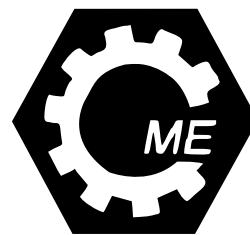


TDK-dolgozat

Horváth Gábor



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR
ÁBRÁZOLÓ GEOMETRIAI TANSZÉK

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Sugárkövetés GPU támogatással

Készítette:

Horváth Gábor

Mérnök informatikus (BSc) alapszak

2007. évfolyam

Infokommunikációs rendszerek szakirány

Konzulens:

Dr. Juhász Imre

Tanszékvezető egyetemi tanár Ábrázoló Geometriai Tanszék

MISKOLC, 2012

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

Tartalomjegyzék

Bevezetés	1
1. Sugárkövetés GPU-n	2
1.1. Ismert megvalósítások	3
1.2. GPGPU	4
1.3. Alkalmazásfejlesztési felületek	5
1.3.1. Gépi kódú programozás	5
1.3.2. GLSL	5
1.3.3. OpenCL	6
1.3.4. Nvidia CUDA	6
1.4. Sugárkövető algoritmusok	7
1.4.1. Befoglaló doboz metszése sugárral	7
1.4.2. Háromszög metszése sugárral	8
2. Térfelosztás	10
2.1. A Befoglaló doboz (<i>bounding box</i>) módszer	11
2.2. Az októlis fa térfelosztási módszer	11
2.3. A tér felosztása kd-fa segítségével	12
2.3.1. A kd-fa generálása	13
2.3.2. A kd-fa tárolása	14
2.3.3. A kd-fa bejárása	16
3. Megjelenítés	18
3.1. Színek és sugarak	18
3.2. Nagy dinamikatartományú fény	19
3.3. Árnyalás	19
4. A rendszer használata	21
4.1. A kd-fát előállító segédprogram	21
4.2. A sugárkövető alkalmazás	22
4.2.1. Erőforrások betöltése	23
4.2.2. Megjelenítés	24

5. Összefoglalás	25
Irodalomjegyzék	30

Bevezetés

Az optikai sugárkövető (*raytracing*, vagy *raycasting*) algoritmusok impresszív, részletgazdag és valósághű grafikai szemléltetést tesznek lehetővé. A rekurzív sugárkövetés módszere már régóta ismert, a fénysugarak útját visszafelé követve, a fénytörési törvényeinek megfelelően számíthatunk tükröződést, árnyékot és akár fénytörést is. A sugárkövetés szépsége abban rejlik, hogy a többi képalkotó algoritmushoz képest ez áll a legközelebb a fizikai valósághoz.

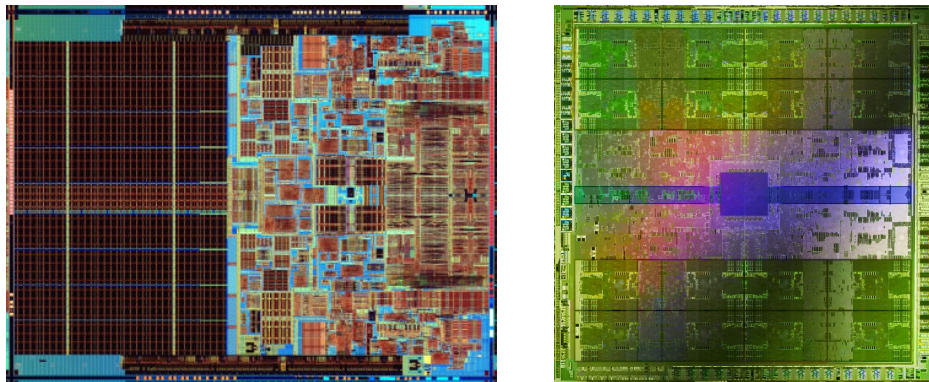
A modern grafikus kártyák inkrementális képszintézist alkalmaznak a térbeli világ objektumainak leképzésére. Ez a módszer elvonatkoztat a pixelek atomi jellegétől és a takarási, illetve árnyalási számításokat az objektumtér egy magasabb szintjén – *fragmentumonként* – végzi el. Ez a módszer azonban azt feltételezi, hogy a virtuális világban csak sokszögek találhatók, ezért például a felületeket sokszögekkel kell közelítenünk [15]. A grafikus kártyák kifejezetten erre az algoritmusra specializálódtak, ám szerencsére az utóbbi időben sok általános célú felhasználásra irányuló törekvés jelent meg. Ez tette lehetővé, hogy a grafikus hardvereken olyan számításigényes algoritmusokat is hatékonyan meg tudjunk valósítani, mint például a rekurzív sugárkövetés. A sugárkövetés egyik fontos tulajdonsága, hogy a vizsgált sugarak teljesen inkoherensek, ezért a hozzájuk rendelt képpont színének kiszámítása könnyen párhuzamosítható feladat.

A kutatásom célja olyan rekurzív sugárkövető rendszer megvalósítása, ami valós időben képes valószerű grafikus szemléltetésre egy napjainkban már átlagosnak számító grafikus hardveren is.

1. fejezet

Sugárkövetés GPU-n

A sugárkövetéses képalkotás folyamatának gyorsítására az egyik legjobb módszer, ha az algoritmust nem a központi feldolgozó egységen (CPU-n), hanem a GPU-n (*Graphics Processing Unit*), azaz a grafikus kártya feldolgozó egységén hajtjuk végre. A GPU-kat elsősorban grafikai szemléltetésre fejlesztették ki, ám a technológia fejlődésével, utasításkészletük kiteljesedett, így egyre inkább általános felhasználású processzorokká válnak. A grafikus hardvereknek egyre több új alkalmazási területe jelenik meg, ezért az új generációkat már inkább *stream processor*-oknak tekinthetjük [12].



1.1. ábra. Az Intel Core2 Duo és az Nvidia Fermi processzormagjának röntgenképe

A stream processzorok abban térnek el a hagyományos műveletvégző egységektől, hogy a számításhoz szükséges adatokat egy adatfolyamból olvassák ki. A grafikus rendszer egy programot, úgynevezett *kernel*t hajt végre a bemenő adatfolyam minden elemére, majd az eredményeket a kimenetre helyezi. A különböző műveletek egymás után való elvégzését grafikus csöveknek (*pipeline*) nevezzük, mely analógia legjobban a CPU-k utasításkészletének SIMD (*Single Instruction Multiple Data*) kiterjesztésére hasonlít.

A GPU magok kialakításukban is alapvetően eltérnek a központi műveletvégző egységektől. Míg a CPU magok fizikai felületének felén általában a gyorsítótárak (L1 és L2 cache-ek) helyezkednek el, addig a GPU magoknál szinte alig van szükség ezekre

a memóriákra, ahogy az 1.1. ábrán is jól megfigyelhető ez a különbség. Ennek az oka az, hogy a CPU esetében gyakoriak az ismétlődő memóriaelérések, melyeket – hogy elkerüljük a memória olvasás okozta késedelmeket – gyorsító tárba kell helyezni, a GPU-knál azonban nincs szükség ilyen kiterjedt gyorsítótárra, hiszen az adatfolyamok elemei egymástól függetlenek és feldolgozásuk folyamatosan történik.

A stream számítási modell két legfontosabb tulajdonsága, hogy az adatfolyam elemeinek számítása egymástól független, ezért a folyamat jól párhuzamosítható, valamint hogy a kernelekkel magas aritmetikai intenzitást érhetünk el. Ezen tulajdonságai miatt a GPU a legjobb választás az olyan algoritmusok számára, mint a sugárkövetés.

1.1. Ismert megvalósítások

Az interaktív sugárkövetésnek már számos megvalósítása létezik. Ezek közül ismertetek néhányat.

Magas – több tízmilliós – poligonszámú modellek kezelése az inkrementális képszintézis algoritmusaival igen nehézkes. A fragmentumok kirajzolásának optimalizálására a távoli és közeli vágósík és a nem látható hátsó lapok elhagyásán alapuló módszereken kívül nem áll rendelkezésünkre más lehetőség. Ilyen esetekben a sugárkövetés gyorsabbnak bizonyulhat, hiszen itt lehetőség van a tér felosztására, illetve egyéb előszámítást igénylő módszerek alkalmazására, mint például a részletességi szint *level-of-detail* (*LOD*), vagy a lehetséges látható halmaz *potentially visible set* (*PVS*) technikák. Ilyen rendszert valósított meg Ingo Wald 2001-ben, amivel bonyolult modelleket járhattunk be valós időben [5].

Hasonló elven működő alkalmazást mutatott be 2010-ben Áfra Attila Tamás, aki az algoritmust tovább gyorsította *voxel* alapú részletességi szint segítségével [1]. Ebben a megvalósításban a modell primitíveit előzőleg *kd-fába* (*kd-tree*) rendezi, ahol minden harmadik mélységnél egy részféval (*treelet*) egy voxelt alkot. Az algoritmus minden sugár által metszett voxelre kiszámol egy *LOD* hiba értéket, ha ez a szám a meghatározott küszöbérték (*pixels of error* (*PoE*)) alatt van, a sugarat a voxelrel metszi és befejezi a további bejárását az aktuális képpontra.

