DIPLOMAMUNKA

TÓTH ÁDÁM

DEBRECEN

2008.

Debreceni Egyetem

Informatikai Kar

GRAFIKUS MOTOR FEJLESZTÉSE DIRECTX 9.0C-VEL

Konzulens:
Dr. Kovács Emőd
főiskolai docens

Készítette:
Tóth Ádám
programtervező matematikus

Debrecen

2008.

Tartalomjegyzék

1. Bevezetés

1. 1. Dióhéjban a modern játékfejlesztés kialakulásáról

Hétköznapjaink meghatározó szereplője a multimédia. Tájékoztatási célja mellett egyre nagyobb teret hódít a szórakoztatás, azon belül is az interaktív szórakoztatás területén. Ide sorolhatók a *számítógépes játék*ok, mint interaktív multimédiás számítógépes alkalmazások.

Az 1990-es évek második felében indult meg igazán a videojátékok forradalma. A hardvergyártók – főként a *grafikus chipsetgyártó*k – felismerték a multimédiás alkalmazásokban, a mérnöki rendszerekben és a videojátékokban rejlő üzleti lehetőségeket – ezen területeken belül igény volt a nagyobb teljesítményű hardverekre. A programozóknak lehetősége nyílt a fejlődő hardverre fejlettebb grafikus alkalmazásokat írni. A matematikusok egyre nagyobb komplexitású grafikus problémákat szemléltethettek számítógépen, interaktív módon. A grafikusok pedig digitális művészekké váltak.

1. ábra. Részlet a Ubisoft Far Cry 2. c. játékából[20].

A videojátékok így egyre komplexebbek, valósághűbbek lettek, kiléptek az ún. „garázsprojektek” sorából, eladható termékké váltak. A kialakult játékpiac, a játékostársadalom (mint a fogyasztók) és a termelés elvárt magas színvonala szükségessé

tette, hogy a termelést egy modern játékfejlesztő cég végezze. A magas nyereség pedig életképessé tette ezen cégeket.

A játékostársadalom ráadásul egy jól definiálható tömeg lett, melyet csoportokra oszthatunk igényeik, elvárásaik és lehetőségeik szerint, s melynek tagjait a megfelelő módon megcélozva kielégíthetők az üzleti érdekek. Vagyis a termék jellegét és minőségét (javarészt) a célközönség elvárásai határozzák meg.

Kétféle játékos létezik. A **hardcore játékos** időt, pénzt és energiát nem sajnálva szórakoztatni szeretné magát. Az általa elvárt játékszínvonal meglehetősen magas. Audionális és vizuális szempontból maximális kielégülésre vágyik. Ebből következően az ilyen játékok igénylik a legrészletesebb kidolgozást, legmodernebb technológiákat (*cutting edge technology*). Ebből kifolyólag hosszú a fejlesztési idejük, drágák, gépigényük nagy. Úgy is szokás nevezni ezeket, hogy „dobozos játékok".

A másikféle játékos a **casual játékos**. Valamilyen tényező korlátozza a játékost: például kevés az ideje, elavult a számítógépe, nem rendelkezik az adott számítógépen megfelelő jogokkal (például dobozos játék telepítéséhez a netkávézóban), azonban mégis szórakozni, játszani szeretne. Őt célozzák meg a minijátékok és a webes játékok (tipikusan *flash* játékok). Ezek fejlesztése nem igényel hosszabb időt, gyakran előfordul, hogy egyetlen ember a fejlesztő, olcsóak vagy ingyenesek, nem kívánnak erős hardvert.

Vizsgáljuk meg a folyamatot a termelők oldaláról is!

A chipsetgyártók hardvereiket már annak szellemében fejlesztették, hogy a multimédia termékei ezen eszközök segítségével megfelelő színvonalon elégítsék ki a célközönséget. A művészi szépség, valósághűség és a sebesség mind-mind fontos tényezők. A grafikus algoritmusok átkerültek áramkörök szintjére, a számítási kapacitásokat megsokszorozták, a szoftverfejlesztők igényeihez alkalmazkodtak.

Megemlítendő még az **API** (*Application Programming Interface*) gyártók versengése (főként az *OpenGL* és a *Microsoft DirectX* API-k között), a játékkonzolok megjelenése (*XBox*, *PlayStation*, *Nintendo Wii*, stb...) illetve a mobiljátékok – mint technológia – megjelenése is; ezek szintén hatottak a játékok világára.

Mindez oda vezetett, hogy a multimédiafejlesztések, azon belül is a játékfejlesztés önálló iparággá nőtte ki magát, nagy vállalatok termelési profiljává avanzsált. Nem ritka, hogy egy-egy ilyen párszáz fős cég több játékprojektet is visz egyszerre. Nem csupán

programozók, de grafikusok, modellezők, designerek, forgatókönyvírók, zeneszerzők, dialógusszerkesztők, koreográfusok, minőségellenőrzők, stb... is betársultak a játékfejlesztésbe. Matematikusok, fizikusok és egyéb szakemberek kutatási tevékenységet folytathatnak egy ilyen cégen belül.

Miért terjed ki ennyi területre a játékfejlesztés?

Természetesen a grafika egy játék szempontjából nem minden. Megfigyelhető, hogy először az érzékszervekre ható tényezők tartják a játékost a gép előtt – grafika, látványvilág, hangok, zene, stb... A következő tényező a játék története, menete (*gameplay*). Végül a játékhoz kiadott újabb fejlesztések, bővítések, hálózaton keresztüli játszhatóság, stb... lesz az, amely a játékos érdeklődését hosszú ideig fenntarthatja. Emiatt kellenek olyan emberek, akik a játékfejlesztés különböző szakaszaiban a saját szakterületükön belül nagy fokú hatékonysággal tudnak dolgozni.

1. 2. Témaválasztás

Azért választottam a háromdimenziós grafikus megjelenítő motor témakörét diplomamunkám tárgyának, mert ez a terület közel áll hozzám, ugyanakkor egy manapság igen lényeges kutatási területe a játékfejlesztésnek, mérnöki rendszereknek, modellező, tervező szoftvereknek. Évről évre újabb és újabb algoritmusok jelennek meg a háromdimenziós grafika és a grafikus megjelenítő motorok legkülönfélébb területein belül, melyek megoldást kínálnak különféle problémákra.

A grafikus technikákon és megoldásokon túl a hardver és az API nyújtotta lehetőségek minél jobb kiaknázása és a korlátok betartása közötti arany középút megtalálását tűztem ki célul. Természetesen erre számos lehetőség kínálkozik, az Interneten is fellelhetők nyílt forráskódú motorok (*open source engine*). Egy általam tervezett és implementált motor ismertetésére szeretnék sort keríteni ebben a munkában. Megpróbáltam hardver, API és programozói tapasztalataimra hagyatkozni, minél kevesebb mástól vett forráskódrészletet felhasználni.

Továbbá azért a DirectX-et választottam az OpenGL-lel szemben, mert ennek programozásában nagyobb tapasztalatom van, felépítése, szerkezete logikusabb számomra. Továbbá a rendszer buktatóit, megoldásait jobban ismerem. Az általam használt DirectX SDK (*Software Development Kit*) verziója: DirectX 9.0c 2007. novemberi SDK-ja.

Fejlesztői környezetként a *Microsoft Visual Studio 2005 Express* változatát használtam, mely ingyenes letölthető és használható.

A fejlesztéshez a C++ nyelvet választottam.

Diplomamunkámban ismertetni fogok egy konkrét, háromdimenziós jelenet megjelenítésére alkalmas, shader alapú grafikus motort, annak szintjeit és egyes elemeinek működését, a bemenő adatokat leíró fájlok szerkezetét, optimalizálási lehetőségeket, kapcsolódási pontokat más játékmotorbeli alrendszerekkel. Továbbá saját *posteffect rendszer*emet is hozzá fogom kapcsolni az elkészült grafikus motorhoz – egy példa az önálló grafikus modulok készítésére és bekapcsolására a grafikus motorba.

2. A játékmotor

2. 1. A játékmotor szerkezete

Minden videojáték mögött meghúzódik egy komplex, jól strukturált felépítésű rendszer, egy program. Ezt nevezzük játékmotornak. Feladata, hogy a jól definiált bemenő adathalmazból jól definiált kimenetet állítson elő.

Ennél konkrétabban fogalmazva a játékmotor feladata, hogy a leíró adatokat, gameplay scripteket, grafikákat, modelleket, zenéket, hangokat, hálózatról érkező csomagokat, felhasználó általi interakciót, stb... feldolgozva előállítson vizuális, zenei, egyéb adat-, stb... kimenetet, melyet a felhasználó (itt *játékos*) az előírt minőségi követelményekben foglalt módon érzékelni fog, feltéve, hogy rendelkezik az előírt rendszerkövetelményekkel.

Látható, hogy a játékmotornak sokféle funkciója van. Adatbeolvasás, képelőállítás, különböző fizikai számítások többek között. Ezen funkciók szerencsére osztályozhatók, elkülöníthetők egymástól. Ez a felosztás fogja indukálni magát a program szerkezetét is.

A játékmotor több **alrendszer**ből vagy motorból (*system*, *subsystem, engine*) épül fel, melyben minden egyes alrendszer egy jól körülhatárolt feladatosztály elvégzésére specializált. Természetesen szükség van egy átfogó **keretrendszer**re (*framework*) is, mely az egyes alrendszerek működését felügyeli. A keretrendszer végzi az ablakozással kapcsolatos műveleteket vagy egyéb operációs rendszerműveleteket is.

Mivel az alrendszerek jól definiáltak, így lehetőség van az alrendszerek párhuzamosítására. Optimalizáció szempontjából ez lényeges kérdés.

Továbbá az alrendszerek mégsem zárhatók el teljesen egymás elől, valamiféle kommunikációs lehetőséget biztosítani kell közöttük. Például közös memóriaterületen keresztül vagy üzenetváltással. Nem jó módszer, ha az alrendszerek egymást közvetlenül tudják befolyásolni. Vagyis a közös memóriaterületet és az üzenetek tárolására szánt alrendszereknek dedikált postafiókokat a keretrendszernek kell biztosítania.

Az alrendszerek ilyen szintű függetlensége garantálja ugyanis az alrendszerek hatékony cseréjét. Ez gyakran előfordul technológiaváltásnál – például az OpenGL alapú grafikus motort lecserélik DirectX alapúra.

Ugyanakkor a teljes függetlenség sem egy tökéletes megoldás. A keretrendszer számára egyes esetekben az információtovábbítás nehézkes, bonyolult lehet, esetleg még a menedzselés miatt lassú is. Egy kritikusabb információcserére érdemes másik módszert találni. Ilyen lehet az alrendszerek egymás alá rendelése.

Legalsó szinten helyezkedhet el például a hiba- vagy kivételkezelő alrendszer, logmentő alrendszer, a memória- és fájlműveleteket végző alrendszer, stb...

Az én esetemben létrehoztam egy fájlbeolvasást kezelő osztályt. Az egyszerű lemezről történő beolvasáson kívül implementálható bele a csomagból való beolvasás is – ha állományaink csomagolva vannak fájlokba, esetleg tömörítve. Látható azonnal ennek óriási előnye: az ezt használó alrendszereknek nem kell törődniük azzal, honnan olvassák be a fájljaikat.

Egy általános játékmotor szerkezete tehát vertikálisan és horizontálisan is felosztott. Horizontális az alrendszerekre való bontása szerint, vertikális az alárendelési viszonyoknak megfelelő rétegzettsége szerint.

2. 2. Az alrendszerek

Az alrendszereknek tehát cserélhetőknek kell lenniük, interfészükön keresztül lehet velük csak kommunikálni, párhuzamosíthatók egymással. Tesztelésnél nem csak egyesével, hanem együtt is kell figyelni és ellenőrizni a működésüket!

Előfordul, hogy egy-egy alrendszer nem saját fejlesztésű – a cégek dönthetnek úgy, hogy egy már létező és jól működő technológiát vásárolnak meg. Ennek oka lehet például az, ha az új alrendszer kifejlesztése többe kerülne mint egy már meglévő beszerzése. Vásárláskor sincs azonban minden rendben, mert az új modult idomítani kell a már meglévő technológiákhoz (például a fájlbetöltő alrendszert felvezetni bele) és tesztelni kell önmagában és a többi alrendszerrel együtt is.

Egy játékmotorban általában az alábbi alrendszerek találhatók meg:

- grafikus motor (render engine),
- fizika motor,
- hang és zene motor,
- mesterséges intelligencia alrendszer,
- hálózati alrendszer,
- inputkezelő alrendszer,
- scriptrendszer,
- erőforrás-kezelő alrendszer (alárendelve a többi alrendszernek),
- egyéb alrendszerek – számításokhoz, videólejátszáshoz, stb...

2. 3. Implementációs kérdések

Miután felosztottuk megfelelő módon a rendszert, megterveztük egyes részeinek működését és jól definiált elvárásaink vannak ezekkel szemben, azt mondhatjuk, a játékmotor tervét specifikáltuk.

A tervezési szakaszt követi az implementációs szakasz, melyben a tervezett interfészek „mögé" tényleges kódot írunk.

Olyan **design minták** (*design pattern*) követhetők, mint a **Singleton** (osztály egyetlen példánnyal – grafikus eszközkezelő osztály), **Factory** (kívánt típusú új objektumot lehet kérni tőle – játékelemek dinamikus létrehozása), **Observer** (több objektum figyeli egynek a működését, amikor annak állapota változik, a megfigyelők értesítést kapnak – konstans–paraméter frissítés), **Utility** (statikus tagokat tartalmazó, nem példányosítható osztály – matematikai függvények gyűjteménye), **Adapter** (interfész elfedése olyan másik interfésszel, melyet egy osztály vár – modulok beépítése), vagy a **Command** (utasítás objektumba csomagolva – üzenet alapú kommunikáció) minta[11][13].

A kritikus, gyakran hívott függvényeket illik **inline** módosítóval ellátni, ekkor ugyanis a fordító, ha teheti, a függvény törzsét a hívás helyére másolja, így a függvény hívásából eredő veremműveletek és egyéb műveletek elmaradnak, gyorsabbá téve a kódot[14].

Legyünk mindig következetesek! Ugyanazzal a dinamikus tömb, sztring, lista, stb... osztályokkal dolgozzunk (például az *STL tárolók*kal), memóriakezeléshez, fájlbetöltéshez használjuk egy erre a célra megírt réteg szolgáltatásait, az osztályok és tagjaik elnevezésében kövessünk egy előre meghatározott mintát, stb...

Megjegyzés. Amikor több ember dolgozik ugyanazon a projekten, a következetesség az egyénektől függő stílus miatt csorbulhat, ennek azonban nem szabad gondot jelentenie!

3. A grafikus motor

3. 1. Általános áttekintés

Diplomamunkám központi témája a grafikus motor (*render engine* vagy *graphics engine*). A játékmotor elhagyhatatlan eleme. Működése jól párhuzamosítható a többi alrendszerével, melyek közül többel is kommunikálnia kell (fizika motor, mesterséges intelligencia alrendszer, számításokat végző alrendszerek, scriptrendszer, stb…).

Egyesekkel a „szélessávú” közös memóriaterületen keresztül kommunikál; tipikusan ilyen eset, amikor a fizika motortól több száz objektum transzformációs mátrixát kell lekérdeznie. Másokkal a kevesebb, kisebb, alkalmanként érkező üzenetek segítségével cserél információt; például jelenetbetöltésnél az új megvilágítási beállításokról kap értesítést.

Az erőforrás-kezelő alrendszer szolgáltatásait közvetlenül használja, hiszen nagyon sok adatot dolgoz fel merevlemezről, memóriából.

Az én esetemben a többi alrendszer hiányában saját maga tölti be a nem közvetlenül hozzá kapcsolódó információkat is (a jelenetet, a világról szóló leírásokat).

A diplomamunkámban ismertetett grafikus motor DirectX alapú, a Direct3D szolgáltatásait használja.

3. 2. A tökéletes grafikus motor

3. 2. 1. Korlátok és igények

Nem beszélhetünk általános értelemben véve tökéletes, univerzális grafikus motorról. Vannak olyan szempontok, melyekből nézve egy adott motor szinte tökéletesen kihasználja a hardver és a platform nyújtotta lehetőségeket, azonban mindenképpen lesznek a motornak gyenge pontjai. Sebesség szempontjából fontos a szűk keresztmetszetek (*bottleneck*) kiszűrése (ld. a **3. 9. 1. Szűk keresztmetszetek kiszűrése** c. fejezetet), látvány szempontjából pedig a vizuális mellékhatások, nem kívánt vizuális jelenségek (*visual artifact*) kijavítása. A sebesség és a valósághűség azonban gyakran egymás ellen dolgozó tényezők.

Továbbá nem hagyhatóak figyelmen kívül a rendszer jellemzői, melyen az alkalmazást futtatni szeretnénk, azon belül is a rendszermemória és a videomemória kapacitása, a videokártya gyártója, típusa, egyéb jellemzői.

A hardveres korlátok (rendelkezésre álló memória, grafikus kártyák jellemzői és lehetőségei – az ún. *card caps*) mérlegelésekre késztetnek bennünket, kompromisszumos megoldásokra van szükség. Gyakran szembesülnek a grafikus algoritmusok programozói a ténnyel, hogy ami gyors, az nem biztos, hogy vizuális mellékjelenségektől mentes illetve, ami igazán látványos, az nem minden hardveren fog működni vagy nem lesz gyors.

Általános motorok helyett speciálisabb motorok léteznek.

A játék koncepciója alapján minden grafikus probléma megoldására kiválasztjuk a legmegfelelőbb algoritmust (vagy akár több algoritmust ugyanazon problémára), és ezt tovább specializálhatjuk, tökéletesíthetjük, optimalizálhatjuk a játéknak megfelelően. Különböző részletezettségi szintek (*level of detail*) szerint is beállíthatjuk a kiválasztott, implementált módszereket.

3. 2. 2. Példa egy szituációfüggő problémára

Árnyékszámítással fokozhatjuk a grafikus élményt, de kizárólag a sebesség rovására. Az árnyékszámítás azonban elengedhetetlen egy modern motorban. Valójában a kérdés

csupán az algoritmus milyensége. Az árnyékszámító algoritmusokat nem tudjuk sorrendbe állítani. A jó nem szinonimája sem a látványosnak, sem a gyorsnak, sem a memóriabarátnak.

Az árnyéktérképezést használó algoritmusok osztálya összehasonlítható az árnyéktérfogatot számító algoritmusok osztályával. Mindkét fajtának van előnye is és hátránya is. De ha egy általános játékmotorba szeretnénk árnyékszámítást beépíteni, gondban lennénk, melyiket célszerűbb. Ilyen esetben mindkét változat implementálását el kell végezni.

Árnyéktérképező algoritmusok	**Árnyéktérfogat-számító algoritmusok**
közepes, nagy látótávolságú jeleneteknél	kicsi látótávolságú jeleneteknél
helyenként áttetsző textúrájú objektumok	nem támogatja az áttetsző textúrákat
sok memóriát igényel	kevés memóriát igényel
puha árnyékhatár képezhető	puha árnyékhatár nehezen képezhető
éles árnyékhatár nehezen képezhető	éles árnyékhatár alaptulajdonság
nagy esély az anomáliákra („kikockásodás”)	főként hardverkorlátok okozta anomáliák
továbbfejlesztett: LiSPSM, CSM, VSM (Variance Shadow Mapping)	továbbfejlesztett: Carmack’s Reverse
egyszerű általános algoritmus 1. árnyéktextúrába mélységszámítás az árnyékvető objektumokra 2. ezen textúra felhasználása a normális rajzolási menetben	több lépéses általános algoritmus 1. objektumok fényforrás szemszögéből vett kontúrját a végtelenbe kell nyújtani 2. stencilpuffert használva kirajzoljuk a térfogat felénk néző lapjait 1 bites növeléssel 3. stencilpuffert használva kirajzoljuk a térfogat hátsó lapjait 1 bites csökkentéssel 4. a stencilpuffert felhasználva számolunk a pixelekben árnyékértéket

Természetesen ez lesarkított ajánlás, hiszen egy jól átgondolt térképező algoritmust felhasználhatunk akár nagy látótávolságú játékoknál is. Ilyen például a *LiSPSM* (*Light Space Perspective Shadow Mapping*) ötvözése a *CSM*-mel (*Cascaded Shadow Mapping*).

A fenti listákat összehasonlítva eldönthetjük, melyik árnyékszámító modellt használjuk játékunkban. Nem árt azonban a továbbfejlesztett változatokat is átböngészni; ezek ugyanis a típusok egyes hiányosságait küszöbölik ki.

