

05/01/2013

FACULTATEA
DE
AUTOMATICA SI
CALCULATOARE

ELEMENTE DE GRAFICA PE CALCULATOR



Laborator 9
Introducere în shadere

Shadere

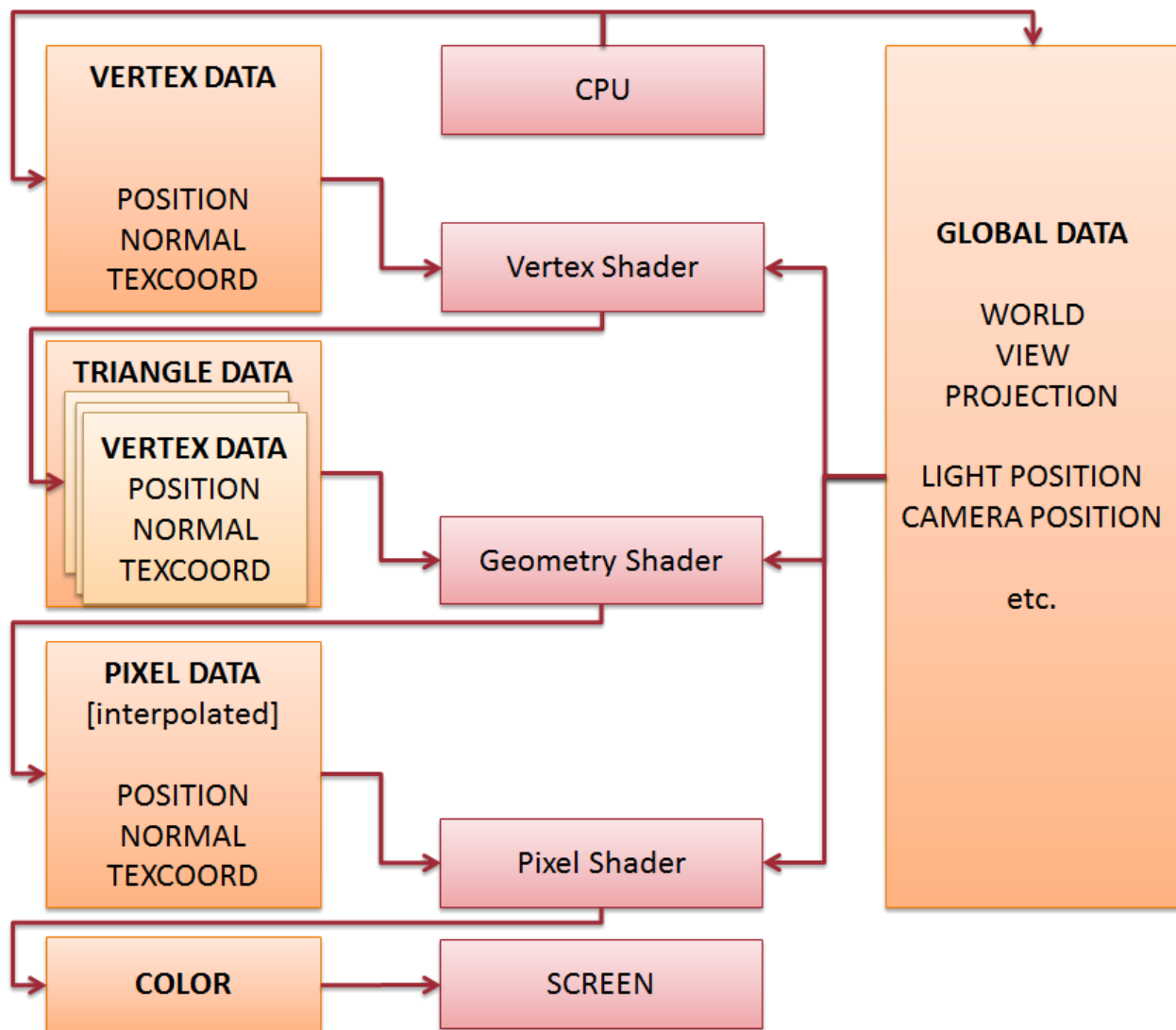
Un shader este un program compilat pe CPU, care ruleaza in intregime pe GPU si executa operatii grafice. Shaderurile sunt in general scrise pentru a aplica transformari pe un set intins de elemente in acelasi timp, fiind prin excelenta paralele. In grafica moderna, shaderurile stau la baza intregului lant de prelucrari grafice, tinzandu-se spre a limita din ce in ce mai mult cantitatea de cod CPU care genereaza/randeaza imagini sau modele 3D.

