Nagy Antal:

Fejlett grafikai algoritmusok

A felsőfokú informatikai oktatás minőségének fejlesztése, modernizációja

TÁMOP-4.1.2.A/1-11/1-2011-0104

Főkedvezményezett:
Pannon Egyetem
8200 Veszprém
Egyetem u. 10.

Kedvezményezett:
Szegedi Tudományegyetem
6720 Szeged
Dugonics tér 13.

2014

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Fejlett grafikai algoritmusok

Dr. Nagy Antal

Képfeldolgozás és Számítógépes Grafika Tanszék
Szegedi Tudományegyetem

Tartalomjegyzék I

Tartalomjegyzék II

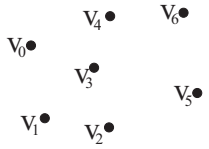
Tartalomjegyzék III

Bevezetés

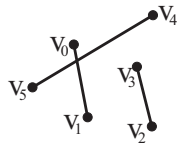
Grafikus csővezeték

Grafikus csővezeték

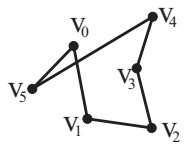
- Vertex feldolgozás
  - A vertexek egyenként a képernyő térbe vannak transzformálva
- Primitív feldolgozás
  - A vertexek primitívekbe vannak szervezve
- Raszterizálás
  - Primitívenként
    - Fragmensek
- Fragmens textúrázás és színezés
  - Fragmensenként



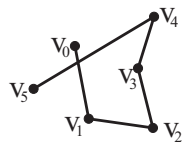
Pontok



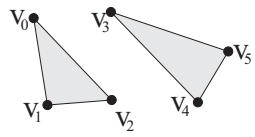
Vonalak



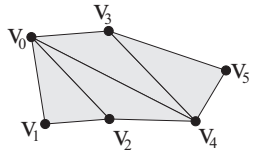
Vonal hurok



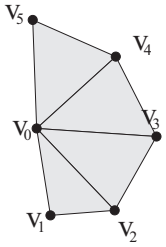
Töredezett vonal



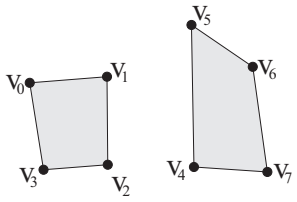
Háromszögek



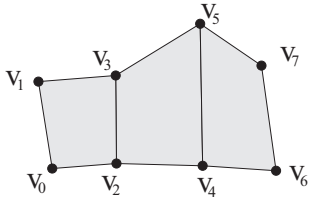
Háromszögsáv



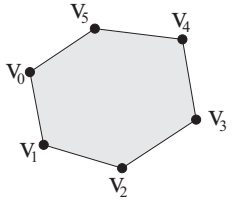
Háromszög-legyező



Négyszögek

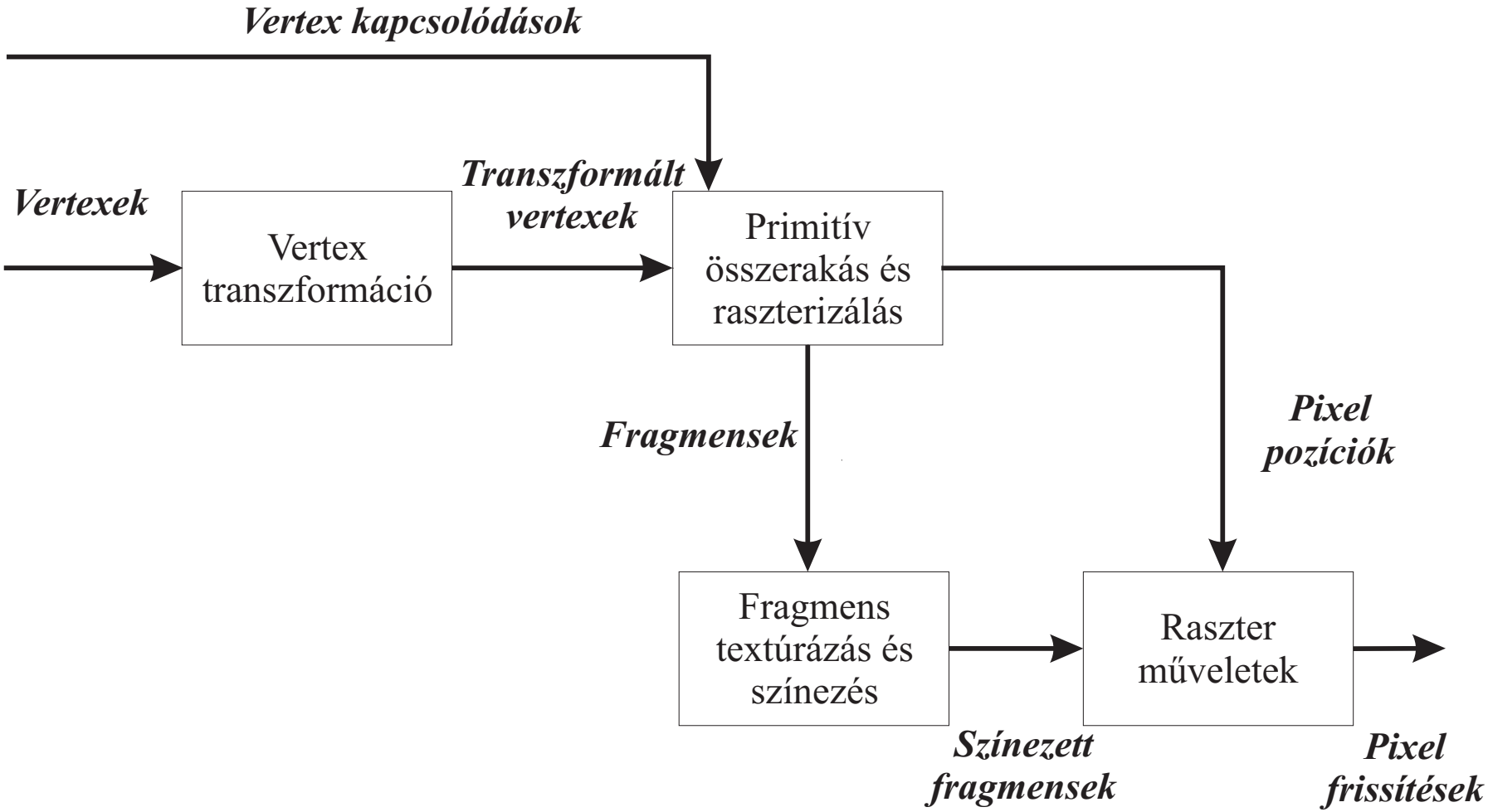


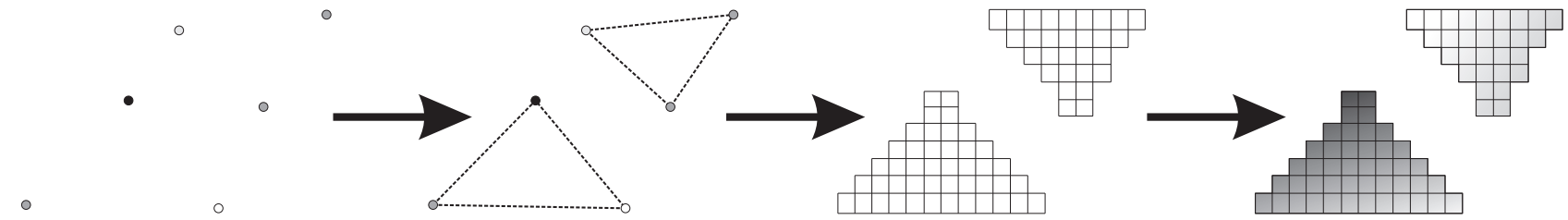
Négyszögsáv



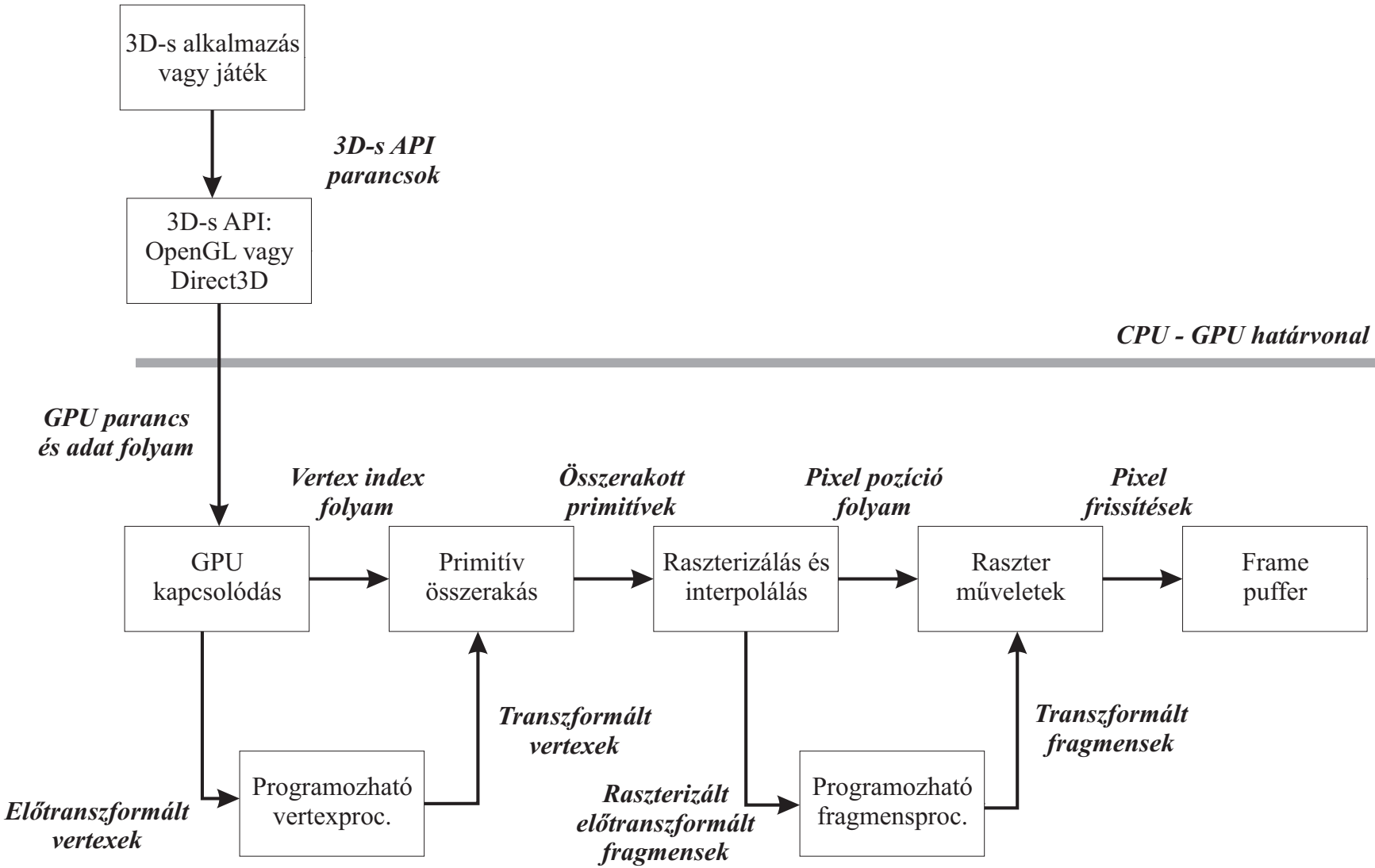
Poligon

Grafikus csővezeték

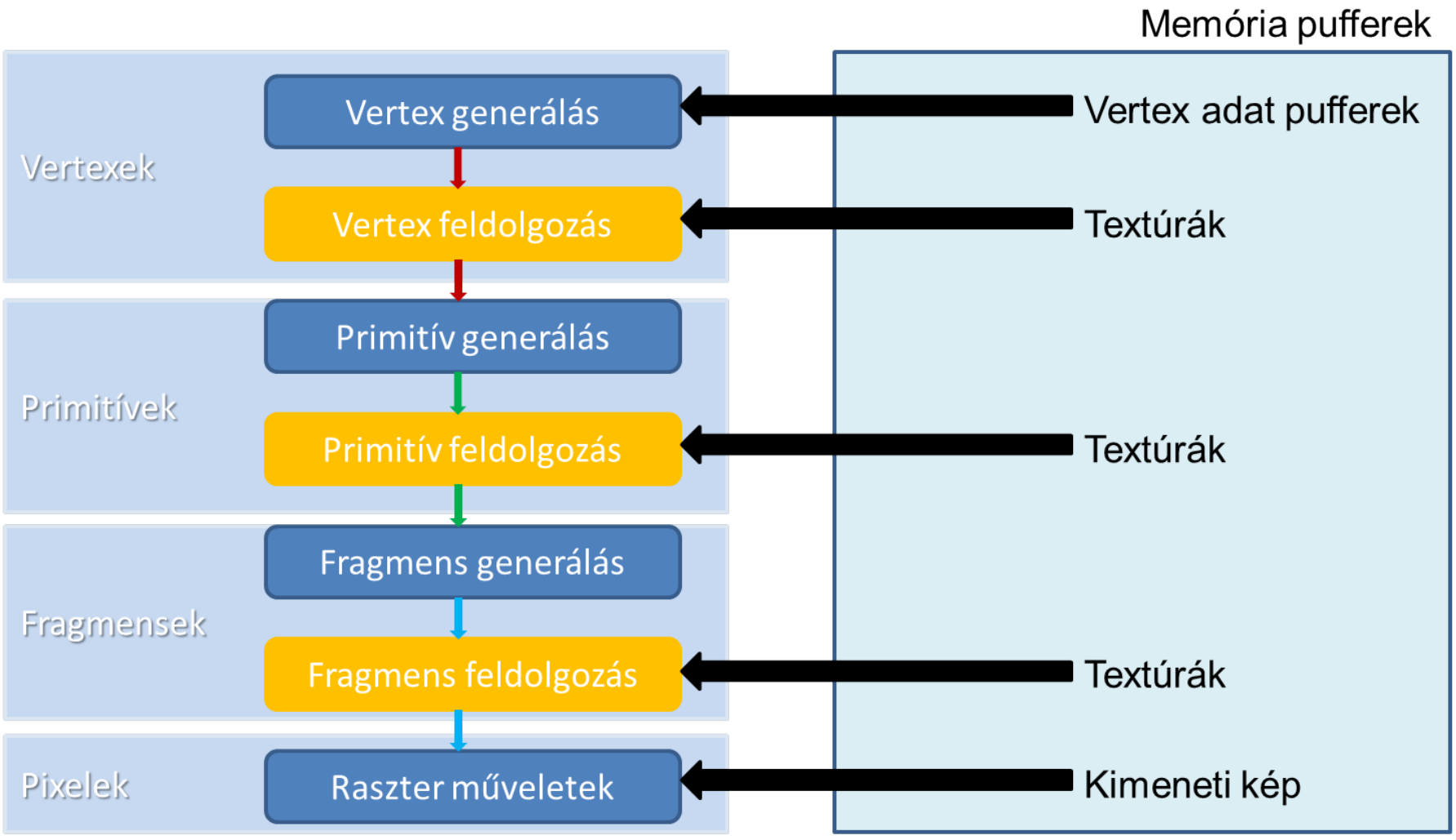



Színezett vertex
vertex transzformáció után
Primitív
összerakás
Raszterizálás
Interpoláció,
textúrázás és színezés

Programozható grafikus csővezeték



Grafikus csővezeték



GPU-k fejlődése

Grafikus csővezeték
SGI RealityEngine

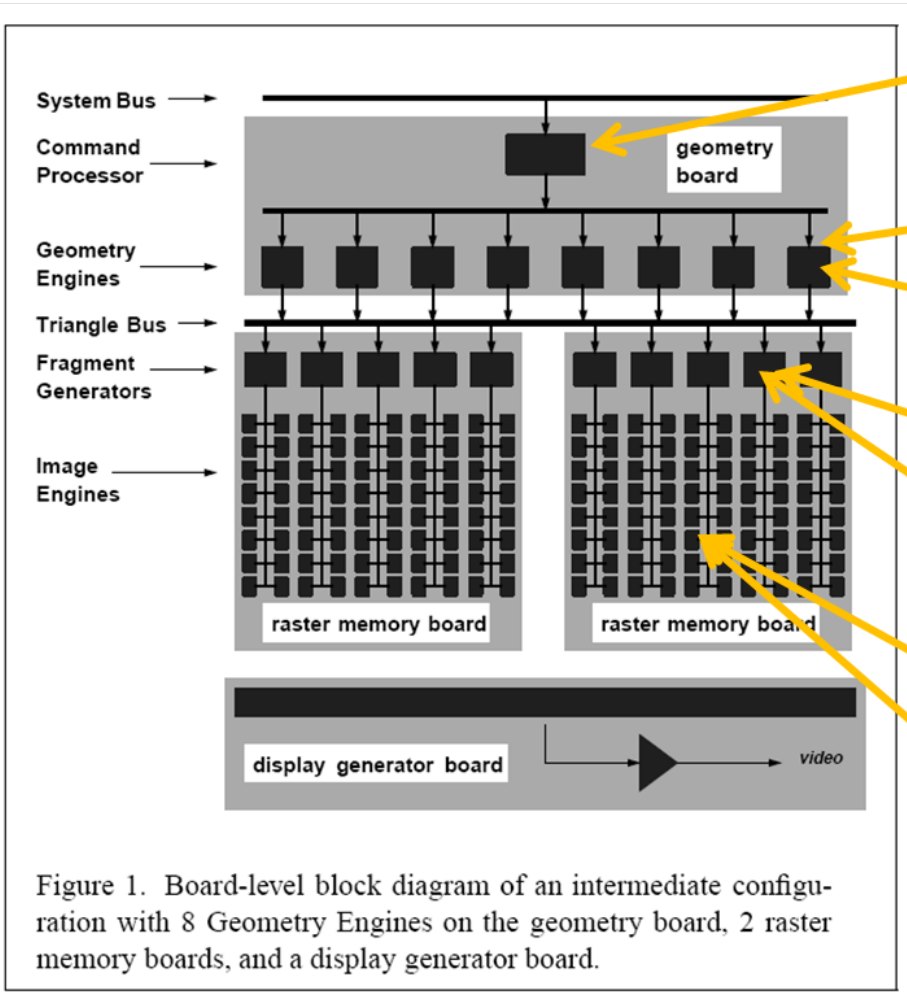
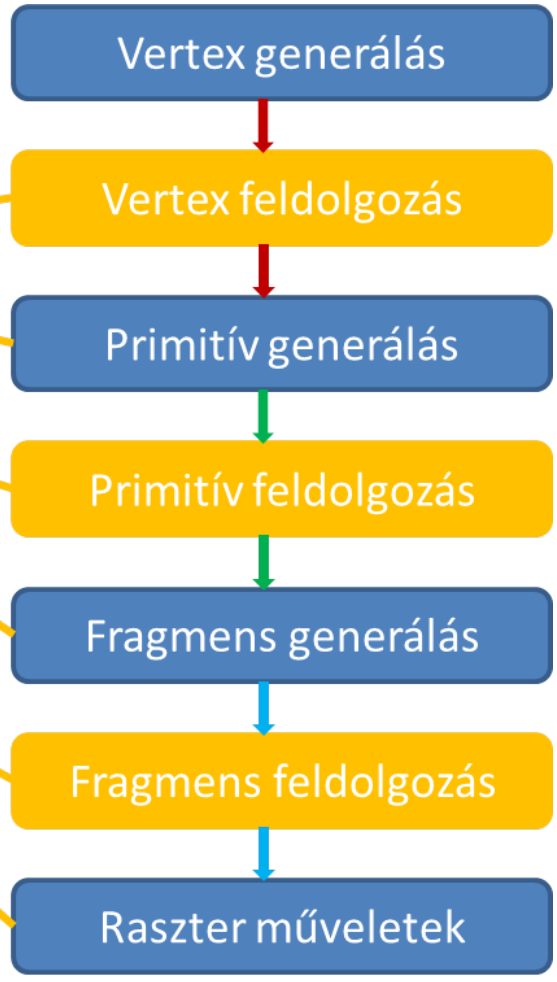


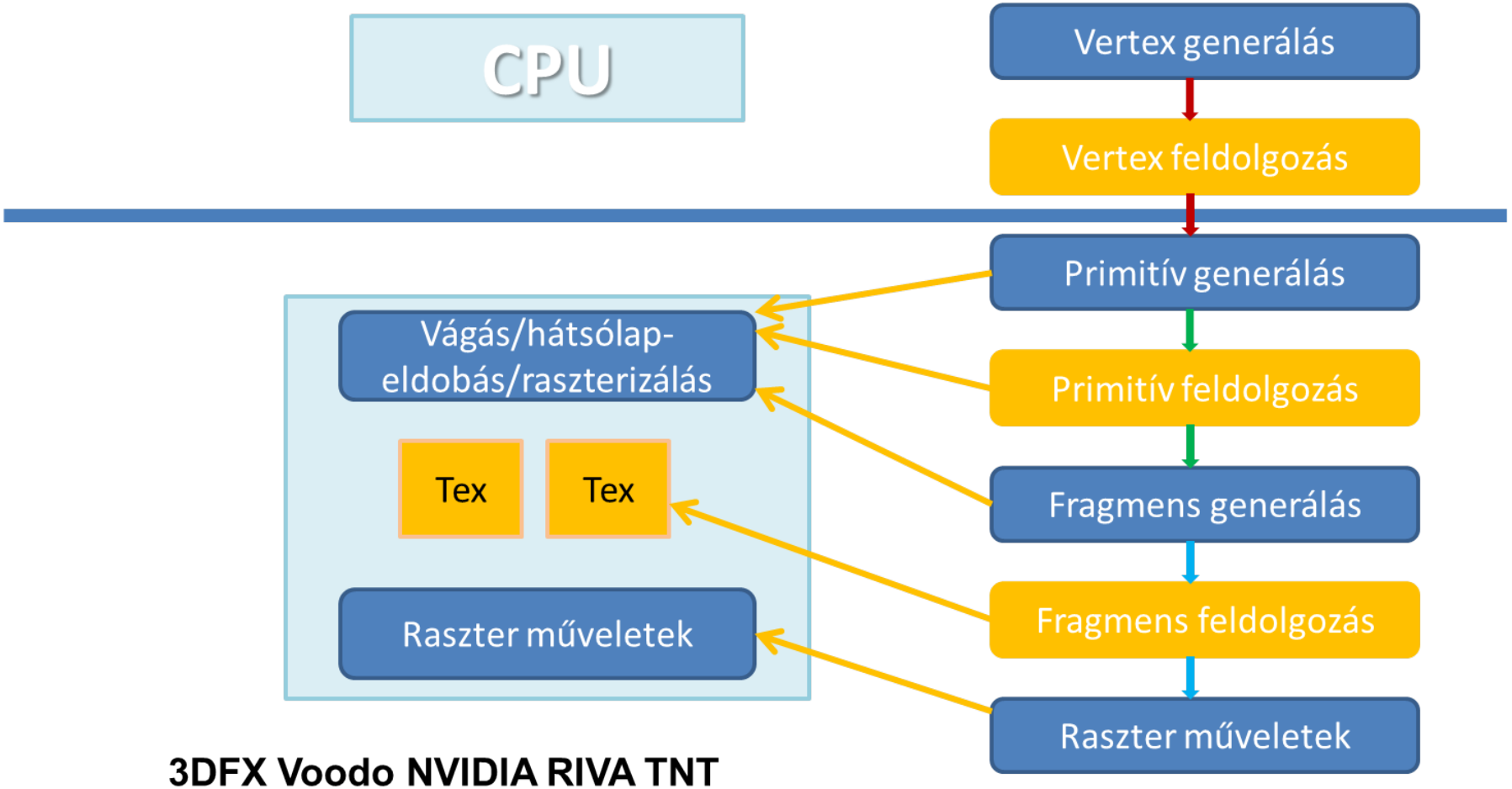
Figure 1. Board-level block diagram of an intermediate configuration with 8 Geometry Engines on the geometry board, 2 raster memory boards, and a display generator board.



Akeley, Kurt. "RealityEngine Graphics". Proceedings of SIGGRAPH '93, pp. 109-116.

Grafikus csővezeték

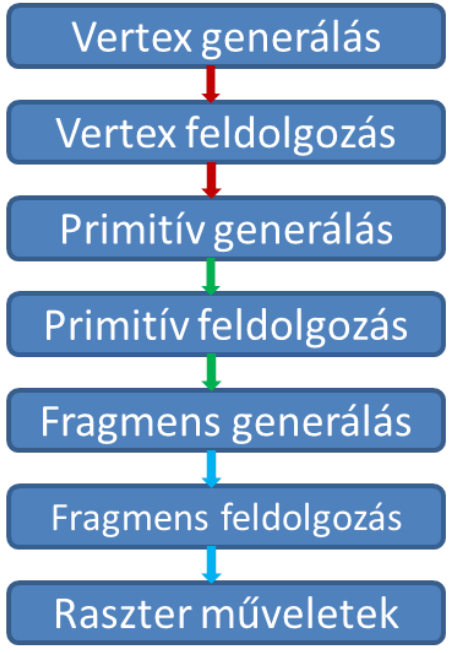
3D-s grafikus gyorsító 1999 előtt



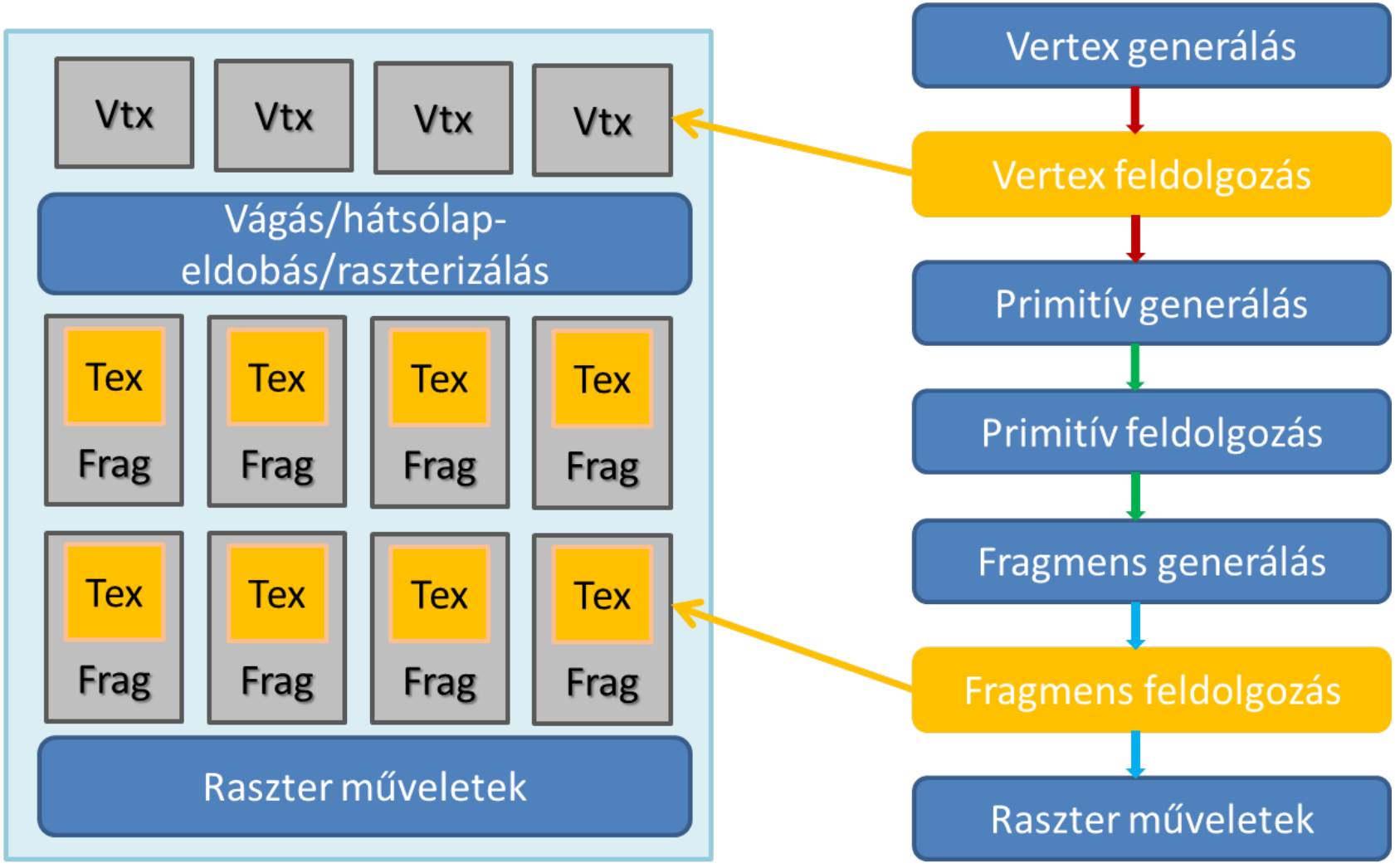
3DFX Voodo NVIDIA RIVA TNT

CPU

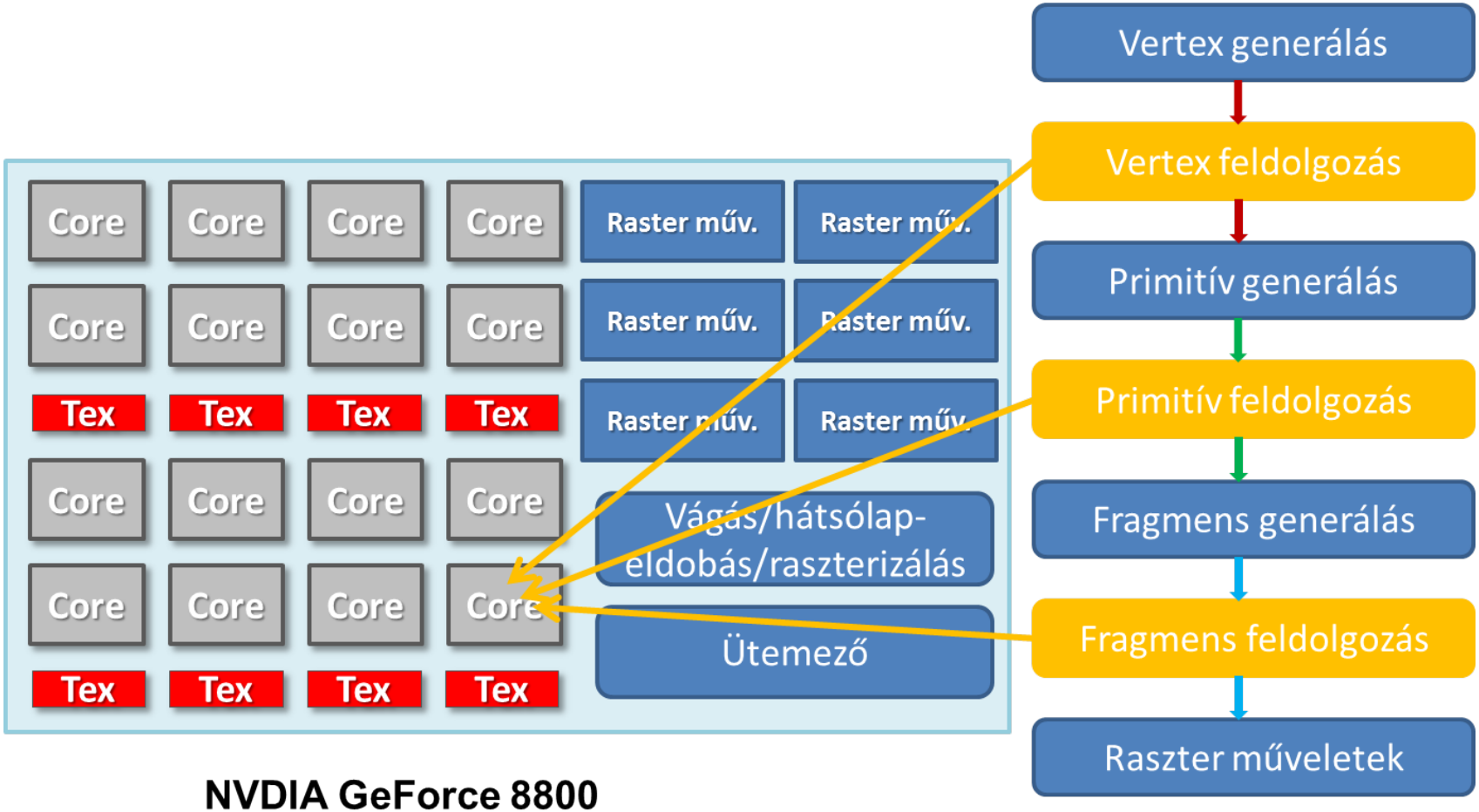
GPU



NVIDIA GeForce 256



ATI Radeon 9700

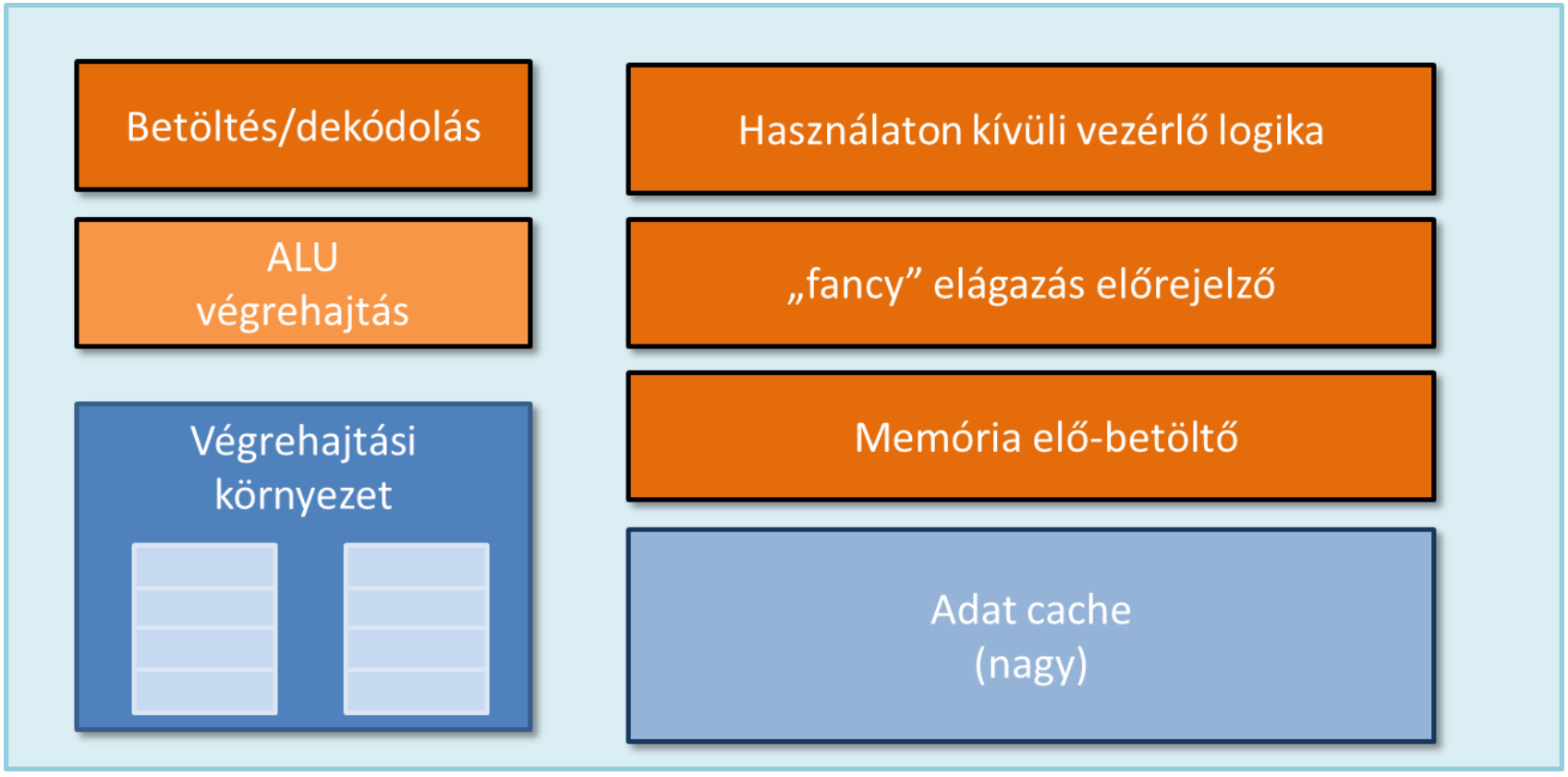


NVDIA GeForce 8800
egységes shader GPU

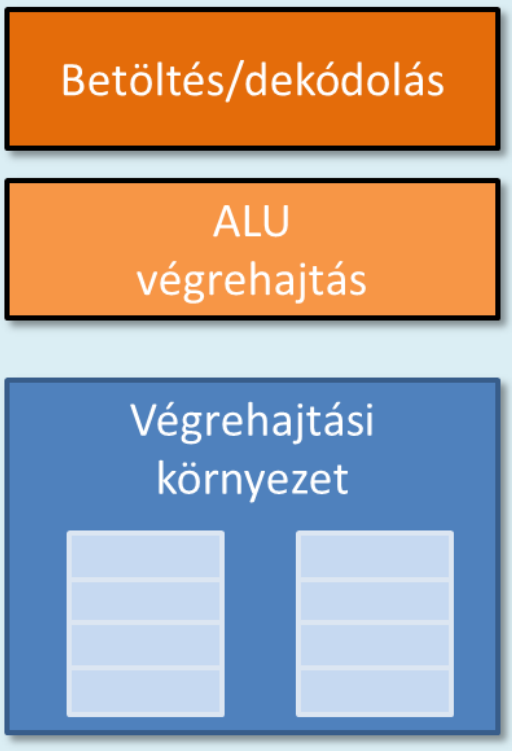
- Egységes shader modell
  - Shaderek megvalósítása közelebb került egymáshoz
    - Egyszerű
    - Kevés utasítás
    - Több száz általános célú végrehajtóegység
    - Hatalmas számítási kapacitás

Ötletek egy GPU felépítéséhez

CPU-stílusú core-ok



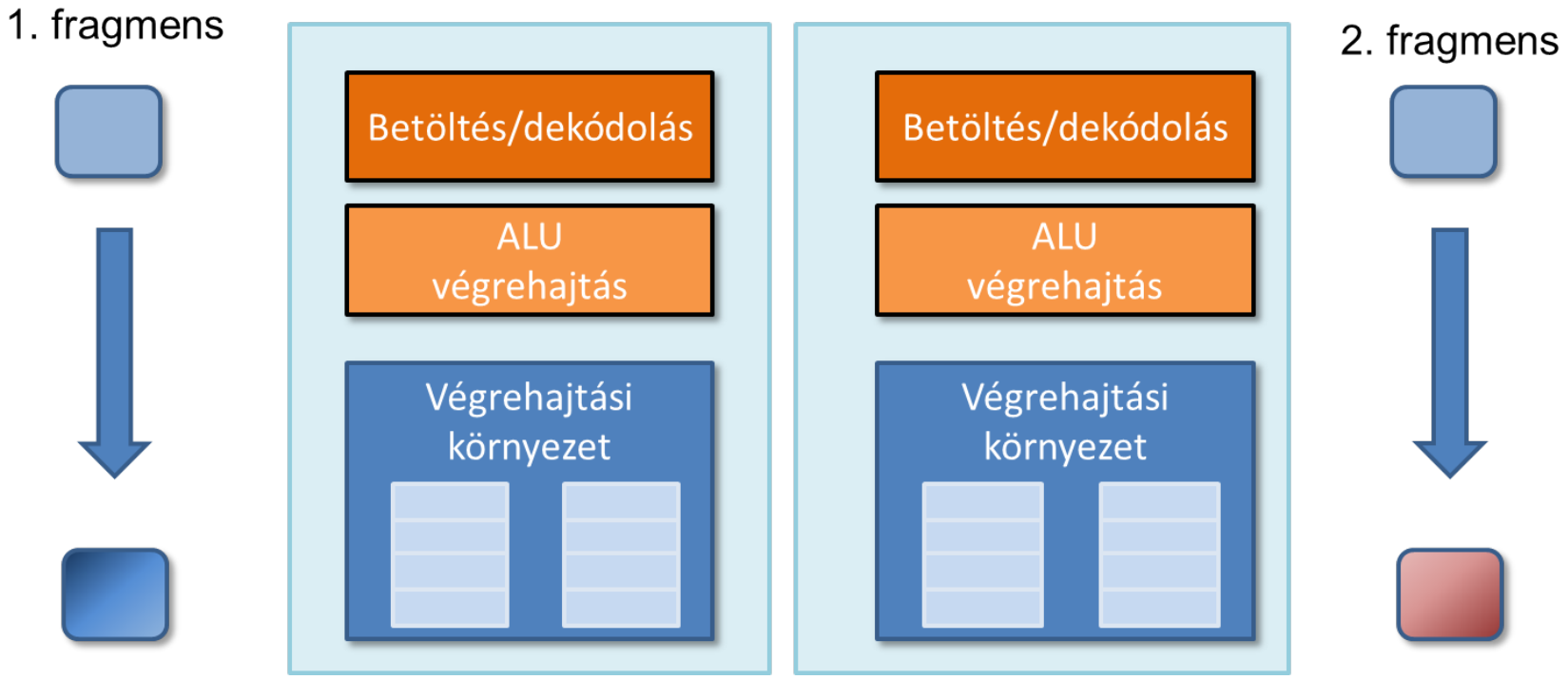
Első ötlet



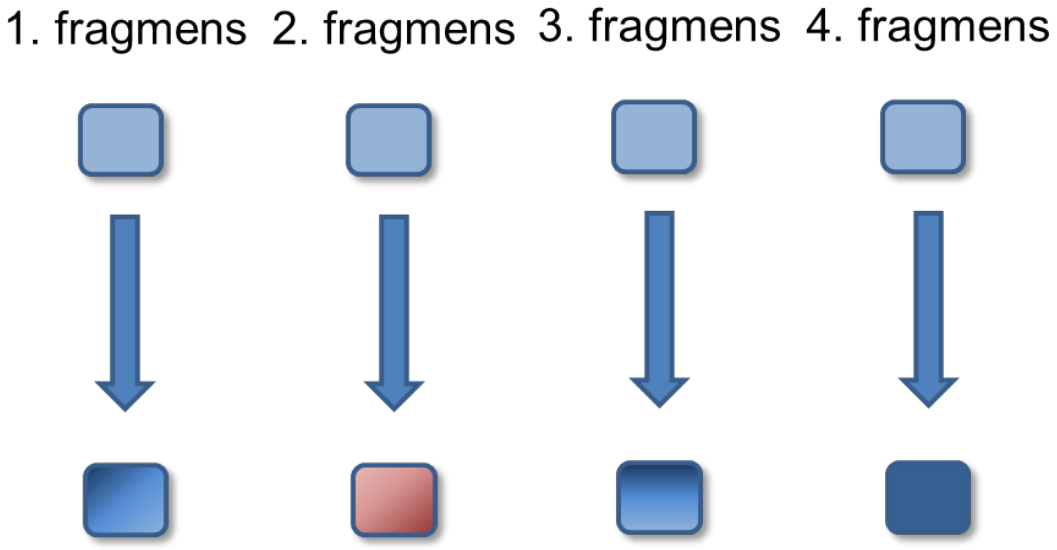
Tüntessük el azokat a komponenseket, amelyek az egyetlen utasítás folyam gyors futását segítik!

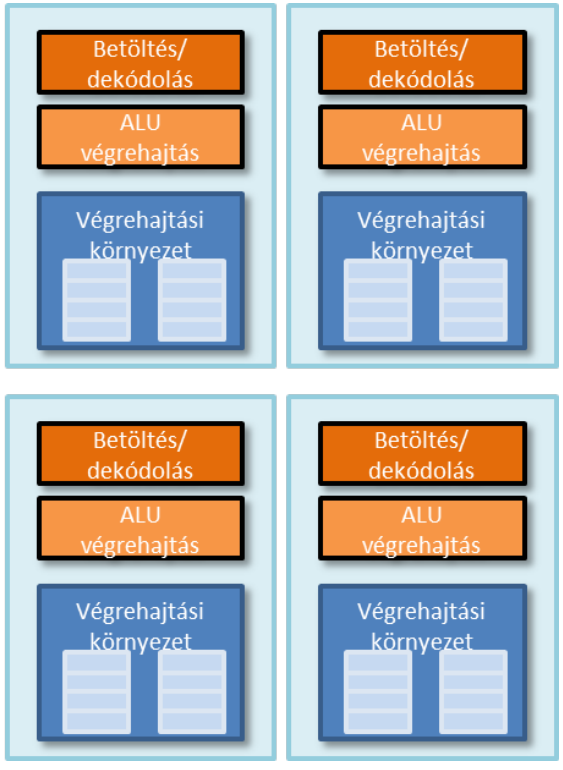
Két mag (core)

Két fragmens párhuzamos feldolgozása



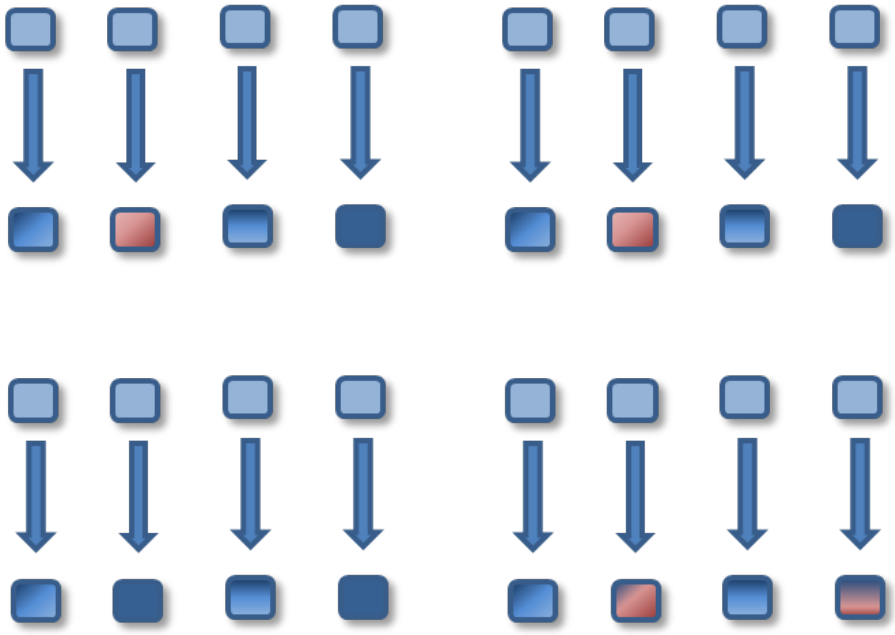
Négy mag (core)
Négy fragmens párhuzamos feldolgozása





Tizenhat mag (core)

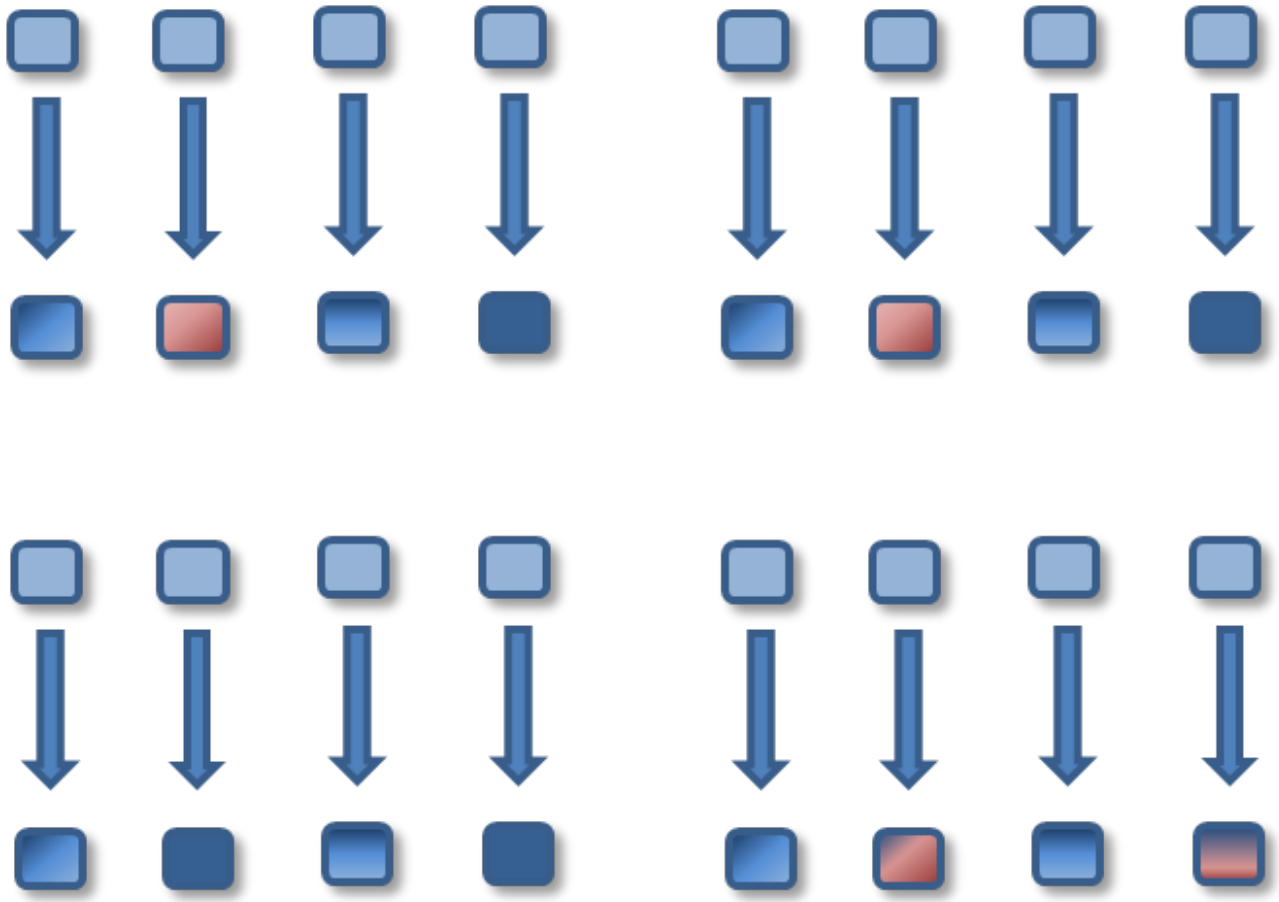
Tizenhat fragmens párhuzamos feldolgozása



16 mag = 16 egyidejű utasítás folyam

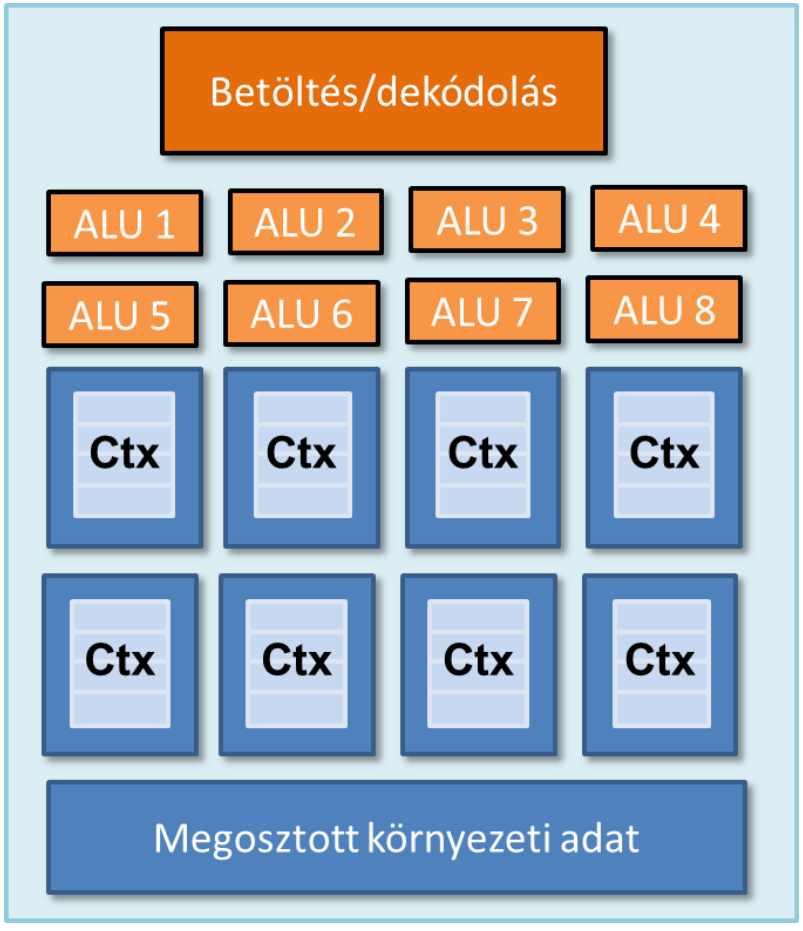
Második ötlet

Fragmensek közötti utasítás folyam megosztása



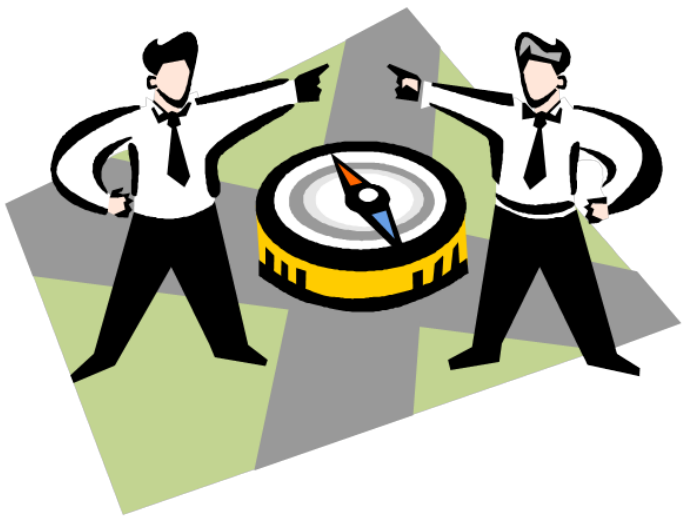
Második ötlet
Single Instruction Multiple Data (SIMD)

- Csökkentsük az ALU-k közötti utasítás folyam
  - Kezelésének költségét
  - Összetettségét



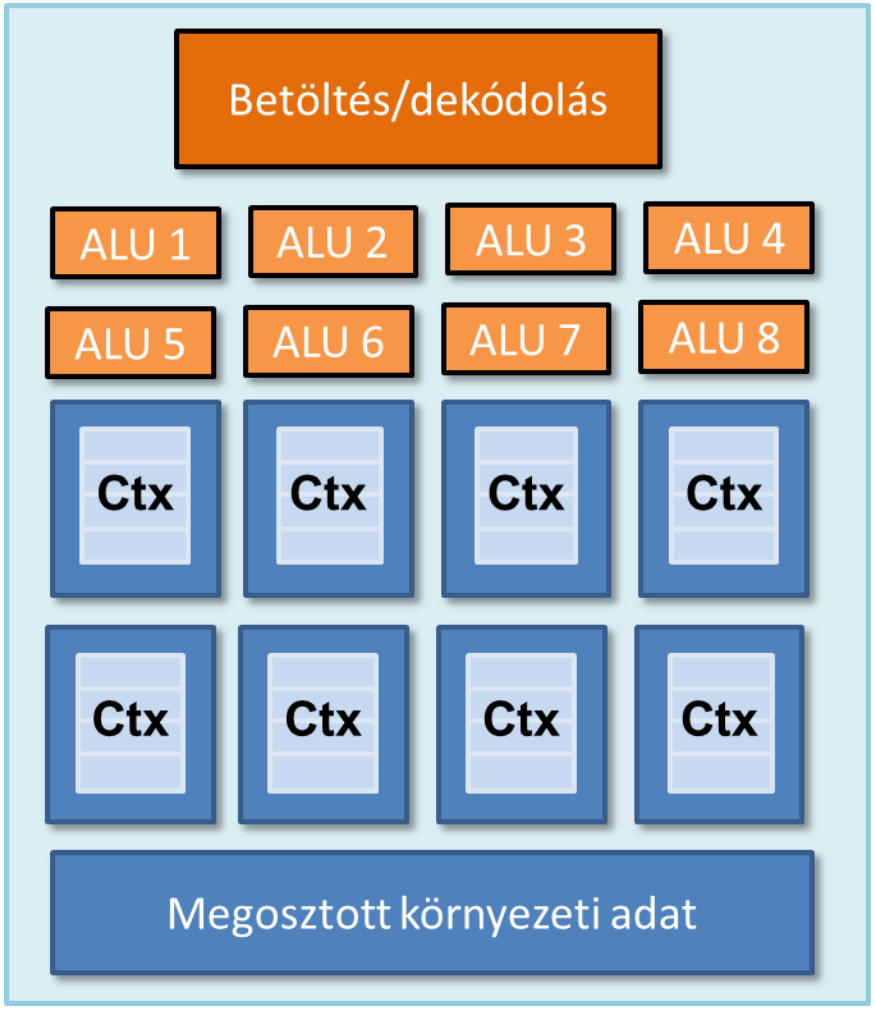
Második ötlet
Shader módosítása

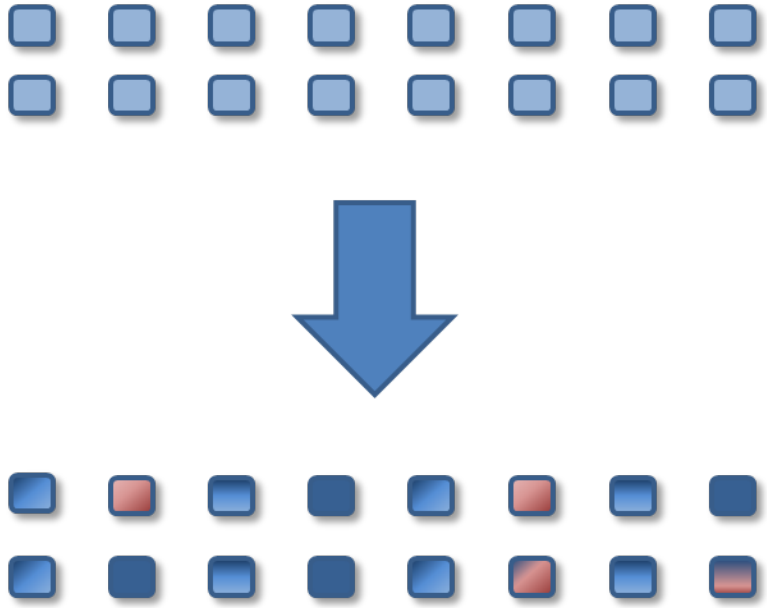
- Előző shader egy fragmenst dolgozott fel
  - Skalár műveletek
  - Skalár operandusok
- Új shader nyolc fragmenst dolgoz fel
  - Vektor műveletek
  - Vektor operandusok



Második ötlet

Fragmensek közötti utasítás folyam megosztása

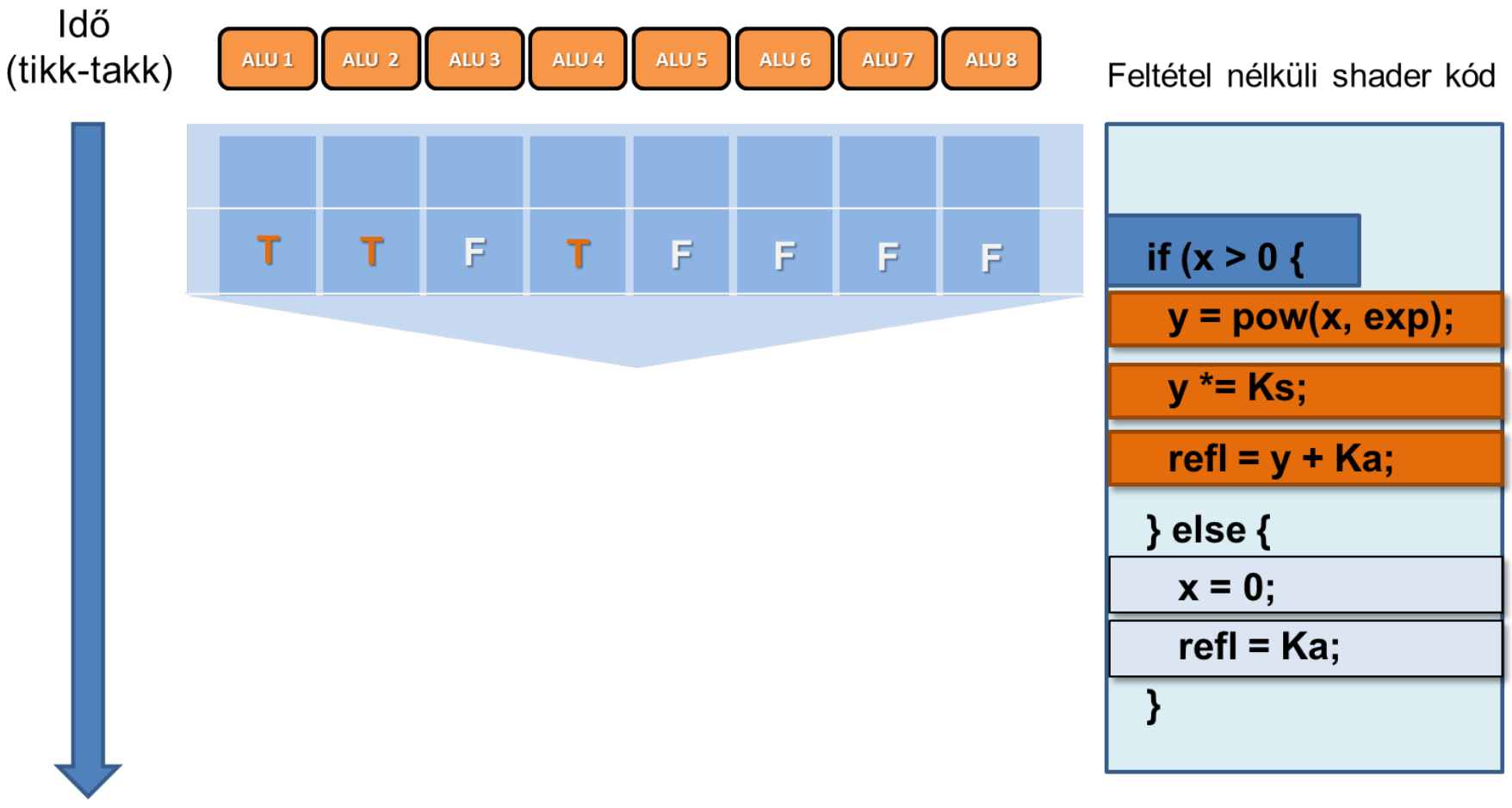




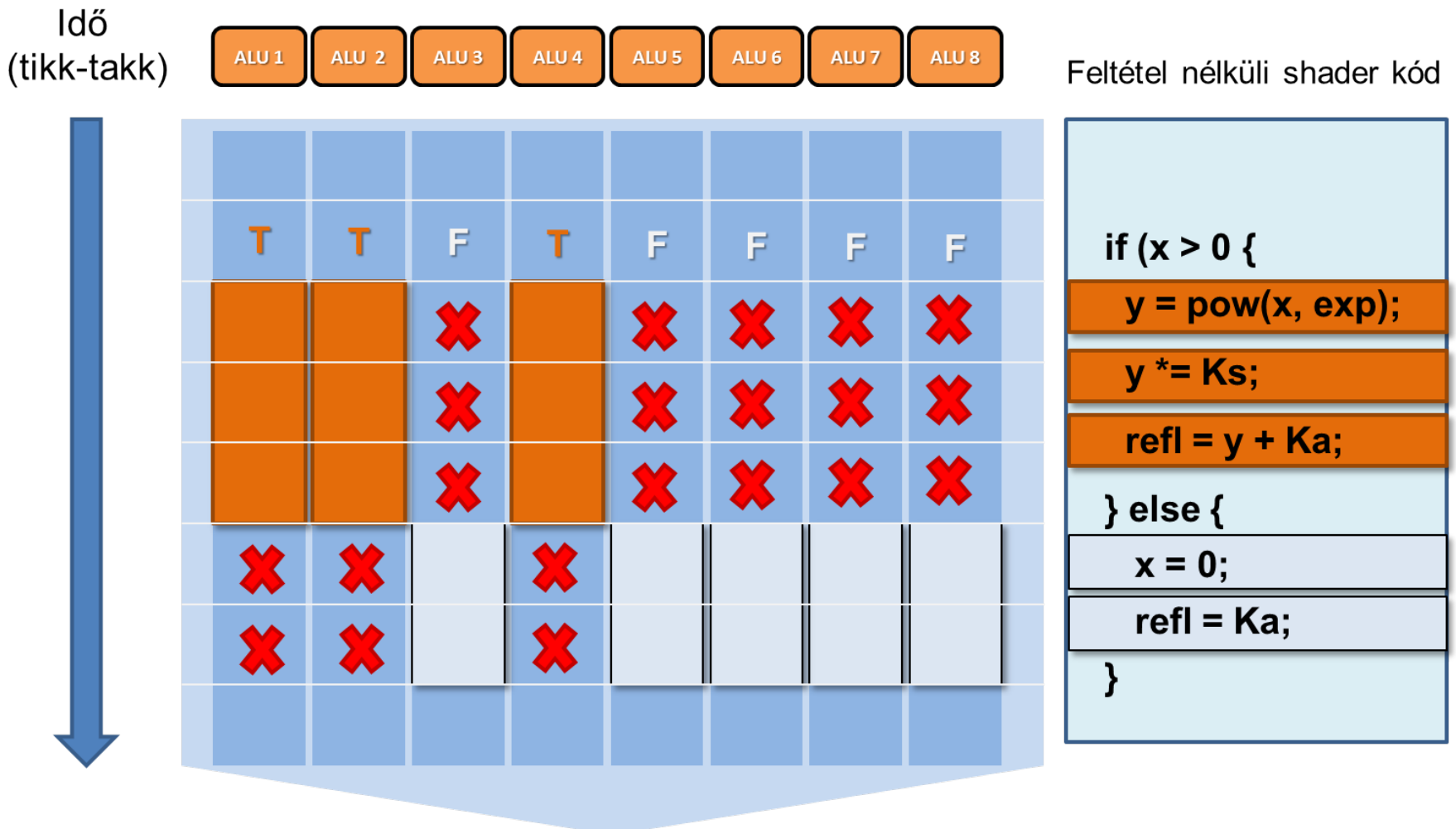
128 fragmens feldolgozása párhuzamosan

- 16 mag = 128 ALU
- 16 egyidejű utasításfolyam
- 128
  - Vertex
  - Primitív
  - Fragmens

Mi van az elágazásokkal?



