Programmation GPU par les shaders

Djihane BABAHENINI

djihane.babahenini@univ-biskra.dz

12/07/2019

1.0

Légende

 Entrée du glossaire

 Abréviation

 Référence Bibliographique

 Référence générale

Table des matières

Objectifs 5

Introduction 6

I - **Qu'est ce qu'un shader ?** 7

1. Définition 7

2. Histoire des GPUs 7

3. Types de shaders 8

3.1. Vertex shader 8
3.2. Geometry shader 9
3.3. Fragment shader 10

4. Exercice 11

5. Exercice : Choisissez la bonne réponse : 12

5.1. Exercice 12
5.2. Exercice 12
5.3. Exercice 12

II - **Langages de programmation des shaders** 13

1. Langage assembleur 13

2. Langage Cg 14

3. Langage GLSL 15

4. Langage HLSL 16

5. Exercice 17

III - **Les shaders en GLSL** 18

1. Evolution d'OpenGL et GLSL 18

2. GLSL : éléments de langage 19

2.1. Types de données 19
2.2. Structures 21
2.3. Qualifiers 22

3. Exercice : Choisissez les bonnes réponses : 23

3.1. Exercice 24
3.2. Exercice 24

3.3. Exercice *24*
3.4. Exercice *24*

IV - **Tester votre compréhension** 25

V - **Conclusion** 28

Solutions des exercices 29

Glossaire 34

Abréviations 35

Références 36

Bibliographie 37

Webographie 38

Objectifs

A l'issu de ce chapitre, l'étudiant sera capable de :

- *Distinguer* entre les différents types de shaders.
- *Connaître* les langages de programmation des shaders.
- *Exploiter* les fonctions de GLSL.

Pré-requis :

L'étudiant doit connaître préalablement les notions suivantes pour pouvoir suivre le maximum de ce cours :

- Les architectures et les algorithmes parallèles.
- Le langage de programmation C et C++.

Introduction

Le GPU est le processus de la carte graphique qui sert à effectuer des rendu 3D accélératrices. Un programme exécutable sous GPU est appelé un shader qui est utilisé pour manipuler des données avant de les dessiner sur un écran

Dans ce chapitre, nous allons décrire les différents types de shaders, et ces intérêts dans le domaine de rendu 3D, ainsi que les fonctions nécessaires à programmer sur GPU avec les shaders.

Qu'est ce qu'un shader ?

Définition	7
Histoire des GPUs	7
Types de shaders	8
Exercice	11
Exercice : Choisissez la bonne réponse :	12

1. Définition

Définition

Un shader est un morceau de code exécutable sur GPU qui se place entre les étages de pipeline graphique. Il a donc :

- un code source,
- une compilation,
- une exécution,

La compilation d'un shader est effectué lors de la compilation de l'application, mais l'exécution se passe au niveau des GPUs.

Les shaders sont très utilisés dans le domaine de rendu 3D pour :

- donner des effets plus réalistes,

rendre les applications qui utilisent les shaders puissants et flexibles.

Le terme "shader" est premièrement utilisé pour se référer à "pixel shader", mais ensuite il apparaît d'autres utilisations de ce terme comme le vertex et geometry shader

p.37

2. Histoire des GPUs

1. *NVIDIA GeForce 3 (2001)*
   - Premier GPU programmable, DirectX 8.0: VS,PS 1.0-1.1
   - ATI Radeon 8500, Microsoft Xbox, NVIDIA GeForce 4 Titanium
   - shaders très limité (fragment shader ressemblait plus à un script de configuration, seulement quelques instructions d'assemblage), texture shaders

- OpenGL: extension ARB_vertex_program (universal)

2. *DirectX 8.1*
   - PS 1.2-1.4 (ATI), VS remains 1.1, Radeon 9000
3. *NVIDIA GeForce FX (2002)*
4. *ATI Radeon 9500-9700*
   - *DirectX 9.0:* VS, PS 2.0
     - La taille de programme a augmenté : 256 instructions
     - Variables uniforms : 256 vecteurs
     - Plus de formats des données : FPU,....
   - *OpenGL :* fonctionnalités équivalentes via des extensionsp.36

3. Types de shaders

Vertex shader	8
Geometry shader	9
Fragment shader	10

Le vertex et le fragment shaders sont les types les plus importants dans un pipeline graphique car ils exposent les fonctionnalités de base de GPU.

Avec l'évolution des GPUs, on distingue d'autres types des shaders tels que :

- Tessellation control shader.
- Tessellation evaluation shader.
- Geometry shader.
- Compute shader.

Dans cette section, on s'intéresse au trois types de shaders : vertex, geometry, fragment shders

3.1. Vertex shader

Définition

Le vertex shader est un programme qui sert à calculer les attributs de chaque sommet. Il est exécuté une seule fois par sommet.

