OpenGL
Mise en œuvre, vue d'ensemble

David Odin

Ce TP est une (re)prise en main d'OpenGL et des shaders aux travers d'exemples assez directs.

Les compétences et connaissances mises en œuvre dans ce TP sont :
— Utilisation directe d'OpenGL
— Utilisation de tous les modes d'envoi des données géométriques à la carte graphique
— Création de shaders simples, avec passage de variables
— Mise en œuvre d'un modèle d'illumination simple (Phong).

1 Les modes d'envoi de la géométrie

1.1 Mode immédiat

En ajoutant directement les lignes suivantes dans la fonction `display()` :

```
1    glBegin(GL_TRIANGLES);
2     glColor3f(1, 0, 0); glVertex3f(-0.75, 0.5, 0);
3     glColor3f(0, 1, 0); glVertex3f(0.25, 0, 0);
4     glColor3f(0, 0, 1); glVertex3f(-0.75, -0.5, 0);
5   glEnd();
```

On obtient bien la le résultat donné sur la figure 1

FIGURE 1 – Un triangle coloré en mode immédiat.

1.2 Tableaux

Pour ajouter un quadrilatère à notre scène en utilisant les tableaux de sommets, on doit bien évidemment définir ces tableaux :

```
1 static GLfloat positions[] = { 0.45,0.45, -0.45,0.45, -0.45,-0.45, 0.45,-0.45 };
2 static GLfloat colors[] = { 1,1,0, 1,0,1, 0,1,1, 1,1,1 };
```

Note : Le mot-clef `static` permet à la variable de continuer à exister même après la fin de la fonction `init()`, ce qui est assez pratique dans notre cas.

Note : J'ai choisi de ne donner que deux coordonnées (x, y) pour les positions, dans ce cas, OpenGL ajoute automatiquement une coordonnée z à 0. Ça permet d'éviter les répétitions. Il faudra en tenir compte lors de l'utilisation de la fonction `glVertexPointer()`. On peut aussi faire ça en mode immédiat en utilisant `glVertex2f()`.

L'initialisation des tableaux se poursuit ainsi :

```
1 // Initialisation des tableaux
2 glEnableClientState(GL_VERTEX_ARRAY);
3 glEnableClientState(GL_COLOR_ARRAY);
4 glVertexPointer(2, GL_FLOAT, 0, positions);
5 glColorPointer(3, GL_FLOAT, 0, colors);
```

Note : Le première paramètre de `glVertexPointer()` est bien 2 pour signifier que l'on ne passe que (x, y).

L'affichage du quadrilatère se fait en ajoutant la ligne suivante dans la fonction `display()` :

```
1 glDrawArrays (GL_QUADS, 0, 4);
```

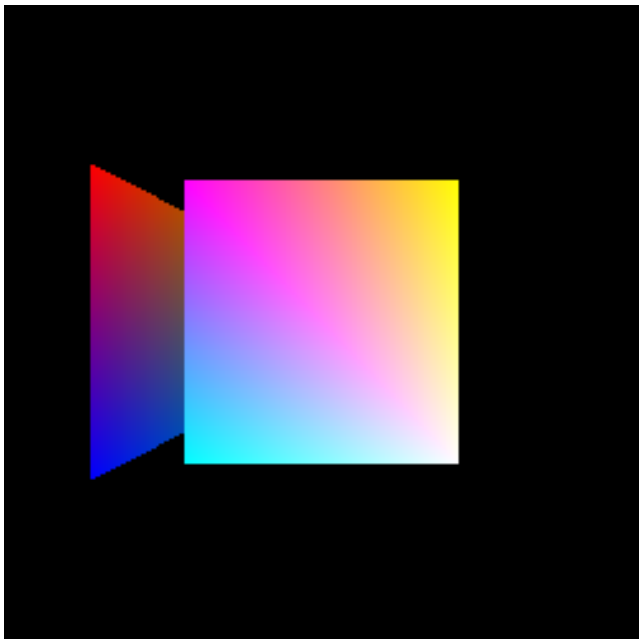

FIGURE 2 – Un quadrilatère multicolor.