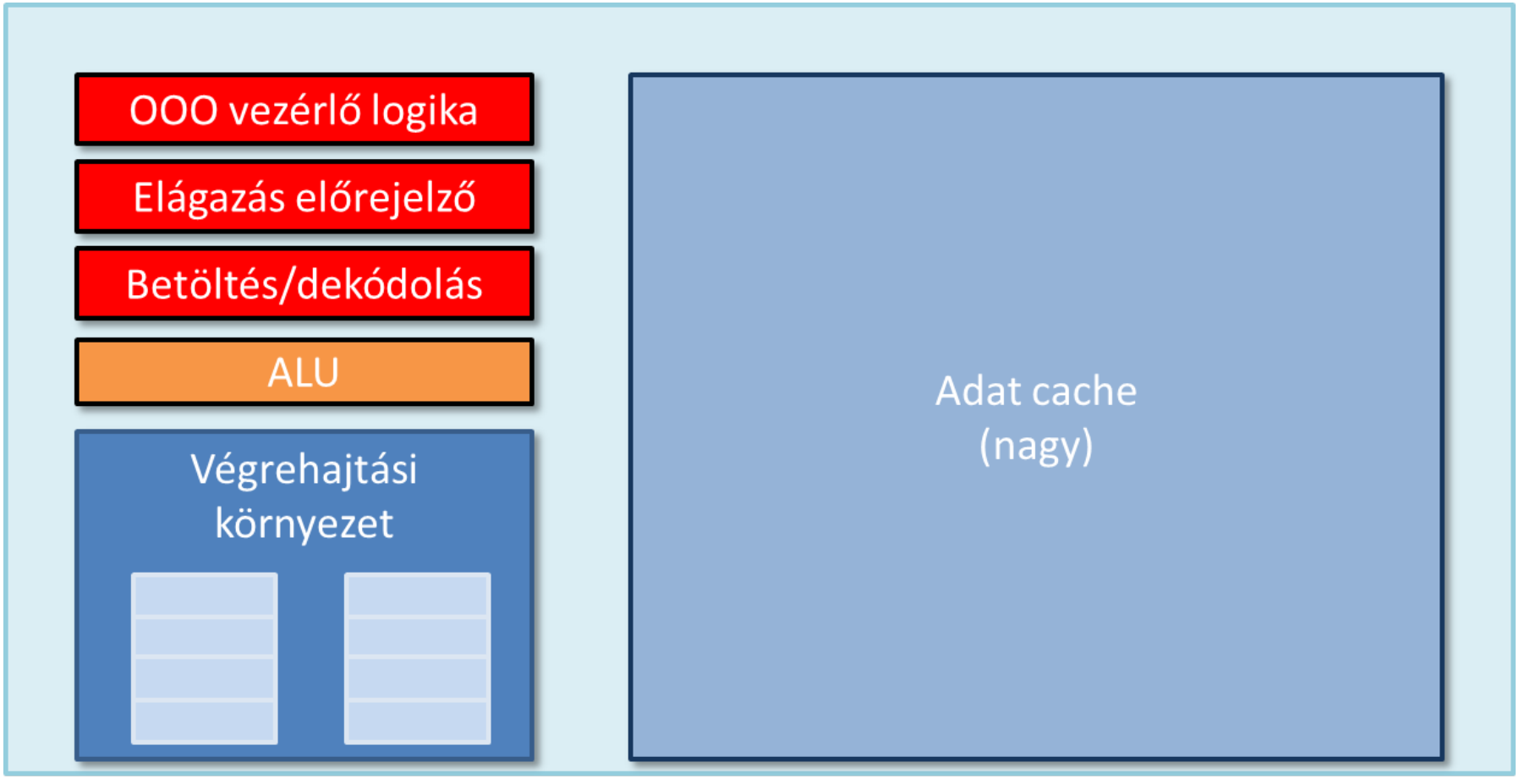
Mi van az elágazásokkal?
Nem mindegyik ALU végez hasznos munkát



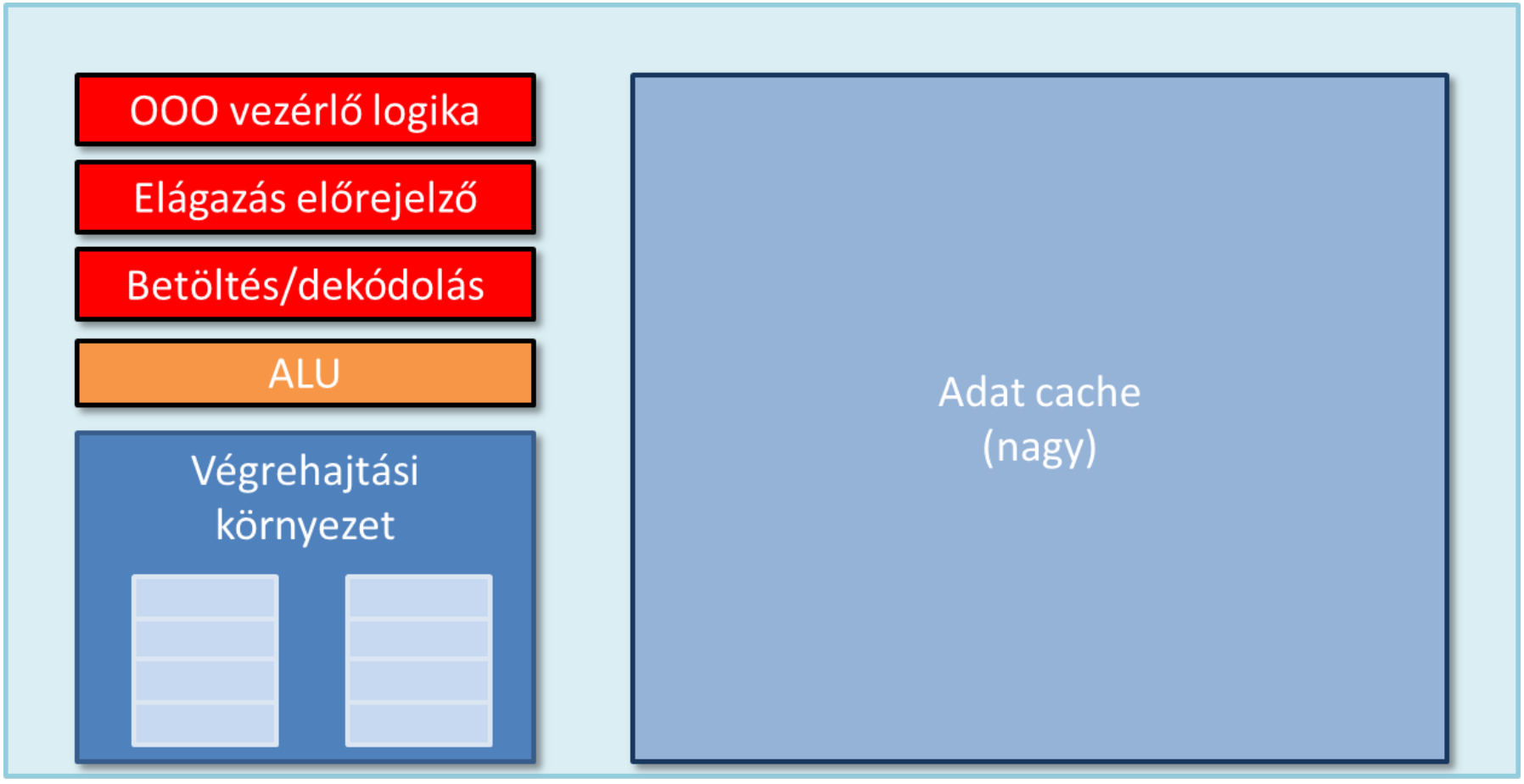
Állások (stalls)

- Állás akkor következik be, amikor egy mag (core) nem tudja futtatni a következő shader utasítást, mivel egy előző utasításra várakozik
  - Függőségek vannak az utasítás folyamban
    - Pl. ADD függ a LOAD befejezésétől
- Késleltetés
  - Adat elérése a memóriából sokszor 1000-nél több ciklust igényel
  - Rossz ötlet volt az első egyszerűsítés?
    - Az eltávolított részek segítenének az állások megoldásában
  - A GPU-k sok független feladatot tételeznek fel
    - Független SIMD csoportok

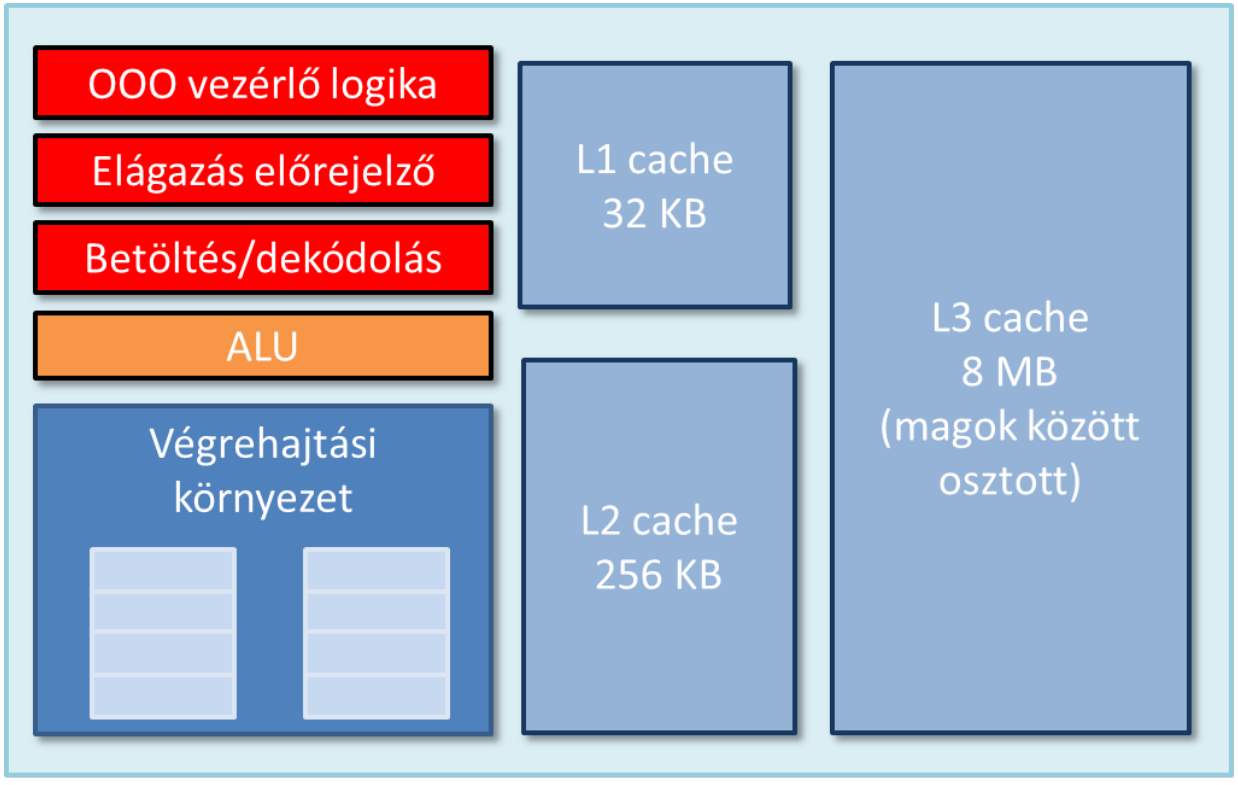
CPU-stílusú magok (core)



CPU-stílusú magok (core)



CPU-stílusú memória felépítés



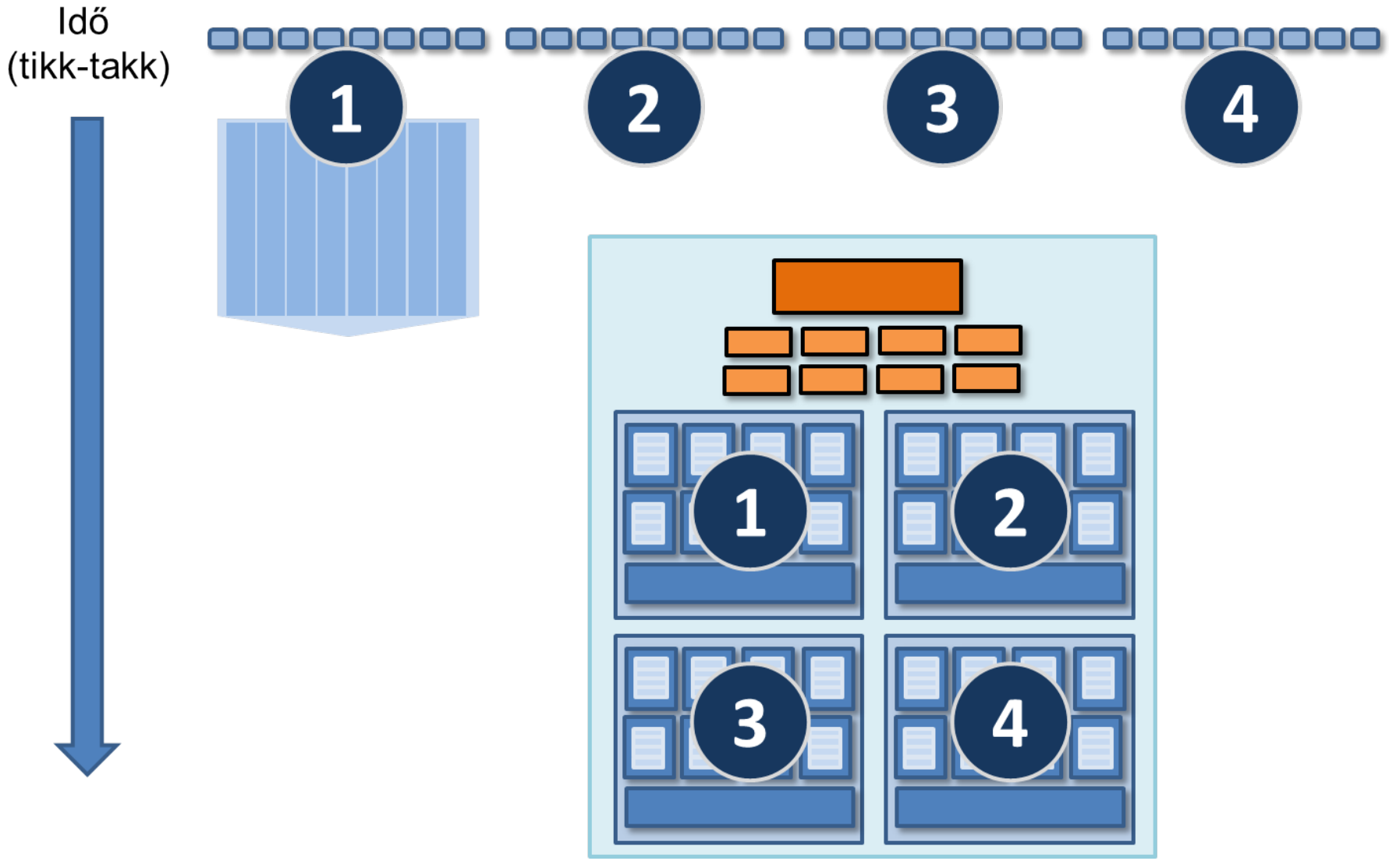
25 GB/sec elérés a memóriához

Magok hatékony kihasználása cache-ben lévő adatok esetén (késleltetés csökkentése, nagy sávszélesség)

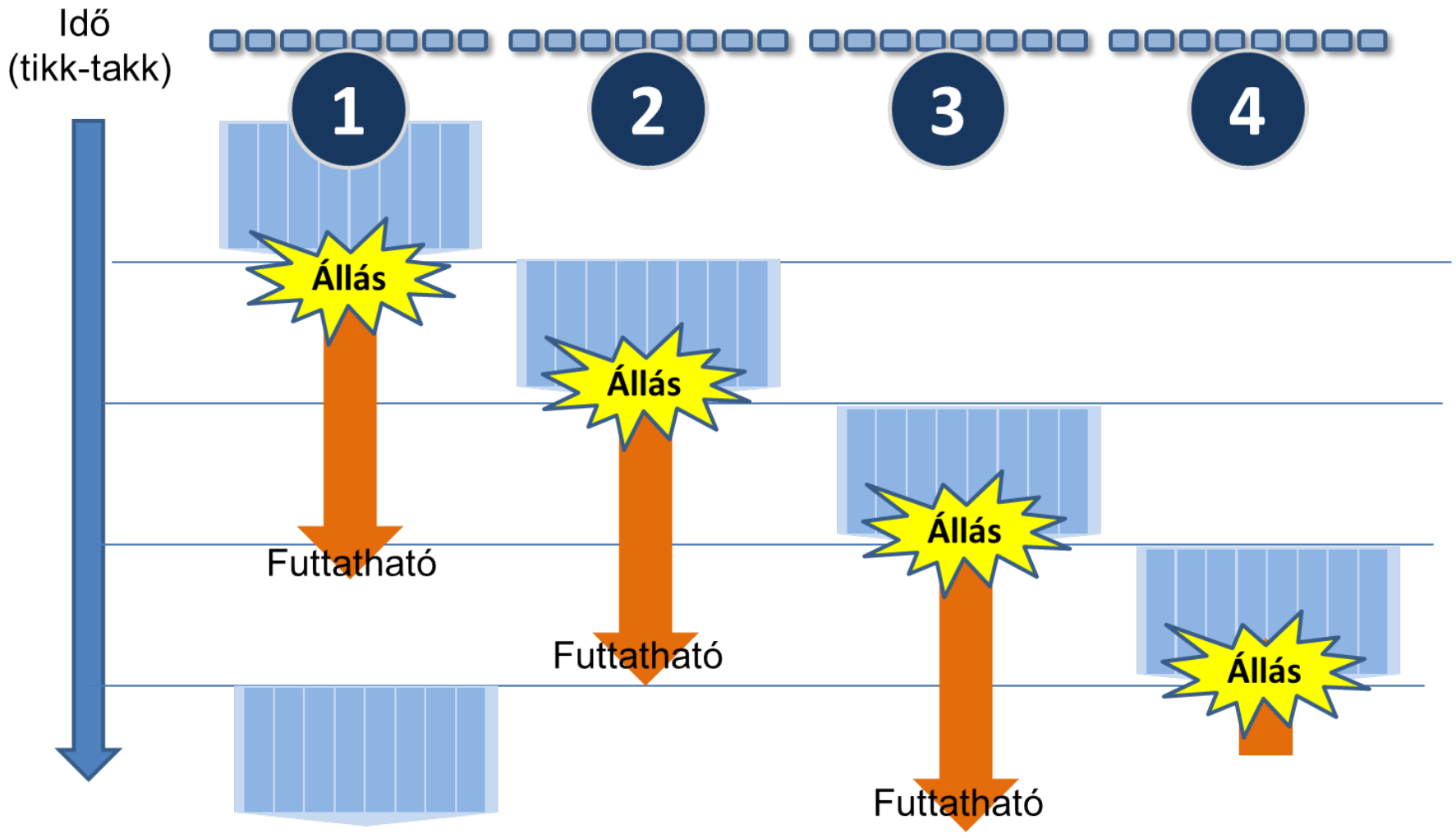
Harmadik ötlet

- Sok független fragmensünk van
- Sok fragmens összefésült feldolgozása egy magon
  - Utasítás folyam váltás egy másik (nem álló) SIMD csoportra, ha az aktív csoport áll
  - GPU hardveresen kezeli
    - Overhead mentesen
    - Ideális esetben teljesen láthatatlan
    - Maximális áteresztőképesség

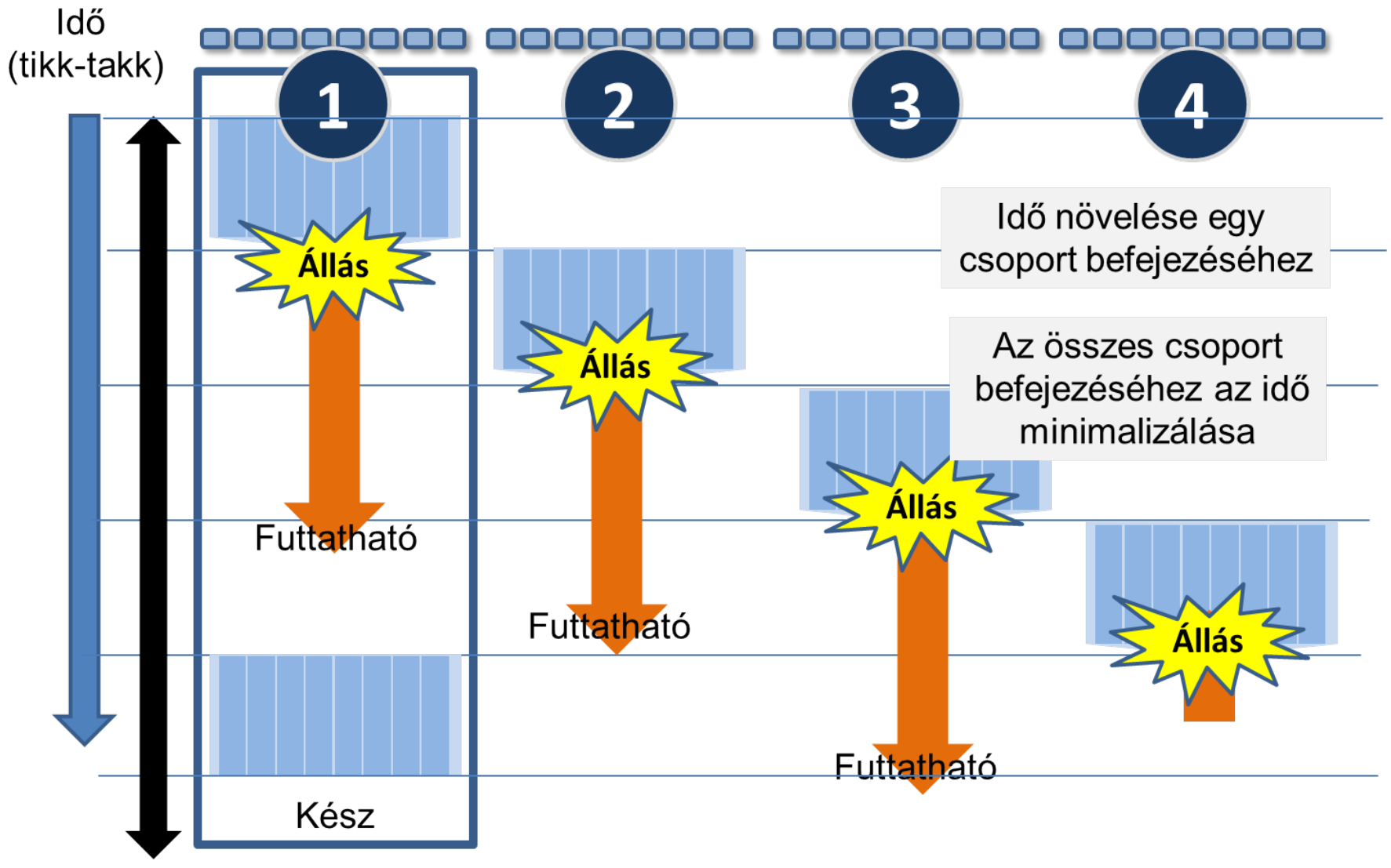
Shader állások elrejtése



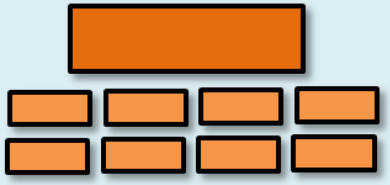
Shader állások elrejtése



Shader állások elrejtése



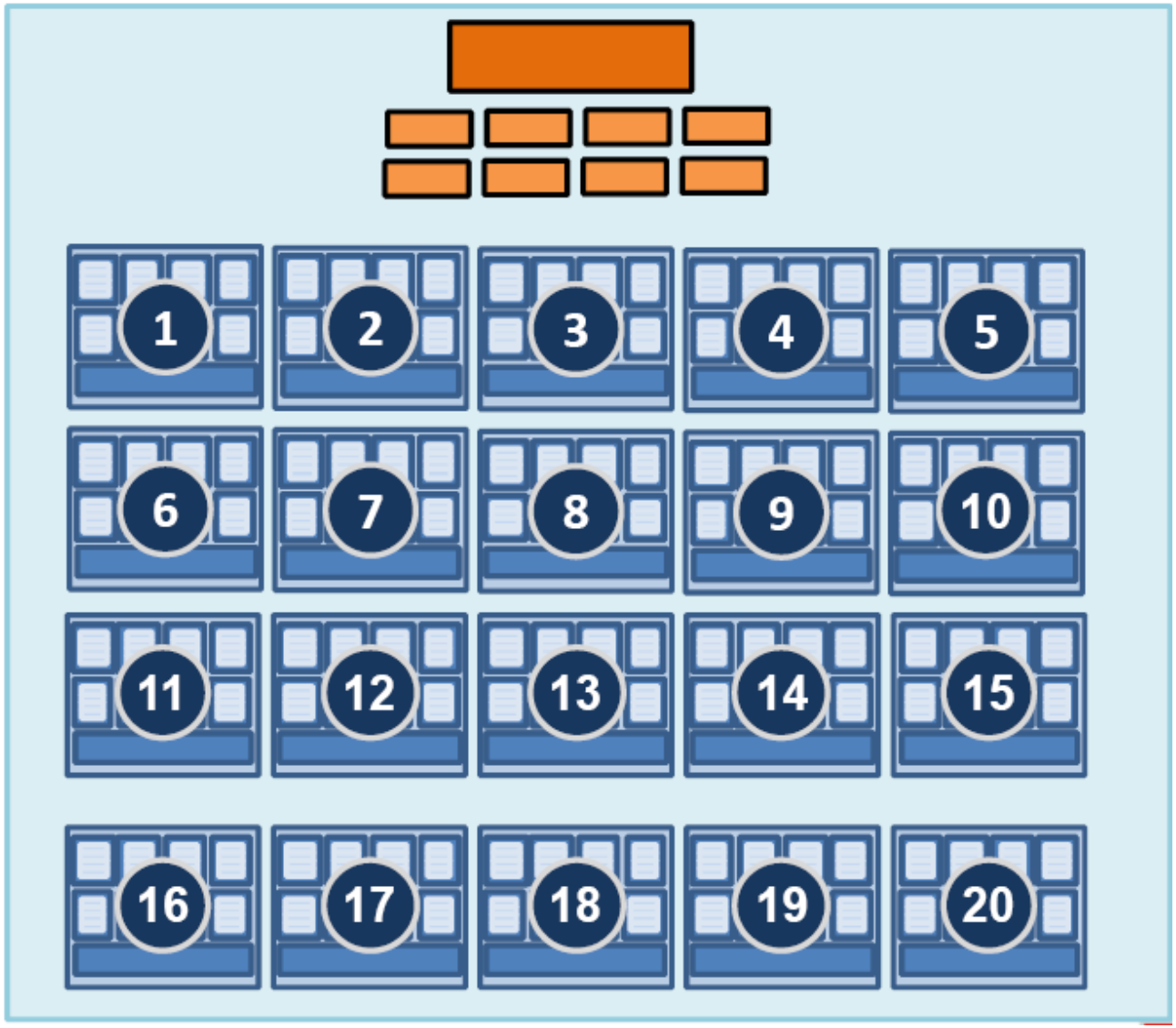
Közös környezet tárolása



Közös környezet készlet tárolás
64 KB

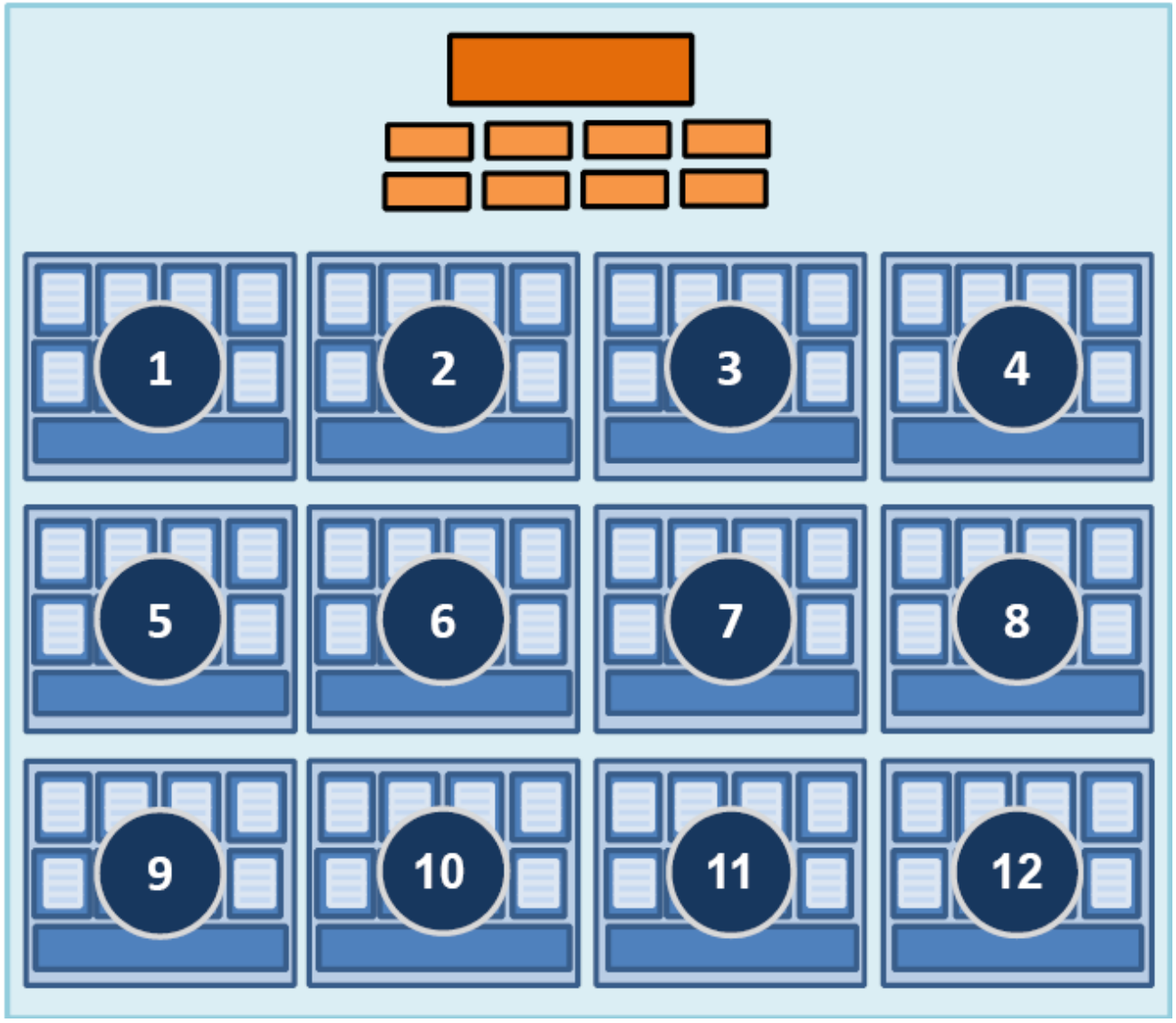
Maximális késleltetés elrejtési képesség

Húsz kicsi környezet



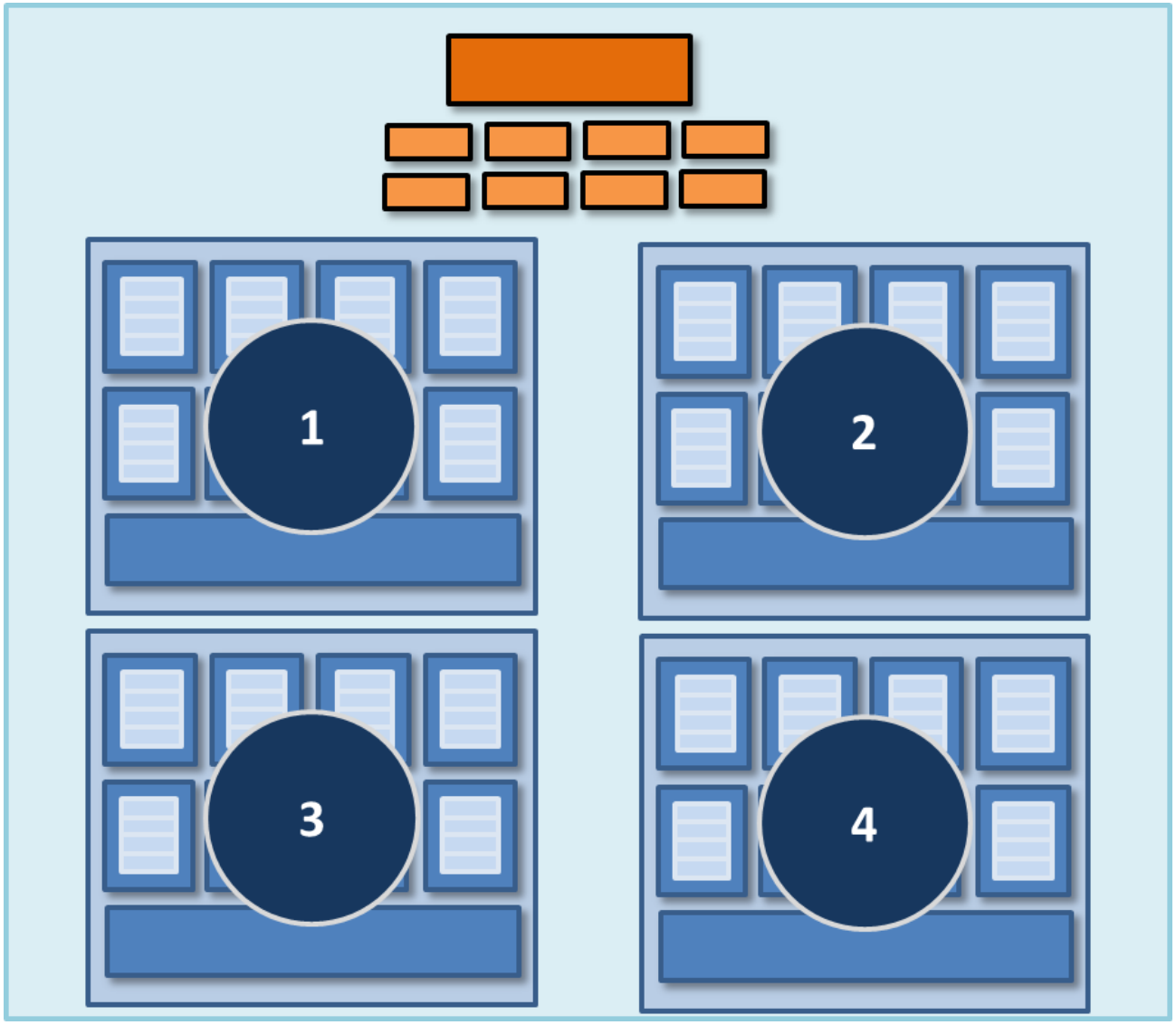
Közepes késleltetés elrejtési képesség

Tizenkét kicsi környezet



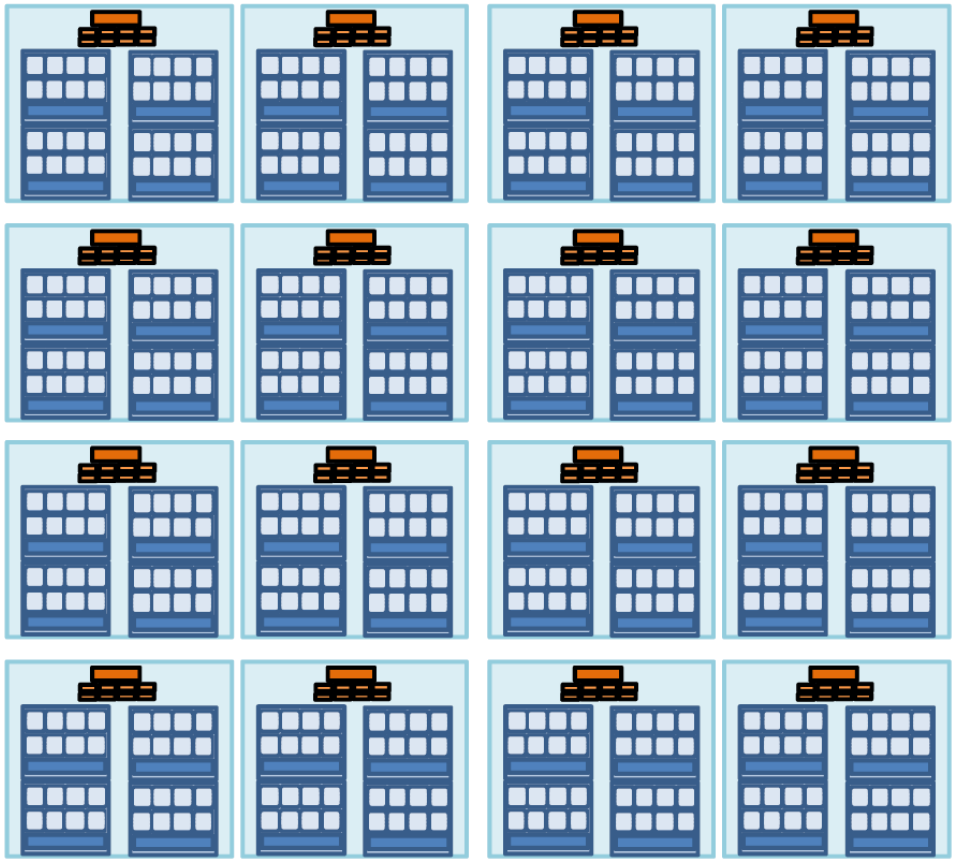
Kicsi késleltetés elrejtési képesség

Négy nagy környezet



GPU shading rendszer

- 16 mag
- 8 mul-add ALU magonként (128 összesen)
- 16 egyidejű utasítás folyam
- 64 konkurens (összefésült) utasítás folyam
- 512 konkurens fragmens



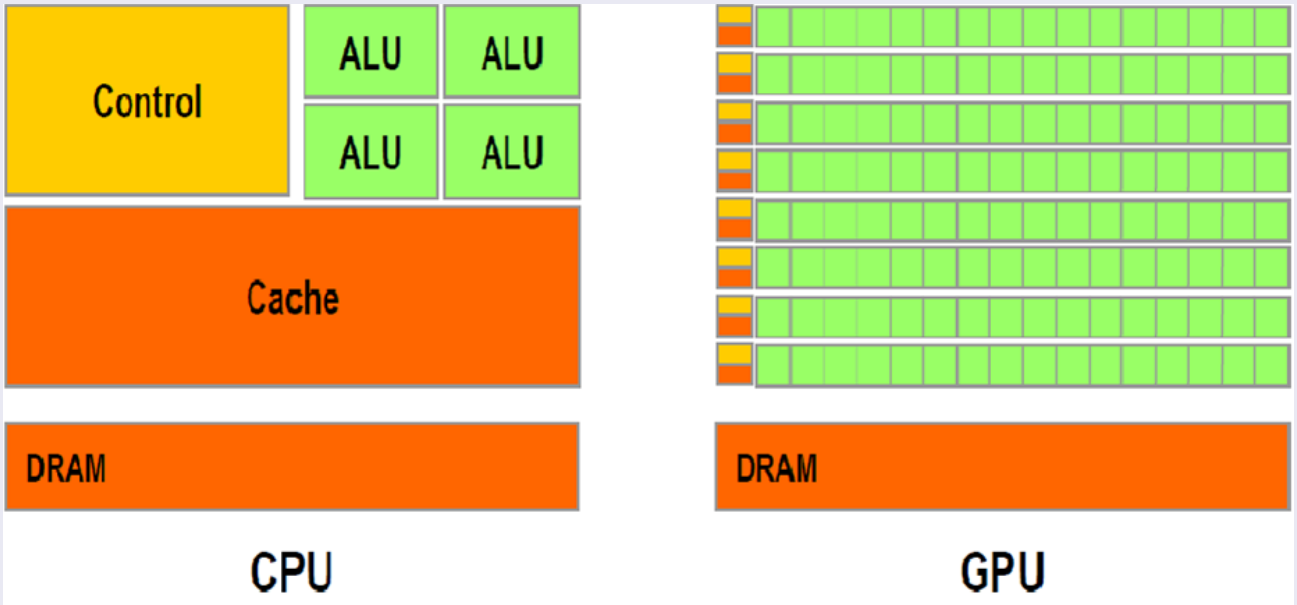
=256 GFLOPS (@1 GHz)

Összefoglalás

Három kulcs ötlet

- Használjunk sok, karcsúsított magot a párhuzamos futtatáshoz
- A magokat rakjuk tele ALU-kkal
  - Megosztott utasítás folyamok fragmensek csoportjainál
- Kerüljük el a késleltetett állásokat fragmens csoportok összefésült végrehajtásával
  - Amikor egy csoport áll, akkor dolgozzunk egy másik csoporton

CPU-GPU összehasonlítás



Összefoglalás

Emlékezzünk a következőkre!

- A GPU-ra egy több magos processzorként gondoljunk, amelyet arra optimalizáltak, hogy
  - A vertex és fragmens adatok maximális „áteresztéssel folynak át” a grafikus csővezetéken
  - Speciálisan támogatja
    - A grafikus csővezeték leképezését ezekre az erőforrásokra
    - A raszterizálást
    - Vágást
    - Hátsó oldal eltávolítást
    - Textúrázást
    - Stb.

- Nagyobb és gyorsabb
  - Több mag
  - Nagyobb FLOPS (manapság 2 TFLOP)
- Milyen fix-funkcióknak kell megmaradnia?
- Néhány CPU-hoz hasonló tulajdonság hozzáadása
  - Általános R/W cache (Fermi)
  - Szinkronizálás

- Alternatív programozási felületek támogatása
  - Általános célú programozás
    - CUDA
    - OpenCL
    - DirectCompute
  - Alkalmazások, amelyek a GPU-t egy több processzoros rendszernek tekintik
- Hogyan változik a grafikus csővezeték absztrakció?
  - Direct3D 11
    - 3 új csővezeték szakasz
- Sugárkövetés

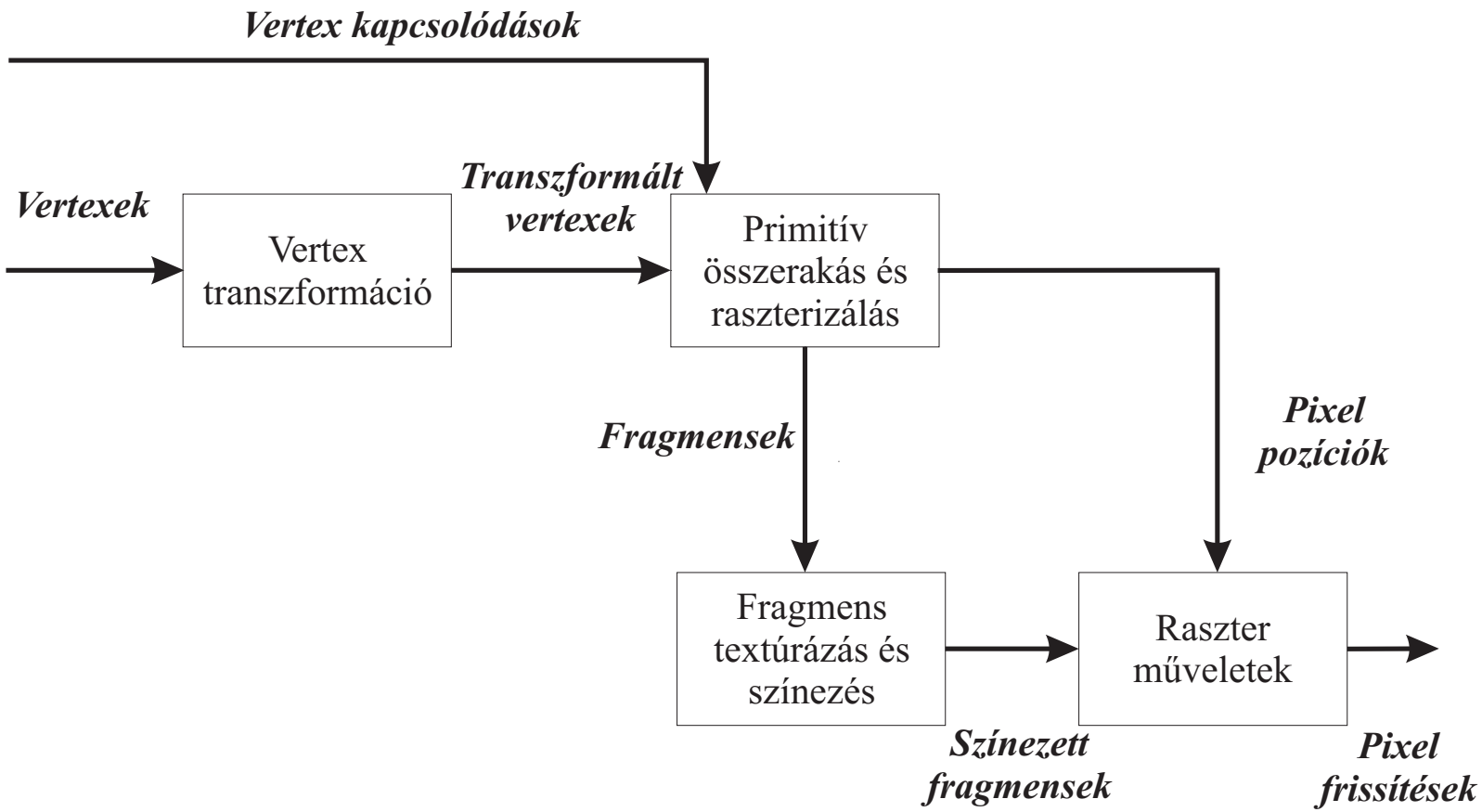
Grafikus csővezeték és az OpenGL függvénykönyvtár

A grafikus csővezeték

- 3D-s színtér objektumainak leírása primitívekkel:
  - pontok,
  - élek,
  - poligonok.
- Primitívek szögpontjait vertexeknek nevezzük
- Adott sorrendben végrehajtott műveletek segítségével áll elő a 2D-s kép
- Műveletek sorrendjét grafikus csővezetéknek nevezzük
  - Rögzített műveleti sorrendű grafikus csővezeték
  - Programozható grafikus csővezeték

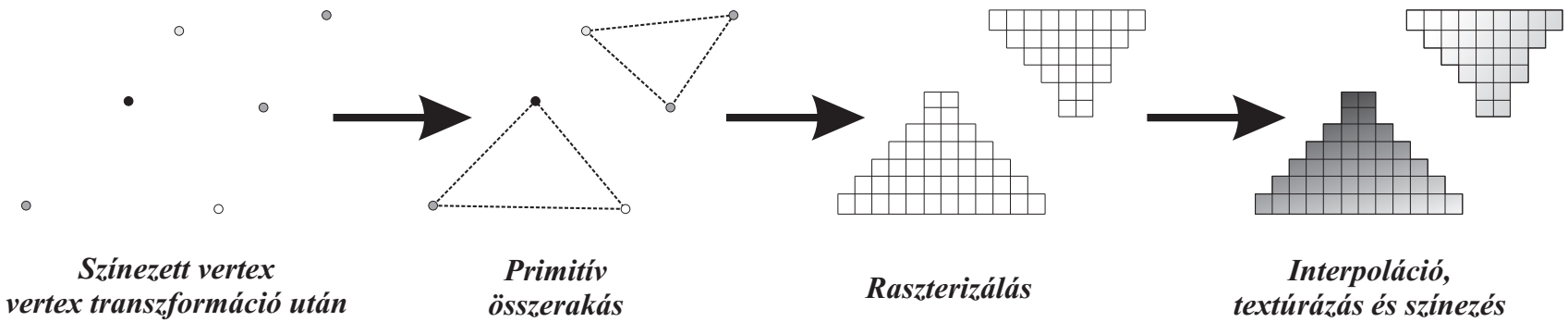
Rögzített műveleti sorrendű grafikus csővezeték

Rögzített műveleti sorrendű grafikus csővezeték



Rögzített műveleti sorrendű grafikus csővezeték

- Vertex transzformációk
  - Matematikai műveletek sorozata
  - Primitívek szögpontjainak meghatározása a képernyőn
  - Vertex attribútumok átadása

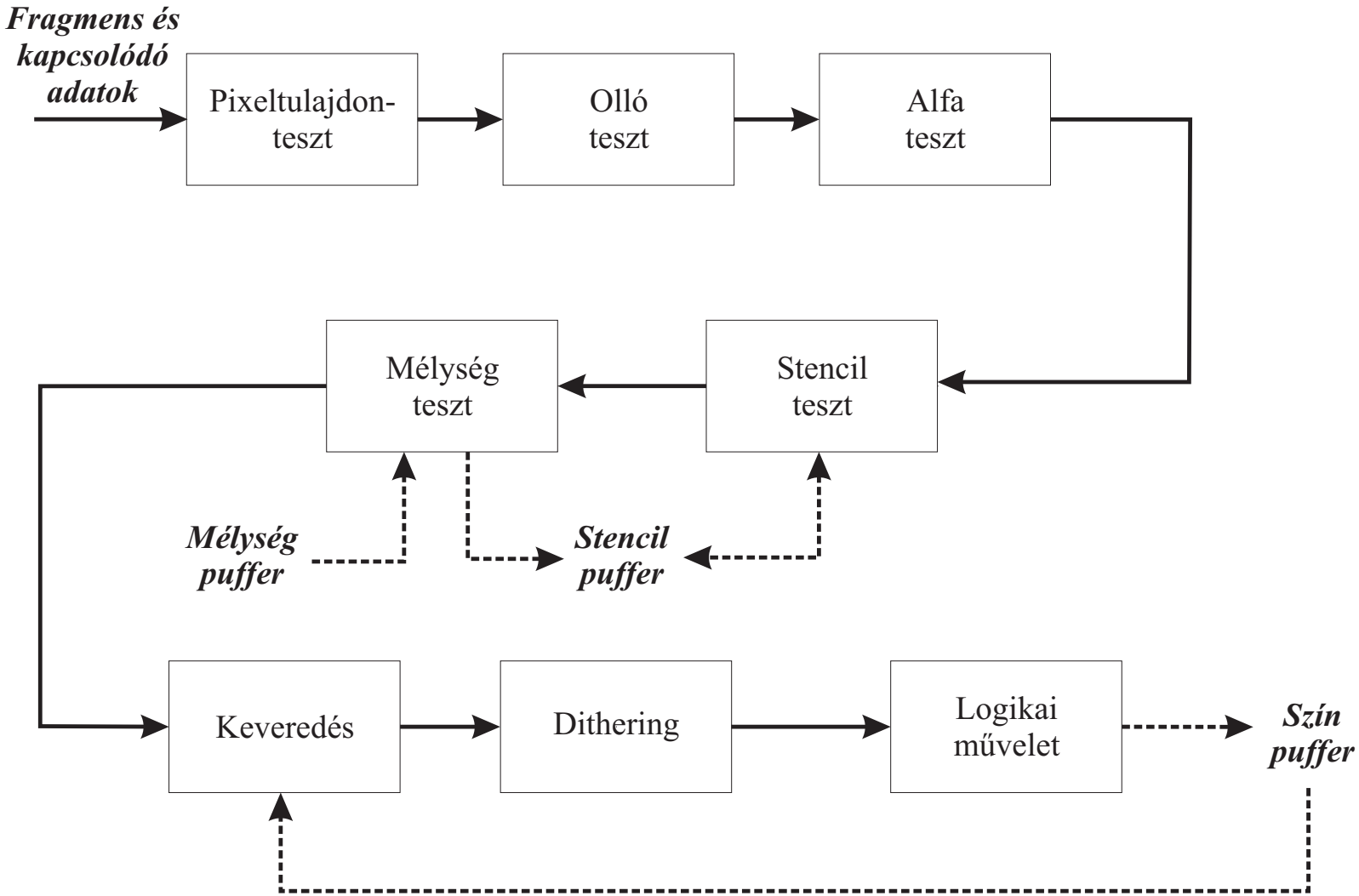


Rögzített műveleti sorrendű grafikus csővezeték

- Primitív összerakás és raszterizálás
  - A vertexek primitívekbe vannak szervezve kapcsolódási információk alapján
  - Vágás
    - A 3D-s színtér látható térfogata
    - Alkalmazás által definiált vágósíkok
    - Raszterizáló eldobhat poligonokat
  - Geometriai primitívek lefedése
  - Fragmensek

Rögzített műveleti sorrendű grafikus csővezeték

- Fragmens textúrázás és színezés
  - Mindegyik fragmensre textúrázás és matematikai műveletek végrehajtása
    - Transzformált vertexekből származó interpolált szín
    - Interpolált textúra koordináták
    - Fragmenshez tartozó texel kinyerése
    - Fragmens szín

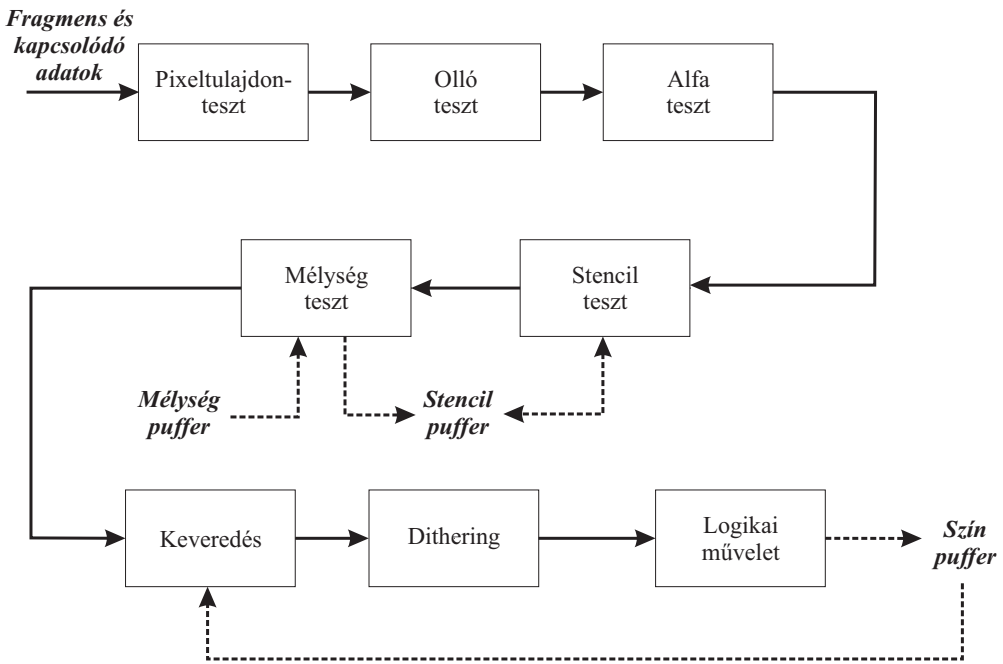

Fragmens és kapcsolódó adatok
Pixeltulajdon-teszt
Olló teszt
Alfa teszt
Mélység teszt
Stencil teszt
Mélység puffer
Stencil puffer
Keveredés
Dithering
Logikai művelet
Szín puffer

- Pixeltulajdon-teszt
  - Képernyő adott pixelére írhatunk-e
- Olló teszt
  - Olló téglalapra korlátozott kirajzolás
- Alfa teszt
  - Fragmens alfa értékének összehasonlítása egy előre megadott értékkel
  - Adott reláció mellett kapott hamis értéknél a fragmens eldobódik

- Stencil-teszt
  - Fragmens pozíciójának megfelelő stencilpuffer érték összehasonlítása egy előre megadott értékkel
  - Ha az összehasonlítás eredménye hamis, akkor a fragmens eldobódik
  - Stencilpuffer értékének módosítása
    - Műveletek megadása sikeres és sikertelen stencil teszt esetén
    - Sikeres stencil teszt és a mélység teszttől függő művelet megadása

