DIPLOMAMUNKA

Rákos Dániel

Debrecen

2011

Debreceni Egyetem

Informatikai Kar

GPU ALAPÚ VÁGÓALGORITMUSOK

Témavezető:

Dr. Schwarcz Tibor

egyetemi adjunktus

Készítette:

Rákos Dániel

programtervező informatikus

Debrecen

2011

TARTALOMJEGYZÉK

Bevezetés

Immár több mint kilenc éve foglalkozok háromdimenziós grafikával, ezen belül is elsősorban OpenGL alapú valós idejű megjelenítéssel, különös hangsúlyt fektetve a mára már alapeszköznek számító shader nyelvekkel, amely témával a 2008-ban írt szakdolgozatom is foglalkozik, melynek címe "A shader nyelvek alkalmazása a valósidejű fotorealisztikus megjelenítésben"[1].

Idő közben a grafikus processzorok (GPU-k) további komoly változásokon mentek keresztül, melyet az ipar és a szabványügyi intézetek is követtek. Mára már a GPU-k általános célú felhasználására és programozására több API is létezik, melyek közül érdemes megemlíteni a nyílt szabványú OpenCL-t[E1], amit a Khronos Group nevű szabványügyi szervezet alkotott, ami az OpenGL szabvány fejlesztéséért is felelős, valamint a CUDA-t, ami az NVIDIA saját fejlesztésű GPGPU (General Purpose GPU) API-ja.

Ennek a fejlődésnek az eredménye az is, hogy ma már valós idejű fizikai szimulációs motorok is léteznek, melyek kiaknázzák a GPU-kban rejlő óriási számítási tejlesítményt és programozhatóságot. Ezen kívül a legújabb szuperszámítógépek is már nagyrészt GPU-kból állnak, hisz ez a hardver mára már egy mondhatni általános célú párhuzamos processzorrá fejlődte ki magát.

Azonban mielőtt ilyen messzire mennénk a GPU-k eredeti felhasználási céljától, vegyük figyelembe, hogy annak ellenére, hogy ezek a processzorok már ekkora programozhatósági rugalmasságot biztosítanak, többnyire a számítógépi grafikában még továbbra is csak az alapvető feladat-végzési módjait használják az iparban. Habár a GPU tényleg elsősorban háromszögek raszterizálására és textúrázására készült, elérkeztünk egy olyan mérföldkőhöz, melyet a valós idejű megjelenítési szoftverek fejlesztőinek is figyelembe kell venni, mégpedig hogy a magasabb szintű megjelítéssel kapcsolatos algoritmusok is már többnyire átültethetőek a GPU-kra, ide értve a jelenet-kezelést (scene management) és a láthatósági algoritmusokat (visibility and culling algorithms).

A diplomamunkám ezen terület kérdéskörével foglalkozik, beleértve az ide tartozó már kiaknázott és még nem kiaknázott lehetőségeket. Ahogy majd a továbbiakban látni fogjuk, a technikai lehetőségek már többé-kevésbé adottak és főként csak a programozói kreativitás és az API-k által nyújtott funkciók szabhatnak korlátot az alkalmazási lehetőségeknek.

Az első fejezetben a vágóalgoritmusok egy olyan osztályozásáról lesz szó, amely a téma feltárásának struktúráltságát hivatott vázolni. Szó lesz olyan alapfogalmakról, melyek szükségesek az egyes módszerek megértéséhez, vizsgálatához.

A második fejezet a közismert elsőrendű vágóalgoritmusokat mutatja be, különös figyelmet fordítva ezek hardveres megvalósítására, illetve a mai eszköztárak által nyújtott lehetőségekre, melyeket kihasználhatunk ezen algoritmusok felhasználásakor.

A harmadik fejezet egy összefogaló az alapvető magasabb rendű vágóalgoritmusokról, melyek bizonyos része már régóta hardveresen támogatott.

A negyedik fejezetben lesz szó olyan jövőbe mutató megközelítésekről, melyek segítségével hatékonyan tudunk nagy mennyiségű adathalmazra láthatósági és vágóalgoritmusokat alkalmazni a GPU-val. A bemutatott technikák köszül néhányat már a gyakorlatban is alkalmaznak bizonyos esetekben, néhányat pedig egyelőre még nem lehet megvalósítani az API-k által nyújtott eszköztárak hiányosságai miatt.

A diplomamunka elsődleges célja, hogy bemutassa azt, hogy miért is van szükség magasabb rendű GPU alapú vágóalgoritmusokra, hogy milyen lehetőségek állnak rendelkezésre most, illetve milyen eszközrendszer várható a közeljövőben, ami további rugalmasságot tesz lehetővé.

A témaválasztásomat az motiválta, hogy az iparban, különösképpen a játékfejlesztők körében az egyre nagyobb és részletesebb virtuális világok megjelenítésénél komoly problémát okoz az, hogy túl sok, a CPU-n eldöntendő láthatósági vizsgálatot és vágóalgoritmust kell végrehajtani, melyek töredezetté teszik a rajzolás folyamatát, ami hatékonysági problémát jelent a GPU-k egyre jobban növekvő teljesítménye miatt, mivel a CPU már azzal is nehezen bírkózik meg, hogy a GPU számára mindig adjon munkát.

A diplomamunkában bemutatott ötletek és algoritmusok a jövőben megoldhatják ezt a töredezettséget azáltal, hogy a láthatósági- és vágóalgoritmusokat, valamint a hozzájuk kötődő döntési folyamatokat is a GPU-n hajthatjuk végre.

1. A vágóalgoritmusok osztályozása

A vágóalgoritmusok itt bemutatott osztályozása egy saját, a problémák tárgyalása szempontjából praktikus megközelítés, amely az algoritmusokat két fő csoportra osztja: elsőrendű- és magasabb rendű vágóalgoritmusok.

Az elsőrendű vágóalgoritmusok csoportjába azok a módszereket soroljuk, melyek alapvető számítógépi grafikai elemeken dolgoznak, mint például a legegyszerűbb geometriai primitívek (pontok, szakaszok, háromszögek), valamint a fragmensek (vagy képpontok ha tetszik).

Minden olyan algoritmus, amely ennél magasabb rendű primitívekkel dolgozik, legyen az egy modell, objektum vagy jelenet, a magasabb rendű vágóalgoritmusok csoportjához tartozik.

2. Elsőrendű vágóalgoritmusok

Az elsőrendű vágóalgoritmusok az egyszerű geometriai és képfeldolgozási primitívekről állapítja meg, hogy ezek láthatóak lesznek-e majd a végső képen, illetve "vágja" azokat amennyiben csak részlegesen láthatóak. Ezen primitívek családjába a következő elemek tartoznak:

- Pont (geometriai értelemben vett pont)
- Szakasz (vagy szakasz-csík)
- Háromszög (vagy háromszög-csík)
- Fragments (vagy képpont)

Az itt felsorolt primitívek választása nem tetszőleges, az alapkoncepció az, hogy csak olyan elemeket nevezünk elsőrendű primitíveknek, amelyekkel a GPU-k is minden további feldolgozás nélkül, közvetlenül dolgozni tudnak.

Természetesen az egyes primitív típusok vágására más és más algoritmust lehet és érdemes használni. Ezek közül ismertet néhányat ez a fejezet.

2.1. Alapfogalmak

A GPU-k, és az alkalmazásukhoz használt API-k egy csővezeték modell alapján épülnek fel, amelynek elsődleges bemenetei a geometriai primitívek, kimenete pedig maga a digitális kép. Ezen csővezeték vázlatos felépítését a 2.1. ábrán láthatjuk.

Az ebben a fejezetben tárgyalt vágóalgoritmusok a bemeneti geometriai primitívek, valamint a raszterizáció eredményeképpen előálló fragmensek láthatóságát hivatottak vizsgálni. Egy fragmens az egyszerűség kedvéért tekinthető úgy, mint egy képpont, azonban a dolgozat folyamán a fragmens fogalmat fogjuk használni, mivel a kép ezen alkotóelemein a gyakorlatban és az API-k specifikiációja szerint is további műveletek hajtódnak végre, melynek eredménye végül a képpont (pixel) lesz. Habár ez a különbség a dolgozatban bemutatott problémakör tárgyalása szempontjából nem annyira lényeges, az egyértelműség kedvéért a továbbiakban a fragmens fogalmat fogom használni.

2.2. Geometriai primitívek vágása

A grafikus megjelenítés csővezetékének sematikus ábrája alapján a geometriai primitívek vágása a primitívek összeállítása (primitive assembly) előtt kell megtörténnie. Az, hogy hol, az elméletben lényegtelen, azonban gyakorlati megfontolásokból ennek a lehető legkésőbb kell megtörténnie, hisz a korábbi fázisokban különböző transzformációkat végezhetünk a bemenő adatokon, melyeket általában figyelembe kell venni a láthatósági vizsgálatoknak is.

Ebben az alfejezetben két közismert vágóalgoritmust fogunk bemutatni: a nézet gúlával való vágást (view frustum culling) és a hátsólap vágást (back-face culling). Ezután pedig arról lesz szó, hogy milyen lehetőségeket kínálnak a GPU-k számunkra ezeknek az algoritmusoknak a használatával, beleértve a beépített (hardveres) és a programozható (pszeudo-hardveres) módszereket.

2.2.1. View frustum culling

Jellemzően a háromdimenziós megjelenítésben a nézet gúla segítségével szemléljük a világot. Centrális projekció esetében ez jellemzően egy csonkagúla, míg párhuzamos vetítésnél egy hasáb.

A nézet gúlával való vágás azt hivatott eldönteni, hogy egy adott geometriai primitív benne van-e ebben a gúlában, illetve a részleges tartalmazás esetében a szakasz és háromszög primitíveket el is vágja részleges láthatóság esetén.

Az algoritmust legegyszerűbben úgy valósíthatjuk meg, ha a jól ismert Cohen-Sutherland vágóalgoritmust[2] terjesztjük ki a harmadik dimenzióra. Ez után az algoritmus megadása analóg módon adódik. Amennyiben szakasz és/vagy háromszög primitívek esetén el is szeretnénk vágni azokat a határok mentén, úgy szintén a Cohen-Sutherland algoritmusban használt módszereket használhatjuk, annyi különbséggel, hogy a négy vágó egyenes helyett hat vágó síkunk van.

Ez az algoritmus közismert és népszerű az egyszerűsége miatt, azonban a gyakorlati alkalmazásával akadhatnak problémák és ahogy azt majd meglátjuk, a GPU-k sem alkalmazzák ezt a módszert.

2.2.2. Hátsólap vágás (back-face culling)

A másik legismertebb geometriai vágóalgoritmus a hátsólap vágás, melynek alapötlete nagyon egyszerű: minden átlátszatlan geometria testnek a tér síkra való leképezése miatt csak az egyik "oldalát" látjuk. Ez a számítógépi grafikában nagyjából azt jelenti, hogy a testjeinket modellező poliéder azon lapjait, amelyek "háttal" vannak a kamerának, eldobhatjuk.

Mivel általában a számítógépi grafikában a poliédereket háromszögekből építjük fel, ez a feladat egy háromszög és egy pont egymáshoz viszonyított helyzetének vizsgálatára redukálódik, amelyet kétféleképpen is vizsgálhatunk.

Egyrészt felhasználhatjuk a háromszög felületi normálját, vagy tekinthetjük a háromszög síkvetületének "irányítottságát", azaz, hogy a háromszög csúcsainak vetületei órajárással megegyező vagy ellentétes irányban helyezkednek el. Természetesen ez utóbbi esetén a háromszögek csúcsait a megfelelő sorrendben kell megadni a rajzoláskor.

2.2.3. Geometriai primitívek vágása GPU-n

Az előző fejezetben tárgyalt vágóalgoritmusokat a GPU-k is támogatják hardveresen. A hátsólap vágáshoz a síkvetület irányítottsága alapján dönt. Érdemes továbbá megjegyezni, hogy mindkét művelet közvetlenül a primitívek összeállítása előtt történik.

Természetesen ezen kívül, a mai GPU-k nagy mértékű programozhatóságának köszönhetően, további pszeudo-hardveres vágóalgoritmusok is implementálhatóak köszönhetően a negyedik programozható GPU generációnak, melyek bevezették a geometry shader-t. Azon kívül, hogy ez a speciális shader egy teljes primitíven dolgozik (legyen az háromszög, szakasz vagy pont), ez az egyetlen geometria feldolgozó shader, amely további primitíveket hozhat létre vagy akár "el tud dobni" egy primitívet. Ez utóbbi tulajdonságot használhatjuk fel további vágóalgoritmusok implementálására.

Egyik jó példa az, amikor a nézet gúlával való vágást a geometry shaderben implementáljuk szoftveresen. Habár első ránézésre ez egy fölösleges művelet, hisz a primitívek összeállításakor úgyis végrehajtódik a hardveresen beépített vágás, azonban a gyakorlatban ez a megközelítés is nagy teljesítménynövekedéshez vezethet. Az oka nagyon egyszerű: a primitívek összeállításával foglalkozó

hardver elem, ami a beépített vágóalgoritmusokat is végzi, nehezen párhuzamosítható anélkül, hogy esetleges nem-determinisztikus képeket eredményezzen a rossz sorrendben történő raszterizálásból kifolyólag. Emiatt ez a hardver elem könnyen a megjelenítés szűk keresztmetszetévé válhat. Ezen tud segíteni egy szoftveres "elővágás" a geometry shader segítségével.

2.3. Fragmensek vágása

A modern grafikus csővezeték alapján raszterizálás eredményeképpen előálló fragmensek több teszten és egyéb műveleten esnek át mielőtt végül a célfelületen mint képpont megjelennek. Ezek közül a legfontosabbakat említve beszélhetünk a következőkről:

- *Fragment shader* – GPU program, mely a fragmens végső színét, mélység értékét és esetlegesen további attribútumait hivatott meghatározni.
- *Mélység teszt* – a fragmenshez tartozó mélység értéket hasonlítja össze egy referencia értékhez és ez alapján dönti el, hogy a fragmenst eldobjuk-e vagy sem.
- *Stencil teszt* – a fragmenshez tartozó stencil értéket (egész szám) hasonlítja össze egy referencia értékhez és ez alapján dönti el, hogy a fragmenst eldobjuk-e vagy sem.
- *Alfa teszt* – a fragmenshez tartozó alfa értéket (általában áttetszőséget mérő szám) hasonlítja össze egy referencia értékhez és ez alapján dönti el, hogy a fragmenst eldobjuk-e vagy sem.
- *Blending* – a célfelületen a fragmenshez tartozó képpont új értékét a fragmens és a képpont aktuális színének faktorizált összegeként, szorzataként vagy egyéb kombinációjaként állítja elő. A mélység és stencil érték is csak ekkor, már a tesztek után íródik a célfelület megfelelő bufferjébe.

