Számítógépes Grafika

Klár Gergely
tremere@elte.hu

Eötvös Loránd Tudományegyetem
Informatikai Kar

2010/2011. tavaszi félév

1. Generáció

- NVIDIA TNT2, ATI Rage, 3dfx Voodoo3
- A standard 2d-s videokártyák kiegészítése
- Csúcspont transzformációkat még a CPU csinálja!
- A kártya csak a textúrázást, Z-buffer kezelést végezte

2. Generáció (1999-2000)

- NVIDIA GeForce 256, GeForce 2, ATI Radeon 7500
- Átveszik a transzformációk és az árnyalás kezelését a CPU-tól.
- Az OpenGL és DirectX 7 is támogatja a hardveres csúcspont transzformációkat
- Multi-textúrázás megjelenése: bump map, light map
- Konfigurálható (driver szinten), de még nem programozható

3. Generáció (2001)

- NVIDIA GeForce 3, GeForce 4 Ti, Xbox, ATI Radeon 8500
- A csúcspont pipeline korlátozott programozhatósága
- Fejlettebb pixel szintű konfigurálás, de még nem programozás
- 3d-s textúrák, többszörös mintavételezés (antialias-hoz)

4. Generáció (2002)

- NVIDIA GeForce FX, ATI Radeon 9700
- A csúcspont és pixel pipeline teljesen programozható (erőforrás-korlátok azért még vannak)
- Magas szintű árnyaló nyelvek (shading languages) megjelenése (NVIDIA Cg, Microsoft HLSL, OpenGL GLSL)
- Shader Model 2.0 (simple branching)

5. Generáció (2004)

- NVIDIA GeForce 6, ATI Radeon X, GeForce 7
- Több puffer szimultán renderelése
- 64bites pipeline
- PCIe busz
- Több memória, hosszabb csúcspont árnyaló programok
- Shader Model 3.0 (branching and looping in the pixel shader (physics))
- HDRI, SLI, TSAA, TMAA

6. Generáció (2007)

- DirectX 10
- Shader Model 4.0 (Unified Shader Model, geometry shader)
- Unified Shading Architecture
- Shading performance 2x pixel, 12x vertex above G71
- 700 Mtransistors
- 130W to 300W

7. Generáció (2009-)

- DirectX 11, OpenGL 4.1
- Shader Model 5.0 (Compute Shader, Tesselation Shaders: Hull Shader & Domain Shader)
- Többszállúság
- *Dynamic Shader Linkage*: OOP jellegű szolgátatások HLSL-ben
- 3000+ Mtransistors
- 1000 GFLOPs

SM3.0 – Grafikus szerelőszalag

- Vertex Shader
- Primitive Assembly
- Rasterisation
- Fragment Shader
- Pixel Rendering

Vertex Shader

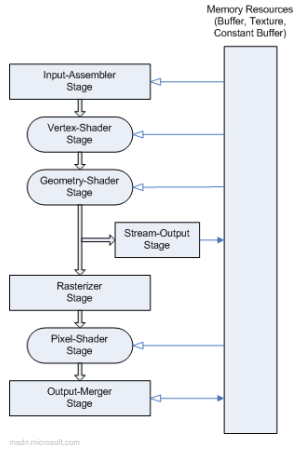
- Feladata: csúcspontok modell k.r. → normalizált eszköz k.r.
- A *vertex buffer* minden elemén lefut
- Bemenet: csúcsok modell k.r.-ben + egyéb
  - legalább a csúcspont pozíciója
  - bármilyen egyéb adat, ami a *vertex buffer*-ben meg van adva
- Kimenet: csúcs normalizált eszköz k.r.-ben + egyéb
  - legalább a csúcspont pozíciója
  - bármilyen egyéb adat

Pixel Shader

- Másik neve: *fragment shader*
- *Fragment*: pixel méretű 3Ds darabka
- *PS* feladata: az egyes *fragment*-ek színének meghatározása
- A raszterizált háromszögek minden egyes *fragment*-jére lefut.
- Az adott *fragment*-et *eldobhatja*.
- Bemenete: A vertex shader kimenő paramétereinek interpoláltjai.
- Kimenete: Az adott *fragment*-hez rendelendő szín.

