myös virtuaalikameran ja valonlähteen sijainti, koska molemmat tekniikat koittavat optimoida käytettyä varjokarttaresoluutiota juuri niiden leikkaukselle. Eli tekniikoiden sisäiset muuttujat ovat riippuvaisia virtuaalikameran ja valonlähteen sijainnista, jolloin lopullinen renderoinnin tulos on aina hieman eri, kun valonlähteen tai kameran ominaisuudet muuttuvat.

Molemmat tekniikat sopivat hyvin varjojen visualisointiin. Tämän työn tulosten perusteella RTW-tekniikkaa voidaan suositella, kun varjokarttatekniikan parempi suorituskyky ja pienempi muistinkulutus ovat tärkeitä. Toisaalta PSSM-tekniikka on yksinkertaisempi toteuttaa ja se tarjoaa paremman varjojen laadun geometriaobjekteille, jotka sijaitsevat kauempina kamerasta.

LÄHTEET

[1] F. Zhang, H. Sun, L. Xu, L-K. Lun, Parallel-split Shadow Maps for Large-scale Virtual Environments, Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications, Hong Kong, China, 2006, ACM Press, pp. 311-318.

[2] P. Rosen, Rectilinear Texture Warping for Fast Adaptive Shadow Mapping, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, Costa Mesa, California, 2012, ACM Press, pp. 151-158.

[3] The OpenGL R Graphics System: A Specification, The Khronos Group Inc, June 2017, 784 p. Saatavissa (viitattu 22.9.2017): https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf

[4] OpenGL Overview, Khronos Group. Saatavissa: https://www.opengl.org/about/

[5] Khronos OpenGL® Registry, Khronos Group. Saatavissa: https://www.khronos.org/registry/OpenGL/index_gl.php/

[6] L. Williams, Casting curved shadows on curved surfaces, Proceedings of the 5th annual conference on Computer graphics and interactive techniques, 1978, ACM Press, pp. 270-274.

[7] Open Toolkit library, OpenTK. Saatavissa: https://github.com/opentk/opentk

[8] Visual Studio Community, Microsoft, ohjelma. Saatavissa: https://www.visualstudio.com/vs/community/

[9] RenderDoc, Baldur Karlsson, ohjelma. Saatavissa: https://renderdoc.org/

[10] Low-poly nature pack, Blend Swap, 3d-malli. Saatavissa: https://www.blendswap.com/blends/view/88462

[11] Wavefront .obj, tiedostoformaatti. Saatavissa (viitattu 26.11.2017): http://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf

[12] ObjLoader, Chris Jansson. Saatavissa: https://github.com/chrisjansson/ObjLoader

[13] Random, .NET Framework, Microsoft. Saatavissa (viitattu 26.11.2017): https://msdn.microsoft.com/en-us/library/system.random(v=vs.110).aspx

[14] Paul Rosen, Source Code and Sample Scenes, RTW: Rectilinear Texture Warping for Fast Adaptive Shadow Mapping, 2013, lähdekoodi. Saatavissa: http://www.cspaul.com/wordpress/projects-rtw/

[15] Stopwatch, .NET Framework, Microsoft. Saatavissa (viitattu 26.11.2017): https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch(v=vs.110).aspx

