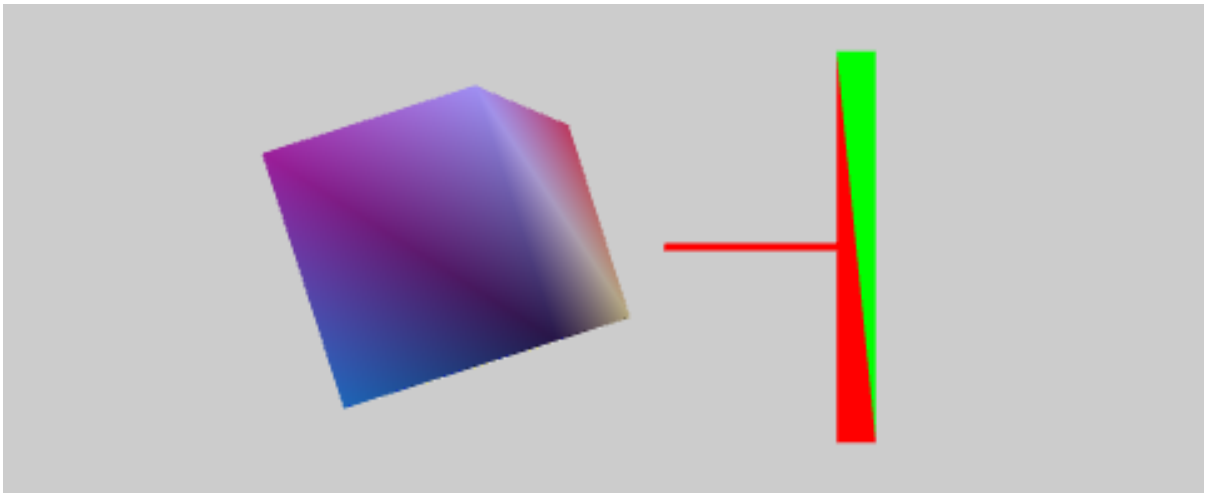

Informatique Graphique

- Compte-rendu de TP n°1 -

Loris Thibaud - Maxime Garibaldi

Dans ce TP, nous avons eu l'occasion de nous familiariser avec OpenGL en affichant une fenêtre, puis en intégrant des renderables (repères spatiaux, triangle, cube) dans l'espace. Ensuite, nous avons étudié la notion d'indexation dans le cadre des modèles de cube, puis nous avons joué sur les transformations matricielles des objets que nous avons créés.

Faire apparaître la fenêtre -

On récupère notre practical 1 qui ressemble à ceci :

```
int main( int argc, char* argv[] ) {
  // Stage 1: Create the window and its OpenGL context
  glm::vec4 background_color = glm::vec4(0.8,0.8,0.8,1.0);
  Viewer viewer(SCR_WIDTH, SCR_HEIGHT, background_color);

  // Stage 2: Load resources like shaders, meshes... and make them part of the virtual scene
  // ...

  // Stage 3: Our program loop
  while( viewer.isRunning() )
  {
    viewer.handleEvent();  // user interactivity (keyboard/mouse)
    viewer.draw();     // rasterization (write in framebuffer)
    viewer.display();      // refresh window
  }

  return EXIT_SUCCESS;
}
```

On set-up la couleur du background avec un vecteur4 (3 pour R-rouge, G-Vert, B-bleu, et un dernier pour la transparence), puis on instancie notre fenêtre viewer. La boucle principale du programme permet d'afficher les objets dans la fenêtre lorsque viewer tourne.

Pour l'instant, on ne charge aucune ressource, on se retrouve donc seul face au vide.

Figure 1 : fenêtre vide

Avoir notre premier rendu -

On va donc chercher à remplir ce vide avec des objets. Pour ça, on a la classe renderable qui représente simplement un objet pouvant être ajouté au Viewer afin d'être affiché dans la fenêtre.

