



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Globális illumináció megvalósítása GPU alapú sugárkövetéssel

Fodor Péter
MSc Mérnök Informatikus

Konzulens:
Dr. Mileff Péter
egyetemi adjunktus
Általános Informatikai Tanszék

MISKOLC, 2013

Tartalomjegyzék

1. Bevezetés	3
2. Képszintézis	4
2.1. Inkrementális képszintézis	4
2.2. Sugárkövetés	5
2.2.1. Sugár és háromszög metszéspontjának számítása	5
2.2.2. Térfelosztási módszerek	7
2.3. Globális illumináció	9
2.3.1. Fotometria	9
2.3.2. Fény-felület kölcsönhatás	10
2.3.3. Monte-Carlo integrálás	11
2.3.4. BRDF modellek	12
2.3.5. Inverz fényútkövetés (<i>path tracing</i>)	13
3. GPGPU programozás	14
3.1. A grafikus processzor általános célú programozása	14
3.2. Programozás CUDA környezetben	15
3.2.1. CUDA <i>kernel</i>	15
3.2.2. Szál hierarchia	15
3.2.3. Memóriakezelés	16
4. Fejlesztői dokumentáció	19
4.1. Háromszögmodellek tárolása	19
4.2. A gyorsítófa felépítése és tárolása	21
4.3. Inverz fényútkövetés a GPU-n	23
4.3.1. Folyam préselés (<i>stream compaction</i>)	25
4.3.2. A számítást végző CUDA kernelek	26
4.4. Eredmények	29
5. Összefoglalás	31
Irodalomjegyzék	32

A bemutatott kutató munka a TÁMOP-4.2.1.B-10/2/KONV-2010-0001 jelű projekt részeként az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

1. fejezet

Bevezetés

A képelőállító grafikai alkalmazásokban gyakran használt lokális árnyalási modellek elhanyagolják, vagy csak nagyon durva egyszerűsítésekkel helyettesítik a felületek közötti többszörös fényvisszaverődésből adódó indirekt megvilágítást. Ennek elsődleges oka, hogy a közvetett megvilágítás pontos meghatározása igen számításigényes feladat. Habár már az egyszerűsítésekkel is el lehet érni egy elfogadható valószerűségi szintet, néhány területen elengedhetetlen a valóságnak megfelelő eredmény. A többségében valós idejű képalkotás során használt raszterizációra támaszkodó algoritmusok segítségével a globális illumináció nem valósítható meg közvetlenül, sőt, például már az árnyékok megállapításához szükséges lépések sem teljesen maguktól értetődőek. A sugárkövetés bár nagyobb számítási igénnyel bír, nyilvánvaló módon teszi lehetővé többek között az árnyékok és az ideális tükröződések kiszámítását is. A globális illumináció megvalósításához azonban a fényforrások és a kamera közötti fényutak minden lehetséges változatát vizsgálni kell, nem csak azok egy részét. Erre a célra véletlen bolyongási módszereket szoktak alkalmazni.

A grafikus kártyák kezdetben csak néhány rögzített feladat gyorsítására voltak képesek, majd idővel a képalkotás folyamatának egyre több és több része vált programozhatóvá. Mivel a grafikus processzor (GPU) rengeteg számítóegységgel rendelkezik, az elmúlt néhány év során felmerült az igény a GPU általános célú használatára is, amelyek túlmutatnak a raszterizációra épülő inkrementális képszintézis feladatain. Ehhez több programozási környezet is született, melyek leegyszerűsítik a GPU általános célú használatát. A dolgozat célja bemutatni egy általam készített prototípus alkalmazást, amely sugárkövetés segítségével, a grafikus processzor nyújtotta lehetőségek felhasználásával végzi a globális illumináció számítását.

2. fejezet

Képszintézis

2.1. Inkrementális képszintézis

A számítógépes grafikában az egyik gyakran használt módszer család az *inkrementális képszintézis*, melyet jellemzően valós idejű megjelenítésre alkalmaznak. Az inkrementális módszer esetében csak egyszerű grafikai primitíveket „tudunk rajzolni” a megjelenítendő képre. A megjelenítendő modelleknek ehhez előbb számos átalakításon kell átesnie. Először a modelleket a primitívekkel kell közelíteni, majd ezután olyan geometriai transzformációkat hajtunk végre a primitíveken amelyek lehetővé teszik a helyes megjelenítést. Az átalakítások után vágással csökkentjük a feldolgozandó elemek számát, mivel fölösleges azok miatt számításokat végezni, amik nem is lennének hatással egyetlen pixelre sem. Ezután – a képernyő transzformációt követően – a primitíveket (háromszögeket) raszterizáljuk. Azt is meg kell oldani, hogy a kirajzolt elemek helyesen takarják egymást. Erre több megoldás is létezik, de a manapság elterjedt módszer – a hátsólap eldobást kiegészítve – a mélységpuffer (z-puffer) használata, amely az egyes pixelekhez tartozó mélységet tárolja, így csak akkor kell az adott pixelt kirajzolni, ha a primitív pixelre eső része közelebb van a mélységpufferben tárolt jelenlegi értékéhez képest. A takarási feladat megoldása mellett a primitíveket árnyalni is kell, hogy minél valószerűbb legyen a végeredmény. Sajnos az inkrementális képszintézis hátránya, hogy az összetettebb jelenségek, mint például az árnyékok, tükröződések vagy a fénytörések megvalósítása nehézkesen, általában csak közelítésekkel valósíthatóak meg hatékonyan. Ez annak a következménye, hogy a primitíveket kezeljük egy egységként a számítások alatt, nem a pixeleket külön és egyben ez az oka annak, hogy a módszer alkalmas a valós idejű felhasználásra.

2.2. Sugárkövetés

A sugárkövetés az inkrementális képszintézissel szemben az egyes pixeleket külön, egymástól függetlenül kezeli, ami miatt sajnos több számítást igényel. Az eljárás alap gondolata a következő: adott a három dimenziós térben egy nézőpont és egy „ablak”, melyen keresztül a megjelenítendő alakzatokat nézzük. Az ablakot pixelekre osztjuk fel és a nézőpontból kiindulva az egyes pixelek irányába egy-egy sugarat irányítunk. A sugár és az alakzatok közül a nézőponthoz legközelebbi metszéspontnak a meghatározása után újabb sugarakat indítunk a fényforrás vagy a fényforrások felé. Ha a sugár a fényforrás előtt elmetesz valamit, akkor tudjuk, hogy a metszéspont árnyékban van. Ha a metszéspont-hoz tartozó alakzat tükröződő, a tükröirányba is elindítunk egy sugarat a tükröződés meghatározására és így tovább amíg egy maximális tükröződési mélységet el nem érünk.

A sugárkövetés nagy előnye, hogy bármilyen alakzatra el lehet végezni a metszéspont-vizsgálatot, továbbá rendkívül egyszerű megvalósítani azokat a jelenségeket amelyeket inkrementális módszerrel körülményes lenne. Továbbá a lokális megvilágítási modellek mellett a globális illumináció kiszámítását is könnyen lehetővé teszi.

2.2.1. Sugár és háromszög metszéspontjának számítása

A sugár az

$$\vec{r}(t) = \vec{o} + t\vec{d}, \quad (t > 0)$$

egyenlettel írható le.

Egy háromszöget három pont határoz meg: $\vec{a}$, $\vec{b}$, $\vec{c}$. Ha ez a három pont nem kollineáris akkor egyértelműen meghatároznak egy síkot. Ennek a síknak egy lehetséges – baricentrikus koordinátával történő – leírása:

$$\vec{p}(\alpha, \beta, \gamma) = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c} \quad (2.1)$$

ahol $\alpha + \beta + \gamma = 1$. Egy $\vec{p}$ pont pontosan akkor van a háromszögön belül, ha

$$0 < \alpha < 1$$

$$0 < \beta < 1$$

$$0 < \gamma < 1$$

egyenlőtlenségek teljesülnek. A (2.1) egyenletbe az $\alpha = 1 - \beta - \gamma$ való helyettesítésével:

$$\vec{p}(\beta, \gamma) = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

Most a $\vec{p}$ pont pontosan akkor van a háromszögön belül, ha

$$\begin{aligned}\beta + \gamma &< 1 \\ 0 &< \beta \\ 0 &< \gamma\end{aligned}\tag{2.2}$$

teljesülnek.