Un vertex shader possède en entrée les attributs de sommet (position, couleur, coordonnées des texture,.....) et à travers des fonctions prédéfinies il génère les sommets transformés et d'autres informations facultatives comme la couleur, la normal, et les coordonnées de texture.

Complément

Un vertex shader offre les possibilités :

1. Calcul des attributs de sommets.
2. Calcul de tout ce qui peut être interpolé linéairement entre sommets.
3. Mélange de transformations (‘Vertex Blend').
4. Mélange de maillages (‘Morphing').
5. Déformation du maillage (‘Vertex Deformation').
6. Calcul des coordonnées de texture.[p.36]

Exemple

Un exemple de vertex shader (écrit en GLSL) pour le calcul de la position transformé d'un sommet :

```
1 in vec3 vtx_position ;
2 out vec3 position ;
3 uniform float MVP ;
4 void main(void) {
5
6  position = vtx_position * MVP ;
7 }
```

Attention

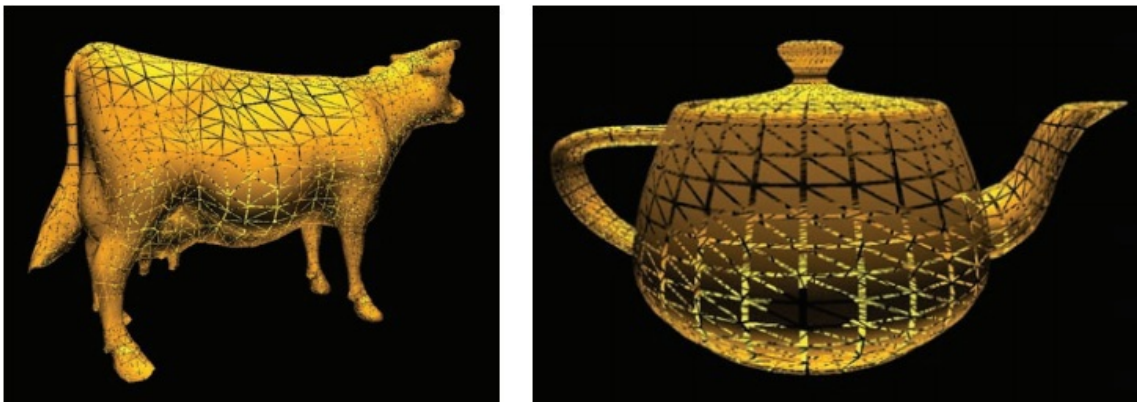
Un vertex shader est exécuté une et une seule fois pour chaque sommet. Donc si on a un objet qui contient 1000 sommets, le vertex shader sera exécuté 1000 fois.

3.2. Geometry shader

Définition

Le geometry shader est un autre type de shader disponible avec les GPUs modernes. Les opérations de ce shader modifient ou développent la géométrie d'origine envoyée au shader en développant de nouveaux sommets et groupes de sommets.[p.37]

Il est donc responsable de la création de nouvelles primitives de rendu. Un geometry shader est exécuté une fois par primitive, par exemple, il est utilisé pour émettre des triangles, où il sera alors exécuté trois fois moins que le vertex shader. Bien que l'exécution du geometry shader puisse être peu coûteux, cela augmente toujours la complexité de la scène, ce qui se traduit par une durée de calcul plus longue du GPU pour restituer la scène.[p.38]



Triangulation d'un modèle avec le geometry shader.

Exemple

Voici un exemple de geometry shader

```
layout( triangles ) in;
layout( triangle_strip, max_vertices=32 ) out;
uniform float uShrink;
uniform mat4 uModelViewProjectionMatrix;
in vec3 vNormal[3];
out float gLightIntensity;
const vec3 LIGHTPOS = vec3( 0., 10., 0. );
```

3.3. Fragment shader

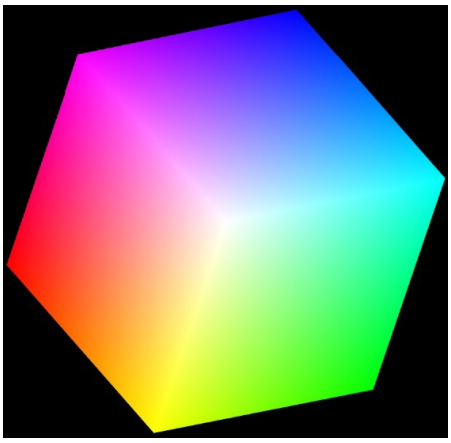
Définition

Un fragment shader est un morceau de code qui est exécuté une fois, et une seule fois, par fragment.

La tâche minimale d'un fragment shader consiste à générer une couleur RGBA. Cette couleur peut être calculer par n'importe quel moyen: de manière procédurale, à partir de textures ou en utilisant les données de sortie de vertex shader.

