Antoine Pizon

Informatique Graphique : TP2

Exercice 1 : Couleur

On cherche ici ajouter un attribut couleur aux différents points.
On ajoute tout d'abord une liste de vec4, dont les coordonnées représentent les valeurs Rouge, Vert, Bleu et Transparence propre à chaque couleur. On donne ensuite une couleur à chaque point de la manière suivante :

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

On crée aussi le buffer approprié avec `glGenBuffer`. Il faut ensuite passer les données au buffer, pour ce faire on lie le buffer avec `glBindBuffer` et on envoie les données avec `glBufferData`.

Il faut ensuite modifier les shaders afin qu'ils prennent en compte cet attribut couleur.
Dans le Vertex Shader, on ajoute l'attribut entrant grâce à `in vec4 vColor`, et l'attribut sortant `out vec4 fragColor`, qui sera ensuite utilisé dans le Fragment Shader.

Dans le Fragment Shader on récupère donc cet attribut avec `in vec4 fragColor` et on le renvoie tel quel avec `out vec4 fragmentColor`.

Ensuite, dans la fonction `do_draw`, on envoie les données du buffer au shader, d'abord en activant l'attribut dans le shader avec `glEnableVertexAttribArray` puis en spécifiant leur format avec `glVertexAttribPointer`.

Enfin on appelle `glDisableVertexAtribArray` pour désactiver l'attribut couleur dans le Vertex Shader.

Exercice 2 : Cube sans Indexing

Pour créer un cube sans utiliser d'indexing, il faut ajouter chaque triangle avec tous les points qui le contiennent, c'est à dire que mêmes les points partagés par plusieurs triangles sont ajoutés plusieurs fois.

On a donc besoin de deux triangles par face, soit 12, et de 3 points pour chacun de ces triangles, soit 36. On ajoute tous ces triangles ainsi que les couleurs qui leur correspondent avec des pushbacks. La classe Utils contient déjà une méthode permettant d'obtenir les coordonnées d'un cube de cette manière.

Les points sont ensuite passés dans les buffers de la même manière que pour un simple triangle avec tout d'abord la génération des buffer (`glGenBuffer`), puis l'envoie des données au buffer (`glBufferData`). L'affichage aussi ne change pas avec d'abord l'activation du buffer auprès du shader (`glEnableVertexAttribArray`),

la spécification du format des données (`glVertexAttribPointer`), l'affichage (`glDrawArrays`) et pour terminer la désactivation (`glDisableVertexAtribArray`).

Exercice 3 : Cube avec Indexing

Cette fois, on se contente de définir les points du cube, puis pour chaque triangle on indique de quel point il est composé. On a donc seulement besoin de 8 points, mais il faudra toujours 12 triangles.

On ajoute d'abord les 8 points avec `positions.push_back(glm::vec3(x,y,z));` Ensuite viennent les indices des 12 triangles avec `indices.push_back(glm::ivec3(i1, i2, i3));`

Le seul changement par rapport au cube non indexé est l'apparition d'un nouveau buffer de type `GL_ELEMENT_ARRAY_BUFFER`, qui contient les indices. Lors de l'affichage, on va cette fois bind ce nouveau buffer après les autres, puis appeler `glDrawElements` à la place de `glDrawArrays`.

Exercice 4 : Transformations basiques

Pour transformer les différents éléments, on va modifier leur `ModelMatrix`, qui correspond à la matrice de position dans la scène, en lui appliquant des transformer grâce aux utilitaires disponibles dans la bibliothèque `glm`.

J'ai donc ajouté trois méthodes dans la classe Renderable :

