Toni Tiainen

Säteenseurannan käyttö reaaliaikaisessa renderoinnissa

Tietotekniikan kandidaatintutkielma

31. toukokuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Toni Tiainen

Yhteystiedot: toni.w.tiainen@student.jyu.fi

Työn nimi: Säteenseurannan käyttö reaaliaikaisessa renderoinnissa

Title in English: Use of ray tracing in real-time rendering

Työ: Kandidaatintutkielma

Sivumäärä: 29+0

Tiivistelmä: Reaaliaikaisessa renderoinnissa käytetään lähes poikkeuksetta rasterointia. Säteenseuranta on kuitenkin viime aikoina noussut varteenotettavaksi vaihtoehdoksi rasteroinnin rinnalle. Tässä tutkielmassa selvitetään, miten säteenseuranta ja rasterointi eroavat, sekä miten säteenseurantaa täytyy optimoida, jotta sitä pystytään hyödyntämään reaaliajassa.

Avainsanat: renderointi, reaaliaikainen renderointi, säteenseuranta, rasterointi, Nvidia OptiX

Abstract: For a long time rasterisation has been considered the only method for real-time rendering. In recent years ray tracing has become a viable alternative. In this paper is explained how ray tracing differs from rasterisation, and how ray tracing must be optimized so it can be used in real time.

Keywords: rendering, real-time rendering, ray tracing, rasterisation, Nvidia OptiX

Kuviot

Kuvio 1. Renrointiputken rakenne ylhäältä alaspäin 5
Kuvio 2. Eri varjostustyyppejä. Vasemmalla tasainen varjostus, keskellä Gouraud-varjostus ja oikealla Phong-varjostus (Bishop ja Weimer 1986). 8
Kuvio 3. Säteenseurannan toimintaperiaate 12
Kuvio 4. Materiaalin vaikutus valon heijastumiseen tai siroamiseen kappaleen pinnasta (Möller ja Haines 2002, s. 206) 14
Kuvio 5. Yksinkertainen ympäröivien alueiden hierarkia (Pharr, Wenzel ja Humphreys 2010, s. 209) 16
Kuvio 6. Säteenseurantaputki (*ray tracing pipeline*) (Parker et al. 2013, s. 206) 20

Sisältö

1 JOHDANTO 1

2 RENDEROINTI 3
2.1 Renderointiputki 4
2.2 Renderointiputken optimointi ja reaaliaikainen renderointi 5
2.3 Varjostus ja valaistusmallit 7

3 SÄTEENSEURANTA JA REALISTINEN RENDEROINTI 10
3.1 Säteenseuranta-algoritmi 11
3.2 Säteenseurannan optimointi 15

4 SÄTEENSEURANNAN KÄYTTÖ REAALIAIKAISESSA RENDEROINNISSA 18
4.1 Nvidia OptiX ja säteenseurantaputki 19
4.2 Säteenseuranta näytönohjaimella 20

5 YHTEENVETO 23

KIRJALLISUUSLUETTELO 24

1 Johdanto

Vain kaksi vuosikymmentä sitten ihmisiä hämmästytti maailman ensimmäinen kolmiulotteinen videopeli Quake (idSoftware, 1996) sekä ensimmäinen täysimittainen animaatioelokuva ToyStory (Pixar, 1997). Monet ihmiset muistavat nuo viihdeteollisuuden järkäleet ensimmäisinä kosketuksinaan aivan uudenkaltaiseen tietokonegrafiikkaan, renderointiin (engl. *rendering*).

Renderoinnilla tarkoitetaan kolmiulotteisen mallin esittämistä kaksiulotteisella pinnalla, yleensä tietokoneen näytöllä. Modernit renderointimenetelmät hoitavat samalla muun muassa tekstuurien piirtämisen sekä valaistuksen, kuten heijastusten ja varjojen, simuloinnin (Salomon 2011, s. 851). Vaikka tietokoneiden laskentateho on kasvanut eksponentiaalisesti aina viime vuosiin asti, on realistisen kuvan renderoiminen silti vaativa operaatio, joka vie aikaa. Renderointimenetelmät voidaankin jakaa kahteen ryhmään sen perusteella, pystytäänkö kyseisellä menetelmällä piirtämään kuva reaaliajassa. Renderointia ja rasterointia käsitellään tämän tutkielman luvussa 2.

Reaaliaikainen renderointi on uuden kuvan luomista reaaliajassa, tyypillisesti käyttäjän syötteiden perusteella, luoden illuusion liikkuvasta kuvasta ja vuorovaikutuksesta (Möller ja Haines 2002, s. 1). Tämä kuitenkin vaatii uuden kuvan piirtämisen useita kymmeniä kertoja sekunnissa, mihin perinteinen säteenseuranta ei nykyisellä prosessointiteholla kykene. Sen takia joudutaan turvautumaan huomattavasti nopeampaan menetelmään, rasterointiin, sekä luomaan keinotekoista realistisuutta erilaisilla valaistusmalleilla ja varjostuksilla (Möller ja Haines 2002, s. 118, s. 345). Reaaliaikaista renderointia käsitellään luvussa 2.2.

Säteenseuranta (engl. *ray tracing*) on yksi suosituimpia renderointimenetelmiä, joka perustuu valon fysikaalisiin ominaisuuksiin. Vaatiessaan valtavasti laskentaa, yksittäisen kuvan renderoiminen säteenseurantaa käyttämällä voi kestää useita tunteja, tuottaen kuitenkin muita renderointimenetelmiä realistisempia tuloksia (Pharr, Wenzel ja Humphreys 2010, s. 4-13). Tässä tutkielmassa selvitetään, miten säteen-

seurantaa toimii ja miten sitä voidaan optimoida siten, että sitä pystyttäisiin käyttämään reaaliajassa. Lisäksi tutustutaan Nvidian OptiX -grafiikkamoottoriin, joka hyödyntää näytönohjaimen rinnakkaislaskentaa säteenseurantaan. Säteenseurantaa ja sen optimointia käsitellään tämän tutkielman luvussa 3, ja Nvidia Optix -grafiikkamoottoria luvussa 4.

2 Renderointi

Koska tietokoneet osaavat käsitellä vain numeroita, joudutaan tietokonegrafiikkaa luomaan käyttämällä pisteitä ja niitä vastaavia vektoreita. Renderoinnilla tarkoitetaankin noiden pisteiden muuttamista kappaleiksi ja niiden piirtämistä kaksiulotteiselle näytölle luoden samalla illuusion kolmiulotteisuudesta.

Renderointialgoritmit voidaan jakaa kahteen ryhmään sen perusteella, miten ne hoitavat kolmiulotteisen kappaleen piirtämisen kaksiulotteiselle pinnalle. Yksi tapa on käydä läpi näytön pikseleitä ja selvittää minkä värisenä kyseinen pikseli kuuluu näyttää. Tätä kutsutaan säteenseurannaksi, koska pikselin väri määritetään lähettämällä säteitä pikseleistä kappaleisiin. Säteenseurantaa käsitellään myöhemmin tämän tutkielman luvussa 3.

Toinen tapa on käydä läpi kappaleita ja määrittää ne pikselit, joissa kappaleen pinnat näkyvät. Kappaleen näkyvistä pinnoista muodostetaan bittikarttoja, eli yhden tai useamman pikselin paloja (engl. *fragment*), joiden perusteella rakennetaan lopullinen kuva. Koska bittikarttoja kutsutaan myös rasterikuviksi, kutsutaan tätä menetelmää rasteroinniksi.