En réalité, un fragment shader est exécuté des millions de fois. Donc, l'optimisation dans un fragment shader est plus critique que dans les vertex shaders.

Un fragment shader possède en entrée les valeurs interpolées des sommets : position, couleur, coordonnées de texture..... La sortie de fragment sera la couleur d'un pixel.



Exemple de résultat de fragment shader.

Complément

Un fragment shader offre les possibilités :

1. Réflexion par pixel.
2. Illumination par pixel (phong, BRDF).
3. Textures procédurales.
4. Calcul de profondeur (depth map).

Exemple

Un exemple de fragment shader (écrit en GLSL) pour la génération de la couleur d'un pixel :

```
1 out vec4 color;
2
3 void main(void)
4 {
5   color = vec4(0.1,0.3,0.3,1);
6 }
```

Remarque

on manipule bien un fragment et non un pixel, par exemple, un fragment shader qui déplacerait un pixel sur l'écran n'a aucun sens.

4. Exercice

[solution n°1 p.29]

Placer les actions ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

OpenGL: extension ARB_vertex_program (universal)

Plus de formats des données : FPU,....

ATI Radeon 8500, Microsoft Xbox, NVIDIA GeForce 4 Titanium

Premier GPU programmable, DirectX 8.0: VS,PS 1.0-1.1

Fonctionnalités équivalentes via des extensions

La taille de programme a augmenté : 256 instructions

Variables uniforms : 256 vecteurs

NVIDIA GeForce 3 (2001)	DirectX 9.0: VS, PS 2.0	OpenGL

5. Exercice : Choisissez la bonne réponse :

[solution n°2 p.29]

5.1. Exercice

Le mélange de maillage se fait au niveau de :

- Geometry shader
- Vertex shader
- Fragment shader

5.2. Exercice

Un vertex shader est exécuté une et une seule fois pour chaque primitive

- Vrai
- Faux

5.3. Exercice

L'illumination par pixel se fait au niveau de :

- Vertex shader
- Fragment shader
- Geometry shader

II

Langages de programmation des shaders

Langage assembleur	13
Langage Cg	14
Langage GLSL	15
Langage HLSL	16
Exercice	17

Il existe plusieurs langages de programmation qui permettent de programmer les shaders. On distingues deux catégories principales :

1. Langage de bas niveau : le langage assembleur.
2. Langages de haut niveau : Cg, HLSL, GLSL.

1. Langage assembleur

Les premiers shaders ont écrit en langage assembleur, c'est un langage de bas niveau qui sert à représenter les shaders comme une séquence d'instructions sur les registres.

L'assembleur est un langage très proche de la machine. Il prend en entrée les attributs de vertex /fragment shaders, et génère en sortie d'autres attributs.p.36

Complément : Avantages de langage assembleur

- aide à comprendre le fonctionnement des cartes graphiques (ATI's r300, NVIDIA's nv30).
- aident à comprendre les APIs haut-niveau (Cg, HLSL, ...).
- Plus faciles à utiliser (extensions d'OpenGL).

Complément : Inconvénients de langage assembleur

- Faible lisibilité
- Écriture complexe
- Réutilisation difficile

- Contrôle de flux très limité.
- Dépendant de la plateforme.

Exemple

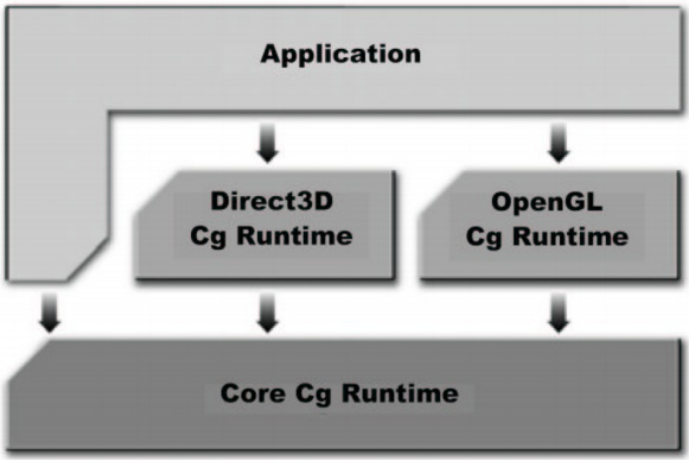
Code écrit en langage assembleur :

```
!!ARBvp1.0
 # ARB_vertex_program généré par NVIDIA Cg compiler
PARAM c12 = { 0, 1, 0, 0 };
TEMP R0, R1, R2;
ATTRIB v18 = vertex.normal;
ATTRIB v16 = vertex.position;
PARAM c0[4] = { program.local[0..3] };
PARAM c10 = program.local[10];
PARAM c5 = program.local[5];
PARAM c9 = program.local[9];
PARAM c4 = program.local[4];
PARAM c8 = program.local[8];
PARAM c11 = program.local[11];
PARAM c7 = program.local[7];
PARAM c6 = program.local[6];
 DP4 result.position.x, c0[0], v16;
 DP4 result.position.y, c0[1], v16;
 DP4 result.position.z, c0[2], v16;
 DP4 result.position.w, c0[3], v16;
 ADD R2.xyz, c7.xyzx, -v16.xyzx;
 DP3 R0.x, R2.xyzx, R2.xyzx;
 RSQ R1.w, R0.x;
 ADD R0.yzw, c6.xxyz, -v16.xxyz;
 DP3 R0.x, R0.yzwy, R0.yzwy;
 RSQ R0.x, R0.x;
 MUL R1.xyz, R0.x, R0.yzwy;
```