In grafica/jocurile moderne, singurele operatii de care se ocupa partea de CPU/C++ sunt de incarcare a modelelor 3D, de compilare a shaderelor si de selectia operatiilor din shader care vor fi aplicate pe geometria din scena. Majoritatea operatiilor de tip GLCeva folosite in laboratoarele trecute sunt considerate deprecate.

Shaderurile pot fi de mai multe tipuri (per pixel, per vertex, etc, vor fi detaliate mai jos) si in trecut placile video contineau unitati de procesare dedicata pentru fiecare tip de shader. In prezent, atat OpenGL cat si DirectX 10+ folosesc „Unified Shader Model”, o tehnologie care permite oricarui tip de shader sa foloseasca acelasi set de instructiuni si aceeasi logica. O exceptie notabila este arhitectura PS3 care pastreaza un model neunificat de shader, spre deosebire de Xbox si toate placile moderne pentru PC.

Banda grafica simplificata (pentru un singur model 3D) :

Cine	Când se apelează	Ce primește	Ce face	Ce trimite
[CPU] C++, C# sau orice	O dată, la început	Model 3D din fișier sau generat Cod shader necompilat	Convertește modelul 3D într-un format de date înțeles de GPU Compilează shaderurile	Geometrie (vertecși și patchuri) Variabile globale shader (date despre lumini, cameră, etc) Cod shader compilat
[GPU] Vertex Shader	Pentru fiecare vertex	Date pentru un vertex (e.g poziție, normală) Variabile globale (e.g view, proj)	Aplică transformări 3D	Date pentru un vertex, transformate
[GPU] Geometry Shader	Pentru fiecare patch (linie, triunghi sau quad)	Date pentru N vertecși (de obicei 3) Variabile globale	Aplică transformări 3D Generează geometrie	Date pentru unul sau mai multe patch-uri (limitat la 256 de vertecși pe DX11)
[GPU] Pixel Shader	Pentru fiecare pixel de pe fiecare triunghi, interpolat și rasterizat pe ecran	Date pentru un pixel (e.g poziție, culoare, etc.) interpolate trilinear. Variabile globale	Aplică transformări 3D Aplică lumini, umbre, alte operații la nivel de pixel.	O culoare (R,G,B,A) pentru pixelul respectiv.



Shaderele sunt compilate pe CPU si codul obiect este transmis direct in placa video. Acestea pot fi scrise fie direct in limbaj de asamblare GPU, fie intr-unul dintre limbajele pe baza de C dedicate shaderelor, HLSL (DirectX), GLSL (OpenGL) sau Cg.

HLSL

HLSL (High Level Shading Language) este un limbaj de programare pentru shadere care foloseste sintaxa C, dezvoltat de Microsoft in colaborare cu Nvidia. Ca sintaxa si functionalitate, este asemanator (in majoritatea cazurilor interschimbabil fara modificari) cu **Cg**, care este dezvoltat in totalitate de Nvidia.

Fisierele HLSL au de obicei extensia **.fx** sau **.hlsl**. Fisierele de tip header au extensia **.fxh** sau **.h**. Acestea sunt compilate la runtime folosind compilatorul HLSL integrat in DirectX. Puteti testa acest lucru modificand fisierele **.fx** gasite in majoritatea jocurilor moderne, e.g Batman: Arkham City. Unele jocuri bazate pe engine-ul Unreal3 folosesc extensia **.usf**. Majoritatea shaderelor gasite in aceste jocuri pot fi destul de utile ca material didactic, daca reusiti sa intelegeti ce date vin din C++ si cum sunt acestea organizate.

