XII. Erdélyi Tudományos Diákköri Konferencia

BONYOLULT, STATIKUS 3D-S MODELLEK VALÓS IDEJŰ MEGJELENÍTÉSE SUGÁRKÖVETÉSSEL

Szerző:
Áfra Attila Tamás

Babeş-Bolyai Tudományegyetem
Matematika és Informatika Kar
Informatikai Modellek Optimalizálása
1. évfolyam (mesteri képzés)

Témavezető:
Dr. Csató Lehel

Babeş-Bolyai Tudományegyetem
Matematika és Informatika Kar
Programozási Nyelvek és Módszerek Tanszék

Kolozsvár, 2009. május 15–17.

Tartalomjegyzék

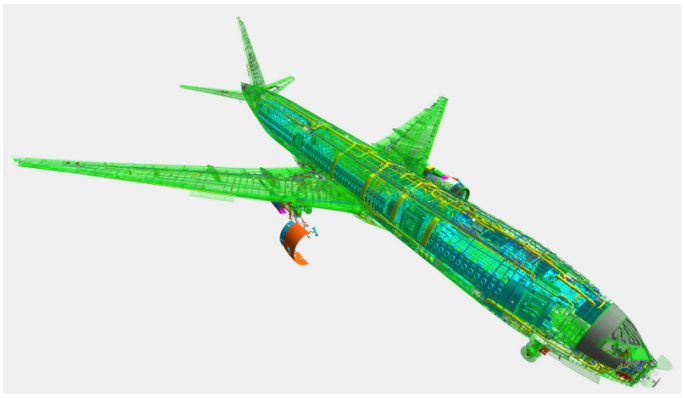
1. fejezet

Bevezető

A nagyon bonyolult 3D-s modellek (pl. többszáz millió háromszögből álló CAD és scannelt modellek: 1.1. ábra) valós idejű megjelenítése a hatalmas számítás- és memóriaigény miatt roppant nehéz feladat.

A dolgozat egy olyan új, speciális módszert vezet be, amely lehetővé teszi az ilyen modellek akadozásmentes bejárását asztali és hordozható számítógépeken is. A renderelés hibrid háromszög/voxel *sugárkövetéssel* (*ray casting*) történik, ami hatékonyan implementálható többmagos processzorokon.

A voxelalapú *level of detail* (LOD) mechanizmusnak és a testreszabott virtuális memóriakezelési rendszernek köszönhetően a teljes modellnek csak egy töredéke kell legyen betöltve a memóriába. A részletek beolvasása a háttértárolóról teljesen aszinkron, így a modell bejárása akkor is folyamatos, amikor a memóriában lévő adatok még nem elegendőek a kért részletességi szint eléréséhez.



(a) Boeing 777 (350 millió háromszög)

(b) St. Matthew (372 millió háromszög)

1.1. ábra. Bonyolult modellek

2. fejezet

Áttekintés

2.1. Sugárkövetés

A *sugárkövetés* (*ray casting*) olyan képalkotási (*renderelési*) módszer, amely segítségével úgy számoljuk ki a kép pixeleit, hogy a kamera pozíciójából, a pixeleken keresztül sugarakat lövünk ki, és megkeressük minden egyes sugár legközelebbi metszéspontját a megjelenítendő modell geometriájával. Miután megkaptuk a metszéspontot, kiszámoljuk az onnan kiinduló és a kamerába érkező fénysugár színét. Ezt *árnyalásnak* (*shading*) nevezzük.

Ha árnyalás során tovább követjük a fénysugár útját, akkor *rekurzív sugárkövetésről* (*ray tracing*) beszélünk.

2.2. Kd-fa

Egy modell primitívek sokaságából áll. A módszerünket háromszögekből álló modellekre terveztük, de könnyedén kibővíthető más primitívekre is (pl. pontok).

Ahhoz, hogy a metszéspont meghatározása során a sugarat ne kelljen tesztelni minden egyes primitívvel, szükségünk van egy hierarchikus adatszerkezetre. A *kd-fa* (*kd-tree*, azaz *k-dimensional tree*) mellett döntöttünk, amely nagyon egyszerű és hatékony [Hav00]. Általában statikus modellekre érdemes alkalmazni, mivel egy jó minőségű kd-fa felépítése nagyon számításigényes.

A kd-fa a *BSP fa* (*binary space partitioning tree*) egy speciális esete. Minden csomópont meghatároz egy *cellát* (térben egy téglatest alakú rész), amelyet egy felezősíkkal két kisebb cellára bont. A felezősík merőleges kell legyen valamelyik koordinátatengelyre, pozíciója viszont tetszőleges lehet. A levelek nem bontják tovább az illető cellát, viszont tartalmazzák az abban a cellában lévő primitíveket. Primitívek tehát csak levelekben vannak.

Fontos megjegyezni, hogy egy primitív több levélben is benne lehet, mivel több cellába is belelóghat. Ez akkor fordulhat elő, ha egy felezősík metszi valamelyik primitívet. A hatékony helykihasználást szem előtt tartva egy háromszöget nem tárolunk el többször, fölöslegesen, hanem csak egyszer, a faszerkezeten kívül, a levelekbe pedig csak egy-egy *primitív ID-t* (azonosítót) teszünk.

A metszéspont keresése (*sugárbejárás*, *ray traversal*) során a kd-fán mindig mélységi bejárást alkalmazunk. Ezt figyelembe véve speciális optimizálásokat lehet kivitelezni.

