SZAKDOLGOZAT

Rákos Dániel

Debrecen

2008

Debreceni Egyetem
Informatikai Kar

A SHADER NYELVEK ALKALMAZÁSA A VALÓS IDEJŰ FOTOREALISZTIKUS MEGJELENÍTÉSBEN

Témavezető:
Dr. Schwarcz Tibor
egyetemi adjunktus

Készítette:
Rákos Dániel
programtervező informatikus

Debrecen
2008

TARTALOMJEGYZÉK

Bevezetés ... **5**
1. A modern GPU ... **7**
1.1. A számítógépi grafika fejlődése ... 7
1.2. A csővezeték modell ... 9
1.3. Modern API-k ... 11
2. Az OpenGL programozható csővezetéke ... **13**
2.1. Fixed function pipeline ... 13
2.2. Vertex shaderek ... 15
2.3. Geometria shaderek ... 16
2.4. Fragmens shaderek ... 16
2.5. Jövőkép ... 17
3. A vertex feldolgozó csővezeték fejlődése ... **18**
3.1. Klasszikus modell ... 18
3.2. A hardware T&L megjelenése ... 18
3.3. Vertex shaderek a CPU-n ... 19
3.4. Vertex shaderek a GPU-n ... 19
4. A raszterizálás fejlődése ... **21**
4.1. Klasszikus modell ... 21
4.2. A multitextúrázás megjelenése ... 21
4.3. A texture combiner mechanizmus ... 22
4.4. Register combinerek és texture shaderek ... 23
4.5. Fragmens shaderek ... 24
4.5.1. Első generáció ... 24
4.5.2. Szabványosított fragmens shader ... 25
5. Alkalmazási területek ... **27**
5.1. Fotorealisztikus megjelenítés ... 27
5.2. Nonfotorealisztikus megjelenítés ... 28
5.3. Általános célú GPU programok ... 29
6. Megvilágítási modellek ... **31**
6.1. Alapfogalmak ... 31
6.2. Lambert törvénye ... 32
6.3. Kétirányú visszaverődés eloszlási függvény ... 32
6.4. Árnyalási módszerek ... 33
6.4.1. Flat shading ... 33
6.4.2. Gouraud shading ... 33
6.4.3. Phong shading ... 33
6.5. Diffúz anyagok ... 34
6.5.1. Lambert modell ... 34
6.5.2. Oren-Nayar modell ... 35
6.5.3. Minnaert modell ... 36
6.6. Tükörszerű anyagok ... 37

6.6.1. Phong modell ... 37
6.6.2. Blinn-Phong modell ... 38
6.6.3. Schlick modell ... 40
6.6.4 Ward modell ... 40
7. Megjelenítési technikák ... 43
7.1. Bump mapping ... 44
7.1.1. Emboss bump mapping ... 44
7.1.2. Environment-mapped bump mapping ... 45
7.1.3. Normal mapping ... 46
7.1.4. Parallax mapping ... 46
7.1.5. Parallax occlusion mapping ... 47
7.2. Árnyékok megjelenítése ... 49
7.2.1. Planáris árnyék vetítés ... 49
7.2.2. Shadow volumes ... 50
7.2.3. Shadow mapping ... 51
7.3. Utófeldolgozási technikák ... 53
7.3.1. Motion blur ... 53
7.3.2. Depth-of-Field ... 53
7.3.3. Bloom ... 54
8. Hatékonysági problémák és potenciális megoldások ... 56
8.1. Szűk keresztmetszet ... 56
8.1.1. A vertex feldolgozó csővezeték túlterhelése ... 56
8.1.2. A fragmens feldolgozó csővezeték túlterhelése ... 56
8.2. Mélység buffer optimalizálás ... 57
8.3. Occlusion culling ... 59
Összefoglalás ... 60
Irodalomjegyzék ... 62
Függelék ... 64

Bevezetés

Már több mint nyolc éve foglalkozok számítógépi grafikával, illetve ezen belül is nagyjából hat éve mélyültem el először a háromdimenziós grafika világában, ezen belül is az OpenGL használatában. Emiatt döntöttem úgy, hogy szakdolgozatom témáját az informatika ezen ágazatából választom.

A háromdimenziós számítógépi grafikával mindenki találkozott bizonyos formában. Vehetjük példának a témakörhöz legszorosabban kötődő népszerű iparágat, a videojáték ipart, vagy beszélhetünk akár a filmiparról vagy a szakterület tudományos célú felhasználásáról. A komputergrafika azért fogott meg a kezdetektől fogva, mivel ez a programozásnak tipikusan egy olyan területe, melyet egy laikus is értékelni tud annak ellenére, hogy általában nem lát a végeredmény mögé, nincs szükség arra, hogy a szemlélő jobban beleássa magát a témába ahhoz, hogy megállapíthassa, hogy tetszik neki amit lát avagy nem.

A szakterület mára már annyira kinőtte magát, hogy nehéz egy konkrét területet kivenni belőle anélkül, hogy valami lényegeset ki ne felejtsük. A választásom a shader nyelvek alkalmazására esett, mivel hatalmas mérföldkövet jelentenek a valós idejű grafikus szoftverfejlesztés számára, de ahogy majd a későbbiekben láthatjuk, sok olyan témáról is szó lesz, melyek közvetlenül nem kapcsolódnak a shader nyelvekhez, azonban szorosan összefonódnak azok gyakorlati alkalmazásával.

Az első fejezetben egy kis történelmi áttekintéssel szeretném kezdeni, kezdve a számítógépi grafika első lépéseitől egészen a modern háromdimenziós grafika fellegkoráig. Szó lesz a tudományág fejlődésének legfontosabb mérföldköveiről, illetve azokról a szervezetekről illetve személyekről, akik ezekhez az evolúciós lépésekhez nagyban hozzájárultak.

A második fejezet egy fajta betekintést nyújt egy konkrét háromdimenziós grafikai függvénykönyvtár működésébe. A választásom az OpenGL API-ra esett, mivel egy nyílt szabványról van szó, melyet a professzionális munkaállomásoktól a személyi számítógépek összes típusán keresztül egészen a mobiltelefonokig szinte minden platformon lehetőségünk van használni. A fejezet az OpenGL grafikus csővezetékének lépéseit mutatja be.

A harmadik és a negyedik fejezet az előzőleg bemutatott grafikus csővezeték egyes szakaszain végbemenő fejlődéseket taglalják, melyek végül a valós idejű komputergrafikában használt shader nyelvek kialakulásához vezetett.

Ezek után az ötödik fejezet a shader nyelvek alkalmazási területeit mutatják be, többek között a fotorealisztikus megjelenítést, ami a szakdolgozat témája, de emellett szó lesz a non-fotorealisztikus megjelenítésről, illetve a shader nyelvek általános célú használatáról is.

A hatodik fejezet keretein belül már a fotorealisztikus megjelenítésre koncentrálunk. Ismertetem az egyes árnyalási módokat, illetve szó lesz a legnevezetesebb megvilágítási módokról, melyeknek az implementációját is bemutatom majd.

A hetedik fejezet már haladó technikák ismertetéséről szól. Ezek közül lesz olyan, ami szorosan kapcsolódik a shaderek masszív alkalmazásával, azonban olyanok is, melyek nem szervesen kötődnek a témához, viszont feltétlen szükségesek egy minőségi fotorealisztikus grafikus alkalmazáshoz. Itt többek között szó lesz arról, hogy hogyan jelenítsünk meg árnyékokat, illetve hogy bizonyos optikai jelenségeket miképp lehet számítógépen valós időben szimulálni.

Az utolsó fejezetben azokat a kérdésekről, technológiákról és technikákról lesz szó, melyeket ismerni ahhoz, hogy a megjelenítést hatékonyan végezzük, a szoftver mögött rejlő hardver adottságait a megfelelő módon használjuk ki.

A szakdolgozat célja egyrészt hogy bemutassa azokat az eszközöket, melyek mára már elengedhetetlen kellékei a valós idejű fotorealisztikus megjelenítésnek. Szintén be szeretném mutatni a múltban, aktuálisan és a jövőben használatos technológiákat, megvilágítási technikákat, melyek az optikai jelenségeket imitálják, legyenek azok akár fizikailag pontos, elméleti alapokkal rendelkező

módszerek vagy akár ad hoc módon kitalált eljárások, melyekkel szemet gyönyörködtető, első ránézésre hihető jeleneteket produkálhatunk. Alapvetően a témához fűződő szakirodalom átdolgozása a cél, hogy bárki egy szempillantás alatt betekintést tudjon nyerni a háromdimenziós grafika szakterületébe, mindezt minimális előismeretek alapján is.

Az egyes gyakorlati módszereket igyekszem majd a hardver implementációk szemszögéből megközelíteni, mivel a számítógépi grafika ezen ágazatánál feltétlen szükséges ennek a nézőpontnak az elsajátítása.

A témaválasztásom másik oka az, hogy magyar nyelven nagyon kevés napra kész szakirodalmi alkotás lelhető fel, ami ezzel a témakörrel foglalkozik. Ennek többek között az is az oka, hogy a számítógépi grafika ezen ágazatával kevés hazai oktatási és ipari szervezet, intézmény foglalkozik. A célkitűzéseim között tehát azt is megjegyezném, hogy szeretném ha a szakdolgozatom által többen megismerkedhetnének ezzel a csodálatos tudományággal, melyre a megfelelő nyelvtudás híján egyeseknek nem volt lehetőségük találkozni.

1. A modern GPU

Ebben a fejezetben a számítógépi grafika bizonyos fejlődési lépéseit fogjuk megismerni, amelyek a modern grafikus feldolgozó egység (GPU) implementációjához vezettek. A mai GPU-kban sok lehetőség rejlik, mivel a fejlődési sebessége messze meghaladja a CPU-k fejlődését. Az elmúlt évek azt mutatják, hogy a GPU-kban lévő tranzisztorok száma hat havonta duplázódik, míg ugyanezt a növekedést a CPU-k tizennyolc hónap alatt érik el. Manapság már egyre gyakrabban használják a GPU-t ezen okok miatt általános célú számításokra is, mivel azonban a GPU elsősorban egy adatfolyam processzor, a felhasználási terület is valamivel kisebb, mint a CPU esetében.

1999-ben készült el a 3D grafikus csővezeték első hardveres implementációja. Azelőtt bizonyos részei, vagy pedig a teljes csővezeték szoftveresen volt megvalósítva, így minden számítás a CPU-n került végrehajtásra. Abban az időben ilyen nehezen hozzáférhető hardverre és specifikus szoftverre volt szükség, ha valaki jobb grafikát szeretett volna, mint ami a piacon fellelhető volt. Ahogy egyre inkább elterjedtek az alacsony árú személyi számítógépek, az igény a jobb számítógépes játékok iránt kiváltotta a hozzáférhető árú 3D grafikus kártyák megjelenését, ami odavezetett, hogy mára 45.000, Ft-ért vehet valaki egy legfelső kategóriás videokártyát, amelyben 320 adatfolyam processzor van, és ezzel messze felül múlja egy felső kategóriás személyi számítógép teljesítményét, ami pedig akár több, mint a tízszeresébe kerülhet.

Hiába azonban a GPU hatalmas számítási képessége, mivel olyan számításokra vannak optimalizálva, amelyek a számítógépi grafika esetében szükségesek. Ez azt jelenti, hogy masszív párhuzamos rendszer, amely tipikusan adott bemenetekre determinisztikusan kimeneteket készít, mindezt hatalmas mennyiségben, azonban nem nagyon van lehetőség adat újrahasznosításra. Hardveres oldalból nézve ez azért van, mert a CPU-k nagy méretű gyorsítótárakkal rendelkeznek, amelyek szándékosan úgy lettek elkészítve, hogy mindenféle használati módot nagyjából azonos mértékben támogasson, ezzel ellentétben a GPU-knak sokkal kisebb a gyorsítótáruk és ezeknek sokkal specifikusabb a funkciójuk, hisz annak ellenére, hogy mára már szinte minden megvalósítható a GPU segítségével is, a GPU elsődleges célja mégis a grafikában használatos számítások elvégzése.

1.1. A számítógépi grafika fejlődése

A „számítógépi grafika” kifejezés megjelenése egy Boeing grafikai tervezőnek köszönhető, aki készített egy számítógép által generált ortogonális nézetet az emberi testről és az elsők között készített háromdimenziós számítógépes animációt. Ez 1960-ban történt és azóta rengeteg technológiai és elméleti fejlődésen ment keresztül a számítógépi grafika, míg elérte az aktuális állását.

A digitális számítógépek fejlődésének egy fontos mérföldköve a MIT (Massachusetts Institute of Technology, egy Cambridge-i magánegyetem) által kifejlesztett légvédelmi rendszer volt, a Whirlwind, majd pedig ennek tovább fejlesztett változata, a SAGE. A fejlesztést Jay Forrester és Ken Olsen kezdte 1944-ben. Az alapötlet egy olyan programozható repülési szimulációs számítógép volt, amely alkalmas lesz a tengerész pilóták kiképzésére anélkül, hogy különböző számítógépeket tervezzenek az egyes repülőgép típusokhoz. A Whirlwind nem az első digitális számítógép volt, azonban az első olyan számítógép, amelyik alkalmas volt interaktív, valós idejű irányításra, illetve valós idejű szöveg és grafika megjelenítésére. Mivel akkoriban a memóriák nem voltak elég gyorsak, hogy lehetővé tegyék a Whirlwind számára, hogy egy igazi valós idejű rendszer lehessen, Jay Forrester egy új memória típust, amit mag memóriának (core memory) nevezett el. Habár a projekt eredetileg egy relülés szimulátornak indult, hamar átalakult egy általános célú, valós idejű digitális számítógép tervezésébe. A légierő sok lehetőséget látott a dologban, így támogatta egy új projekt

indítását. Ez volt a SAGE (Semi-Automatic Ground Environment). 1958-ban elkészült az első teljesen működőképes irányító központ, amelyet a Whirlwind számítógép vezérelt.

A Whirlwind és utódja, a Whirlwind II még mindig elektroncsöveket használtak, azonban a tranzisztor feltalálása után hamar alkalmazták is azt, mivel ezeknek jóval alacsonyabb helyigénye volt és kevesebb hőt produkált, így a hűtés is egyszerűbb volt. A TX-0 (Transistored Experimental Computer Zero) volt az első valós idejű, programozható, általános célú számítógép, amely kizárólag csak tranzisztorokat használt. A TX-0 gyakorlatilag a Whirlwind tranzisztorokkal elkészített változata volt. A Whirlwind egy nagy épület teljes szintjét elfoglalta, míg a TX-0 befért egyetlen szobába és valamivel gyorsabb is volt. A TX-0 továbbfejlesztett változata volt a TX-2 (1956). Ez a két számítógép lényeges fejlődési lépés volt a számítógépi grafika számára, mivel a projekt nagy hangsúlyt fektetett a CRT és a fényceruza használatára, ezzel egy interaktív grafikus számítógépet alkotva, ami a számítógépi grafika egy új szintjét jelentette. A TX-0/TX-2-t használta Ivan Sutherland is, hogy elkészítse interaktív grafikus programját, a Sketchpad-ot. Ezt követően az iparág a korszak lehetőségeihez képest hatalmas fejlődésnek indult. Ennek eredménye többek között a CAD (Computer Aided Design) rendszerek is, amelyek szintén visszavezethetőek Sutherland munkásságára.

A számítógépi grafika modern korának gyökerei az 1980-as évek elejére tehetőek. 1981-ben az IBM elkészítette az első videokártyát PC-k számára. Ez a monokróm videokártya kizárólag szöveg megjelenítésére volt alkalmas, nem tudott címezni egyetlen pixelt, mindig egy 9x14-es méretű pixel tartományt tudott csak egyszerre módosítani. Habár ezek a kártyák nagyon korlátozott képességekkel voltak ellátva, akkoriban tökéletesen megfeleltek a szintén korlátozott képességű PC-k számára. Ahogy a videokártyák fejlődtek egyre magasabb felbontásokat támogattak, egyre több színt, illetve azt a képességet is támogatták, hogy minden egyes pixelt külön címezni tudtunk. Ennek ellenére még mindig a CPU-ra hagyatkoztak minden számítás esetében, így aztán a videokártyák fejlődése csak növelte a CPU-kra nehezedő munkaterhelést.

Az IBM 1984-ben megalkotta az első PC-k számára készített processzor vezérelt videokártyát. Ez volt a PGA (Professional Graphics Adapter), amely egy Intel 8088 mikroprocesszort hordozott, ennek segítségével pedig minden grafikával kapcsolatos feladatot elvégzett a CPU helyett, a CPU feladata egyedül a rajzolási parancsok kiadása volt. Ez egy nagyon lényeges lépés volt a grafikus feldolgozó egység (GPU) fejlődésében, hisz az első kivitelezett ötlet volt, amely azon alapult, hogy elkülönítsék a grafikai számításokat minden mástól. A PGA videokártya három szlotot foglalt el és egy speciális monitort szükségelt, ami több, mint 4.000 dollárba került. A PGA emiatt rövid életű volt. Az első videokártya, ami képes volt 60 FPS (Frame Per Second) sebességgel rajzolni 3D animációkat.

Ahogy megjelent 1983-ban a VisiOn, az első grafikus felhasználói környezet (GUI) PC-re, egyre nagyobb számítási teljesítményre volt szükség. Új szabványokra, irányvonalakra és fogalmakra volt szükség a számítógépi grafikában, így a Silicon Graphics Inc. (SGI) megtette a következő lépést a GPU fejlődésében. Az SGI egy számítógépi grafikai hardver gyártó cég volt, amely arra fektette a hangsúlyt, hogy a lehető legnagyobb grafikai teljesítményű számítógépeket készítse. Ezekkel a számítógépekkel párhuzamosan olyan szabványokat is készítettek, amelyek az alapját képezik a mai komputergrafika hardver és szoftver trendnek. Az egyik mai szoftver szabvány a platformfüggetlen grafikai API (Application Programming Interface), az OpenGL. A szaványt 1989-ben jelent meg és mind máig az ipar által legszélesebb körben alkalmazott és támogatott 2D és 3D API. Egy szintén nagyon lényeges fogalmat is bevezetett az SGI a számítógépi grafikába, ami mára a grafikus hardverek tervezésének egyik leglényegesebb pontja, ez pedig nem más, mint a grafikus csővezeték (graphics pipeline). Mára már ezt a megközelítést szinte minden GPU gyártó alkalmazza a tervezésnél, mint például az ATI, NVIDIA, stb. A legaktuálisabb lendítőerőt a GPU-k tervezésében és megvalósításában mára a háromdimenziós PC játékok szolgáltatják, mint például a Quake, Doom, és még sok más. Ezek

a játékok olyan szinten ösztönözték a nagy teljesítményű GPU-k betörését a PC piacra, hogy mára egyre több területen alkalmazzák a 3D grafikát. Ennek köszönhetően néhány felső kategóriás szuperkomputert mára már PC-k helyettesítenek. Habár mind a PC grafika, mind a nagyipari grafika hatalmas feljődést mutatott a hardver tervezés terén, hozzá kell tenni, hogy ezen két iparág más-más célkitűzés tükrében végzi tevékenységét. Az offline megjelenítő rendszerek, mint például a CAD alkalmazások által használtak, elsősorban a pontosságra fektetik a hangsúlyt, nem pedig az eredmény gyors elérésére, míg a valós idejű megjelenítő rendszerek, mint például a játék motorok és a szimulátorok, elsősorban a nagy képrajzolási sebességre fektetik a hangsúlyt, hogy folyamatos animációkat produkáljanak és nem ritkán áldozzák fel a geometriai és textúra minőséget, hogy mindezt megvalósítsák.

Ahogy már korábban is említettük, mára már a GPU-t nem csak 3D grafikai feladatok elvégzésére használják. Ahogy a GPU-k fejlődtek, sokkal komplexebbé váltak, mint egy általános célú CPU. Ha például megnézzük a tranzisztorok számát az aktuális videokártyák esetében, láthatjuk, hogy például az ATI Radeon HD 3800 sorozatának GPU-ja 320 adatfolyamprocesszort tartalmaz, összesen 666 millió tranzisztorral, mellyel több, mint egy terraFLOPS (Floating-Point Operations Per Second) számítási teljesítményt produkál, míg a négy magos Intel Core 2 Quad processzorban is csak 582 millió tranzisztor van és csak 9.8 gigaFLOPS lebegő pontos számítási teljesítményre képes.

Moore törvénye kimondja, hogy az egységi felületre elhelyezhető tranzisztorok száma duplázódik minden 12 hónapban, azonban azóta ez a sebesség lelassult 18 hónapra, ami gyakorlatilag azt jelenti, hogy a processzorok teljesítménye 18 havonta duplázódik. Az elmúlt 8-10 évben a GPU gyártók megcáfolták Moore törvényét azzal, hogy a tranzisztorszám duplázódás sebességét 6 hónapra csökkentették, ami azt jelenti, hogy a GPU-k teljesítmény növekedése Moore törvényében megfogalmazott mérték négyzetével jellemezhető. Ennek a hihetetlen növekedésnek köszönhetően vetődött fel az az ötlet, hogy a GPU-t ne csak grafikai számításokra használják.

Klasszikus értelemben a CPU egy egyszálas számítási architektúrát valósít meg, amely lehetővé teszi több folyamat időosztásos futtatását ezen az egyszálas csővezetéken, melyek az adataikat egyetlen memória interfészen keresztül érik el. Habár az elmúlt néhány évben megjelentek a több magos processzorok is a PC piacon, ezek nem mások, mint szorosan egymásba ágyazott több processzoros rendszerek. Ezzel szemben a GPU-k teljesen más architektúrát követnek, nevezetesen az adatfolyam feldolgozást (stream processing). Ez a megközelítési mód sokkal hatékonyabb nagy mennyiségű adatok feldolgozására, ami általában az elsődleges feladata a grafikus processzornak. Egy GPU felépítésében akár több ezer ilyen adatfolyam processzor is össze lehet kötve, aminek köszönhetően egy dedikált csővezeték processzort (dedicated pipeline processor) kapunk. A CPU-val ellentétben, mivel itt az adatfolyam processzorok egy csővezetéket alkotnak, nincsenek ütközések és várakozások. Az adatfolyam processzor modell esetében minden egyes tranzisztor folyamatosan dolgozik. Mindezek alapján felmerülhet bennünk a kérdés, hogy lehetséges-e, hogy a GPU átveszi a CPU szerepét, mint a PC elsődleges processzora.

1.2. A csővezeték modell

A grafikus csővezeték (graphics pipeline) feldolgozási szakaszok egy elméleti modellje, amelyen keresztül küldjük a grafikai adatokat, hogy megkapjuk a várt eredményt. Ezt a csővezetéket megvalósíthatjuk szoftveresen (ami jelen van például az OpenGL és a Microsoft Direct3D API-jában is) vagy hardveresen a GPU-ban, illetve ezen kettő kombinációjaként. Ezek a feldolgozási szakaszok gyakorlatilag nem mások, mint adatátalakítási folyamatok: térbeli koordinátákkal kezdünk és rasztergrafikus képet kapunk eredményképpen. Természetesen ez nem ilyen egyszerű, a valóságban ennél sokkal összetettebb dolgok zaljanak le, nem hiába a csővezeték modellnek is születtek

különböző változatai, hisz a GPU-k fejlődésének iránya is nagyban befolyásolja, hogy mely feladatokat kell szétbontani, melyeket pedig összevonni.

A csővezeték modell legegyszerűbb változata az úgynevezett 2-szakaszos csővezeték (2-stage graphics pipeline). Ezt a modellt a következő lépések alkotják:

- Geometria feldolgozó szakasz (geometry stage): bemenetei a háromdimenziós objektumok pontjainak (vertex) koordinátái, kimenete pedig az egyes két dimenziós primitívek.
- Megjelenítő szakasz (rendering stage): bemenetei a két dímenziós primitívek, kimenete pedig a monitoron megjeleníthető rasztergrafikus kép.

Ez a lehető legvázlatosabb grafikus csővezeték modell. Bizonyos modellekben a geometriai feldolgozó szakaszt további két szakaszra szokták bontani:

- Transform and Lighting (T&L): ezen szakasz felelős a ponttranszformációk végrehajtásáért és az egyes pontok (vertex) színeinek meghatározásáért, ami történhet bonyolult fénytani számítások alapján.
- Triangle Assembly: a keletkezett pontokból (vertex) valamilyen konvenció alapján háromszögeket gyártunk, melyek csúcsainak színét már korábban meghatároztuk. Ennek alapján elő tudjuk állítani a kirajzolandó két dimenziós primitíveket, amiket a megjelenítő szakasz fogadni tud.

Akár szoftveresen, akár hardveresen, de gyakorlatilag minden 3D grafikus rendszer támogatja ezt a három szakaszt. Ahogy a GPU-k kezdtek megjelenni, először a csővezeték végét kezdték hardveresen implementálni. Tipikusan a T&L az, amit a korai 3D grafikus kártyák nem támogattak hardveresen, természetesen mára már ez is megváltozott. Amikor már a háromszögek összeállítását is támogatták a GPU-k, a processzor nem volt többé a komputergrafika szűk keresztmetszete. Ez azt a probémát szülte, hogy hogyan gyorsítsuk fel a grafikus hardvert, hogy az lépést tudjon tartani a CPU által küldött háromszögek mennyiségével. A probléma megoldása érdekében a 3D hardver tervezők a párhuzamosítás és a pipeline processzorok alapötletéhez nyúltak. A GPU abból a szempontból hasonlít a CPU-ra, hogy adott mennyiségű műveletet kell elvégezzen bizonyos adatokon, minden órajel ciklusban és a korai hardverek esetében minden egyes pixel feldolgozása rengeteg órajel ciklusba került. A pipeline processzorok alapelve hasonló, mint a gyárak futószalagjáé. Ahelyett, hogy minden ember elkészítene egy terméket a gyártási fázis elejétől a végéig, több embert ültetünk a futószalag mellé, mindegyik ember a gyártási folyamat egyetlen lépését tudja csak elvégezni, azonban miután a termék mindannyiukon végig halad, előáll a kész termék. Ugyanezt teszik a grafikus processzorok az egyes pixelekkel, ennek köszönhetően hiába kerül jó néhány órajel ciklusba egyetlen pixel feldolgozása, mivel az egész folyam szekvenciális, több pixel is jelen van a futószalagon, így elérhető az ideális egy pixel/órajel ciklus sebesség.

Annak ellenére, hogy a futószalag módszer drasztikusan megnövelte az egy órajel ciklusra jutó pixelek számát, a nagy sebességű CPU-k még mindig több háromszöget produkáltak, mint amit a grafikus kártya fel bírt dolgozni. Ezen a problémán a GPU órajelének növelése sem segített kellő mértékben. Mivel a 3D grafika egy szélsőségesen ismétlődő folyamat és mivel az adatok minden esetben ugyanúgy kerülnek feldolgozásra, mindez nagy teret hagy a párhuzamosság kihasználására a GPU-k esetében. Egy újabb futószalag hozzáadásával egyszerűen elérhetjük, hogy mostmár ne egy, hanem két pixelt állítsunk elő egy órajel ciklus alatt. Mára már több, mint 20 milliárd pixel előállítására is másodpercenként. Innentől kezdve azonban ismét a CPU lett a szűk keresztmetszet. Gyakorlatilag nincs határa, hogy hány futószalagot adhatunk még hozzá a grafikus hardverhez, azonban bizonyos mérték fölött értelmetlen tovább bővíteni azok számát, mivel a CPU nem tudja őket elég adattal ellátni, hogy ténylegesen ki legyenek használva.

1.3. Modern API-k

A videokártyák fejlődésével párhuzamosan, de annak erős befolyása alatt készültek különböző alacsony szintű és magas szintű alkalmazásprogramozási felületek (angolul application programming interface, röviden API). Mivel tipikusan a magas szintű API-k az alattuk elhelyezkedő alacsony szintű API-kra alapoznak, elsősorban az alacsony szintű API-k ismerete segíthet bennünket hatékony 3D alkalmazások készítéséhez. Ezek közül érdemes megemlíteni az OpenGL, Direct3D és Glide API-kat.

