Henri Pirinen

GPGPU-säteenseuranta

Tietotekniikan
kandidaatintutkielma
20. joulukuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Henri Pirinen
Yhteystiedot: henri.pirinen@jyu.fi
Työn nimi: GPGPU-säteenseuranta
Title in English: GPGPU ray tracing
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 26
Tiivistelmä: Säteenseuranta on rinnakkaistuva ja laskentaintensiivinen tapa tuottaa kolmiulotteista tietokonegrafiikkaa. Yleiskäyttöiset grafiikkaprosessorit (GPGPU) ovat tehokkaita rinnakkaislaskentaprosessoreita, joiden avulla voidaan kiihdyttää säteenseurantaa. Tässä tutkielmassa käsitellään säteenseurannan toteuttamista yleiskäyttöisillä grafiikkaprosessoreilla ja esitetään rakenne yksinkertaiselle GPGPU-säteenseurantaohjelmalle. Käsittelyn aiheena ovat myös säteenseurantaa kiihdyttävien menetelmien, kuten kiihdytysrakenteiden, toteuttaminen GPGPU--laskennalla.
Abstract: Ray tracing is a parallel and computationally intensive way of producing three dimensional computer graphics. General-purpose graphics processing units (GPGPU) are powerful parallel processors that can be utilized for accelerating ray tracing. This bachelor's thesis discusses about implementing ray tracer on GPGPU and presents a structure for a simple GPGPU ray tracer. This thesis also discusses about GPGPU implementation of methods, such as acceleration structures, that can be used to accelerate ray tracing.
Avainsanat: GPGPU, GPU, säteenseuranta
Keywords: GPGPU, GPU, ray tracing, raytracing

Sisältö

1 Johdanto

Säteenseuranta on valonsäteiden kulkua simuloiva tapa tuottaa kolmiulotteista tietokonegrafiikkaa. Sen avulla pystytään tuottamaan helposti reaalimailman läheisiä valaistusefektejä, joiden toteuttaminen on hankalampaa muiden menetelmien avulla. Ongelmana on säteenseurannan vaatima suuri laskentateho, kun jokaista lähtetettävää sädettä kohti täytyy tarkistaa törmäykset jokaisen avaruuden kappaleen kanssa ja säteitä on lähetettävä ainakin yksi jokaista ruudun pikseliä kohti. (Cook, Porter, & Carpenter, 1984)

Säteenseuranta-algoritmit ovat sopivat hyvin laskettavaksi rinnakkain. Jokainen lähetettävä säde on oma erillinen laskutehtävänsä, joka ei riipu muista säteistä. Tästä syystä säteenseuranta on nykyaikaisten moniytimisten prosessorien valossa erittäin kiinnostava tutkimuksenaihe. (Nery, Nedjah, Franca, & Jozwiak, 2011)

Nykyaikaiset *grafiikkaprosessorit* (GPU) mahdollistavat tehokkaan rinnakkaislaskennan ja niiden laskentateho kasvaa hurjaa vauhtia *keskusprosessorien* (CPU) laskentatehoon nähden. Alun perin grafiikkaprosessorien toiminta oli rajoitettu tiettyihin valikoituihin tehtäviin, mutta niiden hurjan laskutehon kehittymisen myötä, grafiikkaprosessoreista tehdään nykyään ns. *yleiskäyttöisiä grafiikkaprosessoreita* (GPGPU) (Owens et al., 2007).

Tuntuisi siltä, että säteenseurannan laskeminen olisi tehokasta grafiikkaprosessorien avulla niiden rinnakkaislaskentaominaisuuksien vuoksi. Ovatko kuitenkaan grafiikkaprosessorien yleiseen laskentaan kehitetyt ominaisuudet ja rajapinnat tarpeeksi kehittyneitä säteenseurannan tehokkaaseen toteuttamiseen? Millaisia ongelmia säteenseuranta-algoritmien toteuttamisessa voisi olla?

Tutkielman toinen luku käsittelee säteenseurannan perusteita. Sen tarkoituksena on tuoda säteenseurannan perusperiaatteet lukialle tutuksi. Lisäksi toinen luku käsittelee yleisellä tasolla säteenseurannan kiihdyttämiseksi käytettäviä tietorakenteita ja menetelmiä, sekä säteenseurannan rinnakkaistumista.

Kolmas luku keskittyy nykyaikaisten grafiikkaprosessorien arkkitehtuuriin, toimintaan ja hyödyntämiseen rinnakkaislaskennassa. Luvussa selvitetään mitä etuja GPGPU-laskennasta on CPU-laskentaan nähden ja millaisiin tehtäviin se soveltuu.

Neljäs luku käsittelee säteenseurantaa GPGPU-laskennan näkökulmasta ja esittelee tavan toteuttaa GPGPU-säteenseurantaohjelma. Se palaa toisessa luvussa esitettyihin säteenseurantamenetelmiin ja esittää kuinka nämä menetelmät voitaisiin toteuttaa, niiden toimintaa tehostaen, GPGPU-laskennan avulla. Luvussa pohditaan myös eri menetelmien hyötyjä ja haittoja.

2 Säteenseurannan perusteet

2.1 Yleistä

Säteenseuranta perustuu valonsäteiden kulun seuraamiseen avaruudessa. Sen avulla voidaan muodostaa kuva seuraamalla valonsäteitä, jotka osuvat virtuaalisen kameran kennolle (viewport). Valonsäteiden seuraaminen valonlähteistä kameraan on työlästä, koska niiden kulku on vaikeasti ennustettavissa. Tästä syystä valonsäteiden seuraaminen tehdään yleensä käänteisessä järjestyksessä seuraamalla säteitä kameran kennolta valonlähteisiin. (Kuchkuda, 1988)

Monet reaalimaailmassa esiintyvät valoilmiöt on helppo toteuttaa säteenseurannalla. Säteenseuranta on helposti ymmärrettävä ja yksinkertainen säteenseurantaohjelmisto on nopea toteuttaa, mutta sen suurin ongelma on laskennan hitaus. Miljoonien säteiden laskeminen on raskasta, varsinkin piirrettävien näkymien kompleksisuuden kasvaessa. (Kuchkuda, 1988)

Raskautensa vuoksi säteenseurantaa käytetään erityisesti sovelluksissa, joissa piirtämisen ei tarvitse olla tarpeeksi nopeaa interaktiiviseen työskentelyyn. Hyvä esimerkki säteenseurannan käyttökohteesta on elokuvateollisuus, jossa halutaan tuottaa realistisia 3d-grafiikkaefektejä ja piirtäminen tarvitsee tehdä vain kerran, jolloin piirtonopeus ei ole merkityksellinen tekijä. (Purcell, 2004)

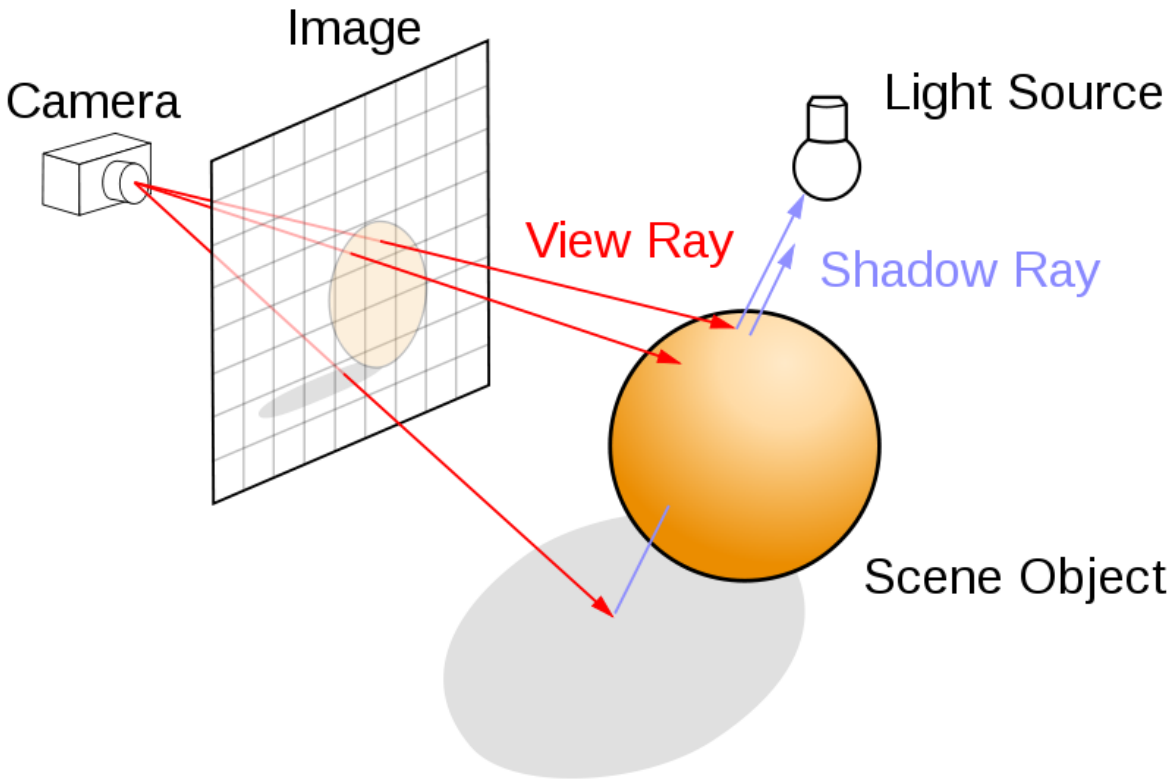
2.2 Säteiden lähettäminen

Säteenseuranta perustuu *säteiden lähettämiseen* (Ray Casting) eli säteiden ja avaruuden kappaleiden välisten törmäysten laskentaan (A.Carr, Hall, & Hart, 2002). Säteellä tarkoitetaan tietorakennetta, joka merkitsee lähtöpisteen ja suunnan. Sitä voisi kuvata suorana, joka lähtee lähtöpisteestä ja jatkuu äärettömyyteen suuntavektorin määrittämään suuntaan.

