INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - TP 3

Modélisation

GRAINDORGE Amance

VERNET Hector

Université de Rennes

I. INTRODUCTION

Dans ce TP, nous découvrirons les techniques de modélisation 3D hiérarchique. Nous commencerons par comprendre le fonctionnement de la modélisation hiérarchique et la compléter, puis nous utiliserons cette technique pour construire un arbre, avant de l'appliquer à un objet de notre projet.

II. EXERCICE 1

Ce premier exercice consiste à comprendre le principe de la modélisation 3D hiérarchique, et à compléter deux des méthodes qui participent à son fonctionnement. En particulier, nous allons compléter les méthodes `computeTotalGlobalTransform` et `updateModelMatrix` de la classe `HierarchicalRenderable`.

La première méthode permet de calculer le total des transformations qui sont appliquées à l'objet, à partir du total des transformations appliquées à son parent, et ce récursivement jusqu'à l'objet racine :

```
1   glm::mat4 HierarchicalRenderable::computeTotalGlobalTransform() const
2   {
3     if (m_parent)
4     {
5       // Si l'objet a un parent, renvoyer les transformations de l'objet appliquées aux
tranformations du parent
6       return m_parent->computeTotalGlobalTransform() * m_globalTransform;
7     }
8     else
9     {
10       // Si l'objet est la racine, renvoyer les transformations de l'objet
11       return m_globalTransform;
12     }
13   }
```

La seconde méthode met simplement à jour le modèle de l'objet, qui équivaut aux transformations locales de l'objet appliquées à ses tranformations globales :

```
1   void HierarchicalRenderable::updateModelMatrix()
2   {
3     m_model = computeTotalGlobalTransform() * m_localTransform;
4   }
```

III. EXERCICE 2

Pour ce deuxième exercice, nous allons créer un arbre récursif en utilisant la modélisation 3D hiérarchique. On utilise la méthode récursive suivante pour créer l'arbre :

```
1   void creer_branche(int level, HierarchicalRenderablePtr parent, ShaderProgramPtr
shaderProgram, float lengthMultiplier, float angle, int branching_branches, float
thickness)
2   {
3     if (level == 0)
4     {
5       return;
6     }
7     for (int i = 0; i < branching_branches; ++i) {
8       // Create child
```

```
9       std::shared_ptr<CylinderMeshRenderable> child = std::
10         make_shared<CylinderMeshRenderable>(shaderProgram, false, 20u, false);
11       // Set global transform
12       glm::mat4 childGlobalTransform;
13       childGlobalTransform *= getTranslationMatrix(0.0, 1.0, 0.0);
14       childGlobalTransform *= getRotationMatrix(glm::radians(angle + i * (360.0f /
branching_branches)), glm::vec3(0.0, 1.0, 0.0));
15       childGlobalTransform *= getRotationMatrix(glm::radians(angle), glm::vec3(1.0, 0.0,
0.0));
16       childGlobalTransform *= getScaleMatrix(lengthMultiplier, lengthMultiplier,
lengthMultiplier);
17       child->setGlobalTransform(childGlobalTransform);
18       // Set local transform
19       glm::mat4 childLocalTransform;
20       childLocalTransform *= getScaleMatrix(thickness, 1.0, thickness);
21       childLocalTransform *= getRotationMatrix(glm::radians(-90.0f), glm::vec3(1.0, 0.0,
0.0));
22       child -> setLocalTransform(childLocalTransform);
23       // Define parent / children relationships
24       HierarchicalRenderable::addChild(parent, child);
25       // Recursive call
26       creer_branche(level - 1, child, shaderProgram, lengthMultiplier, angle,
branching_branches, thickness);
27     }
28   }
```

On utilise ensuite le code suivant pour intégrer l'arbre à la scène :

```
1     // Create root object
2     std::shared_ptr<CylinderMeshRenderable> root = std ::
3       make_shared<CylinderMeshRenderable>(flatShader, false, 20u, false);
4     // Set root global transform
5     glm::mat4 rootGlobalTransform;
6     root -> setGlobalTransform(rootGlobalTransform);
7     // Set root local transform
8     glm::mat4 rootLocalTransform;
9     rootLocalTransform *= getScaleMatrix(0.15, 1.0, 0.15);
10     rootLocalTransform *= getRotationMatrix(glm::radians(-90.0f), glm::vec3(1.0, 0.0,
0.0));
11     root -> setLocalTransform(rootLocalTransform);
12     // Create children
13     creer_branche(5, root, flatShader, 0.75f, 35.0f, 4, 0.15);
14     // Add the root of the hierarchy to the viewer
15     viewer.addRenderable(root);
```