A Glide egy szabadalmazott 3D grafikus API, melyet a 3dfx fejlesztett ki a Voodoo grafikus kártyáihoz. Mivel ezen videokártyák elsősorban játékok hardveres gyorsítására készültek, a Glide API is ennek megfelelően készült, minden az API által használt adatformátum megegyezett a Voodoo videokártyák által használt belső adatformátumokkal. Mivel ezek a grafikus kártyák voltak az első olyanok, amelyek tényleg képesek voltak az akkori háromdimenziós játékokat ellátni, a Glide hamar elterjedt. Ahogy azonban megjelent a Direct3D, illetve az első OpenGL implementációk más gyártóktól, a Glide eltűnt és vele együtt a 3dfx.

A Direct3D a Microsoft DirectX grafikus API-ja, ami elsősorban a Windows 95 operációs rendszerhez készült és mára ez a legelterjedtebb grafikus API, amit Windows operációs rendszerre készült 3D alkalmazásoknál használnak, illetve ezen alapul az Xbox és Xbox 360 konzolok grafikus API-ja is. Annak a ténynek köszönhetően, hogy a DirectX fejlesztése tökéletesen lépést tart a grafikus hardver fejlődésével, a Direct3D a legkedveltebb grafikus API a játékfejlesztők körében.

Az OpenGL (Open Graphics Library) alapvetően egy platformfüggetlen 2D és 3D grafikai API szabvány specifikációja. 1992-ben jelent meg a Silicon Graphics Inc. (SGI) jóvoltából és az elsődleges API, amit a mai CAD rendszerek, VR (virtual reality – virtuális valóság) rendszerek, tudományos megjelenítők és repülőgép szimulátorok használnak. Az OpenGL API-t játékfejlesztők is alkalmazzák, azonban ezen a téren a Direct3D sokkal elterjedtebb. Az OpenGL szabvány fejlesztéséért az ARB (Architecture Review Board) konzorcium volt a felelős, melynek tagjai voltak a legfontosabb szoftverfejlesztő és hardver gyártó cégek (ATI, NVIDIA, Intel, Microsoft, stb.), majd ezt a feladatot 2006 júliusában a Khronos Group nevű konzorcium vette át. Ezek a szervezetek időközönként megbeszéléseket tartanak, amelyben megtárgyalják, hogy melyek azok a funkciók, amelyeket be vesznek a specifikáció következő verziójába. Mivel általában ez egy igen időigényes procedúra, rengetek megválaszolatlan kérdés miatt újabb tárgyalásra van szükség, illetve mivel a specifikációba későn, vagy egyáltalán nem kerülnek be az új grafikus hardverek egyes funkciói, a játékfejlesztők nem nagyon kedvelik. Ezt a problémát valamennyire megoldja az OpenGL kiterjesztés könyvtára. A hardver gyártó és szoftverfejlesztő cégek készíthetnek úgynevezett vendor specifikus kiterjesztéseket, amelyek tartalmazzák az új funkció használatához szükséges függvényeket és konstansokat. Ha egy ilyen vendor specifikus kiterjesztés az implementációk nagy részében megtalálható, többnyire az OpenGL következő verziójába is bekerül.

A fent említett grafikus API-kon kívül érdemes még megemlíteni még egyet. 1997-ben az SGI és a Microsoft elkezdett egy közös projekten dolgozni, melynek neve Fahrenheit volt. Ez valójában már egy magas szintű grafikus API volt, melyet a Direct3D és az OpenGL egységesítésére szerettek volna létrehozni. Végül a fejlesztés abbamaradt, az SGI és a Microsoft feladta a közös munkával való próbálkozást. Végül a Fahrenheit jelenetgráf (scene graph) rendszerét kiadták XSG néven, azonban hamar el is tűnt a színről.

Ahogy mondtam, mára az OpenGL és a Direct3D a két legelterjedtebb grafikus API és igazi konkurenciát kizárólag a játékfejlesztés területén jelentenek egymásnak. Habár mindkét API-nak megvan a maga előnye illetve hátránya, alapvetően mindkettő ugyanannak a hardvernek a funkcióihoz biztosít egy interfészt, tehát elsősorban csak struktúrális különbségek vannak a kettő között, funkcionalitásában a két API majdnem azonos. A következő fejezetek általános, API-független témákat

fognak taglalni, azonban az implementációk és az implementáció közeli magyarázatok elsősorban az OpenGL API-ra fognak vonatkozni. Természetesen minden technika, amiről beszélni fogok megvalósítható Direct3D segítségével is.

2. Az OpenGL programozható csővezetéke

Az OpenGL csővezeték modellje[10] illeszkedik az általános grafikus csővezeték modellre, azonban annál sokkal részletesebb. Az OpenGL bemeneteit képező adatokat közvetlen utasításokkal vagy rajzolási listák futtatásával (ami gyakorlatilag közvetlen utasítások gyüjteménye) adhatjuk meg. Mivel az OpenGL egy nem csak háromdimenziós, hanem két dimenziós megjelenítésre alkalmas API, a hagyományos bemeneteken felül pixel képeket is képes megjeleníteni (ahogy az az OpenGL működésének blokk diagrammján is látható, függelék, 2.1. ábra). Ezen rajzelemektől eltekintve az OpenGL csővezeték modellje tökéletesen megfeleltethető az általános grafikus csővezeték modellnek:

1. Geometria feldolgozó szakasz:
   1.1. Per-vertex műveletek – azoknak a műveleteknek az összessége, melyek minden egyes vertexre végrehajtódnak.
   1.2. Primitívek összeállítása – az a folyamat, mely során az előző lépés eredményeként született vertexekből elkészülnek a rajzolási primitívek (tipikusan háromszögek).
2. Megjelenítő szakasz:
   2.1. Raszterizáció – az a folyamat, mely során a rajzolási primitíveknek megfelelő fragmensek előállnak. A fragmensek gyakorlatilag atomi képelemek és többnyire egy az egyben megfelelnek egy-egy pixelnek a képernyőn vagy általánosabban a framebufferben.
   2.2. Per-fragmens műveletek – azoknak a műveleteknek az összessége, melyek az egyes fragmenseken végrehajtódnak.

2.1. Fixed function pipeline

Az OpenGL 1.0, 1.1 és 1.2 specifikációk egy jól konfigurálható, azonban nem programozható csővezetéket biztosítanak. Ez az ún. fixed function pipeline. Idővel ez a kötött funkcionalitású csővezeték egyre több konfigurációs lehetőséget biztosított, majd egyes részei teljesen programozhatóvá váltak. A fixed function pipeline, melynek részletes leírása megtalálható a specifikációban, a következő elemi műveletekből épül fel:

1. **Transzformáció (transform)** – ebben a lépésben a bejövő vertexek megszorzódnak a modell-nézet mátrixszal, így az objektum térbeli koordinátákból (object space coordinate) előállnak a nézet térbeli koordináták (eye space coordinate). Ugyanebben a lépésben történik a normál vektorok normalizálása, amennyiben engedélyezve van, valamint az eredményül kapott vertexek megszorzódnak a projekciós mátrixszal is.
   Beállítási lehetőségek: a modell-nézet és a projekciós mátrix megadása, normalizálás engedélyezése.
2. **Megvilágitási számítások (lighting)** – a vertex megvilágítottságának kiszámítása a Phong vagy a Blinn-Phong féle lokális megvilágítási modell segítségével, majd a kapott érték és a vertexhez rendelt szín szorzatának kiszámítása. Ennek a lépésnek a működése nagyon jól konfigurálható, azonban manapság már a per-vertex megvilágítás nem kielégítő a legtöbb alkalmazásnál.
   Beállítási lehetőségek: megvilágítás engedélyezése, fényforrások (max. 8) engedélyezése, ambiens, diffúz és spekuláris együtthatók megadása, fényforrás típusának megadása, stb.
3. **Textúra koordináta generálás** – amennyiben valamelyik textúra koordináta generálási módszer ki van választva, akkor a vertexhez rendelt textúra koordináta (amennyiben van ilyen) eldobódik és valamilyen beépített képlet alapján számítódik ki az új.
   Beállítási lehetőségek: a textúra koordináta generálási mód kiválasztása a lehetséges módok közül vagy a generálás kikapcsolása.

4. **Primitívek összeállítása (primitive assembly)** – a rajzolási parancsban megadottak alapján eldől, hogy mely vertexek alkotnak egy primitívet. A legtöbb gyártó esetében a négyszögeket, illetve egyéb poligonokat a hardver vagy a driver háromszögekre bontja. Továbbá ebbel a lépésben történik a hátsó lap eltávolítás, a vágás a nézet csonkagúlájával (view frustum), valamint a felhasználói vágolapokkal.
   Beállítási lehetőségek: a primitív típusának megadása, a nézet csonkagúlájának, a felhasználói vágólapok megadása és a hátsó lap eltávolítás engedélyezése.

Ennek a négy lépésnek az együttesét szokták T&L (Transform & Lighting) vagy TCL (Transform, Culling, Lighting) néven is emlegetni, valamit az első három lépés alkotja az OpenGL csővezeték per-vertex műveleteket végző szakaszát. Az első generációs videokártyák esetében a csővezeték ezen része, mint ahogy már korábban is említettem, a CPU-n került kiszámításra. A következő lépések már a megjelenítő szakaszhoz tartoznak:

5. **Interpoláció** – a pozíció, szín, textúra koordináták és egyéb vertex attribútumok interpolálása a primitív mentén. Értelemszerűen pont típusú primitív esetében nincs.
   Beállítási lehetőségek: perspektivikus korrekció engedélyezése.
6. **Textúrázás** – az interpolált textúra koordináták normalizálódnak a kiválasztott ismétlődési módnak (wrap mode) megfelelően, majd ezzel címezve megkapjuk a textúra egy texel-ét (feltéve ha a textúrázás engedélyezve van, illetve van textúra kiválasztva). A kiválasztott textúra környezeti módnak megfelelően a fragmens színe megszorzódik vagy pedig felülíródik a kapott texel színével.
   Beállítási lehetőségek: textúrázás engedélyezése, az ismétlődési mód, a textúra környezeti mód és a textúra kiválasztása.
7. **Szín összegzés (color sum)** – az OpenGL 1.2 verziójú specifikációjától van jelen. A vertex megvilágítása során a fény spekuláris összetevője egy másodlagos szín attribútumba tárolódik. Ez a lépés felelős azért, hogy a fragmens színéhez hozzáadódjon ez az összetevő.
   Beállítási lehetőségek: a szín összegzés engedélyezése.
8. **Köd számítás (fog)** – a beállított paraméterek és az aktuális fragmens pozíció alapján kiszámolódik a fragmens köd együtthatója. Ez alapján a fragmens végső színe az eddigi szín és a kiválasztott köd szín lineáris kombinációjaként áll elő.
   Beállítási lehetőségek: a köd típusának (linear, exp vagy exp2), színének és egyéb típus specifikus beállítás kiválasztása.
9. **Scissor teszt** – ha a fragmens nem esik az ún. scissor téglalapba (scissor rectangle), akkor eldobódik. Ez a téglalap viewport koordinátákkal kerül megadásra.
   Beállítási lehetőségek: a scissor teszt engedélyezése és a scissor téglalap helyének és méretének megadása.
10. **Alfa teszt** – ha a fragmens színének alfa összetevője és a referencia érték között nem áll fenn a megadott logikai feltétel, akkor a fragmens eldobódik.
    Beállítási lehetőségek: az alfa teszt engedélyezése, a referencia érték és az összehasonlító függvény kiválasztása.
11. **Stencil teszt** – a stencil bufferben lévő érték, a stencil függvény és a stencil operátor alapján módosulhat a stencil buffer illetve eldobódhat a fragmens.
    Beállítási lehetőségek: a stencil teszt engedélyezése, a stencil függvény és operátor kiválasztása az előre definiált lehetőségek közül.
12. **Mélység teszt (depth test)** – ha a fragmens mélység (depth) értéke és a mélység bufferben lévő érték között nem áll fenn a megadott logikai feltétel, akkor a fragmens eldobódik, különben a mélység bufferben lévő érték felülíródik a fragmens mélység értékével

(amennyiben engedélyezett). Tipikusan a mai hardverek esetében a mélység teszt még a textúrázás előtt történik meg (lásd később).
Beállítási lehetőségek: a mélység teszt engedélyezése, a mélység buffer módosításának engedélyezése, valamint az összehasonlító függvény kiválasztása.

13. **Keverés (blending)** – a keverési függvénynek megfelelően a fragmens színe és a szín bufferben lévő érték lineáris kombinációjaként előáll a végső szín mely bekerül a szín pufferbe (az írási maszkoknak megfelelően).
Beállítási lehetőségek: a keverési függvény kiválasztása és a hozzá tartozó paraméterek beállítása.

Ahogy az a lépések ismertetésében is látható, nagyon sok konfigurációs lehetőséget kinál az OpenGL kötött funkcionalitású csővezetéke is, valamint a hardver fejlődésével párhuzamosan rengeteg további lehetőséggel bővült (például új textúra koordináta generálási módok, textúra környezeti módok, keverési egyenlőségek, stb.), mégis az egyre inkább programozható hardver, valamint a szoftverfejlesztők igénye egy sokkal kötetlenebb, programozható csővezeték iránt oda vezetett, hogy mára már az OpenGL csővezetékének több része is programozható, sőt, hamarosan megjelenik a specifikáció 3-as verziója, amely majdnem a teljes csővezeték programozhatóságát támogatja majd.

2.2. Vertex shaderek

A kiterjesztetlen OpenGL API jó néhány konfigurációs lehetőséget biztosít a vertex transzformációk, textúra koordináta generálás és a megvilágítás számításához. Mindezen felül rengeteg kiterjesztés szolgáltat további textúra koordináta generálási módokat (**NV_texgen_reflection**[GL23], **NV_texgen_emboss**[GL24]), új vertex transzformációs módszereket (**ARB_vertex_blend**[GL2], **EXT_vertex_weighting**[GL18]), új megvilágítási módszereket, stb.

Minden ilyen kiterjesztés bővíti a vertex számítási lehetőségek tárházát pár meglehetősen rugalmatlan technikával. Ez a rugalmatlanság ellenmondott a tipikusan nagyon is rugalmas hardvernek (legyen az akár mikrokódolt vertex motor vagy CPU), amit használni szoktak általában az OpenGL per-vertex számításainak implementálására.

A vertex shaderek az OpenGL csővezeték per-vertex műveletekkel foglalkozó részét szándékoznak lecserélni, ami az előbb felsorolt lépések közül az első három: a T&L (függelék, 2.2. ábra). Ennek köszönhetően tetszőleges transzformációkat hajthatunk végre, tetszőleges textúra koodináta generálási technikákat alkalmazhatunk és talán a leglényegesebb, hogy tetszőleges vertex szintű megvilágítási modellt implementálhatunk. Szintén használt a vertex program elnevezés is, azonban elterjedtebb az, hogy shadereknek nevezik ezeket a programokat, pontosan azért, mivel tipikusan a megvilágítási modellek megvalósításához használják őket.

Természetesen a „tetszőleges” kifejezés nem teljesen igaz ez esetben, hisz mind a végrehajtható műveletek száma, mind a használható regiszterek és változók, mind pedig az utasításkészlet kötött, azonban az eddigi kötött funkcionalitású csővezetékhez képest hatalmas szabadságot kínál a fejlesztők számára. A második generációs grafikus kártyák (Radeon 7000, GeForce 256) már támogatták a vertex shadereket, azonban csak szoftveresen, vagyis a vertex shader a CPU-n hajtódott végre. Ennek hátránya az volt, hogy a már hardveres TCL így szoftveresen helyettesítődött, ezzel nagy mértékben csökkentve a rendszer vertex feldolgozó teljesítményét.

A vertex shaderek használata a DirectX API esetében a 7-es verziótól van jelen, az OpenGL esetében csak a 2.0-ás szabványban van jelen, azonban a funkcionalitás különböző formában elérhető az **EXT_vertex_shader**, **ARB_vertex_program**[GL8], **NV_vertex_program** illetve **ARB_vertex_shader**[GL12] kiterjesztéseken keresztül.

2.3. Geometria shaderek

A legújabb programozhatósági lehetőségeket a geometria shaderek nyújták, melyek alig több, mint egy éve támogatottak a grafikus kártyák által. A geometria shaderek a per-vertex műveletek után és a vágási műveletek előtt hajtódnak végre.

A geometria shaderek bemenete egyetlen primitív, legyen az egy pont, szakasz vagy háromszög. A shaderen belül lehetőség van a primitív egyes vertexeinek az atribútumainak felhasználására, melyek segítségével új primitíveket készíthetünk. A geometria shader kimenetének típusa természetesen kötött (vagy pont, vagy szakaszcsík, vagy pedig háromszög csík) és az effektív kimenetei gyakorlatilag vertexek, amelyek leírják ezeket az új primitíveket, lehetőség van továbbá hogy egy geometria shader egy primitívből több különálló primitívet állítson elő. A shader által előállított primitívek természetesen átesnek a vágási műveleteken és a csővezeték összes további szakaszán ugyanúgy, mintha az alkalmazás adta volna meg közvetlenül ezeket.

Ezeket a shadereket használják például tesszeláción alapuló LOD (Level Of Detail) technikák megvalósításához, valamint segítségével a shadow volumes árnyékolási technika teljesen hardveres implementációja is elkészíthető (lásd később). Az API támogatás tekintetében a geometria shaderek a DirectX 10-es verziójában jelentek meg, az OpenGL viszont még nem vette be ezt a funkcionalitást a szabványba, sőt, habár az **`EXT_geometry_shader4`** kiterjesztés lehetőséget biztosít a geometria shaderek használatára, ezt a kiterjesztést egyelőre csak az NVIDIA OpenGL implementációja támogatja.

2.4. Fragmens shaderek

Akárcsak a per-vertex műveletek esetében, az OpenGL (illetve hasonlóképpen a DirectX is) rengeteg lehetőséget biztosít a textúrák alkalmazására, a textúra környezet konfigurálására, a köd számításra, legyenek azok a szabvány által támogatott technikák, avagy különböző kiterjesztések, mint például az **`ARB_texture_env_combine`**[GL3], **`ARB_shadow`**[GL7], **`EXT_fragment_lighting`**, stb. Ismét, ezek relatív kötött funkcionalitást biztosítanak a programozó számára.

A fragmens shaderek ebből a célból jöttek létre. Az OpenGL kötött funkcionalitású csővezetékét tekintve a fragment shaderek a következő szakaszokat definiálják felül:

- Textúrázás
- Szín összegzés
- Köd számítás

A DirectX API, logikailag tévesen, ezeket a programokat pixel shadernek nevezte el, ami azért helytelen, mert egyrészt itt még a csővezeték csak fragmensekkel dolgozik, sehol nincs szó még pixelekről, másrészt habár tipikusan egy keletkezett fragmens a framebuffer egy pixeljét fogja jelenteni, ez általánosan nem igaz. Szintén problémát okozhat a két API által definiált shader fogalmak megfeleltetésekor az a tény, hogy a DirectX már pixel shadernek nevezte az NVIDIA register combiner és texture shader mechanizmusainak együttesét, annak ellenére hogy ezek még inkább csak konfigurálhatóak voltak, mintsem programozhatóak. Továbbá az OpenGL esetében először az ATI_fragment_shader kiterjesztéssel párhuzamosan jelent meg a fragment shader, mint fogalom. Ezt a kiterjesztést azonban kizárólag az ATI OpenGL implementációi támogatták. Ténylegesen fragment shaderekről a szabványosított shaderek megjelenésétől beszélhetünk. Ezt a funkcionalitást az ARB_fragment_program és az ARB_fragment shader OpenGL kiterjesztések nyújtják, ami jelen is van a szabvány 2.0-ás verziójában. A szabványosított fragment shader megfelel a DirectX pixel shader 2.0-val, ami a DirectX 9-es verziójában jelent meg.

Alkalmazásukat tekintve a fragment shadereket tipikusan a per-fragmens (per-pixel) megvilágítási modellek implementációjára használják. Természetesen sok más alkalmazása is van, többek között képek utófeldolgozása, különböző szűrők (filterek) implementálása, stb.

2.5. Jövőkép

A videokártyák többsége mára már támogatják az úgynevezett egységes shader architektúrát, ami annyit jelent, hogy a csővezeték funkcionalitásának egyes részeiért felelős hardver elemek nincsenek már elkülönítve, a GPU egyetlen erősen párhuzamosított adatfolyam processzor, amely a csővezeték teljes műveletigényét kielégíti, illetve annak minden feladatát ellátja. Ez azt a tényt is maga után vonja, hogy gyakorlatilag a grafikus csővezeték összes lépése felül definiálható, programozható. A 3D grafikai API-k fejlesztői ezért további shaderek bevezetését tervezik, többek között a következőket:

- **Texture sample shader** – A textúrákhoz való hozzáférést teszi majd programozhatóvá. Ez azt jelenti, hogy a beépített bilineáris, trilineáris, anizotróp és egyéb szűrők kiválasztása helyett saját szűrőket definiálhatunk. Például elkészíthetjük többek között a Gauss vagy a PCF (Percentage-Closer-Filtering) szűrőket, amelyeket ma is gyakran használnak a 3D grafikában, csak támogatás híján mindezt egy fragmens shader segítségével valósítják meg, ami sokkal kevésbé hatékony.
- **Blend shader** – A kötött funkcionalitású csővezeték utolsó szakaszát szándékozik majd felül definiálni. Segítségével a keverés folyamata teljesen programozhatóvá válik. Akárcsak az előző shader típus esetében is, egy fragmens shader segítségével meg tudjuk valósítani ugyanezt a lehetőséget, azonban sokkal körülményesebben és természetesen nem elég hatékonyan.

Akár az eredeti rendeltetési céljukat tekintjük, azaz a 3D grafikai elemek megjelenítését, akár pedig az általános célú lebegő pontos számítási lehetőséget, a videokártyák egyre nagyobb teret nyernek a nagy mértékű rugalmasságuknak, illetve a CPU-knál sokszorta nagyobb számítási teljesítményüknek köszönhetően.

3. A vertex feldolgozó csővezeték fejlődése

Az első generációs grafikus kártyák kizárólag a megjelenítő szakasz hardveres gyorsítására voltak képesek. Ez annyit jelentett, hogy a vertexekkel kapcsolatos műveletek, valamint a rajzolási primitívek összeállítása továbbra is a CPU feladata volt így nagyon sok esetben a vertex feldolgozási teljesítmény jelentette a szűk keresztmetszetet a 3D grafikai alkalmazások számára. Emiatt a 3D megjelenítő rendszerek fejlesztését ezen irányvonal mentén kezdték el.

3.1. Klasszikus modell

1996-ban jelentek meg az első generációs grafikus kártyák[15], többek között a 3dfx Voodoo (függelék, 3.1. ábra). Ezek a videokártyák kifejezetten a 3D grafikai műveletekre voltak kihegyezve, ebből kifolyólag a hagyományos 2D grafikai funkciókat (amelyeket a standard videokártyáknál jelen voltak) nem támogatták. Ehelyett összekapcsolva működtek egy 2D grafikus kártyával. Az összekapcsolás esetében most nem azt értem, hogy együtt is működtek: a standard videokártya VGA kimenetét rá kellett csatlakoztatni a Voodoo kártyára, majd ennek kimenetét kellett a monitorhoz kapcsolni. Az alapelv az volt, hogy amennyiben nincs szükség 3D grafikai megjelenítésre, akkor a Voodoo egyszerűen továbbította a VGA kártyától kapott képinformációt. Amennyiben viszont egy 3D alkalmazásnak szüksége volt a Voodoo hardveres 3D grafikai adottságaira, abban az esetben a VGA kártya kimenetét ignorálta és kizárólag az általa előállított képet továbbította a monitor irányába.

Ez azt jelentette, hogy a két videokártya közül egyszerre csak egyik produkált aktívan kimenetet, és mivel az egyik funkcionalitása a 2D grafikára, a másiké pedig a 3D grafikára korlátozódott, ezért nem volt lehetőség, legalább is hatékony, a 2D és 3D grafikai elemek kombinálására. Ez többek között lehetetlenné tette az ablakos környezetek számára a 3D grafikai alkalmazások ablakban való megjelenítését.

A Voodoo az egyik első olyan boltokban kapható grafikus kártya volt, mely hardveres textúrázást és Z-buffer támogatást biztosított. Amit viszont nem támogattak ezek a kártyák, az a hardveres vertex transzformáció, melyre szintén nagy szükség van a 3D grafikai elemek megjelenítéséhez. Ezek a műveletek a CPU-n kerültek elvégzésre és tipikusan a grafikus API (OpenGL, DirectX, Glide) implementálta ezeket.

Mivel mind a CPU-k, mind a grafikus kártyák teljesítménye, mind pedig a kettőt összekapcsoló 33MHz-es PCI busz teljesítménye meglehetősen alacsony volt abban az időben, a valós idejű 3D alkalmazás fejlesztők álmodni sem mertek például arról, hogy egy bonyolultabb per-vertex megvilágítási modellt implementáljanak, megelégedtek azzal, hogy lehetőségük van egyszerű textúrázott primitívek hardveres gyorsítással való megjelenítésére.

3.2. A hardware T&L megjelenése

A második generációs 3D grafikus kártyák[14] megjelenésével minden megváltozott. 1999-ben megjelent az NVIDIA Corp. GeForce szériája (függelék, 3.2. ábra) és vele párhuzamosan az ATI Technologies Inc. Radeon szériája.

Ezek a videokártyák több technológiai újítást vezettek be, melyek közül a legfontosabb a grafikus csővezeték geometriai szakaszának a hardveres gyorsítása. Ez az amit úgy szoktak nevezni, hogy hardware T&L (Transform & Lighting) vagy hardware TCL (Transform, Culling, Lighting).

Innentől kezdve a CPU-nak csak annyi feladata maradt, hogy a kirajzolandó primitív vertexeit szolgáltassa. Ez azonban egy nagyobb adatmennyiséget jelentett, hisz korábban csak a látható primitívek vertexeinek képernyő koordinátákban megadott helyét kellett átküldeni a buszon. Ennek optimalizálására készült az AGP busz, ami azon felül, hogy nagyobb sávszélességgel rendelkezett

(AGP 1x - 66MHz, 32 bit, 133MB/s), lehetőséget biztosított arra, hogy a videokártyák a saját lokális memóriájukon felül a rendszer memóriáját is címezni tudják.

Mivel a hardware T&L megjelenése a per-vertex műveletek számítási költségét meglehetősen lecsökkentette, a fejlesztők elkezdtek az összetettebb vertex műveletek irányába kacsintgatni, amelyet részben ki is elégítettek a második generációs grafikus kártyák az új textúra koordináta generálási és megvilágítási módszerekkel.

3.3. Vertex shaderek a CPU-n

Az ezredforduló után, 2001-ben megjelent az ATI Radeon 8500 (függelék, 3.3. ábra) és vele párhuzamosan az NVIDIA GeForce 3 és velük együtt a DirectX 8. Ezek a harmadik generációs videokártyák[15] már több-kevesebb programozhatóságot hordoztak magukban, ami nagyon népszerűvé tette őket a 3D alkalmazás fejlesztők körében. Elsősorban a vertex feldolgozó csővezeték programozhatósága nyerte el a fejlesztők figyelmét, mivel a fragmens feldolgozó csővezeték programozhatósága ezekben a kártyákban egyrészt még meglehetősen limitált volt, másrészt pedig nem voltak elég performensek, hogy ezen funkciók masszív használatát támogassák.

Mivel a programozók nagyon hamar rászoktak a vertex shaderek használatára és mivel ezek a kártyák még nem terjedtek el kellő mértékben a felhasználók körében, a videokártya cégek új ötlettel álltak elő: lehetőséget biztosítanak a vertex shaderek használatára a korábbi kártyáik esetében is, azonban ez esetben megfelelő hardveres támogatás hiányában a geometriai szakasz műveletei ismét a CPU feladata lesz. Mivel ekkorra már a CPU-k is jóval nagyobb teljesítménnyel bírtak, az ötlet a gyakorlatban is működött.

Habár ennek a trükknek köszönhetően a második generációs grafikus rendszerek is képessé váltak vertex shaderek futtatására, természetesen ezek vertex feldolgozó teljesítménye erősen csökkent ezáltal. Természetesen amennyiben a grafikus alkalmazás nem kívánt élni a vertex shaderek által nyújtott lehetőségekkel, akkor a geometriai szakasz műveletei továbbra is a GPU-n futottak ezeknél a kártyáknál is.