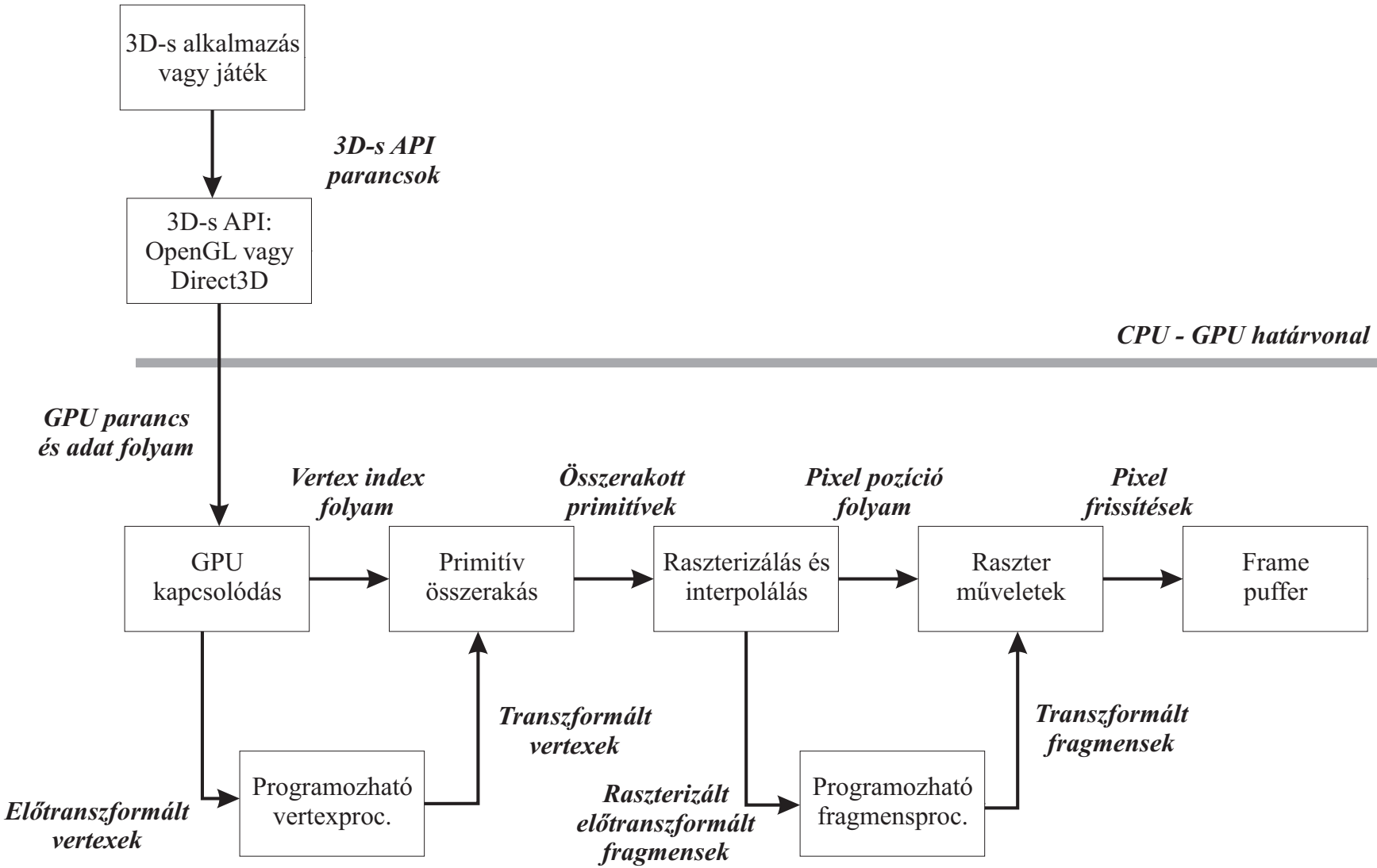
- Mélység teszt
  - Fragmens mélység értékének összehasonlítása a mélységpufferben levő értékkel
  - Sikeres teszt esetén frissül a színpuffer és a mélységpuffer
    - Alap esetben a nézőponthoz közelebbi fragmens fog bekerülni a színpufferbe

- Keveredés: végső fragmens és pixelek egyesítése
- Dithering: a színmélység javítása a térbeli felbontás rovására
- Logikai műveletek: OR, XOR vagy INVERT



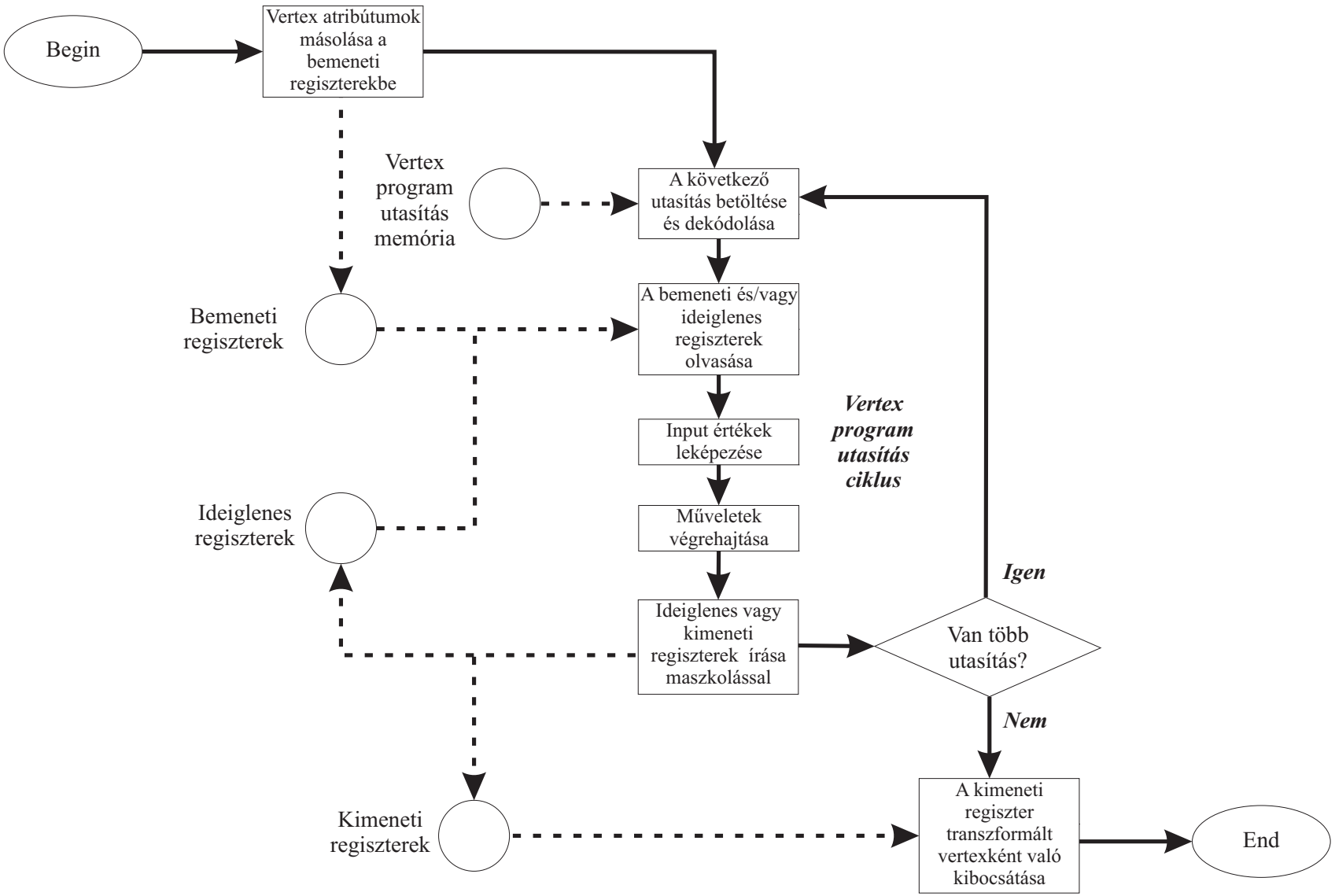
Programozható grafikus csővezeték

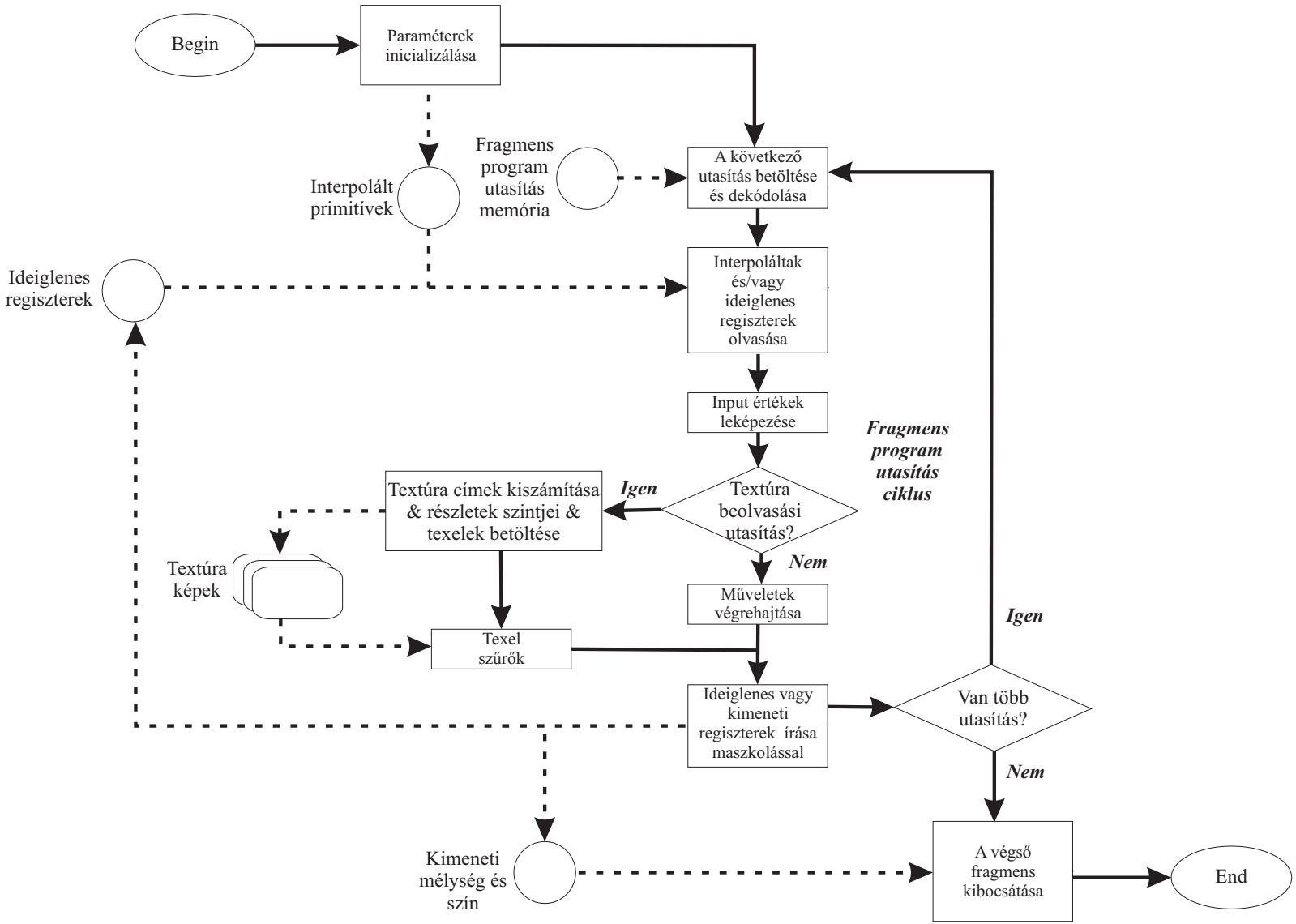
Programozható grafikus csővezeték



Programozható grafikus csővezeték

- Programozható vertex processzor
  - Vertex attribútumok betöltése a megfelelő regiszterekbe
  - Vektorokon végzett matematikai műveletek
  - Fejlettebb vertex processzorok a vezérlési szerkezeteket is támogatják
- Programozható fragmens processzor
  - Hasonló műveletek végrehajtása, mint a vertex processzorok esetén
  - A fragmens processzorok támogatják a textúra műveleteket is

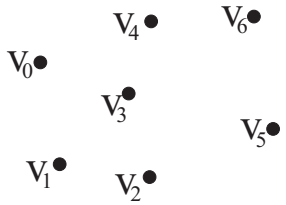

Begin
Vertex atribútumok másolása a bemeneti regiszterekbe
Vertex program utasítás memória
A következő utasítás betöltése és dekódolása
Bemeneti regiszterek
A bemeneti és/vagy ideiglenes regiszterek olvasása
Input értékek leképezése
Vertex program utasítás ciklus
Ideiglenes regiszterek
Műveletek végrehajtása
Igen
Ideiglenes vagy kimeneti regiszterek írása maszkolással
Van több utasítás?
Nem
Kimeneti regiszterek
A kimeneti regiszter transzformált vertexként való kibocsátása
End


Begin
Paraméterek inicializálása
Interpolált primitívek
Fragmens program utasítás memória
A következő utasítás betöltése és dekódolása
Ideiglenes regiszterek
Interpoláltak és/vagy ideiglenes regiszterek olvasása
Input értékek leképezése
Fragmens program utasítás ciklus
Textúra címek kiszámítása & részletek szintjei & texelek betöltése
Igen
Textúra beolvasási utasítás?
Textúra képek
Nem
Műveletek végrehajtása
Igen
Texel szűrők
Ideiglenes vagy kimeneti regiszterek írása maszkolással
Van több utasítás?
Nem
Kimeneti mélység és szín
A végső fragmens kibocsátása
End

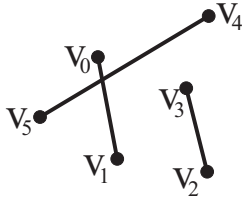
OpenGL függvénykönyvtár

OpenGL függvénykönyvtár

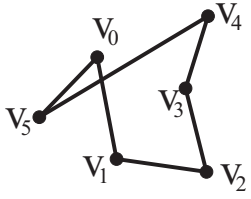
- Hordozható, 3D-s grafikus függvénykönyvtár
- Több száz függvényt és definíciót tartalmaz
- Egy színtér leírásához OpenGL függvény hívások sorozatát kell megadni
- Vertex
  - OpenGL primitívek szögpontjai
  - 2D-s és 3D-s pozíciók
  - Meghatározzák a primitív alakját és helyzetét



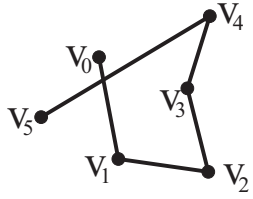
Pontok



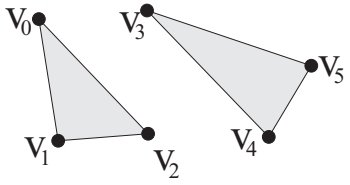
Vonalak



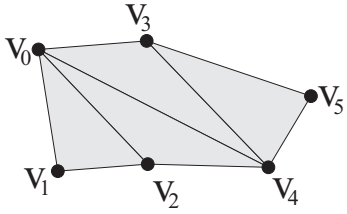
Vonal hurok



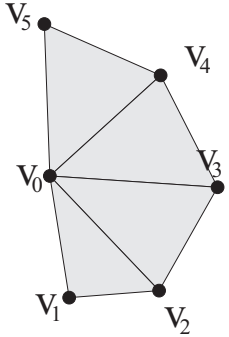
Töredezett vonal



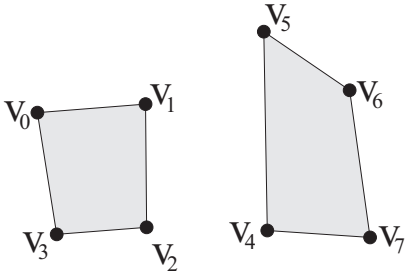
Háromszögek



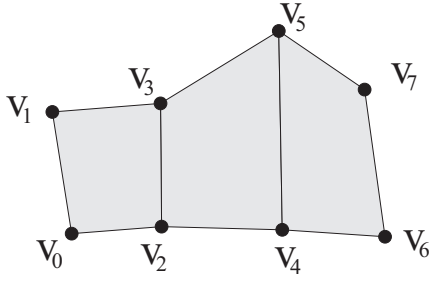
Háromszögsáv



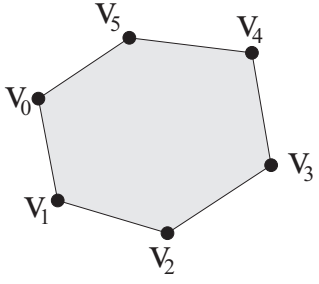
Háromszög-legyező



Négyszögek



Négyszögsáv



Poligon

OpenGL függvénykönyvtár

- Az OpenGL függvénykönyvtár támogatja a
  - Megvilágítást
  - Árnyalást
  - Textúrázást
  - Keveredést
  - Átlátszóságot
  - Más speciális hatásokat és képességeket
- Az OpenGL függvénykönyvtár nem tartalmaz
  - Ablakkezelő függvényeket
  - Felhasználói interaktivitást és I/O műveleteket végrehajtó függvényeket
- Nincs OpenGL file formátum
  - A modellek tárolására
  - A virtuális színtér tárolására

OpenGL adattípus	Belső reprezentáció	C adattípusként definiálva
`GLbyte`	8 bites egész	`signed char`
`GLshort`	16 bites egész	`short`
`GLint`, `GLsizei`	32 bites egész	`long`
`GLfloat`	32 bites lebegőpontos	`float`
`GLclampf`	pont	
`GLuint`, `GLenum`, `GLbitfield`	32 bites előjel nélküli egész	`unsigned long`

<Könyvtár prefix><Alap parancs><Opcionális argumentum szám><Opcionális argumentum típus>

gl Color 3 f

```
glColor3f(0.5, 0.5, 0.5);
```

OpenGL függvénykönyvtár

- Platformfüggetlenség
  - Operációs rendszerekhez kapcsolódó feladatok
    - Ablakkezelés
    - Felhasználói interakciók kezelése
    - Felhasználó leütötte-e az Enter billentyűt?
- GLUT használata
  - Kezdetekben AUX (auxiliary) lib
  - Kereszt-platformos példák szemléltetése
  - Pop-up menük, ablakok, joystick támogatás, stb.
  - GUI programozása adott platformon

```
#include <GL/glut.h>

// a színtér rajzolása
void RenderScene(void)
{
// Az aktuális törlő színnel való ablak törlés
   glClear(GL_COLOR_BUFFER_BIT);

// Flush rajzoló parancs
   glFlush();
}

// A renderelési állapotok beállítása
void SetupRC(void)
{
//A színpuffer törlőszínének a beállítása
   glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
```

```
// A program belépési pontja
int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutCreateWindow("Egyszeru");
  glutDisplayFunc(RenderScene);
  SetupRC();
  glutMainLoop();
  return 0;
}
```

```
glutInit(&argc, argv);
```

- Továbbítja a parancssori paramétereket
- Inicializálja a GLUT függvénykönyvtárat

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
```

- Egyszeresen pufferelt ablak
- RGBA színmód

```
glutCreateWindow("Egyszeru");
```

- Ablak létrehozása
- Címsorában az "Egyszeru" felirat

```
glutDisplayFunc(RenderScene);
```

- RenderScene callback függvény regisztrálása
- Ablak újrarajzolása
  - Ablak első megjelenítésekor
  - Ablak előtérbe helyezésekor
- OpenGL renderelési függvények hívása

`SetupRC();`

- Renderelése előtti inicializálás
- OpenGL állapotok beállítása

`glutMainLoop();`

- GLUT eseménykezelő elindítása
- Vezérlés átadása a GLUT-nak
- Fő ablak bezárásig nem tér vissza
- Üzenetek feldolgozása

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
```

- Ablak törlésére használt szín megadása
- A színpuffer inicializálására használt szín beállítása

```
void glClearColor(GLclampf red, GLclampf green,
    GLclampf blue, GLclampf alpha);
```

- `GLclampf`: 0 és 1 közé leképezett `float`
- A szín vörös, zöld és kék összetevők keverékeként való megadása
- `alpha`: keveredés és speciális hatások

```
glClear(GL_COLOR_BUFFER_BIT);
```

- A színpuffer törlése
- Pufferek törlése

```
glFlush();
```

- OpenGL parancssor ürítése
- Nem vár további utasításokra
- Beérkezett utasítások feldolgozásának folytatása

Összefoglalás

Grafikus csővezeték

- Műveletek meghatározott sorrendje
- Adatok áramlása egyik fázisból a másikba
  - Meghatározott típusú be- és kimenő adatok
  - SIMD
- Programozható grafikus csővezeték
  - Rögzített műveleti sorrendű grafikus csővezeték „kiegészítése”
  - Programozható vertex és fragmens processzor

Összefoglalás

OpenGL függvénykönyvtár

- 3D-s grafikus függvénykönyvtár
- Színtér leírása függvények meghívásával
- Rögzített műveleti sorrendű grafikus csővezeték
- GLUT
- Első program

Cg alapismeretek

- Mi a Cg?
  - C for graphics
- Programozható grafikus hardvert használva
  - Alakzat, megjelenés, mozgásának vezérlése
  - Nagy sebességgel
- Programozási platform
  - Könnyű használni
  - Gyors speciális effekt előállítás
  - Valós idejű mozi minőségű élmény biztosítása

- Nem szükséges a grafikus hardver assembly szintű programozása
  - OpenGl, DirectX, Windows, Linux, Macintosh, Xbox
- CG fejlesztése
  - Microsofttal közösen
  - Kompatibilis
    - OpenGL API
    - High-Level Shading Language (HLSL) DirectX

- A Cg különbözik a C, C++ és Java nyelvektől
  - Nagyon speciális
- Shading/árnyékoló nyelv
  - Fizikai szimuláció és más nem árnyékoló feladatok
  - Cg program
    - Részletes recept egy objektum renderelésére a grafikus hardvert használva

Cg adatfolyam modell

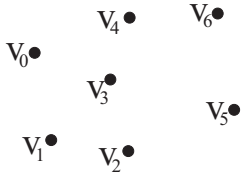
- Grafikára specializálódott
  - Különbözik a többi konvencionális nyelvtől
    - Feldolgozási lépések sorozata az adatokon
  - Vertex-eken és fragmenseken hajt végre műveleteket
  - Minden időpillanatban, amikor egy vertex feldolgozódik vagy egy fragmens jön létre a raszterizálás alatt
    - Vertex/fragmens bemenet
    - Kimenet vertex/fragmens

- CPU
  - Általános célú
  - Alkalmazások végrehajtása
    - C++, JAVA
- GPU
  - Grafikus alkalmazásokra
    - 3D-s színtér
  - Nem képes általános célú programot végrehajtani
    - Speciális
    - Nagy teljesítmény
  - Cg: absztrakt végrehajtási modell

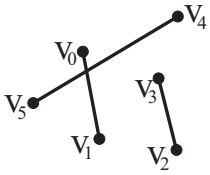
- CPU
  - Helyes program
  - Le lehet fordítani
  - Végre lehet hajtani
    - Operációs rendszer
- GPU
  - Hardware profile-ok
    - Nem minden Cg programot lehet lefordítani egy adott GPU-n
    - Mindegyik profile egy bizonyos GPU architektúrához és grafikus API-hoz tartozik
    - Nem csak helyesnek kell lenni egy programnak
  - Nem képes általános célú programot végrehajtani
  - Korlátozni kell a profile-nak megfelelően

- GPU-k fejlődnek
- Új profile-okat támogat majd a Cg
  - Az új képességekkel rendelkező GPU-okhoz kapcsolódnak
- Idővel a profile-ok nem lesznek olyan fontosak
- A mai Cg programok a jövőbeli profile-okkal gond nélküli fordíthatóak lesznek
  - Superset
- Minél kisebb és hatékonyabb a Cg program, annál gyorsabban fog futni
- A profile-ok nem a Cg korlátozása, hanem a GPU-ké

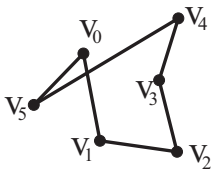
- A csővezeték egy állapotok szekvenciája
  - Párhuzamosan
  - Adott sorrendben
- Mindegyik állapot az előzőből kapja a bemenetét
- A kimenetét pedig a következő állapotba küldi
- Vertexek, geometriai primitívek és fragmensek (lehetséges pixel) sokaságát dolgozza fel



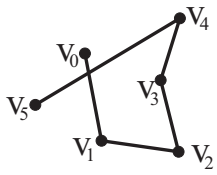
Pontok



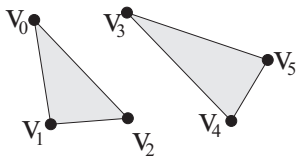
Vonalak



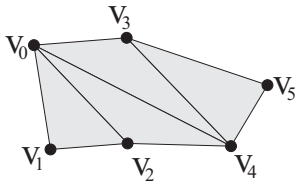
Vonal hurok



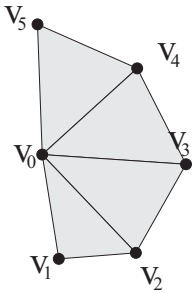
Töredezett vonal



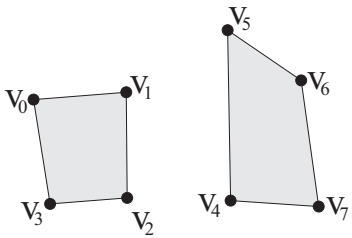
Háromszögek



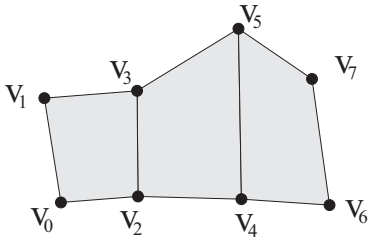
Háromszögsáv



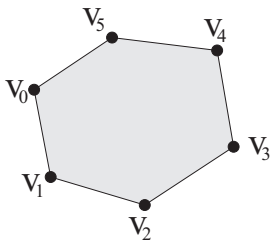
Háromszög-legyező



Négyszögek



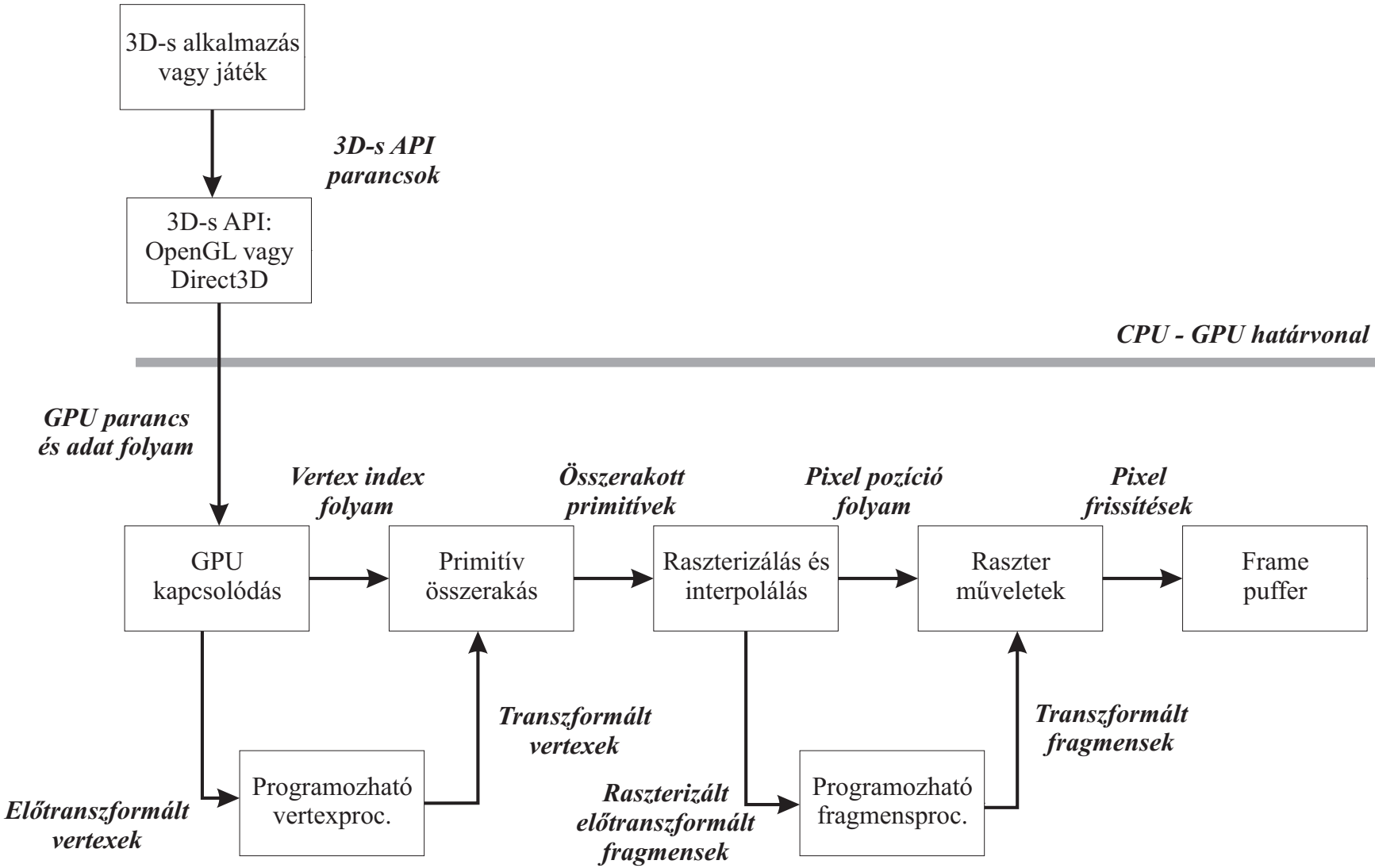
Négyszögsáv



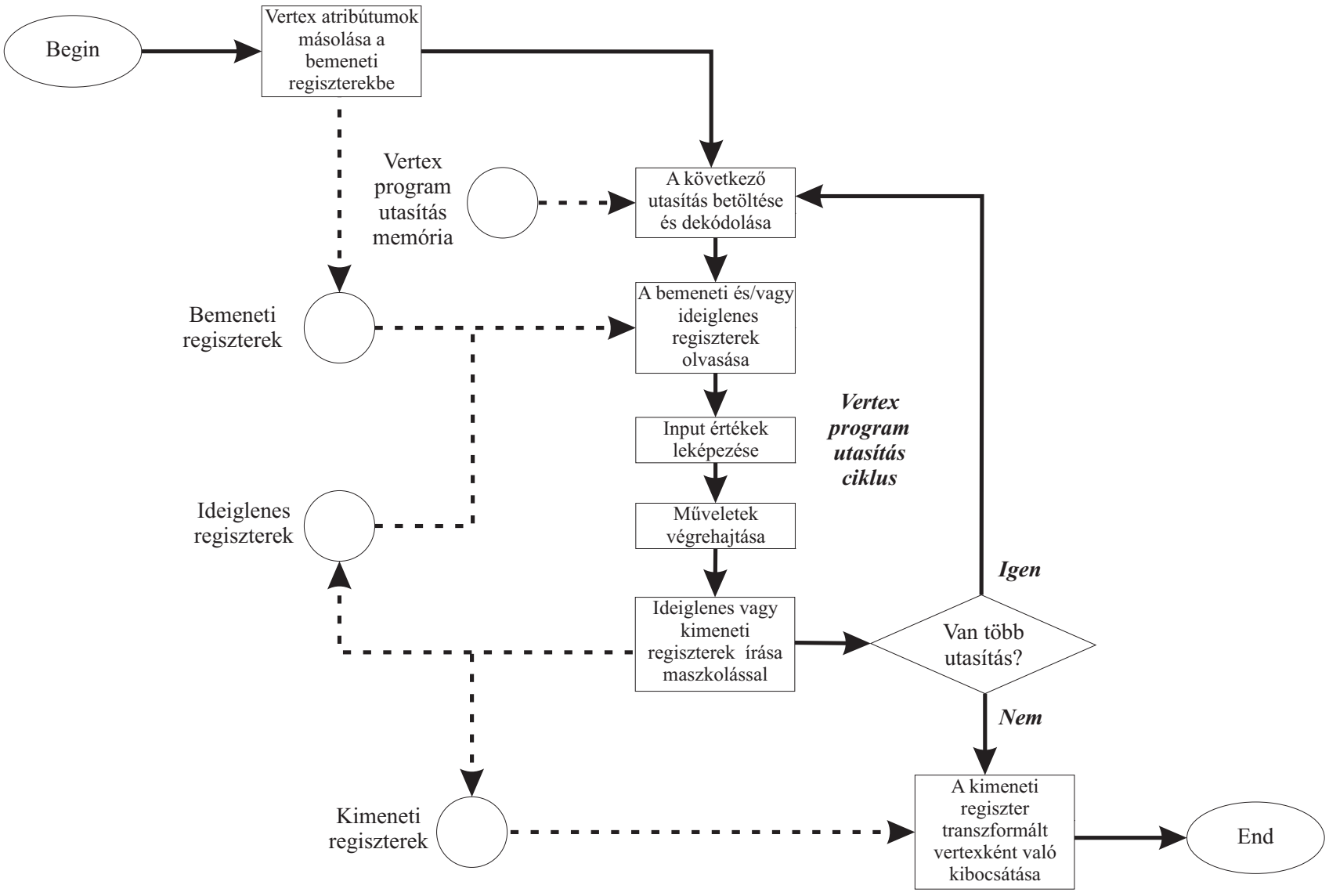
Poligon

- Mindegyik vertex rendelkezik
  - Pozícióval
  - Más attribútumokkal
    - Szín
    - Másodlagos (vagy spekuláris) szín
    - Egy vagy több textúra koordináta halmaz
    - Normál vektorok
    - ...

- Pixel
  - Picture element
  - Frame puffer tartalma egy adott helyen
    - Hasonlóan a szín, mélység és egyéb értékek, melyek ugyanazon a pozícióban találhatóak
- Fragmens
  - Az az állapot, mikor potenciálisan egy bizonyos pixel frissítése szükséges
  - A raszterizálás pixel méretű fragmensekre bontja a geometriai primitíveket
  - Ugyanúgy van pozíciója, mélység értéke, másodlagos színe és egy vagy több textúra koordináta halmaza
  - Potenciális pixel


3D-s alkalmazás vagy játék
3D-s API parancsok
3D-s API: OpenGL vagy Direct3D
CPU - GPU határvonal
GPU parancs és adat folyam
Vertex index folyam
Összerakott primitívek
Pixel pozíció folyam
Pixel frissítések
GPU kapcsolódás
Primitív összerakás
Raszterizálás és interpolálás
Raszter műveletek
Frame puffer
Transzformált vertexek
Előtranszformált vertexek
Programozható vertexproc.
Raszterizált előtranszformált fragmensek
Programozható fragmensproc.
Transzformált fragmensek

Programozható vertex processzor


Begin
Vertex atribútumok másolása a bemeneti regiszterekbe
Vertex program utasítás memória
A következő utasítás betöltése és dekódolása
Bemeneti regiszterek
A bemeneti és/vagy ideiglenes regiszterek olvasása
Input értékek leképezése
Vertex program utasítás ciklus
Ideiglenes regiszterek
Műveletek végrehajtása
Igen
Ideiglenes vagy kimeneti regiszterek írása maszkolással
Van több utasítás?
Nem
Kimeneti regiszterek
A kimeneti regiszter transzformált vertexként való kibocsátása
End

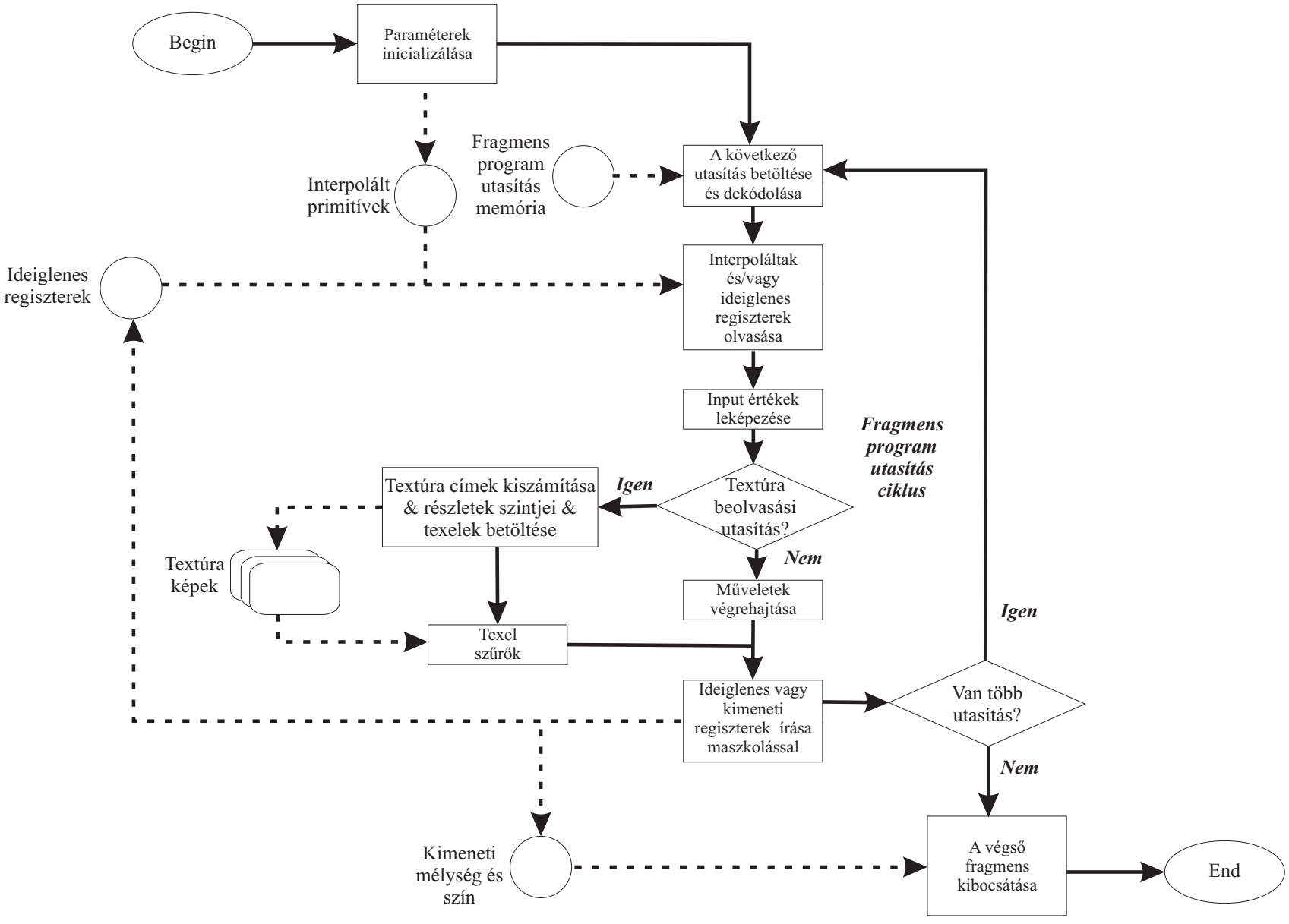
- Vertexek attribútumainak beolvasása a vertex processzorba
  - Pozíció, szín, textúra koordináták, stb.
- A vertex processzor újra meg újra behozza az következő utasítást és addig, amíg a vertex program véget nem ér
- Az utasítások számos különböző regiszter bankok halmazát éri el, melyek vektor értékeket tartalmaznak
  - Pozíció, normál vagy szín

- A vertex attribútum regiszterek csak olvashatóak
  - Az alkalmazás által meghatározott vertex attribútumok halmazát tartalmazza
- Az ideiglenes regiszterek írhatóak és olvashatóak is
  - Köztes értékek kiszámítására használhatóak
- A kimeneti eredmény regiszterek csak írhatóak
- Amikor a vertex program befejeződik, akkor a kimeneti regiszter tartalmazza a transzformált vertexet
- Az eredmény a raszterizálás és interpolálás után a fragmens processzorhoz kerül

- A legtöbb vertex feldolgozás korlátozott számú műveletet használ
- Vektor műveletek
  - 2, 3, 4 komponensű lebegőpontos vektorok
  - Összeadás, szorzás, szorzás-összeadás, skalár szorzat, minimum, maximum
  - Hardver támogatás
    - Vektor negálás, komponensenkénti „keverés” (swizzle)
    - Vektor műveletek általánosítása (Negálás, kivonás, kereszt-szorzat)

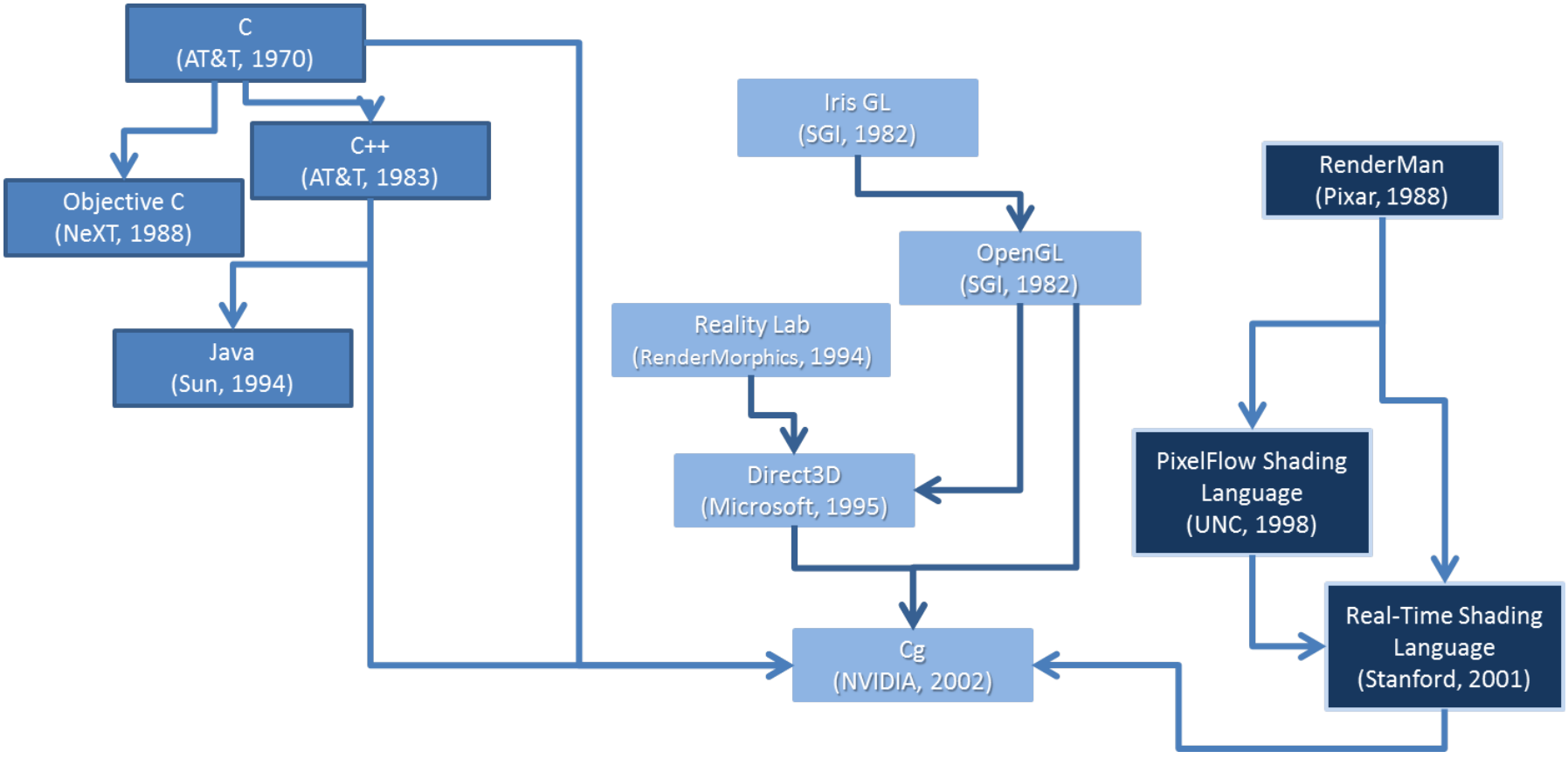
- Komponensenkénti írási maszkolás
  - Műveletek kimenetének szabályozás
- Reciprok és reciprok négyzetgyök kombinálása vektor szorzással és skalár szorzattal
  - Vektor normalizálás
  - Vektor skalárral való osztása
- Exponenciális, logaritmikus és trigonometrikus közelítések
  - Megvilágítás, köd és geometriai számítások elősegítése
- Specializált utasítások
  - Megvilágítás, elnyelődési fv-ek könnyebb kiszámítása

Programozható fragmens processzor


Begin
Paraméterek inicializálása
Interpolált primitívek
Fragmens program utasítás memória
A következő utasítás betöltése és dekódolása
Ideiglenes regiszterek
Interpoláltak és/vagy ideiglenes regiszterek olvasása
Input értékek leképezése
Fragmens program utasítás ciklus
Textúra címek kiszámítása & részletek szintjei & texelek betöltése
Igen
Textúra beolvasási utasítás?
Textúra képek
Nem
Műveletek végrehajtása
Igen
Texel szűrők
Ideiglenes vagy kimeneti regiszterek írása maszkolással
Van több utasítás?
Nem
Kimeneti mélység és szín
A végső fragmens kibocsátása
End

- Hasonló műveletek a programozható vertex processzorhoz
- Textúrázó műveletek
  - Textúra képhez való hozzáférés
    - Textúra koordináták
    - Visszatér a textúra egy szűrt mintájával
- Az új GPU-k támogatják lebegőpontos értékeket használatát
  - A fragmens műveletek hatékonyabbak, ha alacsonyabb pontosságú adat típusoknál

- Sok fragmens feldolgozása egyszerre
  - Nem lehetséges a tetszőleges szétosztás (branching)
- Cg-vel lehetséges ilyen fragmens programokat írni
  - Szimulál
  - Szétoszt és iterál
  - Feltételes „beosztás” operátorok
  - Ciklus „letekerés” (unrolling)
- Bemenő regiszterek
  - Interpolált fragmensenkénti paraméterek
    - Fragmens primitívek vertexenkénti paramétereiből származtatva
- Írható/olvasható ideiglenes regiszterek
- Csak írható kimeneti regiszterek
  - Szín
  - Opcionálisan új mélység érték


C
(AT&T, 1970)
C++
(AT&T, 1983)
Objective C
(NeXT, 1988)
Java
(Sun, 1994)
Iris GL
(SGI, 1982)
OpenGL
(SGI, 1982)
Reality Lab
(RenderMorphics, 1994)
Direct3D
(Microsoft, 1995)
RenderMan
(Pixar, 1988)
PixelFlow Shading Language
(UNC, 1998)
Real-Time Shading Language
(Stanford, 2001)
Cg
(NVIDIA, 2002)

- Általános célú programozási nyelvek
  - GPU-kra specializálódva
- Nem valós idejű árnyaló nyelvek
  - Valós időre optimalizálva
- Programozhat GPU-k és 3D API-k
  - Magas szintű nyelvi támogatás
- Örökség
  - Általános célú C programozási nyelv
  - Offline árnyékoló nyelvek
    - Pl. Renderman árnyaló nyelv
  - Grafikus funkcionalitás
    - OpenGL
    - Direct3D

Cg környezet

- A Cg csak egy komponense a szoftver és hardver infrastruktúrának
  - Összetett 3D-s látvány előállítása programozható GPU-okon valós időben

- Régebben egy PC-n a CPU kezelte le az összes vertex transzformációkat és „pixel-pushing” feladatokat
- A grafikus hardver csak a pixel puffert biztosította
  - A hardver a képernyőn jelenítette meg
- Saját 3D-s megjelenítő algoritmusok
- 3D-s alkalmazások
  - OpenGL
  - Direct3D

OpenGL

- SGI 1991
- OpenGL architecture Review Board (ARB)
- Eredetileg csak erős UNIX munkaállomásokon fut
- Microsoft alapító ARB tag
  - Windows NT
- Multi platformos programozási felület

Direct3D

- Microsoft 1995
  - Direct3D - DirectX
- Windows-s PC-k
- Xbox konzolok
- DirectX 10
  - Windows Vista
  - Aero felület

- Windows-os PC-ken
  - OpenGl vs. Direct3D
    - Melyik a jobb?
    - Melyik lesz az egyeduralkodó?
- A verseny folytatódik
  - Mind a két programozási felület előnyére válik
    - Javul a teljesítményük
    - Javul a minőségük
    - Javuk a funkcionalitásuk

- GPU programozhatóság
- Cg szempontjából összehasonlítható képességekkel bírnak
- Ugyanazon a GPU-n fut mind a kettő
  - Meghatározza a képességeket
- Csekély előny OpenGl esetén
  - A hardware gyártók jobban tudják megmutatni a teljes jellegzetességük halmazát OpenGl-en keresztül
  - A gyártó specifikus kiterjesztések egy kicsit bonyolultabbak a fejlesztők számára
- Mind a két programozási felület támogatja a Cg-t

A Cg fordító és Runtime

- Egyetlen egy GPU sem képes a Cg programot szöveges formában futtatni
  - A fordítás során a Cg programot olyan formátumba kell fordítani, melyet a GPU végre tud hajtani

- Először a Cg program olyan formába kerül, melyet a 3D-s programozási felület elfogad
  - OpenGl
  - Direct3D
- Az alkalmazás továbbítja a Cg program fordítását a GPU-nak
  - Megfelelő OpenGl vagy Direct3D utasításokkal
- Az OpenGl vagy Direct3D meghajtó hajtja végre az utolsó fordítást
  - Hardveren végrehajtható forma

- A fordítás részletei a GPU kombinált képességei és a 3D programozási felülettől függnek
- A köztes OpenGl vagy Direct3D formátum a GPU generációjától függ
  - Előfordulhat, hogy a GPU nem támogat egy érvényes/helyes Cg programot a GPU korlátai miatt

Hagyományos

- A fordítás offline eljárás
  - A fordító a programot a CPU futtható formátumúvá alakítja
  - A fordítás után nincs szükség újra fordításra
    - Programkód megváltozik
    - Másik platformon akarjuk futtatni

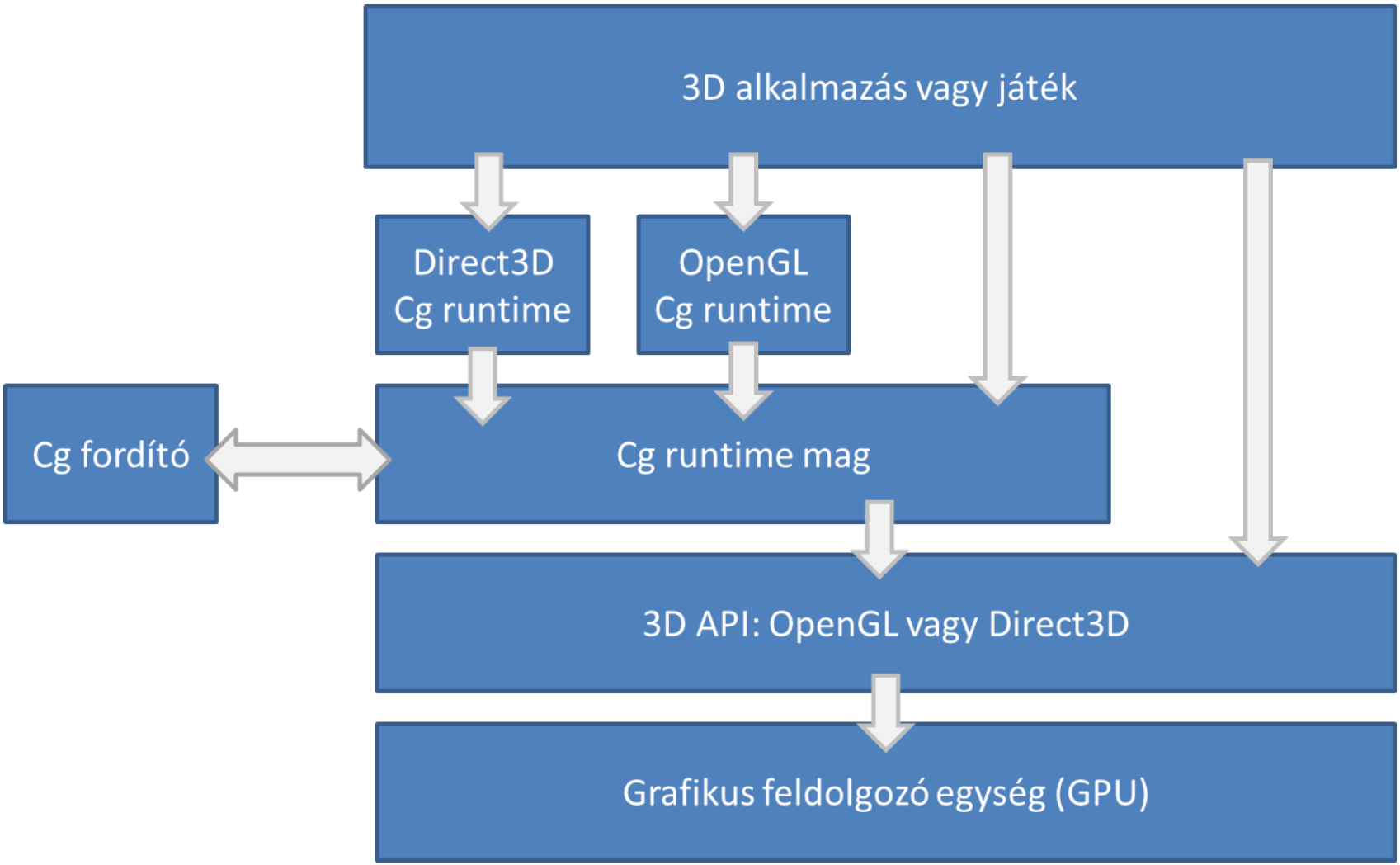
Cg

- Dinamikus fordítás
  - Támogatja a statikust is
- A fordító nem egy különálló program
  - Cg runtime könyvtár része
- Az alkalmazásokat össze kell szerkeszteni a Cg futásidejű könyvtárral
- Az alkalmazás használja a Cg-t, majd meghívja a Cg futásidejű rutinokat cg prefix-szel
- Optimalizált Cg program bizonyos GPU-ra

- OpenGl $\rightarrow$ CgGl $\rightarrow$ `CcgGL`
- Direct3D $\rightarrow$ CgD3D $\rightarrow$ `CcgD3D`
- Egyszerre a kettőt nem lehet használni!
- Összehasonlítva a Cg futásidejű könyvtár magjával, amely tartalmazza a Cg fordítót
  - CgGl és CgD3D könyvtárak viszonylag kicsik
  - Csak a megfelelő OpenGl illetve Direct3D hívásokat tartalmazza, melyek Cg programok végrehajtásának a beállítására szolgálnak
  - Hasonló rutinok

Cg alapismeretek

Hogyan illeszkedik a Cg futásidejű könyvtár az alkalmazásokhoz?



```
struct C2E1v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C2E1v_Output C2E1v_green(float2 position : POSITION)
{
  C2E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color    = float4(0, 1, 0, 1);  // RGBA green

  return OUT;
}
```

```
struct C2E1v_Output {
    float4 position :
        POSITION;
    float4 color    :
        COLOR;
};
```

- Kimeneti értékek
- Korlátozott kimentek
- Hasonló a C/C++-hoz
- Szemantika
  - POSITION
  - COLOR

- Betűvel kezdődő
  - Betűvel vagy számmal folytatódik
  - Tartalmazhat _ (aláhúzás) karaktert
  - Nem lehet kulcsszó
- Más azonosítók
  - Függvény név
  - Függvény paraméter név
  - Helyi változó
  - stb.