SM4 – Grafikus szerelőszalag

- Input Assambler
- Vertex Shader
- Geometry Shader
- Stream Output
- Rasterizer
- Pixel Shader
- Output Merger



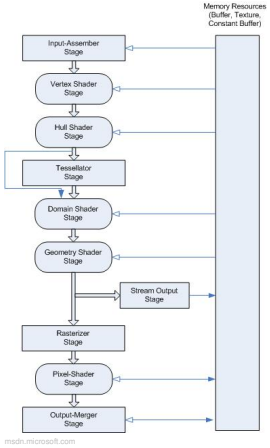
Geometry Shader

- Feladata: új geometria generálása, (vagy régi eltüntetése)
- A vertex shader által már transzformált primitívekre fut let.
- Bemenete: teljes primitív (szomszédsági infókkal, ha vannak)
- Kimenete: újabb primitív[ek], vagy semmi.
- Kimeneti típusok:*PointStream*, *LineStream*, vagy *TriangleStream*

Algoritmusok

- Point Sprite Expansion
- Dynamic Particle Systems
- Fur/Fin Generation
- Shadow Volume Generation
- Single Pass Render-to-Cubemap
- Per-Primitive Material Swapping
- Per-Primitive Material Setup

- Input Assambler
- Vertex Shader
- Hull Shader
- Tesselator
- Domain Shader
- Geometry Shader
- Stream Output
- Rasterizer
- Pixel Shader
- Output Merger
- szerelőszalagon kívül: Compute Shader

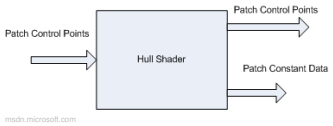


Tesszelációs lépések

- Speciális primitív tipusokat használ
  - négyszög foltok (*patches*)
  - háromszög foltok
  - iso-vonalak
- Alapötlet: a transzformációkat az alacsony felbontású foltokon végezzük, és tesszelációval, hardveresen bontjuk fel kisebb darabokra és tesszük részletgazdaggá.
- Használató:
  - *displacement mapping*
  - nézet függő dinamikus LOD (*level of detail*)
  - *morph*-olások gyorsítása

Hull Shader Stage

- Foltonként dolgozza fel a bementi pontokat.
- Konstansokat rendel a folthoz, amik a tesszelálás módját határozzák meg.
- Megadja a tesszeláció mértékét.
- Két külön kimenet:
  - Vezérlő pontok a *Domain Shader*-nek
  - Konstansok a *Tesselator Stage*-nek

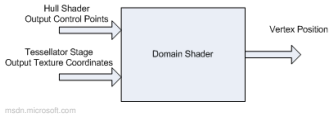


Tesselator Stage

- Nem programozható.
- Felbontja a tartományt kisebb darabokra (négszög, háromszög, szakasz).
- u, v (opcionálisan w) koordinátákat állít elő kimenetként.
- ”Láthatatlan” kimenet: topológiai információk a *primitive assembly*-nek

Domain Shader Stage

- A *Tesselator Stage* során előállított minden egyes pontra lefut.
- Bemenete:
  - A *Tesselator Stage*-től: u, v, (w)-k.
  - A *Hull Shader*-től: vezérlő pontok
  - A *Hull Shader*-től: tesszelációs faktorok
- A bemetekből egyetlen csúcspontot állít elő, és az lesz a kimenet.



Compute Shader

- Általános célú számításokat tesz lehetővé a GPU-n.
- Speciális adatszerkezeteket használ.
- Nincsenek megszorítások a ki- és bemenetekre.
- Felhasználásai:
  - Képfeldolgozás/post-processing
  - Ray-tracing
  - Fizika
  - AI
- ”Testvérei”
  - CUDA
  - OpenCL

Absztrakt fényforrások (ism.)

- Irány fényforrás
  - csak iránya van
  - egyetlen (világ k.r. beli) vektor megadja
  - a fénysugarakat párhuzamosnak tekintjük
  - távoli fényforrások megadására használható
- Pont fényforrás
  - csak pozíciója van
  - egy ponttal adjuk meg, világ k.r.-ben
  - Pl.: villanykörte
- *Spot* fényforrás
  - iránya, pozíciója és "fényköre" van
  - két szög + egy pont és egy vektor adja meg (világ k.r. !)
  - Pl.: asztali lámpa

Jelölések (ism.)