Salomon (2011) huomauttaakin, että renderointi voi yksinkertaisimmillaan olla kolmiulotteisen objektin pinta-alan näyttämistä tietystä kuvakulmasta. Tämä ei usein kuitenkaan riitä, sillä pelkän pinta-alan esittäminen ei luo vaikutelmaa kolmiulotteisuudesta. Tämän takia tietokoneen on luotava kappaleille varjostuksia ja heijastuksia. Rasteroinnin tapauksessa tämä onnistuu käymällä läpi niin kutsuttu renderointiputki (engl. *rendering pipeline*, joskus myös *graphics pipeline*), jonka tehtävänä on antaa rasteroinnille esitietoja siitä, miten annettuja kappaleita kuuluu piirtää. Näihin esitietoihin kuuluu mm. varjostukset, heijastukset ja tekstuurit.

2.1 Renderointiputki

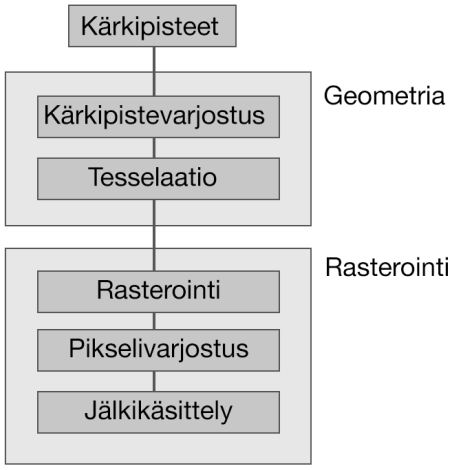
Renderointiputkella tarkoitetaan tapahtumaketjua, joka täytyy suorittaa jokaisen rasteroimalla renderoidun kuvan yhteydessä. Möllerin ja Hainesin (2002, s. 9) määritelmän mukaan renderointiputki muodostuu seuraavista kolmesta vaiheesta:

- Ohjelmisto (*Application*): Rajapinta, joka muodostaa renderoitavat kappaleet, sekä määrittää valonlähteiden ja kameran sijainnit.
- Geometria (*Geometry*): Laskee vektorioperaatioita, joiden tehtävänä on muokata kappaleita, sekä luoda varjostuksia.
- Rasterointi (*Rasterisation*): Rasteroi kolmiulotteiset kappaleet ja suorittaa pikselioperaatioita, joiden tehtävänä on määrittää, minkä väriseksi pikseli piirretään.

Nämä vaiheet sisältävät useita eri operaatioita, jotka täytyy suorittaa ennen, kuin seuraavaan vaiheeseen voidaan siirtyä. Kuviossa 1 ylimpänä on ohjelmistorajapinta, keskimmäisenä geometriavaihe ja alimpana rasterointivaihe.

Kun renderoitavaa näkymää luodaan, käytetään usein ohjelmistoa, esimerkiksi 3D-mallinnusohjelmaa tai pelimoottoria, joka toimii rajapintana käyttäjän ja renderointimoottorin välillä. Nämä rakentavat itse näkymän, joka syötetään renderointiputkessa eteenpäin. Realiaikasessa renderoinnissa ohjelmistot käsittelevät muun muassa kappaleiden liikkumisen, törmäykset sekä muut kappaleisiin kohdistuvat voimat. Tässä vaiheessa luotu matemaattinen malli täytyy syöttää geometriavaiheelle, jossa syötetyistä vektoreista muodostetaan pintoja (kutsutaan myös polygoneiksi), joista luodaan kolmiulotteisia kappaleita (Möller ja Haines 2002, s. 9-24).

Geometriavaihe sisältää suurimmaksi osaksi vektorioperaatioita, jotka käsittelevät kappaleiden sijainteja ja pintojen suuntia. Joissain tapauksissa on tarpeellista muokata kappaleita, esimerkiksi kun luodaan varjostuksia tai pehmennetään kulmia jakamalla pintoja pienempiin osiin (Möller ja Haines 2002, s. 13-19). Yksi tärkeimpiä tehtäviä on kuitenkin tarkastaa, mitkä kappaleet ovat kameran reunojen luoman näkymätilavuuden sisällä ja poistaa ulkopuolelle jääneet kappaleet (kutsutaan rajaamiseksi, engl. *clipping*). Näin pyritään minimoimaan rasterointiin menevien pintojen



Kuvio 1. Renrointiputken rakenne ylhäältä alaspäin

määrä ja nopeuttamaan renderointia.

Renderointiputken viimeisenä osana on itse rasterointi, jossa käsitellään pikseleitä. Tarkoituksena on projisoida näkyvät pinnat näytölle ja lukea geometriavaiheessa määritellyistä tekstuureista, minkä väriseksi yksittäinen pikseli kuuluu piirtää (Möller ja Haines 2002, s. 9-24). Rasterointivaiheessa suoritetaan myös kuvan jälkikäsittely, kuten reunapehmennys (engl. *Anti-aliasing*). Renderoitujen pikselien tiedot tallennetaan näyttöpuskuriin (engl. *framebuffer*), josta pikselien värit lähetetään näytölle. Erilaisia puskureita käytetään myös muihin tarkoituksiin, kuten rasteroidun pinnan ja pikselin etäisyyden tallentamiseen. Tätä niin kutsuttua z-puskuria käytetään rasteroitaessa, jotta tiedetään mikä pinta on lähimpänä kameraa ja täytyy näyttää lopullisessa kuvassa.

2.2 Renderointiputken optimointi ja reaaliaikainen renderointi

Fotorealistisen kuvan renderoiminen voi nykyäänkin viedä kymmeniä minuutteja, jopa tunteja. Tämä ei usein kuitenkaan ole vaihtoehtona. Esimerkiksi videopeleissä on tärkeää, että liike on sujuvaa ja käyttäjän syötteisiin vastataan mahdollisimman lyhyellä viiveellä. Reaaliaikainen renderointi tuokin renderoinnin nopeuden yhdeksi prosessin tärkeimmistä kriteereistä.

Möller ja Haines (2002) toteavat, että renderointiputkessa on aina pullonkaula: paljon laskentaa vaativa operaatio, joka vie enemmän aikaa kuin muut operaatiot. Tuon pullonkaulan löytäminen on tärkeää, kun yksittäisen kuvan renderoimiseen voi käyttää korkeintaan millisekuntteja. Kun pullonkaula on löydetty, voidaan kyseistä prosessia yrittää optimoida.

Möller ja Haines (2002, s. 345) nostavat renderointiprosessin vaativammaksi osuudeksi näkyvien kappaleiden selvittämisen, sillä sen vaativuus kasvaa mitä enemmän kappaleita renderoitavassa näkymässä on. Näkyvien kappaleiden etsimiseen voidaan käyttää säteenlähetystä (engl. *ray casting*), missä ideana on lähettää säteitä pikseleiden läpi ja etsiä ensimmäinen pinta, mihin kyseinen säde osuu (Hughes et al. 2013). Samaa menetelmää käytetään myös osana säteenseurantaa ja sitä käsitellään tarkemmin luvussa 3.

Näkyvien kappaleiden selvittäminen voidaan myös hoitaa osana rasterointia projisoimalla yksittäinen pinta näytölle ja tallentamalla jokaisen pinnan sisältämien pikseleiden etäisyydet kameraan z-puskuriin (Hughes et al. 2013). Näin pystytään tarkistamaan, oliko aiemmin käsitelty pinta lähempänä kameraa ja tallentamaan vain lähimpänä olleen pinnan väri pikselin tietoihin.

Kun kappaleiden määrä ja yksityiskohtaisuus kasvavat, kasvaa myös verrattavien pintojen määrä. Yksi tapa vähentää pintojen määrää ja nopeuttaa näin aiemmin kuvattuja operaatioita, on alentaa yksityiskohtien tasoa (engl. *level of detail, LOD*). Tämä tarkoittaa kuvan kannalta vähemmän merkityksellisten, esimerkiksi kaukana olevien pintojen sulauttamista toisiinsa (Möller ja Haines 2002, s. 368-400).