A $\vec{p}(t) = \vec{o} + t\vec{d}$ sugár (ahol $\vec{o}$ a sugár kezdőpontja, $\vec{d}$ a sugár iránya) ott metszi a síkot, ahol

$$\vec{o} + t\vec{d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})\tag{2.3}$$

teljesül. A sugár pontosan akkor metszi a háromszöget, ha a (2.2) egyenlőtlenségek teljesülnek.

t , β és γ megoldásához a (2.3) vektor-egyenletből felírható egy

$$\begin{cases} o_x + td_x = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ o_y + td_y = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ o_z + td_z = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{cases}$$

lineáris egyenletrendszer. Átrendezve, mátrix alakban:

$$\underbrace{\begin{bmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - o_x \\ a_y - o_y \\ a_z - o_z \end{bmatrix}$$

A három ismeretlen legegyszerűbben a Cramer-szabály alkalmazásával határozható meg.

$$\begin{aligned}\beta &= \frac{\begin{vmatrix} o_x - b_x & a_x - c_x & d_x \\ o_y - b_y & a_y - c_y & d_y \\ o_z - b_z & a_z - c_z & d_z \end{vmatrix}}{|\mathbf{A}|} \\ \gamma &= \frac{\begin{vmatrix} a_x - b_x & o_x - c_x & d_x \\ a_y - b_y & o_y - c_y & d_y \\ a_z - b_z & o_z - c_z & d_z \end{vmatrix}}{|\mathbf{A}|} \\ t &= \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & o_x - c_x \\ a_y - b_y & a_y - c_y & o_y - c_y \\ a_z - b_z & a_z - c_z & o_z - c_z \end{vmatrix}}{|\mathbf{A}|}.\end{aligned}$$

A előző determinánsoknak több közös tényezője is van. Az egyenletrendszert segédváltozókkal felírva

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix},$$

a Cramer-szabály alapján

$$\begin{aligned} u = \beta &= \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M}, \\ v = \gamma &= \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M}, \\ t &= -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M}, \end{aligned}$$

ahol $M = a(ei - hf) + b(gf - di) + c(dh - eg)$. Az $(ei - hf)$, $(gf - di)$, stb. tényezők előre kiszámíthatóak. [7]

Az árnyaláshoz ismerni kell a háromszög normálvektorát. Megegyezés szerint a háromszög pontjait az óramutató járásával ellentétes irányban szokás elnevezni, tárolni. A háromszög (nem egység-hosszúságú) normálvektora

$$\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a}).$$

2.2.2. Térfelosztási módszerek

Ha egy sugár legközelebbi metszéspontját szeretnénk meghatározni egy háromszögekből álló modellel akkor a legegyszerűbb megoldás az, hogy az modellt alkotó összes háromszögre elvégezzük a metszéspontszámítást. Sajnos ez a sugárkövetés egyik legidőigényesebb része, ezért célszerű lenne a sugár-háromszög metszéspont számítások számát csökkenteni.

Jelentős gyorsítást tudnánk elérni, ha a háromszögek egy részét azonnal ki tudnánk zárni a vizsgálandók köréből. Például azokat a háromszögeket, amik a sugár kezdőpontja mögött helyezkednek el, vagy azokat amelyek nem esnek a sugár útjába.

Ezekhez a döntésekhez azonban ismerni kell a modell felépítését, szerkezetét, ezért egy előkészítő fázisban fel kell építeni egy gyorsító adatszerkezetet, amelyek a döntések meghozásához szükséges információkat tárolják.

Szabályos rácsfelosztás

Az egyik módszer, hogy a teret egyenlő nagyságú cellákra osztjuk fel. Az előfeldolgozás során azt kell meghatározni, hogy egy adott cella milyen objektumokat tartalmaz (akár részben akár teljes egészében).

A sugárkövetés során a sugár kezdőpontját tartalmazó cellától kiindulva haladunk a

sugár mentén. Csak azokat az objektumokat kell vizsgálni amik az adott cellához tartoznak.

A módszer előnye, hogy a meglátogatandó cellák egyszerűen előállíthatók, továbbá ha egy cellában megtaláltuk a legközelebbi metszéspontot nem kell a következő cella objektumait vizsgálni sem. Hátránya azonban, hogy gyakran sok felesleges cellát használ, hiszen az objektumok eloszlása a térben nem mindig egyenletes.

Oktális-fa (octree)

Az oktális-fa esetében az objektumokat először egy koordinátatengelyekkel párhuzamos oldalú dobozba helyezzük. Ha egy adott mennyiségnél több objektumot tartalmaz, a dobozt nyolc egyenlő méretű részre osztjuk fel a doboz felezősíkjai mentén. Az előállt részek közül ismét felosztjuk azokat, amelyekben „túl sok” objektum van, és ezt a lépést addig ismételjük amíg a cellákban megfelelő számú objektum található vagy a fa mélysége elért nem ért egy előre megszabott maximális szintet.

Ez a módszer jobb a rácsfelosztásnál abból a szempontból, hogy nincsenek fölösleges cellák, aminek köszönhetően a memóriaigény is csökken, viszont a cellák vizsgálata során a következő ellenőrizendő elem megállapítása összetettebb feladat. További probléma, hogy a felezősík menti felosztások miatt sok üres tér maradhat a cellákban ami fölösleges számításokhoz vezethet.

kD-fa

A bináris térparticionáló fa (Binary Space Partitioning tree, röviden BSP tree) a teret nem cellákra hanem félterekre osztja úgy, hogy a felezősíkok irányára nincs semmilyen megkötés. A kD-fa ezzel szemben egy olyan speciális bináris térparticionáló fa amelyben a térfelező síkok merőlegesek a koordinátatengelyekre.

Hierarchikus befoglaló keret fa (BVH-fa)

A hierarchikus befoglaló keret módszerrel az objektumok köré egy szorosan illeszkedő keretet teszünk. Az objektumok metszéspontjainak kiszámítása előtt először a befoglaló keretet vizsgáljuk, mert ha a sugár nem metszi a keretet akkor az objektumokat sem metszheti. Ezt az elvet követve a kereteket hierarchiába szervezhetjük. Célszerű olyan keretet használni amiről gyorsan eldönthetjük, hogy metszi-e a sugár vagy sem. Erre a célra általában koordinátatengelyekkel párhuzamos oldalú téglatestet (Axis Aligned Bounding Box, röviden AABB) szoktak alkalmazni.

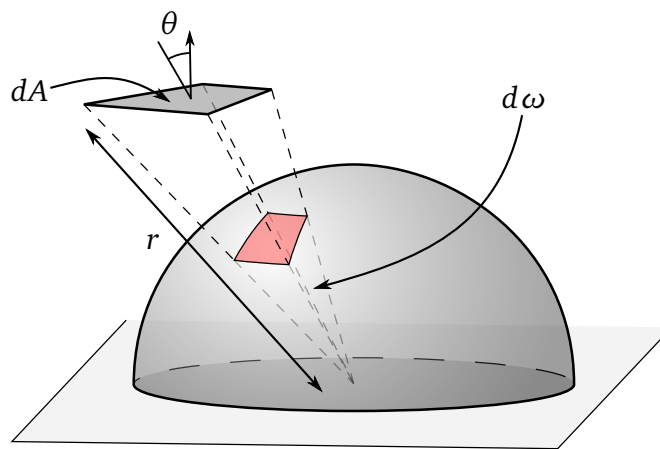
2.3. Globális illumináció

2.3.1. Fotometria

Egy dA infinitezimális felületelem egy $\vec{p}$ pontból

$$d\omega = \frac{dA \cdot \cos \theta}{r^2} \quad (2.4)$$

térszög alatt látszik, ahol r a $\vec{p}$ pont és a dA felületelem távolsága, θ a dA felületelem normálisa és a $\vec{p}$ iránya közötti szög. (2.1. ábra)



2.1. ábra. A $d\omega$ differenciális térszögben látható dA felületelem nagysága

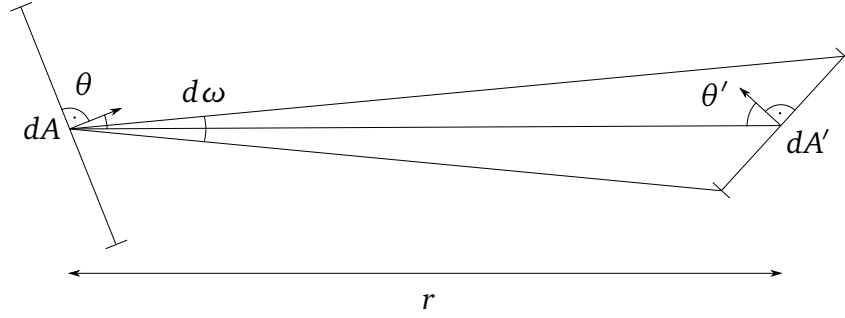
1. definíció (Fluxus). Egy adott felületen egységnyi idő alatt átlépő energiát teljesítménynek, vagy fluxusnak nevezzük, mértékegysége a watt [W].

