

OpenGL Shading Language

Christophe [Groove] Riccio

www.g-truc.net

Creation : 12/01/2005

Mise à jour : 24/02/2006

Sommaire

1. Présentation

- 1.1. A propos de ce document
- 1.2. GLSL, c'est quoi ?
- 1.3. Pourquoi ?
- 1.4. Le matos
- 1.5. Exemple de programmes

2. Les variables

- 2.1. Généralité sur le langage
- 2.2. Type de données simples
- 2.3. Type de données complexes
- 2.4. Qualificatifs

3. Les fonctions intégrées

- 3.1. Introduction
- 3.2. Fonctions trigonométriques
- 3.3. Fonctions exponentielles
- 3.4. Fonctions usuelles
- 3.5. Fonctions pour la géométrie
- 3.6. Fonction de comparaisons
- 3.7. Fonctions sur les fragments
- 3.8. Fonctions sur les nombres aléatoires

4. L'API

- 4.1. Les extensions, le langage et l'API
- 4.2. Handle et Identifiant de textures
- 4.3. Shader object
- 4.4. Compilation et édition des liens
- 4.5. Utilisation et variables
- 4.6. Conclusion

5. Place à la pratique

- 5.1. Introduction
- 5.2. Déclaration
- 5.3. Définition
- 5.4. Conclusion

1. Présentation

1.1. A propos de ce document

Ce document a pour but de décrire le langage GLSL ainsi que l'API C permettant de l'utiliser dans un programme C ou C++. Il ne s'agit en aucun cas de réaliser des effets graphiques avec GLSL. Les techniques utilisées ont pour but de présenter le langage.

Il est indispensable pour le lecteur d'avoir au minimum des notions sur l'API OpenGL notamment sur ses mécanismes d'extensions.

Ce document a été réalisé à partir du livre « OpenGL Shading Language » de Randi J. Rost, des spécifications de GLSL en Version 1.051, de celle d'OpenGL en version 2.0 et des extensions d'OpenGL.

1.2. C'est quoi ?

GLSL pour OpenGL Shading Language est un langage permettant la programmation GPU de scènes OpenGL. Il a été développé par 3D Lab et approuvé par l'ARB (Architecture Review Board), l'organisme chargé de la standardisation d'OpenGL.

La programmation GPU se pratique au moyen de deux types éléments : les vertex shader et le fragment shader. Les shaders sont généralement écrits dans des fichiers textes en dehors du code mais il est tout à fait possible d'imaginer d'autres formes tant que les shaders sont accessibles sous forme de chaînes de caractères.

Un vertex shader réalise des opérations de un vertex alors qu'un fragment shader réalise des opérations sur un fragment. Il est possible d'envoyer des informations du programme C/C++ vers le programme GLSL mais par dans le sens inverse, mise à part le résultat finale rendu à l'écran.

Précisons le vocabulaire au niveau de deux termes très utilisés : « Shader » et « Program ». Le terme « shader » indique du code GLSL, alors que le terme « program » indique soit des programmes GPU de bas niveau soit du code GLSL compilé et lié, ces deux dernières éléments n'étant pas vraiment différents.

Remarques : Vous avez certainement entendu parlé de la notion de Pixel Shader voir de Shader Model 3 mais vous êtes bien surpris car je n'en parle pas. Et bien c'est normal, rassurez vous ! En effet, tout c'est notion ce rapport exclusivement à DirectX / WGF donc elle ne nous regarde pas. Notez cependant, qu'un pixel shader DirectX est l'équivalent d'un fragment shader GLSL.

1.3. Pourquoi ?

Depuis un an (voir plus), la montée en fréquence a été bien faible et la ridicule loi de Moore a encore prouvé à quel point elle est erronée. En effet, Intel stagne à 3.4 GHz et AMD à 2.4 GHz et ce n'est pas avant un bon moment que l'on reverra les fréquences progresser rapidement. Aujourd'hui, les constructeurs sont bien conscients des limites du procédé de fabrication actuel et même si l'augmentation de la fréquence des processeurs est toujours possible en allongeant le pipeline, ceci n'augmente pas obligatoirement les performances. Intel l'a prouvé avec son P4 qui utilise à 30 positions alors que AMD utilise un pipeline à 12 positions sur son Athlon 64 pour des performances supérieures dans les jeux et le rendu 3D temps réel.

L'avenir des CPU c'est le multicore cependant multicore et 3D temps réel ne font pas bon ménage à cause de la logique de la boule d'affichage. L'idée dernière le multi-threading c'est de pouvoir découper les tâches en thread, chaque core exécutant un thread et sa tâche finie (ou non suivant la quantité de travail), il passe au thread suivant. Pour des serveurs webs, ce principe est naturel, chaque connexion crée un thread sur le serveur, la charge de travail est donc bien divisée en une infinité de tâches élémentaires. Plus il y a de cores dans un processeur, plus il y a de threads traités parallèlement. Pour un jeu ou une application 3D temps réel, le découpage en thread n'est pas simple et l'on peut parier qu'il faudra un bon moment pour que les jeux profitent pleinement des processeurs multicore.