Leikkauspisteiden laskentaa voidaan nopeuttaa muillakin tavoilla, esimerkiksi jakamalla koordinaatistoavaruus siten, että yhdessä osassa on vain yksi kappale (ns. binäärinen avaruuden osiointi, engl. *Binary Space Partitioning*) ja luomalla näistä osioista kappaleita ympäröivien tilavuuksien hierarkia (engl. *Bounding Volume Hierarhy, BVH*). Nämä ovat säteenseurannan optimoinnissa elintärkeitä menetelmiä ja niitä käsitelläänkin tarkemmin luvussa 3.2.

Eräitä prosesseja voidaan jopa korvata toisilla. Esimerkiksi varjostuksia pystytään

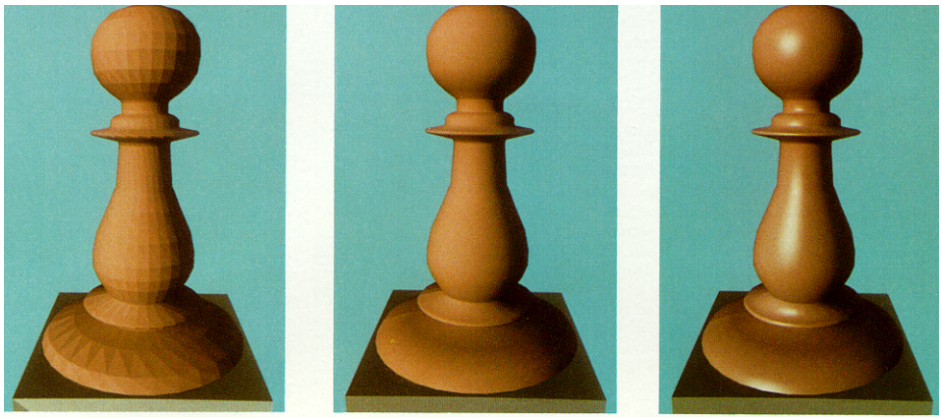
luomaan kärkipistevarjostuksina (engl. *vertex shader*) tai pikselivarjostuksina (engl. *pixel shader*). Se, kumpi näistä on tehokkaampi, riippuu renderoitavasta kuvasta. Jos kappaleita on paljon, on varjostus raskaampaa laskea jokaiselle yksittäiselle pinnalle kerrallaan, kuin kiinteälle määrälle pikseleitä (Möller ja Haines 2002, s. 213-232).

Huolimatta siitä, kuinka paljon renderointiputkea pystytään optimoimaan, ei reaaliaikainen renderointi onnistuisi ilman laitteistokiihdytystä (engl. *hardware acceleration*). Tietokonegrafiikassa laitteistokiihdytyksellä tarkoitetaan usein näytönohjainta, joka on suunniteltu nopeuttamaan renderointiputken suorittamista rinnakkaislaskennalla, eli suorittamalla prosessia samaan aikaan usealla eri syötteellä. Esimerkiksi sama rasterointioperaatio täytyy suorittaa jokaiselle näkymässä olevalle pinnalle, joten on tehokasta ajaa sama ohjelmakoodi usealle pinnalle kerrallaan (Salomon 2011, s. 4-5). Näytönohjaimen toimintaa käsitellään tarkemmin tutkielman loppupuolella, missä perehdytään säteenseurannan suorittamiseen näytönohjaimella.

2.3 Varjostus ja valaistusmallit

Tärkein osa realismiin pyrkivää renderointia on varjostuksien ja heijastusten luominen. Tämä tapahtuu tietokonegrafiikassa hyödyntämällä pinnan normaalia, joka kertoo mihin suuntaan pinta osoittaa, sekä valonlähteiden sijainteja ja suuntia. Näiden perusteella valaistusmallit (engl. *Illumination model*) arvioivat, kuinka paljon valoa pintaan osuu ja millaisen varjon valot luovat. Valaistusmallilla tarkoitetaan yhden tai useamman algoritmin yhdistelmää, joilla saadaan laskettua tietyn pisteen kirkkaus. Näitä valaistusmalleja hyödynnetään varjostusalgoritmeissa, jotka Möller ja Haines (2002, s. 70) jakavat kolmeen tärkeimpään varjostustyyppiin: tasainen, Gouraud ja Phong.

Tasainen varjostus tarkoittaa, että jokaiselle pinnalle lasketaan yksi väri, joka näytetään jokaisessa pikselissä, missä pinta näkyy. Tämä on nopea laskea ja helppo toteuttaa, mutta tuottaa kulmikkaan lopputuloksen, minkä voi huomata vasemmalla kuviossa 2. Tasainen varjostus hyödyntää valaistusmallinaan Lambertin lakia, jonka mukaan pinnan kirkkaus saadaan valon tulokulmasta pinnan suhteen.



Kuvio 2. Eri varjostustyyppejä. Vasemmalla tasainen varjostus, keskellä Gouraud-varjostus ja oikealla Phong-varjostus (Bishop ja Weimer 1986).

Kulmikkaasta varjostuksesta päästään eroon käyttämällä Gouraud-varjostusta, jossa pinnan määrittäville kärkipisteille lasketaan kirkkaudet käyttämällä normaalia, joka on laskettu pisteen jakavien pintojen normaalien perusteella. Tämän jälkeen pinnalle muodostetaan yhtenäinen varjostus käyttämällä interpolaatiota[1] kärkipisteiden välille (Gouraud 1971). Tämä luo sulavamman lopputuloksen kuin tasainen varjostus, kuten kuviossa 2 huomataan. Koska Gouraud-varjostus luo varjostuksen käyttämällä kärkipisteitä, kutsutaan sitä myös kärkipistevarjostukseksi (engl. *vertex shading*).

Phong (1975) huomauttaa, että Gouraud-varjostus kärsii varjostuksen epäjatkuvuudesta pintojen välillä sekä kirkkaiden heijastusten vääristymisestä. Ratkaisuna näihin ongelmiin Phong kehitti omaa nimeään kantavan varjostusalgoritmin, joka interpoloi kirkkauden sijaan normaaleja, joita käytetään varjostuksen määrittämiseen jokaiselle yksittäiselle pikselille erikseen. Tämän takia Phong-varjostusta kutsutaan myös pikselivarjostukseksi.

Phong kehitti varjostusalgoritmiaan varten myös aiempaa kehittyneemmän valaistusmallin, joka voidaan jakaa kolmeen eri osaan:

- Ympäröivä valo (*Ambient light*): Kasvattaa jokaisen näkymässä olevan kappa-

1. Interpolaatio on matemaatiikassa menetelmä kahden ääriarvon välisten arvojen arviointiin.

leen kirkkautta, jos näkymässä on valonlähde. Tämän tarkoituksena on esittää ympäristöstä heijastuvaa valoa.

- Diffuusi valo (*Diffuse light*): Luo perusvarjostuksen käyttämällä Lambertin sääntöä.
- Kiilteet (*Specular Highlight*): Kasvattaa kiiltävien kappaleiden valoa kohti osoittavan pinnan kirkkautta.

Kuten Whitted (1979) huomauttaa, Phong-valaistusmalli ei ota huomioon epäsuoraa valaistusta, eikä luo realistisia heijastuksia. Whitted kehitti valaistusmallin, joka ottaa huomioon ympäröivistä kappaleista heijastuvat valonsäteet, sekä luo realistiset heijastukset kiiltävien kappaleiden pintoihin. Yhdistettynä näkyvien pintojen selvittämisessä käytettyyn säteenlähetykseen, syntyi säteenseuranta, johon perehdytäänkin seuraavassa luvussa.