A kd-fát felhasználva a legközelebbi metszéspontot lineáris helyett logaritmikus időben lehet megtalálni. Ez egy rendkívül fontos tulajdonság, ami miatt a sugárkövetés kiemelkedően jó nagyon sok primitívből álló modellek megjelenítésére.

2.3. *Level of detail* (LOD)

A modell távoli részleteit fölösleges teljes részletességben megjeleníteni, hiszen az elérhető effektív részletességet korlátozza a renderelési felbontás. Bonyolult modelleknél előfordulhat, hogy egy pixelt több háromszög is fed, így azon kívül, hogy a szükségesnél többet számolunk, a minőség is leromlik az *aliasing-effektus* miatt [YLM06]. A fölösleges részletek természetesen be kell legyenek töltve a memóriába. Hatalmas modelleknél ez súlyos problémát okozhat, mert megtörténhet, hogy bizonyos kameraállások esetében (pl. távoli nézet) nem fér be a memóriába az összes szükséges háromszög.

A felsorolt problémák mindegyikét meg lehet oldani egy *level of detail* (részletességi szint) mechanizmus alkalmazásával. Ez abból áll, hogy a modellt eltároljuk különböző részletességi szinteknek megfelelően, renderelés közben pedig csak a szükséges szintek részeit töltjük be és használjuk fel. Ezzel megnöveljük a modell méretét a háttértárolón, viszont jelentősen felgyorsítjuk a renderelést, és drasztikusan csökkentjük a memóriaigényt.

A háromszöghálók egyszerűsítése igen nehéz feladat, mivel a topológia súlyosan meg tudja nehezíteni a kívánt szintre való leegyszerűsítést. Egy másik gond az, hogy renderelés közben a szintek közötti választás és átmenet nem lehet elég finom, mivel ezzel a módszerrel csak nagyobb darabokat lehet egyszerűsíteni.

A tradicionális megközelítés helyett *hierarchikus voxelalapú LOD* mechanizmust használunk, melynek kivitelezése sokkal egyszerűbb, ugyanakkor hatékonyan és elegánsan beépíthető a már gyorsításra használt kd-fába. A kd-fa tehát kettős szerepet játszik. Ennek viszont egyik következménye az, hogy a felépítésekor több szempontot kell szem előtt tartani, hiszen egyszerre két eltérő célra kell megfelelő legyen.

A kd-fa minden csomópontjához opcionálisan tartozhat egy *voxel*, amely az illető csomópont celláját tölti ki, és a cellában levő geometriát approximálja. A voxelek ugyanazokkal az árnyalási paraméterekkel rendelkeznek, mint az eredeti primitívek (pl. normálvektor, diffúz fényvisszaverés stb.). Ha renderelés kor egy voxel a képsíkra vetítve nem fed több mint egy pixelt (kisebb részletesség esetén több pixelt), akkor a csomóponton belüli geometriát helyettesíthetjük a voxellel.

2.4. Virtuális memóriakezelés

Ahhoz, hogy a rendelkezésre álló memóriánál nagyobb méretű modellt is meg tudjunk jeleníteni, virtuális memóriakezelést kell használjunk. A legegyszerűbb megoldás az, hogy az

operációs rendszer *memory mapping* szolgáltatását alkalmazzuk, amely nagyon gyors, mivel a címfordítás (*address translation*) hardveresen történik. Az ilyen virtuális memóriakezelés láthatatlan az alkalmazás számára, így nincs szükség különösebb módosításokra.

A *memory mapping* egyik hátránya, hogy laphiba (*page fault*) esetén a memóriahozzáférés addig blokkol, míg a lap be nem töltődik a memóriába. Renderelésnél ez úgy nyilvánul meg, hogy amíg nincs betöltve az összes szükséges részlet, a megjelenítés szünetel, tehát akadozik.

A módszerünkben *memory mapping* helyett egy saját, testreszabott, szoftveres virtuális memóriakezelési rendszert alkalmazunk, mely az említett problémákat sikeresen kiküszöböli. Az adatszerkezetünket – a gyorsításra és LOD-ra használt kd-fát – ehhez a rendszerhez adaptáltuk, aminek következtében a megjelenítés teljesítménye jelentősen jobb, mint egy általános, operációs rendszer szintű megoldásé.

A részleteket a memóriába teljesen aszinkron töltjuk be, tehát egy memóriahozzáférés sosem blokkol. Amennyiben a kért memóriaterülethez nem lehet hozzáférni, a memóriában lévő LOD-ot felhasználva a modellt az elvártnál kisebb részletességgel jelenítjük meg. Az optimális teljesítmény érdekében a memóriakezelő renderelés közben számon tartja a laphibák gyakoriságát, ami alapján a lapoknak betöltési prioritást feleltet meg.

A címfordítást szoftveresen oldjuk meg, amely relatív lassú és redundáns is, viszont a memóriahozzáférések sugárkövetés során egy jól meghatározott, szigorú mintát követnek, amire optimizálni tudjuk a címfordítási mechanizmust. Így a rendundáns címfordítás gyakorlatilag nem lassítja a renderelést, rendkívül rugalmas, ráadásul teljesen független az operációs rendszertől és a processzor architektúrájától (pl. nem szükséges 64 bites processzor és operációs rendszer).

3. fejezet

Adatszerkezet

A renderelő által használt adatszerkezet két fő részből áll: a voxeleket tartalmazó kd-fából és a háromszögekből. Ezt az adatszerkezetet egy file-ban tároljuk, és egy különálló programmal állítjuk elő az eredeti modellből.