2. Langage Cg

Le langage Cg*p.35* développé par Nvidia (2002-2012) est un langage de programmation des shaders de haut niveau.

- Il est indépendant de la plateforme DirectX ou OpenGL.
- Syntaxe, opérateurs et fonctions ont dérivé de C/C++.
- Il prend en charge des différents types de GPU.
- Le langage utilisé dans le moteur de jeux Unity.*p.34*



Intégration de Cg dans les applications OpenGL et DirectX

Complément : Avantages de langage Cg

- Multi plate-forme : DirectX et OpenGL
- Écriture simplifiée
- Réutilisation par fonctions

Complément : Inconvénients de langage Cg

- Code non optimal
- Offre des fonctions limitées
- Compilateur externe

Exemple

Code écrit en langage Cg :

```
struct pixel_out {
float4 color : COLOR;
} ;
pixel_out main ( float2 texCoord : TEXCOORD0, uniform
sampler2D myTexture ) {
pixel_out OUT;
OUT.color = tex2D(myTexture , texCoord).bgra ;
return OUT;
}
```

3. Langage GLSL

Le langage GLSL[p.35] est un langage de haut niveau conçu par 3DLabs et OpenGL en 2004.

- Langage dérivé de C/C++
- développé pour l'API graphique OpenGL et webGL

Complément : Avantages de langage GLSL

- Offre une grande souplesse de programmation des shaders.
- Compilateur interne.
- Offre des effets de rendu évolué.

Complément : Inconvénients de langage GLSL

- Code complexe
- Un programme GLSL qui compile sur un pilote NVIDIA peut ne pas compiler sur un pilote ATI
- Nombre d'itérations dans la boucle for limité

Exemple

Exemple d'un programme écrit en GLSL

```
vec3 linear_to_gamma(vec3 color)
{
 return pow(color, vec3(1.0 / 2.2));
}
void main()
{
 vec4 color;
 color.rgb = linear_to_gamma(color.rgb);
 frag_color = color;
}
```

4. Langage HLSL

Le langage HLSLp.35 développé par Microsoft en 2003 est un langage de haut niveau pour la programmation des shaders.

- Il est utilisé uniquement pour les applications de DirectX.
- Syntaxe similaire à Cg.

Complément : Avantages de langage HLSL

Rapide par rapport à GLSL

Code compact par rapport à Cg

Complément : Inconvénients de langage HLSL

Utilisé uniquement pour Windows

Exemple

Exemple d'un code écrit en HLSL :

```
sampler2D my_texture ;
float4 MyShader ( float2 Tex : TEXCOORD0 ) : COLOR0 {
```

```
3 float4 Color ;
4 Color.bgra = tex2D ( my_texture , Tex ) ;
5 return Color ;
6 }
```

5. Exercice

[solution n°3 p.30]

Placer les tâches ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

Il est indépendant de la plateforme DirectX ou OpenGL.

aident à comprendre les APIs haut-niveau (Cg, HLSL, ...).

Il est utilisé uniquement pour les applications de DirectX.

Syntaxe, opérateurs et fonctions ont dérivé de C/C++.

Offre une grande souplesse de programmation des shaders.

aide à comprendre le fonctionnement des cartes graphiques (ATI's r300, NVIDIA's nv30).

développé pour l'API graphique OpenGL et webGL

Syntaxe similaire à Cg.

Il prend en charge des différents types de GPU.

Plus faciles à utiliser (extensions d'OpenGL).

Rapide

Assembleur	Cg	GLSL	HLSL

Les shaders en GLSL

Evolution d'OpenGL et GLSL	18
GLSL : éléments de langage	19
Exercice : Choisissez les bonnes réponses :	23

1. Evolution d'OpenGL et GLSL

OpenGL 2.0/GLSL 1.1

Cette version d'OpenGL introduit la programmation graphique basée sur le vertex et le fragment shader. Ces shaders apportent une flexibilité et une créativité énormes à la programmation graphique OpenGL.

elle inclut :

- Les VBO[p.34 p.35] qui permettent de stocker des tableaux de vertex dans la mémoire graphique afin de réduire la communication nécessaire entre le processeur et la carte.
- Les requêtes d'occultation qui permettent de demander combien de pixels un élément de scène particulier occuperait s'il était affiché.
- Les opérations de stencil offrent un meilleur moyen pour l'ombrage.