2. definíció (Radiancia). Egy dA felületelemet $d\omega$ térszögben elhagyó $d\Phi$ infinitezimális fluxusnak a kilépési irányból látható differenciális területtel és a térszöggel ($dA \cdot \cos \theta$) vett hányadosát sugársűrűségnek, radianciának, vagy intenzitásnak nevezzük és L -lel jelöljük:

$$L = \frac{d\Phi}{dA \cdot d\omega \cdot \cos \theta}. \quad (2.5)$$

Ha egy dA és egy dA' felületelemek látják egymást és dA intenzitása dA' irányába L , akkor az átadott fluxus:

$$d\Phi = L \cdot dA \cdot d\omega \cdot \cos \theta.$$



A térszög a (2.4) egyenlet felhasználásával kifejezhető a látható dA' felületelem területével:

3. definíció (Fotometria alaptörvénye).

$$d\Phi = L \cdot \frac{dA \cdot \cos \theta \cdot dA' \cdot \cos \theta'}{r^2}.$$

Az átadott teljesítmény a következő alakban is felírható:

$$d\Phi = L \cdot dA' \cdot \frac{dA \cdot \cos \theta}{r^2} \cdot \cos \theta' = L \cdot dA' \cdot d\omega' \cdot \cos \theta', \quad (2.6)$$

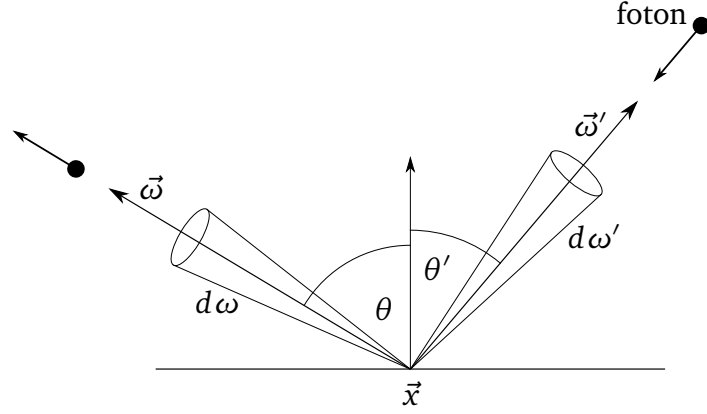
vagyis ugyanaz az egyenlet vonatkozik a sugárzó felületelemre mint a felfogóra. Ez az oka annak, hogy a számítások során a szemből a fényforrás felé is visszakövetjük a fénysugarakat. [8, 9]

2.3.2. Fény-felület kölcsönhatás

A megvilágított felület a beérkező fényt teljesítmény egy részét különböző irányokba visszaveri, míg másik részét elnyeli. Egy $\vec{\omega}'$ irányból a felület $\vec{x}$ pontjába érkező foton a

$$w(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot d\omega = \text{Pr}(\text{a foton az } \vec{\omega} \text{ körüli } d\omega \text{ térszögben halad} \mid \vec{\omega}' \text{ irányból jön})$$

visszaverődési-sűrűségfüggvény szerinti valószínűséggel halad tovább az $\vec{\omega}$ irányba. (2.2. ábra)



2.2. ábra. Az $\vec{\omega}'$ irányból érkező foton visszaverődése az $\vec{\omega}$ irányba

4. definíció (BRDF). A foton haladását leíró valószínűség-sűrűségfüggvény és a kimeneti szög koszinuszának hányadosát kétirányú visszaverődés eloszlási függvénynek (Bi-directional Reflection Distribution Function) nevezzük, mértékegysége 1 per szteradián, $[sr^{-1}]$:

$$f_r(\vec{\omega}', \vec{x}, \vec{\omega}) = \frac{w(\vec{\omega}', \vec{x}, \vec{\omega})}{\cos \theta}$$

Az $\vec{\omega}'$ a fény bejövő irányát, az $\vec{x}$ a felületi pontot, míg az $\vec{\omega}$ a kilépő irányt azonosítja.

5. definíció (Árnyalási egyenlet). A fényátadás alapvető integrálegyenletét árnyalási egyenletnek (rendering equation) nevezzük:

$$L(\vec{x}, \vec{\omega}) = L^e(\vec{x}, \vec{\omega}) + \int_{\Omega} L(h(\vec{x}, -\vec{\omega}'), \vec{\omega}') \cdot f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos \theta' \cdot d\omega',$$

ahol L^e a fénykibocsátás, $h(\vec{x}, -\vec{\omega}')$ pedig a láthatósági függvény, ami azt adja meg hogy egy pontból egy adott irányba milyen másik felület látszik.

Az egyenlet ismeretlenje az L sugársűrűség függvény.

[8], [9]

2.3.3. Monte-Carlo integrálás

Az árnyalási egyenlet dimenzionalitása miatt a megoldást csak Monte-Carlo integrálás segítségével érdemes keresni.

Legyen V egy D -dimenziójú $\mathbf{z} = [z_1, \dots, z_D]$ pontokat tartalmazó tartomány, valamint $f(\mathbf{z})$ a V tartomány felett integrálandó függvény! Ha az $f(\mathbf{z})$ integrandust egy $p(\mathbf{z})$ valószínűség-sűrűségfüggvénnyel beszorozzuk majd elosztjuk, akkor az $f(\mathbf{z})/p(\mathbf{z})$

függvényt egy valószínűség-sűrűségfüggvénnyel súlyozva integráljuk. Abban az esetben ha a $\mathbf{z}$ változó sűrűségfüggvénye $p(\mathbf{z})$ akkor az előbbi integrál épp az $f(\mathbf{z})/p(\mathbf{z})$ várható értéke, amit a nagy számok törvénye miatt véletlen minták átlagával becsülhetünk:

$$\int_V f(\mathbf{z}) d\mathbf{z} = \int_V \frac{f(\mathbf{z})}{p(\mathbf{z})} \cdot p(\mathbf{z}) d\mathbf{z} = E \left[\frac{f(\mathbf{z})}{p(\mathbf{z})} \right] \approx \frac{1}{M} \cdot \sum_{i=1}^M \frac{f(\mathbf{z}_i)}{p(\mathbf{z}_i)}.$$

A becslés szórását úgy csökkenthetjük, hogy a $p(\mathbf{z})$ sűrűségfüggvényt az integrandussal arányosan próbáljuk megválasztani. Ezt a módszert *fontosság szerinti mintavételezésnek* (*importance sampling*) nevezik.

[8], [9]

2.3.4. BRDF modellek

A valóságban képek előállításához a BRDF-modelleknek *fizikailag plauzabilisnek* kell lenniük, ami azt jelenti, hogy nem sérthetik az energiamegmaradás törvényét valamint szimmetrikusnak kell lenniük.

A Helmholtz-féle szimmetria, vagy *reciprocitás* szerint a fénysugár megfordítható, azaz a BRDF-ben a bejövő és kimenő irányok felcserélhetőek:

$$f_r(\vec{\omega}, \vec{x}, \vec{\omega}') = f_r(\vec{\omega}', \vec{x}, \vec{\omega})$$

Egy önállóan nem sugárzó felületelem nem adhat ki több fotont (nagyobb fluxust), mint amit maga kapott, különben sértené az energiamegmaradás elvét, tehát a tetszőleges irányú visszaverődés teljes valószínűsége nem lehet egynél nagyobb.

6. definíció (Albedó). A tetszőleges irányú visszaverődés valószínűségét albedónak nevezzük:

$$a(\vec{x}, \vec{\omega}') = \int_{\Omega_H} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos \theta d\omega \leq 1.$$

[8], [9]

Diffúz visszaverődés

A diffúz anyagokról visszavert sugársűrűség független a nézeti iránytól. A Helmholtz-féle reciprocitás értelmében a BRDF ekkor a bejövő iránytól sem függhet, azaz a BRDF irányfüggetlen konstans:

$$f_r(\vec{l}, \vec{v}) = k_d,$$

ahol $\vec{l}$ a fényforrás irányába mutató egységvektor, $\vec{v}$ a nézőirányba mutató egységvektor.

Az energiamegmaradás miatt az albedó diffúz visszaverődés esetén sem lehet 1-nél nagyobb, így a k_d diffúz visszaverődési együtthatóra

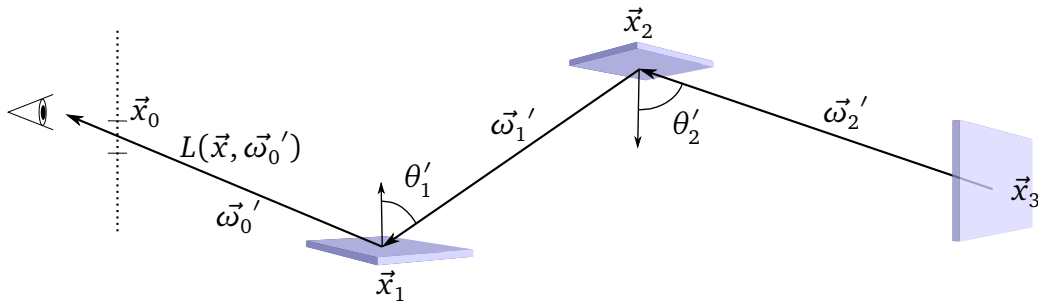
$$a(1) = \int_{\Omega_H} k_d \cdot \cos \theta \, d\omega = k_d \cdot \pi \leq 1.$$

szerint $k_d \leq \frac{1}{\pi}$ korlát állítható fel. [9]

2.3.5. Inverz fényűtkövetés (*path tracing*)

Az inverz fényűtkövetés során az árnyalási egyenletnek úgy keressük a megoldását, hogy a szemből kiindulva a kép pixelein keresztül sugarakat indítunk el a megjelenítendő modell felé, hasonlóan a sugárkövetéshez. A metszett felületekről ezután annak BRDF-je szerint, véletlenszerű irányban tovább folytatjuk a sugárkövetést miközben az felületek fénykibocsátásait az árnyalási egyenletnek megfelelően összegezzük:

$$\begin{aligned} L(\vec{x}_1, \vec{\omega}_0) &\approx L^e(\vec{x}_1, \vec{\omega}_0) + \\ &+ \frac{f_r(\vec{\omega}_1', \vec{x}_1, \vec{\omega}_0) \cdot \cos \theta_1'}{p_{\vec{x}_1}(\vec{\omega}_1')} \cdot L^e(\vec{x}_2, \vec{\omega}_1') + \\ &+ \frac{f_r(\vec{\omega}_1', \vec{x}_1, \vec{\omega}_0) \cdot \cos \theta_1'}{p_{\vec{x}_1}(\vec{\omega}_1')} \cdot \frac{f_r(\vec{\omega}_2', \vec{x}_2, \vec{\omega}_1) \cdot \cos \theta_2'}{p_{\vec{x}_2}(\vec{\omega}_2')} \cdot L^e(\vec{x}_3, \vec{\omega}_2') + \dots \end{aligned}$$