[16] cloc, ohjelma. Saatavissa: https://github.com/AlDanial/cloc

LIITE A: RINNAKKAISJAETUN-VARJOKARTTATEKNIIKAN C# OHJELMAKOODI

```
using OpenTK;
using System;
using System.Collections.Generic;
using OpenTK.Graphics.OpenGL4;

public class ParallelSplitShadowMapRenderer
{
// Varjokartta luokka avustus
class ShadowMap
{
    Framebuffer mFramebuffer;
    Texture2D mDepthTexture;
    public ShadowMap(int size) {
        // Varjokartta eli valon syvyyskartta 'Depth' kanava,
        // 24 bittiä varattu syvyyskanavalle
        mDepthTexture = new Texture2D(size, size,
            PixelInternalFormat.Depth24Stencil8,
            PixelFormat.DepthStencil, PixelType.UnsignedInt248);
        mDepthTexture.SetMinMagFilter(TextureMagFilter.Linear,
                                      TextureMinFilter.Linear);
        mDepthTexture.SetWrapMode(TextureWrapMode.ClampToBorder);
        float[] pBorderColor = new float[] { 1.0f, 1.0f, 1.0f, 1.0f };
        mDepthTexture.SetBorderColor(pBorderColor);

        mFramebuffer = new Framebuffer(size, size);
        mFramebuffer.AttachDepthStencilTexture(mDepthTexture.mHandle);
    }

    public void Activate() {
        mFramebuffer.Activate();
        GL.ColorMask(false, false, false, false);

        GL.ClearDepth(1.0f);
        GL.Clear(ClearBufferMask.DepthBufferBit);
    }

    public void Deactivate() {
        mFramebuffer.Deactivate();
        GL.ColorMask(true, true, true, true);
    }

    public void Bind(int i) {
        GL.ActiveTexture(TextureUnit.Texture0 + i);
        GL.BindTexture(TextureTarget.Texture2D, mDepthTexture.mHandle);
    }
}

const int ShadowMapSize = 4096;
const int NumSplits = 3;

ShaderProgram mSceneShader;
ShaderProgram mShadowShader;

private Camera mCamera = null;
```

```
private Light mLight = new Light();

float[] mSplitPositions = new float[NumSplits + 1];

ShadowMap[] mShadowMap = new ShadowMap[NumSplits];

public ParallelSplitShadowMapRenderer(Camera camera) {
    this.mCamera = camera;

    LoadShaders();

    for (int i = 0; i < NumSplits; ++i) {
        mShadowMap[i] = new ShadowMap(ShadowMapSize);
    }
}

public void LoadShaders() {
    mSceneShader = new ShaderProgram("shaders/pssm_scene.vert",
        "shaders/pssm_scene.frag", true);
    mSceneShader.UseProgram();
    mSceneShader.SetInt("shadowMap", new int[] { 0, 1, 2 });

    mShadowShader = new ShaderProgram("shaders/pssm_shadow.vert",
                                      null, false);
}

// Renderoidaan varjostajat valon näköalueella
private void RenderCasters(List<Scene.Entity> objects, Matrix4 viewProj) {
    GL.Disable(EnableCap.CullFace);

    // Otetaan 'pssm_shadow.vert' varjostin käyttöön
    mShadowShader.UseProgram();
    // set view projection
    mShadowShader.SetMatrix("viewProjection", ref viewProj);

    for (int i = 0; i < objects.Count; ++i) {

        // set object world matrix
        var transform = Matrix4.CreateTranslation(objects[i].position);
        mShadowShader.SetMatrix("worldTransform", ref transform);
        objects[i].Draw();
    }
}

// Renderoidaan kameran näköalue
private unsafe void RenderReceivers(List<Scene.Entity> objects,
                                    ref Matrix4 view,
                                    ref Matrix4 viewProj,
                                    Matrix4[] textureMatrix) {

    mSceneShader.UseProgram();
    GL.Enable(EnableCap.CullFace);
    GL.CullFace(CullFaceMode.Front);

    for (int i = 0; i < NumSplits; ++i) {
        mShadowMap[i].Bind(i);
    }

    mSceneShader.SetMatrixArray("textureMatrix", textureMatrix);
```

```
    mSceneShader.SetFloat("splitPlane", mSplitPositions, 1);

    mSceneShader.SetMatrix("viewProjection", ref viewProj);
    mSceneShader.SetMatrix("view", ref view);

    for (int i = 0; i < objects.Count; ++i) {
        // set object world matrix
        var transform = Matrix4.CreateTranslation(objects[i].position);
        mSceneShader.SetMatrix("worldTransform", ref transform);
        objects[i].Draw();
    }
}

private void UpdateSplitPositions() {
    // Yhdistetty logaritminen ja tasainen leikkaus:
    // CLi = n*(f/n)^(i/numsplits)
    // CUi = n + (f-n)*(i/numsplits)
    // Ci = CLi*0.5 + CUi*0.5
    //

    float near = mCamera.GetNear();
    float far = mCamera.GetFar();

    for (int i = 0; i < NumSplits; i++) {
        float IDM = i / (float)NumSplits;
        float Log = near * (float)Math.Pow(far / near, IDM);
        float Uniform = near + (far - near) * IDM;
        mSplitPositions[i] = Log * 0.5f + Uniform * 0.5f;
    }

    // Varmista että ääriarvot ovat korrektit
    mSplitPositions[0] = near;
    mSplitPositions[NumSplits] = far;
}

// Renderoidaan ympäristö PSSM-varjokarttatekniikalla
public void Render(Scene scene) {

    // Etsitään kaikki kameran näköalueella olevat objektit
    var receivers = mCamera.FindReceivers(scene);
    // Haetaan etä- ja lähitasot
    mCamera.AdjustNearAndFarPlanes(receivers);

    // Päivitetään kameran osa-alueiden sijainnit
    UpdateSplitPositions();

    Matrix4[] lightSplitViewTextureMatrix = new Matrix4[NumSplits];

    int[] viewport = new int[4];
    GL.GetInteger(GetPName.Viewport, viewport);

    GL.Viewport(0, 0, ShadowMapSize, ShadowMapSize);
    // Jokaiselle osa-alueelle renderoidaan oma varjokartta
    for (int i = 0; i < NumSplits; i++) {
        // Luo osa näköalueelle (katkaistu pyramidi)
        var splitFrustum = mCamera.CalculateFrustum(mSplitPositions[i],
                                                    mSplitPositions[i + 1]);
```

```
            // Etsi varjostajat
            List<Scene.Entity> casters = mLight.FindCasters(ref splitFrustum,
                                                            scene);

            // Laske leikkausmatriisi ja valon lopullinen projektiomatriisi
            var cropMatrix = CalculateCropMatrix(mLight, casters, receivers,
                                                 splitFrustum);

            var splitViewProjection = mLight.View * mLight.Projection *
                                      cropMatrix;

            lightSplitViewTextureMatrix[i] = splitViewProjection *
                                             GetTexScaleBiasMatrix();

            // Renderoi varjostajat varjokartalle
            mShadowMap[i].Activate();
            RenderCasters(casters, splitViewProjection);
            mShadowMap[i].Deactivate();
        }
        GL.Viewport(0, 0, viewport[2], viewport[3]);

        // Renderoidaan ympäristö kameran näkökulmasta
        Matrix4 view = mCamera.GetView();
        Matrix4 viewProj = mCamera.GetViewProjection();
        RenderReceivers(receivers, ref view, ref viewProj,
                        lightSplitViewTextureMatrix);
    }

    private Matrix4 CalculateCropMatrix(Light mLight,
                                        List<Scene.Entity> casters,
                                        List<Scene.Entity> receivers,
                                        Frustum splitFrustum) {

        Matrix4 viewProj = mLight.View * mLight.Projection;

        BoundingBox rBB = new BoundingBox();
        BoundingBox splitBB = new BoundingBox();
        BoundingBox cBB = new BoundingBox();

        // Etsi varjostajien rajat valon projektioavaruudessa
        for (int i = 0; i < casters.Count; i++)
            cBB.Union(CreateClipSpaceAABB(casters[i].aabb, viewProj));

        // Etsi objektien rajat valon projektioavaruudessa
        for (int i = 0; i < receivers.Count; i++) {
            rBB.Union(CreateClipSpaceAABB(receivers[i].aabb, viewProj));
        }

        // Hae kameran näköalueen rajat valon projektioavaruudessa
        splitBB = CreateClipSpaceAABB(splitFrustum.aabb, viewProj);

        // Yhdistetään rajat
        BoundingBox cropBB = new BoundingBox();
        cropBB.Min.X = Math.Max(Math.Max(cBB.Min.X, rBB.Min.X), splitBB.Min.X);
        cropBB.Max.X = Math.Min(Math.Min(cBB.Max.X, rBB.Max.X), splitBB.Max.X);
        cropBB.Min.Y = Math.Max(Math.Max(cBB.Min.Y, rBB.Min.Y), splitBB.Min.Y);
        cropBB.Max.Y = Math.Min(Math.Min(cBB.Max.Y, rBB.Max.Y), splitBB.Max.Y);
        cropBB.Min.Z = cBB.Min.Z;
        cropBB.Max.Z = Math.Min(rBB.Max.Z, splitBB.Max.Z);
```

```
    return BuildCropMatrix(cropBB.Min, cropBB.Max);
}

private Matrix4 BuildCropMatrix(Vector3 min, Vector3 max) {
    float fScaleX, fScaleY, fScaleZ;
    float fOffsetX, fOffsetY, fOffsetZ;

    fScaleX = 2.0f / (max.X - min.X);
    fScaleY = 2.0f / (max.Y - min.Y);

    fOffsetX = -0.5f * (max.X + min.X) * fScaleX;
    fOffsetY = -0.5f * (max.Y + min.Y) * fScaleY;

    fScaleZ = 1.0f / (max.Z - min.Z);
    fOffsetZ = -min.Z * fScaleZ;

    var result = new Matrix4(fScaleX, 0.0f, 0.0f, 0.0f,
                             0.0f, fScaleY, 0.0f, 0.0f,
                             0.0f, 0.0f, fScaleZ, 0.0f,
                         fOffsetX, fOffsetY, fOffsetZ, 1.0f);
    return result;
}

private BoundingBox CreateClipSpaceAABB(BoundingBox bb, Matrix4 viewProj) {
    Vector4[] transformed = new Vector4[8];
    for (int i = 0; i < 8; i++) {
        // Muunna piste projektioavaruuteen
        var v = new Vector4(bb.Points[i], 1);
        transformed[i] = Vector4.Transform(v, viewProj);

        transformed[i].X /= transformed[i].W;
        transformed[i].Y /= transformed[i].W;
        transformed[i].Z /= transformed[i].W;
    }
    BoundingBox result = new BoundingBox();
    result.Set(transformed);

    return result;
}

// Muuntaa pisteen akselit [0,1] koordinaatistosta, [-1, 1] koordinaatistoon
Matrix4 GetTexScaleBiasMatrix() {
    return new Matrix4(0.5f, 0.0f, 0.0f, 0.0f,
                       0.0f, 0.5f, 0.0f, 0.0f,
                       0.0f, 0.0f, 0.5f, 0.0f,
                       0.5f, 0.5f, 0.5f, 1.0f);
}
}
```