3.1. Blokkok

A file-t a virtuális memóriakezelő rendszerünk miatt egyenlő méretű *blokkokra* osztjuk. Azért nem a *lap* elnevezést haszáljuk, mert egy blokk nem kell feltétlenül megfeleljen egy operációs rendszer szintű lapnak. Ennek ellenére a blokkméretet a hardveres virtuális memóriakezelésnél széles körben elterjedt 4 KB-nak választottuk meg.

A blokkokat 32 bites *blokk ID-kal* azonosítjuk. Ezzel a maximális file-méretet illetve címterületet 16 TB-ra korlátozzuk, amely gyakorlatban általában több mint elegendő. A legnagyobb ábrázolható ID-t érvénytelen blokk ID-nak tekintjük.

A file először a kd-fa blokkjait tartalmazza, utána pedig a háromszögekét.

3.2. *Treeletek*

A kd-fa egy hatalmas bináris fa, melynek memóriában való elrendezése kritikus a teljesítmény szempontjából. Ismerjük a memóriakezelő pontos működését, így egy speciális *cache-aware* (gyorsítótárról tudomással bíró) elrendezést alkalmazunk.

A kd-fát feldaraboljuk nagyon sok, kicsi alfára, melyeket *treeleteknek* nevezünk. Egy treeletet megpróbulunk úgy felépíteni, hogy teljesen kitöltsön egy blokkot. Ha ez nem lehetséges (a treelet túl kicsi), akkor igyekszünk egy blokkba több kisebb treeletet is eltárolni. A blokkokban fennmaradó üres területeket nem lehet másra felhasználni, ezért ezeknek mennyiségét ajánlatos minimalizálni.

3.3. Csomópontok

A kd-fában két fajta csomópont lehet: belső csomópont (*inner node*) és levélcsomópont (*leaf node*). Egy belső csomóponthoz egy felezősík, mutatók a két gyerekcsomópontra és opcionálisan egy voxel tartozik. Egy levélcsomópont csupán egy *háromszög ID listával* rendelkezik.

A csomópontokat mindig mélységi bejárással érintjük, ezért egy szabályosabb memóriahozzáférési minta érdekében a gyerekcsomópontokat mindig a szülőcsomópont után tároljuk el.

A csomópontmutatók elég nagyok kell legyenek ahhoz, hogy le lehessen velük fedni a teljes címzési tartományt, ami jelen esetben 16 TB. A legtömörebb kd-fa reprezentációk esetében egy csomópont mérete 8 byte, viszont a lefedhető címterület mérete csupán 4 GB, mely átlagos méretű modellekhez elegendő, nagyon nagyokhoz viszont már nem.

Ha ezt a tömör reprezentációt kiterjesztjük 16 TB-os címterületre, akkor a belső csomópontok mérete 16 byte-ossá válik. Tehát a kd-fa mérete jelentősen megnő az eredeti reprezentációhoz képest. Ez a tárolásra, memóriahasználatra és renderelési sebességre (többek között azért, mert a processzor gyorsítótárába fele annyi csomópont fér be) egyaránt erősen negatív hatást fejt ki.

A módszerünkhöz egy ennél sokkal hatékonyabb reprezentációt dolgoztunk ki: kihasználjuk azt, hogy a kd-fát blokkokba csomagolt treeletekre bontjuk. A gyerekmutatók döntő többsége egy olyan csomópontra hivatkozik, amely a szülővel megegyező treeletben és blokkban található. Tehát az ilyen gyerekbe való átmenet nem fog maga után vonni egy másik blokkba való átmenetet.

Ennek következménye kettős. Elsősorban gyerekmutatónak megfelel egy kisméretű, a szülő címéhez viszonyított pozitív offset (*near pointer*, azaz *közeli mutató*), mely nem haladja meg a blokk méretét. Másodsorban az ilyen csomópontátmenet nem igényel címfordítást (ami szoftveresen nagyon költséges), hiszen továbbra is ugyanabban a blokkban maradunk. A belső csomópontok ilyen formában csupán 8 byte-osak, és még marad szabad hely további adatok számára (pl. LOD).

A más blokkban található gyerekcsomópontokra ezzel a módszerrel természetesen nem lehet hivatkozni, ezért bevezetünk egy harmadik, speciális csomóponttípust: a *kapocs-csomópontot* (*link node*). Ez egy olyan mutatót tartalmaz, amely le tudja fedni a teljes címtartományt (*far pointer*, azaz *távoli mutató*), tehát képes bármilyen csomópontra hivatkozni. A szülő a közeli mutatóval nem címezhető gyereke helyett egy kapocs-csomópontra hivatkozik, ami a treeletbe be van építve, s így címezhető.

Ha a fa bejárása során találkozunk egy kapocs-csomóponttal, a célcsomópont címét a távoli mutatót felhasználva címfordítással kiszámoljuk. Címfordítást más alkalommal nem kell végezni, tehát nem kell külön tesztelni a szükségességét.

Egy treelet csomópontjait *DFS* (*depth-first search*) elrendezés szerint tároljuk. Ez azt jelenti, hogy egy bal gyerekcsomópont közvetlenül a szülőcsomópont után kell következzen, a jobb gyerek pedig nem előzheti meg azt (ezt már korábban kikötöttük). A fa mélységi bejárása során nagy a cache-találatok száma, mivel a következő csomópont átlagosan 50% körüli eséllyel

található ugyanabban a cache-vonalban, mint az aktuális.

