OpenGL, notions avancées

CPE – David Odin

Durée – 4h
10 décembre 2010

1 Quelques liens utiles

- Le site officiel concernant OpengGL : http://opengl.org/.
- La documentation en ligne des fonctions OpenGL : http://www.opengl.org/sdk/docs/man/.
- La spécification OpenGL version 2.1 : http://www.opengl.org/registry/doc/glspec21.20061201.pdf.
- La spécificatoin de GLSL 1.20 : http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf.
- Un résumé des fonctions OpenGL et GLSL (malheureusement pour OpenGL 3.2 et GLSL 1.5) : http://www.khronos.org/files/opengl-quick-reference-card.pdf.

2 Utitilisation des didacticiels de Nate Robins

Il y a quelques années Nate Robins a écrit quelques didacticiels permettant de découvirir et de mieux comprendre certains aspects d'OpenGL.

Ils sont un peu vieillots mais ont le mérite de bien présenter chaque notion. Dans chacun de ces programmes, cliquer sur un nombre en vert et bouger la souris permet de modifier une valeur, et cliquer avec le bouton droit permet d'afficher un menu contextuel (qui est différent suivant l'endroit où l'on clique). N'hésitez pas à tester chacun des menus et à modifier chacune des valeurs pour bien comprendre le rôle de tout cela.

2.1 Programme Shapes.

Le premier des programmes que je vous propose de tester est `shapes`. Il montre l'ensemble des primitives de base qu'OpenGL sait dessiner.

Question 1 : *Profitez-en pour bien comprendre l'action de chaque couleur et de chaque position (vertex).*

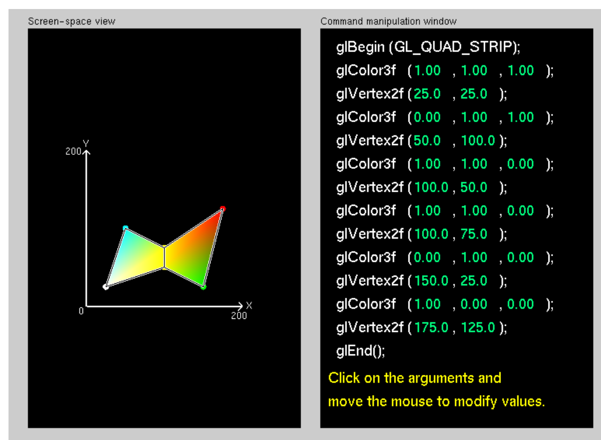

FIGURE 1 – Programme Shapes.

2.2 Programme Transformation.

Le programme `transformation` présente les différentes transformations de bases utilisables dans OpenGL (affinités, translations, rotations).

Question 2 : *Remarquez que l'ordre des transformations est important. Pourquoi ?*

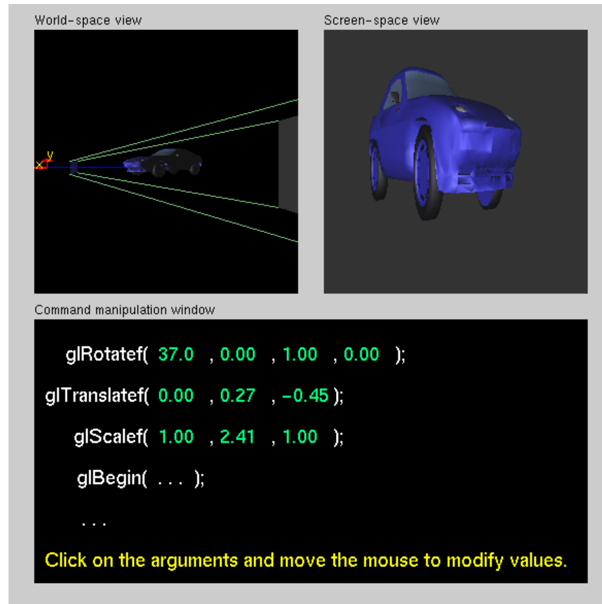

FIGURE 2 – Programme Transformation.

2.3 Programme Projection.

Le programme `projection` permet de comprendre comment les tranformations 3D/2D sont effectués par OpenGL. Il y a deux grands modes pour cela : orthographique et perspective.

