

**Valós idejű Sugárkövetés**

**Erdős Zoltán**

**Budapest**

**2019.**

**Tartalomjegyzék**

1. [**Bevezetés 3**](#_Toc4343601)
2. [**Klasszikus DirectX9 és OpenGL2 3**](#_Toc4343602)
3. [**Újítás: Valós idejű DirectX RayTracing és RadeonRays 4**](#_Toc4343603)
4. [**Saját Sugárkövetés program 4**](#_Toc4343604)
   1. [**Elmélet röviden 4**](#_Toc4343605)
   2. [**Gyorsítási lehetőségek 5**](#_Toc4343606)
      1. [**Sugarak indítása OpenCL segítségével, párhuzamosan 5**](#_Toc4343607)
      2. [**Ütközésdetektálás gyorsítás, tér felosztással (BVH) 10**](#_Toc4343608)
      3. [**BVH gyors újra építése, ha a csúcsok megváltoztak 12**](#_Toc4343609)
5. [**Textúrázás Barycentrikus koordinátákkal 15**](#_Toc4343610)
6. [**TriangleShader, VertexShader, RefitTreeShader, RayShader 16**](#_Toc4343611)
7. [**Árnyék számítása RayShader-ben 16**](#_Toc4343612)
8. [**Továbbfejlesztési lehetőségek 16**](#_Toc4343613)
9. [**Felhasználói kézikönyv 16**](#_Toc4343614)
   1. [**Letöltés 16**](#_Toc4343615)
   2. [**Telepítés 16**](#_Toc4343616)
   3. [**A program használata 17**](#_Toc4343617)
10. [**Irodalomjegyzék 17**](#_Toc4343618)
11. [**Ábrajegyzék 17**](#_Toc4343619)
12. [**Mellékletek 17**](#_Toc4343620)
13. **Bevezetés:**

**Régóta létezik 3D-s megjelenítés. A számítógépek teljesítménye nem képes valós időben fénykép minőségű kép előállítására, ezért közelítő megoldásokat találtak ki. A közelítő megoldások sokkal gyorsabbak, elfutnak kisebb teljesítményű számítógépeken, viszont nem élethű képet adnak eredményűl.**

**Az 1995-ös években az akkori játékok a háromszög csúcsainak adataiból számolták ki egy pixel színét a barycentrikus koordináták segítségével (VertexShader), ez gyors. Majd a 2002-es években lehetett a pixelek színét (PixelShader) egyedileg programozni, ez lassabb, de jelenleg ez is elég gyors már. Itt is még csak korlátozott adatok álltak rendelkezésre egy pixel színének kiszámításához. Nem volt információ (egy pixel színének számításakor) arról, hogy a legközelebbi háromszög, ami a képernyőn megjelenik, az mögött mi van.**

**Mostanság a 2016-os években annyira megnőtt a számítógépek teljesítménye, hogy lehet alkalmazni a „sugárkövetést”. Így nem csak a legközelebb álló háromszög adatait ismerjük, hanem a mögötte lévőket is el tudjuk érni. A „sugárkövetés”, amit be szeretnék mutatni, ez a megoldás a mai számítógépeken elfogadható sebességgel fut, élethűbb képet lehet elő állítani vele. Van tükröződés és törés, amivel, ha egy pixel színét számolom, akkor a szomszédos testekről visszaverődő fény színét is számításba tudom venni. De még ez a megoldás sem fénykép minőségű, hiszen sugárkövetésnél pontosan egy sugarat indítok tükröződés és törés irányba. Míg a valóságban van egy kis fény szóródás, mivel a felületek, amin pattan vagy törik a fény, nem tükörsima.  
A „Globális illumináció” az a megoldás, ami túlmutat a jelenlegi sugárkövetésen. Ott az a megoldás, hogy tükröződéskor/töréskor több sugarat indítok, amik nem pontosan tükör/törés irányból pattannak, hanem egy kicsit szóródva. A mai 2019-es otthoni számítógépeknek ez a számítás nagyon sok időbe telik, valós időben nem megoldható. Maradjunk a sugárkövetésnél.**