A fragmens shadertől eltekintve jellemzően az összes művelet bedrótozott hardveres logika alapján működik. Ezek közül is a számunkra legfontosabb a mélység teszt lesz, amit láthatósági vizsgálatra szoktak használni, mégpedig a klasszikus Z-Buffer algoritmus segítségével.

2.3.1. Z-Buffer

A Z-Buffer vagy mélység tároló algoritmus egy Z-Buffernek nevezett adatstruktúrával dolgozik, mely minden egyes pixelhez a hozzá rendelt Z vagy mélység értéket tárolja a normalizált látótérben.

Az algoritmus egyszerű: kezdetben a buffer összes eleme a hátsó vágósík Z koordinátájának értékét tartalmazza, majd minden egyes raszterizált fragmens esetén összehasonlítja a fragmens mélység értékét a fragmenshez tartozó pixelnek megfelelő mélység értékkel, amit a Z-Buffer tárol.

Amennyiben a fragmens mélység értéke kisebb a tárolténál, úgy a fragmens látható, amennyiben nagyobb vagy egyenlő, úgy a fragmens takarásban van.

A gyakorlatban a mélység értékek összehasonlítása nem csak ily módon történhez, a GPU-k is ennek megfelelően konfigurálhatóak, hogy milyen relációs operátort használjanak az aktuális és a referencia értékek összehasonlításakor. Ez több bizonyos összetettebb algoritmusoknál szükséges lehet.

Szintén érdemes megjegyezni, hogy amennyiben egy fragmens átment a Z-Buffer tesztjén, a fragmens mélység értékét nem feltétlen szükséges beírni a Z-Bufferbe, mint új referencia érték. Ez például áttetsző objektumok megjelenítésénél válhat hasznunkra, amennyiben nem szeretnénk mélység szerint ezen objektumok poligonjait.

2.3.2. Hierarchikus Z-Buffer

A hierarchikus Z-Buffer[3] a klasszikus Z-Buffer algoritmust hivatott tovább fejleszteni oly módon, hogy egyetlen Z-Buffer használata helyett egy sor Z-Buffert használunk. Ezen sor első eleme megegyezik a klasszikus Z-Bufferrel, minden további pedig pontosan fele olyan magas és széles, és értékei úgy állnak elő, hogy az előző buffer méretét 50%-kal csökkentjük és a hagyományos bilineáris szűrő helyett egy maximum szűrőt használunk. Egy így előállított Z-Buffer lánc egy részét láthatunk a 2.2. ábrán.

Habár a hierarchikus Z-Buffer felépítése és karbantartása időigényes művelet lehet, a mélység teszteket nagyban felgyorsíthatja, mivel egyetlen összehasonlítással több fragmenst is el tud dobni. Ezt úgy éri el, hogy a raszterizáláskor "szuperfragmenseket" vizsgál először, melyek NxN-es fragmens lapok. Ezeknek a szuperfragmenseknek a minimális értékét a raszterizáló egyszerűen meg tudja határozni, így amennyiben ez a minimális Z érték nagyobb, mint a szuperfragmensnek megfelelő Z-Bufferben tárolt érték, úgy az egész szuperfragmens eldobható.

2.3.3. Fragmensek vágása GPU-n

A modern GPU-k a hierarchikus Z-Buffer egy speciális változatát támogatják hardveresen, ami két láncot tartalmaz, egyet amit minimum szűrővel állít elő és egyet amit maximum szűrővel. Ez azért szükséges, hogy a hierarchikus Z-Buffer nyújtotta sebességnövekedést akkor se veszítsük el, ha a mélység értékek összehasonlítására használt relációs operátort megváltoztatjuk.

Ezen kívül jellemzően a GPU-k nem építik fel a teljes hierarchikus Z-Buffer láncot, hanem annak csak bizonyos elemeit. Ez helytakarékosság és a frissítéshez szükséges idő csökkentése érdekében történik.

Maga a mélység teszt alkalmazásának szempontjából az, hogy hagyományos Z-Bufferrel vagy pedig hierarchikus Z-Bufferrel dolgozuk, a felhasználó számára transzparens, a végeredményt nem módosítja, azonban érdemes tisztában lenni a hardver működésével ahhoz, hogy azt optimális módon használjuk ki azt.

3. Magasabb rendű vágóalgoritmusok

Magasabb rendű vágóalgoritmusnak fogunk nevezni minden olyan láthatósági vizsgálatot és vágást végző algoritmust, amely az egyszerű geometriai primitíveken, illetve fragmenseken végrehajtott láthatósági vizsgálatokkal ellentétben magasabb rendű primitíveken, úgynevezett objektumokon dolgozik. Objektumok alatt rendszerint komplex geometriai modelleket, illetve nem megjelenítendő virtuális elemeket (például egy szoba, jelenet, stb.) fogunk érteni.

Jellemzően egy ilyen objektumot a következő tulajdonságokkal fogunk jellemezni, illetve a következő tulajdonságok alapján fogunk láthatóságot vizsgálni:

- Az objektumhoz tartozó modell geometriájának leírásával, amely jellemzően egy primitív típusból, valamint egy vertex és egy index listából áll.
- Az objektumhoz tartozó egyszerű maximális tartalmazott térfogat primitívvel (általában egy vagy több gömb, ellipszis vagy téglatest, amelyeket az objektum teljes egészében tartalmaz).
- Az objektumhoz tartozó egyszerű minimális befogadó térfogat primitívvel (általában egy vagy több gömb, ellipszis vagy egy téglatest, amely az objektum minden egyes vertexét tartalmazza).

Az utóbbi kettő olyan láthatósági vizsgálatok esetén lesz hasznunkra, melyek az objektum teljes geometriai leírására alkalmazva túl lassúak vagy egyéb szempontból impraktikusak lennének. Érdemes megjegyezni, hogy többnyire a magasabb rendű vágóalgoritmusok mind ebbe a kategóriába esnek, hisz amennyiben az algoritmusokat a teljes geometriai modellre végezzük, úgy az gyakorlatilag elsőrendű vágóalgoritmusokra redukálódik.

A minimális befogadó- illetve a maximális tartalmazott térfogat primitív meghatározása külön feladat, mellyel ez a szakdolgozat nem foglalkozik, azonban önmagában is érdekes feladatkör.

3.2. View frustum culling

A nézet gúlával való vágás objektumok esetében is triviális. Egyszerűen meg kell határoznunk, hogy az objektumhoz tartozó adathalmaz alapján vizsgáljuk a tartalmazást, illetve esetlegesen a részleges tartalmazás mértékét az adott centrális projekcióhoz tartozó nézet gúlához viszonyítva.

Amennyiben a nézet gúlával való vágást az objektumhoz tartozó modell teljes geometriai információ tekintetében szeretnénk végrehajtani, úgy az algoritmus az elsőrendű nézet gúlával való vágásra redukálódik, hisz ilyenkor a vágást az objektum összes őt alkotó geometriai primitívjére végre

kell hajtsuk. Ezért, illetve mivel ez számunkra további előnyökkel már nem jár, hisz a GPU ezt automatikusan elvégzi, ezzel az esettel nem fogunk foglalkozni.

Természetesen itt a minimális befogadó térfogat primitív jelenthet számunkra hatalmas előnyt. Szemben több száz, több ezer vagy akár több tízezer háromszög egy nézet gúlához viszonyított tartalmazási vizsgálatával, ebben az esetben csak egy gömb, ellipszis vagy egy téglatest helyzetét kell vizsgálni a nézet gúlához képest.

Ezek a vizsgálatok nagyon egyszerűek és gyorsak és természetesen a Cohen-Sutherland féle vágóalgoritmus itt is alkalmazható. A gyakorlatban is bevett szokás az objektumok ily módon történő láthatósági vizsgálata, amit a mai napig többnyire még csak a CPU-n végeznek.

3.3. Takarás vizsgálat

Könnyen észrevehető, hogy az eddig bemutatott vágóalgoritmusok két fő csoportba sorolhatóak:

- A nézet gúlán kívüli primitívek vágása (view frustum culling)
- A nézet gúlán belüli primitívek vágása (back-face culling, Z-Buffer)

Míg az előbbi kategória esetében a feladat triviális, a nézet gúlán belüli primitívek vágása ennél sokkal összetettebb és alapvetőleg egy problémakört céloznak meg: a takarás vizsgálatot.

A takarás vizsgálat feladata az, hogy olyan objektumok láthatóságáról döntsön, melyek a nézet gúlán belül helyezkednek el, azonban egy másik objektum takarása miatt esetlegesen nem láthatóak a végső képen.

Ellentétben az egyszerű lokális láthatósági problémákkal itt egy globális feladatról van szó, hisz a vizsgált objektumok egymáshoz viszonyított helyzetétől függ a láthatósági vizsgálat eredménye. Érdemes megjegyezni azt is, hogy ezek a módszerek általában nem "igen-nem" formájú választ adnak a láthatóság kérésére, kizárólag a nem láthatóságra adnak biztos választ.

A már korábban bemutatott takarás vizsgálatra alkalmas algoritmusok közül a back-face culling például a modell zártságáról tett feltételezés alapján biztosan meg tudja mondani, hogy ha egy lap hátsó lap, akkor az nem is látható. Azonban lehetnek olyan lapok, amelyek nem hátsó lapok, de nem láthatóak és ez ráadásul nem csak globális értelemben, hanem lokális értelemben is igaz, hisz ugyanannak a modellnek egy másik lapja takarhatja a vizsgált lapot.

Ugyanez igaz a Z-Buffer algoritmusra is, hisz habár gyakorlatilag a mélység teszt globális láthatósági vizsgálatnak minősül, kizárólag az eddig már feldolgozott objektumok szempontjából globális, hisz bármikor jöhet egy újabb objektum, mely az összes eddig raszterizált objektum elé kerül.

Ahogy láthatjuk a takarás vizsgálat egy különösen nehéz probléma, melyre habár jó néhány megoldás létezik, azok vagy csak részlegesek, vagy pedig alkalmatlanok a valós idejű megjelenítésben való alkalmazásra.

3.3.1. Analítikus megközelítés

Nyílvánvalóan tetszőleges objektumhalmaz takarás vizsgálatára létezik exakt analítikus vagy geometriai megoldás. Talán a legegyszerűbb megközelítés az, ha meghatározzuk az egyes objektumok a nézőpontból tekintett árnyék vagy vetület térfogatát, majd ehhez viszonyítva vizsgáljuk tartalmazás szempontjából a többi objektumot. Természetesen ennek az algoritmusnak a végrehajtása az objektumokhoz tartozó teljes geometriai adathalmazon számításigényes feladat. Emiatt ez a naív megközelítés teljességgel alkalmatlan valós idejű alkalmazásra.

Az algoritmus hatékonyságának növelésére bevezethetjük a minimális befogadó- és a maximális tartalmazott térfogat primitív használatát. Ebben az esetben az "árnyékoló" objektum teljes modellje helyett annak a maximális tartalmazott térfogat primitívjét használjuk és a vizsgálandó "árnyékolt" objektumok helyett pedig használhatjuk azok minimális befogadó térfogat primitívjét. Így a számítási igény nagyságrendekkel kisebb, azonban továbbra is egy $O(n^2)$ összetettségű algoritmusról beszélünk, ami még ilyen relatív kis méretű adathalmaz esetében is erősen limitálja az algoritmus használhatóságát.

Nem szabad figyelmen kívül hagynunk azt a tényt, hogy habár az analítikus megközelítés adhatja számunkra a legpontosabb megoldást a problémára, mi továbbra is diszkrét adathalmazon dolgozunk, hisz a képernyőn megjelenítendő kép pixeleinek halmaza is diszkrét.

Ezt a megjegyzést figyelembe véve eljutunk a sugárkövetéses algoritmusokhoz (ray tracing). Ekkor minden egyes pixelen keresztül indítunk egy sugarat a nézőpontból és vizsgáljuk, hogy az mely objektum mely pontjával találkozik először (ray casting).

A sugárkövetéses algoritmusok alkalmazása takarás vizsgálatra azonban nagy mértékben attól függ, hogy milyen adatstruktúrával reprezentáljuk a világunkat, hisz a ray casting művelet számítás igénye nagy mértékben ettől függ. Szerencsére rengeteg olyan adatstrukúra létezik, amely $O(\log(n))$ idő alatt elvégzi ezt a feladatot, azonban egy dinamikus világ esetében az objektumok folyamatos mozgásban vannak és sajnos általában ezek az adatstruktúrák nehézkesen módosíthatóak, nem is beszélve arról, hogy még így is egy $O(m \cdot \log(n))$ összetetségű algoritmusról beszélünk, ahol az m (a pixelek száma) meglehetősen nagy.

3.3.2. Occlusion query

Az occlusion query egy harveres lehetőség a takarás vizsgálat feladatának megoldására. A segítségével alapjában véve gyakorlatilag bármilyen számítási költség nélkül megállapíthatjuk egy adott objektumról, hogy a hozzájuk tartozó fragmensek közül mennyi ment át a már korábban tárgyalt mélység teszten. Ezen információ birtokában el tudjuk dönteni, hogy az illető objektum látható-e.

Habár első hangzásra furcsának hat ez a megközelítés, hisz ahhoz, hogy eldöntsük egy objektumról, hogy látható-e, először raszterizálnunk kell azt. A gyakorlatban azonban igen gyakran nagy mértékű sebesség növekedést érhetünk el az occlusion query-k segítségével.

Az egyik megközelítés[E2] alapján abban a lépésben, amikor az occlusion query-t végrehajtjuk, használhatjuk az objektum eredeti geometriai modellje helyett a hozzá tartozó minimális befogadó térfogat primitívet, így az előfeldolgozó lépés számítási- és memória sávszélesség igényét nagyságrendekkel csökkenthetjük. Ráadásul a hierarchikus Z-Buffernek köszönhetően ez a lépés akkor is meglepően gyors, ha ez a primitív a képernyő nagy részét le is fedi. Természetesen ebben az esetben figyelni kell arra, hogy a minimális befogadó térfogat primitív raszterizálásakor a Z-Bufferbe ne írjunk semmit, csak és kizárólag a mélység tesztet hajtsuk végre. Természetesen amennyiben ezt a megközelítést alkalmazzuk, az occlusion query eredménye lehet fals pozitív, hisz az, hogy a minimális befogadó térfogat primitív látható még nem feltétlenül jelenti azt, hogy az eredeti objektum is látható.

Egy másik alkalmazási lehetőség a több lépéses megjelenítési algoritmusok optimalizálása[4]. Ekkor az első lépésben végzünk csak occlusion query-t és ennek eredményét használjuk fel a további megjelenítési lépésekben. Mára már ennek a technikának sokkal kisebb a jelentősége, mivel a mai GPU-k gyakorlatilag tetszőleges hosszúságú és komplexitású programokat képesek futtatni, így maga a több lépéses megjelenítési algoritmusok is kezdenek eltűnni. Minden esetre amennyiben valamilyen ok folytán mégis csak szükségünk lenne több lépéses megjelenítésre, ez esetben mindenképp érdemes felhasználni az occlusion query által nyújtott lehetőségeket.