Question 3 : *Assurez-vous de bien comprendre chaque paramètre et pourquoi le modèle dessiné disparait quelques fois.*

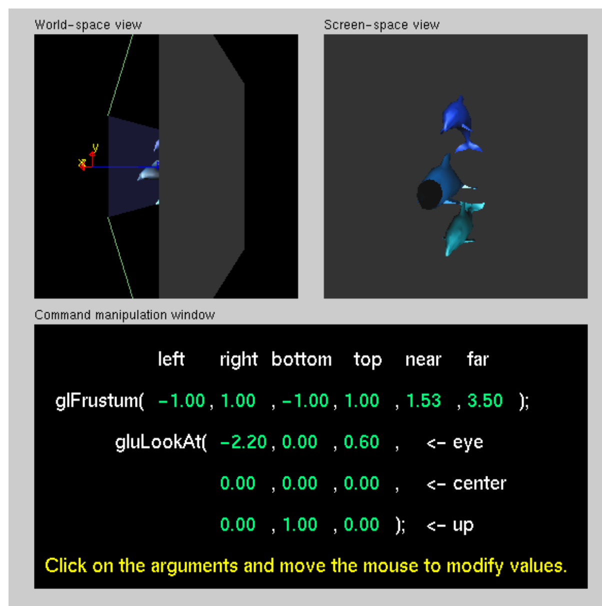

FIGURE 3 – Programme Projection.

2.4 Programme Texture.

Vous avez vu le programme `texture` pendant le cours. C'est le moment de vérifier si vous avez correctement compris les idées de coordonnées de texture.

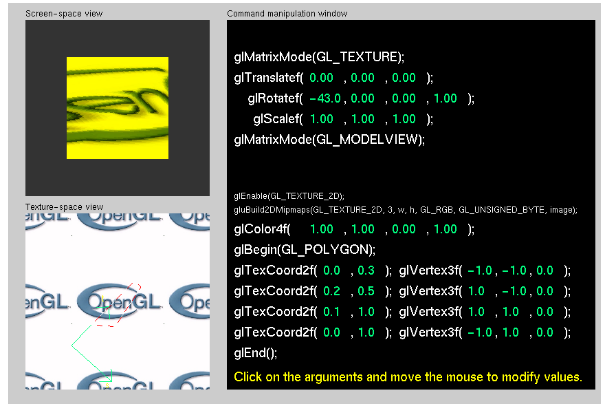

Figure 4 – Programme Texture.

2.5 Programme lightposition.

Il existe d'autre programme faits par Nate Robins, comme par exemple le programme `lightposition`. Celui-ci permet de jouer avec la position d'une source de lumière pour voir comment cela influence le rendu d'un modèle.
Question 4 : *Que se passe-t-il si la quatrième valeur de la position de la lumière est un 0.0 ? Et si c'est un 1.0 ?*

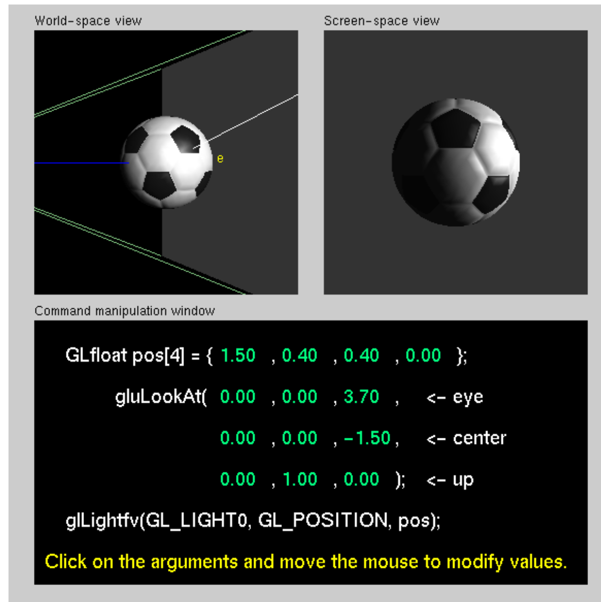

Figure 5 – Programme light position.

3 Les modes d'envoi de la géométrie

Comme vu en cours, il existe plusieurs moyen d'envoyer de la géométrie (des positions de vertex, des couleurs, des normales, des coordonnées de textures, etc.) à OpenGL.

3.1 Mode immédiat

Nommé ainsi car les données sont utilisées immédiatement par OpenGL. Il est assez simple à mettre en œuvre. Pour cela consultez la documentation des fonctions suivantes sur le site d'OpenGL :

- `glBegin (...);`
- `glEnd ();`
- `glVertex*(...);`
- `glColor*(...);`

Question 5 : *Dans le programme donné (simple.tar.gz) ajoutez ce qu'il faut pour afficher un triangle coloré à l'aide du mode immédiat.*

Note : il suffit d'ajouter quelques lignes dans la fonction `display()`.