Une solution alternative est de reporter les calculs sur les cartes graphiques qui elles continuent leurs courses à la puissance. L'idée avec

la programmation GPU est de pouvoir créer de nouveaux effets graphiques mais également et peut-être surtout de faire travailler la carte graphique à la place du CPU. En effet, alors qu'avec un CPU, il est totalement impossible de paralléliser les traitements d'un core avec un GPU c'est parfaitement possible. Un GPU est notamment composée de « pixel units ». Par exemple un modeste GF 5200 contient 4 « pixel units ». Tout se passe comme si l'écran était divisé en 4 et que chaque unité se chargeait d'un quartier. Le nombre de « pixel units » détermine en partie la vitesse d'un GPU. Le GeForce 6800 GT et le Radeon X800 XT en contient 16, ce qui explique en partie leurs bonnes performances. Il faut également compter sur la fréquence et sur la complexité de l'architecture. Par exemple, le GF 5900 possède des unités très complexe disposant de deux unités de textures par unités de pixels cependant elle ne possède que 4 unités de pixels ce qui inquite franchement sur les performances.

1.4. Le matos

Pour utiliser GLSL, il vous faut une carte graphique ATI ou nVidia, de préférence récente.

Chez nVidia, si vous avez au moins une GeForce 256, vous pouvez utiliser GLSL mais vous serez limité au vertex shader si vous n'avez pas au moins une GeForce 5200. Chez ATI, c'est nettement plus simple. Il vous faut au moins un Radeon 9500 pour utiliser GLSL que ce soit en vertex ou en fragment shader.

Notons que si vous disposez d'une carte nVidia au moins du type GeForce 256, vous pouvez utiliser GLSL dans toute son oeuvre via un logiciel d'émulation nommée « nvemulate ».

Remarquons enfin que ce n'est pas par ce que vous avez une carte qui gère GLSL que vous serez a laissez dans la réalisation de vos programmes. Dès que les effets commencent à se complifier, surtout au niveau des fragments shaders, le frame rate ne sera plus acceptable.

1.5. Exemple de programmes

Afin de vous donner un premier aperçu de ce qu'est sont des shaders voici un exemple élémentaire réalisant du multitexturing de type « modulate ».

Le vertex shader :

```
void main ()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Le fragment shader :

```
uniform sampler2D Texture1;
uniform sampler2D Texture2;

void main ()
{
    gl_FragColor = texture2D (Texture1, gl_TexCoord[0].st);
    gl_FragColor *= texture2D (Texture2, gl_TexCoord[0].st);
}
```

Les programmes sont vraiment petits et simples, pour autant tous ne le sont pas. Cependant, la plus part des effets réalisés avec des shaders

sont plus simples à produire que leur équivalent C/C++ et OpenGL. Par exemple, l'utilisation de `GL_ARB_texture_env_combine` est une vraie torture de l'esprit dans le programme C/C++ alors que dans un shader les possibilités offertes par cette extension se réalise naturellement et avec une puissance bien supérieur. GLSL offre une parfaite maîtrise de chaque vertex et fragment ce qui permet de réaliser des effets originaux, totalement impossible à réaliser sans shader.

2. Les variables

2.1. Généralité sur le langage

GLSL reprend beaucoup des langages C et C++. La syntaxe est proche de celle de C, le point d'entrée est la fonction `main`, le pré processeur, etc. Depuis le C++, GLSL a repris le principe de surcharge des fonctions, le type `bool`, la façon de déclarer les variables, le principe des constructeurs, etc. Le langage GLSL fonctionne avec la même logique que le C et le C++ au niveau de la compilation et du link. Tout programme doit subir ces opérations pour être exécuté. Notez cependant, qu'il n'est pas possible de récupérer un fichier issu de la compilation et du link mais l'ARB travail sur ce sujet. De plus, le compilateur est intégré dans les drivers de la carte graphique et son utilisation se fait au travers de l'API OpenGL.

L'utilisation d'un langage de programmation GPU a de fortes implications sur le déroulement d'un programme OpenGL. Le rendu du fog, la transparence ou encore l'éclairage ne sont plus pris en compte. Il faudra reprogrammer le rendu de ces différentes fonctionnalités dans le vertex shader voir le fragment shader.

2.2. Type de données simples

GLSL support les types `void`, `bool`, `int`, `float` ainsi que d'autres type de base dédié à la programme 3D temps réel :

5. `vec2` : un vecteur de `float` à deux dimensions
6. `vec3` : un vecteur de `float` à trois dimensions
7. `vec4` : un vecteur de `float` à quatre dimensions
8. `bvec2` : un vecteur de `bool` à deux dimensions
9. `bvec3` : un vecteur de `bool` à trois dimensions
10. `bvec4` : un vecteur de `bool` à quatre dimensions
11. `ivec2` : un vecteur de `int` à deux dimensions
- `ivec3` : un vecteur de `int` à trois dimensions
12. `ivec4` : un vecteur de `int` à quatre dimensions
13. `mat2` : une matrice de `float` à deux dimensions
14. `mat3` : une matrice de `float` à trois dimensions
15. `mat4` : une matrice de `float` à quatre dimensions
16. `sampler1D` : Accès à une texture à une dimension
17. `sampler2D` : Accès à une texture à deux dimensions
18. `sampler3D` : Accès à une texture à trois dimensions
19. `samplerCube` : Accès à une texture de type cube map
20. `sampler1DShadow` : Accès à une texture de profondeur à une dimension
21. `sampler2DShadow` : Accès à une texture de profondeur à deux dimensions

Remarques :

22. les types de données `sampler*` n'indique pas un identifiant d'objet de textures mais simplement l'unité de texture utilisé. C'est l'équivalent de `GL_TEXTURE0_ARB` dans cette instruction :