A pontos eredmény meghatározásához a fenti a végtelen sort kellene kiszámolni. Ha az összegzést egy bizonyos tag után befejeznénk az torzítaná a végeredményt. Ezért egy *Orosz rulett* nevű módszert használunk amely véletlenszerűen dönt arról, hogy mikor fejezzük be az összegzést. Legyen s_i a valószínűsége annak, hogy folytatjuk a bolyongást és $1-s_i$ annak hogy nem. Ha folytatjuk a bolyongást, a számított L bejövő sugársűrűséget meg kell szorozni $1/s_i$ -vel, amivel a véletlenszerűen elhanyagolt energiát pótoljuk. [8], [9]

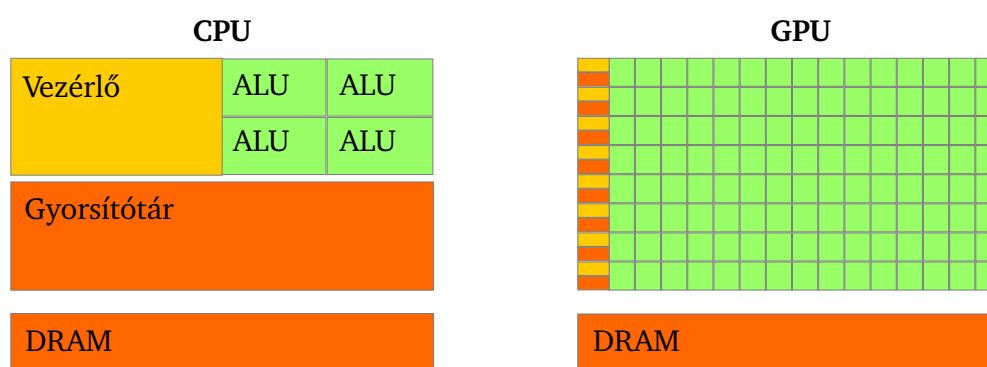
3. fejezet

GPGPU programozás

3.1. A grafikus processzor általános célú programozása

Az elmúlt évtizedben a programozható grafikus processzorok (GPU-k) nagy párhuzamossággal bíró, többszálú, sokmagos processzorokká nőttek ki magukat, aminek köszönhetően igen magas számítási teljesítményre tettek szert.

A számítógép központi processzora összetett vezérlőegységgel, néhány aritmetikai-logikai egységgel (ALU) és viszonylag nagy gyorsítótárral rendelkezik. A grafikai problémák jellegéből adódóan a grafikus processzor párhuzamos számítások elvégzéséhez szükséges felépítéssel rendelkezik, vagyis jóval több ALU van beépítve. (3.1. ábra) Az adat-párhuzamos feladatok miatt nincs szükség összetett vezérlőegységekre, mivel minden adatelemre ugyanazt a műveletet kell elvégezni. Továbbá nincs szükség nagy gyorsítótárakra sem, mert egyrészt a műveletek sok adatelemmel dolgoznak, másrészt a memória-elérésből fakadó késleltetést – azok mennyisége miatt – a számítások elrejtik. [5]



3.1. ábra. A CPU és a GPU vázlatos felépítése [5]

Az egyik lehetőség arra hogy kiaknázzuk a GPU számítási kapacitását, hogy példá-

ul a fragmentum árnyaló segítségével olyan textúrát dolgozunk fel bemenetként, amely valójában nem egy hagyományos kép, hanem a feldolgozandó adatainkat tartalmazza. [2] Ez a megoldás sajnos elég körülményes és összetettebb problémák megoldását különösen megnehezíthetik. Szerencsére az utóbbi néhány évben több *GPGPU* (General Purpose GPU, vagyis általános célú GPU) programozási eszköz is megjelent. A két legelterjedtebb ezek közül az *NVidia CUDA* valamint a platformfüggetlen *Open Computing Language* (OpenCL).

Bár az OpenCL platformfüggetlensége rendkívül előnyös, választásom azért esett a CUDA-ra, mert a Monte-Carlo módszerre támaszkodó inverz-fényűtkövetés megvalósításához szükségem volt egy jó minőségű véletlenszám generátorra. Az OpenCL-hez sajnos még kevés függvénykönyvtár készült, míg a CUDA több hasznos könyvtárral, köztük a cuRAND nevű álvéletlenszám előállító csomaggal együtt érhető el.

3.2. Programozás CUDA környezetben

A CUDA terminológiában a CPU-t és a hozzá tartozó memóriát *host*-nak, a GPU-t és annak memóriáját *device*-nak nevezik. A programok szerkezetileg szekvenciális (*host-on* futó) és párhuzamos (*device-on* végrehajtott) részekből állnak. [5]

3.2.1. CUDA *kernel*

A CUDA *kernel* olyan speciális függvény, amelyet párhuzamosan N darab különböző CUDA szál hajt végre.

A kernel függvényeket a hagyományosaktól a `__global__` deklarációs specifikátor különbözteti meg. A kernel függvények visszatérési típusának továbbá kötelezően `void`-nak kell lenniük.

```
__global__ void kernelFunction(...) { ... }
```

Meghívásuk a C és C++ nyelvben nem létező `<<<...>>>` szintakszisú *végrehajtási konfiguráció* megadásával – aszinkron módon – történik.

Minden kernelt végrehajtó szál egy egyedi *szálazonosítót* kap, ami a kernelen belül a `threadIdx` beépített változón keresztül érhető el.

Lehetőség van csak kernelekből hívható függvények definiálására is a `__device__` deklarációs specifikátor használatával. Rekurzív függvények megadása azonban nem lehetséges, ezt csak az újabb videokártya modellek támogatják. [5]

3.2.2. Szál hierarchia

A `threadIdx` egy három komponensből álló vektor, így a szálak egy, kettő vagy három dimenzióba szervezve azonosíthatóak, és ezt a szervezést *blokk*nak nevezzük. A blokkok

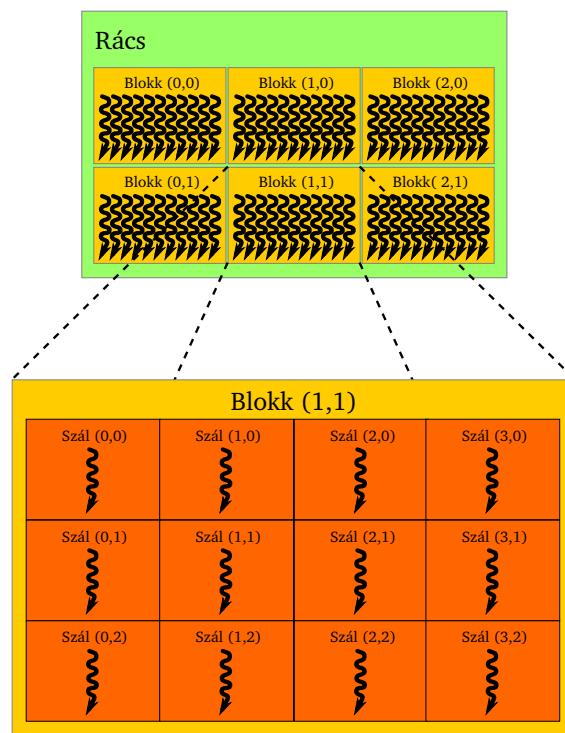
mérete nem lehet tetszőlegesen nagy, mivel egy adott blokk teljes egésze a GPU egy adott magjára van kiosztva. A mai videokártyák esetében egy blokk legfeljebb 1024, régebbiek esetében legfeljebb 512 szálból állhat.

Egy kernel végrehajtása egynél több blokkot is felhasználhat. A szálak szervezéséhez hasonló módon a blokkok egy, kettő vagy háromdimenziós *rácsot* (*grid*) alkotnak. (3.2.2. ábra) A blokk méretét a kernelen belül a `blockDim`, a blokk azonosítóját a `blockIdx` beépített változókon keresztül érhetjük el.

