TD4 - Introduction aux shaders

Les shaders sont des programmes appliqués par le processeur de la carte graphique pour traiter les points et les pixels afin de générer l'image 2D finale. Cela permet de modifier la manière dont sont affichés les points et les pixels et d'appliquer des traitements dessus pour réaliser des effets.

Il existe plusieurs types de shaders, nous nous intéressons aux 2 principaux : les vertex shaders et les fragment shaders.

Il existe également plusieurs langages de shaders : HLSL par Microsoft, Cg par Nvidia, etc, mais nous intéressons au langage de shader standard associé à OpenGL : GLSL, qui ressemble à du C. En GLSL, des types vecteurs (vec3, vec4, etc) et matrices (mat4, etc) sont prédéfinis ainsi que beaucoup d'opérations arithmétiques (multiplications, additions, etc). Le type par défaut est généralement le float.

Exemple de base

Le code est **ici**

Vertex shaders

Un vertex shader est un programme qui s'applique à tous les points à afficher, par exemple en utilisant la fonction glDrawElements. Si on a passé un tableau d'indices et un tableau de points dans le VAO à dessiner, le premier indice est récupéré, on va chercher dans le tableau de points les coordonnées x,y,z associées à ce point qui sont passées ensuite au vertex shader. Ces valeurs se retrouvent stockées dans une variable de type vec3 qui comme son nom l'indique contient 3 valeurs flottantes. On distingue les données en entrée et sortie du shader par les spécificateurs in et out. Pour le point en entrée on déclare donc :

```
in vec3 in_pos;
```

En sortie du vertex shader, on doit produire la position du point une fois les transformations model, view et projection appliquées. Il faut donc passer la matrice de transformation mvp au pixel shader. Dans les vertex shaders, on distingue les variables qui sont différentes pour chaque point et celles qui sont les mêmes pour tous les points. Chaque point possède sa propre position 3D, mais la même matrice de transformation est appliquée à tous les points. Cela ressemble à la distinction entre variables d'instance et de classe en Java et C++. Pour spécifier qu'une variable est partagée entre tous les points, on utilise le spécificateur uniform :

```
uniform mat4 mvp;
```

On peut ensuite écrire le programme pour calculer la position du point en sortie du vertex shader. Cette position est stockée dans une variable prédéfinie nommée gl_Position :

```
void main( void )
{
  gl_Position = mvp * vec4( in_pos, 1.0 );
}
```

Comme mvp est une matrice 4x4, il faut la multiplier par un vecteur de longueur 4, on utilise pour cela le constructeur de vec4 qui prend un vecteur de longueur 3, ici la position en entrée, et une autre valeur, ici 1.0.

Dans le vertex shader on va donc pouvoir modifier les coordonnées des points et donc effectuer des effets de déformation du maillage par exemple.

Fragment shaders

Une fois les points traités par le vertex shader, les points d'une même primitive, dans notre cas un triangle donc 3 points, sont rassemblés pour délimiter un fragment : l'ensemble des pixels qui se trouvent dans le triangle formé par 3 points. Suivant l'orientation du triangle, le nombre de pixels dans le fragment peut donc varier. Un fragment shader va donc pouvoir appliquer des effets sur les pixels d'un fragment, par exemple en changer la couleur. La sortie produite par le vertex shader est donc la couleur d'un pixel d'un fragement, généralement au format RGBA :

```
out vec4 out_color;
```

Dans le fragment shader ci-dessous on définit la couleur de chaque pixel du fragment à vert :

```
void main( void )
{
  out_color = vec4( 0.0, 1.0, 0.0, 0.0 );
}
```

Code OpenGL de base

Dans le code de base fournit, le chargement des shaders est effectué dans la fonction initShaders. Sans trop rentrer dans les détails, les fichiers sont chargés sous forme de chaînes de caractères, sont associés à des shaders, sont compilés pour détecter les erreurs, puis sont assemblés pour former un programme. Ce programme est identifié par la variable progid et est activé par la fonction :

```
glUseProgram( progid );
```

Il est possible d'utiliser plusieurs programmes et de les appliquer à différents maillages. Il suffit pour cela d'activer le bon programme avant l'affichage du maillage sur lequel l'appliquer.