Ensimmäiset säteet lähetetään kameran sijainnista jokaisen virtuaalisen kameran kennon eli muodostuvan bittikartan pikelin läpi (Kuchkuda, 1988). Esimerkiksi 640x480 kuvaa varten tutkitaan 307200 säteen törmäykset avaruuden kappaleiden kanssa (Kuchkuda, 1988). Kuvassa 1 näkyy kuinka kolme sädettä lähetetään kamerasta eri kuvan pikseleiden läpi.

Säteen törmäyskohdan etsimisessä on löydettävä lähin avaruuden kappale johon säde törmää, joka edellyttää kaikkien avaruuden kappaleiden läpikäymistä. Kappaleiden läpikäymistä voi nopeuttaa lajittelemalla ne tietorakenteeseen, jonka avulla voidaan karsia törmäysten tarkistamiseen tarvittavia laskuja. (Kuchkuda, 1988)

Kun lähin törmäys on löydetty, lähetetään törmäyskohdasta sekundaarisia sätei-



Kuva 1: Säteenseurantaohjelman toiminta
Lähde: en.wikipedia.org/wiki/File:Ray_trace_diagram.svg

tä. *Varjosäteet* (shadow ray) lähetetään kohti jokaista valonlähdettä, kuten kuvasta 1 näkee. Jos varjosäde osuu johonkin muuhun kappaleeseen ennen valonlähdettä, tiedetään, että valonlähde ei suoranaisesti valaise kappaletta. Muita sekundaarisia säteitä voidaan lähettää törmäyskohdasta esimerkiksi, jos kappaleen materiaali heijastaa valoa tai se on läpinäkyvä. Nämä säteet käsitellään samalla tavalla kuin ensisijaiset säteet. (Kuchkuda, 1988)

Kuvan pikseleiden lopulliset väriarvot lasketaan yhdistämällä säteiden ja niistä lähteneiden sekundaaristen säteiden törmäyskohdista saadut väriarvot.

Listauksessa 1 on esimerkki rekursiivisesta säteenseuranta-algoritmista. Alussa funktio traceRay kutsuu funktiota findClosestIntersection, joka palauttaa lähimmän törmäyksen säteen kanssa. Jos yhtään törmäystä ei löydy funktio traceRay palauttaa mustan värin. Algoritmissa värit voisivat esimerkiksi olla kolmiulotteisia vektoreita, jotka sisältävät punaisen, vihreän ja sinisen väriarvon.

Rivillä 6 kutsutaan castShadowRays-funktiota, joka laskee törmäyskohdan valaistuksen ja palauttaa reaaliluvun, jolla kerrotaan törmäyskohdan materiaalin väri sen varjostamiseksi. Jos törmäyskohdan materiaali on heijastava, rivillä 9 lähetetään

sekundaarinen säde kutsumalla traceRay funktiota rekursiivisesti ja sen palauttama väriarvo kerrotaan heijastuksen voimakkuudella ja se lisätään säteen lähettämisestä saatuun väriarvoon.

Listing 1: Yksinkertainen rekursiivinen säteenseuranta-algoritmi

```
1  function traceRay(ray):
2    isct = findClosestIntersection(ray)
3    if isct == NULL:
4      return BLACK
5
6    color = isct.color * castShadowRays(isct.normal) * (1-isct.reflect)
7
8    if isct.reflect > 0:
9      color += traceRay(isct.reflect_ray) * isct.reflect
10
11   return color
```

2.3 Rinnakkaistaminen ja skaalautuvuus

Jokaisen ruudun pikselin laskeminen on säteenseurannassa oma tehtävänsä (Segovia, Li, & Gao, 2009). On täysin toisistaan riippumatonta missä järjestyksessä pikseleiden väriarvot lasketaan.

Myös jokaisen säteen lähettäminen voidaan rinnakkaistaa. Jos yhtä pikseliä kohti lähetetään enemmän kuin yksi säde, on säteiden törmäyslaskuista saadut tulokset riippuvaiset toisiensa suhteen. Ennen kuin lopullinen väriarvo voidaan approksimoida, on seurattava jokaista sädettä loppuun asti. (Segovia et al., 2009)

Kun säteenseuranta perustuu saman hyvin rinnakkaistuvan tehtävän ajamiseen monta kertaa, se on erittäin hyvin skaalautuva tehtävä. Säteenseurantatehtävään voidaan lisätä uusia efektejä vain lähettämällä lisää säteitä.

Jos säteenseurannassa lähetetään edellisen kuvatun mallin mukaan jokaista pikseliä kohti yksi säde, lopputulos sisältää sahalaitoja. Reunojen pehmentämiseksi on lähetettävä yhtä pikseliä kohti useampi säde ja approksimoitava pikselin värin niiden pohjalta. Esimerkiksi pikseliä kohti voitaisiin lähettää neljä sädettä, joista jokainen kulkee pikselin kulmien läpi. (Kuchkuda, 1988)

Samalla periaatteella säteenseurannassa voidaan totetutaa muitakin efektejä, kuten pehmeät varjot. Yhden varjosäteen lähettäminen tekee varjojen reunoista suorat, mutta usella varjosäteellä voidaan approksimoida varjon tummuus reunakohdissa.

Tietenkin lisää säteitä lähettämällä myös tarvittava laskentatehon määrä kasvaa. Samoin laitteistoa muuttamalla on helppo lisätä laskutehoa. Esimerkiksi rinnakkai-

sen säteenseurantaohjelman ajaminen nopeutuu lähes lineaarisesti, kun sitä ajavaan laitteistoon lisätään prosessoriytimiä (Römisch, 2009).

2.4 Säteenseurannan nopeuttamisessa käytettäviä menetelmiä

Suurin osa säteenseurantaprosessista on säteiden ja avaruuden kappaleiden törmäysten laskemista. Olisi siis loogista minimoida tarvittavien törmäysten laskeminen. Tässä alaluvussa esitellään tietorakenteita, joiden avulla voidaan vähentää laskettavia törmäyksiä jakamalla avaruus osiin ja jaon avulla määrittää avaruuden osat joiden läpi säde kulkee. Näin voidaan jättää laskematta törmäykset kappaleiden kanssa, jotka eivät kuulu niihin avaruuden osiin, joidenka läpi säde kulkee. Lisäksi tässä aliluvussa esitellään ray packets-menetelma, jonka avulla voidaan vähentää laskettavien törmäysten määrää käsittelemällä lähekkäiset säteet yhdessä.

Yksikköruudukon (uniform grid) ideana on jakaa avaruus yksikkökuutioihin eli vokseleihin (voxel, volumetric pixel). Tietorakenteen läpikäymisalgoritmina toimii teoriassa viivan rasteroimisalgoritmit, jotka voidaan yleistää kolmeen ulottuvuuteen. Esimerkiksi DDA-viivanrasterointialgoritmi sopii tähän tehtävään. (Amanatides & Woo, 1987)

Yksikköruudukko on yksikertaisimpia tietorakenteita säteensuerannan kiihdyttämiseksi. Sen konstruointi on helppoa, sillä avaruus on itsessään jo jaettu vokseleihin koordinaattejen mukaan. Ohjelman täytyy vain merkitä jokaiseen vokseliin kuuluvat kappaleet esimerkiksi lisäämällä jokaista vokselia kohti lista, joka sisältää osoittimet vokselin sisällä oleviin kappaleisiin.

Kahdeksanpuu (octree) jakaa avaruuden rekursiivisesti kahdeksaan osaan. Jako muodostaa puurakenteen, jossa jokaisella puun solmulla on joko kahdekstan tai ei yhtään lasta. Kahdeksanpuun konstruoiminen onnistuu esimerkiksi jakamalla aluksi avaruus kahdeksaan osaan ja tutkimalla mitkä avaruuden kappaleet kuuluvat mihinkin osioon. Sen jälkeen jokaisen kahdeksan osaa kohti suoritetaan edellista jakoalgoritmia rekursiivisesti, kunnes jokainen kappale on omassa puun lehtisolmussa tai valittu puun maksimisyvyys ylittyy. (Römisch, 2009) Kuvassa 2 on kuvattu nelipuu. Se toimii samalla tavalla kahdessa ulottuvuudessa kuin kahdeksanpuu kolmessa. Nelipuu jakaa avaruuden jokaisella jaolla neljään osaan.

K-d puu (kd-tree, k-d tree, k-dimensional tree) on binääripuu, jota käytetään avaruuden jakamiseen (Römisch, 2009). Avaruus jaetaan rekursiivisesti kahteen

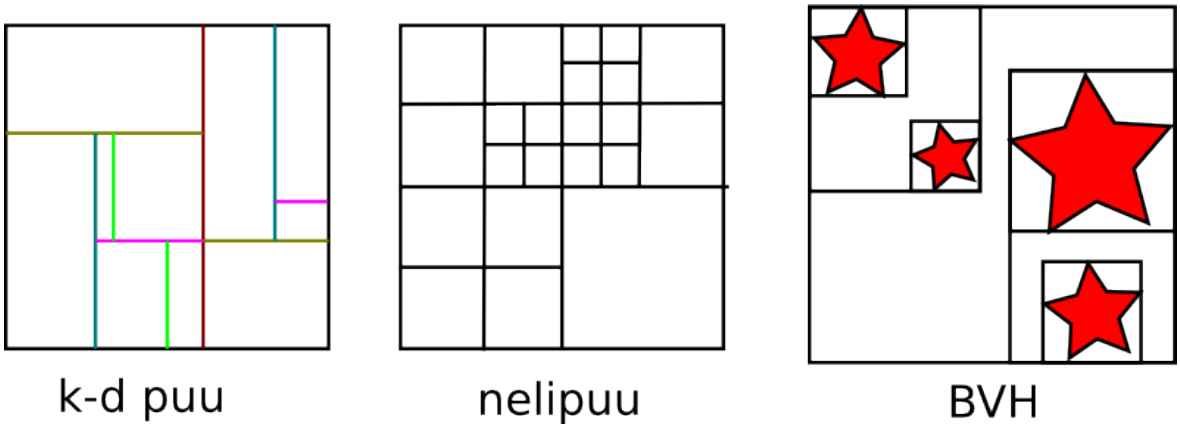
osaan eri avaruuden akseleiden suhteen. Jakojen akselit määritellään vuorotellen siten, että jokainen k-d puun tason solmu jakaa avaruuden samalla akselilla (Römisch, 2009). Jakokohta voi kuitenkin vaihdella tason solmuissa (Römisch, 2009). Jakokohdan määrittämiseen on erilaisia menetelmiä, kuten esimerkiksi SAH (surface area heuristic) metodi (Wald, 2004). Kuvan 2 k-d puu on kaksiulotteinen eli sen jaot tapahtuvat kahdella akselilla. Kuvassa eri tason jaot on kuvattu eri värillä.