A már korábban említett végrehajtási konfigurációban a blokkok és szálak nagyságát kell megadni:

```
kernelFunction<<<rács méret, blokk méret>>>(...);
```

A *rács méret* és *blokk méret* paraméterek típusa vagy `int` (egy dimenzió esetén), vagy `dim3` (kettő vagy három dimenzió esetén). [5]



3.2. ábra. Szál-hierarchia [5]

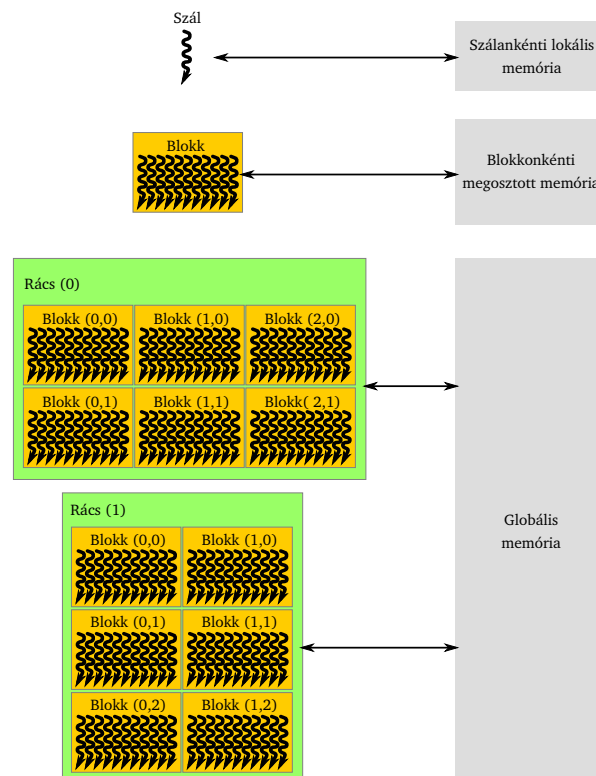
3.2.3. Memóriakezelés

A szálak többféle memória területet érhetnek el. Minden szálnak van egy saját *lokális memóriarésze*, amihez más szál nem tud hozzáférni.

Egy adott blokkon belül az összes szál hozzá tud férni a blokkhoz tartozó *megosztott memóriához* (*shared memory*), így a blokkon belüli szálak kommunikálhatnak egymással.

A blokkon belül a `__syncthreads()` függvény hívásával hozható létre szinkronizációs pont (*barrier*) a végrehajtásban, azaz egy szál csak akkor lépheti át a barriert, ha azt az összes többi blokkon belüli szál elérte.

Minden szál számára elérhető a *globális memória*, függetlenül attól, hogy a szál melyik blokkban fut. Ezen túl minden szál elérhet két csak-olvasható memóriatípust: a textúra- és a konstans-memóriaterületet. Utóbbi három memóriatípus perzisztens az egyes kernel futtatások között, vagyis a kernel indítása előtt nem kell azokat újra feltölteni a videokártya memóriájára. [5]



3.3. ábra. Memória-hierarchia [5]

A globális memóriát a következő CPU-ról hívható függvények segítségével kezelhetjük: A `cudaMalloc` a videokártya memóriáján egy megadott méretet foglal le és a lefoglalt terület címét az első paraméterében megadott változóba tárolja. Adatmásolást – melynek iránya lehet CPU-ról GPU-ra, fordítva, illetve két GPU memóriaterület között – a `cudaMemcpy` segítségével, lefoglalt memóriaterület felszabadítását a `cudaFree` hívásával végezhetünk.

A lefoglalt memóriaterületeket textúraként is kezelhetjük. Ez azért előnyös, mert a globális memória használata a régebbi modelleken nem gyorsítótárazott. Textúraként azonban a kijelölt memóriaterületeket gyorsítótárazva érhetjük el a kernelen belül, ami sokat javíthat a számítások idején. Ilyenkor azonban ezekre a memóriaterületekre a kern-

el futása alatt nem írhatunk, továbbá az elemek által használt típusok is korlátozottak. Ezen kívül a forráskódban globális változóként kell deklarálni egy textúrahivatkozást, amire a `tex1Dfetch` függvénnyel való olvasás miatt van szükség.

Fontos még kiemelni a GPU-n hívható `atomicAdd` függvényt, amely kölcsönös kizárás mellett növeli a megadott változót. [5]

4. fejezet

Fejlesztői dokumentáció

A dolgozatom során olyan prototípus alkalmazást szerettem volna készíteni, amely egy háromszög modell fájlt tölt be és ezt GPU-n végrehajtott inverz-fényűtkövetés segítségével képezi le, a számítás részeredményének folyamatos megjelenítése mellett. A programot Nvidia GeForce 9600 GT típusú videokártyámra fejlesztettem, amely a mai kártyákhoz képest jóval szerényebb számítókapacitással rendelkezik, ezért törekednem kellett a hatékony megvalósításra. A modellek és anyagjellemzők beolvasásához a széles körben elterjed Wavefront OBJ és MTL formátumokra támaszkodtam, mivel egyszerű, olvashatósága elősegíti a hibakeresést, valamint felmerülő igények esetén könnyen kiegészíthető egyedi jellemzőkkel. Annak érdekében, hogy nagy mennyiségű háromszögből felépülő modelleket is le tudja a program képezni, a képalkotó számítások előtt hierarchikus befoglalókeret fát építünk fel.

4.1. Háromszögmodellek tárolása

A program a modellek leírására szolgáló Wavefront OBJ formátum következő részeit támogatja: csúcspontok, normálvektorok és háromszögek megadása, valamint anyagjellemzők beállítása.

A csúcspontok megadását a sorok elején „v” betűvel, a normálvektorokat „vn”-nel kell jelezni, majd ezután szóközzel elválasztva kell megadni az egyes koordinátákat. A számok egész- és tört részét „tizedespontnak” kell elválasztania.

Csak háromszögek megadása lehetséges, melyet „f” betűvel kell jelezni a sor elején, ezután szóközzel elválasztva meg kell adni a csúcspontok azonosítószámait vagy a csúcspontok és az ahhoz tartozó normálvektorok azonosítóit „//”-rel elválasztva.

v x_koordináta y_koordináta z_koordináta

vn x_koordináta y_koordináta z_koordináta

f csúcspont1 csúcspont2 csúcspont3

f csúcspont1//normálvektor1 csúcspont2//normálvektor2 csúcspont3//normálvektor3

Anyagjellemzők megadására is lehetőség van, ehhez az OBJ fájlban meg kell adni az anyagtulajdonságokat leíró fájl nevét a következőképpen:

```
mtllib anyagjellemző_fájl.mtl
```

Adott háromszögek anyagjellemzőinek kiválasztásához a háromszögeket leíró sorok előtt a „usemtl” szót majd az anyagjellemző azonosítónevét kell megadni.

Az anyagjellemzőket leíró fájl a Wavefront MTL formátumra épül. Az egyes anyagokat az alábbi módon lehet megadni:

```
newmtl azonosító_név
Kd diffúz_vörös diffúz_zöld diffúz_kék
Xe fénykibocsátás_vörös fénykibocsátás_zöld fénykibocsátás_kék
%
```

A „Kd” sor a diffúz albedó megadására szolgál, ahol az egyes komponensek [0,1] közötti számok. (Természetesen más is megadható, de ilyenkor nem fog teljesülni az energiamegmaradás.)

Az „Xe”-vel jelölt sor – amely nem része a Wavefront MTL szabványnak – a fénykibocsátás mértékét határozza meg. Ha nincs megadva, akkor az anyag nem bocsát ki fényt.

A betöltött adatokat az alábbi struktúrában tárolja a program:

SceneData
numVertices : int
numTriangles : int
numMaterial : int
numNormals : int
vertices : float4*
triangles : uint4*
materials : Material*
normals : float4*
triangleNormals : uint4*
bvhTree : BVHTree

Az numVertices, numTriangles, numMaterial numNormals mezők az OBJ és MTL fájlból betöltött csúcsok, háromszögek, anyagjellemzők valamint a megadott normálvektorok számát tárolják. A vertices egy float4 elemekből álló tömbre mutat, ebben tároljuk a háromszögek által használt csúcspontokat. A triangles tömbben található elemek az egyes háromszögek csomópontjait és a háromszög anyagtulajdonságát határozzák meg. Az uint4 típusú vektor x, y és z komponense a vertices tömbre hivatkozó, míg a w a materials tömbre hivatkozó index. A materials tömbben tárolt elemek az MTL fájlból betöltött anyagjellemzőket tárolják. A normals és

`triangleNormals` mezők a `vertices` és `triangles` tömbökhöz hasonló módon vannak feltöltve, annyi különbséggel hogy a `triangleNormals` `w` elemét nem hasznosítjuk. A `normals` tárolja a különböző normálvektorokat, amikre a `triangleNormals` tömbben lévő indexek hivatkoznak.

Azért használok négy elemből álló vektor típusokat (`float4`, `uint4`), mert ezeket a tömböket textúraként fogom kezelni annak érdekében, hogy a gyorsítótárazást ki tudjam használni, viszont a textúrák elemei nem állhatnak 3 komponenses vektorokból.

A `bvhTree` mező a BVH-fát tárolja, melyet hamarosan részletesebben bemutatok.

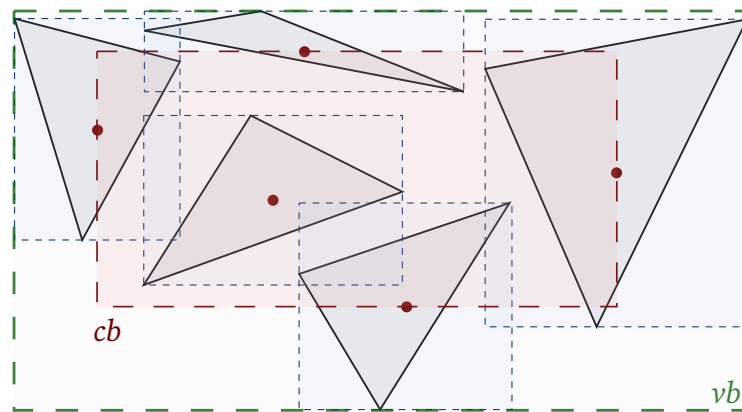
4.2. A gyorsítófa felépítése és tárolása

A BVH-fa felépítéséhez Ingo Wald felületeheurisztikára épülő módszerét valósítottam meg, mely a lehetséges felosztások költségeinek vizsgálatát edényekbe csoportosítás segítségével gyorsítja. [10]

A fa konstrukciója az alábbi lépésekből áll:

1. Először előállítjuk minden i háromszögnek előállítjuk a befoglalódobozát (tb_i), majd az előállított befoglalódobozok középpontjait vagy más néven centroidjait (c_i). A háromszögek adataira a fa előállításánál ezek után már nem is lesz szükségünk.