3.2 Tableaux

Mais lorsque le nombre de données devient important, il est plus sympa de les stocker dans des tableaux. OpenGL permet d'utiliser directement cela grace aux fonctions suivantes :

- `glEnableClientState (...);`
- `glVertexPointer (...);`
- `glColorPointer (...);`
- `glDrawArrays (...);`

Question 6 : *Après avoir regardé les documentations de ces fonctions, modifiez le programme afin d'ajouter un quadrilatère multicolor à l'aide de ces tableaux*

Note : Il faudra placer certaines fonctions dans `init ();` et d'autres dans `display ();`

3.3 Tableaux indexés

Pour passer de l'utilisation de tableaux classiques à l'utilisation de tableaux indexés, il suffit d'ajouter un tableau d'index. Cela est détaillé dans l'aide de la fonction suivante :

- `glDrawElements (...);`

Question 7 : *Modifiez le programme afin d'utiliser un index pour l'affichage du quadrilatère.*

3.4 Tableaux en VRAM (VBO)

Pour plus de rapidité, il est possible de transmettre le contenu de ces tableaux en mémoire vidéo une fois pour toute. Pour cela, il faut créer un tampon (*buffer*) en mémoire video puis y transferer le tableau à l'aide des fonctions suivantes :

- `glGenBuffers (...);`
- `glBindBuffer (GL_ARRAY_BUFFER, ...);`
- `glBufferData (GL_ARRAY_BUFFER, ...);`
- `glVertexPointer (...);`
- `glColorPointer (...);`

Question 8 : *Modifier le programme afin d'utiliser un VBO pour les positions et couleurs du quadrilatère*

3.5 VBO indexés

Les tableaux en mémoire vidéo (VBO) peuvent aussi être indexés, soit par un index en mémoire centrale (RAM) dans quel cas cela s'utilise exactement comme pour les tableaux en RAM, soit par un index en mémoire vidéo. Pour placer les index en mémoire vidéo, on réutilise les fonctions suivantes, mais avec un paramètre différent :

- `glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, ...);`
- `glBufferData (GL_ELEMENT_ARRAY_BUFFER, ...);`
- `glDrawElements (...);`

Question 9 : *Modifier le programme afin d'utiliser un VBO également pour les index.*

4 Jouons avec les shaders

Pour cette partie, vous devrez utiliser l'archive `shader.tar.gz`. Il suffit de recompiler le tout. Vous n'aurez à éditer le fichier `main.cc` que pour modifier la fonction update_uniforms() et encore, uniquement quand il s'agira d'utiliser des variables uniformes.

Le programme `shader` affiche une simple théière (!) que l'on peut manipuler à la souris (les plus curieux peuvent aller voir comment cela est réalisé) en remplaçant les fonctions de calcul de *vertex* et de *fragment* par deux programmes : `shader.vert` et `shader.frag` respectivement.

Ce sont ces deux fichiers qu'il vous faudra modifier avant de relancer le programme `shader` (pas besoin de recompiler quoi que ce soit).

Par défaut `shader.vert` contient ceci :

```
#version 120

void main (void)
{
  gl_Position = ftransform ();
}
```

C'est le minimum que peut contenir un *vertex shader*. Il se contente de calculer la position (gl_Position) de chaque *vertex* en utilisant les transformations et projections normales (c'est ce que renvoie la fonction ftransform():). La ligne `# version 120` signifie que l'on désire utiliser les fonctions définies dans la spécification de GLSL

Et par défaut, le *shader* de *fragment* nommé `shader.frag` contient ceci :

```
#version 120

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

La variable gl_FragColor doit obligatoirement recevoir une valeur pendant un *fragment shader* et le type de cette variable est `vec4`, soit un vecteur à 4 dimensions. Ici, on colore tous les pixels en rouge.

Figure 6 – Une théière toute rouge

4.1 Un peu de fantaisie

On peut heureusement faire mieux que des couleurs unies gràce à l'utilisation de fonctions et de certaines variables. Par exemple, la variable gl_FragColor contient les coordonnées du pixel dans la fenêtre. Ainsi en utilisant la ligne suivante :