3. 2. 3. Grafikus és játékmotorok a piacon

Egy-egy jól eltalált ötleten alapuló grafikus motort vagy teljes játékmotort áruba bocsáthat a fejlesztője. Vagyis forráskódját pénzért publikálhatja a vásárlónak licenszszerződés keretében, korlátozott felhasználási feltételekkel.

Természetesen több kényelmi funkciót is megvalósítanak ezek a motorok, engine csomagok: kitakarásvizsgálat, árnyékszámítás, fénytérképezés, scriptrendszer, fizika, hálózatkezelés (*chat* és *lobby* támogatása többek között); fájlformátumokhoz exportáló *plugin*ek, stb... Az alapdolgokon túl a kiegészítő funkciók fejlesztése prototípusszerűen történhet – kísérletek sikeres vagy kevésbé sikeres eredményeként találkozhatunk új szemléletű technikákkal.

Ilyen motorok, melyek még nem igazán bizonyítottak, **Lengyel Erik C4** nevű motorja[15], az **Irrlicht**[16] vagy a **Virtools**[17].

Másik bevett szokás, hogy egy kész, működő játék motorját adják el licenszre. Ekkor a motor teljesítménye már bizonyított egy adott koncepcióra építve, tehát nem veszünk zsákbamacskát vele – általában ezen motorok jóval drágábbak is. Ide sorolható az **id Software Doom3** motorja, a **Valve Half Life 2** c. játékának **Source** motorja vagy az **Epic Games Unreal 3** motorja.

3. 3. A megjelenítés menete

Mielőtt tervezni kezdenénk a grafikus motor felépítését, fontos megértenünk, hogyan működik a háromdimenziós képalkotás, megjelenítés (*rendering*), mik az egyes lépései, milyen úton áll elő kép a betáplált adatokból, paraméterekből.

1. Adatok előállítása.
    - Objektumleíró pufferek beolvasása a merevlemezről vagy a memóriából. Például egy meshfájl betöltése.
    - Adatgenerálás algoritmikus módon. Például felparaméterezett gömb előállítása.

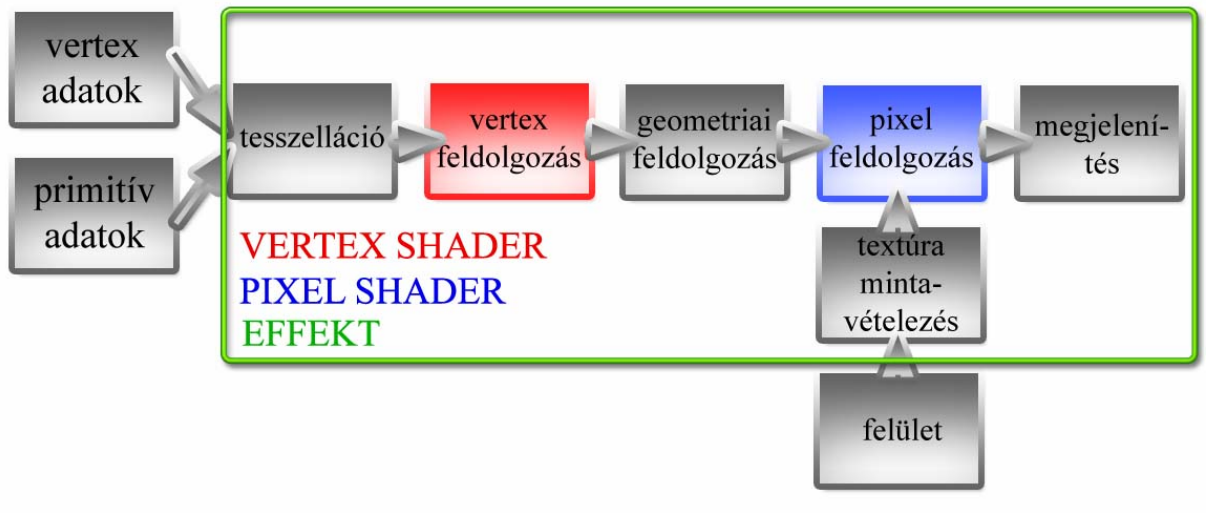
2. Adatok előkészítése esetleges további műveletek és a GPU számára.
    - Közös attribútumok, anyagok, textúrák alapján történő rendezés.
    - Távolság alapú rendezés, például nem alfakeveréses objektumok esetén elölről hátrafelé.
    - Megfelelő vertexdeklaráció. Például optimális 32 bájtos vertexadatok.

3. Geometriai szintű műveletek.
    - Transzformációk és megvilágítás számolása a vertexekben (*TnL, Transformation and Lighting*).
    - Textúrakoordináták generálása, transzformációja.
    - Geometriai primitívek vágása, hátsó lap eldobás, stb...

4. Raszterizáció.
    - A háromszöglapok fragmentumokra bontása interpolációval (szín, textúrakoordináták, mélység, stb...).
    - Különböző tesztelések (mélységi tesz, stencil- és alfateszt).
    - Fényintenzitás-, köd-, színszámítás, alfakeverés.

5. Megjelenítés.
    - Gamma korrekció. Háttérpuffer tartalma a képernyőre kerül.

A felvázolt megjelenítési elv tulajdonképpen a DirectX csővezetékére illik. A shadermodellt támogató grafikus chipsetekben alkalmazható **vertex- és pixelshader**ek olyan programocskák, melyek ezen csővezeték működésébe épülnek be a megfelelő pontokon.



2. ábra. A Direct3D csővezetéke.

A vertexshader a vertextranszformáció és vertexbeli adatokkal való számolások elvégzését szolgálja, a pixelshader pedig a textúrák mintavételezéséből és a poligonok felületén interpolált egyéb adatokból valamilyen színadat előállítását végzi el.

Segítségükkel az állapotokkal szabályzott *fix működésű csővezeték* (*fixed function pipeline*) egy sokkal rugalmasabb, programozható csővezetékké alakítható.

Ugyanakkor lesznek olyan állapotok, melyek hatása shaderekkel nem helyettesíthető. Ilyen a tesszelláció, a megjelenítés, a geometriai feldolgozás és a mintavételezés módjára ható állapotok.

Az effektek olyan shadereket felhasználó eszközök a DirectX-ben, melyekkel többmenetes, állapotváltásokra módot adó rajzolási folyamatokat (*technikák*at) tudunk definiálni.

Modern játékokban a megfelelő látványvilág eléréséhez nélkülözhetetlenek a finoman összehangolt, többféle beállítás mellett is működő shaderek és effektek.

3. 4. Célkitűzések és fejlesztési specifikációk

Mielőtt hozzálátnánk a motor konkrét megtervezéséhez és kifejlesztéséhez, először tisztáznunk kell a koncepciónak megfelelő célkitűzések mellett néhány, a fejlesztésre vonatkozó alapelvet, specifikációt is. Ilyen specifikáció lehet többek között, hogy milyen grafikus API szolgáltatásait fogjuk igénybe venni, milyen platformra, operációs rendszerre fejlesztünk vagy milyen minimális hardveradottságokat várunk el a felhasználói oldalról. Kérdés, hogy melyek lesznek azok a célok, melyeket ezen specifikációk rögzítése mellett meg **szeretnénk** és meg **tudunk** valósítani. Általában a gyengébb specifikációk mellett a koncepcióban foglalt célok egy részéről le kell mondanunk, egy másik részét pedig korlátozott módon tudjuk csak megvalósítani.

A fejlesztési irányvonalat meghatározó specifikáció lehet a *multi-platform* jelleg. Ha a grafikus motort több platformon is futtathatónak szeretnénk elkészíteni, csomagoló osztályok tömkelegét kell megírnunk, melyek elfedik a hardver-, platform-, API-, operációs rendszerspecifikus elemeket (*wrapper level*). Sajnos ez a réteg nehezen egységesíthető a fentebb felsorolt elemek olykor igen erősen eltérő filozófiája miatt. Lásd: Direct3D ↔ OpenGL.

Diplomamunkám esetére az alábbi fejlesztési alapelveket hoztam:

API	: DirectX 9.0c.
Platform	: PC.
Op. rendszer	: Windows XP.
Hardver	: olyan videokártya, mely rendelkezik legalább 1.0-s shadermodellel.

3. 5. Fejlesztő oldali elvárások a grafikus motorral szemben

Vannak alapvető tulajdonságok, melyekkel egy modern, jó grafikus motornak rendelkeznie kell.

A *gyorsaság* egy elengedhetetlen tényező. Mivel interaktív multimédiás alkalmazásról van szó, ezért lényeges a rajzolás sebessége. Ennek mértékegysége az **FPS** (*frame per second*), vagyis egy másodperc alatt hány képet tud kirajzolni. Az FPS nem egy statikus, a motorra jellemző mérőszám. Nagyban függ a hardvertől, a jelenet bonyolultságától és az aktuális rendszerbeállításoktól.

Ugyanakkor optimalizálással, ügyes trükkökkel és egyéb segédinformációkkal (heurisztikával) fokozható egy jelenet sebessége. Törekedni kell a minél magasabb FPS értékre. Az emberi szem számára 20-25 FPS-től tűnik folyamatosnak a lejátszott animáció, ez alá semmiképp sem szabad esni.

Gyakran előfordul, hogy ezt a szintet csak vizuális élményfokozó technikák kikapcsolásával érhetjük el (dinamikus tükröződésszámítás kikapcsolásával például). Ilyenkor megfontolandó más algoritmusok használata, optimalizálás több szinten (ld. a **3. 9. Optimalizáció** c. fejezetet), jelenet poligonszámának csökkentése statikusan vagy számítások útján.

Másik fontos tényező a *könnyű bővíthetőség* újabb modulokkal. Már említettem a **2. A játékmotor** c. témakörben, hogy az egyes alrendszerek cserélhetők legyenek. Hasonló módon lehetnek ilyen részek a grafikus motoron belül is – például az árnyékszámítást végző modult lecseréljük. A bővíthetőség is hasonlóan fontos, hiszen bármikor beleakadhatunk egy szép, gyors technikába, melyet szeretnénk játékunkba integrálni, és a motornak készen kell állnia egy ilyen modul fogadására.

Néhány esetben nem célravezető a modul teljes idomítása, mert átláthatatlanná teheti a modult és hosszú ideig tarthat a fejlesztés.

Erről konkrétabban fogok értekezni a **3. 11. Modulok kapcsolása** c. fejezetben. Diplomamunkám esetén egy posteffekt modult kapcsoltam be a grafikus motoromba.

A grafikus motornak *fejlesztőbarát*nak kell lennie. Nem szabad **anti-design minták**[12] csapdájába esnünk (túl sok osztály, túl nagy osztályok, globális változók, univerzális osztályok, stb...), mert emiatt átláthatatlan lesz az alrendszer. Logikusan

megtervezve, következetesen kell felépítenünk az osztályhierarchiát, mely a követelményeinknek eleget tesz.

Grafikus algoritmusok szempontjából nem rossz dolog, ha többféle algoritmust is támogat. Nem szabad egyetlen megoldásnál leragadnunk, ha létezik több is az adott problémára. Az interneten megtalálható dokumentumok között vannak egész jó eredménnyel kecsegtetők – sajnos a legjobbnak tűnőnek is vannak hátrányai és ún. **artifact**jai (nem kívánt vizuális mellékhatásai). Ezeket a moduloktól eltérően kötelező – és általában elengedhetetlen – imodítani a grafikus motorhoz.

A *robosztusság* szintén fontos szempont. Logmentésekkel, debug technikák alkalmazásával, külső eszközök (*NVidia PerfHUD, PIX*) használatával ellenőrizhetjük a motor kétesen, lassan működő részeit. Kiszűrendők a számítási hibák, melyek artifactot vagy rossz működést okoznak, és a futásidejű hibák.

Továbbá kialakítandók kommunikációs lehetőségek. Erről már írtam a **2. 1. A játékmotor szerkezete** c. fejezetben, tovább nem részletezném.

3. 6. Belső funkciók – a grafikus motor rétegei és moduljai

Arról már volt szó, milyen általános elvárásaink vannak a grafikus motorral szemben. Azonban még nem fejtettem ki, hogy milyen feladatai vannak egy grafikus alrendszernek a megjelenítésen túl.

Ezen feladatok kiépítenek egy rétegzettséget, azaz egy vertikális struktúrát a grafikus motoron belül. Az alsóbb rétegek a videokártyához közelebbi funkciókat valósítják meg, a felsőbb rétegek már egy jóval absztraktabb szintet képviselnek.

Láthatjuk majd, hogyan válik ketté ez alapján az anyag és az anyagshader.

A felsorolás lentről-felfelé történő irányban halad.

3. 6. 1. Erőforrás-kezelés

Habár van külön *erőforrás-kezelő alrendszer*ünk, a grafikus motor olyan speciális erőforrásokat igényel, melyek kezeléséről azon belül kell gondoskodnunk. Ilyen erőforrások a textúrák és a meshleíró adatok (vertex- és indexpufferek).

Megjegyzés. Habár a Direct3D nem tartja erőforrásnak, én szeretném a shadereket is ide sorolni – kezelésük ugyanis hasonló.

Egy erőforrást menedzselni kell: kérésre létrehozni; Direct3D eszközvesztés esetén törölni, ha a videomemóriában létezik, majd visszaállítani; felszabadítani, ha már nincs rá szükség. Alkalmazhatunk referenciaszámlálást is, ekkor automatikus szemétgyűjtő mechanizmus építhető be az erőforrás-kezelőnkbe: amely erőforrásra már nincs több hivatkozás, azt megszüntetjük[19]. Természetesen ez okosításra szorul, prioritást és egyéb információkat adhatunk az erőforrásokhoz (például *„kézileg kell felszabadítani”-flag*).

Többféle módon létezhet erőforrás.

Helye szerint (*pool*) létezhet a videomemóriában és a rendszermemóriában. Ha kérünk rá menedzselést, akkor az API fogja legjobb belátása szerint másolgatni az erőforrást a két

memória között – olykor az AGP (*Accelerated Graphics Port*) bevonásával. Drivertől is kérhetünk menedzselést – ez azonban nem mindig támogatott.

Egy erőforrás lehet dinamikus vagy statikus használatú. Ezen kívül több használati jelzőt is adhatunk neki (például csak olvasható, mélységi-stencil, rendertarget, stb...). Ezeket nevezi a Direct3D *usage*-nak, vagyis használatnak.

Az alábbi táblázat segít eligazodni, hogy milyen erőforráshoz milyen helyet és használatot szokás rendelni.

Hely és használat	Leírás	Példa
Pool: `DEFAULT`. Usage: `0`, `RENDERTARGET`.	Videomemóriában hozzuk létre – eszközvesztés esetén figyelni kell rá! Nem zárolhatók.	Rendertargetek.
Pool: `DEFAULT`. Usage: `DYNAMIC`, `SOFTWAREROCESSING`.	Videomemóriabeli gyakran frissülő erőforrások.	CPU-n számolt részecskerendszerek, szoftveres vertex-feldolgozás.
Pool: `SYSTEMMEM`.	Rendszermemóriában tárolt erőforrások. Ezeket másolni kell explicit módon, ha a videokártyának szüksége van rájuk.	Ritkán változó, statikus erőforrások.
Pool: `SCRATCH`.	Csak és kizárólag a rendszermemóriában létezik az erőforrás – soha nem fogjuk másolni a videokártyának.	Számítások eredményét tárolhatjuk.
Pool: `MANAGED`.	A másolást, menedzselést a Direct3D-re bízzuk.	Bármilyen más erőforrás.

Amit még itt szeretnék megemlíteni. Menedzselésnél, erőforrás-feltöltésnél, explicit másolásnál a megfelelő memóriaterületek zárolódnak. Ezen zárolásnak többféle jelzője lehet és mindig a kívánt művelet és a zárolandó erőforrás helye és használata dönti el, mely jelzőkombinációk használhatók, és melyek a legcélravezetőbbek. Azonban vigyáznunk kell vele, egyes esetekben a zárolás szaggatást okozhat (*stall*)!

Továbbá a nem megfelelő menedzselés, másolás és frissítés (*updating*) is szaggatáshoz vezethet – ha egy textúra például a videomemória telítettsége miatt kilapozódik, majd újra bemásolódik frame-enként többször is (*thrashing*).

3. 6. 2. Egységes matematikai adatszerkezetek

Szükség van egységes matematikai adatszerkezetek és a hozzájuk tartozó műveletek kialakítására.

Ezek tulajdonképpen csomagoló osztályai a Direct3D és kiterjesztése által definiált adatszerkezeteknek, de vállalkozhatunk saját implementációra is. A leglényegesebb adatszerkezetek a vektor, a mátrix, a négy komponensű szín, a kvaternió és a sík.

Mivel ezekhez tömbös adatfeldolgozás esetén egymással jól párhuzamosítható műveletek tartoznak, lehetséges az ún. *batch-feldolgozás*. **SSE** (Intel), illetve **3dNow!** (AMD) optimalizálások használhatók fel.

Az SSE optimalizálást teszteltem mátrix és vektor osztályokra **(**ld. az **5. 1. Nem optimalizált, SSE-optimalizált és D3DX függvények** c. függeléket). Az eredménye az lett, hogy az adatszerkezetek Direct3D-t csomagoló műveletei és a nem SSE-optimalizált műveletei közel azonos idő alatt végeztek, míg az SSE-optimalizált műveletei jóval lassabbnak bizonyultak. Ennek oka az, hogy a fordító optimálisabb kódot tudott fordítani, mint az SSE utasítások.

A D3DX elfedésének használatát javaslom! Ugyanis a Direct3D driverei az aktuális rendszerkonfigurációnak megfelelő optimális módon működnek. És, ha mi már külön nem foglalkozunk az ilyen alacsony szintű optimalizálással, könnyen érthető marad a kód is.

Habár ez a réteg elhagyható lenne, úgy gondoltam, sokkal célszerűbb a Direct3D által nyújtott egyszerű, alapvető adatszerkezeteket és függvényeket becsomagolni ún. *wrapper osztály*okba. Ennek csak az a jelentősége, hogy a fentebbi rétegekben már elhagyható a Direct3D közvetlen bevonása a számításokba.

Nyilván nem fontos erre törekedni, de fejlesztés szempontjából van némi haszna. Például az, hogy ezt a réteget felkészíthetjük a fizikai alrendszer felől érkező adatok konvertálására. Ekkor a felsőbb rétegekben ez a konvertáló mechanizmus már automatikus lesz.

Továbbá azt is fontos megemlíteni, hogy – wrapper osztály lévén – a függvények néhány kivételtől eltekintve mind *inline* függvények. Ez azt jelenti, hogy a fordító a hívásuk helyére közvetlenül a törzsüket illeszti be, feltéve, hogy megteheti, vagyis a függvényhívásokkor fellépő ugrásra és veremkezelési feladatokra nincs szükség[14]. Ez sebességbeli javulást eredményez.

Megjegyzés. Az inline függvények használatát egyébként érdemes más szinteken, más osztályokban is alkalmazni a kritikusabb függvényeknél.

3. 6. 3. Adatfeltöltés a videokártya felé

Szűk keresztmetszete lehet a grafikus motornak az adatfeltöltés a videokártya felé, a shaderváltás illetve a grafikus csővezeték állapotainak (*state*) váltása. Ezen adatmozgatásokra, beállításokra kellene egy jó stratégiát találni, mely a fölösleges feltöltésektől és váltásoktól megkímél minket, ugyanakkor egy könnyen kezelhető, átlátható felületet biztosít fölfelé.

Meg kell oldani a shaderkezeléshez kapcsolódó egyéb problémákat is: megfelelő shader fordítása, mely a grafikus motor beállításaihoz és a hardverhez igazodik.

Sajnos elbonyolítható kellően a shaderkezelés szintje, de mindenképpen szükség van rá, hiszen a videokártya és a grafikus motor közti adattovábbítás közvetítője.

3. 6. 4. Anyagok és anyagshaderek kezelése

A Direct3D által nyújtott anyag (*material*) lehetőség helyett érdemesebb egy komolyabb, több lehetőséget és paraméterezhetőséget nyújtó anyagtípust létrehozni. Egy ilyen anyag tudni fogja ráadásul azt is, hogy egy bizonyos szituációban melyik shader kell az ő megjelenítéséhez és milyen paramétereket vár.

Az anyagokat a grafikusok szerkeszthetik és állíthatják be intuitív módon. Esetleg érdemes létrehozni egy anyagszerkesztőt, mellyel interaktívan paraméterezhetik fel az egyes anyagokat.

Lehetséges az anyaghoz kapcsolódó shaderek felülbírálása. Ez akkor lehet fontos, amikor alacsonyabb megjelenítési beállítást választunk a videokártya lehetőségeihez mérten,

és nincs lehetőségünk a legtöbb részletet kiszámító shadert használni. Ilyenkor érdemesebb egy „butább” verziót használni. De ide sorolható az az eset is, amikor például mélységi rajzolást (*depth rendering*) végzünk – ilyenkor a mélységi értéket kiíró pixelshader egységes minden anyaghoz.