Ajouter des couleurs à notre premier triangle rendu -

On souhaite avoir des couleurs à chaque point de notre triangle. Pour ça, on aura besoin d'un vecteur 4 vColor qu'on définit dans notre glsl flatVertex : chaque vertex prend maintenant un vecteur de couleur.

```
uniform mat4 projMat, viewMat, modelMat;

in vec3 vPosition;
in vec4 vColor;
out vec4 surfel_color;
```

```
void main(){
        gl_Position = projMat*viewMat*modelMat*vec4(vPosition, 1.0f);
        surfel_color = vColor;
}
```

Puis dans CubeRenderable, en s'inspirant des vecteurs de position, on push des vecteurs de couleur.

```
m_colors.push_back(glm::vec4(1, 0, 0, 1));
m_colors.push_back(glm::vec4(1, 0, 0, 1));
m_colors.push_back(glm::vec4(1, 0, 0, 1));
```

et on les place dans un buffer pour le GPU. Ce buffer aura été créé préalablement et comme pour les positions ont lie la variable vColor pour y mettre nos valeurs.

```
glGenBuffers(1, &m_cBuffer);
  glBindBuffer(GL_ARRAY_BUFFER, m_cBuffer);
  glBufferData(GL_ARRAY_BUFFER, m_colors.size() * sizeof(glm::vec4), m_colors.data(),
GL_STATIC_DRAW);
```

```
int colorrrrr = m_shaderProgram->getAttributeLocation("vColor");
glEnableVertexAttribArray( colorrrrr );
glVertexAttribPointer( colorrrrr, 4, GL_FLOAT, GL_FALSE, 0, (void*)0);
glBindBuffer( GL_ARRAY_BUFFER , m_cBuffer );
glDisableVertexAttribArray( colorrrrr);
```

A la compilation, cela nous donne le même triangle que précédemment, mais avec de magnifiques couleurs (figure 2).

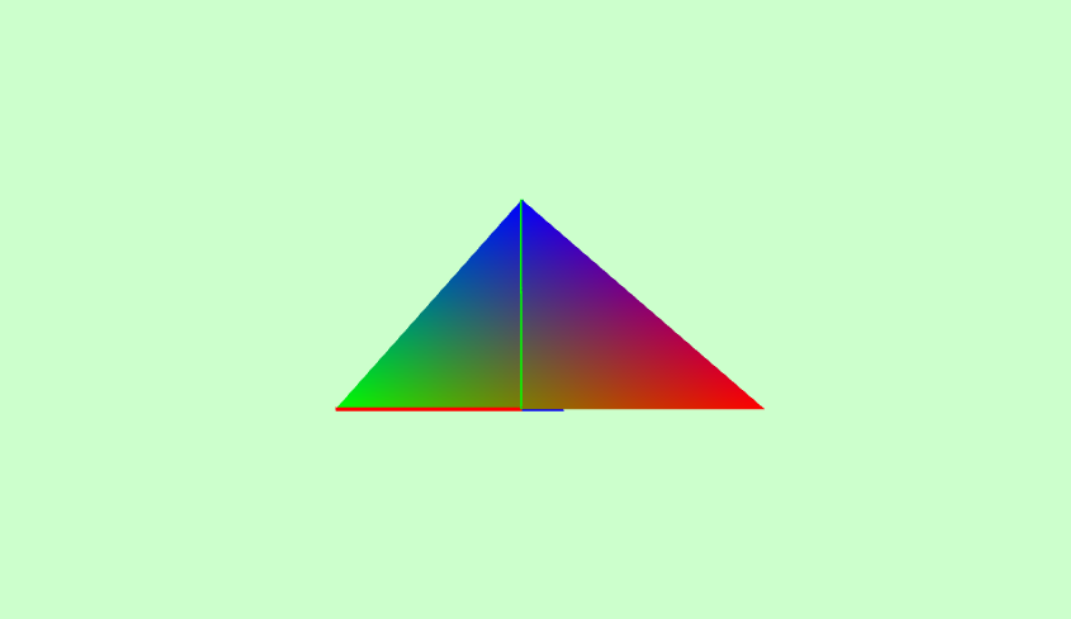

Figure 2 : la triforce RGB, must have des chambres de gamers

Création d'un cube sans utiliser l'indexage -

Un cube n'est qu'une association de triangles, tel qu'une face du cube est composée de deux triangles. Ainsi, l'implémentation d'un renderable cube est très proche de celle d'un triangle ; on va simplement placer 12 triangles pour représenter nos 6 faces, soit 36 sommets. La fonction getUnitCube() dans Utils.cpp le fait déjà pour nous, donc on n'hésite pas à l'appeler, puis on rajoute 36 vec4 de couleur pour rajouter de la couleur à chacun de nos sommets.

```
getUnitCube(m_positions, normals, tcoords);
```

Ce qui nous donne l'affichage suivant (figure 3).

