Olli Koskinen

Säteenseuraajarenderöijän implementointi ja optimointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

10.5.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Olli Koskinen Säteenseuraajarenderöijän implementointi ja optimointi 29 sivua + 1 liitettä 10.5.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikan koulutusohjelma
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Miikka Mäki-Uuro, lehtori Juha Kopu, lehtori

Tämän insinöörityön tavoitteena oli syventyä säteenseuraajarenderöinnin teoriaan, sen toteutukseen ja optimointiin. Säteenseuranta on apuväline, jolla voidaan projisoida 3D-maailma 2D-kuvaksi. Työn toteutukseen käytettiin NVIDIA:n rinnakkaislaskentaa varten kehitettyä CUDA-ohjelmointikieltä. Työn pääpaino oli näytönohjaimella tehtävän rinnakkaislaskennan haasteissa sekä säteen/kolmion törmäystarkistuksen toteutuksessa.

Työn alussa kerrotaan säteenseurannan teoriasta. Tämän jälkeen kerrotaan teoriaa eri rajapinnoista, joita on käytetty työssä säteenseurantarenderöijän toteuttamiseen. Työn toteutusosassa käydään läpi koodiesimerkein säteenseurantarenderöijän implementointi. Työn lopussa perehdytään säteenseurannassa yleisimmin käytettyihin optimointitekniikoihin. Näitä tekniikoita ovat mm. kiihdytystietorakenteet ja eri muistityyppien käyttäminen.

Työn tekijän hypoteesina oli, että noin satatuhatta kolmiota sisältävän 3D-mallin piirtäminen saataisiin reaaliaikaiseksi optimoidulla säteenseurantarenderöijällä. Aiheesta löytyi runsaasti kirjallisuutta ja sen avulla sai koottua kattavan tietopaketin asiasta kiinnostuneille. Samalla opin paljon säteenseurannan teoriasta ja toteutuksesta, sekä näytönohjaimen ohjelmoimisesta CUDA:lla. Optimoinnin osalta saatiin kerättyä paljon tietoa teoriasta, mutta käytännön toteutus vaatii vielä jatkokehitystä. Uskon, että opinnäytetyöni toimii hyvänä tiedonlähteenä säteenseurantarenderöijän ja CUDA:n käytöstä kiinnostuneille.

Säteenseuranta, CUDA, Renderöinti	

Author(s) Title Number of Pages Date	Olli Koskinen Implementation and Optimization of Ray Tracer Renderer 29 pages + 1 appendices 10 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, senior lecturer Juha Kopu, senior lecturer

The aim of the study was to find out about the theory, implementation and optimization of ray tracer renderer. Ray tracing is a tool for projecting 3D-world to 2D-image. The implementation of the ray tracer renderer was done with NVIDIA's parallel computing architecture CUDA. The main challenge in the study were the problems of parallel execution of code and the implementation of ray/triangle intersection.

The study starts with the theory of ray tracing. After this, different interfaces used in the implementation of the ray tracer are introduced. In the implementation part of the study, certain code examples for the basic ray tracer are explained. In the end, a few basic concepts for optimizing the ray tracer renderer are introduced e.g. accelerator data structures and using different memory architecture for storing data.

The hypothesis of the study was that an optimized ray tracer renderer should achieve approximately real time rendering speed with a 3D-model that has 100k triangles. There are quite a lot of studies and literature about the topic and therefore a rather comprehensive package for anyone interested could be gathered. The study serves as a good beginner level guide to ray tracing and CUDA programming.

Ray tracing, CUDA, Rendering	

Sisällys

Lyhenteet

Renderöinti	Kuvan luominen malleista tietokoneohjelman avustuksella.
Ray tracing	Säteenseuranta, kuvantamistekniikka, jossa 3D-maailma projisoidaan näytölle katsojan näkökulmasta säteiden avulla.
Säde	Jana kamerasta ulospäin, jonka avulla tarkistetaan, onko ruudulla näkyviä malleja.
OpenGL	Open Graphics Library, avoin grafiikkakirjasto. Käytetään yksinkertaistamaan grafiikan piirtämistä ruudulle.
CUDA	Compute Unified Device Architecture, NVIDIA:n kehittämä C-ohjelmointikielen kaltainen näytönohjainten ohjelmointirajapinta.
SM	Streaming Multiprocessor, näytönohjaimen osa, joka sisältää CUDA-ytimiä, käytettiin Tesla- ja Fermi-arkkitehtuurissa.
SMX	Next Generation Streaming Multiprocessor, uudistettu SM Kepler -arkkitehtuurissa.
PTX	Parallel Thread eXecution, pseudo Assembly, jonka näytönohjaimen ajuri kääntää binääriksi ennen suoritusta.
Kernel	Ydin, näytönohjaimella rinnakkain suoritettava ohjelma, joka on kirjoitettu C-, C++ tai Fortran-ohjelmointikielillä.
Pikseli	Suorakulmion muotoinen kuvan osa, jonka koko on suoraan verrannollinen kuvan resoluutioon. Tunnetaan myös kuvapisteenä.
PBO	Pixel Buffer Object, näytönohjaimen muistissa sijaitseva pikselipuskuri, johon tallennetaan pikseleitä näyttämistä varten.
API	Application Programming Interface, ohjelmointirajapinta.
Assembly	Matalan tason ohjelmointikieli.

1 Johdanto

Tämän insinöörityön tavoitteena oli syventyä säteenseurantarenderöinnin teoriaan, toteutukseen ja optimointiin. Säteenseuranta on apuväline, jolla voidaan projisoida 3D-maailma 2D-kuvaksi. Työn toteutukseen käytettiin NVIDIA:n rinnakkaislaskentaa varten kehitettyä CUDA-ohjelmointikieltä. Kiinnostuksen kohteena oli erityisesti CUDA:n rinnakkaislaskentaominaisuudet. Uskon, että näytönohjaimella tehty rinnakkaislaskenta tulee olemaan tärkeä osa tietotekniikan kehitystä.

Henkilökohtaisena tavoitteenani oli oppia OpenGL:n, CUDA:n ja säteenseurannan perusteita. Työn pääpaino oli kuitenkin näytönohjaimella tehtävän rinnakkaislaskennan haasteissa sekä säteen/kolmion törmäystarkistuksen toteutuksessa. Työn alussa luvussa 2 kerrotaan säteenseurannan ja rasteroinnin teoriasta ja siitä, miten nämä tekniikat eroavat toisistaan.

Luvussa 3 kerrotaan OpenGL- ja CUDA-rajapinnoista, joita käytetään työssäni säteenseurantarenderöijän toteuttamiseen. Luvussa 3.1 tutustutaan pintapuolisesti OpenGL:n teoriaan. Luku 3.2 käsittelee CUDA:n arkkitehtuuria, yleistä kernel-funktioista ja CUDA:n yhteistyötä OpenGL:n kanssa.

Luvussa 4 tutustutaan koodiesimerkein säteenseurantarenderöijän implementointiin. Tässä luvussa perehdytään tarkemmin mm. säteenseuranta CUDA kernel -funktioon, kolmion törmäystarkistukseen, 3D-mallien lataamiseen muistiin ja 3D-mallien sävytykseen.

Luvussa 5 perehdytään säteenseurannassa yleisimmin käytettyihin optimointitekniikoihin. Näitä ovat mm. kiihdytystietorakenteet ja eri muistityyppien käyttäminen. Kiihdytystietorakenteista käsitellään Bounding Volume Hierarchya (BVH). Luvun lopussa tutustutaan Texture Memoryn käyttöön ohjelman suorituksen nopeuttamisessa.

Luvussa 6 pohditaan säteenseurantarenderöijän implementoinnin onnistumista ja haasteita. Pohdintaosuudessa otetaan myös kantaa jatkokehittelymahdollisuuksiin sekä siihen mitä olisi voitu tehty toisin. Työn hypoteesina oli, että noin satatuhatta kolmiota sisältävän 3D-mallin piirtäminen saataisiin reaaliaikaiseksi optimoidulla säteenseuranta-

renderöijällä. Tässä reaaliaikaisella tarkoitetaan vähintään 24 ruutua sekunnissa. Hypoteesini perustuu edeltäviin tutkimuksiin, joiden mukaan puuhierarkian käyttö nopeuttaa huomattavasti säteenseurantarenderöijän suorituskykyä [16]. Uskon, että opinnäytetyöni toimii hyvänä tiedonlähteenä säteenseurantarenderöijän ja CUDA:n käytöstä kiinnostuneille.