- C, C++-ban nincs natív vektor típus
  - Skalár értékek tömbje
- A vertex és fragmens feldolgozásban alapvető adat típus a vektorok
  - GPU beépített vektor támogatás
  - Cg-ben vektor adat típus
- float4
  - Nem foglalt szó
  - Alap típus definíció Cg Standard Könyvtárban

- Előredefiniált vektor adattípus
  - `float2`, `float3`, `float4`
  - Biztosítják a hatékony vektor feldolgozást a GPU-okon
- `float x[4]` $\neq$ `float4 x`
- Pakolt tömb
  - `data[3]` hatékony
  - `data[i]` nem hatékony

- Natív támogatás
  - `float4x4`
  - `half3x2`
  - `fixed2x4`
- Hasonlóan inicializálhatók, mint C-ben
- Szintén hatékonyak

- Ragasztó, mely hozzáköt egy Cg programot a hátralévő grafikus csővezetékhez
- Megmutatja azt a hardver erőforrást, amelyik feltölti a kimeneti struktúrát a Cg program visszatérésekor
- `POSITION`
  - Transzformált vertex vágási területen lévő pozíciója
- `COLOR`
  - Diffúz vertex szín
- Jelzi, hogy a megelőző változók hogyan kacsolódnak a grafikus csővezeték maradék részéhez
- Nem mindegyik szemantika érhető el az összes profile-ban
- Lehet saját szemantika neveket is létrehozni

- Függvények deklarációja hasonlóan történik, mint C-ben
  - Visszatérési érték
  - Név
  - Vesszővel elválasztott paraméter lista zárójelek között
  - Függvénytörzs
- Belépő vagy belső függvények

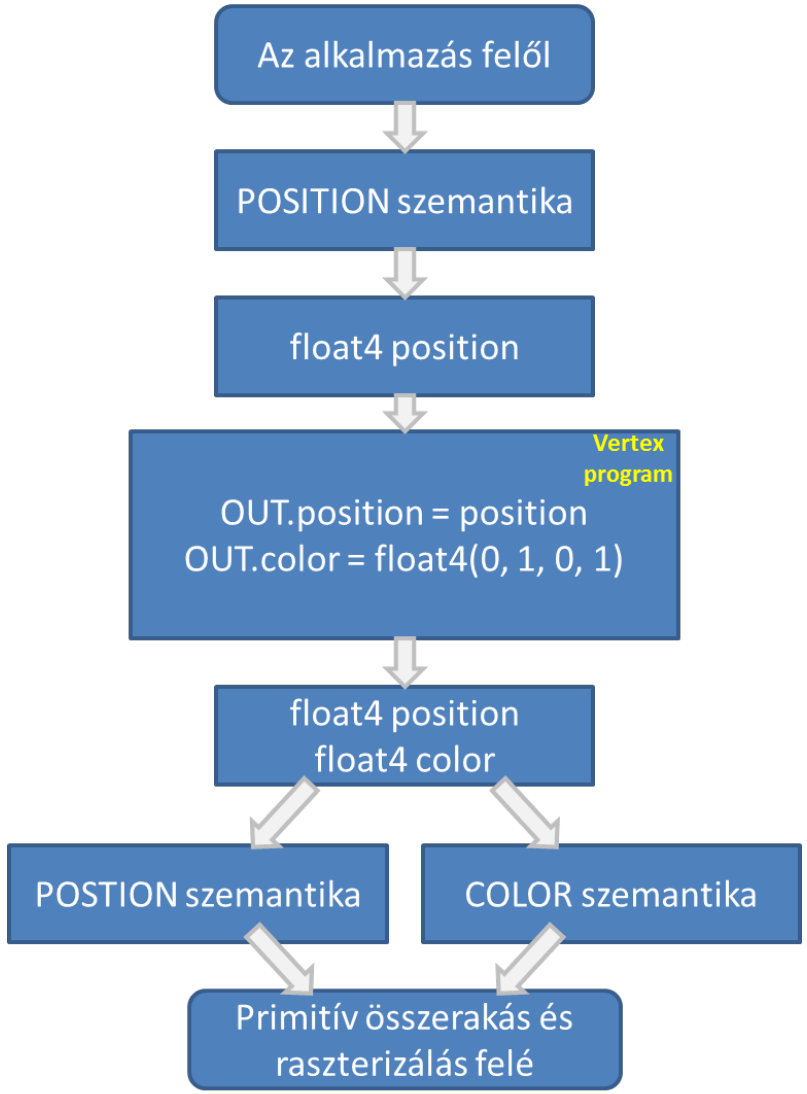
Belépő függvények

- Vertex vagy fragmens program definiálása
  - Analóg a `main` függvényre
  - A program végrehajtása ebben a belépő függvényben kezdődik
  - Amikor egy bemenő paraméter nevét kettőspont és szemantika név követ, akkor ez azt jelzi, hogy szemantika az bemenő paraméterhez van kötve
    - A vertex processzor inicializálja ezt a paramétert az alkalmazás által meghatározott összes vertex pozíciójával

Belső függvények

- A belépő függvények és belső függvények által meghívható függvények
  - Cg Standard Könyvtár
  - Saját belső függvény

- Ugyanaz a nevük
  - Mégis különböznek
    - Alkalmazás által meghatározott vertex pozíciók
    - Vágási területen lévő vertex pozíció
    - Hardver raszterizáló
- A grafikus csővezeték különböző helyén lévő pozíciók
  - Az első programban ez változás nélkül van tovább küldve
  - Nem változik



- A lényeg található a függvény törzsében
- Deklarálni kell a visszatérési értéket tartalmazó változót
- Kimeneti struktúra

```
{
  C2E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color    = float4(0, 1, 0, 1);        // RGBA zöld

  return OUT;
}
```

- `OUT.position = float4(position, 0, 1);`
  - A két komponensű vektor a kimeneti pozíció vektornak megfelelő struktúrává konvertálódik
- `OUT.color = float4(0, 1, 1, 1);`
  - Constructor vektorok és mátrix számára
- `return OUT;`

- Cg futásidejű könyvtár
  - Betöltés és fordítás
  - Meg kell adni
    - A belépő függvény nevét
    - A profile nevét a belépő függvény lefordításához

Profile neve	Programozási felület	Leírás
`arbvp1`	OpenGL	Alap multivendor-os vertex programozhatóság
`vs_1_1`	DirectX 8	Alap multivendor-os vertex programozhatóság
`vp20`	OpenGL	Alap NVIDIA vertex programozhatóság
`vs_2_0` `vs_2_x`	DirectX 9	Fejlett multivendor-os vertex programozhatóság
`vp30`	OpenGL	Fejlett NVIDIA vertex programozhatóság

Hagyományos hibák

- Szintaktikus
- Szemantikus
- Hagyományos fordítói hibák

Profile függő hibák

- Szintaktikailag és szemantikailag hibátlan
- Nem támogatja a megadott profile

Adottság

- Fragmens program
  - Textúra elérés
- Vertex program nem ér el ilyen adatot
- Nem megengedett olyan program lefordítása, amelyet nem lehet végrehajtani

Környezet

- Hibás olyan vertex programot írni, amely nem tér vissza pontosan egy `POSITION` szemantikával rendelkező értékkel
  - A vertex pozícióval rendelkeznek a hátralévő grafikus csővezeték állapotában
  - Fordítva: a fragmens program nem térhet vissza `POSITION` szemantikával rendelkező értékkel

Kapacitás

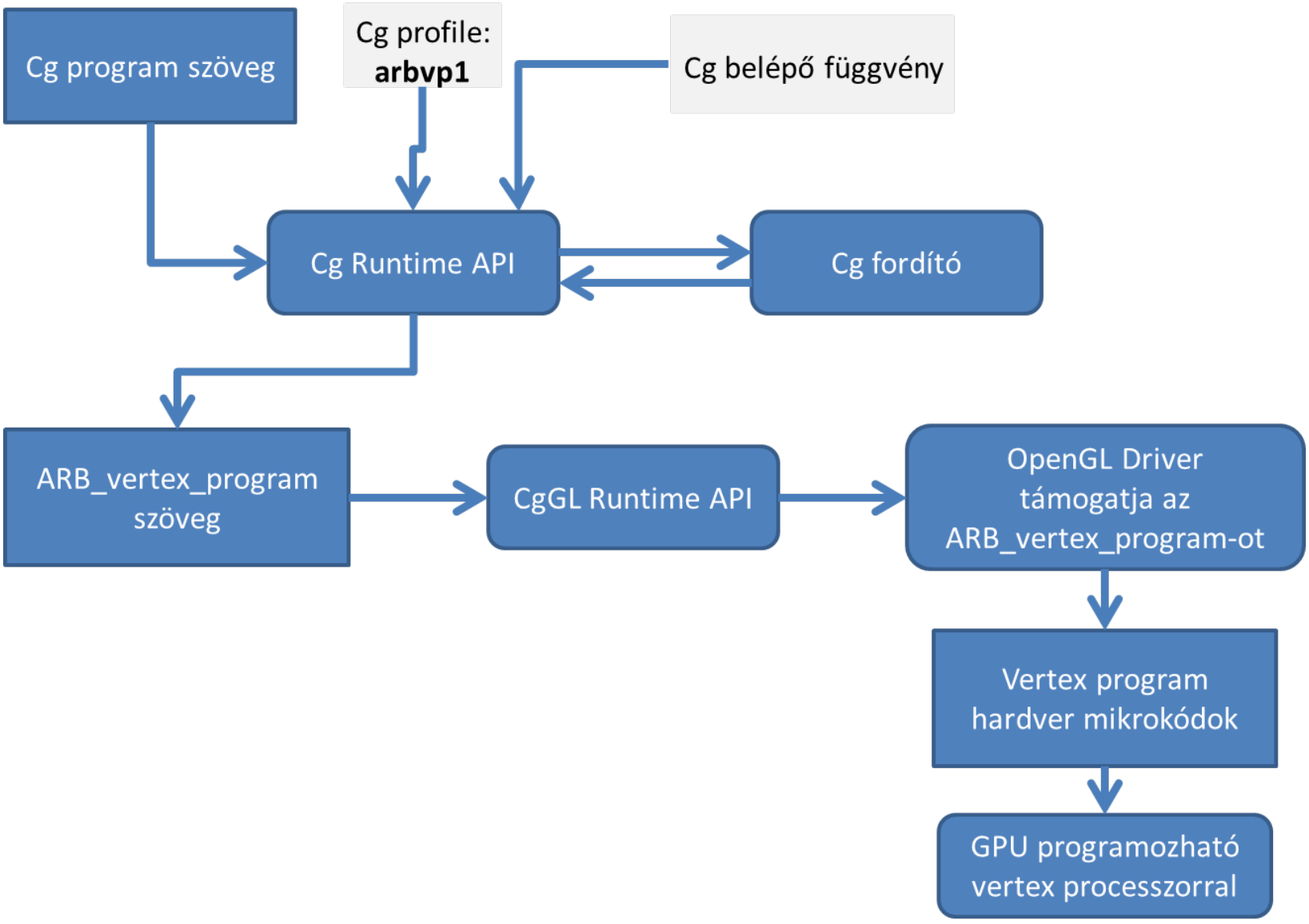
- A GPU képességeinek a korlátaiból származik
  - 4 textúra elérése egy renderelési menetben

Nem nyilvánvaló az, hogy mi haladja meg a GPU képességét

Hibák megelőzése

- Megfelelően nagy képességű profile választása
- Ismerni kell a határokat

- Cg programok gyűjteménye
  - Egy vertex és egy fragmens program
  - Cg programok fordítása futási időben
  - Új Cg programok generálása az alkalmazás futása közben


Cg program szöveg
Cg profile:
arbvp1
Cg belépő függvény
Cg Runtime API
Cg fordító
ARB_vertex_program
szöveg
CgGL Runtime API
OpenGL Driver
támogatja az
ARB_vertex_program-ot
Vertex program
hardver mikrokódok
GPU programozható
vertex processzorral

```
struct C2E2f_Output {
  float4 color : COLOR;
};

C2E2f_Output C2E2f_passthru(float4 color : COLOR)
{
  C2E2f_Output OUT;
  OUT.color = color;
  return OUT;
}
```

- Ez a program nem csinál semmit
- Az output ugyanaz az értek, amelyet a raszterizáló állított elő
- A GPU raszter műveletekért felelős hardvere ezt a színt használja a frame puffer frissítésére, amennyiben a fragmens túlélte a különböző raszter műveleteket

```
struct C2E2f_Output {
  float4 color : COLOR;
};
```

- A fragmens program csak szín értéket frissít a frame pufferben
  - Néhány fejlettebb profile-ban további adat is módosítható pl. a mélység érték
- A `COLOR` szemantika azt jelenti, hogy a color tag egy szín, amely értékét használja majd a frame puffer frissítésénél

- Belépő függvény deklaráció
  - `C2E2f_Output C2E2f_passthru(float4 color : COLOR)`
- Visszatérő struktúra
  - `C2E2f_Output`
  - Négy komponensű vektor
- A függvény törzse

```
{
  C2E2f_Output OUT;
  OUT.color = color;
  return OUT;
}
```

Profile neve	Programozási felület	Leírás

`ps_1_1`
`ps_1_2`
`ps_1_3`
	DirectX 8	Alap multivendor-os fragmens programozhatóság

`fp20`	OpenGL	Alap NVIDIA fragmens programozhatóság `NV_texture_shader`-nek és `NV_register_combiners`-nek megfelelően
`arbfp1`	OpenGL	Fejlett multivendor-os fragmens programozhatóság
`ps_2_0` `ps_2_x`	DirectX 9	Fejlett multivendor-os fragmens programozhatóság
`fp30`	OpenGL	Fejlett NVIDIA fragmens programozhatóság `NV_fragment_program`-nak megfelelően

- Olyan egyszerű, hogy bármelyikkel fragmens profile-lal lefordítható lenne
- Cg fordító használata
  - cgc
  - Tesztelés
  - IDE
    - MS Visual C++

OpenGL

```
glBegin(GL_TRIANGLES);
    glVertex2f(-0.8,  0.8);
    glVertex2f( 0.8,  0.8);
    glVertex2f( 0.0, -0.8);
glEnd();
```

Direct3D

```
static const MY_V3F triangleVertices[] = {
    { -0.8f,  0.8f, 0.0f },
    {  0.8f,  0.8f, 0.0f },
    {  0.0f, -0.8f, 0.0f }
pDev->DrawPrimitive (D3DPT_TRIANGLELIST, 0, 1);
```

```
myCgContext = cgCreateContext();
myCgVertexProgram =
cgCreateProgramFromFile(
   myCgContext,          /*Cg runtime környezet*/
   CG_SOURCE,            /*A program olvasható formában van*/
   myVertexProgFName,    /*A programot tartalmazó file neve*/
   myCgVertexProfile,    /*Profile: OpenGL ARB vertex program*/
   myVertexProgName,     /*Belépő függvény neve*/
   NULL);                /*Nincs extra fordító opció*/

cgGLLoadProgram(myCgVertexProgram);

cgGLEnableProfile(myCgVertexProfile);
cgGLBindProgram(myCgVertexProgram);

cgGLDisableProfile(myCgVertexProfile);

cgDestroyProgram(myCgVertexProgram);
cgDestroyContext(myCgContext);
```

```
static void checkForCgError(const char *situation)
{
  CGerror error;
  const char *string = cgGetLastErrorString(&error);

  if (error != CG_NO_ERROR) {
    printf("%s: %s: %s\n",
    myProgramName, situation, string);
    if (error == CG_COMPILER_ERROR) {
      printf("%s\n", cgGetLastListing(myCgContext));
    }
    exit(1);
  }
}
```

- Uniform paraméterek
  - Az előző példák kiterjesztése
  - `OUT.color = float4(1.0, 0.41, 0.7, 1.0);`
  - Nem lehet minden színre külön Cg programot írni
  - A program általánosítása paraméter átadásával

```
struct C3E1v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C3E1v_Output C3E1v_anyColor(float2 position : POSITION,
                            uniform float4 constantColor)
{
  C3E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color = constantColor;  // Some RGBA color

  return OUT;
}
```

- Jelzi a változók kezdő értékének a forrását
  - Amikor `uniform` változóként van deklarálva egy változó, akkor külső környezetből kapja az iniciális értékét
- Olyan vertex program generálódik, amely a GPU konstans regiszteréből kapja az iniciális értékét
- Cg runtime használatakor
  - A 3D-s alkalmazás le tudja kérni egy paraméter kezelőt a Cg programon belül

Kezelő lekérdezése

```
Cgparemeter myParameter = cgGetNamedParameter(program,
    ``myParameter'');
```

Paraméter értékének beállítása

```
cgGLSetParameter4fv(myParameter, value);
```

Paraméter definiálása

```
float4 myParameter
```

Explicit iniciális érték megadás

```
float4 green = float4(0, 1, 0, 1);
```

Szemantika használata

```
float4 position : POSITION;
```

Profile függő

```
float whatever; //nem definiált vagy 0
```

- Hasonló hatása van, mint C vagy C++-ban
  - Korlátozza a változó használatát a programban
  - A bizonyos érték nem változhat meg soha
  - Hiba üzenet generálódik ellenkező esetben

- A bemeneti adatokon elvégzett számítások
  - Operátorok
  - Beépített függvények a Cg standard könyvtárban
- A Cg támogatja ugyanazokat az aritmetikai, relációs és más operátorokat, amelyeket a C és C++-ban használhatunk
- A Cg mégis különbözik a C és C++-tól
  - Beépített támogatás a vektor mennyiségeken végzendő aritmetikai műveletek esetén
  - C++-ban operátor overloading (túlterhelés) megoldható
  - Amikor skalár az egyik operandus, akkor az adott skalárt egy vektorrá konvertálja

- Folytonos adattípusok ábrázolása
  - `float`, `half`, `double`
  - Csak a Cg-ben `half`: 16 bites fél pontosságú lebegőpontos érték
- A GPU általában nem rendelkezik hardveres támogatással annyi alap adat típusra, mint a CPU
  - Pl. nem támogatja a pointer adat típust
  - Nem támogatják a természetüktől fogva diszkrét mennyiségeket sem
    - Alfa-numerikus karakterek
    - Bit maszkok

- A folytonos mennyiségek nincsenek korlátozva az egész értékekre
  - Fragmens szinten egy szűk intervallumra vannak korlátozva
    - $[0, 1]$ vagy $[-1, +1]$ (szín, normál vektorok)
    - Ezen intervallum korlátozott adattípusok fix-pontos adattípus néven is ismertek
- A `float` nem mindig lebegő-pontos értéket jelent az összes profile-ban, az összes összefüggésben

- Trigonometrikus
- Exponenciális
- Vektor mátrixok
- Textúrák
- Viszont nincs
  - I/O
  - String műveletek
  - Memória foglalás

- A Cg standard könyvtár túlterheli a függvényeket
  - A rutinok sok adattípuson értelmezettek
  - Többszörös megvalósítás
  - Ugyanazon néven több fajta paraméter listával
    - Mindig a megfelelő verziójú függvény hívódik meg
  - Saját belső túlterhelt függvények
- Különböző megvalósítása ugyanannak a rutinnak más profile-ok számára
  - Pl. egy fejlett vertex profile-ban van sin és *cos*
    - De egy alap vertex profile-ban valamilyen módon közelíteni kell az adott értékeket
    - Profile függő függvény túlterhelés
    - Két függvény, mely speciális profile-t követel meg

- A Cg Standard könyvtár matematikai és más műveleteket hatékonyabbak és pontosabbak
  - Speciális GPU utasítások

Saját függvény

```
float myDot(float3 a, float3 b)
{
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
}
```

Beépített függvény (Gyorsabb)

```
dot(a, b)
```

- `out` minősítő jelzi, amivel a rutinnak vissza kell térnie
  - Kezdetben az értéke nem definiált
  - call-by result (copy out)
- `in` minősítő érték szerinti paraméter átadás
  - Az értéket eldobja a Cg
  - Ha `out` minősített is akkor nem
- `inout`
- Az `in` minősítő alapértelmezett
- Nincs funkcionális különbség a két módszer között
- Kombinálhatóak

- Vektor komponensek átrendezése
- Suffix-ek
  - `x, y, z, w`
  - `r, g, b, a`
- Diszjunkt halmazok
- Fordított swizzling
  - `vec1.xw = vec2`

Példák

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

- ._m<row><col>

Példák

```
float4x4 myMatrix;
float myFloatScalar;
float4 myFloatVec4;

myFloatScalar = myMatrix._m32;
myFloatVec4   = myMatrix._m00_m11_m22_m33;
```

- Struktúrák
  - Előző példák
- Tömbök
  - Nincs pointer típus
  - Tömb szintaxist kell használni
- Folyam szabályozás
  - Függvények és a `return` utasítás
  - `if-else`
  - `for`
  - `while` és `do-while`
- Profile specifikusak
  - Pl. a ciklusoknál az iterációk számát előre meg kell tudni határoznia
- `goto` és `switch`
  - Foglaltak
  - Nem támogatják egyelőre

Összefoglalás

- Programozható vertex és fragmens processzor
  - A csővezeték standard folyamatának a megváltoztatása
  - Profile függő vertex és fragmens programok
  - Szemantikák
  - Speciális műveletek vektorokon
  - Dinamikus fordítás program futása alatt
- Példák

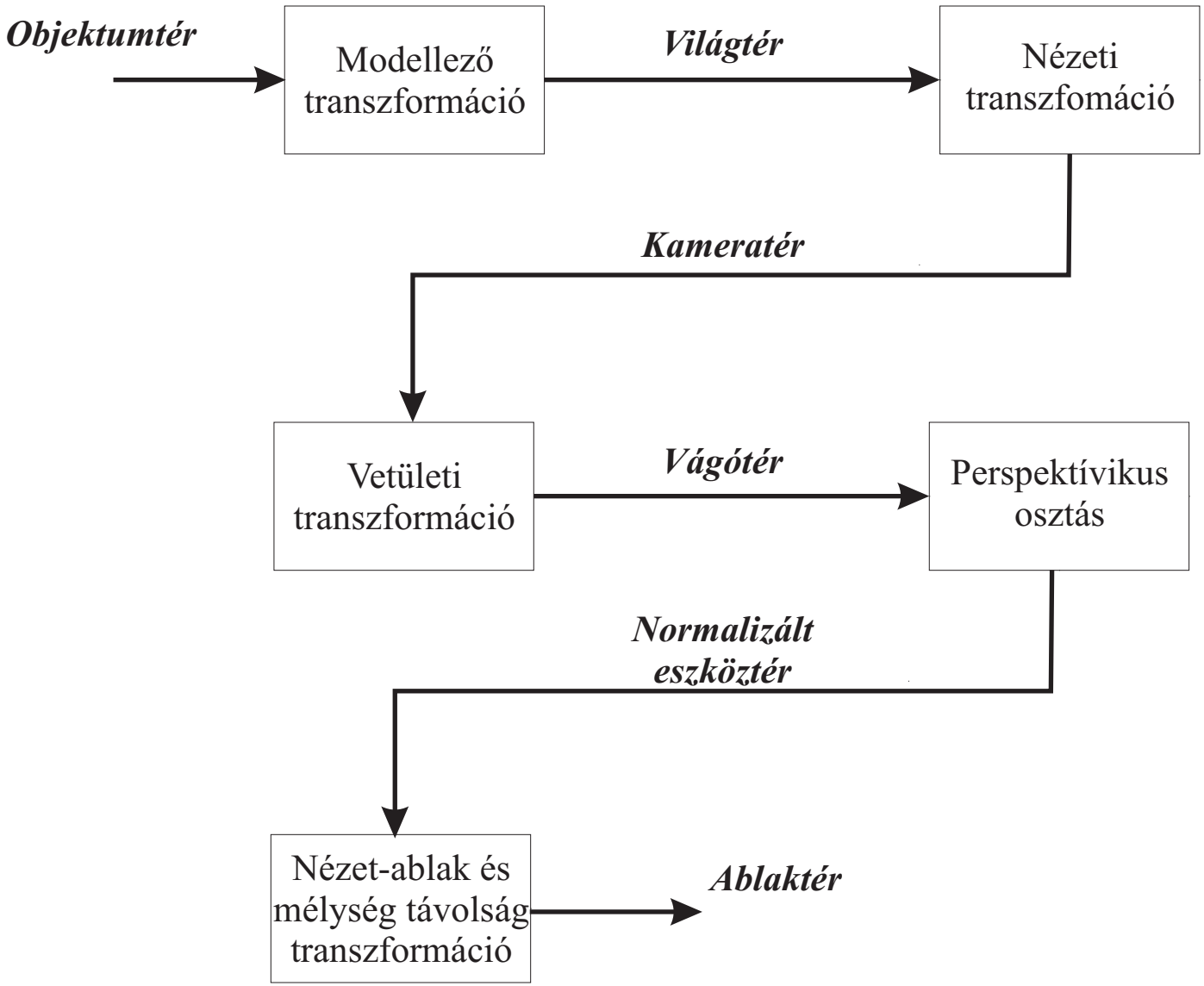
Geometriai transzformációk

Geometriai transzformációk

- Objektumok transzformálása és animálása a 3D-s színtéren
- Mátrix és oszlop vektor szorzása
  - $\mathbf{x}' = \mathbf{A}\mathbf{x}$
    - $\mathbf{x}'$ transzformált oszlopvektor (vertex pozíció)
    - $\mathbf{A}$ transzformációs mátrix (transzformációt leíró mátrix)
    - $\mathbf{x}$ transzformálandó oszlopvektor (vertex pozíció)
  - $\mathbf{x}'' = \mathbf{B}\mathbf{x}' = \mathbf{B}\mathbf{A}\mathbf{x}$

Transzformációs csővezeték

Transzformációs csővezeték



- Vertex pozíció meghatározása egy koordináta rendszerben
- Minden objektum egy saját objektumtérrel rendelkezik
- Pozíció ábrázolása vektorként (pl. koordináta-hármasok)
- Transzformációk segítségével az egyik térben lévő pozíciót egy másik térben lévő pozícióra képezünk le
- Normálvektor
  - Adott felületre merőleges egység hosszú vektor

- Négy-komponensű (x, y, z, w) alak
  - Egy Descartes koordinátával megadott (x, y, z) helyvektor speciális esete
- $(x, y, z, w) = (x', y', z', w')$,
  - ha $\exists$ egy olyan $h \neq 0$,
  - hogy $x' = hx$, $y' = hy$, $z' = hz$, $w' = hw$
- $w = 0$ homogén pozíció esetén végtelenben lévő pont
- $(0, 0, 0, 0)$ homogén koordináta nem megengedett

- $w \neq 0$ esetén (x, y, z, w) szokásos jelölése
  - $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$
- (x, y, z) Descartes koordinátához egy 1-est negyedik komponensként hozzávéve adhatjuk meg a homogén alakot
  - $(x, y, z, 1)$
- Egy (x, y, z, w) homogén pozíció homogén osztás után
  - $(x', y', z', 1)$ pozícióként fog megjelenni
  - Az utolsó 1-es komponens elhagyásával kapjuk meg a homogén pozícióhoz tartozó Descartes koordinátát

- Objektumok között nincsenek térbeli viszonyok definiálva az objektumtéren
- Az objektumok egymáshoz való viszonyuk meghatározása/megadása
- Egy objektumtérben lévő modell homogén vertex pozíciójának ($w = 1$) modellező transzformációval való szorzása
  - A modell világtérbe transzformált pozíciója

- 3D-s modellek elhelyezése a világtérben
  - Forgatás
  - Eltolás
  - Skálázás
  - ...
- Transzformáció megadása
  - 4×4-es homogén transzformációs mátrixok
  - Több transzformáció kombinálása mátrix szorzással egyetlen egy 4×4-es mátrixba

- Egy bizonyos nézőpontból tekintünk a létrehozott színtérre
- A szem a koordináta-rendszer origójában van
- Azt a transzformációt, ami a világtéren lévő pozíciókat a kameratérre viszi át nézeti transzformációnak nevezzük

- 4×4-es mátrix
- Tipikus nézeti transzformáció a világtéren lévő kamera pozícióját a kameratér origójába viszi
  - Eltolás és elforgatás
- Meghatározza a kamera helyét és irányítottságát
- Modell-nézeti mátrix
  - Árnyalási számítások kamera- vagy objektumtérben
  - A modellező és nézeti transzformációs mátrixok összeszorzása

- $\mathbf{T}$ eltolás mátrix

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{T}(\mathbf{t})$ alkalmazásával egy $\mathbf{p}$ pontot a $\mathbf{p}'$ pontba tolhatunk

$$\mathbf{p}' = (p_x + t_x, p_y + t_y, p_z + t_z, 1).$$

- Inverz transzformációs mátrix
  - $\mathbf{T}(\mathbf{t})^{-1} = \mathbf{T}(-\mathbf{t})$

- $\mathbf{R}_x(\phi)$, $\mathbf{R}_y(\phi)$ és $\mathbf{R}_z(\phi)$ forgatási mátrixok
  - x, y és z tengelyek körüli forgatás ϕ szöggel

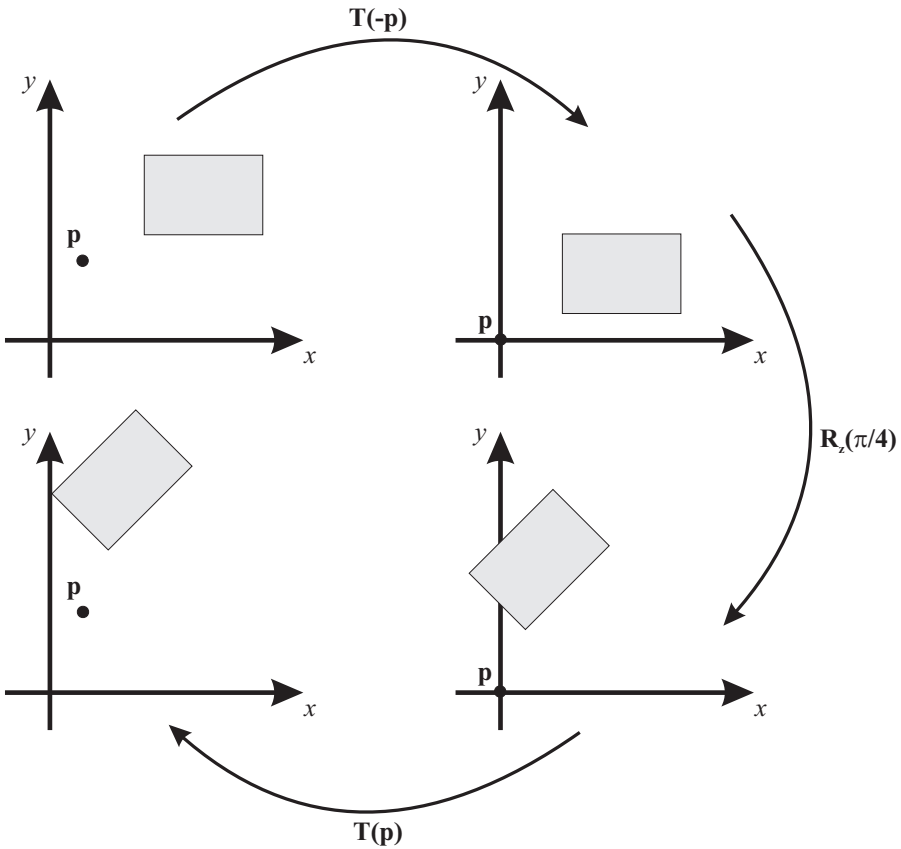
$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Bal-felső 3×3-as részmátrix diagonális elemeinek összege (mátrix nyoma) állandó
  - $\mathrm{tr}(\mathbf{R}) = 1 + 2\cos\phi$
- Forgatási mátrix ortogonális mátrix
  - Inverze megegyezik a mátrix transzponáltjával
  - $\mathbf{R}^{-1} = \mathbf{R}^T$
- Inverz forgatási mátrix
  - $\mathbf{R}_i^{-1}(\phi) = \mathbf{R}_i(-\phi)$
- Forgatási mátrix determinánsa mindig eggyel egyenlő

- A z tengellyel párhuzamos, p ponton átmenő tengely körüli forgatás
  - $\mathbf{X} = \mathbf{T}(\mathbf{p})\mathbf{R}_z(\phi)\mathbf{T}(-\mathbf{p})$



- $\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z)$ (ahol $s_x \neq 0, s_y \neq 0$ és $s_z \neq 0$) skálázó mátrix
  - az x, y és z irányokban az s_x, s_y és s_z értékekkel skáláz (kicsinyít/nagyít)

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Uniform skálázás
  - $s_x = s_y = s_z$
- Inverz skálázás
  - $\mathbf{S}^{-1}(\mathbf{s}) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

- Uniform skálázási mátrix létrehozása
  - Egy homogén koordináta-vektor w komponensének a manipulációjával

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/5 \end{pmatrix}$$

- Tükröző mátrix,
  - ha $\mathbf{s}$ három értékéből egy negatív

- Adott irányba történő skálázás
  - $\mathbf{f}^x$, $\mathbf{f}^y$ és $\mathbf{f}^z$ ortonormált vektorok mentén
- $\mathbf{F}$ mátrix létrehozása

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}^x & \mathbf{f}^y & \mathbf{f}^z & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

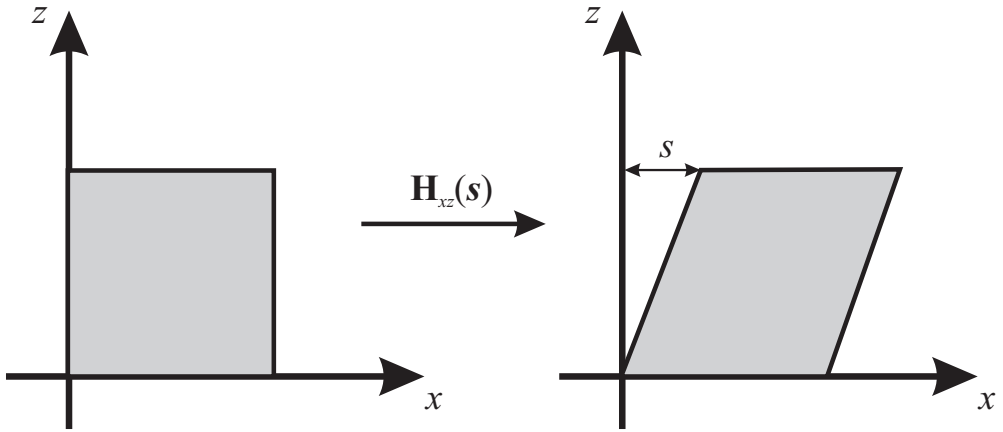
- $\mathbf{F}$ inverzével szorzunk
  - vagyis az $\mathbf{F}$ mátrix transzponáltjával
- Skálázás majd visszatranszformálás

$$\mathbf{X} = \mathbf{F}\mathbf{S}(\mathbf{s})\mathbf{F}^T$$

- Hat alap nyírást $\mathbf{H}_{xy}(s)$, $\mathbf{H}_{xz}(s)$, $\mathbf{H}_{yx}(s)$, $\mathbf{H}_{yz}(s)$, $\mathbf{H}_{zx}(s)$, $\mathbf{H}_{zy}(s)$ mátrixokkal jelöljük
  - Első index azt a koordinátát jelöli, amelyet a nyíró mátrix megváltoztat
  - Második index azt koordinátát jelöli, amely értékétől a változás mértéke függ
- $\mathbf{H}_{xz}(s)$ nyíráshoz tartozó mátrix

$$\mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{P} = (p_x, p_y, p_z)^T$ pontot balról megszorozva
  - $\mathbf{P}' = \mathbf{H}_{xz}(s)\mathbf{P} = (p_x + sp_z, p_y, p_z)^T$



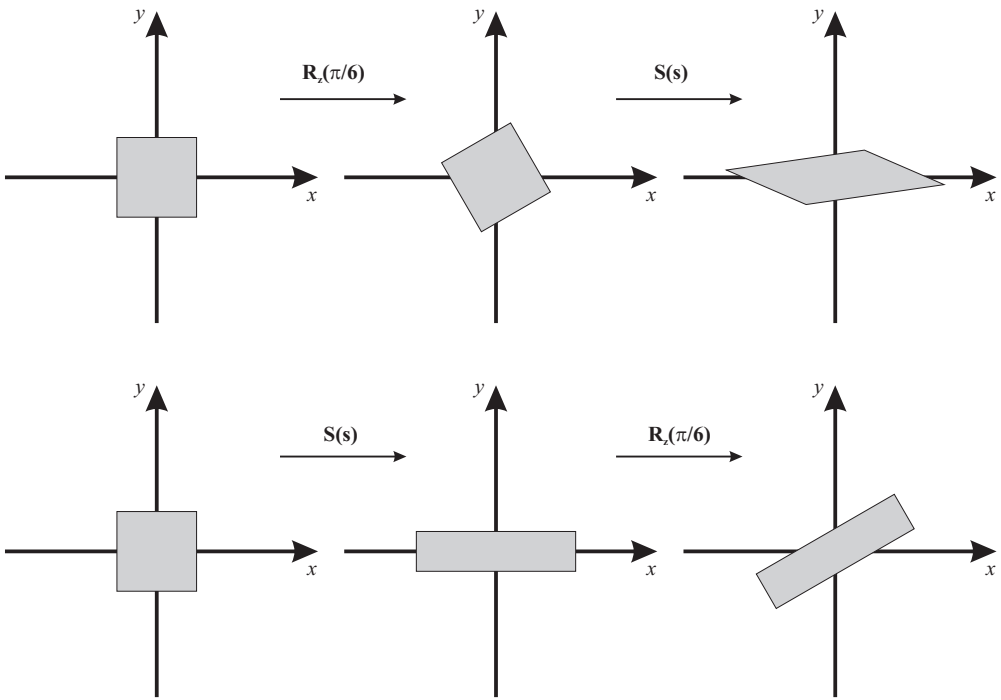
Az y és z értékek nem változnak a transzformáció során

- $\mathbf{H}_{ij}(s)$ (ahol $i \neq j$) mátrix inverze
  - Ellentétes irányba való nyírás
  - $\mathbf{H}_{ij}^{-1}(s) = \mathbf{H}_{ij}(-s)$
- Nyírás más formában való megadása

$$\mathbf{H}'_{xy}(s,t) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  - A két indexszel azt jelöljük, hogy két koordinátát nyírunk a harmadikkal
  - $\mathbf{H}'_{ij}(s,t) = \mathbf{H}_{ik}(s)\mathbf{H}_{jk}(t)$, ahol k a harmadik koordinátát jelöli
- A nyírás térfogat megörző transzformáció
  - $|\mathbf{H}| = 1$

- Mátrix szorzás nem kommutatív
- Transzformációk végrehajtásának a sorrendje hatással van az eredményre
- Tetszőleges **N** és **M** mátrixokra $\mathbf{NM} \neq \mathbf{MN}$
- $\mathbf{TRSp} = \mathbf{T}(\mathbf{R}(\mathbf{Sp}))$



- Csak eltolás és forgatás transzformációk összefűzésével áll elő
  - A hosszakat és szögeket megőrzi
  - Felírható $\mathbf{T}(\mathbf{t})$ eltolás- és $\mathbf{R}$ elforgatásmátrix szorzataként

$$\mathbf{X} = \mathbf{T}(\mathbf{t})\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Inverze kiszámítható

$$\mathbf{X}^{-1} = (\mathbf{T}(\mathbf{t})\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{T}(\mathbf{t})^{-1} = \mathbf{R}^T\mathbf{T}(-\mathbf{t})$$

- Az inverz másképpen is kiszámítható

$$\overline{\mathbf{R}} = (\mathbf{r}_{,0} \quad \mathbf{r}_{,1} \quad \mathbf{r}_{,2}) = \begin{pmatrix} \mathbf{r}_{0,}^T \\ \mathbf{r}_{1,}^T \\ \mathbf{r}_{2,}^T \end{pmatrix}$$

$$\Rightarrow$$

$$\mathbf{X} = \begin{pmatrix} \overline{\mathbf{R}} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

$$\mathbf{X}^{-1} = \begin{pmatrix} \mathbf{r}_{0,} & \mathbf{r}_{1,} & \mathbf{r}_{2,} & -\overline{\mathbf{R}}^T\mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Lineáris transzformációk esetén pontok különbségének a képe a képek különbsége
  - $\mathbf{A}(\mathbf{x_1} - \mathbf{x_2}) = \mathbf{A}\mathbf{x_1} - \mathbf{A}\mathbf{x_2}$
- Nem szögtartó transzformációkat tartalmazó mátrixok
  - skálázás, nyírás
  - Nem mindig használhatóak fel normálvektorok transzformációjára
- A normálvektorokat a geometriai transzformációs mátrix inverzének a transzponáltjával kell megszorozni
  - $\mathbf{M}$ geometriai transzformáció
  - $\mathbf{N} = (\mathbf{M}^{-1})^T$

- Gyakorlatban
  - Csak forgatások alkalmazása esetén
    - Mátrix inverze maga a transzponált mátrix
    - Nem kell kiszámolni az inverzet
  - Csak eltolások alkalmazása esetén
    - Nincs hatással a normálvektor irányára
  - Eltolás és forgatások után nem kell normalizálni
    - Megőrzik a hosszokat
  - Több uniform skálázás alkalmazása esetén
    - Az irány nem módosul
    - Csak a hosszát kell normalizálni
    - Ha ismerjük a skálázás mértékét, akkor osztással elvégezhető
- Ha mégis ki kell számolni az inverzet
  - Elegendő a mátrix bal felső 3×3-as mátrixának az adjungáltjának a transzponáltját meghatározni
  - Osztásra nincs szükség, mivel végül normalizálni kell

- Ha a mátrix egy transzformációt vagy egyszerű transzformációk sorozatát tartalmazza adott paraméterekkel
  - A mátrixot egyszerűen lehet invertálni a paraméterek invertálásával és a mátrixok sorrendjének a megfordításával
  - $\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(\phi)$, akkor $\mathbf{M}^{-1} = \mathbf{R}(-\phi)\mathbf{T}(-\mathbf{t})$
- Ha a transzformációs mátrix ortogonális
  - A mátrix transzponáltja az inverze $\mathbf{M} = \mathbf{M}^T$
- Ha semmilyen információnk nincs
  - ***Adjungált módszer***
  - ***Cramer szabály***
  - LU felbontás
  - Gauss elimináció

- A kameratérben lévő primitíveket a képsíkra kell leképezni
  - A homogén koordináták miatt w súllyal vannak módosítva
- Kanonikus térfogat
  - Normalizált eszköz koordináták
- Vágókockán kívül eső objektumok levágása
- Kameratér koordinátáit a vágótér koordinátáiba a vetületi transzformáció segítségével transzformáljuk át

- A nézeti térfogat pontos alakja a vetületi transzformáció típusától függ
  - A perspektív transzformáció egy csonka gúlát határoz meg
  - Az ortogonális vetítés egy téglatestet visz át vágókockába
- Adott térfogaton belül lévő alakzatok láthatóak

- Párhuzamos vonalak párhuzamosak maradnak
- Legegyszerűbb transzformációs mátrix

$$\mathbf{P}_O = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{P}_O$ nem invertálható, mivel a determinánsa $|\mathbf{P}_O| = 0$

- A néző számára probléma, hogy a pozitív és a negatív z értékeket levetíti a vetítési síkra
  - z irányba nem lehet vágást végrehajtani
  - A takarásban lévő felületek kezelése problémás
  - x, y és z értékek bizonyos intervallumra való lekorlátozása
    - Például a közeli síktól a távoli síkig

- Mátrix megadása egy hatossal (l, r, b, t, n, f)
  - bal, jobb, alsó, felső, közeli és távoli síkok megjelölése
- A mátrix skálázza és eltolja az ezekkel a síkokkal kialakított Tengelyhez Igazított Befoglaló Dobozt
  - Origó középpontú, tengelyhez-igazított vágókockába
  - Bal-alsó sarka (l, b, n)
  - Jobb-felső sarka (r, t, f)

- OpenGL-ben a z-tengely negatív része felé nézünk
  - $-z$ féltérben lévő modelleket szeretnénk megjeleníteni
- $n > f$, mivel a z-tengely negatív irányába nézünk
- Könnyebb dolgunk van akkor, ha a közeli értékek kisebbek, mint a távoliak
  - Felfoghatóak pozitív távolságoknak a nézeti irány mentén
  - Nem pedig z szem koordináta értékekként

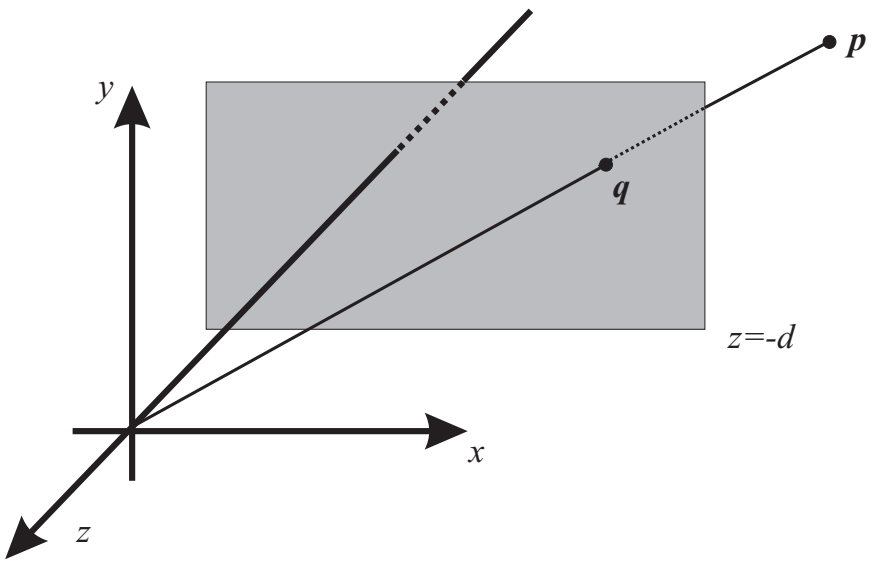
- Ortogonális transzformációs mátrix megadása:

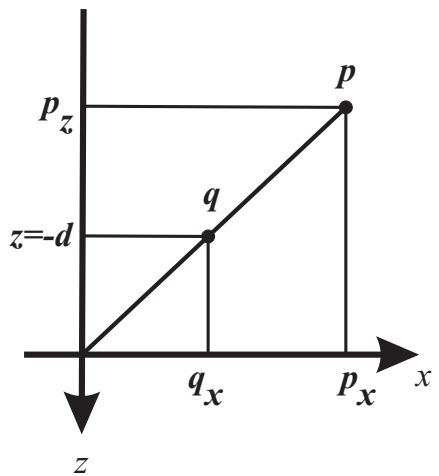
$$\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- A mátrix invertálható
  - $\mathbf{P}_o^{-1} = \mathbf{T}(-\mathbf{t})\mathbf{S}((r-l)/2, (t-b)/2, (f-n)/2))$

- Perspektív vetítési mátrix
  - $z = -d, d > 0$ síkra képez le
  - A nézőpont az origóban van
  - Egy $\mathbf{p}$ pontot vetítünk
  - Képe $\mathbf{q} = (q_x, q_y, -d)$
- Hasonló háromszögek
  - $\frac{q_x}{p_x} = \frac{-d}{p_z} \Longrightarrow q_x = -d\frac{p_x}{p_z}$
  - $q_y = -d\frac{p_y}{p_z}$
  - $q_z = -d$





$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

$$\mathbf{q} = \mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} =$$

$$= \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}$$

- Az utolsó lépésben a w komponenssel elosztjuk a vektort

- Van egy perspektív transzformáció, amely a nézeti csonka gúlát a kanonikus nézeti térfogatba transzformálja
- Csonka gúla nézet esetén feltesszük
  - $z = n$-ben kezdődik és a $z = f$-ben végződik $0 > n > f$ értékek esetén
- Téglalap $z = n$ esetén
  - Bal alsó sarok (l, b, n)
  - Jobb felső sarok (r, t, n)

- A kamera nézeti csonka gúlájának meghatározása
  - (l, r, b, t, n, f)
- A vízszintes látómezőt a csonka gúla bal és jobb síkjai között lévő szög határozza meg
- A függőleges látómezőt az alsó és felső síkok közötti szög
- Minél nagyobb a látószög, annál többet „lát” a kamera
- Aszimmetrikus csonka gúla - sztereó látás
  - $r \neq -l$ vagy $t \neq -b$ értékekkel hozhatunk létre

- Látómező - Színtér észlelése
- Számítógép képernyője
  - Szem fizikai látó mezeje
- $\phi = 2 \arctan(w/(2d))$
  - ϕ a látómező
  - w a színtér szélessége
  - d az objektumtól való távolság

- Szélesebb látómezőt beállítva az objektumok torzítva fognak megjelenni
- Perspektív transzformációs mátrix
  - A csonka gúlát az egység kockába transzformálja

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$ pontot transzformálva
  - w komponens értéke (a legtöbb esetben) nem nulla lesz és nem lesz egyenlő eggyel
- A vetített $\mathbf{p}$ ponthoz osztanunk kell q_w-vel
  - $\mathbf{p} = (q_x/q_w, q_y/q_w, q_z/q_w, 1)^T$

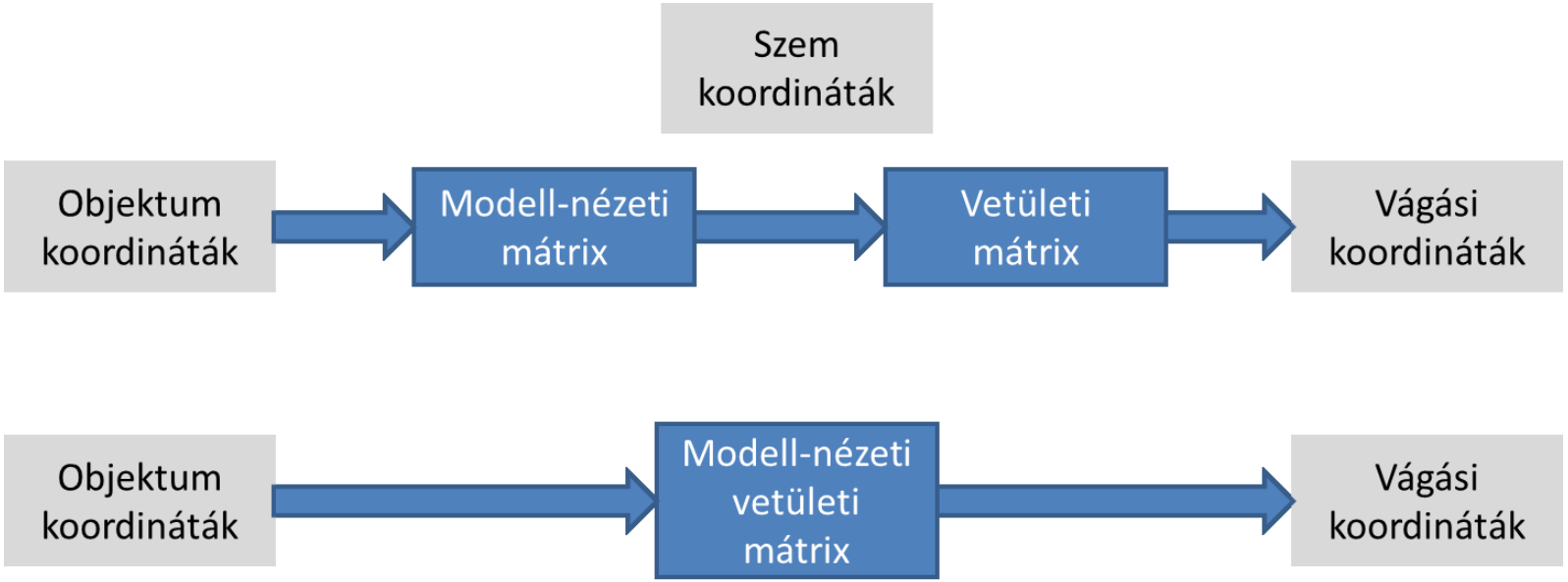
- A $\mathbf{P}_p$ mátrix
  - A $z = f$-et $+1$-re és a $z = n$-et -1-re képezi le
- A perspektív transzformáció végrehajtása után vágással és homogenizálással kapjuk meg a normalizált eszköz koordinátákat
- Először a $\mathbf{S}(1, 1, -1)$ mátrixszal szorzunk
  - A közeli és távoli értékek pozitív értékek lesznek

- A normalizált eszköz koordináták
  - A vágási koordináták homogén formában vannak tárolva
  - x és y értékekre van szükségünk mélységi értékekkel
  - Perspektív osztás
    - w-vel elosztjuk x, y és z értékeket
  - A geometriai adatok egy egységkockán belül lesznek

- Ablak koordináták
  - A normalizált vertex koordinátákat átkonvertáljuk a végső koordináta rendszerbe
  - x és y pixel pozíciókat használunk
  - Nézeti ablak
    - Adott ablakon belüli tér meghatározása
    - aktuális képernyő koordinátákban
  - A raszterizáló a vertexekből pontokat, vonalakat és poligonokat állít elő
- Mélység-távolság transzformáció
  - A vertexek z értékeit skálázza be a mélységpuffer értékeinek az intervallumába

Cg példa
Transzformált vertex pozíciók

- A vertex programok vertex pozíciókat kapnak az objektum térben
- A program mindegyik vertex-et megszorozza a modell-nézeti és vetületi mátrix-szal, mely a vágási területre viszi a vertex-eket
- Gyakorlatban két mátrix konkatenációja
  - Egy mátrix szorzás

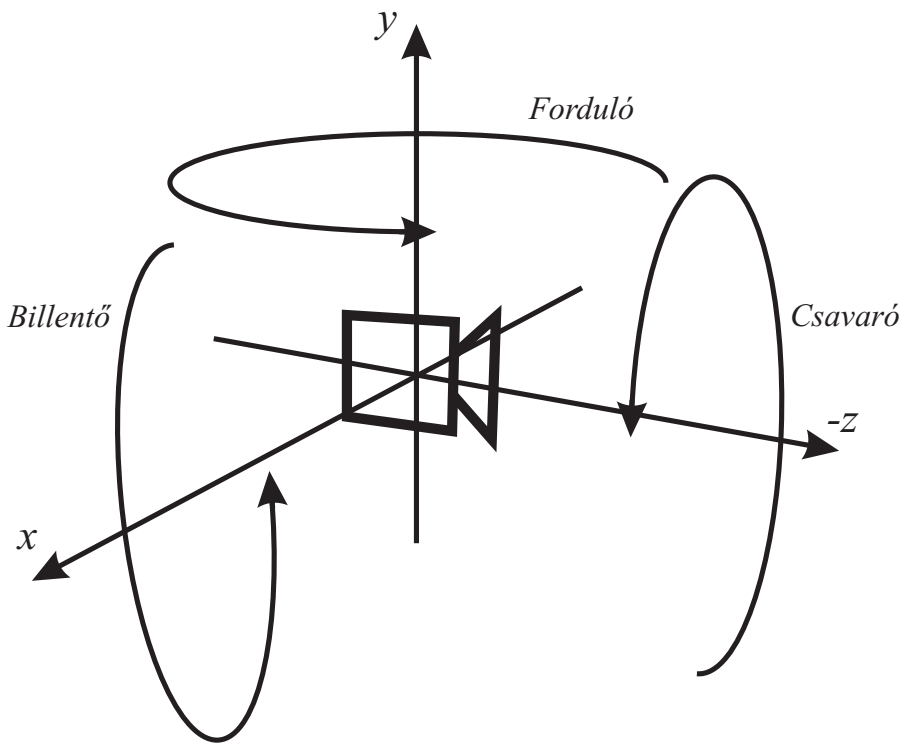


Példa

```
void C4E1v_transform(float4 position  : POSITION,
                     out float4 oPosition : POSITION,
                     out float4 color     : COLOR,
                     uniform float4x4 modelViewProj)
{
// A pozíció transzformálása
// az objektumérből a vágótérbe

  oPosition = mul(modelViewProj, position);

  color = float4(0.8, 0.8, 0.8, 1.0);
}
```

Speciális transzformációk

- Egy tetszőleges forgatás megadására
- Alkalmas egy kamera irányítottság beállítására
- A kamera az origóban, z tengely negatív irányába néz, a felfele mutató irány az y tengely pozitív irányába mutat
- Három mátrix szorzata
  - $\mathbf{E}(h, p, r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)$



- Az $\mathbf{E}$ forgatások összefűzésével áll elő
  - Ortogonális
- Inverze könnyen kiszámítható
  - $\mathbf{E}^{-1} = \mathbf{E}^T = (\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y)^T = \mathbf{R}_y^T\mathbf{R}_x^T\mathbf{R}_z^T$
  - Az $\mathbf{E}$ mátrix transzponáltja egyszerűbb
- Euler szögek
  - (h, p és r)

- Gimbal lock
  - A forgatások miatt egy szabadsági fokot elveszítünk
    - $h = 0$, $p = 90°$
    - Ezután a z-tengely körüli forgatás lényegében az y-tengely körüli forgatásnak felel meg
  - Nem tudunk a z világtengely körül forgatni

$$\mathbf{E}(h,\pi/2,r) = \begin{pmatrix} \cos r\cos h - \sin r\sin h & 0 & \cos r\sin h + \sin r cosh \\ \sin r\cos h + \cos r\sin h & 0 & \sin r\sin h - \cos r\cos h \\ 0 & 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \cos(r+h) & 0 & \sin(r+h) \\ \sin(r+h) & 0 & \cos(r+h) \\ 0 & 1 & 0 \end{pmatrix}$$

- Feladat: a h, p és r Euler paraméterek kinyerése az ortogonális mátrixból

$$\mathbf{F} = \begin{pmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{pmatrix} = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h) = \mathbf{E}(r,p,h)$$

$$\mathbf{F} = \begin{pmatrix} \cos r \cos h - \sin r \sin p \sin h & -\sin r \cos p & \cos r \sin h + \sin r \sin p \cos h \\ \sin r \cos h + \cos r \sin p \sin h & \cos r \cos p & \sin r \sin h - \cos r \sin p \cos h \\ -\cos p \sin h & \sin p & \cos p \cos h \end{pmatrix}$$

- $\cos p \neq 0$

$$h = \arctan(-\frac{f_{20}}{f_{22}}),$$
$$p = \arcsin(f_{21}),$$
$$r = \arctan(-\frac{f_{01}}{f_{11}}).$$

- $\cos p = 0$
  - $f_{01} = f_{11} = 0$
  - $\sin p = \pm 1$

$$\mathbf{F} = \begin{pmatrix} \cos(r \pm h)) & 0 & \sin(r \pm h) \\ \sin(r \pm h) & 0 & \cos(r \pm h) \\ 0 & \pm 1 & 0 \end{pmatrix}$$

- A többi paramétert $h = 0$-ra
  - $\sin r / \cos r = \tan r = f_{10}/f_{00}$
  - $r = \arctan(f_{10}/f_{00})$

- Különböző transzformációk paramétereinek meghatározása
  - Transzformációk összefűzésével előállított mátrixból
- Különböző esetek
  - Csak a skálázási paramétereket nyerjük ki
  - Meghatározni azt, hogy a modellen csak egy merevtest-transzformációt alkalmaztak-e vagy sem
  - Egy animáció kulcspozíciói közötti interpolálás végrehajtása
  - A nyírások eltávolítása a forgatási mátrixból

- Felbontások
  - Eltolás és forgatási mátrix származtatása
  - Triviális eltolási értékek kinyerése merevtest-transzformáció esetén
    - Ortogonális mátrix utolsó oszlopa
  - A mátrix determinánsa meghatározza, hogy végrehajtottak-e tükrözést
- A forgatás, skálázás és nyírás szétválasztása komolyabb erőfeszítést igényel

- **r** forgatási tengely normalizált
- A transzformáció α szöggel forgat **r** körül
- Találni kell két másik egység hosszú tetszőleges tengelyt
  - Egymásra és **r**-rel ortogonálisak (merőlegesek), azaz ortonormáltak
  - Ezek bázist alkotnak
- Ötlet
  - A bázist változtassuk meg az alapról az új bázisra
  - Forgassuk el az adott objektumot α szöggel mondjuk az x-tengely körül
  - Transzformáljunk vissza az alap bázisba

- Számítsuk ki a bázis ortonormált tengelyeit
  - Az első tengely az $\mathbf{r}$
- A második $\mathbf{s}$-tengely megkeresése
  - $\mathbf{t} = \mathbf{r} \times \mathbf{s}$
  - Numerikusan stabil módszer
    - $\mathbf{r}$ abszolút értékben legkisebb komponensének 0-ra állítása
    - A két másik komponens megcserélése után negáljuk az elsőt ezek közül

- Az r, s és t vektort helyezzük el egy mátrix soraiban

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix}$$

- A mátrix az $\mathbf{r}$ vektort az x-tengelybe, az $\mathbf{s}$ vektort az y-tengelybe és a $\mathbf{t}$ vektort a z-tengelybe transzformálja

$$\mathbf{X} = \mathbf{M}^T \mathbf{R}_x(\alpha)\mathbf{M}. \tag{1}$$

- Goldman módszere
  - Tetszőleges normalizált **r**-tengely körüli forgatásra ϕ szöggel

$$\mathbf{R} = \begin{pmatrix} \cos\phi + (1-\cos\phi)r_x^2 & (1-\cos\phi)r_x r_y - r_z \sin\phi & (1-\cos\phi)r_x r_z + r_y \sin\phi \\ (1-\cos\phi)r_x r_y + r_z \sin\phi & \cos\phi + (1-\cos\phi)r_y^2 & (1-\cos\phi)r_y r_z - r_x \sin\phi \\ (1-\cos\phi)r_x r_z - r_y \sin\phi & (1-\cos\phi)r_y r_z + r_x \sin\phi & \cos\phi + (1-\cos\phi)r_z^2 \end{pmatrix}$$

Kvaterniók

Kvaterniók

- A kvaterniókat vektorként ábrázoljuk és $\hat{\mathbf{q}}$-val jelöljük
- A $\hat{\mathbf{q}}$ kvaterniót a következőképpen definiáljuk

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w$$
$$\mathbf{q}_v = iq_x + jq_y + kq_z = (q_x, q_y, q_z)$$
$$i^2 = j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k$$

  - $\hat{\mathbf{q}}$ kvaternió valós része q_w
  - A képzetes része $\mathbf{q}_v$, ahol i, j és k a képzetes egységek
- q_v képzetes részen értelmezve van az összes vektor művelet
  - Például a skálázás, skaláris szorzat és kereszt szorzat

- Két $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ szorzatát a következőképpen tudjuk levezetni
  - A képzetes egységek szorzása nem kommutatív

$$\begin{aligned}\hat{\mathbf{q}}\hat{\mathbf{r}} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w) \\ &= i(q_y r_z - q_z r_y + r_w q_x + q_w r_x) \\ &+ j(q_z r_x - q_x r_z + r_w q_y + q_w r_y) \\ &+ k(q_x r_y - q_y r_x + r_w q_z + q_w r_z) \\ &+ q_w r_w - q_x r_x - q_y r_y - q_z r_z = \\ &= (\mathbf{q}_v \times \mathbf{r}_v + r_w \mathbf{q}_v + q_w \mathbf{r}_v), q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v\end{aligned}$$

Összeadás: $\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v + \mathbf{r}_v, q_w + r_w)$

Konjugált: $\hat{\mathbf{q}}^* = (\mathbf{q}_v, q_w)^* = (-\mathbf{q}_v, q_w)$

Norma: $n(\hat{\mathbf{q}}) = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*} = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \sqrt{\mathbf{q}_v \cdot \mathbf{q}_v + q_w^2} = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$

Egység: $\hat{\mathbf{i}} = (\mathbf{0}, 1)$

- Multiplikatív inverz esetén
  - $\hat{\mathbf{q}}^{-1}\hat{\mathbf{q}} = \hat{\mathbf{q}}\hat{\mathbf{q}}^{-1} = 1$

$$n(\hat{\mathbf{q}})^2 = \hat{\mathbf{q}}\hat{\mathbf{q}}^*$$
$$\Longleftrightarrow$$
$$\frac{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})^2} = 1$$

  - $\hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})^2}\hat{\mathbf{q}}^*$

Konjugálási szabályok

$$(\hat{\mathbf{q}}^*)^* = \hat{\mathbf{q}}$$
$$(\hat{\mathbf{q}} + \hat{\mathbf{r}})^* = \hat{\mathbf{q}}^* + \hat{\mathbf{r}}^*$$
$$(\hat{\mathbf{q}}\hat{\mathbf{r}})^* = \hat{\mathbf{r}}^*\hat{\mathbf{q}}^*$$

Normálási szabályok

$$n(\hat{\mathbf{q}}^*) = n(\hat{\mathbf{q}})$$
$$n(\hat{\mathbf{q}}\hat{\mathbf{r}}) = n(\hat{\mathbf{q}})n(\hat{\mathbf{r}})$$

Linearitás

$$\hat{\mathbf{p}}(s\hat{\mathbf{q}} + t\hat{\mathbf{r}}) = s\hat{\mathbf{p}}\hat{\mathbf{q}} + t\hat{\mathbf{p}}\hat{\mathbf{r}}$$
$$(s\hat{\mathbf{p}} + t\hat{\mathbf{q}})\hat{\mathbf{r}} = s\hat{\mathbf{p}}\hat{\mathbf{r}} + t\hat{\mathbf{q}}\hat{\mathbf{r}}$$

Asszocivitás

$$\hat{\mathbf{p}}(\hat{\mathbf{q}}\hat{\mathbf{r}}) = (\hat{\mathbf{p}}\hat{\mathbf{q}})\hat{\mathbf{r}}$$

- Egységkvaternió $\hat{\mathbf{q}} = (\mathbf{q}_v, q_w)$ normája 1-gyel egyenlő ($n(\hat{\mathbf{q}}) = 1$)
- $\hat{\mathbf{q}}$ megadható
  - $\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi) = \sin\phi\ \mathbf{u}_q + \cos\phi$
    - $\mathbf{u}_q$ egy három-dimenziós vektor
    - $\|\mathbf{u}_q\| = 1$

$$n(\hat{\mathbf{q}}) = n(\sin\phi\mathbf{u}_q, \cos\phi) = \sqrt{\sin^2\phi(\mathbf{u}_q \cdot \mathbf{u}_q) + \cos^2\phi}$$
$$= \sqrt{\sin^2\phi + \cos^2\phi} = 1$$

    - akkor és csak akkor, ha $\mathbf{u}_q \cdot \mathbf{u}_q = 1 = \|\mathbf{u}_q\|^2$

- Komplex számok esetén
  - Egy két dimenziós egység vektor megadható $\cos\phi + i\sin\phi = e^{i\phi}$ formában
  - Alkalmazható kvaterniók esetén is
    - $\hat{\mathbf{q}} = \sin\phi\ \mathbf{u}_q + \cos\phi = e^{\phi\mathbf{u}_q}$
- A logaritmus és hatvány függvények ezek alapján

Logaritmus:

$$\log\hat{\mathbf{q}} = \log(e^{\phi\mathbf{u}_q}) = \phi\mathbf{u}_q$$

Hatvány:

$$\hat{\mathbf{q}}^t = \left(e^{\phi\mathbf{u}_q}\right)^t = e^{\phi t\mathbf{u}_q} = \sin(\phi t)\ \mathbf{u}_q + \cos(\phi t)$$

- Egység hosszú kvaterniók (egységkvaterniók) tanulmányozása
  - Az egységkvaterniók egy három dimenziós forgatást fejezhetnek ki
- Egy pont vagy egy vektor négy koordinátáját $\mathbf{p} = (p_x, p_y, p_z, p_w)^T$ helyezzük el egy $\hat{\mathbf{p}}$ kvaternió komponenseibe
  - Tfh. van egy egységkvaterniónk $\hat{\mathbf{q}} = (\sin\phi\ \mathbf{u}_q, \cos\phi)$
  - Ekkor $\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1}$
    - $\hat{\mathbf{p}}$-t forgatja el (és így $\mathbf{p}$-t is) az $\mathbf{u}_q$-tengely körül 2ϕ szöggel
    - Ezt a forgatást bármely tengely körüli forgatás esetén használhatjuk
  - $\hat{\mathbf{q}}$ és $\hat{\mathbf{q}}^{-1}$ ugyanazt a forgatást hajtja végre
    - Egy mátrixból való kvaternió kinyerése akár $\hat{\mathbf{q}}$-val vagy $-\hat{\mathbf{q}}$-val is visszatérhet

- Adott $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ két egységkvaternió
- Két kvaternióval $\hat{\mathbf{p}}$-n végrehajtott összetett transzformáció
  - $\hat{\mathbf{r}}(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^* = \hat{\mathbf{c}}\hat{\mathbf{p}}\hat{\mathbf{c}}^*$
    - ahol $\hat{\mathbf{c}} = \hat{\mathbf{r}}\hat{\mathbf{q}}$ szintén egy egységkvaternió, amelyet $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ kvaterniók összefűzésével kapunk meg

