Tartalomjegyzék

1. Bevezetés 1

2. A GPU programozása 2
2.1. A Rendszer architektúrája . . . 2
2.2. A GPU architektúrája . . . 2
2.3. Hogyan optimalizáljuk az alkalmazásunkat . . . 6
2.3.1. Pontos mérések készítése . . . 6
2.3.2. A szűk keresztmetszet megtalálása . . . 7
2.3.3. A szűk keresztmetszet a CPU . . . 9
2.3.4. A szűk keresztmetszet a GPU . . . 10
2.4. Általános GPU teljesítményjavító tippek, trükkök . . . 10
2.4.1. Tippek rövid listája . . . 11
2.4.2. „Batch"-ek . . . 13
2.4.3. Vertex shaderek . . . 13
2.4.4. Shaderek vagy árnyalók . . . 13
2.4.5. Textúrázás . . . 18
2.4.6. Teljesítmény . . . 19
2.4.7. Élsimítás . . . 21
2.5. Extra képességekhez tartozó tippek, trükkök . . . 21
2.5.1. A Shader Modell 3.0 . . . 21
2.5.2. sRGB kódolás . . . 24
2.5.3. Lebegőpontos textúrák . . . 25
2.5.4. Több pufferbe renderelés (*MRT*) . . . 25
2.5.5. Vertex textúrázás . . . 26
2.5.6. Általános teljesítmény tanácsok . . . 26
2.5.7. Normálvektor textúrák . . . 27
2.5.8. Hardveres árnyék térképek . . . 28

3. A GPU programozása magasszintű nyelvekkel 28
3.1. A programozhatóság szükségessége . . . 28
3.2. A GPU programozási nyelvek és a grafikus futószalag . . . 31
3.3. Árnyalók fordítása . . . 32
3.4. A nyelv szintaxisa . . . 33
3.4.1. Adattípusok . . . 33
3.4.2. Névtúlterhelés . . . 34
3.4.3. Konstansok . . . 34
3.4.4. Natív mátrixműveletek . . . 35
3.4.5. Új operátorok . . . 35
3.4.6. Összegzés . . . 36

4. Eredmények (*CyGNUsGL*) 37
4.1. Platformfüggetlenség 37
4.2. Nem grafikai célú eszközök 37
4.3. Lineáris matematikai eszközök 38
4.4. A grafikai segédeszközök 40
4.5. Gyakolati alkalmazás 42
4.6. Továbbfejlesztési lehetőségek 42
4.7. Összegzés 43

5. Köszönetnyilvánítás 45

Ábrák jegyzéke

1. *Unreal Engine 3 Technical Demo* 2
2. *A PC-s rendszer architektúrája* 3
3. *Az NV40-es GPU architektúrája* 4
4. *A vertex processzor* 5
5. *A pixel processzor* 6
6. *A GPU általános funkcionalális felépítése* 8
7. *A szűk keresztmetszet keresése* 10
8. *Szűrők balról jobbra: mipmap nélküli „nearest", trilineáris, anizotróp* 19
9. *Élsimítás: balra élsimítás nélkül, jobbra élsimítással* 21
10. *A magas szintű árnyaló nyelvek kialakulása* 29
11. *A GPU-k fejlődése* 30
12. *A GPU-k programozási nyelvei* 30
13. *Shaderek a GPU-ban* 31
14. *Shaderek a különféle API-kban* 32
15. *Példa shaderek* 36

Táblázatok jegyzéke

1. *Pixel shaderek képességeinek összehasonlítása* 22
2. *Vertex shaderek képességeinek összehasonlítása* 23

1. Bevezetés

A dedikált PC-s grafikus hardverek megjelenése előtt az első 3D-s játékok CPU alapú szoftveres megjelenítést használtak. Ebből a korszakból származnak John Carmack úttörő programozási munkái, mint a *Doom* és *Quake*. A lassú CPU-k és az alacsony felbontás ellenére a '90-es évek közepe vízválasztó volt a grafika és játékok számára. Új vizuális effektek majdnem havonta jelentek meg, mint például a *Quake* lighmap-ei – megvilágítási térképei – és árnyékai, vagy az *Unreal* színezett fényei és volumetrikus ködje. Ebben az időszakban jelentek meg a fixed-function – rögzített funkciójú – 3D gyorsítókártyák, amelyek még nem voltak programozhatóak, így a 3D-s játékok látványvilágai nem tértek el jelentősen.

Ma, a teljesen programozható GPU-k – grafikus feldolgozó egységek – korát éljük, amelyek a 10 évvel ezelőtti grafikus teljesítmény több ezerszeresére képesek. Egyesítve a hihetetlen párhuzamos számítási kapacitást a modern magas szintű programozási nyelvekkel, a mai programozható GPU-kal megcsinálhatunk *bármit*, ha találunk egy megfelelő algoritmust az ötleteink kifejezésére.

Nézzük meg, hogy egy grafikai programozó számára milyen erőforrások érhetők el. Először is hozzáférünk a GPU-hoz, amely másodpercenként lebegőpontos műveletek tízmilliárdjainak végrehajtására képes. Ha a problémát a pixelek és vertexek birodalmába át tudjuk helyezni, akkor hasznosíthatjuk a GPU-k erejét. Másodszor rendelkezésünkre áll egy vagy több CPU, a rendszer általános célú számoló egységei. A CPU parancsokat küld a GPU-nak, kezeli az erőforrásokat és kapcsolatot tart a külvilággal. Végül rendelkezünk „művészi tartalommal" is – textúrákkal, modellekkel és más multimédiás adatokkal, amelyeket a GPU egyesíteni, szűrni és procedurálisan módosítani képes a renderelés során.

A GPU-k elég gyorsak és flexibilisek, hogy egy objektumot többször lerendereljenek, egy jelenetet szétszedjenek elemeire – megvilágításra, árnyékolásra, visszaverődésekre, utófeldolgozó effektekre és így tovább. Bár a vizuális valószerűség megvalósítható GPU-kon (1. ábra), nem ez az egyetlen lehetőség, a nem fotorealisztikus renderelési technikák is elérhetőek, mint a „cell shading", túlzott „motion blur" és a „light bloom" vagy az egyéb gyakori Hollywood-i filmeffektek. Nem csak kifejezetten grafikai célra lehet a GPU-kat használni, hanem például ütközés detektálásra, fizikai számításokra, képfeldolgozásra és numerikus módszerekhez is.

A célom egy olyan nyílt forrású többplatformos és több fordítót is támogató C++ nyelvű keretrendszer (*CyGNUsGL*) megalkotása volt, amely a grafikai, és akár GPU-t hasznosító alkalmazások fejlesztését is támogatja, egyszerűsíti.

1. ábra. *Unreal Engine 3 Technical Demo*

2. A GPU programozása

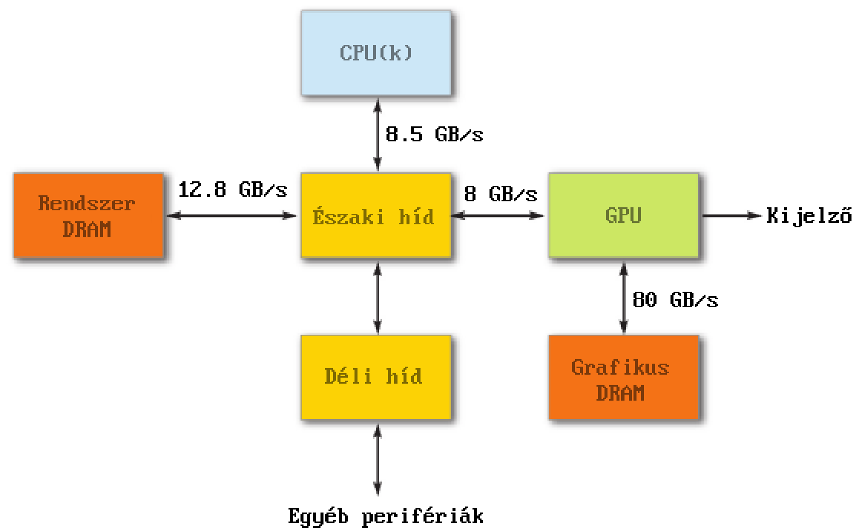
Ez a fejezet egy útmutató, amely a GPU és a grafikus API programozását ismerteti részletesebben, a minél jobb futási tejlesítmény elérése érdekében.

2.1. A Rendszer architektúrája

Mindenek előtt tisztázni kell, hogy a GPU hol is foglal helyet egy korszerű számítógépben. A 2. ábrán látható egy mai számítógép blokkdiagrammja, feltüntetve rajta, hogy hol milyen sávszélességgel haladhatnak az adatok. Az északi híd az alaplapon található, ez biztosítja a memória a CPU és a GPU közötti adatok áramlását. A memória ma 2x64 bites (*dual channel*) konfigurációban a leggyakoribb. A GPU az AGP/PCIE (*PCI-Express*) buszon keresztül áll kapcsolatban a rendszerrel.

2.2. A GPU architektúrája

A pontos részletek előtt tekintsük át egy mai modern GPU architektúrális felépítését, a CPU-tól érkező adatoktól a képernyőre kirajzolt képpontokig. A 3. ábrán a GeForce 6 sorozat architektúrája látható. Először a parancsok, a textúrák és vertex adatok érkeznek meg a gazda CPU-tól vagy egy közös használatú rendszermemória területre, vagy a GPU lokális memóriájába. Az utasítások folyamát a CPU biztosítja, amelyek inicializálnak, állapotot módosítanak. A CPU renderelési utasításokat is küld, amelyek hivatkoznak a vertex és textúra adatokra. Az utasítások megérkezése után, a GPU azokat



2. ábra. *A PC-s rendszer architektúrája*

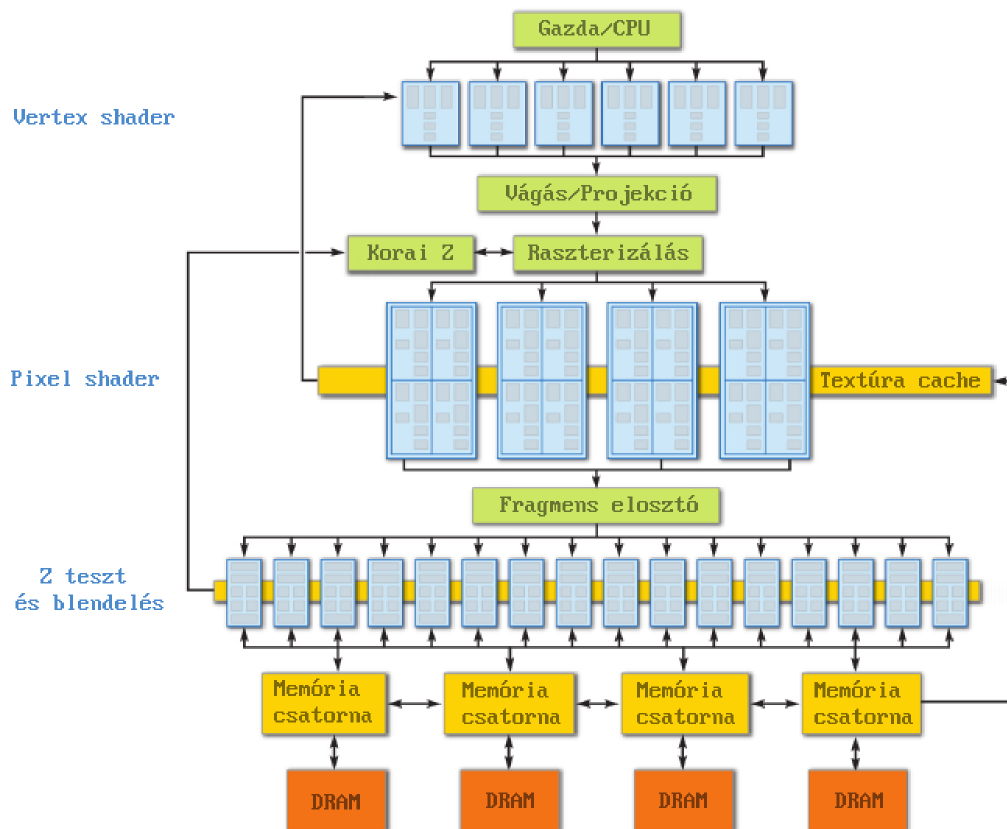
a vertexeket olvassa be, amelyekre az adott renderelési utasítás során szükség lesz.

A vertex processzor (szaknyelven „vertex shader”), minden egyes vertex esetén egy felhasználói program futtatására ad lehetőséget. Minden vertexművelet 32 bites lebegőpontos pontossággal történik. A vertex shaderben lehetőség van textúrából való mintavételezésre is, ezért a vertex és pixel shaderek osztott textúra cache-el rendelkeznek. Ezeken túl van még a vertex processzor előtt és után is egy cache, amely a számítási és memóriaolvasási igényeket csökkenti. Ez pontosabban azt jelenti, hogy, ha kétszer egyazon indexű vertexre történik hivatkozás akkor arra már nem fut le újra a vertex program, hanem a cache-ből felhasználódik a korábban kiszámolt érték. A vertex processzor blokkdiagrammja a 4. ábrán látható.