3 Säteenseuranta ja realistinen renderointi

Rasteroinnissa käytetyt varjostusmenetelmät ovat nopeita, mutta luovat vain matemaattisen arvion siitä, miltä valo näyttäisi kappaleiden pinnoilla. Tietokonegrafiikassa on kuitenkin pyritty aina vain realistisempaan tulokseen, mihin Turner Whittedin vuonna 1979 kehittämä valaistusmalli, säteenseuranta (engl. *ray tracing*, joskus myös *path tracing*), pyrkii antamaan vastauksen.

Säteenseuranta on luotu simuloimaan valon käyttäytymistä sen heijastuessa kappaleista (Pharr, Wenzel ja Humphreys 2010, s. 4). Se mahdollistaa realististen varjostusten ja valaistusten, sekä läpinäkyvien kappaleiden renderoimisen arvioimalla, kuinka valo heijastuu kappaleiden pinnoilta luoden epäsuoran valaistuksen. Koska säteenseuranta ottaa huomioon myös kappaleet, jotka eivät suoraan näy kameralle, vaan vaikuttavat näkymään heijastusten kautta, se pystyy luomaan fysikaalisesti realistisia heijastuksia ja tuottamaan näin muita menetelmiä realistisempia kuvia. Tämän lisäksi Wald ym. (2001) listaavat säteenseurannan eduiksi suhteessa rasterointiin seuraavat ominaisuudet:

- *Näkymättömien kappaleiden leikkaus* (*Occlusion Culling)*: Säteenseuranta-algoritmi käsittelee automaattisesti vain näkyviä kappaleita, eikä näkyvien kappaleiden selvitystä erikseen tarvita.
- *Logaritminen vaativuus*, $O(\log n)$: Hyvin optimoidun säteenseuranta-algoritmin vaativuus on logaritminen näkymässä olevien pintojen määrän suhteen (vrt. rasteroinnin vaativuus on lineaarinen $O(n)$).
- *Muokattavuus*: Säteenseurantaa voidaan käyttää yksittäiseen säteeseen tai säderyppääseen, mahdollistaen tehokkaan tavan laskea vain tarpeellinen tieto, esimerkiksi vain pieni osa heijastuksesta.
- *Tehokas varjostus*: Varjostukset lasketaan vain näkyville pinnoille, toisin kuin muissa menetelmissä, missä varjostukset lasketaan jokaiselle pinnalle ennen rasterointia.
- *Valaistusmallien helpompi käyttö*: Koska säteenseuranta ei noudata lineaarista

renderointiputkea, erilaisten valaistusmallien käyttö on helpompaa.

3.1 Säteenseuranta-algoritmi

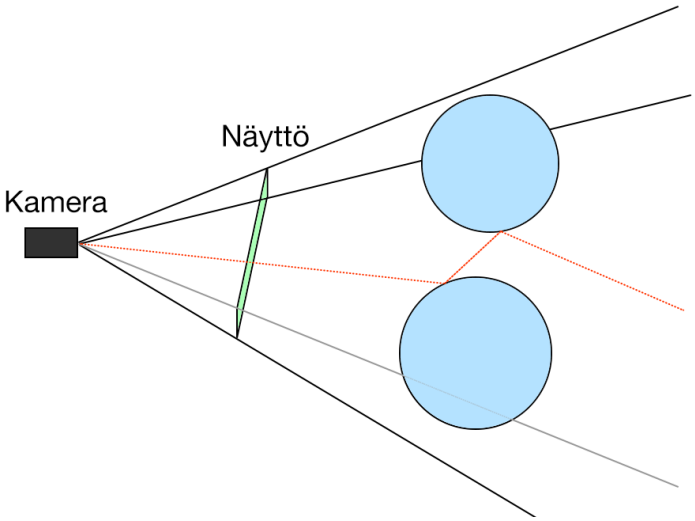
Renderointiputkea hyödyntävät menetelmät sisältävät usein monimutkaisia algoritmeja, joiden tehtävänä on sekä lisätä kuvan realistisuutta, että vähentää laskettavien kohteiden määrää. Näihin verrattuna säteenseurantaan käytetty algoritmi on huomattavasti yksinkertaisempi, mutta myös laskennallisesti vaativampi.

Tässä tutkielmassa käsiteltävä säteenseuranta-algoritmi on Matt Pharrin, Greg Humphreysin ja Jacob Wenzelin (2010) kehittämä PBRT (*Physically Based Ray Tracer*). Se perustuu vahvasti Turner Whittedin (1979) alkuperäiseen valaistusmalliin.

Pharr, Humphreys ja Wenzel listaavat säteenseurannalle olennaiset asiat seuraavasti:

- *Kamerat*: Kertovat mistä näkymää katsellaan, sekä suunta mihin säteitä lähetetään.
- *Säteiden ja kappaleiden leikkauspisteet*: Säteenseurannassa on tärkeä löytää kappaleen pinnalta piste, johon säde osui, jotta tiedetään kyseisen pinnan normaali ja materiaali.
- *Valonlähteet*: Jotta kappaleet näkyisivät kuvassa, täytyy näkymässä olla yksi tai useampi valonlähde, joiden vaikutuksia kappaleiden pintoihin mallinnetaan.
- *Valon heijastuminen ja siroaminen*: Jokainen kappale sisältää tiedon siitä, miten valo käyttäytyy kyseisen kappaleen pinnalla.
- *Rekursio*: Koska säteenseurannassa otetaan huomioon säteiden heijastuminen ennalta määräämättömien kappaleiden kautta, täytyy samaa algoritmia pystyä käyttämään myös heijastuneisiin säteisiin.
- *Säteiden eteneminen*: Säteenseurannan täytyy ottaa huomioon, mitä säteelle tapahtuu sen edetessä tyhjässä tilassa.

Kamera (joissain tapauksissa myös silmä) on säteenseurannan alkupiste, mistä säteet lähetetään. Koska renderoitava kuva koostuu pikseleistä, yhden säteen tehtä-



Kuvio 3. Säteenseurannan toimintaperiaate

vä on selvittää minkä väriseksi yksittäinen pikseli kuuluu värjätä. Tämä onnistuu lähettämällä säde kamerasta virtuaalisen näytön pikselin läpi näkymään. Sädettä käsitellään puolisuorana, jonka alkupisteenä on kamera ja suuntana vektori kamerasta pikseliin. Kuviossa 3 kamerasta lähetetään punaisella katkoviivalla merkitty säde näytön läpi kappaleisiin.

Kun säde on lähetetty, täytyy selvittää mihin pintaan säde osui. Esimerkiksi Möller–Trumbore-leikkauspistealgoritmi (Möller ja Trumbore 2005) on nopea ja muistia säästävä menetelmä, joka palauttaa pinnan etäisyyden säteen lähetyspisteestä, sekä niin kutsutut pinnan barysentriset koordinaatit [1]. Etäisyyttä käytetään ensisijaisesti lähimmän kappaleen löytämiseen, mutta myös selvittämään kuinka pitkän matkan säde kulkee tyhjässä tilassa, mikä vaikuttaa mm. säteen voimakkuuden heikkenemiseen.

Kun lähin säteen reitillä oleva pinta on löydetty, tehtävänä on selvittää kuinka paljon valoa tuo piste heijastaa kameraan. Tämän selvittämiseksi täytyy ensin tietää kuinka paljon valoa pisteeseen tulee. Tämä saadaan laskettua lähettämällä leikkauspisteestä lisää säteitä, joita kutsutaan yleensä näytteiksi (engl. *sample*).