Az occlusion query-k egyetlen komoly problémája, hogy a CPU és a GPU asszinkron működéséből kifolyólag értékes CPU és GPU időt veszíthetünk, amikor egy korábbi occlusion query eredményére várunk. Ezt hivatott részben kiküszöbölni a feltételes rajzolási módszerek[E3] megjelenése a grafikus API-kban.

4. GPU támogatott magasabb rendű vágóalgoritmusok

Ebben a fejezetben a már korábban bemutatott magasabb rendű vágóalgoritmusok GPU támogatott változatait ismertetjük. Habár ma már szinte bármilyen vágóalgoritmust végrehajthatunk a GPU-n, a diplomamunka célkitűzése olyan algoritmusok keresése és bemutatása, amelyeknek gyakorlatban is nagy jelentősége lehet a közeljövőben, tehát arra koncentrálunk, hogy olyan algoritmusokat keressünk, amelyek alkalmasak a valósidejű megjelenítésben való alkalmazásra és mérhetően hatékonyabbak azok CPU alapú implementációitól. Ennek előfeltétele az, hogy minél jobban kihasználjuk a GPU adottságait. A két fő szempont ennek megfelelően a következő:

- Az adatok beetetése a lehető legnagyobb darabokban kell történnie, mivel a GPU leginkább nagy mennyiségű adat párhuzamos feldolgozására alkalmas.
- Az egyes objektumokhoz tartozó adatok függetlenek kell hogy legyenek egymástól, annak érdekében, hogy a lehető leghatékonyabban használjuk ki a GPU párhuzamosított architektúráját.
- Ki kell használnunk a CPU és a GPU asszinkron működését, azaz a szükséges CPU-GPU kommunikáció mennyiségét minimalizálnunk kell, különösen a CPU oldali lekérdezéseket.

4.1. Alapfogalmak

Ahhoz, hogy egy kellően hatékony GPU alapú vágóalgoritmust konstruáljunk, ismernünk kell azokat a módszereket, ahogy bemenetet tudunk adni a GPU-nak, illetve, hogy milyen módon nyerhetünk ki részeredményeket abból.

4.1.1 Rajzolási parancsok

Első körben tekintsük a bemenetek előállítása számára rendelkezésre álló eszközrendszert. Itt úgynevezett rajzolási parancsokról beszélünk. Ezeknek a rajzolási parancsoknak különböző nevei vannak az egyes API-k esetében, így a konvenció az lesz, hogy az OpenGL által használt neveket fogom használni és többnyire a paraméterek is ennek megfelelően lesznek tárgyalva.

4.1.1.1 Klasszikus rajzolási parancsok

```
DrawArrays( mode, first, count )
```

Ez a legegyszerűbb rajzolási parancs, mely egy egyszerű primitív lista feldolgozását, vagy ha úgy tetszik rajzolását eredményezi. A *mode* paraméter a primitív típusát adja meg, mely többek között lehet pont, szakasz vagy háromszög. A *first* és a *count* paraméterek adják meg, hogy a beállított vertex lista bufferből melyik elemétől kezdve és hány elemet akarunk feldolgozni. Maga az adatok egy GPU memóriában lévő bufferben vannak a hatékonyabb feldolgozás érdekében.

```
DrawElements( mode, first, count )
```

Ez a parancs egy egyszerű primitív lista helyett egy indexelt primitív halmazt dolgoz fel. Ez úgy történik, hogy a feldolgozandó vertex adatokat egy index lista határozza meg. A *mode* paraméter jelentése megegyezik az eddig tárgyaltakkal, míg a *first* és *count* paraméterek ezesetben az index lista kezdő elemének sorszámát, és a feldolgozandó indexek számát jelentik. Mind a vertex adatok, mind pedig az index lista természetesen egy GPU memóriában lévő bufferben vannak tárolva.

```
MultiDrawArrays( mode, first[], count[], primcount )
```

Ezzel a rajzolási paranccsal több egyszerű rajzolási parancsot indukálhatunk egy utasítással. Hatása a következő pszeudokóddal ekvivalens:

```
for i = 0 to primcount-1 do
      DrawArrays( mode, first[i], count[i] )
endfor
```

Fontos megjegyezni, hogy a *first* és *count* paraméterek kliens oldali, azaz a központi memóriában tárolt tömbök, de természetesen maga a vertex adatok továbbra is GPU memóriában vannak tárolva.

```
MultiDrawElements( mode, first[], count[], primcount )
```

Teljesen analóg módon definiáljuk az indexelt rajzolási parancs esetében a fenti szignatúrájú utasítást az alábbi pszeudokóddal:

```
for i = 0 to primcount-1 do
      DrawElements( mode, first[i], count[i] )
endfor
```

Itt is természetesen a *first* és *count* paraméterek kliens oldali tömböket jelölnek.

4.1.1.2 Példányosított rajzolási parancsok

```
DrawArraysInstanced( mode, first, count, instcount )
```

Ez a rajzolási parancs megfelel egy egyszerű rajzolási parancsnak, azzal a különbséggel, hogy a *mode*, *first* és *count* paraméterekkel meghatározott objektumból nem egyet, hanem többet rajzol ki. Ezt a rajzolási módszert példányosított rajzolásnak (instanced draw[E4]) nevezzük és az *instcount* paraméter adja meg a megjelenítendő példányok számát. Az egyes példányok egyedi feldolgozásának érdekében természetesen a grafikus API-k biztosítanak eszközöket példány szintű adatok megadására és ezeknek a shaderekben való elérésére.
Fontos megjegyezni, hogy ellentétben a *MultiDraw* parancsokkal, amelyek valójában csak az API által nyújtott egyszerűsített lehetőségek több rajzolási parancs megadására, a példányosított rajzolás egy GPU funkció, tehát effektíve egy rajzolási parancs váltja ki az összes példány feldolgozását.

```
DrawElementsInstanced( mode, first, count, instcount )
```

Természetesen a példányosított rajzolási parancsnak is van indexelt megfelelője. A gyakorlatban ez utóbbi parancs tekinthető az alapértelmezett GPU parancsnak.

4.1.1.3 Indirekt rajzolási parancsok

A GPU-k legújabb generációja már további rajzolási lehetőségeket is nyújt, melyeknek nagy jelentősége van a GPU alapú vágóalgoritmusok szempontjából. Ezek az új parancsok az indirekt rajzolási parancsok[E5], melyek bizonyos paramétereiket, ellentétben a klasszikus lehetőségekkel, nem a központi memóriából veszik, hanem azok szintén GPU memóriában lévő bufferekből származnak. Ez azért ilyen lényeges, mert amennyiben magukat a paramétereket valamilyen GPU alapú algoritmus határozza meg, azokat nem kell a CPU-nak lekérnie a GPU-tól mielőtt a rajzolási parancsot kiadhatná, így a CPU és a GPU fölösleges várakoztatása kiküszöbölődik.

```
DrawArraysIndirect( mode, offset )
```

A parancs *mode* paramétere megegyezik az eddigiekkel, míg az *offset* paraméter egy GPU memóriában tárolt buffer offszet, ahol a rajzolási parancs valódi paraméterei találhatóak. Ezek a GPU memóriában tárolt paraméterek megegyeznek a példányosított rajzolási parancs *first*, *count* és *instcount* paramétereivel, illetve további paramétereket is meg lehet adni, amelyekkel most nem foglalkozunk.

```
DrawElementsIndirect( mode, offset )
```

Analóg módon rendelkezésünkre áll a parancs indexelt változata. A paraméterek jelentése ugyanaz és itt is van mód bizonyos további paraméterek megadására a GPU bufferben, de ezek a probléma tárgyalásának szempontjából szintén nem lényegesek.

4.1.2 Geometria shader

A programozható GPU-k negyedik generációja több fontos áttörést hozott a GPU alapú vágóalgoritmusok fejlődése szempontjából. Az egyik ilyen hardveres funkció a geometria shader[E6]. Ez a grafikus csővezeték egy új fázisa, melyben egy shader segítségével egész primitíveket (pont, szakasz vagy háromszög) dolgozhatunk fel.

Ami ennél még fontosabb az, hogy a vertex és fragmens shaderek működésével ellentétben a geometria shader nem feltétlen kell egy bemenetre pontosan egy kimenetet generáljon, azaz nem szinkron fázisa a csővezetéknek. A geometria shaderben lehetőségünk van egy bemenet primitívre egy vagy több kimenetet generálni, de ami még fontosabb, dönthetünk úgy is, hogy semmilyen kimenetet nem generálunk. Ez utóbbi tulajdonság lesz a hasznunkra a GPU alapú vágóalgoritmusok implementációjánál.

Fontos azonban megjegyezni, hogy a geometria shaderek ezen asszinkron működése nem jár kompromisszumok nélkül. Mivel a kimenetek száma nem kötött, így szükség van egy ideiglenes bufferre a hardverben, ami limitálja a párhuzamosíthatósági lehetőségeket és ezáltal a teljesítményt. Erre különösképpen figyelni kell a korai hardver implementációk esetében, amelyek sokszor bármilyen párhuzamosítás nélkül futtatták a használt geometria shadereket.

Ez a teljesítménycsökkenés jellemzően akkor okozhat problémát, amikor egy bemenetre relatív sok kimenetet generálunk. Szerencsére a vágóalgoritmusok esetében ez nem okoz problémát, mivel esetünkben két döntés születhet: vagy látható az adott objektum, ekkor generálunk egy kimenetet, amely megegyezik a bemenettel, vagy nem látható az objektum, ekkor azonban nem generálunk kimenetet, azaz eldobjuk az objektumot.

4.1.3 Transform feedback

A programozható GPU-k negyedik generációjának második vívmánya a transform feedback[E7], amely lehetővé teszi, hogy a feldolgozott geometriai adatokat (vertexeket és primitíveket) lementsük egy GPU memóriában lévő kimenet bufferbe. Ennek segítségével ahelyett, hogy raszterizálnánk és végül

megjelenítenénk a feldolgozott primitíveket, egy visszacsatolási lépésben lementjük azokat, hogy egy későbbi rajzolási fázisban azokat felhasználjuk.

Habár eredetileg hagyományos geometriai primitívek feldolgozására hivatott a funkció, semmi sem gátolhat meg bennünket, hogy tetszőleges homogén adathalmaz feldolgozására használjuk ezt az eszközt. Ezt úgy tehetjük meg, hogy az adathalmaz egyes elemeire mint egy vertex adataira tekintünk. Ezt a vertex és vagy geometry shaderek feldolgozzák, majd a transform feedback segítségével visszaírják egy memória bufferbe.

4.1.4 Asszinkron lekérdezések

Ahogy láthattuk az indirekt rajzolási parancsok bemutatásánál, lehetőség van bizonyos GPU által generált adatok közvetlen felhasználására, azonban legtöbbször nem leszünk ilyen szerencsések.

A legtöbb GPU alapú vágóalgoritmusnak szüksége van bizonyos hardver számlálók értékének vizsgálatára, hogy eldöntse egy vagy több objektumról, hogy látható-e vagy hogy egy adott objektumhalmazból hány látható.

A probléma ezeknek a számlálóknak az értékének a vizsgálatával, hogy a CPU-nak le kell kérdeznie őket a GPU-tól. Ez még nem is lenne akkora probléma, ha a CPU nem teljesen függetlenül, asszinkron működne a GPU-val szemben.

Ez pontosabban azt jelenti, hogy amennyiben kiadunk egy rajzolási parancsot és utána le szeretnénk kérdezni a rajzolásból származó számláló értékeket, akkor meg kell várnunk, míg a GPU ténylegesen végrehajtja a rajzolást. Ez jellemzően nem azonnal történik, a parancsok általában bekerülnek egy rajzolási listába a driverben, majd alkalmas időpontban a driver kiadja a parancsot a GPU-nak. Ezután amikor a GPU eljut az adott parancs feldolgozásához, nekilát dolgozni és kis idő múlva előáll az eredmény.

Sajnos a gyakorlatban ez a körbejárási idő akár több képkockányi késést is jelenthet, így sokszor nem engedhetjük meg magunknak, hogy a CPU-n "ücsörögjünk" a programunkkal arra várva, hogy egy asszinkron lekérdezés eredménye rendelkezésre álljon.

Jelenleg ez az egyik legnagyobb problémája a GPU alapú vágóalgoritmusoknak is, mivel sokszor elengedhetetlen, hogy bevárjuk egy adott asszinkron lekérdezés eredményét mielőtt továbblépnénk.

Az asszinkron lekérdezéseknek több típusa is van. A számunkra leglényegesebbek a következők:

- Occlusion query[E8]
- Primitive query[E7]

Az occlusion query-ről már korábban is volt szó. Ez a lekérdezés visszaadja, hogy a vizsgált időszakban hány fragmens ment át a mélység teszten.

A primitive query, vagy primitív lekérdezésnek két típusa van: a generált- és a kiírt primitívek számának lekérdezése. A generált primitívek számának lekérdezése arra ad választ, hogy hány primitívet dolgoztunk fel a vizsgált időszakban, a kiírt primitívek számának lekérdezése pedig azt mondja meg, hogy hány primitív adatait írtuk ki bufferbe transform feedback segítségével.

4.1.5 Atomikus számlálók

A modern GPU-k akár több száz párhuzamos processzormagon tudják a shader programokat futtatni. Ezek csak bizonyos speciális hardver eszközrendszerekkel képesek kommunikálni és jellemzően bármilyen közös erőforrás használata a shaderekben komoly teljesítmény csökkenést eredményez. Egy ilyen kommunikációs eszközt képeznek az atomikus számlálók. Ezek olyan egész értékű számlálók, melyet bármely processzormag olvashat és írhat, az atomicitást speciális hardver biztosítja. Az atomikus számlálókon végezhető műveletek a következők:

- Növelés – a számláló értékét eggyel növeli.
- Csökkentés – a számláló értékét eggyel csökkenti.
- Olvasás – visszaadja a számláló aktuális értékét.

Ellentétben a legtöbb közös erőforráson alapuló szinkronizációs lehetőséggel, az atomikus számlálók rettentő gyorsak, használatuk a legtöbb GPU implementációban nem eredményez bármilyen észlelhető teljesítmény csökkenést. Ez annak köszönhető, hogy dedikált áramkörök felelősek minden egyes számlálóért.

Azonban az, hogy minden egyes számláló külön áramköröket igényel, melyek az összes processzormagban elérhetővé teszik azt, általában a GPU-k csak néhány (8-16) atomikus számlálót támogatnak. Az atomikus számlálók egyelőre csak a Direct3D API-n keresztül érhetőek el és ott is csak limitált formában.