Mindkét alkalmazás rendkívül hatékonyan valósítja meg a sugárkövető algoritmust, azonban mindkét esetben csak a CPU erőforrásaira támaszkodtak a szerzők. Azonban léteznek megvalósítások GPU-n történő sugárkövetésre is, ezek általában *shader*ek segítségével valósítják meg az algoritmust. Egyik legkorábbi megvalósítását 2002-ben mutatta be Timothy J. Purcell és a Stanford Graphics Lab csapata [12]. Alkalmazásukban egy *shader* programot használnak a sugárkövetéshez és a szükséges adatstruktúrákat, így például a *grid*-et is *3D texture map*-ekben tárolják, ami egy támogatott memóriaszerkezet a modern videokártyákon.

A sugárkövetés egy rendkívül hasznos alkalmazási területét mutatja be a Budapesti

Műszaki Egyetemen tanító Szirmay-Kalos László, aki hallgatóival egy sugárkövetésen alapuló *environment map* technikát mutatott be [14]. A megoldás lényege, hogy a környezet képének textúráját dinamikusan hozzák létre úgy, hogy az egyes *texeleket* sugárkövetéssel számítják ki. Mivel az így kapott környezet képe függ az objektum térbeli helyétől, ezzel a szemléltetéssel valósághű tükröződést érhetünk el viszonylag alacsony számítási költséggel. Megvalósításukban DirectX alkalmazásfejlesztési felületet és shadereket alkalmaztak.

1.2. GPGPU

A GPU kínálta lehetőségeket kihasználva módunkban áll olyan számításigényes, de jól párhuzamosítható algoritmusok gyors elvégzésére, mint a sugárkövetés. Sajnos a rendelkezésre álló grafikus programozási felületek inkább a inkrementális képszintézis kiegészítésére szolgálnak és nem támogatnak ilyen – elvben más – szemléltetési módszereket. Szerencsére ma már lehetőségünk van általános célú programokat írni a grafikus chip-ekre, ezt a technikát nevezzük *General-Purpose computing on Graphics Processing Units*-nak, azaz GPGPU-nak.

Már régóta használnak GPU-kat a grafikus kártyákban a számításigényes térbeli transzformációk és a képszintézis gyorsítására. Ezeknek az egységeknek a felhasználása régebben csak grafikai megjelenítésre korlátozódott, használatuk csak az erre kifejlesztett alkalmazásfejlesztési felületeken volt lehetséges, melyek közül modern hardverre az OpenGL és a DirectX terjedt el leginkább. Általános célú felhasználásának lehetőségére csak az utóbbi években került sor.

Az első programozható grafikus csöveket (*pipelines*) a DirectX 8 specifikációjában mutatták be, ez volt a Shader Model 1.1. A shader programozás lehetővé tette, hogy a programozó végrehajthassa saját programját minden csúcspontra (*vertex shader*) és minden pixelre (*pixel shader*), így befolyásolva az eddig kötött renderelési folyamatot. A shaderok persze nem távolodtak el a klasszikus API-k fragmentum alapú renderelési eljárásától, csupán lehetőséget kínáltak különböző hatások megvalósítására. Korunkban a legnagyobb videokártya-gyártó cégek létrehozták saját általános célú programozási felületeiket és függvénykönyvtáraikat, azonban ezek tipikusan csak a saját gyártmányú kártyáikon futottak. Az GPU általános célú számításra való használata viszonylag új keletű szemléletmód és csak az újabb grafikus kártyák támogatják.

A GPU legnagyobb előnye, hogy az adatfolyamokon lefutó programok (kernelek) rendkívül jól párhuzamosíthatóak (*stream processing*). A kerneleket a folyamat minden elemére elvégezzük, mint például amikor minden csúcspontot beszorzunk egy mátrixal, azaz transzformálunk. A GPGPU-t támogató grafikus kártyákon lehetőségünk van ilyen kerneleket írni, majd ezeket végrehajtani saját *stream*-ünkön. A GPU általános célú felhasználása nem korlátozódik a grafikus megjelenítésre, bármilyen számítást elvégez-

hetünk rajta, legyen az keresés egy adatbázisban, gyors Fourier-transzformáció, vagy digitális jelfeldolgozás. A felhasználási lehetőségek száma szinte végtelen. Sok szakterületen találkozhatunk számításigényes műveletekkel, melyek könnyen párhuzamosíthatóak, a GPGPU technika erre szolgáltat megoldást.

1.3. Alkalmazásfejlesztési felületek

A sugárkövetés megvalósítására számos általános célú alkalmazásfejlesztési felület áll rendelkezésünkre, ezek közül ismertetek néhányat, a legegyszerűbb és leghardverközelibbtől, a legmodernebb és a legáltalánosabbig.

1.3.1. Gépi kódú programozás

A grafikus kártyák programozhatóak szubrutinhívással és memóriacímek írásával, ilyenkor a videokártya rendszeresen kiolvassa (általában DMA hozzáféréssel) az adott memóriaterületet és megjeleníti azt [8]. Lehetőségünk van gépi kódot futtatni a hardveren, ami rendkívül gyors és jól optimalizálható, valamint a teljes utasításkészlet ismeretében minden algoritmus hatékonyan megvalósítható. Az utasításkészlet függ a gyártótól, sőt a grafikus chip modelljétől is, ezért a fejlesztés kimondottan egy célhardverre irányul. Még ha több hardverre is fejlesztünk, előfordulhat, hogy a gépi kódunk az új – fejlettebb – grafikus kártyák lehetőségeit már nem használja ki teljesen, vagy akár nem is tud futni az adott hardveren. Egy multiplatform sugárkövető algoritmus számára biztosítanunk kell a portabilitást, hogy a lehető legtöbb hardveren gond nélkül futhasson. A VGA gépi kódú programozása ezt nem teszi lehetővé, ezért mindenképpen magasabb szintű általános nyelv használata ajánlott.

1.3.2. GLSL

Az OpenGL 2.0 specifikációjában mutatták be először az OpenGL saját Shader nyelvet, a OpenGL Shading Language-t. A GLSL-ben lehetőségünk van közvetlenül befolyásolni a grafikus csöveket (pipelines). A grafikus csövek programozásának lehetősége már korábban létezett, viszont szigorúan gépi kód szintű nyelvekkel, melyből egy egész készletre volt szükségünk, ha különböző hardvereken szeretnénk volna futtatni grafikus alkalmazásunkat. A GLSL-ben azonban egy C-hez hasonló, magas szintű nyelven írhatjuk meg vertex és fragmentum shadereinket, mely a kártya saját gépi kódjára fordul le. A platformfüggetlenség mellett fontos megemlíteni, hogy a grafikus kártyagyártók az utóbbi időben nagy figyelmet szenteltek a shaderek gyorsítására, skálázhatóságára és optimalizálására. A GLSL is jól támogatott, ismert szabvány, mely ma már a 4.30.6 specifikációnál tart.

A kutatómunkám részeként fejlesztett alkalmazásban, a grafikus megjelenítések során én is használok GLSL-ben írt shadereket, kifejezetten utófeldolgozás céljából. Ez a megoldás nemcsak hatékony, de kézenfekvő is, hiszen a sugárkövetés eredményeként kapott képet OpenGL segítségével jelenítem meg, így a renderelési folyamat során sem másolódik adat az operatív tárba. Ez a megoldás kiküszöböli a memóriamásolások okozta teljesítménybeli szűk keresztmetszetet, ami – a DMA elérés ellenére is –, a rendszer legkritikusabb folyamata lenne.

Bár a GLSL biztosítja a portabilitást és ma már szinte minden grafikus kártya támogatja, mégis a klasszikus inkrementális képszintézishez kötődik. A sugárkövető algoritmus megvalósításakor az elvi különbségek miatt problémák léphetnek fel, ami a teljesítmény vagy akár a további fejlesztések rovására válhat. Éppen ezért megfontolandó ezt az alkalmazásfejlesztési felületet használni.