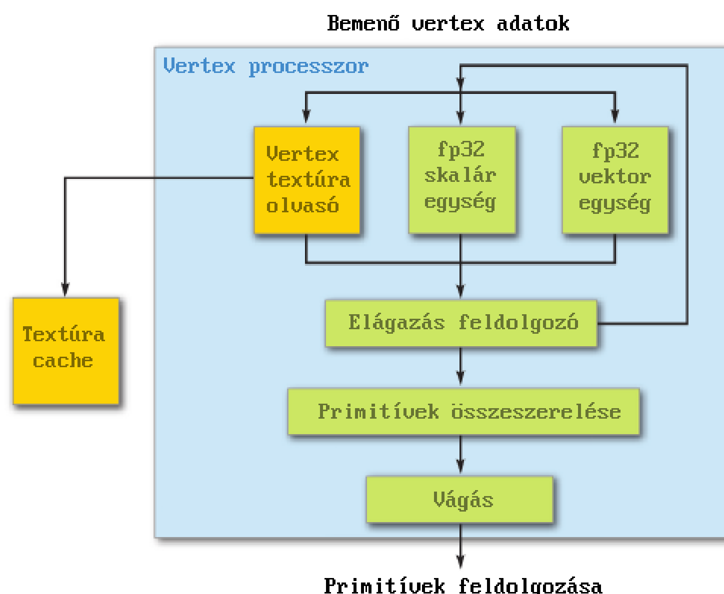
A vertexek ezután primitív csoportokká szerveződnek, pontokká, vonalakká, háromszögekké. Itt primitívenkénti műveletek hajtódnak rajtuk végre, eltávolítódnak azok amelyek egyáltalán nem látszanak, elvágódnak azok amelyek metszik a látómezőt, és előkészítődnek a raszterizálásra.

A GPU a raszterizáláskor kiszámítja, az egyes primitívekhez tartozó pixeleket, és felhasználja az „early Z” – korai Z – egységet a mélység alapján takarásban lévő pixelek gyors és korai kihagyása érdekében. Ha minden teszten átjut a pixel, akkor szín, mélység és egyéb hozzárendelt paraméter interpolálódik hozzá, amelyekkel a pixel processzor fog dolgozni.

A pixel processzor (szaknyelven „pixel shader”) minden pixelre végrehajt



3. ábra. *Az NV40-es GPU architektúrája*



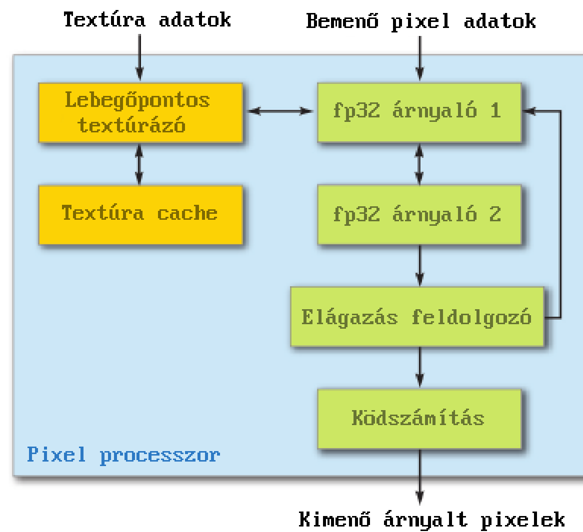
4. ábra. *A vertex processzor*

egy felhasználói programot, blokkdiagrammja az 5. ábrán látható. Hasonlóan a vertex shaderhez itt is a textúra cache segít a memória olvasási igényt csökkenteni. A pixel processzorok 2x2-es blokkokba vannak szervezve. A pixel processzor textúrázó részegységét használja textúrák olvasására, amely opcionális szűrési feladatot is ellát. A textúrázó sok textúraformátum kezelésére képes (1, 2, 4 komponensű, 1, 2, bájtos egész és 2, 4 bájtos lebegőpontos formátumok többféle kombinációját). A szűrés lehet bilineáris, trilineáris vagy anizotróp. Minden pixel shadernek visszaadott adat *fp16* – 16 bites lebegőpontos – vagy *fp32* – 32 bites lebegőpontos – pontosságú.

A GPU-kban több vertex és pixel processzor dolgozik egyszerre párhuzamosan, ezeket szokás folyamfeldolgozó processzornak (szaknyelven „stream-processzor") is nevezni. Minden stream-processzor rendelkezik legalább egy vektor, egy skalárfeldolgozó és egy textúrázó részegységgel is.

A pixelek végső állomása a kép puffer. De előtte kép puffer műveletek kerülnek végrehajtására rajtuk. Konkrétan az alpha teszt, a stencil teszt, a Z teszt, a köd blendelése és a szín pufferbe való alfa blendelés is, illetve a végeredményként előálló szín tárolása, amely legvégül a kijelzőn jelenik meg.

A memóriarendszer partíciókra ún. csatornákra van osztva. Így a memóriamodulok párhuzamosan működtethetőek, vagyis nagyobb memória sávszélesség érhető el velük. Ma egy modern GPU-ban legalább 4x64 bites memóriavezérlő van.



5. ábra. *A pixel processzor*

2.3. Hogyan optimalizáljuk az alkalmazásunkat

Ez az alfejezet betekintést nyújt, a grafikus alkalmazások szűk keresztmetszeteinek megtalálásába és azok eltávolításába.

2.3.1. Pontos mérések készítése

Több megfelelő eszköz létezik teljesímény mérésére (például az *NVPerfHUD* (`http://developer.nvidia.com/object/nvperfhud_home.html`)). A pontos méréshez a következőknek kell teljesülniük:

- **Ellenőrizni kell, hogy az alkalmazás tisztán fut-e.** Például futás közben nem generál-e hibákat vagy figyelmeztetéseket.
- **Biztosítani kell, hogy a teszt környezet alkalmas legyen.** Az optimalizált alkalmazást kell mérnünk. Illetve az alkalmazásunk által használt egyéb szoftverből is az optimalizáltat kell mérés során használni.
- **Meg kell győződni arról, hogy a megfelelő képernyő tulajdonságok vannak-e beállítva.** Az anizotróp szűrés és az élsimítás befolyásolja a teljesítményt.
- **Ki kell kapcsolni a vertikális szinkronizációt.** Ez biztosítja, hogy a képfrissítés ne legyen limitálva a monitor képfrissítési gyakorisága

által.

- **A célhardveren kell futtatni.** Más hardveren más lehet az alkalmazás szűk keresztmetszete.

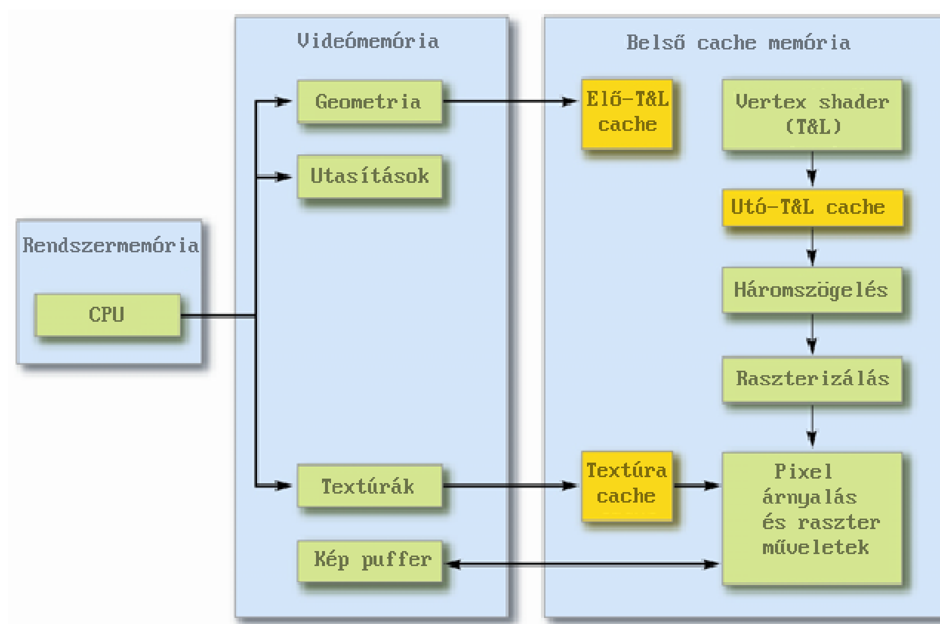
2.3.2. A szűk keresztmetszet megtalálása

- Tételezzük fel, hogy felismerjük, hogy az alkalmazásunk lassú futását. Most meg kell találni a szűk keresztmetszet. A szűk keresztmetszet általában a látványtól függöen változik, azonban egy kép renderelése során is változhat, ezzel is bonyolítva a keresést. Így a szűk keresztmetszet megtalálása a gyakoribb korlátozó tényezők feltárását jelenti. Ezt a korlátot eltávolítva, a legkomolyabb teljesítménybeli javulás tapasztalható.

  Ideális esetben nem lenne szűk keresztmetszet – ha a CPU, AGP/PCIE busz és a GPU csővezetékei azonos terhelés alatt lennének (lásd 6. ábra). Sajnos ez az eset lehetetlen egy valódi alkalmazásban – a gyakorlatban valami mindig visszafogja a teljesítményt. A szűk keresztmetszet vagy a CPU vagy a GPU oldalán van. Ha a GPU csak 1 ezredmásodpercre is felszabadul (üresjárat), akkor az alkalmazás, ha nem is teljesen de részlegesen CPU limitált. A GPU teljesítmény javítása egyszerűen a GPU üresjárati idejének növelését jelenti.

- Alapvető tesztek. Néhány nagyon egyszerű teszttel be tudjuk azonosítani az alkalmazásban lévő szűk keresztmetszet. Semmi különleges eszköz vagy vezérlőprogram nem kell hozzá, így valóban érdemes ezekkel kezdeni az analízist.

  - **Szüntessünk meg minden fájl elérést.** Bármilyen I/O művelet visszavetheti a képfrissítést. Egyszerű észrevenni – csak figyelni kell a számítógép HDD ledjét, vagy használni valamilyen HDD teljesítmény monitorozó programot. Ne feledjük el, hogy a virtuális memória lapozása miatt is létrejöhet fájl I/O, ha az alkalmazásunk túl sok memóriát használ.
  - **A teszteket ugyanazon a gépen végezzük el, csak más CPU órajelen.** A mai CPU-k órajele már dinamikusan változtatható, akár az operációs rendszerből is, így egyetlen gépen elvégezhető többféle mérés. Ha a lassabb CPU órajelével lineárisan csökken az alkalmazás teljesítménye, akkor *CPU limitált.*
  - **Csökkentsük le a GPU órajelét.** Többféle program létezik, amellyel a GPU órajelét és memóriájának órajelét változtatni lehet. Ha a lassabb GPU órajelével lineárisan csökken az alkalmazás



6. ábra. *A GPU általános funkcionalális felépítése*

teljesítménye, akkor a vertex shader, a raszterizáló vagy a pixel shader a korlátozó (*shader limitált*).

– **Csökkentsük a GPU memóriájának órajelét.** Ha a lassabb memória befolyásolja a teljesítményt, akkor vagy a textúra vagy a kép puffer sávszélessége a korlátozó (*GPU sávszélesség limitált*).

Általánosságban a CPU sebességét, a GPU órajelét és a GPU memória órajelét változtatva könnyen kideríthető, hogy hol van a szűk keresztmetszet, a CPU-ban vagy a GPU-ban. Ha a CPU órajelét csökkentjük n százalékkal, és a képfrissítés is n százalékkal csökken, akkor az alkalmazás *CPU limitált.* Ha a GPU és memóriájának órajelét csökkentjük n százalékkal, és a képfrissítés is n százalékkal csökken, akkor az alkalmazás *GPU limitált.*

- Használhatunk más eszközöket is teljesítménymérésre:

  – Különféle „profiler”-eket használhatunk a CPU limitáltság kiderítésére:

    * *Intel VTune* (`http://www.intel.com/software/products/vtune/`)

    * *AMD CodeAnalyst* (`http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html`)
  - OpenGL „profiler”-ek: *gDEBugger*, *BuGLle*, *GLIntercept*.
  - Minden az egyben eszközök: *NVPerfHUD*, *NVPerfKit*, *ATIPerformanceDashboard*.

2.3.3. A szűk keresztmetszet a CPU

Ha egy alkalmazás CPU limitált, akkor „profiler”-el meg kell tudni, hogy mi használ sok CPU időt. A következő modulok rendszerint sok CPU időt igényelnek:

- Az alkalmazás (maga a végrehajtható kód, a hozzátartozó könyvtárakkal).
- A GPU eszközmeghajtó (ismertebb nevén a „driver”).
- A használt grafikus API felhasználó-oldali kliense (OpenGL ICD, DirectX Runtime).