Ezt követően kiszámítjuk az összes háromszög befoglalójának (vb) és az összes befoglaló középpontjának a befoglalóit (cb). (4.1. ábra)



4.1. ábra. A háromszögek befoglalódobozai, azok középpontjai, a középpontok és a háromszög-dobozok befoglalói

2. A háromszögeket edényekhez (*bins*) rendeljük hozzá. A c_i centroidtal rendelkező

háromszög a

$$binID_i = \frac{K(1 - \epsilon)(c_{i,k} - cb_{min,k})}{cb_{max,k} - cb_{min,k}}$$

azonosítójú edénybe tesszük, ahol a cb a centroidok befoglalója, k az edényekre osztás tengelye, K pedig az edények száma. Az $(1 - \epsilon)$ biztosítja hogy a két edény határára eső centroidú háromszögek garantáltan az egyik oldalra kerüljenek.

Észrevehetjük, hogy az előbbi képletben a c_i -t kivéve minden érték konstans, ezért ezeket előre kiszámíthatjuk. Legyen

$$k_1 = \frac{K(1 - \epsilon)}{cb_{max,k} - cb_{min,k}},$$

valamint legyen $k_0 = cb_{min,k}$. Ezek alapján az i . háromszög a $binID_i = k_1(c_{i,k} - k_0)$ edényhez tartozik.

3. Amikor a háromszögeket beletesszük az edényekbe, folyamatosan számoljuk, hogy egy adott edényben mennyi háromszög van (n_i), valamint folyamatosan frissítjük az edény befoglalóját (bb_i). Kezdetben az edények befoglalódoboza „negatív térfogatú” ($[+\infty, -\infty]$), így a háromszögek hozzáadásakor az edény befoglalójának bővítése egyszerűen egy minimum illetve egy maximum függvény használatával történhet.
4. Ezután kiszámítjuk a lehetséges felosztások költségeit. K darab $(B_1, \dots, B_K)$ edény esetén a háromszögek T halmazát $(K - 1)$ -féleképpen bonthatjuk fel két részre: $T_{L,j} = \{B_1, \dots, B_j\}$ a baloldal, míg $T_{R,j} = \{B_{j+1}, \dots, B_K\}$ a jobboldal. ($j = 1 \dots K - 1$) Ismerve az egyes edényekben található háromszögek számát (n_i) a felosztás baloldalán $N_{L,j} = \sum_{i=1}^j n_i$, jobboldalán $N_{R,j} = \sum_{i=j+1}^K n_i$ háromszög áll. A célunk hogy az $A_{L,j}N_{L,j} + A_{R,j}N_{R,j}$ költség minimális legyen, ahol $A_{L,j}$ a baloldal befoglalójának, $A_{R,j}$ a jobboldal befoglalójának a felülete.
5. A kiválasztott felosztás alapján a háromszögeket két részre válogatjuk szét, majd ezekre rekurzívan ismét elvégezzük az előbbi műveleteket. A rekurziót addig ismételjük amíg egy partícióba legfeljebb 2 háromszög kerül. Mivel a háromszögek jellemzői nem változtak, ezért elég a 2. lépéstől végrehajtani az újabb műveleteket.

Az elkészített BVH-fa elemeit a GPU-n való hatékony memóriakezelés érdekében pre-order bejárással több különböző tömbbe tárolja el a program. Ezek a tömbök a BVHTree struktúra részei:

BVHTree
numBoxes : int
min : float4*
max : float4*
childIds : int2*

A numBoxes mező a fában tárolt befoglalódobozok számát tárolja. A min, és max tömbök a dobozokat egyértelműen meghatározó minimális illetve maximális csúcspontját tárolják. A childIds mezőben található vektor x és y komponense a gyerekelemek indexeit tárolják. Ha nincs gyerekelem akkor -1-et tárol a program, továbbá ha a gyerekelem nem befoglalódoboz, hanem háromszög, akkor a tárolt értéket a háromszög azonosítójának (vagyis hogy hanyadik elem a SceneData struktúra triangles tömbjében) és a numBoxes érték összegzéssel kapjuk. Így, ha a kiolvasáskor ez a érték nem kisebb mint numBoxes, tudjuk hogy háromszöget kell vizsgálni.

Ebben a struktúrában is négyelemű vektorokat használtam, mert a struktúra tömbjeit textúráként fogom kezelni.

4.3. Inverz fényűtkövetés a GPU-n

Az első implementációban amit elkészítettem a számításokat mindössze egy kernel végezte. Ezzel több probléma is van. Az egyik, hogy a kernel regiszterigénye a több végrehajtandó művelet miatt nagyobb. Emiatt kevesebb szálát lehet elindítani a blokkokban, mivel a blokkonként felhasználható regiszterek száma korlátozott. A másik probléma a vezérlés divergencia: a szálak végrehajtása a GPU-n – általában 32 szálból álló – ún. warp-okba szervezve történik. A warp-okon belül a szálakhoz egy utasításszámláló tartozik, vagyis ha a vezérlés néhány szálon más ágra kerül mint a többi, az egyes vezérlési ágakat csak egymás után lehet végrehajtani, ami miatt csökken a párhuzamosság. Az Orosz rulett módszer használata miatt lesznek olyan szálak amelyek hamarabb fejezik be a munkát, így sok erőforrás kihasználatlanul marad.

Ezen problémák kiküszöbölésére Laine et al. javaslataihoz [4] hasonló módon több, részfeladatokat végző kernelekre bontottam fel az eredeti változatot. A megoldás hátránya, hogy az eredetileg regiszterekben és lokális memóriában tárolt adatokat a globális memóriába kell helyezni, amin keresztül a kernelek az elvégzett feladatok eredményeit átadják az azokat igénylő kerneleknek. Ezeket az adatokat olyan struktúrában tárolja a program, amelynek minden mezője egy tömb (Structure-of-Arrays röviden SoA), így növelhető a memóriaelérések hatékonysága. A struktúra a következőképpen épül fel:

SoA_PathSegments
numPathSegment : int
rngState : curandState*
origin : float4*
direction : float4*
t : float*
u : float*
v : float*
triangleId : int*
sampleContrib : float4*
throughput : float4*
numSamples : int*
numBounces : int*

A struktúrában az összes tömb azonos – numPathSegment – méretű.

Az rngState az álvéletlenszám-generátor állapotváltozója. A sugarak kezdőpontját és irányvektorát a origin és a direction tömbökben tárolom, míg a metszéspon-
számítás eredményeit a t, u, v és a triangleId tömbök tárolják. A t a metszéspon-
távolsága a kezdőponttól a sugár irányába. Az u és v baricentrikus koordináták, ame-
lyekre a normálvektorok interpolációjához miatt van szükség. A triangleId a metszett
háromszög azonosítója. Amennyiben az adott sugár nem metszett egyetlen háromszöget
sem akkor -1 az értéke.

A minták radiancia-hozzájárulását a sampleContrib tömb, míg a fényútkövetéshez
szükséges áteresztési tényezőt a throughput tömbben tároljuk.

Az Orosz rulett módszer alkalmazása miatt az egyes fényutak nem azonos iteráció-
ban érnek véget ezért mindegyikhez külön kell nyilvántartani az minták számát. Ezt a
numSamples tömb biztosítja. A numBounces a fényutak jelenlegi hosszát tárolják. Ezt
azért tartja nyilván a program, mert az Orosz rulett módszert csak egy adott fényúthossz
felett alkalmazzuk, annak érdekében hogy csökkenjen a Monte-Carlo becslés szórása.

Összesen öt CUDA kernelt tartalmaz az alkalmazás, ebből egy a számításokat meg-
előző inicializáló kernel, a többi számításokat végző képalkotás-logika kernel, a kamera
kernel, az árnyaló kernel és a metszéspon-
számító kernel. A kernelek sorokon keresztül
adják ki egymásnak az igényelt feladatokat. A sorokba a SoA_PathSegment adatszer-
kezet tömbindexei kerülnek az elvégzendő feladatoknak megfelelően. A logikai kernel
nem olvas egyetlen sorból sem, mert a SoA_PathSegment mindegyik elemét feldolgoz-
za és ezek alapján tölti fel a kamera kernel és az árnyaló kernel sorát. Ez a két kernel
a metszéspon-
számító kernel sorába írhat, míg a metszéspon-
számító kernel nem ír
egyetlen sorba sem.