1.3.3. OpenCL

Az OpenCL keretrendszert eleinte az Apple fejlesztette, majd 2008 júniusában adta ki egy non-profit technológiai konzorciumnak, a Khronos Group-nak. Az első specifikáció, az OpenCL 1.0 ugyanebben az évben, november 18-án került kiadásra. Az Open Computing Language lehetőséget ad arra, hogy heterogén platformon futtathassuk C alapú nyelven írt kerneljeinket, legyen az CPU vagy GPU. A OpenCL szabványhoz meglepő módon sok cég csatlakozott, támogatja az IBM, az Intel, sőt még az AMD és az Nvidia is. Az OpenCL egészen új szabványnak számít, így egyelőre kevés új grafikus kártya támogatja, azonban – mivel nyílt és sok cég áll mögötte – a közeljövőben gyorsan elterjedhet, megalapozva egy teljesen platformfüggetlen, hatékony, heterogén programozási felületet. Egyetlen hátránya, hogy a heterogén működés biztosítása miatt a különböző architektúráknak saját kontextust kell biztosítanunk, mely csak a feldolgozó egység számára közvetlenül elérhető memória-címtartományban lévő adatokat tartalmazhat. Ez a megkötés azt eredményezheti, hogy egyes platformoknak szükségtelenül lassú átbocsátó képességű memóriatartományokat használhatnak. Ez a probléma ugyan kiküszöbölhető, de ez költséges konzisztencia őrző és memóriakezelő módszereket von maga után. Kutatásom céljából elsősorban videokártyákra optimalizált rendszert terveztem megvalósítani, ennek érdekében egy kifejezetten GPU-kra specifikált fejlesztési felületet kellett találnom, még ha az hardver-megkötésekkel is jár.

1.3.4. Nvidia CUDA

Az utóbbi időben a két vezető grafikuskártya-gyártó cég, az AMD (korábban Ati) és az Nvidia belekezdett saját GPGPU-t támogató architektúrájának kifejlesztésébe, így jelent meg a FireStream és a CUDA (*Compute Unified Device Architecture*). A fejlesztés hatással volt a grafikus chipek új modelljeire és lehetővé tette a GPGPU elterjedését.

Bár az alkalmazásfejlesztési felületek rendkívül jól használhatóak és a gyártók támogatását élvezik, általános tény, hogy az egyik gyártó függvénykönyvtárával lefordított alkalmazás nem fut a konkurens gyártó grafikus kártyáin. Felhasználását tekintve ezek az API-k a legmegfelelőbbek az olyan algoritmusok megvalósításához, mint a sugárkövető rendszer, de a platformfüggetlenség általában csak a gyártó saját kártyáira korlátozódik. Az említett architektúrák és hozzájuk tartozó programozási felületek közül a CUDA maradt meg aktív fejlesztési státuszban.

A CUDA architektúrák jelentősége abban rejlik, hogy az előző generációk külön számított vertex és pixel shaderjeivel ellentétben ezeknél használtak először egységesített shader csöveket, lehetővé téve az aritmetikai és logikai feldolgozóegységeknek általános célú programkód (kernel) futtatását [13]. A GPU-n futtatható kernelt eddig kizárólag CUDA C illetve CUDA Fortran nyelven lehetett írni, azonban 2011 decemberében az Nvidia nyílt forráskódúvá tette a CUDA fordítóprogramját, így szinte bármilyen nyelven írhatunk GPGPU-s alkalmazásokat.

A CUDA az egyetlen fejlesztői felület, ami lehetőséget ad a videokártya valódi általános célú programozására, mindemellett lehetőséget ad a hardver sajátosságainak hatékony kihasználására is. Éppen ezért kiváló választásnak bizonyult a kutatómunkám során fejlesztett rendszer platformjául is is.

1.4. Sugárkövető algoritmusok

A sugárkövető algoritmus minden olyan térbeli objektum megjelenítésére képes, amelynek egy egyenessel való metszéspontját ki tudjuk számítani. Az inkrementális képszintézissel ellentétben itt nincs szükség a felületek tesszellációjára (poligonhálózattal való közelítésére), vagy a vágósíkkal párhuzamos szeletek leképzésére, ezért például térfiguratmodelleket is könnyen kezelhetünk. A grafikus alkalmazásokban használt modellek többsége, az inkrementális képszintézist használó technológiák elterjedésének következtében, gyakran csak tesszellált, többnyire háromszögekre bontott formában érhető el.

Munkám célja elsősorban a háromszögmodellek hatékony megjelenítése volt, azonban az algoritmus gyorsítása érdekében a tengelyekkel párhuzamos élű befoglaló dobozok (*axis-aligned minimum bounding box*) metszésproblémáját is hatékonyan meg kellett oldani.

1.4.1. Befoglaló doboz metszése sugárral

Egy befoglaló dobozzal való metszés számítása leegyszerűsíthető 6 oldallap, illetve 12 háromszög metszésének számítására is, azonban ha kihasználjuk a térrész azon tulajdonságát, hogy minden oldala merőleges egy tengelyre a folyamat tovább gyorsítható. A befoglaló doboz tesztelésén kívül, hasznos információval szolgál még a sugár belé-

pési és kilépési pontja, amit felhasználva könnyebben elindulhatunk egy térfelosztást alkalmazó modell hierarchiájában.

Munkám során a befoglaló dobozt tesztelő algoritmushoz az Eric Haines által publikált térszelet (*slab*) módszerét alkalmaztam [3]. A módszer lényege, hogy a paraméteresen leírt sugarakat (egyeneseket) a befoglaló doboz tengelyekre merőleges oldallapjainak kiterjesztésével kapott síkpárral elmeteszünk, az így kapott távolságértékek közül megkeressük a legnagyobb közeli és a legkisebb távoli metszéspont távolságát. Ha a közeli érték kisebb, mint a távoli metszéspont távolsága, illetve a távoli metszéspont pozitív, akkor a sugarat leíró félegyenes áthalad a befoglaló dobozon. A távolságadatokból könnyen kiszámíthatjuk a metszéspontot az egyenes paraméteres egyenletének segítségével.

1.4.2. Háromszög metszése sugárral

A sugárkövetés legfontosabb lépése, a térbeli háromszögek és a sugár metszéspontjának kiszámítása. A legegyszerűbb megközelítés szerint az összes sugár esetében minden primitívre (esetünkben háromszögekre) el kell végezni a metszési vizsgálatot. Egy háromszög egyenessel való metszésénél, valójában a három csúcspont által meghatározott sík metszését értjük, ami csak akkor lehetséges, ha az egyenes nem párhuzamos a síkkal. A feladat erőforrás igényes része inkább az, hogy eldöntsük, ez a metszéspont a háromszögben található-e.

A metszéspont kiszámítása jelentősen eltér a takarási vizsgálatától. A sugár legközelebb eső metszéspontjának keresésekor meg kell vizsgálnunk az összes háromszöget, és ezek közül a sugár kiindulópontjához legközelebb esőt kell kiválasztanunk. Zárt felületeknél gyorsíthatjuk az algoritmust a nem látható oldallapok elhagyásával (*back face culling*). A takarási műveletek – például az árnyék – kiszámításánál a hátsó oldallapokat nem célszerű elhagyni, azonban az algoritmust elég csak az első sikeres metszéspontvizsgálatig elvégezni.

A sugárkövető alrendszer fejlesztésekor nagy hangsúlyt fektettem a metszéspontot számító algoritmus hatékony implementálására. Fontos volt, hogy a programkód GPU-ra legyen optimalizálva, annak sajátosságait figyelembe véve. Ez a hardverkörnyezet nagyban különbözik egy általános célú feldolgozóegységtől, hiszen a video processzorok lebegőpontos aritmetikára lehettek optimalizálva. Ezért például egy egész számú osztás műveletideje a lebegőpontos megfelelőjének a többszöröse is lehet. A CUDA architektúra másik sajátossága, hogy multiprocesszoronként egy regisztertár van, így több párhuzamosan futó szál (*thread*) osztozik a rendelkezésre álló regisztereken. Fontos tehát, hogy az algoritmus a lehető legkisebb gyorsítótárat igényelje, és a szükséges regiszterek számát is érdemes csökkenteni, azok újrafelhasználásával. A célhardver másik szűk keresztmetszete a memóriaelérések száma és a szegmensek rendezettsége, ugyan-

is a szekvenciális olvasásokat a hardver könnyen optimalizálhatja, csökkentve ezzel az akár 800 órajelet is felemésztő globális memóriakésedelmet [10].

A háromszögek és a sugár metszéspontjának kiszámítását végző algoritmust a Thomas Möller és Ben Trumbore által kifejlesztett módszer mintájára írtam [9]. A módszer egyik legfontosabb jellemzője az alacsony memóriaigény, valamint a fokozatos ellenőrzések adta optimalizációs lehetőségek.

Az algoritmus a vizsgált háromszög baricentrikus koordinátarendszerébe transzformálja a metszéspontot. Az így kapott (u, v) koordináta-rendszerben könnyen eldönthetjük, hogy a metszéspont a háromszög belsejében van-e. A transzformáláshoz használt mátrix determinánsának vizsgálatával azt is könnyen eldönthetjük, hogy a sugár melyik oldalról metszi a háromszöget, így megvalósítható a hátsó lapok eldobásának módszere is.

A művelet eredményeként – a metszésvizsgálat logikai értékén túl – megkapjuk a normalizált u, v koordinátákat, amiket a normálisok interpolációjánál használhatunk, valamint a metszéspontnak a sugár origójától mért térbeli t távolságát, amennyiben a sugár irányvektora normalizált alakban szerepel. Az algoritmus futásidejét tekintve egyetlen költséges műveletet tartalmaz, ez a determináns reciprokának számítása, melyre az u, v, t paraméterek skálázásánál van szükségünk, hogy azok normalizált alakban szerepeljenek. Ezt a műveletet, csak az árnyalást megvalósító sugaraknál kell végrehajtunk, valamint akkor, ha a még normalizálatlan távolság nemnegatív előjelű.