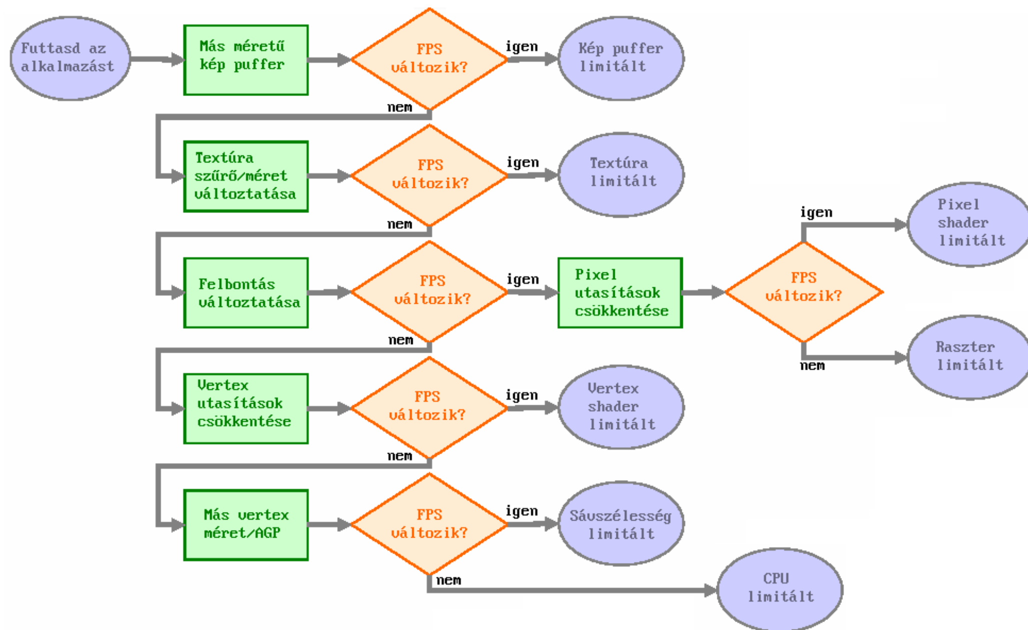
Ebben a szakaszban a fő célunk, hogy a CPU terheltsége lecsökkenjen olyan szintre, hogy ne az korlátozza az alkalmazás futását, így fontos tudni, hogy mi igényel sok CPU időt. Az általános tanácsok vannak érvényben: válasszunk algoritmikus optimalizációt a kód közvetlen alacsony szintű optimalizálása helyett.

Ha a grafikus API kliens oldalán fogy el rengeteg CPU ciklus, az azt jelzi, hogy túl sokszor hívódik meg az API. Ki kell szűrni a redundáns és nem szükséges API hívásokat az alkalmazásból, ill. csökkenteni kell a rajzolási hívásokat úgy, hogy egyszerre több háromszöget rajzolunk ki, és kevesebb állapotváltást végzünk.

További részletek a [9], [7] prezentációkban.

Más jellegű lassító tényezők is lehetnek:

- Ha a CPU-val a GPU adatait akarjuk írni/olvasni, akkor a CPU és a GPU között szinkronizáció alakul ki, vagyis a CPU-nak várni kell a GPU-ra, hogy felszabaduljon. Ezen kívül a GPU memóriájának elérése a CPU-val szintén sok időbe telhet.
- Ha az alkalmazás a CPU-n különféle láthatósági teszteket végez, hogy ezáltal tehermentesítse a GPU-t. Ilyenkor probléma lehet, ha túl sok ilyen teszt van és egy teszt kevés háromszöget spórol meg.



7. ábra. *A szűk keresztmetszet keresése*

- Ha tehermentesíteni tudjuk a CPU-t, úgy hogy még több munkát bízunk a GPU-ra, akkor tegyük ezt meg.
- Használjuk a shadereket, hogy nagyobb csoportokban dolgozzuk fel a geometriát és csökkentsük a meghajtóprogram terhelését. Például összevonhatunk két anyagmintát egy shaderbe, így fele annyi rajzolási hívással terheljük a rendszert.

2.3.4. A szűk keresztmetszet a GPU

A GPU-k mélyen futószalagszerű architektúrájúak. Ha a GPU a szűk keresztmetszet, akkor ki kell találni, hogy a csővezeték mely része korlátozza a sebességet. A gyors keresést a 7. ábrán látható algoritmus segítheti.

További információk Cem Cebenoyan és Matthias Wloka [10] prezentációjában. A következő alfejezetben bővebben lesz szó a GPU teljesítményének javításáról.

2.4. Általános GPU teljesítményjavító tippek, trükkök

Ez az alfejezet ismerteti a legfőbb teljesítménynövelő tippeket, amelyek optimális GPU teljesítmény eléréséhez segítenek. A tippek a grafikus futószalag

szerkezetéhez igazodva tagolódnak, továbbá a sorrendjük durván fontossági sorrend is.

2.4.1. Tippek rövid listája

Ha korrektül használjuk, a mai GPU-k nagyon komoly teljesítménnyel rendelkeznek.

- A rossz „batch"-elés – rajzolásonkénti állapotváltás – szűk CPU keresztmetszethez vezet. Mivel a túl sok állapotváltás és az ezekhez tartozó validálások sok CPU időt emésztenek fel.
  - Használjunk kevesebb „batch"-et:
    - Használjunk textúra atlaszokat, hogy elkerüljük a textúra cseréket. `http://developer.nvidia.com/object/nv_texture_tools.html`
    - Használjuk az „Instancing API"-t – egyazon modell más paraméterekkel történő többszöri rajzolása –, hogy elkerüljük az egyedenkénti állapotváltásokat.
- A vertex shader szűk keresztmetszet lehet.
  - Használjunk indexelt primitíveket.
    - Optimalizáljuk a modelleket, a GPU T&L cache hatékony működéséhez. Erre van többféle eszköz is (*ATI:Tottle*, `D3DX:Optimize()`, *NVTriStrip* (`http://developer.nvidia.com/object/nvtristrip_library.html`))
- A pixel shader szűk keresztmetszet lehet.
  - Használjuk a lehető legalacsonyabb verziójú pixel shader profilt.
    - Amikor shadert fejlesztünk, rendben van, hogy a magasabb verzióját használjuk. Először működjön, azután keressünk olyan optimaizálási lehetőséget, hogy csökkenthessük a pixel shader verzióját.
  - Ha szükség van *ps.2.** funkcionalitásra, haszáljuk a *ps.2.a* profilt.
  - Használjuk a legkisebb precizitású adattípust, amely még alkalmazható.
    - Használjuk a `half`-ot a `float` helyett.
  - Használjuk a `half` típust mindehol, ahol csak tudjuk:

    * Változó paraméterekhez (`varying`)
    * Állandó paraméterekhez (`uniform`)
    * Változókhoz
    * Konstansokhoz
  - Egyensúlyozzuk ki a vertex és pixel shadereket (a *G80* ezt hardveresen megoldja).
  - Helyezzünk át linearizálható számítást a vertex shader-be a pixel shader-ből, ha az korlátoz.
  - Ne használjunk állandó paramétereket (`uniform`) konstansnak, ha azok nem változnak a pixel shader élettartama alatt.
  - Keressünk lehetőségeket, hogy számítást spóroljunk algebra használatával.
  - Helyettesítsük a komplex számításokat textúrából történő olvasással.
    * Pixelenkénti fényvisszaverődési együttható.
    * Pszeudóvéletlen számok, zajok.
    * De a `sincos`, `log`, `exp` natív utasítások, és nem éri meg ezeket textúraolvasással helyettesíteni.

- A textúrázás szűk keresztmetszet lehet.
  - Használjunk „mipmap”-eket.
  - Használjuk a trilineáris és anizotróp szűrést okosan.
    * Nem a leggyorsabb textúraszűrők, csak akkor érdemes használni őket, ha tényleg szükséges.
    * Az anizotróp szűrés szintjét a textúra komplexitásához célszerű igazítani. Ebben segít a következő eszköz: `http://developer.nvidia.com/object/nv_texture_tools.html`
    * Ha a textúra zajos, vagy nagy frekvenciás, akkor anizotróp szűrést használni célravezető.

- A raszterizálás a szűk keresztmetszet.
  - Duplázott sebességű csak Z és stencil renderelés.
  - Korai Z (early Z) optimalizáció.

- Élsimítás.
  - Hogyan hasznosítsuk az élsimítást.

2.4.2. „Batch”-ek

Használjunk kevesebb „batch”-et. A „batch” azt jelenti, hogy olyan sok geometriát kell összecsoportosítani, ami egyetlen API hívással még kirajzolható, ahelyett, hogy (legroszabb esetben) egy API hívással egy háromszöget. Minden API hívásnak van egy kis „overhead”-je a GPU meghajtóprogramjában. A legjobban úgy csökkenthetjük ezt az „overhead”-et, hogy a lehető legkevesebbszer hivjuk az API-t. Más szavakkal, csökkentsük a rajzolási hívások számát, úgy, hogy néhány ezer háromszöget rajzolj ki egyetlen hívással. Kisebb számú nagy „batch”-et használva jelentősen növelhetjük a teljesítményt. Amint a GPU-k, még erösebbé válnak, a hatékony „batch” használat még fontosabb lesz, hogy optimális renderelési sebességet érjünk el.

2.4.3. Vertex shaderek

Rajzoláshoz használjunk indexelt primitíveket. Indexelt primitívek használata esetén a GPU haszálhatja a saját „post-T&L” gyorsítótárát. Ha úgy látja, hogy egy vertexet már transzformált, akkor nem transzformálja újra, hanem egyszerűen felhasználja a gyorstárbeli eredményét.

DirectX-ben használhatjuk a `ID3DXMesh` osztály `Optimize()` metódusát, vagy az *NVTriStrip* segédprogramot, hogy optimalizáljuk a modelleinket a GPU „post-T&L” gyorsítótárához. Így a legoptimálisabb esetben pontosan annyiszor fut le a vertex shader, ahány vertex van a modellben.

2.4.4. Shaderek vagy árnyalók

A magas szintű árnyaló nyelvek hatékony és rugalmas mechanizmust biztosítanak, hogy a shader írása egyszerű legyen. Sajnos ez azt jelenti, hogy lassú shaderek írása egyszerűbb mint valaha. A következő tippek segítenek elkerülni a nem hatékony shaderek írását. Emellett szó lesz arról is, hogy hogyan használható ki teljesen a GPU számítási teljesítménye. Helyesen használva, egy csúcs GPU több mint 40 műveletet tud elvégezni egy órajel ciklus alatt.

- **Használjuk a lehető legalacsonyabb verziójú működő pixel shader profilt.** Például, ha egy egyszerű textúra olvasást és egyszerű interpolációt végzünk egy textúrán, amely komponensenként csak 8 bites, nem szükséges *ps.2.0* vagy magasabb verziójú pixel shader profilt használnunk.
- **Használjuk a legkisebb precizitású működö adattípust.** A másik faktor, amely befolyásolja a teljesítményt és a minőséget az a műveletekhez használt adatok pontossága. A mai GPU-k 32 bites (`float`) és

16 bites (`half`) lebegőpontos formátumokat és egy 12 bites (`fixed`) fixpontos formátumot támogatnak. A `float` típus az IEEE 754-es szabványhoz hasonló (`s23e8` - 1 bit előjel, 23 bit mantissza, 8 bit exponens). A `half` típus szintén IEEE szerű (`s10e5` - 1 bit előjel, 10 bit mantissza, 8 bit exponens). A 12 bites `fixed` típus a [-2,2) intervallumot fedi le, és Direct3D esetén csak a *ps.1.x* profilokban, OpenGL esetén az *NV_fragment_program* kiterjesztéssel használható. A különböző típusok teljesítménye változó a precizitás függvényében:

- A `fixed` típus a leggyorsabb és alacsony precizitású számításokra használható, mint a színszámítás.
- Ha lebegőpontos precizitásra van szükségünk, akkor a `half` típus használatával jobb teljesítmény érhető el, mint a `float` típussal. A `half` típus okos használata megduplázhatja a képfrissítési sebességet. A pixelek 99%-ában, csak a legkevésbé szignifikáns bitben (LSB) tér el az eredmény a `float` típus alkalmazásához képest a legtöbb alkalmazás esetén.
- Ha a legnagyobb pontosság kell, a `float` típus használandó.

Az OpenGL *ARB_fragment_program* kiterjesztés esetén, használjuk az *ARB_precision_hint_fastest* opciót, ha a végrehajtási időt szeretnénk minimalizálni, esetleges csökkentett pontossággal, vagy használjuk az *NV_fragment_program_option* opciót, ha utasítás szinten akarjuk a pontosságot vezérelni. Sok szín alapú művelet elvégezhető minőségveszteség nélkül a `fixed` vagy a `half` típusokkal (például, egy `tex2D*diffuse` művelet). Általában vektorok normalizálásához is bőven elegendő a `half` típus. Egyes GPU-k, a 16 bites lebegőpontos normalizálást 1 utasítással (`NRMH`) el tudják végezni.

- **Algebra használatával számításokat egyszerűsíthetünk.** Miutan a shader működik, nézzük át a számítást és próbáljuk kiegyszerűsíteni matematikai tulajdonságok alapján. Ez különösen igaz a több shaderben használt túl általános könyvtári függvényekre. Például:

  - Az általános gömb térkép (sphere mapping) vetítés gyakran a következő módon van kifejezve:

$$\mathbf{p = sqrt(Rx^2 + Ry^2 + (Rz + 1)^2)}$$

    Ez átalakítható:

$$\mathbf{p = sqrt(Rx^2 + Ry^2 + Rz^2 + 2Rz + 1)}$$

Ha tudjuk, hogy a visszaverődő vektor egység hosszú, akkor az első három term összege 1.0. Ekkor a kifejezés átírható így:

$$\mathbf{p} = \mathbf{sqrt}(\mathbf{2} * (\mathbf{Rz} + \mathbf{1})) = \mathbf{1.414} * \mathbf{sqrt}(\mathbf{Rz} + \mathbf{1})$$

- Sokkal hatékonyabban számítható ki a `dot(normalize(N), normalize(L))` kifejezés.
  * Rendszerint így van kiszámítva: $(\mathbf{N}/|\mathbf{N}|)\mathbf{dot}(\mathbf{L}/|\mathbf{L}|)$, amelyhez két költséges reciprokgyökvonás (`RSQ`) szükséges.
  * Kicsit egyszerűsítve kapjuk:
    $(\mathbf{N}/|\mathbf{N}|)\mathbf{dot}(\mathbf{L}/|\mathbf{L}|)$
    $= (\mathbf{NdotL})/(|\mathbf{N}| * |\mathbf{L}|)$
    $= (\mathbf{NdotL})/(\mathbf{sqrt}((\mathbf{NdotN}) * (\mathbf{LdotL}))$
    $= (\mathbf{NdotL}) * \mathbf{rsq}((\mathbf{NdotN}) * (\mathbf{LdotL}))$
    Amelyhez csak 1 `RSQ` művelet kell.