Sintaxă

Sintaxa HLSL este identica cu cea C/C++ cu mici variatii pe care nu le vom explora in acest laborator.

Tipurile de date cele mai folosite sunt :

- **Scalari:** *int, float, half, double, char, bool, ..*
- **Vectori:** *float4, float3, half4, etc..*
- **Matrici:** *float4x4, half3x3, etc..*
- Orice merge in C, merge si aici, de exemplu structuri, array-uri, etc.

Structura de baza a unui fisier HLSL este:

- **Variabile globale** – orice nu este initializat direct poate veni din C++. Daca C++ nu trimite variabila respectiva, aceasta va fi pur si simplu 0 (sau echivalent).
- **Declaratii de structuri** – definesc ce tip de Vertex Data, Pixel Data, etc. se transmite intre shadere
- **Vertex Shader** – o functie care primeste Vertex Data si intoarce Vertex Data
- **Geometry Shader** – o functie care primeste Vertex Data si intoarce un `TriangleStream<Vertex Data>`
- **Pixel Shader** – o functie care primeste Vertex Data si intoarce o culoare
- **Stari** – initializari ale unor structuri care contin instructiuni de randare (e.g activarea depth-buffer, stencil buffer, alpha blending, etc.)
- **Tehnici de randare** – un model 3D poate fi randat prin mai multe tehnici separate continute intr-un singur shader. Un mecanism simplu de a refolosi cod. Un shader poate avea mai multe tehnici. O tehnica poate avea mai multe render pass-uri.
- **Render Pass** – contine apeluri catre:
 - Initializare de stari (alegem in ce mod vrem sa se faca randarea)
 - Rularea, in secventa, a unui **Vertex Shader**, un **Geometry Shader** si un **Pixel Shader**

Structura este deosebit de flexibila. Putem include alte fisiere, putem defini mai multe shadere intr-un fisier si putem alege, prin pase si tehnici, ce shadere sa folosim.

simple.fx si **simple.fxh**, care insotesc acest laborator, sunt un exemplu bun de un foarte simplu fisier HLSL cu 3 shadere, 2 stari si minimul de operatii necesare afisarii unui model 3D pe ecran.

Important: C++ alege ce tehnici/ render pass-uri sunt folosite pentru fiecare model 3D in parte. In suportul de laborator veti folosi un singur pass (P0) a unei singure tehnici, pentru simplitate.

Restricții

Programele HLSL sunt limitate de GPU-urile pe care rulează și de sistemul de operare. Shaderelor din acest laborator folosesc versiunea 4_0 a standardului HLSL și au nevoie de SDK-ul DX11 pentru a rula (link la sfârșit). Deoarece nu sunt folosite funcții sau shadere cu funcționalități exclusive DX11, laboratorul va funcționa corect și pe GPU-uri care sunt compatibile doar cu DX10, dacă driverele sunt la zi.

În momentul compilării, se specifică versiunea standardului HLSL folosit. De exemplu, standardul 3_0 (DX9, folosit pe Xbox360 în continuare) nu suportă mai mult de 512 de instrucțiuni în Vertex Shader. Încercarea de a scrie o a 513-a linie de cod va rezulta în erori de compilare.

Metode comune în HLSL

Tot ce este trimis către shadere vine din C++. Deși programatorii au o flexibilitate mare în ce vor să transmită, sunt anumite informații pe care este uzual să le vrem în shadere. Așa cum am descris mai sus, datele care vin din C++ sunt de două feluri. Per vertex (vertex data) și per model 3D (globale).