1. **Klasszikus DirectX9 és OpenGL2:**

**Ide vehető a 2002-ben megjelent VertexShader és PixelShader. Nem látni a háromszögek mögé, csak a legközelebbi háromszöget látjuk.**

1. **Újítás: Valós idejű DirectX RayTracing és RadeonRays:**

**2016-ban, jelentek meg az első valós idejű sugárkövetés megoldások. AMD oldalon, ami platformfüggetlen, a „RadeonRays SDK” jelent meg, ami CPU, OpenCL vagy Vulkan segítségével számol. A Microsoft a „DirectX12 RayTracing SDK”, ami naprakész, viszont (úgy tudom) csak Windows 10-en fut. Az Intelnek és az NVidia-nak is vannak sugárkövetés SDK-juk.**

1. **Saját Sugárkövetés program:**

**Ha vannak nagy cégektől sugárkövetés SDK-k, akkor miért írok sajátot (ami butább)?**

**Kíváncsiságból. Azért is írom, hogy megértsem ennek a működését.**

**A forráskód amit írtam/írok, letölthető a** <https://github.com/ezszoftver/OpenCLRenderer> **weboldalról. OpenCL-t használok a párhuzamos számításokhoz. Szabadon felhasználható, módosítható a forráskód, nincs licensz védelem alatt.**

* 1. **Elmélet röviden:**

**A sugárkövetés „futószalagja” elméletben így működik:**

1. **sugarakat indítok a kamerából, amik elmetszhetnek háromszögeket.**
2. **ha egy sugár-háromszög metszés van, akkor abból az ütközéspontból kiszámolom, hogy mennyi fény éri azt a pontot, majd újabb sugarakat indíthatok tükröződési és törési irányba. Ez a sugár újra elmetszhet egy háromszöget, és kezdődik ez a lépés újra.**
3. **Kb. 3 ütközés után abba hagyom az ütközés keresést, meg vannak a szín információk, amiből a képernyőn megjelenő pixel színét ki tudom számolni. Azért hagyom abba a további ütközéskeresést, mert, ha valós idejű képet akarok előállítani, akkor tovább folytatni gépigényes. Így is jobban megközelítem a valóságot, mint a 2002-es DirectX9/OpenGL2 -vel.**

**Elméletben ennyi. Vajon mik azok a megoldások, amivel rövid idő alatt el lehet a „futószalagot” végezni? Most ezt mutatom be.**

**Fontos! Amit leírok megoldásokat, nem a leggyorsabb megoldások, én ezeket ismerem, ezeket tudom bemutatni.**

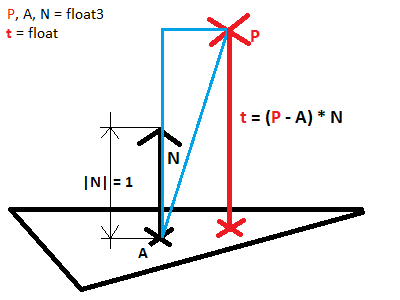
* 1. **Gyorsítási lehetőségek:**
     1. **Sugarak-háromszögek ütközésvizsgálata, párhuzamosan:**

**Először be szeretném mutatni, hogyan lehet egy „sugár-háromszög” metszéspont vizsgálatot elvégezni. Majd bemutatom, hogyan lehet, ha sok sugarunk van, ezt gyorsítani (elöljáróban annyi, hogy ahány sugarunk van, annyi szálat kell indítani OpenCL segítségével, és úgy elvégezni a „sugár-háromszögek” metszéspont vizsgálatot).**

**A „sugár-háromszög” metszéspont vizsgálat lépései:**

* **„pont-sík” távolsága**
* **„sugár-sík” távolsága**
* **metszéspont kiszámítása**
* **metszéspont a háromszögen belül van? vizsgálat**

**pont-sík távolsága:**



**1**. ábra: pont-sík távolsága

Előszöt „float t” -t kell kiszámolni. Tekintsük úgy most a háromszöget, mintha az egy sík lenne. Egy síkot meghatároz egy „**float3 A**” pont, amit a sík elmetsz, és egy „**float3 N**” irány, hogy merre néz a sík.