```
gl_FragColor = vec4 (mod (gl_FragCoord.xy / 30.0, vec2(1.0,1.0)), 0.0, 1.0);
```

On obtient une théière habillé en Tartan écossais du plus bel effet (voir figure 7).

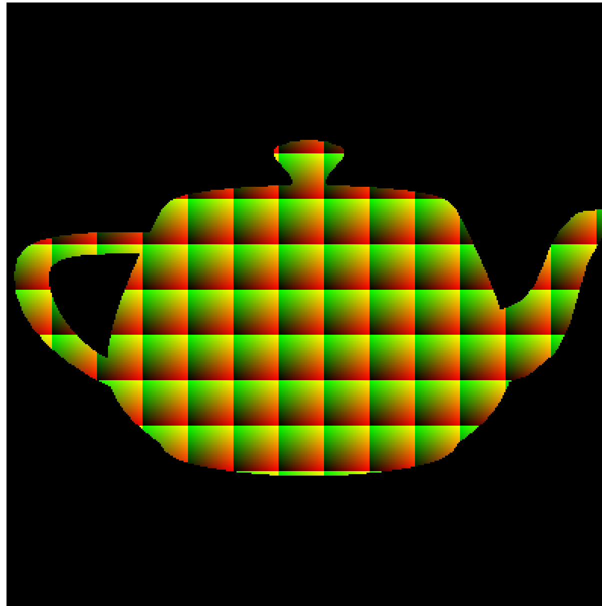

FIGURE 7 – Une théière en Tartan

Question 10 : *Modifiez le fragment shader afin d'afficher un damier ou une fractale de mandelbrot (ou n'importe quoi d'autre qui vous amuse...) sur la théière.*

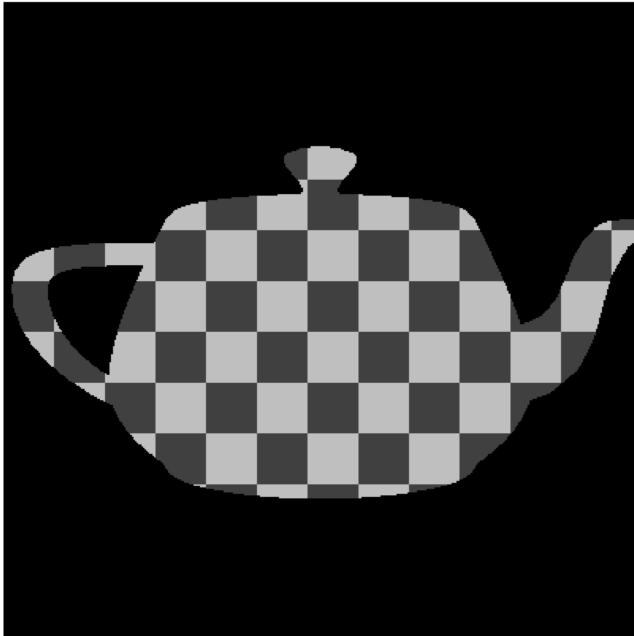

FIGURE 8 – Un damier sur une théière

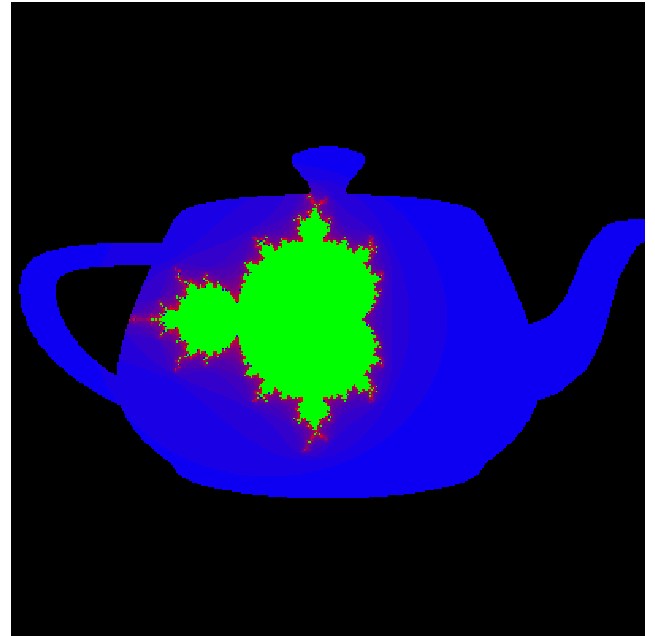

FIGURE 9 – Une fractale sur une théière (ça devient n'importe quoi...)

4.2 Passage de valeur entre les shaders

Heureusement, il est possible de faire plus intéressant. Notamment en passant des variables entre le *vertex shader* et le *fragment shader* Cela se fait en déclarant la même variable dans les deux *shader* en la préfixant par `varying`. Par exemple, on peut avoir ceci comme *vertex shader* :

```
#version 120

varying vec3 position;
varying vec3 normal;

void main (void)
{
  position = gl_Vertex.xyz;
  normal = gl_Normal;
  gl_Position = ftransform ();
}
```

et ceci dans le *fragment shader* :