Az atomikus számlálóknak egy nagyon hasznos funkciója az, hogy megadhatunk minden atomikus számlálóhoz egy GPU memória címet. Amikor a GPU befejez egy rajzolási parancsot az atomikus számlálók értékét a hozzájuk megadott GPU memória címre menti, így lehetőségünk van az értékeket további műveletekhez felhasználni.

Ez utóbbi funkciót egyik grafikus API sem támogatja sajnálatos módon, pedig ahogy azt majd a későbbiekben látni fogjuk ez egy nagyon fontos eszköze lehetne a GPU alapú vágóalgoritmusoknak. Szerencsére ez nem marad sokáig így. Az OpenGL ARB hamarosan ki fogja adni az OpenGL szabvány 4.2-es verzióját, mely az ígéretek szerint pótolni fogja ezt a hiányosságot.

4.1.6 Szinkronizált erőforrások

A klasszikus GPU-k kizárólag csak olvasható vagy csak írható adatstruktúrákkal dolgoznak, ez teszi lehetővé azt, hogy a GPU ilyen hatékonyan dolgozzon fel hatalmas mennyiségű adatot, mivel ez által nem lép fel semmilyen "versengés" a közös erőforrásokért. Ennek megfelelően a klasszikus grafikus processzorok bemeneteit képező vertex bufferek és textúrák csak olvashatóak, míg a kimeneteit képező framebufferek csak írhatóak. Ez alól kivétel a mélység teszt, a stencil teszt és a blending, melyeknek szükségük van az adott pixelhez tartozó aktuális értékre, mely a framebufferben van tárolva. A látszat ellenére azonban ezek koherens olvasás-írás problémák, hisz kizárólag csak az adott pixelhez tartozó értékek kerülnek olvasásra és aztán írásra, így az egyes processzormagok továbbra is egymástól függetlenül, zavartalanul tudnak dolgozni.

Habár már rég óta lehetőség van a textúrákba való írásra az által, hogy a textúrát a framebufferhez csatoljuk, ez azzal a megkötéssel járt, hogy amennyiben egy textúrát hozzácsatoljuk a framebufferhez, úgy azt nem használhatjuk olvasásra, legalább is amennyiben meg is próbálnánk ezt, úgy az eredmény nem definiált, hisz nincs lehetőség szinkronizációra, ami biztosítaná az adatok konzisztenciáját.

A programozható GPU-k ötödik generációjával ez megváltozott. Ennek megfelelően az OpenGL is bevezette a szinkronizált erőforrás típusokat az írható és olvasható textúrák és bufferek formájában[E9] (read-write texture, read-write buffer). Ez egyelőre még csak egy kiterjesztésként érhető el, azonban a szabvány következő verziójában már alapfunkcióként fog szerepelni.

Az írható és olvasható erőforrások esetében lehetőségünk van az adatokon atomikus műveletek végzésére és az adatok szinkronizációjára bármely shader programból. Ez szintén algoritmusok egész tárházát nyitotta meg a GPU-k számára. Az egyik lehetséges alkalmazási lehetőség egy a transform feedbackhez hasonló alternatív visszacsatolási módszerre, melynek segítségével bármelyik shader fázisban hozzáadhatunk egy elemet az eddigi listánkhoz. Ezt a veremhez hasonló adatstruktúrát a Direct3D terminológiája append-consume buffernek hívja. Valójában semmi másról nincs szó, mint egy írható-olvasható bufferről amihez hozzáveszünk egy atomikus számlálót, ami úgy fog viselkedni, mint egy verem mutató. Ezt az adatstruktúrát sok újszerű algoritmus alkalmazza, mint például az AMD által publikált Order-Independent Transparency[5] algoritmus, azaz a sorrendfüggetlen átlátszóság, melynek problémakörére mind addig nem született hatékony megoldás a hardver limitációkból adódóan.

Az írható-olvasható erőforrások természetesen messze nem oldják meg a grafikus programozók minden problémáját, mivel azoknak van egy komoly hátránya: a teljesítmény. Ahogy azt már említettük, a GPU-k nem véletlenül használtak eddig csak olvasható vagy csak írható

erőforrásokat. Az atomikus műveletek, valamint a szinkronizációs primitívek hardveres implementációja és alkalmazása már a CPU gyártók számára is komoly fejfáját okoz a több-magos processzor architektúrák esetében, ez pedig halmozottan igaz a GPU-kra, ahol több száz párhuzamosan működő processzormag számára kell összeszinkronizálni az adatokat. Emiatt az írható-olvasható erőforrások akár több, mint egy nagyságrenddel is lassabbak lehetnek a hagyományos erőforrásoknál, így különös tekintettel kell lennünk arra, hogy mikor használjuk őket.

4.2. Motiváció

Az elmúlt évtizedben a grafikus processzorok fejlődése sokkal gyorsabb ütemben nőtt, mint a CPU-ké. Moore törvényét tekintve a CPU-k nagyjából másfél évente duplázták meg elődjeik teljesítményét, ugyanez a GPU-k esetében fél évente történt. Nem véletlen, hogy mára már a legtöbb szuperkomputer rengeteg GPU-val van felszerelve, mivel azok jóval nagyobb számítási teljesítménnyel rendelkeznek, mint a hasonló fogyasztású és hőleadással bíró alternatív processzorok.

Habár mára már aláfagyott valamennyire ez a hihetetlenül gyors ütemű fejlődés, a probléma már itt van a nyakunkon pár éve. Ez a probléma pedig a GPU-k teljesítményének hatékony kihasználása.

A CPU-k és GPU-k közötti teljesítmény különbség pár éve odavezetett, hogy a valós idejű megjelenítésnél a szűk keresztmetszet a CPU teljesítménye lett, mégpedig azért, mert gyakran a CPU egyszerűen nem képes olyan gyorsan adni az utasításokat a GPU-nak, mint ahogy az fel bírja dolgozni azokat. Így sokszor "üresjáratban" megy a grafikus processzor, így a rendszer teljesítménye csökken.

Nagyon sok eszközt vetettek be már mind a gyártók, mind a szoftverfejlesztők, hogy megpróbálják minimalizálni a szükséges CPU-GPU kommunikációt, így született például a példányosított rajzolás ötlete is.

A célkitűzésünk az, hogy a jelenetünk objektumainak láthatóságát és megjelenítését teljes egészében a GPU-n végezzük, ahol annak a helye van. Szerencsére a láthatósági vizsgálatok többsége nagyon jól párhuzamosítható, így jól illik a GPU architektúrájára. A fő probléma inkább maga az vágóalgoritmusok és a rajzolási lépések koordinálása, valamint az, hogy ezt a lehető legkevesebb CPU-GPU kommunikációval hajtsuk végre.

4.3. Példányosított geometria vágása

Ahogy már korábban láthattuk, a GPU-k hatékonyan tudnak egy parancs hatására adott geometriát többszörösen megjeleníteni, ez a példányosított rajzolás. Szintén említettük, hogy a grafikus API-k lehetőséget nyújtanak példányonkénti adatok megadására. Jellemzően ez egy egyszerű tömb, melynek minden eleme (a példány-adatok) egy-egy példány sajátos információit tartalmazza.

Egy példány-adat általában valamilyen transzformációs információt mindenképp tartalmaz, legyen az egy szimpla eltolás vektor vagy egy teljes affin transzformációs mátrix. Nyilván erre azért van szükség, mert nem szeretnénk minden példányt ugyanoda rajzolni. Természetesen a példány-adat tartalmazhat bármilyen további információt is, legyen az a példányhoz tartozó szín, textúra vagy bármi más.

Ebből az egyszerű reprezentációból könnyen adódik, hogy amennyiben példányosított geometriával dolgozunk, a láthatósági vizsgálatnak csak erre a példány-adat tömbre van szüksége mint bemenet és egy hasonló példány-adat tömb lesz a kimenet, mely már csak a látható példányok adatait fogja tartalmazni.

Ahhoz azonban, hogy vágóalgoritmust hajtsunk végre ezen az adathalmazon, szükségünk lesz még a transzformációs információn kívül még az objektum méreteit meghatározó adatokra is. Ez esetünkben legtöbbször a minimális befogadó térfogat primitív lesz.

Habár a tárgyalt algoritmusok a transzformációt és méretet leíró adatok reprezenciójára nem érzékenyek, így bármilyen ismert megközelítés alkalmazható, az egyszerűség kedvéért esetünkben a transzformáció jellemzően csak egy eltolás vektort fog jelenteni, illetve minimális befogadó térfogat primitívnek vagy gömböt használunk, annak sugarának megadásával, vagy pedig egy AABB-t (Axis Aligned Bounding Box), melyet annak három főtengely szerinti méretével fogunk jellemezni.

4.3.2. Instance Cloud Reduction (ICR)

Az Instance Cloud Reduction, vagyis a példány-felhő redukció valójában csak egy hangzatos név, melyet én adtam a technikának amikor először publikáltam a *Nature* nevet viselő demó programomat[E10]. Gyakorlatilag nincs másról szó, mint geometria példányok nézet gúlával való vágásáról, persze mindezt egy GPU alapú implementáció segítségével, amely OpenGL-t használ.

A munkámat pár korábbi hasonló működésű Direct3D alapú technológiai demó előzte meg. Az egyik ilyen demó az AMD által fejlesztett *March of the Froblins*[E11] nevezetű program, melyet Jeremy Shopf, Joshua Barczak, Christopher Oat és Natalya Tatarchuk készített és mutatott be a 2008-

ban rendezett SIGGRAPH konferencián[6]. A másik hasonló munka az NVIDIA Skinned Instancing[E12] nevű példa programja.

Az algoritmus alapötlete, hogy a példányokhoz tartozó adatok alapján azokat egy geometria shader segítségével vágjuk a nézet gúlával és az eredményt egy bufferbe mentjük transform feedback segítségével. A kapott buffert használjuk az eredeti példány-adatokat tartalmazó buffer helyett a megjelenítésnél és a megjelenítendő objektumok számát egy primitív lekérdezés segítségével határozzuk meg. A teljes algoritmus pszeudokódja a következő:

```
Bind( VERTEX_INPUT, instance_buffer )
Bind( TRANSFORM_FEEDBACK_OUTPUT, culled_instance_buffer )
Enable( TRANSFORM_FEEDBACK )
UseShader( cull_shader )
BeginQuery( PRIMITIVES_WRITTEN )
  DrawArrays( POINTS, 0, instance_buffer.size )
EndQuery()
.........
visible_instances = GetQueryResult( PRIMITIVES_WRITTEN )
Bind( INSTANCE_INPUT, culled_instance_buffer )
DrawElementsInstanced( ..., visible_instances )
```

Az aktuális OpenGL implementáció és a használt shaderek megtekinthetőek mind a *Nature*, mind pedig a *Mountains* demó program forráskódjában.

4.3.3. Hierarchical Z-map based occlusion culling (Hi-Z OC)

Ezt a technikát a GPU által hardveresen is támogatott hierarchikus mélység buffer ihlette. Az alapötlet az, hogy hasonló módon hozzunk döntést az objektumok láthatóságáról, mint ahogy azt a hierarchikus Z-Buffer algoritmus teszi a szuper-fragmensek esetében. Az eredeti ötlet és a hozzá tartozó Direct3D implementáció szintén az AMD által fejlesztett *March of the Froblins* demóból származik.

A klasszikusnak számító occlusion query alapú takarásvizsgálattal szemben ezt a technika nem csak azért sokkal hatékonyabb és praktikusabb, mert kihasználhatjuk a példányosított rajzolás adta lehetőségeket. Az occlusion query esetében minden egyes objektum esetében szükségünk van egy asszinkron lekérdezésre, hogy megtudjuk az takarva van-e vagy sem. Ezzel ellentétben a hierarchical Z-map based occlusion culling, azaz a hierarchikus mélység-térkép alapú takarás vizsgálat (röviden Hi-Z OC) az objektum takartságáról a döntést a GPU-n végzi el így legfeljebb egyetlen

asszinkron lekérdezést kell csak végrehajtsunk egy adott példány lista esetében, amikor is meg szeretnénk határozni, hogy a példányok közül mennyi látható.

Ahogy azt az ICR esetében is tettük, a vágás elvégzéséhez a geometria shader azon tulajdonságát fogjuk felhasználni, hogy el tud dobni egy adott primitívet. A geometria shader csak azokat a primitíveket fogja "átengedni", amelyek sikeresen átmennek a láthatósági vizsgálaton, majd az eredményt transform feedback segítségével egy kimeneti bufferbe mentjük.

Maga az algoritmus, mint ahogy az a legtöbb diszkrét takarás vizsgálati algoritmus esetében történik, azzal kezdődik, hogy a valószínűsíthetőleg takaró objektumokat raszterizáljuk, így a mélység bufferben előáll az az információ, ami alapján már tudunk dönteni egy objektum takartságáról. A takarás vizsgálathoz a példányok minimális befogadó térfogat primitívjét fogjuk használni, azonban nem áll szándékunkban ezen primitív minden egyes fragmensét vizsgálni, mivel ez egy szoftveres implementáció esetében rendkívül időigényes folyamat lenne. Itt kerül előtérbe a hierarchikus mélység buffer ötlete.

A takaró objektumok kirajzolása után a mélység bufferben lévő információ alapján először megkonstruáljuk az úgynevezett hierarchikus mélység térképet (Hi-Z map). Ez egy mélység információt tároló mip-map textúra, melynek legfelső szintjén egy a rajzfelület méreteivel megegyező kép van, melybe minden egyes texel a mélység buffer megfelelő pixelének értékét veszi fel. Természetesen ez a másolás megspórolható, ha az alapértelmezett mélység buffer helyett közvetlenül ezt a textúrát használjuk mélység bufferként a takaró objektumok rajzolásakor.