LIITE B: RINNAKKAISJAETUN-VARJOKARTTATEKNIIKASSA KÄYTETYT GLSL VARJOSTIMET

Kärkivarjostin, jota käytetään varjokartan renderoimiseen, *pssm_shadow.vert*:

```
#version 330

// vakiolohko, kameran projektiomatriisi
uniform mat4 viewProjection;
// objektin transformaatiomatriisi
uniform mat4 worldTransform;

// kärjen tiedot
layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

void main(void) {
	// muunnetaan kärjen sijainti kameran projektion koordinaatistoon
	vec4 worldPosition = worldTransform * vec4(inPosition, 1);
	gl_Position = viewProjection * worldPosition;
}
```

Kärkivarjostin, jota käytetään virtuaaliympäristön renderoimiseen kameran näkökulmasta, *pssm_scene.vert*:

```
#version 330

const int numSplits = 3;
const vec3 lightDirection = vec3(1, -1, 1);

// näköpisteen transformaatiomatriisi
uniform mat4 view;
// näkö ja projektio matriisin yhdistys
uniform mat4 viewProjection;
// objektin transformaatiomatriisi
uniform mat4 worldTransform;
// valon näköalueiden transformaatiomatriisit
uniform mat4 textureMatrix[numSplits];

// kärjen tiedot
layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

// tiedot ulos palavarjostimeen
out vec4 texCoord[numSplits + 1];
out float lighting;

void main(void) {
	// muunnetaan kärjen sijainti maailman koordinaattiavaruuteen
	vec4 worldPosition = worldTransform * vec4(inPosition, 1);
	// muunnetaan se edelleen kameran projektion koordinaattiavaruuteen
	gl_Position = viewProjection * worldPosition;

	// lasketaan yksinkertainen valaistus valon ja kärjen normaalin avulla
	vec3 normalWorld = normalize(mat3(worldTransform) * inNormal));
	lighting = clamp(dot(-lightDirection, normalWorld, 0, 1);
```

```
	// tallennetaan kärjen sijainti kameran koordinaattiavaruudessa
	texCoord[0] = view * worldPosition;

	// tallennetaan kärjen sijainti valon näköalueitten
	// koordinaattiavaruuksissa
	for(int i = 0; i < numSplits; i++) {
		texCoord[i + 1] = textureMatrix[i] * worldPosition;
	}
}
```