A technika, hogy hol hardver által gyorsított vertex feldolgozás történt, hol pedig ennek szoftveres helyettesítése, egy elég kellemetlen problémát eredményezett. Mivel a modern per-fragmens (per-pixel) megvilágítási modelleket és egyéb technikákat ezek a videokártyák csak több lépés során tudták csak megvalósítani, figyelembe kellett venni azt, hogy a szoftveres és a hardveres vertex feldolgozás ugyanazt az eredményt szolgáltassa, mivel egy-egy apró lebegő pontos számítási pontatlanság miatt ugyanazok a poligonok az egyes lépésekben egymástól elcsúszva jelenhettek meg. Ez abból fakad, hogy a CPU-k belső lebegő pontos pontossága nem minden esetében egyezik meg a GPU-k belső pontosságával, sőt az egyes videokártya és processzor gyártók esetében is gyakran különbözik. Ennek a problémának a kiküszöbölésére jött létre a vertex shader 1.1-es verzió, vagy az OpenGL megfelelője, a vertex shadereknél használható ARB_position_invariant opció, ami garantálta hogy a vertex shader tökéletesen ugyanazt az eredményt produkálja a vertex transzformációt követően, mint ahogy azt a kötött funkcionalitású csővezeték teszi. Ezt általában a driverek úgy biztosították, hogy a vertex transzformációt elvégezték a GPU-n is a vertex shader végrehajtásával párhuzamosan és a raszterizációnál a GPU által produkált vertexeket használták fel.

3.4. Vertex shaderek a GPU-n

Ahogy már említettem, a harmadik generációs 3D grafikus kártyáktól fogva a GPU képes volt viszonylag rövig vertex shaderek végrehajtására. Ezek már megfeleltek összetettebb megvilágítási modellek implementálására, akár legyen az egy új per-vertex megvilágítási modell leprogramozása, akár pedig a vertex szintű része egy per-fragmens megoldásnak.

Természetesen ezzel nem ér véget a történet, hisz ezek az 1.1-es verziójú vertex shaderek korlátolt mennyiségű és típusú utasítást, valamint meglehetősen kevés ideiglenes regisztert biztosítottak. Az idő folyamán egyre több és több lehetőséggel bővült a vertex shaderek által nyújtott lehetőségek tárháza. Ma már szinte végtelen hosszú shaderek futtatására alkalmasak a hardverek, valamint lehetőség van például textúra hozzáféréshez egy vertex shaderen belül is, minek segítségével képesek lehetünk például textúrákkal leírt domborzat megjelenítésére egy egyszerű vertex program írásával.

4. A raszterizálás fejlődése

A textúrák alkalmazása a 3D grafikában már az első gyorsító kártyák megjelenése előtt is nagy hangsúlyt kapott, nem hiába már az első hardveres implementációkba is beépített textúrázási lehetőségek voltak. A grafikus kártyák textúrázási lehetőségeinek folyamatos fejlődése mára oda vezetett, hogy a fragmens shaderek segítségével a primitívünkhöz tartozó textúrát vagy textúrákat tetszőleges módon felhasználhatjuk a primitív fragmenseinek színének a meghatározásához, sőt, ezek alapján akár a fragmens mélységét (Z értékét) is manipulálhatjuk vagy éppen eldönthetjük, hogy egyáltalán szeretnénk-e hogy megjelenjen az illető fragmens vagy pedig eldobjuk azt.

4.1. Klasszikus modell

A 3D grafikus kártyák által használt textúrázás klasszikus modellje nagyon egyszerű. Két összetevőből áll:

1. **A fragmenshez tartozó szín** – A primitívhez tartozó vertexek színének interpolációjából származó érték.
2. **A fragmenshez tartozó textúra szín** – A primitív vertexeinek textúra koordinátáinak interpolációjából származó értékből kapjuk a fragmenshez tartozó textúra koordinátát, mellyel megcímezve a primitívhez tartozó textúrát, megkapjuk ezt az értéket.

Mindössze annyi konfigurációs lehetőségünk van ezen felül, hogy eldöntjük, hogy a fragmens végső színének meghatározásában melyik összetevő vesz részt, illetve, ha mindkettő, akkor milyen műveletet alkalmazunk a kettő között (tipikusan csak a szorzás volt a megengedett).

Ez a modell tökéletesen alkalmas volt egyszerű, textúrával rendelkező és per-vertex megvilágítási modellel árnyalt objektumok megjelenítésére. Azonban a per-vertex megvilágítási modellek a legtöbb esetben nem produkálnak megfelelő eredményt, vagy a jó eredményhez szükséges tesszeláció miatt túlságosan túlterheljük a vertex feldolgozó csővezetéket.

4.2. A multitextúrázás megjelenése

Még a második generációs 3D grafikus kártyák előtt megjelentek olyan kártyák, amelyek két textúrázó egységgel rendelkeztek (pl. 3dfx Voodoo2, NVIDIA Riva TNT 2). Ez azt jelentette, hogy egy adott primitívhez tartozó fragmensek színének meghatározásakor az előbb bemutatott komponensen kívül használható egy harmadik is, ami szintén egy textúrából származik. Nyilvánvalóan ahhoz, hogy két teljesen független textúrát használhassunk ugyanahhoz a primitívhez, szükség van két textúra koordinátára vertexenként. OpenGL esetében ezt a funkcionalitást az **`ARB_multitexture`**[GL1] kiterjesztés szolgáltatja, valamint a szabvány 1.3-as specifikációjába is bekerült.

A technika első hardveres implementációiba azon felül hogy a felhasználható operandusok száma egy újabb textúrával háromra nőtt, további funkciók nem kerültek be. Ez azt jelentette, hogy az operandusok között tipikusan az egyetlen művelet a szorzás lehetett.

Annak ellenére, hogy ez látszólag túlságosan kötött és haszontalan, a gyakorlatban nagyon is nagy jelentősége volt. A legelterjedtebb 3D megjelenítési technika ebben az időben a lightmapping volt. A megjelenítendő világ objektumaira, primitíveire egy szoftver segítségével kiszámoltattak bonyolult megvilágítási modelleket, illetve árnyék számításokat, majd eredményképpen megkapták az egyes primitívek darabjainak (ún. patch) a megvilágítottságát. Ezt a megvilágítási mintát egy-egy viszonylag kis felbontású szürkeskálás képbe kimentették. A valós idejű 3D alkalmazások ezután csak a legenerált képeket kellett felhasználják mint második textúra. Így az objektumok mintázat textúrája és ennek a textúrának a szorzataként előállt a végső minta, mely egy összetett algoritmussal megvilágított világ látszatát keltette. Természetesen mivel ezek az úgynevezett lightmap-ek adott

megvilágítási körülmények között lettek legenerálva, kizárólag statikus objektumok és fényforrások megjelenítésekor alkalmazható.

Persze a felhasználható textúrák száma nem limitálódik kettőre, ez csak a technológia megjelenésekor volt így. Mára már az OpenGL kötött funkcionalitású csővezetéke is képes nyolc textúra kezelésére, valamint fragmens shaderek alkalmazásával tizenhat textúrát vagy akár egész textúra tömböket is használhatunk.

4.3. A texture combiner mechanizmus

Ez a technológia a 3D grafikus kártyák második generációjában jelent meg először, pár kivétellel: ATI Rage 128, NVIDIA GeForce 256, Intel 650, stb.

Alapvetően a texture combiner technológia a multitextúrázást egészíti ki új operátorokkal. Mivel a hagyományos multitextúrázás esetében tipikusan csak a szorzás műveletet használhattuk, abban az esetben nem volt szükséges meghatározni egy operandus sorrendet a szorzás kommutativitásából következően, azonban a texture combiner eredeti specifikációjához ezt meg kell tegyük. Az OpenGL API esetében ennek a technológiának a funkcionalitását az **`ARB_texture_env_combine`**[GL3] kiterjesztésen keresztül érhetjük el, mely az **`ARB_multitexture`** kiterjesztéssel együtt bekerült az OpenGL 1.3-as specifikációjába.

A texture combiner mechanizmus úgynevezett textúra szakaszokról beszél. Formálisan ez a következőképpen definiálható:

1. Ha az első textúra szakasz engedélyezve van, akkor az eredménye $C \lozenge_1 T_1$, ahol C az interpolált per-vertex szín, T_1 az első textúra és $\lozenge_1$ az első textúra szakaszhoz beállított operátor. Ha az első textúra szakasz nincs engedélyezve, akkor az eredmény az interpolált szín.
2. Minden i=1..n-1 értékre elvégezzük a 3. Pontot.
3. Ha az i+1-edik textúra szakasz engedélyezve van, akkor az eredménye $S_i \lozenge_{i+1} T_{i+1}$, ahol S_i az i-edik textúra szakasz eredménye, T_{i+1} az i+1-edik textúra és $\lozenge_{i+1}$ az i+1-edik textúra szakaszhoz beállított operátor. Ha az i+1-edik textúra szakasz nincs engedélyezve, akkor az eredmény az i-edik textúra szakasz eredménye.

A fenti definíció alapján a fragmens végső színének kiszámítására a következő formulát kapjuk:

$$\text{fragmens végső színe} = (\ldots((\text{interp?lált szín} \lozenge 1\ 1.\text{textúra}) \lozenge 2\ 2.\text{textúra}) \ldots \lozenge n\ n.\text{textúra}$$

Ahol az $\lozenge_i$ (i=1..n) jelöli az i-edik operátort. Értelemszerűen az operandusok 4-elemű vektorok. A használható operátorok pedig többek között a következőek lehetnek:

1. **Felülírás** – Ebben az esetben az operátortól balra lévő érték eldobódik és a művelet eredménye az operátortól jobbra lévő érték lesz.
2. **Szorzás (*)** – A művelet eredménye a két operandus szorzata.
3. **Összeadás** (+) – A művelet eredménye a két operandus összege. Használata feltételezi az **`ARB_texture_env_add`** kiterjesztés jelenlétét.
4. **Belső szorzat (.)** – A művelet eredménye a két operandus belső szorzata (a homogén koordináta figyelmen kívül hagyásával). Használata feltételezi az **`ARB_texture_env_dot3`**[GL5] kiterjesztés jelenlétét.

Ezen kívül több vendor specifikus kiterjesztés további operátorokat definiál (például az **`ATI_texture_env_combine3`**[GL22] vagy az **`NV_texture_env_combine4`** kiterjesztések), valamint az **`ARB_texture_env_crossbar`**[GL4] kiterjesztés segítségével az operandusok sorrendi kötöttségét is felül definiálhatjuk.

Ez volt az első igazán komoly eszköz a per-fragmens megvilágítási modellek hardveres implementációjához, valamint olyan közismert technológiák megvalósítására, mint például a bump mapping. Érdekes módon ennek a ténynek ellenmond az, hogy a DirectX a texture combiner mechanizmust még nem sorolta a pixel shaderek közé, viszont más, nem sokkal többet nyújtó technológiákat már igen.

4.4. Register combinerek és texture shaderek

A register combinerek és a texture shaderek által kínált funkcionalitások nem jelentettek hatalmas áttörést a grafikus kártyák fragmens shaderek fejlődésében elsősorban azért, mert a technikát egyedül az NVIDIA kártyái támogatták, azonban mégis érdemes megemlíteni őket, mert ezeket nevezi a DirectX az első pixel shader implementációnak.

A register combiner mechanizmus OpenGL esetében az **`NV_register_combiners`**[GL25] és az **`NV_register_combiners2`**[GL26] kiterjesztéseken keresztül használható és ez előbbi már az NVIDIA GeForce szériájától (NV10) támogatott. A register combinerek által nyújtott új funkcionalitások közül többek között érdemes a következőket megemlíteni:

1. Míg a klasszikus texture combiner mechanizmus a számítások esetében minden eredményt a [0, 1] intervallumra szűkít, addig a register combinerek a [-1, 1] intervallumot támogatják.
2. A textúra szakaszok helyett az úgynevezett combiner szakasz fogalmat vezeti be a bővítmény, mely a textúra szakaszok által kínált beállításokon túl további konfigurációs lehetőségeket biztosít.
3. A textúra szakaszok számán túl biztosít továbbá egy speciális combiner szakaszt, mely felhasználva az eddigi combiner szakaszok eredményét, további számításokkal egészítheti ki azt, mint például a szín összegzés vagy a köd számítás.

Ennek értelmében a register combinerek felüldefiniálják az OpenGL csővezeték textúrázó lépésén felül a szín összegzési és köd számítási lépését is, ellentétben az eddigi technikákkal.

Habár a kínált lehetőségek tovább növelik a raszterizálás testre szabhatóságát, az áttörést mégsem a register combinerek nyújtották, hanem a texture shaderek. A texture shaderek az NVIDIA következő generációjában jelentek meg, a GeForce 3 (NV20) kártyákon.

Mind az OpenGL eredeti specifikációja, mind pedig a multitextúrázás illetve egyéb kiterjesztések úgy működtek mind eddig, hogy a textúra koordinátáknak megfelelő textúra színek egy fix mechanizmussal álltak elő. Minden textúra típus esetében, legyen az egy dimenziós, két dimenziós, háromdimenziós textúra, vagy textúra kocka, létezik egy specifikáció, hogy a textúra képeket (texture image) hogyan tudjuk címezni, hogyan tudjuk annak texeleit (texture element) elérni a textúra koordináták segítségével. A texture shaderek ezt a kötöttséget oldják fel bizonyos szintig, ami annyit jelent, hogy a hagyományos textúra elérési módszer helyett egy tucatnyi előre definiált módszert használhatunk. Ez az eredeti **`NV_texture_shader`**[GL27] kiterjesztés specifikációja, melyet további elérési módszerrel egészít ki az **`NV_texture_shader2`**[GL28] és **`NV_texture_shader3`**[GL29] kiterjesztés. A lehetőségek közül párat ismertetnék is, hogy jobban megérthessük az új módszerek jelentőségét. Először is a hagyományos OpenGL a következő elérési módszereket támogatta ([s,t,r,q] jelöli a textúra koordinátákat):

1. **`TEXTURE_1D`** – Egy dimenziós textúra címzésére használt módszer, az (s/q) értékkel címzi a képet.
2. **`TEXTURE_2D`** – Két dimenziós textúra címzése, a képkoordináták (s/q, t/q)-ként állnak elő.
3. **`TEXTURE_3D`** – Háromdimenziós textúra címzése az (s/q, t/q, r/q) képkoordinátákkal.
4. **`TEXTURE_RECTANGLE`** – Téglalap alakú textúra címzése, a képkoordináták (s/q, t/q).

5. **TEXTURE_CUBE_MAP** – Textúra kocka címzése az (s, t, r) képkoordináta hármas segítségével.

Minden esetben a címzési módszert a textúra szakaszhoz rendelt textúra típusa határozta meg, a címzési mód kiválasztására nem volt módszer. Ezzel szemben a texture shader mechanizmus segítségével választhatunk akármelyik hagyományos és új textúra címzési módszer közül. Lássunk néhány nagy jelentőséggel bíró módszert a texture shaderek által kínáltak közül:

1. **PASS_TROUGH** – A textúra szakaszhoz rendelt textúrát figyelmen kívül hagyva egyszerűen az (s,t,r,q) textúra koordináta négyest adja tovább, mint (r,g,b,a) színösszetevők (melyek értékét a [0,1] intervallumra szűkíti).
2. **CULL_FRAGMENT** – Eldobja a fragmenst az alapján, hogy a hozzá tartozó textúra koordináták nem-negatívak, avagy nem-pozitívak-e. Ennek segítségével már a per-fragmens műveleteket megelőzően is lehetőségünk van eldobni a fragmenseket.
3. **OFFSET_TEXTURE_2D** – Egy előző szakasz eredményét, mint (ds, dt) koordinátapárt megszoroz egy 2x2-es mátrixszal, majd ezt, mint egy eltolást, hozzáadja az aktuális szakaszhoz tartozó textúra koordinátákhoz és az eredményül kapott koordinátapárt használja a textúra címzésére.

Ezen felül még jó néhány lehetőséget kínálnak a texture shaderek, melyek hasonlóan a harmadikként felsorolt módszerhez, lehetőséget biztosítanak textúra indirekcióra. A textúra indirekció (texture indirection) annyit jelent, hogy egy textúra címzésére olyan koordinátákat használunk fel, amelyek függenek egy másik textúra címzése eredményéül kapott textúra színtől. Valószínűleg ez volt az oka, hogy a DirectX API először a register combinerek és a texture shaderek által együttesen szolgáltatott funkcionalitást nevezte pixel shadernek. A mechanizmus a DirectX 8-as verziójába került be, azonban az OpenGL esetében a technológia vendor specifikus mivolta miatt nem került be a szabványba. Az egyes DirectX pixel shader verziók és a nekik megfelelő OpenGL kiterjesztések a függelék 4.1. ábráján láthatóak.

4.5. Fragmens shaderek

Habár a DirectX már eddig is beszélt pixel shaderekről (ami nagyjából az OpenGL fragmens shadereknek felel meg), az OpenGL még nem nevezte az eddig említett mechanizmusokat fragmens shadereknek. A DirectX az 1.1-1.3 verziójú pixel shaderei számára biztosított is egy minimális assembly jellegű programozási nyelvet is, az OpenGL-t használók az NVIDIA NVParse nevű eszközét alkalmazva érhették el, hogy függvényhívások helyett programocskákkal vezérelhessék a register combinereket és texture shadereket. Persze ezek messze nem kínáltak igazán komoly lehetőségeket, a pixel shaderek csak néhány utasítást tartalmaz(hat)tak.

A DirectX esetében, ahogy már korábban is említettem, a pixel shader támogatás minimum követelménye a textúra indirekció lehetősége. Az OpenGL esetében ez a minimum követelmény kicsit másképp fogalmazható meg.

4.5.1. Első generáció

Az első OpenGL fragmens shader még jóval a 2.0-ás specifikációban szereplő fragmens shaderek előtt, az **ATI_fragment_shader**[GL20] kiterjesztés formájában jelent meg. Habár ez szintén egy vendor specifikus kiterjesztés, amit kizárólag az ATI kártyái (Radeon 8500+) támogattak, a kiterjesztés által kínált funkcionalitás már nagyon közel áll a vendor független, szabványosított fragmens shaderekhez. A DirectX az ATI fragmens shadereknek megfelelő működést az API 8.1-es verziójától támogatja és pixel shader 1.4 néven emlegeti.

Az **ATI_fragment_shader** kiterjesztés tejlesen újra definiálja a textúrázási lépés működését, félretéve a textúrázási szakaszok szekvencialitásából adódó rugalmatlanságot. A kiterjesztés leglényegesebb innovációi:

- Textúra szakaszok helyett bármely textúrázó egységhez rendelt textúra a fragmens shader bármely pontján hozzáférhető.
- A textúra koordináták, valamint a textúra elérésből származó textúra színek egységesen ugyanúgy kezelhetőek, mint négy-elemű vektorok.
- A legtöbb alapvető számítógép grafikánál használt vektor művelet támogatott (**MUL**, **ADD**, **DOT3**, stb.)
- Tetszőleges swizzle alkalmazása a műveletek operandusaira (swizzle – a négy-elemű vektorok koordinátáinak permutálása).
- Egy fragmens shader két lépésből (pass) áll, melyek egymás után hajtódnak végre, lehetőséget biztosítva értékátadása az egyes lépések között.
- Egy lépés hat utasítást foglalhat magában, melyek akár mind lehetnek textúra hozzáférési utasítások is, ezzel lehetőséget biztosítva akár összesen tizenkét textúra hozzáférésnek egyetlen fragmens esetében, míg az akkori NVIDIA kártyák segítségével csak négyre volt lehetőségünk.

Eredetileg a kiterjesztés egy egyszerű procedurális interfészt biztosított a fragmens shaderek konfigurálásához, azonban az **ATI_text_fragment_shader**[GL21] kiterjesztés (amit csak néhány driver publikált, viszont az újabb driverek is támogatják) lehetőséget biztosít arra, hogy az **ARB_vertex_program**[GL8] kiterjesztés által definiált vertex shader futtató keretrendszer az ATI fragmens shadereit is használni tudja. Az **ATI_text_fragment_shader** kiterjesztés által definiált assembly nyelv nem hiába nagyon hasonló az OpenGL assembly shader nyelvéhez, hisz ez a kiterjesztés indította el azt a folyamatot, ami végül a vendor független fragmens shaderek megjelenését eredményezte.

4.5.2. Szabványosított fragmens shader

Ezen az irányvonalon elindulva 2002-ben megjelent a 3D grafikus kártyák negyedik generációja[14], köztük az ATI Radeon 9700 (függelék, 3.4. ábra), valamint az NVIDIA GeForce FX széria. Ezek voltak az első kártyák, melyek esetében a grafikus csővezeték textúrázó, szín összegző és köd számítási lépéseknek megfelelő részei teljesen programozhatóak voltak. Elvonatkoztatva az ATI fragmens shaderei által bevezetett shader lépések fogalmától, egy a már létező **ARB_vertex_program** kiterjesztés által definiált shader nyelvre jobban illeszkedő alacsony szintű fragmens shader nyelvet vezetett be, melyet az **ARB_fragment_program**[GL9] kiterjesztésen keresztül használhatunk. A DirectX ugyanezt a funkcionalitást a DirectX 9 pixel shader 2.0-ján keresztül biztosítja.

Ez a fragmens shader definíció már teljesen vendorfüggetlen volt, melyet hamar adoptált a többi grafikus kártya gyártó is. Természetesen mind gyártó, mind kártya függő volt, hogy hány és milyen utasítás szerepelhet egy ilyen fragmens shaderben, azonban ettől kezdve az újabb hardverek és shader verziók már tejlesen erre, a már meglévő keretrendszerre alapoztak. Ez elindította a shader nyelvek elterjedését és ezzel együtt annak erőteljes fejlődését, ami oda vezetett hogy az összetett shaderek készítésére már nem volt teljesen kielégítő az OpenGL vagy a DirectX által definiált egyszerű alacsony szintű assembly shader nyelve. Megjelentek magas szintű shader nyelvek is, mind a fragmens, mind pedig a vertex shaderek számára. Ezekből a legismertebbek az OpenGL Shading Language (GLSL[6,13]), a DirectX High-Level Shading Language (HLSL), valamint az NVIDIA Cg shader nyelve.

Az első fragmens shader implementációkban a kötött utasítás szám azért volt elengedhetetlen megszorítás, mert az egyes assembly utasítások egy-egy (vagy néhány esetben több) úgynevezett utasítás szlotba (instruction slot) került és mivel ezek száma kötött volt, természetesen csak annyi utasítást hajthattunk végre egy fragmensre, amennyi belefért a kártyán lévő szlotokba. Ebből adódóan azzal a problémával is szembesülhettünk, hogy nem volt lehetőség elágaztatásra vagy ugrásokra.

A videokártyák következő generációjával ez megváltozott. Szoftver oldalból tekintve ez együtt járt a GLSL 1.10 és a pixel shader 2.x megjelenésével. Ezek a hardverek már lehetőséget biztosítottak elágaztató, ugró és ciklus utasítások használatára, azonban ne felejtsük el, hogy a GPU-k még mindig párhuzamosított adatfolyam (stream) processzorok, tehát a dinamikus elágaztatás lehetősége még nem jelenti azt, hogy érdemes is használni, hisz az ilyen architektúrára épülő processzorok teljesítményét erősen lecsökkenti a dinamikus elágaztatással kapcsolatos feladatok ellátása (lást pipeline processzorok).

Idővel egyre újabb és újabb shader modellek jöttek ki, melyek további utasításokkal és lehetőségekkel egészítették ki a shader nyelveket. Ma már a 4.1-es shader modellnél tartunk, melyben elméletileg végtelen hosszú vertex és fragmens shadereket futtathatunk hardveresen, korlátlan textúra indirekciót alkalmazhatunk, valamint tetszőleges számú textúrát érhetünk el egyetlen fragmens színének meghatározása közben, valamint lehetőségünk van a fragmens Z koordinátájának módosítására is (ami viszont meglehetősen csökkenti a programunk teljesítményét, lásd később).

5. Alkalmazási területek

A 3D grafikus kártyák hatalmas fejlődési ütemének köszönhetően a shader nyelveket egyre szélesebb körben alkalmazzák. Alapvetően a következő három alkalmazási terület a legelterjedtebb:

1. **Fotorealisztikus megjelenítés** – Háromdimenziós környezetek, világok fotorealisztikus megjelenítése. Ez alatt azt értve, hogy az illető világot a lehető legvalósághűbben, legtökéletesebben próbáljuk meg ábrázolni, természetesen kisebb nagyobb csalásokkal, trükkökkel. Az elsődleges cél a való világ optikailag hű lemásolása.
2. **Nonfotorealisztikus megjelenítés** – Speciális megjelenítési technika, mely során nem valósághű képeket állítunk elő. Ide tartozik például az animációs filmek, rajzfilmek készítése, bizonyos tudományos célú képek, animációk előállítása, valamint szürrealisztikus világok megjelenítése.
3. **Általános célú GPU programozás**[19] – Elrugaszkodva a grafikus kártyák rendeltetésszerű használatától, ahelyett hogy képeket állítunk elő vele, egyszerűen matematikai feladatmegoldásra alkalmazzuk azt, legyen az egyszerű vektortranszformációk végrehajtása vagy mondjuk genetikus algoritmusok implementálása a GPU-n.

Az alkalmazási területeket más megközelítésből is osztályozhatjuk. Tekintve például a felhasználási területet, többek között megemlíthetjük a következőket:

- Szórakoztató ipar: filmek, számítógépes játékok és egyéb szórakoztató ipari termékek készítése.
- Orvostudomány: képelemzés, képfeldolgozás, leletek háromdimenziós ábrázolása, sebészeti szimulációk valós idejű fotorealisztikus megvalósítása, stb.
- Építészet: háromdimenziós tervrajzok, látványtervek megjelenítése, stb.
- Fizika: fizikai számítások, rigid body simulator, stb.

5.1. Fotorealisztikus megjelenítés

A fotorealisztikus megjelenítés olyan képek előállításának művészete és tudománya, melyek imitálják a való világot annyira amennyire csak lehet. Jó néhány alacsony- és magas szintű eszköz áll manapság a fejlesztő/művész rendelkezésére, melyek segítségével mindezt elérheti. Tipikusan a fotorealisztikus megjelenítésnek, akár csak a nonfotorealisztikus- és egyéb megjelenítési módszereknek, két fő típusa van:

- **Offline megjelenítés**: Ekkor a képek előállítása egy előfeldolgozási lépésnek tekinthető, mely eredményeképpen előáll például egy video, mely ezután megtekinthető. Mivel ez az előfeldolgozási szakasz elméletileg korlátlan idő és erőforrás igényű lehet, szinte bármilyen bonyolult algoritmust és technikát felhasználhatunk a képeink előállításához, azonban minél összetettebb a feladat, annál tovább tart a végtermék elkészítése. Tipikusan a CPU végzi a számításokat.
- **Valós idejű (real-time) megjelenítés**: A képek előállítása valós időben, tehát a képek megtekintésével szimultán és párhuzamosan történik. Mivel az ember szeme akkor lát folyamatosnak egy animációt, ha legalább 24 képkockát lát abból másodpercenként, természetesen ebben az esetben a kép előállításának az időigénye meglehetősen korlátozott (miliszekundum nagyságrendű). Ebből kifolyólag egyszerűbb algoritmusokat és technikákat kell alkalmazzunk a megjelenítéshez. Tipikusan a GPU végzi a számításokat. Mára már köszönhetően a videokártyák hatalmas teljesítményének, illetve az egyes gyakran használt megjelenítési technikák hardveres implementációjának, közel hasonló eredményeket érhetünk el a a valós idejű megjelenítéssel, mint az offline technikák segítségével.

Mind az offline, mind pedig a valós idejű fotorealisztikus megjelenítés legnagyobb problémája a lokális és globális megvilágítási és fénytükröződési jelenségek valósághű szimulációja. Elsősorban az offline megjelenítésben ennek a problémának a megoldására a következő két technikát alkalmazzák:

1. **Sugárkövetés (ray-tracing)** – Egy technika, mely a képsík egyes pixelein keresztül követi a fény útját és így állítja elő a megvilágított képet. Ez a technika magas szintű fotorealizmussal jeleníti meg a számítógéppel modellezett világot, azonban sokkal nagyobb a számítás igénye, mint a hagyományos scanline megjelenítési technikáknak. A technika tökéletesen alkalmas olyan optikai jelenségek szimulálására, mint például a fényvisszaverődés (reflection), fénytörés (refraction), fényszóródás (scattering) vagy a színtorzulás (chromatic aberration).
2. **Radiozitás (radiosity) számítás** – Egy globális megvilágítási technika, mely elsősorban diffúz felületek megvilágítását tudja jó minőségben szimulálni. Tipikusan ezzel a technikával állítják elő a valós idejű megjelenítésben használt lightmap-eket.