- A mátrixszorzás egyes rendszerekben hardveresen támogatott
  - Hatékonyabb ilyen módon elvégezni a szorzásokat, mint az előzőekben megadott formában
- Szükség van a kvaterniók mátrix formába való átalakítása mindkét irányba

- Egy $\hat{\mathbf{q}}$ kvaternió az $\mathbf{M}^q$ mátrixba a következőképpen alakítható át

$$\mathbf{M}^q = \begin{pmatrix} 1 - s(q_y^2 + q_z^2) & s(q_x q_y - q_w q_z) & s(q_x q_z + q_w q_y) & 0 \\ s(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & s(q_y q_z - q_w q_x) & 0 \\ s(q_x q_z - q_w q_y) & s(q_y q_z + q_w q_x) & 1 - s(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

  - ahol $s = 2/n(\hat{\mathbf{q}})$
- Egységkvaterniók esetén ez a következőképpen egyszerűsödik:

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Nincs szükség trigonometrikus függvények használatára

- Fordított irányú átalakítás egy kicsit bonyolultabb
- Mátrix elemekből előállított különbségek

$$
\begin{aligned}
m^q_{21} - m^q_{12} &= 4 q_w q_x, \\
m^q_{02} - m^q_{20} &= 4 q_w q_y, \\
m^q_{10} - m^q_{01} &= 4 q_w q_z.
\end{aligned}
$$

- Ha q_w-t ismerjük, akkor a $\mathbf{v}_q$ és így $\hat{\mathbf{q}}$ kiszámítható

- A mátrix nyoma

$$\mathrm{tr}(\mathbf{M}^{\mathbf{q}}) = 4 - 2s(q_x^2 + q_y^2 + q_z^2) = 4\left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2}\right)$$

$$= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{n(\hat{\mathbf{q}})}$$

- Ebből következik

$$q_w = \frac{1}{2}\sqrt{\mathrm{tr}(\mathbf{M}^q)} \quad q_x = \frac{m_{21}^q - m_{12}^q}{4q_w}$$

$$q_y = \frac{m_{02}^q - m_{20}^q}{4q_w} \quad q_z = \frac{m_{10}^q - m_{01}^q}{4q_w}$$

- Numerikusan stabil eljárás
  - Kis számokkal való osztások elkerülése
- Legyen $t = q - w^2 - q_x^2 - q_y^2 - q_z^2$

$$
\begin{aligned}
m_{00} &= t + 2q_x^2, \\
m_{11} &= t + 2q_y^2, \\
m_{22} &= t + 2q_z^2, \\
u &= m_{00} + m_{11} + m_{22} = t + 2q_w^2,
\end{aligned}
$$

  - Melyik a legnagyobb a q_x, q_y, q_z és q_w közül
    - m_{00}, m_{11}, m_{22} és u alapján meghatározhatjuk
  - q_w esetén előző fólia

- Ellenkező esetben

$$
\begin{aligned}
4q_x^2 &= +m_{00} - m_{11} - m_{22} + m_{33} \\
4q_y^2 &= -m_{00} + m_{11} - m_{22} + m_{33} \\
4q_z^2 &= -m_{00} - m_{11} + m_{22} + m_{33} \\
4q_w^2 &= \mathrm{tr}(\mathbf{M}^q).
\end{aligned}
$$

  - Mátrix elemekből előállított különbségek segítségével a maradék $\hat{\mathbf{q}}$ komponenseket is meg lehet határozni

- Olyan művelet, amelyet $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ egységkvaternió és egy $t \in [0,1]$ paraméter esetén egy interpolált kvaterniót számít ki
  - Hasznos objektumok animálásánál
  - Kevésbé hasznos kamera irányítottságok interpolálásakor
    - A kamera felfele mutató vektor megdőlhet az interpolálás alatt
- Algebrai formája egy $\hat{\mathbf{s}}$ összetett kvaternió
  - $\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t\hat{\mathbf{q}}$
- Szoftveres megvalósításkor
  - $\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \texttt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(q-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin\phi t}{\sin\phi}\hat{\mathbf{r}}$

- ϕ kiszámításához $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ összefüggést használjuk fel
  - $t \in [0, 1]$ -re a `slerp` függvény egyedi interpolált kvaterniókat számít ki amelyek együtt a legrövidebb ívet alkotják egy négydimenziós egységgömbön $\hat{\mathbf{q}}(t = 0)$-tól $\hat{\mathbf{r}}(t = 1)$-ig
  - Az ív a körön helyezkedik el
    - A $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$ és az origó által meghatározott sík és a négy dimenziós egységgömb metszeteként alakul ki

- A `slerp` függvény tökéletesen alkalmas két orientáció/irányítottság közötti interpolációra
- Euler transzformációval elvégezni nem olyan egyszerű
  - Több Euler szög interpolálásakor gimbal lock állhat elő
- Kettőnél több orientáció esetén
  - $\hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_{n-1}$
  - Amennyiben a `slerp` függvényt alkalmazzuk minden szakaszon, akkor hirtelen irányváltás látható az orientáció interpolálásakor

- Valamilyen spline-t kell használunk interpoláláskor
  - Vezessük be $\hat{\mathbf{a}}_i$ és $\hat{\mathbf{b}}_{i+1}$ kvaterniókat $\hat{\mathbf{q}}_i$ és $\hat{\mathbf{q}}_{i+1}$ között
  - Interpoláció során áthaladunk a kezdeti $\mathbf{q}_i, i \in [0, \hat{\ldots}, n-1]$ orientációkon, de az $\hat{\mathbf{a}}_i$-ken nem
  - Arra használjuk, hogy jelezzük az érintőleges irányítottságokat az eredeti orientációknál

$$\hat{\mathbf{a}}_i = \hat{\mathbf{b}}_i = \hat{\mathbf{q}}_i \exp\left[-\frac{\log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i-1}) + \log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i+1})}{4}\right]$$

- A $\hat{\mathbf{q}}_i$-t, $\hat{\mathbf{a}}_i$-t és $\hat{\mathbf{b}}_i$-t a kvaterniók gömbi interpolációjára használjuk sima köbös spline használatával a következő egyenlet szerint

$$\texttt{squad}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t) =$$
$$\texttt{slerp}(\texttt{slerp}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t), \texttt{slerp}(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t), 2t(1-t))$$

- Egy $\mathbf{s}$ irányvektort egy $\mathbf{t}$ irányvektorba forgatunk
  - Normalizáljuk $\mathbf{s}$-t és $\mathbf{t}$-t
  - Kiszámítjuk az $\mathbf{u}$ egység forgatási tengelyt
    - $\mathbf{u} = (\mathbf{s} \times \mathbf{t}) / \|\mathbf{s} \times \mathbf{t}\|$ egyenlőség alapján
  - Legyen $e = \mathbf{a} \cdot \mathbf{t} = \cos(2\phi)$ és $\|\mathbf{s} \times \mathbf{t}\| = \sin(2\phi)$
    - ahol 2ϕ az $\mathbf{s}$ és $\mathbf{t}$ közötti szög
  - $\hat{\mathbf{q}} = (\mathbf{u} \sin\phi, \cos\phi)$
    - Az adott forgatást hajtja végre $\mathbf{s}$-ből $\mathbf{t}$-be
  - $\hat{\mathbf{q}} = \left(\frac{\sin\phi}{\sin 2\phi}(\mathbf{s} \times \mathbf{t}), \cos\phi\right)$-t egyszerűsítve

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s} \times \mathbf{t}), \frac{\sqrt{2(1+e)}}{2}\right)$$

    - A numerikus instabilitást elkerülhetjük
  - Nullával való osztás
    - Ellentétes irányú $\mathbf{t}$ és $\mathbf{s}$
    - Bármely $\mathbf{s}$-re merőleges forgatási tengely használható $\mathbf{t}$ forgatására

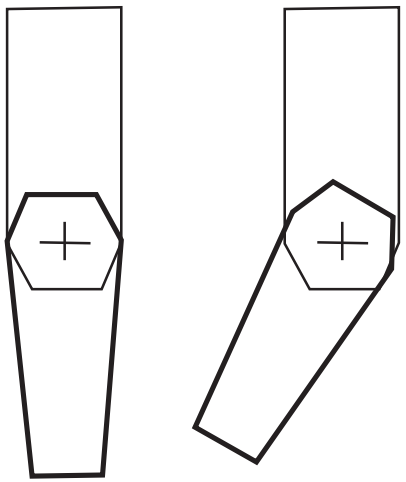
- **s**-ből **r**-be való forgatásmátrix ábrázolása

$$\mathbf{R}(\mathbf{s},\mathbf{t}) = \begin{pmatrix} e + hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e + hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z & hv_yv_z + v_x & e + hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

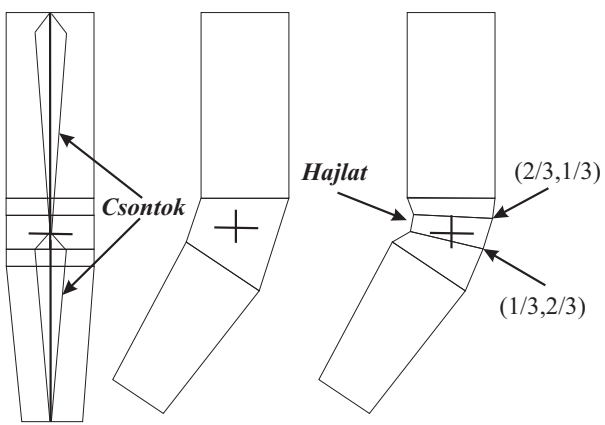
$$\begin{aligned} \mathbf{v} &= \mathbf{s} \times \mathbf{t}, \\ e &= \cos(2\phi) = \mathbf{s} \cdot \mathbf{t}, \\ h &= \frac{1 - \cos 2\phi}{\sin^2(2\phi)} = \frac{1 - e}{\mathbf{v} \cdot \mathbf{v}} = \frac{1}{1 + e}. \end{aligned}$$

Vertex keveredés

- Egy digitális karakter animálásakor az alkarját és felkarját mozgatjuk
- Merevtest-transzformáció
  - Nem hasonlít a két rész kapcsolódása a könyökre
- Csontokat definiálnak
  - Bőr vesz körül és a bőr a csontok mozgásának megfelelően változik
- Az al- és a felkart elkülönülve animáljuk
  - A két rész kapcsolódásánál egy rugalmas bőrrel kapcsoljuk össze a részeket



Merevtest-transzformáció



Vertex keveredés

Vertex keveredés

- A rugalmas rész egy vertex halmazát a felkar
- A másik vertex halmazt az alkar mátrixszal transzformáljuk
- Azok a háromszögek, amelyek vertexeit különböző mátrixokkal transzformáltuk, a transzformáció következtében megnyúlnak illetve összemennek

Vertex keveredés

- Megengedhetjük egy egyszerű vertex esetén
  - Több különböző mátrixszal transzformáljuk
  - Az eredményeket súlyozzuk és keverjük
  - $\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}$,
    - ahol $\sum_{i=0}^{n-1} w_i = 1$, $w_i \geq 0$
    - $\mathbf{p}$ az eredeti vertex
    - $\mathbf{u}(t)$ transzformált vertex
    - $\mathbf{M}_i$ mátrix az iniciális koordinátarendszerből a világ koordináta-rendszerbe transzformál
    - $\mathbf{B}_i(t)$ az i-ik csont világ transzformációja
    - w_i az i-ik csont súlya a $\mathbf{p}$ vertex esetén

Vertex keveredés

- Az $\mathbf{M}_i$ mátrix inverze modelltérből a csont saját terébe transzformál
- A $\mathbf{B}_i(t)$ csont aktuális transzformációjával vissza a modelltérbe transzformál
- A gyakorlatban a $\mathbf{B}_i(t)$ és az $\mathbf{M}_i^{-1}$ mátrixokat összefűzzük
  - Mindegyik csont és az animáció mindegyik képkockája esetén
  - Az eredménymátrixot használjuk a vertex transzformáció végrehajtására

Összefoglalás

Miről volt szó?

- Transzformációs csővezeték
- Speciális transzformációk
  - Euler transzformáció
  - Paraméterek kinyerése az Euler transzformációból
  - Mátrix felbontás
  - Forgatás tetszőleges tengely mentén
- Kvaterniók
- Vertex keveredés

Modellezés

- OpenGL állapotmodell
  - Amikor egy állapot értéke be van állítva, az addig nem változik meg, amíg egy másik függvény meg nem változtatja azt
  - Két lehetséges értékű állapotok
    - `glEnable()`
    - `glDisable()`
  - Különböző értékek
    - Lekérdezés pl. `glGetFloatv(GLenum pname, GLfloat *params)`

Egy objektum modellezése

- Az objektumot felépítő primitívek vertexeit külön-külön is megadhatjuk
  - Több ezer primitívből álló alakzatnál már problémás
- Bonyolult alakzatokat és azok attribútumait egy jól meghatározott struktúrájú adatszerkezetben tároljuk
  - Egyetlen függvényhívás segítségével jelenítjük meg
- Szabályok
  - Poligonoknak síkbeli alakzatoknak kell lenniük
  - Poligon élei nem metszhetik egymást
  - Poligonnak konvexnek kell lennie

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

```
//...
// Négyszögsáv kezdete
glBegin(GL_QUAD_STRIP);
// Egy kör mentén számítjuk ki a henger
// palástjának a vertex pontjait
  for(angle = 0.0f; angle <= (2.0f*GL_PI);
      angle -= 2*(GL_PI/n)) {
// A következő vertex x és y pozíciójának
// a kiszámítása R = 1 esetén
    x = cos(angle);
    y = sin(angle);
//...
// A négyszögsáv alsó és felső pontjainak
// a megadása
    if (h1 < h2) {
      lVertex3f(R1*x, R1*y, h1);
      glVertex3f(R2*x, R2*y, h2);
    } else {
      glVertex3f(R2*x, R2*y, h2);
      glVertex3f(R1*x, R1*y, h1);
    }
  }
glEnd();
```

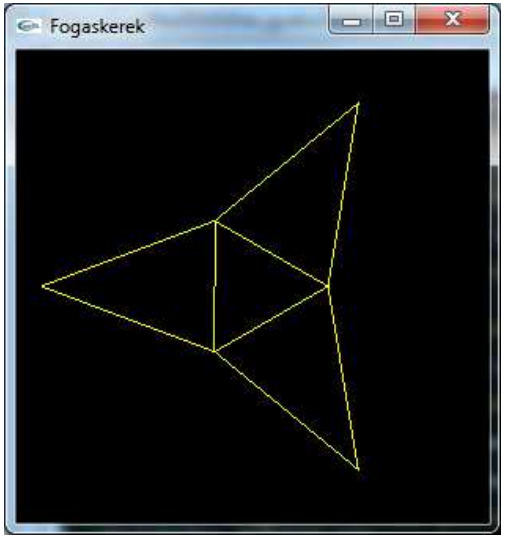
- Nézőponttól vett távolság eltárolása
  - z érték összehasonlítása azzal a z értékkel, amit már korábban eltároltunk
  - z értékek tárolását egy szín pufferrel megegyező méretű pufferrel
- Bizonyos esetekben szükséges a mélységpuffer írásának ideiglenes felfüggesztése
  - `glDepthMask(GL_FALSE)`
  - A mélység teszt ugyanúgy végrehajtódik a korábbi értékekkel

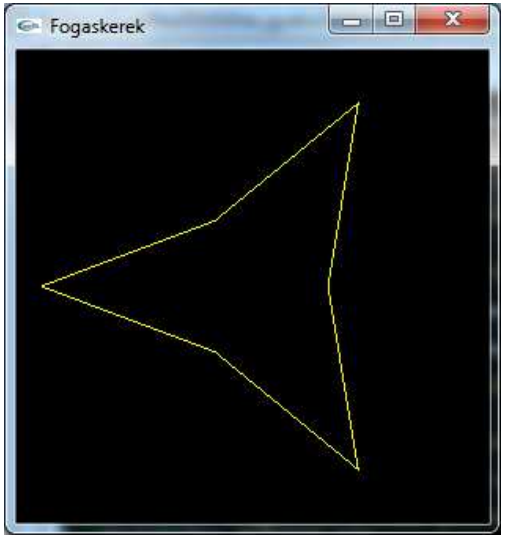
- Alapesetben a mélységellenőrzés a kisebb relációt használ a nem látható objektumhoz tartozó pixelek eltávolítására
- Lehetőség van az összehasonlító reláció megadására
  - `glDepthFunc(GLenum func)`

`GL_NEVER`	Mindig hamis
`GL_LESS`	Igaz, ha a bejövő mélység érték kisebb, mint az eltárolt érték
`GL_EQUAL`	Igaz, ha a bejövő mélység érték megegyezik az eltárolt értékkel
`GL_LEQUAL`	Igaz, ha a bejövő mélység érték kisebb vagy egyenlő, mint az eltárolt érték
`GL_GREATER`	Igaz, ha a bejövő mélység érték nagyobb, mint az eltárolt érték
`GL_NOTEQUAL`	Igaz, ha a bejövő mélység érték nem egyenlő az eltárolt értékkel
`GL_GEQUAL`	Igaz, ha a bejövő mélység érték nagyobb vagy egyenlő, mint az eltárolt érték
`GL_ALWAYS`	Mindig igaz

- A `GL_EQUAL` és `GL_NOTEQUAL` relációk esetén meg kell változtatni az alap $[0.0 - 1.0]$ mélység értékek tartományát
  - `glDepthRange(GLclampd nearVal, GLclampd farVal)`
    - Első paramétere a közeli vágósík
    - Második paramétere a távoli vágósík
  - A vágás és a homogén koordináták negyedik w elemével való osztás után, a mélység értékek a $[-1.0, 1.0]$ tartományba képződnek le
    - A `glDepthRange` adja meg a lineáris leképezését ezeknek a normalizált mélység koordinátáknak az ablak mélységértékeire nézve

- Drótvázas megjelenítéskor a belső alakzat éleit nem kell megjeleníteni
  - Az él flag-et hamisra kell állítani
- `glEdgeFlag`
  - `TRUE`
  - `FALSE`



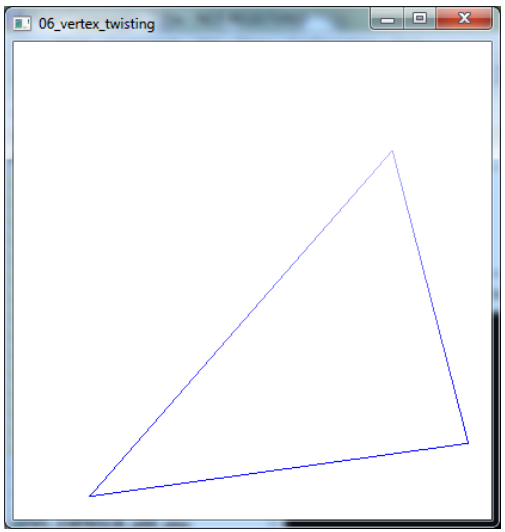


```
glBegin(GL_TRIANGLES);
for(i=0, angle=0.0; i<n; i++, angle += D_Angle)
{
   glEdgeFlag(FALSE);
   glVertex2f(x+radius* cos(angle),
              y+radius*sin(angle));
    glEdgeFlag(TRUE);
   glVertex2f(x+radius*cos(angle+Delta_Angle),
              y+radius*sin(angle+Delta_Angle));
   glVertex2f(x+2.8*radius*cos(angle+Half_DAngle),
              y+2.8*radius*sin(angle+Half_DAngle));
}
glEnd();
```

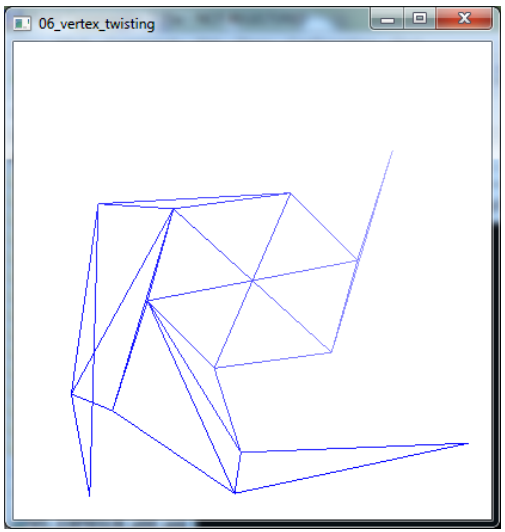
Példa

```
struct C3E4_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};
C3E4_Output C3E4v_twist(float2 position : POSITION,
                        float4 color    : COLOR,
                        uniform float twisting)
{
  C3E4_Output OUT;
  float angle = twisting * length(position);
  float cosLength, sinLength;
  sincos(angle, sinLength, cosLength);
  OUT.position[0] = cosLength * position[0] +
                   -sinLength * position[1];
  OUT.position[1] = sinLength * position[0] +
                    cosLength * position[1];
  OUT.position[2] = 0;
  OUT.position[3] = 1;
  OUT.color = color;
  return OUT;
}
```

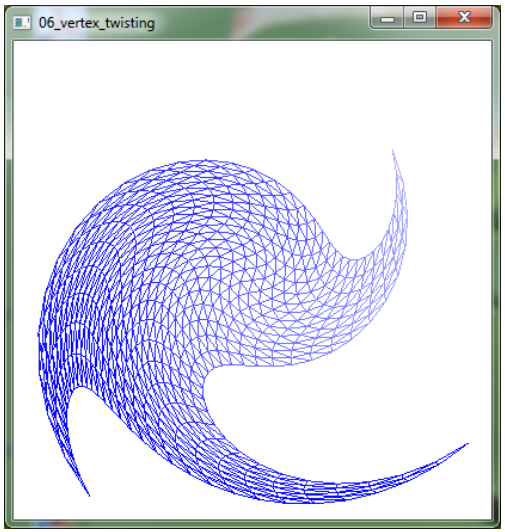
- Vertex elforgatása a középpont körül
  - A forgatási szög növelésével több vertex-re van szükség
- Általában, amikor egy vertex porgram nem lineáris számítást hajt végre, akkor megfelelő tesszalálásra van szükség az elfogadható eredmény eléréséhez



Alacsony



Közepes



Nagy

- Összetett alakzatok létrehozása OpenGL támogatással
  - OpenGL GLU segédfüggvénykönyvtár
    - Gömbök, hengerek, kúpok és sík korongok, illetve korongok lyukkal
    - Szabad-formájú felületekhez
    - Rendhagyó konkáv alakzatok, kisebb jobban kezelhető konvex alakzatokra való felbontása
- GLUT-os objektumok
  - Drótvázas/Kitöltött
  - Kocka, gömb, henger, stb.

Kvadratikus objektumok

- Másodfokú algebrai egyenletekkel leírható felületek
  - Pl. gömb, ellipszis, kúp, henger
- GLU segédfüggvény-könyvtár
  - Objektum orientált modell
    - Nagy paraméter lista elkerülése
    - Paraméterek beállítása függvényekkel
  - A felületekhez további attribútumokat/tulajdonságokat rendelhetünk
    - Normálvektorok
    - Textúra-koordináták
    - ...

- Egy üres kvadratikus objektum létrehozása, használata és törlése

```
GLUquadricObj *pObj;
// . . .
// Kvadratikus objektum létrehozása és inicializálása
pObj = gluNewQuadric();
// Renderelési paraméterek beállítása
// . . .
// Kvadratikus felület rajzolása
// . . .
// Kvadratikus objektum felszabadítása
gluDeleteQuadric(pObj);
```

- Egy üres kvadratikus objektum létrehozása, használata és törlése

```
GLUquadricObj *pObj;
// . . .
// Kvadratikus objektum létrehozása és inicializálása
pObj = gluNewQuadric();
// Renderelési paraméterek beállítása
// . . .
// Kvadratikus felület rajzolása
// . . .
// Kvadratikus objektum felszabadítása
gluDeleteQuadric(pObj);
```

- Egy üres kvadratikus objektum létrehozása, használata és törlése

```
GLUquadricObj *pObj;
// . . .
// Kvadratikus objektum létrehozása és inicializálása
pObj = gluNewQuadric();
// Renderelési paraméterek beállítása
// . . .
// Kvadratikus felület rajzolása
// . . .
// Kvadratikus objektum felszabadítása
gluDeleteQuadric(pObj);
```

- Rajzolási stílus
  - gluQuadricDrawStyle(GLUquadricObj *obj, GLenum drawStyle)

`GLU_FILL`	Solid objektumként jelenik meg
`GLU_LINE`	Drótvázas alakzatként jelenik meg
`GLU_POINT`	Vertex pontok halmazaként jelenik meg
`GLU_SILHOUETTE`	Hasonló a drótvázas megjelenéshez, de a poligonok szomszédos élei nem jelennek meg

- Normál egységvektorok automatikus generálása
  - `gluQuadricNormals(GLUquadricObj *pbj, GLenum normals)`
    - `GL_NONE` Normálvektorok nélkül
    - `GL_SMOOTH` Sima normálvektorok
      - Minden vertexhez külön-külön van meghatározva a normálvektor
    - `GL_FLAT` sík normálvektorok
      - Síkonként, adott háromszögre van kiszámítva a normálvektor

- Normálvektorok iránya
  - `gluQuadricOrientation(GLUquadricObj *obj, GLenum orientation)`

    `GLU_OUTSIDE` Kívülre mutat
    - Pl. megvilágított gömb

    `GLU_INSIDE` Belülre mutat
    - Pl. boltíves mennyezet

- Textúra-koordináták kiszámolása
  - `gluQuadricTexture(GLUquadricObj *obj, GLenum textureCoords)`
    - `GL_TRUE` Igen
      - A textúra-koordináták egyenletesen helyezkednek el a felületen
    - `GL_FALSE` Nem
      - Nem generálódnak textúra-koordináták

- Gömb
  - `gluSphere(GLUQuadricObj *obj, GLdouble radius, GLint slices, GLint stacks)`

    `radius` A gömb sugara

    `slices` A gyűrűkön belül lévő háromszögek/négyszögek száma
    - Hosszúsági körök száma

    `stacks` Gyűrűk száma
    - Szélességi körök száma

- Henger
  - `gluCylinder(GLUquadricObj *obj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks)`

`baseRadius`	Az origóhoz közelebbi oldalhoz tartozó sugár
`topRadius`	A másik oldalhoz tartozó sugár
`height`	A henger magassága
`slices`	Szeletek száma
`stacks`	Gyűrűk száma

  - Amennyiben a `baseRadius` vagy a `topRadius` nullával egyenlő, akkor egy kúpot kapunk eredményül

- Korong
  - `gluDisk(GLUquadricObj *obj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops)`

    `innerRadius` Belső sugár
    - Amennyiben nullával egyenlő, akkor egy tömör korongot kapunk
    - Amennyiben nem nulla, akkor egy lyukas korong az eredmény (csavar alátét)

    `outerRadius` Külső sugár

    `slices` Szeletek száma

    `loops` Gyűrűk szűma

Bézier görbék és felületek

- Parametrikus egyenletek
  - x, y és z egy másik változó függvényeként van megadva
  - A változó egy előre definiált intervallum értékeit veheti fel
  - Egy részecske időbeli mozgása

$$x = f(t),$$
$$y = g(t),$$
$$z = h(t),$$

ahol $f(t)$, $g(t)$ és $h(t)$ egyedi függvények.

- OpenGL-be görbe esetén u-val, felület esetén u-val és v-vel jelöljük a parametrikus görbe/felület paramétereit

- A görbe definiálásakor kontroll pontokkal befolyásolhatjuk annak az alakját
  - Az első és utolsó pont része a görbének
  - A többi kontrollpont mágnesként viselkedik
    - Maguk felé húzzák a görbét
- A görbe rangját a kontrollpontok száma határozza meg
- A görbe foka eggyel kisebb, mint annak a rangja

- A kontrollpontok matematikai jelentése a görbék parametrikus polinom egyenletekre vonatkoznak

  Rang Együtthatók száma
  Fok A legnagyobb kitevője a paraméternek
  - A leggyakoribbak a kubikos (harmadfokú) görbék

- Elméletileg tetszőleges rangú görbét megadhatunk
  - magasabb rangú görbék ellenőrizhetetlenül oszcillálni kezdenek
  - A kontrollpontok kis változtatására nagy mértékben megváltoznak

- Görbék közös vég- és kezdőpontjaikban leírja, hogy mennyire sima az átmenet közöttük
  - Semmilyen: Nincs közös pontja a két görbének
  - C0: Pozícióbeli
    - Egy közös pontban találkoznak
  - C1: Érintőleges
    - A két végpontban a két görbe érintője azonos
  - C2: Görbületi
    - A görbületi sugarak is megegyeznek a töréspontban
    - Az átmenet még simább

- `void glMap1f( GLenum target, GLfloat u1, GLfloat u2, GLint stride, GLint order, const GLfloat * points);`
  - `target` A kiértékelővel előállított adat típusa
    - Pl. `GL_MAP1_VERTEX_3`, `GL_MAP1_NORMAL`, `GL_MAP1_TEXTURE_COORD_1`, ...
  - `u1, u2` Az u lineáris leképezését adja meg
  - `stride` Két kontrollpont közötti float-ok vagy double-ok száma a `points` adatstruktúrában
  - `order` A kontrollpontok száma (pozitív)
  - `points` A kontrollpontokat tartalmazó tömbre mutató pointer

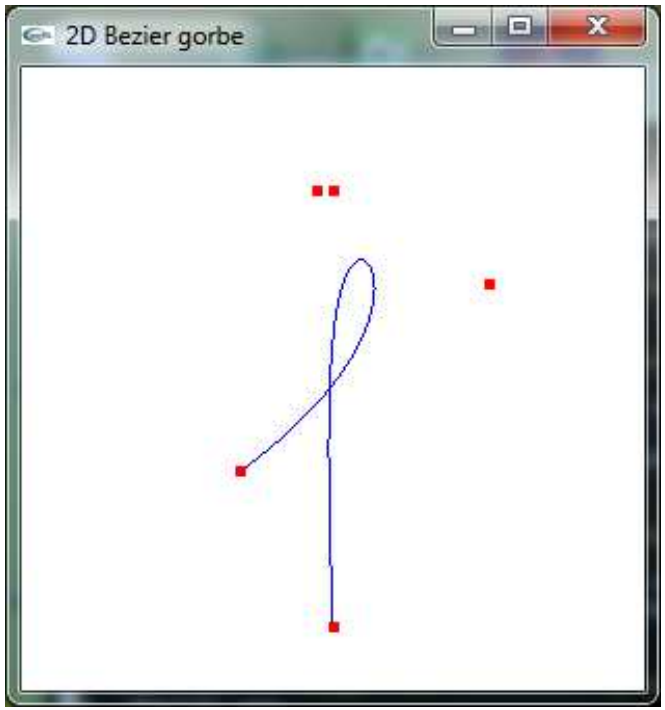
```
//A kontrollpontok száma
GLint nNumPoints = 5;

GLfloat ctrlPoints[5][3]=
// Végpont
{{ -3.0f, -3.0f, 0.0f},
// Kontrollpont
{ 5.0f, 3.0f, 0.0f},
// Kontrollpont
{ 0.0f, 6.0f, 0.0f},
// Kontorllpont
{ -0.5f, 6.0f, 0.0f},
// Végpont
{ 0.0f, -8.0f, 0.0f }};
```

```
glMap1f(
// Az előállított
// adat típusa
GL_MAP1_VERTEX_3,
// u alsó korlátja
0.0f,
// u felső korlátja
100.0f,
// A pontok közötti
// távolság az adatokban
3,
// Kontrollpontok száma
nNumPoints,
// Kontrollpontokat
// tartalmazó tömb mutatója
&ctrlPoints[0][0]);
```

- `glEnable(GL_MAP1_VERTEX_3);`
  - A kiértékelő engedélyezése
- `void glEvalCoord1f(GLfloat u)`

  `u` A paraméter értékének a megadása

```
// A pontok összekötése
// töredezett vonallal
glBegin(GL_LINE_STRIP);
for(i = 0; i <= 100; i++)
{
// A görbe kiértékelése
// az adott pontban
glEvalCoord1f((GLfloat) i);
}
glEnd();
```



- Egyszerűbb megvalósítás
  - `void glMapGrid1d( GLint un, GLfloat u1, GLfloat u2)`

    `un` A rács felosztása

    `u1, u2` Megadja a rácspontok leképezését.
  - `void glEvalMesh1(GLenum mode, GLint i1, GLint i2)`

    `mode` `GL_POINT` vagy `GL_LINE`

    `i1, i2` Az első és utolsó érték a tartományon

```
// Leképezi a 100 pont rácsát a 0 - 100 intervallumra
glMapGrid1d(100,0.0,100.0);

// Kiértékeli a rácsot és vonalakkal megjeleníti azt
glEvalMesh1(GL_LINE,0,100);
```

- `void glMap2f( GLenum target, GLfloat u1, GLfloat u2, GLint ustride, GLint uorder, GLfloat v1, GLfloat v2, GLint vstride, GLint vorder, const GLfloat *points)`

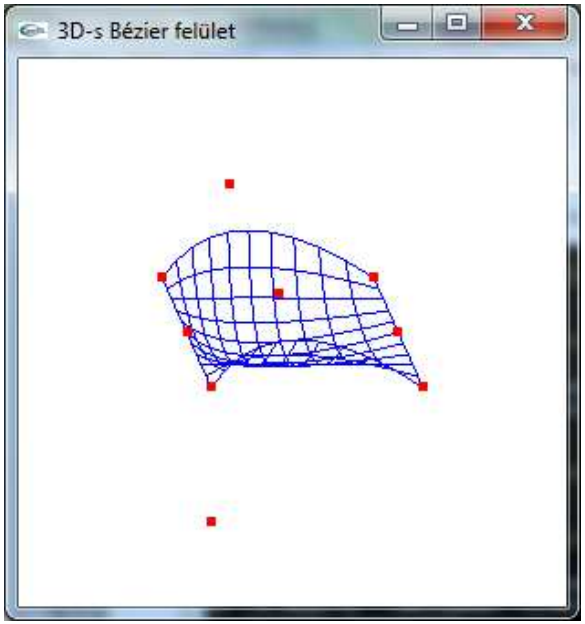
`target`	A kiértékelővel előállított adat típusa
`u1, u2`	Az u lineáris leképzését adja meg
`ustride`	Két u értelmezési tartományban lévő kontrollpont közötti float-ok vagy double-ok száma a `points` adatstruktúrában
`uorder`	A kontroll pontokat tartalmazó tömb dimenziója u tengely mentén
`v1, v2`	Az v lineáris leképzését adja meg
`vstride`	Két v értelmezési tartományban lévő kontrollpont közötti float-ok vagy double-ok száma a `points` adatstruktúrában
`vorder`	A kontroll pontokat tartalmazó tömb dimenziója v tengely mentén
`points`	Kontrollpontokat tartalmazó tömbre mutató pointer

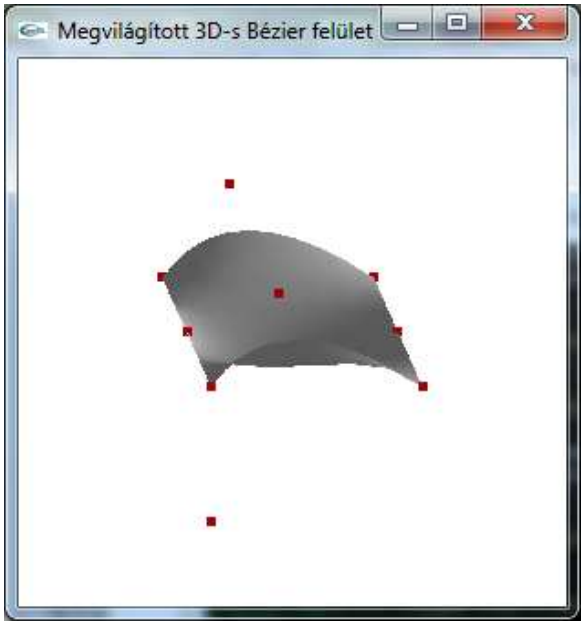
- Egyszerűbb megvalósítás
  - `void glMapGrid2f( GLint un, GLfloat u1, GLfloat u2, GLint vn, GLfloat v1, GLfloat v2 );`

    `un` A rács felosztása u irányban
    `u1, u2` Megadja a rácspontok leképezését
    `vn` A rács felosztása v irányban
    `v1, v2` Megadja a rácspontok leképezését
  - `void glEvalMesh2( GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2)`

    `mode` `GL_POINT` vagy `GL_LINE`
    `i1, i2` Az első és utolsó érték az u tartományon
    `j1, j2` Az első és utolsó érték az v tartományon

```
// Kiértékelő engedélyezése
glEnable(GL_MAP2_VERTEX_3);

// Magasabb szintű függvény a rács leképezésére
// A rács 10 pontjának a leképezése a 0 - 10 tartományra
glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);

// A rács kiértékelése vonalakkal
glEvalMesh2(GL_LINE, 0, 10, 0, 10);
```





GLUT-os objektumok

- A GLUT függvénykönyvtár 3D-s geometriai objektumok létrehozására alkalmas függvények
- Normálvektorokat létrehoz
  - Nem generál textúra-koordinátákat (kivéve teáskanna)
- Pl. gömb és kúp esetén a GLUT-os megvalósítás az előbb ismertetett kvadratikus objektumokat megvalósító függvényeket használja
  - Az adott GLUT-os objektumok paraméterlistája nagyon hasonló a kvadratikus objektumokat megvalósító függvényekhez

Tömör alakzat	Drótvázas alakzat	3D-s objektum
`glutSolidSphere`	`glutWireSphere`	Gömb
`glutSolidCube`	`glutWireCube`	Kocka
`glutSolidCone`	`glutWireCone`	Kúp
`glutSolidTorus`	`glutWireTorus`	Tórusz
`glutSolidDodecahedron`	`glutWireDodecahedron`	Dodekaéder
`glutSolidOctahedron`	`glutWireOctahedron`	Oktaéder
`glutSolidTetrahedron`	`glutWireTetrahedron`	Tetraéder
`glutSolidIcosahedron`	`glutWireIcosahedron`	Ikozaéder
`glutSolidTeapot`	`glutWireTeapot`	Teáskanna

Összefoglalás

- Egy objektum felépítése
- Kvadratikus objektumok
- Bézier görbék és felületek
- GLUT-os objektumok

Megvilágítás-árnyalás, átlátszóság és köd

Árnyalás

Megvilágítás-árnyalás

- Fotorealisztikus előállítása egy háromdimenziós világnak
  - Valósághű geometriai megjelenés
  - Valósághű külső megjelenés
    - Anyagi tulajdonságok felületekhez való hozzárendelése
    - Különböző fajta fényforrások alkalmazása
    - Textúrák hozzáadása
    - Köd
    - Átlátszóság használata

- Fényforrástípusok

  Irányított fények A fényforrás az megvilágított objektumtól végtelen távoli pontban helyezkedik el

  Pontfények, reflektorfények Pozícionális fények, mind a kettő rendelkezik egy pozícióval a térben

- Mind a három fényforrás esetén megadhatunk intenzitásra vonatkozó paramétereket és szín (RGB) értékeket

Jelölés	Leírás
$\mathbf{s}_{amb}$	Ambiens intenzitás szín
$\mathbf{s}_{diff}$	Diffúz intenzitás szín
$\mathbf{s}_{spec}$	Spekuláris intenzitás szín
$\mathbf{s}_{pos}$	Négy elemű fényforrás pozíció

- További reflektorfényre vonatkozó paraméterek
  - $\mathbf{s}_{dir}$ Irányvektor
  - $\mathbf{s}_{cut}$ Levágási szög
  - $\mathbf{s}_{exp}$ Kúpon belüli elnyelődés kontrollálása
- Pozícionális fényforrások távolság alapú intenzitás vezérlésére
  - $\mathbf{s}_c, \mathbf{s}_l$ és $\mathbf{s}_q$ Csillapítás vezérlése

Anyagi tulajdonságok

- Egy felület színét az anyaghoz tartozó paraméterekkel, a fényforrások paramétereivel (melyek megvilágítják a felületet) és egy megvilágítási modellel határozhatjuk meg

Jelölés	Leírás
$\mathbf{m}_{amb}$	Ambiens anyag szín
$\mathbf{m}_{diff}$	Diffúz anyag szín
$\mathbf{m}_{spec}$	Spekuláris anyag szín
$\mathbf{m}_{shi}$	Fényesség paraméter
$\mathbf{m}_{emi}$	Emisszív anyag szín

Megvilágítás és árnyalás

Megvilágítás Megadjuk az anyag és fényforrások paramétereivel meghatározott látható szín értékeit

Árnyalás Az a folyamat, amely végrehajtja a megvilágítási számításokat és meghatározza azokból a pixelek színeit

- Flat, sík
- Goraud
- Phong

Árnyalás

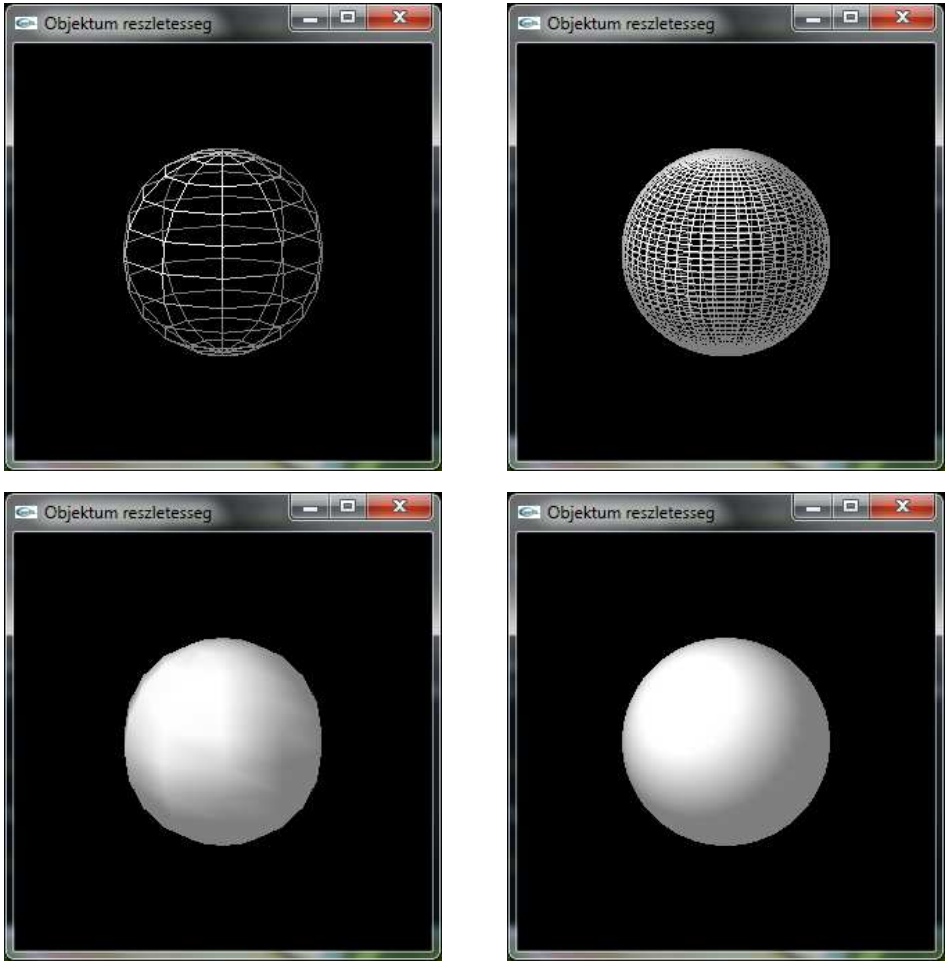
Flat A szín egy háromszögre van kiszámítva és a háromszög ezzel a színnel van kitöltve

Goraud A megvilágítás a háromszög mindegyik vertexe esetén meg van határozva és ezeket a színeket interpolálva a háromszög felületén kapjuk a végső eredményt

Phong A vertexekben tárolt árnyalási normálvektorokat interpolálva határozzuk meg a pixelenkénti normálvektorokat a háromszögben. Ezeket a normálvektorokat használva számítjuk ki a megvilágítás hatását az adott pixelben

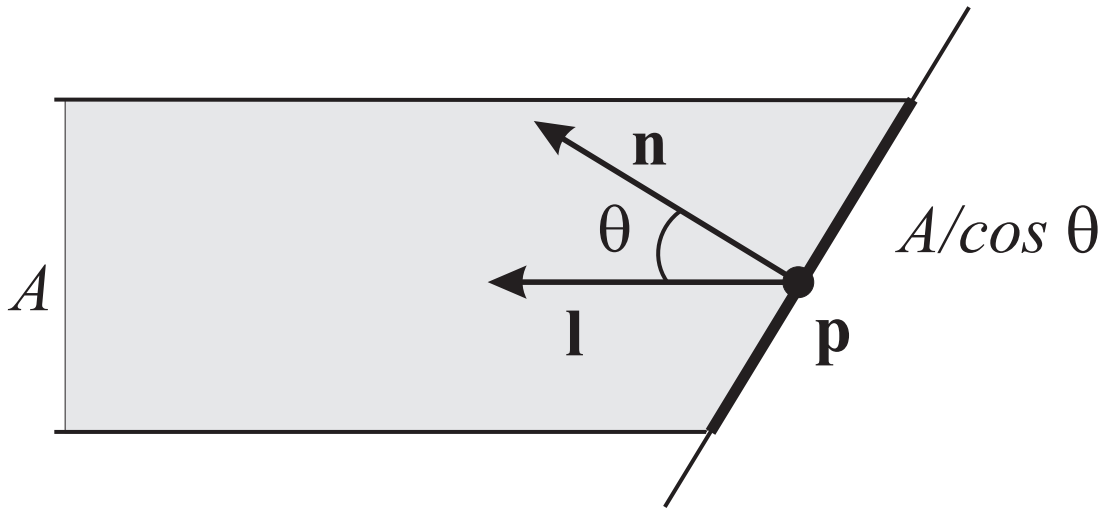
Árnyalás

- A flat árnyalást könnyű megvalósítani
  - Nem ad olyan sima eredményt görbe felület esetén
  - Meg lehet különböztetni a modellt felépítő primitíveket illetve felületeket
- A Gouraud árnyalás függ az objektum részletességétől



- Kevésbé függ az objektum kidolgozottságától
  - Felületi normálvektorok interpolálása
  - Pixelenkénti megvilágítás kiszámítása
- Kiszámítása bonyolultabb és költségesebb
- Korábban ezt a fajta módszert kevésbé használták
- Gouraud árnyalással hasonló eredményt lehet elérni
  - A felület pixelnél kisebb háromszögekre való felosztása
  - Nagyon lassú lehet
  - Nem programozható grafikus hardveren is megvalósítható

- Megfelel a fizikai valóságnak valamint a fény és felület kölcsönhatásának
- Lambert törvényen alapul
  - Az ideális diffúz (teljesen matt és nem csillogó) felületeknél a visszavert fény mértéke az **n** felületi normál és az **l** fényvektor közötti ϕ szög koszinuszától függ



$$\mathbf{i}_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos \phi,$$

$\mathbf{i}_{diff}$ a szem irányában visszavert fény mértékét megadó fizikai mennyiség

$\phi > \pi/2$ esetén nullával egyenlő.

A felület a fénnyel ellentétes irányba néz

- A megvilágítási egyenlet diffúz komponense független a kamera pozíciójától és irányától
- A megvilágított felület bármely irányból ugyanúgy néz ki
- fényforrás $\mathbf{s}_{diff}$ és az anyag $\mathbf{m}_{diff}$ diffúz színét használva

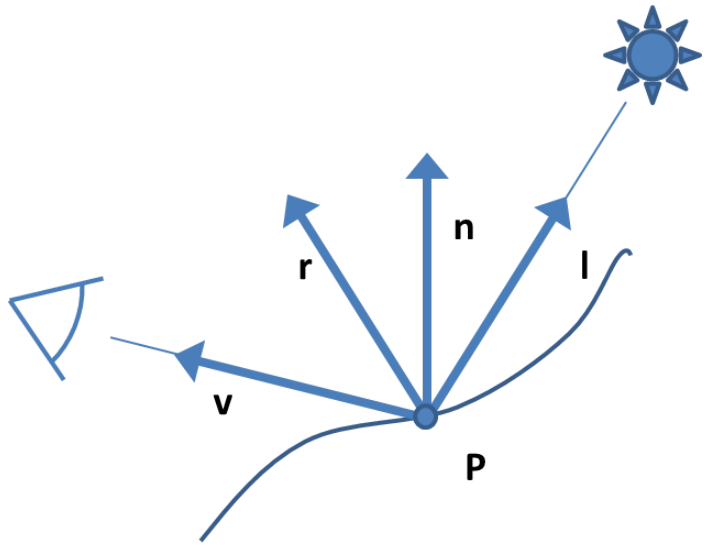
$$\mathbf{i}_{diff} = \max((\mathbf{n} \cdot \mathbf{l}), 0)\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$

  - Az $\mathbf{i}_{diff}$ a szín diffúz tagja
  - Az $\otimes$ operátor a komponensenkénti szorzás
  - $\max((\mathbf{n} \cdot \mathbf{l}), 0)$
    - 0, ha $\mathbf{n}$ és $\mathbf{l}$ közötti szög értéke nagyobb, mint $\pi/2$

- A spekuláris komponens a felület csillogásáért felelős
- Világos foltként jelenik meg a felületen
  - A felület görbeségét hangsúlyozza ki
  - Segít a fényforrások irányának és helyének a meghatározásában
- Phong megvilágítási egyenlet:

$$\mathbf{i}_{spec} = (\mathbf{r} \cdot \mathbf{v})^{m_{shi}} = (\cos \rho)^{m_{shi}}$$

  - **v** a **p** felületi pontból a nézőpont felé mutató vektor
  - az **r** az **l** fény vektor **n** normálvektorral meghatározott visszaverődése
  - A spekuláris összetevő annál erősebb, minél jobban egybeesik az **r** visszaverődési vektor és a **v** nézőpont vektor



- Az **l** fény vektor az **n** normálvektorra nézve az **r** vektor irányában verődik vissza
- Az **r** vektort a következőképpen lehet meghatározni:

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}$$

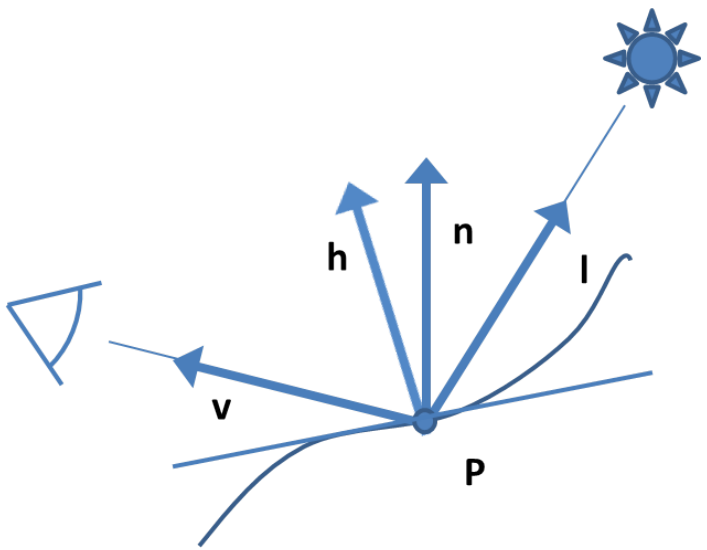
- Amennyiben $\mathbf{n} \cdot \mathbf{l} < 0$
  - A felület nem látható a fényforrásból nézve

- Blinn egyenlete

$$\mathbf{i}_{spec} = (\mathbf{n} \cdot \mathbf{h})^{m_{shi}} = (\cos \phi)^{m_{shi}}$$

  - **h** az **l** és **v** között lévő normalizált vektor

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$



- A **h** annak a síknak a normálisa a **p** pontban, amely a fényforrásból tökéletesen veri vissza a fényt a nézőpontba
  - Az $\mathbf{n} \cdot \mathbf{h}$ tag akkor maximális, ha az **n** normális **p** pontban egybeesik a **h** vektorral
  - Az $\mathbf{n} \cdot \mathbf{h}$ tényező abban az esetben csökken, amikor az **n** és **h** között a szög növekszik
- Nem kell kiszámítani az **r** visszaverődési vektort
- A kétfajta spekuláris megvilágítás közötti közelítés

$$(\mathbf{r} \cdot \mathbf{v})^{m_{shi}} \approx (\mathbf{n} \cdot \mathbf{h})^{4m_{shi}}$$

- OpenGL és a Direct3D megvalósítás

$$\mathbf{i}_{spec} = \max((\mathbf{n} \cdot \mathbf{h}), 0)^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

- m_{shi} a felület csillogásának a mértékét írja le
  - Értékének növelésével azt a hatást érjük el, hogy a világos terület nagysága beszűkül
- Schlick adott egy alternatív megközelítést Phong egyenletére

$$t = \cos \rho,$$

$$\mathbf{i}_{spec} = \frac{t}{m_{shi} - t m_{shi} + t} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

- A megvilágítási modellünkben a fények közvetlenül ragyognak a felületeken
- A valóságban a fény a fényforrásból kiindulva egy másik felületről visszaverődve is elérheti a tárgyat
- A másik felületről érkező fény nem számítható be sem a spekuláris sem pedig a diffúz komponensbe
- Indirekt megvilágítás szimulálása
  - A megvilágítási modellbe belevesszük az ambiens tagot
  - Csak valamilyen kombinációja az anyagi és fény konstansoknak

$$\mathbf{i}_{amb} = \mathbf{m}_{amb} \otimes \mathbf{s}_{amb}$$

- Egy tárgy valamilyen minimális mennyiségű színnel fog rendelkezni
  - Még akkor is, ha nem közvetlen módon lesz megvilágítva
- Azok a felületek, melyek nem a fény felé néznek nem fognak teljesen feketén megjelenni

- OpenGL
  - Támogatja a fényforrásonkénti ambiens értéket
    - Amikor a fényt kikapcsoljuk, akkor az ambiens összetevő automatikusan el lesz távolítva
- Csak ambiens tagot használva nem kapunk megfelelő eredményt
  - Eltűnik a három-dimenziós hatás
- Mindegyik objektum meg legyen világítva legalább egy kicsi direkt megvilágítással
  - Fényeket helyezünk el a színtéren
  - Fejlámpa (headlight) használta, amely egy a nézőponthoz kapcsolt pontfény
  - A spekuláris komponensét kikapcsoljuk, hogy kevésbé zavarjon

- Lokális megvilágítási modell
  - A megvilágítás csak a fényforrásokból származó fénytől függ
  - Más felületről nem érkezik fény
- A megvilágítást az ambiens, diffúz és spekuláris komponensek határozzák meg

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec}$$

```
void C5E1v_basicLight(float4 position   : POSITION,
                      float3 normal     : NORMAL,

                  out float4 oPosition : POSITION,
                  out float4 color     : COLOR,

                  uniform float4x4 modelViewProj,
                  uniform float3 globalAmbient,
                  uniform float3 lightColor,
                  uniform float3 lightPosition,
                  uniform float3 eyePosition,
                  uniform float3 Ka,
                  uniform float3 Kd,
                  uniform float3 Ks,
                  uniform float  shininess)
```

```
oPosition = mul(modelViewProj, position);

float3 P = position.xyz;
float3 N = normal;
```

Cg vertex program - alap megvilágítás

Ambiens és diffúz tag kiszámítása

```
// Ambiens tag
float3 ambient = Ka * globalAmbient;

// Diffúz tag
float3 L = normalize(lightPosition - P);
float diffuseLight = max(dot(N, L), 0);
float3 diffuse = Kd * lightColor * diffuseLight;
```

Cg vertex program - alap megvilágítás
Spekuláris tag kiszámítása

```
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);

float specularLight =
    pow(max(dot(N, H), 0), shininess);

if (diffuseLight <= 0)
    specularLight = 0;

float3 specular =
    Ks * lightColor * specularLight;
```

```
color.xyz = ambient + diffuse + specular;
color.w = 1;
```

Cg fragmens program - alap megvilágítás

Vertex program

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                  float3 normal   : NORMAL,

                  loat4 oPosition : POSITION,
                  out float3 objectPos : TEXCOORD0,
                  out float3 oNormal   : TEXCOORD1,
                  uniform float4x4 modelViewProj)

  oPosition = mul(modelViewProj, position);

  objectPos = position.xyz;
  oNormal = normal;
```

Cg fragmens program - alap megvilágítás

Vertex program

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                  float3 normal   : NORMAL,

                  loat4 oPosition : POSITION,
                  out float3 objectPos : TEXCOORD0,
                  out float3 oNormal   : TEXCOORD1,
                  uniform float4x4 modelViewProj)

  oPosition = mul(modelViewProj, position);

  objectPos = position.xyz;
  oNormal = normal;
```

Cg fragmens program - alap megvilágítás

Vertex program

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                  float3 normal   : NORMAL,

                  loat4 oPosition : POSITION,
                  out float3 objectPos : TEXCOORD0,
                  out float3 oNormal   : TEXCOORD1,
                  uniform float4x4 modelViewProj)

  oPosition = mul(modelViewProj, position);

  objectPos = position.xyz;
  oNormal = normal;
```

Cg fragmens program - alap megvilágítás
Paraméterek

```
void C5E3f_basicLight(float4 position  : TEXCOORD0,
                      float3 normal    : TEXCOORD1,

                  out float4 color     : COLOR,

                  uniform float3 globalAmbient,
                  uniform float3 lightColor,
                  uniform float3 lightPosition,
                  uniform float3 eyePosition,
                  uniform float3 Ka,
                  uniform float3 Kd,
                  uniform float3 Ks,
                  uniform float  shininess)
```

```
void C5E3f_basicLight(float4 position  : TEXCOORD0,
                      float3 normal    : TEXCOORD1,

                  out float4 color     : COLOR,

                  uniform float3 globalAmbient,
                  uniform float3 lightColor,
                  uniform float3 lightPosition,
                  uniform float3 eyePosition,
                  uniform float3 Ka,
                  uniform float3 Kd,
                  uniform float3 Ks,
                  uniform float  shininess)
```

```
void C5E3f_basicLight(float4 position  : TEXCOORD0,
                      float3 normal    : TEXCOORD1,

                  out float4 color     : COLOR,

                  uniform float3 globalAmbient,
                  uniform float3 lightColor,
                  uniform float3 lightPosition,
                  uniform float3 eyePosition,
                  uniform float3 Ka,
                  uniform float3 Kd,
                  uniform float3 Ks,
                  uniform float  shininess)
```

Cg fragmens program - alap megvilágítás

Függvénytörzs

```
float3 P = position.xyz;
float3 N = normalize(normal);

// Ambiens tag
float3 ambient = Ka * globalAmbient;

// Diffúz tag
float3 L = normalize(lightPosition - P);
float diffuseLight = max(dot(L, N), 0);
float3 diffuse = Kd * lightColor * diffuseLight;

// Spekuláris tag
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);
float specularLight = pow(max(dot(H, N), 0), shininess);
if (diffuseLight <= 0) specularLight = 0;
float3 specular = Ks * lightColor * specularLight;

color.xyz = emissive + ambient + diffuse + specular;
color.w = 1;
```

- A valóságban a fény intenzitása fordítottan arányos a fényforrástól mért távolság négyzetével

$$d = \frac{1}{s_c + s_l \left\| \mathbf{s}_{pos} - \mathbf{p} \right\| + s_q \left\| \mathbf{s}_{pos} - \mathbf{p} \right\|^2}$$

  - $\left\| \mathbf{s}_{pos} - \mathbf{p} \right\|$ az $\mathbf{s}_{pos}$ fényforrás pozíciójától vett távolság a $\mathbf{p}$ pontig
  - s_c a konstans, az s_l a lineáris és a s_q a kvadratikus csillapítást kontrollálják
- A fizikailag korrekt távolság csillapításhoz
  - $s_c = 0$, $s_l = 0$ és $s_q = 1$

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec})$$

Cg fragmens program - távolság függés

```
// Anyagi tulajdonságok
struct Material {
  float3 Ka;
  float3 Kd;
  float3 Ks;
  float shininess;
};

// Fény paraméterek
struct Light {
  float3 position;
  float3 color;
  float kC, kL, kQ;
};

// Távolságtól függő skalár meghatározása
float C5E6_attenuation(float3 P,
                       Light light)
{
  float d = distance(P, light.position);
  return 1 / (light.kC + light.kL * d + light.kQ * d * d);
}
```

```
void C5E7_attenuateLighting(Light light,
                            float3 P,
                            float3 N,
                            float3 eyePosition,
                            float  shininess,

                            out float3 diffuseResult,
                            out float3 specularResult)
```

```
// Elnyelődés kiszámítása
float attenuation = C5E6_attenuation(P, light);

// Diffúz komponens kiszámítása
float3 L = normalize(light.position - P);
float diffuseLight = max(dot(L, N), 0);
diffuseResult = attenuation *
                light.color * diffuseLight;

// Spekuláris komponens kiszámítása
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);
float specularLight = pow(max(dot(H, N), 0),
                          shininess);
if (diffuseLight <= 0) specularLight = 0;
specularResult = attenuation *
                 light.color * specularLight;
```

Cg fragmens program - távolság függés

Belépő függvény - paraméterek

```
void oneLight(float4 position : TEXCOORD0,
              float3 normal   : TEXCOORD1,

      out float4 color      : COLOR,

      uniform float3   eyePosition,
      uniform float3   globalAmbient,
      uniform Light    lights[1],
      uniform Material material)
```

Cg fragmens program - távolság függés

Belépő függvény

```
// Ambiens
float3 ambient = material.Ka * globalAmbient;


float3 diffuseLight;
float3 specularLight;
float3 diffuseSum  = 0;
float3 specularSum = 0;

//Diffúz és spekuláris komponensek
//távolság függő kiszámítása
C5E7_attenuateLighting(lights[0], position.xyz, normal,
                       eyePosition, material.shininess,
                       diffuseLight, specularLight);
diffuseSum += diffuseLight;
specularSum += specularLight;