OpenGL 3.0/GLSL 3.0

Cette version d'OpenGL/GLSL inclut :

- L'utilisation de geometry shader.
- Les FBO[p.34 p.35] qui permettent d'indiquer ou le rendu OpenGL sera effectué.
- Des textures beaucoup plus volumineux.
- Ensemble de variables uniforms.

OpenGL 4.0/ GLSL 4.0

Cette version d'OpenGL/GLSL inclut :

- Apparition de tessellation shader.
- Apparition de compute shader.
- Amélioration de geometry shader.

- Interpolation de texture.[p.37]

Texte légal : Evolution de GLSL en parallèle avec OpenGL.

Année	Version GLSL	Version OpenGL
2016	4.50.6	4.5
2014	4.40.9	4.4
2013	4.30.8	4.3
2011	4.20.11	4.2
2010	4.10.6	4.1
2010	4.00.9	4.0
2010	3.30.6	3.3
2009	1.50.11	3.2
2009	1.40.08	3.1
2009	1.30.10	3.0
2006	1.20.8	2.1
2004	1.10.59	2.0

2. GLSL : éléments de langage

Types de données 19
Structures 21
Qualifiers 22

2.1. Types de données

Syntaxe : Scalaires

Entier : dans la programmation GPU, les entiers ont souvent utilisé pour les boucles et les indices de tableau ainsi que pour référencer une texture. Un type entier est stocké sur 2 octets + signe [-65535, +65535].

```
int a = 4;
```

Réel : comme dans le langage C, ce type représente les nombres réels stockés sur 4 octets.

```
float x = 4.0 ;
```

```
float x = 4.f ;
```

Booléen : le type booléen prend seulement deux valeurs vrai (true) ou faux (false).

```
bool b = true ;
```

Syntaxe : Vecteurs

GLSL inclut les vecteurs de rang 2, 3 ou 4 valeurs, ces valeurs peuvent être de type entier, réel, ou booléen. Ce type est très utilisé dans la programmation GPU que le type scalaire.

```
vec4 vector ;
```

On peut définir sa première valeur par :

```
vector[0] = 6.4 ;
```

ou on accède à ses éléments par indexation :

```
vector.x = 6.4 ;
```

```
vector.y = 5.3 ;
```

[0]	[1]	[2]	[3]	utilisables pour les éléments de la boucle
x	y	z	w	utilisables pour les vecteurs qui représentent un point
s	t	p	q	utilisables pour les coordonnées de texture
r	g	b	a	utilisables pour les vecteurs de couleur

```
1 vec4 pos1 = vec4(1.0, 2.0, 3.0, 4.0);
2 vec4 pos2 = pos1.yzwx            // pos2 = (2.0, 3.0, 4.0, 1.0)
3 vec4 pos3 = pos1.xxyy            // pos3 = (1.0, 1.0, 2.0, 2.0)
```

Syntaxe : Tableaux

Les tableaux (arrays) en GLSL ont le même syntaxe que les tableaux en langage C, la taille est optionnelle et peut être calculer automatiquement par le compilateur.

Ils ont limité aux tableaux 1D.

```
1 vec4 vecteur[]; // taille à définir
2 vecteur[0] = vec4(0.,0.,0.,0.);      // vecteur vecteur est maintenant maintenant
  de taille 1
3 vecteur[6] = vec4(1.,0.,0.,0.);      // vecteur vecteur est maintenant maintenant
  de taille 7
```

Syntaxe : Matrices

Les matrices en GLSL ont principalement utilisé pour les transformations géométriques.

Le GLSL a les matrices carrées de 2*2 (mat2), 3*3 (mat3), 4*4 (mat4) ou des combinaisons de 2 à 4 lignes-colonnes (exemple : mat2*3)

Les éléments de la matrice sont toujours de type réel.

Contrairement au langage C, le GLSL donne la priorité au colonnes.

```
1 mat4 projection;
2 projection[2] = vec4(1.0, 3.2, -1.2, 0.0);      //accès à une colonne
3 pojection[1][0] = 1.0;    //accès à une cellule (1ère ligne de la deuxième
  colonne)
```

Syntaxe : Samplers

Le type sampler permet l'accès aux textures d'OpenGL. Pour pouvoir utiliser ce type, il faut tout d'abord créer la texture dans le programme OpenGL, ensuite on envoie son indice au programme GLSL.