A rendszer fejlesztésekor érdemesnek láttam, hogy ennek a módszernek két változatát is megvalósítsam. A legközelebbi metszéspont vizsgálatára használható implementációban, a determináns kiszámítása után lehetőségünk van eldobni a hátsó oldallapokat, az eredményeket pedig normalizált alakban adjuk vissza. A takarási műveletekhez használható verzióban, a nem látható lapokat nem kell eldobni, valamint az eredményeket sem kell normalizálni, hiszen ezeket nem használjuk fel az árnyalási műveletekhez.

2. fejezet

Térfelosztás

A sugárkövető algoritmus leginkább erőforrásigényes folyamata a sugarak és háromszögek metszéspontjának meghatározása. Az 1. fejezetben ismertetett megoldások közül – mint a hátsó oldallap eldobása, illetve a negatív távolság vizsgálat – mindegyik azt feltételezte, hogy az összes háromszöget elemezzük egy sugár metszéspontjainak vizsgálata során. Beláthatjuk, azonban, hogy ez a megoldás nem hatékony, nagy lapszámú poliéderek esetén az algoritmus számításigénye még grafikus kártyán futtatva is hosszú percekig eltartana.

Egy összetett háromszögmodellt átszelő sugár csupán néhány térbeli háromszöget keresztez. Mivel a nem metszett primitívek nincsenek hatással a pixel színére, ezek kiszámítása felesleges. Sajnos nem áll rendelkezésünkre olyan heurisztikus módszer, amellyel meg lehetne állapítani, hogy pontosan mely háromszögeket kell vizsgálni egy adott sugár esetében, azonban léteznek módszerek, melyek a teret felosztva kiszűrhetik a primitívek azon halmazait melyet a sugár biztos hogy nem metsz. Ezek a módszerek a lineáristól jobb, gyakran kvázi-logaritmikus futási időt biztosítanak az algoritmusunknak.

A számítás optimalizálása érdekében a modell térben elhelyezkedő objektumokra először térfelosztást kell alkalmaznunk, mely önmagában ugyan egy plusz számítási folyamatot igényel, de ezt a műveletet elég egyszer elvégeznünk. A tér felosztása után egy hierarchikus adatszerkezetet kapunk, mely fába rendezi a primitíveket. Fontos, hogy a képszintézisért felelős algoritmus képes legyen kezelni ezt a struktúrát és a hozzá tartozó meta adatokat.

A tér felosztására három módszert szoktak alkalmazni, ezeket fogom röviden ismertetni.

2.1. A Befoglaló doboz (*bounding box*) módszer

A befoglaló doboz módszernél a térrész elemeit logikailag csoportosítjuk. A logikai csoportosítást már a modell tervezésekor elvégezhetjük a CAD rendszerben, ilyenkor különböző alkotóelemek, alkatrészek szerint osztjuk fel a modellt. Ezeken a poliéder-csoportokon végezzük el később a bounding box algoritmust. Az algoritmus működési elve egyszerű, a befoglalni kívánt csúcspontok minden egyes koordinátáján minimum és maximumkeresést hajt végre. Az így kapott minimum és maximum koordináta hármasok: $(\min_x, \min_y, \min_z)$ és $(\max_x, \max_y, \max_z)$ egy-egy térbeli pontot alkotnak. Az erre a két a pontra illesztett, a három koordinátatengelyre egyenként merőleges síkok egy téglatestet metszenek ki a térből. Ez a téglatest tartalmazza az összes csúcspontot, illetve oldallapot, ezt nevezzük befoglaló doboznak. A sugárkövető algoritmus először ezekre a téglatestekre végzi el a metszésvizsgálatot. Ha a sugár nem metszi a befoglaló dobozt, akkor biztos, hogy a dobozba foglalt háromszögeket sem metszi, így ezek kiszámításától eltekinthetünk.

Természetesen a téglatestünk éleinek nem kötelező a tengelyekkel párhuzamosnak lenniük, bármilyen téglatestet, sőt egyszerű poliédert is meghatározhatunk egy pont-halmazra, de így az algoritmusok időigénye jelentősen megnőne. Sugárkövetésnél használatos még a befoglaló gömb (*bounding sphere*) módszer is, ebben az esetben a metszésvizsgálatnál, egy egyenes-pont távolság kiszámítása is elegendő.

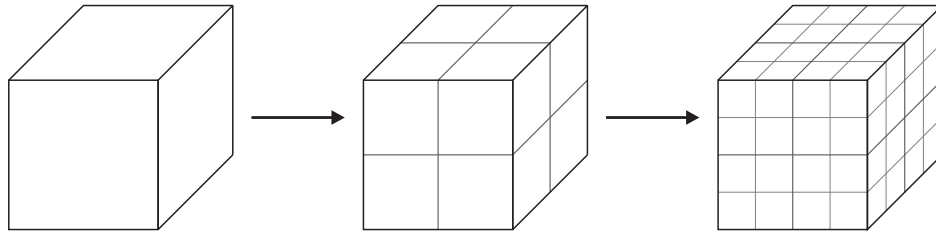
A bounding box algoritmus ugyan nagymértékben csökkenti a renderelés műveletigényét, ám a csúcspontok csoportosítását csak a modell tervezésekor végezzük el. A befoglaló dobozok alkothatnak hierarchiát, amivel az egymást befoglaló téglatestek fába rendezhetők, viszont összetettebb modellek esetén a belső befoglaló dobozok nem különíthetők el egyértelműen. A logikai csoportosítás inkább egy magasabb absztrakciós szintet képvisel, nem garantálja a hatékony metszésvizsgálatot. A fa mélysége is korlátozott, így konkáv poliéderek esetén sok kitöltetlen térrész (*empty space*) maradhat, melyeket a módszer nem kezel megfelelően.

A bounding box módszer jól használható különálló objektumok és figyelmesen megtervezett modellek esetén, azonban a logikailag különálló csoportok csupán a térfelosztási fa legfelső szintjeinél hatékonyak.

2.2. Az oktális fa térfelosztási módszer

Az oktális fa, egy olyan fa struktúra, amelynek minden köztes csomópontjából további 8 ág indulhat ki, innen kapta a nevét is; *octal tree*, röviden *octree*. Az oktális fa térfelosztási módszer használatához először el kell végeznünk a bounding box algoritmust, így megkapjuk a fa gyökerét, melyet további szintekre bonthatunk. A felosztás legegyszerűbb módja, ha egy csomópont által meghatározott téglatest minden olda-

lát a téglatestet lapjainak oldalfelező merőleges síkjaival kettévágjuk, így alakítva ki a csomópont nyolc új gyermekét, ahogy azt a 2.1. ábrán is láthatjuk.



2.1. ábra. Az *octree* felosztás első szintjén 8, a másodikon már 64 különálló részre oszthatjuk a modellteret

Természetesen nem szükséges egy adott mélységben az összes csomópontot felosztanunk további részfákra. Ha egy csomópont kevés elemet tartalmaz vagy esetleg üres, akkor hatékonyabb, ha nem végzünk el rajta több felosztást, hiszen ez további számításokat vonna maga után, nemcsak a felosztás műveletére, de a metszéspontszámítás algoritmusára nézve is.

Az üres területek levágása (*empty space cutting*) egy hatékony térfelosztási stratégia az oktális fa esetében, ilyenkor a vágósíkokat úgy választjuk meg, hogy az a lehető legtöbb üres területet vágja le a szülő térrészében található primitívek halmazából. Érdeemes meghatároznunk egy határértéket is, ami alatt nem érdemes ezt a módszert alkalmaznunk, például ha egy vágósík által levágott üres térrész kevesebb, mint a teljes 15%-a, az eljárás nem számít hatékonynak. Ebben az esetben felező, illetve egyéb heurisztikus algoritmusokkal próbálkozhatunk.

Az *octree* módszerrel ugyan jelentősen gyorsíthatjuk és skálázhatjuk a térbeli metszéspontok keresését, azonban heterogén poliédermodelleken a felosztás nem hatékony. Egy térrész felosztásakor három – különböző tengelyekre orientált – vágósíkot kell meghatároznunk, még akkor is ha a háromból kettő elhanyagolható lenne, illetve ha több, de azonosan orientált vágósík lenne az ideális. Az ilyen esetekben érdekesebb lenne a vágósíkokat egymástól függetlenné tenni, azaz egy térrészre csupán egyetlen felosztást alkalmazni.

2.3. A tér felosztása kd-fa segítségével

A k -dimenziós fa (*kd-tree*) egy térfelosztási adatstruktúra, a BSP fa (*binary space partitioning tree*) egy speciális esete. Lényegében olyan bináris fa, amelynek minden nem levél eleme egy vágósíkot határoz meg úgy, hogy a vágósík merőleges legyen a csomópontban meghatározott k dimenziós tér egyik tengelyére. Az így létrehozott két térrész lesz a köztes elem bal, illetve jobb gyermeke. A fa előállításának számos kimenete lehet, attól függően, hogy milyen sorrendben rendeljük a vágásra jelölt tengelyeket az egyes csomópontokhoz, valamint hogy a térszeleteken belül, hová helyezzük a vágósíkot.