- **Ne tegyünk vektort több interpolálandó vektor skalár komponenseibe.** A számítást túl sok információval megtűzdelve, a fordító számára megnehezítjük a kód hatékony optimalizálását. Például, ha leküldünk egy tangens mátrixot (3 darab 3 komponensű vektor – bázis), akkor a nézőpont irányába mutató vektort, ne csomagoljuk a 3 w komponensbe. A betömörítés és a kitömörítes is 3-3 utasítás! Ez a hiba illusztrálva:

```
// példa hibás megadásra
tangent = float4( tangentVec, viewVec.x );
binormal = float4( binormalVec, viewVec.y );
normal = float4( normalVec, viewVec.z );
```

  Inkább tegyük a `viewVec`-et egy negyedik interpolálandó vektorba.

- **Ne írjunk meg túl általános könyvtári függvényt.** Több shaderben közösen használt függvények gyakran nagyon általánosan vannak megírva. Például a visszaverődés leggyakrabban a következő:

```
float3 reflect( float3 I, float3 N )
{
    return ( 2.0 * dot( I, N ) / dot( N, N ) ) * N - I;
}
```

  Így megírva, a visszaverődés vektor a beérkező és a normálvektor hosszától függetlenül kiszámítható. Azonban a shader írók gyakran legalább

a normálvektort normalizálják, a többi megvilágítási számítás miatt. Ebben az esetben egy belső szorzat, egy reciprok, és egy skalár szorzás eltávolítható a `reflect()` függvényből. Hasonló optimalizációk drasztikusan növelhetik a teljesítményt.

- **Ne számoljuk ki a normalizált vektorok hosszát.** Egy hétköznapi (és költséges) példa a túl általános könyvtári függvényekre az, hogy egy bemenő vektor hosszát lokálisan kiszámítja. Azonban a vektorok sokszor normalizálódnak a függvényhívás előtt. A fordító ezt nem veszi figyelembe, ennek az az eredménye, hogy értékes pixelenkénti műveleteket végez pusztán azért, hogy kiszámítsa az 1.0-t. Ha a könyvtári függvényünknek működnie kell a bemenő vektorok hosszától függetlenül, esetleg megéri plussz egy bemenő skalár paraméternek felvenni a vektor hosszát. Így azok a shaderek, amelyek normalizált vektorral hívják a függvényt, az 1.0 konstans értéket adhatják meg a vektor hosszaként, ha viszont nem egységhosszú vektorral hívják akkor a vektorhossz paraméterbe kiszámolhatják azt (így a hossz csak akkor kerül kiszámításra amikor szükséges).

- **Vonjuk össze a nem változó konstans paramétereket tartalmazó kifejezéseket.** Sok fejlesztő a dinamikusan megadott konstansokkal számol, hogy minél általánosabb legyen egy adott algoritmus, minél többféle paramétert tudjon állítani rajta. Ha több mint egy állandó paramétert (vagy egy állandó paramétert és egy konstanst) használunk egy kifejezésben, akkor gyakori lehetőség a konstansok összevonása, amellyel növelhető a teljesítmény (kevesebb számítást kell végezni). Például:

```
half4 main( float2 diffuse : TEXCOORD0,
            uniform sampler2D diffuseTex,
            uniform half4 g_OverbrightColor ) : COLOR0
{
    return tex2D( diffuseTex, diffuse ) *
                g_OverbrightColor * 3.0;
}
```

  A `g_OverbrightColor`-t előre megszorozhatjuk 3.0-val még a CPU-n, amellyel minden kiszámolt képen kihagyhatunk pixelenként szorzások millióit. Lehet, hogy szükséges lesz felbontani a kifejezéseket, annak érdekében, hogy minél több konstans kifejezést összevonjunk. A másik általános példa a `materialColor * lightColor` vertexenkénti kiszámítása. Mivel ez a kifejezés minden vertexre egyenlő, ezért a CPU-n

kellene kiszámítani. A mátrix inverz és transzponálás számításokat szintén a CPU-n kellene kiszámítani, és nem a GPU-n, mert csak egyszer kell kiszámítani és nem vertexenként, vagy pixelenként. Vannak a shader fordítónak paraméterei, amelyekkel szabályozható a mátrixok tárolása (sor vagy oszlopfolytonos).

- **Ne használjunk olyan állandó paramétereket, amelyek nem változnak a pixel shader élettartama során.** Sok fejlesztő az állandó paramétereket arra használja, hogy általános konstansokat adjon át mint, 0, 1 és 255. Ez a gyakorlat kerülendő. A fordító számára megnehezíti a tényleges konstansok és a shader paraméterek szétválasztását, csökkentve ezzel a teljesítményt.

- **Egyensúlyozzuk ki a vertex és pixel shadereket.** Jó teljesítmény elérése nem jelent mást, mint a szűk keresztmetszetek eltávolítását – amely valójában azt jelenti, hogy a futószalag minden elemét ki kell egynsúlyozni: a CPU-t, az AGP/PCIE buszt és a grafikus csővezeték fokozatait. A döntés néhány tényezőtől függ:

  - **Hogyan vannak az objektumok tesszellálva?** Lehet, hogy csökkenteni kell a vertex shader terhelését, ha képenként vertexek millióit dolgozza fel. Ez különösen igaz akkor, ha a használt algoritmus többmenetes.
  - **Milyen felbontást célzunk meg?** Ha elvárjuk, hogy az alkalmazásunk nagy felbontásban is fusson, akkor a legvalószínűbb, hogy a pixel shader lesz a szűk keresztmetszet. Így minél több számítást át kell helyezni a pixel shaderből a vertex shaderbe.
  - **Milyen hosszúak a pixel shadereink?** Ha komplex árnyalást valósítunk meg, a legvalószínűbb, hogy a pixel shader lesz a szűk keresztmetszet. Ha a pixel shader több mint 20 utasításból áll és a képernyő több mint felét kitölti, akkor valószínüleg a pixel shader korlátozza a képfrissítést a régi 2oo2-es hardvereken. Keressünk lehetőségeket, amellyel a pixel shaderből a vertex shaderbe helyezhetünk át számításokat. Vannak különböző shader tejlesítménymérő programok (*NVShaderPerf*), amelyekkel megtudható, hogy egy adott shader mennyi órajelciklus alatt fut le különböző GPU-kon. Azért jegyezzük meg, hogy az újabb hardverek sokkal komplexebb pixel shaderek futtatását is lehetővé teszik.

- **Ha a pixel shader korlátoz, akkor helyezzünk át linearizálható számításokat a vetex shaderbe.** A raszterizáló veszi a vertexenkénti

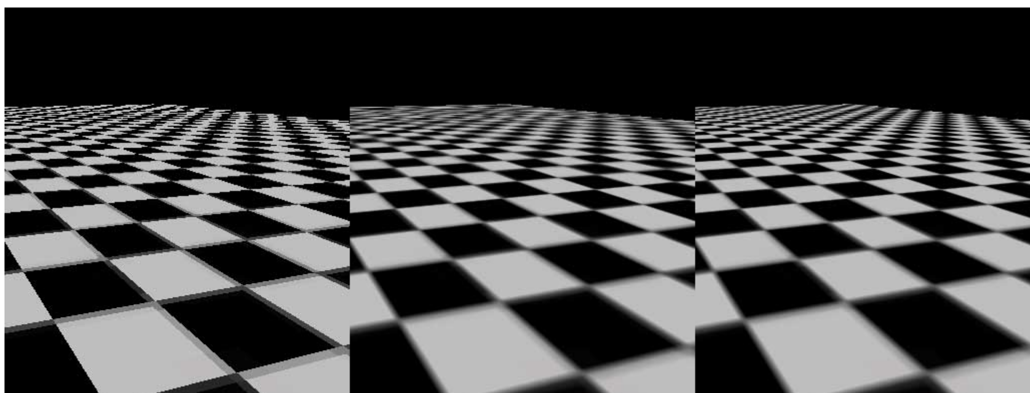
értékeket és interpolálja pixelenként, úgy, hogy közben perspektívikusan korrekt lesz. Használjuk ki ezt az interpolációt, hiszen kevesebb vertexre kell elvégezni a számítást, és a helyesen interpolált értékeket kapjuk meg a pixel shaderben. Például olyan esetben, ahol csak egy vektor iránya a fontos (`texCUBE`), ott már a vertex shaderben kiszámolható ez a vektor.

- **Használjuk a standard könyvtári `mul()` függvényt mátrix szorzásra, a saját függvényeink helyett.** Ezáltal nagyobb az esély, hogy a fordító optimálisabb kódot fog generálni.
- **Használjuk a hardveres textúra koordináta vágást (*GL_CLAMP_TO_EDGE*), ahelyett hogy a `saturate()` függvényt használnánk ilyen célra.** Ugyanis néhány GPU esetén ez nem ingyenes, és csak a magasabb shader profilokban fedhetjük el a használatukat mint utasítás módosító (`add_sat, mul_sat`).
- **Nagyobb teljesítményt érhetünk el, ha először az alacsonyabb számú interpolátort használjuk (`TEXCOORDn`).** Kezdjük a `TEXCOORD0`-val, majd a `TEXCOORD1`, `TEXCOORD2`, és így tovább.

2.4.5. Textúrázás

- **Hogy elkerüljük a kicsinyített textúrák „szikrázó" jelenségét, mindig használjunk „mipmap"-eket.** Jobb képminőséget, jobb textúra cache működést, és jobb teljesítményt fogunk tapasztalni. Ezt 33%-al több memória árán kapjuk, amely elég jó kompromisszum. A 3D-s textúrák még többet profitálnak a „mipmap"-ekből – 30% - 40%-al nagyobb teljesítményt érhetünk el „mipmap"-ek használatával.

  Amikor „mipmap"-eket generálunk ne az egyszerű box-szűrőt használjuk a kisebb „mipmap"-ek elkészítéséhez. Használjunk inkább Gaussian vagy Mitchell szűrőt, melyek több mintavételezéssel jobb minőségű képeket állítanak elő. Egy kicsit több időt áldozva az előfeldolgozásra, az alkalmazásunk jobban nézhet ki futás közben. A „mipmap"-ek előállításához az alábbi címen elérhető eszköz használata javasolt:
  `http://developer.nvidia.com/object/nv_texture_tools.html`
- **Használjuk a trilineáris és anizotróp szűrést okosan.** A trilineáris és anizotróp szűrés is javít a kép minőségén, de csökkentik a teljesítményt, ezért csak ott használjuk ahol ténylegesen szükséges. Általában olyan esetben használatosak, amikor a textúra sok nagy kontrasztú részletet tartalmaz. Ha a textúrát a néző mindig ferdén látja



8. ábra. *Szűrők balról jobbra: mipmap nélküli „nearest”, trilineáris, anizotróp*

(pl.: padló, plafon, stb.), akkor növeljük a textúra anizotróp szűrésének szintjét. A kölönféle szűrőkkel működés közben a 8. ábrán láthatóak.

- **Helyettesítsünk komplex függvényeket textúraolvasással.** A textúrákat bonyolult függvények kódolására is alkalmazhatjuk – úgy képzeljük el, mint egy többdimenziós tömböt, melyet egyszerűen tudunk indexelni. A GPU-k memória sávszélessége lehetővé teszi, hogy egy textúrából való mintavétel egy aritmetikai művelet végrehajtásának idejével összevethető legyen.

  Növelhetjük a teljesítményt, ha komplex aritmetikai utasítások sorozatát le tudjuk kódolni textúrába, azonban tartsuk észben, hogy az olyan függvények mint a `log` és `exp` mikro utasítások a *ps.2.0* és magasabb profilokba, így ezeket nem kell lekódolni textúrába az optimális teljesítményért.

2.4.6. Teljesítmény

- **Duplázott sebességű csak Z és stencil renderelés.** Ma már az összes GPU kétszeres sebességgel tud a mélység és stencil pufferekbe renderelni. Ahhoz, hogy elérjük ezt a jó tulajdonságot, a következőt kell tennünk:
  - A szín pufferbe való írást le kell tiltani.
  - Az aktív mélység és stencil puffer nem lehet élsimított.
  - Mélységmódosító utasítás nincs alkalmazva egyetlen pixelre sem (`oDepth, texm3x2depth, texdepth`).

  - Alpha teszt kikapcsolva.
  - Nincs színkulcs alkalmazva egy aktív textúrában sem.
  - Nincs felhasználói vágósík bekapcsolva.