Ezen felül vannak hibrid megoldások is, melyek a két technikát együttesen használják, hogy segítségükkel a való világ optikai jelenségeit szinte száz százalékosan imitálják. Azonban mivel mindkét technikához tartozó algoritmus meglehetősen időigényes, a valós idejű megjelenítésben egyenlőre még nem nagyon használják, viszont egyes spekulációk alapján nem sokára ez is lehetségessé válik.

A valós idejű megjelenítésben tipikusan különböző trükkökkel, technikákkal érik el a közel valósághű képeket. Ezen technikák közül érdemes megemlíteni a következőket, melyeket később bővebben is ismertetek:

1. **Bump mapping** – Felületek kisebb egyenetlenségének szimulálására szolgáló technika. Különböző implementációi léteznek, kezdve az egyszerű közelítésektől összetett, nagyobb egyenetlenségek imitálására alkalmas technikákig: emboss bump mapping, environment bump mapping, normal mapping, parallax mapping, parallax offset mapping, relief mapping, cone step mapping, stb.
2. **HDR (High Dynamic Range) megjelenítés** – Ennek a technikának a segítségével lehetőségünk van az emberi szem dinamikus fényerősség tartomány érzékelésének valósághű szimulálására.
3. **Motion blur** – A gyorsan mozgó tárgyak képének homályosságát próbálja imitálni, szokták bemozdulásos életlenségnek is fordítani.
4. **Depth of field** – Az ember szeme egyszerre csak egy távolságra tud fókuszálni, tehát a fókusztávolságtól távolabb, vagy annál közelebb lévő tárgyak elmosódottnak tűnnek. Mivel a számítógép képernyője fix távolságra van szemünktől, függetlenül attól, hogy a képernyőn megjelenő kép egy háromdimenziós világot szeretne ábrázolni. A depth of field technika segítségével ezeket, a szem változó fókusz távolságából adódó életlenségeket hivatott utánozni.

5.2. Nonfotorealisztikus megjelenítés

A nonfotorealisztikus megjelenítés (NPR – Non-Photorealistic Rendering) a számítógépes grafika azon területe, ami a művészet és kifejezési stílusok digitális megjelenítésére koncentrál. Az NPR mint fogalom 1994-ben került be a köztudatba David Salesin és George Winkenbach által. Ellentétben a hagyományos komputergrafikával, ami a valósághűségre fekteti a hangsúly, a nonfotorealisztikus megjelenítés a festményekből, rajzokból, rajzfilmekből és a tudományos illusztrációkból meríti a ihletet.

A háromdimenziós NPR alkalmazása tipikusan a videojátékok, filmek, rajzfilmek területén elterjedt. Általában ugyanazokat a háromdimenziós geometriai modelleket alkalmazzák, mint az a

fotorealisztikus megjelenítésnél teszik, azonban a nonfotorealisztikus megjelenítésben olyan anyagokat és megvilágítási modelleket használnak, amelyek valamilyen speciális művészi látványt, végeredményt produkálnak, nagyon gyakran az ilyen technikák arra szolgálnak, hogy a háromdimenziós képet úgy jelenítsék meg, hogy az két dimenziós, sík képként hasson. A számítógépes játékok és rajzfilmek esetében használt NPR technikákat tipikusan úgynevezett toon (rajzfilm) shaderekkel valósítják meg. A rajzfilmek és játékokon túl az NPR technikák nagyon hasznosak a tudományos ábrák, illusztrációk készítésekor is, mivel ilyenkor nem feltétlen szükséges vagy akár nem is célszerű a valósághűség.

Habár a nonfotorealisztikus megjelenítés, mint szakterület már több, mint egy évtizede létezik, sokáig nem vették komolyan, ami annak is köszönhető, hogy nem volt mögötte egy konzisztens elméleti háttér, nem volt jól definiált az, hogy mi is tartozik ebbe a szakterületbe, sőt, még az elnevezését is sokan kritikusan fogadták.

Néhány a legismertebb NPR technikák közül:

1. **Cel-shading** – Egy technika, mely segítségével a megjelenített objektumok úgy néznek ki, mintha kézzel rajzolták volna őket. A megvilágításkor az objektum színárnyalatai közül csak párat használunk fel, ezáltal egy lapos látványt keltve.
2. **Edge-detect** – Egy kép éleit meghatározó algoritmus. A meghatározás történhet például a megjelenített jelenet mélység pufferje alapján, az egyes pixelekhez tartozó normál vektorok alapján, vagy akár a kettő kombinálásával. Ez egy alapvető technika, amit szinte minden NRP technikánál alkalmaznak.
3. **Color grading** – Általános képfeldolgozó technika, mely azon alapszik, hogy minden pixel színére alkalmaz egy függvényt. Ennek a technikának egy speciális változata többek között a sepia és a szürkeskála megjelenítés.

5.3. Általános célú GPU programok

Általános célú GPU programozásnak (GPGPU vagy GPGP – General Purpose GPU Programming) nevezzük azt, amikor a videokártya processzorát (GPU), amellyel alapvetően számítógépi grafikai számításokat szoktunk végezni, olyan számítások elvégzésére használjuk, amit tipikusan a CPU szokott végezni. A GPGPU, mint fogalom és szakterület annak hatására alakult ki, hogy a videokártyák grafikus csővezetékében egyre több szakasz, lépés vált programozhatóvá, ennek köszönhetően a szoftverfejleszők kihasználhatják a GPU-k nagyon számítási teljesítményét és pontosabb aritmetikáját, hogy nem grafikai adatok adatfolyam orientált feldolgozását végezzék.

Maga a GPU-ra írt programok nem mások, mint vertex, fragmens vagy egyéb shaderek. Természetesen amikor általános célú GPU programozásról beszélünk, akkor ezek a shaderek nem egészen a rendeltetési céluk szerint vannak használva, tehát például a vertex shader nem feltétlen térbeli pontok, vertexek színét, pozícióját és egyéb paramétereit szándékozott meghatározni a bemeneti paraméterei alapján. Egy shader gyakorlatilag nevezhető egy kernelnek (mag, transzformációs kód), amely egy nagy mértékben párhuzamosított adatfolyamprocesszoron fut. Minden bemenethalmazra, függetlenül a többi bemenethalmaztól, alkalmazza ezt a kernelt, majd kimeneteket produkál.

A klasszikus értelemben vett általános célú számításokhoz a következő erőforrásokra van feltétlen szükség:

1. **Programozható processzor** – Ez tipikusan a CPU.
2. **Operatív memória** – Egy írható és olvasható memória interfészt biztosít a processzor számára. Innen jönnek a bemenetek, majd pedig ide kerülnek a program kimenetei is.

Tekintve a GPU programozást, nem beszélhetünk olyan lehetőségről, mellyel közvetlenül hozzá tudnánk férni a videomemóriához, akár írásról, akár olvasásról van szó, viszont ugyanígy itt is rendelkezésünkre állnak az általános célú számításokhoz szükséges erőforrások:

1. **Programozható processzor** – Vertex, geometria vagy fragmens shaderek, melyek a GPU megfelelő adatfolyam processzorán futnak.
2. **Vertex adatok** – Csak olvasható memória interfész.
3. **Textúrázó egységek** – Csak olvasható memória interfész.
4. **Framebuffer** – Csak írható memória interfész.

Leggyakrabban az általános célú GPU programok a textúrákat használják fel, mint bemeneti adatfolyam és fragmens shaderekkel végzik el a szükséges számításokat. Ez úgy valósítható meg, hogy egy teljes képernyős négyszöget rajzolunk ki a használt grafikus API segítségével, melynek négy csúcsa a képernyő négy sarkának felel meg. Ehhez a primitívhez hozzárendelünk egy vagy több két dimenziós textúrát, a négyszög csúcsainak megfelelő textúra koordináták pedig általában minden egyes textúrázó egység esetében a textúra négy sarkát címzi. Ennek a technikának a segítségével például szinte bármilyen képfeldolgozó algoritmust végre tudunk hajtani nagyságrendekkel nagyobb sebességgel, mint azt a CPU-val tennénk.

Az általános célú GPU programozási területek közül érdemes megemlíteni a következőket:

1. **Grid számítás (elosztott számítás)** – a GPU mint egyfajta szuperszámítógép viselkedik.
2. **Fizikai szimulációk** – elsősorban a newtoni fizikán alapuló számítások, rigid body szimulátorok megvalósítása a GPU-n.
3. **Számítógépes tomográfia készítés**.
4. **Gyors Fourier transzformáció (FFT – Fast Fourier Transform)** – segítségével implementálhatjuk a GPU-n az FFT algoritmust, mely alkalmas a diszkrét Fourier transzformáció és annak inverzének számítására.
5. **Hang és egyéb digitális vagy analóg jelfeldolgozás** – tipikusan adatfolyam orientált algoritmusokat igénylő művelet, mely hatékonyan implementálható shaderekkel.
6. **Neurális hálók** – neurális hálókkal kapcsolatos és egyéb a mesterséges intelligencia témakörébe tartozó számítások elvégzése.
7. **Kriptográfia** – kódolási algoritmusok hatékony implementációja a GPU-n.
8. **Adatsűrítés** – veszteség nélküli (loss-less) és veszteséges (lossy) tömörítő algoritmusok megvalósítása a videokártyán.

6. Megvilágítási modellek

Habár a shader nyelvek alkalmazási területe egyre inkább nő és mára már szinte bármilyen célra felhasználhatjuk a shader programokat, nem szabad elfelejtsük azt, hogy a shader nyelvek kialakulását az az igény ösztönözte, hogy minél összetettebb, valósághűbb és testreszabhatóbb eredményeket produkálhassanak a fotorealisztikus megjelenítő szoftverek. Nem hiába kapták a shader nevet, ami tükörfordításban „árnyaló”-t jelent.

Már jóval a számítógépes grafika modern kora előtt létezett jó néhány matematikai módszer, mellyel megközelítőleg szimulálni tudjuk a valódi világ optikai jelenségeit, azonban a 3D grafikus kártyák első generációi csak egy-két előre definiált egyszerű megvilágítási modellt ismertek, melyek csak konfigurálhatóak voltak. Ez többek között az akkori kártyák relatív alacsony teljesítményének és korlátozott működésének volt betudható. Ahogy a videokártyák és elsősorban a rajtuk elhelyezkedő GPU-k fejlődtek, egyre több lehetőség állt a szoftverfejlesztők rendelkezésére, mind a teljesítmény, mind pedig a programozhatóság, testreszabhatóság tekintetében. Mára már bármilyen komplex megvilágítási modellt tudunk implementálni a grafikus processzorokon, melyek közül a legtöbb a valós idejű alkalmazásban is megállja a helyét.

Ebben a fejezetben megismerkedünk a megvilágítási modellek megértéséhez szükséges alapfogalmakkal, ismertetek néhány megvilágítási modellt, anyag modellt, illetve azok implementációjáról is szó lesz.

6.1. Alapfogalmak

Amikor megvilágítási modellről beszélünk, akkor a következő két dolog együttesére gondolunk:

1. **Megvilágítás** – a fényforrások által kibocsátott fénynek az a része, amely a tér egy adott pontjába eljut (általában feltételezve, hogy semmi sem zárja el a fény útját a fényforrástól az illető pontig). Ez tipikusan egy több paraméteres függvény, melyben elsősorban a fényforrás típusa, valamint a pontnak a fényforrástól való távolsága vesz részt.
2. **Visszaverődés** – egy tárgy által visszavert fény mennyiségét határozza meg a tárgy egy adott pontján. Ez egy sokkal összetettebb függvény, mely függ többek között a tárgy anyagától, valamint a beérkező fénysugarak irányától.

Ahhoz, hogy egy felület látható legyen a szem számára, ahhoz a következő három feltételezésből legalább egynek igaznak kell lennie:

1. Van egy fényforrás, melynek fénye visszaverődik ezen a felületen.
2. A felület fényt bocsát ki.
3. A felület valamely fényforrás fényét átereszti.

Ebből adódóan a megvilágítási egyenletünk ettől a három tényezőtől függ. Az egyszerűség kedvéért viszont tételezzük fel, hogy a felületeink átlátszatlanok. Ekkor a megvilágítási egyenletünk a következő sematikus formát ölti fel:

$$Kimenő\ fény = Kibocsátott\ fény + Visszaverődési\ függvény * Beeső\ fény$$

Mivel a kibocsátott fény megint csak nem a leglényegesebb és legösszetettebb része az egyenletnek, tételezzük fel azt is, hogy a felületünk nem bocsát ki fény. Ekkor a következő egyenletet kapjuk:

$$Kimenő\ fény = Visszaverődési\ függvény * Beeső\ fény$$

Ebből az egyeletből pedig azt tudjuk meg, hogy a megvilágítási egyenletünk két tényezőtől függ. Az egyik tényező a beérkező fény mennyisége, ami a fényforrások definiálása alapján viszonylag egyszerűen meghatározható. Az egyenlet másik tényezője az úgynevezett visszaverődési függvény. Ennek meghatározása jóval összetettebb és nagyon sok formát ölthet fel. Vannak fizikán alapuló megközelítések, illetve egyszerű tapasztalati értékeken alapuló megoldások is. Mindkét esetben a visszaverődési függvény arra szolgál, hogy a beérkező fény mennyiségét egy olyan [0, 1]

intervallumbeli értékkel szorozza meg, mely végül megközelítőleg visszaadja azt a látványt, amit az eredeti fizikai felület adna.

6.2. Lambert törvénye

Az előzőleg ismertetett megvilágítási egyenlet azon a feltételezésen alapul, hogy a fény egy ideális pontba esik, mely esetében a beérkező fénymennyiség teljes intenzitásával hat az illető pontra. A valóságban természetesen ez nem így van, mivel bármilyen pontossággal is számolunk, a fény akkor is minden esetben egy nagyon kicsi, de mérhető telületre esik. Az ideális ponttal való közelítés megfelelő, amikor a fényforrásból érkező fénymennyiség csillapítását számoljuk (ami a pontnak a fényforrástól vett távolságától függ), de ebben az esetben nem. Ez abból adódik, hogy egy pont megvilágítottsága (illiuminance) nem más, mint az egységnyi felületre eső fényenergia mennyisége, ami ideális pont esetében nem értelmezhető. Geometriai és trigonometriai számításokkal bizonyítható, hogy egy fény nyaláb által megvilágított felület mérete nő a felület normálisa és a fényforrástól az illető pontba mutató vektor közötti szöggel, emiatt a fény nyaláb energiája nagyobb felületen oszlik el, tehát a felület adott pontjában a megvilágítottság csökken, ahogy az a 6.1. ábrán látható. Lambert törvénye adja meg számunkra a matematikai összefüggést a beesés szöge és a megvilágítottság között:

$$Megvilágítottság = Beérkező\ fény * \cos(\theta)$$

Ahol θ a beeső fény iránya és a felület normálisa által bezárt szög. Beépítve ezt a megvilágítási egyenletünkbe a következőt kapjuk:

$$Kimenő\ fény = Visszaverődési\ függvény * Beérkező\ fény * \cos(\theta)$$

6.3. Kétirányú visszaverődés eloszlási függvény

A kétirányú visszaverődés eloszlási függvény (BRDF – Bidirectional Reflectance Distribution Function[8]) egy négy dimenziós függvény ami meghatározza hogy hogyan verődik vissza a fény egy átlátszatlan felületről. A függvénynek két paramétere van: a beérkező fény iránya (ω_i) és a kimenő irány (ω_o), mindkettő a felület normálisához képest van megadva. A függvény eredménye a ω_o vektor irányába visszavert fénymennyiség és a ω_i irányából érkező megvilágítottság (illuminance) aránya. Ennek alapján a BRDF függvények a következőképpen írhatóak fel matematikailag:

$$BRDF: \mathbb{R}^4 \times \mathbb{R}^4 \to [0,1]$$

$$BRDF(\omega_i, \omega_o) = \frac{L_r}{L_i}$$

Habár meghatározás szerint az így definiált függvényeket nevezzük BRDF-eknek, egy BRDF nem feltétlen csak a fent említett paraméterektől függhet, például bizonyos BRDF-ek eredménye függhet a hullámhossztól, így egy újabb paraméterrel bővül a függvényünk.

A legtöbb valódi anyag optikai viselkedése így például a fa, a kő vagy a csiszolt fém, nem modellezhető egy egyszerű BRDF függvénnyel, azonban általában egy ilyen függvény megadásával nagyon jól lehet közelíteni a valódi anyagok látványát.

További bonyodalmakat okozhatnak az úgynevezett anizotróp anyagok. Az anizotróp anyagok tükröződési mértéke függ attól is, hogy a nézőpont a felület normálisához képest milyen irányban van. Habár valójában minden valódi anyag hordoz magában bizonyos mértékű anizotróp viselkedést, nagyon sok anyag esetében ennek mértéke elhanyagolható. Ezeket nevezzük izotróp anyagoknak. Vannak azonban erősen anizotróp anyagok, mint például a csiszolt fém, mely estében az anizotrópia figyelmen kívül hagyása gyenge eredményekhez vezethet.

Habár a BRDF-ek nagyon jól definiálják egy adott anyag optikai viselkedését, a számítógépi grafikában használt megvilágítási/árnyalási modellek (shader model) ezeknél általában sokkal inkább ad hoc módon próbálják utánozni a valódi anyagok viselkedését.

6.4. Árnyalási módszerek

Eddig arról beszéltünk, hogy hogyan határozzuk meg a felületekről visszaverődő fénymennyiséget, de mielőtt a konkrét technikákat, módszereket vennénk sorra, meg kell értenünk, hogy a kiszámított megvilágítottsági értékeket hogyan használjuk fel a primitívek (tipikusan háromszögek) kifestéséhez.

6.4.1. Flat shading

A flat shading a legegyszerűbb árnyalási módszer, ami háromszögenként egy normál vektorral dolgozik. Ezen normál vektor, illetve maga a háromszög pozícióját felhasználva kiszámítja a háromszög által visszavert fénymennyiséget, majd egy egyszínű háromszöget eredményez. A technika nevezhető háromszögönkénti árnyalásnak (per-triangle shading) is.

Előnye, hogy háromszögenként csak egyszer kell kiszámolni a megvilágítási egyenletet, továbbá köszönhetően annak, hogy az egész háromszög ebből fakadóan monokróm lesz, a háromszög kifestése minimális számítást igényel.

Hátránya azonban hogy ahhoz, hogy viszonylag realisztikus eredményt kapjunk, az objektumaink rengeteg apró háromszögből kell hogy álljanak, különben hiába jó a megvilágítási egyenletünk, nem felelünk meg annak a megszorításnak, hogy a megvilágítási egyenletet egy pontszerű felületre alkalmazzuk. Természetesen ez szinte lehetetlen, mivel egyrészt hatalmas mennyiségű vertex definiálására lenne szükség, másrészt mind a mai napig a videokártyák szűk keresztmetszete a feldolgozható vertexek száma, tehát teljesítmény szempontjából sem célszerű ez a módszer. Másrészt görbe felületeket képtelenség lenne szimulálni, mivel abból adódóan hogy minden háromszög egy normálissal rendelkezik, a görbe felületek, mint például egy gömb, gyémántszerűnek, töredezettnek tűnnek.

6.4.2. Gouraud shading

A Gouraud árnyalási módszert Henri Gouraud publikálta 1971-ben. Egy olyan módszert keresett, mely segítségével szimulálni lehet a számítógépes grafikában a fény és a színek különböző hatásait egy felületen. Alapvetően Gouraud technikája abból áll, hogy a megvilágítási egyenletet egyenként alkalmazzuk a vertexekre, melyek saját, a háromszög síkjának normálisától akár teljesen különböző normál vektorokkal rendelkezhetnek. Ez által minden vertexhez rendeltünk egy megvilágítottsági értéket avagy színt, majd a háromszögek raszterizálása folyamán ezeket az értékeket interpoláljuk a háromszög mentén. A technikát szokás még per-vertex árnyalásnak (per-vertex shading) is nevezni.

Habár ez a technika már valamennyivel lassabb, mint a flat shading, a megvilágítási egyenletet maximum háromszor annyiszor kell alkalmazni (ideális esetben, például egy gömb árnyalásakor ugyanannyiszor, mint a flat shading esetében), valamint a színek interpolálása sem jelent sokkal több műveletet a raszterizáláskor, ugyanakkor az eredmény messzemenően jobb az előző technikáétól. További előnye, hogy a görbe felületek látványát is elég jól lehet közelíteni, ha a vertexek normálja nem az őt tartalmazó háromszög síkjához képest, hanem az elméleti görbéhez képest kerül meghatározásra.

A Gouraud shading jól alkalmazható diffúz megvilágítási modellek esetében, ha a háromszögek vertexei közötti távolság jóval kisebb, mint a fényforrás „hatótávolsága” (az a távolság, ahova még mérhető fénymennyiséget juttat el), azonban szükséges tesszeláció hiányában a csúcsfényeket, illetve egyéb éles visszatükröződéseket gyengén imitálja.

6.4.3. Phong shading

A Phong shading[TN2] kifejezés több komputergrafikai technikára is utal. Esetünkben arról a technikáról van szó, amelyet Phong interpolációnak is szoktak nevezni. A technika Bui Tuong Phong

1973-ban publikálta és a következőképpen működik: ahelyett hogy a megvilágitási egyenletet a vertexekre alkalmazná és annak eredményét interpolálná, mint ahogy azt Gouraud technikája teszi, a vertexekhez tartozó normál vektorokat interpolálja a háromszög mentén és minden pixelre alkalmazza a megvilágítási egyenletet felhasználva az interpolált normál vektort. Emiatt szokták a Phong interpolációt per-pixel árnyalásnak (per-pixel shading) is nevezni.

Mivel a megvilágítási egyenlet kiértékelése minden egyes pixelre megtörténik, ez az árnyalási módszer produkálja a lehető legvalósághűbb eredményt. A legtöbb megvilágítási modell implementációjakor, de legfőképpen a tükörszerű felületek megjelenítésekor ezt a technikát alkalmazzák.

A bemutatott három technika közül azonban a Phong interpoláció messze számítás igényesebb, mint a másik kettő, hisz általában a pixelek száma nagyságrendekkel nagyobb a vertexek számánál, így aztán jóval többször kell alkalmazni a megvilágítási modellt. Ezért gyakran a per-pixel árnyalás használatakor egyszerűbb megvilágítási modelleket szoktak alkalmazni, illetve nem ritkák a hibrid megoldások is, melyek esetében bizonyos számítások, mint például a diffúz megvilágítás, vertexenként kerül kiszámításra, a csúcsfények pedig, amelyek esetében a pontosság sokkal lényegesebb, pixelenként kerülnek kiszámításra.

További elengedhetetlen előfeltétele a pixelenkénti árnyalás alkalmazásának az, hogy a grafikus hardver támogasson valamilyen fragmens/pixel shadert vagy legalább is valamilyen összetettebb textúrázó mechanizmust.

6.5. Diffúz anyagok

Vannak anyagok, így például a matt festék, melyek a beérkező fényt egyenletesen szórják szét minden irányban. Lambert törvénye kimondja hogy a megvilágítottság mértéke függ a beérkező fény irányától és a felület normálisától, azonban nem mond semmit a visszavert fény mennyiségéről. Matt festék esetében például bármely irányból nézzük a felületet, a felületről visszaverődött fény nagyjából ugyanaz minden esetben. Ez amiatt van, hogy a festék nagyon durva felületű, ezért a beeső fény szinte minden irányban ugyan olyan mértékben verődik vissza, tehát a nézőpont változtatása nem nagy mértékben módosítja az érzékelt fénymennyiséget. Habár tökéletesen diffúz anyagok nem léteznek, nagyon sok anyag optikai viselkedése jellemezhető és viszonylag realisztikusan visszaadható egy teljesen diffúz megvilágítási modellel.

6.5.1. Lambert modell

A legegyszerűbb diffúz anyag modell a Lambert féle megvilágítási modell, aminek a Lambert féle törvény az alapja. A Lambert féle modell esetében a megvilágítási egyenletünk a következő alakban írható fel:

$$Kimenő\ fény = Beérkező\ fény * \cos(\theta)$$

Ahol θ a beeső fény iránya és a felület normálisa által bezárt szög. Ez a legegyszerűbb diffúz megvilágítási modell, melyet előszeretettel használtak a háromdimenziós grafika terén, főképp annak korai fejlődési szakaszában, amikor még nem volt megfelelő hardver az összetettebb modellek implementálására.

A modell egyszerűen implementálható az OpenGL magas szintű shader nyelvében, akár vertexenkénti árnyaláshoz (egy vertex shader segítségével), akár pedig pixelenkénti megoldáshoz (egy fragmens shader segítségével). Mivel mára már nem luxus minden technika megvalósításához pixelenkénti megvilágítást alkalmazni, lássuk hogy néz ki a per-pixel Lamber féle modellt implementáló GLSL shader:

```
[Vertex_Shader]
varying vec3 normal, lightDir;
```

```
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
}

[Fragment_Shader]
varying vec3 normal, lightDir;
void main() {
    float lum = dot(normalize(normal), normalize(lightDir));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

A shader az OpenGL beépített 0-ik fényforrását használja, mint fehér fényt kibocsátó pontszerű fényforrás. A shader nem alkalmaz sem csillapítást sem egyetlen további paramétert a megvilágítás számításakor, egyedül Lambert törvényét alkalmazza.

Észrevehető hogy a fragmens shaderben mind a normál vektort, mind pedig a beeső fény irányát normalizáljuk. Ez szükséges, mivel még akkor is, ha a vertex shader futásakor egység hosszúak ezek a vektorok, hisz az interpoláció folyamán rövidülés léphet fel, ami elrontja a megvilágítási modellünk pontosságát.

6.5.2. Oren-Nayar modell

Az Oren-Nayar féle diffúz megvilágítási modell olyan anyagok árnyalására alkalmazzák, melyek nem teljesen egyenletesen szórják szét a beérkezett fénymennyiséget. Néhány durva felület, mint például a föld az agyag vagy egyes textilek nagyobb mennyiségben vernek vissza fényt a beeső fény irányába. Ezt a jelenséget önvisszaverődésnek (retroreflection) nevezik. Az említett anyagok ezen tulajdonsága egy laposabb érzetet kelt a Lambert féle modellénél, különösen ha a nézet iránya megegyezik a beeső fény irányával. Szintén hasonló vizuális élményt kelt a hold, amely szintén egy laposabb látszatot keltő diffúz anyagú objektumnak tekinthető. Az Oren-Nayar féle modell ezt a sajátosságot próbálja szimulálni. A megvilágítási modell matematikai egyenlete a következő:

$$Kimenő\,fény = (\mathrm{N} \cdot \mathrm{L}) * \left(A + B * \sin(\alpha) * tan(\beta) * MAX(0, \cos(C))\right) * Beérkező\,fény$$

Ahol az $\mathrm{N} \cdot \mathrm{L}$ tényező nem más, mint Lambert féle összetevő. N a felület normálisa, L a beeső fény iránya. A és B két tényező, melyek az Oren-Nayar modellben szereplő durvasági paramétertől (σ) függnek:

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33} \qquad B = 0.45\frac{\sigma^2}{\sigma^2 + 0.99}$$

Továbbá:

$$\alpha = MAX(\mathrm{acos}(\mathrm{N} \cdot \mathrm{L}), \mathrm{acos}(\mathrm{N} \cdot \mathrm{V})) \qquad \beta = \mathrm{MIN}(\mathrm{acos}(\mathrm{N} \cdot \mathrm{L}), \mathrm{acos}(\mathrm{N} \cdot \mathrm{V}))$$

Ahol V a nézet vektor és C pedig a nézet vektor és a beeső fény iránya közötti azimuth-i szög. Ez azt jelenti, hogy a normál vektor által meghatározott síkra levetítjük a nézet vektort, illetve a beeső fény irányvektorát. Ez a két vektor által bezárt szög lesz a C.

Amikor az Oren-Nayar modellt szeretnénk implementálni, érdemes figyelembe venni, hogy az A és B tényezők nem függnek sem a geometriától, sem a beeső fény irányától, sem pedig a nézet vektortól, tehát ezeket célszerű már az alkalmazásunkban kiszámolni, kivéve ha a σ értéke pixelenként változhat. Ebben az esetben egy textúrában érdemes ezt a plusz információt tárolni. Lássuk akkor az Oren-Nayar diffúz megvilágítási modell GLSL implementációját:

```
[Vertex_Shader]
varying vec3 normal, lightDir, viewDir;
void main() {
```

```
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
}

[Fragment_Shader]
uniform float A, B;
varying vec3 normal, lightDir, viewDir;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    float dotNL = dot(N, L);
    float dotNV = dot(N, V);
    float acosNL = acos(dotNL);
    float acosNV = acos(dotNV);
    float alpha = max(acosNL, acosNV);
    float beta = min(acosNL, acosNV);
    vec3 lightPlaneProj = L - N * dotNL;
    vec3 viewPlaneProj = V - N * dotNV;
    float cosC = dot(normalize(lightPlaneProj), normalize(viewPlaneProj));
    float lum = dotNL * (A + B * sin(alpha) * tan(beta) * max(0.0, cosC));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

Ahogy látható a többször használt értékeket, mint például a nézet vektor és a beeső fény irányának, valamint a felület normálisának a skaláris szorzatát külön letároljuk az optimalizálás érdekében. Általában a videokártya driverekbe beépített shader fordítók automatikusan optimalizálják az elkészített shaderek kódját, azonban ha biztosra akarunk menni, akkor ezt mi magunk tesszük meg. Természetesen azt nem lehet garantálni, hogy ezek az értékek tényleg letárolódnak és nem kerülnek többszörös kiszámításra, hisz hardver implementáció függő, hogy hány regiszter áll rendelkezésünkre a fragmens shaderekben, azonban általában a modern hardverek esetében emiatt nem kell aggódnunk.