```
glMultiTexCoord2fARB (GL_TEXTURE0_ARB, 0, 0);
```

23. Les extensions OpenGL peuvent étendre le langage de nouveaux types de données. Par exemple, l'extension `GL_ARB_texture_rectangle` d'OpenGL 2.0 ajoute les types `samplerRect` et `samplerRectShadow`.
24. Le type `void` n'est utilisé que pour les fonctions qui ne retournent rien.

2.3. Type de données complexes

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

```
struct SVertex
{
    vec3 Normal ;
    vec2 TexCoord ;
    vec3 Position ;
};
```

Remarques :

25. Le mot clé `typedef` a été réservé mais n'est pas disponible.
26. La définition de variables de types structures se réalise ainsi :
- ```
SVertex UnVertex;
```

GLSL permet la création de tableaux qui fonctionnent globalement comme les tableaux C mais avec une souplesse sur sa taille beaucoup plus grande. Il est ainsi possible de déclarer un tableau sans indiquer sa taille puis on adapte sa taille en fonction de nos besoins :

```
vec3 Vertex[];
Vertex[3] = vec4 (1.0); //Le tableau dispose à présent de 4 éléments.
Vertex[7] = 2.0; //Le tableau dispose à présent de 8 éléments.
```

## 2.4. Qualificatifs

Les variables GLSL peuvent être déclarées avec quatre qualificatifs différents indiquant la façon de communiquer des variables avec d'autres programmes.

Le qualificatif « `const` » fonctionne de la même manière qu'en C. La valeur de la variable n'est pas modifiable lors de l'exécution. Cette variable est accessible seulement par le programme qui la déclare.

Le qualificatif « `varying` » offre une solution pour envoyer des informations du vertex program au fragment program. Si la valeur `GL_SMOOTH` a été passée à `glShadeModel ()` dans le programme C/C++ alors la valeur de la variable dans le fragment program sera interpolé suivant la valeur qu'elle prend pour chaque vertex qui génère la surface dont le fragment est issu.

Le qualificatif « `uniform` » indique que la valeur de la variable est valable et identique pour tous les vertrices et tous les fragments. La valeur de la variable est transmise par le programme C/C++.

Le qualificatif « `attrib` » indique que la valeur de la variable est valable pour un vertex uniquement. La valeur de la variable est transmise par le programme C/C++.

Il existe un grand nombre de variables prédéfinis. En voici une liste :

Liste des attribues prédéfinis pour chaque vertex :

```
attribute vec4 gl_Color; // Couleur du vertex, donnée par glColor*
```

```

attribute vec4 gl_SecondaryColor; // Couleur secondaire du vertex, donnée
par glSecondaryColor* définit par l'extension GL_EXT_secondary_color.
attribute vec3 gl_Normal; // Normal du vertex, donnée par glNormal*
attribute vec4 gl_Vertex; // Position du vertex, donnée par glVertex*
attribute vec4 gl_MultiTexCoord0; // Coordonnées de la texture utilisé par
la 1ère unité
attribute vec4 gl_MultiTexCoord1; // Coordonnées de la texture utilisé par
la 2ème unité
attribute vec4 gl_MultiTexCoord2; // Coordonnées de la texture utilisé par
la 3ème unité
attribute vec4 gl_MultiTexCoord3; // Coordonnées de la texture utilisé par
la 4ème unité
attribute vec4 gl_MultiTexCoord4; // Coordonnées de la texture utilisé par
la 5ème unité
attribute vec4 gl_MultiTexCoord5; // Coordonnées de la texture utilisé par
la 6ème unité
attribute vec4 gl_MultiTexCoord6; // Coordonnées de la texture utilisé par
la 7ème unité
attribute vec4 gl_MultiTexCoord7; // Coordonnées de la texture utilisé par
la 8ème unité
attribute float gl_FogCoord; // Valeur du fog du vertex, donnée par
glFogCoord* définit par l'extension GL_EXT_fog_coord

```

Listes des variables définies pour chaque fragment :

```

vec4 gl_FragCoord; // Coordonnée du fragment par rapport à la fenêtre. La
valeur en z est identique à celle donnée par gl_FragDepth et la valeur w
permet d'exprimer la position en coordonnées homogènes. Cette variable est
accessible en lecture seule.
bool gl_FrontFacing; // Indique si le fragment appartient à la face avant
d'un polygone. Cette variable est accessible en lecture seule.
vec4 gl_FragColor; // Couleur du fragment dans le tampon chromatique
float gl_FragDepth; // Valeur de la profondeur du fragment dans le Z-Buffer

```

Listes des constantes définies pour chaque vertex et chaque fragment, les valeurs indiquent sont les valeurs minimums, les véritables valeurs dépendent des cartes graphiques utilisées :