Teljesítmény és tárolás szempontjából nagyon fontos a csomópontok hatékony kódolása, ezért az adatokat összecsomagoljuk, azaz csak annyi biten ábrázoljuk, amennyi szükséges. A csomagolt adatokat 32 bites egészekbe csoportosítjuk, és úgy rendezzük őket, hogy a kicsomagolás minimális mennyiségű bitműveletből (AND és SHIFT) álljon.

A hatékony cache-használat érdekében mutatók segítségével szétválasztjuk a fa bejárása során gyakran érintett adatokat (*hot data*) a ritkán érintettektől (*cold data*) [Eri04]. Ezt a módszert *hot/cold data separation-nek* nevezzük.

3.3.1. Belső csomópontok

A belső csomópontok mérete 8 byte. A felezési tengelyt (az a tengely, amelyre a felezési sík merőleges) 2 biten kódoljuk, a felezési pozíciót pedig 32 biten (lebegőpontosan).

Csak a jobb gyerekcsomópont offsetjét kell eltárolni, melynek elegendő 10 bit. Normális esetben szükség lenne 12 bitre (a blokkméret $2^{12} = 4096$ byte), viszont minden csomópont 4 byte-ra van igazítva, tehát az offset alsó 2 bitje mindig 0.

A kd-fában általában nagyon sok üres csomópont (primitíveket nem tartalmazó levélcsomópont) van. Ezeket fölösleges eltárolni, ezért a belső csomópontokban kódoljuk azt is, hogy egy gyerek üres-e vagy sem. Ha a jobb gyerek üres, akkor offsetje 0-vál egyenlő. Ez az offset más esetben nem használt, mivel önmagára mutat. A bal gyerek offsetjét nem tároljuk el, ezért szükségünk van egy jelzőbitre.

Egy belső csomóponthoz opcionálisan tartozhat egy voxel, melynek memóriabeli pozícióját – a gyerekcsomópontokhoz hasonlóan – egy 10 bites offset segítségével ábrázoljuk. Amennyiben a csomópont nem rendelkezik voxellel, az offset értéke 0.

Renderelés során a LOD mechanizumusnak szüksége van a voxel köré írható gömb sugarára (a voxel méretének egy approximációja), melyet ki lehetne számolni menet közben is, viszont az túl költséges lenne. A megoldás tehát az, hogy ezt az értéket előre kiszámoljuk, és eltároljuk a belső csomópontokban. Azonban még egy 32 bites lebegőpontos érték már nem fér el 8 byte-ban, és valójában nincs is szükségünk ekkora pontosságra.

Ismét kihasználjuk azt, hogy mélységi bejárást kell végrehajtanunk a fán: nem magát a sugarat tároljuk el, hanem az aktuális és a legközelebbi voxellel rendelkező ős sugarának arányát. Az ős sugara nem lehet kisebb mint az aktuálisé, tehat az arány csak 0 és 1 között vehet fel értéket. Ezt 8 biten kvantálva tároljuk el, viszont gyakran hozzáfért adat léven nem a voxelhez, hanem a csomóponthoz kötjuk.

3.3.2. Levélcsomópontok

Egy levélcsomópontot csupán 4 byte-on ábrázolunk. Tartalmazza a háromszögek számát és egy offsetet az ID listára.

A lista nem más, mint háromszög ID-k sorozata. Az implementációnk 32 bites ID-kkal dolgozik, lekorlátozva a maximálisan kezelhető háromszögek számát kb. 4 milliárdra. Szükség

esetén az ID-k könnyedén szélesíthetőek.

3.3.3. Kapocs-csomópontok

A kapocs-csomópontok 8 byte-osak, és csak egy távoli mutatót tartalmaznak. Közönséges offset helyett egy blokk ID-t és egy blokkon belüli offsetet kódolunk. Ezzel le tudjuk egyszerűsíteni és fel tudjuk gyorsítani a virtuális memóriakezelő működését.

3.4. Voxelek

A voxeleknek tartalmazniuk kell az árnyaláshoz szükséges összes paramétert. A két legalapvetőbb ilyen paraméter a normálvektor és a diffúz visszaverődés.

A normálvektort 30 bites kvantált formában (egy vektorkomponens 10 bites) ábrázoljuk, a diffúz visszaverődést pedig standard 24 bites RGB alakban. Összevetve tehát egy voxel mérete 8 byte.

3.5. Háromszögek

A háromszögeket olyan speciális formában tároljuk el, amely nagyon gyors sugárral való metszést tesz lehetővé. A geometriai adatokon kívül diffúz visszaverődést is elkódolunk.

Az implementációnk háromszögstruktúrája 48 byte-ot foglal.

4. fejezet

Előfeldolgozás

Az előfeldolgozási fázisban a megjelenítendő nyers modellt átalakítjuk az előzőleg bemutatott optimizált reprezentációba. A rendereléshez hasonlóan az előfeldolgozás is kell működjon olyan modellekre, melyek nem férnek be a memóriába.

4.1. A kd-fa felépítése

A kd-fa minősége kritikus a renderelési teljesítményre nézve. Elsősorban egy megfelelő voxelhierarchiát kell magába foglaljon, másodsorban pedig fel kell gyorsítsa a legközelebbi metszéspont keresését. Emiatt nem használhatunk módosítatlan tradicionális kd-fa építési algoritmust, és sok esetben kénytelenek vagyunk kompromisszumokat kötni.