Látható, hogy az anyaghoz több fizikai shader is tartozhat, és beállítástól függően válogathatunk ezek között. Ugyanakkor amikor anyagról beszélünk, már nem szeretnénk a konkrét megvalósításba belemerülni, csak a paraméterezéssel akarunk foglalkozni. Az alsóbb funkciókat az ún. anyagshader (*materialshader*) fogja kezelni.

Az anyag és az anyagshader közti határ lesz az elválasztóvonal a shaderfüggetlen és a shaderfüggő elemek között.

3. 6. 5. Meshek kezelése

A meshek vagy modellek lesznek a háromdimenziós világunk építőkövei. Vertex- és indexpufferekből állnak, anyagonként eggyel-eggyel. Optimalizálás szempontjából megkövetelhető lenne egy bizonyos elemszám-korlátozás ezen pufferekre, azonban ettől itt most eltekintek. (Bővebben ld. a **3. 9. 5. Geometriai optimalizáló modulok** c. fejezetet.)

Egy mesh több darabból állhat. Egy darabja tartalmaz egy anyagot, egy vertex- és egy indexpuffert. Amikor a meshhez kérés érkezik valamelyik darabjának rajzolásához, beállítja a darabhoz tartozó anyagot, saját paramétereit feltölti, és a Direct3D primitívrajzoló mechanizmusát felhasználva kerül a mesh adott darabja a háttérpufferbe vagy egy rendertargetbe.

Minden egyes mesh tartalmaz továbbá egy referenciát egy csontvázobjektumra is – feltéve, hogy csontozottan animálható és tartalmazza a megfelelő információkat erre vonatkozóan.

3. 6. 6. Csontvázak és animációk kezelése

Minden egyes meshhez tartozhat egy csontváz leírás. Ez csontozott mesheknél (*skinned mesh*) jelenik meg csupán, ahol az egyes vertexek végleges pozícióját az objektumra vonatkozó

világtranszformációs mátrixon (*world transformation matrix*) kívül a csontadatok is meghatározzák.

Egy csontváz tulajdonképpen egy mátrixpaletta, melyben minden egyes mátrix egy adott csont transzformációját tárolja. Természetesen mátrix helyett másféle módon is elraktározhatjuk ezen információkat – pl. a csont forgáspontját (*pivot point*) és az orientációját (*yaw-pitch-roll* vagy *kvaternió*) tároljuk.

Amikor egy vertex pozícióját szeretnénk kiszámolni, a hozzárendelt csontok (maximálisan négy csont) mátrixát a vertexben tárolt mátrixtömb-indexek (*blending indices*) alapján a vertexben tárolt értékekkel (*blending weights*) súlyozva véve kapunk egy újabb mátrixot, ezzel transzformáljuk a vertex pozícióját, majd még ezután a világtranszformációs mátrixszal is beszorozzuk.

Ugyanez a műveletsor érvényes a vertexekbeli tangensterek vektoraira is.

Hardverlimitáció miatt – konkrétan 1.1-es vertexshader esetén – általában 20 csontmátrixnál többet nem adunk át a grafikus kártyának. Ugyanis a vertexshader-regiszterek száma maximálisan 96. Egy mátrix 4 regisztert foglal le. Így látható, hogy 20 * 4 = 80 regisztert foglalnak le csak a csontadatok. Ezt áthidalandó a nagyobb csontvázakat több részre osztják fel úgy, hogy az egyes részek 20 vagy annál kevesebb csontot tartalmazzanak. Nyilván ekkor a mesh is több darabra esik szét, hiszen megváltoznak a vertexekben tárolt csontindexek. Ezt a felosztást végezheti a grafikus motor is – természetesen nem valós időben,– vagy egy külső alkalmazás is, mint egy *exportáló plugin*.

Ha a shader engedi, több mátrixot is feltölthetünk, vagy 4x3-as mátrixokat alkalmazhatunk, ahol a negyedik oszlopot automatikusan (0, 0, 0, 1)-nek vesszük. Továbbá használhatunk kvaterniókat is erre a célra.

Az egyes animációs lépésekhez (*keyframe*) létezik tehát egy-egy ilyen mátrixpalettánk. Ezek között interpolálni is lehet és mindig a megfelelő mátrixlistát – vagy annak egy részét – továbbítjuk a vertexshader felé. Kvaterniós megadás esetén ajánlott a gömbös interpoláció[64] (*spherical interpolation*), de kellően sok keyframe esetén kielégítő eredményt szolgáltat a lineáris interpoláció is – és gyorsabb is.

Továbbá elég letárolni keyframe-enként csak azokat a mátrixokat, melyek megváltoztak az előző vagy egy adott keyframe-beli értékükhöz (*referenciamátrix*) képest.

Lehetőség van másodlagos vagy kevert animáció lejátszására is (például sétálás közben bólogat a karakter feje).

Megjegyzés. Csontozott animáció esetén szoftveresen is transzformálhatók a vertexek. Ekkor a végleges vertexpozíciókat tároló puffereket ennek megfelelően kell beállítani: videomemóriában létező, dinamikus erőforrásként, melynek zárolása írásra *discard* jellegű, vagyis törli a korábbi tartalmat (ld. **3. 6. 1. Erőforrás-kezelés** c. fejezetet).

3. 6. 7. Objektumok kezelése

A megjelenő karaktereket, tárgyakat, a pályát, a környezet elemeit, stb... nevezzük objektumoknak. Minden objektumhoz tartoznak meshek, melyeket egymástól részletezettségi szintjeik (*level of detail, LOD*) különítenek el. A LOD-ot választhatjuk megjelenítés-minőségi beállításként vagy a részletező modul (LOD-rendszer) kalkulálhatja ki, attól függően, hogy az objektum a képernyő mekkora részét tölti ki vagy milyen távol van a kamerától.

Az objektumnak tudnia kell saját transzformációs mátrixát, animációinak leírását, az aktuális animációjának fázisát, befoglaló térfogatának méreteit (*bounding box*, illetve ha az axisokhoz igazított: *axis-aligned bounding box - AABB*). Ezen kívül természetesen a LOD információkat és a kapcsolódó meshеket is tartalmazza.

Statikus (nem csontozott) objektumok esetén megadható egy statikus térfogattextúra is (*volume texture*), mellyel az objektum ambiens lesötétítő hatása (*ambient occlusion*) definiálható. Ennek lényege, hogy a jelenetbe helyezett objektum a körülötte lévő objektumokat lesötétíti, ha elég közel van hozzájuk.

További jelzők is kellenek, melyek elárulják, az objektum tartalmaz-e áttetsző részeket, vet-e árnyékot, animált-e, fénysugárzó-e (*emitter*), önmegvilágító-e (*self-illuminating*), stb... Ezek szükségessége egyrészt koncepciótól függ, másrészt rajzolási sorokba való pakoláshoz kell.

3. 6. 8. A fények és a megvilágítás kezelése

Az árnyékszámító megoldásoknál már tapasztaltuk, hogy többféle járható út lehetséges. A megvilágítási rendszer szintén egy ilyen pontja a grafikus motornak – fő kérdés tehát, hogy az adott koncepcióhoz és tervezethez milyen megvilágítási modell illik.

Ha egyetlen fényforrásunk van, akkor nem sok probléma adódhat. Viszont több fényforrás esetén már erős technológia bevetésére van szükség. Ez a technológia lehet az ún.

deferred shading, ahol előbb mélységi, normál és egyéb információkat tartalmazó képet készítünk a jelenetről, majd a fényforrások hatókörét – mint objektumot – megjelenítve számolunk minden pixelbe intenzitásértéket. Ez a technika nem egyszerű, és nagyszámú, nagy hatókörű fényforrás esetén sebességvesztést is okozhat (*performance hit*).

Másik megoldás lehet, hogy objektumonként csak a legközelebbi adott számú fényforrást gyűjtjük össze és az objektum kirajzolásakor ezeket vesszük számításba. Ennek előnye, hogy a maximálisan összegyűjthető fényforrások számával paraméterezhető a shader, viszont okozhat egyéb artifactokat (fantomfények, hiányzó fények, stb…).

Statikus megvilágítás esetén, vagyis amikor a fényforrások nem mozognak, intenzitásuk folyamatos és feladjuk a dinamikus árnyékokat, alkalmazhatók a fénytérképek (*lightmap*). Ezeknek többféle változatuk létezik és nemcsak színinformáció tárolása lehetséges. Lényege, hogy előre leszámolunk textúrába megvilágítási információt az egyes felületekre (tetszőleges modell alapján), majd ezen textúrát valós időben a felületekre alkalmazzuk valamilyen művelettel.

Természetesen a megvilágítási technikák tárháza kellően nagy, így minden motorba megtalálhatjuk a játék számára megfelelőt. Több technika ötvözése is lehetséges, ezáltal a statikus és a dinamikus megvilágítás keverhető, esetleg részletezhető.

A fényforrásokról még nem beszéltünk. A motor támogathat többféle típusú fényforrást is – körsugárzó (*omni light*), vetített (*directional light*), irányított (*spot light*), területi (*area light*) és egyéb fényforrástípusokat is.

Típuson kívül pozíciót, irányt, színt, intenzitás paramétereket, vetítési nyílásszöget, stb... is rendelhetünk hozzájuk.

A fényforrásokat célszerű listába rendezni. Kell továbbá olyan algoritmus, mely egy térbeli ponthoz vagy befoglaló dobozhoz megtalálja az ahhoz legközelebb eső adott számú fényforrást ebből a listából.

3. 6. 9. Kamerák kezelése

A kamerarendszert nem érdemes elbonyolítani, általában tömör, kevés szolgáltatást nyújtó osztály elegendő hozzá. Egy kamerának olyan alaptulajdonságok adhatók meg, mint a pozíció, nézési irány, felfele mutató vektor iránya (ezek alkotják a nézeti mátrixot – *view matrix*), a közeli és a távoli vágósík távolsága a kamerától (*near plane, far plane*), a kamera

nyílásszöge (*field of view*), a képarány (*aspect ratio*) és az, hogy bal- vagy jobbkezes koordinátarendszert használunk (ezek a vetítési mátrixot adják meg – *projection matrix*).

Megjegyzés. Fontos a két vágósík megfelelő beállítása, ugyanis az értékeik befolyásolják a mélységi puffer felbontását.

Ezeken az adatokon kívül létrehozhatunk egy kameralistát is, melyben többféle kamerát tárolhatunk más-más paraméterekkel; természetesen egy rajzolásnál mindig csak az egyik kamera aktív a listából.

Esetleg görbéket is definiálhatunk, melyekhez kamerát köthetünk.

Kameramozgási tulajdonságokat is definiálhatunk: rögzített pozíció, rögzített célpont, mozgási sebesség, görbén való mozgás, objektumkövető kamera, fizikai objektumhoz kapcsolt kamera, stb…

A lényeg, amit elvárunk ettől a modultól az az, hogy lekérdezhető legyen az aktív kamera nézeti és projekciós mátrixa, esetleg az aktív ablak (*viewport*) adatai, melybe az aktív kamera szemszögéből rajzolunk.

3. 6. 10. Rajzolási sorok kezelése

A rajzolandó objektumokat tulajdonságaik alapján **rajzolási** vagy **rendersor**okba (*render queue*) pakolja be ez az egység. Ilyen rendersorok például az árnyékrajzolás, a tükröződésrajzolás, a *deferred* rajzolás, a normális rajzolás és az átlátszósági rajzolás. Egy objektum több sorba is bekerülhet. Egy ilyen sor saját működését határozza meg, vagyis definiálhat mátrix, anyag és egyéb felülbírálásokat, azt, hogy milyen célterületre (*rendertarget*) rajzolja az objektumait és egyéb paramétereket, algoritmusokat. Tulajdonképpen egy rendersort egy gépként lehet elképzelni, ahol egyik végén beöntjük az objektumokat, változtathatunk néhány beállítást, aztán a másik végén kijönnek az adatok (tipikusan egy leszámolt kép), melyeket majd egy másik gépnek paraméterként átadhatunk.

Mivel egyetlen videokártyával rendelkezünk, ezért ezen rendersorok működése szekvenciális, hiszen mindig csak az egyik használhatja a videokártyát, ráadásul némelyiknek várnia kell a másik eredményére. Vagyis lényeges a rajzolási sorok sorrendje.

A nem GPU-algoritmusaik azonban kiemelhetők ebből a szekvenciális feldolgozásból. Például amíg az árnyékszámítás fut, addig az átlátszósági sorban elindíthatjuk a távolság alapú rendezést, és mire végeztünk az árnyéktérképpel, az átlátszó objektumainkból már rendezett poligonlistánk lesz.

3. 6. 11. A háromdimenziós világ kezelése – vezérlési szint

A háromdimenziós világ magát a megjelenítendő jelenet elemeit gyűjti magába. A megjelenítendő objektumokon kívül a kamerák, fények és a rajzolási sorok listáit is tartalmazza.

Főbb feladata a rajzolási sorok végrehajtatása, az objektumok rendezése, vágása, eldobása, LOD információk beállítása; a fizikai, a mesterséges intelligencia, az input alrendszerektől fogadott adatok feldolgozása, továbbítása a megfelelő célelemek felé, üzenetek fogadása és kezelése, a jelenet betöltése, feldolgozása, megjelenítése.

Látható, hogy ez a szint lesz a legfelső rétege a grafikus motornak. A motoron belüli párhuzamosítást és a más alrendszerekkel való kommunikációt ez a réteg oldhatja meg.

3. 6. 12. Különálló modulok, bővítmények

Léteznek olyan modulok is, melyek nem feltétlenül szükségesek a megjelenítéshez, csupán látványfokozó vagy optimalizáló szerepük van. Ezek a modulok azok, melyek a grafikus motort modern grafikus motorrá tehetik.

Optimalizáló modul a kitakarásvizsgáló modul (*occlusion culling engine*), a LOD számító modul, a portál technikát felhasználó modul beltéri játékok esetén, vagy az imposztor technikát felhasználó modul nagy látótávolságú játékok esetén.

Látványfokozó modul a gyors növényzetrajzoló modul (például a *SpeedTree engine*[65]), a posteffekt alrendszer, a részecskerendszerek, a csapadékszimulációs motorok, egyéb speciális trükköket megvalósító modulok (fénytörés a lencsén – *lens flares*).

Ezeknél a legtöbb problémát a már létező grafikus motorhoz való idomítás okozza. Ugyan önállóan (egy kisebb szemléltető alkalmazásban) képesek működni (*prototípus*ok), de

nagyon sok tényező van, melyet módosítanunk kell ahhoz, hogy igazán beépüljenek a motorba.

Gondoljunk csak arra, hogy megvásároltunk egy növényzetrajzoló modult, mely imposztor és részletező technikákat valósít meg, saját shaderbetöltő egysége van, több API-t is támogat. Viszont a megvilágítási modellünket, az árnyékszámítási módszerünket, az anyagfelülbírálási lehetőségeket nem ismeri – ezt mind bele kell programoznunk, mert ezek nélkül nem fog a növényzet látványra a jelenetbe illeni.

Lényeges az ilyen modulokhoz kapcsolódási pontok kialakítása a grafikus motorban. Ezekről részletesebben a **3. 11. Modulok kapcsolása** c. fejezetben értekezem.

3. 7. Adatvezéreltség

3. 7. 1. Miért fontos?

Az adatvezéreltség játékfejlesztés szempontjából a játék tartalmának cserélhetőségét, működésének nem csak programozók általi befolyásolását, a lehető legtöbb elem paraméterezhetőségét jelenti.

Vagyis elkerülendő a tartalom és a paraméterek „bedrótozása” a játékmotorba. Az a cél, hogy tetszőleges egyén (grafikus, designer, gameplay-programozó) tudja intuitív – és lehetőleg interaktív – módon változtatni az egyes játékelemeket.

Ebből a szemszögből tekintve a játékmotor csupán egy olyan automata, mely az adatok beolvasása és feldolgozása alapján hajt végre bizonyos tevékenységeket. Ilyen bemenő adat lehet bármi az egyszerű algoritmusparaméterektől a játékmenetet leíró scriptekig.

Túlzásokba esni sem szabad, hiszen egy nagyon általános motor a tökéletes motor csapdájába fog esni: minden megvalósítható vele, de az adatfeldolgozás bonyolult folyamat, és a komplexebb algoritmusok hiánya miatt (illetve azok scriptnyelv-szerű változata miatt) sebességvesztés, nehezen javítható artifactok is felléphetnek.

Az adatvezérelt filozófia fontosságának másik oka a szabad tartalommódosításon kívül az adatok ellenőrzése. Bedrótozott adatokat menet közben nehézkesen lehet módosítani, ellenőrizni (C++-ban például függvényeken belül `static` tárolási deklarációval megoldható a futás közbeni értékmódosítás), ráadásul nem biztos, hogyha egy adat hibás, könnyen megtaláljuk, melyik.

Az adatokat viszont már beolvasásnál tudjuk ellenőrizni, ha az adatvezéreltség elvét követjük, ráadásul az interaktív adatmódosítás nem igényel programozási ismereteket (ld. fentebb).

3. 7. 2. Megvalósítás

Az adatvezéreltségnek az *anti-design mintát*[12] követő ún. **magic numbers** (**varázsszámok**) kiiktatása lenne az egyik legfontosabb feladata.

A varázskonstansokat fejlesztés közben hozta létre a programozó, aki ezen értekek mellett tesztelte algoritmusait. Ha ezek a számok, sztringliterálok nem tartoznak szorosan az algoritmus számításába, vagyis a számítás paraméterei, akkor kötelező ezeket a varázskonstansokat megszüntetni, és helyettük a megfelelő paramétereket kivezetni a programból.

Előfordulhat, hogy több helyen is ugyanolyan jelentéssel alkalmaztunk egy varázsszámot. Mennyivel kényelmesebb lenne egy fájlban állítani ennek az értékét, mint visszakeresni az összes deklarációt, definíciót a program szövegében és minden értékadás helyén megváltoztatni azt!

A másik fontos dolog a hasonló feladatot ellátó, hasonló szerepkörű algoritmusok összegyűjtése, rendszerezése, egységesítése. Ezeknek nem csak paramétereik vezethetők ki, de a feldolgozott adatok alapján lehetne közülük kiválasztani valamelyik végrehajtását. Ezzel tehát megoldható lenne, hogy az adat vezérelje a játék és egyéb részeinek működését.

Ide sorolhatók a script technikák. A könnyen megtanulható, magas szintű nyelven megírt kódot (scriptet) a program beolvassa, a szintaktikus – esetleg szemantikus – hibákat kiszűri benne, feldolgozza a benne foglalt utasításokat. Ilyen nyelvek például a *Python*, *Lua*, de *Java* vagy akár *C#* nyelvű scripteket is feldolgozhatunk. Természetesen ezeknél egyszerűbb nyelvek is kialakíthatók játékunk számára.

Összetettebb nyelvek esetén létezik egy futtató környezet, például Java esetén a *Java Virtual Machine* (JVM), ahol a nyelvhez tartozó elemeket a futtató környezet kezeli, a játékmotorhoz tartozó függvényeknek pedig léteznek *natív* megfelelői a motorban. Egyszerűbb nyelveknél elegendő lehet az interpreteres végrehajtás is.

Az adatvezéreltség speciális megjelenési formája az alrendszerek üzenetváltása. Egy-egy ilyen üzenet utasításokat, de akár rövidebb scripteket is tartalmazhat. Ekkor az alrendszereknek maguknak kell megoldaniuk ezen üzenetek feldolgozását.

3. 8. A grafikus motor részletesebb mechanizmusai

3. 8. 1. Erőforrás-menedzsment

Az erőforrások kezelése nem nehéz feladat, csupán meg kell oldani az erőforrások betöltését, létrehozását, törlését. Betöltéshez a fájlkezelő réteg szolgáltatásait is használni fogjuk. Továbbá a Direct3D megfelelő függvényeit is meghívjuk a megfelelő módon paraméterezve.

Textúránál fontos figyelni, hogy minden textúrát csupán egyszer töltsünk be. A már betöltött textúrákat egy listán vezetjük. Ha olyan textúra betöltésére érkezik igény, amely ezen a listán szerepel, akkor a listán szereplő textúra címét adjuk vissza az igénylőnek.

Felszabadításnál figyeljük, hányszor kellett volna betöltenünk a textúrát, és ebből hányszor kellett volna már felszabadítani. Ha a két érték egyenlő, a textúrát felszabadítjuk. Az alkalmazás bezárásakor a listán található textúrákat töröljük – kérésre szintén felszabadítható a teljes lista.