Vertex Data

Pentru fiecare vertex, C++ trimite un set flexibil de informații. În majoritatea cazurilor, vom avea aceleași date de bază la care vom adăuga date suplimentare doar dacă este nevoie de ele:

- **VERTEX_DATA**
 - float4 POSITION (*Poziția vertexului în spațiu 3D, netransformată*)
 - float4 NORMAL (*Normala vertexului, normalizată, de obicei normala feței pe care se află*)
 - float2 TEXCOORD (*Coordonatele de textură ale vertexului*)
 - Se interpolează automat în pixel shader. (de la 0,0 la 1,1)
 - float4 COLOR (*Culoarea vertexului*)
 - Se interpolează automat în pixel shader

Variabile globale

Variabilele globale conțin în laboratorul de față valorile de mai jos (descrise în **simple.fxh**). Majoritatea sunt destul de uzuale (le veți întâlni în multe alte exemple de shadere) dar, în cazuri concrete, se poate adăuga orice doriți:

- float4x4 **view, proj**
 - Matricile de vizualizare și proiecție, calculate ca și până acum pe CPU și trimise direct.
- float 4x4 **world**
 - Matricea de transformări care plasează obiectul în scenă. Inițial toate punctele sunt în spațiul obiect (e.g. direct din fișierul care conține modelul 3D). Dacă dorim să aplicăm transformări (rotatii, translații, scalări) putem trimite această matrice. Bineînțeles, orice alte transformări pot fi aplicate direct în shader.

- float4 **center**
 - Centrul obiectului, calculat pe CPU. Este pur si simplu suma vertecilor impartita la numarul lor. Nu putea fi calculata in shader!
- float3 **lightPosition, eyePosition**
 - Pozitia luminii si a camerei
- Texture2D **modelTexture**
 - Textura modelului. Aceasta este incarcata din fisier pe CPU intr-un stream binar.
- float **loop**
 - Un contor simplu care este incrementat la fiecare frame. Poate fi folosit pentru animatii diverse in shader.

Exemple concrete

Ce putem face in Vertex Shader?

Vertex shader-ul standard nu face decat sa inmulteasca fiecare vertex cu matricile world, view si proj. Aceste operatii nu sunt obligatorii (le putem efectua si in geometry shader, daca vrem).

Avand in vedere ca vertex shader-ul este apelat pentru fiecare vertex in parte, avem control total asupra tuturor datelor pertinente la acest vertex, in principal asupra pozitiei lui.

De exemplu, daca am vrea sa „impingem” vertexul intr-o directie sau alta, este suficient sa ii modificam pozitia inainte (sau dupa, depinde de efect dorim) de a aplica inmultirile cu matricile de mai sus. Daca vrem sa il impingem in lungul propriei sale normale, putem face:

```
Out.position += In.position + distance * In.normal
```

Ce putem face in Geometry Shader?

Geometry shader-ul face exact ce face si un vertex shader, doar ca in loc sa se aplice pe fiecare vertex, se aplica pe fiecare patch (e.g triunghi) de geometrie. Shader-ul are la dispozitie un stream de iesire in care poate scrie patch-uri, astfel ca el poate sa genereze mai multe patchuri la iesire, daca acesta este rezultatul dorit. Un geometry shader de baza copiaza pur si simplu patchul de la intrare pe stream-ul de iesire.

Presupunand ca avem 3 vertecsi la intrare In[3], cel mai simplu geometry shader ar face:

```
OutputStream.append(In[0]);  
OutputStream.append(In[1]);  
OutputStream.append(In[2]);  
OutputStream.RestartStrip();
```

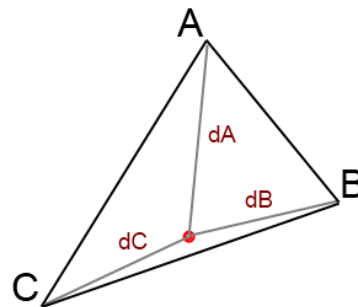
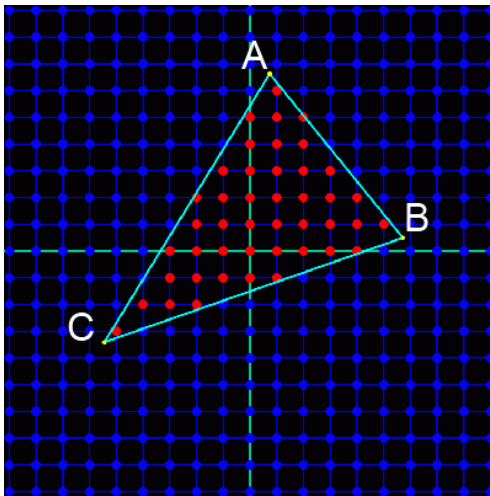
Acest sir de instructiuni scrie vertecsi in stream si apoi „inchide” triunghiul. Ultima instructiune este foarte importanta si trebuie apelata dupa fiecare triunghi complet. Daca am dori sa dublam triunghiul primit in In, nu avem decat sa continuam sa scriem in OutputStream. Ar fi de