BVH (Bounding volume hierarchy) on puurakenne, joka kapseloi rekursiivisesti avaruuden kappaleet yksinkertaisten geometristen muotojen sisälle. Esimerkiksi kapseloinnissa voidaan käyttää palloja tai kuusitahokkaita. Jokainen puun sisäsolmu kuvaa yhtä geometrista muotoa, joka sisältää lapsisolmuja eli muita avaruuden kappaleita kapseloivia kappaleita. Lehtisolmut sisältävät yhden tai useamman avaruuden kappaleen. BVH tulisi konstruoida siten, että lähekkäiset lehtisolmut sisältäisivät avaruudessa lähekkäitä kappaleita. (Lauterbach, Yoon, Tuft, & Manocha, 2006) Kuvan 2 BVH kapseloi kaksiulotteisessa avaruudessa olevat tähdet.

Puurakenteiden läpikäyminen voidaan toteuttaa saman periaatteen mukaan. Säteen leikkaus kiihdytysrakenteen juurisolmun kanssa muodostaa suoran (min, max), jossa min on säteen lähdettä lähempi piste ja max on kauempi piste (Horn, Sugerman, Houston, & Hanrahan, 2007). Läpikäyminen tapahtuu tutkimalla pistettä min lähinnä oleva juurisolmu ja siirtämällä pistettä min lähemmäksi pistettä max, niin kauan kunnes törmäys löytyy tai suoran pituus on nolla (Horn et al., 2007).

Puurakenteiden läpikäymisalgoritmissa solmujen läpikäyminen voidaan esimerkiksi toteuttaa rekursiolla. Tällöin jokaisen sisäsolmun kohdalla kutsutaan läpikäymisalgoritmia rekursiivisesti ja annetaan parametreiksi säteen leikkaaman lapsisolmun leikkaussuora.

Sädepakettejen (Ray packets) avulla voidaan vähentää avaruuden kappaleiden kanssa laskettavia törmäyksiä käsittelemällä monia lähekkäisiä säteitä yhdessä. Erityisesti ensimmäiset lähetetyt säteet kulkevat lähekkäin lähes samaa reittiä samojen kiihdytysrakenteen alkioiden läpi. Esimerkiksi joukosta säteitä voidaan muodostaa katkaistu kartio, jonka leikkaukset kiihdytysrakenteen alkioiden kanssa tarkastetaan. Jos kartio ei seuraa yhtä reittiä kiihdytysrakenteessa, jaetaan säteet uusiin sädepaketteihin tai seurataan säteitä erikseen. (Reshetov, Soupikov, & Hurley, 2005)



Kuva 2: Kaksiulotteisia versioita kiihdytysrakenteista

3 Yleinen laskenta grafiikkaprosessoreilla

Tämä luku käsittelee grafiikkaprosessoriteknologian historiaa, sekä nykyaikaisen grafiikkaprosessorien laskentaominaisuuksia. Ensimmäinen aliluku keskittyy grafiikkaproessorienteknologian historiaan, ja sen kehittymisen syihin ja vaiheisiin. Toinen aliluku esittelee nykyaikaisen grafiikkaprosessorin toimintaan, sekä arkkitehtuuriin. Kolmas aliluku käsittelee GPGPU-laskentarajapintoja yleisesti, sekä ottaa kantaa niiden eroavaisuuksiin. Viimeinen aliluku tutustuu CUDA-rajapintaan.

3.1 Grafiikkaprosessoriteknologian historiaa

Viimeisen vuosikymmenen aikana grafiikkaprosessorien laskentateho on kasvanut hurjasti keskusprosessoreihin nähden. Grafiikkaprosessorit koostuvat sadoista prosessoriytimistä, jotka mahdollistavat erittäin tehokkaan rinnakkaislaskennan. Tästä syystä grafiikkaprosessorit pystyvät laskemaan keskusprosessoreja suuremman määrän liukulukulaskutoimituksia sekunnissa, vaikka keskusprosessorien kellonopeudet ovat selvästi grafiikkaprosessoreita suurempia. Kuitenkaan suurta laskentanopeutta ei voida hyödyntää, jos laskentatehtävää ei voida laskea rinnakkain grafiikkaprosessorin prosessoriytimillä. Hyvin rinnakkaistuvat tehtävät ovat sopivia laskettavaksi grafiikkaprosessorilla, kuin taas peräkkäin laskettavat tehtävät keskusprosessoreilla. (Owens et al., 2008)

Alun perin grafiikkaprosessoreiden toiminta oli *grafiikkaputken* (graphics pipeline) mukaan ennalta määriteltyä ja niiden tehtävänä oli kiihdyttää rasteroidun grafiikan piirtämistä. Grafiikkaputki on malli tehtävistä ja tehtävien järjestyksestä, joita suoritettiin grafiikkaprosessorilla. Kuitenkin grafiikan piirtämiseen haluttiin enemmän kontrollia, jonka takia grafiikkaprosessorien toimintaan lisättiin ohjelmoitavia

osia, ns. *varjostinohjelmia* (shader program). Aluksi varjostinohjelmia oli kahdenlaisia: vertex-varjostinohjelmia, jotka tekevät operaatioita jokaista piirrettävän monikulmion kulmapisteitä kohti, sekä fragment-varjostinohjelmia, jotka tekevät operaatioita jokaiselle kuvan pikselille. Varjostinohjelmien avulla pystyy muokkaamaan grafiikkaputken toimintaa, ja luomaan täysin uudenlaisia efektejä rasteroituun grafiikkaan. (Owens et al., 2008)

Vaikka grafiikkaprosessoreilla pystyi suorittamaan omaa koodia, sen ajaminen oli sidottu grafiikkaputkeen (Owens et al., 2008). Ainoa tapa ajaa omaa koodia grafiikkaprosessorilla oli käyttää OpenGL- tai Direct3D-rajapintoa, jotka eivät pelkästään olleet ohjelmallisia elementtejä, vaan niiden toiminta oli määritelty grafiikkaprosessorien raudassa asti (Kirk & Hwu, 2009). Varjostinohjelmien käskykannat olivat erillisiä, ja molemmille varjostinohjelmille oli omat laskentayksiköt. (Kirk & Hwu, 2009). Edellisistä syistä grafiikkaproessorien käyttäminen muuhun kuin grafiikan piirtämiseen oli hankalaa, joka hidasti yleiskäyttöisen GPU-laskennan yleistymistä. (Kirk & Hwu, 2009).

Iso askel kohti yleiskäyttöistä GPU-laskentaa oli varjostinohjelmien käskykantojen yhdistäminen. Sen sijaan, että eri varjostinohjelmien ajamista varten olisi ollut omat prosessoriytimet, niiden ajaminen toteutettiin käyttämällä samoja yleiskäyttöisiä prosessoriytimiä. Yleiskäyttöiset prosessoriytimet mahdollistivat prosessorin tehojen jakamisen eri varjostinohjelmien kesken paremmin, sillä tarvittava laskentateho ei välttämättä jakaannu tasaisesti niiden kesken. (Kirk & Hwu, 2009)

Nykyaikaisten grafiikkaprosessorien arkkitehtuuri mahdollistaa ohjelmakoodin ajamisen grafiikkaputken ohi. GPU-laskentarajapinnat ovat viime vuosina kehittyneet nopeasti ja uudet ominaisuudet ovat tehneet oman ohjelmakoodin suorittamisesta helpompaa ja tehokkaampaa grafiikkaprosessoreila. Grafiikkaprosessoreille on tullut täysin uudenlaisia käyttökohteita yleisen GPU-laskennan myötä, muun muassa tieteellisessä laskennassa. (Owens et al., 2008)

3.2 Nykyaikaisten grafiikkaprosessorien arkkitehtuuri

Virtaprosessori (streaming processor, SP) on grafiikkaprosessorin pienin laskentayksikkö. Yhdessä grafiikkaprosessorissa voi olla satoja virtaprosessoreja, jotka sijaitsevat *monivirtaprosessoreissa* (streaming multiprocessror, SM). Monivirtaprosessorien tehtävä on suorittaa *ydinohjelmia* (kernel program) jakamalla laskenta virtaprosessorien kesken. Jokaisella monivirtaprosessorilla on oma välimuisti ja rekisterit, jotka jaetaan virtaprosessorien käytettäväksi. (Kirk & Hwu, 2009)

Ydinohjelmat ovat grafiikkaprosessorin käskykannalle kirjoitettuja ohjelmia, joi-

ta virtaprosessorit ajavat. Kaikissa nykyaikaisissa grafiikkaprosessoreissa ei ole mahdollisuutta suorittaa aliohjelmakutsuja ydinohjelmissa, joten tässä tutkielmassa oletetaan, että se ei ole mahdollista. Tällöin ei ole siis mahdollista tehdä rekursiivisia ydinohjelmia. (Kirk & Hwu, 2009)

Grafiikkaproesssori koostuu useista monivirtaprosessorista, sekä yleisestä muistista. Yleinen muisti toimii tiedonsiirtokanava isäntälaitteen ja näytönohaimen, sekä eri monivirtaprosessorien välillä. Grafiikkaprosessorien muistit ovat erittäin nopeita verrattuna tietokoneen keskusmuistiin, jotta tiedonsiirto isäntäkoneelta grafiikkaprosessorille, sekä grafiikkaprosessorin muistejen välillä olisi mahdollisimman nopeaa. (Kirk & Hwu, 2009)