Rendertargetek létrehozását kérésre támogatjuk. A rendertarget méretét megadhatjuk konkrétan vagy az aktuális felbontással arányosan. Amikor eszközt vesztünk (*device lost*), a rendertargetek automatikusan újragenerálódnak. A textúrákkal ellentétben itt nem memóriacímmel hivatkozhatunk a textúrára, hanem egy azonosító egésszel. Ennek oka, hogy újragenerálásnál megváltozik a textúra címe, hiszen újat hozunk létre!

A shaderek betöltését is ez a réteg végzi. Megadott nevű, belépési pontú (*entrypoint*), fordítású (*target*), makrókkal ellátott shaderek fordítása lehetséges. Nem fordítja kétszer ugyanazt a shadert, már meglévő címét adja vissza.

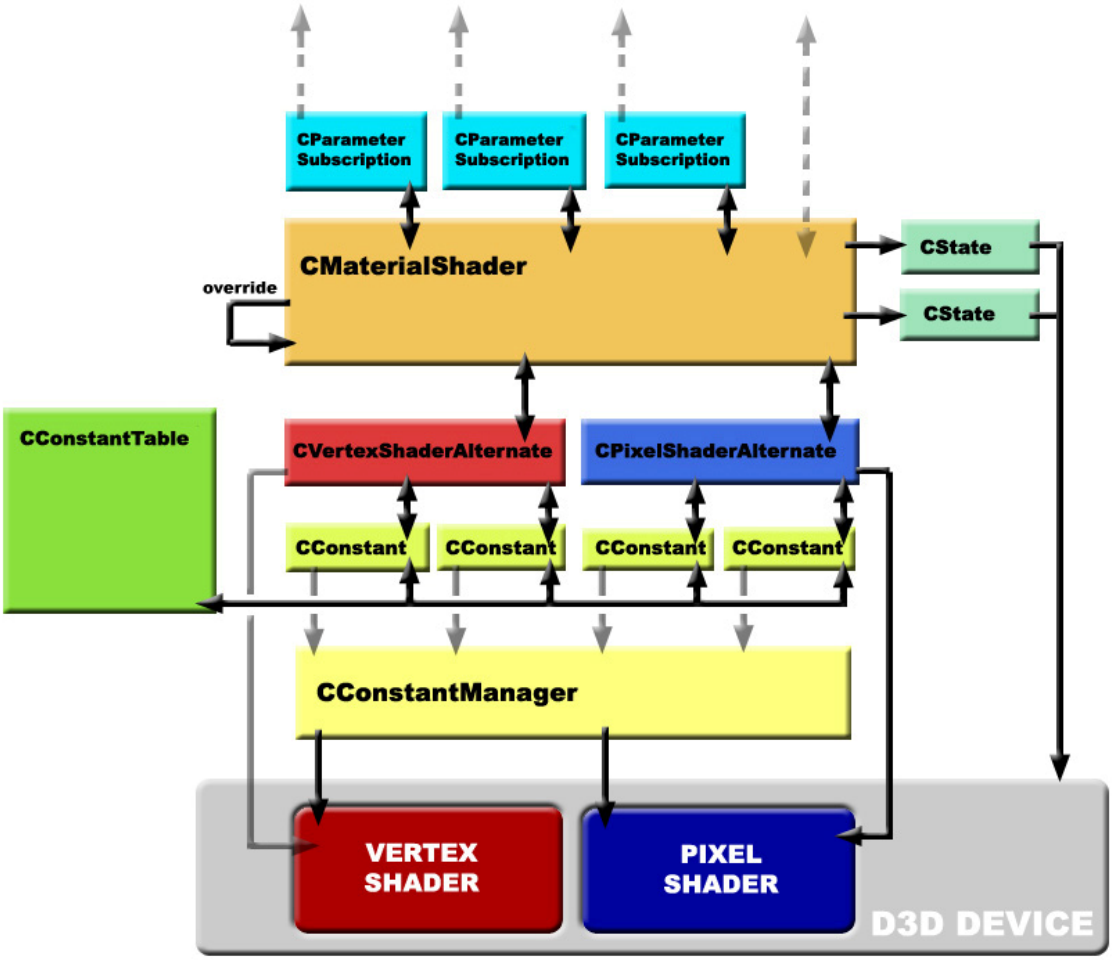
Vertex- és indexpufferek létrehozása. A beolvasott meshadatoknak megfelelő méretű tömböket allokálunk a kívánt *vertexformátum*mal. Ezeket egy alacsony szintű meshdarabba fogjuk csomagolni. Lásd az **5. 3. A magas és az alacsony szintek közti határ** c. függeléket!

3. 8. 2. A shaderkezelés lelke – az anyagshader

Lehetőségünk van a shaderek beállításoktól függő makrózott fordítására, de miért szükséges ez? Miért nem elég minden anyaghoz egyetlen shader hozzárendelése?

Az belátható, hogy egy anyagtípust mindig ugyanazzal a shaderrel szeretnénk megjeleníteni. De miért kell mégis több shadert hozzárendelnünk?

Grafikus opcióktól és a videokártya által támogatott shaderverzióktól kell függővé tenni a shaderek fordítását annak érdekében, hogy minél több szabályozhatóságot biztosíthassunk a felhasználó számára, és gyengébb hardveren is működjenek valamilyen szinten a shaderek.



3. ábra. Az anyagshader és a shaderváltozatok kapcsolatai. Látható, hogy a magasabb szintek csupán az anyagshaderen keresztül tudnak kommunikálni a shaderekkel.

Ezt a feladatot látja el az anyagshader osztálya, a `CMaterialShader`. Az egyes shaderlehetőségeket a `CBaseShaderAlternate` osztályból származó két osztály, a `CVertexShaderAlternate` és a `CPixelShaderAlternate` osztály példányai realizálják. Az anyagshader mindig az aktuális beállításoknak megfelelő shadereket fogja kiválasztani, melyeket egy-egy jól definiált listáról választ ki.

A kiválasztott shadereket szükség szerint lefordítja, majd kigyűjti a konstansokat belőlük. Ezek címét bemásolja egy globális konstanstáblába (`CConstantTable` singleton példány), amely regisztrációs feladatokat old meg. Továbbá a konstansokból ún. paraméterelőjegyzéseket is készít (ld. **3. 8. 3. A shaderek globális paraméterezhetőségének kérdése** c. fejezetben).

Az anyagshaderben megadhatók állapotváltások is, melyek segítségével a csővezeték egyes mechanizmusait módosíthatjuk.

Amikor az anyagshadert aktivizáljuk, a megfelelő shadereket teszi aktívvá (`SetVertexShader`, `SetPixelShader`), összegyűjti konstansaik aktuális értékeit (ld. **3. 8. 3. A shaderek globális paraméterezhetőségének kérdése** c. fejezetben), ezeket optimalizált módon feltölti a kártyának a `CConstantManager` singleton példányon keresztül (ld. **3. 9. 2. Konstansfeltöltési stratégia** c. fejezetben), és a rajzolási és mintavétezelési állapotokat is beállítja (`SetRenderState` illetve `SetSamplerState`).

Lehetőség van anyag-felülbírálásokra (*material override*). Ez azt jelenti, hogy például a mélységi rajzolást egy másik anyagshaderrel kell elvégeztetni.

3. 8. 3. A shaderek globális paraméterezhetőségének kérdése

3. 8. 3. 1. Konstansok és textúrák

A shaderek az alkalmazástól nemcsak geometriai adatokat kapnak, melyeken transzformációkat és egyéb számításokat végrehajtva előáll a megjelenítendő pixelsorozat, hanem olyan adatokat is, melyek ezen transzformációkat és számításokat, tehát a shaderek működését befolyásolják, paraméterezik.

Ezen adatokat két típusba sorolhatjuk – konstansok és textúrák (vagy *mintavételezők*).

3. 8. 3. 2. Konstansok

A konstansok HLSL nyelven többféle típussal rendelkezhetnek (float, int, bool), azonban két dolgot érdemes megjegyeznünk velük kapcsolatban.

HLSL-ben célszerű mindig float típusú konstansokat használni., ugyanis gyakorlatilag float típusú adatok feldolgozására találták ki a grafikus kártya regisztereit.

Továbbá négyeseket tudunk feltölteni. Vagyis ha egyetlen float értéket szeretnénk átadni a shadernek, akkor ezt úgy tehetjük meg, hogy egy négyelemű floattömbbe bemásoljuk valamelyik elembe (akár mindbe) ezt az értéket, és ezt a tömböt adjuk át.

A shaderek egymástól független regisztermezőkkel rendelkeznek. A konstansregiszter-mező egy elemi egysége (*regiszter* vagy *slot*) 4 float érték tárolására elegendő. A shader konkrét slotsorozatot (regisztersorozatot) rendel minden konstansához. Egy ilyen slotsorozat folytonos, így elég csak a kezdő slot sorszámát megadni (0-tól indul a sorszámozás). HLSL nyelven konkrétan megadhatjuk a konstansainknak ezt a kezdő slot indexet. Ha nem adjuk meg, fordításkor a fordító olyan sorrendben kezdi kiosztani a sorszámokat, ahogyan a lefordított shader felhasználja őket. A konstansok nevei alapján ezt a sorszámozást – és további hasznos adatokat is – le tudunk kérdezni az alkalmazásban.

A videokártya és a shadermodell meghatározza, mekkora méretűek lehetnek az egyes regisztermezők. A pixelshader konstansregiszter-mezője jóval kisebb szokott lenni mint a vertexshaderé. A legkisebb közös metszet: a vertexshader 96 konstansregiszterrel, a pixelshader 8-cal rendelkezik.

Nemcsak konstansregiszter létezik – input-, output-, cím-, szín-, ciklusszámláló, ideiglenes, stb... regiszterek is vannak. Ezek azonban csak a shaderek assembly kódjában jelennek meg, így itt nem foglalkozom velük.

Az általam használt konstansfeltöltő függvény:

```
SetVertexShaderConstantF(first_slot, float_array, float4_count)
```

illetve

```
SetPixelShaderConstantF(first_slot, float_array, float4_count).
```

Megjegyzés. A konstans elnevezés onnan ered, hogy a shader ezeket – mint a bemenetére írt adatokat – konstansokként kezeli, értékeiket nem tudja megváltoztatni, rajtuk keresztül nem tud visszajelezni az alkalmazás felé.

3. 8. 3. 3. Mintavételezők

A shaderek fejlődésének köszönhetően már nemcsak 4 mintavételezési szakaszt használhatunk, és a mintavételezési szakaszok és a textúrakoordináták már nincsenek

összekapcsolva. Azonban ha 1.4-es verzió alatti pixelshadert írunk, vigyázni kell, mert a `TEXCOORDx` koordinátával csak az x. mintavételezőből olvashatunk be színt ($x \geq 0$)!

A mintavételezési beállításokat az egyes textúrákhoz a shaderben is közölhetjük (ez a közlés csak HLSL nyelven lehetséges és csak akkor használható, ha effektként olvassuk be a shadert), egyébként a mintavételezési beállításokat az alkalmazás végzi el:

```
SetSamplerState(stage, sampler_option, value),
```

a shader pedig egy mintavételezőt (*sampler*) vár. A mintavételezőket hasonlóképpen töltjük fel mint a konstansokat:

```
SetTexture(stage, texture).
```

3. 8. 3. 4. Paraméterezési stratégia keresése

Sajnos a paraméterezés egy többszintű dolog. A világ, az objektumok, a meshek és az anyagok mind hordoznak olyan információt, amelyet a shader felhasznál. Szerencsére kizárható az a lehetőség, hogy ugyanarra a dologra vonatkozó információ – mint paraméter több szinten is megjelenjen; egy szinten viszont szerepelhet többször is.

Például a világ tudja, hol van a kamera, mi a `ViewProjection` mátrix, milyen a napfény iránya, és tudomása van az árnyéktextúráról is. Viszont a shaderekhez nem szabad hozzáférnie!

A shaderekhez csak az anyag férhet hozzá, így az anyagban kellene összegyűjteni a világ paramétereit. Nem lenne okos dolog, ha a világ a paramétereit bemásolgatná objektumaiba, azok a mesheikbe, azok az anyagaikba, aztán az anyagok töltenék fel a shaderekbe ezen értékeket.

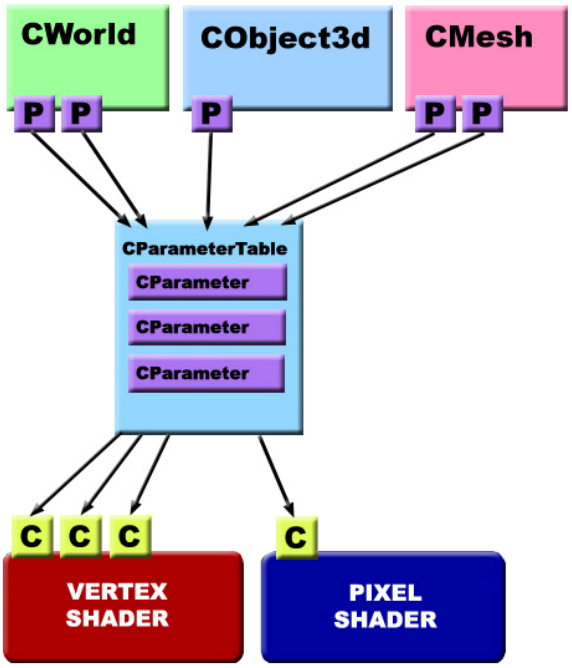
Vagy esetleg a shadernek szüksége van egy vetítési mátrixra, ezért az anyaghoz fordul, az anyag a meshhez, a mesh az objektumhoz, az objektum a világhoz, aztán az információt visszaküldik ugyanezen a láncon a shaderhez.

Ez a megközelítés nem szép, és mi történne, ha egy újabb szintet kellene építeni a hierarchiába? Minden szinten meg kellene valósítani lefelé másolgatásokat, illetve felfelé mozgó paraméterigényeket.

3. 8. 3. 5. Paramétertábla

Ehelyett egy paramétertáblát fogunk használni, amelyet a shaderen kívül más nem láthat.

Biztosítunk a felsőbb rétegek számára egy paraméter típusú objektumot. Ilyennel bármelyik osztály tetszőleges számban rendelkezhet. Egy paramétert a neve fog azonosítani, viszont ha több ugyanolyan nevű paraméterünk van, azok ugyanazt a paraméterbejegyzést fogják módosítani a táblában. A paramétertáblában ezért minden paraméternév csak egyszer szerepel. (A neveket a gyors elérés céljából indexekre oldjuk fel.)



4. ábra. Paraméterek és konstansok.

Amikor a shader megkapja a videokártyát rajzoláshoz, akkor konstansainak értékét a paramétertábla aktuális értékei szerint állítja be.

Ez úgy is elképzelhető, mintha tetszőleges példány előfizethetne néhány paraméterre, és ha változtatja a paraméterei értékét, akkor a shader tudni fog ezekről a módosításokról.

Ezzel a módszerrel az is megvalósul, hogy teljes rajzolásonként a világot leíró paraméterek csak egyszer kerüljenek kiszámításra és regisztrálásra a paramétertáblában, vagyis amikor a világot frissítjük, akkor elég a paramétereibe az újonnan kiszámolt értékeket beleírni, a többi dologgal már nem kell foglalkoznunk ezen a szinten.

Röviden szólva egyszerűvé és kényelmessé válik a felsőbb rétegek számára az információközlés a shader felé.

Munkát spórolhatunk meg, ha egy okos megoldás segítségével jelezni tudjuk a felsőbb rétegek felé a paramétereken keresztül, hogy szükség van-e egy adott paraméter számítására – elképzelhető, hogy a `ViewProjectionInverseTranspose` mátrixot például nem használja egyetlen használt shader sem, ezért a kiszámítása felesleges.

Ezt később látni fogjuk, hogyan oldható meg.

3. 8. 3. 6. A shader szintje – összekapcsolás a paramétertáblával

Már beszéltünk a felsőbb szinteken a paraméterezés kérdésének megoldásáról és van egy rejtett paramétertáblánk is, azonban még nem esett szó arról, hogy a shader mit kezdjen ezzel az adathalmazzal.

Amikor egy shadert feltöltünk a videokártyának, a paramétertáblában a shadert közvetlenül használó anyag, illetve a shadert közvetetten használó mesh, objektum és világ aktuális paraméterértékei kell, hogy legyenek.

A shader konstansainak tudnia kell, hogy melyik paraméterbejegyzést figyeljék; itt az összepárosítás megint a nevek alapján történik. A neveket a gyors elérés céljából itt is indexekre oldjuk fel. A feloldás úgy történik, hogy amikor konstanst hozunk létre, a paramétertáblából kikeressük az azonos nevű paraméter indexét. Ha még nem létezik ilyen paraméter, akkor -1-et írunk.

Amikor viszont új paraméterbejegyzést hozunk létre a paramétertáblába, az azonos nevű konstansokban ezt az indexet tárolni kell. A konstansokat ilyenkor úgy lehet összeszedni, hogy egy tömbben tároljuk őket referenciaként, és ezt a tömböt bejárva módosítjuk a konstansokban a paramétertáblába mutató értéket.

Akkor lép fel hiba, amikor egy olyan konstanst szeretnénk a shadernek feltölteni, amihez nem tartozik bejegyzés a paramétertáblában – vagyis a paramétertáblába mutató érték -1. Ez azt jelenti, hogy a shader egy olyan paramétert várna, amelyet senki nem jegyzett elő!

Ezt úgy hidaljuk át, hogy a shaderek konstansaiból készült paraméter-előjegyzések (*parameter subscription*) már paraméterbejegyzést is elhelyeznek a paramétertáblában.

3. 8. 3. 7. Szintenkénti paramétergyűjtés

Az előző probléma ellentetje, amikor olyan paraméterek értékeit számolgatjuk ki feleslegesen, amelyeket egyetlen shader sem fog a továbbiakban felhasználni.

Itt már sajnos szükség van a szintek közötti kommunikációra. Egy felsőbb szinthez tartozó objektum elkérheti alatta lévő szinthez tartozó gyerekei azon paraméter-előjegyzéseinek listáját, amelyekkel azok nem tudtak mit kezdeni.

Mivel az anyagshader közvetlenül hozzá tud férni a shaderekhez, ezért az anyagshaderleíró fájlban szerepeltetjük az anyaghoz tartozó paramétereket illetve a meshhez,

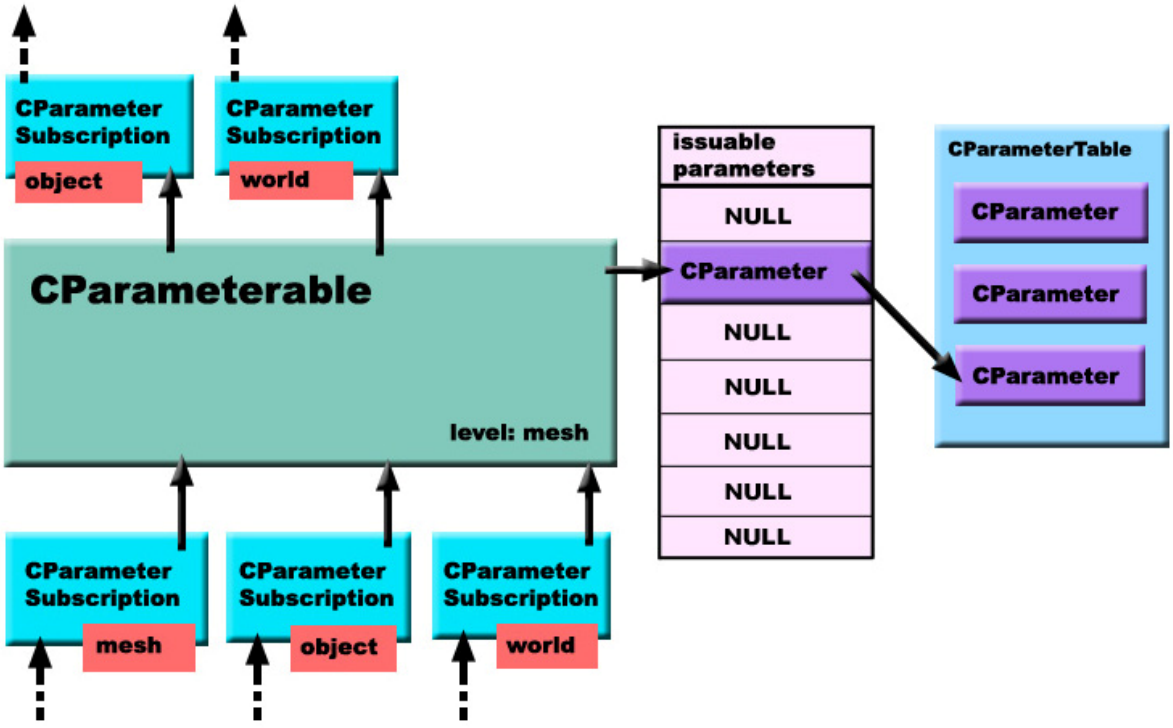
objektumhoz, stb... tartozó nem egyértelmű paramétereket a szint megjelölésével

Például a `HairColor` nevű shaderparaméternél jelezhetjük, hogy értékét az objektum szintjétől várjuk, így ez a paraméterelőjegyzés felgyűrűzik a láncon az objektumig, ott paraméterobjektum készül belőle, és az objektum paraméterlistájában jelenik meg.