4.1.1. Üres tér levágása

A kd-fa egy csomópontja általában két részre osztja a benne levő primitíveket. Előfordulhat, hogy a cellák többszörös kettéosztása után nagyobb üres zonák alakulnak ki. A sugárkövetés teljesítményét jelentősen meg tudjuk növelni, ha ezeket minél hamarabb levágjuk (*empty space cutting*). Ezt olyan csomópontokkal tudjuk megvalósítani, melyek egyik gyereke üres, a másik pedig változatlanul tartalmazza a csomópont összes primitívjét.

Az üres tér levágása a mi esetünkben a szokásosnál is sokkal fontosabb, mivel a csomópontokhoz voxelek is tartozhatnak, a voxelek méretét pedig az illető csomópont cellája határozza meg. Ahhoz, hogy a voxelhierarchia pontosan tudja approximálni a geometriát, az átlagosnál aggresszívebb levágási stratégiát kell alkalmazzunk.

Mielőtt egy csomópontot megpróbálnánk a megszokott módon kettéosztani vagy levélle alakítani, megvizsgáljuk, hogy nem tudunk-e levágni üres teret a csomópont cellájának valamelyik széléről. Vágást csak egy bizonyos küszöbérték felett érdemes végezni (pl. akkor, ha az üres tér a cella több mint 25%-át teszi ki).

4.1.2. Kettéosztás

Amennyiben nem lehet vagy nem érdemes levágni üres teret, a csomópontot megpróbáljuk kettéosztani. Szükségünk van egy megállási kritériumra, azaz egy bizonyos pont után nem osszuk tovább a csomópontokat. Az optimális felezősík és megállási pont megtalálása sajnos nem triviális.

Felezési tengelynek mindig a cella leghosszabb éle menti tengelyt vesszük. Ideális esetben a voxelek kocka alakúak kell legyenek, mivel a megfelelő LOD kiválasztása során a voxeleket egy négyzetrácsra vetítjük. A mi esetünkben a voxelek tetszőleges téglatest alakot vehetnek fel, viszont ezzel a tengelyválasztási stratégiával el tudjuk kerülni a túlságosan elnyújtott voxeleket. A sugárkövetés teljesítménye szempontjából ez szuboptimális, viszont az esetek többségében ez a tengely a legjobb, tehát a sebességkülönbség nem túl nagy. Ezzel szemben a kd-fa felépítésének sebessége jelentősen megnő.

A megfelelő felezési pozíció megtalálása jóval összetettebb probléma, mivel ahhoz már hozzá kell férjünk a háromszögekhez is. Két algoritmus közül választunk, attól függően, hogy a csomóponthoz tartozó összes háromszög befére-e egyszerre a memóriába vagy sem.

Out-of-core kettéosztás

Ha a háromszögek nem férnek be a memóriába, akkor egy egyszerű *out-of-core* algoritmust használunk. A csomópont háromszögei egy file-ban vannak eltárolva lista formájában (gyakran használatos a *triangle soup* elnevezés).

A feladat az, hogy határozzunk meg egy elfogadható felezési pozíciót, és annak megfelelően hozzunk létre két temporális listafile-t, melyek a gyerekcsomópontok háromszögeit tartalmazzák.

A felezősíkot a cella középébe tesszük, így a kapott két gyerekcsomópont cellája azonos méretű. A bemeneti listán egyetlen egyszer végighaladva (*streaming*) megvizsgáljuk, hogy egy háromszöget a bal, a jobb vagy mindkét gyerek listájába kell beírni. A kettéosztás után nincs már tovább szükség az eredeti listára, ezért azt a file-t letörölhetjük.

Ez a faépítési algoritmus távol áll az optimálistól, viszont csak kevés számú gyökérközeli csomópontra kell alkalmazni.

In-core kettéosztás

Ha a csomópont háromszögei befférnek a memóriába, az előzőnél sokkal hatékonyabb, költségalapú heurisztikus algoritmust használunk.

A felezősík pozícióját lokálisan mohó *felület terület heurisztikával* (*surface area heuristic*, *SAH*) [Hav00] határozzuk meg. Az algoritmus röviden abból áll, hogy megkeressük azt a felezősíkot, amelyre a SAH költségfüggvény értéke minimális.

Ha ez a minimális költség nagyobb, mint a költség abban az esetben, ha nem osztanánk tovább a csomópontot, akkor a csomópontból levelet csinálunk.

4.1.3. Voxelek létrehozása

Nem erdémes minden egyes csomópontba voxelt tenni, mert általában egy gyerekcsomópont cellája nem sokkal kisebb, mint a szülőé. Egy lehetséges megoldás az, hogy csak mélységben minden harmadikba teszünk [YLM06].

Mi ehelyett egy adaptív stratégiát javasolunk: egy csomópontba csak akkor teszünk voxelt, ha annak sugara legfeljebb feleakkora, mint a legközelebbi voxellel rendelkező ősé. A sugárarány kódolása során a nagyobb pontosság érdekében ezt ki tudjuk használni (így az arány csak 0 és 0.5 között mozoghat).

4.2. A kd-fa treeletekre való bontása

A kd-fa felépítését és treeletekre való bontását nem választjuk szét két külön fázisra, hanem egyszerre valósítjuk meg.

A treeleteket egy ún. *weight-greedy* heurisztika [BDFC02] felhasználásával építjük fel. Adott csomópont esetén megpróbálunk egy olyan treeletet felépíteni, amely kitölt egy blokkot, és gyökere az illető csomópont.