Palavarjostin, jota käytetään virtuaaliympäristön renderoimiseen kameran näkökulmasta, *pssm_scene.frag*:

```
#version 330

const int numSplits = 3;
// siirtää syvyyseroa,
// auttaa vähentämään liukukujen tarkkuuden aiheuttamaa virhettä
float shadow_bias = 0.00001f;

const vec3 ambient = vec3(0.05,0.05,0.05);
const vec3 maxClamp = vec3(1,1,1);
const vec3 minClamp = vec3(0.05,0.05,0.05);

// Näköalueen leikkauksien sijainnit
uniform float splitPlane[numSplits];
// Varjokartta per leikkaus
uniform sampler2D shadowMap[numSplits];

// palan tiedot
in float lighting;
in vec4 texCoord[numSplits + 1];

float GetShadowIntensity(sampler2D shadowMap, vec4 shadow_coord) {
	// hae lähemmän geometrian etäisyys
	// tämän palan kohdalla valon näkökulmasta
	float tex_d = texture(shadowMap, shadow_coord.xy).x;
	// etäisyyksien ero
	float z_diff = shadow_coord.z - tex_d;
	return 1 - clamp( (z_diff - shadow_bias) / shadow_bias, 0, 1);
}

void main() {
	vec3 lightColor = vec3(lighting, lighting, lighting);
	float lightingFactor = 1.0;
	float distance = texCoord[0].z; // palan etäisyys

	// valitaan oikea varjokartta etäisyyden
	// ja näköalueen leikkaussyvyyden perusteella
	if(distance < splitPlane[0]) {
		lightingFactor = GetShadowIntensity(shadowMap[0], texCoord[1]);
	}
	else if(distance < splitPlane[1]){
		lightingFactor = GetShadowIntensity(shadowMap[1], texCoord[2]);
	}
	else {
		lightingFactor = GetShadowIntensity(shadowMap[2], texCoord[3]);
```

```
        }

        vec3 finalColor = ambient + lightColor * lightingFactor;
        gl_FragColor.xyz = clamp(finalColor, minClamp, maxClamp);
        gl_FragColor.a = 1.0;
}
```