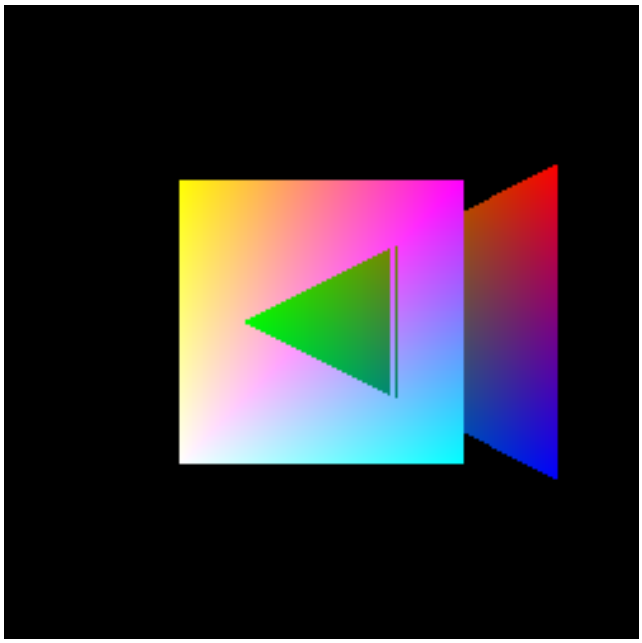

FIGURE 3 – Un quadrilatère multicolor.

Cela permet d'obtenir la figure 2. Si on change l'angle de rotation, il est possible d'avoir des bugs d'affichage comme sur la figure 3. Les deux primitives étant co-planaires, la moindre imprécision fait passer une partie de l'un devant ou derrière l'autre. Il s'agit d'un problème classique nommé *Z-fighting*.

1.3 Tableaux indexés

Pour passer de l'utilisation de tableaux classiques à l'utilisation de tableaux indexés, il suffit d'ajouter un tableau d'index.

Il suffit de remplacer la ligne `glDrawArrays(...);` par les deux lignes suivantes :

```
1  GLuint indices[] = { 0, 1, 2, 3 };
2  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_INT, indices);
```

Pour tracer un seul quadrilatère, l'intérêt des index ne saute pas aux yeux, car aucun sommet n'est réutilisé. Le résultat est le même que pour les tableaux non indexés ! (Pas de nouvelle figure)

1.4 Tableaux en VRAM (VBO)

La création des VBO rend le code d'initialisation un peu plus complexe, mais rend l'affichage plus rapide.

L'initialisation des données pour l'utilisation des VBO est la suivante :

```
1  glGenBuffers(1, &vbop);
2  glBindBuffer(GL_ARRAY_BUFFER, vbop);
3  glBufferData(GL_ARRAY_BUFFER, sizeof positions, positions, GL_STATIC_DRAW);
4  glEnableClientState(GL_VERTEX_ARRAY);
5  glVertexPointer(2, GL_FLOAT, 0, nullptr);
6
7  glGenBuffers(1, &vboc);
8  glBindBuffer(GL_ARRAY_BUFFER, vboc);
9  glBufferData(GL_ARRAY_BUFFER, sizeof colors, colors, GL_STATIC_DRAW);
10 glEnableClientState(GL_COLOR_ARRAY);
11 glColorPointer(3, GL_FLOAT, 0, nullptr);
```

Note : Les données elles-même n'ont plus besoin d'être déclarées `static` car elles sont copiées en mémoire vidéo.

La fonction `display()` n'est pas du tout modifiée, et l'affichage non plus.

1.5 VBO indexés

Les indices peuvent également être placés en mémoire video. Pour cela il suffit d'ajouter les lignes suivantes dans la fonction d'initialisation :

```
1  GLuint indices[] = { 0, 1, 2, 3 };
2  glGenBuffers(1, &vboi);
3  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboi);
4  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof indices, indices, GL_STATIC_DRAW);
```

L'appel dans la fonction `display()` devient alors simplement :

```
1  glDrawElements(GL_QUADS, 4, GL_UNSIGNED_INT, nullptr);
```

2 Jouons avec les shaders

2.1 Utilisation de gl_FragCoord

La variable prédéfinie `gl_FragCoord` contient les coordonnées du pixel dans la fenêtre. `gl_FragCoord.x` et `gl_FragCoord.y` prennent donc des valeurs entre 0 et 400.

En les divisant par 400, on pourrait ramener leur valeur entre 0 et 1.

Ainsi en utilisant la ligne suivante :

```
1  color = vec4(gl_FragCoord.x/400.0);
```

On obtient une gris dégradé de gauche à droite comme sur la figure 4.