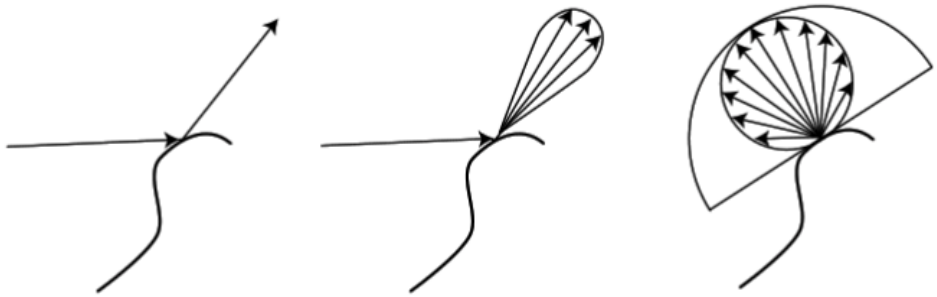
1. Lineaarialgebrassa barysentrisillä koordinaateilla tarkoitetaan pinnan kärkipisteiden painoarvojen avulla määriteltyä pistettä pinnan sisältä.

Pinnan tärkein ominaisuus on sen normaali, joka on yleensä laskettu etukäteen ja tallennettu pinnan tietoihin sen kärkikoordinaattien lisäksi. Rasteroinnissa käytettyä Phong-varjostusta voidaan hyödyntää myös pyöreiden pintojen pehmentämisessä, sillä Phongin algoritmissa lasketaan normaalien interpolaatioita kärkipisteiden avulla. Modernit säteenseuranta-algoritmit, kuten PBRT, voivat käyttää myös normaalimuunnoskarttaa (engl. *normal mapping* tai *bump mapping*) eli tekstuuria, joka kertoo mihin suuntaan normaalia kuuluu kääntää tietyssä kohdassa pintaa (Salomon 2011, s. 879). Näin voidaan esimerkiksi luoda rosoinen tai kulunut pinta käyttämällä pelkkää tekstuuria mallintamatta jokaista yksityiskohtaa erikseen.

Pelkästään normaalin avulla voidaan jo selvittää, valaiseeko jokin valonlähde pisteen. Tämä on varsin yksinkertainen tehtävä, joka pystytään ratkaisemaan lähettämällä näytteet jokaisen valonlähteen suuntaan ja tarkastamalla onko jokin pinta lähempänä kuin valonlähde. Tällöin pinta on valonlähteen edessä ja piste on varjossa. Jos pisteen ja valonlähteiden välissä ei ole kappaleita, voidaan pisteen kirkkaus laskea näkyvien valonlähteiden voimakkuuksien summana, johon vaikuttaa valonlähteen etäisyys, sekä valon heijastuskulma normaalin suhteen (Möller ja Haines 2002, s. 8). Pelkkien valonlähteiden näkyvyyden selvittäminen ei kuitenkaan riitä realistisen kuvan renderoimiseen, vaan huomioon täytyy ottaa myös kappaleiden kautta heijastuva valo.

Eri materiaalit heijastavat valoa eri tavalla. Esimerkiksi kirkkaan metallipinnan heijastus on terävä, kun taas puinen pinta ei näytä heijastavan valoa lainkaan. Todellisuudessa jokainen pinta heijastaa valoa, mutta valo hajoaa (kutsutaan myös *sironnaksi*) pinnassa siten, että heijastus ei näy terävänä. Näitä fysikaalisia ominaisuuksia pyritään mukailemaan kappaleiden materiaaleilla (Möller ja Haines 2002, s. 11). Näitä materiaalitietoja käytetään heijastusfunktioissa, joita kutsutaan lyhenteellä BSDF (*Bidirectional Scattering Distribution Function*), joka muodostuu kahdesta eri funktiosta: BRDF (*Bidirectional Reflectance Distribution Function*) ja BTDF (*Bidirectional Transmittance Distribution Function*).

BRDF pyrkii mallintamaan kuinka kuinka valo heijastuu kappaleen pinnasta, kun taas BTDF mallintaa valon kulkeutumista kokonaan tai osittain läpinäkyvien kap-



Kuvio 4. Materiaalin vaikutus valon heijastumiseen tai siroamiseen kappaleen pinnasta (Möller ja Haines 2002, s. 206)

paleiden läpi (Möller ja Haines 2002, s. 194). Ne ottavat syötteenä valon tulo- ja lähtökulman, voimakkuuden, sekä aallonpituuden (kuvataan tietokonegrafiikassa RGB-väriarvona). Heijastusfunktiot palauttavat arvion siitä, kuinka paljon valoa heijastuu lähtökulman suuntaan.

Perinteisessä säteenseurannassa näytteitä lähetetään valonlähteiden lisäksi kuvan kannalta tärkeisiin kappaleisiin, esimerkiksi metallipintoihin, jotka heijastavat paljon valoa (Möller ja Haines 2002, s. 284). Realistisemman tuloksen tarjoavat kuitenkin Monte Carlo ja Las Vegas -menetelmät, jotka perustuvat samannimisiin todellisen ja näennäisen satunnaisotannan (engl. *random sampling*) algoritmeihin (Pharr, Wenzel ja Humphreys 2010, s 637). Käytännössä tämä tarkoittaa sitä, että näytteitä lähetetään satunnaisesti alueelle, joka riippuu kappaleen materiaalista.

Esimerkiksi kuviossa 4 vasemmalla oleva malli kuvaa valon heijastumista täydellisestä peilipinnasta ja keskellä oleva heijastumista kiiltävästä pinnasta, joka ei kuitenkaan tuota terävää heijastusta. Oikealla oleva malli kuvaa taas valon siroamista heijastamattomasta kappaleesta. Se, kuinka monta sädettä lähetetään, vaikuttaa kuvan realistisuuteen ja vähentää rakeisuutta, mutta kasvattaa laskennan vaativuutta.

Turner Whitted korosti tutkielmassaan (1979) säteenseurannan rekursiivistä luonnetta. Hänen ideanaan oli, että samaa säteenseuranta-algoritmia voitaisiin käyttää kaikkiin heijastuviin säteisiin. Yhdessä satunnaisotantafunktioiden kanssa, heijastusfunktioita voidaan käyttää laskemaan yksittäisen pisteen kirkkautta rekursiivi-

sesti, lähettämällä jokaisesta säteen ja pinnan leikkauspisteestä lisää säteitä satunnaisiin suuntiin. Rekursio katkaistaan, kun säteen voimakkuus on laskenut ennaltamääritellyn rajan alle. Tällöin voidaan laskea jokaisen kappaleista heijastuneiden valojen summa ja tallentaa tuo tieto yksittäisen pikselin väriksi. Sama toistetaan jokaiselle näytön pikselille.

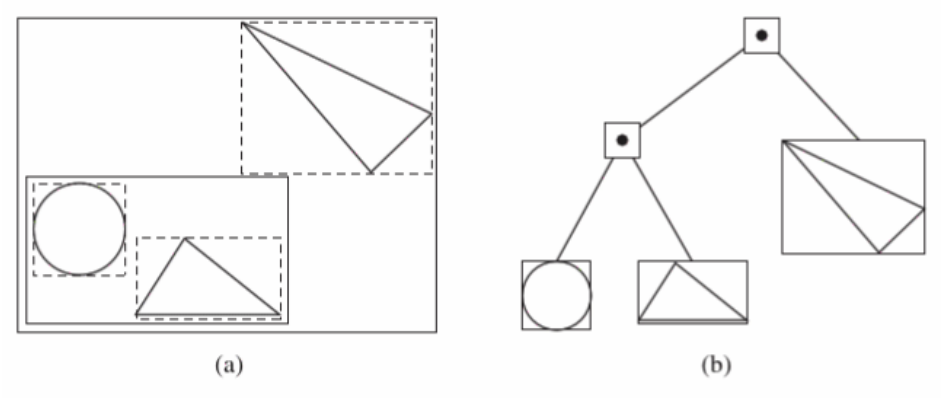
On siis selvää, että säteenseuranta on vaativa operaatio. Seuraavassa kappaleessa käsitellään säteenseurannan optimoinnin kannalta tärkeitä teoreettisia yksityiskohtia ja luvussa 4 tutkitaan säteenseurannan käyttöä reaaliaikaisessa renderoinnissa.