- **A korai Z optimalizálás úgy növeli a renderelési sebességet, hogy a takarásban lévő felületeket nem hagyja kiszámolni.** Ha a takarásban lévő felületre hosszú pixel shadernek kéne lefutnia, akkor sok számítást spórol meg. Annak érdekében, hogy teljesen kihasználjuk a korai Z optimalizációt, a következőket kell betartani:

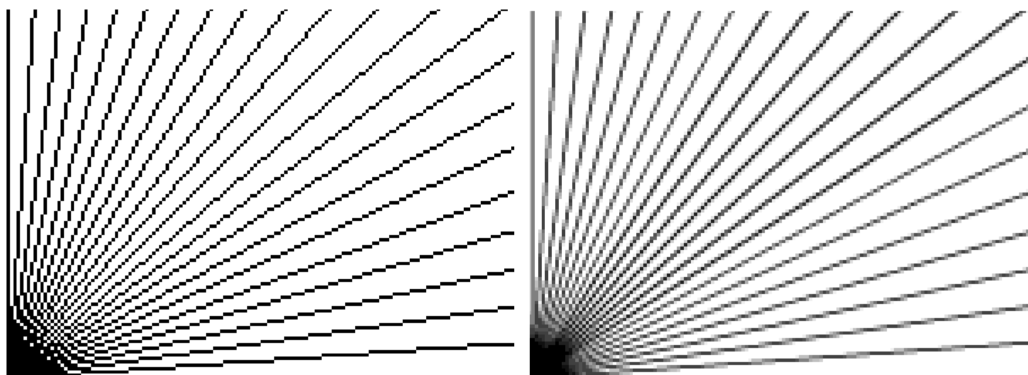
  - Ne rajzoljunk olyan háromszögeket, amelyeknek lyuk van a közepén (kerüljük az alpha tesztet és a `texkill` utasítást).
  - Ne módosítsuk a mélységi értéket a pixelek shaderben (így a GPU használhatja az interpolált értékeket).

  Ezeket a szabályokat megszegve a GPU számára hasznos adatok elvesznek, így nem tudja akalmazni a korai Z optimalizációt, amíg a mélység puffert újra nem töröljük.

- **Először a Z puffert töltsük fel.** A legjobban úgy használhatjuk ki a fentebb említett optimalizációkat, ha először a Z pufferbe renderelünk. Ez azt jelenti, hogy a kétszeres sebességgel kiszámított a látványhoz tartozó Z puffert árnyalás nélkül, mint első menet létrehozzuk. Így megkapjuk a nézőhöz legközelebbi felületeket. Majd újra rendereljük a képet csak most teljes árnyalással. A korai Z automatikusan ki fogja hagyni azokat a részeket, amelyek nem látszanak, így sok számítást spórol meg. Azonban ez a két optimalizáció veszít a hatékonyságából, ha sok kis méretű háromszöget renderelünk.

- **Memória allokáció.** Annak érdekében, hogy minimalizáljuk a videó memória veszteséget, a shaderek és renderelési pufferek legjobb allokációs stratégiája:

  1. A renderelési puffereket kell először lefoglalni.
     - Az allokáció sorrendjét szélesség alapján kell rendezni.
     - Az azonos szélességűeket használatuk gyakorisága alapján kell rendeni.
  2. Hozzuk létre a vertex és pixel shadereket.
  3. Töltsük be a textúrákat.



9. ábra. *Élsimítás: balra élsimítás nélkül, jobbra élsimítással*

2.4.7. Élsimítás

A mai GPU-k nagyon hatékony élsimító motorral rendelkeznek. Bekapcsolt élsimítással teljesítenek a legjobban – arányosan sokkal több pixelt tudnak úgy kiszámítani másodpercenként, mint élsimítás nélkül (4x-es élsimítás nem 4x lassabb, csak 35%-al, pedig 4x annyi pixelt renderel a GPU) –, így ajánlott az alkalmazást az élsimítás támogatására felkészíteni. Régebben volt egy olyan probléma, hogy az utófeldolgozó effektek nem működtek az élsimítással együtt. Ez ma már meg van oldva OpenGL-ben az *EXT_framebuffer_blit* kiterjesztéssel, DirectX-ben pedig a `StrechRect()` hívással.

Például ha az élsimítás 4x-es, akkor a 100×100 kép puffer esetén, belsőleg egy 200×200-as kép puffer jön létre. Többek között ezért nem lehet egy élsimított Z pufferhez egy nem élsimított szín puffert rendelni. A fentebb említett megoldások a pufferek közötti átméretezett másolásban segítenek, egy $n \times m$-es puffert átmásolhatunk egy $k \times l$-esbe, úgy hogy még akár lineárisan meg is szűrjük az eredményt. Az élsimítást működés közben a 9. ábrán láthatjuk.

2.5. Extra képességekhez tartozó tippek, trükkök

2.5.1. A Shader Modell 3.0

A DirectX 9 új szabványt definiált a vertex és pixel shaderek technológiájában, konkrétan a 2.0-ás és a 3.0-ás verziót. A legtöbb ma használatos GPU a shaderek 2.0-ás vagy újabb verzióját hardveresen támogatja. A shader modell 2.0 (*SM2*) olyan technológiákat is magába foglal, amelyek fejlett megvilágítást és animálást tesznek lehetővé, azonban korlátozva van a shader program hosszában és komplexitásában, amely az elérhető effektek minőségét

Pixel shader képesség	**Shader 2.0**	**Shader 3.0**
Függő textúraolvasás	*4*	*korlátlan*
Textúrázó utasítások	*32*	*korlátlan*
Utasítások száma	*32+64*	*512+*
Végrehajtható utasítások	*32+64*	*65536*
Interpolált regiszterek	*2+8*	*10*
Feltételes utasítások	*nincsenek*	*vannak*
Gradiens utasítások	*nincsenek*	*vannak*
Dinamikus elágazás	*nincs*	*van*
Dinamikus elágazás mélység	–	*24*
Indexelt bemenő regiszterek	*nincsenek*	*vannak*
Lokális regiszterek	*12*	*32*
Konstans regiszterek	*32*	*224*
Pozíció regiszter	*nincs*	*van*
Hátoldali regiszter	*nincs*	*van*
Ciklusszámláló regiszter	*nincs*	*van*
Tetszőleges permutálás	*nincs*	*van*
Interpolált szín formátum	*8 bites egész*	*32 bites lebegőpontos*
Több pufferbe renderelés	*opcionális*	*legalább 4*
Köd	*8 bites rögzített*	*shaderből 16/32 bites*

1. táblázat. *Pixel shaderek képességeinek összehasonlítása*

korlátozza. A 3.0-ás (*SM3*) több területen is fejlettebb, mind vertex mind pixel feldolgozás terén.

Az új és az azt megelőző shader modellek összehasonlítását táblázatba foglaltam. A 1. táblázat összegzi és kiemeli a pixel shader 2.0 és 3.0 közötti külömbségeket. A 2. táblázat a különböző vertex shaderek képességeinek eltéréseit mutatja.

Vertex shader képesség	Shader 2.0	Shader 3.0
Utasítások száma	*256*	*512+*
Végrehajtható utasítások	*65536*	*65535*
Feltételes utasítások	*nincsenek*	*vannak*
Lokális regiszterek	*12*	*32*
Konstans regiszterek	*256+*	*256+*
Statikus elágazások	*vannak*	*vannak*
Dinamikus elágazások	*nincsenek*	*vannak*
Dinamikus elágazás mélység	–	*24*
Vertex textúrázás	*nincs*	*van*
Textúrázók száma	–	*4*
„Instancing” támogatás	*nincs*	*szükséges*

2. táblázat. *Vertex shaderek képességeinek összehasonlítása*

- **Dinamikus elágazások**

  A 3.0-ás shader modell egyik fő újítása a dinamikus elágazások. Így lehetőség van valódi feltételek használatára és ciklusok szervezésére. Például írhatunk egy olyan vertex shadert, amely rögzített számú fényforrásokból egy ciklusban kiszámolja, hogy melyeknek van hatása az adott vertexre, ezt az információt indexelve továbbadja a pixel shadernek, amely végigiterálva csak a megfelelő fényforrásokat használja fel a végeredmény kiszámítására és dinamikusan kilép a ciklusból.

  A legtöbb fény típus csak az objektumok egyik oldalát világítja meg – a fény felé nézőt. Az új hátoldali regiszter segítségével a vertex és pixel shaderben dinamikusan kihagyható azoknak a fényforrásoknak a feldolgozása, amelyek nem a felület felé néznek. Ez jelentős időt takaríthat meg, felgyorsíta ezzel a renderelést. Hasonló teljesítményjavítás érhető el karakterek csont alapú animációinak renderelésekor, úgymint más hasonló algoritmusnál is. A dinamikus és statikus elágazásokra egy példa:

```
void Shader(
  ...
    // pixelenkénti vagy vertexenkénti
    // input paraméterek
    in float3   normal,

    // állandó paraméterek
```

```
        uniform float3  lightDirection,
        uniform bool    computeLight,
      ...
    )
    {
      ...
        // statikus elágazás, mivel csak konstanstól függ
        if( computeLight )
        {
          ...
            // dinamikus elágazás, hiszen kiszámolt
            // eredménytöl függ
            if( dot( lightDirection, normal ) )
            {
              ...
            }
          ...
        }
      ...
    }
```

- **Könnyebb kód karbantartás**

  Amint a játékok egyre bonyolultabbá válnak, gyakran létrehozzák egyazon shader több változatát, hogy azok még beleférjenek a 2.0-ás shaderek korlátozott hosszába. Ez plussz időt jelent a kód karbantartásához, a shaderek lefordításához, és szintek betöltéséhez, illetve futás közben több rendszermemóriát igényel. A 3.0-ás shaderek eltüntetik ezt a problémát és lehetővé teszik a ciklusokon és elágazásokon keresztül, hogy egyetlen megfelelően megírt pixel és vertex shaderrel a helyes végrehajtási út kerüljön lefuttatásra.

  Összegezve a 3.0-ás shader modell egy komoly lépés az egyszerű használat, a teljesítmény és a shader komplexitás irányába.

2.5.2. sRGB kódolás

Az sRGB egy olyan kódolás, amely gamma korrekciót használ, hogy nagyobb pontosságot biztosítson a nullához közel. Az sRGB a DXT textúratömörítéssel együtt is használható, így az alkalmazás profitálhat a nagyobb színhűségből és a csökkentett helyigényből.

Néhány esetben viszont a lebegőpontos formátumok használatosak, előnyeik:

- A teljes tartományra vonatkozóan nagyobb precizitás.
- Sokkal nagyobb, lineáris, dinamikus tartomány.

A legtöbb esetben az sRGB textúrák használata ajánlott a lebegőpontos típusokkal szemben, a kisebb memória és sávszélesség igény miatt.

2.5.3. Lebegőpontos textúrák

A mai GPU-k kiemelten támogatják a lebegőpontos textúrákat, azonban a támogatás még nem teljes. A korlátozások:

- A 16 bites egy komponensű textúra (`R16F`) nem támogatott, helyette a kétkomponensű (`G16R16F`) használata javasolt.
- Blendelni csak `A16B16G16R16F` pufferbe lehet, `G16R16F`-be vagy `R32F`-be nem.
- Az összes szűrés alkalmazható a `G16R16F` és `A16B16G16R16F` formátumokra.
- A 32 bites (`R32F, G32R32F, A32B32G32R32F`) formátumokra csak a „nearest” – legközelebbi pixel – szűrés alkalmazható.

2.5.4. Több pufferbe renderelés (*MRT*)

Az újabb GPU-k támogatják a több pufferbe egyszerre történő renderelést, amely azt jelenti, hogy egy pixel shader egyszerre maximum négy különböző pufferbe renderelhet egyidejűleg. A MRT-k akkor hasznosak, ha a pixel shader több mint négy lebegőpontos értéket számol ki és, azokat le is kell textúrákba tárolni.

A GPGPU-s – General-Purpose Computing on Graphics Processing Units (a GPU-k általános célrú számításokra történő felhasználása) – algoritmusok, vagy a részecske rendszerek fizikáját szimuláló algoritmus, használja a GPU-k ezen tulajdonságát.

A mai GPU-k egyidejűleg legalább négy pufferbe képesek írni, a pixel shaderben ez a négy kimeneti regiszterbe (`oC0, oC1, oC2, oC3`) történő írással tehető meg. Az MRT más GPU képességeket viszont korlátoz:

- A legfontosabb, hogy a hardveresen gyorsított élsimítás nem működik együtt a MRT-el.

- Az egyidejűleg kiválasztott kimeneti puffereknek a szélessége, magassága és bitmélysége megegyező kell, hogy legyen.
- Alpha teszteléshez az `oC0.a` érték használódik fel.
- Nagy a memória sávszélesség igénye. Például négy `A32B32G32R32F` pufferbe történő renderelés 16-szor költségesebb mint egy egyedüli `A8R8G8B8` pufferbe. Ezért minimalizálni kell az egyszerre használandó pufferek számát és bitmélységét.

2.5.5. Vertex textúrázás

A legújabb GPU-k a vertex shaderből történő textúraolvasást is lehetővé teszik, de nem tekinthetőek RAM beli konstanstárolónak. A vertex textúrák olvasása késleltetéssel jár, így a legjobb, ha a textúraolvasás és az eredményét felhasználó utasítás közé több független aritmetikai utasítás kerül, ezzel elfedve azt a bizonyos késleltetést.

A vertex textúrák nem nagy konstanstároló tömbök, inkább arra lettek kitalálva, hogy egy kevés vertexenkénti adatot biztosítsanak, nagyobb dinamizmust adva a fejlesztők kezébe. A vertex pufferbe történő renderelés – ami másolással jár – is kioptimalizálható vertex textúrák alkalmazásával, ugyanis ebben az esetben nincs szükség a textúrából a vertex pufferbe történő köztes másolásra, hanem a textúrát közvetlenül fel lehet használni a vertex programban.

A vertex textúrázásról bővebbe Simon Green [2] ír.