2 Renderöintitekniikat

2.1 Yleistä

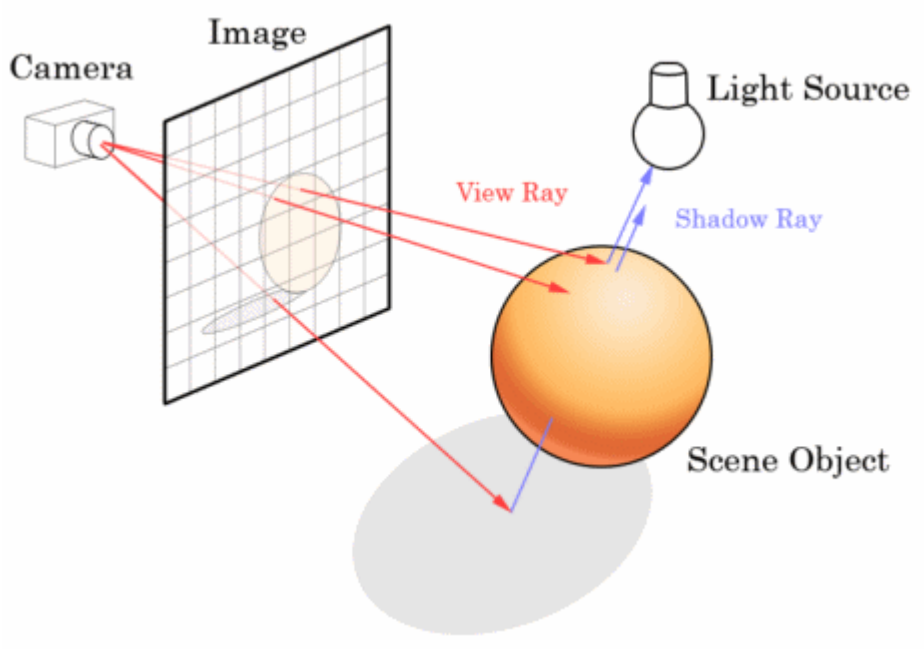
Yksi tietokonegrafiikan päätavoitteesta on realistisen kuvan piirtäminen. Tavoite pyritään saavuttamaan vertaamalla ympäröivää fysikaalista maailmaa tämän hetken edistyneimpiin tietokoneilla luotuihin kuviin. Havaitut puutteet tietokoneella luodussa kuvassa pyritään ratkaisemaan tutkimalla fysikaalisia ilmiöitä. Tietokonegrafiikan alkuhetkillä kuvaa saatiin paranneltua nopeasti. Näihin parannuksiin kuuluvat monet, nykyään jo standardina olevat tekniikat kuten objektin valonläpäisevyys, pintojen valontaittokyky ja heijastavuus. [9.] Säteenseuraajarendereröijä pyrkii havainnollistamaan valonlähteen ja havaitsijan välillä kulkevia säteitä. Tarkasteltavan kohteen väri määräytyy pohjamateriaalin väristä ja potentiaalisista valon absorboitumisesta ja heijastumisesta.

2.2 Säteenseuranta ja rasterointi

Säteenseuranta ja rasterointi ovat renderöintitekniikoita. Säteenseuranta on apuväline, jolla voidaan projisoida 3D-maailmaa 2D-kuvaksi. Kun halutaan piirtää 2D-projektio 3D-maailmasta mahdollisimman realistisesti, täytyy pystyä laskemaan, kuinka paljon valoa kukin pikseli vastaanottaa [10]. Säteenseurannan metodien sanotaan saaneen alkunsa fysiikan kirjallisuudesta, jossa tutkittiin valon kulkemista linssien läpi. Linssejä suunniteltaessa fyysikot piirsivät usein säteen valon lähteestä linssin läpi. Tätä prosessia kutsuttiin säteenseurannaksi. [9.]

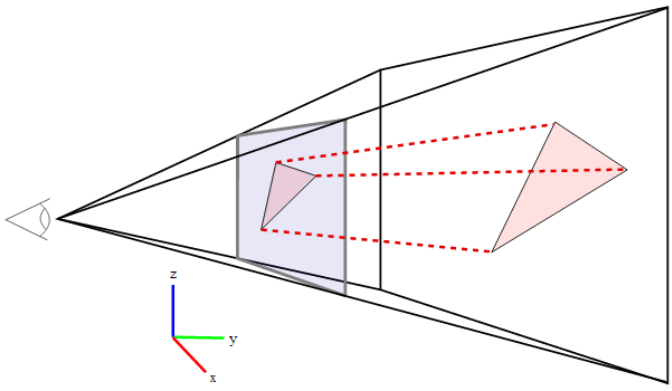
Tällä hetkellä rasterointi on kuitenkin yleisin apuväline esitettäessä 3D-maailmaa 2D-kuvana ja sitä varten on suunniteltu suurin osa tällä hetkellä markkinoilla olevista näytönohjaimista. Fundamentaalinen ero säteenseurannan ja rasteroinnin välillä on seuraavanlainen: säteenseurantarenderöinnissä valitaan yksi pikseli kerrallaan, josta

sitten ammutaan säde kohti kaikkia muistissa olevia geometrisiä objekteja (kuva 1), kun taas rasteroinnissa valitaan yksi kolmio, jonka 3D-koordinaatit pyritään muuttamaan ruudun 2D-koordinaateiksi ja näin sovittamaan ruutuun (kuva 2).



Kuva 1. Säteenseurannan perusperiaate. [25]

Pseudokoodina ajateltuna säteenseurannan uloin silmukka käy läpi pikselin yksi kerrallaan, kun rasteroinnissa uloimmassa silmukassa tutkittaisiin esimerkiksi kolmioita.



Kuva 2. Rasteroinnin peruskäsite. [24]

Vaikka molemmat renderöintitekniikat ovat olleet käytössä jo 1960-luvulla, rasterointi yleistyi näytönohjaimissa, koska sen avulla voitiin toteuttaa isompia 3D-malleja käyttämällä säteenseurantatekniikkaa vähemmän muistia. Reaaliaikaisen suorituskyvyn parantamisen nimissä hyväksyttiin, että valon realistinen simulointi ei olisi mahdollista. Realistisen valon simuloinnin sijasta käytettiin approksimaatiotekniikoita, joilla päästiin tyydyttävään lopputulokseen. Säteenseuranta soveltuu kuitenkin approksimaatiotekniikoita paremmin valon realistiseen simulointiin. Säteenseurantaa toteutettaessa tulee mieleen, onko prosessia mahdollista optimoida ampumalla säde kaikista pikseleistä samaan aikaan? Juuri tähän kysymykseen otetaan kantaa työssäni.

3 Ohjelmointirajapinnat

Ohjelmointirajapinnalla tarkoitetaan tässä työssä kahta eri koodikirjastoa, joiden avulla näytönohjaimella voidaan suorittaa sekä laskentaa että graafisia toimintoja, kuten näytölle piirtämistä. OpenGL- ja CUDA-rajapinnoista, joita käytetään säteenseurantarenderöijän toteuttamiseen. Luvussa 3.1 tutustutaan pintapuolisesti OpenGL:n teoriaan. Luku 3.2 käsittelee CUDA:n arkkitehtuuria, yleistä kernel-funktioista ja CUDA:n yhteistyötä OpenGL:n kanssa.

3.1 OpenGL

OpenGL on alustariippumaton API-spesifikaatio tietokonegrafiikan renderöintiä varten [7]. Alustariippumattomuuden takia OpenGL-implementaatiot toimivat monen eri valmistajan näytönohjaimella nyt ja tulevaisuudessa. Moderni OpenGL-spesifikaatio toteuttaa ainoastaan grafiikan renderöinnin, mutta se ei määrittele mitään animointiin, ajoituksiin, tiedosto-operaatioihin, graafisiin käyttöliittymiin tai siirto-operaatioihin liittyvää [8].

Työssäni oli käytössä OpenGL:n lisäksi SDL-apukirjastoa ikkunointiin sekä syötteiden käsittelyyn. Tämän lisäksi työssä käytettiin myös GLEW-apukirjasto, jotta Windows- ja Unix-pohjaisia käyttöjärjestelmiä voitaisiin tukea ilman lisävaivaa. Työtä aloittaessani en aivan ymmärtänyt kuinka vähän loppujen lopuksi tarvitsisin itse OpenGL:n toimintoja.

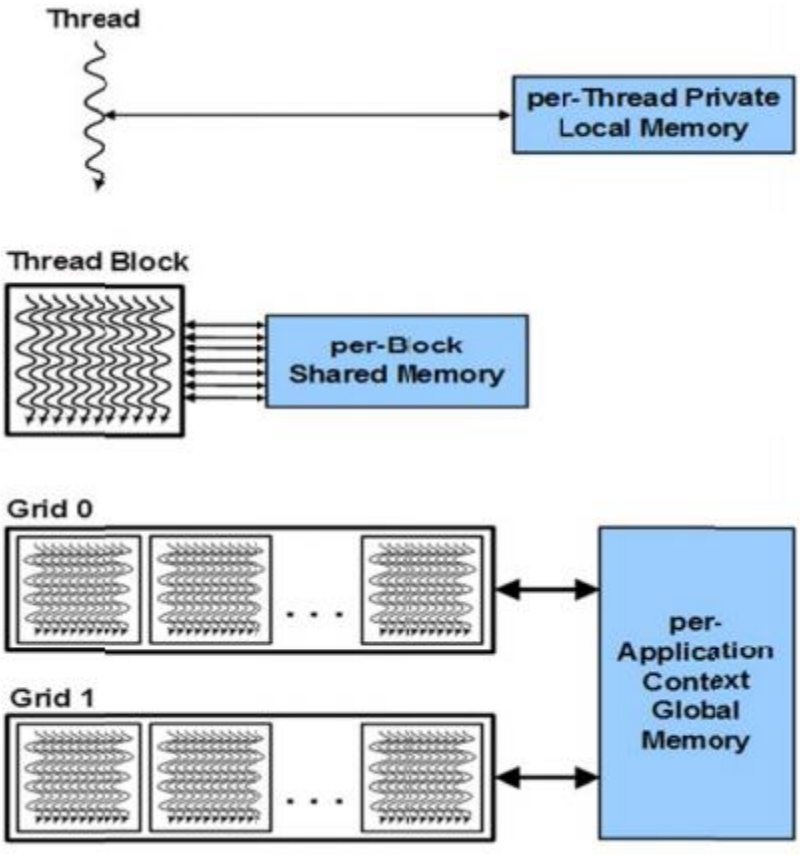
Työssäni ei tarvittu OpenGL:n shadereita, vaan ohjelmoijan tuli ainoastaan tarjota CUDA:lle pikselipuskuri, johon tulisi kirjoittaa pikselien väriarvot ja tämän jälkeen piirtää

kyseinen puskuri ruudulle. Asiaa käsitellään tarkemmin osiossa ”CUDA:n ja OpenGL:n yheistyö”.