A treeletet inkrementálisan építjük fel. Első lépésként a treeletbe betesszük a gyökércsomópontot. Ezután minden lépésben a treeletbe megpróbáljuk beépetni azt az aktuális treelettel szomszédos csomópontot, amely maximális látogatási valószínűséggel rendelkezik. Ez a valószínűség egyenesen arányos a csomópont cellafelületének területével.

Ha a treeletet a blokk telítettsége miatt nem tudjuk tovább építeni, akkor rekurzívan felépítjük a gyerektreeleteket.

A kapott treeleteket szintekbe csoportosítjuk. Egy szintet folytonosan tárolunk el a file-ban. A blokkokba több egymás melletti treeletet is teszünk, ha van rá mód.

4.3. A háromszögek elrendezése

A háromszögeket a file-on belül a fa után tároljuk el. A tárolási sorrend nagyon fontos, mivel minimizálni szeretnénk a megjelenítés során szükséges blokkhozzáférések mennyiségét.

A megoldás egyszerű: a háromszögeket a levelek létrehozásának sorrendjében írjuk be a file-ba. Az algoritmusunkra jellemző építési sorrend (DFS és weight-greedy keveréke) miatt a háromszögek elrendezése hierarchikus mintát követ.

5. fejezet

Megjelenítés

A megjelenítési rendszer két fő komponensből áll: a sugárkövetést végző renderelőből és a virtuális memóriakezelőből.

5.1. A virtuális memóriakezelő

A rendszer kliens-szerver modellt követ. A kliensek azok a szálak, amelyek hozzáférnek a menedzselt memóriához, a szerver pedig maga a virtuális memóriakezelő.

A memóriakezelő előre megadott, konstans mennyiségű blokkot képes eltárolni a memóriában. A blokkokat *blokk-kerettekben* tároljuk. A blokkhelyettesítő algoritmus egy saját fejlesztésű *CLOCK* [CH81] variáns, ami az LRU (*least recently used*) algoritmust approximálja. Minden memóriában levő blokkhoz hozzárendelünk egy időpecsétet, amely a legutolsó hozzáférés idejét mutatja. Egy időegység két szinkronizációs pont közti intervallumnak felel meg.

5.1.1. A *fetcher* szál

A blokkok aszinkron betöltését egy *fetcher* (behozó) szál végzi, amely folyamatosan fut a kliens szálakkal párhuzamosan. A rendszert úgy terveztük meg, hogy a kliens szálak és a fetcher szál közötti szinkronizáció mennyisége minimális legyen: csak egyetlen szinkronzációs pont van, amelyet valamelyik kliens kell kezdeményezzen.

Két szinkronizációs pont között a kliensek számára hozzáférhető blokkok halmaza garantáltan nem változik, azaz konstans. Ez bővebben azt jelenti, hogy egy előzőleg betöltött blokk elérhető marad legalább a következő szinkronizációs pontig, egy idő közben betöltott blokkhoz pedig addig nem lehet hozzáférni.

A fetcher szál rendelkezik egy *fetch listával* és egy *szabad blokk-keret listával*, amelyek alapján sorban betölti a szükséges blokkokat. A fetch lista a betöltendő blokkok ID-ját tartalmazza, a szabad blokk-keret lista pedig olyan blokk-kereteket, ahova be lehet tölteni a kért blokkokat.

5.1.2. Blokklekérdezés

A memóriakezelő elsődleges funkciója a blokklekérdezés.

Ha a kért blokk elérhető, akkor címfordítással kiszámoljuk a blokk memóriacímét, és visszatérítjük ezt a kliensnek. Ugyanakkor frissítjük a blokk időpecsétjét, ha szükséges.

5.1.3. Blokk-kérés

A kliens kérheti egy blokk betöltését, aminek következtében a blokk ID-t betesszük egy kérési listába. Minden kliens szál rendelkezik egy saját kérési listával, így nincs szükség szinkronizációra.

5.1.4. Szinkronizáció

A szinkronizáció során három lényeges dolog történik: előrhetővé tesszük a frissen betöltött blokkat a kliensek számára, létrehozzuk a fetch listát, és előre lefoglalunk blokk-kereteket.

Elsőször is ideiglenesen leállítjuk a fetcher szálat, és megvizsgáljuk, hogy az előző szinkronizációs pont óta mely blokkat sikerült betöltenie. Ezeket a blokkat beépítjük a címfordítási táblákba.

Ezután a régi fetch listát teljes egészében eldobjuk, az újat pedig úgy állítjuk össze, hogy az elején a legsürgősebb kérések legyenek. Ezt úgy valósítjuk meg, hogy összegezzük a kliensek kérési listáját, és megszámoljuk a kérések számát az egyes blokkokra. Ezek alapján a kérésekhez egy-egy prioritást rendelünk. Az összegzett kéréseket elhelyezzük a fetch listába, amit ezután rendezünk elsősorban prioritás, másodsorban blokk ID szerint.

Az azonos prioritású blokkok ID-juk szerinti növekvő sorrendbe kerülnek, ami miatt a beolvasás során kevesebb lesz a véletlenszerű keresések száma. Merevlemezről való betöltés esetén ez jelentős teljesítménynövekedéshez vezet, de SSD-knél is tapasztalható észrevehető javulás.