GLSL offrent les types :

- *sampler1D :* accès à une texture à une dimension
- *sampler2D :* accès à une texture à deux dimensions
- *sampler3D :* accès à une texture à trois dimensions
- *samplerCube :* accès à une texture de type cube map
- *sampler1DShadow :* accès à une texture de profondeur à une dimension
- *sampler2DShadow :* accès à une texture de profondeur à deux dimensions

L'accès est en lecture seule, c'est à dire on ne peut pas modifier les textures via les shaders

```
1 uniform sampler2D Grass;       //déclaration de la texture
2 vec4 color = texture2D(Grass, coord);        //utilisation de la texture
```

2.2. Structures

Syntaxe

GLSL offre la possibilité de créer des structures de données à la manière du langage C :

```
struct vertex

{

vec3 Position ;

vec3 Normal ;

vec2 UV ;

}vertexP;
```

`vertex` sera le nom d'un nouveau type :

```
vertex v ;
```

2.3. Qualifiers

Syntaxe : Uniform

Uniform en GLSL désigne une variable globale constante. Ce qualifier permet d'identifier une variable qu'il passe sa valeur de programme OpenGL au programme GLSL.

La variable uniform est une variable en lecture seule.

Commun au vertex et fragment shader.

```
uniform float intensity ;
```

Syntaxe : Varying

Varying est une variable globale dont chaque shader possède sa propre copie. Ces variables servent à faire passer des valeurs du VS au FS par interpolation.

- Faire le lien entre vertex et fragment Shader
- Obsolète à partir de la version 150 du GLSL.
- Accessible en écriture dans le vertex Accessible en écriture dans le vertex shader
- Accessible en lecture dans le fragment Accessible en lecture dans le fragment shader

```
varying vec3 position ;
```

Syntaxe : Attribute

Variables correspondant à ce qui est injecté fréquemment dans le pipeline : sommets, normales ... Il fait le lien entre OpenGL et vertex shader

- Accessible en lecture uniquement Accessible en lecture uniquement
- Réservé au vertex Réservé au vertex shader
- Certaines variables préséfinies : glPosition, glNormal,....
- Obsolète à partir de la version 150 du GLSL.

```
attribute float scale ;
```

Syntaxe : In

Variable globale provenant d'un stage précédent du pipeline (possiblement avec interpolation).

```
in vec3 color ;
```

Syntaxe : Out

Variable globale provenant d'un stage suivant du pipeline (possiblement avec interpolation).

out vec3 color ;

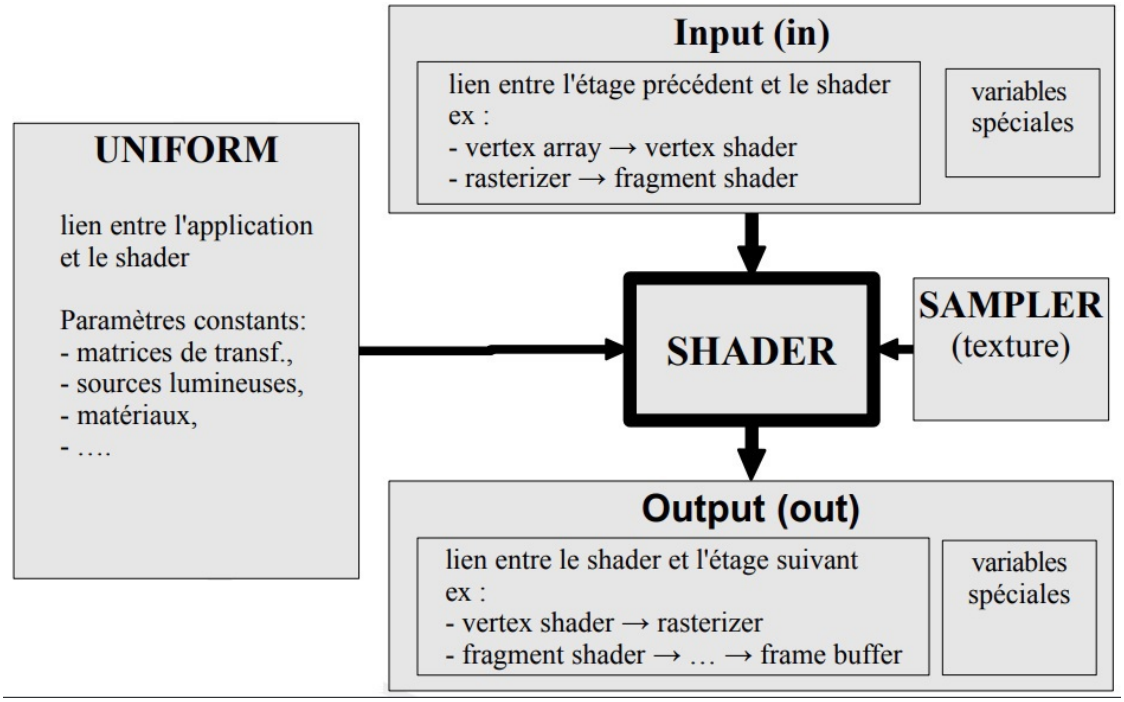
Syntaxe : Const

Initialisation obligatoire et accessible par référence

```
const PI = 3.14 ;
```

Remarque : Passage par valeur ou par références pour les fonctions

- *in :* passage par valeur
- *out :* accessible en écriture seulement
- *inout :* passage par référence
- *const :* lecture uniquement



Programme shader avec les différents qualifiés.