3.2 CUDA

3.2.1 CUDA-arkkitehtuuri

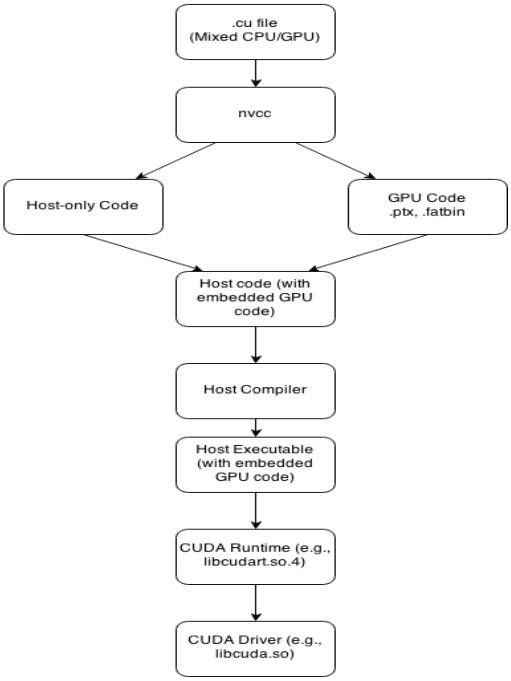
Compute Unified Device Architecture on NVIDIA:n kehittämä C-ohjelmointikielen kaltainen näytönohjainten rajapinta, jonka avulla voidaan ajaa rinnakkain funktioita monella eri säikeellä [21]. Nicholas Wilt (2013) kertoo kirjassaan The Cuda Handbook näytönohjaimella rinnakkain suoritettavista funktioista, joita kutsutaan CUDA:ssa kernel-funktioiksi. Kerneleitä ajetaan joko ohjelmoijan tai kääntäjän järjestämissä ruudukoissa (”grids”) jotka taas koostuvat säieblokeista (”thread blocks”) (kuva 3). [1.]



Kuva 3. CUDA-säie- ja muistihierarkia. [3]

Jokainen säie yhdessä säieblokissa suorittaa instanssin kernel-funktiosta. Tähän säikeeseen assosioidaan myös yksilöllinen säie- ja blokki-id, rekistereitä, säiekohtainen muisti (”per-thread private memory”), funktion parametrit (”inputs”) ja laskennan tulokset (”output results”). Säie- ja blokki-id:hin voidaan kernel-funktion sisällä viitata niille varatuilla avainsanoilla. Lisää säie- ja blokki-id:stä kerrotaan luvussa ”4.1 Säteenseuranta kernel -funktio”. Yhden säieblokin säikeet voivat kommunikoida vain saman blokin säikeiden kanssa käyttämällä hyväksi blokin jaettua muistia (”per-Block Shared Memory”). Jokaisen globaalin kernel-funktion synkronisoinnin jälkeen (”kernel-wide global synchronization”) blokit kommunikoivat keskenään globaalin muistin välityksellä (”per-Application Context Global Memory”). Kaikki tämä yhdistyy NVIDIA:n uusimpien näytönohjainten Kepler-arkkitehtuurin SMX-yksikössä, jossa voidaan suorittaa 2048 rinnakkaista säiettä. [1.]

CUDA kääntäjä (NVCC) pitää huolen siitä, että ”.cu”-päätteiset tiedostot jaetaan kahteen osaan heti käännösvaiheessa. Tiedostosta erotellaan näytönohjaimella ja prosessorilla ajettava koodi, jonka jälkeen kummatkin osat käännetään erikseen (kuva 4). Prosessorilla ajettava koodi käännetään omaan omaksi suoritettavaksi ohjelmakseen, ja CUDA käännetään niin sanotuksi PTX pseudo Assemblyksi, joka lisätään juuri luotuun suoritettavaan ohjelmaan merkkijonona. Vasta näytönohjaimen ajuri kääntää PTX:n binääriksi. Näin varmistutaan, että CUDA-ohjelmat toimivat myös tulevaisuudessa julkaistavilla näytönohjaimilla. [1.]



Kuva 4. CUDA-käännösprosessi. [1]

Juuri ennen ohjelman suoritusta näytönohjaimen ajuri kääntää itse PTX:n binääriksi. Tämän ansiosta konekäskyt ovat aina ajan tasalla, vaikka kyseistä näytönohjainta ei olisi ollut olemassa vielä ohjelmaa kirjoitettaessa. [1.]

3.2.2 Kernel -funktiot

NVIDIA on toteuttanut CUDA kernel -funktioiden ohjelmoinnin käyttäen C-,C++- tai Fortran-ohjelmointikieliä. C- ja C++-kielien tuet ovat melko kattavat, ja NVIDIA pyrkii joka CUDA-version mukana laajentamaan tukea entistä enemmän. Pieniä eroja kuitenkin on, esimerkiksi itse CUDA kernel -funktion kutsuminen tapahtuu NVIDIA:n itsensä lisäämällä laajennuksella, jossa käytetään erisuuruusmerkkejä (koodiesimerkki 1).

```
raytrace<<<blocks,threads>>>(devPtr, s, numSpheres);
```

Koodiesimerkki 1. CUDA kernel -funktiokutsun syntaksi.

Funktion nimen jälkeen tulevat erisuuruusmerkit sulkevat sisäänsä kaksi parametria, joista ensimmäinen muuttuja määrää, montako säieblokkia ohjelmamme suorittaa, ja toinen parametri määrittää säikeiden määrän säieblokkia kohden. Esimerkiksi jos blocks

-muuttujan arvo on 1 ja threads -muuttujan arvo on 10, suoritamme tällöin vain 10 rinnakkaista säiettä.

Kernel-funktioiden määritteleminen tapahtuu muuten samalla tavalla kuin C++:ssa, mutta funktion palautusarvon tyypin eteen lisätään ”__global__” -avainsana (koodiesimerkki 2). Myös seuraavat avainsanat ovat käytössä CUDA-funktioiden kanssa; ”__device__” ja ”__host__”. Device-avainsana määrittelee funktion, jota voidaan kutsua kernel-funktion sisältä, jolloin se suoritetaan vain näytönohjaimella. Host-avainsanalla tarkoitetaan tietokoneen prosessorilla ajettavaa funktiota. Nämä kaksi avainsanaa voidaan myös yhdistää funktiokutsussa, jolloin funktiota voidaan käyttää sekä tietokoneen prosessorilla että näytönohjaimella.

```
__global__ void raytrace(uchar4 *devPtr, Sphere * s, unsigned int SPHERES)
```

Koodiesimerkki 2. CUDA kernel -funktion määrittely.

CUDA-koodi kirjoitetaan erilliseen ”.cu”-päätteiseen tiedostoon, joka saa sisältää sekä tavallista C-, C++-, Fortran- että CUDA-koodia.

Kernel-funktiot voivat hyödyntää prosessorin tuottamaa dataa varaamalla ensin tarpeeksi muistia näytönohjaimelta, siirtämällä tarvittava datan käyttäen muistin kopiointia (”memcopy”) ja antamalla tämän jälkeen kernerille osoitin tähän juuri varattuun muistiin (koodiesimerkki 3).

```
Sphere *s = nullptr;
//Allocate memory from gpu
cudaMalloc((void **)&s,sizeof(Sphere)*SPHERES+sizeof(Vector3f) * SPHERES);
//Allocate memory from host
Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)*SPHERES);
//Copy models to device
cudaMemcpy(s,temp_s, sizeof(Sphere)*SPHERES,cudaMemcpyHostToDevice);
//Free the host side data if not needed anymore
free(temp_s);
```

Koodiesimerkki 3. Muistin varaus ja siirto näytönohjaimelle.

Koodiesimerkki 3:ssa esiteltyä osoitinta ”s” voidaan käyttää viittaamaan näytönohjaimen muistissa sijaitsevaan dataan, kuten koodiesimerkki 2:ssa on tehty. Erityisesti tulisi kiinnittää huomiota CUDA:n omiin malloc- ja memcopy-funktioihin, joilla tehdään muistin varaus ja siirto näytönohjaimelle.

3.3 CUDA:n ja OpenGL:n yheistyö

CUDA:n ja OpenGL:n yheistyöllä (Graphics Interoperability tai ”graphics interop”) tarkoitetaan toiminnallisuutta, jossa CUDA kernel -funktiot kirjoittavat dataa näytönohjaimen muistissa sijaitseviin OpenGL:n tietorakenteisiin, joita käytetään grafiikan piirtämiseen. Tämän yhteistyön ansiosta tietokoneen ei tarvitse tehdä prosessorin ja näytönohjaimen välillä hidasta muistin kopiointia PCI-E -väylän kautta. [1.] Tämä on tärkeää ohjelman suorituskyvyn kannalta, sillä näytönohjaimen sisäisen muistin siirtonopeus on parhaimmillaan 192.2 G/s [3], kun taas PCI-E-väylä harvoin ylittää 8G/s-nopeutta [4].

Näytönohjaimen ja OpenGL:n ajurit voivat koordinoida toimintaansa paremmin (koodiesimerkki 4), jos heti ohjelman alustusvaiheessa tehdään selväksi, että käytetään graphics interop -toimintoa.

```
void initCuda()
{
    //Setup Cuda device and map it to OpenGl
    cudaDeviceProp prop;
    int dev;
    memset(&prop, 0, sizeof(cudaDeviceProp));
    prop.major = 1;
    prop.minor = 0;

    //Find GPU with atleast 1.0 Cuda support and use that
    cudaChooseDevice(&dev,&prop);

    //Enable CUDA & OpenGl interop
    cudaGLSetGLDevice(dev);
}
```

Koodiesimerkki 4. CUDA:n ja OpenGL:n yhteistyön alustaminen.