Ami talán magyarázatra szorul, az a cos (C) kiszámításának módja. A `lightPlaneProj` és a `viewPlaneProj` változók a fény irányvektorának és a nézet vektornak a felület normálisa által meghatározott síkra képezett vetületeit jelöli. Látható, hogy ezek kiszámításánál a sík egyenlete helyett egyszerűen csak a normál vektort használtuk, amit megtehetünk, hisz a sík áthalad az origón ez esetben és így a síkot meghatározó egyenlet negyedik együtthatója 0 lesz, a másik három pedig rendre megegyezik a normál vektor koordinátáival.

6.5.3. Minnaert modell

Habár a Marcel Minnaert által kifejlesztett megvilágítási modellnek nem ez volt az eredeti célja, a gyakorlatban nagyon hasznosnak bizonyult a bársony anyagának utánzásában. Eredetileg azt a látványt szerette volna elérni, hogy a hagyományos diffúz anyag modellektől eltérően az objektumok szélei minden esetben sötétebb árnyalatokat kapjanak. Az eredmény egy olyan látványt sugall, mintha a megjelenített objektum bársonnyal vagy hasonló finom anyaggal lenne bevonva. A Minnaert féle diffúz árnyalási modell szintén nem nézet független, a megvilágítási egyenlet a következő alakban írható fel:

$$Kimenő\ fény = \left((N \cdot L) * (N \cdot V)\right)^{m-1} * Beérkező\ fény$$

Ahol az N, L és V ezúttal is rendre a felület normálist, a fény irányvektorát és a nézet vektort jelöli, az m paraméter pedig egy tapasztalati tényező. Kísérletezve ennek a paraméternek a különböző

értékeivel, elérő eredményeket kapunk. A Minnaert féle modell GLSL implementációja ez alapján a következő:

```
[Vertex_Shader]
varying vec3 normal, lightDir, viewDir;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
}

[Fragment_Shader]
uniform float m_minus_one;
varying vec3 normal, lightDir, viewDir;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    float lum = pow(dot(N, L) * dot(N, V), m_minus_one);
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

Ahogy látható, célszerűen az m tényező helyett rögtön az $m-1$ értéket adjuk át paraméterként a shadernek, hisz nincs értelme minden egyes fragmens esetében ugyanazt a plusz egy kivonást elvégezze a GPU. A vertex shaderünk ugyanaz, mint az előző esetben, tipikusan nem is nagyon fog változni az egyes megvilágítási modellek tekintetében, hisz a legtöbb megvilágítási egyenletnek csak erre a három paraméterre van szüksége (N, L és V).

6.6. Tükörszerű anyagok

Nagyon sok anyag a durva felületéből adódóan nagyjából egyenletesen szórja szét minden irányban a beérkező fénymennyiséget. Vannak azonban olyan anyagok, melyeknek a felülete közel tökéletesen egyenletes, ebből adódóan a fényvisszaverődés törvényéből következetve kiderül, hogy a beeső fény nagy részét egy adott irányba veri vissza a felület. Az ilyen anyagú tárgyak, melyek esetében észlelhetőek az úgynevezett csúcsfények, tükörszerű anyagoknak nevezzük. Az ezen tulajdonsággal rendelkező anyagokat, elsősorban a fém és a különböző műanyag felületeket nem lehet élethűen visszaadni egy diffúz megvilágítási modell segítségével.

A tisztán tükörszerű anyagokat nevezhetjük a tisztán diffúz anyagok ellenkezőjének. Ahogy egy tisztán diffúz anyag tökéletesen egyenletesen veri vissza minden irányba a beérkező fényt, egy tökéletesen tükröződő felület teljesen a fényvisszaverődés törvénye által megkapott visszaverődési irányba veri vissza a beeső fénysugarakat. A tükör jó példa majdnem teljesen tükröződő anyagra, azonban a valódi tükrök sem számítanak tökéletesen tükröződő felületnek. Ez letesztelhető azzal is, hogy egy erős, fókuszált lámpát fordítunk a tükör irányába és ez esetben látható, hogy a fény nem csak a tökéletes visszaverődés irányában tér vissza a tükör felületéről.

Ahogy a korábbi technikák, az ebben az alfejezetben ismeretetettek is csak közelítések, egyes anyagok optikai viselkedésének modellezésére alkalmasak másokéra nem, azonban jó kiindulási pontot adhatnak akár egy saját fémes tükröződésű anyag árnyalási modelljének elkészítéséhez.

6.6.1. Phong modell

A valódi anyagok esetében a csúcsfény a tökéletes visszaverődési szög mentén és ahhoz közel is érzekelhető. A nagyon tükröződő felületek esetében csak egy kis szögtartományban érzékelhető a

csúcsfény, más, kevésbé tükröződő anyagoknál pedig jóval nagyobb ez a szögtartomány. Mindezt matematikailag kifejezve ezt a jelenséget egy olyan egyenlettel lehet jellemezni, mely tekinti a tökéletes visszatükröződési irány és a nézet vektor közötti távolságot, különbséget, majd ezt valamilyen hatványra emeli és ez alapján határozza meg a csúcsfény mértékét. Ezen alapszik Bui Tuong Phong 1975-ben kifejlesztett megvilágítási egyenlete is:

$$Kimenő\ fény = (R \cdot V)^n * Beérkező\ fény$$

Ahol n a csúcsfény hatvány (specular exponent), R pedig a tökéletes visszatükröződés irányvektora, melyet a következőképpen számíthatunk ki:

$$R = 2N(L \cdot N) - L$$

A többi jelölés megegyezik a korábban használtakkal. A Phong féle tükörszerű anyagoknál alkalmazott megvilágítási modellt szintén szokták Phong shadingnek nevezni, akár csak a Phong interpolációs technikát, ezért érdemes elkerülni az előbb említett elnevezést. A Phong modell per-pixel implementációja az OpenGL magas szintű shader nyelvében a következőképpen néz ki:

```
[Vertex_Shader]
varying vec3 normal, lightDir, viewDir;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
}

[Fragment_Shader]
uniform float n;
varying vec3 normal, lightDir, viewDir;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    vec3 R = vec3(2.0, 2.0, 2.0) * N * dot(L, N) - L;
    float lum = max(0.0, pow(dot(R, V), n));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

Természetesen a fenti shader kizárólag a csúcsfény meghatározására alkalmas. Tipikusan minden tükörszerű anyag árnyalási modellt ötvözni szoktak egy diffúz megvilágítási modellel, például a Phong modellt a Lambert féle modellel együtt használják. Ez esetben a diffúz és spekuláris összetevőket egy-egy együtthatóval látják el és így a két érték lineáris kombinációjaként áll elő a végső megvilágítási érték. Ezek az együtthatók természetesen nem feltétlen kell skalár értékek legyenek, sőt, a legtöbb három vagy négy elemű vektorokként adják meg őket, ezzel akár a diffúz és a spekuláris összetevők színét is manipulálni lehet (például a műanyagok esetében a szórt fény megegyezik a műanyag színével, a csúcsfények azonban általában fehérek).

6.6.2. Blinn-Phong modell

Habár a Phong modell nagyon jól közelíti a legtöbb anyag optikai jellemzőit, valamint a megvilágítási egyenlete is viszonylag egyszerű, a valós idejű háromdimenziós grafikai alkalmazások esetében sokáig nem volt célszerű használni, hisz a tökéletes visszaverődés irányának meghatározása meglehetősen számítás igényes, ráadásul ezt a számítást minden fragmens esetében végre kell hajtani. James Blinn két évre a Phong által készített megvilágítási modell publikálása után kidolgozott egy módszert, hogy hogyan egyszerűsítse Phong egyenletét úgy, hogy minimális veszteség árán sokkal hatékonyabbá

tegye azt. Ahogy már mondtam, Phong egyenletében a tökéletes visszaverődés irányának meghatározása a legköltségesebb számítás, mely ahhoz szükséges, hogy meghatározzuk az $R \cdot V$ belső szorzatot. Blinn tapasztalati eredmények alapján arra a következtetésre jutott, hogy az előbb említett skaláris szorzat eredménye a legtöbb esetben közel ugyanazt az eredményt szolgáltatja, mint az $N \cdot H$ belső szorzat, ahol N a felület normál vektora, H pedig az úgynevezett félszög vektor (half angle vector, half vector), amely a nézet vektor és a fény beesési irányának átlagaként áll elő. Ezek alapján az úgynevezett Blinn-Phong modell matematikai megfogalmazása az alábbi módon írható fel:

$$Kimenő\ fény = (N \cdot H)^n * Beérkező\ fény \qquad H = \frac{L + V}{2}$$

Az elméleti háttér azt sugallja hogy csak annyit értünk el, hogy a tökéletes visszaverődési irány meghatározásához szükséges négy műveletet kettőre csökkentettük a félszög vektor segítségével. A gyakorlatban ennél sokkal több haszna van a Blinn-Phong modellnek[TN2]: a félszög vektort, ellentétben a tökéletes visszaverődési irányvektorral, még az összetett megjelenítési technikáknál is, mint például a bump mapping is elég vertexenként kiszámolni és a fragmens shaderben elég az interpolált értéket használni. Ez azt jelenti, hogy a négy per-fragmens művelet helyett csak két per-vertex műveletre van szükségünk, valamint a nézet vektorra sincs szükség a fragmens shaderben, így aztán egy normalizálással is csökken a szükséges számítások mennyisége, ez pedig nagy mértékű sebesség növekedést jelent, természetesen a kép minőség rovására. Ez alapján lássuk a Blinn-Phong csúcsfény shadert:

```
[Vertex_Shader]
varying vec3 normal, halfv;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    vec3 viewDir = -vertex;
    halfv = (lightDir + viewDir) * vec3(0.5, 0.5, 0.5);
}

[Fragment_Shader]
uniform float n;
varying vec3 normal, halfv;
void main() {
    vec3 N = normalize(normal);
    vec3 H = normalize(halfv);
    float lum = pow(dot(N, H), n);
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

A programozható shaderek megjelenése előtt a grafikus csővezetékben akár szoftveresen, akár pedig hardveresen implementált megvilágítási modellt a legtöbb esetben a Lambert féle diffúz megvilágítás és a most bemutatott Blinn-Phong technika együttes használatának segítségével valósították meg. Természetesen szükséges hardver támogatás hiányában akkoriban mindezt egy per-vertex szintű megoldással valósították meg, minek köszönhetően, ahogy már korábban is volt szó róla, a csúcsfények a szükséges tesszeláció hiányában nem voltak minden igényt kielégítőek. Szintén ez volt az első csúcsfényeket is megjelenítő technika, amit fragmens shaderekkel implementáltak valós idejű alkalmazásokban.

6.6.3. Schlick modell

A Schlick modell tovább optimalizálja Phong modelljét annak valós idejű alkalmazásához. A technika hasonló eredményeket produkál, mint az előző két megvilágitási modell által előállított csúcsfények anélkül, hogy hatványozást végezne. Ez azért volt szükséges, mert a negyedik generációt megelőző 3D grafikus kártyák nem voltak képesek hatványozási műveletek végrehajtására, ezért ugyanazt az eredményt vagy többszöri szorzással vagy pedig segéd textúrák használatával tudták csak elérni. Értelemszerűen ez meglehetősen időigényes és körülményes volt. A Schlick féle spekuláris megvilágítási egyenlet a következő:

$$Kimenő\ fény = \left(\frac{R \cdot V}{n - \big((n-1)(R \cdot V)\big)}\right) * Beérkező\ fény$$

Az n paraméter megegyezik a Phong és a Blinn-Phong modelleknél megismert fényességi együtthatóval, R pedig ez esetben is a tökéletes fényvisszaverődés irányvektora. Ez alapján a megvilágítási modellt implementáló shader így néz ki:

```
[Vertex_Shader]
varying vec3 normal, lightDir, viewDir;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
}

[Fragment_Shader]
uniform float n, n_minus_one;
varying vec3 normal, lightDir, viewDir;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    vec3 R = vec3(2.0, 2.0, 2.0) * N * dot(L, N) - L;
    float dotRV = dot(R, V);
    float lum = max(0.0, dotRV / (n - n_minus_one * dotRV));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

Természetesen ennek az algoritmusnak a további optimalizálására is alkalmazhatjuk Blinn technikáját, mi szerint helyettesítjük az $R \cdot V$ tényezőt az $N \cdot H$ belső szorzattal. Ha ezt is megtesszük, akkor egy olyan megvilágítási modellt kapunk, ami már implementálható olyan videokártyákra is, melyek nem rendelkeznek semmilyen pixel vagy fragmens shaderrel.

6.6.4 Ward modell

A korábban bemutatott Phong modellt és ebből adódóan a Blinn-Phong és a Schlick modellt is gyakran kritizálták amiatt, hogy az ezen megvilágítási módszer segítségével árnyalt objektumok műanyagszerű látszatot keltenek[8]. Ez nagy részt igaz is volt, hisz a Phong modell elméleti háttere elsősorban a műanyag és hasonló felületek optikai jellemzőire épít. Ahhoz hogy fém és egyéb fényes felületeket imitálhassunk a virtuális világunkban, más módszerek bevezetésére van szükség.

A Ward féle modell több szempontból is érdekes. Egyrészt fizikailag sokkal pontosabb, mint a Phong féle modell, így akár egy igazi BRDF függvénynek is nevezhető. Másrészt a segítségével lehetőségünk van mind izotróp, mind pedig anizotróp felületek szimulálására, amire az eddig bemutatott technikák nem voltak képesek. A megvilágítási modell diffúz összetevőjeként ez a technika

is Lambert törvényét alkalmazza, úgyhogy most mi csak a spekuláris összetevőről fogunk csak beszélni. A Ward féle csúcsfény számítási módja jóval komplexebb az eddig bemutatott technikáknál, ezért sokáig nem nagyon használták a valós idejű megjelenítés területén:

$$Kimenő\ fény = \left(\frac{e^{-\frac{\tan^2 \mathrm{acos}(N \cdot H)}{\sigma^2}}}{2\pi\sigma^2\sqrt{(N \cdot L)(N \cdot V)}}\right)(N \cdot L) * Beérkező\ fény$$

Ez Ward izotróp modelljének megvilágítási egyenlete, ahol σ egy durvasági faktor, ami tipikusan konstans a felület egészén, a többi jelölés a megszokott. Most pedig lássunk egy egyszerű izotróp Ward shadert:

```
[Vertex_Shader]
varying vec3 normal, lightDir, viewDir, halfv;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
    halfv = (lightDir + viewDir) * vec3(0.5, 0.5, 0.5);
}

[Fragment_Shader]
uniform float sigma2_inverse;
uniform float sigma2_mul_2pi_inverse;
varying vec3 normal, lightDir, viewDir, halfv;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    vec3 H = normalize(halfv);
    float tangent = tan(acos(dot(N, H)));
    float exponent = exp(-tangent * tangent * sigma2_inverse);
    float factor = exponent * sigma2_mul_2pi_inverse;
    float dotNL = dot(N, L);
    float dot_mul = dot(N, V) * dotNL;
    float lum = factor * inversesqrt(dot_mul) * dotNL;
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

A shader ránézésre furcsa dolgokat tartalmaz, amiket az optimalizálás érdekében használtam. Először is két tényező is kiemelhető konstansként a shader teljes egészében, melyek egyedül a σ durvasági tényezőtől függenek, mely megegyezésünk alapján konstans a felület egészén:

$$\texttt{sigma2_inverse} = \frac{1}{\sigma^2} \qquad \text{valamint} \qquad \texttt{sigma2_mul_2pi_inverse} = \frac{1}{2\pi\sigma^2}$$

Másrészt megbeszéltük a Blinn-Phong modell esetében hogy a félszög vektort elegendő vertexenként kiszámítani és annak interpolált értékét használni a fragmens shaderünkben.

Tekintve az anizotróp esetet a durvasági tényezőt két irányban kell megadni, mind a tangens irányú (σ_x), mind pedig a binormál menti (σ_y) durvasági tényezőt. Ez esetben a megvilágítási egyenlet a következő formában írható fel:

$$Kimenő\ fény = \left(\frac{e^{-tan^2 \mathrm{acos}\,(N \cdot L)\left(\frac{cos^2\phi}{\sigma_x^2}+\frac{sin^2\phi}{\sigma_y^2}\right)}}{4\pi\sigma_x\sigma_y\sqrt{(N \cdot L)(N \cdot V)}}\right) * (N \cdot L) * Beérkező\ fény$$

Ahol ϕ a fény beesési irányának a tangens irányú síkon vett azimuth-i szögét jelöli. A megvilágítási modell egy lehetséges shader implementációja pedig a következő:

```
[Vertex_Shader]
uniform vec3 direction;
varying vec3 normal, lightDir, viewDir, halfv, tangent, binormal;
void main() {
    gl_Position = ftransform();
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightDir = vec3(gl_LightSource[0].position.xyz - vertex);
    viewDir = -vertex;
    halfv = (lightDir + viewDir) * vec3(0.5, 0.5, 0.5);
    tangent = cross(normal, direction);
    binormal = cross(normal, tangent);
}

[Fragment_Shader]
uniform float sigmax2_inverse;
uniform float sigmay2_inverse;
uniform float sigmaxy_mul_4pi_inverse;
varying vec3 normal, lightDir, viewDir, halfv, tangent, binormal;
void main() {
    vec3 N = normalize(normal);
    vec3 L = normalize(lightDir);
    vec3 V = normalize(viewDir);
    vec3 H = normalize(halfv);
    vec3 T = normalize(tangent);
    vec3 B = normalize(binormal);
    float dotHT = dot(H, T);
    float dotHB = dot(H, B);
    float Af = (dotHT * dotHT) * sigmax2_inverse;
    float Bf = (dotHB * dotHB) * sigmay2_inverse;
    float exponent = exp(-2.0 * (Af + Bf) / (1 + dot(H, N)));
    float factor = exponent * sigmaxy_mul_4pi_inverse;
    float dotNL = dot(N, L);
    float dot_mul = dot(N, V) * dotNL;
    float lum = factor * inversesqrt(dot_mul) * dotNL;
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

Ami magyarázatra szorul az a `direction` nevű shader paraméter. Ez a paraméter fog arra szolgálni, hogy megadjuk a felület anizotrópiájának tangens térbeli irányát. Láthatóan a számítások egyszerűsítésének érdekében szükségünk van a tangens és binormál vektorokra, melyeket a félszög vektorral együtt már a vertex shaderben meghatározunk, ezzel is növelve a shader sebességét.

A fejezet folyamán megismerhettük egy megvilágítási egyenlet összeállításához szükséges alapfogalmakat, valamint különböző nevezetes megvilágítási modelleket. Némelyek ezek közül az optikai jelenségeket fizikai szempontból közelíti meg, mások pedig sokkal inkább ad hoc módon összeállított, tapasztalati értékeken alapuló technikák, melyek az alacsonyabb számítási igényük miatt szélesebb körben alkalmazhatóak a valós idejű fotorealisztikus megjelenítés területén és ugyanakkor viszonylag élethű eredményeket produkálnak. Az, hogy mikor melyik megvilágítási modellt érdemes használni és hogy mikor van szükségünk esetlegesen egy saját technikát összeállítani, nagyban függ az alkalmazás céljától, illetve a célközönség által birtokolt grafikus hardverektől.

7. Megjelenítési technikák

Az eddigi fejezetekben szó volt a 3D grafikus kártyák fejlődéséről, illetve különböző programozhatósági és konfigurálhatósági lehetőségekről, valamint szó volt fotorealisztikus megvilágítási modellekről, melyeknek az implementációját is ismertettük. Ez a fejezet olyan haladó megjelenítési technikákat mutat be, melyek a videokártyák korábban ismertetett technológiai adottságait eszközként használja fel összetett jelenetek közel tökéletesen valósághű megjelenítését.

Természetesen lehetetlen ilyen szűk keretek között az összes ilyen technikát bemutatni, azonban a fejezet kiemel majd néhány igen széles körben használt módszert, melyek mára már elengedhetetlen eszközei a valós idejű fotorealisztikus 3D alkalmazásoknak.

Ezek a megjelenítési technikák különböző szempontok szerint osztályozhatóak. Az egyik megközelítése az osztályozásnak az alapján soroja be a technikákat az egyes típusokba, hogy milyen formájú információt használnak fel a technika megvalósításához. Ez esetben a következő osztályokról beszélhetünk:

1. **Geometria alapú technikák (geometry based rendering techniques)** – Ebbe az osztályba sorohatóak azok a megjelenítési technikák, melyek elsősorban a primitívek geometriai adottságai alapján valósítják meg a technika célját.
2. **Kép alapú technikák (image space rendering techniques)** – Az ide tartozó megjelenítési technikák elsődleges eszközei a képek, tipikusan textúrák, melyekből, akár geometriai információk felhasználásával, akár anélkül, leképezések sorával előállítják a várt eredményt.

A módszerek egy másik lehetséges osztályozása lehet az, amikor azokat a rendeltetési céljuk alapján szeretnénk besorolni csoportokba. Mivel azonban rengeteg különböző ilyen cél létezik, továbbá ezek száma folyamatosan bővül, ezek közül csak párat említenék meg, amelyeknek nagyobb jelentőséget tulajdonítanak a valós idejű fotorealisztikus megjelenítés szakterületén:

- **Részletesség megőrzés (detail preservation)** – Olyan megjelenítési módszerek, melyek segítségével nagy részletességű modellek, képek látványát tudjuk imitálni különböző trükkökkel, eljárásokkal.
- **Árnyék rajzolás (shadowing techniques)** – Technikák, melyek segítségével a egyes fényforrásokra az objektumokhoz tartozó árnyék vetületeket tudjuk meghatározni és megjeleníteni. Banális feladatnak tűnhet, azonban ne felejtsük el, hogy a klasszikus értelemben vett 3D grafikában csak lokális megvilágítási modellek megvalósítására van lehetőség, amelyek nem biztosítanak semmilyen árnyék megjelenítő lehetőséget.
- **Utófeldolgozás (post-processing)** – Olyan módszerek, melyeket a virtuális világ megjelenítése után hajtunk végre. Ezen technikák bemenete gyakorlatilag a framebuffer, amiben a már megjelenített világ képe van tárolva.

Ebben a fejezetben az előbb felsorolt kategóriákba tartozó technikák közül emelek majd ki néhányat, melyek mára már elengedhetetlen részei minden real-time fotorealisztikus grafikai alkalmazásnak, legyen az egy számítógépes játék, egy virtuális valóság szimulátor, vagy csak egy egyszerű tech-demo.

7.1. Bump mapping

Szinte bárki aki ért valamennyire a háromdimenziós grafikához, legyen az akár szoftverfejlesztő vagy akár grafikus művész, valószínűleg már hallott arról, hogy mi is az a bump mapping. Ez a technika talán a legismertebb részletesség megőrzésre alkalmas technika és elsősorban a megvilágítási modellek alkalmazásában kap nagy jelentőséget.

Az eredeti technikát James Blinn találta ki 1978-ban és a „Simulation of Wrinkled Surfaces" című publikációja keretén belül jelent meg[4]. A technika alapja, hogy a normál vektorokat egy textúra segítségével pixelenként módosítja. Ezeket a módosított normál vektorokat felhasználva a megvilágítási egyenlet kiértékelésekor a felület részletessége látszólag nagyban megnő anélkül, hogy további geometriai részletességre vagy tesszelációra lenne szükség. Blinn egy szürkeskálás textúrát alkalmazott, melyben a felület adott pontbeli magasságát, kiemeltségét tárolta. Ez az úgynevezett magasság textúra, vagy közismertebb nevén height map. A módosított normál vektorokat a felület paraméterei (tipikusan a textúra koordináták) és a height mapben lévő értékek parciális deriváltja alapján számolja ki. A deriváltak megadják, hogy a felület kiemeltsége milyen mértékben nő vagy csökken, ezáltal megkapjuk a felület aktuális meredekségét, ami alapján a normál vektorok meghatározása egyszerű.

Ahogy már említettük, ezeket a módosított felület normálisokat kell alkalmazni a megvilágítási egyenlet kiértékelésekor, ami feltételezi azt, hogy egy per-pixel megvilágítási modellt alkalmazunk. A bump mapping technikát már évtizedek óta alkalmazzák az offline megjelenítés területén, azonban köszönhetően a grafikus kártyák robbanásszerű fejlődésének, már közel tíz éve használhatjuk a technikát a valós idejű fotorealisztikus megjelenítés szakterületén is.

Tekintve a technika valós idejű alkalmazását, hamar rájöhetünk, hogy a parciális deriváltak számítása minden egyes fragmensre nem feltétlen a leghatékonyabb módszer, mivel egyrészt a korai GPU-k nem is voltak képesek parciális deriváltak számítására, másrészt az újabb grafikus processzorok esetében, amelyek már képesek is erre, megint csak nem célszerű ezt használni, mivel a művelet meglehetősen számítás igényes. A következő alfejezetekben az eljárás különböző valós időben alkalmazható megvalósításait mutatjuk majd be, kezdve a legegyszerűbbektől, egészen a haladó módszerekig melyek már a legszemfülesebb szemlélőkkel is elhitetik azt a látszatot, hogy a technikával megjelenített felületek tényleg hepehupásak.

7.1.1. Emboss bump mapping

A bump mapping technika első valós idejű imitálása az emboss bump mapping eljárás révén jött létre. Nem hiába használtam az imitálás szót, hisz ez a módszer gyakorlatilag nem is számolja ki a módosított normál vektorokat, csak megpróbálja azt a látszatot kelteni, mintha ez megtörtént volna.

Az emboss bump mapping[TN1] technika szintén a kiemelkedettséget tároló szürkeskála textúrát használja fel a látvány megvalósításához ugyanúgy, mint James Blinn eredeti módszere, azonban a hasonlóságok a két eljárás között nagyjából ki is merülnek ebben. Ahelyett hogy megpróbálná meghatározni a módosított normál vektorokat, hogy azokat használhassa a megvilágítási modell alkalmazásakot, ez a technika közvetlenül a felület megvilágítottságának meghatározását tűzi ki célul. Természetesen ebből adódóan Blinn bump mapping technikájával ellentétben az emboss bump mapping nem teszi lehetővé bármilyen megvilágítási modell alkalmazását.