A memóriában levő blokkokat a legutolsó hozzáférésük óta eltelt idő alapján három kategóriába soroljuk: *working set* (*munkahalmaz*), *hot data* és *cold data*. A working setbe azok a blokkok tartoznak, amelyekhez az aktuális időegység során hozzáfértek a kliensek. A hot datahoz tartozó blokkokhoz csak régebben fértek hozzá, de nem telt el azóta sok idő. Végezetül cold data-ba soroljuk azokat a blokkokat, melyekhez már régóta nem fértek hozzá.

A blokk-keretek lefoglalását egy *óramutató* [CH81] segítségével oldjuk meg, amely kezdetben az első blokk-keretre mutat. A blokk-kereteket körkörösen kezeljük. Egyetlen szinkronizíció alkalmával a mutatóval legfeljebb két teljes kört teszünk.

Az első körben megpróbáljuk csak a cold blokkokat kidobni, miközben statiszikát készítünk a hot blokkok koráról (a legutolsó hozzáférés óta eltelt idő). Ha így nem sikerült lefoglalni elég blokk-keretet, akkor egy második kör során már hot blokkokat dobunk ki. Felhasználva a statisztikát, a felszabadítást először a régebbi blokkokra végezzük el.

Ha még így sem sikerült elég blokk-keretet lefoglalni, akkor megállunk, mivel már csak a working sethez tartozó blokkok maradtak. Ezeket nem ajánlatos felszabadítani, mert különben instabillá válna a renderelés. A megoldás az, hogy csökkentjük a renderelési részletességet.

Mindezek után növeljuk az aktuális időpecsétet, és újraindítjuk a fetcher szálat.

5.2. A renderelő

A renderelő feladata az, hogy előállítson egy képkockát a modellről a megadott kameraállásból. Minden képkocka lerenderelése után szinkronizál a virtuális memóriakezelővel.

5.2.1. Optimizálás több processzorra

A sugárkövetés hatékonyan párhuzamosítható, mivel a pixeleket egymástól teljesen függetlenül ki lehet számolni. Ezért minden processzoron futtatunk egy-egy renderelő szálat, melyek a teljes képkockának csak egy bizonyos részéért felelősek.

A szálak közti hatékony munkafelosztás nem triviális, mivel a pixeleket nem azonos idő alatt számoljuk ki. Azt is érdemes figyelembe venni, hogy a processzorok más folyamatok futása miatt különböző mértékben lehetnek leterhelve. A munkafelosztás tehát dinamikus kell legyen.

Egy szál által végezhető legkisebb munkaegység a *tile*, azaz egy kisméretű, négyzet alakú pixelterület. Implementációnkban 16x16 pixelnyi tile-okat használunk. A renderelő szálak egy cikluson belül kiválasztanak egy renderelésre váró tile-t, és sugárkövetéssel kiszámolják annak pixeleit. Ez szinkronizálást igényel, amely jelen esetben nagyon gyors kell legyen. Mi nagyteljesítményű *lock-free* szinkronizálást alkalmaztunk atomikus műveletek felhasználásával.

5.2.2. Sugárkövetés

A legközelebbi metszéspontot és az ahhoz tartozó árnyalási paramétereket a Wald [Wal04] által javasolt sugárbejárási algoritmus általunk kiegészített változatával határozzuk meg. Az eredeti algoritmusba beépítettük a voxelalapú LOD mechanizmusunk és a virtuális memóriakezelési rendszerünk támogatását.

(a) Betöltés közben (b) Betöltés után

5.1. ábra. Egy modell részleteinek aszinkron betöltése

Amikor bejárás során egy voxellel rendelkező csomópontba lépünk, megbecsüljuk, hogy az a képsíkra vetítve hány pixelt foglal. Ha ez az érték kisebb, mint egy tetszőleges küszöbérték (pixel hibatolerancia), akkor megállunk a sugárbejárással. Metszéspontnak azt a pontot vesszük, ahol a sugár az illető voxelt metszi (5.2. ábra). Mivel a voxelt a csomópont cellája határolja be, a metszést nem kell külön elvégezni.

5.2. ábra. Hibrid voxel/háromszög megjelenítés (a vörös zónák háromszögekből állnak)

Egy kapocs-csomópontba való kerüléskor megpróbáljuk lekérdezni a virtuális memóriakezelőtől a célblokk címét. Ha a blokk nem elérhető, akkor megállunk, és kérjük annak betöltését. Helyettesítő geometriának azt a legközelebbi voxelt vesszük, amely az aktuális csomópont egyik ősében található (5.1. ábra). A blokk betöltési prioritása a kérések számától függ, amely így a helyettesítő voxel által foglalt pixelek számával egyenlő.