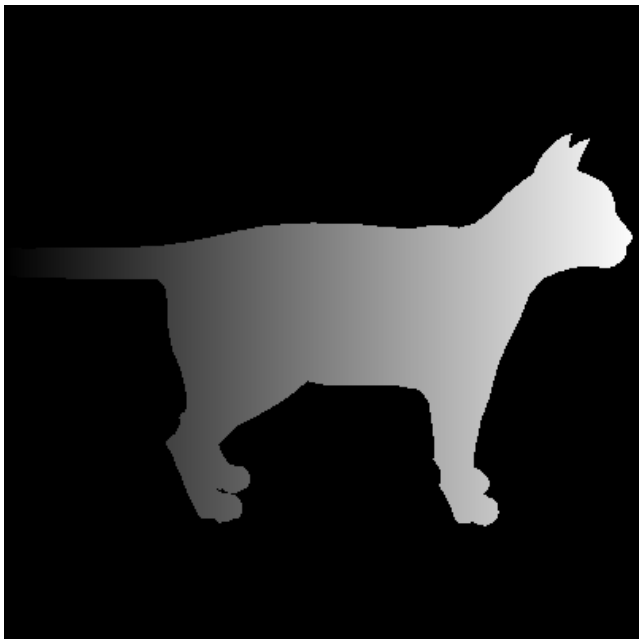

FIGURE 4 – Un chat qui se dégrade

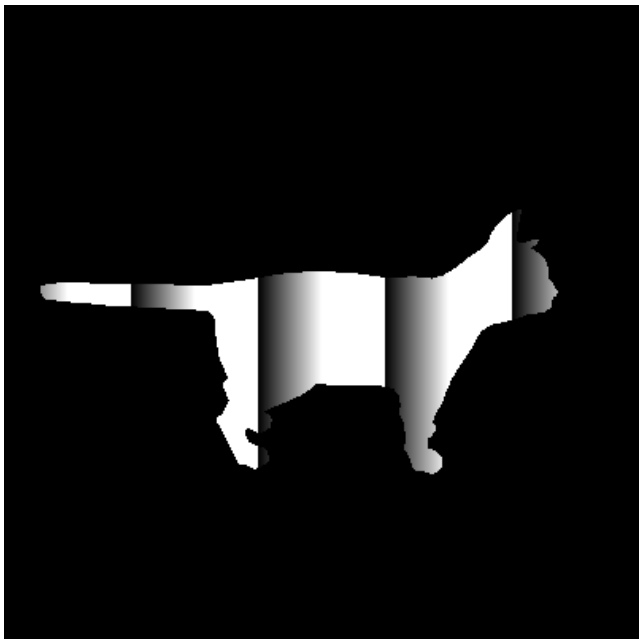

FIGURE 5 – plusieurs dégradés

Si on veut que ce dégradé se répète plusieurs fois, on peut utiliser la fonction `mod()`, qui garde le reste d'une division, ainsi :

```
1  color = vec4(mod(gl_FragCoord.x/40.0,2.0));
```

Notre dégradé va de 0 à 2, mais on remarque que les parties entre 1 et 2 restent blanche. Pour se rapprocher d'un damier, on peut utiliser la fonction `floor()` qui ne garde que la partie entière d'une valeur (0 pour tout ce qui est entre 0 et 1) :

```
1  color = vec4(floor(mod(gl_FragCoord.x/40.0,2.0)));
```

On obtient alors des bandes verticales (figure 6).