Egyes konstansokról világosan eldönthető, hogy melyik szinthez tartoznak, például a nézeti mátrix a világ szintjéhez, a csontadatok a mesh szintjéhez, stb...

Ha egy objektum több mesht is felhasznál, akkor az azoktól kapott paraméterelőjegyzések közül kiválogatja az objektumszinthez tartozókat, a többit (világszintűeket) pedig a világ felé továbbítja. A megtartottakból paraméterobjektumokat készít, melyeket rajzolása előtt feltölt a kiszámított értékekkel.

Látható, hogy ez a feladat szintenként azonos: összegyűjteni a paraméterelőjegyzéseket a regisztrált gyerekektől, a szinthez tartozókat kiszűrni (a szint által felismerteket vagy az anyagshaderben explicit módon ehhez a szinthez rendelteket), belőlük paramétereket példányosítani, a többit pedig továbbadni feljebb.



5. ábra. Paraméterelőjegyzések gyűjtése a regisztrált elemekből, paraméterek készítése a szinthez tartozó előjegyzésekből, a többi raktározása továbbadás céljából.

Ezt egy kis osztály implementálásával oldhatjuk meg egyszerűen, amely majd a paraméterező rétegek őse lesz: `CParameterable`.

Figyelni kell arra is, mi történik abban az esetben, ha opcióváltás során az anyagshaderek shadert váltanak. Ilyenkor az összes paraméterezhető objektumban érvényteleníteni kell a paramétereket és az előjegyzéseket, majd az anyagshaderből újra fel kell gyűrűztetni ezeket – lehetőleg egy teljes rajzolás (*frame*) alatt.

A `CParameterable` osztály ezt a mechanizmust automatikusan megoldja. Egy objektum figyeli, hogy egy globálisan megadott frissítő értékhez képest a saját frissítő értéke mennyi. Ha eltérő, frissítenie kell gyerekeit, belőlük összegyűjteni, szűrni a paraméterelőjegyzéseket, a szinthez tartozó paramétereket létrehozni, majd a frissítő értékét beállítania a globálisra. Ez a módszer egyfajta *timestamp* technikának is felfogható[21].

Vagyis opcióváltásnál az összes anyagshaderben összegyűjtjük az új shaderekhez tartozó konstansokat, ezekből paraméterelőjegyzéseket készítünk. Majd a globális frissítő értéket megváltoztatjuk. Amikor a világ rajzolni kezd, a paraméterei nem lesznek érvényesek, így végigfrissíti a gyerekeit – és ezután már újra érvényes paraméterei lesznek.

Hogy a paraméterelőjegyzések továbbítása és a frissítés automatikus legyen, egy paraméterezhető objektum konstruktorában meg kell mondani az objektum szintjét a `SetLevel` függvénnyel, továbbá implementálnia kell a `SetupParameters` függvényt, és lehetőség szerint az alábbi módon kell kezdődnie ezen függvénynek:

```
if (NeedsUpdate())
{
	RegisterParameterableChild(m_pThing);
	RegisterParameterableChildren((vector<CParameterable*> &)m_pvMoreThings);
	UpdateNode();
}
```

Ezután kezdődhet a létező paraméterek értékeinek beállítása.

3. 8. 4. Hatékony rajzolás

Tudjuk, hogy a világ illetve az aktuális rajzolási sor mely objektumokat fogja megjeleníteni. Az objektumokból a LOD szint alapján kiválasztható a rajzolásra szánt mesh a hozzákapcsolt animált csontvázzal. Egy ilyen mesh több darabból áll, anyagonként – esetleg a csontváz felosztása miatt – csoportosítva a poligonokat. Ez a felosztás azt is garantálja, hogy az egy

meshdarabhoz tartozó poligonokat ugyanaz az anyagshader, vagyis ugyanazok a shaderek fogják rajzolni.

Habár a shaderváltás nem történik meg, ha kétszer egymás után alkalmazzuk ugyanazt az anyagshadert, sok mesh kirajzolásánál már problémát okozhat: ugyanaz a néhány shader váltogatja majd egymást a csővezetékben.

A shaderváltás sajnos nem olcsó folyamat, ezért szükséges valamilyen optimális utat találni erre a problémára. Erre kínál megoldást a **3. 9. 3. Shaderváltási stratégia** c. fejezetben ismertetett módszer.

3. 8. 5. A vezérlési szint problémái

A grafikus motor szintjeinek működését már ismerjük. Azonban még nem tudjuk, hogyan fog együttműködni a játékmotor többi alrendszerével, illetve magával a keretrendszerrel. Miután létrehoztunk egy ablakot, valahogy szeretnénk, ha a grafikus motor azt használná.

Ilyen célokat szolgál a `TheGraphicsEngine` nevezetű singleton példány, az ún. *vezérlési szint*, mely a grafikus motorral való kommunikációt (a grafikus eszköz létrehozását, a rajzolási fázis megkezdését), stb... teszi lehetővé a többi alrendszer és a keretrendszer felé.

A kommunikációs pontok kialakítása feladatfüggő. Ha a fizikai motor kapcsolatot szeretne létesíteni a grafikus motorral, üzenetet küld, hogy mely objektumok mozgásáról szeretné tájékoztatni a grafikus motort. Ekkor a grafikus motor kérvényez a keretrendszertől egy megfelelő nagyságú memóriaterületet (vagy őmaga létrehozza azt), erről tájékoztatja a fizikai motort, és informálja világának megfelelő objektumait, hogy transzformációs mátrixaikat ebből a memóriaterületből fogják tudni kinyerni.

Hasonlóan kell eljárni a törésszimulációnál, animációnál, anyagváltozásnál (például egy karakter sebesülését folyamatos textúraváltásokkal akarjuk érzékeltetni, mint ahogy a *Quake 2.* c. játékban volt), kamerák mozgatásánál, és egyéb hasonló jellegű problémánál.

Megjegyzés. Mivel a grafikus motor képes párhuzamosan működni a többi alrendszerrel, a közös memóriaterületekre *zárolási szabály*ok alkalmazandók. Például amíg a fizikai ír, addig a grafikus motor nem olvashatja, mert inkonzisztens állapotú a memóriaterület. Ez *szaggatás*hoz vezethet, így érdemes jó *adatmegosztó-stratégiá*t kitalálni.

Miután a `TheGraphicsEngine` létrehozta a grafikus eszközt, azt az alsóbb rétegeknek is el kell érniük, melyek a Direct3D függvényeire támaszkodnak. Nem lenne szép, ha mondjuk a textúrabetöltő szintnek használnia kellene a grafikus motor legfelső szintjét.

Ezt a problémát a `TheDeepCore3d` alsó szint bevezetése fogja megoldani, ahol a grafikus eszköz elérhető. Ugyanitt lesz két függvény: az egyik eszközvesztés (*device lost*), a másik eszközvisszaállítás (*device reset*) esetére.

Amikor egy egység videomemóriabeli erőforrásai érvénytelenné válnak, lépéseket kell tennie az erőforrás újbóli létrehozására. Ezt úgy fogja tudni megtenni, ha saját függvényeit regisztrálja *callback* függvényként a `TheDeepCore3d` szintjén. Mikor az eszköz elveszik, a `TheDeepCore3d` két függvénye sorban megkeresi és meghívja az összes regisztrált callback függvényt.

3. 9. Optimalizáció

3. 9. 1. Szűk keresztmetszetek kiszűrése

Mivel játékfejlesztésről van szó, ahol elengedhetetlen a gyors megjelenítés, lényeges a grafikus motor egyes elemeinek, a kritikusabb függvényeknek a futási és a válaszideje. Ez a válaszidő nagyban függ az aktuális paraméterektől, beállításoktól, a megjelenítendő jelenet összetettségétől, az objektumok számától és egyéb tényezőktől, melyeket nem kellene optimalizálni.

Azonban van egy tényező, amin mindig lehet gyorsítani: maga az algoritmus.

A lassú megjelenítést, amely szaggatást okoz, a **szűk keresztmetszet**ek (*bottleneck*) okozzák. Ezek a keresztmetszetek azon pontjai a programnak, melyeken az átfolyó adatmennyiséghez képest kicsi a kapacitás, így a teljes folyamat kapacitása erre az keresztmetszetre korlátozódik[66].

A szűk keresztmetszetek feloldásának lépései:

1. megkeressük a legszűkebb keresztmetszetet (*performance analysis*),
2. megnöveljük a kapacitását,
3. ha a teljes folyamat még mindig nem optimális, ugrás az első pontra!

A szűk keresztmetszeteket **profilozás** útján találhatjuk meg[67]. Egyes algoritmusok, függvényhívások, programrészletek futási idejét mérhetjük, függvények hívási számát számolhatjuk össze időegység alatt, stb... Lényege, hogy az elkészült statisztika alapján ki tudjuk szűrni a lassú algoritmusokat, melyeken gyorsítani kell, a sokszor hívott függvényeket, melyeket inline függvényekké teszünk, gyorsítunk, vagy más technikával megpróbáljuk a hívások számát korlátozni.

Az optimalizálásnak több módja létezik a keresztmetszet jellegéből adódóan.

Vannak nem grafikus eredetű keresztmetszetek, melyek a CPU-t veszik maximálisan igénybe. Például bonyolult számítások, gyakori memóriamásolások, gyakori objektumlétrehozások.

A grafikus keresztmetszeteket szintén több probléma okozhatja. Az alábbi táblázatban felsoroltam általános problémákat, melyek szűk keresztmetszeteket okoznak, illetve ezek javítási lehetőségeit.

Probléma	**Probléma oka**	**Probléma megoldása**
Sok vertex- és pixelművelet számítása történik fölöslegesen.	Túl részletes objektumok a távolban.	Részletezettségi szintek bevezetése (*LOD*), *PatchMesh*ek használata, ahol mód van rá, *imposztortechnika*.
	Sok objektumot kell megjeleníteni a képernyőn egyszerre.	Kitakarásvizsgálat (*occlusion query*), térbeli struktúrált felosztás (például *Binary Space Partitioning*), *portáltechnika*.
	Túl messze van a hátsó vágósík.	Hátsó vágósík közelebb hozása köddel kombinálva, új vágósík definiálása például tükröződésnél.
Sok pixelművelet számítása történik fölöslegesen.	Egy pixelt többször írunk, de mindig csak a legfelső látszik. Ez akkor a legkritikusabb, amikor bonyolult, hosszú pixelshadereink vannak.	Előre hozott mélységi rajzolás (*pre z-pass*), stenciltesztelés, előlről hátrafelé történő rajzolás, kivéve az alfakeveréses objektumokra.

Probléma	Probléma oka	Probléma megoldása
Lassú pixelshaderek.	Bonyolult, ciklust tartalmazó pixelshader.	Az assembly kódot figyelve optimalizálni kell a shadereket, a ciklusokat el kellene kerülni. Műveletek átvezetése a motorba vagy a vertexshaderbe. Esetleg részletezettségi szinteket adni a shaderhez.
	Hatalmas textúrák. Ki-be lapozás történhet a videomemóriában (*thrashing*).	Kisebb, tömörített textúrákat, textúraatlaszokat kell használni.
	Hatalmas textúrák kicsi felületeken, bonyolult szűrésekkel.	Kis felületekre használjunk kisebb textúrákat, és, ha lehet, kerüljük a lineáris filternél bonyolultabb szűréseket! (Mipmap.)
Sok pixelművelet egy pixelre (*fillrate*) problémája.	Alfakeveréses módot használunk és sok nagy egymást átfedő poligont rajzolunk.	Ha nem szükséges, kapcsoljuk ki az alfakeverést! Csökkentsük az alfakeveréses lapok számát!
Lassú erőforrás-másolások és frissítések.	A művelet, az erőforrások helye és foglalási módja, és a zárolás együttes kombinációja nem megfelelő.	A DirectX SDK-jában több szabály van, melyet be kell tartunk az erőforrások másolásánál és frissítésénél.
Lassú shaderbeállítás.	A shaderek beállítása, a konstansok feltöltése, a rajzolási és mintavételezési állapotok váltása lassú.	Észszerű stratégiát kell találni ezek kezelésére.

Szűk keresztmetszetek szűrésnél az alábbi módon járhatunk el. A grafikus minőséget szándékosan rontjuk: rövid shadereket, kisebb rendertargeteket használunk, minimális beállításokat adunk meg. Ha a teljes rajzolási folyamat kapacitása nem nő ettől, akkor a CPU-n van a keresztmetszet. Másik irány, ha a CPU számításait és memóriamásolásait kapcsoljuk ki. Ha a kapacitás nem nő, a GPU-n van a keresztmetszet.

A feladatok egyenlő elosztását a két feldolgozó egység között **terheléselosztás**nak nevezzük (*load balancing*)[68].

Kitűnő grafikus profilozó eszköz az **NVidia PerfHUD** nevű ingyenesen letölthető program, mely a DirectX alapú grafikus alkalmazásba beépülve interaktív módon használható. Statisztikákat készít, a shaderek assembly kódját kilistázza, a rendertargetek tartalma megtekinthető, a rajzolásokat futási idő szerint sorrendbe helyezi, egy frame rajzolása lépésről-lépésre nyomon követhető, stb...

3. 9. 2. Konstansfeltöltési stratégia

A shaderváltás, az állapotok módosítása, a mintavételezők és a konstansregiszterek feltöltése egy viszonylag lassú folyamat. Ezért törekedni kell arra, hogy az egy frame-beli számukat a lehető legkisebbre korlátozzuk.

Először a konstansregiszterek feltöltésére fogunk egy stratégiát adni. Ezt a stratégiát a `CConstantManager` nevű singleton osztály egyetlen példánya valósítja meg.

Shaderenként kell egy regisztermező, melynek egy eleme (cellája) 4 float értéket tárol, egy indexet a paramétertáblába, egy darabsorszámot (*piece*), egy *dirty* és egy *valid* flaget.

Az *index* jelzi, hogy a paramétertábla melyik bejegyzésének értékét írtuk be utoljára a cellába.

A *4 float érték* az az érték, amit utoljára a cellába írtunk.

A *dirty* flag jelzi, hogy az adott cella még nincs a shaderben.

A *valid* flag jelzi, hogy a cellában tárolt érték egyezik a paramétertáblabeli értékkel.

A *piece* szám azt jelenti, hogy a többregiszteres konstansnak hanyadik regisztere található az adott cellában.

Kezdetben minden cellában az index -1, a dirty flag ki van kapcsolva, a valid flag ki van kapcsolva, a többi érték lényegtelen.

Amikor egy paraméter értéke módosul a paramétertáblában, akkor a hozzátartozó cellákban a valid flaget kikapcsoljuk. Ez fogja garantálni, hogy a legutóbbi feltöltés óta változott a paraméter értéke, ezért mindenképpen ki kell írni.

Amikor értéket akarunk beírni a paramétertáblából, akkor a regisztermező írni kívánt szakaszán az alábbi feltételek együttes teljesülése kell ahhoz, hogy ne írjunk a regisztermezőbe: a cellákban az index legyen azonos az írandó paraméter indexével (ugyanaz a paraméter), a piece értéke 0-tól kezdve folyamatosan növekedjen (ugyanaz volt a kezdő regiszter és nem volt „beleírás”), a valid flagek be legyenek kapcsolva (a paraméter értéke két írás között nem változott).

Amikor beírjuk az adott szakaszra a paraméter értékét, akkor a dirty és a valid flageket itt végig bekapcsoljuk, az indexekbe a paraméter indexét másoljuk, a piece értékét 0-tól kezdve cellánként eggyel növelve beírjuk. A valid flagek bekapcsolása jelenti azt, hogy amit beírtunk, az megegyezik a paramétertáblában található adattal. A dirty flagek pedig biztosítják a szakasz feltöltését a shadernek. A piece gondoskodik arról, hogy a konstanselcsúszások miatti feltöltési hibát elkerülhessük.

Miután végeztünk az összes konstans beírásával a fent említett módon, sor kerülhet a tényleges feltöltésre: a dirty szakaszokat töltjük fel, majd az összes dirty flaget kikapcsoljuk – hiszen minden regiszter értéke már a shaderben van!

Ezen algoritmus hatékonyságához az kell, hogy a konstansokat shaderenként nagyjából azonos módon osszuk ki – itt jön jól az explicit HLSL-beli megadási mód. Például a `View`, `Projection`, `Sun` konstansok kerüljenek a legtöbb shaderben a 0., 4., 8. slotra! Ennek következetes betartásával ezen konstansok feltöltését máris optimalizálta a rendszer!

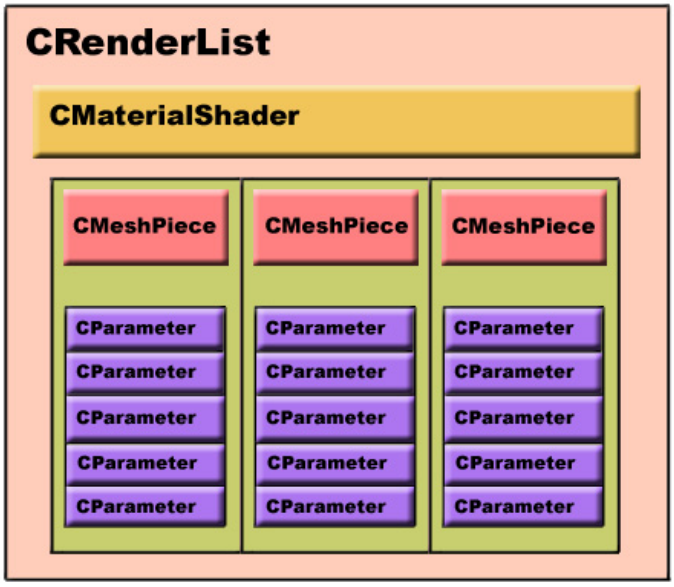
Egy példa található az **5. 4. Egy példa a konstansfeltöltések** optimalizálására c. függelékben!

Mintavételezések feltöltésekor szintén egy regisztermezőt hozunk létre, ám ennek elemei csupán fizikai címek lesznek! Elég a textúrák címét összehasonlítani, hogy tudjuk, fel kell-e tölteni az adott regiszterbe textúrát.

3. 9. 3. Shaderváltási stratégia

Minden anyagshader más-más shadert alkalmazhat. Nagyon ritkán fordul elő olyan eset, hogy két anyagshader ugyanazt a shadert fordította és használja. Ebből következik, hogy a shaderek váltásának figyelése helyett elég az anyagshaderek váltását figyelni. Vagyis, ha az előzőleg használt anyagshader megegyezik azzal az anyagshaderrel, amivel most akarok rajzolni, akkor nem kell a shadereket sem váltani.

Ez odavezet, hogy létrehozunk egy anyagshaderlistát, melyben minden anyagshader csak egyszer szerepel. Két anyagshader akkor egyezik meg, ha ugyanabból a leíró fájlból töltöttük be őket, vagyis nevük azonos.



6. ábra. Egy rajzolási lista elemének felépítése.

Azonban meg kell oldani még azt is, hogy rajzoláskor az azonos anyagshaderű meshdarabok egymást kövessék.

Ezt késleltetett rajzolási módszerrel oldjuk meg. Létrehozunk rajzolási listákat, annyit, ahány anyagshaderünk van. Amikor egy meshdarabot ki kell rajzolni, a hozzátartozó anyagon keresztül megkeressük, melyik anyagshader fogja rajzolni, és az ahhoz tartozó rajzolási listába betesszük.

Amikor a jelenet összes meshének összes darabját bepakoltuk rajzolási listába, akkor indulhat a listák rajzolása.

Ez a módszer fogja garantálni, hogy teljes rajzolásonként minden anyagshadert – és ezáltal lehetőleg minden shadert – csak egyszer állítsunk be a videokártyának.

Szükség van a meshdarab feletti szinteken beállított paraméterek mentésére, hiszen a rajzolási listák összekeverik a meshdarabok rajzolásának sorrendjét, így a fentebbi rétegek beállításai el fognak veszni. Elegendő a paraméterek referenciáit tárolni, nincs szükség az értékek másolására egy új paraméterobjektumba, hiszen egy frame-en belül egy paraméterezhető objektum csak egyszer állítja be saját paramétereit.