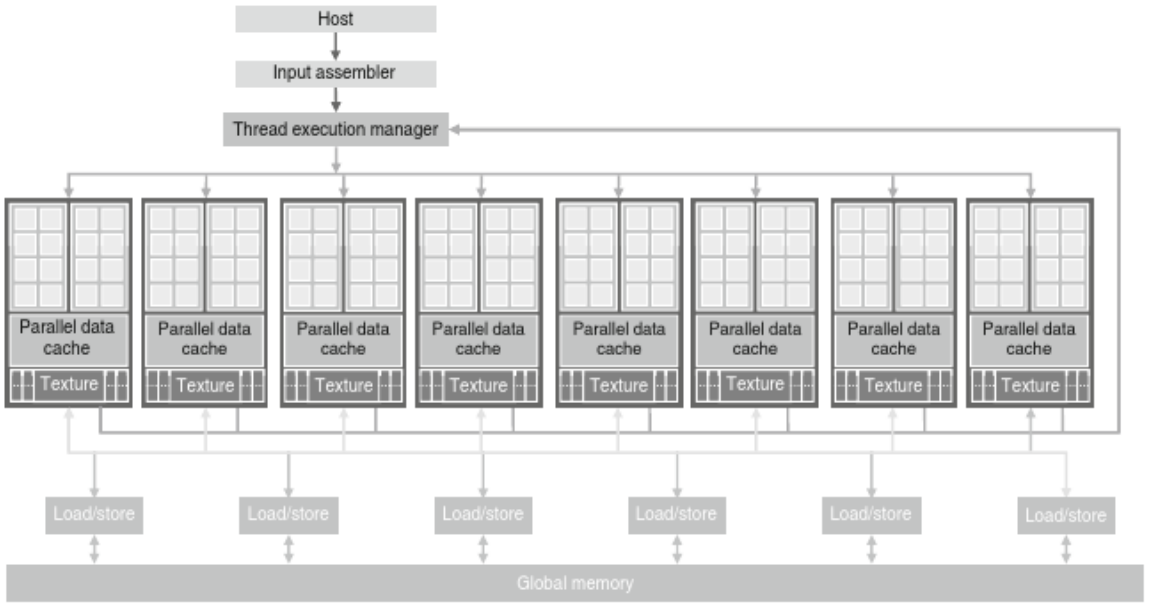
Monivirtaprosessorit pystyvät siirtämään tietoa yleisen muistin ja oman välimuistinsa välillä. Paras ratkaisu on käyttää mahdollisimman paljon monivirtaprosessoreiden välimuistia, koska se on vielä nopeaampaa kuin lukeminen ja kirjoittaminen yleiseen muistiin. Pelkän välimuistin käyttäminen ei ole mahdollista, koska isäntäkone ei voi siirtää tietoa suoraan monivirtaprosessorien välimuistiin ja välimuistit ovat paljon pienempiä kuin grafiikkaprosessorin yleinen muisti. (Kirk & Hwu, 2009)

Grafiikkaprosessorit on suunniteltu siten, että ne pystyvät käynnistämään ydinohjelmien suorittamisen mahdollisimman nopeasti. Näin nykyaikaisten grafiikkaprosessorien tehoista saa parhaiten tehoja irti jakamalla ohjelman mahdollisimman moneen yksinkertaiseen ydinohjelmaan, koska silloin se pystyy jakamaan laskennan parhaiten virtaprosessorien kesken. (Kirk & Hwu, 2009)

Kuvassa 3 kuvataan nykyaikaisen grafiikkaprosessorin arkkitehtuuri. Kuvan grafiikkaprosessorissa on 128 virtaprosessoria, jotka on jaettu 16 virtaprosessorin joukkoihin kahdeksaan monivirtaprosessoriin. Jokaisessa monivirtaprosessorissa sijaitsee myös välimuisti, sekä erityisesti tekstuureita varten tarkoitettu tekstuurivälimuisti. Kuvassa näytetään myös kuinka data liikkuu eri muistejen välillä.

3.3 GPGPU-rajapinnat

Isäntäjärjestelmän ja grafiikkaprosessorin välinen tiedonsiirto tapahtuu GPGPU-rajapinnan avulla. Tällä hetkellä yleisimmät käytössä olevat kaksi GPGPU-laskentarajapintaa ovat OpenCL ja CUDA. CUDA on Nvidian kehittämä GPGPU-laskentarajapinta, jonka rajoite on, että se toimii vain Nvidian valmistamilla näytönohjaimilla. OpenCL on avoin rinnakkaisohjelmointistandardi, joka ei pelkästään rajoitu GPGPU-laskentaan, vaan se pystyy jakamaan tehtäviä esimerkiksi myös keskusprosessorin kanssa. (Karimi, Dickson, & Hamze, 2010)



Kuva 3: Nykyaikaisen grafiikkaprosessorin arkkitehtuuri. (Kirk & Hwu, 2009)

Pääpiirteiltään molemmat rajapinnat ovat samankaltaisia. Molemmat rajapinnat toteuttavat C-typpisen ohjelmointikielen, jonka avulla kirjoitetaan ydinohjelmia, sekä kääntäjät näille kielille (Kirk & Hwu, 2009). OpenCL pystyy kääntämään ydinohjelmia dynaamisesti, toisin kuin CUDA, jonka ydinohjelmien pitää olla esikäännettyjä (Karimi et al., 2010). Ohjelmien ajaminen tapahtuu alustalle natiivin ohjelman kautta kutsumalla rajanpinnan avulla ydinohjelmaa (Karimi et al., 2010).

Vaikka rajapinnat ovat samankaltaisia, CUDA:n ja OpenCL:n käsitteistö eroaa hieman. Tässä tutkielmassa käytetään CUDA:n käsitteistöä, sillä valtaosa lähdemateriaalista käsittelee CUDA-rajapintaa.

3.4 CUDA-arkkitehtuuri

CUDA-rajapinnassa *laite* (device) tarkoittaa yhtä grafiikkaprosessoriyksikköä. Laite sisältää *yleiskäyttöisen muistin* (global memory) ja *vakiomuistin* (constant memory). Yleiskäyttöinen- sekä vakiomuisti toimivat tiedonvälittäjinä isäntälaitteen ja näytönohjaimen *säikeiden* (thread) välillä. Molemmat voivat lukea tietoa näistä muisteista, sekä kirjoittaa tietoa yleiskäyttöiseen muistiin. (Kirk & Hwu, 2009)

Säikeet ovat CUDA-arkkitehtuurin prosesseja, joita suoritetaan virtaprosessoreilla. Yksi *lohko* (block) sisältää monta säiettä, jaetun muistin, sekä säikeiden kesken jaetut rekisterit. Lohkot suoritetaan siten, että jokainen lohkon säie ajetaan sa-

malla monivirtaprosessorilla. Tällöin jaettu muisti ja säikeiden rekisterit sijaitsevat monivirtaprosessorin välimuistissa. (Kirk & Hwu, 2009)

Jokaisella suoritettavalla lohkolla ja säikeellä on indeksi, jonka perusteella voidaan määrittää ohjelman käyttäytymistä. Säikeiden indeksit voivat olla moniulotteisia, joidenka avulla voidaan esimerkiksi toteuttaa operaatioita moniulotteisille taulukoille. (Kirk & Hwu, 2009)

Ydinohjelman ajaminen tapahtuu aluksi varaamalla ja siirtämällä laskentaan tarvittava tieto näytönohjaimen yleiseen muistiin. Tämän jälkeen voidaan ajaa ydinohjelma käynnistämällä se isäntäohjelmasta. Kun ydinohjelma on suoritettu kopioidaan laskutoimituksen tulos keskusmuistiin ja vapautetaan näytönohjaimen varattu muisti. (Kirk & Hwu, 2009)

4 Säteenseurannan toteuttaminen GPGPU-laskennan avulla

Tässä luvussa käsitellään säteenseurantaohjelmiston toteuttamista käyttäen apuna GPGPU-laskentaa. Ensimmäinen aliluku pohtii mitä hyötyjä säteenseurannan toteuttamisesta GPGPU-laskennan avulla on ja käsittelee GPGPU-laskennan avulla nopeutetun säteenseurannan käyttömahdollisuuksia. Toinen aliluku esittää mahdollisen rakenteen yksinkertaiselle GPGPU-säteenseurantaohjelmalle, pohtii sen toimivuutta ja esittää vaihtoehtoisia menetelmiä. Kolmas aliluku käsittelee sekundaaristen säteiden lähettämistä GPGPU-säteenseurannalla. Aliluvut 4, 5 ja 6 käsittelevät säteenseurannan kiihdytysmenetelmiä GPGPU-laskennalla. Viides aliluku käsittelee kiihdytysrakenteiden konstruointia, ja kuudes aliluku niiden läpikäyntiä.

4.1 GPGPU-säteenseurannan käyttömahdollisuuksia

Säteenseuranta on GPGPU-laskennan valossa erittäin kiinnostava tutkimuksenkohde. Sen rinnakkaistuvuus mahdollistaa laskennan suorittamisen tehokkaasti grafiikkaprosessoreilla, ja monet tutkijat ovat saaneet jopa GPGPU-säteenseurannalla moninkertaisia piirtonopeuksia pelkkään keskusprosessorilaskentaan nähden (Günther, Popov, Seidel, & Slusallek, 2007).

Perinteisesti säteenseurannan hitaus on mahdollistanut sen käyttämisen vain sovelluksiin, jotka eivät vaadi interaktiivista piirtonopeutta. Nykyaikaisilla keskusprosessoreilla on mahdollista päästä interaktiivisiin nopeuksiin säteenseurannassa (Römisch, 2009). GPGPU-laskennan avulla voidaan kuitenkin toteuttaa keskusprosessoreita nopeampia säteenseurantaohjelmia (Römisch, 2009). Nopeutusta voidaan tuoda epäinteraktiivisiin sovelluksiin, mutta ehkä kiinnostavampia GPGPU-

säteenseuranna tutkimuksenkohteita ovat interaktiivisella nopeudella toimivat ja reaaliaikaiset säteenseurantasovellukset.