LIITE C: SUORALINJAISESTI VÄÄNNÄTETYSSÄ VARJOKARTTATEKNIIKASSA KÄYTETTY C# OHJELMAKOODI

```
using System.Collections.Generic;
using OpenTK.Graphics.OpenGL4;
using OpenTK;

public class RTWShadowMapRenderer
{
ShaderProgram mColorDepthNormalShader;

Texture2D mColorTexture; // unit 0
Texture2D mCameraDepthMap; // unit 1
Texture2D mDotNormalTexture; // unit 2
Framebuffer mColorDepthDNormalFBO;

ShaderProgram mImportanceShader;

Texture2D mImportanceTexture; // unit 3
Framebuffer mImportanceFramebuffer;

ShaderProgram mCompactMapShader;
ShaderProgram mBlurMapShader;
ShaderProgram mBuildWarpMapShader;

Texture2D[] mWarpTexture = new Texture2D[3];
Framebuffer[] mWarpFramebuffer = new Framebuffer[3];

ShaderProgram mShadowMapShader;
Texture2D mShadowMapTexture;
Framebuffer mShadowMapFBO;

ShaderProgram mCompositeShader;

int mClientWidth;
int mClientHeight;

Camera mCamera;
Light mLight = new Light();

const int ImportanceSize = 512; // tärkeyskartan koko / leveys
const int SmSize = 4096; // varjokartan sivun koko

public RTWShadowMapRenderer(Camera camera, int width, int height) {
    mCamera = camera;
    // piirtoikkunan leveys ja korkeys
    mClientWidth = width;
    mClientHeight = height;
    LoadShaders();
    CreateFramebuffers();
}


// varjostimien lataus
private void LoadShaders() {
    mColorDepthNormalShader =
            new ShaderProgram("shaders/rtw_depth_image.vert",
```

```
                                "shaders/rtw_depth_image.frag", true);
        mImportanceShader = new ShaderProgram("shaders/rtw_importance.vert",
                                              "shaders/rtw_importance.frag",
                                               false);
        mCompactMapShader = new ShaderProgram("shaders/rtw_warp.vert",
                                              "shaders/rtw_compact.frag",
                                              false);

        mBlurMapShader = new ShaderProgram("shaders/rtw_warp.vert",
                                           "shaders/rtw_blur.frag", false);
        mBuildWarpMapShader = new ShaderProgram("shaders/rtw_warp.vert",
                                                "shaders/rtw_buildwarp.frag",
                                                false);

        mShadowMapShader = new ShaderProgram("shaders/rtw_shadow.vert", null,
                                             false);
        mCompositeShader = new ShaderProgram("shaders/rtw_scene.vert",
                                              "shaders/rtw_scene.frag", false);
    }

        // Luodaan tekstuurit ja ruutupuskurit niille
    private void CreateFramebuffers() {
        // Värikartta RGBA, 8 bittiä per kanava
        mColorTexture = new Texture2D(mClientWidth, mClientHeight,
                                      PixelInternalFormat.Rgba,
                                      PixelFormat.Rgba, PixelType.UnsignedByte);
        mColorTexture.SetMinMagFilter(TextureMagFilter.Nearest,
                                      TextureMinFilter.Nearest);
        mColorTexture.Bind(0);

        // Kameran syvyyskartta 'Depth' kanava, 24 bittiä varattu syvyyskanavalle
        mCameraDepthMap = new Texture2D(mClientWidth, mClientHeight,
                                        PixelInternalFormat.Depth24Stencil8,
                                        PixelFormat.DepthStencil,
                                        PixelType.UnsignedInt248);
        mCameraDepthMap.SetMinMagFilter(TextureMagFilter.Linear,
                                        TextureMinFilter.Linear);
        mCameraDepthMap.SetWrapMode(TextureWrapMode.ClampToBorder);
        float[] pBorderColor = new float[] { 1.0f, 1.0f, 1.0f, 1.0f };
        mCameraDepthMap.SetBorderColor(pBorderColor);
        mCameraDepthMap.Bind(1);

        // Kameran normaalikartta yksikanava, 16 bittinen liukuluku
        mDotNormalTexture = new Texture2D(mClientWidth, mClientHeight,
                                          PixelInternalFormat.R16f,
                                          PixelFormat.Red, PixelType.Float);
        mDotNormalTexture.SetMinMagFilter(TextureMagFilter.Nearest,
                                          TextureMinFilter.Nearest);
        mDotNormalTexture.Bind(2);

        mColorDepthDNormalFBO = new Framebuffer(mClientWidth, mClientHeight);
        mColorDepthDNormalFBO.AddColorTexture(mColorTexture.mHandle, 0);
        mColorDepthDNormalFBO.AddColorTexture(mDotNormalTexture.mHandle, 1);
        mColorDepthDNormalFBO.AddDepthStencilTexture(mCameraDepthMap.mHandle);

        // Tärkeykartta kaksikanavainen, 32 bittinen liukuluku per kanava
        mImportanceTexture = new Texture2D(ImportanceSize, ImportanceSize,
                                           PixelInternalFormat.Rg32f,
                                           PixelFormat.Rg, PixelType.Float);
```

```
    mImportanceTexture.SetMinMagFilter(TextureMagFilter.Nearest,
                                       TextureMinFilter.Nearest);
    mImportanceTexture.Bind(3);
    mImportanceFramebuffer = new Framebuffer(ImportanceSize, ImportanceSize);
    mImportanceFramebuffer.AddColorTexture(mImportanceTexture.mHandle, 0);

    for (int i = 0; i < 3; ++i) {
        // 3x vääntymiskartta yksikanavainen, 32 bittinen liukuluku
        mWarpTexture[i] = new Texture2D(ImportanceSize, 2,
                                        PixelInternalFormat.R32f,
                                        PixelFormat.Red,
                                        PixelType.Float);

        if (i < 2) {
            mWarpTexture[i].SetMinMagFilter(TextureMagFilter.Nearest,
                                            TextureMinFilter.Nearest);
        } else {
            mWarpTexture[i].SetMinMagFilter(TextureMagFilter.Linear,
                                            TextureMinFilter.Linear);
            mWarpTexture[i].SetWrapMode(TextureWrapMode.ClampToEdge);
        }
        mWarpTexture[i].Bind(4 + i);

        mWarpFramebuffer[i] = new Framebuffer(ImportanceSize, 2);
        mWarpFramebuffer[i].AddColorTexture(mWarpTexture[i].mHandle, 0);
    }

    // Varjokartta eli valon syvyyskartta 'Depth' kanava,
    // 24 bittiä varattu syvyyskanavalle
    mShadowMapTexture = new Texture2D(SmSize, SmSize,
                                      PixelInternalFormat.Depth24Stencil8,
                                      PixelFormat.DepthStencil,
                                      PixelType.UnsignedInt248);
    mShadowMapTexture.SetMinMagFilter(TextureMagFilter.Linear,
                                      TextureMinFilter.Linear);
    mShadowMapTexture.SetWrapMode(TextureWrapMode.ClampToBorder);
    mShadowMapTexture.SetBorderColor(pBorderColor);
    mShadowMapTexture.Bind(7);