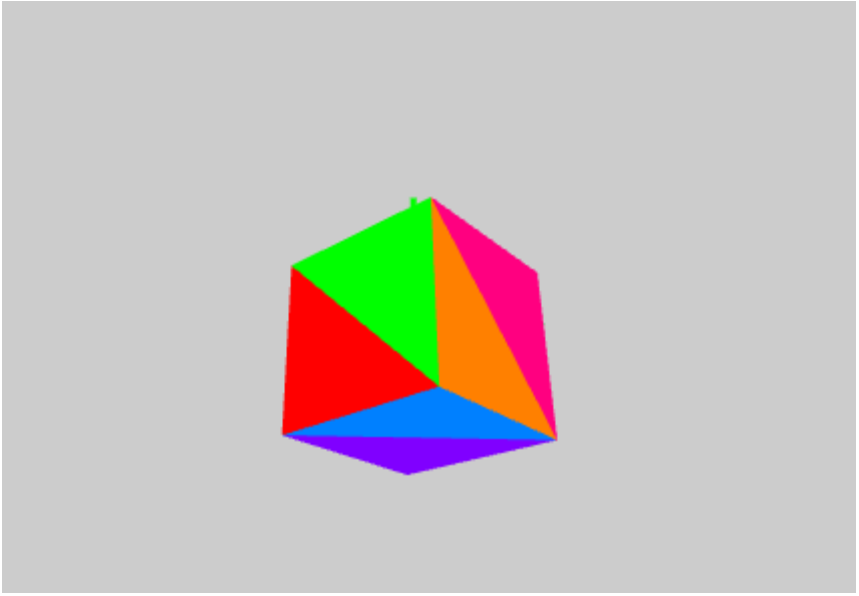

Figure 3 : Le cube pas indexé

Création d’un cube avec indexage -

Cette fois-ci nous allons créer un Cube mais en utilisant une liste d’index. Comme il est facile de le comprendre dans la solution précédente, nous avons créé 36 sommets alors qu’en réalité nous en avons besoin de seulement 8. Nous ajoutons donc la liste d’index qui permet d'indiquer une liste de sommets par triangle, ce qui permet donc de réduire le nombre de sommets. Comme pour le cube précédent une fonction getUnitIndexedCube() est prédéfinie dans Utils.cpp. Il nous reste donc plus qu'à ajouter un vecteur dans IndexedCubeRendable.hpp ainsi que son buffer pour pouvoir le remplir par la suite dans le IndexedCubeRendable.cpp à l’aide du tutoriel mis à notre disposition.

```
glGenBuffers(1, &m_iBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_iBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, m_index.size()*sizeof(unsigned
int), m_index.data(), GL_STATIC_DRAW);
glDrawElements(GL_TRIANGLES, m_index.size(), GL_UNSIGNED_INT,
(void*)0);
```

Ce qui nous donne le résultat suivant (figure 4).