Egy kézenfekvő megoldás lehet, hogy a vágott tengelyeket sorban, egymást felváltva rendeljük a fa különböző szintjeihez, így például egy x tengelyre merőleges vágás után mindkét gyermekére y vágást alkalmazunk, és így tovább. Ezzel a módszerrel a kd-fában minden harmadik szint egy részfat (*treelet*) alkot, amivel meghatározhatunk egy voxel-t. A voxelre kiszámított befoglaló gömbön *pixels of error* vizsgálatot alkalmazhatunk. Ez arra szolgál, hogy elkerülhessük a metszéspontszámítást az egy pixelnél kisebb háromszögekre, amely azon túl, hogy szükségtelen, még *aliasing* jelenséghez is vezet [1].

Természetesen kd-fa esetén is lehet használni olyan térfelosztási stratégiákat, mint az üres területek levágása, azonban a vágólapok helyének meghatározása mindenképpen valamilyen heurisztikus módszerrel, mediánnal vagy súlyozással kell történnie, hiszen a felezés ugyanazt az eredményt adná, mint az octree. Természetesen itt is bevezethetünk egy küszöbértéket, ami alatt nem érdemes tovább bontani egy csomópontot, ezzel növelve a fa bejárásának hatékonyságát. Belátható, hogy a metszéspontok keresésének futási ideje kd-fa esetén lineáris helyett közel logaritmikus lesz.

A köztes csomópontok nem tartalmaznak primitíveket, azok csak a levelekben vannak. Azonban előfordulhat, hogy egy vágósík metszi valamely térbeli háromszöget, ilyenkor az adott primitív mindkét levélbe belekerül. Megfontolandó, hogy a levelekben a primitívek helyett csak azok referenciáit tároljuk el, ez jelenthet némi memóriakésleltetést, viszont az algoritmus tárigénye csökken.

2.3.1. A kd-fa generálása

A fa generálása egyszeri feladat, éppen ezért fontos, hogy az eredmény a lehető leghatékonyabb működést biztosítsa a sugárkövető algoritmusnak. A vágósíkok kiválasztására és elhelyezésére több heurisztikus módszer is ismert. Az üres területek elkülönítése (*empty space cutting*) hasznos térfelosztási stratégiának bizonyulhat, ha a modell összetett és sok üres térrésszel rendelkezik. Szóba jöhet még a térrészek súlyozásos felosztása is, ami úgy igyekszik meghatározni a vágósíkok helyzetét, hogy a keletkezett térrészek lehetőleg egyező számú primitívet tartalmazzanak. Ez a módszer ugyan csökkentheti a fa végleges mélységét, de ritkábban fordulnak elő kevés elemszámú csomópontok, melyeken általában jobb továbbhaladni. Figyelembe vehetjük a vágás után elkülönített térrészek térfogatát, illetve területét is, hiszen ezek azok a szabadságfokok, amikből a sugár áthaladásának valószínűségét megkaphatjuk. A kutatásom során fejlesztett rendszer számára az Ingo Wald által publikált Felületheurisztika (*Surface Area Heuristic – SAH*) megközelítést vettem alapul [6].

A SAH módszer egy p vágósík pozíciójának és irányításának függvényében egy $C_v(p)$ költséget számol ki, ami

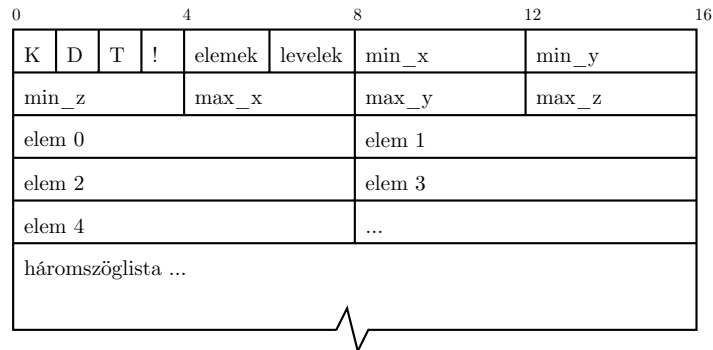
$$C_v(p) = \frac{A(V_l)}{A(V)}C(V_l) + \frac{A(V_r)}{A(V)}C(V_r)$$

alakban írható fel, ahol a vágás előtti V térrész és a két felosztott V_l, V_r térrészek ismeretében meghatározható azok A felszíne és a bennük található háromszögek C száma. Az algoritmusban egy felosztás elvégzése előtt a térrészben található háromszögek csúcspontjainak megfelelő koordinátáin haladunk végig, majd a legkevesebb költséggel járó vágást hajtjuk végre.

Ez a módszer előnyben részesíti az üres térrészek levágását, ha azok kellőképpen nagy felületű térrészre vonatkoznak.

2.3.2. A kd-fa tárolása

A modelltér primitíveinek felbontását elvégezhetjük minden esetben, egy új modell betöltésekor, ez azonban időigényes folyamat és belátható, hogy az eredmény minden modell esetében azonos. Kutatásom során ezért egy különálló segédprogramot fejlesztettem a kd-fa létrehozására, mely egy szabványos Wavefront Technologies által kifejlesztett, .OBJ kiterjesztésű térbeli alakzatokat leíró fájlformátumból [16] egy saját, az alrendszerek számára egységes, könnyen kezelhető, bináris fájlformátumot hoz létre. A sugárkövető rendszerben használt fájlformátumnak a .KDT kiterjesztést választottam.

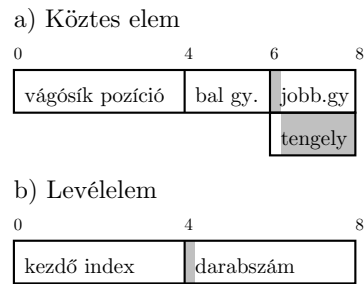


2.2. ábra. A .kdt fájl formátuma, fejléce és tartalmának elrendezése

A fájl felépítését tekintve három részre különíthető el, az első a fejléc, ezt követi a csomópontok listája, majd a háromszöglista. A fejléc első 4 bájtja a *magic number*, ez segíti a fájl egyértelmű azonosítását, valamint ez különbözteti meg az egyéb, hasonló kiterjesztésű fájlformátumoktól. A következő 4 bájtban két 16 bites egész számot tárolunk, ezek jelzik a fa összes elemének és levélelemének számát. A fa-hierarchia gyökerelem, egy befoglaló doboz, ebből indul ki a felosztás. A 8. bájtól kezdődően ezt a befoglaló dobozt tároljuk a 2.2. ábrán látható sorrendben.

A csomópontok listája szabályszerűen a 32 bájt eltolással (*offset*) kezdődik. A fa listában való tárolására szélességi keresést (*breadth-first*) alkalmaztam. Minden elemet 8 bájtban tárolunk, így a fejlécben jelezett összes elem számából megállapítható a teljes

lista mérete is. A lista egyaránt tartalmaz köztes és levélelemeket is, ezek mérete ugyan megegyezik, de felépítésük jelentősen eltérő, ahogy az a 2.3. ábrán is látható.



2.3. ábra. A .kdt fájlban tárolt fa elemeinek két típusa

A köztes elemek egy vágósíkot határoznak meg. Az első 4 bájtban a vágósík pozícióját tároljuk egy egyszeres pontosságú lebegőpontos számmal, a második 4 bájtban pedig két 16 bites egész szám jelöli a bal, illetve jobb gyerek indexét. A levélemben két 32 bites egész számot tárolunk, az első szám a levélhez tartozó háromszöglista kezdő indexe, a második pedig azt adja meg, hogy hány darab primitívet tartalmaz a levél.

A két típus megkülönböztetésére egy bitmintát (*pattern*) alkalmaztam, ennek tárolására a 8 bájtos adatstruktúra második felének legnagyobb helyiértékű két bitje szolgál. Ez a minta 4 különböző értéket vehet fel. Ha ez az érték 0, akkor levélelemről van szó, az 1–3 értékek pedig köztes elemeket jeleznek, attól függően hogy x , y , vagy z tengelyen vágunk. A levélelem esetében ez a bitminta nem játszik szerepet a feldolgozásnál, csupán a darabszámot jelölő egész típus maximális értéke csökken 2^{30} -ra, ez azonban nem jelent gondot a rendszer számára. A köztes elemeknél ezeket a biteket azonban le kell maszkolnunk, mivel e nélkül helytelen értékeket kapnánk.