Ha a „**float3 (P - A)**” vektort skalárisan össze szorozzuk az 1.0 egységnyi hosszú „**float3 N**” vektorral, akkor megkapjuk „float t” -t.

Két „**float3 a,b**” **egységnyi** hosszú vektorok **skaláris szorzat**ára, igaz ez a képlet:

**a \* b = cos(alpha)**

**„(P - A)**” és „**N**” vektorok skaláris szorzatának képlete:

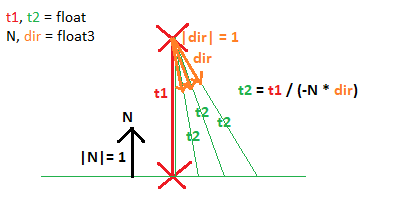
**(P - A) \* N = cos(alpha) \* |P - A|**

**cos(alpha)** eredménye, **[-1.0 .. +1.0]** intervallum béli számot ad eredményűl.

Tehát **|P - A|** hosszt összeszorozzuk egy +1.0 -nál kisebb számmal, így eredményül egy

**|P - A|** -nál rövidebb, „t” hosszt ad eredményül, ami a sík és a „P” pont távolsága.

**sugár-sík távolsága:**



2. ábra: sugár-sík távolsága

**Ha meg van a t távolság, akkor, ha figyelembe vesszük azt, hogy a „P” pontnak van iránya is, akkor egy félegyenest kapunk. Ha az irány megegyezik a -N -el, akkor az a legrövidebb távolság. Ahogyan a „t2” -k mutatják a 2. ábrán úgy, ha minél nagyobb a „-N, dir” közötti szög, annál hosszabb „t2” -ket kapunk eredményűl.**

**metszéspont kiszámítása:**

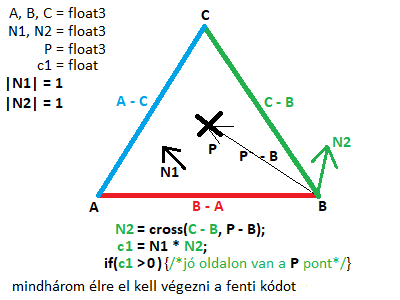
**Azt a pontot, hogy a félegyenes hol metszi el a síkot ezzel a képlettel kapjuk meg:**

**P2 = P + (dir \* t2);**

**„float3 P2” az a pont, amit a sík elmetsz. A „float3 P” és „float3 dir” a félegyenes (sugár) kezdőpontja és iránya.**

**A „float3 dir” hossza, 1 egység.**

**metszéspont a háromszögen belül van?:**

****

**3**. ábra: a metszéspont a háromszögen belül van?

A harmadik lépés, hogy a síkot metsző pont a háromszögen belül van-e? Én azt a megoldást választottam, hogy vektoriális szorzattal ellenőrzöm ezt. A háromszög mindhárom élére el kell végezni, a „jó oldalon van a pont?” ellenőrzést. Én egy élre mutatom be az ellenőrzést:

Vegyük B-C szakaszt. Ha vektoriálisan össze szorzom (C - B) és (P - B) vektorokat, akkor a képen látható N2 vektort kapom eredményűl. Fontos, hogy a vektoriális szorzat nem kommutatív, vagyis, ha nem ebben a sorrendben, hanem fordítva végzem el a vektoriális szorzatot (cross(P - B, C - B)), akkor -N2 -t kapok. Majd szög ellenőrzéssel (skaláris szorzat) ellenőrzöm, hogy jó oldalon van-e a P pont. Ha a szög 90foknál kisebb, akkor jó oldalon vagyok. De mind a három élre igaznak kell lennie ennek a feltételnek ahhoz, hogy a „háromszögen belül vagyok-e?” kérdés igaz legyen. A vektoriális szorzatnál figyelembe kell venni azt is, hogy nem mindegy, hogy (C - B) vagy (B - C), mert, ha felcserélem a pontokat, akkor ellenkező irányba fog mutatni a vektor. Én az óramutató járásával ellentétes irányt választottam

