Tutorial Cg (OpenGL)

25/04/2005 by B0b.K !113r

Qu'est ce que Cg?

De nos jours, les cartes graphiques évoluent très rapidement et coder un shader en assembleur est très laborieux. En plus, le langage assembleur peut être spécifique au matériel utilisé. Il faut donc un langage de haut niveau pour nous simplifier la tâche. Le Cg (C for graphics), langage de shader de haut fait abstraction du type de GPU de la carte graphique.

Avantages du Cg :

- ⇒ Le compilateur optimise le code automatiquement et effectue lui-même les tâches de bas niveau tel que l'allocation des registres.
- ⇒ Un shader écrit en langage de haut niveau est bien plus facile à lire qu'un code assembleur
- ⇒ Les shaders écrits en langage de haut niveau sont portables

Le Cg porte bien son nom, car il a la syntaxe du C. Mais il permet la manipulation simple de données vectorielles et matricielles. La déclaration des variables se fait de la même manière qu'en C, il est possible de créer des fonctions et de faire des appels de fonctions, les types sont quasiment identiques au types C.

Le Cg permet d'écrire des VertexShader et des PixelShader. Le code Cg peut être compilé à la volé (lors de l'exécution du programme) ou être compilé auparavant. Il existe des profils de compilation, permettant de préciser le type de langage assembleur que le compilateur doit générer, mais vous pouvez lui dire de choisir le meilleur profil à utiliser.

Les Profiles :

GL_ARB_vertex_program

- macro = CG_PROFILE_ARBVP1
- option du compilateur = -profile arbvp1

GL_ARB_fragment_program

- macro = CG_PROFILE_ARBFP1
- option du compilateur = -profile arbfpl

GL_NV_vertex_program

- macro = CG_PROFILE_VP30
- option du compilateur = -profile vp30

GL_NV_fragment_program

- macro = CG_PROFILE_FP30
- option du compilateur = -profile fp30

Installation :

Pour programmer en Cg il vous faut le toolkit de nVidia téléchargeable ici :

http://developer.nvidia.com/object/cg_toolkit.html

Il vous suffit d'installer, de linker les variables d'environnements (s'il ne l'a pas fait), et de préciser le répertoire des includes et des libs à votre compilateur.

Structure d'un programme Cg (VertexShader):

```

struct myVertex
{
    float4 position : POSITION;
    float4 color0   : COLOR0;
    float2 texcoord0 : TEXCOORD0;
};

myVertex main( fragment IN, const uniform float4x4 worldViewProj )
{
    myVertex OUT;

    OUT.position = mul( worldViewProj, IN.position );
    OUT.color0   = IN.color0;
    OUT.texcoord0 = IN.texcoord0;

    return OUT;
}

```

La fonction principale est appelée 'main' comme en C. Mais vous pouvez la nommer comme bon vous semble si vous le précisez au compilateur Cg.

Le code d'un VertexShader est semblable au code d'un PixelShader. Les données d'entrées correspondent à votre structure de Vertex dans votre programme OpenGL ou Direct3D. Par exemple ici, nous avons 'myVertex' qui a une coordonnée x,y,z et w, une couleur r,g,b et a et une coordonnée de texture u et v. Les données renvoyées par le Vertex Programme entreront dans le Fragment Programme. Les sémantiques sont : **POSITION**, **COLOR0** et **TEXCOORD0**. Elles indiquent au programme Cg le type de variables traitées. Il en existe beaucoup d'autres. Comme vous l'aurez compris les types : **float4**, **float2** et **float4x4** sont assez explicite. Un **float4** est une structure de 4 **float**, **float2** de 2 **float** et **float4x4** une matrice de 4 sur 4 **float**. Le **const** indique qu'on ne peut changer la variable. Et le mot clé important : **uniform** indique que cette donnée est envoyée par le programme qui fait appel au Shader. La fonction **mul** est implémentée en Cg. Elle multiplie les 2 composantes passées en paramètre. Cette fonction est surchargée pour multiplier n'importe quoi, dans le domaine du faisable. Par exemple 2 **float2**, 2 **float4** ou encore 2 **float4x4**... Sinon, on accède aux données de la structure comme en C.

Autre exemple pour montrer les différentes façons de traiter les entrées :

```

void main(
    in float4 iposition : POSITION,
    in float4 icolor0   : COLOR0,
    in float2 itexcoord0 : TEXCOORD0,
    const uniform float4x4 worldViewProj,
    out float4 oposition : POSITION,
    out float4 ocolor0   : COLOR0,
    out float2 otexcoord0 : TEXCOORD0,)
{
    myVertex OUT;

    oposition = mul( worldViewProj, iposition );
    ocolor0   = icolor0;
}

```

```
otexcoord0 = itexcoord0;
}
```

Ce code génère le même code assembleur que le premier.

Nous avons vu l'écriture d'un VertexShader, car il s'occupe de modifier les vertexes. Passons maintenant aux PixelShader.

Structure d'un programme Cg (PixelShader):

```
struct fragment
{
    float4 position : POSITION;
    float4 color0 : COLOR0;
    float2 texcoord0 : TEXCOORD0;
};

struct pixel
{
    float4 color : COLOR;
};

pixel main( fragment IN, uniform sampler2D testTexture )
{
    pixel OUT;

    OUT.color = lerp(tex2D( testTexture, IN.texcoord0 ) , IN.color0, 0.5);

    return OUT;
}
```

Un Fragment programme va donc prendre les résultats du Vertex programme et lui ne va renvoyer qu'une couleur TOUJOURS.