// Anyagi tulajdonságok figyelembevétele
float3 diffuse = material.Kd * diffuseSum;
float3 specular = material.Ks * specularSum;

color.xyz =  ambient + diffuse + specular;
color.w = 1;
```

- A reflektorfény a színteret különböző módon világítja meg
  - c_{spot}-tal jelölt szorzótényező

$$c_{spot} = \max(-\mathbf{l} \cdot \mathbf{s}_{dir}, 0)^{s_{exp}}$$

$\mathbf{l}$	A fény vektor
$\mathbf{s}_{dir}$	A reflektor iránya
s_{exp}	Az exponenciális faktor a reflektor középpontjától való halványodását vezérli

- Módosított megvilágítási egyenlet

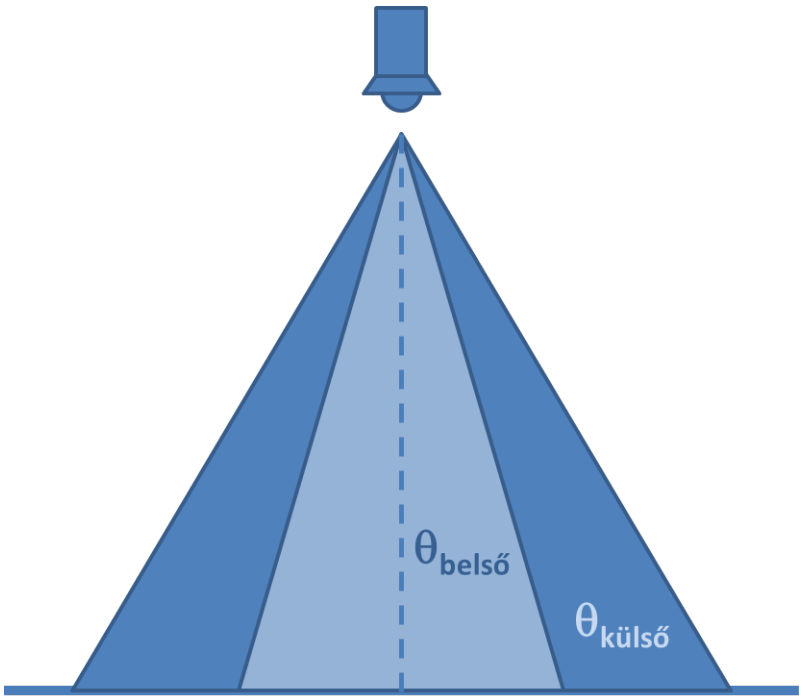
$$\mathbf{i}_{tot} = c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec}))$$

  - Ha a fényforrásunk nem reflektorfény, akkor $c_{spot} = 1$

Cg fragmens program - reflektorfény

Intenzitás változás

- Belső és külső "kúpok"
  - Könnyű eldönteni, hogy melyik részbe esik egy pont
  - Változtatni kell az intenzitás kiszámítását



Cg fragmens program - reflektorfény

Intenzitás változás

```
float saturate(float x)
{
  return max(0, min(1, x));
}

float smoothstep(float min,
                 float max,
                 float x)
{
  float t =
  saturate((x - min)/(max -
      min));
  return t*t*(3.0 - (2.0*t));
}
```

- 0, ha $x < min$
- 1, ha $x > max$
- Hermite interpolált érték 0 és 1 között
  - $-2t^3 + 3t^2$

Cg fragmens program - reflektorfény

Intenzitás változás

```
float C5E9_dualConeSpotlight(float3 P,
                             Light  light)

  float3 V = normalize(P - light.position);
  float cosOuterCone = light.cosOuterCone;
  float cosInnerCone = light.cosInnerCone;
  float cosDirection = dot(V, light.direction);

  return smoothstep(cosOuterCone,
                    cosInnerCone,
                    cosDirection);
```

Cg fragmens program - reflektorfény

Intenzitás változás

```
float C5E9_dualConeSpotlight(float3 P,
                             Light  light)

  float3 V = normalize(P - light.position);
  float cosOuterCone = light.cosOuterCone;
  float cosInnerCone = light.cosInnerCone;
  float cosDirection = dot(V, light.direction);

  return smoothstep(cosOuterCone,
                    cosInnerCone,
                    cosDirection);
```

- A felület mennyi fényt bocsát ki
  - Az anyag rendelkezik egy $\mathbf{m}_{emi}$ emisszív paraméterrel
- Globális ambiens fényforrás paraméter
  - Konstans háttérfényt közelít
  - Minden irányból körülveszi a tárgyakat
- Módosított megvilágítási egyenlet

$$\mathbf{i}_{tot} = \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \mathbf{m}_{emi} + c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec}))$$

Cg fragmens program részlet
Emisszív paraméter

```
struct Material
  float3 Ke;
  float3 Ka;
  float3 Kd;
  float3 Ks;
  float shininess;
;

...

float3 emissive = material.Ke;
```

Cg fragmens program részlet
Emisszív paraméter

```
struct Material
  float3 Ke;
  float3 Ka;
  float3 Kd;
  float3 Ks;
  float shininess;
;

...

float3 emissive = material.Ke;
```

- Tegyük fel, hogy n fényforrásunk van és mindegyiket k indexszel azonosítjuk

$$\mathbf{i}_{tot} = \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \mathbf{m}_{emi} + \sum_{k=1}^{n} c^k_{spot}(\mathbf{i}^k_{amb} + d^k(\mathbf{i}^k_{diff} + \mathbf{i}^k_{spec}))$$

  - A fényforrás intenzitás összege 1-nél nagyobb is lehet
    - Az eredmény megvilágítási színt $[0, 1]$ intervallumra korlátozzuk le
  - Túlcsorduló szín skálázása a legnagyobb komponenssel
  - A túlcsordulások gyakran a geometriai részletességet csökkentik

Cg fragmens program részlet
Több fényforrás

```
    float3 diffuseLight;
    float3 specularLight;

    float3 ambientSum  = 0;
    float3 diffuseSum  = 0;
    float3 specularSum = 0;

// Az ambiens, diffúz és spekuláris komponensekre való
     számítások
    for (int i = 0; i < 2; i++) {
      C5E5_computeLighting(lights[i], position.xyz, normal,
                           eyePosition, material.shininess,
                           diffuseLight, specularLight);
      ambientSum += ambientLight;
      diffuseSum += diffuseLight;
      specularSum += specularLight;
    }

    // Anyagi tulajdonságok figyelembevétele
    float3 amibient = material.Ka * ambientSum;
    float3 diffuse = material.Kd * diffuseSum;
    float3 specular = material.Ks * specularSum;
```

Átlátszóság

Átlátszóság

- Megvalósításához szükség van az átlátszó tárgy színének és a mögötte lévő objektumok színének a keverésére
- Egy RGB szín és egy Z-puffer mélység van hozzákötve mindegyik pixelhez a képernyőn való megjelenítésekor
- α komponens
  - Az az érték, amely leírja a tárgy átlátszóságának a fokát egy adott pixelben
  - $\alpha = 1$ azt jelenti, hogy az objektum nem átlátszó és teljes egészében kitölti a pixel területet
  - $\alpha = 0$ pedig azt jelenti, hogy a pixel egyáltalán nem látszik

Átlátszóság

- Egy objektum átlátszóvá tételéhez a meglévő színtéren kell megjeleníteni egynél kisebb alfa értékkel

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + (1 - \alpha_s)\mathbf{c}_d \qquad \text{[over operátor]}$$

- $\mathbf{c}_s$ Az átlátszó objektum színe (forrás)
- α_s A tárgy alfa értéke
- $\mathbf{c}_d$ A keveredés előtti (a színpufferben lévő, cél) pixel szín érték
- $\mathbf{c}_o$ Az eredmény szín
  - Az átlátszó objektumot a meglévő színtér elé (over) helyezzük

Átlátszóság

- Helyes megjelenítéséhez általában szükségünk van rendezésre
  - Először a nem átlátszó tárgyakat kell renderelni
  - Aztán az átlátszó objektumokat kell hátulról előre haladva összekeverni a háttérben lévő alakzatok pixel értékeivel
- Tetszőleges sorrendben való összekeverés esetén súlyos artifaktumokat kaphatunk
  - A művelet sorrendfüggő vagyis feltételezi, hogy a háttérben lévő tárgyak már a színpufferben vannak
  - Speciális esetben, amikor két átlátszó tárgy van megjelenítve és mind a kettő alfa értéke 0.5, akkor a keveredésnél nem számít a sorrend

Átlátszóság

- Amennyiben a rendezés nem lehetséges vagy csak részben lett végrehajtva
  - Legjobb a Z-puffer használata
    - A z-mélység írását kikapcsolva az átlátszó objektum esetén
    - Az összes átlátszó objektum legalább meg fog jelenni
- Más technikák
  - Hátsó oldalak eldobásának kikapcsolása
  - Az átlátszó poligonok kétszeri renderelésével és a mélység tesztelést valamint a Z-puffer írásának az engedélyezését váltogatva

Átlátszóság

- Ki lehet számítani több menetben, két vagy több mélységpuffer használatával (mélység hámozás)
  - Első megjelenítési menetben a nem átlátszó felületek z-mélység értékeit helyezzük el az első Z-pufferben
  - Ezután az átlátszó objektumokat rendereljük le
    - A második menetben a mélység tesztet úgy módosítjuk, hogy elfogadjuk azt a felületet, amely az első pufferben lévő z-mélység értéknél közelebb van és az átlátszó objektumok közül pedig a legtávolabb van
    - A legtávolabbi átlátszó objektum bekerül a színpufferbe a mélység értéke pedig a második Z-pufferbe
    - Ezt a puffert aztán arra használjuk, hogy a következő legközelebbi átlátszó felületet határozzuk meg a következő menetben és így tovább

Köd

Köd

- A ködöt több céllal is lehet használni
  - A külső tér realisztikusabb megjelenítés szintjének a növelése
  - Mivel a köd hatása a nézőponttól távolodva növekszik, ezért ez segít meghatározni, hogy milyen távol találhatóak az objektumok
  - Ha megfelelően használjuk, akkor ez segít a távoli vágósík hatásának az elrejtésében
  - A köd gyakran hardveresen van megvalósítva, így egy elhanyagolható plusz költséggel lehet azt használni.

- $\mathbf{c}_p$ végső pixel szín értékének meghatározása

$$\mathbf{c}_p = f\mathbf{c}_s + (1 - f)\mathbf{c}_f$$

$\mathbf{c}_f$ A köd színe
$f \in \lfloor 0, 1 \rfloor$ A köd együtthatója
$\mathbf{c}_s$ Az árnyalt oldal színe

- Ahogy f értéke csökken, a köd hatása növekszik
- Különböző egyenleteket használhatunk a f megadására

- Egy köd konstans
  - Lineárisan csökken a nézőponttól távolodva
- Hol kezdődik és hol végződik a köd a néző z-tengelye mentén?

$$f = \frac{z_{end} - z_p}{z_{end} - z_{start}}$$

  - z_p az a z érték, ahol a köd hatását kell meghatározni

- f ködegyüttható

$$f = e^{-d_f z_p},$$
$$f = e^{(-d_f z_p)^2}$$

d_f A köd sűrűségét vezérli

- A kapott értéket a $[0, 1]$ intervallumra csonkoljuk és a köd egyenletét használjuk a végső érték kiszámításához
  - $\mathbf{c}_p = f\mathbf{c}_s + (1 - f)\mathbf{c}_f$

- Néha táblázatokat használnak a ködfüggvény hardveres megvalósítása esetén
  - Minden mélységre egy f ködegyütthatót előre kiszámítanak és eltárolnak.
  - Kiolvassák a táblázatból (vagy lineáris interpolációval határozzák meg két szomszédos tábla elemből)
    - Bármilyen értéket el lehet helyezni a köd táblázatban, nem csak az iménti egyenletekben megadottakat

Köd

- A ködfüggvényeket alkalmazni lehet vertex vagy pixel szinten

  Vertex-szintű A köd hatása a megvilágítási egyenlet részeként lesz kiszámítva és a kiszámított szín értéket interpolálja a poligonon keresztül Gouraud árnyalást használva

  Pixel-szintű A pixelenként tárolt mélység értéket használva számítjuk ki

- A pixel-szintű köd jobb eredményt ad

Összefoglalás

Témakörök

- Megvilágítás
  - Fényforrások és anyagi tulajdonságok
- Árnyalás
  - Megvilágítási számítások
- Megvilágítási modell
  - Cg példaprogramok
- Átlátszóság
- Köd

Textúrázás

Általánosítás textúrázás

Textúrázás

- Olyan eljárás, amely egy felület megjelenését módosítja egy bizonyos kép, függvény vagy adat segítségével
- A geometriai hiányosságoktól eltekintve eltérhet a valóságtól
- Két példa
  - Csillogó textúrázott felület
    - Világos spekuláris részt tartalmazó textúrakép
    - Színes textúrakép
  - Egyenetlen felület
    - Bump mapping
    - Felületi normálvektorok tárolása textúraképben
    - Normálvektorok megváltoztatása

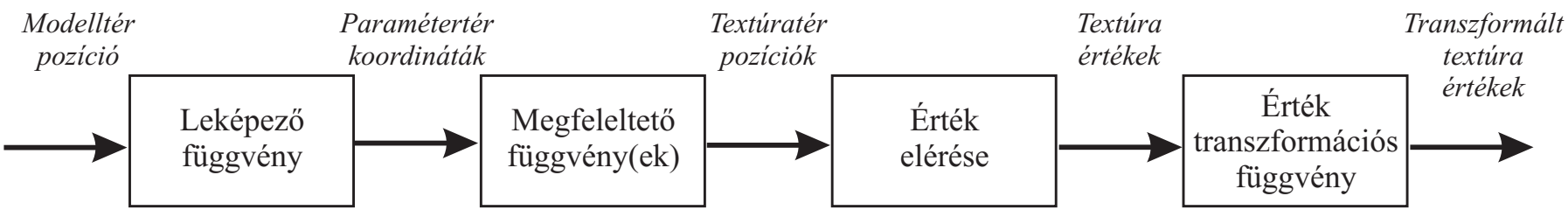
Általánosított textúrázás

- Hatékony technika a felületi tulajdonságok modellezésére
- Egyik megközelítési módja
  - Egy színértéket meghatározása egy poligon vertexe esetén
  - A színt a megvilágítási paraméterek valamint az anyagi tulajdonságok és a nézőpont helyének a figyelembe vételével számoljuk ki
  - Az átlátszóság és köd szintén befolyásolhatja ezt az értékét
  - A textúrázás módosítja a megvilágítási egyenletben kiszámított értékeket

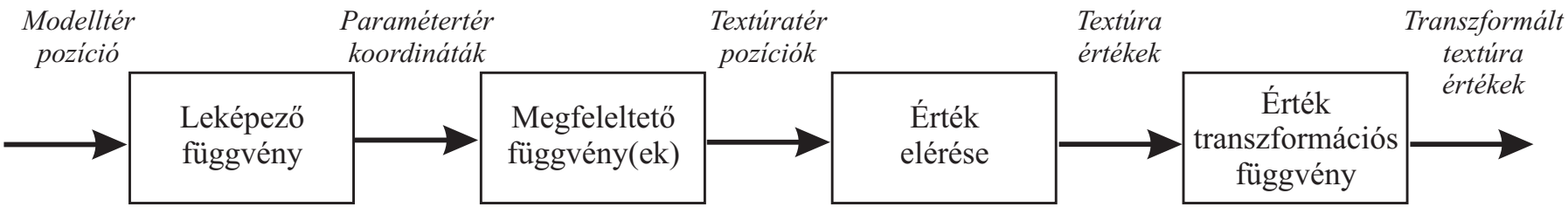
Általánosított textúrázás

- Alapesetben az adott színértékeket a felületi pozíciók alapján módosítjuk
  - A megvilágítási egyenletben a világos sepkuláris textúrakép a fényességi értéket módosítja
  - A felületi egyenetlenséget tartalmazó textúrakép esetén pedig a normálvektorok irányát változtatjuk

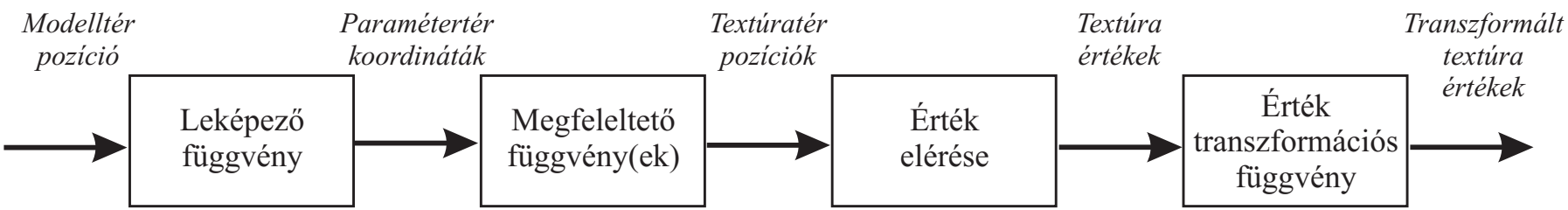
1. Felületi pontokhoz tartozó textúra értéket meghatározása
   - Egy térbeli ponthoz kell megkeresnünk az ennek megfelelő textúratérbeli pozíciót
   - Leggyakrabban a modellhez van rögzítve
   - Lehet a világtéren is



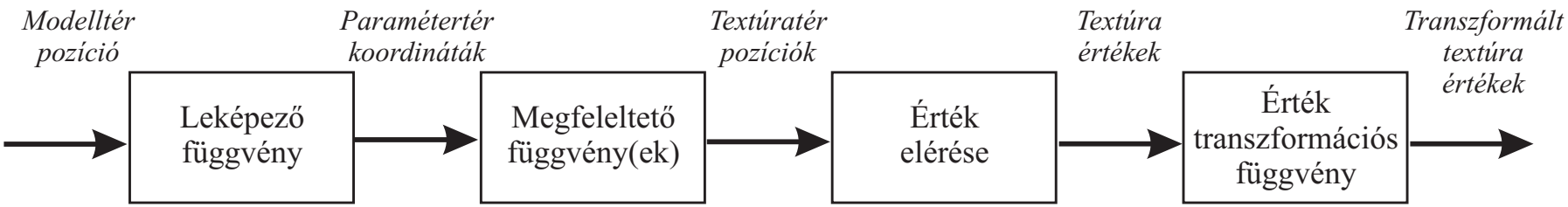
2. Leképező függvény alkalmazása
   - Megkapjuk paramétertér értékeket
   - A textúrakép eléréshez használjuk



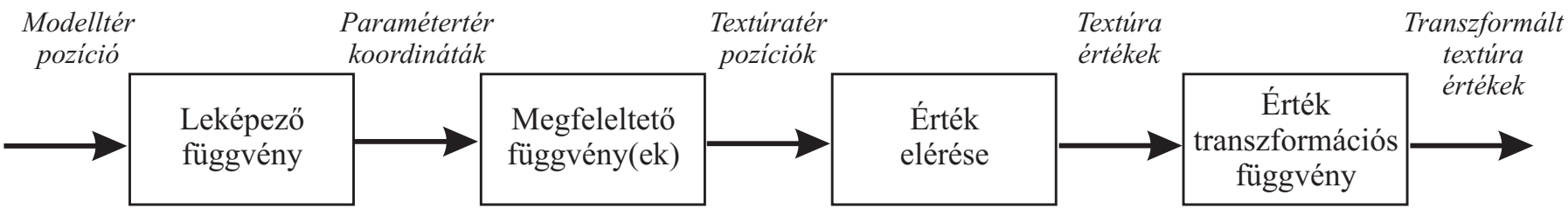
3. Megfeleltető függvény alkalmazása
   - A paramétertér értékeinek a textúraképtérbe való transzformálása
   - A textúraképtérbeli értékeket használjuk fel a textúrakép értékeinek eléréséhez
   - Lehetnek tömb indexek a textúrakép pixeleinek a kinyeréséhez



4. Textúra értékek kinyerése



5. Transzformációs függvény alkalmazása
   - Kinyert értékek transzformálása



- Szükségünk van egy textúraképre
  - Színek
  - Árnyalás során használt optikai/felületi jellemzők
- A textúraképek általában kétdimenziósak
  - Textúratérbeli pozíció
    - (u, v) $(u, v \in [0, 1])$ textúra-koordináták
    - Néha (u, v, w) három elemű vektorként adjuk meg a w mélységgel együtt
    - Más rendszerekben (s, t, q) koordinátákat használnak
  - Textúra-leképezés során ezeket a textúra-koordinátákat rendeljük hozzá a felületi pozíciókhoz

- Textúra-leképezés során megadjuk
  - Egy pontban mivel kell lecserélni a színértéket
  - Az adott pozícióban milyen értékkel kell módosítani az árnyalást
- A felületet alkotó poligonok szögpontjaiként határozzuk meg
  - Az adott szögponthoz hozzárendeljük a megfelelő textúra-koordinátákat
  - A modellt alkotó primitívekre illesztjük a textúraképből kivágott területeket

- A modellező szakaszban manuálisan adjuk meg
- A leképezés eredményét a vertexekben tároljuk
- OpenGL
  - `glTexGen`

    Gömbi leképezés A pontokat egy pont körül elhelyezkedő képzeletbeli gömbre vetíti

    Henger leképezés Az u textúra-koordinátákat a gömbi leképezésekhez hasonlóan számítja ki. A v textúra-koordinátákat a henger tengelye mentén, mint távolságot számítja ki

    Sík leképezés Egy irány mentén képez le és a textúrát a teljes felületre alkalmazza

- Más bemenő adatokat is lehet használni egy leképező függvényben
  - Felületi normálvektor használata sík kiválasztására cube map leképezéskor
  - Bizonyos leképező függvények egyáltalán nem is hasonlítanak a vetítésre
    - A parametrikus görbe felületek definíciójuk alapján rendelkeznek egy (u, v) értékhalmazzal
  - Textúra-koordinátákat például a nézőpont iránya vagy a felület hőmérséklete alapján is elő lehet állítani

- Nem-interaktív rendereléskor gyakran hívják meg ezeket a leképező függvényeket a renderelési eljárás részeként
  - Elegendő lehet a teljes modell számára
  - Gyakran a felhasználónak kell a modellt szétdarabolni és a leképező függvényeket alkalmazni

- A paramétertérben lévő értékeket konvertálják textúratérbeli pozíciókra
- Megadható egy transzformációs mátrixszal
  - Eltolás
  - Forgatás
  - Skálázás
  - Nyírás
  - A textúra leképezése/vetítése egy adott felületre
- Milyen módon alkalmazzuk a textúrát?
  - Az (u, v)-k a $[0, 1)$ intervallumban vannak alap esetben
  - Mi történik a $[0, 1)$ intervallumon kívül?
    - A megfeleltető függvény meghatározhatja ezt a viselkedést

- Megfeleltető függvények az OpenGL-ben

  repeat A kép ismétli önmagát a felületen. A paraméter egész részét eldobjuk.

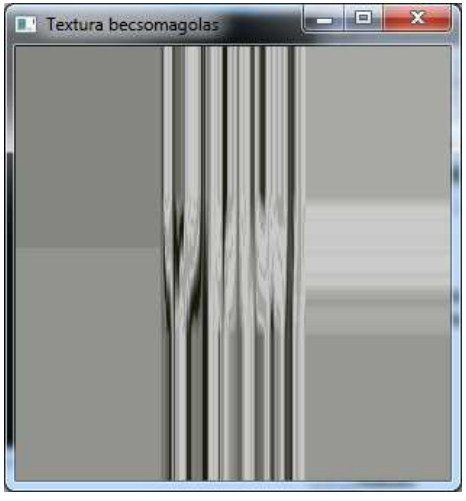
  mirror A kép szintén ismétli önmagát a felületen, de minden egyes ismétléskor tükrözve van.

  clamp to edge A $[0, 1)$ intervallumon kívüli értékek esetén a textúrakép első és utolsó sorának vagy oszlopának az ismétlését eredményezi.

  clamp to border A $[0, 1)$ paraméter értékeken kívül a textúra betöltéskor megadott határ színét használja hasonlóan a **clamp to edge** esethez. A határ nem tartozik textúrához. Határszín beállítása `glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor)`

Repeat

Mirror



Clamp to edge

Clamp to border

- A paramétertér értékeit használják a textúratér pozícióinak az előállítására
- Implicit és a kép méretéből származtatott megfeleltető függvény
  - u és v értékei a $[0, 1)$ intervallumon belül vannak alapesetben
  - Az intervallumban lévő paraméterértékeket megszorozva az adott kép méretével a pixelek pozícióját kapjuk meg a képtérben
  - (u, v) értékek nem függenek a kép méretétől

- Leggyakrabban kétdimenziós képeket használunk
- A textúraképek direkt kiterjesztése a háromdimenziós képadat
  - (u, v, w) koordinátákon keresztül érhetünk el
  - Más rendszerekben (s, t, q) textúra-koordinátákat használnak
  - A modell csúcspontjaihoz közvetlenül rendelhetjük a pozíciókat textúra-koordinátaként
  - Elkerülhetjük a kétdimenziós textúrázáskor előforduló torzításokat
  - Olyan mint, ha az anyagot reprezentáló háromdimenziós textúrából lenne kifaragva a modell
    - Például gránit tömbből kifaragott szobor

- A felületi tulajdonságok kicserélésére vagy módosítására használhatunk fel
  - RGB hármas
  - Szürkeárnyalatos érték
  - RGBα
    - α érték alapesetben a szín átlátszóságát fejezi ki
  - Felületi normálvektorok
- Textúra értékek kinyerése után közvetlenül vagy transzformálva használhatjuk fel

- A legtöbb valósidejű rendszer Gouraud árnyalást használ
  - Csak bizonyos értékek vannak interpolálva a felületen
  - Csak ezeket az értékeket tudja a textúra módosítani
- Alapesetben a megvilágítási egyenlet RGB eredményét módosítjuk
  - Vertexenkénti kiértékelés után interpoláljuk a színt

- Felület színértékének módosítása
  - Egyesítő függvények
  - Textúra keveredés operátorok

replace — Az eredeti felületi színt lecseréli a textúra színére. Megjegyezzük, hogy ez eltávolítja az árnyalás során meghatározott értéket.

decal — Hasonló a **replace** egyesítő függvényhez, de amikor a textúrakép tartalmaz egy α értéket, akkor a textúraszín az eredeti felületi színnel keveredik, de az eredeti α érték nem módosul.

modulate — Megszorozza a felületi színértékét a textúra színével. Az árnyékolt felület a textúra színével van módosítva, amely egy árnyékolt textúrázott felületet ad.

- A textúrakép (textúra) a poligon felületére van illesztve
- A kép felületre való alkalmazása
- Textúra mérete gyakran korlátozva van $2^m \times 2^n$
  - Néha $2^m \times 2^m$
- Példa
  - Textúra
  - Négyzeten akarjuk használni
- Egyező méret esetén ugyanúgy fog kinézni, mint az eredeti kép
- Mi történik, ha
  - a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúra?
    - Nagyítás
  - a leképezett négyzet csak a textúra pixeleinek a töredékét fedi le?
    - Kicsinyítés

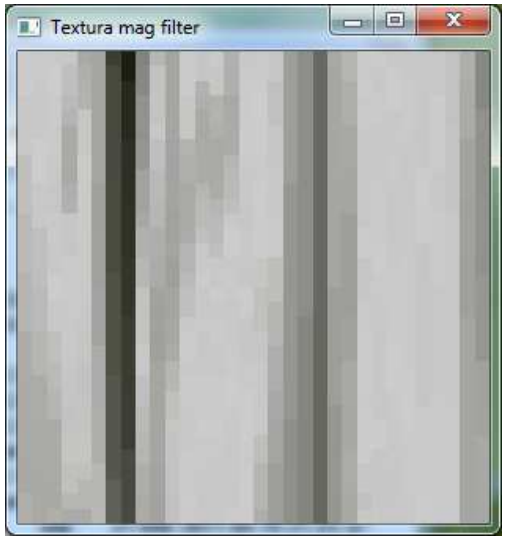
- A textúrakép (textúra) a poligon felületére van illesztve
- A kép felületre való alkalmazása
- Textúra mérete gyakran korlátozva van $2^m \times 2^n$
  - Néha $2^m \times 2^m$
- Példa
  - Textúra
  - Négyzeten akarjuk használni
- Egyező méret esetén ugyanúgy fog kinézni, mint az eredeti kép
- Mi történik, ha
  - a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúra?
    - Nagyítás
  - a leképezett négyzet csak a textúra pixeleinek a töredékét fedi le?
    - Kicsinyítés

- A textúrakép (textúra) a poligon felületére van illesztve
- A kép felületre való alkalmazása
- Textúra mérete gyakran korlátozva van $2^m \times 2^n$
  - Néha $2^m \times 2^m$
- Példa
  - Textúra
  - Négyzeten akarjuk használni
- Egyező méret esetén ugyanúgy fog kinézni, mint az eredeti kép
- Mi történik, ha
  - a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúra?
    - Nagyítás
  - a leképezett négyzet csak a textúra pixeleinek a töredékét fedi le?
    - Kicsinyítés

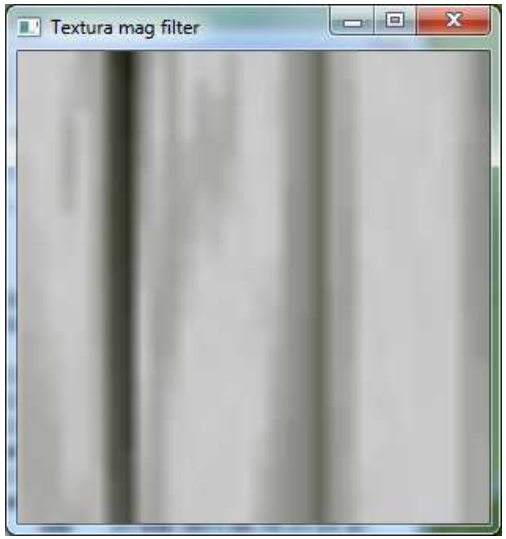
- A textúrakép (textúra) a poligon felületére van illesztve
- A kép felületre való alkalmazása
- Textúra mérete gyakran korlátozva van $2^m \times 2^n$
  - Néha $2^m \times 2^m$
- Példa
  - Textúra
  - Négyzeten akarjuk használni
- Egyező méret esetén ugyanúgy fog kinézni, mint az eredeti kép
- Mi történik, ha
  - a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúra?
    - Nagyítás
  - a leképezett négyzet csak a textúra pixeleinek a töredékét fedi le?
    - Kicsinyítés

- A textúrakép (textúra) a poligon felületére van illesztve
- A kép felületre való alkalmazása
- Textúra mérete gyakran korlátozva van $2^m \times 2^n$
  - Néha $2^m \times 2^m$
- Példa
  - Textúra
  - Négyzeten akarjuk használni
- Egyező méret esetén ugyanúgy fog kinézni, mint az eredeti kép
- Mi történik, ha
  - a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúra?
    - Nagyítás
  - a leképezett négyzet csak a textúra pixeleinek a töredékét fedi le?
    - Kicsinyítés

- Nagyítás szűrési technika
  - Legközelebbi szomszéd
    - A texelek különállóan láthatóvá válnak
  - Bilineáris interpoláció

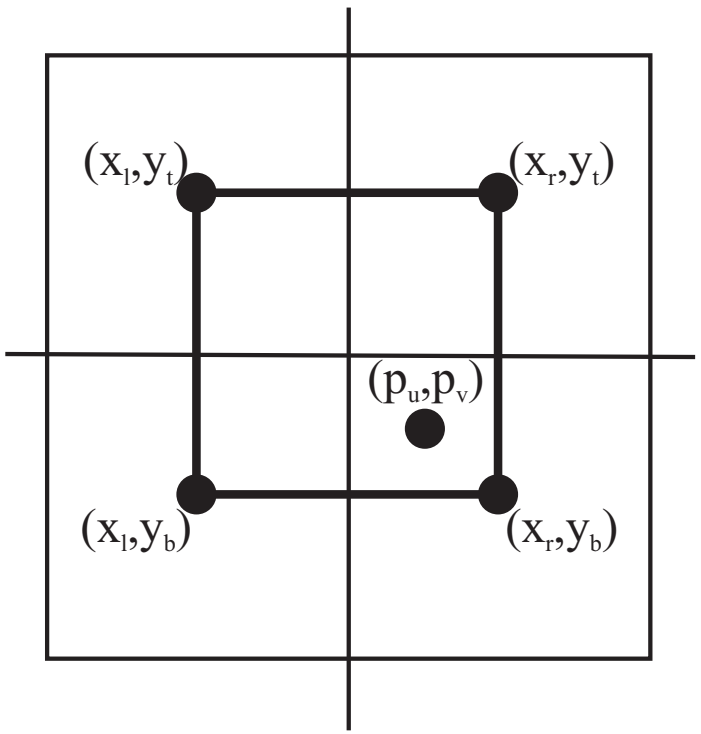


Legközelebbi szomszéd



Bilineáris interpoláció

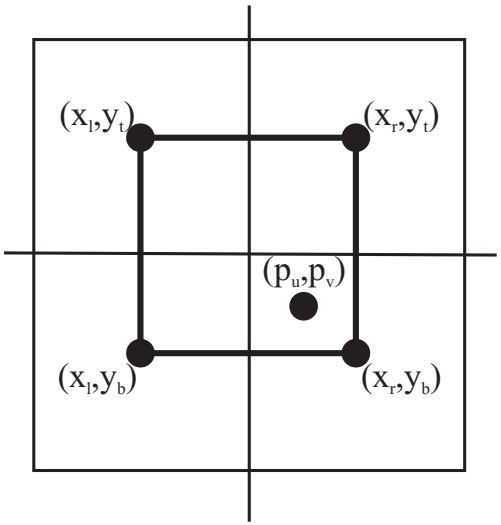
- A bilineáris interpoláció mindegyik pixel esetén négy szomszédos texelt vesz
  - Két dimenzióban lineárisan interpolálja azokat
  - Az eredmény homályosabb
  - A legközelebbi szomszéd módszernél tapasztalt élek recésségének nagy része eltűnik



- A bilineáris interpolált $\mathbf{b}$ szín a (p_u, p_v) pozícióban

$$\mathbf{b}(p_u, p_v) = (1-u')(1-v')\mathbf{t}(x_l, y_b) + u'(1-v')\mathbf{t}(x_r, y_b) + (1-u')v'\mathbf{t}(x_l, y_t) + u'v'\mathbf{t}(x_r, y_t)$$

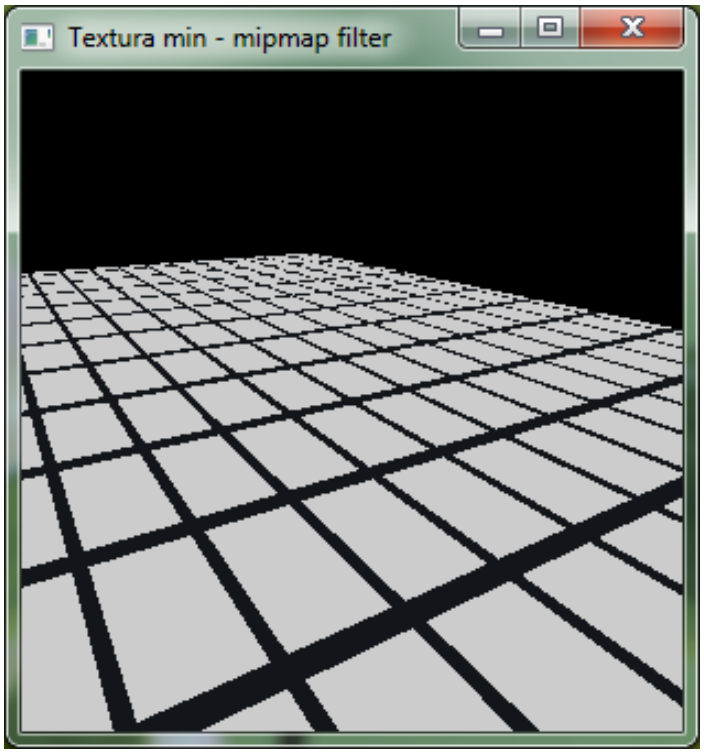
  - $\mathbf{t}(x, y)$ a texel színét jelöli a textúrában
  - x és y egészek



- Szűrő kiválasztása függ az elérendő eredménytől
- A bilineáris interpoláció használata az esetek többségében jó eredményt ad

- Textúrát kicsinyítéskor
  - Több texel fedhet be egy pixelt
- Összegezni kell a textúra értékek hatását, amelyeket az adott pixel lefed
- Nehéz pontosan meghatározni a textúra értékek átlagát egy bizonyos pixel pozícióban
  - Nem tudjuk, hogy hány textúra értéket kell az átlagszámításkor figyelembe venni

- Legközelebbi szomszéd
  - Hasonló a nagyításnál használt módszerhez
  - Komoly aliasing probléma
  - Éles szög esetén artifaktumok/műtermékek jelenek meg
  - Csak egyetlen egy texel befolyásolja a pixel értékét adott pozícióban
  - Időbeli aliasing
    - Az artifaktumok sokkal jobban észrevehetőek, amikor a nézőponthoz viszonyítva a felület mozog



- Bilineáris interpolációt szintén használhatunk kicsinyítéskor
  - Négy texel értékét átlagoljuk ebben az esetben is
  - Ha egy pixel értékére több, mint négy textúra érték van hatással, akkor a szűrő hibázni fog

- Textúra jelfrekvenciája attól függ, hogy milyen közel helyezkednek el a texelek a képernyőn
- A textúra jelfrekvenciája nem lehet nagyobb, mint a mintavételezési frekvencia fele
  - Vagy a mintavételezési frekvenciát növeljük
  - Vagy a textúra frekvenciáját kell csökkenteni
- Antialiasing módszerekkel növelhetjük a mintavételezési frekvenciát
  - Ezek a módszerek csak korlátozott mértékben tudják növelni a mintavételezés frekvenciáját

- Multum in parvo
  - Sok dolog kis helyen
- A legtöbb mipmap technika a legegyszerűbb grafikus hardvereken is meg van valósítva
- Az eredeti textúra mellett a textúra kicsinyített változatait is felhasználjuk

- Kicsinyített textúrák előállítása
  - Az eredeti textúrából kiindulva, mindig az előző textúra méretét csökkentjük a negyedére
    - Minden új texelt négy szomszédos texel átlagaként számítunk ki
  - A csökkentést addig hajtjuk végre addig, amíg a textúrának egyik vagy mind a két dimenziója egy texellel lesz egyenlő
- Az így kialakult textúra felbontási szintek mentén egy harmadik tengelyt definiálunk
  - d-vel jelöljük

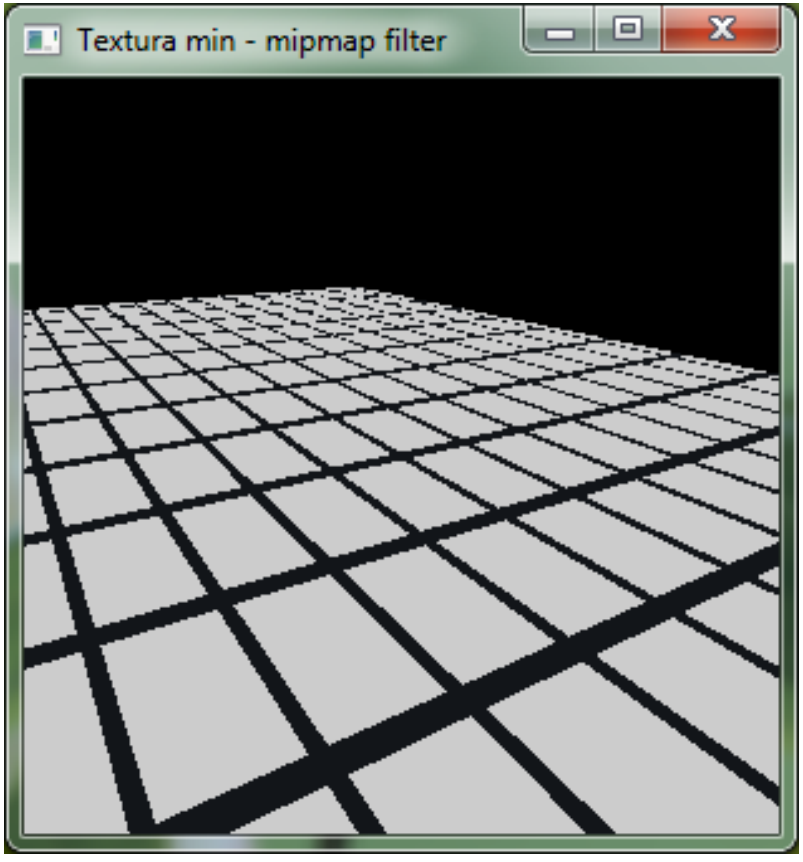
- Jó minőségű mipmap textúra létrehozásához szükség van jó szűrésre és gamma korrekcióra
- Gamma korrekció nélkül az átlagos fényessége a mipmap szinteknek el fog térni az eredeti textúra fényességétől
  - Az objektumtól távolodva az objektum sötétebben fog megjelenni és a kontrasztja és a részletessége is megváltozhat

- Meghatározzuk, hogy hol mintavételezzünk a mipmap piramis tengely mentén
  - A cél az, hogy nagyjából meghatározzuk azt, hogy a pixelre mekkora textúra terület van hatással
- Azt szeretnénk elérni, hogy egy pixel-texel arány legalább 1 : 1 legyen

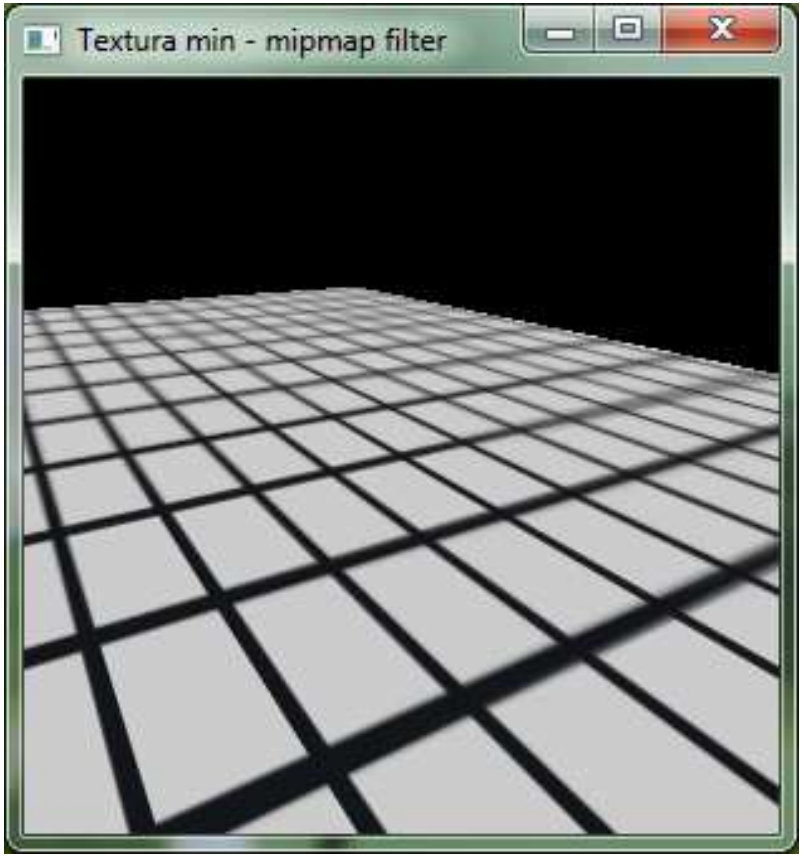
- d koordináta kiszámítása
  - A pixel által formált négyszög hosszabb élét használja a pixel kiterjedésének a megközelítésére
  - A legnagyobb abszolút értékű ($\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$) differenciát használja mértékként
- Mindegyik differencia azt határozza meg, hogy mekkora a változás nagysága a textúra-koordinátákban a képernyő adott tengelye mentén
  - Például a $\frac{\partial u}{\partial x}$ az u érték változásának a nagyságát jelenti egy pixelre nézve az x tengely mentén

- Az (u, v, d) hármast használjuk a mipmap textúra elérésére
- Trilineáris interpoláció
  - A d értéke egy valós szám
  - d nem egész, ezért a d textúra szint felett és a szint alatt mintavételezünk
  - Az így kapott (u, v) pozíciót használjuk egy-egy bilineárisan interpolált minta kinyerésére a két szomszédos textúra szintből
  - Az eredményt ezután lineáris interpolációval kapjuk meg attól függően, hogy d milyen távolságra van a két szomszédos textúra szinttől
  - Pixelenként hajtjuk végre
- Két mipmap textúra elérése
  - Végrehajtása kétszer olyan drága lehet néhány hardveren

Textúra kicsinyítés



Legközelebbi szomszéd



Mipmap

Textúrázással kapcsolatos kiegészítő OpenGL függvények

Színpuffer használata

- Egy- vagy kétdimenziós textúrákat meg lehet adni a színpufferből származó adatokkal
- Kép beolvasása színpufferből és annak új textúraként való felhasználása
- Forrás szín puffer kiválasztása a pixelek számára
  - `void glReadBuffer(GLenum mode);`

    `mode` Például `GL_FRONT` és `GL_BACK`

Színpuffer használata

- Egy- vagy két-dimenziós textúrát definiálása az aktuális `GL_READ_BUFFER`-ből

```
void glCopyTexImage1D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y,
GLsizei width, GLint border);

void glCopyTexImage2D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y,
GLsizei width, GLsizei height, GLint border);
```

Színpuffer használata

```
void glCopyTexImage2D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y,
GLsizei width, GLsizei height, GLint border);
```

`target` Meghatározza a cél textúrát

`level` A részletesség szintjének beállítása

`internalformat` A textúra belső formátumának a beállítása

`x, y` A másolandó téglalap bal alsó koordinátái

`width` A textúra kép szélessége

`height` A textúra kép magassága

`border` A keret szélessége

Színpuffer használata

- `glCopyTexImage3D` függvény nem létezik
  - Egy kétdimenziós színpufferből nem lehet térfogati adatokat kinyerni
- Képi adat kinyerése a színpufferből
  - `data` visszatérő pixel adat
  - Akár mélység puffer adatokat is ki lehet nyerni

```
void glReadPixels( GLint   x,
  GLint y,
  GLsizei width,
  GLsizei height,
  GLenum format,
  GLenum type,
  GLvoid *data);
```

Textúrák frissítése

- Textúrák többszöri betöltése a valós-időben teljesítmény problémákat okozhat
- Részben vagy teljes egészében le lehet cserélni
  - Sokkal gyorsabban lehet végrehajtani, mint egy új textúrát megadni
  - `glTexSubImage`

```
void glTexSubImage1D(GLenum target, GLint level,
GLint xOffset,GLsizei width,
GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage2D(GLenum target, GLint level,
GLint xOffset, GLint yOffset,
GLsizei width, GLsizei height,
GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage3D(GLenum target, GLint level,
GLint xOffset, GLint yOffset, GLint zOffset,
GLsizei width, GLsizei height, GLsizei depth,
GLenum format, GLenum type, const GLvoid *data);
```

Textúrák frissítése

- A legtöbb paraméter megfelel a `glTexImage` függvény argumentumainak

  `dOffset` Texel eltolása meghatározása `d` irányba, amelyet a meglévő textúra képen cserélünk le a megadott textúra adattal

  `dSize` Beillesztendő textúra mérete `d` irányba

Textúrák frissítése

- Színpufferből való olvasás és egy textúra részének beszúrása vagy lecserélése

```
void glCopyTexSubImage1D(GLenum target, GLint level,
     GLint xoffset, GLint x, GLint y,
     GLsizei width);

void glCopyTexSubImage2D(GLenum target, GLint level,
     GLint xoffset, GLint yoffset,
     GLint x, GLint y,
     GLsizei width, GLsizei height);

void glCopyTexSubImage3D(GLenum target, GLint level,
     GLint xoffset, GLint yoffset, Glint zoffset,
     GLint x, GLint y,
     GLsizei width, GLsizei height);
```

Textúrák frissítése

- Az argumentumok jelentése megegyezik a `glCopyTexImage` és `glTexSubImage` függvények hasonló paramétereivel
- A `glCopyTexSubImage3D` függvény esetén a színpuffert felhasználhatjuk egy 3D-s textúra egy kétdimenziós szeletének lecserélésére is.

Textúra mátrix

- Textúra-koordinátákat transzformálhatjuk egy textúramátrixszal
  - Eltolás, skálázás, forgatás
- A `glMatrixMode(GL_TEXTURE);` utasítás meghívása után
  - Használhatóak a `glRotatef()`, `glScalef()` és `glTranslatef()` függvények
  - Speciális OpenGL mátrix műveletek
- A textúramátrix verem hasonlóan működik, mint a transzformációknál ismertetett
  - `glPushMatrix()`, `glPopMatrix()`
  - A verem mélysége maximálisan kettő lehet

Mipmap szintek automatikus előállítása

- Szükség van az eredeti textúrakép kicsinyített változataira
- GLU segéd-függvénykönyvtár
  - `gluScaleImage` függvény ismételt meghívásaival előállíthatóak a kicsinyített képek
  - Kényelmesebb módszer

```
int gluBuild1DMipmaps(GLenum target, GLint
    internalFormat,
    GLint width,GLenum format,
    GLenum type, const void *data);

int gluBuild2DMipmaps(GLenum target, GLint
    internalFormat,
    GLint width, GLint height,
    GLenum format, GLenum type, const void *data);

int gluBuild3DMipmaps(GLenum target, GLint
    internalFormat,
    GLint width, GLint height,
    GLint depth, GLenum format,
    GLenum type, const void *data);
```

Mipmap szintek automatikus előállítása

- A `glTexImage` használatával egyezik meg a használatuk
- Nem rendelkeznek
  - `level` paraméterrel, ami megadja a mipmap szinteket
  - Textúra határ kezeléssel
- Nem fogunk olyan minőséget kapni, mint egy professzionális képszerkesztő esetén

Mipmap szintek automatikus előállítása

- Használhatjuk az OpenGL hardveres gyorsítását a mipmap szintek előállításához
  - Amennyiben előre tudjuk azt, hogy az összes mipmap szintet be fogjuk tölteni
- `glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);`
  - Minden `glTexImage` vagy `glTexSubImage` függvény hívásakor automatikusan frissíti az alacsonyabb szintű mipmap szinteket is
- OpenGL 1.4-es vagy annál későbbi változata esetén használható

Részletesség szintjeinek befolyásolása

- Mipmapping engedélyezésekor az OpenGL egy formula alapján meghatározza, hogy melyik mipmap szintet kell kiválasztani
- Be lehet állítani azt az OpenGL-ben
  - A kiválasztási feltételt hátrébb (a nagyobb mipmap szintek felé)
  - A kiválasztási feltételt előrébb (a kisebb mipmap szintek felé) tolja

Részletesség szintjeinek befolyásolása

- Hatása
  - A teljesítmény javul a kisebb mipmap szintek használatakor
  - A textúrázott objektum „élessége” növekszik nagyobb mipmap szintek használatakor
    - A textúra feldolgozása kissé hosszabb ideig fog tartani
- Példa
  - Nagyobb részletességű szintek felé mozgatjuk el a részletességet
  - `glTexEnvf(GL_TEXTURE_FILTER_CONTROL, GL_TEXTURE_LOD_BIAS, -1.5);`

Textúrázással kapcsolatos Cg ismeretek

```
struct C3E2v_Output {
  float4 position : POSITION;
  float3 color    : COLOR;
  float2 texCoord : TEXCOORD0;
};

C3E2v_Output C3E2v_varying(float2 position : POSITION,
                           float3 color    : COLOR,
                           float2 texCoord : TEXCOORD0)
{
  C3E2v_Output OUT;

  OUT.position = float4(position,0,1);
  OUT.color    = color;
  OUT.texCoord = texCoord;

  return OUT;
}
```

- Vertexenkénti szín és textúra-koordináta megadás

`COLOR`	Vertex szín
`TEXCOORDO`	Nullás textúra-koordináta halmaz

Fragmens shader

```
struct C3E3f_Output {
  float4 color : COLOR;
};

C3E3f_Output C3E3f_texture(float2 texCoord : TEXCOORD0,
                           uniform sampler2D decal)
{
  C3E3f_Output OUT;
  OUT.color = tex2D(decal,texCoord);
  return OUT;
}
```

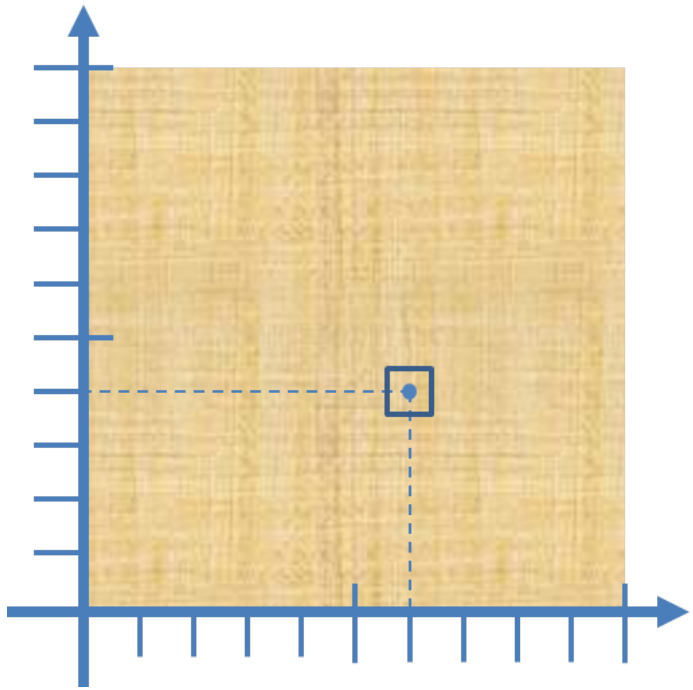
Textúra mintavételező

- A fragmens program megkapja az interpolált textúra-koordinátákat
  - Figyelmen kívül hagyja az interpolált színt
- Egy uniform bemenő paramétert is megkap
  - Külső objektum
  - Mintavételezni képes
    - Mint egy textúrát

Textúra mintavételező

Mintavételező típusa	Textúra típusa	Alkalmazások
`sampler1D`	Egy dimenziós textúra	1D függvények
`sampler2D`	Két dimenziós textúra	Matricák, normál térképek
`sampler3D`	Három dimenziós textúra	Térfogati adat
`samplerCUBE`	Cube map textúra	Környezeti térkép
`samplerRECT`	Nem kettő-hatványú, nem mipmap-elt két dimenziós textúra	Videó képek, ideiglenes pufferek

- A textúra koordináták meghatározzák azt, hogy milyen pozíción érjük el a textúra értékét
  - $[0-1]$ intervallumon belül
  - $[0-1]$ intervallumon kívül



- Decal (matrica) textúrához való hozzáférés interpolált textúra koordinátákkal
  - `OUT.color = tex2D(decal, texCoord);`
- Az eredmény a mintavételezett adat az adott pozícióban, az adott textúrában
- Gyakorlatban ez egy textúra elem kikeresése
- A mintavételezés és szűrés függ
  - Textúra típusa
  - Textúra paraméterek
- Textúra objektum tulajdonságainak beállításai

- Dimenzióknak meg kell egyeznie
  - `tex2D` `sampler2D`
- Az alap fragmens profile-ok korlátozzák a textúra mintavételezési rutinokat
- A fejlettebb fragmens profile-ok megengedik
  - A Cg programban kiszámított textúra koordináták használatát

- **Vertex program**
  - Vertexenkénti pozíció
  - Szín
  - Textúra koordináta halmaz
- **Fragmens program**
  - Figyelmen kívül hagyja az interpolált színt
  - A textúra képet mintavételezi az interpolált textúra koordinátákkal

Megoldható nem programozható 3D-s hardver nélkül is

Példa textúrázásra

Kettős látás hatás - vertex program

```
void C3E5v_twoTextures(
            float2 position : POSITION,
            float2 texCoord : TEXCOORD0,

            out float4 oPosition      : POSITION,
            out float2 leftTexCoord   : TEXCOORD0,
            out float2 rightTexCoord  : TEXCOORD1,

            uniform float2 leftSeparation,
            uniform float2 rightSeparation)
{
  oPosition     = float4(position, 0, 1);
  leftTexCoord  = texCoord + leftSeparation;
  rightTexCoord = texCoord + rightSeparation;
}
```

Példa textúrázásra

Kettős látás hatás - fragmens program

```
void C3E6f_twoTextures(
                float2 leftTexCoord  : TEXCOORD0,
                float2 rightTexCoord : TEXCOORD1,

                out float4 color : COLOR,

                uniform sampler2D decal)
{
  float4 leftColor  = tex2D(decal, leftTexCoord);
  float4 rightColor = tex2D(decal, rightTexCoord);
  color = lerp(leftColor, rightColor, 0.5);
}
```

- Lineáris interpoláció
  - `VECTOR lerp(VECTOR a, VECTOR b, TYPE weight)`
- Nem fog lefordulni alap fragmens profile-okkal
  - Két textúra koordináta halmazt használ a 0-s textúra egység eléréséhez

Példa textúrázásra

Kettős látás hatás - fragmens program