Yhteistyön alustamisen jälkeen ohjelmoijan tulee varata Pixel Buffer Object (PBO) kernel -funktion ajon ajaksi CUDA:n käyttöön (koodiesimerkki 5). Jos OpenGL yrittää päästä käsiksi CUDA:lle varattuun resurssiin, seuraukset ovat arvaamattomat. Kun CUDA varaa PBO:n, cudaGraphicsMapResources()-funktio varmistaa, että kaikki OpenGL -grafiikkakutsut on suoritettu loppuun asti ennen lopullista varausta [6].

```
GLuint bufferObj;
//Bind the pixel buffer on which we will draw on
glGenBuffers(1,&bufferObj);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj);
//Use DYNAMIC_DRAW_ARB since the data changes all the time in the PBO
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, WINDOW_WIDTH * WINDOW_HEIGHT * 4, nullptr, GL_DYNAMIC_DRAW_ARB);

//Specify to cuda runtime that we want to use opengl pbo
cudaGraphicsResource *resource;
cudaGraphicsGLRegisterBuffer(&resource, bufferObj,cudaGraphicsMapFlagsNone);

//Map the graphics resource to cuda so we can manipulate it in our kernel
cudaGraphicsMapResources(1,&resource,nullptr);
cudaGraphicsResourceGetMappedPointer((void **) &devPtr, &size, resource);

//Run the kernel
rayTraceWrapper(grid,threads,devPtr,s,SPHERES);

//Unmap the resource from cuda for OpenGl to access draw it
cudaGraphicsUnmapResources(1,&resource, nullptr);
```

Koodiesimerkki 5. CUDA-ja OpenGL-yhteistyö, grafiikkaresurssien varaaminen.

4 Renderöijän implementointi

Työn toteutusosan tarkoituksena oli implementoida säteenseurantarenderöijä käyttämällä CUDA- ja OpenGL-rajapintoja. Toteutus alkoi määrittelemällä näytönohjaimella suoritettava säteenseuraajan kernel-funktio. Kernel-funktion määrittelyn jälkeen implementoitiin säteenseurannassa käytettävä kolmioiden törmäystarkistus, jonka jälkeen ladataan 3D-malli muistiin. Lopuksi malli sävytetään käyttäen Lambertian diffuse -funktiota.

4.1 Säteenseuraajan kernel-funktio

Säteenseuranta koostuu yhdestä pääfunktiosta nimeltä raytrace (käsitelty aikaisemmin koodiesimerkissä 2, luku 3.2.2). Tämä kernel-funktio ottaa vastaan ensimmäisenä parametrina osoittimen OpenGL:n alustamaan PBO:iin, johon lopuksi kirjoitetaan tarvittavat väritiedot kuvaa varten. Kernel-funktion toinen parametri on osoitin taulukkoon, joka sisältää Object-luokan olioita. Kolmas parametri on renderöitävien kolmioiden lukumäärä.

Jokaisella säikeellä ja säieblokilla on oma uniikki id, jonka ansiosta funktiossa ensimmäisenä voidaan laskea ammuttavan säteen ja pikselin x- ja y-koordinaatit. Tässä laskennassa hyödynnetään säikeen ja säieblokin omaa numerointia (koodiesimerkki 6).

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
if(offset > WINDOW_WIDTH * WINDOW_HEIGHT)
    return;
```

Koodiesimerkki 6. Lasketaan säteen aloituskoordinaatit.

Koordinaattien laskennan suorittamisen jälkeen voidaan lasketut pisteet keskittää niin, että origo on aina keskellä ruutua (koodiesimerkki 7).

```
//Center the rays
ray.x = (x - WINDOW_WIDTH/2);
ray.y = (y - WINDOW_HEIGHT/2);
ray.z = -1;
```

Koodiesimerkki 7. Säteiden keskittäminen.

Koska työn toteutuksessa käytetään suorakulmaista perspektiiviä (”orthogonal perspective”), ja kamera on toistaiseksi kiinnitetty yhteen paikkaan avaruudessa, voidaan mielivaltaisesti määrittää säteiden z-komponentin kulkemaan z-akselilla aina negatiiviseen suuntaan. Täten kamera katsoo aina negatiivista z-akselia pitkin siten, että x-akseli on vaakasuorassa ja y-akseli pystysuorassa.

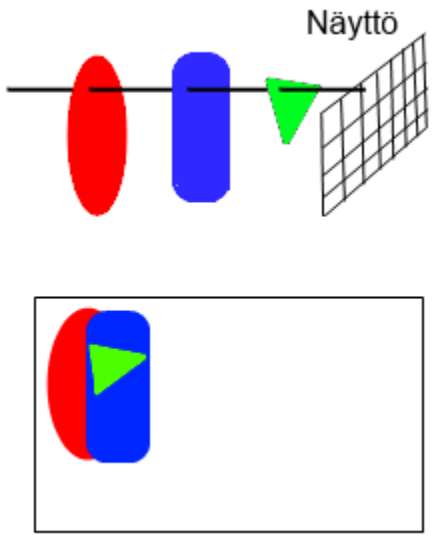
Säteenseuraajan pääsilmukka on loppujen lopuksi hyvin yksinkertainen. Pääsilmukassa käydään läpi kaikki geometriset objektit ja tarkistetaan, osuuko kyseessä oleva säde objektiin. Osuman tapahtuessa tallennetaan säteen ja kolmion törmäyskohdan väri talteen. Object-tietorakenteeseen kuuluu materiaalien määrittely, josta voidaan päätellä osumakohdan väri ennen varjojen tai muiden efektien laskemista. Pikselin lopullinen väri voidaan määritellä annettujen olosuhteiden mukaan, jos maailmassa on valoja, tulisi tässä kohtaa laskea potentiaaliset varjo- ja heijastus-kohdat. Sama prosessi käydään läpi jokaiselle kolmiolle. Vain kameraa lähimpänä oleva osumakohta tallennetaan piirtämistä varten (koodiesimerkki 8).

```
//Main intersection loop
for(int i=0;i<ObjectS;i++)
{
    //Decide intersection test
    if(s[i].type == SPHERE)
        s[i].intersection(ray,s,depth,hitd, false, ObjectS);
    else
        triangleIntersection(s+i,ray,hitd);

    if(hitd->distance > maxz)
    {
        //Remember our object of choice. Save index for reflection and Lambertian diffuse
        targetIndex = i;
        hitd->r  = s[i].r;
        hitd->g  = s[i].g;
        hitd->b  = s[i].b;
        maxz = hitd->distance;
    }
}
```

Koodiesimerkki 8. Säteenseuraajan kernel-funktion pääsilmukka törmäyksien tarkistusta varten.

Tällä saadaan esiin objektien asettuminen toisiinsa nähden syvyyssuunnassa (kuva 5).



Kuva 5. Tallennetaan vain viimeisen osumakohdan väri, jotta objektien syvyyserot näkyisivät kuvassa.

Kun varmistutaan, että juuri kyseessä olevan pikselin takana on jokin objekti, voidaan tarpeen tullessa ampua uusi säde esimerkiksi heijastuksien laskemiseksi (koodiesimerkki 9). Jos mallin lisäksi avaruudessa ei ole erillistä valonlähdettä, voi maailma näyttää liian tasaiselta (valo jakautuu tasaisesti mallin pinnalle), eikä monimutkaisia muotoja voida erottaa mallin pinnalla. Tämä ongelma voidaan ratkaista kolmioiden sävytyksellä.

Olen alustavasti toteuttanut sävytyksen käyttämällä Lambertin kosinilakia [11]. Lambertin kosinilain avulla pikselin väri voidaan tallentaa OpenGL:n tarjoamaan PBO-objektiin piirtämistä varten.

```
//The actual reflection cast
if(s[targetIndex].type == SPHERE)
{
    hitd->reflection = true;
    s[targetIndex].intersection(ray,s,depth,hitd, hitd->reflection, ObjectS);
}