3. Exercice : Choisissez les bonnes réponses :

[solution n°4 p.30]

3.1. Exercice

Quels sont les qualifiers de langage GLSL :

- ☐ uniform
- ☐ vector
- ☐ attribute
- ☐ varying
- ☐ mat
- ☐ struct

3.2. Exercice

Les vecteurs de langage GLSL sont de type :

- ☐ char
- ☐ int
- ☐ float
- ☐ long
- ☐ bool

3.3. Exercice

Les types de données fournis par le GLSL sont:

- ☐ scalairs
- ☐ vecteurs
- ☐ chaîne de caractère

3.4. Exercice

Le qualifier in est utilisé dans :

- ☐ passage par valeur
- ☐ passage par adresse
- ☐ interpolations des données entre le vertex et le fragment shader

Tester votre compréhension

IV

Exercice

[solution n°5 p.31]

Placer les tâches ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

Apparition de tessellation shader.

Les FBO*p.34 p.35* qui permettent d'indiquer ou le rendu OpenGL sera effectué.

Les VBO*p.34 p.35* qui permettent de stocker des tableaux de vertex dans la mémoire graphique afin de réduire la communication nécessaire entre le processeur et la carte.

Ensemble de variables uniforms.

Les opérations de stencil offrent un meilleur moyen pour l'ombrage.

Apparition de compute shader.

Amélioration de geometry shader.

L'utilisation de geometry shader.

Les requêtes d'occultation qui permettent de demander combien de pixels un élément de scène particulier occuperait s'il était affiché.

OpenGL 2.0/GLSL 1.1	OpenGL 3.0/GLSL 3.0	OpenGL 4.0/ GLSL 4.0

Exercice

[solution n°6 p.32]

Quel est le langage utilisé pour la plateforme de jeux Unity ?

Exercice

[solution n°7 p.32]

La méthode qui permet d'envoyer des données 3D (position, normal, couleur, coordonnées de texture, ...) au GPU est le :

Exercice

[solution n°8 p.32]

Ensemble de buffers logiques et d'états définissant l'emplacement où le rendu OpenGL sera effectué dans le :

Exercice

[solution n°9 p.32]

Les qualifiers qui permettent le passage par valeur ou par adresse dans une fonction sont :

- ☐ uniform
- ☐ in
- ☐ out
- ☐ varying
- ☐ inout
- ☐ attribute
- ☐ const

Exercice

[solution n°10 p.33]

Complétez les trous suivants :

GLSL (OpenGL): est un langage permettant la programmation des scènes . Le langage GLSL est basé sur le langage

Exercice

[solution n°11 p.33]

Complétez les trous suivants :

Les variables servent à la communication entre le programme exécuté par le et les exécutés par le

Exercice

[solution n°12 p.33]

Complétez les trous suivants :

Les variables servent à la communication entre le shader exécuté et le shader

Conclusion

IV

Dans ce chapitre, nous avons :

1. Montré l'utilité de la programmation GPU via les shaders dans le rendu 3D.
2. Vu les différents types de langage de programmation GPU.
3. Expliqué les types et les fonctions nécessaires pour programmer avec le langage GLSL.

Solutions des exercices

> **Solution** n°1

Exercice p. 11

Placer les actions ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

NVIDIA GeForce 3 (2001)	DirectX 9.0: VS, PS 2.0	OpenGL
ATI Radeon 8500, Microsoft Xbox, NVIDIA GeForce 4 Titanium	Plus de formats des données : FPU,....	Fonctionnalités équivalentes via des extensions
OpenGL: extension ARB_vertex_program (universal)	La taille de programme a augmenté : 256 instructions	
Premier GPU programmable, DirectX 8.0: VS,PS 1.0-1.1	Variables uniforms : 256 vecteurs	

> **Solution** n°2

Exercice p. 12

Exercice

Le mélange de maillage se fait au niveau de :

- ○ Geometry shader
- ◉ Vertex shader
- ○ Fragment shader

Exercice

Un vertex shader est exécuté une et une seule fois pour chaque primitive

- ○ Vrai

- ⦿ Faux

Exercice

L'illumination par pixel se fait au niveau de :

- ○ Vertex shader
- ⦿ Fragment shader
- ○ Geometry shader

> **Solution** n°3

Exercice p. 17

Placer les tâches ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

Assembleur	**Cg**	**GLSL**	**HLSL**
Plus faciles à utiliser (extensions d'OpenGL).	Il est indépendant de la plateforme DirectX ou OpenGL.	développé pour l'API graphique OpenGL et webGL	Il est utilisé uniquement pour les applications de DirectX.
aide à comprendre le fonctionnement des cartes graphiques (ATI's r300, NVIDIA's nv30).	Il prend en charge des différents types de GPU.	Offre une grande souplesse de programmation des shaders.	Rapide
aident à comprendre les APIs haut-niveau (Cg, HLSL, ...).	Syntaxe, opérateurs et fonctions ont dérivé de C/C++.		Syntaxe similaire à Cg.