Ici dans ce programme, nous recevons les données du VertexShader précédent et une texture `sampler2D` passée en `uniform` donc par l'application principale. Nous créerons l'effet souhaité et nous renvoyons une couleur pour le pixel. `tex2D` prend en paramètre une texture et une coordonnée de texture pour pouvoir calculer la couleur du pixel. `Lerp` est une fonction d'interpolation de 2 valeurs. Elle est surchargée comme la fonction `mul` vu précédemment. Ici nous modulons la couleur de la texture avec la couleur du vertex.

Quelques mots clés :

Entré / Sortie

Si l'on n'utilise pas de structures pour spécifier les variables d'entrées et de sorties, on peut préciser qu'une variable est un paramètre d'entrée avec le mot clef

`in`

Et pour un paramètre de sortie

`out`

Types de base :

Les types de base sont :

`float half int fixed bool`

Les objets de texture sont représentés par les types suivants

`sampler1D sampler2D sampler3D samplerCUBE samplerRECT`

Pour déclarer un vecteur il suffit d'ajouter la dimension au type de base (jusqu'à 4), par exemple pour déclarer un vecteur 4 dimensions on utilisera

`float4`

De même pour les matrices, il suffit de préciser la dimension de la façon suivante

`float4x4`

Pour une matrice 4x4 dont les éléments sont des flottants.

Fonctions Cg :

Voici quelques fonctions de base du langage :

- `min( a, b )` : retourne le vecteur minimum
- `max( a, b )` : retourne le vecteur maximum
- `mul( a, b )` : multiplie a par b, a et b peuvent être des vecteurs ou des matrices
- `rsqrt( a )` : retourne l'inverse de la racine carrée du scalaire a
- `sqrt( a )` : retourne la racine carrée de a
- `length( a )` : retourne la norme du vecteur a
- `normalize( a )` : retourne un vecteur de norme unitaire et de même direction que a
- etc ...

Comment introduire l'appel à un Shader d'un programme OpenGL :

Tout d'abord les includes nécessaires :

```
#include <Cg/cg.h>
#include <Cg/cgGL.h>
```

Une fois OpenGL initialisé, on va créer un context Cg :

```
CGcontext Context = cgCreateContext() ;
```

On va ensuite récupérer les profiles des fragment et vertex programme à utiliser :

```
CGprofile profil_vertex = cgGLGetLatestProfile(CG_GL_VERTEX) ;
CGprofile profil_pixel = cgGLGetLatestProfile(CG_GL_FRAGMENT) ;
/*CG_PROFILE_UNKNOWN
```

```
CG_PROFILE_ARBVP1
CG_PROFILE_ARBFP1
CG_PROFILE_VP30
CG_PROFILE_FP30*/
```

On va ensuite créer les programmes Cg dont nous avons besoin :

```
CGprogram prog_vertex = cgCreateProgramFromFile(Context,
                                                CG_SOURCE ,
                                                const char *program_file,
                                                profil_vertex,
                                                NULL,
                                                NULL ) ;

CGprogram prog_pixel = cgCreateProgramFromFile(Context,
                                                CG_SOURCE ,
                                                const char *program_file,
                                                profil_pixel,
                                                NULL,
                                                NULL ) ;
```

Il faut ensuite charger les programmes :

```
cgGLLoadProgram(prog_vertex) ;
cgGLLoadProgram(prog_pixel) ;
```

Nous devons récupérer les noms des paramètres du programme Cg pour pouvoir leur attribuer des valeurs. Le nom : 'name' correspond au nom donné dans la fonction `uniform` ...

```
CGparameter param_vertex = cgGetNamedParameter(prog_vertex, const char * name ) ;
CGparameter param_pixel = cgGetNamedParameter(prog_pixel, const char * name ) ;
```

Différentes fonctions sont accessibles pour mettre à jour ces paramètres :

```
cgGLSetParameter*{fd}( CGparameter param, type ... ) ;
cgGLSetStateMatrixParameter( CGparameter param,
                             CGGLenum matrix,
                             CGGLenum transform ) ;
```

Pour sélectionner le vertex/pixel programme on utilise la fonction :

```
void cgGLBindProgram( CGprogram prog ) ;
```

Et pour activer ou désactiver le Shader les fonctions suivantes sont nécessaires :

```
void cgGLEnableProfile( CGprofile profile ) ;
void cgGLDisableProfile( CGprofile profile ) ;
```

Les astuces :

```
float4 position : POSITION;  
float4 color0   : COLOR0;  
float2 texcoord0 : TEXCOORD0
```

Pour accéder aux données de la position on a plusieurs moyens :

```
float pos = position[0] = position.x = coordonnée x ;
```

On peut aussi prendre plusieurs valeurs en même temps :

```
float4 pos = position.xyzw ;
```

On peut aussi reprendre plusieurs fois la même coordonnée ou les prendre dans le sens que l'on souhaite :

```
float4 pos = position.zxxy ;
```

Pour les couleurs c'est :

```
float4 color = color0.rgba ;
```

Pour les textures :

```
float2 tex = texcoord0.uv ;
```

J'espère que mon tutorial vous aidera ;)

Liens utilisés :

http://www.supinfo-projects.com/fr/2003/shaders_opengl/3/

<http://www.codesampler.com>