float lambertian = calculateLambertianDiffuse(ray, s+targetIndex, maxz);
devPtr[offset].x = (lambertian * hitd->r * 255);
devPtr[offset].y = (lambertian * hitd->g * 255);
devPtr[offset].z = (lambertian * hitd->b * 255);
```

Koodiesimerkki 9. Lasketaan heijastus ja sävytys. Tallennetaan väritieto PBO:hon.

PBO voidaan piirtää heti kun säteenseurantaan käytetty kernel-funktio on palauttanut suorituksen prosessorille.

Säteenseurantaan käytetty kernel-funktion debuggaus ei ollut niin yksinkertaista. Yksi suurimmista ongelmista oli itse kernel-funktion suoritusaika. Windows-käyttöjärjestelmään on määritelty, että näytönohjain saa suorittaa oletuksena maksimissaan 2 sekuntia kerrallaan, ennen kuin Windows keskeyttää suorituksen ja käynnistää näytönohjaimen ajurit uudelleen (”TDR”, Timeout Detection and Recovery) [1; 12]. Tilanne on hankala varsinkin, kun kernel-funktioni pyörii ilman optimointeja ja debugprofiililla noin 120 – 600 s ajan. Jopa release-profiililla suoritus kestää 5 – 20 s ajan. Tämä oli mahdollista kiertää muuttamalla TDR -aikaa korkeammaksi. Huono puoli tässä on se, että tietokone on toimintakyvytön niin kauan, kunnes kernel-funktion suoritus on saatu loppuun.

4.2 Kolmion törmäystarkistus

Olen valinnut työhöni kolmion törmäystarkistukseksi yhden tämän hetken parhaimmista algoritmeista nimeltään Möller-Trumbore: Fast, Minimum Storage Ray/Triangle intersection [13; 14; 15]. Useat kolmion ja säteen törmäystarkistusalgoritmit käyttävät laskuihin tason kaavoja. Möller-Trumbore ei näin tee, ja tästä syystä sen käyttö vie hieman vähemmän muistia. Ainoastaan kolmion kulmapisteet säilötään muistissa, jolloin saadaan parhaimmillaan 50 % muistisäästöt [13]. Möller-Trumbore onkin siis erinomainen kompromissi muistin ja nopeuden kannalta.

Törmäystarkistusalgoritmit perinteisesti laskevat, osuuko säde tasolle, jolla itse kolmio sijaitsee, ja vasta tämän jälkeen, osuuko säde kolmion rajaaman alueen sisälle [13]. Möller-Trumboren tekniikassa kolmion kärki asetetaan origoon, ja muut kärkipisteet esitetään y- ja z-akseleilla yksikkövektoreina. Samalla myös säde käännetään kulkemaan x-akselin suuntaisesti.

Voimme esittää säteen kaavalla 1.

$$R(t) = O + tD \tag{1}$$

Kaavassa O on säteen alkupiste (”Origin”), t on säteen pituus ja D säteen suunta (”Direction”). Tämän yhtälön avulla voidaan esittää säteen osumakohta barysentrisillä koordinaateilla (u,v), koska piste T(u,v) kolmiolla saadaan kaavasta 2, jossa V_0, V_1 ja V_2 ovat kolmion kärkipistevektorit (kaava 2).

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2 \tag{2}$$

Kaavassa u ja v ovat barysentrisiä koordinaatteja, joiden arvo vaihtelee välillä u ≥ 0, v ≥ 0 ja u + v ≤ 1. Törmäyspisteen barysentrisiä koordinaatteja hyödynnetään myös teksturoimiseen. Säteen, R(t), ja kolmion, T(u,v), törmäyspiste saadaan seuraavasti: R(t) = T(u,v), josta saadaan kaava 3.

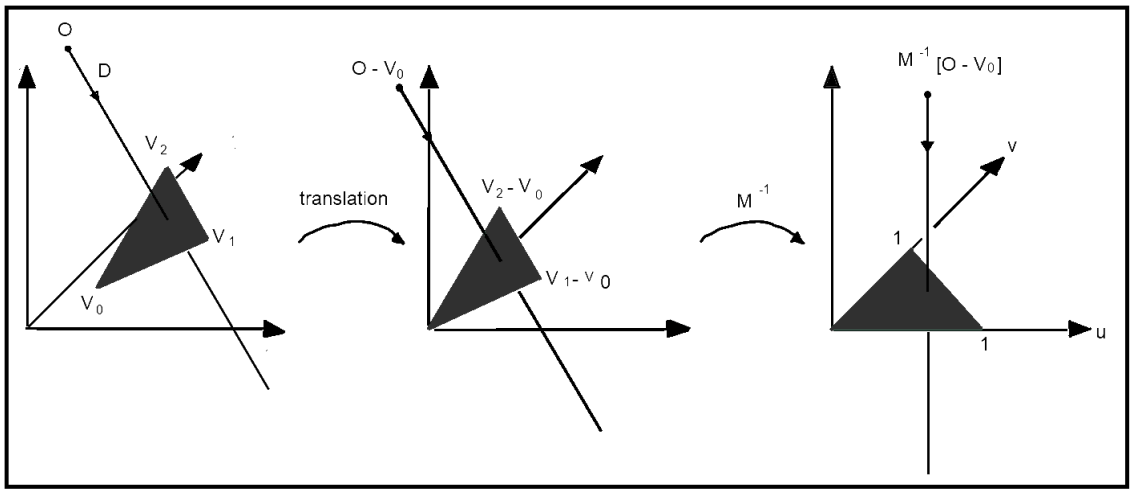
$$O + tD = (1-u-v)V_0 + uV_1 + vV_2 \tag{3}$$

Järjestämällä termit uudelleen saadaan kaava 4.

$$[\,-D,\ V_1 - V_0,\ V_2 - V_0\,] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{4}$$

Kaavasta 4 nähdään, että barysentriset koordinaatit (u,v) ja etäisyys (t) säteen lähtöpaikasta törmäyskohtaan saadaan ratkomalla lineaarinen yhtälöryhmä (kaava 4).

Merkitsemällä $M = [\,-D,\ V_1 - V_0,\ V_2 - V_0\,]$ saadaan algoritmista kuvan 6 tapainen esitys (kuva 6).



Kuva 6. Kolmion pingottaminen y- ja z-akseleille. Säde O suuntaan D käännetään x-akselin suuntaisesti. (Kuva muokattu Möller-Trumbore -työn mukaan [13])

Implementaatiooni Möller-Trumbore-algoritmistä voi tutustua liitteessä 1.

4.3 Wavefront OBJ -tiedoston lataus

Lopputyöni aiheena oli implementoida säteenseurantarenderöijä, joten en nähnyt tarpeelliseksi lähteä itse implementoimaan 3D-mallien lataamista. Sen sijaan päädyin käyttämään mahdollisimman yksinkertaistettua latauskirjastoa. Kirjaston nimi on Tiny obj loader, jonka päämäärä on yksinkertainen: yhden tiedoston obj-lataus muistiin mahdollisimman tehokkaasti [22]. Kirjasto on toteutettu käyttämällä ainoastaan C++-standardikirjastoa. Työtä varten oli toteutettu oma tietorakenne kolmion esittämistä varten, koska kolmioiden törmäystestaus aloitettiin ennen 3D-mallien latausta. Tästä aiheutui, että jouduin ensin lataamaan OBJ-tiedostot tietokoneen muistiin käyttäen Tiny obj loaderia, jonka jälkeen ladattu data täytyi muuttaa käyttämääni muotoon. OBJ-tiedostoilla mallinnetaan 3D-objektia käyttämällä sen kulmapisteitä.

OBJ-spesifikaatio on pelkistettynä seuraavan lainen:

- Data esitetään ASCII-muotoisena tekstitiedostona.
- Kommenttirivit alkavat risuaidalla (#).

- Jokainen kulmapiste rivi alkaa kirjaimella v (”Vertex”).

- Kolmion/monikulmion pintaa kuvataan f-kirjaimella (”Face”).

OBJ-tiedostossa jokainen viittaus kulmapisteisiin indeksoidaan alkaen numerosta 1 (kuva 7).

```
# Raytracer in Cuda static test scene
# Olli Koskinen

v 150.0 100.0 -1.00001
v 150.0 250.0 -1.00001
v 250.0 100.0 -1.00001

f 1 2 3
```

Kuva 7. Pelkistetty Wavefront OBJ -tiedosto. Kolmion pinta kuvataan kolmen kärkipisteen avulla.

Käyttäessäni Tiny obj loader -kirjastoa ongelmaksi muodostui, että kirjasto ei aivan noudattanut OBJ-tiedostospesifikaatioita, vaan latasi tiedot niin, että indeksointi alkoi yhden sijaan nollasta. Tiedostoa ladatessani oletin, että käyttämäni Tiny obj loader -kirjasto noudattaisi OBJ-tiedostospesifikaatiota, vaikka myöhemmin selvisi, ettei näin ollutkaan. Toinen ongelma tiedostojen lataamisessa oli oma virheellinen käsitys pintojen indeksoinnista. OBJ -spesifikaation mukaan pintamäärittelyt (f-rivit) eivät välttämättä viittaa kulmapisteisiin täysin lineaarisessa järjestyksessä vaan esimerkiksi pinta saattaa muodostua aivan satunnaisista kulmapisteistä.

Naiivi tapa yrittää ladata kaikki kulmapisteet järjestyksessä ei siis aivan toiminut. Lähdin selvittämään ongelmia kirjoittamalla datakonversion uudelleen ottaen huomioon kaksi edellä mainittua ongelmaa (koodiesimerkki 10).

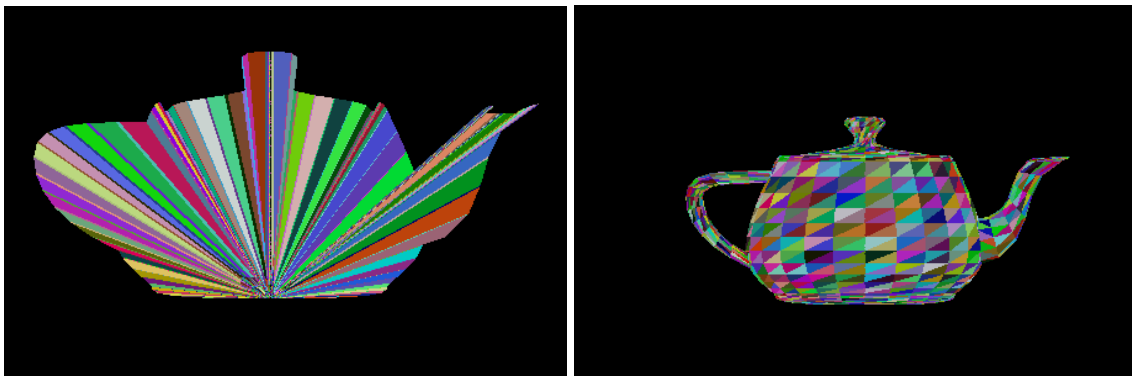
```
unsigned int triangleCounter = 0;
for (size_t v = 0; v < loadedObject[0].mesh.indices.size(); v+=3)
{
    //For every three vertex we increase our counter
    if (v != 0 && v % 3 == 0)
        triangleCounter++;
    //Give triangles random colours for funzies
    model[triangleCounter].type = TRIANGLE;
    model[triangleCounter].r = rnd(0.8f);
    model[triangleCounter].g = rnd(0.8f);
    model[triangleCounter].b = rnd(0.8f);

    //Convert the indexed vertices to our model
    model[triangleCounter].pos[0].x = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+0]*3)+0];
    model[triangleCounter].pos[0].y = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+0]*3)+1];
    model[triangleCounter].pos[0].z = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+0]*3)+2];
    model[triangleCounter].pos[1].x = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+1]*3)+0];
    model[triangleCounter].pos[1].y = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+1]*3)+1];
    model[triangleCounter].pos[1].z = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+1]*3)+2];
    model[triangleCounter].pos[2].x = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+2]*3)+0];
    model[triangleCounter].pos[2].y = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+2]*3)+1];
    model[triangleCounter].pos[2].z = loadedObject[0].mesh.positions[(loadedObject[0].mesh.indices[v+2]*3)+2];