```
#version 120

varying vec3 position;
varying vec3 normal;

void main(void)
{
  gl_FragColor = ...   ;
}
```

Cela permet de récuperer dans le *fragment shader* la position sur la théière et la normale. Ceci permet d'avoir des effets qui suivent la surface de la théière. Par exemple, en utilisant ceci :

```
  gl_FragColor = vec4 (abs(position), 1.0);
```

On obtient une couleur qui ne dépend que de la position sur la théière (voir figure 10).

En utilisant la normale, il est possible d'obtenir une impression d'éclairage et de relief comme on peut le voir sur la figure 11.

Question 11 : *Comment ?*

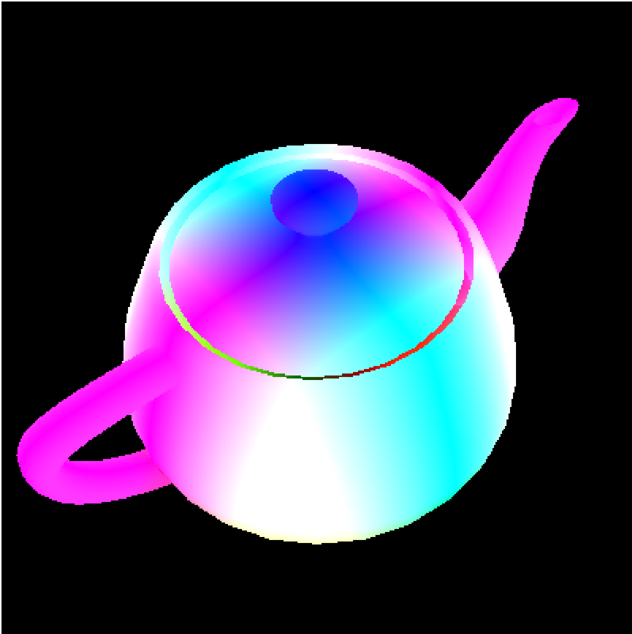

Figure 10 – Une théière (légèrement) plus réaliste

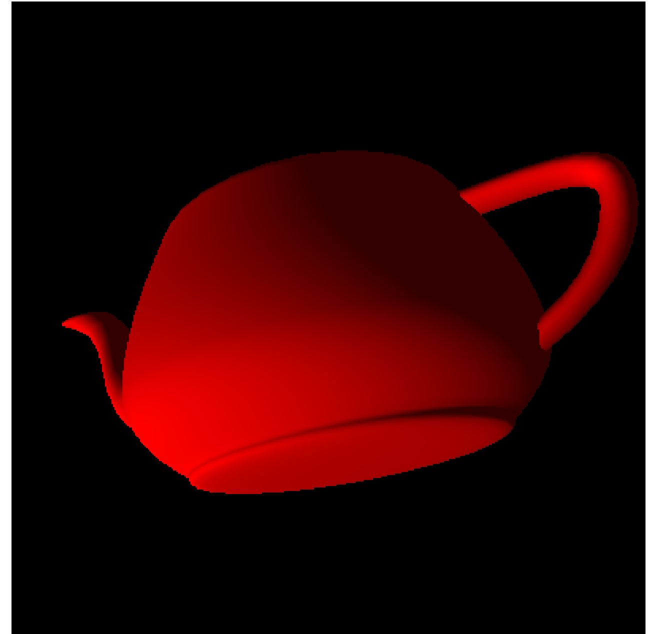

Figure 11 – Une théière éclairée

4.3 Variables globales (uniformes)

Il est aussi possible d'avoir des variables globales, qui ont une valeur qui ne change pas (qui reste uniforme donc...). Ces variables sont préfixées par le mot-clef `uniform` et peuvent être modifiée via le programme principal en utilisant les fonctions `glUniform*`.

Question 12 : *Modifiez la fonction update_uniforms() et vos shaders afin d'utiliser une telle variable.*

4.4 Utilisation des textures

Une utilisation très fréquente des variables uniformes concerne les textures.

Pour cela, il suffit d'ajouter les lignes suivantes dans le *fragment shader* :

```
uniform sampler2D  myTexture;

...

  vec3 color = vec3 (texture2D (myTexture, gl_TexCoord[0].st));
```

Et la ligne suivante dans le *vertex shader* :

```
  gl_TexCoord[0]  = gl_MultiTexCoord0;
```

Question 13 : *Essayez d'obtenir un affichage ressemblant à la figure 12.*

FIGURE 12 – Une théière en brique pour toujours plus de réalisme