A mélység mip-map textúra további szintjeit az előző szint alapján konstruáljuk meg, mégpedig úgy, hogy az adott szint minden egyes texeléhez az előző szint megfelelő négy texelének vesszük a maximumát. Ezt gyakorlatilag úgy tesszük meg, hogy sorra a következő szintet választjuk mélység buffernek, az előző szintet csatoljuk, mint textúra képet és egy a teljes nézetet lefedő téglalapot rajzolunk. Az algoritmus pszeudokódja ennek megfelelően a következő:

```
Bind( TEXTURE_INPUT, hi_z_map )
num_levels = 1 + floor( log2( max( screen_width, screen_height ) ) )
current_width = screen_width
current_height = screen_height
for i = 1 to num_levels do
  current_width = max( current_width / 2, 1 )
  current_height = max( current_height / 2, 1 )
  SetViewportSize( current_width, current_height )
  Bind( TEXTURE_INPUT, hi_z_map, LEVEL = i-1 )
  Bind( DEPTH_BUFFER, hi_z_map, LEVEL = i )
  DrawFullscreenQuad()
```

```
endfor
```

Maga a forrás textúra olvasásáért, a maximum érték számolásért és az eredmény mélység bufferbe történő írásáért egy speciális fragmens shadert fogunk használni. Ennek a fragmens shadernek az egyszerű változata a következő képpen néz ki GLSL-ben:

```
uniform sampler2D LastMip;
uniform ivec2 LastMipSize;

in vec2 TexCoord;

void main(void)
{
  vec4 texels;
  texels.x = texture( LastMip, TexCoord ).x;
  texels.y = textureOffset( LastMip, TexCoord, ivec2(-1, 0) ).x;
  texels.z = textureOffset( LastMip, TexCoord, ivec2(-1,-1) ).x;
  texels.w = textureOffset( LastMip, TexCoord, ivec2( 0,-1) ).x;

  float maxZ = max( max( texels.x, texels.y ), max( texels.z, texels.w ) );

  gl_FragDepth = maxZ;
}
```

Ez a shader azonban azt feltételezi, hogy a mélység textúránk négyzet alakú és annak mérete kettő valamely hatványa. Amennyiben nem ilyen textúrával dolgozunk, ami legtöbbször igaz, hisz a rajzfelületünk általában téglalap alakú, a mip-mapping alkalmazásánál egy adott szinten egy adott texelhez nem feltétlen pontosan négy texel felel meg az előző szinten. Ilyen esetben a páratlan sorokat vagy oszlopokat, amelyek az API konvenciók alapján a legalsó és legjobboldalibb texelsor, külön le kell kezelni. Így az alsó és jobb széleken hat texel értékének maximumát kell hogy vegyük, sőt a legszélsőségesebb esetben, amennyiben az előző szintnek mind a hosszúsága és szélessége páratlan, összesen kilenc texelnek kell a maximumát vegyük. A *Mountains* demó forráskódjában megtalálható az erre alkalmas shader.

Habár a Hi-Z map megkonstruálása meglehetősen időigényes feladatnak tűnhet, meglepően egy középkategóriás GPU-t használva is bőven egy miliszekundum alatt előállítható. Az általam rendelkezésre álló ATI Radeon 5770-en ez 0,2 miliszekundumot vett igénybe. Ez az idő akár még tovább is csökkenthető egy szintén újkeletű hardver funkció segítségével, mely lehetővé teszi, hogy egyszerre négy texel értékét olvassuk ki, azonban mivel a Hi-Z map megkonstruálása az algoritmus

teljes futási idejének csak nagyon kis részét teszi ki, így erre nem fektettem külön hangsúlyt az implementációban.

Miután megkonstruáltuk a hierarchikus mélység-térképünket, maga a láthatósági vizsgálatot úgy végezzük, hogy az adott objektum minimális befogadó térfogat primitívjének a rajzfelületre vonatkozó helye és mérete alapján kiválaszjuk a Hi-Z map megfelelő szintjét és texeljeit úgy, hogy egy olyan 2x2-es texel halmazt választunk, amelyiknek a rajzfelületre vonatkozó képének mérete minimális, de teljesen lefedi az objektum minimális befogadó térfogat primitívjének képét, majd pedig az objektum minimális befogadó térfogat primitívjének a nézőponttól vett távolságát hasonlítjuk ezeknek a texeleknek az értékéhez. Amennyiben az objektum minimális befogadó térfogat primitívjének a távolsága nagyobb mind a négy texel értékénél, az objektum biztos takarásban van, ellenkező esetben valószínűleg látható.

Mivel az algoritmus minden egyes objektum esetében csak négy texel értékét kell hogy vizsgálja és mivel a döntéshez szükséges összes információ azonnal rendelkezésre állhat számunkra egy tetszőleges shaderben, a hierarchikus mélység-térkép alapú takarás vizsgálat hihetetlenül hatékony alternatíva a klasszikus takarást vizsgáló algoritmusokkal szemben, mely egyszerre és függetlenül alkalmazható teljes objektum listákra egy rajzolási parancs kiadásával, így példányosított objektumok vágására ideális. Természetesen mivel sokkal kevesebb információ alapján döntünk, az eredmény is sokkal pontatlanabb, nem szolgál olyan pontossággal, mint a klasszikus algoritmusok és habár a láthatóság kérdésére sok fals pozitív eredményt adhat, amikor takarást azonosít, azt teljes biztossággal teszi. Mivel napjainkban a valós idejű megjelenítésben hatalmas mennyiségű dinamikus objektumokkal kell megbírkozzanak a megjelenítők, egy ilyen algoritmus, amelyik lineáris időben végez takarás vizsgálatot az objektumokon és még a rendelkezésre álló hardverek architektúráját is jól kiaknázza, az algoritmus nagy jelentőséggel fog bírni a közeljövőben a megjelenítő motorok hatékonyságának növelésében.

4.3.4. Limitációk

Ahogy láthattuk az ICR és Hi-Z OC technikák eredményesen használhatóak példányok láthatóságának vizsgálatára. A két algoritmus ráadásul minkét alapvető láthatósági kérdésre választ ad: a nézet gúlán kívüli objektumok vágását és a nézet gúlán belüli, takart objektumok vágását is képes elvégezni. Jellemzően ezért érdemes mindkét technikát egyidejűleg alkalmazni, első lépésben az ICR algoritmust hajtjuk végre, második lépésként pedig a Hi-Z OC algoritmust, melynek a bemenetét az ICR algoritmus kimenete fogja képezni. Sajnos azonban annak ellenére, hogy az algoritmusok nagyon jól

teljesítenek és példányhalmazok vágására ideálisak, jó néhány API hiányosságból fakadó limitációjuk van, melyekről érdemes pár szót váltanunk.

Az első komoly probléma, hogy az algoritmusok teljesítményét jelenleg nem tudjuk maximalizálni. Ahogy már korábban említettük, habár maga a konkrét eredményhalmazt az algoritmusok képesek egy GPU bufferbe menteni, ahonnan azokat azonnal fel lehet használni egy következő rajzolási lépésben, maga a bufferbe mentett elemek számát jelenleg csak egy asszinkron lekérdezéssel tudjuk meghatározni. Ez azonban a CPU és a GPU "kiéheztetéséhez" vezethet a két feldolgozó egység asszinkron működéséből fakadóan, melyet a 4.1. ábra szemléltet.

Fontos megjegyezni azonban, hogy az asszinkron lekérdezés elméletileg nem szükséges, hisz a mai GPU-k képesek lennének maguk meghatározni és felhasználni a megállapított látható objektumok számát, azonban jelenleg egyik grafikus API sem alkalmas ezeknek a hardver funkcióknak a kihasználására.

Az ideális implementáció az atomikus számlálók és az indirekt rajzolás lehetőségét kiaknázva kiküszöbölhetné az asszinkron lekérdezés szükségességét. Mindezt a következő módon tenné:

- A példányosított geometria rajzolási parancsának paramétereit lementjük egy GPU bufferbe az indirekt rajzolási parancsok által definiált módon.
- Egy atomikus számlálót nullázunk és hozzárendeljük ennek a GPU buffernek azt a címét, ahol az indirekt rajzolási parancs *instcount* paramétere helyezkedik el.
- Lefuttatjuk az ICR vagy Hi-Z OC algoritmusok egy módosított változatát mely azon kívül, hogy dönt az egyes példányok láthatóságáról, minden egyes látható objektum esetében növeli az előzőleg bekonfigurált atomikus számlálót.
- Végül kiadjuk az indirekt rajzolási parancsot a korábban említett GPU buffert használva a paraméterek forrásaként.

Ezt az implementációt azonban a grafikus API-k jelenlegi eszközrendszerével nem tudjuk elkészíteni. Mindez azonban hamarosan lehetségessé válik az OpenGL 4.2-es verziójának hamarosan esedékes megjelenésével.

Az algoritmusok egy másik komoly hiányossága az, hogy csak példányosított objektumok vágására alkalmasak. Ez főként megint csak az API limitációkból fakad, mivel maga a vágóalgoritmusok, ahogy azt a későbbi fejezetekben látni fogjuk, érzéketlenek az objektumok természetére, azonban a generált eredmények felhasználása jelenleg nem kifizetődő tetszőleges objektumok esetében, mivel további asszinkron lekérdezésekre és CPU oldali logikára lenne szükség ahhoz, hogy végül csak a látható objektumokat rajzoljuk ki.

Végül pedig meg kell említeni a Hi-Z OC algoritmus takarás vizsgálatának egy nagyon lényeges megkötését. Ahogy azt az algoritmus ismeretésénél jeleztük, a Hi-Z OC a már megjelenített takaró objektumok mélység képéből állítja elő a hierarchikus mélység-térképet, amelyet majd az algoritmus a példányok láthatósági vizsgálatánál fel fog használni. Ez azonban két komoly problémát vet fel: melyek azok az objektumok, amelyek jól alkalmazhatóak takaró objektumokként és amennyiben ezek megrajzolása a láthatósági vizsgálat előfeltétele, azok láthatóságát hogyan vizsgáljuk?

Az első kérdésre a válasz egy olyan GPU alapú előfeldolgozási lépés lehetne a válasz, mely kiválasztja a takaró tulajdonsággal rendelkező objektumokat. Ezt természetesen több szempont alapján megtehetjük, lehetséges lokális és globális vizsgálatok használata, amennyiben viszont szeretnénk egy egyszerű és hatékony megoldást választani, elsősorban a lokális döntéshozatalra kell törekedjünk. Egy ilyen lokális döntéshozatal jó takaró tulajdonsággal rendelkező objektumokat tud kijelölni a következő szempontok alapján:

- Az objektum távolsága a nézőponttól – amennyiben az objektum közelebb van, úgy valószínűbb, hogy más objektumokat takar.
- Az objektum képének mérete – amennyiben az objektum a rajzfelületen nagyobb területet fed le, úgy valószínűbb, hogy más objektumokat takar.
- Az objektum geometriai komplexitása – mivel leginkább a magas geometriai komplexitású objektumokat szeretnénk vágni, így lehetőleg takaró objektumnak alacsony geometriai komplexitású objektumokat szeretnénk választani.

A második kérdés kicsit tyúk-tojás kérdésnek tűnhet, azonban ha nem is a Hi-Z OC algoritmust alkalmazva, szükségünk van a takaró objektumok vágására is. Erre a legegyszerűbb válasz az ICR algoritmus alkalmazása. Mivel a takaró objektumok kiválasztása azon a feltételezésen alapul, hogy ezek az objektumok majd láthatóak lesznek a végső képen, esetükben nagy valószínűséggel egy takarás vizsgálat fölösleges lenne, így a nézet gúlával vágásnak elegendőnek kell lennie.

4.4. Tetszőleges objektumhalmaz vágása

Az előző fejezetekben megismerhettünk néhány GPU alapú vágóalgoritmust, amelyek segítségével példányosított rajzolás esetében gyorsan és meglehetősen pontosan tudunk a példányok láthatóságáról dönteni.

Azt is láthattuk, hogy maga a láthatósági vizsgálatok bármilyen objektumokra alkalmazható, azonban bizonyos hardveres és API limitációk miatt a bemutatott vágóalgoritmusok nem alkalmasak

tetszőleges objektumhalmazok vágására, legalább is az általuk előállított kimenetek nem megfelelőek a láthatósági vizsgálaton átment objektumok megjelenítése számára.

Ahhoz, hogy a bemutatott algoritmusokat kiterjesszük tetszőleges objektumhalmazokra további tárgyalásra van szükség, amely azt tűzi ki célul, hogy az objektumokat oly módon írja le, hogy az alapján már képesek legyünk olyan kimenetet generálni, amelyet a GPU már közvetlenül értelmezni tud és az adatok alapján megjeleníti a látható objektumokat.

Az alkalmas adatreprezentáció hasonló lesz a példányosított objektumok vágásánál használtával, legalább is abban, hogy az objektumokhoz adatok egy homogén listáját fogjuk rendelni, melynek minden eleme egy adott objektum jellemzőinek leírását fogja tartalmazni. Itt is szükségünk lesz transzformációs információra és szintén alkalmazni fogunk valamilyen minimális befogadó térfogat primitívet.

A megszokott adatokon kívül, amennyiben tetszőleges objektumhalmazzal szeretnénk dolgozni, szükségünk van olyan adatok tárolására is, amelyek az objektumhoz tartozó geometriáról szolgáltatnak információt. Erre a példányosított objektumhalmazok vágásánál nem volt szükség, mivel ott ez implicit adódott, hisz az összes példány ugyanazokat a geometriai adatokat használták. A kérdés már csak az, hogy milyen formában adjuk meg ezt a geometriával kapcsolatos információt.

Már korábban szó volt a rajzolási parancsokról és az indirekt rajzolásról. Mivel a célunk az, hogy az algoritmus olyan kimenetet generáljon, mely alapján már a GPU el tudja végezni az adott objektumok rajzolását. Ennek biztosításához szükségünk lesz a rajzolási parancsoknál használt paraméterekre: első index, indexek száma és a példányok száma. Az egyszerűség kedvéért most példányosított objektumokkal nem foglalkozunk, így gyakorlatilag két további adat tárolásával biztosítottunk a szükséges geometriai információ rendelkezésre állását.

Mindezek alapján az algoritmusaink célja az, hogy a kimenetük minden egyes objektumra egy indirekt rajzolási parancs paraméter listája legyen, amelyeket már csak egy-egy indirekt rajzolási paranccsal ki tudunk rajzolni. Ezzel csak egy probléma van: hiába végeztuk el a láthatósági vizsgálatot a GPU-n még továbbra is szükségünk van arra, hogy az egyes objektumok megjelenítését külön rajzolási paranccsal végezzük. A meglepő az, hogy már létezik olyan GPU, ami erre képes lenne, de mivel sem a Direct3D, sem az OpenGL, sem pedig az általános célú GPU programozási függvény könyvtárak nem biztosítanak ennek a hardver lehetőségnek a kihasználására és ismét egy API limitációba ütköztünk.

4.4.1. Második generációs indirekt rajzolás

Amikor az OpenGL szabvány 4.0-s verziója megjelent, ezzel együtt pedig az indirekt rajzolási parancsok, észrevettem, hogy a funkció hatalmas potenciált rejt magában a GPU alapú vágóalgoritmusokban való alkalmazásra. A szabvány új verziója a programozható GPU-k ötödik generációját hivatott támogatni, melyek közel másfél éve jelentek meg, maga a szabvány pedig alig egy éve. Hamar észrevettem azonban, hogy habár a funkció hatalmas hardveres rugalmasságról tesz tanúbizonyságot, a hardverben valószínűleg sokkal több lehetőség lapul, mint amit a két népszerű grafikus API a rendelkezésünkre bocsát.