Az emboss bump mapping implementációjának első lépése, hogy a height map alapján létrehozzunk két másik textúrát: az első fele olyan világos kell legyen, mint az eredeti height map, a második pedig a height map invertált változatából képezzük és szintén fele olyan világos kell legyen, mint az invertált kép. Ez után maga a megjelenítés három lépésből áll:

1. A megjelenítendő objektumot kirajzoljuk felhasználva az első létrehozott textúrát.

2. A második lépésben az invertált textúrát használjuk fel, azonban úgy, hogy az eredeti textúra koordinátákat kis mértékben a fényforrás irányába toljuk és a kapott képet hozzáadjuk az első szakasz eredményéhez. Ez által egyfajta per-pixel diffúz megvilágítottságot kap a felületünk, ahogy az a 7.1. ábrán is látható.
3. Az utolsó lépés estében egy per-vertex megvilágítási modellel árnyaljuk az objektumot, valamint esetlegesen egy hagyományos értelemben vett textúrát is ráhúzunk. A lépés eredményével megszorozzuk az eddigi képet, így megkapva a látszólag bump mapping alapú per-pixel megvilágítással megjelenített objektumot.

Mivel a módosított normál vektorok sosem kerülnek kiszámításra, az emboss bump mapping csak Lambert modelljéhez hasonló diffúz megvilágítás szimulálására alkalmas. Ennek ellenére egy nagyon figyelemre méltó technika, hisz nem szükségel semmi féle hardveres támogatást, fragmens shadert vagy vertex shadert, sőt, akár még multitextúrázási támogatottság nélkül is megvalósítható több lépéses megjelenítés és keverés segítségével.

7.1.2. Environment-mapped bump mapping

Ez a technika szintén nem egészen Blinn módszerén alapul. Az environment-mapped bump mapping (EMBM[18]) nem a megvilágítási modellt, hanem az environment mapping technikát egészíti ki olyan lehetőségekkel, hogy a megjelenített objektumok hepehupás felület látszatát keltsék.

Az eljárást a Bitboys finn cég fejlesztette ki és az első videokártya a piacon a Matrox G400 volt, amelyik a technikát hardveresen támogatta, amit hamarosan a Radeon 7000 formájában az ATI is adoptált[GL19]. Az EMBM ahelyett hogy a normál vektorokat módosítaná pixelenként, a textúra koordinátákkal teszi ugyanezt. Mindezt egy két összetevőből álló textúra segítségével, az úgynevezett DU/DV textúrával valósítja meg. Ennek texeleit (texture element) az U és V (más jelölés esetében S és T) textúra koordináták eltolására használja a módszer. Más szavakkal ez azt jelenti, hogy egy adott fragmenshez tartozó textúra koordináták (U+DU, V+DV) formában írhatóak fel, ahol az U és V az interpolált per-vertex textúra koordináták, DU és DV értéke pedig a segéd textúrából kerül ki.

Természetesen jobban ismerve az environment mapping technika által nyújtott lehetőségeket, megvalósíthatóak konkrét megvilágítási modellek is az úgynevezett environment mapek segítségével, legyen szó diffúz vagy tükörszerű anyagról. A bump mapping szerű effektus megvalósításához használatos DU/DV textúrát szintén egy height map alapján hozzuk létre, mégpedig a Blinn módszerében ismertetett parciális deriváltak alapján állítjuk elő a texelekben tárolt eltolások mértékét. Természetesen ezek az eltolások előjeles számokként állnak elő, ezért egy eltolást és egy skálázást is végre kell hajtsunk, hisz a klasszikus értelemben vett textúrák csak a [0,1] intervallumba eső értékek tárolására alkalmas, vagy fix pontos ábrázolás esetében a $[0,2^n-1]$ intervallumba eső értékek tárolására, ahol n a texel komponenseinek tárolására használt bitek száma. Értelemszerűen ugyanennek a műveletnek az ellenkezőjét végre kell hajtani a textúra felhasználásakor, amit azonban általában a hardver automatikusan elvégez.

Ahogy látható az EMBM technika alkalmas mind megvilágítási modellek szimulálására, mind pedig környezeti tükröződések megjelenítésére. Ennek ellenére ez a technika még mindig nem nevezhető igazi bump mapping implementációnak, hisz nem jönnek létre módosított normál vektorok, tehát a felhasználható megvilágítási modell lehetőségei korlátozva vannak az által, hogy mennyi információt tudunk eltárolni egy environment mapben.

7.1.3. Normal mapping

A normal mapping[TN1] technikának az eredeti ötlete Mark Peercy, John Airey és Brian Cabral 1997-ben publikált „Efficient Bump Mapping Hardware”[9] című kiadványához fűződik és már évek óta ez Blinn bump mapping technikájának leggyakrabban használt valós idejű implementációja.

Értelemszerűen az eljárás bemenetét képező textúra az úgynevezett normal map, ami egy olyan textúra, melynek szín csatornáiban a már módosított felületi normálisok koordinátái vannak tárolva. Akár csak a DU/DV textúrák előállításakor, ez esetben is a height map alapján történik a normal map létrehozása, mégpedig Blinn eredeti eljárása segítségével.

Ez alapján kaptunk egy textúrát, melynek R, G és B összetevőiben letároltuk a módosított normál vektorok X, Y és Z koordinátáit. A kérdés már csak a következő: a normál vektor koordinátái milyen koordináta rendszerben lettek letárolva? Erre különböző válaszokat is adhatunk. Megtehetjük például azt, hogy a normal mapben lévő felület normálisokat az objektumtér koordináta rendszer alapján tároljuk le, azonban hamar rájöhetünk, hogy ez nem a legcélszerűbb, hisz amennyiben az objektumainkon transzformációkat szeretnénk végrehajtani, ugyanezt meg kell tegyük a módosított normál vektorokkal is. Ez egyrészt körülményes, másrészt időigényes, hisz a transzformációk alkalmazását a normál vektorokra minden egyes fragmens esetében meg kéne tegyük. Ezzel a problémával már Blinn is szembesült az eredeti bump mapping módszer kidolgozása közben is.

A normal mapping esetében a problémára a tangens tér adta a megoldást. A tangens tér egy olyan háromdimenziós koordináta rendszer, melyben az X és Y tengelyek, melyeket tangens és binormál vektornak is szoktak nevezni, az U és V textúra koordináta tengelyeket követik, a Z tengely pedig merőleges a felületre, ami gyakorlatilag megegyezik a módosítatlan normál vektorral.

Blinn módszerében minden fragmensre megkonstruálta a tangens tért definiáló koordináta rendszert és ezt használta fel a módosított normál vektorok objektumtérbe való transzformálásához. Peercy és társai a publikációjukban egy sokkal hatékonyabb megoldást találtak ki ennek a problémának a megoldására. A tangens és binormál vektorokat előre kiszámították, de csak vertex szinten, és a primitív vertexeinek attribútumaiként hozzáférhetővé tették a vertex shaderek számára. Megjelenítéskor a fény beesési irányvektorát átviszik a tangens térbe minden egyes vertex esetében és ezt interpolálják. Fragmens szinten ezt az interpolált irányvektort normalizálják és így már közvetlenül elvégezhető a megvilágítási modell alkalmazása. Természetesen abban az esetben, ha egy olyan megvilágítási modellt szeretnénk együtt alkalmazni a bump mapping technikával, melynek szüksége van a nézet vektorra is, ez esetben a vertex shaderben ezt is át kell vinni a tangens térbe.

Mivel ezen technika esetében már közvetlenül hozzá tudunk férni a módosított normál vektorokhoz fragmens szinten, egyedül a hardver korlátozások szabhatnak határt abban, hogy milyen megvilágítási modellt implementálunk, mely felhasználná a bump mapping által szolgáltatott felület normálisokhoz. A fragmens shader támogatottság előtti videokártyák, így az első Radeon és GeForce kártyák esetében is megvalósítható ez a technika, mivel a texture combinerek segítségével végezhetünk DOT3 műveleteket (lást texture combiner mechanizmus), mellyel egy egyszerű per-pixel Lambert féle diffúz megvilágítást alkalmazhatunk.

7.1.4. Parallax mapping

Habár a bump mapping nagyban növeli a megjelenített 3D világok valósághűségét, nem szabad elfelejteni, hogy a bump mapping így is úgy is egy textúrázás alapú technika, ami azt jelenti, hogy nem változtat azon a tényen, hogy a háromszögeink gyakorlatilag laposak maradnak.

A parallax mapping[18] vagy más néven offset mapping az eredeti bump mapping technika egy tovább fejlesztett változata, mely azt a látványt imitálja, mintha a height mapben tárolt domborulatok ténylegesen kiemelkednének. Ezt az optikai jelenséget nevezik parallaxisnak. A látványt az eljárás úgy

szimulálja, hogy az egyes fragmensekhez tartozó textúra koordinátákat a nézet vektor irányával ellentétes irányban eltolja, melynek mértéke attól függ, hogy a height map adott texelében milyen érték található.

A technika implementációja a normal mapping implementációjához szükséges per-fragmens műveletek számát alig egy néhánnyal növeli, azonban a segítségével elért látvány magáért beszél. Mivel a technika megvalósításához szükségünk van height mapre, értelemszerűen még egy textúrára van szükségünk, vagy mégsem? Mivel lehetőségünk van RGBA textúrák használatára, megtehetjük azt, hogy az RGB komponensekben tároljuk a normal mapet és az A csatornában pedig a szürkeskálás height mapet.

A parallax mapping természetesen nem változtatja meg a tényleges geometriát, ebből adódóan nagyon éles szögekből nézve a felületünket, az ismét laposnak tűnik, sőt, figyelembe kell venni azt a tényt, hogy a technika csak közelítéseken alapszik, matematikailag nem helyes a megfelelő eltolás meghatározása, így aztán különösen sok lejtőt és emelkedőt tartalmazó height mapek esetében teljesen téves eredményeket is produkálhat (függelék, 7.2. ábra). Azonban ahhoz, hogy a hepehupás felület és a nézetvektor tényleges metszéspontját meghatározzuk, valamilyen kereső algoritmusra van szükségünk és nem szabad elfelejtenünk azt, hogy ennek meghatározását minden egyes fragmens esetében el kell végezzük, ami természetesen nem túl hatékony.

Összességében véve a parallax mapping egy nagyon jó eszköz mind a művészek, mind pedig a szoftverfejlesztők kezében, hogy a meglévő bump mapping implementációjukat minimális számítási költség árán tovább tökéletesítsék és amennyiben a height mapjeink nem tartalmaznak nagyon gyakori lejtő-emelkedő váltásokat, a parallax mapping által megvalósított „csalás" ránézésre optikailag helyesnek látszik.

7.1.5. Parallax occlusion mapping

Ahogy már korábban is említettem, a parallax mapping csak egy nagyon durva közelítése a valós parallaxis jelenségnek, mivel a használt eltolás mértéke nem feltétlen esik egybe a fizikailag helyes értékkel. Ennek a problémának a kiküszöbölésére rengeteg technikát találtak ki időközben: relief mapping, cone step mapping, parallax occlusion mapping. Mindezek közül a gyakorlatban a leghatékonyabbnak és legpontosabbnak ez utóbbi technika bizonyult.

A technikát az ATI Technologies munkatársai, Z. Brawley és N. Tatarchuk[11,20] mutatták be először a „Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing" című publikációjukban, 2004-ben. Az eljárás, ahogy az a cikk nevéből is adódik, egy fordított irányú sugárkövetést hajt végre a height mapen, így megtalálva a felület és a nézet vektor kvázi pontos metszéspontját. Természetesen mivel ez az algoritmus is közelítéseken alapszik, a pontos metszéspontot itt sem fogjuk meghatározni, azonban nagyon jól közelíteni fogjuk azt.

Ahhoz, hogy meghatározzuk a magasság térkép és a nézet vektor metszéspontját, a height mapet egy görbével fogjuk közelíteni. Mindig két mintát használunk fel egyszerre a height mapből: az első a t_0 textúra koordinátának megfelelő érték, mely megegyezik az interpolált textúra koordinátával, a többi pedig a $t_{i+1} = t_i + \delta P$ $(i = 0..n-1)$ textúra koordinátának megfelelő érték, ahol P az úgynevezett parallaxis vektor, melyet a parallax mapping esetében használtunk eltolásként, $\delta = \frac{1}{n}$ pedig a lépésköz mérete, n pedig a mintavételek száma. Vesszük az első két ilyen height map értéket (a t_0-hoz és a t_1-hez tartozókat) és megnézzük, hogy az így előállt szakaszt metszi-e a nézet vektor. Amennyiben nem, vesszük a következő ilyen szakaszt. Mindezt nagyon egyszerű megállapítani, hisz csak azt kell megnézni, hogy az adott textúra koordinátákhoz tartozó height map értékek a nézet vektor fölött helyezkednek-e el. Amikor ez a helyzet előáll, akkor a szakasz metszi a nézet vektort.

Ilyenkor ténylegesen meg is határozzuk ezt a metszéspontot és ez alapján a neki megfelelő textúra koordinátákat. Értelemszerűen mivel a metszéspont meghatározására a szakaszt használtuk, nem pedig a tényleges görbét, ezért az eredmény matematikailag nem helyes, azonban a tényleges metszéspontot sokkal jobban megközelíti, mint azt az eredeti parallax mapping technika tette (függelék, 7.3. ábra).

Ugyanezzel az eljárással meghatározhatjuk a fény beesési irányvektorának és a felületi görbének a metszéspontját is és így akár önárnyékolást is szimulálhatnánk, mindezt pedig még mindig úgy, hogy egyetlen további háromszöggel se kellett kiegészítsük a modellünket.

Nyilvánvalóan a parallax occlusion mapping és a hozzá hasonló eljárások sokkal számítás igényesebbek a korábban bemutatott bump mapping technikáknál, másrészt pedig a hardveres implementációjához elengedhetetlenül szükséges a dinamikus elágaztatás támogatása a fragmens shaderekben, hisz minden egyes lépésben döntenünk kell arról, hogy hogyan tovább. Ennek ellenére köszönhetően a performens grafikus kártyáknak, mára már széles körben alkalmazzák a valós idejű megjelenítés területén is.

7.2. Árnyékok megjelenítése

Az eddigi fejezetekben megismertünk jó néhány megvilágítási modellt, azok implementációját, illetve a bump mapping technika különböző megvalósításait, melyek segítségével nagyban tudjuk növelni a megjelenített háromdimenziós világunk valósághűségét, azonban nem szabad elfelejtsük, hogy minden eddig ismeretett technika kizárólag lokális megvilágítási modellek implementálására alkalmas, hisz a jelenetek minden egyes objektumát egymástól függetlenül, külön-külön jelenítjük meg. Ez természetesen hatékonyabb, mint bármilyen globális megvilágítási eljárás, nem hiába ez a bevett módszer a valós idejű megjelenítésben, azonban mindez maga után vonja minden nemű árnyék hiányát, ami sajnos erősen csökkenti a megjelenített képek hihetőségét.

Ebben a fejezetben néhány árnyék számítási és megjelenítő módszert fogok ismertetni, kezdve a legegyszerűbbektől, egészen olyan általános módszerekig, melyek szinte minden igényt kielégítő árnyékokat produkálnak még a legösszetettebb jelenetek esetében is.

7.2.1. Planáris árnyék vetítés

Az egyik legegyszerűbb árnyék megjelenítési módszer. Típusát tekintve a planáris árnyék vetítés egy geometria alapú megjelenítési technika, melyet szintén James Blinn fejlesztett ki 1988-ban. Az eljárás csak sík felületekre vetített árnyékok megjelenítésére alkalmas, ami egy az algoritmus egyszerűségéből adódó megszorítás.

Adott egy $\boldsymbol{P}: \boldsymbol{n}\boldsymbol{x} + d = 0$ módon megadott sík, ahol $\boldsymbol{n}$ a felület normálisa, illetve egy pontszerű fényforrás, melynek térbeli pozíciója az $\boldsymbol{l}$ vektorral van megadva. Konstruáljunk egy olyan $\boldsymbol{M}$ transzformációs mátrixot, ami bármely $\boldsymbol{v}$ vertexre meghatározza annak a $\boldsymbol{P}$-re képzett vetületét. Legyen $\boldsymbol{v}$ ilyen formán előállított vetülete $\boldsymbol{p}$. Ekkor a vetítési transzformáció a következő alakban írható fel:

$$\boldsymbol{p} = \boldsymbol{l} - \frac{d + \boldsymbol{n} \cdot \boldsymbol{l}}{\boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{l})}(\boldsymbol{v} - \boldsymbol{l})$$

A transzformáció mátrixos alakja ez alapján a következő:

$$\boldsymbol{M}\boldsymbol{v} = \boldsymbol{p}, \qquad \boldsymbol{M} = \begin{pmatrix} \boldsymbol{n} \cdot \boldsymbol{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \boldsymbol{n} \cdot \boldsymbol{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \boldsymbol{n} \cdot \boldsymbol{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \boldsymbol{n} \cdot \boldsymbol{l} \end{pmatrix}$$

Ha az árnyékot vető objektum összes vertexére végrehajtjuk ezt a transzformációt, akkor megkapjuk a kirajzolandó árnyék vertexeit. Figyelemmel kell lenni azonban arra a speciális esetre, amikor a fényforrás az árnyékot vető objektum és az árnyékot fogadó sík között helyezkedik el. Mivel a transzformáció egyszerűen az adott vertexet érő fénysugár és a $\boldsymbol{P}$ sík metszéspontját határozza meg, ez esetben úgynevezett anti-árnyékok jelenhetnek meg a síkon.

Természetesen az árnyékokat tipikusan nem explicit jelenítjük meg, hanem arra használjuk őket, hogy az aktuális fényforrás esetében az adott területen ne számítsunk semmilyen diffúz vagy csúcsfényt. Ezt a geometria alapú árnyék rajzoló módszerek esetében a stencil buffer segítségével valósítják meg. A stencil buffer a framebuffer része és egy vagy több logikai érték tárolására képes minden egyes pixel esetében. Először az árnyékot rajzoljuk meg, de a szín buffert nem módosítjuk, egyedül a pixelekhez tartozó logikai bufferben tárolt értéket módosítjuk. Majd ez után annyi a dolgunk, hogy a síkot kirajzoljuk a megfelelő megvilágítási modellt használva, azonban csak azokra a pixelekre alkalmazzuk a megvilágítási egyenletet, amelyikhez tartozó logikai bufferbeli értéket nem módosítottuk az előző lépésben. Ez által a sík árnyékban lévő része nem kerül megvilágításra.

7.2.2. Shadow volumes

A shadow volume algoritmus szintén egy geometria alapú technika, melynek szüksége van az objektumok poligonjaihoz tartozó kapcsolódási információkra, hogy kiszámítsa az árnyékot vető objektumok sziluettjét. Az eljárás szintén a stencil buffert használja az árnyékban lévő pixelek elkülönítésére és alapvetően az algoritmus a következő lépésekből áll:

1. Minden objektumot kirajzolunk, de csak a környezeti- és a kibocsátott fény mennyiségét használjuk fel a fragmensek megvilágítottságának számolásakor.
2. Minden árnyékot vető objektumra kiszámítjuk a poligon modelljének a potenciális sziluettjét (potencial silhouette edge – PSE) és az így előállt sziluett éleket a fénysugarak irányába extrudálva megkapjuk az árnyék térfogatot alkotó négyszögeket.
3. Ezután kirajzoljuk az előző lépésben kapott poligonokat úgy, hogy minden első lap esetén növeljük a stencil bufferben lévő értékeket, a hátsó lapok esetében pedig csökkentjük azt.
4. A végső lépés az, hogy a világ objektumait kirajzoljuk felhasználva a diffúz és csúcsfényt alkalmazó shaderek segítségével, de csak azokra a pixelekre alkalmazzuk a megvilágítási modellt, melyek esetében a stencil bufferben tárolt érték 0.

Az eredeti algoritmust Heidmann mutatta be 1991-ben, majd jó néhányan átdolgozták, finomítottak rajta, többek között Kilgard 1999-ben. Ez után is azonban rengeteg szélsőséges esetet nem kezelt jól az algoritmus, ezért nem is igazán nevezhető a legáltalánosabb árnyék megjelenítő módszernek, ennek ellenére ez a talán legszélesebb körben alkalmazott technika.

Az eljárás egyik nagy hibája többek között az, hogy csak akkor működik, ha a nézőpont nincs benne egyetlen árnyék térfogatban sem, mert ebben az esetben pontosan azok a pixelek kerülnek majd megvilágításra, amelyeknek pont hogy árnyékban kéne lenniük. Ez a probléma több féle képpen elkerülhető, például az árnyék meghatározás fordított működtetésével vagy például úgy, hogy a stencil buffer értékeit eredetileg 2^n-1 –re állítjuk, ahol n a stencil buffer értékek pontossága bitben kifejezve. John Carmack 2000-ben folytatott kutatásai alapján Everitt és Kilgard 2002-ben előállt egy sokkal elegánsabb megoldással, melyet ma már csak Z-fail módszernek neveznek. A módszer az eredeti algoritmus harmadik lépését cseréli ki a következőre:

3. Az árnyék térfogat négyszögeit úgy rajzoljuk ki, hogy minden első lap esetén csökkentjük és minden hátsó lap esetén növeljük a stencil bufferber lévő értéket, azonban mindezek előtt a mélység buffer összehasonlító operátorát az ellenkezőjére váltjuk, tehát pontosan azok a fragmensek fognak átmenni a mélység teszten, melyek eredetileg nem mentek volna át. Ezért is a Z-fail név, utalva az eljárás menetére.

Az eredeti algoritmus, melyet szoktak Z-pass módszernek is nevezni, illetve az előbb bemutatott Z-fail módszer tökéletesen ekvivalensek, tehát ugyanazt az eredményt produkálják, attól az apró különbségtől eltekintve, hogy ez utóbbi esetében nem számít speciális esetnek az, ha a nézőpont egy rnyék térfogatban helyezkedik el. Van egy dolog, ami miatt mégis csak érdemes használni az eredeti módszert szemben a Z-fail módszerrel: a Z-fail módszer használatakor többször is meg kell változtassuk a mélység teszt irányát, ami meglehetősen csökkenti a megjelenítés teljesítményét, köszönhetően az utóbbi videokártya generációkba hardveresen beépített mélység buffer optimalizálási módszereknek (lásd később).

Tekintve az algoritmus egyes lépéseinek hardveres támogatását, mivel minden modern grafikus kártyában van hardveresen gyorsított stencil buffer támogatás, a második lépés kivételével a teljes algoritmus implementálható a GPU-n. Azonban a harmadik lépést a közelmúltig a CPU-n végezték, hisz nem volt semmi beépített vagy programozhatósági lehetőség a grafikus csővezetékben, mely segítségével poligonokat állíthattuk volna elő, illetve mely esetében több vertexhez is egyidejűleg hozzáférésünk van. A DirectX 10-es támogatással megjelent grafikus kártyák azonban ezt

is megváltoztatták, hisz ettől fogva lehetőség van geometria shaderek használatára, melyek pontosan azokat a feladatokat tudják ellátni, ami a shadow volume algoritmus második lépésének hardveres implementációjához szükséges. Ennek köszönhetően az itt bemutatott algoritmus a közeljövőben valószínűleg még nagyobb népszerűségnek fog örvendeni szemben a többi árnyék megjelenítő technikával.

7.2.3. Shadow mapping

A shadow mapping egy teljesen kép alapú árnyék megjelenítő algoritmus, ami azt jelenti, hogy az eljárásnak nincs szüksége semmilyen geometriai információra a jelenetet alkotó objektumokról ahhoz, hogy elvégezze a megfelelő számításokat. Még pontosabban meghatározva, a technika az úgynevezett mélység térkép (depth map vagy shadow map) textúrákat alkalmazza, melyek esetében nem alkalmazható egyetlen hagyományos textúra szűrő (bilineáris, trilineáris vagy anizotróp), ezáltal a textúra töredezettsége sokszor szemmel látható. Gyakorlatilag ebben ki is merül az algoritmus minden rugalmatlansága. Az eljárás két lépésből áll: a shadow map megkonstruálása, majd pedig a világ megjelenítése a shadow map felhasználásával.

A shadow map elkészítése egy teljes rajzolási lépésnek számít: megjelenítjük a teljes világunkat a fényforrás szemszögéből, azonban nem használunk semmilyen textúrát, sem megvilágítást, sőt, kizárólag csak a mélység bufferbe való írást engedélyezzük. Ennek köszönhetően a lépés viszonylag gyorsan végrehajtható. A mélység bufferben lévő képet, mint textúrát használjuk fel. Ez lesz az úgynevezett shadow map, mely minden egyes, a fényforrás szemszögéből látható fragmens esetében tárolja annak távolságát a fényforrástól, hisz ez volt esetünkben a nézőpont.

Amikor a világot az eredeti nézőpontból, az adott fényforrás megvilágításával szeretnénk megjeleníteni, felhasználjuk az előbb megkonstruált shadow mapet, hogy eldönthessük az egyes fragmensek esetében, hogy az árnyékban van-e vagy sem. Ahhoz hogy ezt megtehessük, szükségünk van az illető fragmensnek a fényforráshoz viszonyított koordinátáira. Ez azt jelenti, hogy az előző lépésben használt koordináta rendszerbe kell transzformáljuk a fragmenshez tartozó koordinátákat. Természetesen ezt a transzformációt érdemes a vertex shaderben elvégezni és a fragmens shaderben az interpolált koordinátákat felhasználni. Az így megkapott (X, Y, Z) koordinátákat felhasználva meghatározzuk a shadow map (X, Y) textúra koordinátákkal címzett texelének értékét. Ez után már csak annyi a dolgunk, hogy az így megkapott értéket összehasonlítjuk a Z koordinátával. Amennyiben az összehasonlítás eredménye az, hogy a shadow mapben lévő érték a kisebb, akkor az aktuális fragmens árnyékban van.

Ideális esetben a fényforrás által megvilágított felületek esetében ez a két érték tökéletesen megegyezik, azonban a lebegő pontos számítások és maga a shadow mapben lévő értékek tárolási pontatlanságából kifolyólag ez legtöbbször nem teljesül. Az ebből fakadó abnormális optikai jelenségek elkerülése érdekében különböző trükköket kell alkalmazni. Az egyik legegyszerűbb módszer, hogy a fényforrás szemszögéből történő megjelenítéskor, vagyis a shadow map elkészítésekor a nézőpontot kicsit előrébb toljuk, vagy pedig a tényleges megjelenítési szakaszban a vertex shaderben végrehajtott transzformáció esetében tesszük meg ennek az ellenkezőjét.

A korábbi hardverek esetében a shadow mapping technika az árnyék textúrák limitált felbontásából adódóan sokszor nagyon töredezett árnyékokat produkált. Mára már a videokártyák jóval nagyobb teljesítményének és textúrázási képességük növekedésének köszönhetően akár megfelelő felbontású shadow mapek előállítására is van lehetőségünk, azonban mégsem ez a bevett szokás, hisz textúra szűrési lehetőségek híján ebben az esetben ez az algoritmus is ugyanolyan éles árnyékokat eredményez, mint azt a shadow volume algoritmus tette. Ez azonban nem felel meg a valóságnak, hisz mind a fényjelenségek fizikai hátteréből adódóan, mind pedig érzékelési

tapasztalataink alapján láthatjuk, hogy az árnyékok fokozatosan jelennek meg. A háromdimenziós grafikában előforduló éles határú árnyékok annak a feltételezésnek köszönhetőek, hogy a megvilágítási modellek többségében pontszerű fényforrásokkal dolgozunk. A valóságban azonban tökéletesen pontszerű fényforrások nem léteznek, így az árnyékok szélén vannak részben árnyékolt területek. Ezek az úgynevezett penumbrák (félárnyékok). Ennek a jelenségnek a szimulálására mindegyik árnyék megjelenítő technika esetében van lehetőség, azonban a shadow mapping a legegyszerűbb és leghatékonyabb módszer a penumbrák megjelenítésére.

Mivel eredetileg semmilyen textúra szűrő nem alkalmazható a shadow mapping esetében használt árnyék textúrák finomítására, a fragmens shaderben kell implementáljunk egy ilyet. A leggyakrabban használt ilyen szűrő az úgynevezett PCF (Percentage Closer Filtering) szűrő, melynek magja (kernel) az 7.4. ábrán látható. Ennek a szűrőnek a használatával egyrészt elkerülhetjük az árnyékok szélének töredezettségét anélkül, hogy az árnyék textúra felbontását növeljük, ezáltal növelve a shadow mapek előállítási idejét, másrészt pedig egy finomabb, penumbra szerű, árnyalt területet is kapunk az árnyékaink szélén.

Azt eldönteni, hogy a bemutatott három árnyék rajzolási technikából melyik a legjobb, nem lehet. Mindig az alkalmazástól függ, hogy melyiket és milyen formában érdemes megvalósítani. Összesítésképpen nézzük meg az egyes módszerek előnyeit és hátárnyait:

1. **Planáris árnyék vetítés**

   Előnyei: egyszerű, gyors, valamint az eredeti technika megvalósításához semmilyen különleges hardver nem szükséges.