2.5.6. Általános teljesítmény tanácsok

Az új GPU architektúrák sok olyan bővítést tartalmaznak, amelyekkel sokkal hatékonyabbá tehetők. Itt van néhány megoldás, amely a képességek kihasználásában segíthet:

- **Használjuk az írási maszkokat és a „swizzling"-et (regiszter komponens permutációt).** Az új shader architektúra lehetővé teszi, hogy egy négy komponensű vektorműveletet több különböző végrehajtó egységgel hajtsa végre (*co-issue, dual-issue*), amely a shader kihasználását javítja. Írási maszkokkal és a „swizzling"-el a fordító felismerheti ezeket az ütemezési lehetőségeket.
- **Használjunk részleges precizitást ahol csak lehetséges.** Két ok is van, amiért a részleges precizitást érdemes használni. Először is az új GPU-k rendelkeznek egy speciális szabadon használható 16 bites lebegőpontos normalizáló egységgel, amely nagyon hatékony és

párhozamosan hajtható végre más utasításokkal. Ahhoz, hogy ezt az alkalmazásban kihasználjuk a részleges precizitást célszerű használni, ahol csak lehetséges. A korábbi architektúrákban ezt három különböző utasítással lehetett elérni. A második ok, hogy a használt regiszterek száma csökkenthető, így nagyobb teljesítmény érhető el.

- **A dinamikus elágazásokat csak elég koherens esetben használjuk.** Ahogy az már korábban említésre került, a dinamikus elágazásokkal a kód gyorsabbá és könyebben implementálhatóvá válik. Azonban a hatákony működéséhez az elágazásoknak elég koherenseknek kell lenniük (például egy közel 30x30 pixeles területen).

2.5.7. Normálvektor textúrák

Ha a normálvektor textúrák tárolására nincs elég hely az alkalmazás számára, akkor az egység hosszú tangens térbeli normálvektor tömörítése elérhető a következő módon:

```
N.z = sqrt( 1 - N.x * N.x - N.y * N.y );
```

Ez szükségteleníti az egyik komponens (`N.z`) letárolását, és 3-5 pixel shader utasításra fordul le, így a teljesítményre gyakorolt hatása attól függ, hogy mennyire párhuzamosodnak a környező utasításokkal, vagy hogy a textúra olvasás-e a szűk keresztmetszet.

Ha úgy döntünk, hogy ezt a technikát használjuk, akkor az ajánlott textúra formátum OpenGL-ben a `GL_LUMINANCE8_ALPHA8`. Ez a formátum pixelenként 16 bites és 2:1 arányú tömörítést tesz lehetővé.

Azonban jegyezzük meg, hogy ez a technika a flexibilitást rontja, hiszen csak pozitív normálvektorok kódolását teszi lehetővé. Olyan technikák, amelyek negatív normálvektort használnak (mint az objektum-térbeli normálvektor textúrázás), vagy nem egységhosszú normálvektorokkal számol, akkor ez a technika nem fog működni.

Csúcs minőségű normálvektor textúrák létrehozásához, amelyek alacsony részletességű modelleknek nagy részletességű kinézetet adhat, több eszköz is létezik (*ATINormalMapper, NVMelody* (*http://developer.nvidia.com/object/melody_home.html*)).

A normálvektor textúrák „mipmap"-eléséről bővebb Michael Toksvig [8] ír.

A normálvektor textúrák tömörítéséről bővebben Simon Green [3] írásában olvashat.

A normálvektorok normalizálásáról szóló technikai információkat Mark J. Harris [4] szolgáltathat.

2.5.8. Hardveres árnyék térképek

A GPU-k rég óta támogatják a hardveres árnyék térképeket (GeForce3 óta). A GPU-k rendelkeznek speciálisan erre a célra dedikált tranzisztorokkal, amelyek az árnyék térképek összehasonlításait és szűrését végzik. Ajánlott ennek a képességnek a használata, hiszen hatékonyabban és magasabb minőségben képes az árnyék térképeket szűrni. Mivel dedikált tranzisztorok vannak erre a célra fenntartva, ezért ha a pixel shaderben próbáljuk emulálni, akkor vagy gyengébb teljesítményt vagy gyengébb minőséget érünk el.

Bővebb információ a hardveres árnyéktérképekről az alábbi cikkekben találhatók: [1] és [5].

A perspektívikus árnyéktérképekről pedig a *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* című könyvben olvashat bővebben: [6].

3. A GPU programozása magasszintű nyelvekkel

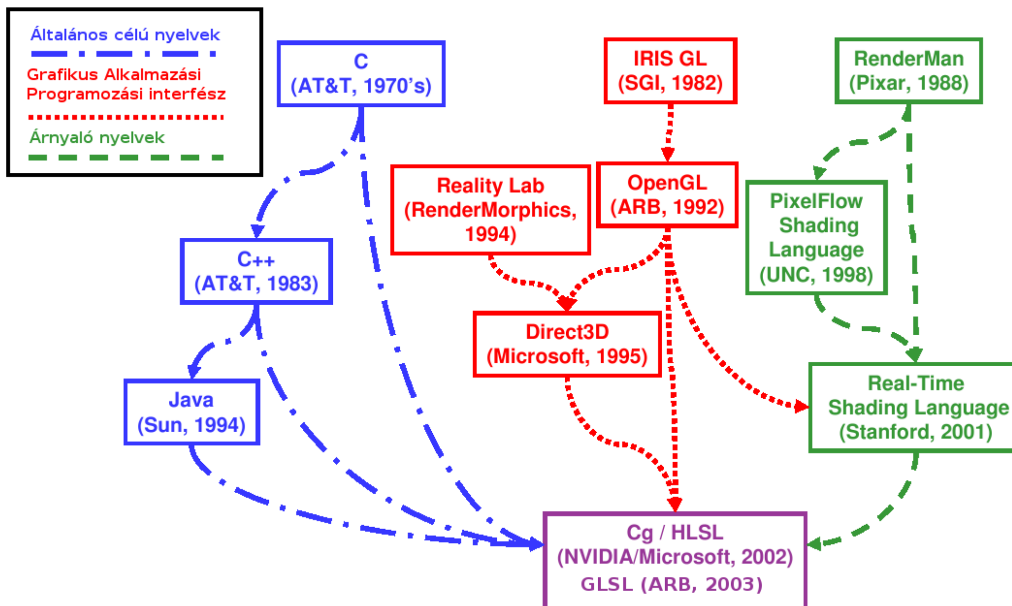
A modern GPU programozási nyelvek öröksége három forrásból ered. Először is a szintaktika és szemantika a *C* programozási nyervre van alapozva. Másodszor olyan nem valós idejű árnyaló nyelvek koncepciója mentén fejlődtek mint a RenderMan. A harmadik az, hogy a modern GPU programozási nyelvek a valós idejű megjelenítést az OpenGL vagy a Direct3D programozási interfészen keresztül érik el. A 10. ábrán látható a magas szintű árnyaló nyelvek fejlődésének menete.

A régi PC-s 3D grafikában (a GPU-k megjelenése előtt) a CPU kezelte a vertex és pixel számításokat a 3D-s látványvilág létrehozásához. Ekkoriban a grafikus hardver csak a kijelzőn megjelenő kép pixeleinek tárolásához biztosított kép puffert. A programozóknak a saját 3D-s algoritmusaikat szoftveresen kellett implementálniuk. Így a vertex és pixel számítások teljes mértékben programozhatóak voltak. Azonban a CPU-k túl lassúak voltak látványos 3D-s effektek valós idejű kiszámításához.

Ma a 3D-s alkalmazásoknak nem kell a saját 3D-s renderelő algoritmusaikat a CPU-n implementálniuk, ezt a feladatot a két 3D-s programozási interfész valamelyikére, az OpenGL-re vagy a Direct3D-re bízhatják.

3.1. A programozhatóság szükségessége

Idővel a GPU-k minden mérhető tulajdonságukban gyorsabbak lettek a CPU-knál. A vertex feldolgozási sebesség tízezres nagyságrendről százmilliósra



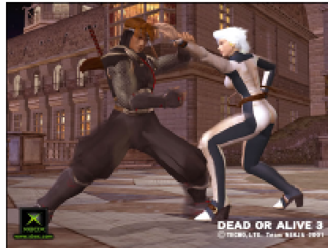
10. ábra. *A magas szintű árnyaló nyelvek kialakulása*

nőtt. A pixel feldolgozási sebesség pedig milliósról tízmilliárdosra emelkedett. Nem csak az, hanem a GPU-k funkcionalitása is megnőtt, amelyek új renderelési algoritmusok implementálását teszik lehetővé. Mindennek az eredménye a jobb minőségű renderelt képeken túl a moziszerű renderelés. A fejlődést a 11. ábra szemlélteti.

A hardver és képességeinek csodálatos fejlődése ellenére és a magas szintű árnyaló nyelvek eljövetele előtt a GPU-k assembly nyelven voltak programozhatóak. A magas szintű nyelvek és az assembly közötti külömbségeket a 12. ábra mutatja.

A mai modern GPU-k, amelyek programonként utasítások tízezreinek végrehajtására képesek, az assembly nyelvek közvetlen használata értelmetlenné vált. Azon túl, hogy nehéz kódolni, a benne írt kód nem újrafelhasználható és nehezen debuggolható.

Ezen okok miatt az ipar felismerte, hogy szükség van magas szintű GPU programozási nyelvekre, mint a *HLSL*, *GLSL*, és a *Cg*.



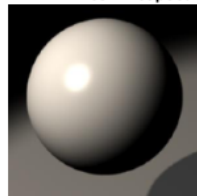
11. ábra. *A GPU-k fejlődése*

Assembly

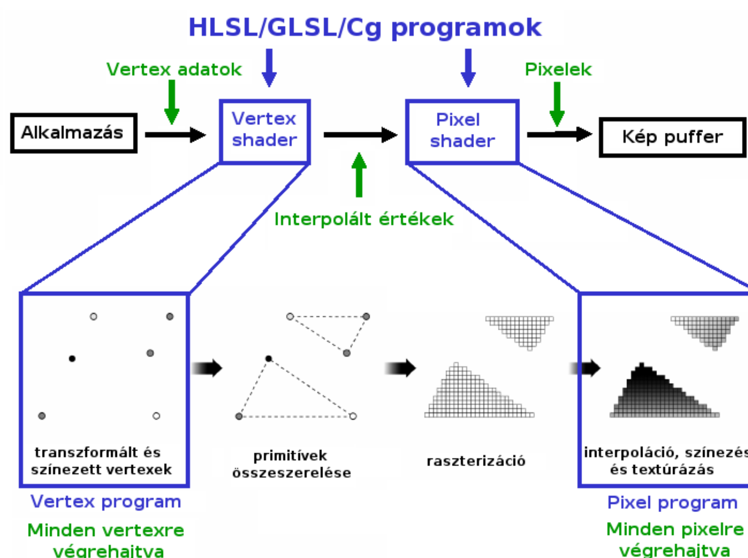
```
…
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
…
```

Magas szintű nyelv

```
…
float3 cSpecular = pow(max(0, dot(Nf, H)),
                       phongExp).xxx;
float3 cPlastic = Cd * (cAmbient + cDiffuse) +
                  Cs * cSpecular;
…
```



12. ábra. *A GPU-k programozási nyelvei*



13. ábra. *Shaderek a GPU-ban*

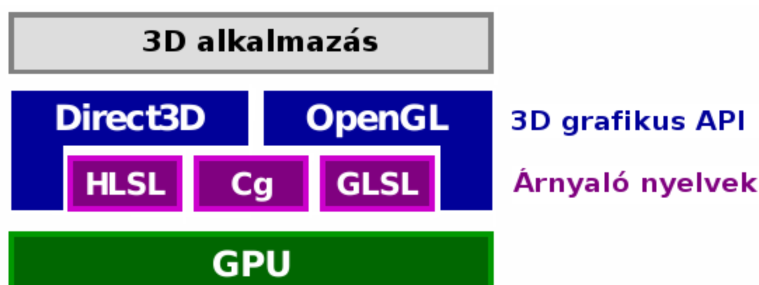
3.2. A GPU programozási nyelvek és a grafikus futószalag

A tradícionális rögzített funkciójú grafikus futószalagban az alkalmazás elküldi a vertexeket a grafikus kártyának és egy sor mágikus művelet megtörténése után végül a kép pufferben a színezett pixelértékek jelennek meg. Ezen műveletek némelyike a programozó által is befolyásolható volt, de a legtöbb funkcionalitás kőbe volt vésve.

A programozható árnyalás eljövetelével ezek a rögzített funkciók eltűntek és testreszabható feldolgozókkal helyettesítődtek. A shaderek a grafikus futószalagban elfoglalt helyét a 13. ábra szemlélteti.

Az első GPU, amely ezt a fajta shader programozást lehetővé tette a GeForce 3 volt 2001-ben. Bár nagy előrelépés volt, de igazából csak vertex programozást tett lehetővé. Csak 2003-ban a GeForce FX GPU-k voltak teljesen pixel programozhatóak, amely pixel programonkénti több mint 1000 utasítás végrehajtását is támogatta. A mai GPU-knál ez a szám még magasabb a dinamikus elágazásokkal és ciklusokkal.

A HLSL, GLSL, és Cg használatával pontosan meg tudjuk mondani a GPU-nak, hogy mit szeretnénk minden a grafikus futószalagon végighaladó vertexen és pixelen végrehajtani. A jövőben a grafikus csővezeték még több része válik majd programozhatóvá.