Az indirekt rajzolás legnagyobb hiányosságának azt találtam, hogy segítségükkel csak egyetlen rajzolási parancsot lehet csak kiadni, amely a paramétereit egy GPU bufferből veszi. Rögtön az a kérdés merült fel bennem, hogy lenne-e lehetőség paraméterek egész listáját egy körben "beetetni" a GPU-nak, így egyetlen rajzolási paranccsal akár több, teljesen független objektumot is meg tudnánk jeleníteni bármilyen CPU oldali közbeavatkozás nélkül.

Ezek alapján összeállítottam egy ajánlást a második generációs indirekt rajzolási parancsok implementációjára, mely a következő négy új rajzolási parancsot vezetné be:

```
MultiDrawArraysIndirect( mode, offset, stride, primcount )
```

Ez az első parancs, ami több, akár példányosított rajzolási parancsot is kiad, melyeknek a paraméterei mind egy GPU bufferben helyezkednek el egymás után. A *mode* paraméter megegyezik a korábbiakban tárgyaltakkal, az *offset* paraméter, hasonlóan az első generációs indirekt rajzolási parancsokhoz, a paraméterek GPU memóriában lévő címét adja meg, azonban itt nem csak egy paraméter listáról van szó, ezért is a *stride* paraméter, amely azt adja meg, hogy az egyes paraméter listák memória címe között mennyi a különbség. A parancs ekvivalens eredményhez vezet, mint az alábbi pszeudokód:

```
for i = 0 to primcount-1 do
     DrawArraysIndirect( mode, offset + i * stride )
endfor
```

Fontos azonban megjegyezni, hogy ideális esetben a parancs nem egy kliens oldali (CPU) for ciklust fog eredményezni, hanem a GPU fog végig iterálni az egyes rajzolási parancsok paraméterein. Ennek mintájára hasonló módon bevezethetjük az előző parancs indexelt változatát:

```
MultiDrawElementsIndirect( mode, offset, stride, primcount )
```

A paraméterek ez esetben is ugyanazzal a jelentéssel bírnak, annyi különbséggel, hogy a GPU bufferben tárolt rajzolási paraméterek másképp vannak megadva.

Ez a két új parancs ránézésre elégnek tűnik, fontos, hogy megfigyeljük, hogy a kirajzolandó objektumok számát (*primcount* paraméter) még továbbra is a CPU határozza meg. Habár bizonyos esetekben ez praktikus, más esetekben sokkal ideálisabb lenne számunkra ha ezt a paramétert is egy GPU bufferből venné a parancs, például ha az objektumok számát egy GPU alapú vágóalgoritmus határozza meg egy atomikus számláló segítségével. Emiatt szükség van további két rajzolási parancsra:

```
MultiDrawArraysIndirect2( mode, offset, stride, count_offset )
MultiDrawElementsIndirect2( mode, offset, stride, count_offset)
```

A két új parancs megegyezik a korábban bemutatottakkal, eltekintve attól, hogy az objektumok számát nem közvetlenül, hanem egy GPU memória címmel adjuk meg.

Miután a koncepcióval előálltam, felkerestem azzal a két legnagyobb grafikus processzor gyártó céget, az AMD-t és az NVIDIA-t. Míg ez utóbbitól semmilyen pozitív vagy negatív megerősítést nem kaptam arról, hogy a kívánt funkciót támogatná-e a jelenleg piacon lévő hardverük vagy hogy terveznek-e hasonló funkciót implementálni a közeljövőben, az AMD oldaláról volt szerencsém megvitatni a témát Pierre Boudier-vel, aki az AMD OpenGL drivereinek a szoftver architektje. Kiderült, hogy már a piacon lévő hardver generáció is képes elméletben az ajánlott rajzolási parancsok feldolgozására, azonban eddig nem merült fel hasonló funkció hozzáadása az OpenGL szabványba.

Az AMD képviselői meglátták a lehetőséget az ötletben és be is terjesztették azt az OpenGL ARB megbeszélésein, azonban mivel még jó néhány API hiányossággal kell megbírkózzanak a szabványt fejlesztő mérnökök, a funkció megjelenése a szabványban még jó ideig elhúzódhat, leghamarabb az OpenGL 4.3-as verziójában van rá esély hogy bekerül. Figyelembe véve, hogy egyelőre a Khronos Group, az OpenGL ARB-nak otthont adó szabványügyi intézet csak mostanában fogja bejelenteni a szabvány 4.2-es verzióját, a bemutatott API funkcióra még legalább egy évet várnunk kell. Mindaddig a GPU alapú vágóalgoritmusok hatékony implementációja tetszőleges objektumhalmazokra valószínűsíthetőleg csak elméleti lehetőség marad.

4.4.2. Heterogeneous Object Cloud Reduction (HOCR)

A HOCR algoritmus az ICR technika kiterjesztése tetszőleges heterogén objektumhalmazra. Az algoritmus magja gyakorlatilag megegyezik az eredetivel, egyedül az algoritmus bemenete és kimenete változik az előző fejezetekben tárgyaltak alapján.

A HOCR algoritmus bemenete egy objektum lista, azok geometriai leíró adataival, transzformációs mátrixával és minimális befogadó térfogat primitívjeivel. A kimenet ennek megfelelően változik. Minden olyan objektum, amely átmegy a nézet gúlával való vágás tesztjén, a hozzá tartozó leíró adatokat (szín, textúra index, stb.) egy kimeneti GPU bufferbe írjuk, másrészt a rajzoláshoz szükséges adatokat egy másik GPU bufferbe mentjük, melyet majd az indirekt rajzolási parancsok hatására a GPU felolvas és ez alaján jeleníti meg az objektumokat. A jelenlegi eszközrendszerrel az algoritmust a következő pszeudokód adja meg:

```
Bind( VERTEX_INPUT, object_data_buffer )
Bind( TRANSFORM_FEEDBACK_OUTPUT, culled_data_buffer, STREAM = 1 )
Bind( TRANSFORM_FEEDBACK_OUTPUT, indirect_buffer, STREAM = 2 )
Enable( TRANSFORM_FEEDBACK )
UseShader( cull_shader )
BeginQuery( PRIMITIVES_WRITTEN )
  DrawArrays( POINTS, 0, object_data_buffer.size )
EndQuery()
.........
visible_objects = GetQueryResult( PRIMITIVES_WRITTEN )
Bind( INSTANCE_INPUT, culled_data_buffer )
Bind( INDIRECT_BUFFER, indirect_buffer )
for i = 0 to visible_objects-1 do
       DrawElementsIndirect( ... )
endfor
```

Ahogy látható, ebben az esetben két, a teljesítményt erősen befolyásoló probléma van. Egyrészt a látható objektumok számát egy asszinkron lekérdezéssel határozzuk meg, másrészt az objektumokat továbbra is egyenként tudjuk csak megjeleníteni. Emiatt felmerül a kérdés: megéri-e akkor egyáltalán a HOCR algoritmust használnunk a nézet gúlával való vágásra? A válasz pedig az, hogy ilyen formában semmiképp sem.

Egy felől a nézet gúlával való vágás nem túl nagy számítás igényű, így megkérdőjelezhető, hogy nyerünk-e bármilyen érzékelhető mennyiségű időt azáltal, hogy a számítást a GPU-val végeztetjük. Persze, ha kellően nagy mennyiségű adatról van szó, ez az idő akár még lényeges is

lehetne, azonban az objektumokat továbbra is csak egyenként tudjuk megjeleníteni, így aztán a szűk keresztmetszet áttolódik a rajzolási parancsok feldolgozására.

Végül, de nem utolsó sorban a másik problémát az fogja okozni, hogy a megjelenítendő objektumok számát továbbra is egy asszinkron lekérdezéssel kell hogy meghatározzuk, így értékes processzor időt veszíthetünk az eredményre várakozva. Természetesen ezen még javíthatunk egy keveset ha az occlusion query-knél használt predikált rajzoláshoz hasonló funkciót implementálunk, azaz amennyiben az asszinkron lekérdezésnek az eredménye még nem áll rendelkezésünkre, akkor megpróbáljuk kirajzolni a következő objektumot. Ennek persze előfeltétele, hogy a vágóalgoritmus eredményeképpen előálló indirekt buffert inicializáljuk olyan értékekkel, amelyek a lehető legkevesebb ideig fogják fogni a GPU erőforrásait és nem eredményeznek semmilyen látható változást a rajzfelületen (például egy null-elemű geometria rajzolása). Erre azért van szükség, mert az is előfordulhat, hogy több indirekt rajzolási parancsot adunk ki, mint ahány objektum látható, azaz ahány rajzolási paraméter listát ír ki az algoritmus az indirekt bufferbe. Ez akkor fordulhat elő, ha az asszinkron lekérdezés eredménye már csak jóval később áll rendelkezésünkre.

Amennyiben az előző fejezetben ismertetett indirekt rajzolási parancsok rendelkezésünkre állnának, az algoritmus jóval egyszerűbb és hatékonyabb lenne, oly annyira, hogy már indokolttá tenné annak a gyakorlatbani alkalmazását. Az ennek megfelelő javított algoritmus pszeudokódja a következő:

```
Bind( VERTEX_INPUT, object_data_buffer )
Bind( TRANSFORM_FEEDBACK_OUTPUT, culled_data_buffer, STREAM = 1 )
Bind( TRANSFORM_FEEDBACK_OUTPUT, indirect_buffer, STREAM = 2 )
Bind( ATOMIC_COUNTER_BUFFER, counter_buffer )
Enable( TRANSFORM_FEEDBACK )
UseShader( cull_shader )
BeginQuery( PRIMITIVES_WRITTEN )
  DrawArrays( POINTS, 0, object_data_buffer.size )
EndQuery()
Bind( INSTANCE_INPUT, culled_data_buffer )
Bind( INDIRECT_BUFFER, indirect_buffer )
Bind( INDIRECT_PRIMCOUNT_BUFFER, counter_buffer )
MultiDrawElementsIndirect( ... )
```

Az algoritmusnak ez a formája teljesen minimalizálja a felhasznált CPU processzor időt, mivel egyetlen rajzolási paranccsal megjeleníti az összes objektumot és még üresjáratra sem kényszeríti a processzort, amíg az egy asszinkron lekérdezés eredményére várna. Emellett ez utóbbi okból kifolyólág a GPU kihasználtsága is növekedne, mivel nem éheztetnénk ki azt.

4.4.3. A Hi-Z OC kiterjesztése tetszőleges objektumhalmazra

A hierarchikus mélység-térkét alapú takarás vizsgálat is teljesen analóg módon lehet kiterjeszteni tetszőleges objektumhalmazokra, hisz maga a kliens oldali befoglaló kód ugyanaz ebben az esetben is, így ugyanazok az optimalizálási módszerek alkalmazhatóak ebben az esetben is.

Az egyetlen lényegi különbség talán az, hogy míg a HOCR esetében a második generációs indirekt rajzolási parancsok támogatottsága elengedhetetlen annak érdekében, hogy hatékonyan implementáljuk az algoritmust, a Hi-Z OC sok esetben ezen optimalizálások nélkül is jobban teljesíthet, mint egy hasonló megfontoláson alapuló kliens oldali algoritmus. Ez egyszerűen abból adódik, hogy a hierarchikus mélység-térkép megkonstruálását, beleértve maga az eredeti mélység kép előállítását, a CPU valószínűleg számottevően hosszabb idő alatt tudná csak elvégezni, mint a GPU, nem is beszélve a rengeteg extra implementációs munkáról, amit egy ilyen program megalkotása igényelne.

4.5 Heterogén példányosított objektumhalmaz vágása

A további fejezetekben bemutatott algoritmusok alkalmasak példányosított geometria vagy heterogén objektumhalmaz vágására. Annak ellenére, hogy a gyakorlatban az objektumok többsége rendszerint csak egyszer, vagy legfeljebb néhányszor kerül megjelenítésre, vannak bizonyos objektum típusok, amelyekből gyakran több ezer példányt kell megjelenítsünk. Ebbe a kategóriába esik például a részecske megjelenítés vagy a vegetáció.

Habár az előző fejezetben tárgyalt algoritmusok tetszőleges objektumhalmazra alkalmazhatóak, így példányosított geometria esetében is, hisz végül is ezeket is tekinthetjük önálló objektumoknak, a valós idejű megjelenítés esetében nem engedhetjük meg, hogy feláldozzuk a példányosított megjelenítés adta teljesítmény növekedést annak érdekében, hogy egységesen tudjunk kezelni minden objektum típust. Megoldás lehet erre a problémára az, hogy külön kezeljük a példányosított és az egyedi objektumokat az eddig bemutatott algoritmusokkal, azonban ez a következő hátrányokkal jár:

- Nő a szoftver komplexitása, mivel nem egységesen kezeljük az összes objektumot.
- A példányosított objektumok esetében minden példányosított geometriára külön-külön kellene végrehajtanunk az algoritmusokat ami egy bizonyos szint után komoly hatással lehet a teljesítményre.

Ahhoz, hogy ezeket a problémákat kiküszöböljük, szükségünk van egy olyan algoritmusra, amely segítségével egységesen kezelhetjük az egyedi és a példányosított objektumokat, ugyanakkor

megőrizzük mind a példányosított geometria vágására alkalmas, mind pedig a heterogén objektumhalmaz vágására alkalmas algoritmusok előnyeit.

Az objektumaink ábrázolására egy két-szintű rendszert fogunk használni. Az első szint egy lista, melyben az egyes geometriákhoz tartozó adatokat tároljuk, míg a második szint egy olyan lista, melyben az egyes objektumokhoz tartozó adatokat tároljuk. A megfeleltetés egy a geometriát leíró adatokban tárolt mutató. Természetesen feltételezi azt, hogy példányosított geometria esetében az egyes példányok adatai egymás után helyezkednek el az adatstruktúránk második szintjén, ez azonban könnyedén biztosítható. Így kapunk két GPU buffert, mely a megfelelő szint adatait tartalmazza.

Az egyedi objektumok vágására továbbra is elegendő lesz számunkra egy a geometria shaderben meghozott döntés a láthatóságról, azonban szükségünk van valamilyen módszerre, hogy amennyiben egy példányosított geometria adatához érünk, valahogy döntést hozzunk több objektumról. Egy naív implementáció egy egyszerű ciklussal oldaná meg a dolgot, így a geometria shader egyetlen példánya dolgozná fel az adott példányosított geometriához tartozó összes példányt. A probléma ezzel csak az, hogy nem használja ki a GPU adta nagy fokú párhuzamosságot és futási szinkronizációt, így a leghosszabb példánylista könnyedén az algoritmus szűk keresztmetszetévé válna. Szerencsére, GPU-ról lévén szó, rendelkezésünkre áll bedrótozott logikák egész sora, melyek a grafikus csővezeték bizonyos részeit hivatottak kiszolgálni. Esetünkben ezeket teljesen más célra, hierarchikus feldolgozásra fogjuk használni.