```

const int gl_MaxLights = 8; // Nombre maximum de lampes OpenGL
const int gl_MaxClipPlanes = 6; // Nombre maximum de plans de clipping
const int gl_MaxTextureUnits = 2; // Nombre d'unités de textures de la
carte graphique
const int gl_MaxTextureCoordsARB = 2; // Nombre de coordonnées de textures
disponibles
const int gl_MaxVertexAttributesGL2 = 16; // Nombre d'attribues maximums
par vertex.
const int gl_MaxVertexUniformFloatsGL2 = 512; // Nombre de variables
uniformes par vertex program
const int gl_MaxVaryingFloatsGL2 = 32; // Nombre de variables varying
maximum par programme
const int gl_MaxVertexTextureUnitsGL2 = 1; // Nombre d'unités de vertex de
la carte graphique
const int gl_MaxFragmentTextureUnitsGL2 = 2; // Nombre d'unités de fragment
de la carte graphique
const int gl_MaxFragmentUniformFloatsGL2 = 64; // Nombre de variables
uniformes par fragment program

```

Listes des variables varying définies pour chaque vertex et fragment:

```

varying vec4 gl_FrontColor; // Couleur sur la face avant
varying vec4 gl_BackColor; // Couleur sur la face arrière
varying vec4 gl_FrontSecondaryColor; // Couleur secondaire sur la face
avant

```

```

varying vec4 gl_BackSecondaryColor; // Couleur secondaire sur la face avant
varying vec4 gl_TexCoord[]; // Coordonnées de textures, au maximum autant
qu'indiqué par gl_MaxTextureCoordsARB.
varying float gl_FogFragCoord; // Valeur du fog du fragment

```

Il existe un très grand nombre de variables uniformes correspondant à chacun des paramétrages réalisables avec les commandes OpenGL. La plus part des variables uniformes prédéfinies sont définis sous la forme de structures telle que la suivante produite à partir de l'extension `GL_ARB_point_parameters`:

```

struct gl_PointParameters
{
 float size; // Taille d'une point définit par glPointSize
 float sizeMin; //Taille minimum d'un point définit par
glPointParameterfARB (GL_POINT_SIZE_MIN_ARB, min) grâce à l'extension
GL_ARB_point_parameters
 float sizeMax; //Taille maximal d'un point définit par
glPointParameterfARB (GL_POINT_SIZE_MAX_ARB, min) grâce à l'extension
GL_ARB_point_parameters
 float fadeThresholdSize;
 float distanceConstantAttenuation; //Premier coefficient d'atténuation
de la taille d'un point définit par glPointParameterfvARB
(GL_POINT_DISTANCE_ATTENUATION_ARB, Quadratic3) grâce à l'extension
GL_ARB_point_parameters
 float distanceLinearAttenuation; //Deuxième coefficient d'atténuation
de la taille d'un point float distanceQuadraticAttenuation;
//Troisième coefficient d'atténuation de la taille d'un point
};
uniform gl_PointParameters gl_Point;

```

Pour en connaître la liste complète, reportez vous à la page 42 des spécifications de GLSL en version 1.051.

## 3. Les fonctions intégrées

### 3.1. Introduction

Ce langage propose un large ensemble de fonctions intégrées. Bien que beaucoup de ces fonctions soient simples à reprogrammer, Il est important de les connaître car elles sont une bonne source d'optimisations. En effet, certaines fonctions sont câblées dans les cartes graphiques. Par exemple, la fonction cosinus est câblée dans les GeForce FX 5XXX ce qui promet des performances exceptionnelles sur une fonction demandant beaucoup de calculs.

### 3.2. Fonctions trigonométriques

GLSL propose un ensemble de fonctions permettant de manipuler les nombres, les espaces vectoriels et les textures. Concernant la trigonométrie, les fonctions **cos** (cosinus), **sin** (sinus), **tan** (tangente), **acos** (arc cosinus), **asin** (arc sinus), **atan** (arc tangente) sont disponibles. Il y a également les fonctions **radians** et **degrees** pour respectivement la conversion de degré en radian et radian en degré. Toutes ces fonctions sont valables pour les **float**, les **vec2**, les **vec3** et les **vec4**.

### 3.3. Fonctions exponentielles

GLSL définit également des fonctions pour le calcul d'exponentiels (**exp2**) de logarithmes (**log2**) de racines (**sqrt**) de racines inverses (**inversesqrt**) et de puissances (**pow**). Mise à part **pow**, toutes les fonctions prennent un seul paramètre de type **float**, **vec2**, **vec3** ou **vec4**. La fonction **pow** prend deux paramètres, le premier est la valeur dont l'on veut la puissance et le deuxième est la valeur de la puissance.

### 3.4. Fonctions usuelles

Il existe plusieurs fonctions pour la manipulation des nombres, en voici la liste sachant que **type** peut-être un **float**, un **vec2**, un **vec3** ou un **vec4** :

- **type abs (type x)** : Retourne la valeur absolue d'un nombre
  - **type sign (type x)** : Indique le signe d'un nombre, retourne 1.0 si le nombre est positif et -1.0 si le nombre est négatif
  - **type floor (type x)** : Fournit (sous la forme du type) le plus grand entier qui ne soit pas supérieur à **x**.
  - **type ceil (type x)** : Fournit (sous la forme du type) le plus petit entier qui ne soit pas inférieur à **x**.
  - **type fract (type x)** : Retourne **x - floor (x)**.
  - **type mod (type x, float y)** : Retourne **x - y \* floor (x / y)**
  - **type mod (type x, type y)** : Retourne **x - y \* floor (x / y)**
  - **type min (type x, type y)** : Retourne la plus petite valeur entre **x** et **y**
  - **type min (type x, float y)** : Retourne la plus petite valeur entre **x** et **y**
  - **type max (type x, type y)** : Retourne la plus grande valeur entre **x** et **y**
  - **type max (type x, float y)** : Retourne la plus grande valeur entre **x** et **y**
  - **type clamp (type x, type min, type max)** : Retourne **min (max (x, min), max)**
  - **type clamp (type x, float min, float max)** : Retourne **min (max (x, min), max)**
  - **type mix (type x, type y, type a)** : Permet la réalisation d'interpolation linéaire en retournant **x \* (1 - a) + y \* a**
  - **type mix (type x, type y, float a)** : Permet la réalisation d'interpolation linéaire en retournant **x \* (1 - a) + y \* a**
  - **type step (type bord, type x)** : Retourne 0.0 si **x** est inférieur ou égale à **bord** et sinon 1.0.
  - **type step (type bord, float x)** : Retourne 0.0 si **x** est inférieur ou égale à **bord** et sinon 1.0.
  - **type smoothstep (type bord0, type bord1, type x)** : Identique à l'autre variante de **smoothstep** mais avec des valeurs de **x** différente pour chaque composante de **bord0** et **bord1**
  - **type smoothstep (type bord0, type bord1, float x)** : Retourne 0.0 si **x** <= **bord0** et 1.0 si **x** >= **bord1** sinon retourne une valeur interpolé.
- Equivalent aux calculs suivants :