Tehát el tudjuk mostantól végezni a félegyenes-háromszög ütközésvizsgálatot. Félegyenes alatt sugarat, vagy ray -t is mondhatnék, mindhárom szó most ugyanazt jelenti. Amikor egy képet akarok előállítani sugárkövetésesl, akkor minden egyes pixelből sugarat kell indítani a világba. Tehát nagyon sok sugarat kell indítanom. Két pixelnek a textúrából, nincs köze egymáshoz, tehát két sugárnak sincs köze egymáshoz. Nincs függőség, vagyis pl. a (0,0) pixelből indított sugár, nem várakozik a (0,1) pixelből indított sugárra, tehát párhuzamosan, külön-külön szálakon lehet elindítani a sugár-háromszög metszésvizsgálatot.

Egy videokártyában kb. 100-2000 darab kis órajelű (500Mhz – 1GHz) processzor van. Ezeket a processzorokat el lehet érni C nyelven, „OpenCL” segítségével. Tehát tudok párhuzamosítani. A CPU abban más a GPU-k tól, hogy magasabb órajelen működnek, kevesebb van belőlük (1 - 16), viszont összetettebb, az egész számítógépet vezérlik. CPU-n lassabban fut egy kép számítás, mint egy videokártyán. Pont azért találták ki a videokártyát, vagyis egy külön hardvert képszámításra, mert az a képszámításra van optimalizálva, gyorsabban kiszámolja a képet.

OpenCL-ben, „Buffer” -ekben vannak tárolva az adatok. Egy „Buffer” osztály, más néven tömb. Meg lehet adni a „Buffer” -nek, hogy milyen típusú adatokat akarok benne tárolni: Buffer<int> egesz\_szamok;. Ez a buffer a videokártya memóriájában jön létre. Általában van egy bemeneti Buffer, amivel számol, és van egy kimeneti Buffer, ahova az eredmények kerülnek. Majd a Buffer-t ha kell, vissza lehet másolni az operációs rendszer memóriába, és lehet az eredményekkel tovább számolni.

Tehát van sok sugaram, ez egy „Buffer” (Buffer<Ray> rays). Illetve van a háromszögeket tartalmazó „Buffer” (Buffer<Triangle> triangles). ezek a bemenő bufferek. Kell egy kimeneti buffer is, ami megmondja, hogy egy sugár elmetszette-e a „triangles” bufferben lévő háromszögek valamelyikét. Ezek az adatok az eredmény (Buffer<hit> hits).

Két párhuzamosan, külön-külön szálon futó Ray, ugyanazt a „triangles” buffert használja, baj ez? Nem, mert csak olvasnak belőle, nem módosítják.

Egy videokártya, kb. 10x gyorsabban kiszámol egy képet a párhuzamosítás miatt, mint egy vele egyenértékű CPU.

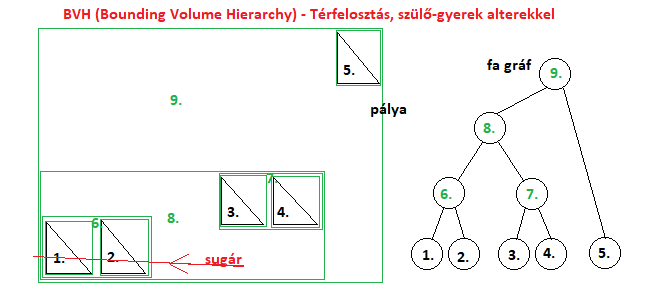
A sugarakat párhuzamosítottuk. Most nézzük meg, hogy mit lehet tenni a háromszögekkel, ott is gyorsítani kéne.