A sorokat úgy valósítottam meg, hogy egy fix – MAX_SIZE – méretű memóriaterüle-
tet lefoglallok és egy számlálóval (numItems) tartom nyilván a sorban található elemek

(items) számát:

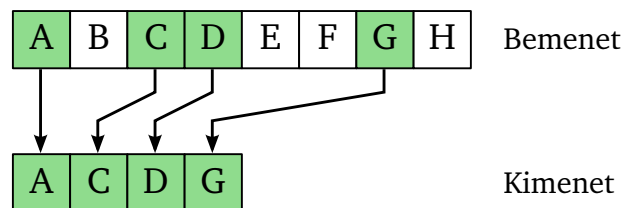
Queue
MAX_SIZE : int
items : int*
numItems : int*

Fontos megjegyezni, hogy az numItems mező csak 1 db egész számra mutat. Azért kell itt mutatót alkalmazni, hogy a kernelek frissíthessék a változó értékét.

4.3.1. Folyam préselés (stream compaction)

A bemeneti sorokkal rendelkező kernelek a sor minden elemét feldolgozzák. Minden egyes szál a sor egy-egy elemét kezeli: Az első szál az első elemet, a második a másodikat, és így tovább. Az adott feladat elvégzése után előfordulhat, hogy egy szál nem fogja a saját elemét a kimeneti sorhoz hozzáadni. Azonban ahhoz hogy a kimenet folytonos maradjon és ezáltal a következő kernel minden szála dolgozzon, minden sorba író szálnak tudnia kellene a sorban lévő elemek addigi számát, tehát azt hogy a tömb melyik elemébe kellene írni. Ezt a legegyszerűbben úgy lehetne megvalósítani, hogy a szálak az atomicAdd függvény segítségével kiolvassák, majd 1-el növelik az elemszámlálót. A kiolvasott érték alapján már ismertté válik a írandó elem helye. Ez a megoldás sajnos nem elég hatékony mivel a atomicAdd függvény gyakori hívása miatt a teljesítmény jelentősen romlik, ezért célszerű lenne a hívások számát valamilyen módon csökkenteni.

Laine et al. az atomicAdd hívásának számát warp-szintű szavazás felhasználásával csökkenti [4]. Ezt a műveletet a videokártyám nem támogatja ezért Dietgen van Antwerpen által alkalmazott módszer [1] segítségével oldottam meg a problémát: a sorba írás előtt minden blokkon belül folyam préselés (stream compaction) történik, így az atomicAdd függvény az adott blokkban csak egyszer kerül meghívásra.

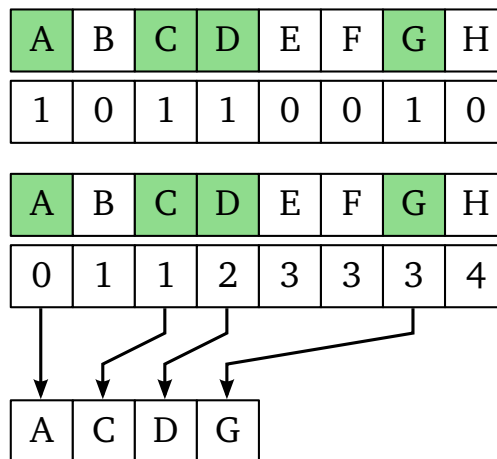


4.2. ábra. Folyam préselés [3]

A folyam préselés elvégzéséhez prefixszámítási (prefix-sum) feladatot kell megoldani. A prefixszámítás egy olyan feladat, ahol adott egy kétváltozós, I identitáselemmel rendelkező $\oplus$ asszociatív művelet, valamint egy n -elemű $[a_0, a_1, \dots, a_{n-1}]$ tömb és az $[I, a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{n-2}]$ tömböt szeretnénk meghatározni. [3]

A folyam préselés lépései:

1. Kiválasztjuk a megtartandó elemeket
2. Azokhoz az elemeket amelyeket a kimeneti folyamban meg szeretnénk tartani 1-t, a többihez 0-t rendelünk és ezeket egy tömbbe írjuk
3. A tömbre elvégezzük a prefixszámítást
4. A kapott tömbben azokon a helyeken ahol eredetileg 1-es volt most az az index áll, ahova a megtartandó elemeket kell másolni



4.3. ábra. Folyam préselés megvalósítása prefixszámítással [3]

A prefixszámítást a GPU Gems-ben [3] ismertetett Blelloch-féle párhuzamos algoritmus felhasználásával valósítottam meg. Az algoritmus két fázisból áll, melynek első fázisa párhuzamos redukció (összegzés). Fontos, hogy a redukciós fázis elvégzése után a tömb utolsó elemét eltároljuk: az adott blokkon belül ennyi elem kerül majd a kimeneti sorba, vagyis ennyivel kell növelni az atomicAdd segítségével a sor elemszámlálóját. Csak ezután folytatódhat a prefixszámítás a második fázissal.

4.3.2. A számítást végző CUDA kernelek

A kernelek indítása előtt a CPU-n előállított SceneData típusú struktúra tartalmát – a felépített BVH-fával együtt – felmásoljuk a videokártya memóriájára, majd beállítjuk textúramemóriaként. Ezt követően lefoglaljuk a számításokhoz szükséges pathSegments névvel elérhető SoA_PathSegments típusú struktúrának a memóriaterületet. A struktúrában található tömbök minden eleme a kép egy-egy pixeléhez tartozik. Ezen kívül a három sornak, a cameraQueue, a shadingQueue és az intersectionQueue számára is memóriát allokálunk. A lefoglalt sorokban legfeljebb annyi elemet lehet elhelyezni, mint a pathSegments elemszáma.

Inicializáló kernel

Beállítja a `pathSegments` tömbjeinek összes elemét a szükséges kezdőértékekre:

- A `curand_init` függvény hívásával inicializálja az álvéletlenszám-generátor állapotváltozókat
- A minták számát és a fényutak hosszát 0-ra állítja
- A radiancia-hozzájárulást (0, 0, 0)-ra állítja
- Az áteresztési tényezőt (1, 1, 1)-re állítja
- A metszett háromszög azonosítóját -1 -re állítja, ezzel biztosítva, hogy a logika kernel majd minden elemet a kamera kernelnek osszon ki, így elindítva a képalkotást

Az inicializáció után, a következő négy kernel egymás utáni hívásának ismétlésével történnek a számítások.

Képalkotási-logika kernel

Mivel a kernel a `pathSegments` összes elemét feldolgozza, nincs bemeneti sora. A kernel feladata elvégezni a képalkotáshoz szükséges teendőket és kiosztani a többi kernel számára a feladatokat.

Ha a kernel futását megelőző iterációban a sugár metszett háromszöget akkor először ellenőrizzük, hogy a háromszög fényforrás-e, mert ha igen hozzá kell adni a minta radiancia-hozzájárulásához a fénykibocsátás mértékének és az áteresztési tényezőnek a szorzatát. Ezután mindenféleképpen frissíteni kell az áteresztési tényezőt a BRDF-nek megfelelően, majd ha fényút hossza elér egy előre megszabott értéket Orosz rulett segítségével döntünk, hogy folytassuk-e a bolyongást. Ha igen, a szál „feljegyzí”, hogy ezt az elemet az árnyaló kernel sorához hozzá kell majd adni.

Ha viszont egyetlen háromszöget sem metszett a sugár, vagy ha az előbbi az Orosz rulett miatt leállítjuk a bolyongást, vagy esetleg az áteresztési tényező nullára csökken, akkor a pixelt – természetesen színkomponensenként –

$$\frac{prev \cdot n + c}{n + 1}$$

értékűre változtatjuk, ahol a *prev* az előző érték, *n* a minták száma, *c* pedig az jelenlegi minta radiancia-hozzájárulása, vagyis újraszámoljuk az átlagot. Ezután 1-gyel növeljük a mintaszámlálót (`numSamples`) és az fényút hosszát és a radiancia-hozzájárulást lenullázzuk és az áteresztési tényezőt (1, 1, 1)-re állítjuk. Végül „feljegyezzük”, hogy az elemet a hozzá kell adni a kamera kernel sorához.

Ezek után, a feljegyzéseknek megfelelően a szálak a blokkszintű folyamprézelés és az elemszámláló értéke alapján kapott helyre írják az elemeket a `cameraQueue` és `intersectionQueue` sorokat megvalósító tömbökben.

Kamera kernel

A kamera kernel a bemeneti sorból kiolvasott index és a kernel hívásakor paraméterben megadott képméret alapján meghatározza, melyik pixelhez kell a sugarat beállítani, kiszámolja a kezdőpontot és az irányt, majd beállítja a `sceneData` struktúrában lévő `origin` és `direction` tömbök megfelelő elemeit. Ezután a bemeneti sor összes elemét hozzáadja a metszéspon-számító kernel sorának végére. Mivel az összes elem átkerül a kernel kimeneti sorába ezért itt most nem kell folyam prézelést végezni ahhoz, hogy a szálak el tudják dönteni hogy a sort megvalósító tömb hanyadik elemét kell írni.

Árnyaló kernel

Az árnyaló kernel feladata, hogy a korábbi metszéspon-t eredmények felhasználásával és az adott háromszög anyagjellemzőjének figyelembevételével, vagyis a BRDF-nek megfelelően előállítsa a sugarak új irányát. A véletlenszámok előállításához a `cuRAND` függvénykönyvtárt használom.

Hasonlóan a kamera kernelhez, a bemeneti sor összes elemét „tovább küldjük” a metszéspon-t-számító kernelnek, így itt sincs szükség folyam prézelésre.