   Hátrányai: csak síkra vetített árnyékok megjelenítésére alkalmas, valamint a penumbrák megjelenítése nagyban megnöveli az algoritmus számítás igényét.

2. **Shadow volumes**

   Előnyei: elég általános, amennyiben megfelelően van implementálva, az árnyékok szélét pontosan határozza meg, egyetlen alapkövetelménye a stencil buffer támogatás.

   Hátrányai: geometria shader támogatás hiányában nem lehet teljesen hardveresen implementálni, ezáltal elég nagy mennyiségű számítást hárít a CPU-ra, nagyon sok speciális eset van, melyekre az algoritmus nem működik és sokszor ezek lekezelése meglehetősen körülményes, illetve szintén erősen degradálja a teljesítményt a penumbrák megjelenítése.

3. **Shadow mapping**

   Előnyei: általános algoritmus, melynek teljesítményét nem határozza meg nagy mértékben az objektumaink poligon száma, illetve viszonylag alacsony teljesítmény csökkenés árán lehetőségünk van a penumbrák szimulálására.

   Hátrányai: a hardveres implementációhoz legalább egy harmadik generációs grafikus kártyára van szükségünk, valamint legalább egy negyedik generációs kártya kell ahhoz, hogy szűrést alkalmazzunk az árnyék textúrára, ennek hiányában pedig a megjelenített árnyékok széle gyakran töredezettnek látszik.

7.3. Utófeldolgozási technikák

Az utófeldolgozási lépések teljesen kép alapú technikák. Ezek a módszerek a hagyományos eljárásokkal előállított képeket használják fel bemenetként. Vannak technikák, melyek a framebuffernek csak a színeket tároló részét, esetleg részeit használják fel, mint például a motion blur vagy a bloom effektus, mások pedig a mélység buffernek a tartalmát is felhasználják, ilyen például a depth-of-field technika. Ebben a fejezetben az előbb említett utófeldolgozási technikák elméletét és azok gyakorlati megvalósítását fogjuk megismerni.

7.3.1. Motion blur

Azt az optikai jelenséget, mely következtében a gyorsan mozgó tárgyak képét elmosódottan látjuk motion blurnak vagy magyarul bemozdulásos életlenségnek nevezzük. Ennek a jelenségnek a számítógépes szimulálására az időben egymás utáni képkockák összemosását (temporal blurring) használjuk. Mivel a technika megvalósításához nem szükséges, hogy az aktuális képkockához tartozó jelenetet többször megjelenítsük, nagyon hatékonyan implementálható. A technika legegyszerűbb megvalósításához egyedül annyit kell csináljunk, hogy az aktuális képkocka eredményéül az előző képkockának és az aktuális képkockához tartozó hagyományos módon előállított kép lineáris kombinációját (interpolációját) használjuk fel:

$$Aktuális\ eredmény = Előző\ eredmény * \alpha + Aktuális\ kép * (1 - \alpha)$$

Ahol α az úgynevezett áthallási faktor, mely értékét a [0, 1] intervallumból veszi fel. Ha ennek a tényezőnek az értéke 0, akkor nincs motion blur. Minél nagyobb az α értéke, annál nagyobb mértékben befolyásolja az előző képkockához tartozó kép az aktuális megjelenítés végeredményét. Fontos azonban megjegyezni, hogy ennek a tényezőnek az értéke valós idejű alkalmazások esetében általában nem konstans, hisz a szemünk által érzékelt elmosódás mértéke függ az aktuális képrátától (FR – Frame Rate, FPS – Frames Per Second, ami megadja az egy másodperc alatt előállított képkockák számát), tehát az α értékét ennek alapján dinamikusan módosítani kell a megfelelő eredmény eléréséhez.

Tekintve ezen egyszerű technika implementációját, szinte semmilyen hardveres támogatásra nincs szükségünk, egyedül az additív keverés támogatására, ami gyakorlatilag minden 3D grafikus kártya esetében adott. Az aktuális megjelenítési lépést úgy végezzük el, hogy annak eredménye egy textúrába kerüljön (ami általában a legtöbb utófeldolgozási lépésre igaz). Ezután a valódi framebuffert használva célként, megjelenítünk egy a teljes képernyőt átölelő négyszöget, melynek textúra koordinátái oly módon kerülnek beállításra, hogy az előbbi lépésben előállított textúra minden egyes texele egy-egy pixellel kerül összerendelésre. A grafikus csővezeték keverési lépését bekonfigurálva a megfelelő módon, előállítjuk a textúra és a framebufferben aktuálisan tárolt kép segítségével a kívánt eredményt. Mivel ugyanezt végezzük el minden egyes képkocka esetében, adott lépésben az aktuálisan a framebufferben tárolt kép gyakorlatilag az előző képkocka, tehát pont azt kaptuk, amit szerettünk volna.

7.3.2. Depth-of-Field

Minden lencse, legyen az egy fényképezőgép lencséje vagy pedig az emberi szemlencse, rendelkezik egy fókusztávolsággal, ami természetesen az emberi szem esetében dinamikusan változtatható, mely meghatározza azt a távolságot, mely esetében a tárgyakat tökéletesen élesnek látjuk, azonban minden objektum, mely ettől a távolságtól közelebb vagy távolabb helyezkedik el, bizonyos mértékben elmosódottnak látszik.

A klasszikus számítógépi grafika az úgynevezett tűlyuk (pinhole) kamera modellt alkalmazza, ami azt jelenti, hogy a fénysugarak csak egyetlen pontban juthatnak át a virtuális szemlencsénken,

mint ahogy az a 7.5. ábrán látható. Ez által minden egyes képelem a megjelenített világunkban tökéletesen élesen látszik, mintha minden fókuszban lenne.

A valódi kamerák lencséi nem tekinthetőek pontszerűnek, így aztán rendelkeznek egy adott fókusztávolsággal (függelék 7.3 ábra). Emiatt valójában csak egy adott távolságra lévő tárgyak lesznek optikailag tökéletesen élesek egy képen, de mivel a szemünk közel sem tökéletes, így más, ettől a távolságtól eltérő távolságban lévő tárgyak is élesnek látszódhatnak. Azt a tartományt, amin belül az élességet elfogadhatónak találjuk, mélységélesség tartománynak hívjuk (DOF – Depth-of-Field). A fókusztávolságnál közelebb lévő tárgyak a szenzorunk mögé, a távolabb lévő tárgyak pedig a szenzorunk elé fókuszálódnak majd. Az így képződő köröket hívjuk lágy pontoknak (CoC – Circle of Confusion). Ha a CoC kellően kicsi, akkor a tárgy élesnek tűnik, tehát a DOF-on belül van.

Ahhoz, hogy a megjelenített világunkra alkalmazzunk egy olyan utófeldolgozási lépést, mely élethűen imitálja a depth-of-field hatást, szükségünk lesz az előállt képhez tartozó szín és mélység bufferre, valamint néhány paraméterre: a fókusztávolságra, illetve a rekesznyílás méretére, mely meghatározza majd, hogy milyen mértékben nő a lágy pontok átmérője a tárgypont fókusztávolságtól mért távolságának arányában. A szín és a mélység buffer tartalmát az eljárás mint textúrákat fogja felhasználni, hogy előállítsa a framebufferben a végleges képet. Szintén egy a teljes framebuffert beterítő négyszöget rajzolunk, melyhez az előbb említett két textúrát csatoljuk. A mélységélességi hatást a fragmens shader fogja előállítani. Minden egyes fragmensre tekintjük a mélység információt tartalmazó textúra adott texelét. Ezen érték alapján, illetve a fókusztávolságot és a rekesznyílás méretet felhasználva kiszámítjuk a CoC méretét. Ez után már csak arra van szükség, hogy a szín információt tartalmazó bemeneti textúránkból előállítsuk a CoC-t. Ehhez nyilvánvalóan a textúra többszöri olvasása szükséges, melyet egy poisson eloszlású szűrő maggal (filter kernel) fogunk megvalósítani (függelék, 7.6. ábra).

Ez a technika viszonylag alacsony számítás igényű és elég jó eredményeket produkál, azonban vannak esetek, amikor a fókusztávolságban lévő objektumok pixelei „átfolynak” az életlen objektumokra, hisz a poisson szűrő alkalmazásakor nem tudhatjuk biztosan, hogy az adott texel minták ténylegesen ugyanabban a távolságban vannak-e. Az ilyen mellékhatások elkerülése érdekében érdemes a poisson filter minden mintavétele esetében ellenőrizni, hogy nincs-e nagy mértékű eltérés az aktuális minta és az eredeti minta mélység értékei között. Amennyiben ilyen előfordul, akkor az illető mintát nem érdemes belevenni a CoC szimulálására használt interpolációba.

A depth-of-field utófeldolgozási technika habár egy jóval komplexebb és időigényesebb technika, mint például a motion blur, valamint az implementálásához elengedhetetlen valamilyen pixel vagy fragmens shader jelenléte, mégis egy hatalmas eszköz a művészek kezében. Egyre gyakrabban használják az utóbbi években a valós idejű megjelenítésben is, mivel nagy mértékben növeli a megjelenített képek hihetőségét, valósághűségét.

7.3.3. Bloom

A bloom hatás azt az optikai jelenséget hivatott szimulálni, mely során a kép nagyon fényes részei beletüremkednek a kevésbé fényes részek képébe. Ezt a hatást például akkor vehetjük észre, ha egy viszonylag sötét szobában vagyunk, kinn pedig erősen süt a nap és kinézünk az ablakon. Ilyenkor az ablakon beszűrődő erős fény sokszor még az ablakkeretet is átfedő erős fényérzetet kelt mind az emberi szemben, mind pedig egy fényképezőgép vagy kamera által készített képen.

Mivel az emberi szem dinamikusan változtatni tudja a fényérzékelési tartományát, a bloom hatás csak egy gyenge közelítése a valóságban végbemenő optikai jelenségnek, hisz a bloom hatás hagyományos implementációi fix pontos framebufferrel dolgoznak, mely esetében lehetetlen elkapni a fény teljes tartományát. A DirectX 9-et támogató grafikus kártyáktól kezdődően lehetőség van lebegő

pontos framebuffer használatára (HDR – High Dynamic Range) és ezáltal a modern alkalmazásokban a bloom hatás fizikailag is helyes eredményeket tud produkálni.

Alapvetően a megjelenítési technika megvalósítása mindkét esetben ugyanaz. A bloom hatás alkalmazása, hasonlóan az eddigi utófeldolgozási technikákhoz, a hagyományos megjelenítés eredményét használja fel és ezt dolgozza át. Az algoritmus lépései a következők:

1. Megjelenítjük a világunkat a hagyományos módon, azonban a framebuffer helyett egy textúrát használunk célként.
2. A szín buffernek megfelelő textúrát felhasználva előállítjuk annak az 1/16 méretű mását.
3. Erre a csökkentett méretű szín információt tároló textúrára alkalmazunk egy Gauss szűrőt, ezzel előállítva a kép elmosódott mását.
4. Felhasználva az eredeti szín buffer alapján elkészült textúrát és a csökkentett méretű elmosódott képet, előállítjuk a végső képet, mely a következő képlet alapján áll elő:

$$Végső\ kép = (Eredeti\ kép - B_1) * S_1 + (Elmosott\ kép - B_2) * S_2$$

Ez gyakorlatilag azt jelenti, hogy mindkét összetevőre alkalmazunk egy skálázást és egy eltolást (scale and bias). Ha fix pontos framebuffert használunk akkor általában a $B_1 = 0$ és az $S_1 = 1$ tehát az eredeti képet változatlanul hagyjuk, hisz nincs dinamikus érték tartományunk.

Az algoritmus felvázolt lépései között van azonban egy, melynek megvalósítása első körben meglehetősen nehézkesnek tűnhet. Ez a harmadik lépés, amikor a Gauss szűrőt alkalmazzuk. A Gauss féle elmosódást azért nehéz implementálni, mert nagyon sok textúra hozzáférésre van szükség, esetünkben például célszerű egy 7x7-es Gauss szűrőt használni a megfelelő eredmény eléréséhez, ami 49 textúra hozzáférést jelentene. Ez egyrészt meglehetősen időigényes és viszonylag új típusú hardvert igényel, melyben lehetőség van olyan shader írására, ami ilyen sok textúra hozzáférést támogat. A Gauss féle szűrőnek azonban van az a jó tulajdonsága, hogy felbontható az X és Y tengelyek mentén, ami azt jeneti, hogy elegendő az eredeti képre egy 7 lépéses horizontális Gauss szűrőt alkalmazni, majd az eredményül kapott képre pedig egy szintén 7 lépéses vertikális Gauss szűrőt alkalmazni, mivel matematikailag és ebből kifolyólag a gyakorlatban is, eltekintve a lebegő pontos és a framebuffer pontatlanságtól, ez a két lépés ugyanazt az eredményt produkálja, mint a 7x7-es Gauss szűrő.

Az emberi szemnek természetesen minden értelemben dinamikus a fényérzékelési tartománya, gondoljunk csak arra, hogy amikor egy sötét szobából lépünk ki a napra, az első pillanatban szinte megvakulunk az erős fénytől, majd pár másodperc múlva a szemünk alkalmazkodik az új fényviszonyokhoz és elég élesen látunk az új körülmények között is. Ugyanez történik az ellenkező esetben is, amikor az erős napfényről visszamegyünk a sötét szobába és először szinte semmit nem látunk, majd a szemünk ismét adaptálódik a változásokhoz. Lebegő pontos framebufferek használatakor (HDR) a bloom hatás mellé alkalmazni szokták az úgynevezett tone mapping eljárást, hogy a végső kép előállítása esetében a B_1 és S_1 paramétereket frame-enként változtatják, hogy a szemnek ezt a tulajdonságát is imitálják.

Ez az utófeldolgozási technika nem nagyban növeli a képkockák (frame-ek) előállítási idejét, ugyanakkor nagy mértékben növeli az előállított képek valósághűségét, nem hiába manapság nem nagyon adnak ki számítógépes játékokat sem a bloom hatás valamilyen implementációja nélkül.

8. Hatékonysági problémák és potenciális megoldások

A korábbi fejezetek arról szóltak, hogy hogyan implementáljuk az egyes fotorealisztikus megvilágítási modelleket, illetve, hogy milyen haladó megjelenítési technikákkal növeljük az előállított képek élethűségét. Mindazonáltal a bemutatott eljárásokat, algoritmusokat valós idejű alkalmazások esetében szeretnénk felhasználni, így aztán néhány fontos hatékonysági kérdést és optimalizálási lehetőséget is meg kell hogy említsünk.

8.1. Szűk keresztmetszet

Ideális esetben az algoritmusok implementációi tökéletesen ki kéne hogy használják a hardver által kínált erőforrásokat. Ekkor elérhetnénk az adott hardver esetében az optimális teljesítményt. Természetesen ez általában nem így van. Legtöbbször a grafikus csővezeték egy része jobban le van terhelve, mint a többi. Ilyenkor ez a szakasz lesz a rendszer szűk keresztmetszete. Rendszer alatt ez esetben a grafikus alkalmazás, a driver és a videokártya együttesét értjük. Ilyen esetekben a tesztelők feladata, hogy megtalálják a rendszer szűk keresztmetszetét, mely alapján a fejleszőnek lehetősége van optimalizálási stratégiákat alkalmazni, melyek segítségével a túlterheltség mértékét csökkenthetik az által, hogy néhány feladatot a rendszer más részeivel végeztetnek el.

8.1.1. A vertex feldolgozó csővezeték túlterhelése

A vertex feldolgozó csővezeték túlterhelése lehet szoftveres vagy hardveres szűk keresztmetszet attól függően, hogy a grafikus csővezeték ezen része a driverben vagy pedig a grafikus kártyán van implementálva. Ezen probléma tipikus forrásai a következők:

- Túl részletes geometriával rendelkező modellek megjelenítése.
- Túl összetett vertex shaderek használata, melyeket már a grafikus kártya nem támogat hardveresen vagy csak egyszerűen nagyon sok számítást igényelnek.

Ezen problémák általában amiatt jönnek elő, hogy vertex shaderekkel szeretnénk implementálni a megvilágítási modellünket és az egyéb megjelenítési technikáinkat, azonban a szükséges minőség eléréséhez részletes háromdimenziós modellekre van szükségünk. Ez az a tipikus szemléletmód, amely mentén a modellezésben járatos művészek gondolkodnak, azonban ami működik az offline megjelenítésben, az nem feltétlenül hatékony a valós idejű alkalmazások esetében, hisz ez esetben nagy számú vertexre futnak le az amúgy is összetett vertex shaderek.

Általában a probléma kiküszöböléséhez át kell térjünk egy per-pixel megvilágítási implementációra, mely együtt egy részletesség megőrzési technikával, mint például a bump mapping, hasonló eredményeket produkál egy pár ezer vertexből álló modellel, mint az eredeti megközelités tette volna egy milliós nagyságrendű vertexből álló modell segítségével. Ezáltal elértük azt, hogy a vertex feldolgozó csővezetékünk jóval kevesebb feladat elvégzését kell csak megoldja, természetesen ezzel egyidejűleg növeltük a fragmens feldolgozó csővezeték terheltségét.

Szintén csökkenthető a grafikus csővezeték ezen részének terheltsége azáltal, hogy csak az olyan poligonokat, objektumokat jelenítjük meg, amelyek valószínűleg látszani is fognak a végső képen, hisz semmi értelme nincs például a kamera mögött elhelyezkedő tárgyak megjelenítésére. Ennek megvalósításához a hatékonyság szempontjából érdemes valamilyen tér felosztási adatszerkezetet és algoritmust használni, mint például a BSP fák vagy a octree-k.

8.1.2. A fragmens feldolgozó csővezeték túlterhelése

Mivel általában egy képkocka megjelenítésekor feldolgozott fragmensek száma általában nagyságrendekkel nagyobb a feldolgozott vertexek számánál, a grafikus csővezeték ezen része jóval

könnyebben túlterhelődik még egy tapasztalt grafikus alkalmazás fejlesztő kezében is. Ennek tipikus okai a következők:

- Túl összetett fragmens shaderek alkalmazása.
- A fragmens mélység értékének módosítása egy fragmens shaderben (ebből adódóan nincs a hardvernek nincs lehetősége bizonyos mélység buffer optimalizálások alkalmazására, lásd később).
- Dinamikus elágaztatás alkalmazása a fragmens shaderekben, ami a pipeline architektúrájú grafikus processzor áteresztőképességét erősen csökkenti.
- Összetett textúra szűrők használata (trilineáris, anizotróp vagy fragmens shaderekben implementált szűrők).
- A stencil és az alfa tesztek illetve a keverés túlzott vagy fölösleges használata.

Ez utóbbi elkerülésére egyszerű a megoldás: csak akkor használjuk ezeket, ha feltétlen szükség van rá, például amennyiben a keveréshez nem szükséges felhasználni a szín bufferben tárolt szín és alfa értékeket, érdemes ezt kikapcsolni. A többi probléma kiküszöbölésére alkalmazhatjuk a következő módszerek valamelyikét vagy azok együttesét:

- Dinamikus elágaztatások helyett inkább használjunk több számítást, mivel gyakran ez esetben jobb teljesítményt érünk el.
- Amennyiben mindenképpen szükség van dinamikus elágaztatásokra, igyekezzünk ezeket a fragmens shader elejére helyezni, mivel ilyenkor a pipeline processzorok éhezési mértéke csökken.
- A fragmens shaderek azon számításait, melyeket elegendő vertexenként elvégezni és azok eredményeinek az interpolált értékét felhasználni, helyezzük át a vertex shaderekbe.
- Amennyiben nem szükséges, kapcsoljuk ki a mélység bufferbe való írást (például több lépéses megjelenítés esetében).
- A mélység buffer kikapcsolása és a hátulról előre haladó megjelenés alkalmazása helyett inkább használjuk a mélység buffert és esetlegesen használjunk ezzel együtt előről hátra haladó megjelenítést.

Manapság az egységesített shader architektúra alkalmazásának köszönhetően a grafikus processzor ugyanazon része futtatja a vertex és fragmens shadereket, ellentétben a korábbi hardverekkel, ami miatt nem beszélhetünk tényleges szűk keresztmetszetről a grafikus csővezeték ezen részei esetében, azonban az itt megemlített optimalizálási technikákat ettől függetlenül alkalmazhatjuk a megjelenítés teljesítményének növeléséhez, hisz habár ez az egységesített végrehajtó egység folyamatosan egyenletes terhelés alatt van, nem mindegy, hogy milyen mennyiségű műveletet kell elvégezzen.

8.2. Mélység buffer optimalizálás

Mivel a modern valós idejű grafikus alkalmazások szinte minden esetben masszívan használják a mélység buffer által nyújtott lehetőségeket érdemes figyelmet szentelni a mélység buffer és a mélység teszt hardveres megvalósítására.

Az első nagyon lényeges dolog, amit itt meg kell hogy említsünk az az a tény, hogy a grafikus hardver alapértelmezett működése eltér a grafikus csővezeték elméleti modelljétől. A modell azt mondja, hogy a mélység teszt a fragmens végső színének meghatározása után kerül sorra, azonban a grafikus hardver általában ezt még a textúrázási lépés előtt hajtja végre, tehát a fragmens shaderek lefutása előtt. Ez által a mélység buffer alapján nem látható fragmensekre le se fut az aktuálisan kiválasztott fragmens shader. Ebből adódóan olyan számítások alól menti fel a fragmens shaderek futtatására szolgáló végrehajtó egységet, melyeket amúgy is fölöslegesen számított volna ki, hisz a

fragmens a következő lépésben úgyis eldobásra került volna a fragmens mélység értékének ellenőrzésekor. Ezt a hardveres optimalizálást nevezik korai mélység tesztelésnek (early depth test)[2].

Itt utalnék vissza egy korábbi témára, még pedig arra, hogy habár a fragmens shaderekben lehetőség van a fragmens mélység értékének módosítására, ez nem javasolt. Értelemszerűen amikor egy olyan fragmens shader van kiválasztva futtatásra, mely tartalmaz olyan utasítást, ami módosíthatja a fragmens mélység értékét, a grafikus kártya ki kell hogy kapcsolja a korai mélység tesztelést ahhoz, hogy a grafikus csővezeték elméleti modelljének megfelelő működést produkáljon. Nyilvánvaló, hogy egy a fragmens feldolgozási teljesítmény rovására megy, ezért amennyiben nem feltétlen szükséges, érdemes ezt elkerülni.

A mélység bufferek megfelelő hardveres támogatása előtt gyakran alkalmazták azt a módszert a grafikus alkalmazás fejlesztők, hogy a poligonokat hátulról előre haladó sorrendben jelenítették meg és így többnyire nem volt szükség a mélység buffer használatára ahhoz, hogy megfelelő eredmény kapjunk. Mára már a mélység bufferek teljesítménye nagyon nagy köszönhetően a hierarchikus mélység buffereknek. A hierarchikus mélység buffer a hagyományos mélység buffer mellett rendelkezik az eredeti buffer kicsinyített másaival is (1/4 méretű, 1/16 méretű, stb.). Egy adott szinten a buffer 1/4 méretű másolata úgy készül el, hogy az eredeti buffer minden egyes 2x2-es méretű cellanégyesből kiválasztja a legkisebb vagy a legnagyobb értéket (attól függően, hogy milyen összehasonlító operátort választottunk ki a mélység teszthez) és ezt írja a csökkentett méretű buffer megfelelő cellájába.

Amikor egy adott poligon raszterizálásakor elvégezzük a mélységtesztet, első körben a legkisebb méretű mélység buffer megfelelő értékét nézzük. Amennyiben a mélység buffer adott értékéhez tartozó fragmensek nem mennek át a mélység teszten, egyidejűleg több fragmens eldobását is elvégzi a hardver. Ha valamely fragmensek átmennek a teszten, akkor vesszük az egyel nagyobb méretű mélység buffert. Ugyanezt az algoritmust hajtjuk végre minden szintre. Ezáltal egy fragmens csak akkor megy át a mélység teszten, ha az összes szinten átmegy azon. Az optimalizálási módszer legfőbb előnye, hogy lehetőség van a fragmensek tömeges eldobására akár egyetlen mélység buffer hozzáférés segítségével.

Mivel a bemutatott optimalizálásoknak köszönhetően mára a mélység buffer megfelelő alkalmazása az egyik legjobb módszer a fragmens csővezeték terheltségének csökkentésére. Ebből kiidulva nem meglepő, hogy ha a klasszikus hátulról előre haladó megjelenítési sorrend helyett előről hátra haladó sorrendet alkalmazunk, a megjelenítés teljesítménye akár nagyságrendekkel jobb lehet, hisz optimális esetben minden egyes pixelre pontosan egyszer fut le a fragmens shader.

Ezt a gondolatot vitte tovább John Carmack a Doom 3 grafikus motorának készítésekor. A játék grafikus motorja egy meglehetősen összetett per-pixel megvilágítási modellt alkalmazott, így a megjelenítés teljesítményét erősen befolyásolta az, hogy mennyi fragmensre alkalmazta azt. Annak biztosítására, hogy minden egyes pixel esetében pontosan egyszer értékelődjön ki a megvilágítási egyenlet, Carmack bevezette az úgynevezett „csak mélység” lépést. Ez azt jelentette, hogy első lépésben a világot úgy jelenítette meg, hogy a szín bufferbe való írás és minden fragmens és vertex shader ki volt kapcsolva, a lépés egyedül a mélység buffer megfelelő feltöltésére szolgált. A további megjelenítési lépésekben pedig a mélység bufferbe való írás volt kikapcsolva és a mélység teszt alkalmazása miatt minden egyes pixelre pontosan egyszer futott le az alkalmazott fragmens shader, ezzel optimalizálva a szükséges számítások mennyiségét.

Amit még érdemes a hierarchikus mélység bufferrel kapcsolatosan megemlíteni az az, hogy maga a buffer hierarchia megkonstruálása viszonylag sok időbe telik, tehát nem ajánlott a mélység teszt összehasonlítási irányát megváltoztatni, mivel ekkor a mélység buffer hierarchiát újra kell

generálni. Többek között emiatt lassabb jóval a korábban bemutatott shadow volume árnyék megjelenítő technika Z-fail változata a Z-pass módszernél.

8.3. Occlusion culling

Az occlusion culling egy hardveres lehetőség, hogy bármilyen számítási költség nélkül megtudhassuk adott primitívre vagy primitívekre, hogy a hozzájuk tartozó fragmensek közül hány ment át a mélység teszten. Ezen információ birtokában el tudjuk dönteni, hogy az illető primitív(ek) láthatóak-e. Első hangzásra a módszer olyan érzetet kelt, mintha azt akarnánk eldönteni, hogy „mi volt előbb, a tyúk vagy a tojás", hisz ahhoz hogy eldöntsük, hogy a primitívek egyáltalán láthatóak-e, előtte meg kell hogy jelenítsük őket. Természetesen a gyakorlatban nem feltétlenül így történik mindez és nagyon gyakran növelhetjük a segítségével a megjelenítés teljesítményét.

Az első hasznos alkalmazása[TN3] ennek a lehetőségnek az, hogy egy geometriailag összetett modell megjelenítése helyett, ami esetlegesen túlterhelné a vertex feldolgozó csővezetéket, egy egyszerűbb befogadó tér modellt jelenítünk meg először, mindezt úgy, hogy a mélység bufferbe és a szín bufferbe való írást letiltjuk, így ennek megjelenítése nagyon gyorsan megtörténik. A befogadó tér modell megjelenítésekor a mélység teszten átment fragmensek számát megkapjuk az occlusion culling segítségével és ezt értelmezzük. Nyilvánvaló, hogy az eredeti összetett modellnek legfeljebb annyi fragmense lenne látható, mint amennyi a befogadó tér modellből látható. Ez alapján a következő döntéseket tudjuk hozni:

- Ha a befogadó tér modell megjelenítésekor egy fragmens sem ment át a mélység teszten, akkor az eredeti modell sem látszana, tehát nem szükséges annak megjelenítése.
- Az alapján, hogy hány fragmens ment át a teszten, eldönthetjük, hogy az eredeti modellt milyen részletességgel jelenítsük meg, hisz például ha a modellből legfeljebb 100 pixel fog látszani, akkor fölösleges, hogy egy 100000 poligonból álló modellt jelenítsünk meg, elegendő, hogyha az eredeti modell egy redukált változatát rajzoljuk csak ki.