* + 1. **Ütközésdetektálás gyorsítás, tér felosztással (BVH):**

**Eddíg egy sugár végigjárta, az összes háromszöget, és úgy kereste a hozzá legközelebbi, elmetsző háromszöget. Nem lehet ezen gyorsítani? De lehet, tér felosztással.**

**Képzeljük el, hogy van a világ. Az legyen mondjuk 1km hosszu, 1km magas, 1km széles). Itt vannak a háromszögek össze-vissza. Van egy sugarunk, pl. (10,10,10) pontban. Felesleges azokkal a háromszögekkel ütközéstdetektálást végezni, amik nagyon messze, pl. (1km, 1km, 1km) távolságra vannak a Ray-től. Jobb lenne, csak a Ray-hoz közeli háromszögeket vizsgálni csak. Megoldás, daraboljuk fel a teret, minden altérbe másoljuk bele, a teret elmetsző háromszögeket. Majd ha jön egy sugár, akkor számoljuk ki hogy a sugár megy tér részeket metsz el, és csak az azokban a tér részekben lévő háromszögekkelveégezzünk ütközés detektálást.**

**Én itt, a BVH (Bounding Volume Hierarchy) megoldást választottam. Jelentése: alterek, szülő-gyerek kapcsolatban, vagyis hierarchiában.**



**4**. ábra: BVH (alterek a gyors kereséshez)

A kép bal oldali része az altereket, világot mutatja jól, míg a kép jobb oldali része, a gráf, a szülő gyerek kapcsolatot mutatja jól.

Az 1,2,3,4,5 csomópontok, háromszögek, míg a 6,7,8,9 csomópontok, tér részek. A 9. csomópont, a gráf szerint, az a gyökér, az az egész világot magába foglalja.

Vizsgáljunk a sugár szemszögéből: amit elmetsz az a 9, 8, 6 -os alterek, és az 1, 2 -es háromszögek. Felesleges vizsgálni a 3, 4, 5-ös háromszögekkel a metszésvizsgálatot.

Hogyan kapjuk meg az 1 és 2-es háromszögeket? Kezdjük a metsző háromszögek keresést a fa gráf bejárásával.

* A sugár elmetszi a gyökeret? (9) => igen, tehát vizsgáljuk meg a 9. csomópont gyerekeit.
* A sugár elmetszi a 5 alteret? nem, tehát erre nem folytatjuk a keresést.
* A sugár elmetszi a 8-as alteret? igen, vizsgáljuk meg ennek az altérnek/csomópontnak a gyerekeit.
* a sugár elmetszi a 7-es alteret? nem, erre nem keresünk tovább.
* a sugár elmetszi a 6-os alteret? igen, akkor vizsgáljuk ennek az altérnek/csomópontnak a gyerekeit.
* a sugár elmetszi az 1 háromszöget? igen
* a sugár elmetszi a 2-es háromszöget? igen

Levél: Az a csomópont, akinek nincsen gyereke, vagyis egy háromszög van benne.

Ha eljutottunk egy levélig, akkor háromszög-sugár metszést kell vizsgálni.

Ha nem levélben vagyunk, akkor „sugár-altér” metszésvizsgálatot kell csinálni.

Mindkét háromszögre megkapjuk a „t” távolságokat. Nekünk a kisebb értékű „t” (háromszög) kell, számoljuk ki milyen textúra szín és fény éri azt a pontot, és tároljuk el azt a színt.

Ez a térfelosztás azért jó, mert, ha pl. 1millió háromszögből áll egy pálya, akkor az 1millióhoz képest kevés altér vizsgálattal eljutok a „nagy valószínűségű, hogy metsző” háromszögekig.

* Tegyük fel, hogy van 1millió háromszögem. Ha nem lenne térfelosztás, akkor egyesével, minden háromszöget vizsgálni kéne, ez 1 millió vizsgálat.
* De ha BVH segítségével keresünk, akkor kb. 100 vizsgálattal megkapom a metsző háromszögeket. Kevesebb így a vizsgálat.