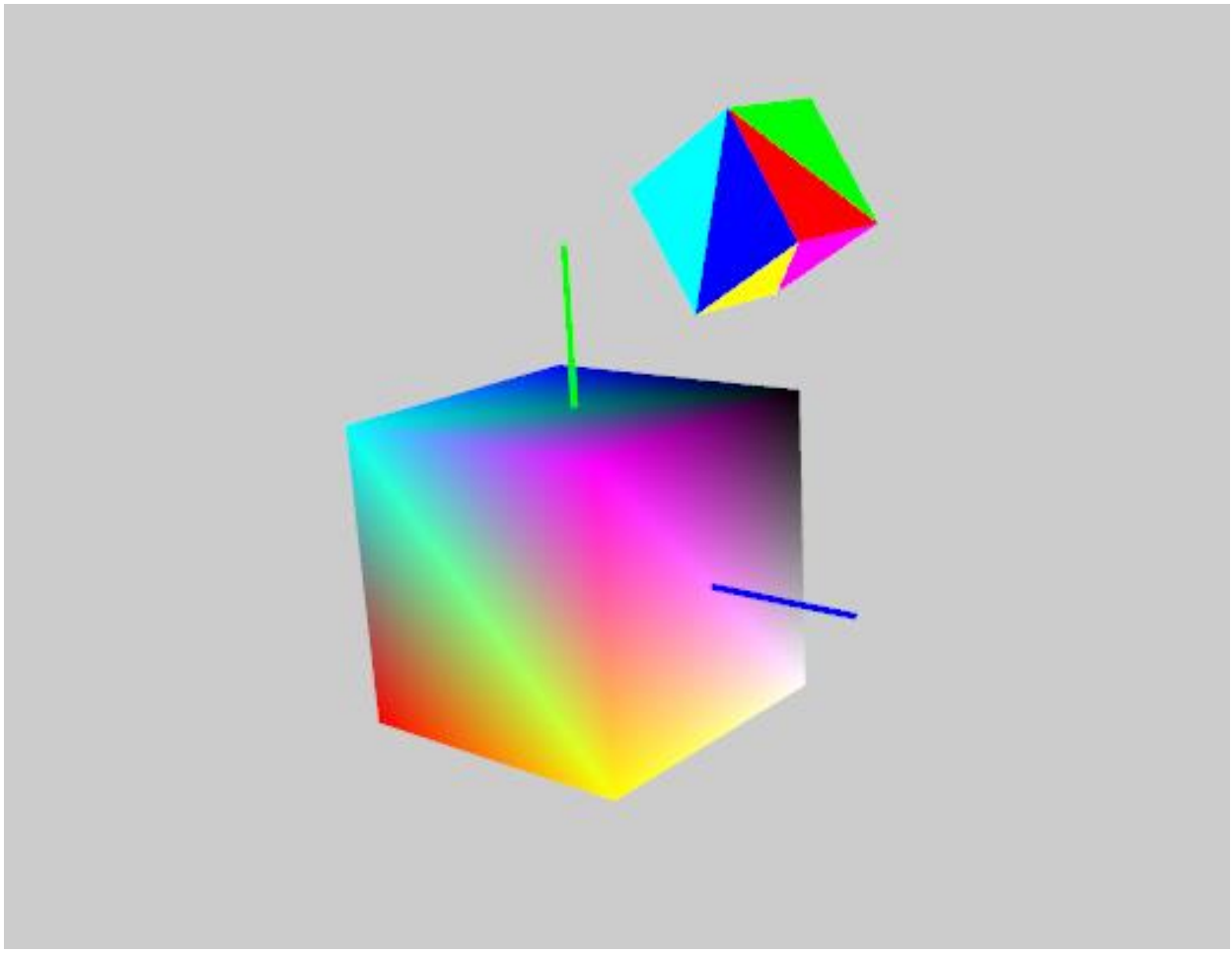
```
void Renderable::translate(float x, float y, float z) {
     setModelMatrix(glm::translate(getModelMatrix(),
                                   glm::vec3(x, y , z)));
}

void Renderable::rotate(float angle, float x, float y, float z) {
     setModelMatrix(glm::rotate(getModelMatrix(), angle,
                                   glm::vec3(x, y , z)));
}

void Renderable::scale(float x, float y, float z) {
     setModelMatrix(glm::scale(getModelMatrix(),
                                   glm::vec3(x, y , z)));
}
```

On appelle ensuite ces méthodes sur notre forme dans le main par exemple :

```
cube->translate(1.0f, 1.0f, 0.0f);
cube->rotate(45, 1.0f, 0.0f, 0.0f);
cube->scale(0.5f, 0.5f, 0.5f);
```

Aucune autre modification du code n'est necessaire, la ModelMatrix est utilisée dans le shader qui applique donc automatiquement la transformation.

Antoine Pizon



Résultat obtenu en utilisant les différentes notions du TP

Sur cette image, le cube central est un cube indexé, chaque point à sa couleur propre et le shader interpole donc les couleurs comprises entre ces points d'où le dégradé.
Le second est un cube non indexé, les transformations indiquées plus haut lui ont été appliqué, dans ce cas chaque triangle (donc les 3 points le composant) a une couleur.