- $\vec{v} := \omega$ a nézeti irány
- $\vec{l} := \omega'$ a megvilágító, a fényt "adó" pont fele mutató vektor
- $\vec{n}$ a felületi normális
- $\vec{v}, \vec{l}, \vec{n}$ egységvektorok
- θ' a $\vec{l}$ és a $\vec{n}$ által bezárt szög

Ambiens tag

- A színtéren mindenütt jelenlevő fénymennyiség.
- Képlet: $k_a \cdot L_a$.
- k_a a felülettől függ, legyen `float4 ambientColor`.
- A `float4` az egy *rgba* négyes, ahol *a* az átlátszógás.
- L_a fénytől, felülettől független. Szintén legyen `float4 ambientLight`, de $a \equiv 1$.
- *Pixel shaderben* használható: `ambientColor*ambientLight`

Lambert-törvény

- BRDF: $L_{ref} = L_i k_d \cos^+ \theta'$
- Ezt neveztük *diffúz színnek*.
- k_d és L_i mint az előbb, de L_i az aktuális fényforrás tulajdonsága:
  `float4 diffuseColor`
  `float4 diffuseLight`
- Kéne még: $\cos^+ \theta'$.
- Kiszámítása: `saturate(dot(normal, toLight))`
- Ismerni kell hozzá `normal` $= \vec{n}$-t és `toLight` $= \vec{l}$-t.
- *Spot* fényforrásnál a fénykört is figyelembe kell majd venni.

Spekuláris visszaverődés - Phong modell

- BRDF: $L_{ref} = L_i k_s (\cos^+ \phi)^n$, ahol ϕ az $\vec{r}$ tükörirány és a $\vec{v}$ nézeti irány által bezárt szög.
- k_d és L_i (ez egy másik L_i) megint mint az előbb, L_i szintén az aktuális fényforrás tulajdonsága:
  `float4 specularColor`
  `float4 specularLight`
- n felület függő konstans, legyen `float specularPower`

Spekuláris visszaverődés - Phong modell

BRDF: $L_{ref} = L_i k_s (\cos^+ \phi)^n$; $\vec{r}$ tükörirány és a $\vec{v}$ nézeti irány.

- Kéne $\cos^+ \phi$, amihez meg kéne $\vec{r}$ és $\vec{v}$.
- $\vec{r}$ a $\vec{l}$ vektor $\vec{n}$-re vett tükörképe. Számítása:
  `float3 reflection = reflect(-toLight, normal)`
- $\vec{v}$ a nézeti irány, azaz a feületi pontból a camerába mutató egységvektor.
  `float3 directionToEye = normalize(eyePosition-worldPos)`
- $(\cos^+ \phi)^n$ számítása:
  `pow(saturate(dot(reflection, directionToEye)), specularPower)`

Összefoglalva

Felület tulajdonságai

- `ambientColor`
- `diffuseColor`
- `specularColor`
- `specularPower`
- `normal`
- `worldPos`

Fényforrás tulajdonságai

- `diffuseLight`
- `specularLight`
- `toLight`

Színtér tulajdonságai

- `ambientLight`
- `eyePosition`

Tudd, hogy értsd

- Minden felületi-optikai tulajdonságot meg lehet adni konstanssal, vagy akár textúrával.
- (Sőt, még többet, ha ügyesek vagyunk!)
- Minden vektor és pont világkoordináta-rendszerben adott.
- `eyePosition`-t frissíteni kell, ha változik a nézet!
- Probléma: a modellünk modell k.r.-ben van, és a *Vertex Shader* normalizált eszköz k.r.-be visz át!
- Megoldás: számoltassunk a *Vertex Shader*-rel világ k.r.-beli koordinátákat is!

Adatstruktúrák

```
struct VS_INPUT
{
  float4 pos    : POSITION;
  float3 normal : NORMAL;
};

struct VS_OUTPUT
{
  float4 pos      : POSITION;
  float3 normal   : TEXCOORD0;
  float3 worldPos : TEXCOORD1;
};
```

Vertex shader

```
VS_OUTPUT LightingVS(VS_INPUT input)
{
  VS_OUTPUT output;
  output.pos = mul(input.pos, WorldViewProjection);
  output.worldPos = mul(input.pos, World).xyz;
  output.normal =
    mul(InvWorld, float4(input.normal, 0)).xyz;

  return output;
}
```

Pixel shader

```
float4 LightingPS(float3 normal   : TEXCOORD0,
                  float3 worldPos : TEXCOORD1)
                    : COLOR0
{
  normal = normalize(normal);
  float3 toLight = ??? // Fenyforras fuggo
  float diffuseIntensity =
    saturate(dot(normal, toLight));
  float3 reflection = reflect(-toLight, normal);
  float3 directionToEye =
    normalize(eyePosition-worldPos);
  //...
```