0	4	8	12
0. pozíció x	0. pozíció y	0. pozíció z	
1. pozíció x	1. pozíció y	1. pozíció z	
2. pozíció x	2. pozíció y	2. pozíció z	
0. normális x	0. normális y	0. normális z	
1. normális x	1. normális y	1. normális z	
2. normális x	2. normális y	2. normális z	
0. textúra u	0. textúra v	1. textúra u	
1. textúra v	2. textúra u	2. textúra v	

2.4. ábra. A .kdt fájlban használt, háromszöget leíró adatszerkezet

A háromszögeket tartalmazó lista a csúcspontok listája után következik. A háromszöghöz tároljuk a három csúcspontját, a csúcspontbeli normálisokat, és a hozzájuk rendelt textúra-koordinátákat. Minden háromszöget 96 bájtban írunk le, függetlenül attól, hogy rendelkezik-e textúra-koordinátákkal, vagy ismerjük-e az csúcspontok normálisát. A háromszög adatstruktúrájának felépítését a a 2.4. ábra szemlélteti.

A .KDT fájlformátum egyik legnagyobb előnye, hogy a benne tárolt bináris adatokat átalakítás nélkül, egyszerűen felmásolhatjuk a videokártya memóriájába. A fájl beolvasása sem igényel nagy erőforrást, hiszen a fájl már tartalmazza a tér felosztását meghatározó kd-fát, ezért a szolgáltatást végző alrendszer akár futás közben is betöltheti a fájlokat.

2.3.3. A kd-fa bejárása

A modelltér felosztásának módszere csak akkor növelheti a képszintézis átbocsájtó képességét, ha a fát bejáró algoritmus elég hatékony. A megvalósítást tovább nehezítette, hogy ezt az algoritmust a videokártya feldolgozó egységén kellett futtatnom, ami számos technikai problémához vezetett. A fa bejárása során gyakran előfordul, hogy egy köztes elem vizsgálata során mindkét gyermeket meg kell látogatnunk. Intel architektúrákon ilyenkor a fa bejárását végző függvényben csupán egy elem vizsgálatát kell elvégezni, majd annak végén az egyik, vagy mindkét gyermekre rekurzív függvényhívást kell alkalmaznunk. A CUDA architektúrák is támogatják a rekurzív függvényhívást a 2.0-ás számítási képességtől kezdődően (*compute capability*), ez azonban indokolatlanul növeli a sugárkövető rendszer hardverkövetelményét, tehát leszűkíti a támogatott platformok számát. A rekurzív függvényhívás programozástechnikailag másképpen is megvalósítható, ebben az esetben nem a feldolgozó egység menti a verembe (*stack*) a regisztereket, hanem a programozónak magának kell gondoskodnia a szükséges regiszterek tartalmának megőrzéséről. Ez azon túl, hogy lehetővé teszi a korábbi számítási képességű videokártyák használatát, további optimalizációs lehetőséget biztosít, hiszen bizonyos részeredményeket újra felhasználhatunk.

A dolgozatomban használt fa bejáró algoritmust Vlastimil Havran et al. publikációjában ismertetett módszer alapján fejlesztettem [4]. Az említett módszer egyik jó tulajdonsága az implicit hibakezelés. A speciális eseteket – mint mikor a sugár kiindulópontja a vágósíkon helyezkedik el, vagy iránya azzal párhuzamos – helyesen oldja meg, és ehhez nincs szükség külön feltételvizsgálatra. A korábbi algoritmusoktól eltérően ez a módszer nem a metszéspontok előjeles távolságát vizsgálja, hanem azok koordinátáit, attól függően, hogy a vágás melyik tengelyre vonatkozik. A belépési és kilépési pontok koordinátái a befoglaló doboz tesztelése során előállnak, a vágósíkok pozíciói pedig a fában vannak tárolva. Egy köztes elem vágósíkjának ismert pozíciója mellé, csak akkor szükséges kiszámolnunk a döféspont további két koordinátáját, ha mindkét gyerek elemét vizsgálnunk kell a fa bejárása során. Több gyerek vizsgálata esetén mindig azon az elemen folytatjuk tovább a fa bejárását, ahol a belépési pont volt, ezzel garantálhatjuk, hogy a levélelemek térrészeit a sugár kiindulópontjától mért távolság tekintetében sorban vizsgáljuk. Ez teszi lehetővé, hogy találat esetén, a legközelebbi metszéspont keresését, csupán a vizsgált levélelem háromszögeire kell elvégezni, az így

kapott metszéspont a teljes modell legközelebbi metszéspontja is egyben. A gyakorlatban ez a módszer jelentősen megnövelte a fa bejárásának sebességét, ideális esetben az algoritmusnak csupán egy levélelemet kell vizsgálnia.

3. fejezet

Megjelenítés

3.1. Színek és sugarak

Egy szín meghatározásához a számítástechnikában leggyakrabban három színkomponenst adunk meg: vöröset, zöldet és kéket. Ezt a szín-előállítási módot szokás RGB színrendszernek nevezni. A modern grafikus kártyák is ezt a színrendszert támogatják, komponensenként 8 bitet tárolva. Egy pixel színének megadásához tehát összesen 24 bitre van szükségünk, azonban a buszrendszer miatt érdekesebb 32 biten tárolni ezt az információt. A fennmaradó 8 biten általában az átlátszóságot (*alfa*) adjuk meg, ezzel szabályozhatjuk, hogy egy pixel kitöltésénél az új pixel színe mennyire fedje el a pixel régi színét. A sugárkövetéses képábrázolásban az alfának nem tulajdonítunk olyan nagy jelentőséget a végleges pixel esetében, mivel az átlátszóság vizsgálata a sugár által metszett felületek kiértékelésekor történik.

A sugárkövetés lehetőséget ad a valóságos optikai kölcsönhatások modellezésére is. A nézőpontba jutó fény leghitelesebb ábrázolását úgy érhetjük el, ha a fizikai tulajdonságait vesszük figyelembe. A fény elektromágneses sugárzás, melynek négy legfontosabb tulajdonsága az intenzitása, hullámhossza (frekvenciája), polarizációja és fázisa. Az emberi szemben található fotoreceptorok érzékelik a fény intenzitását és hullámhosszát a 380-750 nm-ig terjedő tartományban (látható fény), azonban a polarizációját és fázisát közvetlenül nem [2]. A fény fázisától és tömegétől nyugodtan eltekinthetünk, a polarizációnak azonban jelentős hatása lehet a képre, tükröződő felületek esetén. A fény fizikai tulajdonságainak kezelése és tárolása nehézkes lehet az általános grafikus kártyákon, azonban valószerűbb képek előállítását teszi lehetővé.

3.2. Nagy dinamikatartományú fény

A grafikus rendszerekben, mint például az OpenGL-ben is, a színeket egy háromdimenziós tér pontjaiként fogjuk fel. Minden színt egy számhármassal azonosítunk, melyek általában a $[0, 1]$ intervallumba eső valós számok [7]. Megvilágítás esetén előfordulhat, hogy a színtkomponensek értékei ezen intervallumon kívülre esnek. Ezeket az OpenGL csonkolja (*clamp*), mielőtt a raszteres megjelenítőkön használt diszkrét 8 bites egészekké alakítaná. Az így kapott kép ezért alul-, illetve túlexponált lehet.

A nagy dinamikatartományú fény (*High dynamic range lighting*) esetében nincs ilyen csonkolás. A beérkező fény intenzitásától függően lehetőségünk van változtatni a megjeleníteni kívánt intervallum terjedelmét, így modellezve az emberi pupilla tágulását és összehúzódását a beérkező fény intenzitásának függvényében.

Videokártyával gyorsított sugárkövetés esetén a képpontok meghatározása párhuzamosan, egymástól függetlenül történik. A nézőpont dinamikatartományának kijelöléséhez ismernünk kell az összes képpont fényintenzitását. A képkocka képpontjainak teljes halmazával azonban a párhuzamosan futó szálak nem rendelkezhetnek, hiszen nem tudni milyen sorrendben fejeződnek be az egyes blokkműveletek, az atomiság megvalósításával pedig közel soros feldolgozást érünk el, ami pont ellentétes szándékainkkal.

A jelenlegi implementáció nem rendelkezik automatikus dinamikatartomány beállításal, azonban az utófeldolgozást végző shader bemenetén a színtkomponensek valós értékeként szerepelnek, így a felhasználónak lehetősége van állítani a dinamikatartomány eltolását (*shift*), és kiterjedését (*stretch*), a fent említett HDR hatást érve el ily módon. Ezzel a módszerrel szemléletesen bemutatatható például a tükröződő fénykomponens exponenciális jellege is.

3.3. Árnyalás

A rendszer árnyalási technikája Bui Toung Phong technikájára épít [11]. Ez a technika a *Gouraud* árnyalási technikától eltérően nem a csúcspontok között interpolálja a színeket egy fragmentumra, hanem a felületi normálisokat interpolálja minden pixelre, majd ezekre számítja ki a fényvisszaverési együtthatókat. Ezek az együtthatók a környezeti fény visszaverődési együtthatója *ambient reflection*, a szórt fény visszaverődési együtthatója (*diffuse reflection*) és a nézőpontba visszatükröződő fény együtthatója (*specular reflection*).