14. ábra. *Shaderek a különféle API-kban*

3.3. Árnyalók fordítása

A GPU programozási nyelvek használata a következő:

- Használjuk a 3D-s API hívásait, hogy megadjuk a vertex és pixel shadereket.
- Engedélyezzük a vertex és pixel shadereket.
- Töltsük be és engedélyezzük a textúrákat ahogy általában tennénk.
- Rajzoljuk ki a geometriát ahogy eddig tettük.
- Állítsuk be a blendelést mint korábban.
- A vertex shader le fog futni minden vertexre.
- A pixel shader le fog futni minden képernyőre kerülő pixelre.

Az alkalmazás és a 3D-s API egymásra épülését és a grafikus hardverrel való kapcsolatát a 14. ábra szemlélteti.

Néha az árnyaló nyelvek többet is ki tudnak fejezni, mint amire a GPU-k képesek (a GPU-któl függően). Erre a problémára a nyelvek tervezői létrehozták a profilokat. Minden profilnak körvonalazva van a képessége, amelyet egy GPU támogat a vertex vagy pixel shaderében. Ezúton hiba üzenetet kapunk, ha olyan kódot akarunk fordítani, amelyet az nem támogat.

A fordítási profilok segítségével hardver specifikus kód is fordítható:

- Optimalizált teljesítmény.
- Az extra hardver képességek kihasználása.

- Limitálhatja a nyelvi konstrukciókat a kisebb képességű hardverek számára.

Fordítási profilok:

- Vertex profilok: *vs_1_1, vs_2_0, vs_2_x, vs_3_0, arbvp1, vp20, vp30, vp40*
- Pixel profilok: *ps_1_3, ps_1_4, ps_2_0, ps_2_x, ps_3_0, arbfp1, fp30, fp40*

3.4. A nyelv szintaxisa

A magasszintű shader nyelvek közül a HLSL, Cg, GLSL általában bőségesen elegendőek, de mind különböző. A HLSL-t a Microsoft és az NVIDIA fejlesztette ki közösen, a Cg-t az NVIDIA, így nem meglepő, hogy a HLSL és a Cg elég hasonlóak, a GLSL azonban kicsit komolyabban különbözik tőlük. Az árnyaló nyelvek szintaktikájának bemutatásához csak az egyik nyelvet (Cg) használom fel. Az indokok:

- Az OpenGL shader nyelv (GLSL), csak az OpenGL-el használható.
- A HLSL csak Direct3D-ben használható és csak Windows platformon.
- A GLSL-ES csak OpenGL-ES-el használható együtt.
- A Cg (*C for Graphics*) API független és platform független is.
  - Használható: PlayStation3, ATI, NVIDIA, Linux, MacOS X, Windows, stb.
  - A legjelentősebb DCC (digitális tartalom készítő) alkalmazások is támogatják (*Maya, 3DS Max, XSI*)

A C-hez hasonlóan itt is vannak feltételes fordítási direktívák: `#define`, `#ifdef`. Viszont assembly nyelvű betéteket nem lehet alkalmazni.

3.4.1. Adattípusok

Az alábbi adattípusok használhatóak Cg-ben:

`float`	32 bites IEEE 754 szabványnak megfelelő lebegőpontos típus
`half`	16 bites IEEE szabványhoz hasonló lebegőpontos típus
`bool`	logikai típus
`sampler`	Textúra mintavételező
`struct`	C/C++ struktúra analógiája

Mutatók egyelőre nincsenek. Természetesen itt is lehetőség van tömbök, vektorok és mátrixok definiálására. A hardverhez közel álló összetett típusok előre definiálva vannak. A natívan támogatott összetett típusok:

- Vektorok: `float2`, `float3`, `float4`, `half2`, `half3`, `half4`
- Mátrixok:
  `float2x2`, `float2x3`, `float2x4`,
  `float3x2`, `float3x3`, `float3x4`,
  `float4x2`, `float4x3`, `float4x4`,
  `half2x2`, `half2x3`, `half2x4`,
  `half3x2`, `half3x3`, `half3x4`,
  `half4x2`, `half4x3`, `half4x4`

Általánosabb tömbök deklarációja C szerűen történik: `float lightexp[8];` de a `float v[4];` nem egyenlő a `float4 v;` deklarációval.

3.4.2. Névtúlterhelés

A függvényekre érvényes a névtúlterhelés, amely sok típus esetén nagyon hasznos. Példák:

```
float funcA(float3 x);
float funcA(half3 x);
float funcB(float2 a, float2 b);
float funcB(float3 a, float3 b);
float funcB(float4 a, float4 b);
```

3.4.3. Konstansok

Különböző konstans használati szabályok:

- C-ben könnyű véletlenül magasabb precizitással számolni:

```
half    x, y;
x = y * 2.0;    // a szorzás float
                // pontossággal történik!
```

- De Cg-ben nem:

```
x = y * 2.0;    // a szorzás half
                // pontossággal történik y miatt!
```

- Mígnem ezt akarjuk:

```
x = y * 2.0f;   // a szorzás float
                // pontossággal történik!
```

3.4.4. Natív mátrixműveletek

A vektorok és mátrixok nem csak összetett típusként támogatottak, vannak natív műveletek a használatukhoz:

- Komponensenkénti műveletek: összeadás (+), kivonás (-), szorzás (*), osztás (/)
- Skaláris (belső) szorzat:

```
dot( v1, v2 );  // visszaad egy skalár értéket
```

- Mátrixszorzások:

```
float4x4    M, N, O;    // 4x4-es float mátrixok
float4      v, r;       // 4 elemü float vektorok

// mátrix-vektor szorzás:
r = mul( M, v );        // visszaad egy vektort

// vektor-mátrix szorzás:
r = mul( v, M );        // visszaad egy vektort

// mátrix-mátrix szorzás:
O = mul( M, N );        // visszaad egy mátrixot
```

3.4.5. Új operátorok

Új operátorok alkalmazására is lehetőség nyílik:

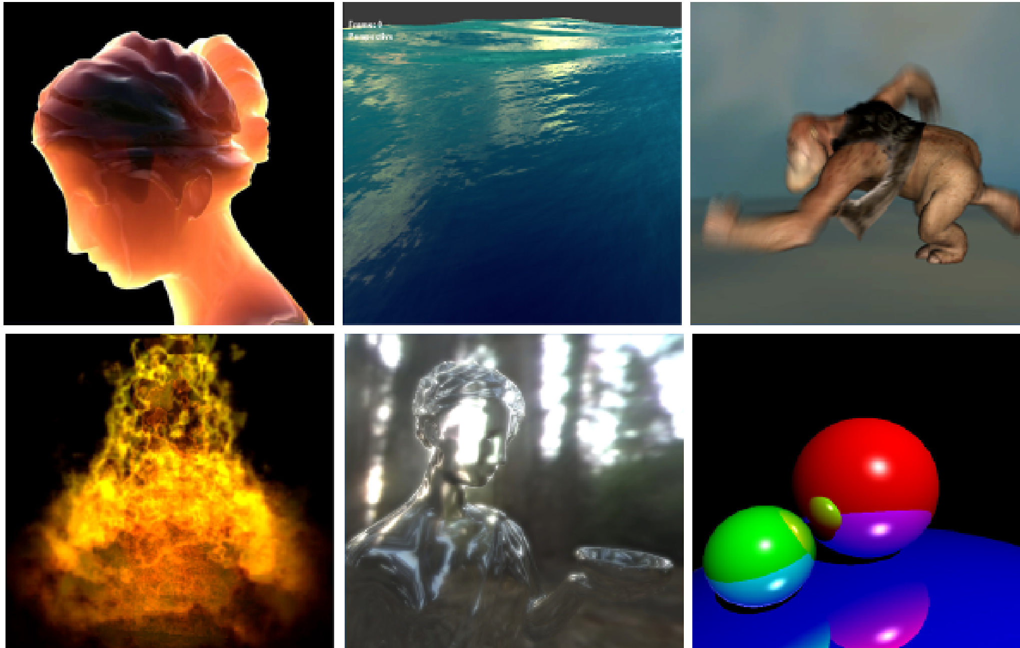
- Vektor konstruktor:

```
float4  a = float4( 1.0, 0.0, 0.0, 1.0 );
```

- A „swizzle” operátorral kiemelhetünk elemeket vektorokból és mátrixokból, illetve permutálhatjuk azokat:

```
a = b.xxyy;
```

- Példák:



15. ábra. *Példa shaderek*

```
float4  vec1 = float4( 4.0, -2.0, 5.0, 3.0 );
float2  vec2 = vec1.yx;     // vec2 = ( -2.0, 4.0 )
float   scalar = vec1.w;    // scalar = 3.0
float3  vec3 = scalar.xxx; // vec3 = ( 3.0, 3.0, 3.0 )
float4x4 myMatrix;

// myFloatScalar = myMatrix[3][2];
float    myFloatScalar = myMatrix._m32;
```

3.4.6. Összegzés

A 15. ábra néhány shaderekkel megvalósított effektet mutat. Amint látható, a Cg szintaxisa nagyon hasonlít a C-hez, de néhány bővítéssel is rendelkezik, amely a grafikai programozást segíti.

4. Eredmények (*CyGNUsGL*)

4.1. Platformfüggetlenség

A keretrendszertől az alábbi fordítókkal való teljes kompatibilitás várható el:

- *GNU Compiler Collection* C++ 3.3+/4.0+
- *Intel C++ Compiler 9.0+*
- *Microsoft Visual C++ 14.0+ (Visual Studio 2oo5)*

Ezek a fordítók a legújabb C/C++ ANSI/ISO szabványokon túl a *„Streaming SIMD Extensions intrinsics”* C++ kiterjesztést is támogatják, ami annyit jelent, hogy az *x86*-os és *x86-64*-es architektúrájú processorok SSE utasításkészletét C++ nyelvi szinten használhatjuk. Mi csak az utasítások logikai sorrendjét határozzuk meg, a fizikailag előállított kódot és a regiszterallokációt a fordító optimalizálja a célprocesszorra (például: Pentium3, Pentium4, AthlonXP, Athlon64, Core2Duo).

4.2. Nem grafikai célú eszközök

A *CyGNUsGL* általános nem grafikai célú eszközrendszere (makrók, inline-ok, template-ek, wrapperek):

- Különböző architektúrájú processzorok extra képességeinek kihasználása globálisan makrók és inline-ok segítségével (X87/SSE/SSE2/SSE3/SSSE3), így az elérhető leggyorsabb kód fordul le a binárisba.
- Használhatók debuggoló makrók (pl.: `error, warning, info` stb.), amelyek csak adott debuggolási szint fölött fordítódnak a programkódba: `CY_DEBUG={0|1|2|3}`. A debug információk kimenete is irányítható. Figyelhetők egyéb pl. OpenGL futás közben adott hibajelzései is.
- A GCC estén a generált assembly kódba C++ szinten megjegyzés tehető. Mivel a fordító elég jelentős transzformációt, és optimalizációt hajt végre a kódon, és szeretnénk a generált kódot vizsgálni, akkor jól jöhet, ha a kód bizonyos részeit meg tudjuk jelölni.
- A hagyományos `abs(), min(), max(), saturate(), clamp()` makrók úgy lettek megírva, hogy a fordító a PentiumPro vagy újabb processzor esetén a `CMOVx` utasításokkal elkerülhesse a feltételes elágazásokat, ezzel javítva a generált kód minőségét.

- Idő mérését CPU ciklusokban kifejezhető módon is lehetővé teszi, az alacsony szintű `RDTSC` utasítással, erre két külön wrapper osztály is épül (`timer4, timer8`).

- A nem konstans paraméterrel hívott `sin(), cos()` függvények hívása költséges, hiába van belőlük SSE2-es verzió. Bizonyos esetekben, például egérrel való forgatások esetén mindössze néhány száz előre kiszámított `sin(), cos()` érték táblázatból történő kiolvasása sokkal gyorsabb lehet, erre a feladatra egy `sc_cache` template osztály szolgáltat megoldást.

- A `<cstdio>` standard fájlkezelése nehézkes nem objektum-orientált, az `<iostream>` pedig tejlesítményben nem a legjobb. Az `std_file` wrapper a standard fájlkezelést egyszerűsíti objektum-orientáltan.

4.3. Lineáris matematikai eszközök

Grafikus API-tól független lineáris matematikai számításokat segítő eszközök:

- A `float4` osztály egy 4 komponensű `float` vektor osztály. A `float4x4` pedig 4x4 db `float` elemeket oszlopfolytonosan tároló mátrix. SSE optimalizációt engedélyezve a műveleteik SSE/SSE3 utasítások használatával fordítódnak le, kihasználva a mai CPU-k tejles lebegőpontos tejlesítményét. Operátor-túlterhelést nem tartalmaznak, mivel azt tapasztaltam, hogy operátor felüldefiniálás során a sok felesleges másolást a fordítók nem voltak képesek tejlesen eliminálni – még az *Intel* fordítója sem –, jelentősen csökkentve ezzel a teljesítményt.

- Az alapvető vektor és mátrixműveletek is megtalálhatóak. A mátrixokból viszont többféle létezik tartalom alapján. Vannak tejles transzformációs mátrixok, és affin transzformációs mátrixok. Az affin transzformációs mátrix eltolásokat elforgatásokat stb. tartalmaz, így az inverze a transzponáltja is egyben, ill. tartalmaz 3 db 0 és egy 1-es értéket is a mátrixa. Ez további optimalizációs lehetőséget jelent az affin mátrixok összeszorzására, valamint egy affin és egy teljes mátrix összeszorzására. Ezek is implementálásra kerültek.