Epäinteraktiivisella nopeudella toimivat säteenseurantasovellukset voivat piirtää kuvia määrittelemättömän pitkän aikaa. Yleensä näiden sovellusten tarkoitus on tuottaa staattisia kuvia 3d-näkymistä tai esipiirrettyä 3d-animaatiota. Kun grafiikkaprosessoriteknologia mahdollistaa nykyaikaisten piirtotekniikoiten nopeuttamisen, ei olisi ihme, jos esimerkiksi suurten elokuvastudioiden piirtofarmit käyttäisivät hyödyksi GPGPU-laskentaa.

Interaktiivinen sovellus tarkoittaa sovellusta, joka vastaa käyttäjän toimintoihin ilman odotusaikoja. Säteenseuranta interaktiivisessa ajassa mahdollistaa kolmiulotteisen datan esittämisen ja muokkaamisen sovelluksessa. Sen etuna rasteroituun grafiikkaan on mahdollisuus esittää erilaista grafiikkaa kuin pelkkiä polygonirakenteita (Römisch, 2009). Hyvänä esimerkkinä on pistepilvet, joita voidaan tuottaa 3d skannerilla (Römisch, 2009). Skannattua 3d-mallia voidaan käsitellä vokselidatana, joka antaa hyvän tavan visualisoida pistepilviä (Römisch, 2009).

Reaaliaikaiset sovellukset, kuten videopelit vaativat yli 30 kuvan ruudunpäivitysnopeuden sekunnissa. Reaaliaikaisella säteenseurannalla voidaan toteuttaa reaalimailmaa lähenteleviä valaistusefektejä paremmin kuin reaaliaikaisella rasteroinnilla (Meseth, Guthe, & Klein, 2005). Kuitenkin säteenseuranta on vielä selvästi hitaampaa, kuin rasterointi, joka voi olla syynä siihen, että se ei ole yleistynyt reaaliaikaisissa sovelluksissa.

Eräs GPGPU-säteenseurannan käyttökohde reaaliaikaisissa sovelluksissa voisi olla yhteiskäyttö rasteroinnin kanssa. Esimerkiksi säteenseurannalla voidaan toteuttaa todenmukaiset heijastusefektit rasteroiduille pintamateriaaleille. (Meseth et al., 2005)

4.2 GPGPU-säteenseurantaohjelman rakenne

Tässä aliluvussa pohditaan GPGPU-säteenseurantasovelluksen rakennetta. Tärkeä kysymys sovelluksen rakenteen suunnittelussa on kuinka tehtävät jaetaan keskusprosessorin ja grafiikkaprosessorin kesken. Lisäksi on mietittävä miten säteenseurannan tehtävät toteutetaan grafiikkaprosessorin ydinohjelmina, kuinka ne suoritetaan ja miten laskentaan tarvittava data siirretään grafiikkaprosessorille?

Jotta GPGPU-säteenseurantasovellus voitaisiin toteuttaa tehokkaasti on selvitettävä tehtävät, jotka ovat suoritettavissa rinnakkain. Jokaiselle rinnakkain suoritettavalle tehtävälle voidaan tehdä oma ydinohjelma grafiikkaprosessorilla laskettavaksi. Peräkkäin suoritettavat tehtävät kannattaa laskea keskusprosessorilla.

GPGPU-säteenseurantasovellus muodostuu useasta peräkkäin suoritettavasta ydinohjelmasta. Ydinohjelmat kutsutaan järjestyksessä isäntäjärjestelmästä. Ohjelman rakennetta voisi kuvata putkella, joka ottaa syötteeksi näkymän tiedot ja palauttaa valmiin kuvan. Tietojen siirtyessä putkessa eteenpäin, niille suoritetaan putkessa olevat operaatiot järjestyksessä. (Britton, 2010)

Käsitellään aluksi yksinkertaista säteenseurantaa, joka ei sisällä sekundaarisia säteitä. Tällaisessa tilanteessa ruudun pikseleiden lopullinen väriarvo saadaan laskettua, kun kaikki yhden pikselin ensisijaiset säteet on prosessoitu. Laskettavia tehtäviä ovat kiihdytysrakenteen konstruointi, ensisijaisten säteiden suuntavektoreiden muodostaminen, lähimmän törmäyksen etsiminen ja varjosäteen lähettäminen. Jokaisesta näistä tehtävistä voidaan muodostaa oma ydinohjelma.

Ensimmäiset tehtävät, eli kiihdytysrakenteen konstruointi ja ensisijaisten säteiden muodostaminen ovat täysin toisistaan riippumattomat tehtävät. Kiihdytysrakenteiden konstruointia käsitellään luvussa 4.5. Ensisijaisten säteiden muodostamista varten täytyy varata muistilohko grafiikkaprosessorin muistista, johon sijoitetaan muodostetut säteet. Säteet muodostetaan omassa ydinohjelmassaan, joka suoritetaan kerran jokaista kuvan pikseliä kohti. Käyttämällä kaksiulotteisia säikeen indeksiä voidaan ydinohjelmassa määrittää, minkä pikselin läpi säde lähetetään. Kun säteet generoidaan omassa ydinohjelmassaan on myös jokaista sädettä kohti merkittävä pikseli, johon säteen seuraaminen vaikuttaa.

Säteitä ei välttämättä tarvitse muodostaa omassa ydinohjelmassa, vaan se voidaan tehdä ensisijaisten säteiden lähettämisen kanssa samassa ydinohjelmassa. Tämä vähentäisi grafiikkaprosessorien yleisen muistin käyttöä ja säteet voisi tallettaa väliaikaisesti säikeen lokaaliin muistiin. Kuitenkin olisi hyvä grafiikkaprosessorien suorituskyvyn kannalta minimoida lokaalin muistin käyttö, sillä se on jaettu kaikkien saman lohkon säikeiden kesken. Lisäksi jos halutaan lähettää sekundaarisia säteitä rinnakkain tulee joka tapauksessa tarpeelliseksi varata yleistä muistia sädelistalle.

Lähimmän törmäyksen etsimisestä eli säteen lähettämisestä voidaan muodostaa yksi ydinohjelma. Ydinohjelmassa käydään läpi kiihdytysrakenne, lasketaan lähin törmäys, lähetetään varjosäteet ja talletetaan väriarvo bittikarttaan. Jos lähetettävät säteet on talletettu listaan, ydinohjelmassa käytetään yksiulotteista säikeen indeksiä, jonka avulla määritetään ydinohjelmassa käsiteltävä säde.

Kiihdytysrakenteen läpikäyminen ja varjosäteen lähettämisestä voitaisiin myös muodostaa omat ydinohjelmat. Kuitenkin tämä vaatisi sen, että kiihdytysrakenteen läpikäymisestä saatu kappalejoukko, sekä säteen törmäyksesta saatu informaatio tulisi tallettaa yleiseen muistiin jatkokäsittelyä varten. Tämä tekisi ydinohjelmista

pienempiä, mutta kasvattaisi yleisen muistin käyttöä.

Yksi suurimmista ongelmista säteenseurantaohjelmiston toteuttamisessa on muistin riittämättömyys. Kuvan resoluution kasvaessa, myös tarvittavan muistin määrä kasvaa eksponentiaalisesti. Eräs ratkaisu muistinkäytön ongelmiin on muodostaa kuva palasissa peräkkäin. Kuvan muodostaminen palasissa hidastaa säteenseurantaprosessia, koska kaikkia säteenseurantatehtäviä ei ole rinnakkaistettu, mutta se mahdollistaa suuremman muistimäärän käyttämisen pikseliä kohti.

4.3 Sekundaaristen säteiden lähettäminen

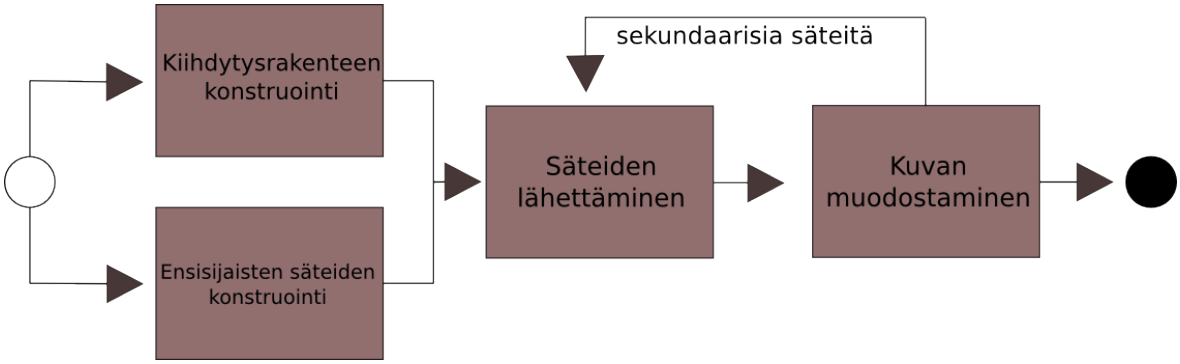
Jos haluamme lähettää sekundaarisia säteitä, ongelmana on, että perinteinen rekursiivinen säteenseuranta ei onnistu nykyaikaisen grafiikkaprosessoriteknologian avulla. Rekursiivisessa säteenseurannassa pinomuistia käytetään sekundaarisen säteiden seurannasta saadun värien yhdistämiseen. Jotta tämä onnistuisi ilman pinomuistia, on jokaisen sekundaarisen säteen kohdalla muistettava minkä pikselin väriarvoon se vaikuttaa.

Myöskään pinomuistin toteuttaminen ydinohjelmiin ei ole kovin tehokasta, koska tehokkain ratkaisu suorittaa laskenta grafiikkaprosessorilla on rinnakkaisesti. Säteen seuraaminen ydinohjelmassa tekisi ongelmasta peräkkäisen, sillä ydinohjelmat eivät voi käynnistää uusia säikeitä. Siksi jokaisen säteen lähettämisen kannattaisi olla oma ydinohjelmakutsunsa.

