Informatique Graphique : TP4

Interface HierarchicalRenderable

Dans ce TP, on cherche à créer une hiérarchisation des objets de sorte à ce que chaque objet soit localisé et transformé par rapport à un parent, et que le mouvement de ce parent face bouger l'objet en conséquence.

On commence par compléter la méthode `computeTotalParentTransform`, qui est chargée de renvoyer la matrice qui localise notre objet par rapport à la racine de la hiérarchie. On va donc remonter dans ses parents jusqu'à la racine, en ajoutant à chaque fois les différentes transformations.

```
glm::mat4 computeTotalParentTransform() const {
      if( m_parent )
            return m_parent->computeTotalParentTransform()
                        * m_parentTransform;
      return m_parentTransform;
}
```

La fonction est récursive et on a donc deux cas. Soit l'objet est la racine (i.e. il n'a pas de parent) dans quel cas on renvoie sa matrice de transformation. Soit l'objet a un parent dans quel cas on renvoie sa matrice de transformation multipliée par la matrice de toutes les transformations du parent.

On complète ensuite la méthode `updateModelMatrix`, qui se charge de mettre à jour la position de l'objet dans le modèle à chaque frame. Dans cette méthode on se contente de modifier le model de sorte à ce qu'il prenne en compte à la fois les transformations locales de l'objet et ses transformations par rapport à la hiérarchie.

```
void updateModelMatrix() {
      m_model = computeTotalParentTransform() * m_localTransform;
}
```

Manipulation

La création et l'initialisation des objets hiérarchisés est un peu plus longue qu'un objet classique. On commence par créer l'objet sous forme de pointeur partagé comme on le ferait normalement, puis on doit lui définir ses matrices de transformation et enfin définir la hiérarchie.

Chaque objet a deux matrices de transformation, la première représente sa transformation par rapport à son élément parent, la seconde la transformation dans son repère local.

Dans cet exemple j'ai décidé de créer trois cylindres, un père, son fils et son petit-fils. On commence par définir la matrice de transformation du père, qui correspond à la matrice identité.

```
glm::mat4 p_cylParentTransform(1.0f);
p_cyl->setParentTransform(p_cylParentTransform);
```

Vient ensuite le premier fils, ici sa matrice parente est une rotation de 90° selon tous les axes, j'ai créé cette matrice grâce à la fonction `rotate` fournie par `glm`. Cela signifie que le cylindre aura toujours une rotation de 90° par rapport à son père.

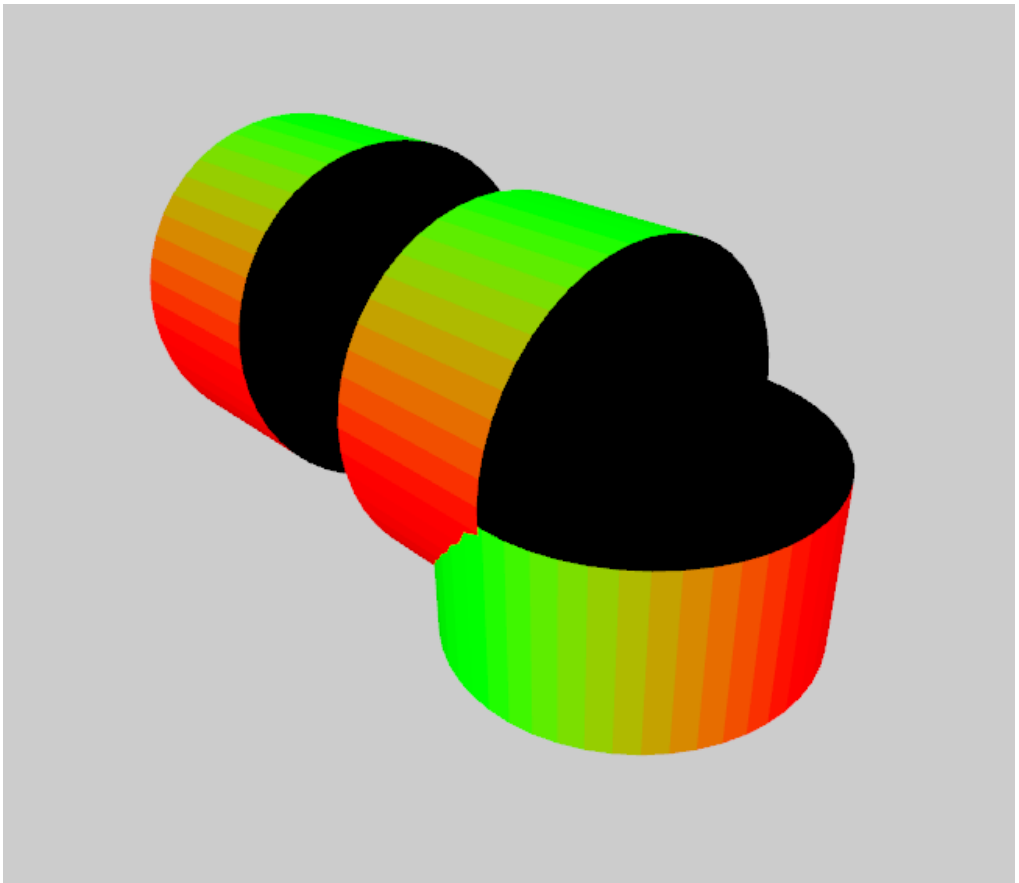
```
glm::mat4 c_cylParentTransform = glm::rotate(glm::mat4(1.0f), 90.0f,
glm::vec3(1.0f, 1.0f, 1.0f));
c_cyl->setParentTransform(c_cylParentTransform);
glm::mat4 c_cylLocalTransform(1.0f);
c_cyl->setLocalTransform(c_cylLocalTransform);
```

On fait ensuite la même chose pour le second fils, mais cette fois avec une translation de deux unités selon l'axe X grâce à la fonction `translate` de `glm`. Cela signifie que le cylindre aura toujours une translation de deux unités par rapport au premier fils, mais il aura aussi toutes les transformations du premier fils par rapport au père, c'est-à-dire ici une rotation de 90°.

```
glm::mat4 cc_cylParentTransform = glm::translate(glm::mat4(1.0f),
glm::vec3(2.0f, 0.0f, 0.0f));
cc_cyl->setParentTransform(cc_cylParentTransform);
glm::mat4 cc_cylLocalTransform(1.0f);
cc_cyl->setLocalTransform(cc_cylLocalTransform);
```

Enfin, on définit les liens de parenté en indiquant les fils de chaque objet.

```
HierarchicalRenderable::addChild(p_cyl, c_cyl);
HierarchicalRenderable::addChild(c_cyl, cc_cyl);
```



Résultat obtenu (de gauche à droite : petit-fils, fils, père)