```
type t;
t = clamp ((x - edge0) / (edge1 - edge0), 0, 1);
return t * t * (3 - 2 * t);
```

### 3.5. Fonctions pour la géométrie

GLSL a prévu des fonctions pour les calculs liés à la géométrie, voici la liste des fonctions disponibles :

- **float length (type)** : Retourne la norme d'un vecteur
- **float distance (type p0, type p1)** : Retourne la distance entre p0 et p1 ainsi : `length (p0 - p1)`
- **float dot (type x, type y)** : Retourne le produit scalaire de **x** et **y**
- **vec3 cross (vec3 x, vec3 y)** : Retourne le produit vectoriel de **x** et **y**
- **type normalize (type x)** : Retourne un vecteur de même direction que **x** mais de norme égale à 1.
- **vec4 ftransform ()** : Garantie que la transformation effectuée par un vertex shader est identique à celle effectuée dans le code OpenGL.
- **type reflect (type I, type N)** : Retourne la direction d'un rayon réfléchi par une surface de normal **N** et de rayon incident **I**
- **type faceforward (type N, type I, type Nref)** : Permet de déterminer les faces avant au moyen du produit scalaire :  
`return dot (Nref, I) < 0 ? N : -N`
- **mat\* matrixCompMult (mat\* a, mat\* b)** : Multiplication terme à terme de deux matrices.

### 3.6. Fonctions de comparaisons

GLSL définit un ensemble de fonctions de comparaisons dédiées

- **bvec\* lessThan(vec\* a, vec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a < b$
- **bvec\* lessThan(ivec\* a, ivec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a < b$
- **bvec\* lessThanEqual(vec a, vec b)** : Retourne pour chaque composante la valeur de la comparaison  $a \leq b$
- **bvec\* lessThanEqual(ivec a, ivec b)** : Retourne pour chaque composante la valeur de la comparaison  $a \leq b$
- **bvec\* greaterThan(vec\* a, vec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a > b$
- **bvec\* greaterThan(ivec\* a, ivec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a > b$
- **bvec\* greaterThanEqual(vec\* a, vec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a \geq b$
- **bvec\* greaterThanEqual(ivec\* a, ivec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a \geq b$
- **bvec\* equal(vec\* a, vec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a == b$
- **bvec\* equal(ivec\* a, ivec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a == b$
- **bvec\* equal(bvec\* a, bvec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a == b$
- **bvec\* notEqual(vec\* a, vec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a != b$
- **bvec\* notEqual(ivec\* a, ivec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a != b$
- **bvec\* notEqual(bvec\* a, bvec\* b)** : Retourne pour chaque composante la valeur de la comparaison  $a != b$
- **bool any (bvec\* a)** : Retourne **true** si l'une des composantes de **a** sont **true**
- **bool all (bvec\* a)** : Retourne **true** si toutes composantes de **a** sont **true**
- **bvec\* not (bvec\* a)** : Retourne le complément pour chaque composante de **a**

## 3.7. Fonctions sur les textures

Les fonctions sur les textures permettent de consulter la valeur des texels en fonction de la valeur des **sampler\*** et des coordonnées de textures. Les fonctions disponibles a cet effet sont les suivantes :

- **vec4 texture\*D[Proj](sampler\* sampler, float coord [, float bias])**
- **vec4 texture\*D[Proj]Lod(sampler\* sampler, float coord , float lod)**
- **vec4 textureCube(samplerCube sampler, vec3 coord [, float bias])**
- **vec4 textureCubeLod(samplerCube sampler, vec3 coord , float lod)**
- **vec4 shadow\*D[Proj](sampler\* sampler, float coord [, float bias])**
- **vec4 shadow\*D[Proj]Lod(sampler\* sampler, float coord, float lod)**

Le paramètre **bias** est optionnel et n'est disponible que pour les fragments shaders. Il permet de personnaliser le calcul du niveau de détail. S'il n'est pas précisé alors les calculs seront réalisés avec la méthode par défaut.

Les fonctions ayant le suffixe « **Lod** » ne sont utilisable que dans les vertex shaders. Le paramètre « **lod** » est alors utilisé pour le calcul du niveau de détails.

Les fonctions ayant le suffixe « **Proj** » on leur composante divisé par la dernière composante.

## 3.8. Fonctions sur les fragments

Certaines fonctions sont dédiés aux fragment programs. Les fonctions **dFdx**, **dFdy** et **fwidth** permettent d'approximer le calcul de dérivées.

## 3.9. Fonctions sur les nombres aléatoires

GLSL définit un ensemble de fonctions pour la recherche de nombres aléatoires. Toutes les fonctions retournes des valeurs pour chaque composante comprises dans l'intervalle [-1.0, 1.0]. Le nombre dépend de la valeur passée en paramètre. Voici la liste de ces fonctions :

- **float noise1(type x)** : Retourne une valeur aléatoire à une dimension basée sur x
- **vec2 noise2(type x)** : Retourne une valeur aléatoire à deux dimensions basée sur x
- **vec3 noise3(type x)** : Retourne une valeur aléatoire à trois dimensions basée sur x
- **vec4 noise4(type x)** : Retourne une valeur aléatoire à quatre dimensions basée sur x

# 4. L'API

## 4.1. Les extensions, le langage et l'API

Pour utiliser GLSL, les drivers de la carte graphique doivent supporter 4 extensions. La première nommée **GL\_ARB\_shading\_language\_100** définit le langage lui-même, tel que nous l'avons vue sur dans les deux précédant tutoriaux. Celle-ci n'est pas suffisant car nous avons également besoin de **GL\_ARB\_shader\_objects** qui définit un nouvel objet OpenGL à la

manière des objets de textures, `GL_ARB_vertex_shader` pour la programmation de vertex shader et `GL_ARB_fragment_shader` pour la programmation de fragment shader. Notez qu'avec OpenGL 2.0, ces extensions ont été intégrées au corps de la spécification. En conséquence, avec OpenGL 2.0, il n'est pas nécessaire d'indiquer 'ARB' à la fin des fonctions. Cependant, les drivers des cartes ne sont pas encore totalement OpenGL 2.0 (En tout cas chez nVidia), donc toutes les fonctions sont présentés sous la forme des extensions.

## 4.2. Shader object

Les objects de shader sont des objets OpenGL mais contrairement aux objets de textures, il n'utilise pas les termes habituelles de l'API OpenGL, donc pas de fonctions 'gen', 'is' et 'bind' mais des fonctions 'create', 'use', 'attach' et 'get'. Notons que l'ARB a statué que l'ancienne méthode (à la manière des objets de textures) pour gérer les objets étaient recommandés pour la définition des extensions. Nous pouvons donc supposer de la méthode utilisé pour les objets de shaders ne se développera pas, ce qui laisse septique sur l'intérêt de cette méthode pour GLSL.

## 4.3. Compilation et édition des liens

Pour rappel, avec GLSL il faut dissocier deux notions : shader et program. Un program contient au plus un vertex shader et un fragment shader. Nous pouvons comparer les shaders aux fichiers compilés (.o avec GCC, .obj avec VC) et le program à l'exécutable généré après l'édition des liens des fichiers compilés. Ainsi la fonction `glCreateShaderObjectARB` crée un objet de shaders et `glCreateProgramObjectARB` crée un objet de program. Pour lire le code source nous utilisons la fonction `glShaderSourceARB` puis nous compilons le shader avec `glCompileShaderARB`. Nous indiquons au programme les shaders compilés à l'aide de la fonction `glAttachObjectARB` puis nous réalisons l'édition des liens avec `glLinkProgramARB`. Enfin, il est possible de détruire les deux types d'objets avec `glDeleteObjectARB`. Notons que lorsque les shaders compilés ont été lié nous pouvons les détruire.