```
void C3E6f_twoTextures(
                float2 leftTexCoord  : TEXCOORD0,
                float2 rightTexCoord : TEXCOORD1,

                out float4 color : COLOR,

                uniform sampler2D decal0
                uniform sampler2D decal1)
{
  float4 leftColor  = tex2D(decal0, leftTexCoord);
  float4 rightColor = tex2D(decal1, rightTexCoord);
  color = lerp(leftColor, rightColor, 0.5);
}
```

A textúrát kétszer kell „binding”-olni !!!

Összefoglalás

Témakörök

- Általánosított textúra csővezeték
  - Nagyítás
  - Kicsinyítés
- Kiegészítő OpenGL függvények
- Cg ismeretek

Ütközés-detektálás

Ütközés-detektálás

- Alapvető alkotórésze sok számítógépes grafikai és virtuális valóság alkalmazásnak
- Ütközés kezelés
  - Ütközés-detektálás
    - Eredménye egy logikai érték
    - Kettő vagy több tárgy ütközött-e vagy sem
  - Ütközés-meghatározás
    - Megtalálja az aktuális objektumok metszéspontjait
  - Válasz az ütközésre
    - Milyen műveletet kell végrehajtani két tárgy ütközésekor?

Ütközés-detektálás

- Egy színtér több száz objektumot tartalmazhat
- Ha a színtér n mozgó és m statikus objektumot tartalmaz
- Naiv megközelítés esetén végrehajtandó objektum tesztek száma minden egyes képkocka esetén
  - $nm + \frac{n}{2}$
  - (statikus és dinamikus objektumok) + (dinamikus objektumok)
- m és n növekedésével az elvégzendő tesztek száma nagy mértékben megnő
  - Az algoritmusok függnek az aktuális ütközési forgatókönyvtől
  - Nincs olyan algoritmus, amely minden esetben a legjobban viselkedik

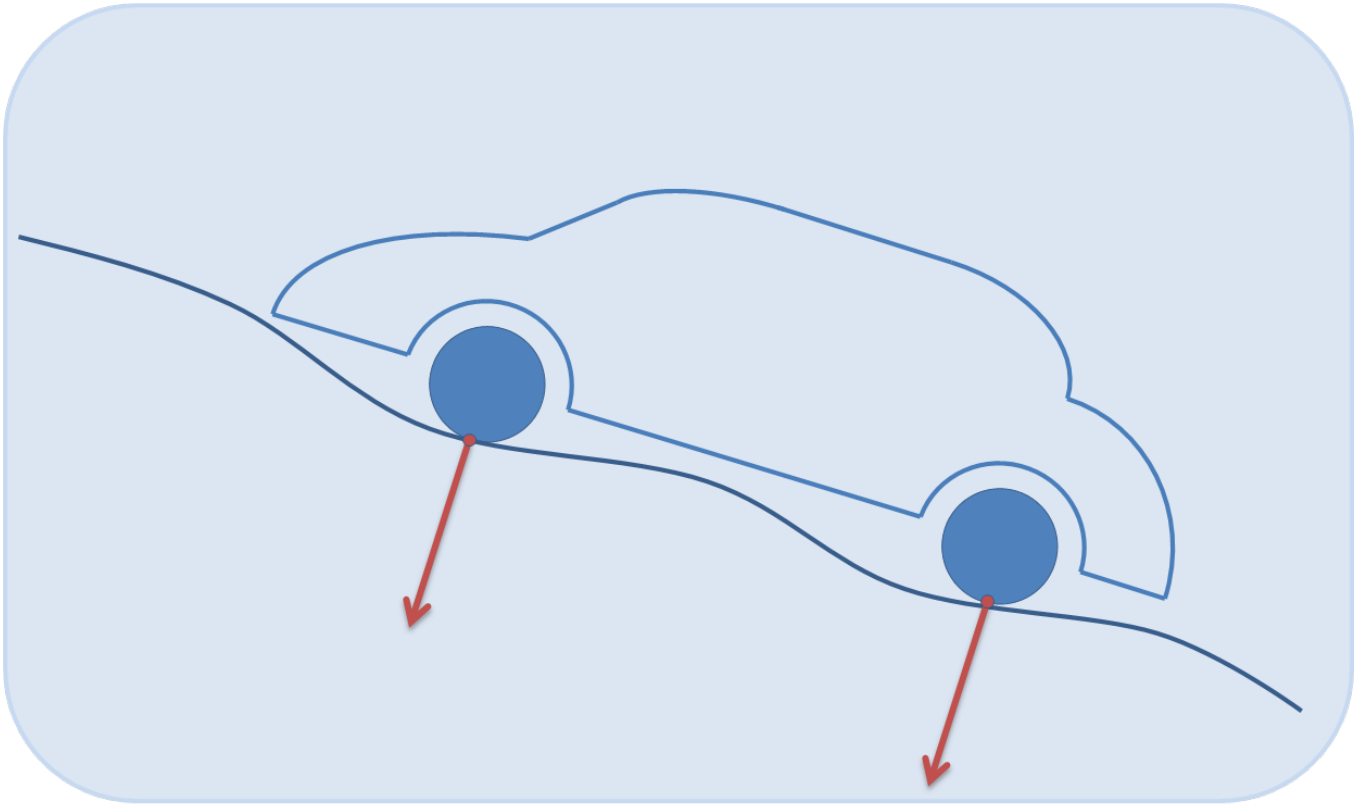
Ütközés-detektálás sugarakkal

Ütközés-detektálás sugarakkal

- Bizonyos feltételek esetén jól működik
- Példa
  - Egy gépkocsi halad felfelé egy emelkedőn
  - Információ az útról
    - Az utat felépítő primitívek
  - Kocsi kerekeit az úton tartjuk animáció közben
- A kerekek és az utat alkotó összes primitív esetén elvégezzük az ütközés-detektálást
- A mozgó objektumot közelíthetjük egy sugárhalmazzal

Ütközés-detektálás sugarakkal

- Egy-egy sugarat helyezünk el a négy keréknél
- A közelítés addig jó, amíg feltesszük, hogy csak a négy kerék van kapcsolatban az úttal



Ütközés-detektálás sugarakkal

- Tegyük fel hogy
  - A gépkocsi egy síkon áll a kezdetekben
  - A sugarakat úgy helyezzük el, hogy a kezdőpontjaikat a kerekek és a környezet érintkezési pontjában legyenek
- A kerekeknél elhelyezett sugarak metszését teszteljük a környezettel
  - Ha a sugár origója és a környezet közötti távolság nulla
    - A kerék pontosan a talajon van
  - Ha a távolság nagyobb, mint nulla
    - A kerék nem érintkezik a környezettel
  - Negatív érték esetén
    - A kerék behatol a környezetbe

Ütközés-detektálás sugarakkal

- Ütközés-válasz kiszámítására használható a távolság
  - Negatív távolság a gépkocsit felfele mozgatná
  - A pozitív távolság a kocsit lefele mozgatná
    - Kivéve, ha a kocsi nem a levegőben repül egy rövid ideig
- Amire szükségünk van az a sugár útjában lévő legközelebbi objektum
  - A negatív sugárparaméterhez tartozó metszéseket is vizsgálnunk kell
  - Annak érdekében, hogy két irányba kelljen keresni
    - A tesztelő sugár origóját mozgatjuk vissza addig, amíg kívül nem esik az út geometriájának határoló térfogatán
    - Ez csak azt jelenti, hogy a 0 távolságban kezdődő sugár helyett, negatív távolságnál kezdődik a sugárnyaláb

BSP fák

BSP fák

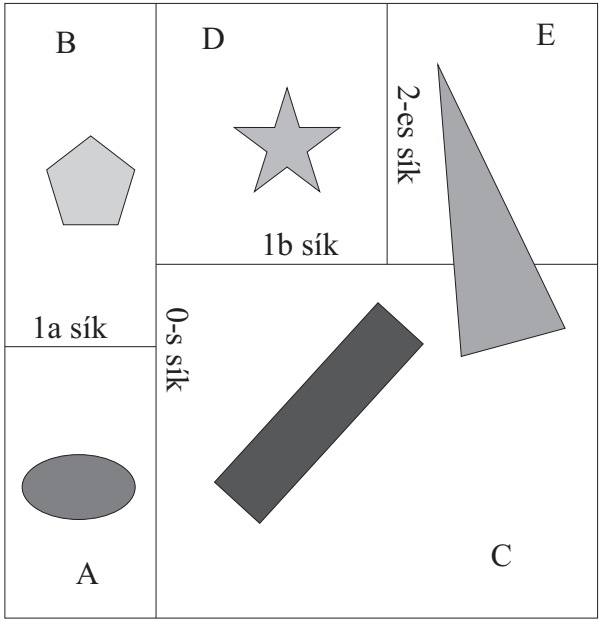
- Metszéstesztek felgyorsításához a hierarchikus ábrázolást alkalmazhatunk
- A környezetet BSP (bináris térparticionáló/Binary Space Partitioning) fával ábrázolhatjuk
- Attól függően, hogy milyen primitíveket használunk a környezetben, különböző sugár-objektum metszési módszerek szükségesek
- Két fajtáját különböztetünk meg
  - Tengely-igazított (axis-aligned)
  - Poligon-igazított (poligon-aligned)

BSP fák

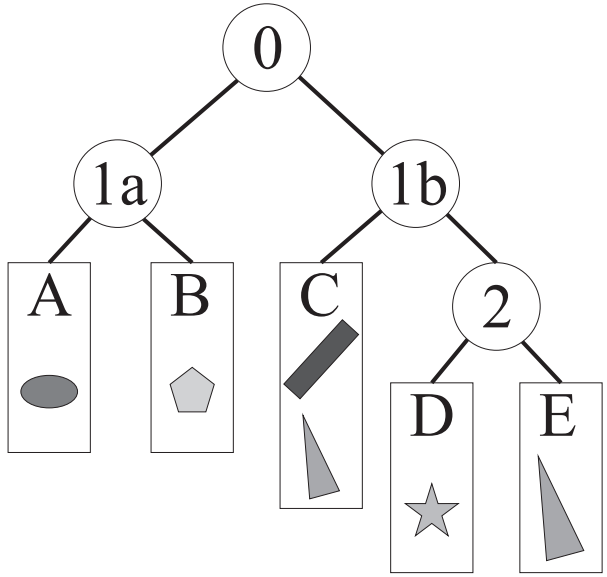
- A fákat a felosztás művelet rekurzív végrehajtásával hozzuk létre
  - A felosztáskor egy sík segítségével a teret két részre osztjuk
    - Ez rendszerint a tér egy poligonja
  - A geometriákat ebbe a két részbe rendezzük
- A geometriai tartalma a fának lerendezhető tetszőleges nézőpontból
  - Ha a fákat egy bizonyos módon járjuk be

- A tengely-igazított BSP fa létrehozása
  - A teljes színteret bekerítjük egy tengelyhez-igazított befoglaló dobozba
  - Ezt követően rekurzívan felosztjuk ezt a dobozt kisebb dobozokra
    - A doboz egyik tengelyét kiválasztjuk és egy merőleges síkot állítunk elő, amely kettévágja a teret két dobozra
    - Néhány esetben rögzítik ezt a felosztó síkot, amely két egyenlő részre osztja fel a dobozt
- Feltétel, ami megállítja a felosztást
  - Maximális fa mélység elérése
  - Egy dobozban lévő primitívek száma egy definiált küszöbérték alá nem esik

- Egy olyan objektum, amelyet a sík elmetsz
  - Vagy ezen a szinten van eltárolva
  - Vagy mind a két részhalmaz eleme lesz
  - Vagy pedig ténylegesen szét van vágva a síkkal két különálló objektumra



Tér felosztás



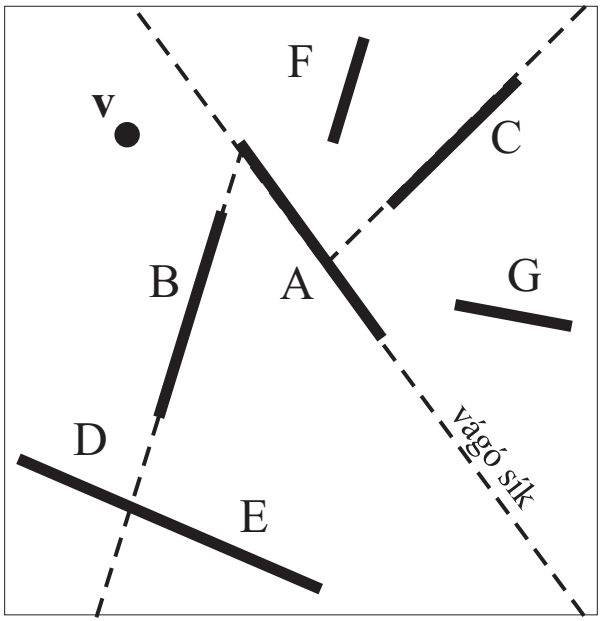
BSP fa struktúra

- Stratégia a doboz felosztására a tengelyek ciklikus váltogatása
  - A gyökérnél az x-tengely mentén vágunk
  - A gyerekeknél az y-tengely mentén vágunk
  - Az unokák esetén a z tengely mentén vágunk
  - ezt ismételjük
- Másik stratégia
  - A doboz legnagyobb oldalát keressük meg és e mentén daraboljuk a dobozt
- Kiegyensúlyozott fához az adott tengelyhez tartozó felosztási értékét kell úgy beállítani, hogy a két tér részbe egyenlő számú primitív kerüljön.
  - Gyakran a primitívek átlag vagy medián középpontját választjuk

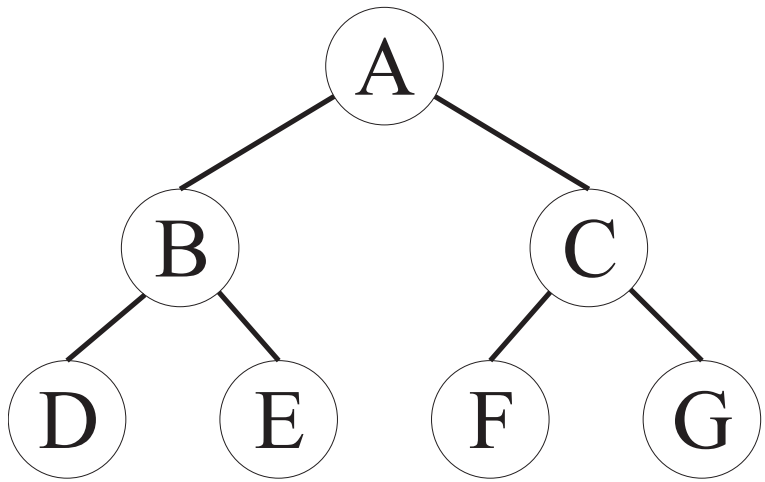
- Elölről-hátra rendezés
  - Tegyük fel, hogy egy N-nel jelölt csomóponton éppen áthaladunk
  - N a bejárás gyökere
  - Az N síkját megvizsgáljuk és a fa bejárását rekurzívan folytatjuk
    - A sík azon oldalán, ahol a nézőpont elhelyezkedik
  - A közelebbi rész bejárása a fának befejeződhet
    - Amikor egy csomópont doboza teljesen a nézőpont (pontosabban a közelebbi sík) mögött van
  - Ez nem ad pontos rendezést, mivel az objektumok a fa több csomópontjában is lehetnek
  - A bejárást a nézőponthoz viszonyított csomópont síkjának a másik oldalán elkezdve az objektumok egy hozzávetőleges rendezését kapjuk hátulról előre haladva

- Egy poligont választunk ki, mint felosztót felező sík
  - Ketté osztja a teret
  - Ez lesz a fa gyökere
- Azt a síkot választjuk ki, amelyiken a poligon fekszik
  - Arra használjuk ezt a síkot, hogy a színtér maradék poligonjait felosszuk két halmazra
    - Azokat a poligonokat, amelyeket a felosztó sík elmetsz, szétválasztjuk két elkülönülő darabra a metsző vonal mentén
    - Ezután a felosztó sík mindegyik félsíkjában egy másik poligont választunk felosztóként, amely csak az adott féltérben lévő poligonokat választja szét
  - Addig folytatjuk rekurzívan, amíg az összes poligon be nem kerül a BSP fába

- Hatékony poligon-igazított BSP fa előállítása időigényes eljárás
- Általában egyszer számítjuk ki
  - Az eltárolt változatot újra hasznosítjuk



Tér felosztás



BSP fa struktúra

- Az a legjobb, ha kiegyensúlyozott fát alakítunk ki
  - Minden levelének a mélysége ugyanaz vagy legfeljebb csak eggyel tér el a többitől
- Kiegyensúlyozatlan fa nem hatékony
- Legkevésbé-keresztezett feltétel
  - Több lehetséges poligont véletlenszerűen kiválasztunk
  - Azt a poligont használjuk, melyet legkevesebb alkalommal metszi el a többi poligon
- Egy 1000 poligonból álló teszt színtér esetén empirikus úton bebizonyították, hogy elegendő csak 5 poligont vizsgálni vágási műveletenként ahhoz, hogy jó fát kapjunk eredményül
  - Ha a színtéren található poligonok száma nagyobb, akkor ezt a számot valószínűleg növelni kell

- Hasznos tulajdonságok
  - Adott nézet esetén a struktúra pontosan bejárható hátulról-előre (vagy elölről-hátulra) haladva
    - Egy egyszerű pont/sík összehasonlítással lehet meghatározni azt, hogy a kamera a gyökér sík melyik oldalán található
    - Ettől a síktól távolabb lévő poligonok kívül esnek a kamerához közelebbi oldal poligonjaitól
    - Ezután a távolabbi oldal halmaza esetén vesszük a következő szint felosztó síkot és meghatározzuk, hogy a kamera melyik oldalán van
    - Az a részhalmaz, ahol a kamera található az a korábbi közelebbi részhalmaztól távolabb van és a távolabbi részhalmaz pedig a legtávolabb lévő részhalmaz a kamerától
  - Rekurzívan folytatva az eljárás létrehoz egy pontos hátulról-előre haladó sorrendet

- A sorrend nem garantálja, hogy az egyik objektum közelebb van, mint a másik
- Példa
  - A $\mathbf{v}$ az A vágó sík bal oldalán helyezkedik el
    - a C, F és G a B, D és E mögött vannak
    - C vágó síkkal összehasonlítva $\mathbf{v}$-t azt kapjuk, hogy G a sík ellentétes oldalán van
    - B sík egy tesztje megadja, hogy E-t D előtt kell megjeleníteni
  - A hátulról előre haladó sorrend ekkor, G, C, F, A, E, B, D

Dinamikus ütközés-detektálása BSP fák használatával

- Ütközések meghatározása
  - BSP fával leírt geometria
  - Ütköző
    - Gömb, henger vagy egy objektum konvex burka
- Alkalmas dinamikus ütközés detektálására
  - Egy gömb az n-edik frame-en lévő $\mathbf{p}_0$ pozícióból az $n+1$e-dik frame-en a $\mathbf{p}_1$ pozícióba mozog
  - Történt-e ütközés a $\mathbf{p}_0$ és $\mathbf{p}_1$ -et összekötő egyenes szakaszon?

- Az alap BSP fát hatékonyan lehet vonal darabok tesztelésekor használni
  - A vonal szegmenst egy pontként lehet ábrázolni
  - $\mathbf{p}_0$-ból $\mathbf{p}_1$-be mozog
- Az első metszés (ha van egyáltalán) adja meg az ütközést a pont és a BSP fában ábrázolt geometria között
- Könnyen ki lehet terjeszteni r sugarú gömb kezelésére
  - $\mathbf{p}_0$-ból $\mathbf{p}_1$-be mozog
  - Mindegyik síkot r távolságra mozgatjuk az egység normál mentén
    - Vonal szegmensek és a BSP fa csomópontokban tárolt síkok tesztelése helyett
  - Ütközés-kéréskor röptében hajtjuk végre

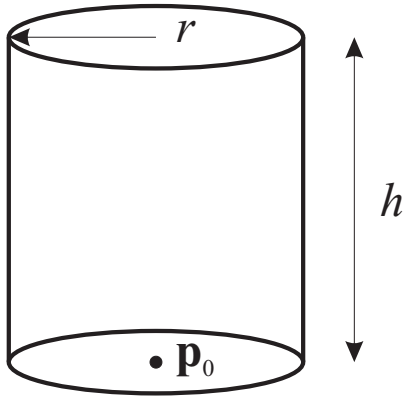
- BSP fát bármilyen méretű kör esetén használhatunk
- Feltéve, hogy egy sík $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$,
  - Kiigazított sík egyenlete $\pi : \mathbf{n} \cdot \mathbf{x} + d \pm r = 0$
  - Ahol az r előjele attól függ, hogy a sík melyik oldalán folytatjuk a tesztelést egy ütközés keresésében
- Feltéve, hogy a karakter a sík pozitív félterében van
  - $\mathbf{n} \cdot \mathbf{x} + d \geq 0$ ki kell vonnunk az r sugarat a d-ből
  - A negatív félteret *tömörnek* tekintjük
    - Valami, amit a karakter nem léphet át

- A gömb, egy karaktert nem igazán jól közelít meg
  - Konvex burka a karaktert alkotó vertexeknek, vagy egy henger, amely körülveszi a karaktert pontosabb eredményt ad
- Ahhoz, hogy ezeket a határoló térfogatokat használjuk a d értékét különböző módon kell kiigazítani a sík egyenletében
- Egy mozgó konvex burok S vertex halmazának a BSP fával való teszteléséhez a következő skalár értéket kell a síkegyenlet d értékéhez hozzáadni

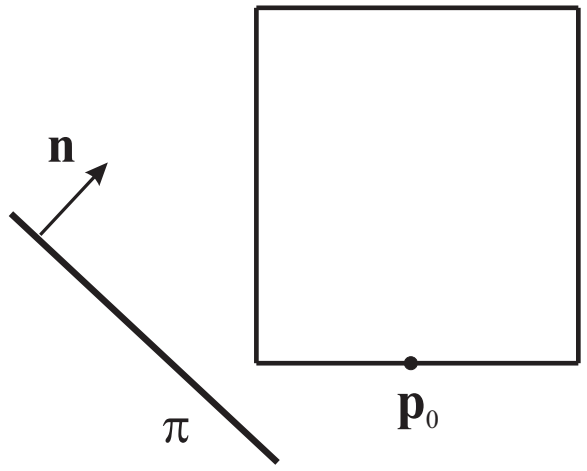
$$-\max_{\mathbf{v}_i \in S}(\mathbf{n} \cdot (\mathbf{v}_i - \mathbf{p}_0))$$

- A negatív előjel azt tételezi fel, hogy a karakter a síkok pozitív félterében mozog
- $\mathbf{p}_0$ pont tetszőlegesen megválasztott referencia pont
  - Gömb esetén a gömb középpontja
  - Egy karakternél egy lábhoz közeli pont választható
- A $\mathbf{p}_0$ pontra vizsgáljuk az ütközést a kiigazított BSP fában található síkokra
  - Dinamikus kéréskor a $\mathbf{p}_0$ pontot a vonal szegmens kezdő pontjaként használjuk
- Amennyiben egy képkocka alatt $\mathbf{w}$ vektorral mozdul
  - A vonal szegmens végpontja $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{w}$ lesz

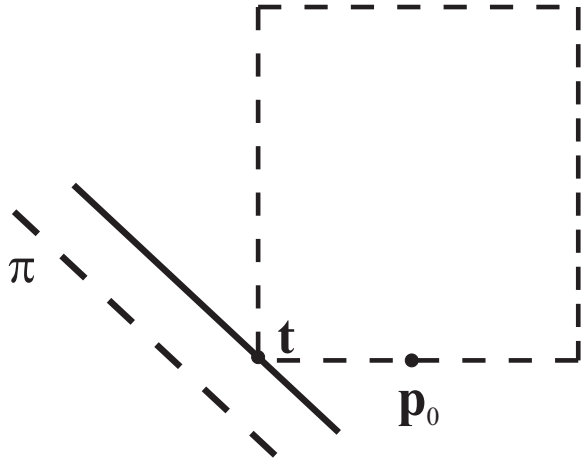
- Henger esetén
  - Gyorsabban lehet elvégezni a tesztet
  - Hasonlít egy karakterhez
- A sík egyenletét kiigazító érték származtatása bonyolultabb
- A határoló térfogat tesztelését a BSP fával átfogalmazzuk
  - Egy $\mathbf{p}_0$ pont tesztelése
  - Kiigazított BSP fával
- Ezt terjesztjük ki egy mozgó objektumra
  - A $\mathbf{p}_0$ pontot cseréljük ki egy $\mathbf{p}_0$-ból induló és $\mathbf{p}_1$-ben végződő vonal szegmensre



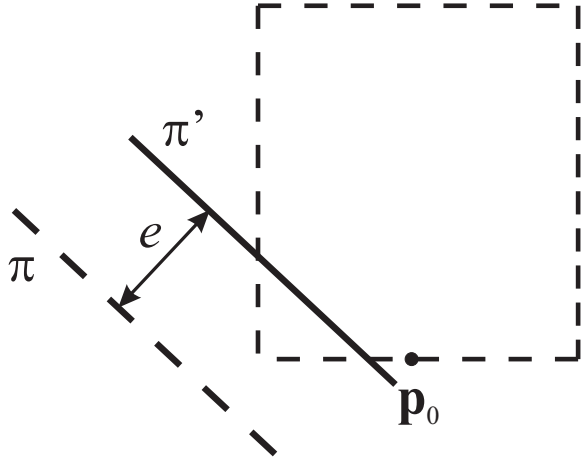
Henger teszt paraméterei



Henger tesztelése π síkkal



π sík mozgatása úgy, hogy a sík éppen csak érinti a hengert



π síkot π' síkba mozgatjuk az e távolsággal

- A tesztelést redukáltuk a $\mathbf{p}_0$ pont π' síkkal való tesztelésére
- e értékét online számítjuk ki mindegyik síkra és képkockára
  - Kiszámítjuk a $\mathbf{p}_0$-ból a $\mathbf{t}$ pontba mutató vektort
    - Ahol az elmozgatott sík érinti a hengert
  - Ezután e-t a következőképpen számítjuk ki

$$e = \left|\mathbf{n} \cdot (\mathbf{t} - \mathbf{p}_0)\right|$$

  - Ezután már csak $\mathbf{t}$-t kell kiszámítani

- A $\mathbf{t}$ z-komponense esetén, ha $n_z > 0$
  - $t_z = p_{0\,z}$
- Különben $t_z = p_{0\,z} + h$
- Ha n_x és n_y nulla
  - A henger aljának tetszőleges pontját használhatjuk
  - Egy választás a henger aljának a középpontja $(t_x, t_y) = (p_x, p_y)$
- Különben a henger aljának a szélén lévő pontot választjuk

$$t_x = \frac{rn_x}{\sqrt{n_x^2 + n_y^2}} + p_x,$$
$$t_y = \frac{rn_y}{\sqrt{n_x^2 + n_y^2}} + p_y,$$

  - Ahol a sík normálisát az xy-síkra vetítjük le, normalizáljuk és ezután r-rel skálázzuk

- Pontatlanság előfordulhat a módszer használata során
  - Hegyes kiszögellés esetén az ütközést korábban detektálhatjuk
  - Ennek a problémának a megoldására extra ferde síkokat vezetünk be
    - Gyakorlatban a külső szögét számítjuk ki a két szomszédos síknak
    - Egy extra síkot veszünk be, ha a szög nagyobb, mint 90°
    - Nem adnak megoldást az összes problémára

- A BSP fa `N` gyökerével hívjuk meg
- A gyerekei
  - `N.negativechild`
  - `N.positivechild`
- A vonal szakaszt
  - `p0` és `p1` pontok határozzák meg
- Az ütközés pontját (ha van ilyen)
  - Egy globális `p_impact` változóban kapjuk meg

```
HitCheckBSP(N,v0,v1)
returns({TRUE, FALSE});
if(not isSolidCell(N)) return FALSE;
else if(isSolidCell(N))
    p_impact = v0;
return TRUE;
end
hit = FALSE;
if(clipLineInside(N shift out, v0, v1, &w0, &w1))
hit = HitCheckBSP(N.negativechild, w0, w1);
if(hit) v1 = p_impact
end
if(clipLineOutside(N shift out, v0, v1, &w0, &w1))
hit = HitCheckBSP(N.positivechild, w0, w1);
end
return hit;
```

- Az `isSolidCell` `TRUE` értéket ad vissza
  - Ha elértük a fa levelét
  - A negatív féltérben vagyunk
- A `clipLineInside` `TRUE` értékkel tér vissza
  - ha a vonal szakasz (`v0` és `v1`) belül van a csomópont eltolt síkjában, ami a negatív féltérben van
  - Szintén metszi a vonalat a csomópont eltolt síkjával és visszatér az eredmény szakasszal (`w0` és `w1`)-ben
- A `clipLineOutside` hasonlóan működik
  - A `clipLineInside` és a `clipLineOutside` eljárások által visszaadott vonal szakaszok átfedik egymást
- Korrigált sík egyenletek gömbök, hengerek és konvex burok esetén
  - `N shift out` és `N shift in`

Összefoglalás

- Ütközés kezelés
- Ütközés-detektálás sugarakkal
- BSP fák
  - Tengely-igazított BSP fa
  - Poligon-igazított BSP fa
- Dinamikus ütközés-detektálása BSP fák használatával

Térbeli adatstruktúrák

Térbeli adatstruktúrák

- A nagy teljesítmény elérésének gyakori akadálya a valós-idejű alkalmazásokban a geometriai áteresztőképesség
  - A színtér és a modellek sok ezer vertexből épülnek fel
  - Normálvektorok, textúra koordináták és más attribútumok kapcsolódnak hozzájuk
- A CPU-nak és a GPU-nak kell feldolgoznia
- Az adatok másolása grafikus hardver felé szintén szűk keresztmetszet lehet
- Az OpenGL számos lehetőséget biztosít
  - Gyors geometria áteresztő képesség
  - Adatok rugalmas és kényelmes kezelése

Display listák

Display listák

- Primitívek kötegei
  - `glBegin()`/`glEnd()` függvény párosok
  - Egyedi `glVertex()` hívások
- Ez a lehető legrosszabb módja a geometria továbbítására a GPU felé
  - A teljesítményt is figyelembevételekor

```
glBegin(GL_TRIANGLES);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
glEnd();
```

Display listák

- Egy háromszög létrehozásához
  - 11 függvényhívást kell elvégeznünk
  - Mindegyik függvény potenciálisan drága ellenőrző kódot tartalmaz az OpenGL meghajtóban
  - 24 különböző négy byte-os paramétert kell a veremre helyezni
  - Visszaadni a hívó függvénynek
- Elképzelhető, hogy a grafikus hardver a CPU-ra várakozik
  - Amíg összerakja és továbbítja a geometriai kötegeket

Display listák

- Használhatóak
  - Vektor-paraméterű függvényeket
    - Konszolidálni lehet a köteget
  - Háromszög sávokat és legyezőket
    - Redundáns transzformációk és másolások csökkentésére
- Szükség van több ezer nagyon kicsi, potenciálisan költséges művelet geometriai kötegekben való elküldésére

- A meghajtó program az OpenGL
  - A parancsokat „valamilyen” módon speciális hardver utasításokra vagy műveletekre alakítja át
  - A grafikus kártyára küldi azokat
- A parancsok nem hajtódnak végre egyből
  - Egy lokális pufferben gyűlnek össze
  - Amíg egy határt el nem érnek
    - Ekkor ezek a parancsok a hardverhez kerülnek/ürítődnek (flush)
- A grafikus hardverhez vezető út sok időt vesz igénybe

- A puffer küldése a grafikus hardverhez egy aszinkron művelet
  - A CPU egy másik feladatot kezdhet el
  - Nem kell várnia az elküldött kötegelt renderelési utasítások befejeződéséig
- A hardver egy adott parancshalmaz renderelése alatt
  - A CPU azzal van elfoglalva, hogy egy új grafikus képhez tartozó parancsokat dolgoz fel
- Hatékonyan működik a CPU és a grafikus hardver között

- Három esemény idéz elő ürítést az aktuális renderelő parancsok kötegénél
  1.) A meghajtó program parancs puffere tele van
    - A pufferhez nem férünk hozzá és nem módosíthatjuk annak méretét sem
  2.) Akkor is történik ürítés, amikor egy puffer cserét hajtunk végre
    - A művelet addig nem hajtódik addig, amíg a sorban álló parancsok mindegyike végre nem hajtódik
    - Az adott színtér létrehozása befejeződött és az elküldött parancsok eredményének meg kell jelennie a képernyőn
  3.) Manuálisan idézzük elő az ürítést
    - Egyszeres színpuffert használunk
    - Az OpenGL nem tudja azt, hogy mikor fejeztük be a parancsok küldését

- Néhány OpenGL parancs azonban nem pufferelt későbbi végrehajtás céljából
  - A függvények közvetlenül érik el a frame puffert
    - Direkt módon olvassák és írják
  - Sebesség csökkenést idéznek elő a csővezetéken való áthaladásban
    - Az aktuális sorban álló parancsokat először ki kell üríteni és végre kell hajtani azokat, mielőtt a színpuffert közvetlenül módosítanánk
    - Erőszakosan kiüríthetjük a parancs puffert és várhatunk arra, hogy a grafikus hardver befejezze az összes renderelési feladatát a `glFinish()` függvény meghívásával
    - Ezt a függvényt csak nagyon ritkán használjuk a gyakorlatban

- OpenGL parancsok hívásához kapcsolódó tevékenységek költségesek azonnali renderelési mód esetén
  - A parancsok lefordulnak és átalakítódnak alacsony szintű hardver utasításokká
- Gyakran a geometria vagy másik OpenGL adat nem változik képkockánként
  - Például csak a modellnézeti mátrix változik
- Megoldás
  - Elmentjük a parancs pufferben található, előre kiszámított adatdarabot, amely valamilyen ismételt renderelési műveletet hajt végre
  - Ezt az adatdarabot később bemásolhatjuk egyszerre a parancspufferbe
    - Ezzel sok függvényhívást és fordítási munkát takarítunk meg, amely az adatot létrehozza

- Előfeldolgozott parancsok létrehozására OpenGL-ben
  - Display listák
  - Az OpenGL primitíveket `glBegin`/`glEnd` utasításokkal határoljuk el
  - A display listákat `glNewList`/`glEndList` függvény hívásokkal különítjük el egymástól
  - Egy egész értékkel azonosítjuk

```
glNewList(<unsigned integer name>,GL_COMPILE);
// ...
// OpenGL függvényhívások
// ...
glEndList();
```

- A `GL_COMPILE` paraméter
  - Csak lefordítja a listát
  - Még ne hajtja azt végre
- Használhatjuk a `GL_COMPILE_AND_EXECUTE` értéket is
- Rendszerint a display listákat csak felépítjük a program inicializálási részében és csak a rendereléskor hajtjuk végre azokat

- A display lista azonosító tetszőleges előjel nélküli egész lehet
  - Ha ugyanazt az értéket kétszer használjuk, akkor a második display lista felülírja az előzőt
    - `GLuint glGenLists(GLsizei range)`
    - Visszatérési értékként egy egyedi display lista azonosító sorozat első elemét kapjuk vissza
- Display lista felszabadítása
  - `glDeleteLists(GLuint list, GLsizei range)`
  - Felszabadítja
    - A display lista neveket
    - A listák számára lefoglalt memória területeket

- `glCallList(GLuint list)` függvényhívással tudjuk végrehajtani az előre lefordított OpenGL parancsokat tartalmazó listákat
- Display listákat tartalmazó tömbök esetén
  - `glCallLists(GLsizei n, GLenum type, const GLvoid *lists)` utasítás segítségével tudjuk lefuttatni
  - Az első paraméter a display listák száma
  - Második paramétere a tömb adat típusa
    - Rendszerint `GL_UNSIGNED_BYTE`
  - `lists` nevű tömb

- A display lista létrehozásának és végrehajtásának optimalizálása
  - Függ a megvalósítótól
- Akkor érdemes használni, amikor a lista állapotváltozásokat tartalmaz
  - Például fény ki- és bekapcsolása
- Egyes megvalósítások esetén a `glGenLists` utasítás használata szükséges a működéshez

- A display listában ne használjunk
  - `glReadPixels` függvény hívást
    - A frame puffert betölti egy memória területre mutató pointerbe
  - `glTexImage2D` függvény hívást
    - A textúra kétszer akkora memória területen lesz eltárolva
- A display lista nem tartalmazhat display lista létrehozást
  - Az egyik display listában meghívhatjuk a másik display listát

Vertextömbök

Vertextömbök

- A modell vertex adatait előre kiszámítva egy tömbben eltárolhatjuk

  Hátránya Végig kell menni a teljes tömbön, és vertexenként kell az adatokat az OpenGL-nek átadni

  Előnye A geometria változhat a műveletek során

- Mind a két megoldás jó tulajdonságát kihasználhatjuk vertextömbök használatával
- A CPU és GPU között lehet átvinni
  - Előre kiszámított vagy módosított geometriákat
  - Nagy mennyiségben
  - Egy időben

1.) Össze kell rakni a geometriához tartozó adatokat egy vagy több tömbben.
   - Ezt algoritmikusan, illetve egy állományból betöltve is el lehet végezni
2.) Meg kell mondani az OpenGL-nek, hogy hol van az adat
   - Renderelés során az OpenGL a vertex adatot a megadott tömbökből „húzza” be
3.) Világosan meg kell mondani, hogy mely tömböket használja az OpenGL
   - Elkülönített tömbökben tárolhatunk vertexeket, normálvektorokat, színeket, stb.
4.) Végre kell hajtani az OpenGL parancsokat
   - A megadott adatok alapján előállítják a megfelelő objektumot/objektumokat

- Modelljeinket tömbökben kell tárolni

```
// Vertex száma
GLuint VertCount = 100;

// Vertex pointer
GLfloat *pData = NULL;
// Normálvektor pointer
GLfloat *pNormals = NULL;
// ...

// Megfelelő méretű memória terület foglalása
  pData = malloc(sizeof(GLfloat) * VertCount * 3);
  pNormals = malloc(sizeof(GLfloat) * VertCount *
      3);

// ...
// Adatok feltöltése
// ...
```

- A `RenderScene` függvényben engedélyeznünk kell a vertexek és normálvektor tömbök használatát

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
```

- A letiltásra a `glDisableClientState(GLenum array)` függvényt használhatjuk
- Lehetséges paraméterek
  - `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXURE_COORD_ARRAY` és `GL_EDGE_FLAG_ARRAY`

- Miért van szükség egy új függvényre?
  - A `glEnable`-t is használhatnánk erre a feladatra
- Az OpenGL kliens-szerver modell alapján van megtervezve

  Szerver A grafikus hardver

  Kliens A gazda CPU és memória
- Az engedélyezett/letiltott (enable/disable) állapotot a kliens oldali képre alkalmazzuk

- A vertex adatok használata előtt, meg kell mondani, hogy hol tároljuk az adatokat

```
glVertexPointer(2, GL_FLOAT, 0, pData);
```

- A többi vertextömb adattípus tartozó függvények

```
void glVertexPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glColorPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glTexCoordPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glSecondaryColorPointer(GLint size, GLenum type,
GLsizei stride, const void *pointer);
void glNormalPointer(GLenum type,
GLsizei stride, const void *pData);
void glFogCoordPointer(GLenum type,
GLsizei stride, const void *pointer);
void glEdgeFlagPointer(GLenum type,
GLsizei stride, const void *pointer);
```

- `type` paraméter adja meg a tömb adattípusát

Parancs	Elemek	Érvényes adattípusok
`glColorPointer`	3, 4	`GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, `GL_DOUBLE`
`glEdgeFlagPointer`	1	nem meghatározott (mindig `GLboolean`)
`glFogCoordPointer`	1	`GL_FLOAT`, `GL_DOUBLE`
`glNormalPointer`	3	`GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`
`glSecondaryColorPointer`	3	`GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, `GL_DOUBLE`
`glTexCoordPointer`	1, 2, 3, 4	`GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`
`glVertexPointer`	2, 3, 4	`GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`

- textttstride paraméter byte-ban adja meg két egymást követő tömbelem közötti eltolás mértékét
  - Amennyiben ez az érték 0-val egyenlő, akkor ez azt jelenti, hogy az elemek a tömbben szorosan egymásután vannak elhelyezve
- Az utolsó paraméter, az adat tömbre mutató pointer
- Vertextömbök esetén a cél textúra egységet `glClientActiveTexture(GLenum texture)` függvény hívással lehet beállítani a `glTexCoordPointer` számára
  - A `target` paraméter `GL_TEXTURE0`, `GL_TEXTURE1` ... értékeket veheti fel

- Vertex adatok kinyerése

```
glBegin(GL_POINTS);
for(i = 0; i < VertCount; i++)
glArrayElement(i);
glEnd();
```

  - Veszi a megfelelő tömb adatokat azokból a tömbökből, amelyek a `glEnableClientState` függvénnyel engedélyezve lettek
  - Vertex, normál, szín, stb.

- Adott blokk teljes átküldése
  - `glDrawArrays(GLenum mode, GLint first, GLint count)`

    `mode` Milyen primitívet akarunk előállítani
    `first` A tömb melyik elemétől kezdve
    `count` Hány elemet akarunk a tömbökből kinyerni

```
glDrawArrays(GL_POINTS, 0, VertCount)
```

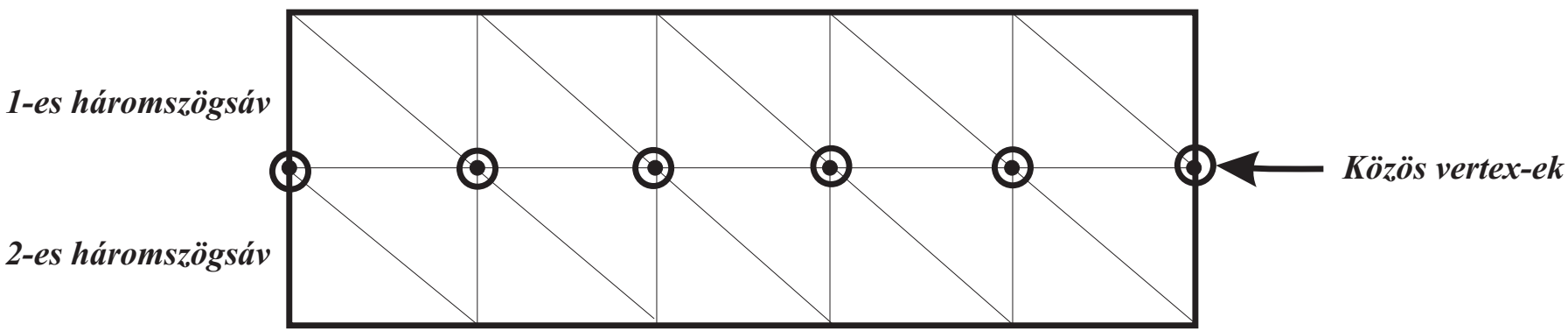
Indexelt vertextömbök

Indexelt vertextömbök

- A vertextömbhöz tartozik egy külön index értékeket tároló tömb
  - Megadja azt, hogy melyik vertexeket és milyen sorrendben kell felhasználni az adott objektum felépítésekor
- Memória területet takaríthatunk meg és csökkenthetjük a transzformációs költségeket is
- Kapcsolódó primitívek közös vertexekkel rendelkezhetnek, melyeket nem lehet egyszerűen háromszögsávok, háromszög-legyezők, négyszögsávok használatával megoldani

Indexelt vertextömbök

- Két szomszédos háromszögsáv esetén, nem létezik olyan módszer, amellyel vertexek halmazát meg lehetne osztani



Indexelt vertextömbök

- Amennyiben a vertexeket vagy a normálvektorokat újra felhasználjuk a vertextömbökben
  - Csökkenteni tudjuk a memória használatot
  - Csökkenteni lehet a transzformációk számát
  - A transzformációval töltött időt is jelentősen csökkenteni lehet

Indexelt vertextömbök

```
// ...
// A kocka nyolc sarka
static GLfloat sarkok[] =
// A kocka előlapja
    { -25.0f, 25.0f, 25.0f,
    25.0f, 25.0f, 25.0f,
    25.0f, -25.0f, 25.0f,
    -25.0f, -25.0f, 25.0f,
// A kocka hátlapja
    -25.0f, 25.0f, -25.0f,
    25.0f, 25.0f, -25.0f,
    25.0f, -25.0f, -25.0f,
    -25.0f, -25.0f, -25.0f };

// Az index tömb,
static GLubyte indexek[] =
    // Előlap
    { 0, 1, 2, 3,
    //Felső lap
    4, 5, 1, 0,
    Alsó lap
    3, 2, 6, 7,
    // Hátlap
    5, 4, 7, 6,
    // Jobb oldali lap
    1, 5, 6, 2,
    // Bal oldali lap
    4, 0, 3, 7 };

// ...
```

```
void RenderScene(void)
{
// ...

// A vertextömb engedélyezése és
    megadása
glEnableClientState(
    GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0,
    sarkok);

// Négyszögsávokkal való
    megjelenítés
glDrawElements(GL_QUADS, 24,
    GL_UNSIGNED_BYTE, indexek);

// ...
}
```

Indexelt vertextömbök

- `glDrawElements` függvény nagyon hasonlít a `glDrawArrays` függvényre
  - Az index tömböt is meg kell adni
  - Milyen sorrendben kell a vertextömböt tekinteni
- `glDrawRangeElements`
  - Az indexek mely részét kell felhasználni az objektum létrehozásához
- `glMultiDrawArrays`
  - Több index tömböt lehet elküldeni egyetlen függvényhívás segítségével
- `glInterleavedArrays`
  - Több tömböt egy összesített tömbben helyezzünk el
  - A tömbök memória szervezése javíthatja a teljesítményt néhány hardveres megvalósítás esetén

Vertex puffer objektumok

Vertex puffer objektumok

- A display listák egy gyors és könnyű módszert adnak az azonnali kódolási módban
  - Legrosszabb esetben a display lista egy előre lefordított OpenGL adatot fog tartalmazni
    - Gyorsan a parancs pufferbe lehet másolni és elküldeni a grafikus hardverhez
  - Legjobb esetben
    - Egy display listát a grafikus hardverbe másolhat
    - A hardver sávszélességét lényegében nullára csökkentve
  - Vertextömbök biztosítják számunkra az összes rugalmasságot
  - Az indexelt vertextömbök segítségével csökkenthetjük a vertex adatok mennyiségét

Vertex puffer objektumok

- Még egy lehetőséget biztosít a geometriai áteresztőképesség vezérlésére
  - Vertextömbök használatakor át lehet küldeni a CPU kliens oldaláról egyedi tömböket a grafikus hardverre
  - Vertex puffer objektum
    - A vertextömböket ugyanúgy használjuk és kezeljük, mint a textúra adatok betöltését és kezelését

- Vertextömböket kell használnunk
- Puffer objektumokat kell létrehoznunk
  - `glGenBuffers` függvény meghívásával
    - Első paramétere a kért objektumoknak a számát adja meg
    - Második paraméter pedig egy olyan tömb, ami az új puffer objektumok neveivel van feltöltve
  - `glDeleteBuffers` utasítással lehet felszabadítani
- A `glBindBuffer` függvény összeköti az aktuális állapotot egy bizonyos puffer objektummal
  - A `target` paraméter határozza meg azt, hogy milyen típusú tömböt fogunk kijelölni
    - `GL_ARRAY_BUFFER` a vertex adatok esetén
    - `GL_ELEMENT_ARRAY_BUFFER` index tömbök számára

- Először hozzá kell csatolni a szóban forgó puffer objektumot
- Aztán kell meghívni a `glBufferData` függvényt

```
glBufferData(GLenum target, GLsizeiptr size,
 GLvoid *data, GLenum usage)
```

`target` `GL_ARRAY_BUFFER` vagy `GL_ELEMENT_ARRAY_BUFFER` értékeket kaphatja meg

`size` Adja meg a vertextömb méretét byte-ban

`data` Inicializálásként az adat tárolóba bemásolandó adat

`usage` A várható használati minta

Használati utasítás	Leírás
`GL_DYNAMIC_DRAW`	A puffer objektumban tárolt adat valószínűleg sokszor fog változni, de a változások között a kirajzolásra többször fogja használni a forrást. Ez a segítség a megvalósítás számára jelzi, hogy az adatot olyan helyen kell tárolni, melynek **időnkénti megváltoztatása nem okoz nagy gondot**.
`GL_STATIC_DRAW`	A puffer objektumban tárolt adat nem valószínű, hogy változni fog és sokszor ez lesz a forrása a rajzolásnak. Ez azt jelzi, hogy a megvalósításnak az adatot olyan helyen kell tárolnia, ami **gyorsan olvasható, de nem szükséges gyorsan frissíteni azt**.
`GL_STREAM_DRAW`	A pufferben tárolt adat gyakran változik és a változások között csak néhányszor lesz szükség a forrásra. Ez a segítség azt jelzi a megvalósításnak, hogy időben változó geometriai (például animált geometria) objektumot tárolunk, amit egyszer használunk és utána meg fog változni rögtön. Elengedhetetlen, hogy az ilyen jellegű adatot olyan helyen tároljuk, amelyet **gyorsan lehet frissíteni még a gyorsabb renderelés kárára is**.

- Hozzá kell rendelni az adott vertextömböt a renderelési módhoz mielőtt a vertex mutató függvényt meghívnánk
- Az aktuális tömbre mutató pointer ezentúl egy eltolás lesz a vertex puffer objektumban

```
glVertexPointer(3,
                GL_FLOAT,
                0, pVerts);
```

```
glBindBuffer(GL_ARRAY_BUFFER,
             bufferObjects[0]);
glVertexPointer(3,
                GL_FLOAT,
                0, 0);
```