A sebesség megtartása érdekében a rajzolási lista osztályában ragaszkodtam a statikus tömbökhöz az STL tárolók használata helyett.

A meshdarabokhoz letárolt paraméterek száma az anyagshadertől függ, így csak annak frissítésekor kell ezen a számon változtatni (a paramétertömböket újraallokálni).

A beérkező meshdarabok száma azonban eltérő lehet, így erről frame-enkénti rajzolási számot kell nyilvántartani. Amikor a megadott kapacitásnál több meshdarab érkezik, a tömb méretét, vagyis a kapacitást automatikusan megnagyítjuk valamennyi elemmel. Esetleg lehetőség van a rajzolt meshdarabok számának figyelésével arra, hogy a több frame-en keresztül fölöslegesen nagy kapacitást csökkentsük.

3. 9. 4. Állapotváltási stratégia

A fix működésű csővezeték (*fixed function pipeline*) pixelműveleteit pixelshaderek végzik. Így megszabadulunk a keverési állapotok beállításától (*texture stage states*). Ezekkel több textúra, a vertexszín, konstans színek, stb... közti műveleteket állíthatjuk be korlátozott és nem túl intuitív módon.

Azonban megmaradnak a rajzolásra vonatkozó állapotok (*render states*) és a mintavételezési állapotok (*sampler states*). Az előbbiek a vágásra, lapon belüli árnyalásra, alfakeverésre, alfatesztre, stenciltesztre, mélységi tesztre, stb... vonatkoznak, az utóbbiak a textúrák mintavételezési módját szabályozzák.

Az állapotváltások az anyagshaderváltásokhoz kapcsolódnak, így az ott bemutatott listázó technika miatt az állapotváltások száma is csökken.

Tovább csökkenthetjük azonban, ha figyeljük, mely állapotok változnak meg ténylegesen. Ehhez figyelnünk kell korábbi értéküket is, és ha változást észlelünk, csak abban az esetben „szólunk róla” a shadernek. Amennyiben nem *pure* beállítású eszközt használunk, erre a lépésre nincs szükség, hiszen az eszköz megszűri a redundáns állapotváltásokat[2].

Pure eszköz esetén: mivel az anyagshaderlistát – így az anyagshaderek sorrendjét – ismerjük, lehetőség van úgy rendezni az egyes anyagshadereket, hogy az összes állapotváltás száma a lehető legkevesebb legyen. Ehhez a problémához a mesterséges intelligencia témaköréből ismeretes optimális megoldás-keresőket implementálhatunk, melyek előállítják ezt az optimális anyagshaderlistát.

3. 9. 5. Geometriai optimalizáló modulok

Léteznek olyan több éves tapasztalatra épülő, megvásárolható modulok, melyek kifejezetten a grafikus optimalizálásra lettek kitalálva. Előnyük, hogy elegendő geometriai adatokat átadnunk nekik, ők pedig visszaadják az optimális geometriai adatokat.

Az optimalizálás lehet kitakarás vizsgálat (*occlusion culling* vagy *occlusion query*), meshek szalagosítása (*stripify*) optimális módon, portáltechnika, stb... Gyakran párhuzamosan működő elemeket tartalmaznak és bonyolult, de kipróbált matematikai modelleken alapszanak.

Ha alkalmazásunkban nem készültünk fel egy ilyen modul használatára – tehát sok ideig tartana a geometriai adatok összegyűjtése – akkor nem biztos, hogy érdemes ilyen modulok beépítésével próbálkozni. Ekkor ugyanis az optimalizáló algoritmust megelőző gyűjtési, konvertálási idő (*overhead*) miatt még tovább eshet az FPS.

Mint ahogy már említettem, tökéletes játékmotor, tökéletes grafikus motor nem létezik. Tökéletes optimalizáló modul sem létezik. Ha komolyan gondolkodunk alkalmazásunk sebességjavításában, akkor célszerű a már meglévő technikákhoz illeszkedő saját optimalizáló algoritmus implementálása, mely ugyan nem épül bonyolult matematikai modellekre, de sokszor az apró trükkök oldják meg a legégetőbb problémákat.

3. 9. 5. 1. További ötletek az optimalizálásra

A legjobb eset az, ha rajzolásonként (valamely `DrawPrimitive` függvény egyetlen hívásakor) kb. 1000 vertexhez tartozó poligon kerül kirajzolásra[2].

Statikus objektumok esetén ún. *degenerált háromszög*eket illeszthetünk be az objektumok egy-egy vertexei közé, ezzel láthatatlanul össze lehet őket kötni. Ennek eredményeként jól szalagosíthatók és több kis objektum befér egyetlen rajzolásba (*batching*). Ilyenkor azonban a kitakarásvizsgálat és a LOD-ozás nem lesz rájuk a kívánt hatással.

Előre ismert számú dinamikus objektum esetén, mikor minden objektum geometriailag egyforma, lehetőség van arra, hogy mátrixpalettát feltöltve a vertexshadernek (hasonlóan mint a *csontozás*) oldjuk meg a más-más pozíció és orientáció kérdését[44; 64. o.]. Ekkor a vertex- és az indexpufferekbe sokszorosítjuk az objektum transzformálatlan adatait, a vertexformátum

pedig kibővül egy `BlendIndices` elemmel, mely a mátrixpalettába mutató index – objektumonpéldányonként azonos minden vertexben, a súlyozás pedig nyilván most nem kell.

Nemcsak transzformáló mátrixtömb tölthető fel a vertexshadernek, hanem akár megvilágítási vagy egyéb hasznos információkat tartalmazó tömb is.

Ezt a technikát nevezzük a meshek példányosításának (*instancing*).

3. 10. Fájlformátumok

Mivel igen fontos az adatvezéreltség, szólnunk kell mindenképpen a fájlformátumokról, vagyis a grafikus motor által beolvasott fájlok szerkezetéről.

Kiemelendő az **XML**, mint fájlformátum. Előnye a rugalmas, magában a fájlban definiált adatszerkezet és a könnyű olvashatóság, szerkeszthetőség. Megszabadít az adatbázisok és a szigorú szabályok szerint szerkesztett fájlok (bináris fájlokban például bájtokban kifejezett adatelem-pozíció, adatelemhossz, stb...) merevségétől.

Beolvasásához a *Sourceforge*-on megtalálható *TinyXML* nevű nyílt forráskódú függvénykönyvtárat fogom használni, mely számos lehetőséget biztosít az XML fájlok kezelésével kapcsolatban[69].

3. 10. 1. Textúrák

Az egyik legfontosabb erőforrás a textúra. Ennek fájlformátumaként a DirectX fájlformátumát fogjuk használni, a **Direct Draw Surface**-t (*DDS*).

Többféle tárolási módot engedélyez, a kétdimenziós textúrán kívül a kockatextúrát (*cube texture*) illetve a volumetrikus textúrát (*volume texture*).

*Mip-map szintek*et is támogat, de azok az alkalmazásból is generáltathatók.

A lehetséges csatornaformátumok listája a DirectX SDK súgójában található meg, és a kártya lehetőségeinek lekérdezésével (*card caps*) kapunk megerősítést az egyes formátumok támogatottságáról.

Több szintű veszteséges tömörítésre is lehetőségünk van, és amikor ez nem okoz látható vizuális minőségromlást, ajánlott is tömöríteni (*DXT szintek*). Itt megemlítendő, hogy a videokártyák a tömörített DDS formátumot támogatják, így a textúrát nem kell feltöltés előtt visszafejteni. Habár a tömörítés minőségromlást okoz, a videomemóriába ezáltal több textúra férhet be.

Sok kicsi textúrát, melyeket nem fogunk a felületeken ismételtetni, célszerű ún. **textúraatlasz**ba (*texture atlas*) összemásolni. Ekkor a mip-map szintek által okozott artifactok (a textúradarabok színe a szűrés miatt összevegyülhet) elkerülése végett fontos a textúradarabok közti megfelelő színes rés biztosítása (*padding*).

3. 10. 2. Objektumok

Az objektumok tárolására létrehoztam egy saját adatszerkezetet. Az objektum alaptulajdonságain kívül az objektumhoz tartoznak paraméterbeállítások és LOD-onként szétválogatott meshek.

Egy mesh több darabból állhat (*meshpiece*), anyagok szerint csoportosítva. Erre azért van szükség, mert a vertexformátum anyagonként más és más lehet, fölösleges lenne az egész mesht a legkisebb közös vertexformátum alapján tárolni. Valamint biztosítani kell a rajzolási listák számára is a mesh darabjait: az anyagok szerinti felbontás megoldja az anyagshaderek szerinti felbontást is.

Ezen kívül a mesh tartalmaz egy csontvázleírást is. A csontváz – és ezáltal a mesh – darabokra bontásáról a grafikus motor gondoskodik.

Egy meshdarab a hozzá felhasznált anyag nevét tartalmazza, a vertexformátumot, a vertexlistát és a poligonlistát.

Amikor exportáljuk az objektumot a modellező programból (*3ds max 7*-ből), nem fogjuk tudni az anyagok pontos nevét, sem a paramétereit az objektumnak. Emiatt az exportált objektumhoz elő kell állítani egy valódi objektumleíró fájlt is, melyben a fenti információk és a geometriai adatok mind megtalálhatóak.

Ide kapcsolódik két **plugin**, melyeket én fejlesztettem. Az egyikkel a *3ds max 7* anyagában állíthatjuk be a kívánt vertexformátumot, a másikkal pedig egy objektumot tudunk exportálni. Az előálló fájl a geometriai adatokon kívül nem tartalmaz egyéb információkat.

3. 10. 3. Anyagshaderek

Szükség van az anyagokat és a shadereket összekapcsoló anyagshaderekre. Ezek leírását külön fájlokban találjuk meg.

Egy ilyen fájl feltételes, makrózott shaderfordításra ad lehetőséget, a shaderek konstansaiból előálló paraméter-előjegyzésekhez szint rendelhető, és megadhatók még az anyagshader által igényelt állapotváltások és az anyagshader-felülbírálások is.

Hogy milyen anyagshadereket kell betölteni és feldolgozni, azt egy külön listafájl tartalmazza. A rajzolás, világfeldolgozás közben érkező igény alapján történő betöltés nem túl

szerencsés, hiszen ha az egyik fájl hibás, nem jó, ha már csak közvetlenül rajzolás előtt értesülünk erről.

A shaderek fordítását szintén előre kell elvégezni – sok, bonyolult shader fordítása akár több percet is igénybe vehet!

3. 10. 4. Anyagok

Az anyagok leírása egyszerű: egy lista, melyen minden anyagnak egyedi neve van, és mindhez tartozik egy-egy anyagshader megnevezés illetve paraméterbeállítások.

Anyag–mesh összerendeléshez illetve anyagparaméterezéshez szükséges lehet egy anyagszerkesztő program (*material editor*) is, mellyel a grafikus interaktív és intuitív módon állíthatja be és szerkesztheti az anyagokat.

Ez a program lehet önálló program, a grafikus motorba integrált szerkesztő (ajánlott) vagy a modellezőben *plugin*ként megjelenő szerkesztő. Ez utóbbi nagyban függ a modellező szoftver lehetőségeitől.

3. 10. 5. Világleírás

Objektumaink, anyagaink, shadereink és textúráink már vannak. Azonban egy világot is kell valamilyen formában definiálni. Erre ad választ a világleíró fájl, melyben felsorolhatók a világban található objektumok, ezek transzformációs mátrixa, a hozzájuk rendelt tulajdonságok, a megvilágítási leírások, a kamerákkal kapcsolatos leírások, stb...

Nem szabad összetéveszteni a gameplay-hez tartozó világleírással! Az egyes elemek működése, reakciója bizonyos dolgokra, nem itt kell, hogy helyet kapjanak!

A nem befolyásolható, egyszerűbb animációk leírásai itt szerepelhetnek, de a megállítható, elindítható illetve egyéb összetett animációk definíciói egy külön gameplay-scriptben kell, hogy legyenek!

Ugyanez igaz a nem szorosan a világhoz tartozó objektumok definiálására, mint például az egyes karakterek, melyekhez a puszta geometriai leírásokon kívül mesterséges intelligencia, viselkedésmód leírása is társul, és a megadott módon, időben és helyen rakjuk le őket a világba.

3. 11. Modulok kapcsolása

A grafikus motor egyes részeinek – különösen speciális grafikus, matematikus algoritmusainak – implementálása helyett dönthetünk úgy, hogy megvásárolunk egy kisebb szemléltető alkalmazásban működő modult, mely elvégzi helyettünk a munkát. Léteznek ingyenes, nyílt forráskódú modulok is, de a komolyabbak, melyek évek tapasztalatai alapján születtek, általában pénzbe kerülnek.

Egy modulhoz tartoznak dinamikus illetve statikus **függvénykönyvtár**ak (*Dynamic Link Library: DLL; static link LIBrary: LIB* fájlok), és „olvasható" programrészek, melyek egy része *interfész* a könyvtárakhoz.

Amennyiben úgy döntünk, hogy dinamikusan linkeljük programunkhoz a modult (lehetővé téve annak egyszerű cseréjét, frissítését újabb verzióra a teljes alkalmazás frissítése nélkül), szükséges az alkalmazáshoz a DLL fájlokat is csatolnunk.

A forráskódok általában azt a célt szolgálják az egyszerű projekthez kapcsoláson kívül (*including*), hogy a már meglévő motorhoz tudjuk *idomítani* a modult. Az idomítás két féle lehet: funkcionális részek idomítása (fájlbetöltés, memóriakezelés, kommunikáció) illetve grafikus idomítás (a modul grafikus eredményének minél jobb beillesztése a meglévő világba). Grafikus idomítási lehetőség például a shaderek módosítása, ahol a motorunk megvilágítási, árnyalási, stb... modelljét alkalmazhatjuk.

A grafikus eszköz modullal való megosztása lesz az, ami a leglényegesebb – enélkül ugyanis egy grafikus modul nem lesz működőképes. Az eszközátadásra, eszközvesztésre és egyéb információk cseréjére (kamerapozíció például) lesznek kialakított csonkjai a modulnak – illetve, ha nincsenek, megfelelő módon ki kell őket alakítani –, és ezen csonkokat össze kell kapcsolni a motor megfelelő egységeivel.

Tehát például az eszközvesztés jelzésére a `TheDeepCore3d` szintjén már ismertetett callback-mechanizmuson alapuló listába kell regisztrálni a modul eszközvesztés-kezelő függvényét.

Másik példa: a fájlbetöltés felvezetése egyszerűen úgy történik, hogy a modul forráskódjában az összes fájlbetöltést átírjuk a saját fájlbetöltő függvényünkre.

A visszaállíthatóság érdekében használhatunk makrózott fordítási technikát.

Saját programomban egy posteffekt modult kapcsoltam a grafikus motorhoz.

3. 11. 1. Példa egy funkcionális idomítási problémára

Kezdőknek problémát okozhat a shaderek betöltésekor implementálandó *includemanager*. Az alábbi módon azonban problémamentesen megoldható.

Beágyazott osztály és az *includemanager* változó deklarációja a shader betöltését végző (például `Shader` nevezetű) osztályban:

```
// private nested include-manager class
class IncludeManager : public ID3DXInclude
{
	HRESULT __stdcall Open(D3DXINCLUDE_TYPE, LPCSTR, LPCVOID, LPCVOID*, UINT*);
	HRESULT __stdcall Close(LPCVOID);
} m_pTheIncludeManager;
```

Implementáció:

```
HRESULT Shader::IncludeManager::Open(D3DXINCLUDE_TYPE, LPCSTR pFileName, LPCVOID,
LPCVOID *ppData, UINT *pBytes)
{
	return TheFileStream.LoadFromFile(pFileName, (char **)ppData,
	(long *)pBytes) ? S_OK : E_FAIL;
}

HRESULT Shader::IncludeManager::Close(LPCVOID pData)
{
	if (pData)
	{
		free((void *)pData);
	}
	return S_OK;
}
```

A `TheFileStream` egy `CFileStream::Instance()` makró, a fájlbetöltő egység singleton osztályának egyetlen példányát hivatkozza.

Amikor pedig shaderfordításra kerül a sor és át kell adni az *includemanager*t paraméterként:

```
D3DXCompileShader((LPCSTR)data,                    // datastream
                  size,                            // data size
                  defines,                         // defines
                  &m_pTheIncludeManager,           // include manager
                  (LPCSTR)m_sEntrypoint.c_str(),   // entry point
                  (LPCSTR)m_sTarget.c_str(),       // compilation target
                  flags,                           // flags
                  &buff,                           // buffer for the shadercode
                  &error,                          // buffer for errors
                  &constanttable);                 // table of constants
```

4. Összefoglalás

Láthattuk, miért fontos egy jól működő grafikus motor. A játékokon kívül a szórakoztató multimédia és a modellező alkalmazások mögött is egy grafikus motor működik.

Azt is láttuk, milyennek kell lennie egy modern, shadereket támogató grafikus motor felépítésének; a sajátom szerkezetét, működését ismertettem.

A szerkezetében és egyes problémákra adott megoldásaiban már magában hordoz bizonyos korlátokat, azonban ezek a korlátok csak a tökéletes grafikus motorban nem léteznek, melynek tökéletességét optimális voltának hiánya csorbítja. Vagyis tökéletes motor nem létezik, de lehet törekedni a koncepciónak megfelelő ideális motor kialakítására.

Játékfejlesztésnél fő szempont a sebesség, melyet ugyan a grafikus motoron kívül álló tényezők is befolyásolnak (sok, bonyolult objektum, sok, egymást átfedő alfakeveréses lap, magas grafikus beállítások, stb...), azonban lehetőség szerint törekedni kell a motor több szintű optimalizálására a szűk keresztmetszetek iteratív szűrő-javító folyamatával.

Az sebességnövekedéshez, a megjelenítési minőség szintjének emeléshez a megírt algoritmusok, shaderek optimalizálásán túl segítséget nyújtanak optimalizáló modulok is.

A grafikus motor felépítése rétegekre osztott, ahol minden egyes szint vagy réteg csak az alatta lévők szolgáltatásait használhatja.

Kiemelendő az anyagshader és az anyag szintje. Az előbbi a shaderváltás, shaderbetöltés, konstansgyűjtés feladatát oldja meg, az utóbbi pedig már ezen bonyolult implementációktól mentes objektumként látható a felsőbb szintek számára, kellemes eszközt biztosítva a meshek anyagainak kialakításához.

A paraméterek kezelése – melyekben beállított értékek a shaderek konstansaiban jelennek meg – egy fontos, megoldandó probléma. Találtam rá egy jól működő stratégiát, mely ráadásul optimális konstansfeltöltő mechanizmus kialakítására is lehetőséget adott.

A lényege, hogy minden szint létrehozhat paramétert, és ezek értékét akkor számítja ki és tölti fel, ha van rá paraméter-előjegyzése. A paraméter-előjegyzések a shaderek fordításából származó konstansokból erednek, melyek frissítéskor, gyűjtéskor szintenként haladnak felfelé a motorban. Minden szint kiválogatja az általa felismert vagy neki címzett paraméter-előjegyzéseket, paramétert készít belőlük, melybe a számított értékeit írja majd bele, a többi előjegyzést pedig biztosítja a közvetlenül felette álló szintnek.

A paraméterekhez tartozik egy rejtett paramétertábla, melyekben egyedi paraméterek találhatók. A hozzárendelés n:1 arányú, vagyis több paraméter, paraméter-előjegyzés és konstans hivatkozhatja ugyanazt a bejegyzést – név alapján. Az aktuális shader így az utoljára beállított paraméterértékeket fogja megkapni, amikor rajzolni kezd.

Fontos volt a megfelelő fájlformátumok kialakítása is.

Például a DirectX mesheit tároló X fájlformátum helyett mindenképpen kellett egy XML alapú objektumtároló adatszerkezet. A cél a könnyű olvashatóság és a grafikus motor által könnyen értelmezhető, feldolgozható megadási mód.

Exportáló pluginek írását is szükségessé tette az új fájlformátum.

Szükség szerint lehetőség van anyagszerkesztő létrehozására is, mellyel a grafikusok interaktív és intuitív módon szerkeszthetik az objektumok anyagait.

Modulok kapcsolásához megfelelő módon kell eljárni a grafikus motorban, azt fel kell készíteni külső bővítmények fogadására.

Modul illesztésénél szükség van a modul funkcionális illetve grafikus idomítására is, hogy működésében és a megjelenítés minőségében is alkalmazkodjon a már meglévő részekhez, modellekhez.

A grafikus motor kommunikációs pontját a legfelső, ún. vezérlő (`TheGraphicsEngine`) szinten alakítjuk ki, mely kapcsolatban áll a világ szintjével és a teljesen alul elhelyezkedő `TheDeepCore3d` szintjével is. Ez utóbbi feladata a grafikus eszköz és alapfunkciók biztosítása a rá épülő rétegek számára. Ilyen alapfunkció például az eszközvesztés esetén fellépő callback lista elemeinek meghívása.