3.2 Säteenseurannan optimointi

Whitted (1979) toteaa tutkielmassaan, että säteenseurantaan käytetystä ajasta yli 95% menee säteen ja leikkauspisteen etsimiseen, kun näkymässä on paljon kappaleita. Tämä viittaa siihen, että säteenseurantaa pystytään nopeuttamaan merkittävästi tehokkaalla leikkauspistealgoritmilla. Valitettavasti lähetettävien säteiden määrä on niin suuri, että säteenseurantaa täytyy optimoida muillakin tavoilla, esimerkiksi näytteiden määrän radikaalilla vähentämisellä, jotta päästäisiin rasteroinnin tehokkuuteen ja reaaliaikaisen renderoinnin aikavaatimuksiin.

Optimoimattomana leikkauspistealgoritmi joutuu käymään kaikki näkymän pinnat läpi, jotta tiedetään mihin pintaan säde osuu ja mikä osumista on lähimpänä. Tällöin säteenseurannan vaativuus on lineaarinen näkymässä olevien kappaleiden suhteen (Pharr, Wenzel ja Humphreys 2010, s. 183). Onneksi leikkauspisteen etsintää voidaan nopeuttaa niin kutsutuilla kiihdytysmalleilla (engl. *acceleration structures*), jotka laskevat säteenseurannan vaativuuden logaritmiseksi (Pharr, Wenzel ja Humphreys 2010, s. 183). Tämä tarkoittaa sitä, että säteenseurannan vaativuus kasvaa hitaammin kuin rasteroinnin, kun näkymässä olevien pintojen määrä kasvaa.

Yleinen säteenseurannassa käytetty kiihdytysmalli on jo aiemmin mainittu kappaletta ympäröivien alueiden hierarkia (engl. *Bounding Volume Hierarchy, BVH* tai *Bounding Box Hierarchy, BBH*) (Pharr, Wenzel ja Humphreys 2010, s. 208). Ympäröivällä alueella tarkoitetaan yhden tai useamman kappaleen rajaamista saman alueen



Kuvio 5. Yksinkertainen ympäröivien alueiden hierarkia (Pharr, Wenzel ja Humphreys 2010, s. 209)

sisälle. Näin pystytään eliminoimaan kaikki alueen sisällä olevat kappaleet, jos säde ei osu aluetta rajaaviin pintoihin.

Näistä alueista voidaan luoda puu (hierarkia), jossa juurisolmu sisältää kaikki näkymän kappaleet ja lehtisolmut yksittäisen kappaleen tai pinnan. Näiden välissä olevat solmut toimivat linkkeinä aina vain pienempiin alueisiin, kunnes päädytään yhden kappaleen tai pinnan sisältämään lehtisolmuun. Ideana ympäröivien alueiden määrittämisessä on luoda geometrisesti yksinkertaisempia kappaleita, kuin alkuperäiset kappaleet, käyttämällä esimerkiksi palloja tai koordinaattiakselien suuntaisia nelikulmioita (Möller ja Haines 2002, s. 347). Tällöin säteen ja pinnan leikkauksen etsiminen nopeutuu.

Kun ympäröivien alueiden hierarkia on luotu, riittää säteenseuranta-algoritmille kulkea puun läpi ja tarkistaa pelkkien alueiden rajaavien pintojen leikkaukset, ennen viimeiseen lehtisolmuun päätymistä, jolloin etsitään varsinainen säteen ja pinnan leikkauspiste. Kuviossa 5 esitetään ympäröivien alueiden muodostaminen kappaleiden ympärille (a) ja näiden alueiden jakaminen binääripuuksi (b).

Yksinkertainen tapa nopeuttaa säteenseurantaa on vähentää leikkauspisteistä lähetettävien näytteiden määrää. Elokuvatasoisen kuvan renderointiin käytetään yleensä tuhansia näytteitä, mutta reaaliaikaisessa säteenseurannassa näytteiden määrää joudutaan pienentämään radikaalisti. Ongelma tässä optimoinnissa on renderoidun

kuvan rakeus, eli vierekkäisten pikselien kirkkausero. Säteenseurannan optimointi onkin viime vuosina keskittynyt suurelta osin rakeisen kuvan jälkikäsittelyyn.

Jälkikäsittelyyn käytetään yleensä niin kutsuttuja apusyötteitä (engl. *Auxiliary inputs*), joiden perusteella pyritään pehmentämään kuvaa säilyttäen kuitenkin kuvan terävyyden. Apusyötteitä voidaan saada rasteroimalla kuva ensin ja tallentamalla geometriapuskuriin näkyvien pintojen normaaleja ja syvyyksiä (etäisyyksiä pinnan ja kameran välillä). Schiedin ym. (2017) kehittämä menetelmä hyödyntää näiden lisäksi aiemmin renderoitujen kuvien sisältämiä väritietoja uuden kuvan rekonstruktiossa. Lisäksi heidän menetelmänsä hyödyntää apusyötteitä säteenseurannalla tuotetun kuvan jakamisessa suoriin ja epäsuoriin valaistuksiin, jotta vältytään tekstuurien ja terävien reunojen sumentamiselta. Aiemmista renderoinneista saatujen historiatietojen käyttö on kuitenkin haastavaa, varsinkin jos kuvien liike on nopeaa. Tämän takia Schiedin ym. menetelmä ei sovikaan esimerkiksi videopeleihin.

Toisen näkökulman jälkikäsittelyyn antaa Chaitanya ym. (2017), jotka ovat kouluttaneet neuroverkkoa hyödyntämään apusyötteitä renderoidun kuvan rekonstruktiossa. Sekä Schieldin ym., että Chaitanyan ym. menetelmissä käytettiin todella alhaista näytemäärää; yksi näyte pikseliä kohden. Prosessi sisältää tällöin yhden kamerasta lähetetyn pääsäteen, yhden uudelleensuunnatun näytteen, sekä kummassakin osumakohdassa suoritetun valonlähteiden etsinnän. Yhtä pikseliä kohden tulee siis suorittaa yhteensä neljä sädeoperaatiota.

Tietokoneiden muistin määrän ja lukunopeuden kasvaessa, nykyaikaiset optimointimenetelmät ovat selkeästi siirtyneet muistin ja puskurien käyttämiseen laskennan nopeuttamiseksi. Grafiikkamoottorit pyrkivät laskemaan ja tallentamaan näkymästä mahdollisimman paljon tietoa etukäteen, jotta itse renderointivaiheessa vaadittaisiin mahdollisimman vähän uutta laskentaa. Säteenseurannan käyttö reaaliaikaisessa renderoinnissä vaatii kuitenkin näytönohjaimen käyttöä, johon Steven G. Parkerin ym. (2013) kehittämä NVIDIA OptiX -grafiikkamoottori pyrkii tuomaan ratkaisun.

4 Säteenseurannan käyttö reaaliaikaisessa renderoinnissa

Rasterointi on perinteisesti mielletty nopeammaksi renderointimenetelmäksi kuin säteenseuranta. Sen lisäksi, että rasterointi on laskennallisesti vähemmän vaativaa, on nykyaikainen laitteisto kehitetty nimenomaan renderointiputken nopeuttamiseen. Parker ym. (2013) kuitenkin huomauttavat näytönohjainten kehittyneen yksittäiseen operaatioon erikoistuneista piireistä täysin ohjelmoitaviin yleiskäyttöisiin prosessoreihin (engl. *General Purpose GPU*). Näytönohjainten mahdollistaman rinnakkaislaskennan käyttö säteenseurannassa on heidän mukaansa välttämätön kehitysaskel, jotta säteenseurantaa voitaisiin käyttää reaaliaikaisesti.