- Renderelési függvényhívások

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufferObjects[3])
    ;
glDrawElements(GL_TRIANGLES, nNumIndexes,
    GL_UNSIGNED_SHORT, 0);
```

Poligon technikák

Poligon technikák

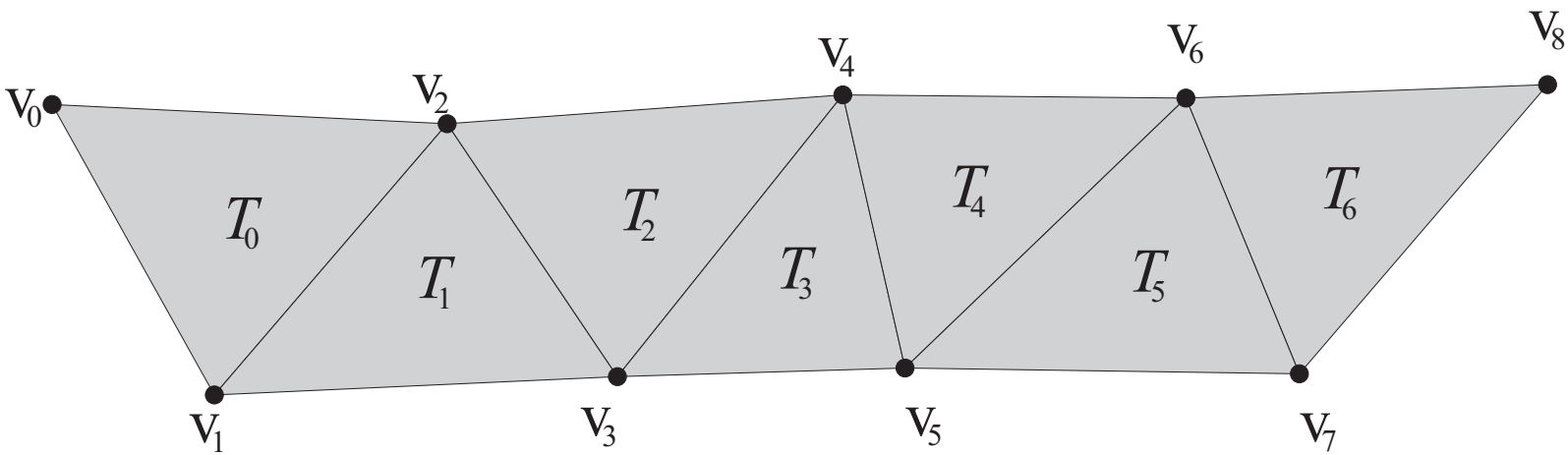
- A poligon ábrázolás általános célja a vizuális pontosság és a sebesség
- A pontosság mindig az adott környezettől függ
  - Egy modell adott pontossággal jelenik meg
  - Egy repülőgép-szimulátor esetén
    - A mérnök irányítani és pozicionálni akarja a modelleket valós időben és minden részletet minden időpillanatban látni akar a számítógépen
    - Egy játék esetén, ahol ha a képkockák sebessége elég magas, kisebb hibák vagy pontatlanságok az adott képkockán belül megengedettek

- Poligon felbontás
  - Az a folyamat, amikor a felületet poligonok halmazára bontjuk fel
- Poligon felületek felbontásával fogunk foglalkozni
- Háromszögek, mint elemi alkotó részek vesznek részt a renderelés során
  - Tetszőleges felületek előállíthatóak belőlük

- A renderelő lehet, hogy csak konvex poligonokat tud kezelni
- Előfordulhat, hogy a felületet kisebb részekre kell vágni azért,
  - Az árnyalás vagy a visszatükröződő fény megfelelően jelenjenek meg
- Poligon felbontására vonatkozó feltételek lehetnek
  - Egyetlen egy poligon se legyen nagyobb egy előre megadott területnél
  - Háromszögek esetén a háromszögek szögeinek nagyobbaknak kell lenniük egy előre megadott minimum szögnél
    - Nem-grafikai alkalmazások (pl. véges elem analízis) esetén megszokottak
    - Felület megjelenését is javítják
  - A hosszú, vékony háromszögeket jobb elkerülni
    - Különböző árnyalások esetén a vertexek közötti nagy távolságok miatt interpoláláskor a hibák jobban megjelennek

- Első lépés egy felület felbontása esetén
  - Egy 3D-s poligon esetén meghatározzuk azt, hogy melyik a legjobb vetítés 2D-re
  - A problémát és a probléma megoldásául szolgáló algoritmust leegyszerűsítsük
    - A poligont leképezzük az xy, yz és xz síkokra
    - Az a legjobb sík, amikor az adott poligon esetén a legnagyobb leképezett területet kapjuk
    - A legnagyobb nagyságú koordinátát hagyjuk el a poligon normálvektorában
  - A poligon normálvektor tesztjével nem mindig lehet meghatározni a legnagyobb területű vetületet

- A grafikus teljesítmény növelésének egy nagyon gyakori módja
  - Kevesebb vertexet küldünk át háromszögenként a grafikus csővezetéken
- A háromszögsávok és háromszöglegyezők esetén a közös vertexeket csak egyszer kell elküldeni



- Egy **szekvenciális háromszögsávot**, rendezett vertexek listájával definiálhatjuk
  - $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$
    - Ahol a T_i az i-ik háromszöget $\triangle \mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli
    - $0 \leq i < n-2$ esetén
  - A vertexeket a megadott sorrendben küldjük a GPU felé
- A szekvenciális háromszögsáv n vertexe $n-2$ háromszöget határoz meg
- A vertexek v_a átlagos száma egy m hosszú szekvenciális háromszögsáv esetén a következőképpen fejezhető ki

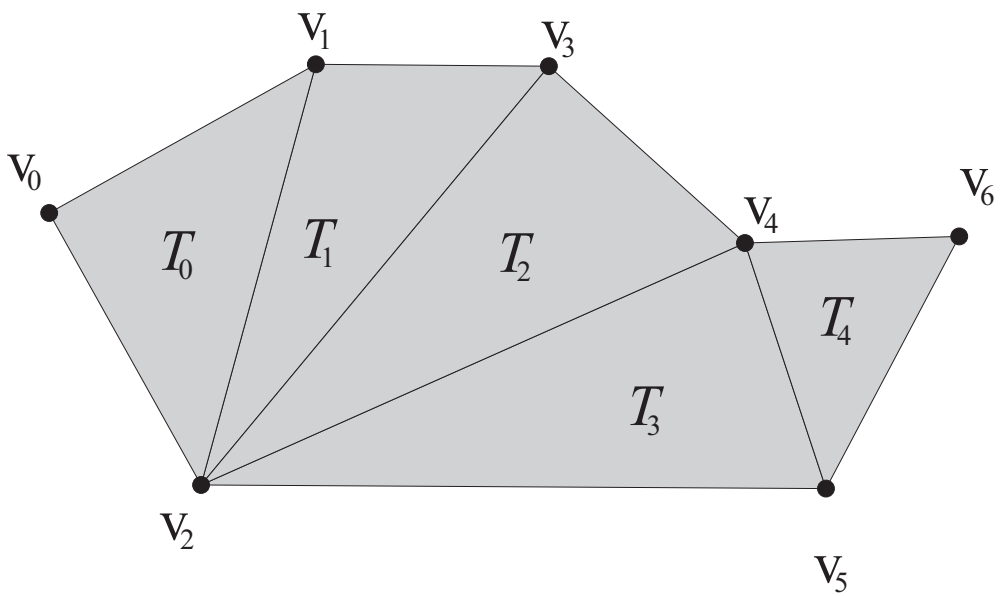
$$v_a = \frac{3 + (m-1)}{m} = 1 + \frac{2}{m}$$

  - $m \to \infty$ esetén $v_a \to 1$

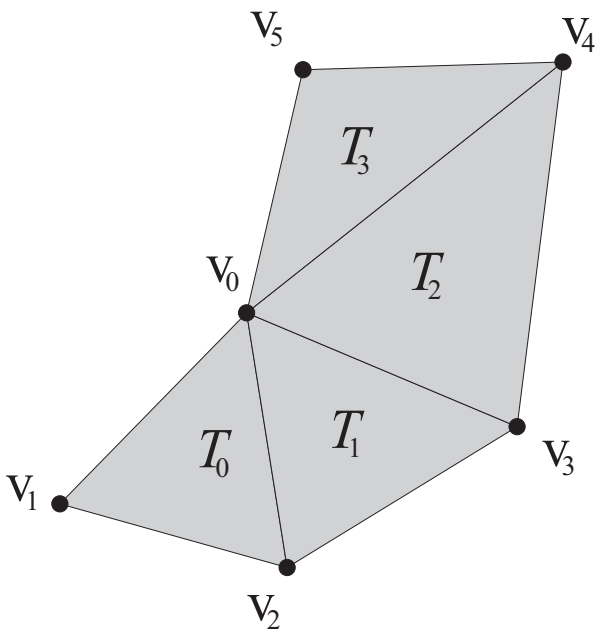
- Ha nem követeljük meg a szigorú szekvenciáját a háromszögeknek
  - Hosszabb és ezáltal hatékonyabb sávokat hozhatunk létre
  - **Általánosított háromszögsávok**
    - Szükség van vertex gyorsítótárra a grafikus kártyán
    - A transzformált és a megvilágított vertexeket tárolja
    - A vertexeket el lehet érni és ki lehet cserélni rövid bit kódok küldésével
    - Amikor a vertexek a gyorsítótárba kerülnek, akkor más háromszögek is felhasználhatják azokat elenyésző költséggel

- A szekvenciális háromszögek „általánosításához” a *csere* műveletet kell bevezetnünk
  - Megcseréli a két utolsó vertexnek a sorrendjét
  - OpenGL-ben és Direct3D-ben a csere parancsot egy vertex újra küldésével valósíthatjuk meg
    - Cserénként egy vertexnyi plusz költséget jelent
  - Olyan háromszöget hoz létre, melynek nincs területe
- Egy új háromszögsáv létrehozásának a költsége két vertex
- Az aktuális API hívások, melyek a sáv küldésért felelősek, további költségeket jelentenének

- Egy háromszögsáv, amely a $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathit{csere}, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$ vertexek átküldésére várakozik megvalósítható a következőképpen
  - $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$
    - Ahol a csere a $\mathbf{v}_2$ vertex újraküldésével lett megvalósítva



- A háromszög-legyező hasonló jó tulajdonsággal rendelkezik, mint a háromszögsáv
- A legtöbb alacsony szintű grafikus API támogatja a háromszög-legyezők létrehozását
- Egy általános konvex poligont könnyen lehet háromszög-legyezővé alakítani
  - Háromszögsávvá is könnyű alakítani



- A *középső vertexen* az összes háromszög osztozik
- Egy új háromszöget a középső vertex, az előzőleg elküldött vertex és az új vertex segítségével hozzuk létre
- Az n vertexből felépülő háromszög-legyezőt a $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ rendezett vertex listái alkotják
  - $\mathbf{v}_0$ a középső vertex
  - Az i-ik háromszög $\triangle \mathbf{v}_0, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli, $0 \leq i < n-2$ esetén
- Bármelyik háromszög-legyező átalakítható háromszögsávvá
  - Sok cserét fog tartalmazni
  - Fordítva nem hajtható végre

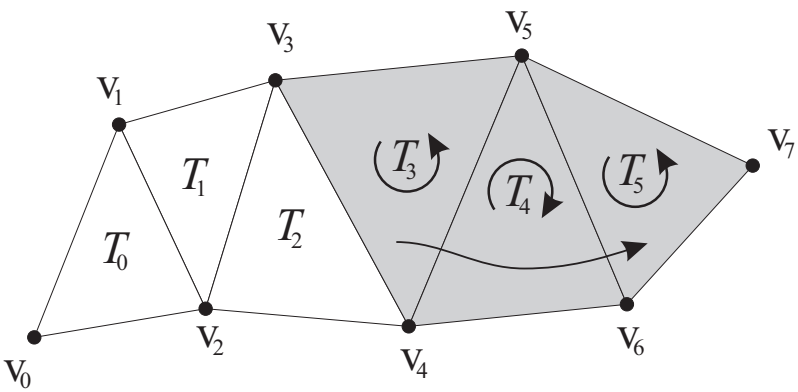
- Általános háromszöghálót érdemes hatékonyan szétbontani háromszögsávokra
- Optimális háromszögsávok előállítására NP-teljes probléma
- Meg kell elégednünk heurisztikus módszerekkel
- Mindegyik háromszögsáv létrehozó algoritmus a poligon halmaz szomszédsági adat struktúrájának a létrehozásával kezdődik
  - Mindegyik poligonhoz tartozó él esetén szomszédos poligonra való hivatkozást is el kell tárolni
  - A poligonok szomszédjainak a számát *foknak* nevezzük
  - Egész értéke 0 és a poligon vertexeinek a száma között van

- *Euler* síkbeli kapcsolódó gráfokra vonatkozó tétele: $v - e + f - 2g = 2$
  - v a vertexek száma
  - f a lapok száma
  - g az objektumban lévő lyukak száma
- Meghatározhatjuk a vertexek átlagos számát, amit a csővezetékbe küldünk
  - A $2e \geq 3f$ mindig teljesül
    - Kapcsolódó gráfok esetén minden él mentén két lap helyezkedik el
    - Minden lapnak legalább három éle van
  - emphEuler tételbe behelyettesítve
  - Egyszerűsítés után $f \leq 2v - 4$
  - Ha mindegyik lap háromszög, akkor $2e = 3f \Rightarrow f = 2v - 4$
- A háromszögek száma kisebb vagy egyenlő a vertexek számának a kétszeresénél a háromszögekre való felbontáskor
  - Minden vertexet (átlagosan) legalább kétszer kell elküldeni a szekvenciális háromszögsáv használatakor

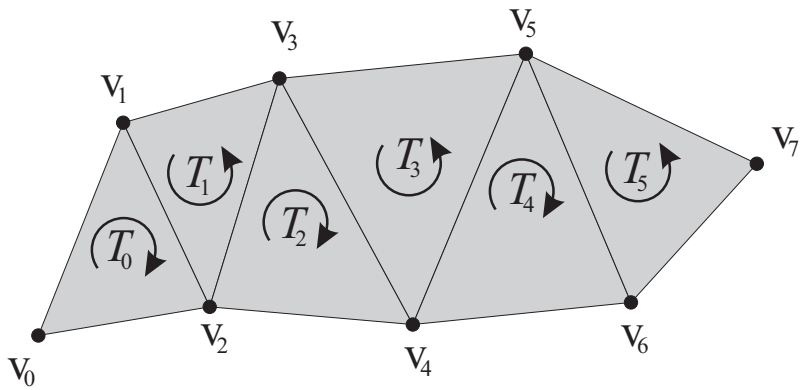
- Ez az algoritmus csak olyan modellek esetén működik, amelyek teljesen háromszögekből épülnek fel
- Lokális optimumok alapján dönt
  - Mindig azt a kezdő háromszöget választja, amelyiknek a legkisebb a foka (legkevesebb szomszédos oldala van)
- **SGI háromszögsáv-képző algoritmus**
  1.) Válasszuk ki a kezdő háromszöget.
  2.) Építsünk 3 különböző háromszögsávot a háromszög minden éle mentén.
  3.) Terjesszük ki ezeket a háromszögsávokat az háromszögsáv első elemétől az ellenkező irányba.
  4.) Válasszuk ki a három közül a leghosszabb háromszögsávot és töröljük a többit.
  5.) Ismételjük meg a folyamatot az 1-es lépéstől addig, amíg az összes háromszög be nem került a sávba.

- Az első lépésben az algoritmus a legkisebb fokszámú háromszöget választja ki
  - Több ilyen megegyező fokszámú háromszög esetén szomszédos háromszögek szomszédjainak a fokszáma alapján dönt
  - Ha ezek után még mindig nem egyértelmű a háromszög kiválasztása, akkor tetszőlegesen kiválaszt egyet
- A sávot a háromszögsáv kezdő és végső háromszögének az irányba terjesztjük ki
  - Az elkülönített háromszögek nem szerepelnek egyetlen egy háromszögsávban sem
  - Ezeknek a háromszögeknek a számát minimalizálja
- Lineáris idejű algoritmus megvalósítható
  - Hash táblák segítségével szomszédsági adatokat tárolására
  - Prioritási sorokkal, amelyeket mindegyik új sáv kezdő háromszögének keresésére használunk fel
- Tetszőleges háromszöget választva az algoritmus során ugyan olyan jó eredményt kaphatunk

- Praktikus szempont
  - A háromszögek irányítottságát meg kellene őrizni
  - A háromszögsáv első háromszöge határozza meg a háromszögek körbejárását
- Mi történik abban az esetben, ha a körbejárás megváltozik a kiterjesztés során?

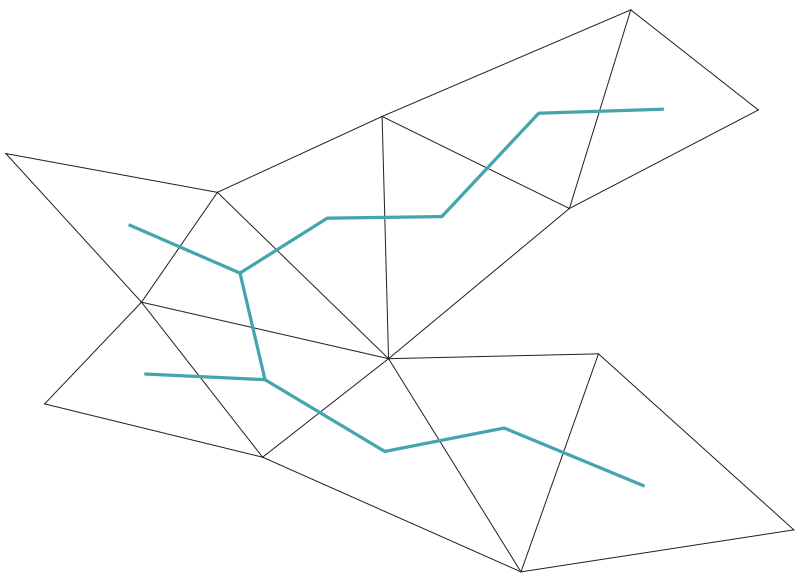


T_3 háromszög órajárással ellentétes körbejárású



T_0 háromszög órajárással megegyező körbejárású

- A háromszög-háló duális gráfjának a használata
- A gráf élei a szomszédos lapok középpontjait összekötő élei lesznek
- Ezen élek gráfját *feszítőfának* nevezzük
  - A jó háromszögsávok keresésére használhatunk fel

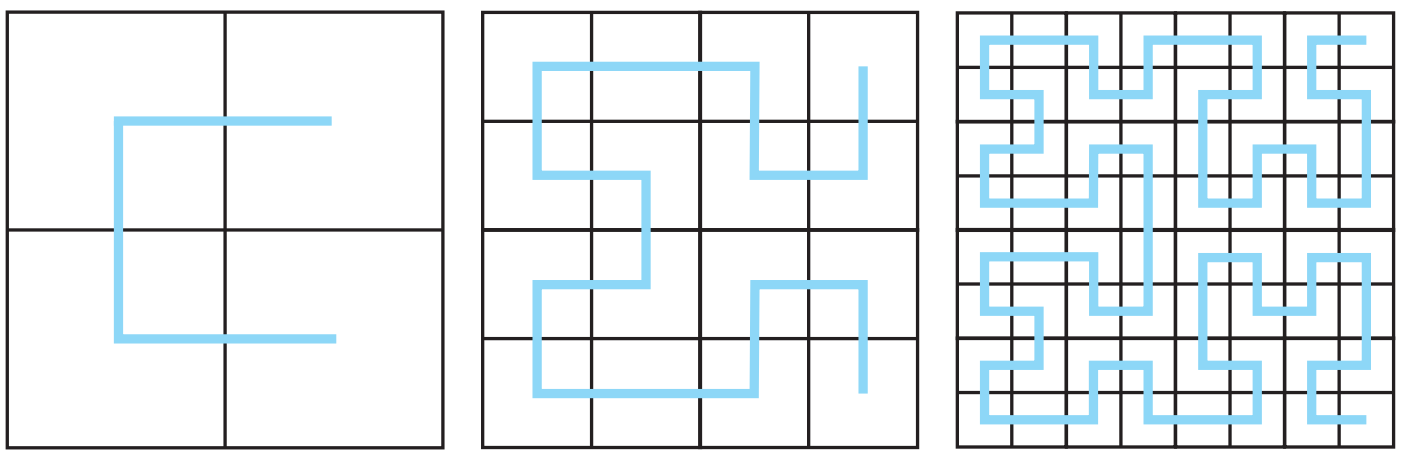


- Vertexek listájából és körvonalak halmazából állnak
- Minden vertex pozíció és további adatokat tartalmaz
  - Diffúz és spekuláris színeket
  - Árnyalási normálvektort
  - Textúra-koordinátákat
- Mindegyik háromszög körvonal egész értékű indexek listájával rendelkezik
- Ezek az indexek a vertexekre mutatnak a listában

- Tételezzük fel, hogy a háromszöghálókat háromszögsávokkal hoztuk létre
  - A grafikus kártyák többségének van vertex puffere
  - A csővezetéken átküldendő sávok sorrendjei különböző vertex puffer teljesítményt fognak mutatni
    - A gyorsítók különböző tárméretekkel rendelkeznek
- Előfeldolgozási művelettel javíthatjuk a sorrendet
  1.) Vegyünk egy háromszögsávot, melynek a vertexeit a vertex gyorsítótár (FIFO) egy szoftveres szimulációjában helyezzük el
  2.) A fennmaradó háromszögsávok közül kiválasztjuk azt, amelyik a legjobban használja ki a tár tartalmát
  3.) Az eljárást addig ismételjük, amíg az összes háromszögsávot fel nem dolgoztuk
- Az NVIDIA NVTriStrip függvénykönyvtár is pontosan ezeket a lépéseket követi
  - http://developer.nvidia.com/object/nvtristrip_library.html

- Adott egy indexelt háromszög háló
  - Amely hatékonyan renderelhető egy primitív vertex gyorsítótárral rendelkező grafikus hardver segítségével
- A kérdés továbbra is az indexek megfelelő sorrendjének meghatározása
  - Különböző hardverek különböző méretű vertex gyorsítótárral rendelkeznek
  - Mindegyik méretre más és más módon állítjuk elő az indexek sorrendjét?!?
- Az igazi megoldás egy *univerzális* index sorozat előállítása
  - Az összes lehetséges vertex gyorsítótár méret esetén jól viselkedik

- A *terület-kitöltő görbe* fogalma
  - Egy egyszerű, folytonos görbe
  - Nem metszi önmagát
  - Kitölt egy olyan négyzetet vagy téglalapot, amely uniform négyzetrácsait csak egyszer érint annak bejárása során
- A jó terület-kitöltő görbe jó térbeli összefüggőséggel rendelkezik
  - Amikor bejárjuk azt mindig az előzőleg meglátogatott pontok közelében maradunk
  - A Hilbert görbe egy példa a terület-kitöltő görbére



- A háromszög háló esetén a vertex gyorsítótárban nagy találati arányra számíthatunk terület-kitöltő görbe használatakor
- Egy rekurzív algoritmussal jó index sorozatot lehet létrehozni a háromszöghálókra
- Alapötlet
  - Szétvágjuk a hálót megközelítőleg két egyforma méretű hálóra
  - Majd az egyik illetve a másik hálót rendereljük le
  - Ezt ismételjük rekurzívan minkét hálóra
- A háló szétvágását előfeldolgozási lépésként hajtjuk végre kiegyensúlyozott él-vágás algoritmussal
  - Minimalizálja az él vágások számát
- Az algoritmus komplexitása lineáris a hálóban lévő háromszögek számára nézve

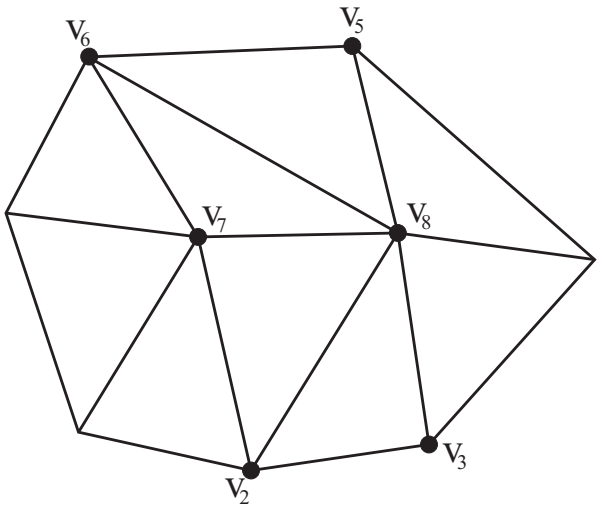
1.) Hozzuk létre a háromszögháló duális gráfját.
2.) Ezután a kiegyensúlyozott él-vágás algoritmussal eltávolítjuk a minimális élek halmazát a duális gráfban
    - A hálót két különálló, nagyjából megegyező méretű hálóra vágjuk szét
3.) A jó gyorsítótár teljesítmény eléréséhez ajánlott, hogy az utolsó háromszög az első hálóban közel legyen a második háló első háromszögéhez.
    - Az első háló utolsó háromszöge és a második háló első háromszöge esetén megengedjük, hogy a duális gráfban egy élen osztozzanak, amelyet átvágtunk

- A *háló egyszerűsítéssel adat csökkentésként* vagy *decimálásként* is találkozhatunk
  - Egy részletes modell poligonjainak a számának csökkentése
  - Megpróbálja megőrizni annak megjelenését
- Valósidejű munka során ez az eljárás a csővezetéken átküldendő vertexek számát csökkenti
  - Fontos lehet az adott alkalmazás régebbi számítógépeken való futtatásakor
- Modellek redundánsak lehetnek egy elfogadható megjelenítés esetén is

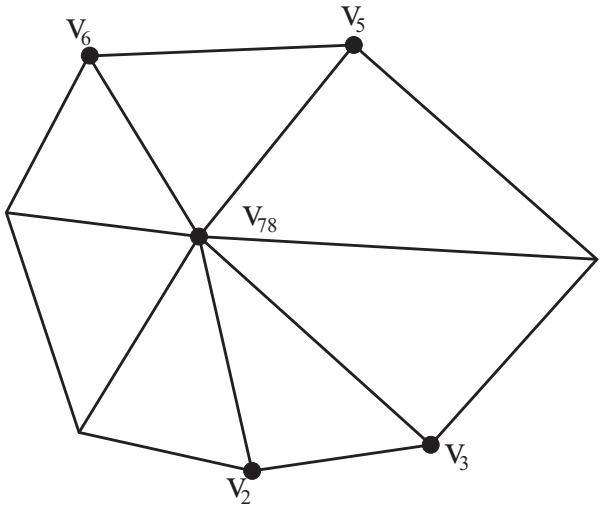
- Poligon egyszerűsítési technikát
  1.) Statikus
    - Elkülönített részletességi szinteket (Level of Detail, röviden LOD) hozunk létre még a renderelés előtt és a megjelenítő ezek közül választ
  2.) Dinamikus
    - Folytonos spektruma az LOD modellek néhány diszkrét modelljével szemben
  3.) Nézőpont-függő
    - Például egy terep renderelésekor a közeli területeket részletesebben, míg a távolabbiakat a távolságtól függően kisebb részletességgel jelenítjük meg

- Az LOD algoritmusoknak 3 fő része van
  - Generálás
    - Egyszerűsítéssel
    - Kézzel
  - Kiválasztás
    - Valamilyen szempont alapján az LOD modell kiválasztása
    - A képernyőn lévő becsült terület alapján például
  - Váltás
    - LOD szintek közötti váltás

- Az egyik módszer a poligonok számának csökkentésére az élek összevonása művelet
- Két vertex egybe olvasztásával lehet elvégezni



A $\mathbf{v}_6\mathbf{v}_7$ és $\mathbf{v}_6\mathbf{v}_8$ illetve a $\mathbf{v}_2\mathbf{v}_7$ és $\mathbf{v}_2\mathbf{v}_8$ élek összevonása



Eltűnik a $\mathbf{v}_7\mathbf{v}_8$ él, a $\triangle\mathbf{v}_6\mathbf{v}_7\mathbf{v}_8$ és $\triangle\mathbf{v}_7\mathbf{v}_2\mathbf{v}_8$ háromszögek

- Az élek összevonása művelet visszafordítható
- Él összevonásokat rendezve, az egyszerűsített modellből kiindulva visszaállíthatjuk az eredeti modellt
- A $\mathbf{v}_7$ és $\mathbf{v}_8$ vertexeket összeolvasztottuk $\mathbf{v}_{78} = \mathbf{v}_7$ vertexbe
  - A másik vertexbe való olvasztás is ($\mathbf{v}_{78} = \mathbf{v}_8$) elfogadható lett volna
- Ha korlátozzuk a lehetőségek számát, akkor értelemszerűen kódolhatjuk az aktuálisan végrehajtott választást

- Az *optimális elhelyezés* eléréshez több lehetőséget kell megvizsgálni
- Ahelyett, hogy az egyik vertexet a másikba olvasztanánk az élen lévő mindkét vertexet egy új pozícióban húzunk össze
  - Jobb minőségű háló jön így létre
  - Hátránya, hogy több műveletet kell végrehajtani és több memóriát is kell felhasználni a nagyobb területen való elhelyezési lehetőségek kiválasztására
- Bizonyos vertex összeolvasztásokat a költségekre való tekintet nélkül el kell kerülni
  - Például konkáv alakzatok esetén a vertex összeolvasztás után egy új él a poligonon kívülre kerül
    - A szomszédos poligonok normálvektorának az iránya megfordul-e az összeolvasztás következtében

- Garland és Heckbert alap célfüggvénye
  - Egy adott vertexhez megadhatjuk azon háromszögek halmazát, amelyeknek eleme ez a vertex és mindegyik háromszöghöz adott a sík egyenlete
  - A célfüggvény egy mozgó vertexre a síkok és az új pozíció távolságainak négyzet összegei

$$c(\mathbf{v}) = \sum_{i=1}^{m} (\mathbf{n}_i \cdot \mathbf{v} + d_i)^2$$

    - A $\mathbf{v}$ az új pozíció
    - Az $\mathbf{n}$ az adott sík normálvektora
    - d az eredeti pozícióhoz viszonyított eltolási értéke

1.) Képzeljünk el két háromszöget, amelyek egy közös éllel rendelkeznek.
   - Ez az él egy nagyon éles él, amely pl. egy turbina lapát része lehet.
   - A célfüggvény értéke a vertex összeolvasztás esetén ebben az esetben alacsony
     - Az egyik háromszögön csúszó pont nem kerül távol a másik háromszög síkjától
   - Egy olyan síkot veszünk hozzá az objektumhoz, amely tartalmazza az élet és az él normálvektora a két háromszög normálisának az átlaga
     - Azoknál a vertexeknél, amelyek nagyon eltávolodnak ettől az éltől, nagyobb költség függvény értéket fogunk kapni

2.) A költség függvény másik fajta kiterjesztése másfajta felületi tulajdonságok megőrzésére szolgál.
- Például a modell gyűrődési és határ élei fontosak a megjelenítésben, ezért kisebb mértékben szabad csak azokat módosítani
- Érdemes más felületi tulajdonság esetén is megőrizni a pozíciókat
  - Változik az anyag
  - Textúra élek vannak és változik a vertexenkénti színek száma

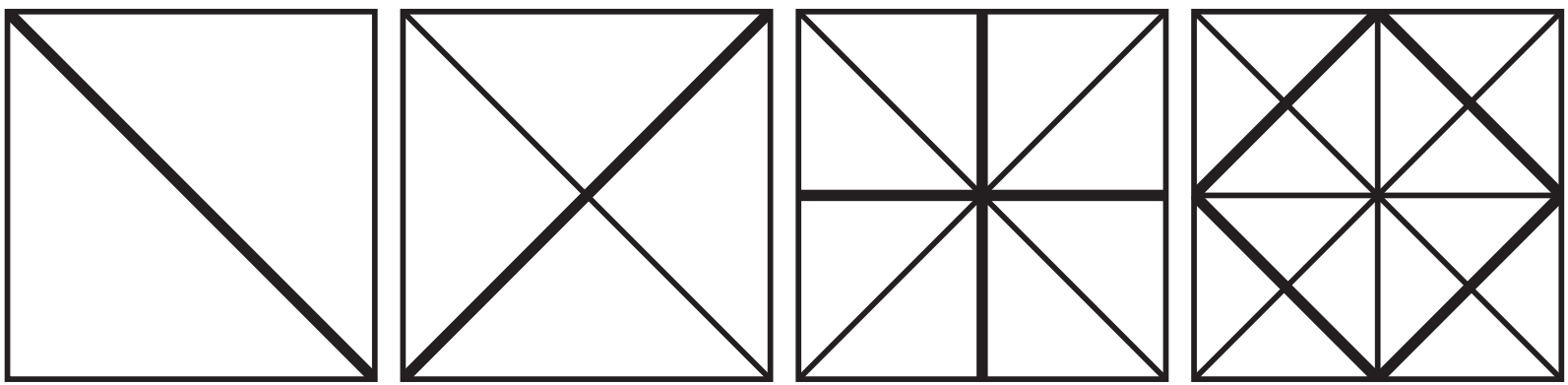
3.) Leginkább azokat az éleket érdemes összevonni, amelyek a legkisebb észlelhető változást eredményezik.
- Kép-vezérelt függvényt használjunk ilyen esetek kezelésére
  - Az eredeti modellből különböző nézetből (mondjuk 20), legyártjuk a képeket
  - Ezután minden potenciális élre kipróbáljuk az összevonásokat az adott modell esetén és előállítjuk a képeket, amelyeket összehasonlítjuk az eredetivel
  - Azt az élet vonjuk össze, amelyik vizuálisan a legkisebb különbséget adja
- Ennek a célfüggvény értékének a kiszámítása igen drága és természetesen nem valósítható meg valós-időben
  - Az egyszerűsítési műveletet el lehet végezni előzetesen, amelyet később fel lehet használni

- Komoly problémát jelent az egyszerűsítési algoritmusoknál az, hogy gyakran a textúrák észrevehetően eltérnek az eredeti megjelenésüktől
  - Ahogy az élek eltűnnek, a felület mögött lévő textúrázási leképezés eltorzulhat
- A poligon csökkentési technikák hasznosak lehetnek
  - Egy tehetséges modellező létrehozhat egy alacsony poligon számú modellt, amely minőségben sokkal jobb, mint az automatikus eljárással előállított
  - A legtöbb redukáló algoritmus nem tud semmit a vizuálisan fontos elemekről vagy a szimmetriáról
    - Például a szemek és az orr a legfontosabb része az arcnak
    - Egy naiv algoritmus elsimítja ezeket a területeket, mivel lényegtelenek

- A terep az egyik olyan modell típus, amelyik egyedi tulajdonságokkal rendelkezik
- Az adatokat rendszerint egyenletes rácson megadott magassági értékekként tárolják el
- A nézőpont-függő módszerek általában valamilyen feltételben megadott kritérium határértékéig folytatják az egyszerűsítést
- Élösszevonás kiegészítve egy célfüggvénnyel, ami a nézőpontot is figyelembe veszi
  - A terepet nem egy egyszerű hálóként kell ilyen esetben kezelni, hanem kisebb részterületekre bontva

- Algoritmusok másik osztálya
  - A magasságmező rácsból származtatott hierarchikus adatstruktúrát használ
  - Egy hierarchikus struktúrát építünk fel az adatok felhasználásával és kiértékeléskor pedig csak a bonyolultság szintjének megfelelően állítjuk elő a terep felszínt
- Hierarchikus struktúra lehet a bináris háromszögfa
  - Egy nagy, jobb háromszöggel kezdődik a magasságmező darab sarkaiban lévő vertexekkel
  - Ez a háromszög felosztható az átfogón lévő középpont és a szemközti sarokpont összekötésével
  - A felosztást addig folytathatjuk, amíg el nem érjük a magasságmező rácsának a részletességét

- Bináris háromszögfa létrehozása
  - A baloldalon a magasságmező két háromszöggel van közelítve
  - A következő szinteken, mindegyik háromszög újból ketté van osztva
  - Az osztásokat a vastag szakaszok jelölik



- Mindegyik háromszöghöz előállítunk egy hibahatárt
  - Ez a hibahatár fejezi ki azt a maximális mennyiséget, amivel a magasságmező eltérhet a háromszögekkel kialakított síktól
  - A hibahatár és a háromszög együtt határozzák meg egy torta-alakú szeletét a térnek
    - Tartalmazza a teljes terepet, amely kapcsolatban áll ezzel a háromszöggel
  - A futás alatt ezeket a hibahatárokat leképezzük a nézősíkra és kiértékeljük a megjelenítésre kifejtett hatásukat

Összefoglalás

Témakörök

- Display Listák
  - Állapot változások
- Vertex tömbök
  - Változó vertex pozíciók
- Vertex pufferek
- Poligon technikák
  - Háromszögsávok és hálók
  - Poligon egyszerűsítési technikák

Realisztikus színtér

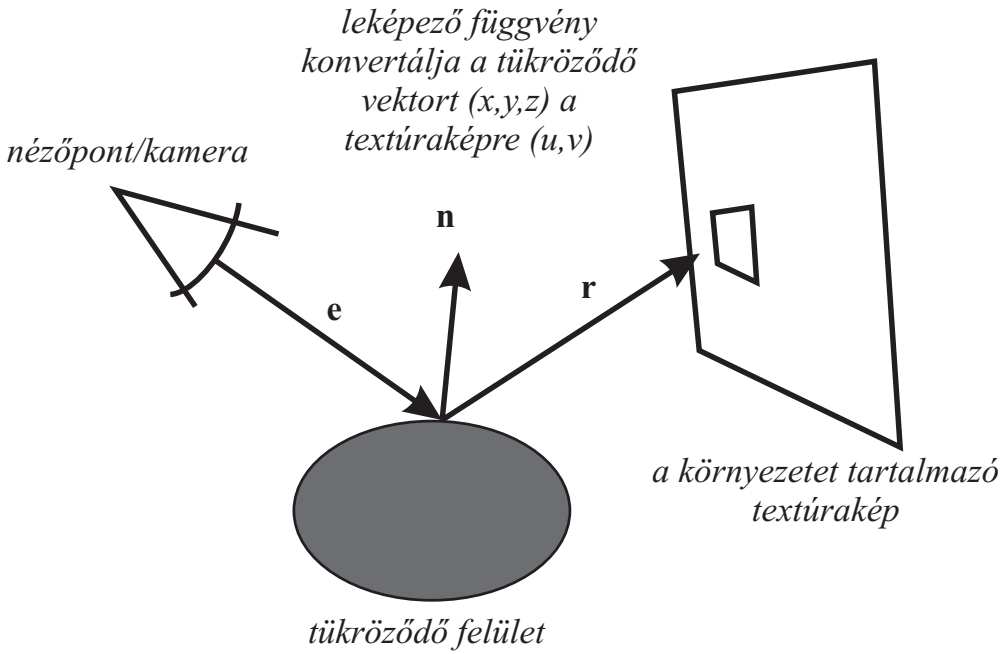
Környezet leképezés

Környezet leképezés

- Hatékony módszer görbe felületeken való tükröződés megjelenítésére
  - Egy sugarat indít a nézőpontból a tükröződő objektum egy pontjába
  - Ez a sugár ezután a pontban lévő normálvektor alapján visszaverődik
    - Ahelyett, hogy megkeresnénk a legközelebbi felülettel való metszését a visszavert fényvektornak az irányát használja a környezetet tartalmazó kép indexének a meghatározására

Környezet leképezés

- A kamera egy objektum felé néz
- Az **r** visszavert fényvektorát az **e** és **n** vektorokból számítjuk ki
- A visszavert fényvektor eléri a környezetet tartalmazó textúraképet
- Az elérési információt a leképező függvény felhasználásával számítjuk ki, amely az (x, y, z) visszatükröződő vektort alakítja át (u, v) értékre



Környezet leképezés

- A környezet leképezés feltételezi,
  - Az objektumok és fények, melyek a felületen tükröződnek, messze vannak
  - A tükröződő felület önmagát nem tükrözi
- A környezet leképezési algoritmus lépései a következőek
  1.) A környezetet ábrázoló kétdimenziós kép előállítása és betöltése
  2.) A tükröződő objektum mindegyik pixelére az objektum felületén lévő pozíciókban kiszámítjuk a normál egységvektorokat
  3.) A visszavert fényvektornak a kiszámítása a nézőpontvektor (nézőpont iránya) és a normál egységvektorból
  4.) A visszavert fényvektor segítségével meghatározzuk a környezeti térkép egy indexét, ami a környezet színe az objektum adott pontjában
  5.) A környezeti térképből kinyert texel adatokat használjuk fel az aktuális pixel színezésére

- Mindegyik leképezett pixelre kiszámítjuk a visszavert fényvektort és (ρ, ϕ) gömbi koordinátákba transzformáljuk azokat
  - A $\phi \in [0, 2\pi]$-t hosszúsági körnek nevezzük
  - $\rho \in [0, \pi]$-t szélességi körnek nevezzük
- (ρ, ϕ)-t a következő összefüggések alapján számítjuk ki

$$\rho = \arccos(-r_z)$$

$$\phi = \arctan(\frac{r_y}{r_x}), \text{ ha } r_x \neq 0$$

  - $\mathbf{r} = (r_x, r_y, r_z)$ a normalizált visszavert fényvektor

- A nézőponthoz tartozó visszavert fényvektort, hasonlóan számítjuk a fény tükröződési vektoréhoz

$$\mathbf{r} = \mathbf{e} - 2(\mathbf{n} \cdot \mathbf{e})\mathbf{n}$$

  - $\mathbf{e}$ a normalizált vektor a felület pozícióban
  - $\mathbf{n}$ az egység normálvektor az adott pozícióban

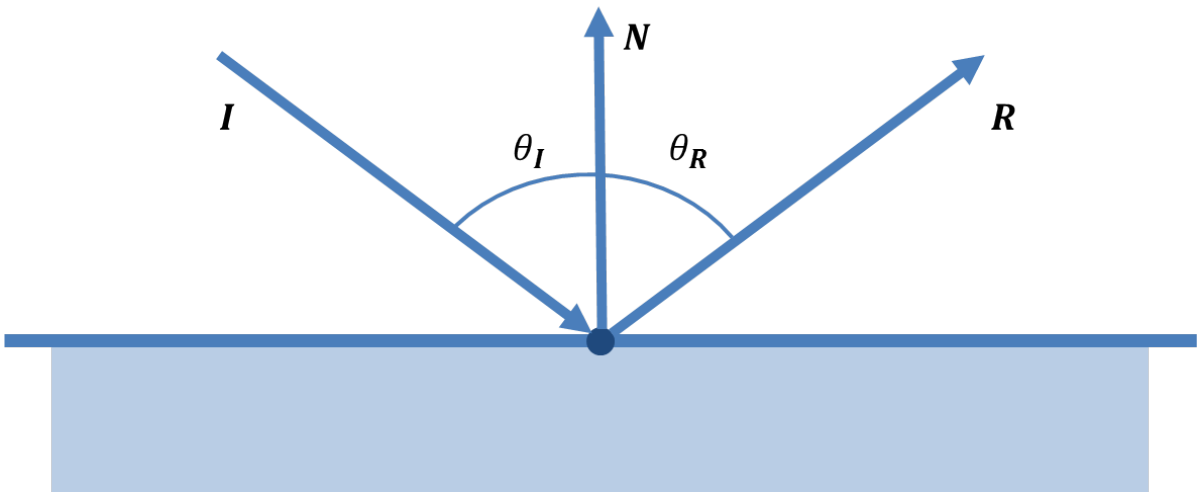
- A (ρ, ϕ) gömbi koordinátákat a $[0, 1)$ tartományra képezzük le
- (u, v) koordinátaként használjuk a környezet textúra eléréséhez, a tükröződő szín előállítására
- A tükröződési vektort transzformáljuk gömbi koordinátákba
  - A környezetet tartalmazó textúrakép egy „kiterített" gömb képe
- A textúra befed egy gömböt, ami körbeveszi a tükröződési pontot
- Ezt a leképező függvényt néha szélességi-hosszúsági leképezésnek is hívják
  - v a szélességi körökkel, u pedig a hosszúsági körökkel egyezik meg

- Hátrányok
  - $\phi = 0$-ban van egy határ
  - A térkép összefut a sarkoknál
- A környezet leképezésben használt képnek egyeznie kell a szegélyeknél a függőleges élek mentén
- El kell kerülni a torzítási problémákat a felső és alsó élek környezetében
- Használhatjuk az indexek kiszámítására a vertexekben és ezután interpolálhatjuk ezeket a koordinátákat
- Hiba fordul elő abban az esetben is, amikor egy háromszög vertexei olyan indexekkel rendelkeznek a környezeti térképen, melyek a sarkokon mennek keresztül

- A kamerát egy kocka középpontjában helyezzük el és levetítjük a környezetet a kocka oldalaira
- A környezeti térképet bármely renderelővel könnyen elő lehet állítani valós-időben
- A visszavert fényvektornak az iránya meghatározza, hogy a kocka melyik oldalát használjuk
  - A visszavert fényvektor abszolút értékben legnagyobb komponense meghatározza, hogy milyen kapcsolatban van az oldallal
    - Például a $(-3.2, 5.1, -8.4)$ a $-Z$ oldalt jelöli ki
  - A maradék két komponenst a legnagyobb komponens abszolút értékével elosztva, majd a $[0, 1]$ intervallumra leképezve kapjuk meg a textúra-koordinátákat a kiválasztott lapon

- Tükrözött vektorok kiszámítása
  - Beeső sugár I
  - Tükrözött sugár R
  - Felületi normál N
  - Tökéletes tükröződésnél $\theta_I = \theta_R$
  - $R = I - 2N(N \cdot I)$

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



- Konvex vagy majdnem konvex objektumok esetén működik jól
- Csak az iránytól függ nem pedig a pozíciótól
  - Sík tükröződő felületek esetén nem jól viselkedik
  - Legjobban a görbefelületeken alkalmazható

Cg - Vertex program

```
void C7E1v_reflection(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,
                      float3 normal   : NORMAL,
                  out float4 oPosition  : POSITION,
                  out float2 oTexCoord  : TEXCOORD0,
                  out float3 R          : TEXCOORD1,
              uniform float3   eyePositionW,
              uniform float4x4 modelViewProj,
              uniform float4x4 modelToWorld)
{
  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;
  // A világtérben való pozíció és normálvektor kiszámítása
  float3 positionW = mul(modelToWorld, position).xyz;
  float3 N = mul((float3x3)modelToWorld, normal);
  N = normalize(N);
  // A beeső és tükröződő vektorok kiszámítása
  float3 I = positionW - eyePositionW;
  R = reflect(I, N);
}
```

Cg - Fragmens program

```
void C7E2f_reflection(float2 texCoord : TEXCOORD0,
                      float3 R        : TEXCOORD1,

                  out float4 color : COLOR,

                uniform float reflectivity,
                uniform sampler2D decalMap,
                uniform samplerCUBE environmentMap)
{
  // A tükröződő környezeti szín meghatározása
  float4 reflectedColor = texCUBE(environmentMap, R);

  // A matrica textúra szín meghatározása
  float4 decalColor = tex2D(decalMap, texCoord);

  color = lerp(decalColor, reflectedColor, reflectivity);
}
```

- A tükröződő vektor kiszámításához használhatnánk fragmens programot
  - Jobb képminőség érhető el
  - Spelkuláris megvilágításnál a tükröződési vektor nem-lineáris módon változik fragmensről-fragmensre
  - A lineárisan interpolált vertexenkénti tükröződési értékek nem megfelelőek
  - Az objektum körvonalánál artifaktumok jelennek meg
- Nem biztos, hogy észrevehető a különbség

- A textúraképet egy tökéletesen tükröződő gömbön megjelenő környezet ortogonális nézetéből állítjuk elő
  - A textúrát gömbtérképnek nevezzük
- Előállítása
  - Egy csillogó gömbről készítünk fényképet
    - Ezt az kör alakú eredmény gömbtérképet néha fényvizsgálatnak is hívják
  - A gömbtérképet szintetikus színtér esetén való előállítása
    - Sugárkövetéssel
    - A cube map környezeti térképnél használt képek gömbre való vetítésével

- A gömbtérképnek van egy bázisa
- A képet egy **f** tengely mentén nézzük a világtérben **u** felfele mutató vektorral és feltesszük, hogy a **h** vektor vízszintesen jobbra mutat (mindegyik vektor normalizált)
- Ez egy bázis mátrixot ad

$$\begin{pmatrix} h_x & h_y & h_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- A gömbtérkép egy elemének eléréséhez
  - Az $\mathbf{n}$ felületi normált és a szem pozíciójából a vertexbe menő $\mathbf{e}$ vektort transzformáljuk
  - Ez az $\mathbf{n}'$ és $\mathbf{e}'$ vektorokat állítja elő a gömbtérkép terében
- A visszavert fényvektort a következőképpen állítjuk elő

$$\mathbf{r} = \mathbf{e}' - 2(\mathbf{n}' \cdot \mathbf{e}')\mathbf{n}'$$

  - Ahol az $\mathbf{r}$ eredmény vektor a gömbtérkép terében van

- A tükröződő gömb a teljes környezetet mutatja meg, ami a gömb előtt található
  - Ez mindegyik visszavert irányt leképezi a gömb kétdimenziós képének egy pontjára
- Ha meg akarjuk határozni a tükröződési irányt a gömbtérkép egy adott pontjában
  - Szükségünk van a gömb pontjában a felületi normálvektorra

- Fordítsuk meg az eljárást és vegyük a gömbön a pozíciót
  - Vezessük le a felületi normált a gömbön, ami az (u, v) paramétereket határozza meg a textúra adatok eléréséhez
- A gömb normálvektora (r_x, r_y, r_z) a visszavert fényvektor és a szem iránya $(0, 0, 1)$ között fél úton található
  - Az $\mathbf{n}$ normálvektor egyszerűen felírható a szem és a visszavert fényvektor összegeként, amelyet ezután normalizálunk

$$m = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2},$$

$$\mathbf{n} = \left(\frac{r_x}{m}, \frac{r_y}{m}, \frac{r_z + 1}{m} \right)$$

- Hátrány
  - A gömbtérképen két pont közötti mozgás nem lineáris
  - A gömbtérkép csak egyetlen nézőpont irány esetén érvényes
    - Ha változik a nézőpont iránya, akkor a leképezést újra végre kell hajtani
  - Mivel a gömbtérkép nem tartalmazza a teljes környezetet
    - Előfordulhat, hogy képkockáról-képkockára ki kell számolni a környezeti leképezés textrúra-koordinátáit az új nézőpont irányra az alkalmazás szakaszban
  - Amennyiben a nézőpont iránya változik, akkor érdemesebb nézőpont független környezeti leképezést használni

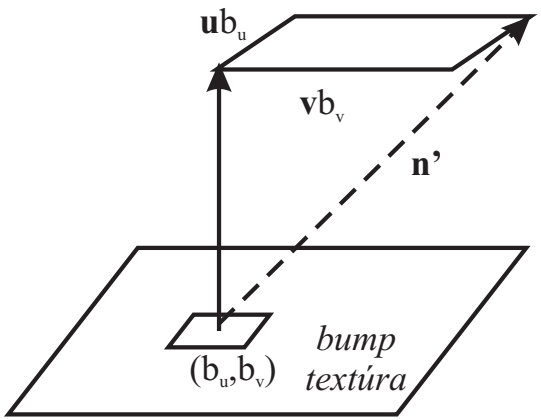
Felületi egyenetlenség leképezés - Bump mapping

Felületi egyenetlenség leképezés - Bump mapping

- A felületek megjelenését teszi egyenetlenné
  - Olyan tulajdonságot szimulálhat, amit ellenkező esetben sok poligon felhasználásával lehet csak modellezni
- Az alap ötlet az, hogy a textúra nem a szín komponenst változtatja meg a megvilágítási egyenletben, hanem a felületi normálvektorokat módosítja
  - A normálvektorokat egy textúrában tárolunk el
- A felületi geometria normálja változatlan marad
  - A megvilágítási egyenletben használt normálvektorokat változtatjuk meg pixelenként

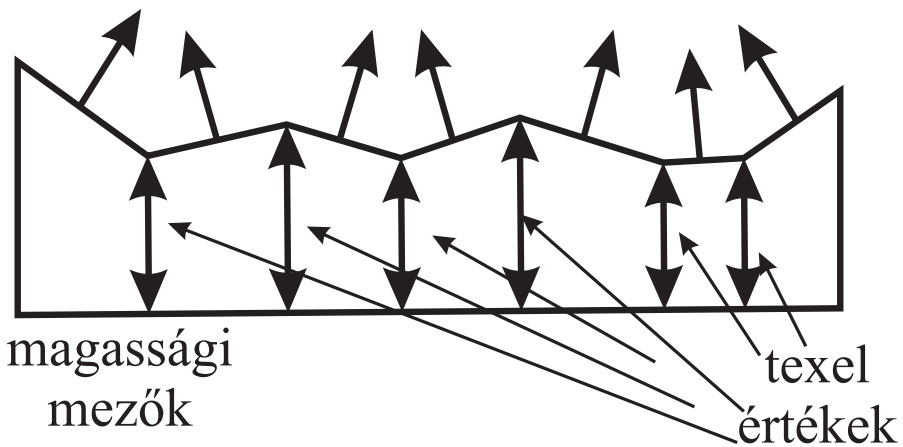
Felületi egyenetlenség leképezés - Bump mapping

- Az egyik felületi egyenetlenség textúrázási technika esetén b_u és b_v előjeles értékeket tárolnak el egy textúrában
- Ez a két érték a normál változásának a mennyiségét tárolja u és v tengelyek mentén
- Ezeket a textúra értékeket használjuk a normálisra merőleges két vektor skálázására
  - A textúra értékek általában bilineárisan interpoláltak
- A két b_u és b_v adja meg, hogy a felület milyen irányba néz a pontban



Felületi egyenetlenség leképezés - Bump mapping

- Magassági mezőket használunk a felületi normálvektorok irányainak a módosítására
- A szomszédos oszlopok különbsége adja meg az u, valamint a szomszédos sorok különbsége a v meredekségét



Felületi egyenetlenség leképezés - Bump mapping

- Meggyőző és olcsó módja a geometriai részletesség látszatának növelésére
- Hátrány
  - Az objektumok körvonalai körül azonban a hatás eltűnik
  - A felületi egyenetlenség alkalmazásakor az egyenetlenségek nem vetnek árnyékot a saját felületükön

Felületi egyenetlenség leképezés - Bump mapping

- Léteznek fejlettebb valósidejű renderelő módszerek, melyek használatával önárnyalási hatást is el lehet érni
  - Statikus színterek esetén a megvilágítást előre is ki lehet/kell számítani
  - Ha egy felületen nincs spekuláris megvilágítás és a fények nem mozognak a felülethez viszonyítva
    - A felületi egyenetlenséghez tartozó árnyalást ki lehet számítani egyszer és az eredményt egy szín textúraként használjuk az adott felületen
  - Ha a felület, fény és kamera mindegyike rögzítve vannak egymáshoz
    - A fényes, egyenetlen felületet elegendő egyszer előállítani

Cg - Vertex program

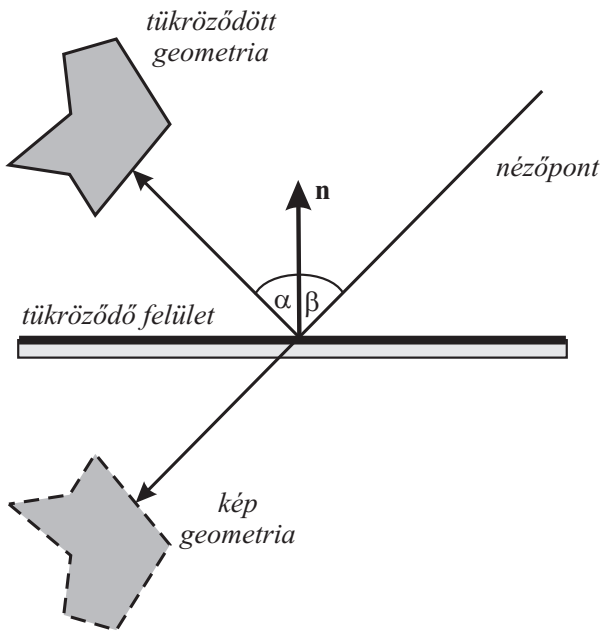
```
void C8E1v_bumpWall(float4 position : POSITION,
                    float2 texCoord : TEXCOORD0,

                    out float4 oPosition      : POSITION,
                    out float2 oTexCoord      : TEXCOORD0,
                    out float3 lightDirection : TEXCOORD1,

                uniform float3   lightPosition,  // Objektumtér
                uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;
  // Különbség vektorok az objektumtér
  // megvilágítási irányhoz
  lightDirection = lightPosition - position.xyz;
}
```

Cg - Fragmens program

```
float3 expand(float3 v)
{
  return (v-0.5)*2;  // Kiterjesztése a vektornak
}

void C8E2f_bumpSurf(float2 normalMapTexCoord : TEXCOORD0,
                    float3 lightDir          : TEXCOORD1,

                out float4 color : COLOR,

            uniform sampler2D   normalMap,
            uniform samplerCUBE normalizeCube)
{
  // Normalizes light vector with normalization cube map
  float3 lightTex = texCUBE(normalizeCube, lightDir).xyz;
  float3 light = expand(lightTex);

  // A normál map textúra mintavételezése és kiterjesztése
  float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  float3 normal = expand(normalTex);

  // Diffúz megvilágítás
  color = dot(normal,light);
}
```

Tükröződések

- Sík tükröződést könnyebb megvalósítani és végrehajtani, mint egy általános tükröződést
- Ideális tükröződő felületre érvényes a *tükröződési törvény*, amely szerint a beesési szög megegyezik a visszavert fény kilépési szögével
- Ennek a törvénynek köszönhetően az objektum tükrözött képe egyszerűen maga a tükrözött objektum



- A visszatükrözött sugár követése helyett a beeső fényt követhetjük a tükröződő felületen
  - Egy tükröződést előállíthatunk egy objektum másolatának a transzformálásával a tükröződő pozícióba
  - A pozíció és az irány figyelembevételével a fényforrásokat is tükrözni kell
- Ha feltesszük, hogy a tükröződő felület $\mathbf{n}$ normálvektora $(0, 1, 0)$ és ez az origón megy keresztül
  - A mátrix, ami erre a síkra tükröz egy egyszerű tükröző $\mathbf{S}(1, -1, 1)$ skálázó mátrix

- Általános esetben az **M** tükröződési mátrix
  - **n** normálvektor
  - A tükröződő felület **p** pontja

$$\mathbf{F} = \mathbf{R}(\mathbf{n}, (0,1,0))\mathbf{T}(-\mathbf{p})$$

  - A síkot eltoljuk a $\mathbf{T}(-\mathbf{p})$ transzformációval úgy, hogy az origón keresztül menjen
  - A tükröződő felület **n** normálvektorát forgatjuk, hogy párhuzamos legyen az $(0,1,0)$ y-tengellyel
    - A forgatást az $\mathbf{R}(\mathbf{n}, (0,1,0)$ felhasználásával hajtjuk végre
  - A tükröződő felület ezek után az $y = 0$ síkhoz lesz igazítva

$$\mathbf{M} = \mathbf{F}^{-1}\mathbf{S}(1,-1,1)\mathbf{F}$$

- A mátrixot újra kell számolni, ha a pozíció vagy a tükröződő felület irányítottsága megváltozik

- Először a megjelenítendő színtér **M**-mel transzformált tükröződő objektumait
- A színtér többi részét rajzoljuk ki a tükröződő felülettel együtt
- A tükröződő felületnek részlegesen átlátszónak kell lenni
  - Azért, hogy a tükröződés látható legyen
- Amennyiben éles szögben nézünk rá az adott színtérre, akkor a tükröződő geometria láthatóvá válhat
  - A „kilógó ” rész eldobásával ez a probléma megoldható
  - A legalkalmasabb ennek a problémának a megoldására a *stencil puffer* használata

OpenGl példa

```
// A szín és mélység puffer frissítésének letiltása
glDisable(GL_DEPTH_TEST);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// A stencil műveletek beállítása
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);

// Puffer 1-esekkel való feltöltése ott ahol a padló van.
drawFloor();

// Újra engedélyezése a szín és mélység puffernek.
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glEnable(GL_DEPTH_TEST);
```

OpenGl példa

```
// Csak oda rajzolunk, ahol 1-esek vannak
glStencilFunc(GL_EQUAL, 1, 0xffffffff);

// Az 1-esek maradnak a pufferben.
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

// A tükrözött dinó kirajzolása, ahol 1-esek vannak.
glPushMatrix();
glScalef(1.0, -1.0, 1.0);
setLightSourcePositions();
drawNinja();
glPopMatrix();
glDisable(GL_STENCIL_TEST);
```

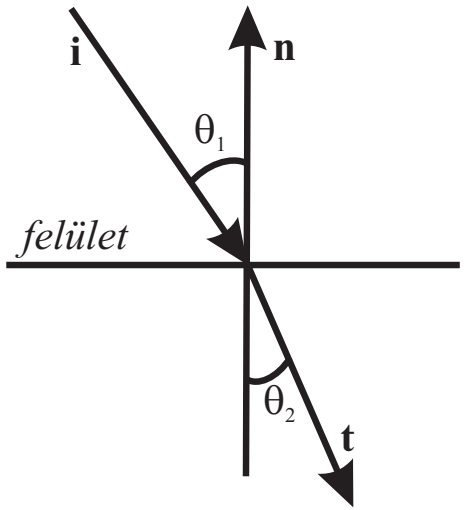
OpenGl példa

```
// Összemosott padló és az aktuális objektum kirajzolása

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor4f(0.7, 0.0, 0.0, 0.40);
drawFloor();
glDisable(GL_BLEND);
drawNinja();
```

- Egy másik probléma a hátsólap-eldobás miatt fordul elő
- Amennyiben a hátsólap-eldobás be van kapcsolva és egy objektumot skálázunk a tükröződési mátrixszal
  - A hátsólap-eldobás helyett az előlapok lesznek eldobva
- A megoldás az, hogy a hátsólap-eldobásból az előlap-eldobásra váltunk

- Snell törvénye
  - A beérkező és kilépő vektorok kapcsolatát állapítja meg
  - Az egyik közegből (mint például levegő) a másikba (például a víz) lép a fény

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$



- n_k az adott közeg törésmutatója
- θ_k ($k \in \{1, 2\}$) a felületi normálishoz viszonyított szög

$$\mathbf{t} = r\mathbf{i} + (w - k)\mathbf{n}$$
$$r = n_1 / n_2,$$
$$w = -(\mathbf{i} \cdot \mathbf{n})r,$$
$$k = \sqrt{1 + (w - r)(w + r)}$$

- A kiértékelés eléggé költséges
- A fénytörés mértéke a horizont környékén csökken
  - Kis bejövő szögek esetén a következő közelítést használhatjuk

$$\mathbf{t} = -c\mathbf{n} + \mathbf{i}$$

  - c víz szimulálása esetén 1.0 körül van
  - Ebben az esetben $\mathbf{t}$ vektort normalizálni kell

Cg példa - Vertex program

```
float3 refract (float3 I, float3 N, float etaRatio)
{
 float cosI = dot(-I, N);
 float cosT2 = 1.0f - etaRatio * etaRatio *
            (1.0f - cosI * cosI);
 float3 T = etaRatio * I +
          ((etaRatio * cosI - sqrt(abs(cosT2))) * N);
     return T * (float3)(cosT2 > 0);
}
```

Cg példa - Vertex program

```
void C7E3v_refraction(float4 position : POSITION,
                      float2 texCoord : TEXCOORD0,
                      float3 normal   : NORMAL,
                  out float4 oPosition  : POSITION,
                  out float2 oTexCoord  : TEXCOORD0,
                  out float3 T          : TEXCOORD1,
              uniform float etaRatio,
              uniform float3 eyePositionW,
              uniform float4x4 modelViewProj,
              uniform float4x4 modelToWorld)
{
  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;

  // A pozíció és a normál kiszámítása a világtérben
  float3 positionW = mul(modelToWorld, position).xyz;
  float3 N = mul((float3x3)modelToWorld, normal);
  N = normalize(N);

  // A beeső és a kilépő vektorok kiszámítása
  float3 I = normalize(positionW - eyePositionW);
  T = refract(I, N, etaRatio);
}
```

Cg példa - Fragmens program

```
void C7E4f_refraction(float2 texCoord : TEXCOORD0,
                      float3 T        : TEXCOORD1,

                  out float4 color : COLOR,

              uniform float        transmittance,
              uniform sampler2D    decalMap,
              uniform samplerCUBE  environmentMap)
{
  // Textúra szín betöltése
  float4 decalColor = tex2D(decalMap, texCoord);

  // A fénytörés által meghatározott szín
  float4 refractedColor = texCUBE(environmentMap, T);

  //A végső szín kiszámítása
  color = lerp(decalColor, refractedColor, transmittance);
}
```

- A Fresnel egyenletek azt írják le,
  - Mennyi fény verődik vissza és mennyi fény „törik” meg
- Alacsony szög esetén nagy a tükröződés és nincs/alig van fénytörés
  - Nem lehet látni mi van a víz alatt
- Realisztikusabb lesz az előállított kép
- A Fresnel egyenletek bonyolultak
- Empirikus közelítés
  - $reflectionCoefficient = max(0, min(1, bias + scale \times (0 + I \cdot N)^{power}))$

- A fénytörés egyszerűsítve volt
  - Függ a felszín normál vektorától
  - Függ a beesési szögtől
  - Függ a fénytörési hányadostól
- Fénytörés mértéke függ még a bejövő fény hullámhosszától
  - Például a vörös fény jobban elhajlik, mint a kék
  - Szimulálhatjuk, hogy mi történik a komponensekkel
    - Az adott komponensek sugaraihoz tartozó környezeti térkép értékét kell kikeresni
    - A fénytörési hányadosokat külön adjuk meg a vörös, zöld és kék komponensekre

Cg példa - Vertex program

```
void C7E5v_dispersion(float4 position : POSITION,
                      float3 normal   : NORMAL,
                  out float4 oPosition         : POSITION,
                  out float  reflectionFactor  : COLOR,
                  out float3 R                 : TEXCOORD0,
                  out float3 TRed              : TEXCOORD1,
                  out float3 TGreen            : TEXCOORD2,
                  out float3 TBlue             : TEXCOORD3,
              uniform float fresnelBias,
              uniform float fresnelScale,
              uniform float fresnelPower,
              uniform float3 etaRatio,
              uniform float3 eyePositionW,
              uniform float4x4 modelViewProj,
              uniform float4x4 modelToWorld)  {
  oPosition = mul(modelViewProj, position);
  // A pozíció és a normál kiszámítása a világtérben
  float3 positionW = mul(modelToWorld, position).xyz;
  float3 N = mul((float3x3)modelToWorld, normal); N = normalize(N);
  // A beeső, a visszavert és a kilépő vektorok kiszámítása
  float3 I = positionW - eyePositionW;
  R = reflect(I, N);  I = normalize(I);
  TRed   = refract(I, N, etaRatio.x);
  TGreen = refract(I, N, etaRatio.y);
  TBlue  = refract(I, N, etaRatio.z);
  // A tükröződési faktor kiszámítása
  reflectionFactor = fresnelBias + fresnelScale * pow(1 + dot(I, N),
                     fresnelPower ); }
```

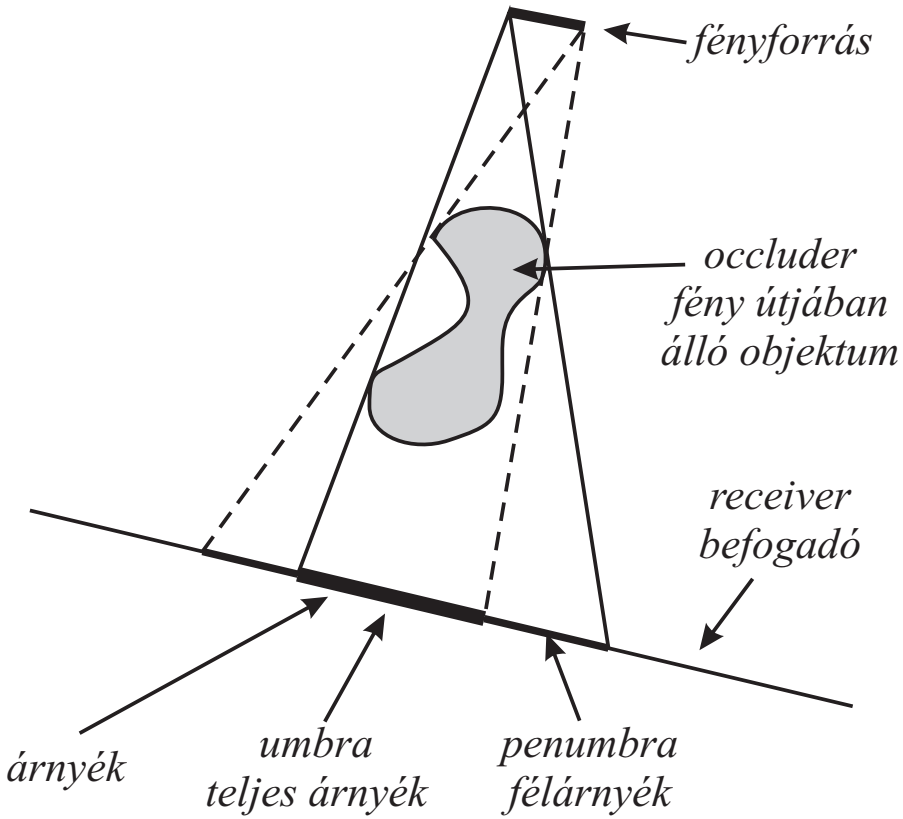
Cg példa - Fragmens program

```
void C7E6f_dispersion(float  reflectionFactor : COLOR,
                      float3 R                : TEXCOORD0,
                      float3 TRed             : TEXCOORD1,
                      float3 TGreen           : TEXCOORD2,
                      float3 TBlue            : TEXCOORD3,
                  out float4 color : COLOR,
              uniform samplerCUBE environmentMap0,
              uniform samplerCUBE environmentMap1,
              uniform samplerCUBE environmentMap2,
              uniform samplerCUBE environmentMap3)
{
  // A tükröződött szín betöltése
  float4 reflectedColor = texCUBE(environmentMap0, R);
  // A fénytörés színének kiszámítása
  float4 refractedColor;
  refractedColor.x = texCUBE(environmentMap1, TRed).x;
  refractedColor.y = texCUBE(environmentMap2, TGreen).y;
  refractedColor.z = texCUBE(environmentMap3, TBlue).z;
  refractedColor.w = 1;
  // A végső szín meghatározása
  color = lerp(refractedColor,
               reflectedColor,
               reflectionFactor);
}
```

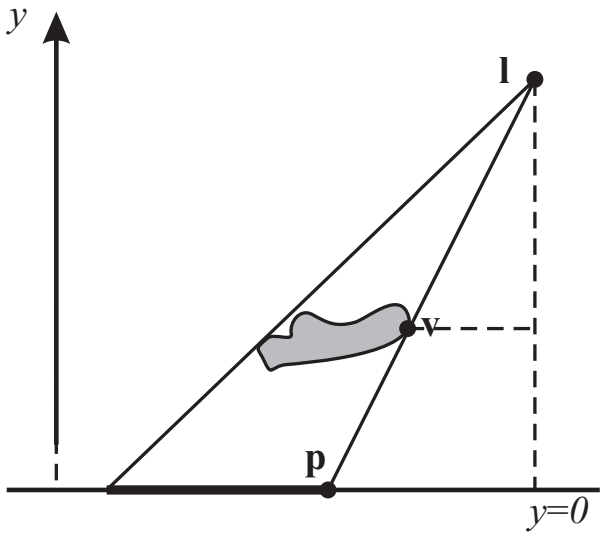
Árnyék síkfelületen

Árnyék síkfelületen

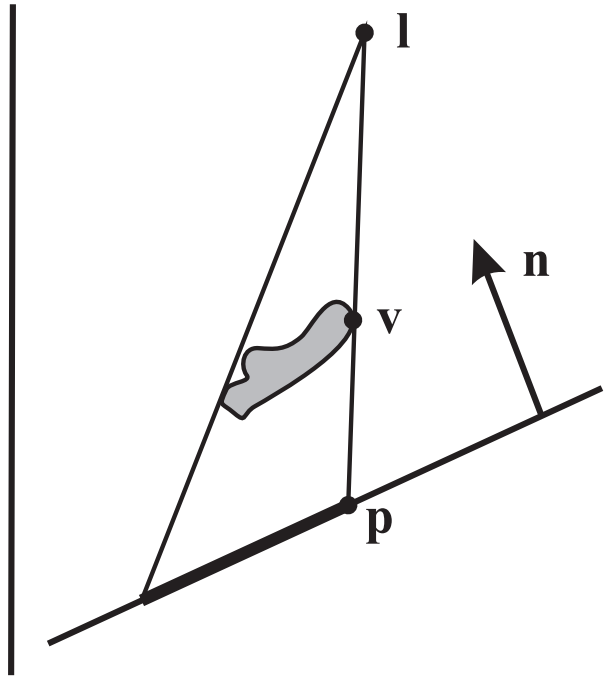
- Az árnyékok fontos elemei a valósághű képek előállításánál
  - A felhasználónak adnak némi információt az objektumok elhelyezéséről



- Egy egyszerű esete az árnyalásnak
  - Az objektumok árnyékai egy sík felületen jelennek meg
- Egy mátrixot hozunk létre
  - Az objektum vertexeit vetíti le egy síkra
  - Az árnyék előállításakor a háromdimenziós objektumot kétszer jelenítjük meg



$y = 0$ síkra vetett árnyék.



$\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ síkra vetett árnyék.

- Az x koordináták vetítésével kezdjük a levezetést
  - Hasonló háromszögek alapján a következő egyenlőségeket írhatjuk fel
    - $\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y} \Longleftrightarrow p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$
- A z koordinátát hasonlóan kapjuk
  - $p_z = (l_y v_z - l_z v_y)/(l_y - v_y)$
- y koordináta 0-val egyenlő
- $\mathbf{M}$ projekciós mátrixot a következőképpen írhatjuk fel

$$\mathbf{M} = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$

- Általános esetben a sík egyenlete, amelyikre az árnyék vetődni fog
  - $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$
  - $\mathbf{v}$ pontot vetíti le $\mathbf{p}$ pontba

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

- Mátrix alakba felírva

$$\mathbf{M} = \begin{pmatrix} \mathbf{n} \cdot \mathbf{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{n} \cdot \mathbf{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{n} \cdot \mathbf{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{l} \end{pmatrix}$$

- Az általános mátrixba behelyettesítve
  - Az $y = 0$ síkhoz tartozó speciális értékeket
    - $\mathbf{n} = (0, 1, 0)^T$
    - $d = 0$
  - A speciális $\mathbf{M}$ mátrixot kapjuk
- Az árnyék előállításához egyszerűen ezt a mátrixot kell alkalmaznunk az objektumokra
  - A π síkra vetnek árnyékot
- Sötét színnel és megvilágítás nélkül kell megjeleníteni a vetületeket
  - A fény útjában álló objektumot kétszer rendereljük
    - Először a vetített poligonokat árnyékként
    - Másodszor pedig az eredeti objektumként

- Szükség van egy olyan módszerre, amely megakadályozza azt, hogy a vetített poligonokat a befogadó felület mögött állítsuk elő
  - A talajt rajzoljuk ki először
  - Aztán a vetített poligonokat kikapcsolt Z-puffer ellenőrzéssel
  - Azután az összes többi geometriát
- A vetített árnyék a síkon kívül is megjelenhet
  - A megoldása is hasonló az ott alkalmazott módszerrel, el kell távolítani a kilógó részt

OpenGL példa

```
void RenderScene(void)
  {
    // A szín puffer és a mélység puffer törlése
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // A sík felület (talaj) kirajzolása
    DrawGround();

    glPushMatrix();

    // A jet kirajzolása az új pozícióban
    // a fényforrás megfelelő pozícióba helyezése
    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    DrawJet(FALSE);

    glPopMatrix();
```

OpenGL példa

```
void RenderScene(void)
    {
...

    // Az árnyék rajzolása a talajon
    // A mélység ellenőrzés és a fény számítások tiltása
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glPushMatrix();

    // Az árnyék vetítési mátrixszal való szorzás
    glMultMatrixf((GLfloat *)shadowMat);

    // Az árnyék beforgatása
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Árnyék rajzolása
    DrawJet(TRUE);

    glPopMatrix();
```

OpenGL példa

```
void RenderScene(void)
    {
...

    // A fényforrás kirajzolása
    glPushMatrix();
    glTranslatef(lightPos[0],lightPos[1], lightPos[2]);
    glColor3ub(255,255,0);
    glutSolidSphere(5.0f,10,10);
    glPopMatrix();

    // A mélység teszt engedélyezése
    glEnable(GL_DEPTH_TEST);

    // Az eredmény megjelenítése
    glutSwapBuffers();
    }
```

- Hátrány
  - Csak sík felületek esetén működik
  - Az árnyékot mindegyik képkocka esetén elő kell állítani
    - Még akkor is, ha az árnyék nem változik
- Egy jól működő módszer az, amikor az árnyékot egy textúrában állítjuk elő
  - Egy textúrázott téglalapként jelenítünk meg
  - Csak akkor kell újra kiszámítani, ha az árnyék megváltozik
    - A fényforrás vagy a fény útjában álló objektum vagy a befogadó felület mozog

Összefoglalás

- Környezeti leképezés
  - Blinn és Newell módszere
  - Cube map környezet leképezés
  - Sphere map környezet leképezés
- Felületi egyenetlenség
- Tükröződések
  - Sík tükröződés
  - Fénytörés
    - Snell törvénye
    - Fresnel hatás
    - Kromatikus szóródás
- Árnyék sík felületen
  - Vetített árnyék

Irodalomjegyzék

Irodalomjegyzék

Tomas Akenine-Moller and Eric Haines.

Real-Time Rendering (2nd Edition).

A K Peters/CRC Press, July 2002.

Randima Fernando and Mark J. Kilgard.

The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.

Addison-Wesley Professional, March 2003.

Eric Lengyel.

Mathematics for 3D Game Programming and Computer Graphics, Second Edition, chapter Transforming Normal Vectors.

CHARLES RIVER MEDIA, 2004.

Richard S. Wright, Jr. Benjamin Lipchak, and Nicholas Haemel.

OpenGL SUPERBIBLE, Fourth Edition, Comprehensive Tutorial and Reference.

Addison-Wesley Professional, June 2007.