Amennyiben példányosított geometriát kell vágjuk, a láthatósággal kapcsolatos döntésfolyamatot és az eredmény bufferbe való írást egy fragmens shaderre fogjuk bízni. A fragmens shader egy-egy példánya a geometria egy-egy példányát fogja feldolgozni. Mindezt úgy tesszük, hogy amennyiben a geometria shaderünk egy példányosított geometriát kell feldolgozzon, helyette kimenetként egy olyan háromszög csíkot ad kimenetként, amely pontosan annyi (vagy legalább annyi) fragmenst fog eredményezni, ahány példányt kell vágnunk. Végül a fragmens shader meghatározza a hozzá tartozó példány sorszámát a fragmens koordinátái alapján és feldolgozza azt.

Mivel az eredményként előálló indirekt buffert mostmár nem csak a geometria shaderben akarjuk szerkeszteni, a transform feedback már nem használható, helyette egy írható-olvasható buffert kell használnunk, mégpedig egy a már korábban is említett append-consume buffert.

Amikor a generálandó háromszög csík méretét szeretnénk megállapítani, fontos figyelembe venni, hogy a GPU-k architektúrájukból adódóan jellemzően valamilyen csempe-szerű struktúrában futtatják a fragmens shadereket. Ez gyártótól és típustól függően lehet 16x16, 32x16, 32x32, 64x32 vagy 64x64 fragmens méretű.

Ezt figyelembe véve érdemes arra törekedni, hogy hasonló méretű téglalapot írjunk le a háromszög csíkkal, mivel amennyiben egy ilyen "csempében" legalább egy fragmenst fel kell dolgozzon a GPU, akkor a "csempe" minden fragmensére lefoglal egy magot. Ezért is egy négyszöget raszterizálunk egy szakasz helyett. Ezt azt is jelenti, hogy nyugodtan "fel-kerekíthetjük" a téglalapunk méretét és egyszerűen ellenőrizzük a fragmens shaderben, hogy az adott fragmenshez tartozó példány sorszám érvényes-e, azaz nem lépte-e túl a feldolgozandó példányok számát. A téglalap használata a GPU architektúrájának hatékony kihasználásán kívül biztosítja a determinisztikus működést is, mivel általa elkerülhetők, vagy legalább is minimalizálhatók a számítási pontatlanságokból illetve a raszterizáló hardverek implementációs különbségeiből adódó hibák.

Maga az objektumokhoz tartozó adatok kiírásához természetesen szükségünk lesz egy második append-consume bufferre, amelybe minden esetben kiírjuk az adott objektumhoz tartozó adatokat. Összesítve a tárgyaltakat az algoritmus szerver oldali (GPU oldali) részének leírása a következőképpen adható meg:

Geometria shader

- Ha a geometriához tartozó példányok száma egy, akkor végrehajtjuk a láthatósági vizsgálatot az objektumon. Ha az objektum látható, akkor növeljük az első append-consume buffer számlálóját és beírjuk a geometriához tartozó rajzolási parancsot a buffer végére (az *instcount* paraméter értéke 1 lesz), aztán növeljük a második append-consume buffer számlálóját is és beírjuk az objektumhoz tartozó adatokat a buffer végére.
- Ha a geometriához tartozó példányok száma nagyobb mint egy, akkor a következőket tesszük:
    - Növeljük az első append-consume buffer számlálóját és beírjuk a geometriához tartozó rajzolási parancsot a buffer végére (az *instcount* paraméter kezdeti értéke 0 lesz).
    - Egy a példányok számának megfelelő méretű téglalapot leíró háromszög csíkot bocsátunk ki és átadjuk a fragmens shadernek az append-consume bufferben lefoglalt rekord indexét.

Fragmens shader

- Meghatározzuk a példány sorszámát a fragmens koordinátái alapján, így meghatározva a példányhoz tartozó adatok helyét a bemeneti bufferben.
- Elvégezzük a láthatósági vizsgálatot az objektumon. Ha az objektum látható, akkor egy atomikus művelettel növeljük annak a kiment buffer azon rekordjának az *instcount* mezőjét, amelyik indexét a geometria shadertől kaptuk, utána pedig növeljük a második append-

consume buffer számlálóját és kiírjuk a példányhoz tartozó objektum adatokat a buffer végére.

Ennek megfelelően a kliens oldali pszeudokód pedig a következőképpen alakul:

```
Bind( VERTEX_INPUT, geometry_data_buffer )
Bind( READ_ONLY_BUFFER, object_data_buffer )
Bind( READ_WRITE_BUFFER, indirect_buffer, INDEX = 1 )
Bind( READ_WRITE_BUFFER, culled_data_buffer, INDEX = 2 )
Bind( ATOMIC_COUNTER_BUFFER, counter_buffer )
UseShader( cull_shader )
DrawArrays( POINTS, 0, geometry_data_buffer.size )
Bind( INSTANCE_INPUT, culled_data_buffer )
Bind( INDIRECT_BUFFER, indirect_buffer )
Bind( INDIRECT_PRIMCOUNT_BUFFER, counter_buffer )
MultiDrawElementsIndirect( ... )
```

Összefoglalás

A diplomamunkában sikerült bemutatnunk azokat az okokat, amelyek indokolttá teszik a láthatósági vizsgálatok GPU-ra történő adaptálását és azt, hogy hogyan járult hozzá ehhez a programozható GPU-k fejlődése. Ismertettük a legfontosabb hardver és API eszközrendszereket, amelyek elérhető közelségbe hozták a GPU alapú jelenetkezelési- és vágóalgoritmusokat. Ezen belül is szó volt arról, hogy az egyes GPU generációk milyen mérföldköveket tettek le erre a fejlődési útvonalra és megpróbáltunk egy jövő képet adni arról, hogy ez a folyamat hova vezethet.

Kiindulási pontnak bevezettük a vágóalgoritmusok egy ad hoc osztályozását, mely mentén bemutattuk a diplomamunka témájául szolgáló GPU alapú vágóalgoritmusokat. Először a klasszikus elsőrendű és magasabb rendű vágóalgoritmusokat mutattuk be, külön kitérve arra, hogy a grafikus hardverek milyen lehetőségeket biztosítanak ezek megvalósításához, majd azt is tárgyaltuk, hogy miért nem elegendőek a CPU alapú- és a finom granularitással végzett GPU támogatott vágóalgoritmusok.

A diplomamunka bemutatott két olyan keret algoritmust, amelyik alkalmas a láthatósági vizsgálatok két legfontosabb problémakörére választ adni. Az első azon objektumok kiszűrésére ad konzervatív választ, amelyek a nézet transzformációk miatt nem lesznek láthatóak a végső képen. A második algoritmus pedig a hardverekben implementált hierarchikus mélység-buffer algoritmusán alapulva egy olyan diszkrét módszerhez juttatott, amelyik lineáris időn belül ad konzervatív választ teljes objektumhalmazok takartságának kérdésére, melyet a hagyományos analítikus módszerek csak jóval nagyobb időkomplexitás árán tudnának csak megválaszolni.

A keret algoritmuson kívül nagy hangsúlyt fektettünk arra, hogy csak is olyan módszerekre koncentráljunk, amelyek a jelenleg adott és a közeljövőben várható hardveres és API eszközrendszerre alapulnak és azok segítségével hatékonyan implementálhatóak a valós idejű megjelenítésbeni alkalmazásra.

Emiatt külön tárgyaltuk azokat az objektum reprezentációkat, amelyek esetében már most is nagy teljesítmény növekedést érhetünk el a gyakorlatban a GPU alapú vágóalgoritmusok alkalmazásával.

Továbbá bemutattunk egy olyan, jelenleg még nem létező API eszközrendszert, amelyet indítványoztam az OpenGL ARB (Architecture Review Board) felé, mint potenciális jövőbeli fejlesztés. Ennek kapcsán arról is beszéltünk, hogy sok esetben a rendelkezésre álló API megoldások azok,

amelyek limitálják az alkalmazási lehetőségeket és hogy több olyan hardver funkció van jelen a piacon lévő GPU-kban, melyeket egyik API sem tett eddig elérhetővé, itt beleértve mind a grafikus API-kat (OpenGL, Direct3D), illetve az általános célú GPU programozási API-kat is (OpenCL, CUDA).

Az ajánlott API eszközrendszer segítségével bemutattunk olyan további algoritmusokat, melyek a láthatósági vizsgálatot tetszőleges heterogén objektumhalmazokra képesek elvégezni a lehető legkisebb CPU oldali közbeavatkozással, így maximalizálva a CPU és a GPU asszinkron működési elve adta lehetőségeket.

Ezen kívül a diplomamunka mellékleteként két implementációt is társítottam, melyek többek közt azokat az algoritmusokat alkalmazzák, amelyeket már a meglévő eszközrendszerekkel is meg lehet valósítani.

Az első demó program a *Nature* nevet kapta és első verzióját 2010 februárjában publikáltam az *Instance culling using geometry shader*[E10] címet kapott cikkemben. A program az ICR algoritmust mutatja be, mely bármely negyedik generációs programozható GPU-n képes futni (Radeon HD2000 series, GeForce 8 series). Az eredeti implementáció OpenGL 3.2 használatával készült, majd egy további, OpenGL 3.3-ra optimalizált változatát is megjelentettem 2010 júniusában[E13].

A második demó program a *Mountains* névre hallgat és 2010 októberében látott napvilágot[E14]. Ez utóbbi több GPU alapú jelenetkezelési- és láthatósági algoritmust is bemutat. Amellett, hogy a program szintén alkalmazza az ICR algoritmust, itt implementáltam először a Hi-Z OC algoritmust, melyről külön cikket is írtam *Hierarchical-Z map based occlusion culling*[E15] címmel. Ezen kívül a *Mountains* demó, kihasználva a vágóalgoritmusok számára rendelkezésre álló információt egy GPU alapú LOD (level-of-detail) rendszert is tartalmaz, melyről bővebben a *GPU based dynamic geometry LOD*[E16] című cikkemben írtam.

Köszönetnyilvánítás

Először szeretnék köszönetet mondani azoknak a családtagoknak, barátoknak, akik lelki támogatást nyújtottak a témábal folytatott kutatásaim folyamán és a diplomamunkához mellékelt programok fejlesztéséhez.

Szintén köszönetemet szeretném nyílvánítani az AMD volt munkatársainak, Jeremy Shopf-nak, Joshua Barczak-nak, Christopher Oat-nak és Natalya Tatarchuk-nak, akiknek a *March of the Froblins* demó program alkotásában nyújtott csodálatos munkája szolgált elsődleges inspirációs forrásként az általam készített programok és ezen diplomamunka megalkotásához.

Ugyancsak megköszönném az AMD OpenGL driver architektjének, Pierre Boudier közreműködését, aki segített tökéletesíteni a *Nature* demó programomat, illetve aki elindította a második generációs indirekt rajzparancsokra benyújtott indítványomat az OpenGL ARB-hoz.

Végül, de nem utolsó sorban pedig szeretném megköszönni a Debreceni Egyetem Komputergrafika és Képfeldolgozás tanszékének dolgozoinak munkáját és közülük is elsősorban témavezetőm, Dr. Schwarcz Tibor egyetemi adjunktus segítségét, akik szakmailag támogatták ezen diplomamunka elkészítését.

Irodalomjegyzék

Könyvek, cikkek, tanulmányok:

[1] Rákos Dániel – A shader nyelvek alkalmazása a valósidejű fotorealisztikus megjelenítésben, Debreceni Egyetem Informatikai Kar

[2] Dr. Schwarcz Tibor – Bevezetés a számítógépi grafikába, mobiDIÁK könyvtár

[3] Ned Greene, Michael Kass, Gavin Miller – Hierarchical Z-Buffer Visibility, SIGGRAPH '93

[4] Evan Hart, John Spitzer – OpenGL Performance Tuning, Game Developers Conference, 2004

[5] Justin Hensley – Order-Independent Transparency in DirectX11, SIGGRAPH 2008

[6] Jeremy Shopf, Joshua Barczak, Christopher Oat, Natalya Tatarchuk – March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU, SIGGRAPH 2008

Elektronikus források:

[E1] Khronos OpenCL Working Group – The OpenCL Specification, Version 1.1, Khronos Group http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[E2] Tom Nuydens – Occlusion Culling http://www.delphi3d.net/articles/viewarticle.php?article=occlusion.htm

[E3] Eric Werness, Pat Brown – NV_conditional_render http://www.opengl.org/registry/specs/NV/conditional_render.txt

[E4] Michael Gold, James Helferty, Daniel Koch – ARB_draw_instanced http://www.opengl.org/registry/specs/ARB/draw_instanced.txt

[E5] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Bruce Merry, Graham Sellers, Greg Roth, Nick Haemel, Pierre Boudier, Piers Daniell – ARB_draw_indirect http://www.opengl.org/registry/specs/ARB/draw_indirect.txt

[E6] Pat Brown, Barthold Lichtenbelt – ARB_geometry_shader4 http://www.opengl.org/registry/specs/ARB/geometry_shader4.txt

[E7] Barthold Lichtenbelt, Pat Brown, Eric Werness – EXT_transform_feedback http://www.opengl.org/registry/specs/EXT/transform_feedback.txt

[E8] Ross Cunniff, Matt Craighead, Daniel Ginsburg, Kevin Lefebvre, Bill Licea-Kane, Nick Triantos – ARB_occlusion_query

http://www.opengl.org/registry/specs/ARB/occlusion_query.txt

[E9] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Eric Werness, Graham Sellers, Greg Roth, Nick Haemel, Pierre Boudier, Piers Daniell – EXT_shader_image_load_store
http://www.opengl.org/registry/specs/EXT/shader_image_load_store.txt

[E10] Rákos Dániel – Instance culling using geometry shaders
http://rastergrid.com/blog/2010/02/instance-culling-using-geometry-shaders/

[E11] Jeremy Shopf, Joshua Barczak, Christopher Oat, Natalya Tatarchuk – March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU, Advances in Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH 2008
http://developer.amd.com/.../Chapter03-SBOT-March_of_The_Froblins.pdf

[E12] NVIDIA Direct3D SDK 10 Code Samples
http://developer.download.nvidia.com/SDK/10/direct3d/samples.html