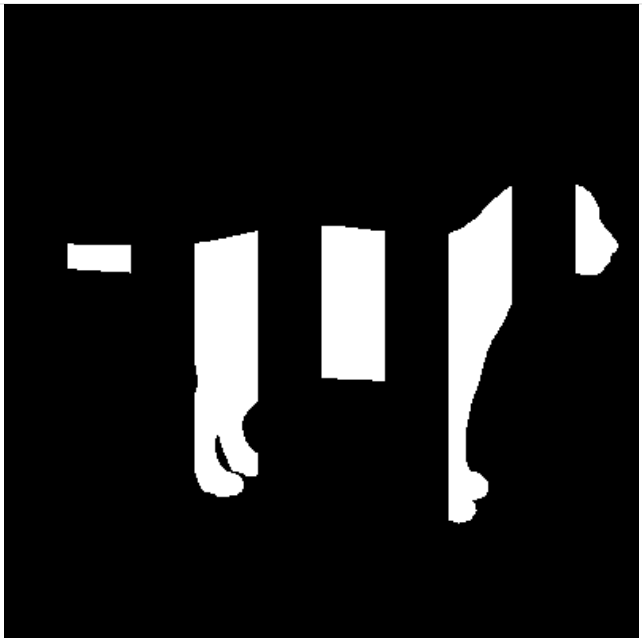

FIGURE 6 – Un chat découpé en (grosses) tranches

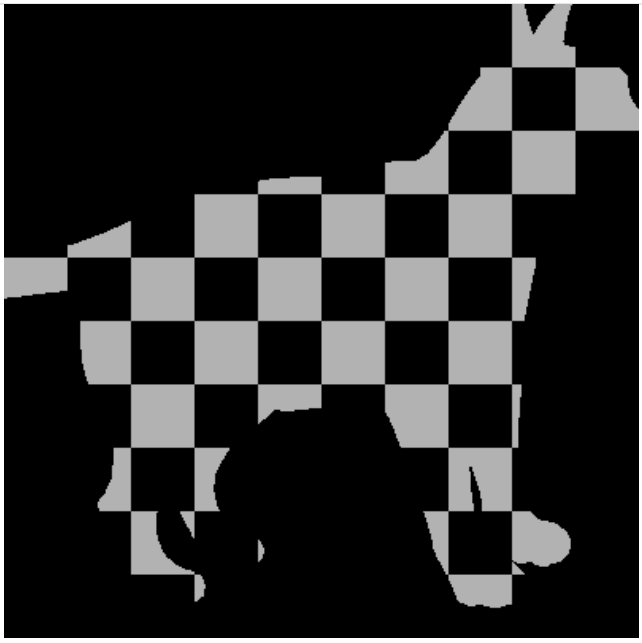

FIGURE 7 – Un chat-damier

En utilisant la même technique, dans les deux directions, on peut obtenir un damier comme sur la figure 7. Pour simplifier le code, on utilise le fait que toutes les fonctions GLSL fonctionnent aussi bien avec des *float* qu'avec des *vec2* :

```
1  vec2 v = mod(floor(gl_FragCoord.xy / 40.0), vec2(2.0));
2  color = .7*vec4(mod(v.x+v.y,2.0));
```

En utilisant `gl_FragCoord` dans des calculs plus complexes, on peut obtenir toutes sortes d'habillage pour notre chat. Le langage GLSL est très complet, on peut utiliser des tests (`if`), des boucles (`while`, `for`), ce qui permet de dessiner une fractale avec le code suivant (voir figure 8) :

```
1  vec2 col01 = vec2(0., 1.);
2  vec2 z0 = gl_FragCoord.xy / 80. - vec2(2.5), z = z0;
3  float iter;
4  for (iter = 0.; iter < 20. && length(z) < 2.; iter++)
5    z = vec2(z.x * z.x - z.y * z.y + z0.x, // Partie réelle
6             2. * z.x * z.y + z0.y);       // Partie imaginaire
7  if (length (z) < 2.)
8    color = col01.xyxy;
9  else
10   color = mix(col01.xxyy, col01.yxxy, iter / 20.);
```

Notez l'écriture `col01.yxxy`, qui permet de simplifier des notations, toutes les combinaisons sont possibles !

À vrai dire, le chat ici n'est pas très utile. Il ne sert que de pochoir, et un quadrilatère couvrant toute la fenêtre ferait bien mieux l'affaire pour tester ce genre de calculs qui ne mettent en jeu que des coordonnées de pixels. Un site web permet d'explorer cela : https://www.shadertoy.com/. Certaines réalisations sont vraiment délirantes, mais elles montrent la puissance des cartes graphiques et du langage GLSL.

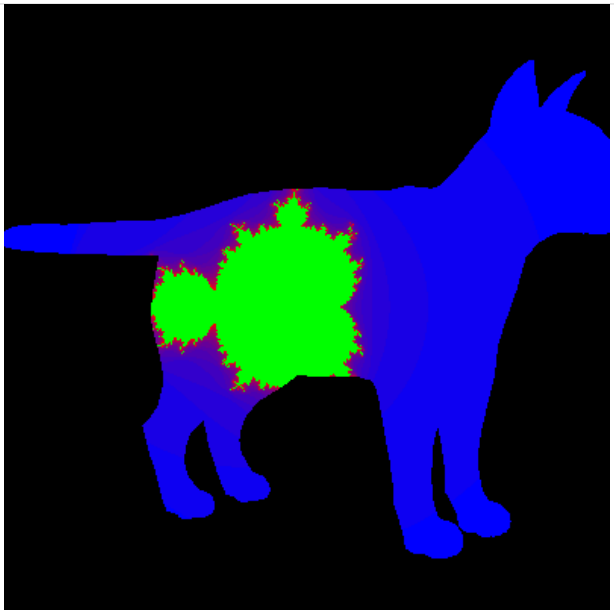

FIGURE 8 – Un chat fractale

2.2 Passage de valeur entre les shaders

L'utilisation de `gl_FragCoord` n'est cependant pas la norme dans les *fragment shaders*.

Les données d'entrées proviennent en général du *vertex shader*. En utilisant la position et la normale, il est possible de simuler un éclaraige de Phong avec le code suivant (voir figure 9) :

```
1  vec3 s = -normalize(v_position - vec3(50., 50., 150.));
2  vec3 n = normalize(v_normal);
3  vec3 r = normalize(reflect(s, n));
4  vec3 view = normalize(-v_position);
5  vec4 diffuse  = vec4(1., .3, 0., 1.) * max(0., dot(n, s));
6  vec4 specular = vec4(vec3(pow(max(.0, dot(r, view)), 80.)), 1.);
7  color = diffuse + specular;
```

FIGURE 9 – Un chat en porcelaine

Nous verrons d'autres utilisations des shaders, notamment l'utilisation de variables *uniform* ou de textures dans les prochains TP.