```
GLhandleARB glCreateProgramObjectARB()
GLhandleARB glCreateShaderObjectARB(GLenum shaderType)
void glShaderSourceARB(GLhandleARB shader, GLuint nstrings, const GLcharARB
**strings, GLint *lengths)
void glCompileShaderARB(GLhandleARB shader)
void glAttachObjectARB(GLhandleARB program, GLhandleARB shader)
void glLinkProgramARB(GLhandleARB program)
void glDeleteObjectARB(GLhandleARB object)
```

Notons qu'il y a deux fonctions différentes pour créer les objets et une seule pour les détruire ce qui n'est pas évident d'un point de vue sémantique ou par rapport à nos habitudes avec OpenGL.

## 4.4. Utilisation et variables

Lorsque l'on souhaite qu'une série de vertrices subissent les effets des shaders GLSL, il faut utiliser la fonction `glUseProgramObjectARB`. Elle doit être utilisé en dehors d'un bloc `glBegin` / `glEnd` et nous lui passons l'objet de program en paramètre.

Lors du rendu de la scène nous pouvons envoyer des données personnalisées du programme C/C++ vers le programme GLSL. Soit nous

envoyons des variables valables pour tous les vertrices, les variables uniforms, soit nous envoyons des variables pour chaque vertex, les vertex attributes. Par exemple lorsque l'on appelle la fonction `glColor*` nous envoyons une variable aux programmes GLSL. Cette variable est prédéfini et porte le nom `gl_Color`, c'est un vertex attribute prédéfini. Admettons que l'on souhaite faire de la réflexion de lumière et que pour chaque vertex nous souhaitons définir un coefficient de réflexion différent. Comment envoyer ce coefficient au vertex shader ? En créant un vertex attribute personnalisé. Pour cela nous récupérerons l'identifiant de la variable incriminée avec `glGetAttribLocationARB` puis nous utilisons les fonctions `glVertexAttrib*ARB`. Ces fonctions prennent en paramètre des vecteurs de 1 à 4 valeurs, soit par tableau soit par valeur et pour beaucoup de types de bases possibles. Les fonctions `glVertexAttrib*ARB` doivent être utilisées dans un bloc `glBegin / glEnd`. Il est également possible d'envoyer des valeurs qui sont normalisées au passage, avec les fonctions `glVertexAttrib*N*ARB`. Enfin, l'envoi des informations à l'aide des vertex array ou des VBO est possible avec la fonction `glVertexAttribPointerARB`.

Il est également possible que nous souhaitons envoyer des données identiques pour tous les vertrices, par exemple, pour indiquer le temps. Pour cela nous utilisons les variables uniforms. Avec la fonction `glGetUniformLocationARB` nous récupérerons l'identifiant de la variable puis avec les fonctions `glUniform*ARB` nous passons les valeurs. Notons que pour utiliser une texture dans un vertex ou fragment shader il faut obligatoirement utiliser les fonctions `glUniformliARB` ou `glUniformvARB`. Enfin, le passage de matrices est possible avec `glUniformMatrix*vARB`.