Az occlusion culling egy másik gyakori alkalmazási területe a több lépéses megjelenítésbeli felhasználása[2]. Mivel sok megvilágítási modell annyira összetett, hogy egy lépésben csak egy fényforrás által végzett megvilágítást tudunk csak megjeleníteni, az összes fényforrás esetében újra és újra meg kell hogy jelenítsük a világunkat. Amennyiben a lépések előtt végrehajtottunk egy „csak mélység" lépést, akkor már az első lépésben, ha nem, akkor pedig a második lépésben rendelkezhetünk az occlusion culling segítségével minden olyan információval, amivel eldönthetjük, hogy mely objektumok vagy poligonok láthatóak a végső képen, illetve melyek nem. Ezen információ birtokában a további lépésekben megspórolhatjuk ezeknek az objektumoknak/poligonoknak a kirajzolási idejét.

Az occlusion culling magyarul takarás alapján történő eldobást jelent, pedig maga a mechanizmus csak egy egyszerű visszajelzési érték hozzáférésére biztosít lehetőséget. Azonban nyilvánvaló, hogy segítségével tényleg lehetőség van láthatósági vizsgálatok elvégzésére és gyakorlatilag ez alapján akár a megjelenítendő objektumok nagy többségének kirajzolását kihagyhatjuk, ezzel ismét nagy mértékben növelve a shadereinkkel megvilágított világok megjelenítését.

Összefoglalás

A szakdolgozat folyamán sikerült megismerkednünk a valós idejű háromdimenziós grafika fejlődési szakaszaival megismerkedve a grafikus hardverek egyes generációival és azok sajátosságaival, képességeivel. A kártyák által támogatott lényegesebb technológiákat is be tudtam mutatni, melyek közül a lényegesebbekről részletesebben is beszéltem.

Ezen felül megismertük a nevezetes megvilágítási modellek kialakulásának rövid történelmét, vázlatosan láthattuk azok matematikai hátterét és azok implementációját is be tudtuk mutatni. Megemlítettük a háromdimenziós számítógépi grafika feljődésében nagy szerepet betöltő személyeket, mint például Bui Tuong Phong vagy James Blinn, akik nagy úttörői voltak ennek az egyre népszerűbb informatikai szakterületnek.

Láthattunk haladó szintű megjelenítési technikákat, algoritmusokat, melyeknek sajnos az implementációs lépéseit csak nagyon felületesen sikerült bemutatni, mivel azok többsége meghaladja egy szakdolgozat keretein bemutatható módszer korlátait. Ennek ellenére a shader nyelvek legtöbb gyakorlati alkalmazási területét sikerült bemutatni, melyek a fotorealisztikus megjelenítés témakörébe tartoznak. Nem sikerült azonban ezeknek a shader nyelveknek a felépítéséről, szintaxisáról, illetve egyéb részleteiről beszélni, ami szintén egy nagyobb terjedelmű publikáció keretein belül lenne csak megvalósítható, másrészt pedig az idő folyamán jó néhány shader nyelvezet alakult ki, melyeknek történelmi szempontból nagy jelentősége volt, azonban mára már csak néhányat használnak a gyakorlatban ezek közül, illetve többnyire vendor vagy API függő nyelvekről van szó, tehát alapvetően azok az általános tulajdonságok, melyek mindegyikben fellelhetőek, azok többnyire említésre is kerültek.

Szintén fontos szerepet kapott az optimalizálás szükségességének kihangsúlyozása. Szó volt általános optimalizálási ötletekről, technikákról és ezeket elősegítő hardver és szoftver technológiákról, valamint az egyes implementációk és implementáció közeli magyarázatok esetében megemlítettünk néhány specifikus optimalizálási lehetőséget. Így például a megvilágítási egyenletek implementációi esetében igyekeztem megfelelően megosztani a vertex és a fragmens shaderek között a számításokat.

Azt is láthattuk, hogy a valós idejű háromdimenziós grafikában szinte minden számítást csak közelítésekkel valósíthatunk meg, mivel hiába a grafikus hardverek hihetetlen számítási teljesítménye, a való világban végbemenő részecske szintű optikai jelenségeket lehetetlen egyenlőre teljes mértékben szimulálni a mai technológiákkal. Természetesen ez ugyanígy igaz az offline megjelenítésre is, azonban a valós idejű megjelenítés még inkább ezekre a durva közelítésekre kell hogy helyezze a hangsúlyt. Láthattunk példákat arra, hogy legtöbbször a megjelenítési módszereket nem a programozásban megszokott hagyományos módon optimalizálunk, hanem általában további közelítéseket vezetünk be a megfelelő teljesítmény eléréséhez. Ahogy azt a gyakorlat mutatja és ahogy arról a szakdolgozatban is szó van, olyan esetek is előfordulhatnak, amikor egy ad hoc módon megkonstruált durva közelítés hihetőbb eredményeket produkál a számítógépi grafikában, mint egy fizikai alapokon nyugvó közelítés.

A szakdolgozatban bemutatott hardver és szoftver technológiák, a megjelenítési algoritmusok és módszerek ismeretében egy rutinos programozó több-kevesebb utánaolvasással képes lehet látszólag tökéletesen élethű álló- és mozgóképek megjelenítésére, illetve szintén jó kiindulópont a kreatívabb fejlesztők számára, hogy saját megjelenítési technikáit kifejlessze.

Ez úton szeretnék köszönetet mondani mindazoknak, akik elindítottak és vezettek az informatika ezen szakterületének megismerésében. Legelőször Bertók Tamásnak szeretném megköszönni, aki középiskolás informatikát tanuló diákként felkeltette a tizenkét éves önmagam érdeklődését a számítógépi grafika iránt, és sok technikai is elméleti segítséget nyújtott az induláskor.

Szintén megköszönném azoknak a családtagoknak, barátoknak, akik hatalmas érdeklődéssel követték figyelemmel az egyes grafikai szoftvereim, játékaim feljődését. Végül, de nem utolsó sorban pedig szeretném megköszönni a Debreceni Egyetem Komputergrafika és Képfeldolgozás tanszékének dolgozóinak munkáját (Dr. Kovács Emőd, Dr. Tornai Róbert) és közülük is elsősorban témavezetőm, Dr. Schwarcz Tibor egyetemi adjunktus segítségét, mellyel pótolták a számítógépi grafikával kapcsolatos ismereteim elméleti hiányosságait és továbbra is fenn tartották érdeklődésemet az informatika ezen oly érdekes területe iránt.

Irodalomjegyzék

Könyvek, cikkek, tanulmányok:

[1] Arnold Gallardo – 3D Lighting: History, Concepts, and Techniques, Charles River Media Inc., 2001

[2] Evan Hart (ATI), John Spitzer (NVIDIA) – OpenGL Performance Tuning, Game Developers Conference, 2004

[3] Fábio Policarpo, Manuel M. Oliveira, João L. D. Comba – Real-Time Relief Mapping on Arbitrary Polygonal Surfaces, 2005

[4] James Blinn – Simulation of Wrinkled Surfaces, 1978

[5] James C. Leiterman – Learn Vertex and Pixel Shader Programming with DirectX 9, Wordware Publishing Inc., 2004

[6] John Kessenich – The OpenGL Shading Language (Version 1.20), 3Dlabs Inc. Ltd., 2006

[7] Jonathan Dummer – Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm

[8] Kelly Dempski, Emmanuel Viale – Advanced Lighting and Materials with Shaders, Wordwar Publishing Inc., 2005

[9] Mark Peercy, John Airey, Brian Cabral – Efficient Bump Mapping Hardware, 1997

[10] Mark Segal, Kurt Akeley – The OpenGL Graphics System: A Specification (Version 2.1 – July 30, 2006), Silicon Graphics Inc., 2006

[11] Natalya Tatarchuk – Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows, 3D Application Research Group, ATI Research Inc., 2006

[12] Paul Martz – OpenGL Distilled, Addison Wesley Professional, 2006

[13] Randi J. Rost – OpenGL Shading Language, Second Edition, Addison Wesley Professional, 2006

[14] Renate Kempf, Jed Hartman – OpenGL on Silicon Graphics Systems, Silicon Graphics Inc., 2005

[15] Thomas Scott Crow – Evolution of the Graphical Processing Unit, 2004

[16] Tom McReynolds, David Blythe – Morgan Kaufmann Publishers, Elsevier Inc., 2005

[17] Wolfgang F. Engel – ShaderX: Vertex and Pixel Shader Tips and Tricks, Wordware Publishing Inc., 2002

[18] Wolfgang F. Engel – ShaderX2: Introductions and Tutorials with DirectX 9.0, Wordware Publishing Inc., 2004

[19] Wolfgang F. Engel – ShaderX3: Advanced Rendering with DirectX and OpenGL, Charles River Media Inc., 2005

[20] Z. Brawley, Natalya Tatarchuk – Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing, 2004

Internetes források:

OpenGL kiterjesztés specifikációk (típusonként megjelenési sorrendbe rendezve):

[GL1] http://www.opengl.org/registry/specs/ARB/multitexture.txt

[GL2] http://www.opengl.org/registry/specs/ARB/vertex_blend.txt

[GL3] http://www.opengl.org/registry/specs/ARB/texture_env_combine.txt

[GL4] http://www.opengl.org/registry/specs/ARB/texture_env_crossbar.txt

[GL5] http://www.opengl.org/registry/specs/ARB/texture_env_dot3.txt

[GL6] http://www.opengl.org/registry/specs/ARB/depth_texture.txt

[GL7] http://www.opengl.org/registry/specs/ARB/shadow.txt

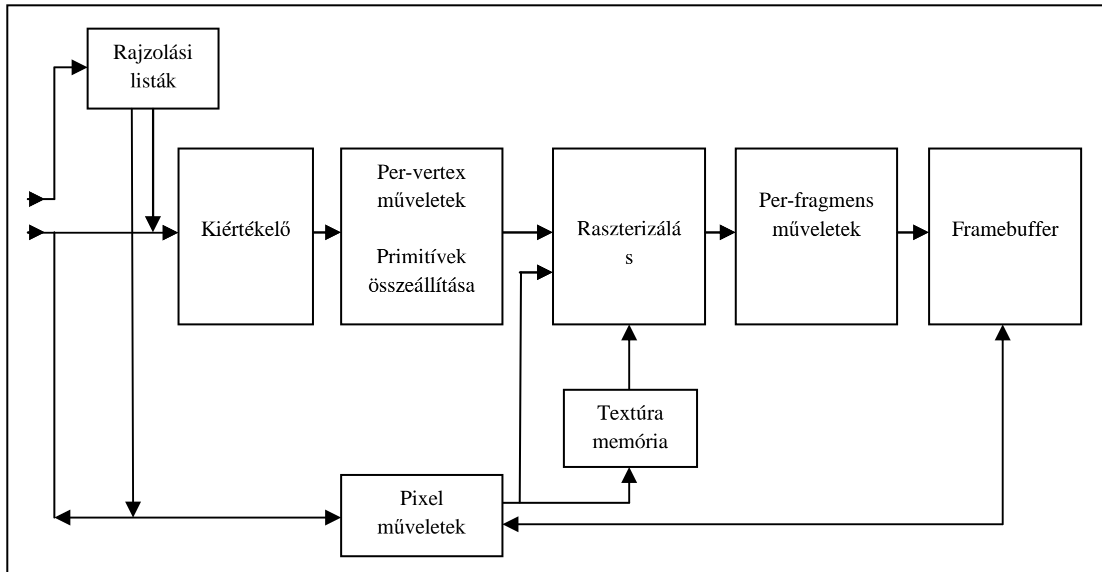
[GL8] http://www.opengl.org/registry/specs/ARB/vertex_program.txt

[GL9] http://www.opengl.org/registry/specs/ARB/fragment_program.txt
[GL10] http://www.opengl.org/registry/specs/ARB/occlusion_query.txt
[GL11] http://www.opengl.org/registry/specs/ARB/shader_objects.txt
[GL12] http://www.opengl.org/registry/specs/ARB/vertex_shader.txt
[GL13] http://www.opengl.org/registry/specs/ARB/fragment_shader.txt
[GL14] http://www.opengl.org/registry/specs/ARB/shading_language_100.txt
[GL15] http://www.opengl.org/registry/specs/ARB/color_buffer_float.txt
[GL16] http://www.opengl.org/registry/specs/ARB/texture_float.txt
[GL17] http://www.opengl.org/registry/specs/EXT/separate_specular_color.txt
[GL18] http://www.opengl.org/registry/specs/EXT/vertex_weighting.txt
[GL19] http://www.opengl.org/registry/specs/ATI/envmap_bumpmap.txt
[GL20] http://www.opengl.org/registry/specs/ATI/fragment_shader.txt
[GL21] http://www.opengl.org/registry/specs/ATI/text_fragment_shader.txt
[GL22] http://www.opengl.org/registry/specs/ATI/texture_env_combine3.txt
[GL23] http://www.opengl.org/registry/specs/NV/texgen_reflection.txt
[GL24] http://www.opengl.org/registry/specs/NV/texgen_emboss.txt
[GL25] http://www.opengl.org/registry/specs/NV/register_combiners.txt
[GL26] http://www.opengl.org/registry/specs/NV/register_combiners2.txt
[GL27] http://www.opengl.org/registry/specs/NV/texture_shader.txt
[GL28] http://www.opengl.org/registry/specs/NV/texture_shader2.txt
[GL29] http://www.opengl.org/registry/specs/NV/texture_shader3.txt
[GL30] http://www.opengl.org/registry/specs/NV/fragment_program.txt

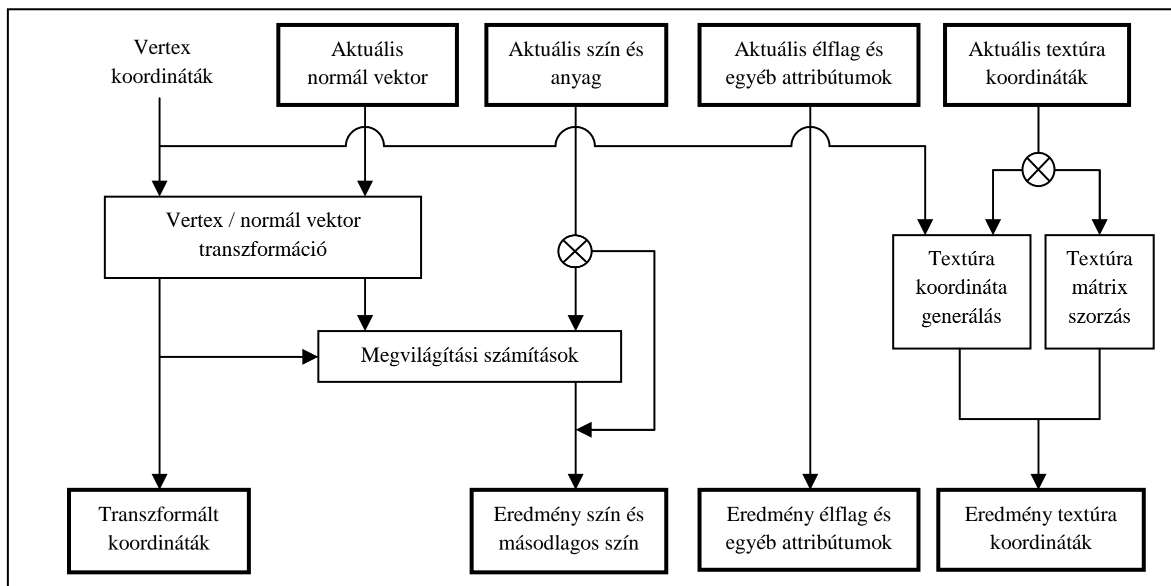
Tom Nuydens publikációi:

[TN1] An Overview of Bump Mapping Techniques,
http://delphi3d.net/articles/viewarticle.php?article=bumpmapping.htm
[TN2] Phong For Dummies,
http://delphi3d.net/articles/viewarticle.php?article=phong.htm
[TN3] Occlusion Culling,
http://delphi3d.net/articles/viewarticle.php?article=occlusion.htm

Függelék



2.1. ábra. Az OpenGL működésének blokk diagrammja.



2.2. ábra. Az OpenGL vertex feldolgozó csővezeték működésének blokk diagrammja.

3.1. ábra. Első generációs 3D grafikus kártya: 3dfx Voodoo.

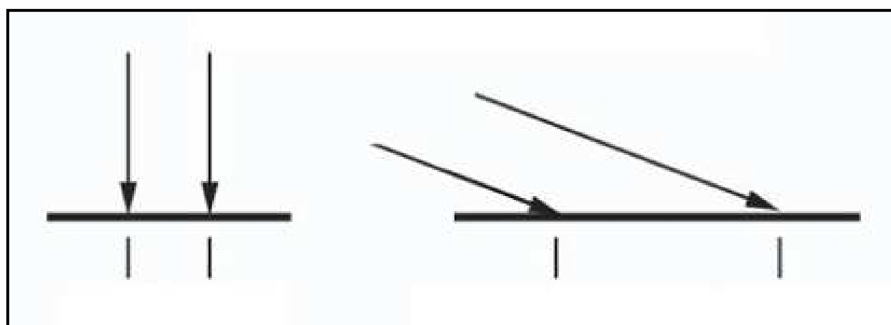
3.2. ábra. Második generációs 3D grafikus kártya: NVIDIA GeForce 256.

3.3. ábra. Harmadik generációs 3D grafikus kártya: ATI Radeon 8500.

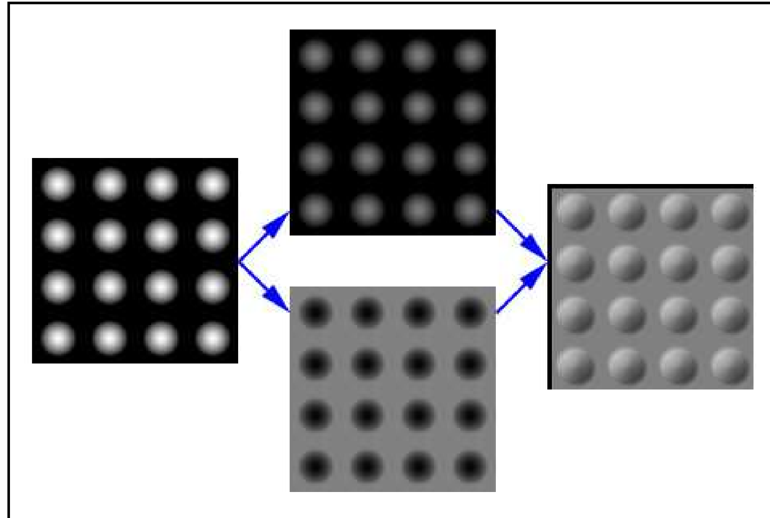
3.4. ábra. Negyedik generációs 3D grafikus kártya: ATI Radeon 9700.

DirectX Pixel Shader verzió	Megfelelő OpenGL kiterjesztések	Hardver támogatás
Pixel Shader 1.1	NV_register_combiner2 NV_texture_shader	GeForce3+ NV20+
Pixel Shader 1.2	NV_register_combiner2 NV_texture_shader2	GeForce3+ NV20+
Pixel Shader 1.3	NV_register_combiner2 NV_texture_shader3	GeForce4 Ti+ NV25+

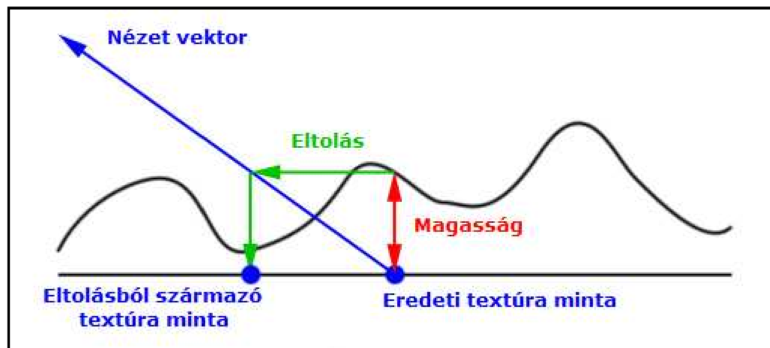
4.1. ábra. DirectX 8 pixel shaderek és a neki megfelelő OpenGL kiterjesztések.



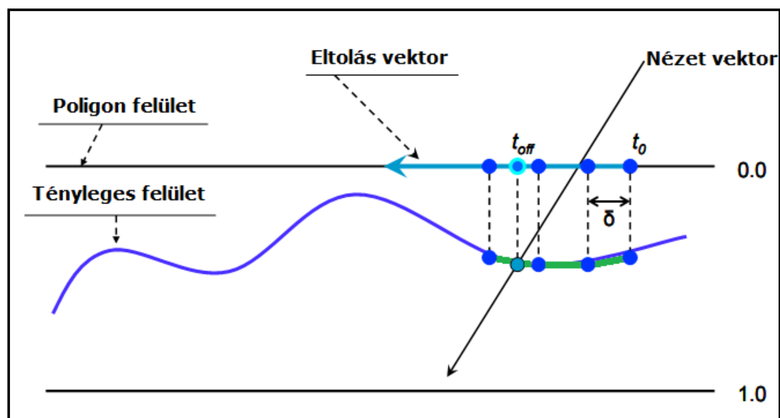
6.1. ábra. Lambert törvényének megfelelően a meredekebb szögben érkező fény nyalábok energiája nagyobb felületen oszlik el.



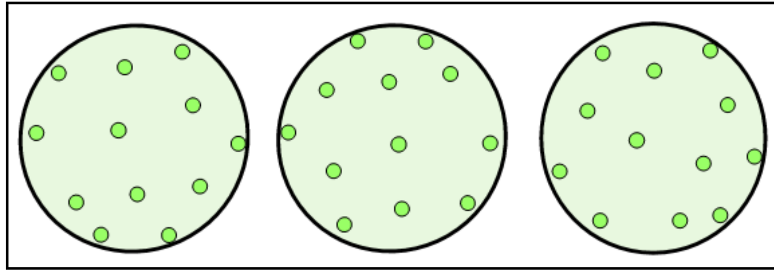
7.1. ábra. Emboss bump mapping.



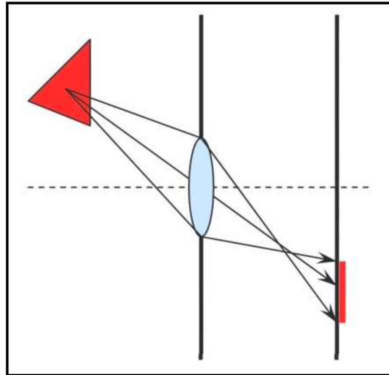
7.2. ábra. Téves eltolási érték a parallax mapping technika alkalmazásakor.



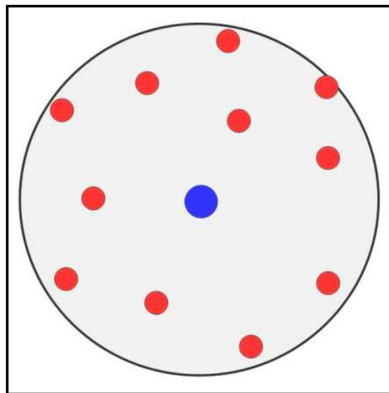
7.3. ábra. A parallax occlusion mapping által kiszámított eltolási mérték.



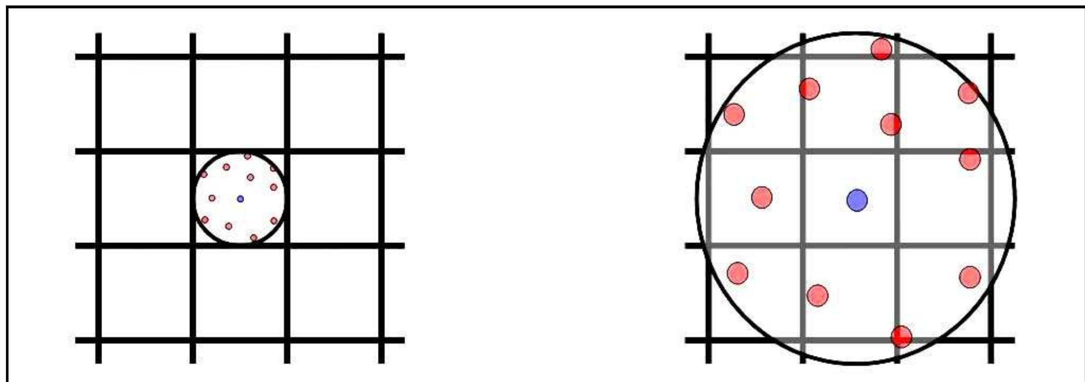
7.4. ábra. Percentage Closer Filter Kernel.



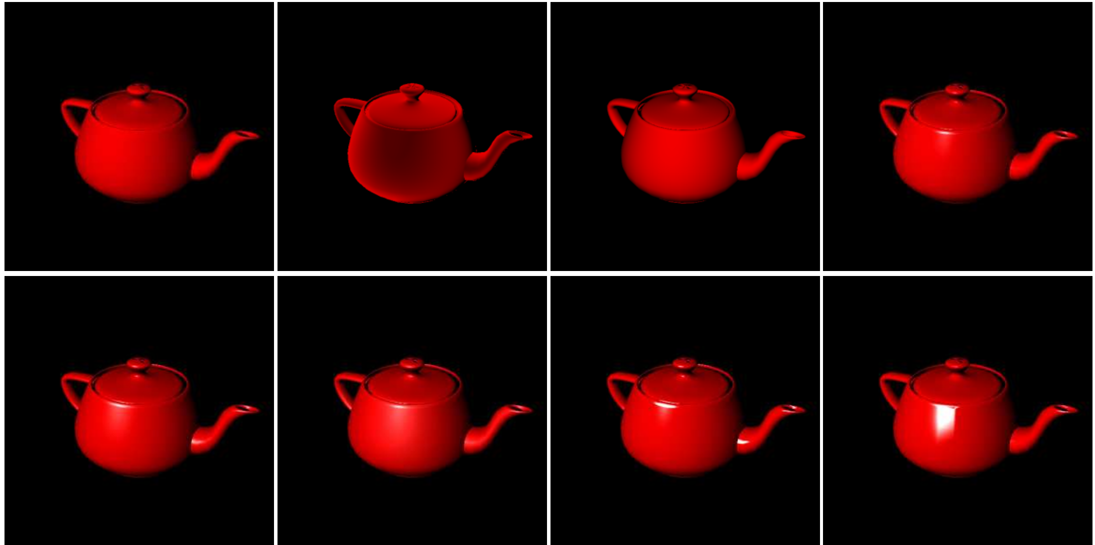
7.5. ábra. Valódi lencsék esetében kialakulnak az úgynevezett lágy pontok.



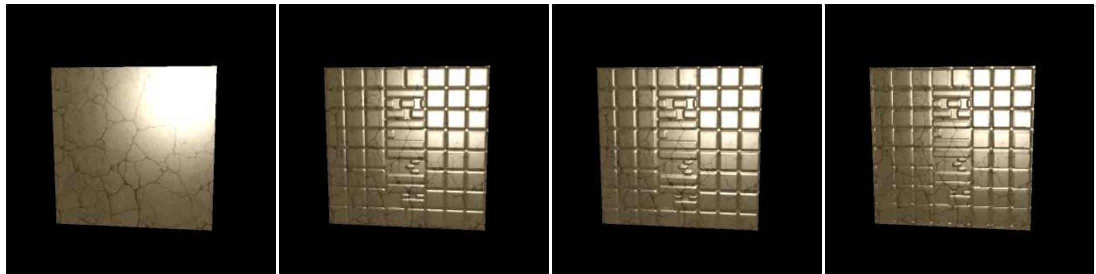
7.6. ábra. Poisson Filter Kernel.



7.7. ábra. Lágy pontok előállítása a poisson szűrő segítségével.



Megvilágítási modellek: (felső sor, balról jobbra) Lambert, Oren-Nayar, Minnaert, Phong, (második sor, balról jobbra) Blinn-Phong, Schlick, izotróp Ward, anizotróp Ward modell.



Bump mapping módszerek: (balról jobbra) bump mapping nélkül, normal mapping, parallax mapping, parallax occlusion mapping.

Utófeldolgozási technikák: (balról jobbra) utófeldolgozás nélkül, motion blur, depth of field effektus, bloom hatás.