Parkerin ym. (2013) tutkimus keskittyy NVIDIA OptiX -grafiikkamoottorin kehittämiseen, joka pyrkii nopeuttamaan säteenseuranta-algoritmin eri osia hyödyntämällä näytönohjaimen rinnakkaislaskentaa. He kuitenkin korostivat, että OptiX on pelkkä rajapinta, eikä varsinainen renderointimoottori. Se tarjoaa säteenseurantaan vaadittavan laskennan ja antaa käyttäjän luoda itse esimerkiksi heijastusfunktiot ja virheenkäsittelyt.

Parkerin ym. tutkimuksesta nousi esiin neljä säteenseurannan optimointiin vaadittavaa ominaisuutta, jotka on toteutettu OptiX-grafiikkamoottorissa:

- *Ohjelmoitava säteenseurantaputki*: Säteenseurannan operaatioiden jakaminen renröintiputkea vastaavaan suoritusmalliin.
- *Näkymähierarkian luominen*: Kappaletta ympäröivien alueiden, geometriatietojen ja käyttäjän määrittämien prosessien tallentaminen samaan tietorakenteeseen
- *Täsmäkääntäminen*: Koodin täsmäkääntäminen (engl. *domain specific compilation*) suoritusvaiheessa (engl. *just-in-time compilation*) näytönohjaimelle suunniteltuun konekieleen.
- *Hienojakoinen suoritusmalli*: Säteiden tilan tallentaminen ja samassa tilassa ole-

vien säteiden suorittaminen samanaikaisesti.

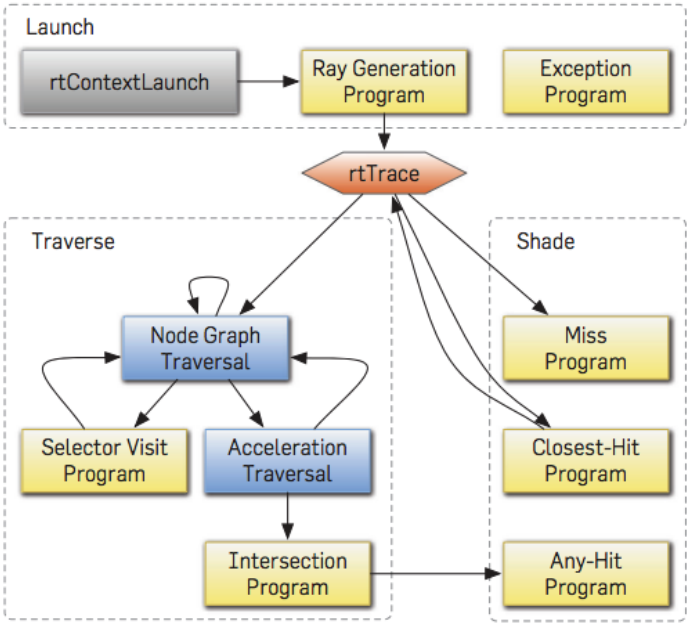
4.1 Nvidia OptiX ja säteenseurantaputki

Koska nykyaikainen grafiikkalaitteisto on optimoitu renderointiputken suorittamiseen, oli tärkeää muodostaa säteenseurannalle vastaava pieniin osiin jaettu suoritusmalli. Säteenseurantaputkella (engl. *ray tracing pipeline*) tarkoitetaan siis säteenseurantaalgoritmin pilkkomista yksittäisiin operaatioihin, joita voidaan suorittaa tietyssä järjestyksessä (Parker et al. 2013). Säteenseurannan rekursion takia putki ei kuitenkaan kata koko renderointiprosessia, vaan kuvaa yhteen säteeseen liittyvien prosessien kulkua.

Kuvio 6 kuvaa säteenseurantaputken rakennetta ja operaatioiden etenemistä. Koska OptiX on tarkoitettu rajapinnaksi, on säteenseurantaputkeen merkitty keltaisella ohjelmat, jotka ovat käyttäjän määritettävissä. Sinisellä on taas merkitty OptiX:n sisäiset ohjelmat, jotka keskittyvät pääosin kappaleiden hakemiseen ympäröivien alueiden hierarkiasta ja tarvittavien kappaletietojen noutamiseen muistista.

Säteenlähetysohjelma (kuvassa *Ray Generation Program*) päättää mihin suuntiin säteitä lähetetään ja mitä säteillä tehdään. Tämä antaa kehittäjälle vastuun siitä, käytetäänkö säteenseurantaa koko kuvan renderoimiseen, vai muihin operaatioihin, kuten fotonikartoitukseen (engl. *photon mapping*) tai varjostustekstuurien luomiseen (Parker et al. 2013).

Jokainen lähetetty säde käynnistää uuden säteenseurantaprosessin (kuvassa *rtTrace*) uuteen säikeeseen (engl. *thread*), joka vaihtelee leikkauspisteen löytämisen (kuvassa *Traverse*) ja varjostuksen (kuvassa *Shade*) välillä. Leikkauspiste kyetään löytämään tehokkaasti käyttämällä ympäröivien alueiden hierarkiaa. Jotta siirtymä varjostusprosessiin olisi mahdollisimman nopea, sisältää kappaleet tiedon käyttäjän määräämästä prosessista, joka ajaa kyseiselle kappaleelle. Käyttäjän määräämien prosessien tarkoituksena on tallentaa säteen väri, päättää lähetetäänkö lisää säteitä eteenpäin ja tarvittaessa lopettaa säteen seuranta.



Kuvio 6. Säteenseurantaputki (*ray tracing pipeline*) (Parker et al. 2013, s. 206)

4.2 Säteenseuranta näytönohjaimella

Pelkkä säteenseurantaputken määrittely ei kuitenkaan mahdollista sen tehokasta suorittamista näytönohjaimella, sillä näytönohjain ei kykene kunnolla käsittelemään prosessien haarautumista (engl. *branching*), toisin kuin keskussuoritin (Aila ja Laine 2009). Haarautumisella tarkoitetaan samalle datalle suoritettavien prosessien vaihtumista siten, että vaihtumiseen vaikuttaa aiemmin suoritetun prosessin tulos. Tällöin vaihtumista ei voida ennakoida. Esimerkiksi säteenseurannassa yhdelle säteelle suoritettavat prosessit voivat vaihdella sen mukaan, mihin kappaleisiin säde osuu. Koska osumakohdat eivät ole tiedossa etukäteen, haarautuu säteiden seuraaminen voimakkaasti. Tämä voidaan huomata jo säteenseurantaputkea tarkastelemalla: toisin kuin lineaarinen renderointiputki, säteenseurantaputken kulku haarautuu useaan eri suuntaan.

Haarautuvia prosesseja kutsutaan yleensä MIMD-prosesseiksi (*Multiple Instructions, Multiple Data*), sillä niissä datalle suoritetaan useita eri käskykantoja[1]. Tämän vas-

1. Käskykannalla tarkoitetaan prosessin sisältämien operaatioiden ketjua, jotka syötetään suorittimelle.

takohta on SIMD-prosessi (*Single Instruction, Multiple Data*), missä isolle määrälle dataa suoritetaan sama operaatio. Esimerkiksi rasteroinnissa jokaiselle näkymässä olevalle pinnalle suoritetaan sama käskykanta, mikä on tehokasta laskea rinnakkain näytönohjaimella, missä on useita laskevia ytimiä ja operaatioita ketjuttavia säikeitä (engl. *threads*). Rinnakkain suoritettavien säikeiden rypästä kutsutaan SIMT-yksiköksi (*Simple Instructions, Multiple Threads*).