Eräs ratkaisu sekundaaristen säteiden seuraamiseen on käsitellä ne tasopohjaisesti. Ideana tässä menetelmässä on palauttaa jokaisen säteen lähettämisen jälkeen lista sekundaarisista säteistä. Kun kaikki ensisijaiset säteen on lähetetty, muodostetaan lista sekundaarisista säteistä ja lähetetään sekundaariset säteet. Ne muodostavat uuden kuvan ja palauttavat taas listan säteistä. Tätä suoritetaan kunnes sekundaaristen säteiden lista on tyhjä. (Segovia et al., 2009)

Tämä menetelmä luo joukon bittikarttoja, yhden jokaista säteen tasoa kohti (Segovia et al., 2009). Bittikartan yksi pikseli on sen tason säteiden palauttama väriarvo, joka rekursiivisessa säteenseurannassa palautetaan rekursiivisesta aliohjelmakutsusta. Lopuksi bittikarttojen pikseleiden väriarvot yhdistetään, josta saadaan lopullinen kuva (Segovia et al., 2009). Bittikarttojen yhdistämistä varten voidaan toteuttaa oma ydinohjelma, joka rinnakkaistaa yhdistämisprosessin.

Ongelmallisen tästä menetelmästä tekee sen muistintarve. Jokainen bittikarttaa varten on varattava tilaa yleisestä muistista jokaista säteiden tasoa kohti. Myös säteille pitää varata oma muistialue. Tietenkin säteille varatun muistialueen suuruus riippuu siitä kuinka monta sekundaarista sädettä yksi törmäys maksimissaan lä-



Kuva 4: Esimerkki rinnakkaisen säteenseurantaohjelman ydinohjelmista ja niiden suoritusjärjestyksestä

hettää, sekä maksimimäärästä säteiden tasoja. On myös huomattava, että säteiden määrä voi maksimissaan kasvaa eksponentiaalisesti joka tasolla.

Järkevästi tämän metodin voisi toteuttaa yhdistämällä tasot jokaisen uuden tason jälkeen. Ensisijaisten säteiden väriarvot laskettaisiin lopulliseen bittikarttaan, ja toisen tason säteet väripuskuriin. Kun toisen tason säteet olisi lähetetty yhdistettäisiin puskuri bittikarttaan, jonka jälkeen puskuriin laskettaisiin kolmas taso. Tällä menetelmällä grafiikkaprosessorin yleisestä muistista ei tarvitsisi varata kuin kaksi bittikarttaa väriarvojen tallettamiseksi.

Säteiden muistintarpeen vähentämiseksi voidaan käyttää aikaisemmin mainittua menetelmää, jossa säteenseurantatehtävä jaetaan peräkkäisiin osiin paloittelemalla laskettava kuva. Tällöin muistia tarvitaan vähemmän, kun pikseleitä on laskettavana kerralla vähemmän. Muistin varaamisessa on myös huomioitava, että grafiikkaprosessorilta varattua muistialuetta ei voi suurentaa ydinohjelman suorituksen aikana, jonka takia säteitä varten on varattava tarpeeksi muistia ennen ydinohjelman suoritusta.

Kuvassa 4 esitetään kuinka GPGPU-säteenseurannan ydinohjelmat voisivat muodostua. Kiihdytysrakenteen ja ensisijaisten säteiden konstruointi suoritettaisiin rinnakkain. Sen jälkeen lähetettäisiin konstruoidut säteet rinnakkain ja kun kaikki säteet on lähetetty yhdistetään lopulliseen kuvaan säteiden seuraamisesta saadut väriarvot. Kahta viimeistä kohtaa suoritetaan niin kauan kunnes säteiden lähettäminen ei tuota uusia sekundaarisia säteitä.

4.4 Säteenseurantaa kiihdyttävät menetelmät GPGPU-laskennalla

Säteenseurantaa kiihdyttävät menetelmät ovat tärkeä osa sen nopeuttamisessa. Hyvin konstruoidut kiihdytysrakenteet vähentävät tarvittavien säteen törmäyksien las-

kentaa radikaalisti. Perinteisesti kiihdytysrakenteiden konstruointi ja läpikäyti ovat peräkkäisiä tai rekursiivisia algoritmeja. Esimerkiksi useasti puurakenteet konstruoidaan ja käydään läpi käyttämällä rekursiivisia aliohjelmakutsuja.

GPGPU-laskennalla olisi hyvä saada kiihdytysrakenteiden konstruointi- ja läpikäymisalgoritmit toimimaan ilman rekursiota, sillä rekursiivisten algoritmien toteuttaminen ydinohjelmissa kuluttaa paljon välimuistia. Kuten aiemmin mainittiin kiihdtysrakenteiden konstruointi on täysin oma tehtävänsä, joten paras valinta konstruointialgoritmiksi olisi mahdollisimman hyvin rinnakkaistuva algoritmi. Kiihdytysrakenteen läpikäymisalgoritmit ovat taas sidottuja säteiden lähettämiseen, jolloin voisi olla hyvä käydä kiihdytysrakenne peräkkäisesti läpi samassa ydinohjelmassa.

Seuraavissa kahdessa aliluvussa palataan luvussa 3.4 mainittuihin kiihdytysrakenteisiin ja pohditaan niiden kanssa käytettäviä algoritmeja GPGPU-laskennan näkökulmasta. Seuraava aliluku käsittelee kiihdytysrakenteiden konstruointia ja viimeinen aliluku käsittelee niiden läpikäytiä.

4.5 Kiihdytysrakenteiden konstruointi

Monesti säteenseurannan kiihdytysrakenteet konstruoidaan keskusprosessorilla. Varsinkin epäinteraktiivisten sovellusten tapauksissa tämä tapa on siedettävä, sillä kiihdytysrakenteiden konstruointi on paljon säteenseurantaa nopeampi tehtävä. Kuitenkin interaktiivisissa sovelluksissa kiihdytysrakenteet on konstruoitava uudelleen tai niitä on muokattava pienien aikavälien sisällä useasti. Kun kiihdytysrakenne saatetaan joutua konstruoimaan useamman kerran sekunnin aikana, olisi järkevää tällöin toteuttaa kiihdytysrakenteiden konstruointi rinnakkain grafiikkaprosessorilla.

Kun grafiikkaprosessorin varatun muistin kokoa ei voi muuttaa ydinohjelman suorituksen aikana, tietorakenteen konstruointia varten tarvittava muistimäärä on varattava ennen konstruointiydinohjelman ajamista. Kiihdytysrakenteet eivät myöskään voi olla dynaamisesti varattuja linkitettyjä rakenteita, koska niiden siirtäminen grafiikkaprosessorin ja isäntäjärjestelmän välillä on vaikeaa erillisten muistiavaruuksien takia.

Eräs menetelmä tallettaa kiihdytysrakenteita grafiikkaprosessorin muistiin on käyttää kahta erillistä muistipuskuria. Ensimmäinen muistipuskureista sisältää kiihdytysrakenteen jakojen sisällön tiedot ja toinen osoittimia sisältöpuskurin alkioihin. Osoittimina ei tarvitse käyttää muistiosoittimia, vaan riittää merkitä tavu, josta alkio alkaa sisältöpuskurssa. Tällä tavalla jokaisen jaon sisältämät tiedot löydetään lisäämällä sisältöpuskurin alkuun osoittavaan osoittimeen osoitinpuskurin sisältä-

män arvon. Oleellisin idea tässä menetelmässä on, että toinen puskureista sisältää saman kokoisia muistilohkoja ja toinen vaihtelevan kokoisia muistilohkoja. Tällä tavalla voidaan antaa eri kiihdytysrakenteen jaoille eri verran muistia käytettäväksi.

Kiihdytysrakennetta konstruoidessa on huomioitava, että muistilohkot ovat tarpeeksi isoja. Osoitinpuskurin koko on yleensä helppo määritellä, sillä kiihdytysrakenteen jakojen määrä on säädeltävissä ja muistilohkot ovat saman kokoisia. Sisältöpuskurin muistilohkot ovat eri kokoisia ja tästä syystä on vaikea arvioida, kuinka paljon sisältöpuskuria varten olisi varattava tilaa. Tarvittavan tilan määrä on approksimoitava kappaleiden määrän pohjalta tai laskettava vaadittu tila etukäteen.

Edellistä menetelmää voidaan esimerkiksi käyttää yksikköruudukon kanssa. Osoitinpuskuri sisältää osoittimia yksikköruudukon ruutujen määrän, eli jokaista ruutua kohti on yksi osoitin sisältöpuskuriin. Tämä osoitin osoittaa kohtaan, jossa sisältöpuskurissa alkaa kyseisen yksikköruudun sisältämät kappaleet. Sisältöpuskuri sisältää siis erimittaisia listoja kappaleista. (Ivson, Duarte, & Celes, 2009)

Kiihdytysrakenteiden konstruointi voidaan toteuttaaa usean ydinohjelman avulla, jotta se saataisiin rinnakkaistettua parhaiten. Esimerkiksi yksikköruudukon konstruointi voidaan toteuttaa neljällä ydinohjelmalla (Ivson et al., 2009). Ensimmäinen ydinohjelma suoritetaan jokaista avaruuden kappaletta kohti ja se laskee yhteen kaikki yksikköruudukon ruutujen ja kappaleiden leikkaukset (Ivson et al., 2009). Tämän tiedon avulla voidaan varata tarvittava määrä muistia sisältöpuskuriin (Ivson et al., 2009). Toinen ydinohjelma suoritetaan myös jokaista kappaletta kohti ja se muodostaa numeroparilistan sisältöpuskuriin, joka sisältää ruutujen ja niitä leikkaavien kappaleiden indeksejä (Ivson et al., 2009). Kolmas ydinohjelma järjestää edellisen välituloksen ruutujen indeksejen suhteen, jotta neljännen ydinohjelman avulla voitaisiin muodostaa helposti osoitinpuskuri hakualgoritmin avulla (Ivson et al., 2009).

Yksikköruudukon käyttämisen etuja ovat nopea konstruointi, sekä muokkaamismahdollisuudet. Kuitenkaan yksikköruudukolla ei voida kapseloida avaruuden kappaleita yhtä tehokkaasti kuin puurakenteilla.