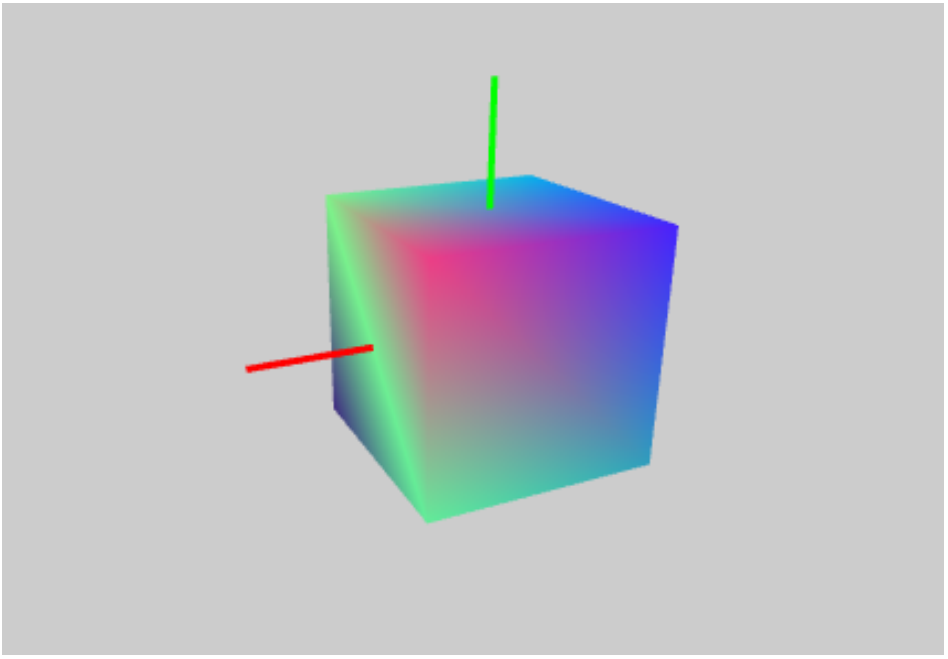

Figure 4 : Le cube indexé

Introduction à la transformation de matrice -

Maintenant que nous avons instancié nos deux cubes, ils se retrouvent au même endroit, l'un plus grand que l'autre. Ils entrent en collision. On va donc chercher à faire effectuer une translation à notre cube indexé pour le mettre à côté du cube classique.

Pour ça, on utilise d'abord la fonction translate() d'un glm qui permet d'enregistrer les informations d'une translation dans l'espace.

Puis, on utilise une fonction setModelMatrix() dont IndexedCubeRenderable hérite et qui permet d'appliquer n'importe quelle transformation à une matrice.

```
glm::mat4 t = glm::translate(glm::mat4(), glm::vec3(2.0f, 0.0f, 0.0f));
indexedCube -> setModelMatrix(t*r);
```