Hogyan lehet egy háromszögek listából, BVH fát felépíteni?:

Adottak a háromszögek. Első lépésben veszek egy háromszöget, és megkeresem a hozzá legközelebbi másik háromszöget. „Csúcs-csúcs” vizsgálat elég, mert általában a felületek folytonosak, zártak mindig van egy háromszögnek egy olyan csúcsa, ami egy másik háromszöghöz is tartozik. Ebből a két szomszédos háromszögből, egy csomópontot lehet csinálni. A csomópontot egy „List<Node> nodes” listába teszem és a két háromszöget törlöm a „háromszögek listájából”. Majd veszem a következő háromszöget a háromszögek listájából, és ugyan ezt a „szomszéd keresés” algoritmust futtatom, majd a megszületett csomópontot hozzá fűzöm a „nodes” listához. Addíg keresem egy háromszög szomszédos háromszögeit, amíg vannak háromszögek a „triangles” listában. Előfordulhat, hogy csak egy háromszög maradt a listában, annak nem tudok szomszédot találni, így belőle egy olyan csomópontot hozok létre, aminek csak egy gyereke van.

a „nodes” listában, most sok kicsi altér van. Ezekkel az alterekkel ugyanúgy elvégezzük a szomszéd keresést, ugyan úgy, mint a háromszögeknél. Mindig, az újjonan keletkező csomópontokat bele tesszük egy új „List<Node> out” listába, majd ha az „List<Node> in” listából, ha elfogytak a csomópontok, akkor az „in = out; out = new List<Node>();” (az out lista az in listába kerül, és egy új, üres out Lista jön létre), és kezdődik elölről a szomszédok keresése.

Egyszer eljutunk egy olyan állapothoz, amikor az in Listában egy elem lesz. Az lesz a gyökér elem.

(Amikor létre hozunk egy csomópontot, akkor mindig kiszámoljuk a gyerekei altérből, az aktuális alteret, amiben mindkét gyerek altér benne van).

Így létre jön egy BVH fa.

Ez a gráf addig „jó”, amíg a háromszögek mozdulatlanok. De a számítógépes grafikában a háromszögek mozognak. Pl. animáció. Hogyan lehet egy már felépített fát, amiben, ha elmozdul egy háromszög (valamelyik csúcsa), akkor újra „jóvá” tenni? Lehet ezt párhuzamosan számolni? Igen.

A következő rész egy fa „újra jóvá tételét” mutatja be.

* + 1. **BVH gyors újra építése, ha a csúcsok megváltoztak:**

**Mi van akkor, ha egy BVH fa háromszögeinek csúcsai transzformálódott? Újra kell építeni az egész fát? Nem.**

**Két eset lehet:**

* **Animációkor, a háromszögek szomszédsága megmarad, pl. felemeljük a kezünket. Hagyományos animációkor ez az állítás érvényes.**
* **Animációkor a háromszögek szomszédsága nem marad meg, pl. levágják egy ember kezét, és messzire eldobják.**

**Én azt a megoldást választottam itt, hogy mindkét esetben a háromszögek szomszédságán nem változtatok a BVH-n belül.**

**Első esetben ez nem gond, a háromszögek szomszédsága ugyanúgy megmarad, csak az altereket kell a gyerekektől a szülők felé újra számolni.**

**Második esetben, „amikor messzire repül a kéz”, akkor is csak az altereket számoljuk újra, igaz ilyenkor nem lesz optimális a BVH bejárás, mert több 10 méter is lehet a távolsága a kéz és ember között, pedig biztosan lenne közelebbi háromszög.**

**Sajnos nem tudok megoldást, amivel gyorsan a háromszögek szomszédságát újra lehetne számolni. De ha a csomópont altereket újra számoljuk, függetlenül attól, hogy nem tökéletes a szomszédság, így is gyorsabb a fa bejárással, a metsző háromszögek keresése, mintha egyesével járnánk végig az összes háromszöget, metszéspontot keresve.**