4. 1. További kutatások és fejlesztések

További fejlesztési lehetőség kínálkozik azzal, ha az alsóbb rétegekben a Direct3D 10-es változatát is felhasználjuk, ezáltal újabb szolgáltatásokhoz jutva. Makrózott fordítással elérhető, hogy a Direct3D függvényeiből mindig a megfelelő változatút hívjuk meg.

Ebből mérhető, hogy a grafikus motor mennyire készült fel az API-beli módosításokra. Ha csak néhány fájlban kell a megfelelő helyeken módosítani a kódot, lehetőleg az alsóbb szintekhez tartozó osztályokban, akkor mondhatjuk azt, hogy felkészült rá.

A Direct3D 10 lehetővé teszi a geometriashader használatát is, feltéve, hogy a videokártya támogatja a 4.0-ás shadermodellt. Ennek integrálása a grafikus motorba nem szabad, hogy problémát okozzon. A `CBaseShaderAlternate` osztályból származtatva létrehozható a geometriashader osztálya a `CVertexShaderAlternate` és a `CPixelShaderAlternate` osztályok mintájára, az anyagshaderben pedig gondoskodni kell ezen új elem kezeléséről is.

A leíró fájlokban szintén nem okoz problémát a regisztráció – ha ott fel van tüntetve, foglalkozunk vele.

A grafikusok, gameplay-programozók, designerek interaktív és intuitív módon szeretnék elkészíteni a játékot, lehetőleg minél kevesebb külső eszköz használatával. Emiatt gondolhatunk arra is, hogy a grafikus motor működésébe egy interaktív szerkesztő modult építsünk be.

Olyan dolgok lennének állíthatóak, mint az anyagparaméterek, az objektumok részletezettségi szintjei, effektparaméterek, kamerapályát leíró görbék, objektumok elhelyezése, stb...

Természetesen a shaderprogramozó feladatát is megkönnyíthetjük hasonló módon, ezáltal lehetővé téve a magas hatékonyságú shaderkutatást és –fejlesztést. Egy ilyen modullal a shaderek működése nem csak összehangolható, de újabb shadereket is ki lehet fejleszteni hatékony módon.

A többmagos processzorok megjelenésével érdemes elgondolkodni a grafikus motor moduljai működésének párhuzamosításán. Nyilván vannak olyan szakaszok, melyek egymással logikailag nem párhuzamosíthatók, vagy nem lesz optimálisabb a párhuzamosítás után a működésük. Ennek vizsgálatáról külön esettanulmányok készíthetők.

5. Függelék

5. 1. Nem optimalizált, SSE-optimalizált és D3DX függvények

Példa egy vektor hosszát visszaadó vektor osztálybeli függvényre. Az alábbi kódrészletek ugyanazon állományból (`math3d.h`) származnak, bennük elkülönítve narancssárga háttérrel a nem optimalizált számításokhoz, sárgával az SSE-optimalizált számításokhoz, zölddel pedig a D3DX függvényhívásokhoz tartozó sorok.

```
#if (IMPLEMENTATION == IMPL_SSE_OPTIMIZED)
union __declspec(align(4)) sse4
{
	__m128 m;
	float  f[4];
};
#endif
```

```
class Vector
{
	#if (IMPLEMENTATION == IMPL_NOOPTIMIZATION)
	float                    m_sVector[4];
	#elif (IMPLEMENTATION == IMPL_SSE_OPTIMIZED)
	sse4                     m_sVector;
	#elif (IMPLEMENTATION == IMPL_D3DX)
	D3DXVECTOR3              m_sVector;
#endif
[...]
};
```

```
__forceinline float Vector::Length() const
{
	#if (IMPLEMENTATION == IMPL_NOOPTIMIZATION)
	return sqrt(m_sVector[0] * m_sVector[0] + m_sVector[1] * m_sVector[1] +
		m_sVector[2] * m_sVector[2]);
	#elif (IMPLEMENTATION == IMPL_SSE_OPTIMIZED)
	sse4 temp;
	temp.m = _mm_mul_ps(m_sVector.m, m_sVector.m);
	temp.f[3] = temp.f[0] + temp.f[1] + temp.f[2];
	temp.m = _mm_sqrt_ps(temp.m);
	return temp.f[3];
	#elif (IMPLEMENTATION == IMPL_D3DX)
	return D3DXVec3Length(&m_sVector);
	#endif
}
```

A különböző függvények és operátorok működésének helyesség- és pontosságellenőrzése után több milliószor futott le ugyanaz a számítási folyamat mind a három esetben. A mért időeredmények az alsó sorokban láthatók.

Nem optimalizált esetben:

```
v1: (0.5000, 0.2000, 0.8000).
v2: (-20.0000, 0.6700, 5.6667).
Length of v1: 0.9644.
Length of v2: 20.7981.
Dot product of v1 and v2: -5.3327.
v3: addition of v1 and v2: (-19.5000, 0.8700, 6.4667).
v3 scaled by -10.0f: (195.0000, -8.7000, -64.6667).
Normalized v1: (0.5185, 0.2074, 0.8296), length: 1.000000.
Cross product of v1 and v2: (0.6194, -19.5293, 4.4952), length: 20.0495.
----- Speed testing -----
Calculations [non optimized] took 187 ms.
```

SSE-optimalizált esetben:

```
v1: (0.5000, 0.2000, 0.8000).
v2: (-20.0000, 0.6700, 5.6667).
Length of v1: 0.9644.
Length of v2: 20.7981.
Dot product of v1 and v2: -5.3327.
v3: addition of v1 and v2: (-19.5000, 0.8700, 6.4667).
v3 scaled by -10.0f: (195.0000, -8.7000, -64.6667).
Normalized v1: (0.5185, 0.2074, 0.8296), length: 1.000000.
Cross product of v1 and v2: (0.6194, -19.5293, 4.4952), length: 20.0495.
----- Speed testing -----
Calculations [SSE optimized] took 10375 ms.
```

D3DX függvényhívások csomagolásával:

```
v1: (0.5000, 0.2000, 0.8000).
v2: (-20.0000, 0.6700, 5.6667).
Length of v1: 0.9644.
Length of v2: 20.7981.
Dot product of v1 and v2: -5.3327.
v3: addition of v1 and v2: (-19.5000, 0.8700, 6.4667).
v3 scaled by -10.0f: (195.0000, -8.7000, -64.6667).
Normalized v1: (0.5185, 0.2074, 0.8296), length: 1.000000.
Cross product of v1 and v2: (0.6194, -19.5293, 4.4952), length: 20.0495.
----- Speed testing -----
Calculations [D3DX] took 188 ms.
```

5. 2. A programban megjelenő fontosabb osztályok

5. 2. 1. A grafikus motor legalsó szintjén található osztályok

CDeepCore3d singleton osztály egyetlen példánya:
A Direct3D eszköz kezelésével kapcsolatos feladatokat látja el. Továbbá megtalálhatók itt callback függvényeket tároló listák is, mint például az eszközvesztés esetén lefuttatandó függvények listája.

CVector, CMatrix, CColor példányok:
A megfelelő D3DX adatszerkezeteket és függvényeket becsomagoló osztályok objektumai.

5. 2. 2. Low (alsó) szinteken található osztályok

CTexture2d, CVolumeTexture, CCubeTexture osztályok:
Adott típusú textúra tárolását biztosító csomagoló osztályok. Textúra létrehozására csak a CResourceManager osztályból van lehetőség. Ugyanitt rendertarget is létrehozható, mely speciális 2d-s textúra.

CParameter példánya:
Egy paraméterobjektum, melyet a felsőbb szinten akármelyik objektum létrehozhat és értékét változtathatja. Ezen változásokról az alsó rétegekben dolgozó mechanizmusok tudni fognak.

Amikor létrehozzuk, egy másolata kerül a paramétertáblába, ha még ott ilyen nevű nem létezik (ezt paraméterbejegyzésnek nevezzük), továbbá tárolni fogjuk a paraméterobjektumban ezen másolat indexét is.

CParameterSubscription példánya:
Shaderbeolvasás után a konstansokból generálódnak az előjegyzések, és ezek a szinteken fokozatosan haladnak fölfelé, miközben minden szint elveszi a listából a hozzá tartozó előjegyzéseket és elkészíti ezekből megfelelő paramétereit.

CParameterable osztálytól öröklő osztály példánya:
A paraméterelőjegyzések gyűjtését, szűrését és továbbadását biztosítja automatikusan. A szinthez tartozó paraméter-előjegyzésekből tényleges paraméterobjektumokat készít.

Amikor opcióváltás történik, timestamp technikán alapuló módszerrel újrafrissíti a hálózatba szervezett paraméterezhető példányokat. A frissítést és a paraméterelőjegyzések gyűjtését regisztrált gyerekein végzi el.

CConstant példánya:

Speciális paraméterobjektumnak tekinthető. Feltöltési információkat tartalmaz. Konstanshoz a felsőbb rétegek nem férhetnek hozzá. Shaderfordítás után megkapjuk a shaderek konstansait, ezekből paraméter-előjegyzések generálódnak, amik majd aztán a szinteken felfelé haladva paraméterobjektumok létrehozását eredményezik.

Az egyszerűség kedvéért a floattömböket és a textúrákat egyaránt konstansként kezelem.

CParameterTable egyetlen példánya:

Egy globális tároló a paraméterek számára. Az azonos nevű paraméterek csupán egyszer vannak eltárolva. Amikor egy paraméter értéke megváltozik a felsőbb szinteken, ezen táblában az egyező nevű paraméter-bejegyzés értéke erre módosul.

Amikor paramétert vagy paraméter-előjegyzést hozunk létre (ez utóbbit csak az anyagshader hozhat létre), megnézzük, van-e már azonos nevű a táblában. Ha van, arra fog hivatkozni, ha nincs, először beszúrunk a táblába egy ilyen nevűt, és ekkor erre fog hivatkozni.

Ennek célja az, hogy a konstansok tudják, mik az aktuális paraméterértékek. Továbbá az is nyomon követhető, hogy mely paraméterek módosították utoljára az egyes bejegyzéseket.

A táblabeli paraméterobjektumok kezelése automatikus, rejtve marad a felső szintek elől.

CConstantTable egyetlen példánya:

Egy tároló az összes konstans számára, amelyek shaderek létrehozásakor jöttek létre. Ez a tábla csupán címeket tárol, és az összes konstanst tartalmazza, így azonos nevű konstansokból többet is tárolhat.

Feladata a paraméter–konstans frissítések megkönnyítése.

CConstantManager egyetlen példánya:

Célja a regiszterek feltöltésének optimalizálása. Ezt azáltal éri el, hogy a tényleges regisztermezők mintájára töltődik fel, majd amikor végeztünk a konstansokkal, az egybefüggő, piszkos regisztermező-darabokat tölti fel ténylegesen.

Egy regiszter piszkos, ha a legutóbb feltöltött változatával nem egyezik meg.

Egy regiszter nem érvényes, ha az őt legutóbb író paraméter értéke megváltozott, és/vagy már nem egybefüggő a többi darabjával.

CVertexShaderAlternate, CPixelShaderAlternate példánya:

Egy shader és a hozzá kapcsolódó technikai információk tárolására szolgáló objektum. Nem feltétlenül van lefordítva a shader egy ilyen objektumban. Fordításkor viszont előáll a konstansok listája, mely lekérdezhető.

Kapcsolódó opciók segítségével makrózott fordításra is lehetőségünk van.

CMaterialShader példánya:

Több vertex- és pixelshader alternatívát tartalmaz. Ezek közül természetesen mindig egy-egy az aktív – az opcióknak megfelelően. Kérésre ezen shadereket feltölti a grafikus kártyának a konstansok értékeivel egyetemben. Állapotokat is beállít (render és sampler stage állapotokat).

Továbbá tartalmaz egy-egy utalást az árnyék- és a mélység/HDR számítására alkalmas anyagshaderekre.

CMeshPieceLow példánya:

Direct3D-specifikus vertex- és indexpuffert tartalmaz, ezeket feltölti az XML-ből beolvasott adatoknak megfelelően. Rajzolás is kérhető tőle: ebben az esetben beállítva a megfelelő vertexdeklarációt hozzálát az indexelt primitívlista kirajzolásához.

CResourceManager singleton osztály egyetlen példánya:

Erőforrások létrehozását, felszabadítását biztosító osztály. Ez vonatkozik nem csak a textúrákra, meshdarabokra (vertex- és indexpufferek), hanem a shaderekre és az anyagshaderekre is.

5. 2. 3. High (felső) szinteken található osztályok

CMaterial paraméterezhető példánya:

Az anyagok leírására szolgáló réteg. Egyetlen materialshader kapcsolódik hozzá. Paraméterei XML-ből olvasott értékeket vesznek fel.

CMeshPiece paraméterezhető példánya:

Egy mesh egy darabját jelöli. A megrajzolandó geometriához (CMeshPieceLow objektum) tartozik egy anyagleírás is (CMaterial objektum), továbbá egyéni paraméterei is lehetnek.

CRenderList példánya:

Egy anyagshaderhez tartozó meshdarabok összegyűjtését, majd azok egymás után való kirajzolását végző objektum. Az ilyen objektumok célja a minél kevesebb shaderváltás. Számuk megegyezik az

ismert anyagshaderek számával és minden anyagshader frissítéskor (opcióváltás) frissíteni kell a rajzolási lista objektumokat is.

Az egyes meshdarabokhoz az azok kirajzolásához szükséges aktuális paramétereket is elmenti – mivel a felsőbb szintek paraméterbeállításai elvesznek a megjelenítés keverése miatt. Ezt az elmentést az anyagshader alapján oldja meg, amely képes visszakeresni az adott szint feletti paraméterek aktuális értékét (mit írtak utoljára az anyagshader által ismert paraméterekbe a megadott szint felett?).

CAnimation példánya:

Egy előre definiált animációt leíró objektum. Az animáció minden egyes keyframe-jében letároljuk, mely csontok adata változik meg, és ezekben mi az új csontadat. Tárol továbbá az animáció lejátszására vonatkozó egyéb információkat is, mint például az animáció sebessége.

Lineárisan interpolált értékek kiszámítását is elvégzi egy ilyen objektum.

CSkeleton példánya:

Kezdeti csontvázat és az erre lejátszható animációk listáját tároló objektum. Egy adott nevű vagy azonosítójú animáció lejátszását, illetve annak szabályozását teszi lehetővé.

Esetleg lehetőséget biztosíthat több animáció keverésére, mellyel megoldhatók olyan problémák, mint „a fej forgatása vagy bólogatás sétálás vagy futás közben” tetszőleges kombinációja.

CMesh paraméterezhető példánya:

Egy objektum valamelyik LOD szintjén megjelenő mesh. Több meshdarab kapcsolódik hozzá. Ezt a felosztást az exportált XML fájl határozza meg, általában anyag szerint vannak szétosztva meshdarabokra a mesh poligonjai.

Rajzolásnál előbb a rajzolási listába pakoljuk be a mesh darabjait.

Csontvázat (CSkeleton objektumot) kapcsolhatunk a meshhez, ha vertexformátumában megjelennek a csontadatok is, vagyis a mesh csontozott (skinned mesh).

CObject3d paraméterezhető példánya:

Egy objektumot ír le. Több objektumhoz tartozhat ugyanaz a mesh, és egy objektumhoz több mesh tartozhat, ez utóbbi a LOD szintek alapján. Az objektumnak van koordinátatengelyhez igazított befoglaló doboza is (AABB), mely meghatározza a méretét, valamint kapcsolható hozzá például volumetrikus textúrába számolt ambiens kitakarás, mellyel dinamikus ambiens kitakarási effekt valósítható meg – feltéve, hogy az objektum nem csontozottan animált.

A lejátszandó animációt is itt állíthatjuk be.

Az objektumokhoz továbbá flagek is rendelhetők, melyek alapján rajzolási sorba pakolhatók vagy optimalizáláshoz extra információt szolgáltathatnak.

CCameraSystem példánya:

Kamerakezelést megvalósító objektum. Több kamera kezelésére van lehetőség.

A statikus kamerabeállítások helyett esetleg görbéket is lehet rendelni a kamerákhoz, és így a kamerák ezeken a görbéken mozoghatnának bizonyos paraméterek mellett.

Mindig csak egyetlen kamera aktív, ennek mátrixai és pozíciója fontos paraméterek értékei lehetnek, így ezek lekérdezése biztosítandó.

CLightingSystem példánya:

Többféle fényforrást is támogató objektum. Létrehozhatunk több fényforrást is. Lekérdezhető a legközelebbi adott számú fényforrás adata.

A fények típusa, pozíciója, iránya, vetítési nyílásszöge, színe, stb... kérdezhetők le és állíthatók be. Itt szintén lehetne görbéket használni a fényforrások animálása végett.

CRenderQueue paraméterezhető példánya:

Egy rajzolási sorba olyan objektumokat válogatunk össze, melyek valamilyen feltételnek eleget tesznek. Ezeket a rajzolási sorhoz rendelt rendertarget textúrába vagy a háttérpufferbe rajzoljuk, alkalmazva – és ezáltal felülbírálva a világot – a megadott paramétereket. A rajzolási sor felülbírálhatja az anyagshadereket is, ezáltal árnyék- vagy mélységi textúra nyerhető.

Az objektumok rendezésére is lehetőség van egy rajzolási soron belül. Ezáltal például a nem alfakeveréses objektumok elölről-hátrafelé való rendezése optimalizálja a rajzolást.

CWorld singleton osztály egyetlen paraméterezhető példánya:

Maga a világ, a jelenet, amit meg szeretnénk jeleníteni. Objektumokat, fény- és kamerarendszert, rajzolási sorokat tartalmaz. Rajzolásnál a paraméterek beállítása után a rajzolási sorokat futtatja le sorban.

Speciális grafikus effekteket megvalósító modulok beszúrása itt lehetséges. Ilyen például a posteffekt, füst, fénytörés a lencsén, stb... modul.

5. 2. 4. A grafikus motor legfelső szintjén található osztály

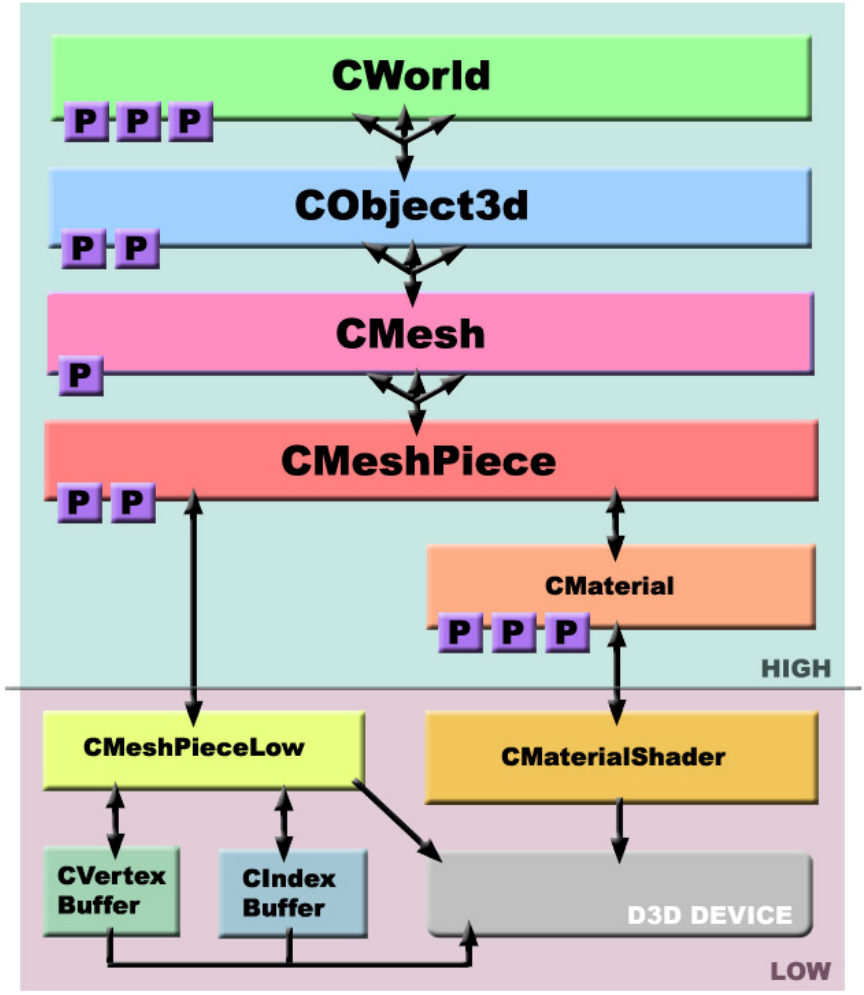
CGraphicsEngine singleton osztály egyetlen példánya:

Biztosítja a TheDeepCore3d és az alkalmazás közti kommunikációt, valamint a világ kirajzolását és paraméterezését intézi a többi alrendszerrel való kommunikáció alapján.