Puurakenteiden konstruointialgoritmit ovat perinteisesti toteutettu rekursiivisesti. Kuitenkin rekursiivisten algoritmien toteuttaminen GPGPU-laskennalla on tehotonta ja on käytettävä ratkaisuita, jotka eivät tarvitse rekursiota. Lisäksi konstruoinnin tulisi olla hyvin rinnakkaistuva.

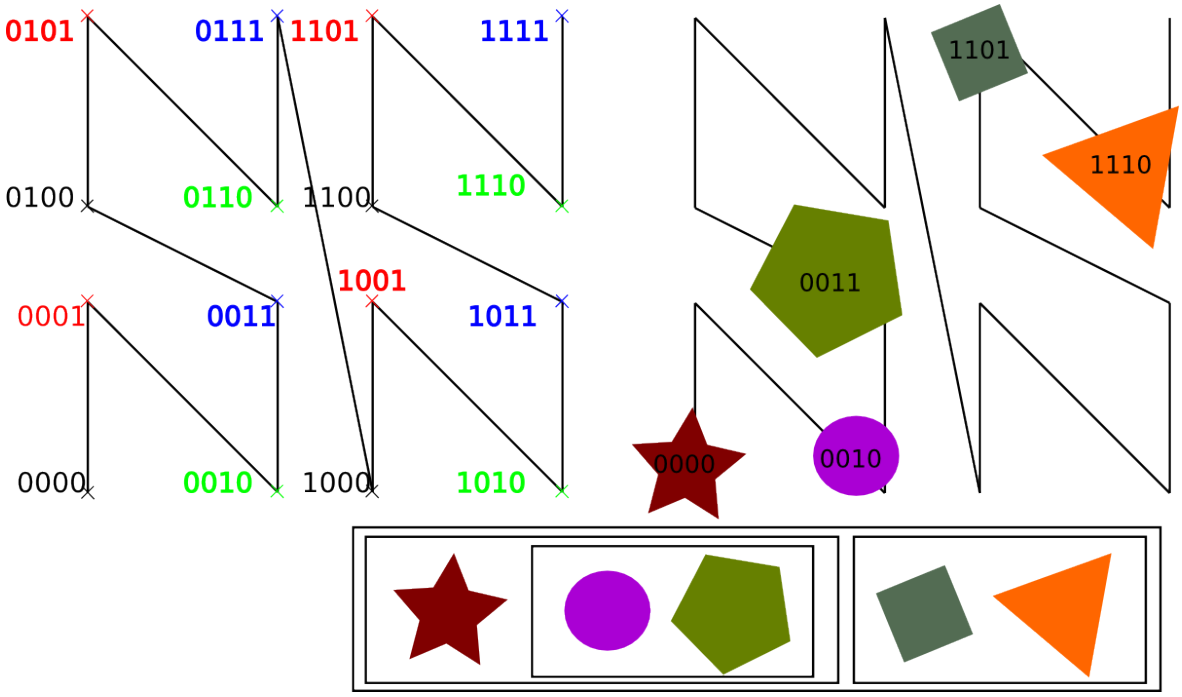
Esimerkki puurakenteiden konstruoinnista on LBVH-algoritmi (Linear Bounding Volume Hierarchy), joka on yksinkertaisimpia menetelmiä BVH:n konstruointiin grafikkaprosessorilla. Se perustuu avaruuden kappaleiden lajittelemiseen Morton Curve-metodilla, jonka avulla voidaan määrittää avaruuden kappaleille järjes-

tys ns. *Morton koodit* (Morton Code), niiden paikan perusteella avaruudessa. Tämän numeroarvon perusteella kappaleet voidaan järjestää kapselointia varten. (Lauterbach, Garland, Sengupta, Luebke, & Manocha, 2009)

Morton koodin bitit osoittavat miten kappaleet ovat sijoittuneet avaruuteen toistensa kanssa (Lauterbach et al., 2009). Tämän tiedon avulla voidaan kappaleet kapseloida järjestämisalgoritmien avulla (Lauterbach et al., 2009). Kuvassa 5 näytetään kuinka Morton curve määritetään, sekä miten sen avulla voidaan rakentaa BVH.

LBVH-algoritmin etuna on sen yksinkertaisuus ja nopeus. Kuitenkin se ei välttämättä konstruoi hyvin optimoituja BVH-rakenteita säteenseurantaa varten. Tästä syystä se ei ole paras valinta konstruointialgoritmiksi, mutta sitä voidaan hyödyntää muiden algoritmien kanssa yhdessä. (Lauterbach et al., 2009)

Puukiihdytysrakenteiden konstruointialgoritmit ovat monimutkaisia ja -osaisia. Ne koostuvat useista ydinohjelmista, jotta algoritmista saataisiin mahdollisimman rinnakkainen. Esimerkkejä rinnakkaisista konstruointialgoritmeista on esitetty ((Lauterbach et al., 2009) ja (Zhou, Hou, Wang, & Guo, 2008)), mutta kuitenkin suurin osa GPGPU-säteenseurannan tutkimisesta on liittynyt puurakenteiden läpikäymiseen.



Kuva 5: Morton koodin määrittäminen ja BVH:n konstruointi LBVH:n avulla

4.6 Kiihdytysrakenteiden läpikäyti

Yksikköruudukon läpikäyminen voidaan toteuttaa iteratiivisesti GPGPU-laskennalla käyttämällä viivanrasterointialgoritmia. Kuitenkin toinen kiinnostava ratkaisu olisi käydä yksikköruudukko läpi rinnakkain siten, että ydinohjelma käsittelee kerrallaan yhden avaruuden ruudun (Nery et al., 2011). On kuitenkin kyseenalaista, onko tämä metodi nopeampi, kuin iteratiivinen läpikäymistapa. Erityisesti jos ruudukko on iso, joudutaan rinnakkaisella läpikäymistavalla käsittelemään paljon ylimääräisiä ruutuja.

Rinnakkainen lähestymistapa yksikköruudukon läpikäymiseen on kuitenkin parempi ratkaisu, jos ei käsitellä yksittäisiä säteitä, vaan sädepaketteja. Läpikäyminen sädepakettejen kanssa vaatii yksikköruudukon ruutujen, sekä katkaistun kartion välisen leikkauksen tarkastelemista. Tällöin ei ole mahdollista käyttää viivanrasteroimisalgoritmeja.

Perinteisesti puurakenteiden läpikäyminen tehdään rekursiivisesti. Kuten aiemmin on todettu, rekursiiviset algoritmejen toteuttaminen ei ole tehokasta GPGPU-laskennalla. Tästä syystä läpikäymiseen on käytettävä muita algoritmeja.

Kd-restart on k-d puiden läpikäymistä varten esitetty algoritmi, joka toimii ilman rekursiota. Sen ideana on aloittaa k-d puun läpikäyminen alusta joka kerta, kun läpikäymisessä joudutaan väärään polkuun. Muuten k-d puun läpikäyminen tapahtuu luvussa 2.4 esitetyllä tavalla.(Foley & Sugerman, 2005)

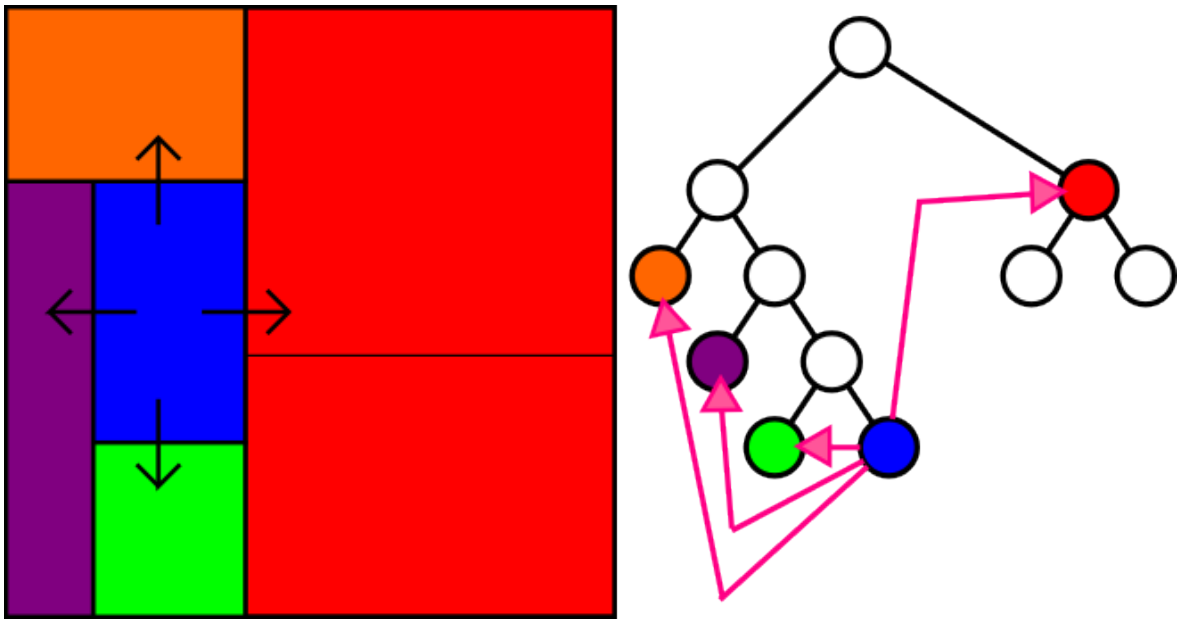
Kd-restart on hidas algoritmi, koska se voi joutua laskemaan samoja leikkauksia säteen ja solmujen välillä monia kertoja. Sen etuna on pieni muistinkäyttö ja iteratiivinen toimintatapa, jotka ovat hyviä ominaisuuksia GPGPU-laskennan näkökulmasta. Kd-restart on toiminutkin puurakenteiden iteratiivisen läpikäymisen pohjaideana ja siihen pohjautuen on tehty monia laskentatehokkaampia muunnelmia.