**Megoldás: „ne szakadjon le a kéz”. Maradjon meg a szomszédsági viszony. De ez nem lehetséges (mindig).**

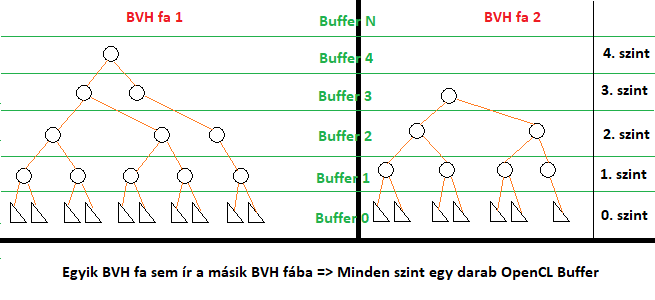
**Az altereket hogyan lehet párhuzamosan számolni?:**

* **tudjuk azt egy fa gráfról, hogy vannak szintjei. Én most megfordítom a szintek sorszámozását. Legyen a 0. szint a levelek, vagyis a háromszögek szintje. az 1. szint, a háromszögek szülői, … az N. szint pedig a gyökér.**
* **Tudjuk azt is, hogy egy pl. 5. szintet csak akkor tudom párhuzamosan számolni, ha a 4. szint alterei már kiszámításra kerültek.**

**Tehát a 0. szintű háromszögekből inicializáláskor készítek egy OpenCL „Buffer<Node> level0” buffert, ő egyben in/out buffer és kiszámolom a háromszögek altereit OpenCL-el. Így a 0. szint altereit kiszámolta az OpenCL, párhuzamosan, el lehet kezdeni az 1. szintű Node -k altereinek számolását. Fontos: Egy Node egyszerre altér és háromszög. Onnan tudom hogy egy Node levél (vagyis hogy háromszög), hogy nincs egy gyereke sem.**

**Minden szint egy „Buffer<Node>”. Ezt előfeldolgozási lépésben ki lehet számolni, hiszen egy BVH fa szintjei mindig ugyan azok maradnak, csak a levelek változnak.**

**Sorra kiszámolom a 2,3,4,5 … N -edik szintig párhuzamosan a szintek csomópontjainak altereit OpenCL-el, a gyerekek altereiből. Így frissítettem egy BVH fát.**



**5**. ábra: BVH fák szintjei. Egy szint elemei párhuzamosíthatók OpenCL-el

**Sőt ahogy a kép is mutatja, ha sok BVH fa van, legyen mondjuk 2 darab (vagy több), amikor készítem pl. 0. szintet, akkor össze lehet fűzni mind a 2 darab 0.ás szintű level-eket, és egy nagy level0 Buffer keletkezik. Ugyanígy level N-ig. Mivel egyik BVH fa sem ír/olvas a másik BVH fa Node -jéből/-jébe, függetlenek egymástól, ezért összefűzhetők.**

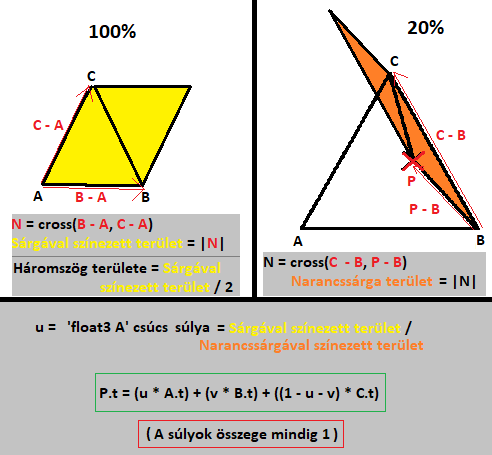
**Egy fa szintje, kb. 15-25 lehet. Ha kiszámoljuk, ha 25 szintű a fa, akkor elfér benne (2^25) 33 millió háromszög a levelekben, BVH-nként. És csak 25 darab OpenCL függvényhívást kellett a megfelelő sorrendben meghívni, akkor is, ha több BVH fa van, és újra „jóvá” tettük a fát.**

**Különbséget kell tenni animált és nem animált BVH fa között. Az animált BVH fa „Dynamic” típusú, a nem animált BVH fa „Static” típusú. Elég csak a „Dynamic” típusú BVH-knak a altereit frissíteni. Azt hogy egy BVH fa Static vagy Dynamic lesz-e, azt a programozó dönti el. Egy Static típusú BVH-nak sohasem változnak meg a háromszög csúcsainak pozíciói, így azt elég egyszer, inicializáláskor az altekreket frissíteni.**