asemenea o idee buna sa modificam vertecsii (e.g sa ii impingem pe normala, ca mai sus) ca sa se vada cele 2 triunghiuri distincte.

Pentru ca GPU-ul sa stie la ce sa se astepte, se foloseste marker-ul [maxvertexcount(K)] inainte de functie, unde K este numarul maxim de vertecsii pe care ne asteptam sa il scriem. Nu este neaparat nevoie sa scriem K vertecsii, dar daca incercam sa scriem mai multi, orice peste K va fi ignorat.

Ce putem face in Pixel Shader?

Destul de multe. Pixel shader-ul este deosebit de puternic, deoarece controleaza in mod direct culoarea care apare pe ecran in punctul respectiv. Este apelat pentru fiecare pixel rasterizat pe modelul 3D curent.



Pentru triunghiul de, presupunand ca grila ros/albastra corespunde cu pixelii de pe ecran, Pixel Shader-ul va fi apelat pentru fiecare bulina rosie. **Parametrii de intrare ai Pixel Shader-ului vor fi interpolati trilinear!** De exemplu, normala si pozitia in orice punct rosu vor fi interpolate folosind valorile acestora in extremitatile triunghiului si distanta intre punctele rosii si acestea.

Mult mai simplu decat suna, pentru orice parametru V pe care il stim in A, B, C, valoarea acestuia in punctul rosu (P) poate fi determinata prin:

$$V_P = \frac{V_A \cdot dA + V_B \cdot dB + V_C \cdot dC}{dA + dB + dC}$$

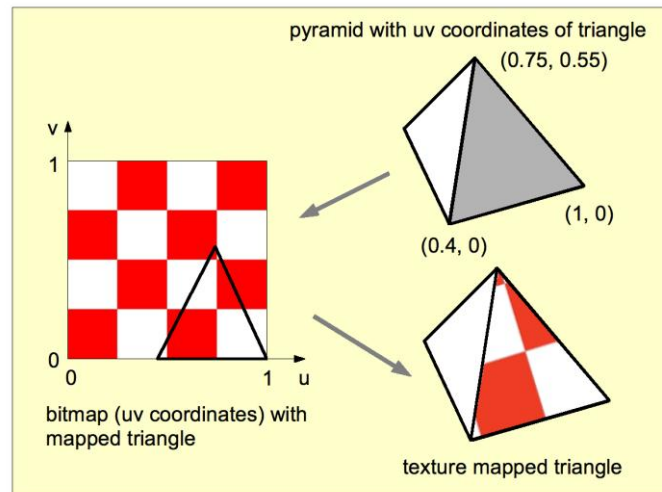
Majoritatea valorilor care intra in Pixel Shader sunt interpolate in acest fel, ca de exemplu culoarea de intrare, pozitia sau normala.

Cateva tehnici comune folosite in Pixel Shader :

Texture Mapping

Asa cum ati vazut mai sus, fiecarui vertex i se asociaza coordonate de textura, notate in mod uzual cu U,V si care pot avea valori intre 0 si 1.