    //Calculate triangle surface normal
    Vector3f v1 = model[triangleCounter].pos[1] - model[triangleCounter].pos[0];
    Vector3f v2 = model[triangleCounter].pos[2] - model[triangleCounter].pos[0];
    model[triangleCounter].normal = v1.crossproduct(v2);
}
```

Koodiesimerkki 10. Konvertoidaan Tiny obj loader -kirjaston lataama data yhteensopivaksi renderöijän kanssa kolmion kärkipisteiden eri indeksien järjestyksestä välittämättä.

Ongelman korjauksen jälkeen selvisi, että osa kolmion kärkipisteistä skaalautui väärin (kuva 8). Tätä ei näkynyt vielä alkuperäisessä muutaman kolmion testimallissani, jonka olin kovakoodannut mukaan ohjelman alustukseen. Monistin tämän testimallin omaan OBJ -tiedostoon, jonka latauduttua selvisi, että ammuttu säde ei kulje aivan z-komponentin suuntaisesti niin kuin oli tarkoitus. Ongelma ilmeni säteen suuntavektorin z-komponentin ollessa pienempi kuin -1. Kuvassa tämä näkyi kärkipisteiden skaalautumisena virheellisesti origoa kohti. Kärkipisteet, joiden paikkavektorin z-komponentti oli < -1, skaalautuivat pienemmiksi suhteessa etäisyyteen. Tästä johtui, että ladatut monimutkaiset mallit näyttivät mallin silhuetin oikein muiden pisteiden ollessa väärin.



Kuva 8. Vasen kuva: kolmioiden kärkipisteet skaalautuvat virheellisesti origoa kohti säteen suuntavektorin määrittelyvirheen takia. Oikea kuva: skaalaus korjattu, säde kulkee vain z-komponentin suuntaisesti.

Kaikki nämä ongelmat johtuivat työssä käytetystä staattisesta ja ortogonaalisesta kamerasta, joka katsoi vain negatiivista z-akselia pitkin. Näin jokaisen säteen tuli kulkea vain z-akselin suuntaisesti. Alkuperäisessä versiossa asetin säteelle x- ja y- komponentit säikeiden juoksevan numeroinnin mukaan ja laskin tästä säteen suunnan. Ongelma ratkesi asettamalla säteen paikkavektori säikeiden juoksevien numeroiden mukaan, mutta pakottamalla suuntavektorin ainoaksi komponentiksi z = -1. Tämän jälkeen mallit näkyivät oikein ruudulla (kuva 5, oikea kuva).

Wavefront OBJ -tiedoston latauksessa ilmenneiden kolmen ongelman korjauksen seurauksena onnistuin piirtämään ladattuja 3D-malleja näytölle. Seuraava ongelma olikin mallien valaistuksen puuttuminen, ja tästä johtuva mallien yksityiskohtien vaikea hahmottaminen.

4.4 Mallien sävytys

Mallien yksityiskohtien näkyvyyden kannalta on tärkeää, että samassa tilassa mallin kanssa on ainakin yksi valonlähde tuomassa sävyeroja mallin pinnalle. Tämä olikin seuraava askel työssäni. Toteutin sävytyksen tekniikalla nimeltä ”Lambertian Diffuse”, joka perustuu Johann Heinrich Lambertn kehittämään Lambertin kosinilakiin [11]. Laki määrittää, että pinnan sävy on suoraan verrannollinen valonsäteen ja pinnan normaalin kulman kosiniin [11].

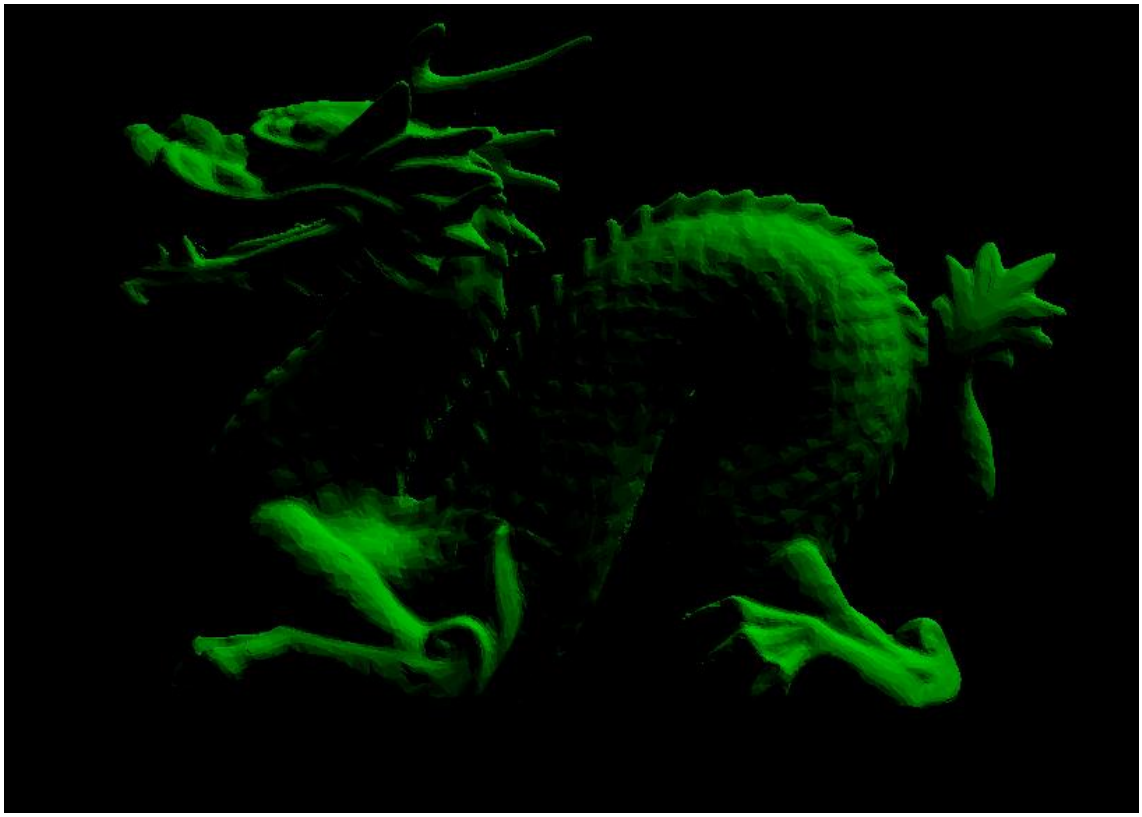
$$L \cdot N = |N||L| \cos \alpha = \cos \alpha \quad (5)$$

Toistaiseksi mallissani on vain yksi staattinen valonlähde, mutta jatkokehityksenä on tarkoitus mahdollistaa täysin dynaamiset valonlähteet (koodiesimerkki 11).