Több fényforrás esetén a fényösszetevőket összegezni kell. A nézőpontba jutó fény V értéke n fényforrás esetén

$$V = I_a * k_a + \sum_{i=0}^n (I_{di} * k_d * (N \cdot L_i) + I_{si} * k_s * (R \cdot L_i)^\alpha)$$

alakban írható fel, ahol I jelöli a fényintenzitásokat a három együtthatóra, k az anyag fényvisszaverődési együtthatóinak vektora, az α pedig az anyag ragyogási kitevője. Ezek a tulajdonságok az egész képkockára nézve fényforrásonként állandóak. A felületi pont normálisa N , a fényforrás irányvektora L és a tükröződő fény R irányvektora azonban minden képpont esetében eltér.

A jelenlegi megvalósításban a sugárkövetést végző alrendszer csupán a modell takarási és árnyalási feladatainak számítását végzi, majd az ebből kapott szórt és tükröződő fényegyütthatókat küldik el az utófeldolgozónak, ami a végleges képet állítja elő. A két fénykomponensen kívül átadásra kerül még a sugár által metszett felület u, v koordinátái is, melyek a textúrázáshoz nélkülözhetetlenek. Az egy képpontra értelmezett 4 komponensű lebegőpontos vektor, melyet röviden $DSUV$ vektornak nevezünk, a rendszer egyik sajátossága.

Fontos megjegyezni, hogy a környezeti fény együtthatója, valamint a modell anyagtulajdonsága egységes *uniformális* komponenseknek tekinthetők. Éppen ezért elég, ha az árnyalást végző sugárkövető rendszer helyett az utófeldolgozást ellátó shader számol velük.

4. fejezet

A rendszer használata

A kutatómunkám során fejlesztett szoftvercsomag tartalmazza a sugárkövető rendszert megvalósító grafikus felületű programot. Emellett a csomagban egy segédprogram is megtalálható, amivel a sugárkövető program számára állíthatunk elő bemeneti .KDT fájlokat. A sugárkövető rendszer működtetéséhez elengedhetetlen, hogy ismerjük a hozzá tartozó programok használatát. A programok használatáról ebben a fejezetben szeretnék bővebb információt nyújtani.

4.1. A kd-fát előállító segédprogram

A sugárkövetést végrehajtó grafikus felületű programhoz szükséges .KDT fájlokat a **treegen** programmal tudjuk létrehozni. Ez a program a paraméterként megadott szabványos .OBJ modellfájlból képes kd-fát előállítani, és eltárolni azt a rendszer számára betölthető fájlformátumban.

Ez a segédprogram végzi a modelltér felosztását és a csomópontok előállítását. Egy térrészre értelmezett felosztás műveletnek igen nagyszámú kimenete lehetséges, a program által megvalósított heurisztikus algoritmus ezért minden lehetőségre költséget számol, melyek közül a legkedvezőbbet választja ki és azon halad tovább. A költség számítás függ a térrészek méretétől, a bennük található háromszögek számától és a relatív bejárása költségtől is. Ezen a paraméterek helyes megválasztására nincs általános megoldás, mely minden modellre jó eredményt adna, ezért a segédprogram számos opciót kínál, melyeket parancssori argumentumként lehet megadni futtatáskor. Amennyiben nem adunk meg egyéni értékeket, úgy a program az alapbeállításokat használja.

A programot parancssorban futtatva a következő argumentumokkal hívhatjuk meg.

```
treegen input.obj  
    [-d depth] [-p population] [-b] [-t cost] [-e bias]  
    [-o output.kdt]
```

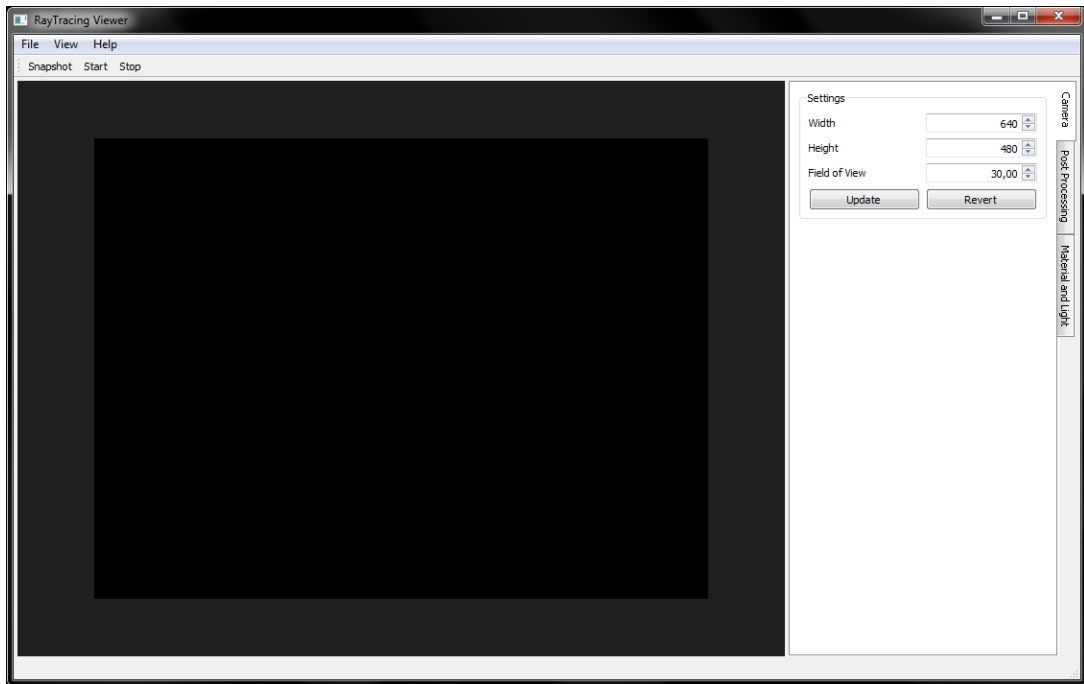
A kötelezően megadandó `input` argumentummal adjuk meg a bemeneti fájlt. A program működését befolyásoló opcionális kapcsolók rendre a következők.

- d** Ez a kapcsoló állítja be a fa maximális mélységét.
Alapértelmezés szerint ez az érték 12.
- p** Ezzel adhatjuk meg a levélelemek népességének küszöbértékét; a megadott értéknél kevesebb elemszám esetén nem osztjuk tovább a teret.
Az alapértelmezés 32.
- b** Felező síkok vizsgálata. Ezt bekapcsolva a program a csúcspontok közötti távolság felénél is végez metszésvizsgálatot.
Ez az opció alapesetben ki van kapcsolva.
- t** A fa relatív bejárási költségét állíthatjuk be ezzel a kapcsolóval. Az algoritmus nem osztja tovább a teret, ha a fa további bejárása költségesebb, mint az összes levélelem vizsgálata. Nagy költségérték esetén a fa kevesebb csomópontból áll majd.
Alapértelmezés szerint ez az érték 1.0, vagyis a fa bejárási költsége arányos egy háromszög vizsgálatának költségével.
- e** Az üres térrészek levágásakor számított költséget befolyásoló valós érték. Ezzel az értékkel szorozódik meg az üres térrészekre számolt költség értéke.
Az alapbeállítás 80%, azaz 0.8.
- o** A kimeneti fájl neve. A kapcsoló nélkül ez a bemeneti fájl névével fog meg egyezni, csupán a kiterjesztése változik.

4.2. A sugárkövető alkalmazás

A sugárkövető rendszert megvalósító `RTViewer` alkalmazáshoz grafikus felület tartozik. Ezen a felületen állíthatjuk be a sugárkövető algoritmus paramétereit, módosíthatjuk a kép megjelenítését az utófeldolgozóban, és ebben az ablakban jelenik meg a kép, mint végeredmény. Az alkalmazás főablakát a 4.1. ábrán láthatjuk.

A főablak két részre oszlik. Balra helyezkedik el az képszintézis eredményét megjelenítő OpenGL panel, jobb oldalt pedig a vezérlő panelek foglalnak helyet. A vezérlő



4.1. ábra. A sugárkövető alkalmazás főablaka

panelek három különböző kategóriára vannak bontva, ezek külön lapfülön helyezkednek el, és más-más funkciót látnak el. A vezérlő panelek lapfülönként felülről lefelé a következők:

- *Camera* - a nézeti beállításokért felelős panel;
- *Post processing* - az kép utófeldolgozását vezérlő panel;
- *Material and light* - az anyagtulajdonságot, textúrát és fényt állító panel.

4.2.1. Erőforrások betöltése

Az alkalmazás működtetéséhez be kell töltenünk egy .KDT kiterjesztésű modellfájlt. Ezt a **File** menü **Load Model** menüpontjával tehetjük meg. A felugró ablakban kiválasztott modellfájlt ezután a rendszer betölti és a videokártya memóriájába másolja a szükséges adatokat. Bemeneti fájl nélkül az alkalmazás nem működik, csak fekete képernyőt láthatunk.