Kd-backtrack on muunnelma kd-restartista, jossa uudelleen läpikäymisen sijaan palataan puussa taaksepäin. Jotta puussa voitaisiin palata taakseppäin, on puun solmuihin oltava talletettu osoitin sen vanhempaan. Kun tutkitusta solmusta palataan taaksepäin, siirretään leikkauksen alkukohtaa ja otetaan leikkaus solmun vanhemmasta. Kd-backtrack on hieman nopeampi algoritmi kuin kd-restart, kun puuta ei tarvitse käydä aina läpi alusta asti. Se kuitenkin vaatii hieman enemmän muistia vanhempiosoittimien takia.(Foley & Sugerman, 2005)

Toinen mahdollisuus kiihdyttää kd-restart algoritmia on lisätä osoittimet k-d puun solmun vierekkäisiin solmuihin. Vierusosoittimet mahdollistavat nopean k-d puun läpikäymisen, kun voidaan siirtyä suoraan seuraavaan säteen leikkaamaan avaruuden jakoon. Kuitenkin vierusosoittimet kasvattavat k-d puun muistinkulu-

tusta vielä enemmän kuin kd-backtrack-algoritmin vaatimat konstruktiot, kun jokaista puun jaon muodostaman kuusitahokkaan tahkoa kohti täytyy tallettaa osoitin, tahkoa vastapäätä olevaan solmuun. Vierusoittimet voidaan lisätä k-d puuhun konstruoimisen jälkeen, joka hidastaa k-d puun konstruoimista. Lisäksi vierusosoittimien muodostaminen kannattaa tehdä siten, että viereisiksi solmuiksi valitaan mahdollisimman matalan tason solmut, jotta läpikäytävien solmujen määrä voitaisiin minimoida. (Popov, Günther, Seidel, & Slusallek, 2007)

Kuvassa 6 näytetään kuinka vierusosoittimet määritetään kaksiulotteiseen k-d puuhun. Kuvan solmut ovat värikoodattu siten, että kuvassa näkyvät vierusosoittimet lähtevät sinisestä solmusta ja muut värilliset solmut ovat sinisen solmun viereisiä solmuja.



Kuva 6: Vierusosoittimet k-d puussa

Eräs idea kd-restart algoritmin kiihdyttämiseksi on käyttää push-down metodia. Sen ideana on, että uudeksi juurisolmuksi merkitään solmu jonka ulkopuolisia solmuja ei tarvitse enää käydä läpi. Tämä tehdään siis aina, kun puun läpikäyminen aloitetaan uudelleen ja säde leikkaa vain toista puun lapsisolmua. (Horn et al., 2007)

On myös esitetty, että pinomuistilla voisi kiihdyttää kd-restart algoritmia. Kun säde leikkaa jonkun solmun molempia lapsisolmuja, voitaisiin toinen lapsisolmu ja säteen leikkauksen alkukohta ja loppukohta toisesta lapsisolmusta juurisolmun loppuun työntää pinoon. Sen jälkeen tutkittaisiin solmut jotka sijaitsevat ensimmäisen lapsisolmun sisällä, eli käsiteltäisiin säteen leikkauskohdat ensimmäisen lapsisolmun alusta, sen loppuun. Kun kaikki leikkaukset on käyty läpi, nostetaan pinosta

pois toinen lapsisolmu ja leikkausväli käsittelyä varten. (Horn et al., 2007)

Vaikka edelliset läpikäymisalgoritmit on suunniteltu k-d puun läpikäymistä varten ne kaikki on toteutettavissa kahdeksanpuulle (Römisch, 2009). Erona k-d puun läpikäymiseen on kuinka solmujen ja säteen leikkaukset lasketaan (Römisch, 2009). Lisäksi vierusosoittimen konstruointi tapahtuu hieman eri tavalla.

Ei ole myöskään mitään syytä miksi edellisistä algoritmeista kaikki, jotka perustuvat puun läpikäymisen kontrollointiin eivät olisi myös toteutettavissa BVH:lle. Ainoa algoritmeista joka ei sovellu BVH:n kanssa käytettäväksi on vierusosoittimet, jotka on mahdotonta totetutaa BVH:lle, sillä BVH:n solmuilla ei välttämättä ole viereisiä solmuja avaruudessa.

5 Yhteenveto

Säteenseurannasta grafiikkaprosessoreilla on tehty monenlaista tutkimusta. Suurin osa tutkimuksista ja kirjallisuudesta liittyvät erillisiin GPGPU-säteenseurannan osa-alueisiin, eikä aiheesta ole yleispätevää kirjallisuutta. Kuitenkin GPGPU-laskentaan ja säteenseurantaan molempiin on olemassa paljon kirjallisuutta ja julkaisuja, joidenka pohjalta on mahdollista totetuttaa tutkimusta liittyen GPGPU-säteenseurantaan ja suunnitella GPGPU-säteenseurantaohjelmia.

Suurin puute GPGPU-säteenseurantaan liittyvässä kirjallisuudessa on se, että GPGPU-säteenseurantaohjelman perusrakenteen selittävää julkaisua ei löydy. Syynä tähän voisi olla se, että GPGPU-säteenseurantaohjelman rakenne on triviaali ongelma ihmisille joille säteenseurannan ja GPGPU-laskennan perusteet ovat tuttuja. Valtaosa kirjallisuudesta käsittelee kiihdytysrakenteita ja muita GPGPU-säteenseurannan kiihdytysmenetelmiä. Tämä on ymmärrettävää, sillä kiihdytysrakenteet ja -menetelmät ovat erittäin oleellinen osa-alue tehokkaan säteenseurantaohjelmiston totetuttamisessa, ja perinteiset algoritmit eivät ole tehokkaita totetuettuna GPGPU-laskennalla.

Tässä tutkielmassa on esitetty GPGPU-säteenseurannan ongelmakohtia, sekä joitakin ratkaisuja niihin. Suurimmat ongelmat GPGPU-säteenseurannassa tulevat grafiikkaprosessorien muistinhallinnasta. Luonteensa mukaan, perinteisesti säteenseuranta on toteutettu rekrusiivisena ohjelmana. Tämä ei kuitenkaan GPGPU-laskentaa käyttämällä ole mahdollista. Muistinkäytöstä aiheutuu myös ongelmia kiihdytysrakenteiden käyttämisessä, kun perinteiset keskusprosessorille suunniteltujen algoritmien käyttäminen ei ole enää tehokasta.

Kuten monissa tutkimuksissa on todettu, GPGPU-laskenta mahdollistaa tehokkaan rinnakkaisen säteenseurannan toteuttamisen. GPGPU-säteenseurantaa varten

on kehitetty erilaisia algoritmeja, jotka soveltuvat paremmin grafiikkaprosessorille. On tarpeellista tutkia näiden uusien menetelmien tehokkuuksia ja vertailtava niitä muihin menetelmiin, jotta voitaisiin löytää parhaat tavat käyttää ja nopeuttaa GPGPU-säteenseurantaa.

Viitteet

A.Carr, N., Hall, J. D., & Hart, J. C. (2002). *The ray engine.*

Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. In *In eurographics 87* (pp. 3–10).

Britton, A. D. (2010). *Full cuda implementation of gpgpu recursive ray-tracing.*

Cook, R. L., Porter, T., & Carpenter, L. (1984). Distributed ray tracing. *SIGGRAPH Comput. Graph.*, *18*(3), 137–145.

Foley, T., & Sugerman, J. (2005). *Kd-tree acceleration structures for a gpu raytracer.*

Günther, J., Popov, S., Seidel, H.-P., & Slusallek, P. (2007). Realtime ray tracing on gpu with bvh-based packet taversal. In *Proceedings of the ieee/eurographics symposium on interactive ray tracing 2007* (pp. 113–118).

Horn, D. R., Sugerman, J., Houston, M., & Hanrahan, P. (2007). *Interactive k-d tree gpu raytracing.*

Ivson, P., Duarte, L., & Celes, W. (2009). *Gpu-accelerated uniform grid construction for ray tracing dynamic scenes.*

Karimi, K., Dickson, N. G., & Hamze, F. (2010). A performance comparison of cuda and opencl. *CoRR*, *abs/1005.2581*.

Kirk, D. B., & Hwu, W.-m. W. (2009). *Programming massively parallel processors.* Morgan Kaufmann publications.

Kuchkuda, R. (1988). An introduction to ray tracing. *Theoretical Foundations of Computer Graphics and CAD*, *40*, 1039-1060.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., & Manocha, D. (2009). Fast bvh construction on gpus. *Computer Graphics Forum*, *28*(2), 375–384.

Lauterbach, C., Yoon, S.-E., Tuft, D., & Manocha, D. (2006). Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *Interactive ray tracing 2006, ieee symposium on* (p. 39 -46).

Meseth, J., Guthe, M., & Klein, R. (2005). Interactive fragment tracing. *Visual Computer*, *21*, 591–600.

Nery, A., Nedjah, N., Franca, F., & Jozwiak, L. (2011). *A parallel ray tracing architecture suitable for application-specific hardware and gpgpu implementations.*

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J. (2008). Gpu computing. *Proceedings of the IEEE*, *96*(5), 879 -899.

Owens, J. D., David, L., Naga, G., Mark, H., Jens, K., E., L. A., & J., P. T. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, *26*, 80–113.

Popov, S., Günther, J., Seidel, H.-P., & Slusallek, P. (2007). Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, *26*(3), 415–424. ((Proceedings of Eurographics))

Purcell, T. J. (2004). *Ray tracing on a stream processor.*

Reshetov, A., Soupikov, A., & Hurley, J. (2005). Multi-level ray tracing algorithm. *ACM Trans. Graph.*, *24*(3), 1176–1185.

Römisch, K. (2009). *Sparse voxel octree ray tracing on the gpu.* Aarhus Universitet, Datalogisk Institut.

Segovia, A., Li, X., & Gao, G. (2009). *Iterative layer-based raytracing on cuda.*

Wald, I. (2004). *Realtime Ray Tracing and Interactive Global Illumination.* Unpublished doctoral dissertation, Computer Graphics Group, Saarland University.

Zhou, K., Hou, Q., Wang, R., & Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, *27*(5), 126:1–126:11.