- A `cam32` egy olyan kamera osztály, amely két `float4x4`-et tartalmaz, az egyik a kamera bázisa, a másik pedig a tejles transzformációs mátrixot tartalmazza. A GPU számára hasznos adat a kamera pozíciója, ami a bázis utolsó vektora, illetve a tejles transzformációs mátrix, amely a bázis inverz transzponáltjának egy projekciós mátrixszal való szorzata.

A kamera műveleteihez tartoznak a tengelyek menti elmozdulások és forgatások, és maga a projektív transzformáció:

- `left()`, `right()`, `up()`, `down()`, `forward()`, `back()`
- `rot_x()`, `rot_y()`, `rot_z()`
- `calc_trviewproj()`

Ezeket a műveleteket nem úgy valósítottam meg, hogy feltöltöttem egy megfelelő transzformációs mátrixot és összeszoroztam a bázissal, hanem a szorzandó mátrixokban lévő 0 és 1-es értékeket kioptimalizálva csak a szükséges számításokat végztem el. Természetesen ezeknek a műveleteknek is van SSE/SSE3-ra optimalizált verziója. Egy példa ezeknek ez optimalizációknak a bemutatására:

```
inline void cam32::rot_x(const brc& co, const brc& si)
{
#if CY_IS == CY_IS_X87  // az x87-es verzió

    register float   t;

    t = basis.fb;
    basis.fb = basis.fb*co + basis.fc*si;
    basis.fc = basis.fc*co - t*si;

    t = basis.ff;
    basis.ff = basis.ff*co + basis.fg*si;
    basis.fg = basis.fg*co - t*si;

    t = basis.fj;
    basis.fj = basis.fj*co + basis.fk*si;
    basis.fk = basis.fk*co - t*si;

#elif CY_IS >= CY_IS_SSE    // az SSE-s verzió

    register __m128 tbfjn = basis.bfjn;

    basis.bfjn = _mm_add_ps(
                    _mm_mul_ps( basis.bfjn, co.xyzw ),
                    _mm_mul_ps( basis.cgko, si.xyzw ) );

    basis.cgko = _mm_sub_ps(
```

```
                        _mm_mul_ps( basis.cgko, co.xyzw),
                        _mm_mul_ps( tbfjn, si.xyzw ) );
#endif
}
```

4.4. A grafikai segédeszközök

A grafikai programozáshoz az OpenGL-t választottam a Direct3D-vel szemben. Az OpenGL-t azért részesítettem előnyben mert minden Linux, UNIX, Windows és MacOS operációs rendszer alatt támogatott, és az Ada, C, C++, Fortran, Python, Perl és Java nyelvekből is használható. A *PlayStation3* is OpenGL-en keresztül programozható (konkrétan OpenGL-ES-en keresztül). Az OpenGL másik nagy előnye a bővíthetőség, a különböző hardvergyártók saját hardvereikhez jobban hozzáigazíthatják az API-t, amelytől az adott hardver a lehető legjobban kihasználható mind funkcionalitásban mind sebességben, azonban hardverfüggővé is válik. Gondolhatnánk, hogy a platform függetlenség a teljesítmény rovására megy, de ez nem így van, hiszen most már a legnagyobb grafikus hardvergyártók (*NVIDIA, ATI-AMD*) fejlesztik az OpenGL-t, és a hardvereik meghajtóprogramjának nem csak a kernel oldali részét, hanem a kliensét is ők biztosítják, ellenben a Direct3D-vel, ahol ezt a *Microsoft* Direct3D Runtime vállalja át.

A keretrendszer a grafikai programozást az OpenGL grafikus API használatának megkönnyítésével teszi:

- Az OpenGL-es alkalmazás inicializálásához három lehetőség kínálkozik:

    1. Az ablakozó rendszer függetlenségéhez a *Simple Directmedia Layer (libSDL)*-en keresztül biztosít hozzáférést (`http://libsdl.org/`).
    2. Az *X Windowing System*-hez az *X GLX* kiterjesztésén keresztül.
    3. A *Windows*-os rendszereken pedig a *WGL* segítségével.

- Az OpenGL kiterjesztéseinek kezelését is könnyíti, bár léteznek különféle függvénykönyvtárak (*GLEW, GLEE*), amelyek pontosan erre valók, én mégis úgy döntöttem, hogy sajátot írok a következő okok miatt:

    1. Mások az összes lehetséges kiterjesztést betöltik amit éppen tud a rendszer, és ellenőrizni kell a kiterjesztések jelenlétét. Dinamikus megoldás, a nagyobb memóriaigényen túl statikus linkelés esetén nagyban növeli a lefordított bináris méretét, vagy a dinamikusan hozzálinkelt könyvtárat kell adni az alkalmazáshoz. A licencekről nem is beszélve.

2. Az én megvalósításomban viszont feltételes fordítási makrók segítenek, hogy csak a szükséges kiterjesztések töltődjenek be, mindenből pontosan annyit használva, amennyi kell. Ráadásul Linux-os környezetben betöltésről szó sincs, hiszen az OpenGL driver-hez linkelődik az alkalmazás közvetlenül, tehát az összes kiterjesztés elérhető, amit a driver biztosít. Windows-ban viszont be kell tölteni a kiterjesztéseket. A `glext_loader` a két eltérő működést elfedi, és még a betöltött függvények érvényességét is leellenőrzi, amennyiben a megfelelő makróval ezt is kérjük.

- A szabvány OpenGL-re és a `glext_loader`-re épül a `glwrapper`, amely az OpenGL függvények meghívását csak helyes paraméterezéssel engedi meghívni. Azért függ a `glext_loader`-től, mert a megengedett paraméterek a kiterjesztésektől is erősen függnek, hiszen egy kiterjesztés új belépési pontokon túl új paraméterzési lehetőséget is hoz.

- Az OpenGL-ben az adatok objektumokként jönnek létre, érhetők el. Ezek használatához könnyítésként az alábbi osztályok állnak rendelkezésre:
  - A `gl_list` a display listák használatához.
  - A `gl_query` az OpenGL query-khez: *GL_ARB_occlusion_query, GL_EXT_timer_query*.
  - A `gl_fence` az *GL_NV_fence* kiterjesztéshez.
  - A `gl_program` az alacsony szintű shaderekhez: *GL_ARB_vertex_program, GL_ARB_fragment_program*.
  - A `gl_vbo` az OpenGL-es vertex és index pufferekhez (*GL_ARB_vertex_buffer_object*).
  - A `gl_rbuffer` és `gl_fbo` az általánosított kép pufferekhez és a textúrára rendereléshez (*GL_EXT_framebuffer_object*).
  - A `gl_tex` pedig a textúrák kezeléséhez.

- Egy tejlesen új vertexek és indexek tárolását végző fájlformátum is a rendszer része, csak ennek a támogatása van megoldva, viszont a keretrendszert felhasználó egyéb kisebb programok segítenek a különféle fájlformátumok közötti konvertálásokban.

- Új textúra fájlformátum is került a rendszerbe, speciálisan OpenGL-re kihegyezve. Azon túl, hogy az összes textúra típust ismeri (1D, 2D, 3D, CUBE, COMPRESSED), támogatja az összes létező textúra formátumot is. A textúrák „mipmap”-jeinek tárolásáról is gondoskodik.

4.5. Gyakolati alkalmazás

A saját shaderek, demók fejlesztésén kívül az orvosi képfelolgozás számára is fontos feladatok elvégzésében könnyítette meg munkámat a *CyGNUsGL*. A PET – *Pozitron Emissziós Tomográf* – vizsgálatokhoz szükséges számítások GPU-val történő gyorsítását kellett megoldanom. A feladat a *radon* transzformáció megvalósítása volt egy 3D-s volume-on. Ez nem jelent mást, mint a 3D-s volume-ot levetíteni több előre megadott síkra (tipikusan 180db síkra), a vetítés során, pedig vagy a maximum értéket vagy az átlagot kellett kiszámítani. Továbbá feladat volt a *radon* transzformációból létrejött adatokból a *szinogrammok* kiszámítása. Ezeket a feladatokat a hagyományos CPU-s megvalósításnál jóval gyorsabban lehetett elvégezni GPU-val.

Bár legjobb tudomásom szerint a CPU-s verzió nem volt optimális, egy 8 CPU-t tartalmazó klaszteren a számítás **483** másodpercig tarott, a GPU-s verzió ugyanazokon az adatokon egyetlen GeForce 8800 GTX-el pedig **456** ezredmásodperc alatt végezte el a transzformációt (a bemutatón ekkor először valós időben).

További feladat egy olyan képfeldolgozási csomag kifejlesztése, ami a GPU-t, mint általános nagy teljesítményű stream-processzort használja, az orvosi képfeldolgozás területén.

4.6. Továbbfejlesztési lehetőségek

A legfőbb továbbfejlesztési lehetőség a következő OpenGL verziók (*„OpenGL Longs Peak" és „OpenGL Mount Evans"*) támogatásának integrálása. Megjelenése 2oo7 nyarára várható. Az új OpenGL teljesen inkompatibilis lesz a ma létezővel, és csak a legújabb GPU-k fogják támogatni (jelenleg egyedül az *NVIDIA GeForce 8* sorozata képes erre).

A gyökerektől fogva újratervezik az egész API-t, hiszen az aktuális 1992-ben lett bemutatva, amely mára elavultnak számít. Az új modell a futás közbeni tejlesítményre lesz optimalizálva, szemben a maival, ahol az objektum létrehozás áll a középpontban, emiatt minden renderelési hívás egy sor validálást és adminisztrációt von maga után. Az új DirecX 10-ben is hasonló újítást hozott be a *Microsoft*, de az csak a *Windows Vistában* érhető el. További részletek az új OpenGL-ről itt találhatók:
`http://opengl.org/pipeline/vol001/`
`http://opengl.org/pipeline/vol002/`
`http://opengl.org/pipeline/vol003/`

A *CyGNUsGL*-t a jövőben mint nyílt forráskódú projektet szertném továbbfejleszteni. A bővítése egyszerű, főleg makrók, wrapperek és templatek.

A közeljövőre tervezett bővítések a shadereken és renderelési technikákon túl:

- Az új OpenGL kiterjesztésekhez a megfelelő wrapperek kifejlesztése, viszont ehhez új GPU-ra (GeForce 8800) is szükség lesz.
- Az új GPU architektúrával megvalósítható effektek, főleg az igazi „displacement mapping”.
- Az *„OpenGL Longs Peak”* támogatása.
- Az alapvető láthatósági teszteken túl bonyolultabb befoglaló objektum hierarchiák alkalmazása (BSP/KD fa), és portál technikák.
- A meglévő geometriai fájlformátumba animációk tárolásának kifejlesztése.
- Alapvető fizikai számítások, akár a GPU-val is.
- Részecskerendszerek GPU-s megvalósítása.
- 3D-s fraktálok vizsgálata a GPU-val.

4.7. Összegzés

Sikerült egy olyan nagyon jól bővíthető többplatformos (*Linux, *BSD, Win**) és több fordítót is támogató C++ nyelvű keretrendszert megalkotnom, amely a grafikus alkalmazások fejlesztését egyszerűsíti, megkönnyíti, de nem korlátozza le a fejlesztő elképzeléseit, és nem erőltet rá semmit. Kiemelten támogatja a korszerű processzorok utasításkészleteit (SSE-SSSE3), és az OpenGL-t.

Amint fentebb látható, nem feltétlenül grafikai célokra alkalmazható, hanem használható ott is, ahol a GPU-k félelmetes számítási kapacitását általánosabb feladatok elvégzésére szeretnénk használni, például az orvosi képfeldolgozásban.

A fentiekből látható, hogy egy nagyon jól alkalmazható keretrendszer került létrehozásra, amellyel nagy teljesítményű tejesen testreszabható alkalmzások fejleszthetők.

Hivatkozások

[1] Cass Everitt. Shadow mapping. Technical report, NVIDIA Corporation, 2001.

[2] Philipp Gerasimov; Randima Fernando; Simon Green. Shader model 3.0 using vertex textures. Technical report, NVIDIA Corporation, 2004.

[3] Simon Green. Bump map compression. Technical report, NVIDIA Corporation, 2004.

[4] Mark J. Harris. Normalization heuristics. Technical report, NVIDIA Corporation, 2004.

[5] Gary King. Shadow mapping algorithms. Technical report, NVIDIA Corporation, 2004.

[6] Simon Kozlov. Perspective shadow maps: Care and feeding. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*.

[7] Ashu Rege. Optimization for directx9 graphics. In *Game Developers Conference*, 2004.

[8] Michael Toksvig. Mipmapping normal maps. Technical report, NVIDIA Corporation, 2004.

[9] Matthias Wloka. Batch, batch, batch: What does it really mean? In *Game Developers Conference*, 2003.

[10] Cem Cebenoyan és Matthias Wloka. Optimizing the graphics pipeline. In *Game Developers Conference*, 2003.

5. Köszönetnyilvánítás

Ezúton szeretném megköszönni mindazoknak akik segítettek abban, hogy a diplomamunkámat elkészíthessem:

- a családomnak a bíztatást
- a barátaimnak az építő jellegű kritikákat
- azoknak a tanáraimnak akik tanulmányaim során olyan információkkal gazdagítottak, melyeket jól hasznosíthattam munkám során
- és főként témavezetőmnek, Dr. Tornai Róbertnek a hasznos tanácsait és a dolgozat írása közben mutatott áldozatkészségét