Metszéspon-t-számító kernel

A kernel feladata, hogy a kamera vagy az árnyaló kernel által előállított sugarak háromszögekkel való metszéspon-tját kiszámítsa.

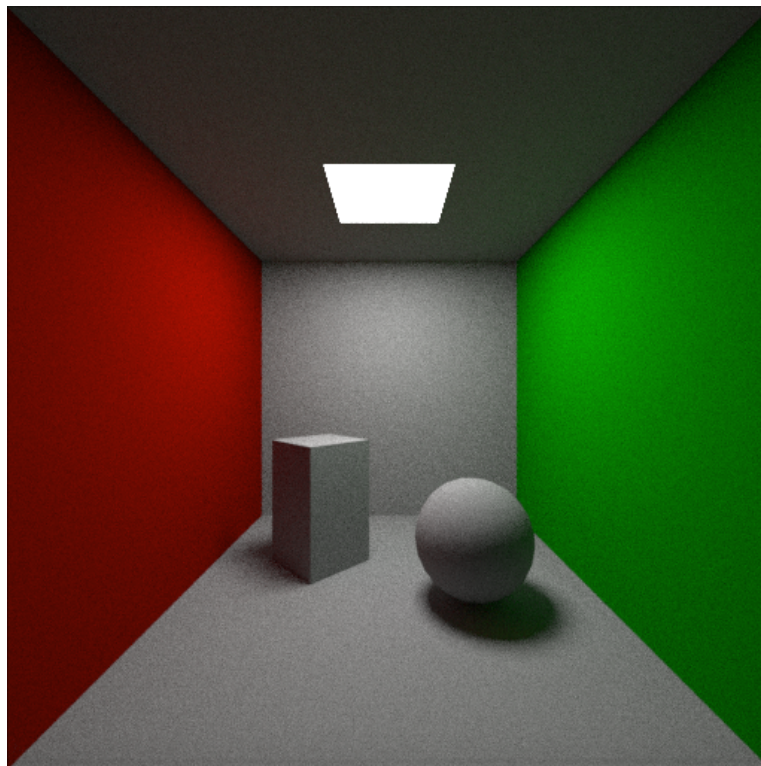
Itt használjuk fel a korábban elkészített BVH-fát a sugár-háromszög metszéspon-t számítások csökkentése érdekében. A fa bejárását hagyományosan rekurzió alkalmazásával lenne célszerű megvalósítani, azonban a CUDA ezt nem teszi lehetővé a régebbi grafikus processzorokon. Szerencsére a problémát megkerülhetjük, ha a fa bejáráshoz szükséges vermet mi magunk kezeljük.

A kernelben minden szál külön veremmel rendelkezik és a veremben azokat a csúcso-kat tároljuk amelynek a gyerekelemeit vizsgálni kell. A fa bejárásának megkezdése előtt ellenőrizzük, hogy a gyökérelemt metszi-e a sugár. Ha igen a verem tetejére beszu-rjuk a gyökérelemt. A fa bejárása addig tart amíg van a veremben üressé nem válik. A bejárás lépéseiben először kivesszük a verem tetején található elemet és a gyerekelemeivel elvé-gezzük a metszéspon-t vizsgálatot. Ha volt metszéspon-t akkor azokat beszu-rjuk a verem tetejére.

A számítások után az eredményeket a kernel a `pathSegments` eredményt tároló `triangleId`, `t`, `u` és `v` tömbjeibe írja.

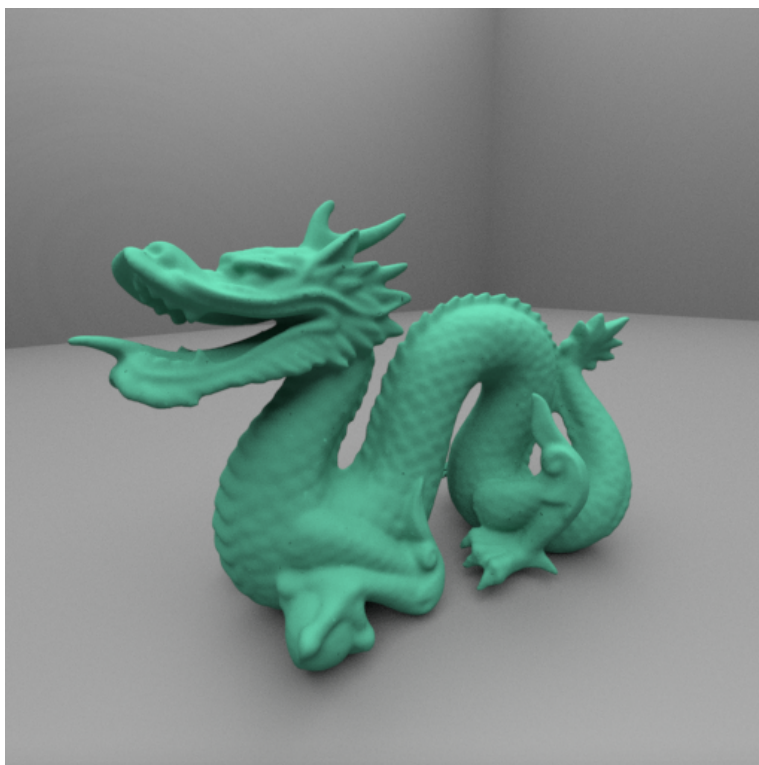
4.4. Eredmények

A képek 512×512 méretűek, és nagyjából 10 perc számítás eredményei. A kernelek futási idejét a CUDA programozási környezetében elérhető nvprof programmal mértem.



4.4. ábra. Cornell-box változat – 350 háromszög

	Teljes futási idő	Átlagos futási idő
Képalkotás-logika	133,057 s	8,1800 ms
Kamera	40,6660 s	2,4998 ms
Árnyalás	148,671 s	9,1389 ms
Metszéspont számítás	310,851 s	19,108 ms



4.5. ábra. Stanford sárkány – 201039 háromszög

	Teljes futási idő	Átlagos futási idő
Képalkotás-logika	46,3606 s	7,7852 ms
Kamera	18,2073 s	3,0575 ms
Árnyalás	52,9279 s	8,8880 ms
Metszéspont számítás	497,524 s	83,547 ms

5. fejezet

Összefoglalás

A dolgozat során elkészített alkalmazással alapvetően sikerült elérni a kitűzött célokat. Megvalósítottam a sugárkövetés gyorsításához szükséges hierarchikus befoglalókeret fa felépítését, aminek köszönhetően még több százezer háromszögből álló modelleket is képes a program kezelni. Az kernelek futási idejéből kiderült, hogy a metszéspontszámítás valóban a képszintézis legidőigényesebb része. A fejlesztés során fellépő technikai kihívások ellenére sikerült megvalósítanom az inverz fényútkövetést a grafikus processzoron több CUDA kernel együttműködésével, valamint a számítások részeredményének folyamatos megjelenítését.

Mindezek mellett, még számtalan lehetőség kínálkozik a továbbfejlesztésre. A gyorsítófa felépítését jelenleg a CPU végzi, ezért érdemes lenne a fa GPU-n való előállításának lehetőségeit is megvizsgálni.

A program több különböző anyagjellemzőt képes betölteni, de jelenleg csak fénykibocsátó és diffúz anyagok vannak megvalósítva. A valóságban ettől azonban jóval összetettebb visszaverődési modellek is léteznek, amelyeket még implementálni lehetne.

Az alkalmazásban megvalósított inverz fényútkövetés csak akkor működik jól, ha a megjelenítendő modelleket nagy méretű fényforrások világítják meg. Kis felületű fényforrások esetén lassabban szűnik meg a becslés zaja, aminek az az oka, hogy a megvalósított algoritmus csak a BRDF fontossági mintavételezésével állítja elő a mintákat, vagyis csak „nehezen találja meg” a fényforrásokig vezető utakat. Ezt a problémát az algoritmus fényforrás-mintavételezéssel való kiegészítésével lehetne orvosolni.

Irodalomjegyzék

- [1] Dietger van Antwerpen. *Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU*. 2011.
- [2] Iványi Antal (szerk.). *Algorithms of Informatics, Chapter 28. General Purpose Computing on Graphics Processing Units*.
http://progmatt.hu/tananyagok/algorithms_of_informatics_volume3/book.html#d5e20797
- [3] Mark Harris, Shubhabrata Sengupta, John D. Owens. *Parallel Prefix Sum (Scan) with CUDA*. GPU Gems 3.
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
- [4] Simuli Laine, Tero Karras, Timo Aila. *Megakernels considered harmful: Wavefront Path Tracing on GPUs*. Proc. High-Performance Graphics, 2013.
- [5] NVIDIA. *CUDA C Programming Guide*. 2013.
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [6] <http://qt-project.org>
- [7] Peter Shirley, R. Keith Morley. *Realistic Ray Tracing, Second Edition*. A K Peters/CRC Press, 2003.
- [8] Szirmay-Kalos László. *Számítógépes grafika*. ComputerBooks, 1999.
<http://sirkan.iit.bme.hu/~szirmay/grafika/graf.pdf>
- [9] Szirmay-Kalos László, Antal György, Csonka Ferenc. *Háromdimenziós grafika, animáció és játékfejlesztés*. ComputerBooks, 2003.
- [10] Ingo Wald. *On fast Construction of SAH based Bounding Volume Hierarchies*. Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing, 2007