Pixel shader

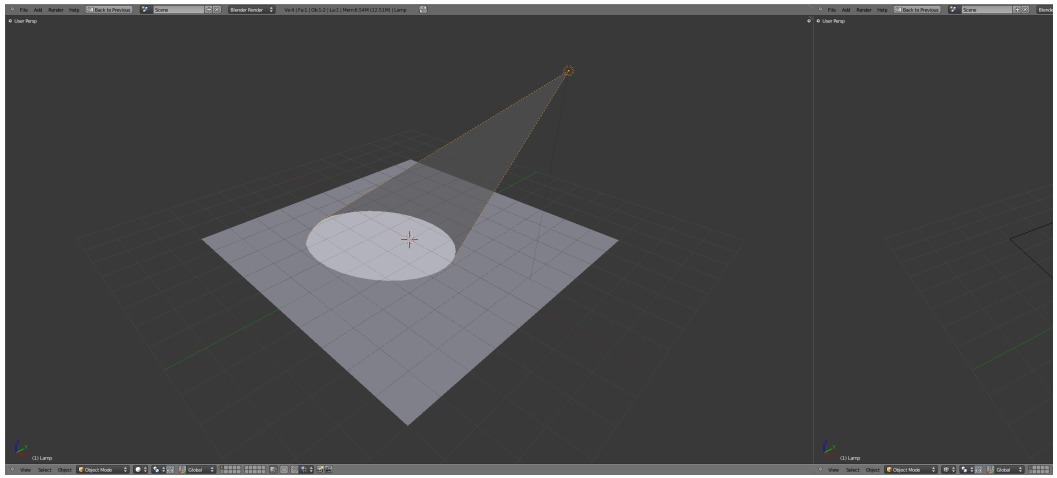
```
  //...
  float specularIntensity =
    pow(saturate(dot(reflection, directionToEye)),
      specularPower);
  if (diffuseIntensity <= 0)
    specularIntensity = 0;

  return ambientColor*ambientLight +
diffuseIntensity*diffuseColor*diffuseLight +
specularIntensity*specularColor*specularLight;
}
```

`toLight` számítása

- Irány fényforrás
  - Fény iránya, normalizált irányvektor: `float3 lightDirection`
  - `toLight = -lightDirection`
- Pont fényforrás
  - Fény pozíciója, helyvektor: `float3 lightPosition`
  - `toLight = normalize(lightPosition-worldPos)`
- *Spot* fényforrás
  - Fény iránya, normalizált irányvektor: `float3 lightDirection`
  - Fény pozíciója, helyvektor: `float3 lightPosition`
  - `toLight = normalize(lightPosition-worldPos)` mint pont fényforrás esetén

Spot fényforrás hatása

- Két extra paraméter:
  - belső fénykör: amin belül teljes intenzitással hat
  - külső fénykör: amin kívül abszolút nem hat
- A kettő között folyamatosan csökken a fény intenzitása.
- A fénykörökét a fényforrásból induló, a fény irányával megeggyező állású (végtelen)kúpoknak tekintjük.
- Egy felületi pont akkor van benne egy kúpban, ha a pontot a fényforrás pozíciójával összekötő egyenes a kúpon belül van.



Spot fényforrás hatása

- Az előbbi szakasz iránya pont a `toLight`.
- Ha a `lightDirection` és a `-toLight` által bezárt szög, kissebb, mint a kúp nyílásszögének a fele, akkor a szakasz benne van a kúpban, azaz a felületi pont is.
- A szögek helyett elég vizsgálni azok cos-át, ha a max. nyílásszög $\leqslant 180°$.
- Adjuk meg a két fénykört, a hozzájuk tartozó kúpok nyílásszögének felének cos-áva:
  - belső fénykör: `cosInnerCone`
  - külső fénykör: `cosOutterCone`

Spot fényforrás hatása

- `smoothstep(min, max, x)`:
  - 0, ha `x<min`
  - 1, ha `x>max`
  - lin. átmenet 0 és 1 között kül.
- `float spotFactor = smoothstep(cosOutterCone, cosInnerCone, dot(lightDirection, -toLight))`
- Ezzel kell beszorozni a diffúz és a spekuláris tagot.
- Pl. `diffuseIntensity`-t és `specularIntensity`