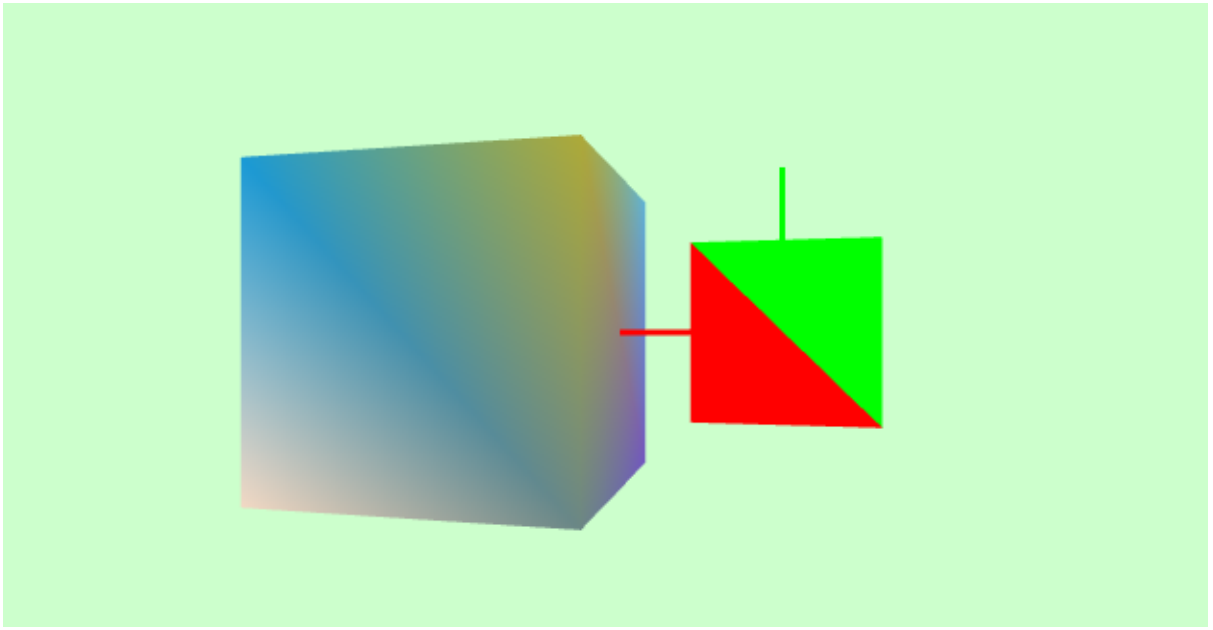

Figure 5 : nos deux cubes après la translation du cube indexé

Puis, on s'entraîne à la transformation matricielle avec de la rotation et de mise à l'échelle sur nos deux objets qui héritent tous les deux des fonctions de renderable.

```
glm::mat4 t = glm::translate(glm::mat4(),glm::vec3(2.0f,0.0f,0.0f));
glm::mat4 r = glm::rotate(glm::mat4(), -60.0f,glm::vec3(0.0f,0.0f,2.0f));
glm::mat4 s = glm::scale(glm::mat4(),glm::vec3(1.0f,1.2f,1.4f));

Indexedcube-> setModelMatrix(t * r *s);

glm::mat4 s2 = glm::scale(glm::mat4(),glm::vec3(0.2f,2.0f,0.2f));

cube ->setModelMatrix(s2);
```

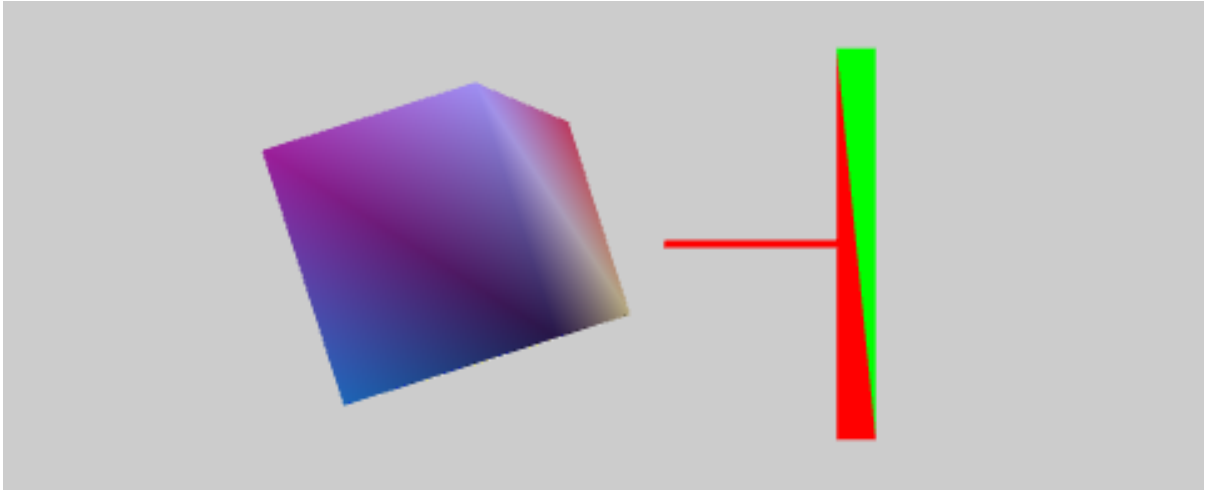

Figure 6 : L'une rotationne, l'autre pas (1)

Enfin, pour simplifier la démarche, nous pouvons utiliser les fonctions get de Utils qui rendent plusieurs fonctions surchargées pour n'avoir qu'à renseigner le vecteur de transformation (et l'angle pour la matrice de rotation).

```
glm::mat4 getTranslationMatrix(const glm::vec3 & tvec){
   return glm::translate(glm::mat4(), tvec);
}
glm::mat4 getRotationMatrix(float angle, const glm::vec3 & avec){
   return glm::rotate(glm::mat4(), angle, glm::normalize(avec));
}
glm::mat4 getScaleMatrix(const glm::vec3 & svec){
   return glm::scale(glm::mat4(), svec);
}
```

Conclusion -

En conclusion, cette première approche sur OpenGL nous a permis de se rendre compte de la difficulté des tâches dans le logiciel et de comprendre la hiérarchie et l'organisation d'un projet OpenGL, Nous avons rencontré quelques difficultés au début mais celles-ci nous ont permis de mieux comprendre à nous servir de OpenGL.

Notes anecdotiques :

(1) : mauvaise référence au titre d'un film d'Agnès Varda : l'une chante, l'autre pas...