    mShadowMapFBO = new Framebuffer(SmSize, SmSize);
    mShadowMapFBO.AddDepthStencilTexture(mShadowMapTexture.mHandle);
}

// Luodaan väri, normaali ja syvyyskartat
private void GenerateSurface(List<Scene.Entity> receivers,
                             ref Matrix4 view,
                             ref Matrix4 viewProj) {
    // Otetaan 'rtw_depth_image.frag/.vert' varjostimet käyttöön
    mColorDepthNormalShader.UseProgram();
    mColorDepthDNormalFBO.Activate();
    // Aktivoidaan kaksi ruutupuskuria
    // (ens. väri/syvyys, toinen normaalille)
    GL.DrawBuffers(2, new DrawBuffersEnum[] {
        DrawBuffersEnum.ColorAttachment0,
        DrawBuffersEnum.ColorAttachment1
    });
    GL.Clear(ClearBufferMask.ColorBufferBit |
             ClearBufferMask.DepthBufferBit);
```

```
    // Asetetaan varjostimen vakiolohko tiedot
    mColorDepthNormalShader.SetMatrix("viewProjection", ref viewProj);
    mColorDepthNormalShader.SetMatrix("view", ref view);

    // Piirretään objektit
    for (int i = 0; i < receivers.Count; ++i) {
        // Asetetaan objektin transformaatio
        // matriisi varjostimen vakiolohkoon
        var transform = Matrix4.CreateTranslation(receivers[i].position);
        mColorDepthNormalShader.SetMatrix("worldTransform", ref transform);

        receivers[i].Draw();
    }
    mColorDepthDNormalFBO.Deactivate();
    GL.DrawBuffers(1, new DrawBuffersEnum[] {
        DrawBuffersEnum.ColorAttachment0
    });
}

// Tärkeyskartan luominen
private void GenerateImportance(ref Scene.Entity rtwPoints) {
    // Otetaan 'rtw_importance.frag/.vert' varjostimet käyttöön
    mImportanceShader.UseProgram();
    // Liitetään syvyys ja normaalikartta liukuhihnalle
    mCameraDepthMap.Bind(1);
    mDotNormalTexture.Bind(2);
    mImportanceShader.SetInt("depthMap", 1);
    mImportanceShader.SetInt("dotNormalMap", 2);

    mImportanceFramebuffer.Activate();
    GL.ClearColor(0, 0, 0, 0);
    GL.Clear(ClearBufferMask.ColorBufferBit);

    // Muutosmatriisi kameran projektiokoordinaatistosta
    // valon projektiokoordinaatistoon
    var cameraViewToLight = mCamera.GetViewProjection().Inverted() *
                            (mLight.View * mLight.Projection);
    mImportanceShader.SetMatrix("cameraToLight", ref cameraViewToLight);

    // Renderoidään pistejoukko
    rtwPoints.Draw(PrimitiveType.Points);
    mImportanceFramebuffer.Deactivate();
}

// Vääntymiskartan luominen
private void GenerateWarpMap(ref Scene.Entity fsq) {
    // Otetaan 'rtw_compact.frag/rtw_warp.vert' varjostimet käyttöön
    mWarpFramebuffer[0].Activate();
    GL.Clear(ClearBufferMask.ColorBufferBit);
    mCompactMapShader.UseProgram();
    // Liitetään tärkeykartta liukuhihnalle
    mImportanceTexture.Bind(3);
    mCompactMapShader.SetInt("importanceMap", 3);
    fsq.Draw(PrimitiveType.TriangleStrip);
    mWarpFramebuffer[0].Deactivate();

    // Otetaan 'rtw_blur.frag/rtw_warp.vert' varjostimet käyttöön
    mWarpFramebuffer[1].Activate();
    GL.Clear(ClearBufferMask.ColorBufferBit);
```

```
    mBlurMapShader.UseProgram();
    // Liitetään edellisen renderoinnin tulos,
    // eli supistettu tärkeyskartta liukuhihnalle
    mWarpTexture[0].Bind(4);
    mBlurMapShader.SetInt("compactMap", 4);
    fsq.Draw(PrimitiveType.TriangleStrip);
    mWarpFramebuffer[1].Deactivate();

    // Otetaan 'rtw_buildwarp.frag/rtw_warp.vert' varjostimet käyttöön
    mWarpFramebuffer[2].Activate();
    GL.Clear(ClearBufferMask.ColorBufferBit);
    mBuildWarpMapShader.UseProgram();
    // Liitetään edellisen renderoinnin tulos, eli supistettu
    // ja sumennettu tärkeyskartta liukuhihnalle
    mWarpTexture[1].Bind(5);
    mBuildWarpMapShader.SetInt("blurMap", 5);
    fsq.Draw(PrimitiveType.TriangleStrip);
    mWarpFramebuffer[2].Deactivate();
}

// Luodaan varjokartta
private void GenerateShadowMap(List<Scene.Entity> casters) {
    mShadowMapShader.UseProgram();
    // Liitetään vääntymiskartta(t) liukuhihnalle
    mWarpTexture[2].Bind(6);
    mShadowMapShader.SetInt("warpMap", 6);

    Matrix4 lightViewProj = mLight.View * mLight.Projection;
    mShadowMapShader.SetMatrix("viewProjection", ref lightViewProj);

    mShadowMapFBO.Activate();
    GL.Clear(ClearBufferMask.DepthBufferBit);

    // Piirretään varjostajat
    for (int i = 0; i < casters.Count; ++i) {
        // set object world matrix
        var transform = Matrix4.CreateTranslation(casters[i].position);
        mShadowMapShader.SetMatrix("worldTransform", ref transform);

        casters[i].Draw();
    }
    mShadowMapFBO.Deactivate();
}

// Asetetaan varjot jo renderoidun värikuvan päälle
private void RenderComposite(ref Scene.Entity fsq) {

    mCompositeShader.UseProgram();
    // Liitetään väri-, varjo-, syvyys- ja vääntymiskartat liukuhihnalle
    mShadowMapTexture.Bind(7);
    mCameraDepthMap.Bind(1);
    mColorTexture.Bind(0);
    mWarpTexture[2].Bind(6);
    mCompositeShader.SetInt("shadowMap", 7);
    mCompositeShader.SetInt("depthMap", 1);
    mCompositeShader.SetInt("colorMap", 0);
    mCompositeShader.SetInt("warpMap", 6);

    // Muutosmatriisi kameran projektiokoordinaatistosta
```

```
    // valon projektiokoordinaatistoon
    var cameraViewToLight = mCamera.GetViewProjection().Inverted() *
                            (mLight.View * mLight.Projection);
    mCompositeShader.SetMatrix("cameraToLight", ref cameraViewToLight);
    fsq.Draw(PrimitiveType.TriangleStrip);

}

// Renderoidaan ympäristö RTW-varjokarttatekniikkaa käyttäen
public void Render(Scene scene) {
    // Haetaan objektit jotka on kameran näköalueella
    var receivers = mCamera.FindReceivers(scene);
    // Haetaan etä- ja lähitasot
    mCamera.AdjustNearAndFarPlanes(receivers);

    Matrix4 view = mCamera.GetView();
    Matrix4 viewProj = mCamera.GetViewProjection();
    // Piirretään ympäristö kameran kulmasta,
    // tallennetaan väri, syvyys ja normaalit
    GL.Enable(EnableCap.CullFace);
    GL.CullFace(CullFaceMode.Front);
    GenerateSurface(receivers, ref view, ref viewProj);
    GL.Disable(EnableCap.CullFace);
    GL.Disable(EnableCap.DepthTest);

    // Luodaan tärkeyskartta ja vääntymiskartta
    GenerateImportance(ref scene.RTWImportancePoints);
    GenerateWarpMap(ref scene.FullScreenQuad);
    GL.Enable(EnableCap.DepthTest);

    // Haetaan objektit jotka toimivat varjostajina
    Frustum cameraFrustum = mCamera.CalculateFrustum(mCamera.GetNear(),
                                                     mCamera.GetFar());
    var casters = mLight.FindCasters(ref cameraFrustum, scene);
    // Renderoidaan varjokartta
    GenerateShadowMap(casters);

    // Yhdistetään varjokartta sekä värikartta
    GL.Viewport(0, 0, mClientWidth, mClientHeight);
    RenderComposite(ref scene.FullScreenQuad);
}
}
```