> **Solution** n°4

Exercice p. 23

Exercice

Quels sont les qualifiers de langage GLSL :

- ☑ uniform

- ☐ vector
- ☑ attribute
- ☑ varying
- ☐ mat
- ☐ struct

Exercice

Les vecteurs de langage GLSL sont de type :

- ☐ char
- ☑ int
- ☑ float
- ☐ long
- ☑ bool

Exercice

Les types de données fournis par le GLSL sont:

- ☑ scalairs
- ☑ vecteurs
- ☐ chaîne de caractère

Exercice

Le qualifier in est utilisé dans :

- ☑ passage par valeur
- ☐ passage par adresse
- ☑ interpolations des données entre le vertex et le fragment shader

> **Solution** n°5

Exercice p. 25

Placer les tâches ci-dessous dans les endroits appropriés par l'opération"glisser-déposer puis cliquer sur correction"

OpenGL 2.0/GLSL 1.1	**OpenGL 3.0/GLSL 3.0**	**OpenGL 4.0/**

Les requêtes d'occultation qui permettent de demander combien de pixels un élément de scène particulier occuperait s'il était affiché.	Ensemble de variables uniforms.	**GLSL 4.0**
Les VBOp.34 p.35 qui permettent de stocker des tableaux de vertex dans la mémoire graphique afin de réduire la communication nécessaire entre le processeur et la carte.	Les FBOp.34 p.35 qui permettent d'indiquer ou le rendu OpenGL sera effectué.	Amélioration de geometry shader.
Les opérations de stencil offrent un meilleur moyen pour l'ombrage.	L'utilisation de geometry shader.	Apparition de compute shader.
		Apparition de tessellation shader.

> **Solution** n°6

Exercice p. 26

Quel est le langage utilisé pour la plateforme de jeux Unity ?

Cg

> **Solution** n°7

Exercice p. 26

La méthode qui permet d'envoyer des données 3D (position, normal, couleur, coordonnées de texture, ...) au GPU est le :

VBO

> **Solution** n°8

Exercice p. 26

Ensemble de buffers logiques et d'états définissant l'emplacement où le rendu OpenGL sera effectué dans le :

FBO

> **Solution** n°9

Exercice p. 26

Les qualifiers qui permettent le passage par valeur ou par adresse dans une fonction sont :

- ☐ uniform
- ☑ in

- [x] out
- [ ] varying
- [x] inout
- [ ] attribute
- [x] const

> **Solution** n°10

Exercice p. 26

Complétez les trous suivants :

GLSL (OpenGL `Shading Language`): est un langage permettant la programmation `GPU` des scènes `OpenGL`. Le langage GLSL est basé sur le langage `C`

> **Solution** n°11

Exercice p. 27

Complétez les trous suivants :

Les variables `uniform` servent à la communication entre le programme exécuté par le `CPU` et les `shaders` exécutés par le `GPU`

> **Solution** n°12

Exercice p. 27

Complétez les trous suivants :

Les variables `varying` servent à la communication entre le `vertex` shader exécuté et le `fragment` shader

Glossaire

FBO

Ensemble de buffers logiques et d'états définissant l'emplacement où le rendu OpenGL sera effectué.

Unity

Moteur de jeux multiplateforme, rapide et il permet de sortir les jeux sur tous les supports.

VBO

méthode qui permet d'envoyer des données 3D (position, normal, couleur, coordonnées de texture, ...) au GPU.

Abréviations

Cg : C for graphics

FBO : Frame Buffer Object

GLSL : OpenGL Shading Language.

HLSL : High-Level Shading Language

VBO : Vertex Buffer Object

Références

GPU programming (shaders) Josef Pelikán, 2016, CGG MFF UK Praha

Les shaders Alexandre Topol, Valeur C CAM & DEA MM, 2003-2004

Shaders : une introduction Alexandre Topol, Ecole nationale du jeu et des médias interactifs numériques,

Bibliographie

IV

Mike Hergaarden, *Graphics shaders,* VU Amsterdam, Janvier 2011

Mike Bailey, Steve Cunningham, *Graphics shders- theory and practice,* CRC Press, deuxième édition

Webographie

https://subscription.packtpub.com/book/hardware_and_creative/9781849698009