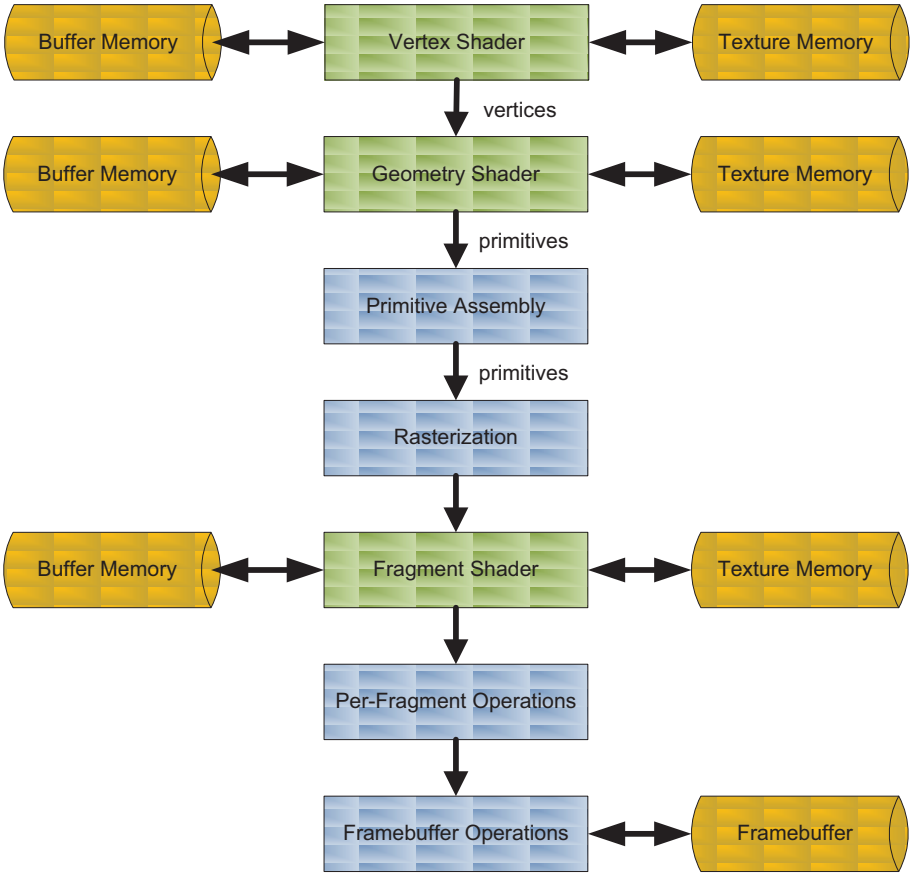
[E13] Rákos Dániel – Instance Cloud Reduction reloaded
http://rastergrid.com/blog/2010/06/instance-cloud-reduction-reloaded/

[E14] Rákos Dániel – OpenGL 4.0 – Mountains demo released
http://rastergrid.com/blog/2010/10/opengl-4-0-mountains-demo-released/

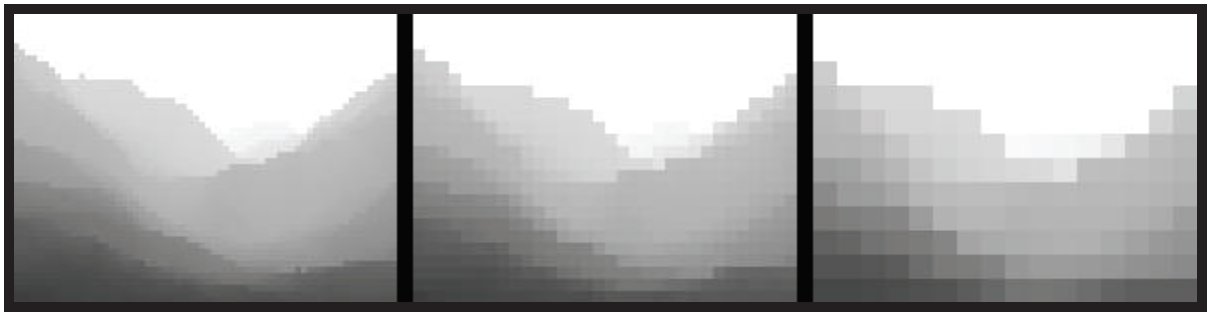
[E15] Rákos Dániel – Hierarchical-Z map based occlusion culling
http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/

[E16] Rákos Dániel – GPU based dynamic geometry LOD
http://rastergrid.com/blog/2010/10/gpu-based-dynamic-geometry-lod/

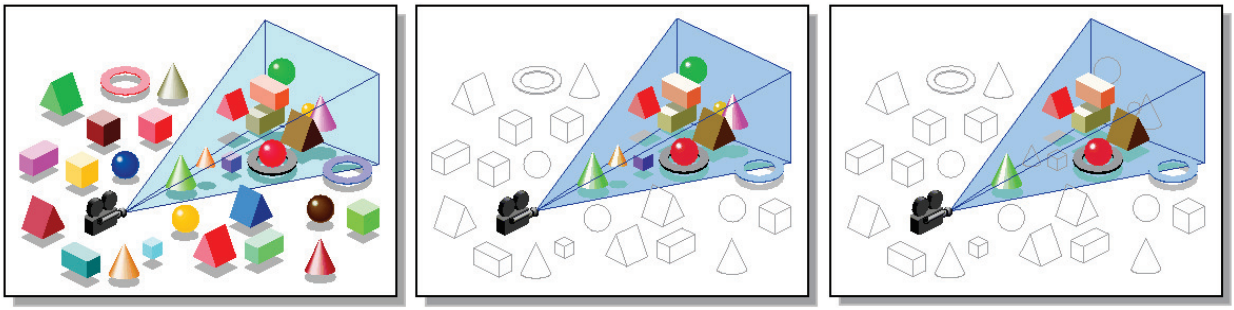
Függelék



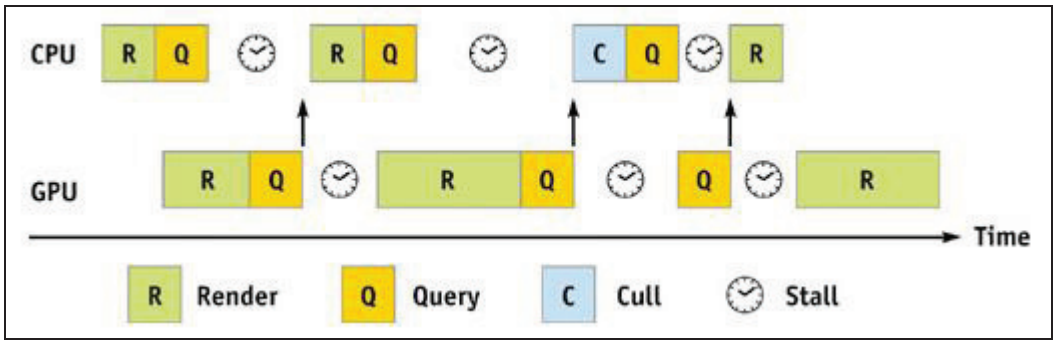
2.1. ábra. A modern grafikus csővezeték sematikus ábrája.



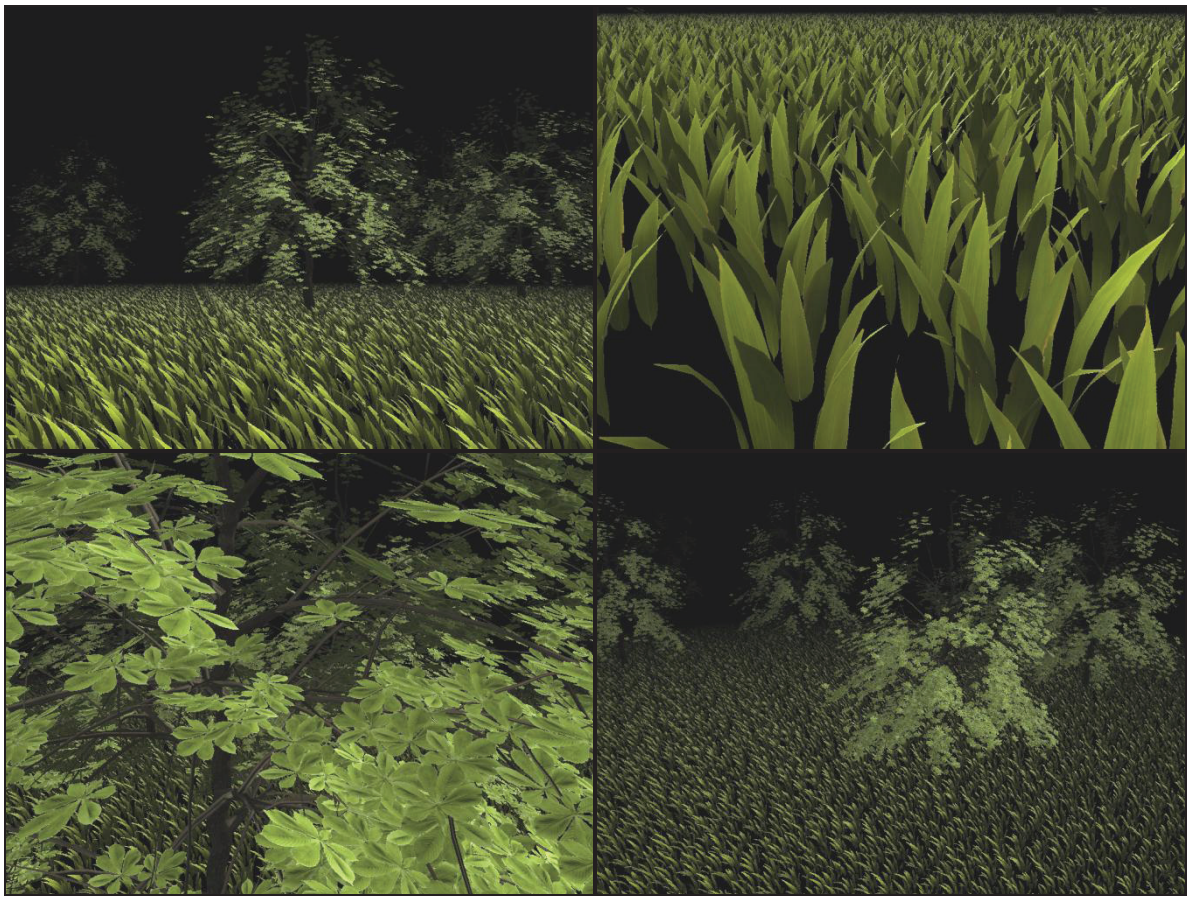
2.2. ábra. A hierarchikus mélység buffer három egymás utáni szintje.



3.1. ábra. A láthatósági vizsgálat két alapproblémájának szemléltetése. Eredeti objektumhalmaz (bal), látható objektumok a nézet gúlával való vágás (közép) és látható objektumok a takarás vizsgálat után.

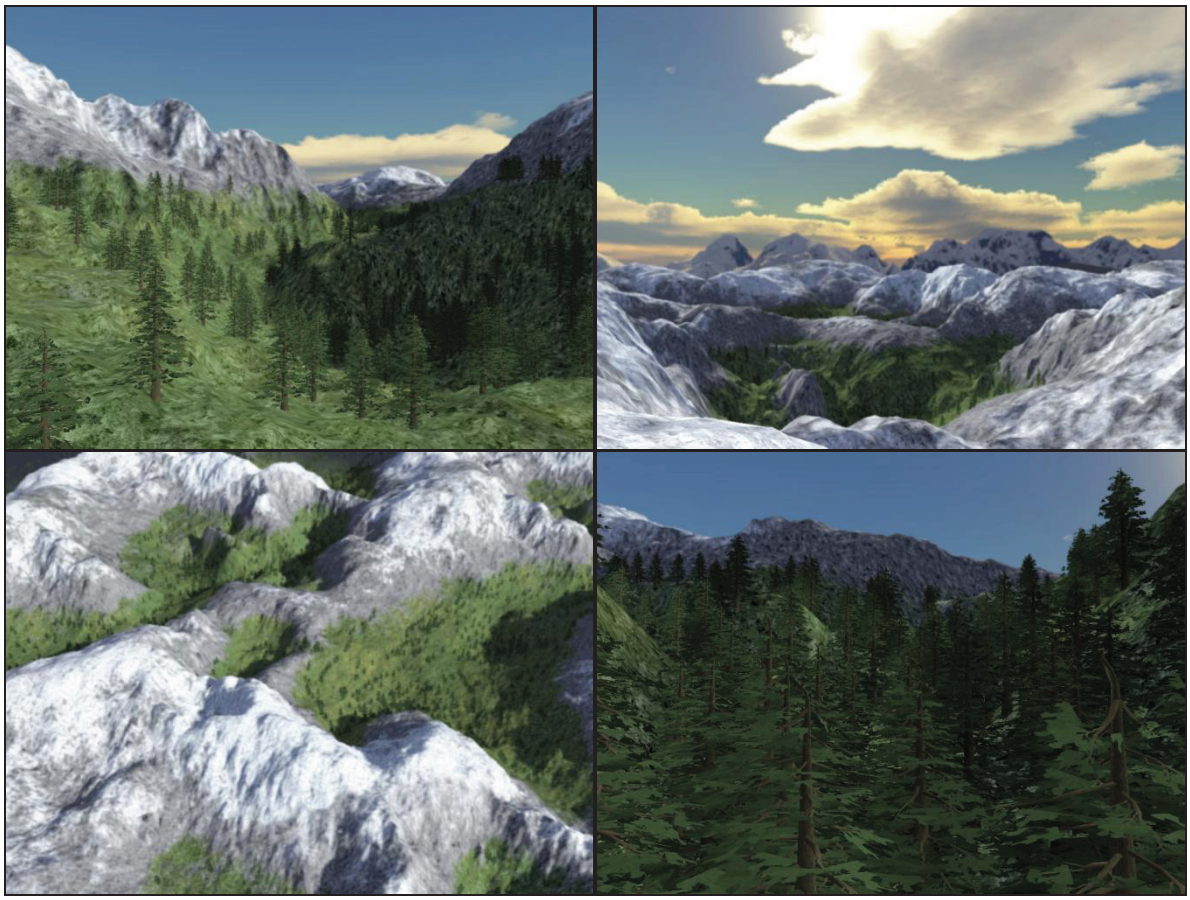


4.1. ábra. Az asszinkron lekérdezések folyamatának szemléltetése, mely során a CPU és a GPU kiéheztetése lép fel.



Képek a *Nature* demó programból.

A megjelenített világ több mint 2.7 milliárd háromszögből áll. Minden egyes fűszál és levél egyenként van háromszögekre bontva. A fa modell több, mint 33 ezer háromszögből áll. Mégis, egy kellően alacsony látótávolságot beállítva az ICR algoritmus mindezt képes 2 millió háromszög alá redukálni.



Képek a *Mountains* demó programból.

A megjelenített világ több mint 400 millió háromszögből épül fel. Az ICR algoritmus segítségével a megjelenítendő geometria mérete nagyjából 50 millió háromszögre csökken. Miután a Hi-Z OC algoritmust is alkalmazzuk a megjelenítendő geometria mérete akár egy millió háromszög alá is csökkenhet, de legfeljebb néhány millió háromszög.