```
//Lambertian diffuse shading. Take the cosine of triangle s's surface normal and the light direction
__device__ float calculateLambertianDiffuse(Object *s, Vector3f &lightSource)
{
    //Vector3f normalized = s->normal.normalize();
    Vector3f v = lightSource.direction();
    return fmaxf(s->normal.dotproduct(v), 0);
}
```

Koodiesimerkki 11. Lambertin sävytys. Kolmion pintanormaalin ja valonsäteen suunnan kosini.

Sävytyksessä varmistetaan myös, että sävy on aina vähintään 0. Sävytyksen tuloksena on huomattavasti tarkempi 3D-malli, jossa yksityiskohdat ovat erotettavissa (kuva 9).



Kuva 9. Kuvan lohikäärme on toteutettu Lambertian diffuse -sävytyksellä. Tätä sävytystä kutsutaan flat shadingiksi.

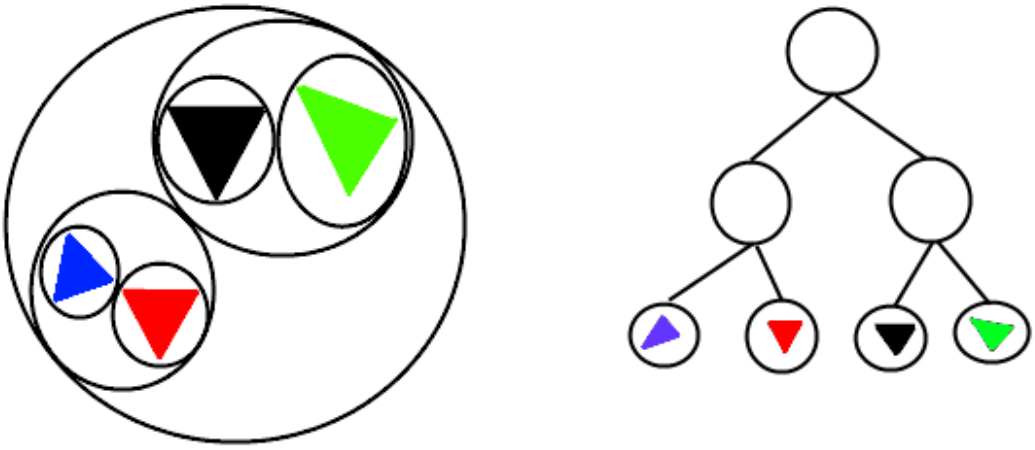
Lambertian diffuse on ns. flat shading -tekniikka, joka toteuttaa sävytyksen kolmioiden avulla pehmentämättä niiden reunoja luoden karkean näköisen pinnan. On olemassa myös ns. Phong shading -tekniikka, joka pehmentää kolmioiden ääriviivoja saaden sävytyksen huomattavasti tasaisemmalta [23].

5 Optimointi

Optimoinnilla tarkoitetaan toiminnan suorittamisen nopeuttamista toimintatapoja muuttamalla. Tässä luvussa käsitellään kahta yleisintä optimointitapaa, joita käytetään näytönohjaimella suoritettavassa säteenseurannassa: Bounding Volume Hierarchyä (BVH) ja CUDA:n Texture memoryä. BVH on kiihdytystietorakenne ja Texture memory on NVIDIA:n kehittämä muistiarkkitehtuuri. Näitä menetelmiä voidaan käyttää nopeuttamaan törmäystarkistukseen käytettävien kolmioiden etsimistä.

5.1 Bounding Volume Hierarchy

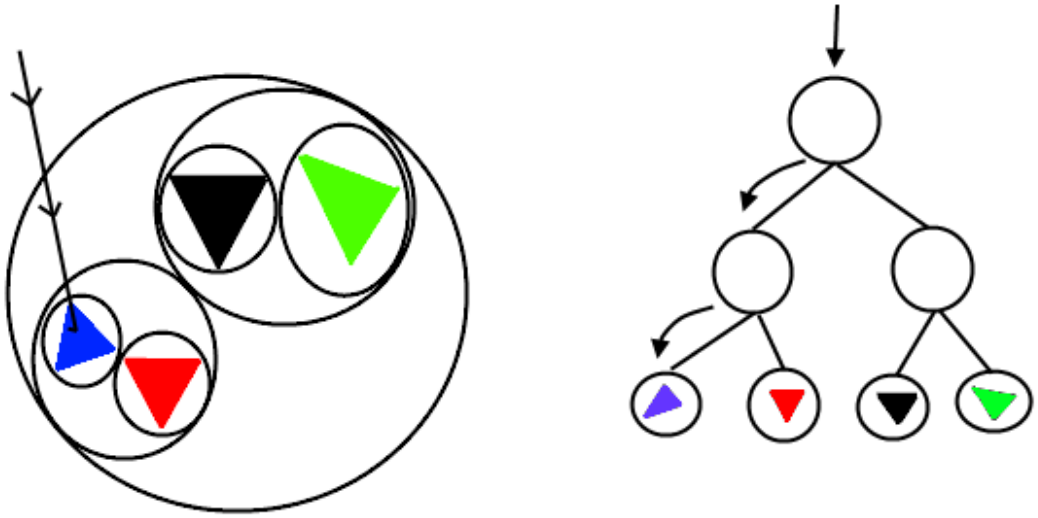
Bounding Volume Hierarchy eli BVH on säteenseurannassa tällä hetkellä eniten käytetty kiihdytystietorakenne [17]. BVH koostuu solmuista. Nämä solmut sisältävät aina ns. ”bounding volumen” eli rajatun alueen, joka sulkee sisäänsä kaikki muut sen alueen lapsisomut. Nämä rajatut alueet voivat olla esimerkiksi neliöitä, kuutioita tai palloja. Tässä mallissa lopulta lehtisolmut sulkevat sisäänsä aina objektin, jota vastaan törmäystarkistus halutaan suorittaa. Näitä objekteja voivat olla neliöt, kolmiot tai vaikka kokonaiset 3D-mallit (kuva 10). BVH:ta rakennettaessa pyritään maailma jakamaan osiin niin, että objektit, jotka ovat fyysisesti rinnakkain, tulisivat myös puurakenteessa samoihin tai vierekkäisiin solmuihin.



Kuva 10. Kolmitasoinen Bounding Volume Hierarchy sulkee sisäänsä kolmiot. Jokainen taso on tässä tapauksessa ympyrä, joka sulkee sisäänsä lapsisolmujen rajaavat muodot.

Kiihdytystietorakenteet, kuten BVH, on kehitetty vähentämään säteenseurannassa tehtävien törmäystarkistuksien määrää. Naiivissa törmäystarkistuksessa jokaisen säteen tulee tehdä törmäystarkistus jokaista objektia vastaan. Tällä törmäystarkistus määrittää, mihin kolmioon säde osuu, jos se ylipäänsä osuu ollenkaan. Puurakenteen myötä renderöijän dynamiikka hieman muuttuu. Aikaisemmin törmäystarkistukset tehtiin vain kolmioita vastaan, kun taas puurakenteen myötä suurin osa törmäystarkistuksista tehdään rajattuja alueita vastaan. Sädettä ammuttaessa tulee ohjelman kulkea puuta pitkin tutkien samalla kumpaan lapsihaaraan, jos kumpaankaan, tulisi siirtyä. Oikea

haara päätetään tekemällä yksinkertainen törmäystesti solmun rajaavia muotoja kohtaan (kuva 11).



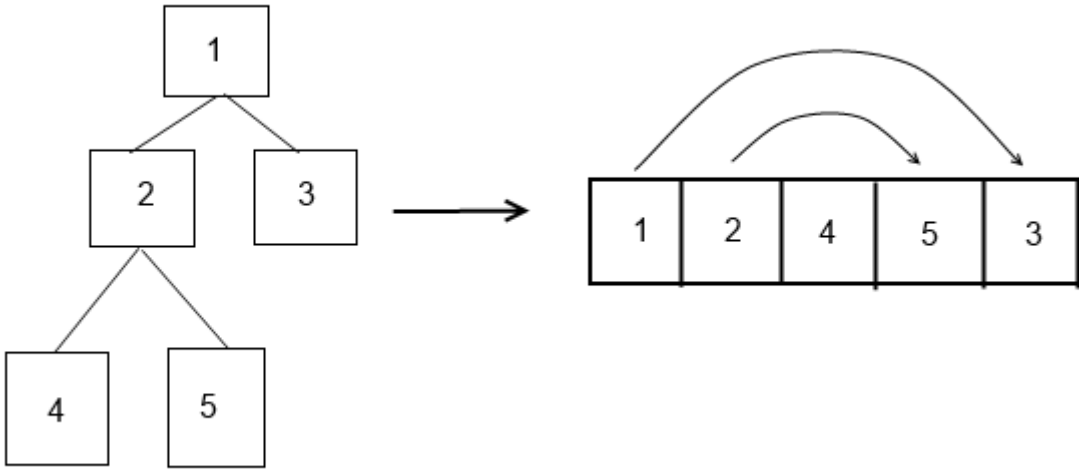
Kuva 11. Jos säde osuu BVH:n suureen juurisolmuun, osuu se mahdollisesti myös muihin objekteihin.

Primitiivimuotojen, kuten kolmioiden, etsiminen puusta on yksi BVH:n tärkeimmistä osista [18]. Puu tulee rakentaa siten, että se toimii mahdollisimman tehokkaasti. Tehokkaan puun rakennukseen kuuluvat seuraavat vaiheet:

- Jokaiselle primitiivimuodolle lasketaan sen ympäröivä rajausalue.
- Jaetaan maailma siten, että samalla alueella olevat primitiivit suljetaan saman rajausalueen sisään.
- Toistetaan rekursiivisesti kunnes jäljellä on vain yksi iso laatikko.

Tuloksena on puuhierarkia, jossa jokainen solmu viittaa sisältämiinsä lapsisolmuihin tai primitiiviin.

Puun rakenteen takia sen yksittäisten solmujen sijainti muistissa ei ole aina täysin ideaali. Tästä johtuen puu tulee muuttaa vielä yksiulotteiseksi taulukoksi. Taulukkomuotoon muutettaessa BVH:n rakenteesta tulee huomattavasti tiiviimpi, mikä parantaa välimuistiin tehtävien hakujen määrää ja samalla suorituskykyä (kuva 12) [18].

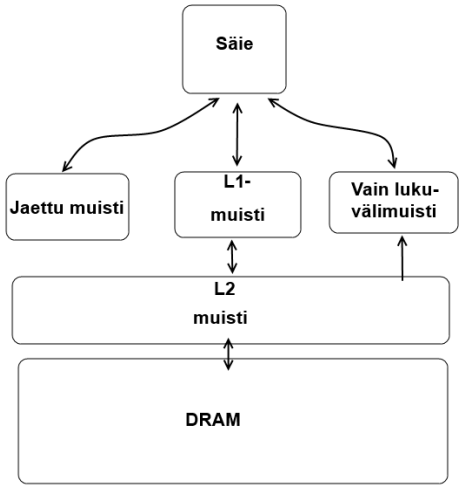


Kuva 12. BVH tiivistetään käyttäen syvyys-ensin-hakua. Tiivistyksen jälkeen ensimmäinen lapsisolmu löytyy heti seuraavana muistista. [19]

Tiivistyksen jälkeen ensimmäinen lapsisolmu löytyy muistista heti seuraavasta paikasta, ja toinen lapsisolmu löytyy sille annetun siirtoluvun (offset) päästä.

5.2 CUDA Texture Memory

CUDA Texture Memory on NVIDIA:n Kepler-näytönohjaimilla sijaitseva erikoismuisti, joka on tarkoitettu vain lukua varten. Tämä soveltuu hyvin BVH:n säilyttämiseen, sillä sitä vasten tehdään suuria määriä lukuja, mutta ei kirjoituksia. Ensimmäinen haku tekstuurimuistista on kohtalaisen hidas, mutta kaikki hakujen tulokset säilytetään huomattavasti nopeammassa 48 KT:n kokoisessa vain lukuun tarkoitetussa välimuistissa seuraavia hakuja varten [20]. Tämä muisti on täysin erillinen esimerkiksi L1- ja Shared-muistista (kuva 13).



Kuva 13. Kuvaus Kepler-arkkitehtuurin muistirakenteesta. [20]

Muistin lokaalisuuden ollessa heikko muistiin tehtäviä hakuja tulee myös huomattavasti enemmän.

Tekstuurimuistin käyttäminen on helppoa CUDA:lla. Määritellään tekstuurireferenssi, jota käytetään näytönohjaimella ajattavassa kernel -funktiossa, ja kiinnitetään tekstuurireferenssi haluttuun dataan (koodiesimerkki 12).

```
//Allocate enough memory for our primitives from the GPU
#define PRIMITIVES 1024
float4 *device_pointer;
cudaMalloc((void **)&d_ptr,sizeof(float4)*PRIMITIVES);

//....Copy primitive data specified somewhere else to GPU with cudaMemcpy..