Lehetőségünk van anyagmintázat (*textúra*) használatára is. Anyagmintázatot a **Material and light** lapfülön lévő **Texture** gombbal tölthetünk be. A rendszer ezután ezt a textúrát fogja alkalmazni a modell megjelenítése során, amennyiben az rendelkezik csúcspontonkénti u, v koordinátákkal. Amennyiben a modell nem tartalmaz explicit u, v koordinátpárokat, a rendszer minden csúcsponthoz a 0,0 értéket rendeli hozzá. Ez ahhoz vezet, hogy a modell mintázata homogén lesz, színe pedig megegyezik a

textúraként használt kép bal felső sarkában található pixel színével. Amennyiben ez a pixel fekete, vagy áttetsző, úgy fekete képet láthatunk a kimeneten.

4.2.2. Megjelenítés

Egy kép megjelenítését kétféleképpen kérhetjük a rendszertől. Létezik egy pillanatkép (**Snapshot**) opció, ebben az esetben egyetlen képkockát számítunk ki. A másik funkció a valós idejű renderelés elindítása (**Start**), ekkor a képkockák kiszámítása és megjelenítése folyamatos. Így – az egér segítségével mozgatva, forgatva a modellünket – azonnal látjuk a transzformáció eredményét, míg a pillanatkép esetében ehhez újból meg kell nyomnunk a gombot. A valós idejű renderelést az eszköztár **Stop** gombjával tudjuk megállítani.

Fontos, hogy folyamatos renderelés közben nincs lehetőségünk a nézeti kép (*viewport*) változtatására, ezért előtte mindenképpen meg kell állítanunk a valós idejű renderelést.

5. fejezet

Összefoglalás

A videokártyák processzorainak általános célú felhasználására irányuló kezdeményezések újra előtérbe helyezték a sugárkövető algoritmusok kutatását. Ahogy a dolgozatom eredménye is igazolja, a valósidejű sugárkövetéshez nincs szükségünk ipari felszerelésre, vagy szervertermekre. Örömmel konstatáltam, hogy egy átlagos, asztali munkaállomáson futtatva is képes a rendszerem – az ember számára már nem zavaró – magas képfrissítési frekvenciát elérni.

Az általam fejlesztett .KDT fájlformátum a kutatás kezdete óta sokat fejlődött, az azt előállító és használó programok által. Kijelenthetem, hogy a kutatás során lefedtetett specifikáció jól megállta a helyét az egész rendszer területén, és sikerült egy olyan jól skálázható és portolható bináris formátumot fejleszteni, ami önmagában is hasznosnak bizonyulhat hasonló projektek esetében is.

A rendszerben heterogén módon használom a videokártya grafikus processzorát, egyaránt futtatok CUDA platformon írt kerneleket a sugárkövetéshez, valamint GLSL-ben írt shadereket az utófeldolgozáshoz. A teljesítmény növelése érdekében osztott memóriahasználatot vezettem be, vagyis a két alrendszer között nem történik költséges memóriamásolás – még az eltérő platform ellenére sem. A hoszt oldal a vezérlésen kívül más szerepet nem tölt be – a teljes folyamatot a videokártya számolja ki és jeleníti meg. Erre a megoldásra a témával foglalkozó szakirodalomban nem találtam példát.

A grafikus processzorra, mint architektúrára való fejlesztés számos új megoldás keresését vonta maga után. Kutatásom során csak részben támaszkodhattam az irodalomban közölt – jól bevált, de elsősorban CPU-ra fejlesztett –, algoritmusokra. Olyan triviális problémákat kellett átgondolnom, mint a rekurzív függvényhívás, vagy a veremkezelés. A regiszterekkel való takarékoskodás, vagy a memória elérések egyesítése az architektúra olyan sajátossága, melyre különösképpen ügyelnem kellett. A kutatómunkám során rengeteg új ismeretre sikerült szert tettem ezen a területen.

A sugárkövetéses képkalkotás eredményének megjelenítésére sikerült egy egyedi, speciális módszert kidolgoznom. A fény együtthatónkénti kezelése és továbbítása, nemcsak a rendszer memóriaigényét csökkentette, hanem lehetőséget adott a fény nagy dinami-

katartományú megjelenítésére is. A jelenleg alkalmazott árnyalási és takarási módszerek élethű fényhatásokat biztosítanak, de a sugárkövetésben alkalmazható optikai hatások tárháza azonban ettől sokkal nagyobb. Mi értelme sugárkövetéssel foglalkozni, ha nem valósíthatunk meg olyan valóságghű optikai jelenségeket, mint a tükröződés, vagy a fénytörés? A jelenlegi implementáció azonban nem vonultathatta fel mindezeket, hiszen az elsődleges cél egy olyan rendszer kifejlesztése volt, ami képes sugárkövetéssel készült képek valósidejű előállítására.

Dolgozatom írása során számos kihívás merült fel. A kutatómunka és a fejlesztéskor felmerülő problémák megoldása során – úgy vélem – sikerült széleskörű és naprakész tudást szereznem a képszintézis tudományos világában.



5.1. ábra. Angyal (474 048 háromszög)



5.2. ábra. Armadillo (345 948 háromszög)



5.3. ábra. Suzanne a majom (15 744 háromszög)



5.4. ábra. Stanford bunny (69 451 háromszög)



5.5. ábra. Stanford dragon (871 414 háromszög)



5.6. ábra. Stanford happy buddha (290 633 háromszög)

Irodalomjegyzék

- [1] Áfra, A. T., Interactive Ray Tracing of Large Models Using Voxel Hierarchies, Computer Graphics Forum, **31** (1), 75–88, 2012., doi:10.1111/j.1467-8659.2011.02085.x
- [2] Czap, L., Képfeldolgozás, Miskolci Egyetem, Miskolc, 2007.
- [3] Haines, E., Essential Ray Tracing Algorithms, in Glassner, A. (edt.) An Introduction to Ray Tracing, Chapter 2, 33–77, Academic Press, 1989.
- [4] Havran, V., Kopal, T., Bittner, J., and Žára, J., Fast robust BSP tree traversal algorithm for ray tracing, J. Graph. Tools, **2** (4), 15–23, 1998.
- [5] Ingo W., Philipp S., Carsten B., Interactive Distributed Ray Tracing of Highly Complex Models, In Rendering Techniques 2001: 12th Eurographics Workshop on Rendering, 277–288, 2001.
- [6] Ingo, W., Havran, V., On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$, in Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 61–69, 2006.
- [7] Juhász, I., Lajos, S., Számítógépi grafika, Miskolc, 2007.
http://193.6.8.43/segedlet/dokumentumok/TISZK/Szamitogepi_grafika.php
- [8] László J., A VGA-kártya programozása Pascal és Assembly nyelven, Computer-Books, Budapest, 1995.
- [9] Möller, T., Trumbore, B., Fast, minimum storage ray/triangle intersection, in SIGGRAPH '05 ACM SIGGRAPH 2005 Courses, Art. no. 7, ACM, New York, 2005.
- [10] NVIDIA CUDA C Programming Guide version 4.0, NVIDIA Corporation, 2011.
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- [11] Phong B. T., Illumination for computer generated pictures, Communications of ACM, **18** (6), 311–317, 1975.
doi:10.1145/360825.360839

- [12] Purcell, Timothy J., Buck, I., Mark, William R., Hanrahan, P., Ray Tracing on Programmable Graphics Hardware, SIGGRAPH '02 ACM Trans. Graph. **21** (10), 703–712, 2002.
- [13] Sanders, J., Kandrot, E., CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.
- [14] Szirmay-Kalos, L., Aszódi, B., Lazányi I., Premecz M., Approximate Ray-Tracing on the GPU with Distance Impostors, 2005. sirkan.iit.bme.hu/~szirmay/ibl3.pdf
- [15] Szirmay-Kalos, L., Antal, Gy., Csonka, F., Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks, Budapest, 2007.
- [16] Wavefront OBJ Specification
<http://www.martinreddy.net/gfx/3d/OBJ.spec>

Ábrák jegyzéke

1.1. Az Intel Core2 Duo és az Nvidia Fermi processzormagjának röntgenképe	2
2.1. Az <i>octree</i> felosztás első szintjén 8, a másodikon már 64 különálló részre oszthatjuk a modellteret	12
2.2. A .kdt fájl formátuma, fejléce és tartalmának elrendezése	14
2.3. A .kdt fájlban tárolt fa elemeinek két típusa	15
2.4. A .kdt fájlban használt, háromszöget leíró adatszerkezet	15
4.1. A sugárkövető alkalmazás főablaka	23
5.1. Angyal (474 048 háromszög)	27
5.2. Armadillo (345 948 háromszög)	27
5.3. Suzanne a majom (15 744 háromszög)	28
5.4. Stanford bunny (69 451 háromszög)	28
5.5. Stanford dragon (871 414 háromszög)	29
5.6. Stanford happy buddha (290 633 háromszög)	29