LIITE D: SUORALINJAISESTI VÄÄNNÄTETYSSÄ VARJOKARTTATEKNIIKASSA KÄYTETYT GLSL VARJOSTIMET

Kärkivarjostin, jota käytetään väri-, normaali- ja syvyyskartan renderoimiseen kameran näkökulmasta, *rtw_depth_image.vert*:

```
#version 330

const vec3 lightDirection = vec3(1, -1, 1);

// vakiolohko
uniform mat4 view;
uniform mat4 viewProjection;
uniform mat4 worldTransform;

// kärjen tiedot
layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

// tiedot ulos palavarjostimelle
out float lighting;
out float cameraView_dot_normal;

void main(void) {
	// muunna kärjen sijainti kameran projektio koordinaattiavaruuteen
	vec4 worldPosition = worldTransform * vec4(inPosition, 1);
	gl_Position = viewProjection * worldPosition;

	// laske yksinkertainen kärjen valaistus normaalia
	// ja valon suuntaa käyttäen
	vec4 normalWorld = normalize(worldTransform * vec4(inNormal,0));
	lighting = clamp(dot(-lightDirection, normalWorld.xyz), 0, 1);

	// laske kärjen normaali kameran koordinaattiavaruudessa
	vec3 nm = (view * normalWorld).xyz;
	cameraView_dot_normal = normalize(nm).z;
}
```

Palavarjostin, jota käytetään väri-, normaali- ja syvyyskartan renderoimiseen kameran näkökulmasta, *rtw_depth_image.frag*:

```
#version 330
// vakiolohko
uniform mat4 view;
uniform mat4 viewProjection;
uniform mat4 worldTransform;
in float lighting;
in float cameraView_dot_normal;

layout (location = 0) out vec4 color;
layout (location = 1) out float dotN;
void main(void) {
	// tallenna väri ja normaali tiedot
	color = vec4(lighting, lighting, lighting, 1.0f);
	dotN = cameraView_dot_normal;
}
```

Kärkivarjostin, jota käytetään tärkeyskartan renderoimisessa, *rtw_importance.vert*:

```
#version 330

uniform mat4 cameraToLight;
uniform sampler2D depthMap;
uniform sampler2D dotNormalMap;

// kärjen tiedot
layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

// pisteen syvyys ja normaali kamerankoordinaatti avaruudessa
out float oDpt;
out float oNorm;

void main(void) {
      // haetaan syvyys ja normaali jo renderoiduista bittikartoista
      float  dpt = texture(depthMap, inPosition.xy).x;
      float  n   = texture(dotNormalMap, inPosition.xy).x;

      // muunna kameran projektiokoordinaatti, valon projektiokoordinaattiin
      vec4 pos = vec4(inPosition.xy, dpt, 1) * 2.0f - 1.0f;
      gl_Position = cameraToLight * pos;

      oDpt = (dpt >= 1.0f) ? 2.0f : dpt;
      oNorm = n;
}
```

Palavarjostin, jota käytetään tärkeyskartan renderoimisessa, *rtw_importance.frag*:

```
#version 330

const float direction_bonus = 2.0f;

// pisteen syvyys ja normaali kamerankoordinaatti avaruudessa
in float oDpt;
in float oNorm;

layout (location = 0) out vec2 outColor;

// etäisyyden tärkeysfunktio
float importanceIZ(float dpt){
    return clamp((1.0f - dpt), 0.0001f, 1.0f);
}

// normaalin tärkeysfunktio
float importanceND(float n) {
    return 1.0f + direction_bonus * clamp(n, 0, 1);
}

void main(void) {
    outColor.r = 1.0f * importanceIZ(oDpt) * importanceND(oNorm);
    outColor.g = (oDpt >= 1.0f) ? 2.0f
                  : (1.0f - clamp(outColor.r / 100.0f, 0, 1));
}
```

Kärkivarjostin, jota käytetään vääntymiskarttaan liittyvissä operaatiossa (supistus, sumennus ja luonti), *rtw_warp.vert*:

```
#version 330

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

void main(void) {
    gl_Position = vec4(inPosition, 1);
}
```

Palavarjostin, joka supistaa tärkeyskartan, *rtw_compact.frag*:

```
#version 330

int width = 512;
// tärkeyskartta
uniform sampler2D importanceMap;

// laskettu painoarvo (ulostulo) palalle
out float outWeigth;

void main(void) {
    vec2 uv0 = vec2(gl_FragCoord.x,         0) / width;
    vec2 uv1 = vec2(gl_FragCoord.x, width - 1) / width;

    // käännä, kun kirjoitetaan alempaa bittikartta riviä
    if (gl_FragCoord.y > 0.5f) {
        uv0.xy = uv0.yx;
        uv1.xy = uv1.yx;
    }

    float wgt = -1;
    for (int v = 0; v < width; v++) {
        float  t  = float(v) / (width-1);
        vec2 tc = mix(uv0, uv1, t);
        float  cw = texture(importanceMap, tc).x;
        wgt = max(wgt, cw);
    }

    outWeigth = wgt;
}
```