//Setup texture reference for texture memory acces
texture<float4,1,cudaReadModeElementType> textureRef;

//Bind the texture to memory
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float4>();
cudaBindTexture2D(NULL,textureRef,device_pointer,channelDesc,PRIMITIVES * sizeof(float4));
```

Koodiesimerkki 12. Tekstuurimuistin alustaminen ennen käyttämistä. Alustus tapahtuu prosessorin puolella.

Tekstuurimuistista voidaan hakea tietoa heti alustuksen jälkeen käyttäen yhtä ainoaa funktiota CUDA kernel -funktiossa (koodiesimerkki 13).

```
__global__ raytrace() {
//...
float x = tex1d(textureREf,offset).x;
//..
}
```

Koodiesimerkki 13. Tiedon hakeminen CUDA-tekstuurimuistista.

Tekstuurimuistia käyttämällä varmistetaan tarvittavan tiedon pysyminen välimuistissa, mikä nopeuttaa ohjelman suoritusta vältyttäessä turhilta muistihauilta.

Projektissa toteutettiin Bounding Volume Hierarchy -kiihdytystietorakenne, mutta sen tuomat edut eivät päässeet esille viimehetken bugien takia.

6 Pohdinta

Työn tavoitteena oli syventyä säteenseurannan teoriaan, sen toteutukseen ja optimointiin. Aiheista löytyy runsaasti kirjallisuutta, ja sen avulla sain koottua kattavan tietopaketin asiasta kiinnostuneille. Samalla opin paljon säteenseurannan teoriasta ja toetutuksesta,

sekä näytönohjaimen ohjelmoimisesta CUDA:lla. Optimoinnin osalta sain kerättyä paljon tietoa teoriasta, mutta käytännön toteutus vaatii vielä jatkoperehtymistä. Varsinkin Bounding Volume Hierarchyn implementointi oli odotettua haastavampaa. Puurakenteen oikeellisuuden tarkistaminen on hyvin työlästä sen monimutkaisuuden takia. Toinen ongelma BVH:n kanssa oli myös itse kolmioiden etsiminen puusta riittävän tehokkaasti.

Työn edetessä huomasin, että itse BVH:n rakentamisen ja etsinnän tutkimiseen olisi ollut hyvä käyttää enemmän aikaa ennen ohjelmoinnin aloittamista. Käytetyt tietorakenteet eivät täysin sopineet käytettyihin optimointitekniikoihin, mikä tuotti ylimääräistä päänvaivaa työn loppuvaiheessa. Työtä tehdessä kiinnostuin myös näytönohjaimen muistiarkkitehtuurista ja myös siihen olisi ollut hyvä tutustua enemmän jo heti työn alkuvaiheessa.

Uskon työstäni olevan hyötyä henkilöille, jotka ovat vasta tutustumassa säteenseurannan konsepteihin ja haluavat tutusta näytönohjaimella suoritettuun säteenseurantaan. Työssäni käsitellyt koodinpätkät ovat vain ohjenuoria tärkeimpiin konsepteihin sekä säteenseurannassa että CUDA:n suorituksessa.

Lähteet

1 Wilt N. 2013. The Cuda Handbook. Crawfordsville, Indiana: RR Donnelley.

2 NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110, [PDF], < http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, Viitattu: 7.4.2014.

3 GeForce GTX 670 Specifications, [verkkodokumentti] <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-670/specifications> Viitattu: 8.4.2014.

4 PCI Express Base Specification Revision 3.0, [PDF] <http://komposter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf> Viitattu: 8.4.2014

5 Whitepaper NVIDIA GeForce GTX 680 The fastest, most efficient GPU ever built, [PDF]<http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf> Viitattu: 8.4.2014.

6 CUDA Toolkit Documentation, [verkkodokumentti] < http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__INTEROP.html#group__CUDART__INTEROP_1gb7064fb72e54d89d0666e192b45d35cc > Viitattu: 8.4.2014.

7 What is OpenGL, [verkkodokumentti] <https://www.opengl.org/wiki/FAQ#What_is_OpenGL.3F> Viitattu 9.4.2014.

8 What is NOT OpenGL?, [verkkodokumentti] <https://www.opengl.org/wiki/FAQ#What_is_NOT_OpenGL.3F> Viitattu: 9.4.2014.

9 Glassner AS. An Introduction to Ray Tracing, 9. painos. San Francisco, CA 2002: Morgan Kaufmann Publishers, Inc. Saatavissa: <http://books.google.fi/books?id=YPblYyLqBM4C&printsec=frontcover&hl=fi#v=onepage&q&f=false>.

10 Hughes JF, van Dam A, McGuire M, Sklar DF, Foley JD, Feiner SK, Akeley K. Computer Graphics Principles And Practice, 3. painos. Willard, Ohio 2014:RR Donneley.

11 Lambert's cosine law, [verkkodokumentti] http://en.wikipedia.org/wiki/Lambert's_cosine_law Viitattu: 10.4.2014.

12 Timeout Detection and Recovery of GPUs (TDR), [verkkodokumentti] <http://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff570088(v=vs.85).aspx> Viitattu: 10.4.2014.

13 Möller T, Trumbore B. Fast, Minimum Storage Ray/Triangle Intersection, Saatavissa: [PDF] <http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf> Viitattu 10.4.2014.

14 Shumskiy V, Parshin A. GPU Ray Tracing – Comparative Study of Ray-Triangle Intersection, GraphiCon'2012. Saatavissa: [PDF] < http://www.graphicon.ru/proceedings/2012/conference/EN2%20-%20Graphics/gc2012Shumskiy.pdf> Viitattu: 14.04.2014.

15 Segure RJ, Feito FR. Algorithms To Test Ray-Triangle Intersection. Comparative Study, Departamento de Informática, Universidad de Jaén 2001. Saatavissa: [PDF] <http://wscg.zcu.cz/wscg2001/Papers_2001/R75.pdf> Viitattu: 14.04.2014.

16 Rubin SM, Whitted T. A 3-Dimensional Representation for Fast Rendering of Complex Scenes, ACM 1980. Saatavissa: [PDF] < http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.6937&rep=rep1&type=pdf > Viitatu: 24.04.2014.

17 Karras T, Aila T. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies, NVIDIA 2013. Saatavissa: [PDF] < https://mediatech.aalto.fi/~timo/publications/karras2013hpg_paper.pdf > Viitattu: 24.04.2014.

18 Chang AY. A survey of Geometric Data Structures for Ray Tracing, Department of Computer and Information Science, Polytechnic University, Brooklyn, 2001. Saatavissa: [PDF] < http://web1.poly.edu/cse/_doc/technical/tr-cis-2001-06.pdf > Viitattu: 24.04.2014.

19 Pharr M, Humphreys G. Kappale 4: Primitives and Intersection Acceleration, kirjassa: Physically Based Rendering, Saatavissa: [PDF] < http://www.pbrt.org/chapters/pbrt-2ed-chap4.pdf > Viitattu 24.04.2014.

20 NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, [PDF] <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf> Viitattu 24.4.2014.

21 NVIDIA's CUDA Page, [verkkodokumentti] <http://www.nvidia.com/object/cuda_home_new.html> Viitattu: 1.5.2014.

22 Tiny obj loader, [verkkodokumentti] <http://syoyo.github.io/tinyobjloader/> Viitattu: 1.5.2014.

23 Illumination & Shading, [PDF] <http://www.vis.uky.edu/~ryang/teaching/cs535-2012spr/Lectures/09-Shading.pdf> Viitattu:1.5.2014.

24 Introduction to 3D-graphics, [verkkodokumentti] <http://schabby.de/introduction-to-3d-graphics/> Viitattu: 1.5.2014.

25 Real-Time Raytracing, [verkkodokumentti] <http://blog.codinghorror.com/real-time-raytracing/> Viitattu: 1.5.2014.

Möller-Trumbore-kolmion törmäystarkistus näytönohjaimella.

```
//Intersect ray with triangle using Möller - Trumbore method
//Introduced in   "Fast, Minimum Storage Ray/Triangle Intersection" study.
__device__ bool triangleIntersection(  Object *s,  // Triangle vertices
                                       Vector3f  &O,  //Ray origin
                                       HitData *hitData ){
    Vector3f edge1, edge2,P, Q, T;
    Vector3f D = Vector3f(0,0,-1);  //Static camera direction always down the -z axel
    float det, inv_det, u, v, t;

    //Find vectors for two edges sharing V1
    edge1 = s->pos[1] - s->pos[0], edge2 = s->pos[2] - s->pos[0];
    //Begin calculating determinant - also used to calculate u parameter
    P = D.crossproduct(edge2);

    //if determinant is near zero, ray lies in plane of triangle
    det = edge1.dotproduct(P);
    //NOT CULLING
    if(det > -EPSILON && det < EPSILON){
        hitData->distance = -INF;
        return false;
    }
    inv_det = 1.0f / det;
    //calculate distance from V1 to ray origin
    T = O - s->pos[0];
    //Calculate u parameter and test bound
    u = T.dotproduct(P) * inv_det;
    //The intersection lies outside of the triangle
    if(u < 0.0f || u > 1.0f){
        hitData->distance = -INF;
        return false;
    }
    //Prepare to test v parameter
    Q =  T.crossproduct(edge1);
    //Calculate V parameter and test bound
    v = D.dotproduct(Q) * inv_det;
    //The intersection lies outside of the triangle
    if(v < 0.0f || u + v  > 1.0f) {
        hitData->distance = -INF;
        return false;
    }
    t = edge2.dotproduct(Q) * inv_det;

    if(t > EPSILON) { //ray intersection
        //update hit only if we are closer than the previous hit
        if (hitData->distance <= (-INF+EPSILON) || t < hitData->distance)
        {
        Vector3f lightSource = Vector3f(50.0f,75.0f,1.0f);
        float lambertian = calculateLambertianDiffuse(s, lightSource);
        hitData->distance = t;
        hitData->scale = 1;
        hitData->r = lambertian * s->r;
        hitData->g = lambertian * s->g;
        hitData->b = lambertian * s->b;
        }
        return true;
    }

    // No hit
    hitData->distance = -INF;
    return false;
}
```