- Coordonatele de textura U, V corespund modului in care se mapeaza vertexul curent pe o textura virtuala aflata in primul cadran (de la U,V = 0,0 la U,V = 1,1)
- De exemplu, daca vrem sa mapam pe un quad o textura in intregime, coltul stanga sus va avea U,V = 1,0, coltul dreapta sus U,V = 1,1 si coltul stanga jos va avea U,V = 0,0. Exemplu:



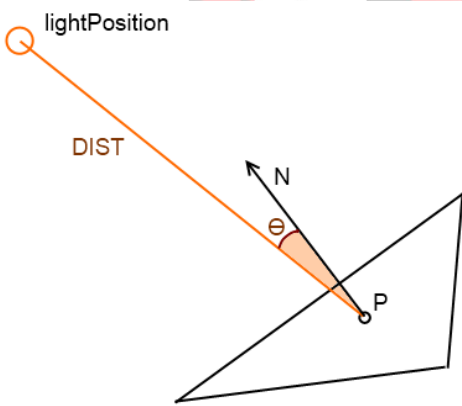
Coordonatele UV vin din C++. Un procedeu foarte des intalnit este maparea acestor coordonate UV pe o textura, pentru a obtine culoarea pixelului din locul respectiv.

Pentru a face acest lucru in Pixel Shader, se poate folosi urmatoarea functie pe un TextureSampler (in laborator aveti o textura numita modelTexture si un sampler pe aceasta textura modelTextureSampler. Sampler-ul nu face decat sa specifice cum se doreste sa se faca interpolarea) :

```
float4 color = texture.Sample(textureSampler, texcoord)
```

Iluminarea per-pixel

O varianta foarte simpla de iluminare calculata pe fiecare pixel este :



DIST – distanta de la originea luminii pana in acel punct (lungimea vectorului diferenta intre pozitia luminii si pozitia punctului)
 $DIST = \text{length} (\text{position} - \text{lightPosition})$

DIR – directia din care vine lumina (diferenta normalizata intre pozitia luminii si pozitia punctului)
 Atentie! Vertecsii contin si un camp numit **absolutePosition** care stocheaza pozitia inainte de a fi aplicate matricile view si proj. Acest camp trebuie folosit, deoarece **position** va fi exclusiv 2D in acest moment:
 $DIR = \text{normalize}(\text{position} - \text{lightPosition})$

INT – intensitatea luminii in acel punct
 $INT = \text{saturate} (\text{dot} (\text{normal}, - DIR))$

(Folosim dot product pentru ca este dependent de unghiul dintre vectori. Astfel lumina va creste in intensitate pe masura ce unghiul este mai ascutit)

A – lumina ambientala

$$\text{color} = A + DIST * INT$$

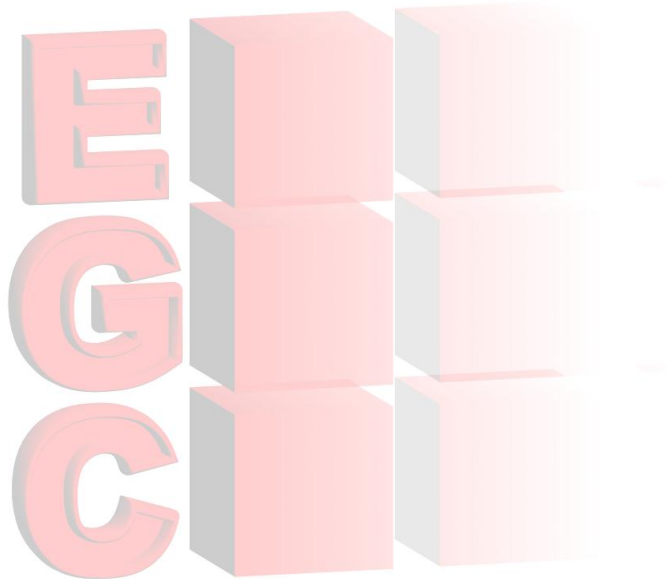
Laboratorul a fost scris in C# si HLSL, folosind .NET 4.0, DX11 SDK si SlimDX11. Codul sursa nu este inclus.

Dependințe:

- [DX11 SDK](#)

Alte informații:

- [MSDN HLSL Documentation](#)
- [RB Whitaker HLSL Tutorials](#)
- [Wikipedia HLSL page](#)
- [Wikipedia Unified Shader page](#)
- [Riemer's Per Pixel Lighting Tutorial](#)



Responsabil laborator:
Daniel Flamaropol