Ensuite, on peut récupérer les emplacements des variables uniform des shaders du programme à l'aide de la fonction glGetUniformLocation. Par exemple pour récupérer l'emplacement de la variable mvp dans le vertex shader ci-dessus on utilise :

```
mvpid = glGetUniformLocation( progid, "mvp" );
```

Où mvpid est une variable entière globale qui va être utilisée dans la fonction display pour envoyer la nouvelle matrive mvp au programme à chaque nouvel affichage :

```
glUniformMatrix4fv( mvpid , 1, GL_FALSE, &mvp[0][0]);
```

Pour associer les données d'entrée du programme associées à chaque point, il convient également de récupérer les emplacements de ces variables, dans notre exemple la variable in_pos, au moment de la création du VAO :

```
auto pos = glGetAttribLocation( progid, "in_pos" );
//
glVertexAttribPointer( pos, 3, GL_FLOAT, GL_FALSE, 0, 0 );
glEnableVertexAttribArray( pos );
```

La variable in_pos est associée à l'emplacement pos, décrit comme 3 flottants consécutifs (x, y, z).

Compilation et test

Pour lancer le code (vérification des shaders + compilation) : make run.

Exercices

Redimensionnement du maillage

Modifier uniquement le vertex shader pour que le modèle apparaisse 2 fois plus gros à l'écran.

Modification du maillage

Ajouter le vecteur vec3(1.0, 0.0, 0.0) à tous les points dont les coordonnées en x sont positives. Pour récupérer la composante x de in_pos : in_pos.x.

Modification de la couleur

Modifier uniquement le fragment shader pour que le modèle apparaisse en jaune.

Couleur en fonction de la position du point

On peut transmettre des variables d'un vertex shader vers un fragment shader. Dans cet exercice, on utilise la position des points comme couleur donc il faut passer la position du vertex shader au fragment shader à l'aide de la variable color.

Dans le vertex shader, créer une variable de sortie color :

```
out vec3 color;
```

puis définir dans la fonction main :

```
color = vec3( 1.0, 0.0, 0.0 );
```

Cette variable de sortie est ensuite passée comme entrée au fragment shader :

```
in vec3 color;

out vec4 frag_color;

void main( void )
{
  frag_color = vec4( color, 1.0 );
}
```

Le maillage doit apparaître en rouge.

1. Modifier ensuite la couleur pour quelle dépende de la position des points du maillage.
2. Modifier ensuite la couleur pour quelle dépende uniquement de la coordonnée x de chaque point.

Couleur en fonction des coordonnées des pixels

Dans le fragment shader, la variable prédéfinie gl_FragCoord de type vec4 permet de récupérer les coordonnées de chaque pixel dans l'espace de la fenêtre.

On souhaite modifier le fragment shader de manière à attribuer une couleur de chaque pixel du fragment en fonction de sa position 2D dans la fenêtre. On peut ainsi créer une sorte de texture « échiquier » à son objet avec une taille de bloc de 10 par 10.

Pour cela, on utilise les fonctions ceil (pour arrondir une valeur flottante à l'entier supérieur) et la fonction mod (modulo). L'idée de base est la suivante : si la somme des coordonnées x et y d'un pixel est paire alors on assigne une couleur au pixel, sinon on en assigne une autre. De cette manière si un pixel a une couleur, les pixels à sa gauche, droite, au dessus et au dessous auront l'autre couleur. Pour créer des blocs de 10 par 10 pixels avec la même couleur il suffit de diviser leurs coordonnées par 10. Ce qui nous donne le code suivant :

```
frag_color = vec4( mod(ceil(gl_FragCoord.x/10)+ceil(gl_FragCoord.y/10), 2), 0.0, 0.0
```

On peut obtenir des résultats différents avec une fonction sinus :

```
frag_color = vec4( sin(gl_FragCoord.x/5), 0.0, 0.0, 1.0 );
```

Redimensionner la fenêtre pour observer le comportement du shader. Pourquoi le motif ne passe pas à l'échelle ?

Essayer d'autres paramètres et d'autres fonctions mathématiques.