**Statikus BVH-nak számít a mozdulatlan pálya, míg Dynamic BVH-nak számít az animált-, vagy a térben máshová kerülő tárgyak.**

**Az OpenCL level1,2, .. 25 Bufferekbe csak a Dynamic típusú BVH fákat kell bele tenni.**

1. **Textúrázás Barycentrikus koordinátákkal:**



**6**. ábra: textúrakoordináta számítása, a P pontban

Ha ismerem az A-, B, és C csúcshoz tartozó textúra koordinátákat, és ki szeretném számolni a P metszéspont textúra koordinátáját, azt hogyan kell? Súlyokkal.

Képzeljük el, hogy ha egy P metszéspont közel van az A csúcshoz, akkor az A csúcs textúra koordinátához közeli értékű lesz a P csúcs textúra koordinátája. Minél messzebb kerül a P pont az A csúcstól, és minél közelebb kerül a B csúcs felé, annál inkább a B csúcs textúra koordinájához közeli értéket fog a P csúcs textúra koordinátája fel venni. Ugyan ez a C csúccsal.

Súlyokat kéne létre hoznom, amik megmondják [0.0 .. 1.0] intervallumban, hogy milyen közel vagyok egy csúcshoz. Ha nagyon közel vagyok pl. az A csúcshoz, akkor az A csúcs súlya 1.0-hoz közeli szám, és a B és C csúcsok súlya 0.0-hoz közeli szám. Az A csúcs textúra koordinátája fog jobban részt venni a P csúcs textúrakoordinátájának számítása közben, a B és C csúcsok, közel 0%-al fognak részt venni. A súlyok összege 1.0-et kell hogy kiadjon.

Hogyan tudom az A csúcs „float u” súlyát kiszámolni? Ahogyan a kép is mutatja, ha kiszámolom a teljes háromszög területét, ez legyen „t1”. Majd, ha kiszámolom a vele szemközti al háromszög területét, legyen ez „t2”. Majd „float u = t2 / t1”. Így egy kisebb számot osztok egy nagyobb számmal, pont az A csúcs súlyát fogom megkapni. A mellékelt ábra az A csúcs súlyának kiszámítását mutatja be.

Számoljuk ki a B csúcsal szemközti kis háromszög területét, ez legyen „t3”. Majs a súly: „float v = t3 / t1”. A C csúcs súlyát ugyan ezen elven lehet kiszámolni, de felesleges, mivel tudjuk hogy „u + v + w = 1”, ebből következik, hogy a C scsúcs súlya: „1 – u - v”.

Ismerjük a három súlyt, alkalmazzuk, ezt képletet: „P.t = (u \* A.t) + (v \* B.t) + ((1 – u - v) \* C.t)”. Így megkapjuk a P pontban lévő textúra koordinátát. Ugyan ezzel a módszerrel, nem csak textúra koordinátát, hanem normálvektort, vagy pozíciót is lehet számolni. A DirectX9, OpenGL2.0 ezzel a megoldással számolta ki egy pixel koordinátáit, ez a gyors, nem „háromszög-sugár” ütközésvizsgálatot végzett.

Hogyan kell kiszámolni egy (nem derékszögű) háromszög területét? Ahogyan a kép is mutatja, egy A, B, C csúcsú háromszög területe egyenlő

„length(cross(B - A, C - A)) / 2.0”. „cross()” a vektoriális szorzat, „length()” a vektor hossza. És osztani kell kettővel.

1. **TriangleShader, VertexShader, RefitTreeShader, RayShader:**
2. **Árnyék számítása RayShader-ben:**
3. **Továbbfejlesztési lehetőségek:**
4. **Felhasználói kézikönyv:**
   1. **Letöltés:**
   2. **Telepítés:**
   3. **A program használata:**
5. **Irodalomjegyzék:**
6. **Ábrajegyzék:**
7. **Mellékletek:**