```

GLuint glGetAttribLocationARB(GLhandleARB program, const GLcharARB* name)
void glVertexAttrib*ARB(GLuint index, ***)
void glVertexAttrib*N*ARB(GLuint index, ***)
void glVertexAttribPointerARB(GLuint index, GLint size, GLenum type,
GLboolean normalized, GLsizei stride, const GLvoid * pointer)
GLuint glGetUniformLocationARB(GLhandleARB program, const GLcharARB* name)
void glUniform*ARB(GLint location, ***)
void glUniformMatrix*vARB(GLint location, GLuint count, GLtranspose, const
GLfloat *v);

```

## 4.5. Conclusions

Voici cette partie terminer. La description de l'API n'est pas complète mais les principales informations sont inscrites ici. Elles nous permettrons de comprendre l'implémentation d'une classe nous aidant à gérer tout ça dans la partie 5.

# 5. Place à la pratique

## 5.1. Introduction

Il est tout a fait possible d'utiliser les extensions concernant les shaders telles quelles. Cependant, l'encapsulation a cette faculté de simplifier la programmation. C'est pourquoi nous allons réaliser une classe pour générer des shaders qui aura comme faculté de nous offrir un support minimal.

## 5.2. Déclaration

Voici, la déclaration de la classe que nous allons construire :

```

class CShader
{
private:
 // Sauvegarde le l'objet OpenGL identifiant le programme
 Int m_uiShaderProg;

 // Chargement des fichiers textes contenant le code GLSL
 char* _Load(const char* szFilename);

public:
 CShader();
 ~CShader();

 bool Load(const char* szVertex, const char* szPixel = 0);
 // Activation du shader
 void Enable() const;
 // Deactivation du shader
 void Disable() const;
 // Récupération de l'identifiant d'une variable uniform
 unsigned int Uniform(const char* szNameVar) const;
 // Récupération de l'identifiant d'une variable d'attribut
 unsigned int attrib(const char* szNameVar) const;
};

```

Lors de l'utilisation cette classe, la fonction Load sera appelé en premier pour charger les shaders du programme. Cependant, il est possible de n'utiliser qu'un Vertex Shader seul en n'indiquant que le chemin du fichier de ce shader à la fonction Load. Il est alors possible de récupérer les identifiants des variables uniformes et d'attributs. Ensuite, s'il l'on veut appliquer le programme GLSL à un groupe de sommets il faut activer le programme avant, par exemple, un bloc « glBegin() / glEnd() » puis le désactiver à la fin de ce bloc au moyen des fonctions « Enable() » et « Disable() ». L'indication des valeurs prise par les variables d'attributs et uniformes ne peuvent être fait que entre ces deux fonctions mais il ne faut pas perdre de vue que les variables uniformes sont valables pour un groupe de sommets et les variables d'attributs le sont pour chaque sommet. Ainsi, il est interdit d'exécuter une initialisation d'une variable uniform dans un bloc « glBegin() / glEnd() » alors que c'est naturellement fessable pour les variables d'attributs.

### 5.3. Définition

Nous savons à présent à quoi ressemble notre classe, explorons son âme, le core des fonctions.