Parker ym. (2013) kuitenkin huomauttavat, että yksittäisen säteen prosessin vaihtuminen on kuitenkin väliaikaista, sillä säteelle suoritettava prosessi palaa jokaisen osuman jälkeen säteen ja lähimmän pinnan leikkauspisteen etsimiseen. Tällöin säteenseurannan suorittaminen rinnakkaislaskennalla on ainakin osittain mahdollista.

OptiX hoitaa rinnakkaislaskentaa jakamalla seurattavat säteet säikeisiin ja tallentamalla niihin kuorman (engl. *payload*), johon tallennetaan mm. säteen väri, uudelleen lähetetyt säteet sekä säteen tila. Säteen tilalla tarkoitetaan kyseiselle säteelle seuraavaksi suoritettavaa prosessia, joita kuvataan säteenseurantaputkessa.

Koska säteitä suoritetaan rinnakkain useita säikeitä sisältävässä SIMT-yksikössä, voidaan kyseiselle yksikölle suorittaa vain yksi prosessi kerrallaan. Suoritusvuorossa oleva prosessi voi kuitenkin vaihtua hyvinkin nopeasti säteiden osuessa eri kappaleisiin. Välttääkseen turhaa käskykannan muuttamista, OptiX ajastaa prosessit siten, että kaikki samassa tilassa olevat säikeet ajetaan samanaikaisesti (Parker et al. 2013). Säikeet, jotka eivät vaadi kyseistä prosessia, odottavat oikean prosessin vaihtumista. Nopeuttaakseen prosessien vaihtamista, OptiX suosii samankaltaisten prosessien suorittamista peräjälkeen.

Prosessien ajoitus pyrkii pitämään mahdollisimman monta säiettä samassa tilassa, jotta niiden rinnakkainen prosessointi olisi mahdollisimman tehokasta. Koska leikkauspisteiden etsiminen on säteen todennäköisin tila, muuttuu säteenseuranta hieman lineaarisemmaksi.

Jotta yllä kuvattu prosessien aikataulutus olisi mahdollista, täytyy prosessit täsmäkääntää (engl. *Domain-Specific Compilation*), mikä tarkoittaa käyttäjän kirjoittaman

koodin kääntämistä tiettyyn tarkoitukseen suunniteltuun konekieleen. OptiX kääntää käyttäjän kirjoittaman koodin näytönohjaimelle suunnitelluiksi PTX-funktioiksi (Parallel Thread Execution) analysoidessaan näkymähierarkiaa (Parker et al. 2013). Funktioita pyritään lisäksi optimoimaan siten, että samaa funktiota pystyttäisiin käyttämään useaan kappaleeseen, jolloin säikeiden suoritustilan vaihtaminen vähentyisi.

5 Yhteenveto

Onko säteenseuranta sitten aivan liian työläs renderointimenetelmä, että sitä pystyttäisiin käyttämään reaaliajassa? On, ainakin toistaiseksi. Tietokoneiden laskentateho kasvaa jatkuvasti, joten kukaan ei voi tietää, mihin tulevaisuuden näytönohjaimet ja grafiikkasuorittimet pystyvät. Säteenseuranta on kuitenkin niin vaativa menetelmä, että kestää vielä tovin, että sitä voidaan käyttää reaaliaikaiseen renderointiin ilman jälkikäsittelyä tai muita apukeinoja.

Syynä tähän on yksinkertaisesti säteenseurannan tuoma vaativuuden uusi ulottuvuus: säteiden määrä. Rasteroinnissa huolenaiheena on ollut renderoitävien kappaleiden määrä, mikä ei nykyaikaisilla näytönohjaimilla ja käytettävissä olevalla muistilla ole juurikaan enää ongelma. Säteitä seuratessa kuitenkin kohdataan ongelma, kun heijastuneiden säteiden tai heijastumisten maksimimäärää kasvatetaan.

Esimerkiksi Chaitanyan ym. käyttämässä säteenseurannassa säteen annetaan kimmota osumasta vain *kerran* ja tuostakin osumasta lähetetään vain *yksi* seurattava säde. Yhteensä neljän säteen laskeminen pikseliä kohden vie nykyisen näytönohjainsukupolven äärirajoille. Realistisen kuvan renderoiminen ilman huomattavaa rakeisuutta vaatii satoja näytteitä jokaisesta leikkauspisteestä, joka sekä lisää laskettavien säteiden määrää, myös täyttää käyttömuistin nopeasti.

Muuta ratkaisua, kun rinnakkaislaskennan nopeuttaminen sekä muistin määrän kasvattaminen ei juuri ole. Tämän takia onkin keskitytty enemmän jälkikäsittelyyn ja säteenseurannan käyttämiseen yksittäisiin grafiikkaa parantaviin yksityiskohtiin, kuten realistisiin heijastuksiin tai valaistukseen. Reaaliaikaisen renderoinnin näkökulmasta säteenseurantaa ei kuuluisikaan käsittellä renderointimenetelmänä, vaan lähinnä apukeinona, jolla pystytään resurssien mukaan tuomaan kuvaan lisää realistisuutta. Vaikkei säteenseuranta ehkä vielä ole syrjäyttämässä rasterointia, on ensimmäiset askeleet siihen suuntaan jo otettu.

Kirjallisuusluettelo

Aila, Timo, ja Samuli Laine. 2009. "Understanding the efficiency of ray traversal on GPUs". Teoksessa *Proceedings of the conference on high performance graphics 2009*, 145–149. ACM.

Bishop, Gary, ja David M Weimer. 1986. "Fast phong shading". Teoksessa *ACM SIGGRAPH Computer Graphics*, 20:103–106. 4. ACM.

Chaitanya, Chakravarty R Alla, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai ja Timo Aila. 2017. "Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder". *ACM Transactions on Graphics (TOG)* 36 (4): 98.

Gouraud, Henri. 1971. "Continuous shading of curved surfaces". *IEEE transactions on computers* 100 (6): 623–629.

Hughes, John F., Akeley Kurt, Steven K. Feiner, James D Foley, Morgan McGuire, David Sklar ja Andries van Dam. 2013. *Computer Graphics: Ray Casting and Rasterization.* Luku 15. Boston, Massachusetts: Addison-Wesley Professional.

Möller, Tomas, ja Eric Haines. 2002. *Real-Time Rendering*. Natick, Massachusetts: AK Peters.

Möller, Tomas, ja Ben Trumbore. 2005. "Fast, minimum storage ray/triangle intersection". Teoksessa *ACM SIGGRAPH 2005 Courses*, 7. ACM.

Parker, Steven G, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys et al. 2013. "GPU ray tracing". *Communications of the ACM* 56 (5): 93–101.

Pharr, Matt, Jacob Wenzel ja Greg Humphreys. 2010. *Physically Based Rendering: From Theory to Implementation.* Burlington, Massachusetts: Morgan-Kaufmann.

Phong, Bui Tuong. 1975. "Illumination for computer generated pictures". *Communications of the ACM* 18 (6): 311–317.

Salomon, David. 2011. *The Computer Graphics Manual.* London: Springer.

Schied, Christoph, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn ja Marco Salvi. 2017. "Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination". Teoksessa *Proceedings of High Performance Graphics*, 2. ACM.

Wald, Ingo, Philipp Slusallek, Carsten Benthin ja Markus Wagner. 2001. "Interactive rendering with coherent ray tracing". Teoksessa *Computer graphics forum*, 20:153–165. 3. Wiley Online Library.

Whitted, Turner. 1979. "An improved illumination model for shaded display". Teoksessa *ACM SIGGRAPH Computer Graphics*, 13:14. 2. ACM.