5. 3. A magas és az alacsony szintek közti határ

Az alábbi ábra szemlélteti a magas és az alacsony szintek közötti határt a `CMeshPiece` osztályt a középpontba helyezve.

Természetesen lehetőség van ezen kívül bármely alacsonyszintű osztály használatára, például rendertarget készítéséhez a `CResourceManager` és a `CTexture2d` osztályok használandók.

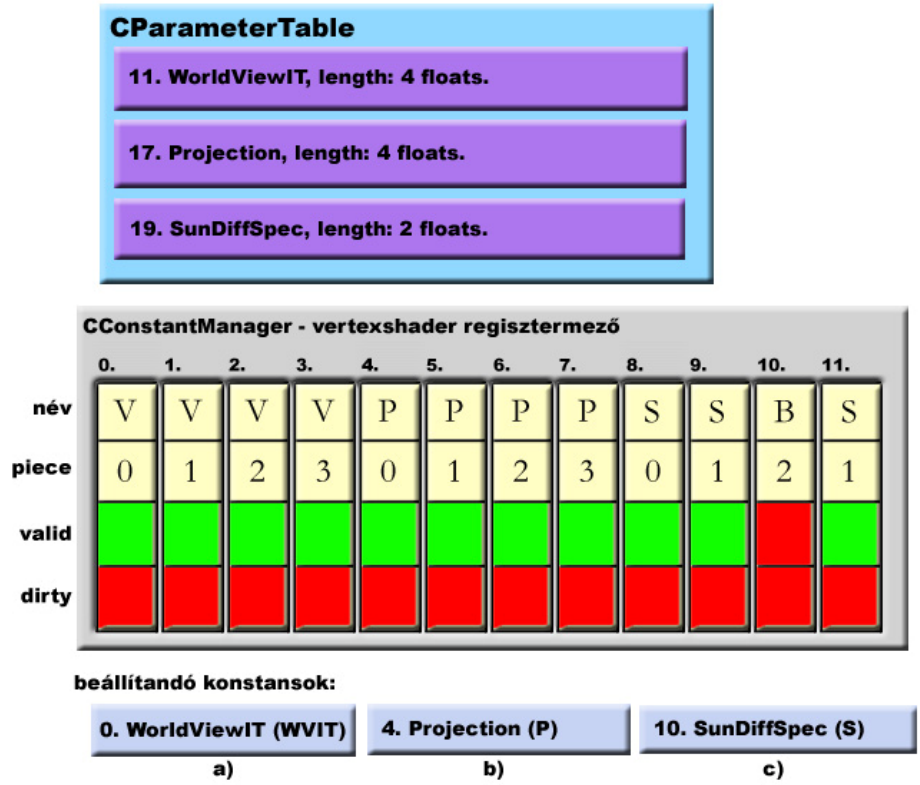


Megjegyzés. Ha dinamikusan számított vertex- vagy indexpufferre van szükségünk, egy `CMeshPiece` objektumot kell létrehoznunk, `CMeshPieceLow` objektumának pedig a vertex- illetve lapgeneráló függvényt kell *callback* függvényként megadni.

5. 4. Egy példa a konstansfeltöltések optimalizálására

Az alábbi ábra szemlélteti a vertexshaderhez tartozó regisztermező aktuális tartalmát. Látható, hogy előzőleg minden regisztert feltöltöttünk a shadernek, hiszen az összes dirty flag ki van kapcsolva. Az is látható, hogy a „B” értékét váró 10. regiszter valid flagje ki van kapcsolva, ami azt jelenti, „B” értéke megváltozott előző feltöltése óta. A többi regiszter értéke érvényes.

Három konstansfeltöltési igény érkezik.

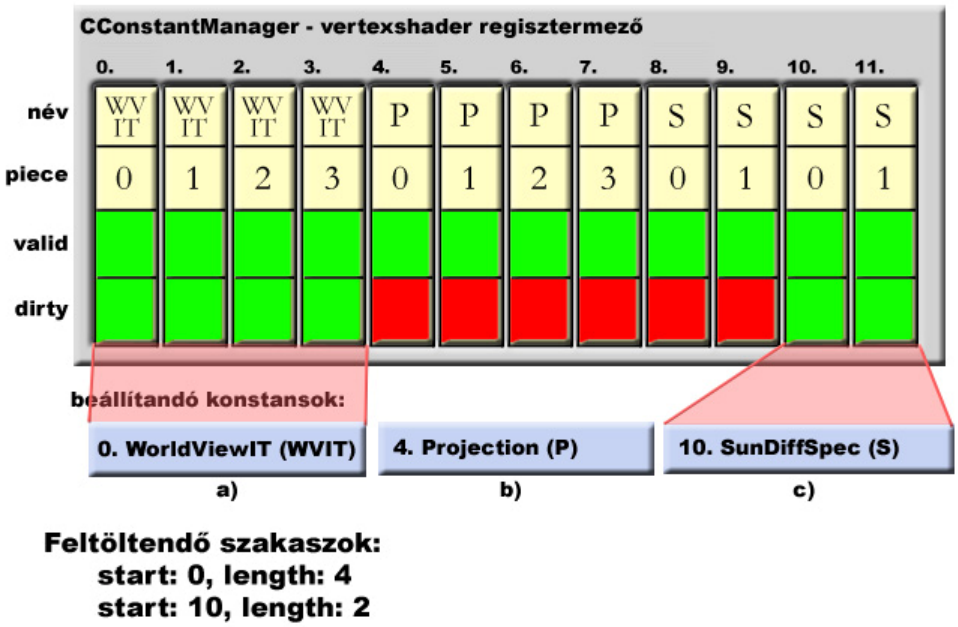


Az a) esetben a célregiszterekben más paraméterhez tartozó értékek találhatóak, ezért szükséges ezeket felülírni: a „WVIT” paramétert feltölteni a 0. regisztertől kezdve!

A b) esetben „P” paraméter értékét szeretnénk kiírni, de előzőleg pont az írtuk. A piece-kódok sincsenek elcsúszva, és „P” paraméter értéke sem változott az előző kiírás óta. Ezért b) feltöltést mellőzzük.

A c) esetben szintén nem az van a célregiszterekben, amit írni szeretnénk. A piece-kód sem jó (2-től indul 0 helyett), ráadásul a mező valid flagje sincs beállítva. Vagyis a c) feltöltésre szükség van a 10. regisztertől!

A következő ábrán már az a) és a c) feltöltések utáni állapot látható.



A regiszterekben már az új értékek szerepelnek, ezért minden írt regiszter értéke érvényes, így az összes valid flag bekapcsolt. Továbbá a név (az eredeti algoritmusban index) és a piece értékek is megfelelő módon változtak.

A dirty flagek bekapcsolásával pedig azonnal láthatóvá válik, mely regisztermező-szakaszok töltendők fel ténylegesen a vertexshadernek.

6. Irodalomjegyzék

[1] **Michael Root – James Boer: DirectX**. Panem Könyvkiadó, 2000.

[2] **The DirectX Software Development Kit documentation** (a DirectX 9.0c, 2007. novemberi SDK-jához csatolt helpfájl).

[3] **Nyisztor Károly: Grafika és játékfejlesztés DirectXszel**. Szak Kiadó, 2005.

[4] **3ds max 6 Plug-In Software Development Kit** (a 3ds max 7 maxsdk-hoz csatolt helpfájl).

[5] **Kelly L. Murdock: 3ds max 7 Biblia**. Wiley Publishing, Inc., 2005.

[6] **Dr. Szirmay-Kalos László: Számítógépes grafika.** ComputerBooks, 1999.

[7] **Johannes Staffans: Online Occlusion Culling**. Abo Akademi, 2006. Elérhetőség: http://web.abo.fi/~jstaffan/occlusion/thesis.pdf

[8] **Guennadi Riguer: Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0.** 2002. Elérhetőség: http://ati.amd.com/developer/dx9/ATI-DX9_Optimization.pdf

[9] **Frank Luna: Skinned Mesh Character Animation with Direct3D 9.0c.** 2004. Elérhetőség: http://www.moon-labs.com/resources/d3dx_skinnedmesh.pdf

[10] **Kent Knox: AMD's Direct3D Optimizations.** Software Research & Development DirectX Team, 2004. Elérhetőség: http://www.amd.com/us-en/assets/content_type/ DownloadableAssets/dwamd_Dx9Software.pdf

[11] **Matthias Bauchinger: Designing a modern rendering engine.** Elérhetőség: http://www.cg.tuwien.ac.at/research/publications/2007/bauchinger-2007-mre/ bauchinger-2007-mre-Thesis.pdf

[12] **Wikipedia: Anti-Pattern.** Elérhetőség: http://en.wikipedia.org/wiki/Anti-pattern

[13] **Wikipedia: Design pattern.** Elérhetőség: http://en.wikipedia.org/wiki/Design_pattern_ (computer_science)

[14] **Marshall Cline: Inline functions.** C++ FAQ Lite, 2006. Elérhetőség: http://www. parashift.com/c++-faq-lite/inline-functions.html

[15] **Terathon Software: C4 engine.** Elérhetőség: http://www.terathon.com/

[16] **Irrlicht játékmotor.** Elérhetőség: http://irrlicht.sourceforge.net/

[17] **Virtools játékmotor.** Elérhetőség: http://www.virtools.com/

[18] **NVidia PerfHUD 5.1 user guide.** 2007. Elérhetőség: http://developer.download.nvidia. com/tools/NVPerfHUD/5beta4/UserGuide.pdf

[19] **Bruno R. Preiss: Reference Counting Garbage Collection.** 1998. Elérhetőség: http:// www.brpreiss.com/books/opus5/html/page421.html

[20] **Ubisoft: Far Cry 2.** 2007. Elérhetőség: http://farcry.uk.ubi.com/

[21] **Wikipedia: Timestamp.** Elérhetőség: http://en.wikipedia.org/wiki/Timestamp

[22] **M. Malmer, F. Malmer, U. Assarsson, N. Holzschuch: Fast Precomputed Ambient Occlusion for Proximity.** 2005. Elérhetőség: http://www.ce.chalmers.se/~uffe/Fast%20Precomputed%20 Ambient%20Occlusion%20for%20Proximity-RR-5779.pdf

[23] **Ińigo Quilez: Screen Space Ambient Occlusion.** 2007. Elérhetőség: http://rgba.scenesp.org/iq/ computer/articles/ssao/ssao.htm

[24] **Frederick M. Waltz, John W. V. Miller: An efficient algorithm for Gaussian blur using finite-state machines.** 1998. Elérhetőség: http://www-personal.engin.umd.umich.edu/~jwvm/ece581/ 21_GBlur.pdf

[25] **David Lanier: 3D Studio Max SDK.** 3D Studio Max SDK Training session, 2003. Elérhetőség: http://dl3d.free.fr/resources/3DSMAX_SDK_DavidLanier.pdf

[26] **SirMike: Geometry Exporter - 3D Studio MAX.** Elérhetőség: http://www.sirmike.org/wp-content/uploads/2006/09/3dstudiomax_eng.pdf

[27] **gamedev.net: Normal calculation by using smoothing groups in meshes.** 2008. Elérhetőség: http://www.gamedev.net/community/forums/topic.asp?topic_id=486058

[28] **N. Hazzard: Strategies and Techniques for Realtime Shaders**. Game Developers Conference, 2006.

[29] **Natalya Tatarchuk: Practical Parallax Occlusion Mapping For Highly Detailed Surface Rendering.** 3D Application Research Group ATI Research, Inc., 2006. Elérhetőség: http://ati.amd.com/developer/gdc/2006/GDC06-Tatarchuk-Parallax_ Occlusion_Mapping.pdf

[30] **Wikipedia: PerPixel Lighting With Offset(Parallax) Mapping.** Elérhetőség: http://www.ogre3d.org/wiki/index.php/PerPixel_Lighting_With_Offset(Parallax)_ Mapping

[31] **Marshall Cline:** Operator overloading. 1991. Elérhetőség: http://earth.uni-muenster.de/~joergs/doc/cppfaq/operator-overloading.html

[32] **AMD: 3DNow! Technology manual**. 2000. Elérhetőség: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf

[33] **AMD: 3DNow! Technology FAQs**. 2008. Elérhetőség: http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_861_1028,00.html

[34] **California Institute of Technology: C++ Operator Overloading Guidelines**. 2007. Elérhetőség: http://www.cs.caltech.edu/courses/cs11/material/cpp/donnie/cpp-ops.html

[35] **C. Danger, W. Folberth Stratton: Optimizing for SSE: A Case Study.** 2002. Elérhetőség: http://www.cortstratton.org/articles/OptimizingForSSE.php

[36] Chesnokov Yuriy: 2D Vector Class Wrapper SSE Optimized for Math Operations. 2007. Elérhetőség: http://www.codeproject.com/KB/cpp/SSE_optimized_2D_ vector.aspx

[37] **Vsrajeshvs: Singleton Pattern & its implementation with C++.** 2002. Elérhetőség: http://www.codeproject.com/KB/cpp/singletonrvs.aspx

[38] **David Lanier: Choosing Between Utility and Modifier Plug-Ins for 3D Studio Max**. 2000. Elérhetőség: http://www.gamedev.net/reference/articles/article1140.asp

[39] **Apple: SSE Performance Programming.** (2008) Elérhetőség: http://developer. apple.com/hardwaredrivers/ve/sse.html

[40] **Intel, Intel C++ Compiler for Linux Intrinsics Reference.** 1998. Elérhetőség: ftp://download.intel.com/support/performancetools/c/linux/v9/intref_cls.pdf

[41] **gamedev.net: mmx matrix multiply.** 2009. Elérhetőség: http://www.gamedev.net/community/forums/topic.asp?topic_id=470231

[42] **V. Freeman: Multimedia Extensions: Do MMX, 3DNow!, and SSE Still Matter?** 2002. Elérhetőség: http://www.cpuplanet.com/features/article.php/1487301

[43] **Optimizing for SSE: Code Samples.** Elérhetőség: http://www.cortstratton.org/articles/HugiCode.html

[44] Á. Moravánszky: Dense Matrix Algebra on the GPU. NovodeX AG. Elérhetőség: http://www.shaderx2.com/shaderx.PDF

[45] **Tomas Akenine-Möller 3d grafika oktató pdf-sorozat.** Department of Computer Engineering Chalmers University of Technology, 2002. Elérhetőség: http://www.ce.chalmers.se/edu/year/2004/course/EDA425/course2002/ lectures2002/

[46] **A. Lauritzen: Variance Shadow Maps.** RapidMind, 2004. Elérhetőség: ftp://download.nvidia.com/developer/presentations/2006/gdc/2006-GDC-Variance-Shadow-Maps.pdf

[47] **C. Everitt, A. Rege, C. Cebenoyan: Hardware Shadow Mapping.** Elérhetőség: http://citeseer.ist.psu.edu/cache/papers/cs/26339/http:zSzzSzdeveloper.nvidia.comzSzdocszSzIOzSz1830zSzATTzSzshadow_mapping.pdf/everitt02hardware.pdf

[48] **F. Zhang, L. Xu, C. Tao, H. Sun: Generalized Linear Perspective Shadow Map Reparameterization.** The Chinese University of Hong Kong.

[49] **Tobias Martin: Experience with Perspective Shadow Maps.** 2003. Elérhetőség: http://www.comp.nus.edu.sg/~tants/tsm/psm.html

[50] **Marc Stamminger, George Drettakis: Perspective Shadow Maps**. REVES - INRIA Sophia-Antipolis, France. Elérhetőség: http://www-sop.inria.fr/reves/publications/data/2002/SD02/PerspectiveShadowMaps.pdf

[51] **Wikipedia: Shadow mapping**. Elérhetőség: http://en.wikipedia.org/wiki/Shadow_mapping

[52] **W. Donnelly, A. Lauritzen: Variance Shadow Maps.** Computer Graphics Lab, School of Computer Science, University of Waterloo. Elérhetőség: http://www.punkuser.net/vsm/vsm_paper.pdf49

[53] **M. Wimmer, D. Scherzer, W. Purgathofer: Light Space Perspective Shadow Maps.** Vienna University of Technology, Austria, 2004. Elérhetőség: http://www.cg.tuwien.ac.at/research/publications/2004/Wimmer-2004-LSPM/Wimmer-2004-LSPM-Paper.pdf

[54] **Leigh Davies: Optimized CPU-based Skinning for 3D Games**. 2006. Elérhetőség: http://cache-www.intel.com/cd/00/00/17/21/172124_172124.pdf

[55] **Benjamin Freidlin: DirectX 8.0: Enhancing Real-Time Character Animation with Matrix Palette Skinning and Vertex Shaders**. MSDN Magazine, 2001. Elérhetőség: http://msdn.microsoft.com/en-us/magazine/cc301765.aspx

[56] **Keshav B. Channa: Geometry Skinning / Blending and Vertex Lighting - Using Programmable Vertex Shaders and DirectX 8.0**. Elérhetőség:

http://www.flipcode.com/archives/Geometry_Skinning_Blending_and_Vertex_Lighting-Using_Programmable_Vertex_Shaders_and_DirectX_80.shtml

[57] **David Gosselin: Character Animation with Direct3D Vertex Shaders**. ATI Research. Elérhetőség: http://ati.amd.com/developer/shaderx/shaderx_characteranimation.pdf

[58] **Wikipedia: HyperZ**. Elérhetőség: http://en.wikipedia.org/wiki/HyperZ

[59] **Agner Fog: Optimizing subroutines in assembly language**. Copenhagen University College of Engineering, 1996. Elérhetőség: http://www.agner.org/optimize/optimizing_assembly.pdf

[60] **Emil Persson: Render to Vertex Buffer Programming**. Elérhetőség: http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/R2VB_programming.pdf

[61] **Anirudh S. Shastry: High Dynamic Range Rendering**. Elérhetőség: http://www.gamedev.net/columns/hardcore/hdrrendering/

[62] **Nyisztor Károly: Gyakorlati C++.** Kossuth Kiadó, 2004.

[63] **Herb Sutter, Andrei Alexandrescu: C++ kódolási szabályok**. Kiskapu Kiadó, 2005.

[64] **Brian Martin: Quaternion interpolation**. 1999. Elérhetőség: http://www.theory.org/software/qfa/writeup/node12.html

[65] **IDV: SpeedTree**. Elérhetőség: http://www.speedtree.com/

[66] **Wikipedia: Bottleneck (engineering).** Elérhetőség: http://en.wikipedia.org/wiki/Bottleneck_%28engineering%29

[67] **Wikipedia: Performance analysis**. Elérhetőség: http://en.wikipedia.org/wiki/Performance_analysis

[68] **Wikipedia: Load balancing (computing).** Elérhetőség: http://en.wikipedia.org/wiki/Load_balancing_(computing)

[69] **Grinning Lizard: TinyXML**. Elérhetőség: http://www.grinninglizard.com/tinyxmldocs/index.html

7. Köszönetnyilvánítás

Köszönetet szeretnék mondani konzulensemnek, **Dr. Kovács Emőd**nek, hogy segített kiválasztani diplomamunkám témáját, hogy ösztönzött a DirectX megismerésére, forrásokat javasolt a tanulmányozásához és, hogy figyelemmel kísérte a kutatásaimat ezen a területen belül.

Köszönetet szeretnék mondani munkatársamnak, **Kocsis Attilá**nak, a grafikus algoritmusokkal, a Direct3D-vel és a 3ds max 7 pluginfejlesztésekkel kapcsolatos tapasztalatai megosztásáért, tanácsaiért és az internetes forrásokért, melyeket ajánlott nekem.

Köszönetet szeretnék mondani munkatársamnak, **Nagymáthé Dénes**nek, a modern játékfejlesztéssel kapcsolatos tanácsaiért, ötleteiért.

Köszönetet szeretnék mondani munkatársamnak, **Svantner Dávid**nak, a modellezéshez, a 3ds max 7 haladó szintű használatához, a modellekhez és a textúrákhoz adott ötleteiért és tanácsaiért.

Köszönetet szeretnék mondani mindazon egri főiskolai hallgatóknak, akik figyelemmel kísérték a modern játékfejlesztésről, a grafikus motorról és grafikus algoritmusokról szóló előadásaimat és az **Eszterházy Károly Főiskola Matematikai és Informatikai Intézet**ének a felkérést.

Köszönetet szeretnék mondani **Jakab Roland Gábor**nak az irodalomjegyzék szerkesztésében való szíves segítségéért.