```

// Chargement et compilation des shaders
bool CShader::Load(const char* szVertex, const char* szPixel)
{
 // Identifiant du succès de la compilation des shaders
 GLint iResult;

 // Log pour les erreurs de compilations
 char szBuffer[BUFFER];
 memset(szBuffer, '\\0', BUFFER);

 // Pointeurs sur les sources des shaders (Vertex et Fragment)
 const char* szShaderSources[2];

 // Création d'un objet "program"
 m_uiShaderProg = glCreateProgramObjectARB();

```

```

// Vertex shader
// Chargement du source du vertex shader
szShaderSources[0] = _Load(szVertex);
// Création d'un objet OpenGL vertex shader.
GLhandleARB ShaderVert =
glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
// Indique le code source à l'objet vertex shader.
glShaderSourceARB(ShaderVert, 1, &szShaderSources[0], 0);
// Compilation du vertex shader
glCompileShaderARB(ShaderVert);
// Liaison du programme compilé avec l'objet programme
glAttachObjectARB(m_uiShaderProg, ShaderVert);
delete[] szShaderSources[0];

// Récupération du log de compilation
glGetObjectParameterivARB(ShaderVert, GL_OBJECT_COMPILE_STATUS_ARB,
&iResult);
if (iResult == GL_FALSE)
{
 glGetInfoLogARB (ShaderVert, sizeof (szBuffer), 0, szBuffer);
 printf("%s\n", szBuffer);
 return false;
}

// Suppression de l'objet vertex shader
glDeleteObjectARB(ShaderVert);

// Fragment shader : Equivalent au vertex shader
if (szPixel)
{
 szShaderSources[1] = _Load(szPixel);
 GLhandleARB ShaderFrag =
glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
 glShaderSourceARB (ShaderFrag, 1, &szShaderSources[1], 0);
 glCompileShaderARB (ShaderFrag);
 glAttachObjectARB (m_uiShaderProg, ShaderFrag);
 glDeleteObjectARB (ShaderFrag);
 delete[] szShaderSources[1];

glGetObjectParameterivARB(ShaderFrag, GL_OBJECT_COMPILE_STATUS_ARB,
&iResult);
if (iResult == GL_FALSE)
{
 glGetInfoLogARB (ShaderFrag, sizeof (szBuffer), 0, szBuffer);
 printf("%s\n", szBuffer);
 return false;
}
}

// Edition des liens entre le objet compile : le vertex shader et le
fragment shader
glLinkProgramARB(m_uiShaderProg);

// Récupération du log d'édition des liens
glGetObjectParameterivARB(m_uiShaderProg, GL_OBJECT_LINK_STATUS_ARB,
&iResult);
if (iResult == GL_FALSE)
{
 glGetInfoLogARB(m_uiShaderProg, sizeof (szBuffer), 0,
szBuffer);
}

```

```

 printf ("%s\n", szBuffer);
 return false;
 }

 return true;
}

// Récupération de l'identifiant d'une variable uniform
unsigned int CShader::Uniform(const char* szNameVar) const
{
 return glGetUniformLocationARB (m_uiShaderProg, szNameVar);
}

// Récupération de l'identifiant d'une variable d'attribut
unsigned int CShader::Attrib(const char* szNameVar) const
{
 return glGetAttribLocationARB (m_uiShaderProg, szNameVar);
}

// Activation du programme GLSL
void CShader::Enable() const
{
 glUseProgramObjectARB(m_uiShaderProg);
}

// Désactivation du programme GLSL
void CShader::Disable() const
{
 glUseProgramObjectARB (0);
}

```

## 5.4. Conclusion

Toutes les fonctions ne sont pas définies ici, vous trouverez la classe au complet dans l'archive jointe. J'ai également intégré un programme d'exemple afin que vous vous fassiez un peu les dents avec GLSL. Il s'agit d'un simple cube avec multitexture et éclairage dans GLSL.

Ainsi se termine cette série d'articles. Elle n'est certainement pas sans reproche donc si vous avez des suggestions pour l'améliorer voir pour corriger des éventuelles erreurs, n'hésitez pas à me contacter.

Pour continuer dans cette voie de la connaissance, vous trouverez dans les futures versions du pack de tutoriaux OpenGL de G-Truc Creation de plus en plus exemple de code GLSL.

L'archive contenant la classe et l'exemple <http://www.g-truc.net/article/gsl.zip>, avec des projets ou makefile pour Visual C++ 7.1, DevCpp, MinGW et GCC.