Palavarjostin, joka sumentaa supistetun tärkeyskartan, *rtw_blur.frag*:

```
#version 330

int width = 512;
float blur_factor = 0.75;
int blur_width = 10;

uniform sampler2D compactMap;

out float oColor;
```

```
void main(void) {

    vec2 uv = vec2( gl_FragCoord.x, gl_FragCoord.y * width / 2 );
    float my_w = texture( compactMap, uv / width ).x;
    float c_factor = blur_factor;

    for(int u = 1; u < blur_width; u++){
        float w0 = texture( compactMap, (uv-vec2(u,0)) / width ).x;
        float w1 = texture( compactMap, (uv+vec2(u,0)) / width ).x;
        float cw = max( w0, w1 ) * c_factor;
        if(cw >= 0){
            my_w = max(my_w, cw);
        }
        c_factor *= blur_factor;
    }

    oColor = my_w;
}
```

Palavarjostin, joka luo vääntymiskartan sumennetusta ja supistetusta tärkeyskartasta, *rtw_buildwarp.frag*:

```
#version 330

int width = 512;
uniform sampler2D blurMap;
out float oColor;

ivec2 GetBoundingRegion(float v) {
    int first_u = width, last_u = 0;
    for (int u = 0; u < width; u++){
        float cw = texture(blurMap, vec2(u, v) / width).x;
        if (cw > 0){
            first_u = min(first_u, u);
            last_u  = max(last_u,  u);
        }
    }
    if (last_u < first_u) { return ivec2(0, width); }
    return ivec2(first_u, last_u + 1);
}

float GetWeight(int u0, int u1, float v) {
    float ret = 0;
    for (int u = u0; u < u1; u++){
        float cw = texture(blurMap, vec2(u, v) / width).x;
        ret += max(cw, 0.001f);
    }
    return ret;
}

void main(void) {
    int my_u = int(gl_FragCoord.x);
    ivec2 range = GetBoundingRegion(gl_FragCoord.y * width / 2);

    float st_loc = mix(-1.0f, 1.0f, floor(gl_FragCoord.x) / width);
    float my_loc = st_loc;
```

```
    if (my_u < range.x) {
        my_loc = -1.05f;
    }
    else if (my_u >= range.y) {
        my_loc = 1.05f;
    }
    else {
        float w0 = GetWeight(range.x, my_u, gl_FragCoord.y * width / 2.0f);
        float w1 = GetWeight(my_u, range.y, gl_FragCoord.y * width / 2.0f);

        // lineaarinen interpolaatio
        my_loc = mix(-1.0f, 1.0f, w0/(w0+w1));
    }

    oColor = (my_loc - st_loc) / 2.0f;
}
```

Kärkivarjostin, jota käytetään varjokartan renderoimisessa, *rtw_shadow.vert*:

```
#version 330

uniform mat4 viewProjection;
uniform mat4 worldTransform;
uniform sampler2D warpMap;

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

vec2 ClipToTexcoord(vec2 c) { return c * 0.5f + 0.5f; }
vec3 ClipToTexcoord(vec3 c) { return c * 0.5f + 0.5f; }

vec2 TexcoordToClip(vec2 c) { return c * 2.0f - 1.0f; }
vec3 TexcoordToClip(vec3 c) { return c * 2.0f - 1.0f; }

// lue vääntymä vääntymiskartasta
vec2 GetOffsetLocation(vec2 ts){
    ts.x += texture(warpMap, vec2(ts.x,0.25f)).x;
    ts.y += texture(warpMap, vec2(ts.y,0.75f)).x;
    return ts;
}

void main(void) {
    vec4 position = worldTransform * vec4(inPosition, 1);
    position = viewProjection * position;

    vec2 tex_loc = ClipToTexcoord(position.xy / position.w);
    vec2 off_loc = GetOffsetLocation(tex_loc);

    position.xy = TexcoordToClip(off_loc) * position.w;

    gl_Position = position;
}
```

Kärkivarjostin, jota käytetään lopullisen lopputuloksen renderoinnissa, *rtw_scene.vert*:

```
#version 330

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;

out vec2 texcoord;

void main(void) {
    gl_Position = vec4(inPosition, 1);
    // muunna kärjen koordinaatti [-1, 1], tekstuurikoordinaatiksi [0, 1]
    texcoord = inPosition.xy * 0.5f + 0.5f;
}
```

Palavarjostin, jota käytetään lopullisen lopputuloksen renderoinnissa, *rtw_scene.frag*:

```
#version 330
float shadow_bias = 0.00004f;
uniform sampler2D   shadowMap;
uniform sampler2D   depthMap;
uniform sampler2D colorMap;
uniform sampler2D warpMap;
uniform mat4 cameraToLight;

in vec2 texcoord;
out vec4 oColor;
vec2 GetOffsetLocation(vec2 ts){
    ts.x += texture(warpMap,vec2(ts.x,0.25f)).x;
    ts.y += texture(warpMap,vec2(ts.y,0.75f)).x;
    return ts;
}
vec3 ClipToTexcoord(vec3 c) { return c * 0.5f + 0.5f; }

void main(void) {
    // haetaan palan syvyys ja normaali jo renderoiduista bittikartoista
    float dpt = texture( depthMap, texcoord ).x;
    vec4 col = texture( colorMap, texcoord );
    float intensity = 1;

    if (dpt < 1.0f) {
        vec4 pos = vec4(texcoord, dpt, 1) * 2.0f - 1.0f;
        vec4 sc = cameraToLight * pos;
        vec3 shadow_coord    = ClipToTexcoord(sc.xyz / sc.w);

            shadow_coord.xy = GetOffsetLocation(shadow_coord.xy);

        // hae lähemmän geometrian etäisyys
        // tämän palan kohdalla valon näkökulmasta
        float tex_d = texture(shadowMap, shadow_coord.xy).x;
        // etäisyyksien ero
        float z_diff = shadow_coord.z - tex_d;
        intensity =  1 - clamp((z_diff - shadow_bias) / shadow_bias, 0, 1);
    }

    oColor = col * mix(0.0f, 1.0f, intensity);
}
```