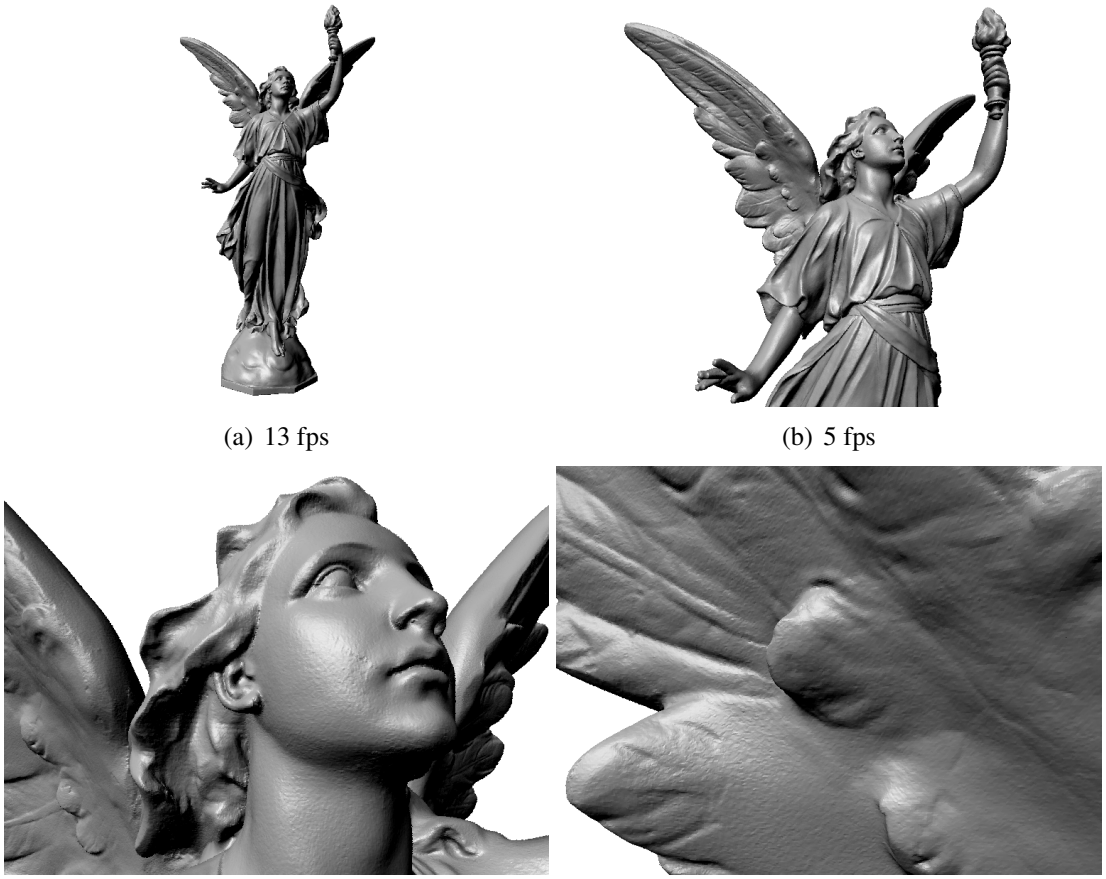
6. fejezet

Eredmények

6.1. Implementáció

A módszerünket teljes mértékben C++-ban implementáltuk. Minden számítást a CPU-n végzünk el.

A megjelenítő a modellt egy direkcionális fényforrással világítja meg, amihez Phong árnyalást használ.



(a) 13 fps (b) 5 fps

(c) 2 fps (d) 2 fps

6.1. ábra. Renderelési sebesség különböző kameraállásokból

6.2. Teljesítmény

Az implementációt a következő laptopkonfiguráción futtattuk: *Intel Core 2 Duo T5500* (1.66 GHz, két mag) processzor, 2 GB RAM, 7200 RPM HDD.

A teljesítménymméréseket a rendelkezésünkre álló legnagyobb modellel végeztük: *Stanford Lucy* (kb. 28 millió háromszög). A modell háromszöglista formátumban kb. 1 GB-ot, a speciális formátumban pedig 2.9 GB-ot foglal. Az előfeldolgozás kb. 20 percig tartott.

A renderelési felbontás 768x576 volt, a hibatoleranciát pedig 1.25 pixelre állítottuk. A modell bejárása során a sebesség 2-13 fps között mozgott (6.1. ábra), tehát interaktív volt.

6.3. További kutatások

A renderelési sebességet jelentősen fel lehet gyorsítani, ha különálló sugarak helyett *sugárcsomagokat* (*ray packet*) követünk [Ben06]. Ez a módszer növeli a memóriakoherenciát, és ki tudja használni a processzorok SIMD utasításkészletét (pl. SSE, AVX, VMX stb.). Előzetes tesztjeink alapján a módszerünk esetében körülbelül *háromszoros* teljesítménynövekedésre lehet számítani.

További látványos teljesítménynövekedést érhetünk el, ha a módszerünket optimizáljuk speciális, masszívan párhuzamos processzorokra (pl. *Intel Larrabee*, *IBM Cell*, GPU-k az *OpenCL* API-n keresztül programozva).

A jobb renderelési minőség érdekében a LOD rendszert érdemes lenne kibővíteni *view-dependent* (látószögtől függő) voxelekkel [GM05].

6.4. Konklúzió

Egy olyan 3D-s megjelenítési módszert sikerült kidolgoznunk, amely segítségével átlagos teljesítményű hardveren interaktívan bejárhatunk bonyolult – a rendszermemóriánál akár sokkal nagyobb méretű – modelleket.

Irodalomjegyzék

[BDFC02] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Efficient Tree Layout in a Multilevel Memory Hierarchy. In *In Proc. 10th Annual European Symp. on Algorithms (ESA), volume 2461 of LNCS*, pages 165–173, 2002.

[Ben06] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2006.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK – A Simple and Effective Algorithm for Virtual Memory Management. In *SOSP '81: Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 87–95, New York, NY, USA, 1981. ACM.

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[GM05] Enrico Gobbetti and Fabio Marton. Far Voxels – A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. *ACM Transactions on Graphics*, 24(3):878–885, August 2005. Proc. SIGGRAPH 2005.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[YLM06] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Fast LOD-based Ray Tracing of Massive Models. *The Visual Computer: International Journal of Computer Graphics*, 22(9):772–784, 2006.