Ce code nous donne le résultat suivant :

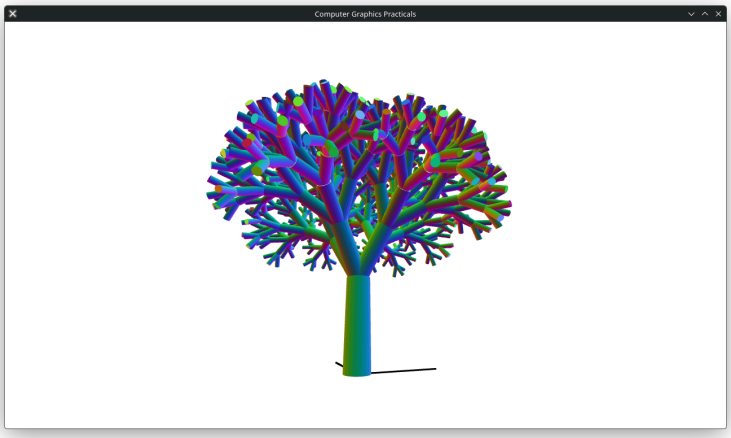

Figure 1 : Arbre récursif hiérarchique

IV. EXERCICE 3

Nous allons enfin appliquer la modélisation hiérarchique à un objet de notre projet. Pour cela, nous allons utiliser le modèle de tortue que nous avons créé avec Blender. Il est composé d'une carapace, qui sera la racine du modèle, de 4 nageoires et d'une tête. De la même façon que dans l'exercice précédent, on ajoute chacun des composants dans la scène :

```
1   void initialize_scene(Viewer& viewer)
2   {
3     // Create a shader program
4     ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(
5         "../../sfmlGraphicsPipeline/shaders/flatVertex.glsl",
6         "../../sfmlGraphicsPipeline/shaders/flatFragment.glsl");
7
8     // Add the shader program to the viewer
9     viewer.addShaderProgram(flatShader);
10
11     const std::string shell_path = "../ObjFiles/Carapace.obj";
12     const std::string nag_ard_path = "../ObjFiles/Nag-ArD.obj";
13     const std::string nag_arg_path = "../ObjFiles/Nag-ArG.obj";
14     const std::string nag_avd_path = "../ObjFiles/Nag-AvD.obj";
15     const std::string nag_avg_path = "../ObjFiles/Nag-AvG.obj";
16     const std::string tete_path = "../ObjFiles/Tete.obj";
17     MeshRenderablePtr shell = std::make_shared<MeshRenderable>(flatShader, shell_path);
18     MeshRenderablePtr nag_ard = std::make_shared<MeshRenderable>(flatShader,
nag_ard_path);
19     MeshRenderablePtr nag_arg = std::make_shared<MeshRenderable>(flatShader,
nag_arg_path);
20     MeshRenderablePtr nag_avd = std::make_shared<MeshRenderable>(flatShader,
nag_avd_path);
21     MeshRenderablePtr nag_avg = std::make_shared<MeshRenderable>(flatShader,
nag_avg_path);
22     MeshRenderablePtr tete = std::make_shared<MeshRenderable>(flatShader, tete_path);
23     HierarchicalRenderable::addChild(shell, nag_ard);
24     HierarchicalRenderable::addChild(shell, nag_arg);
25     HierarchicalRenderable::addChild(shell, nag_avd);
```

```
26      HierarchicalRenderable::addChild(shell, nag_avg);
27      HierarchicalRenderable::addChild(shell, tete);
28      viewer.addRenderable(shell);
29      viewer.addRenderable(nag_ard);
30      viewer.addRenderable(nag_arg);
31      viewer.addRenderable(nag_avd);
32      viewer.addRenderable(nag_avg);
33      viewer.addRenderable(tete);
34    }
```

On obtient ainsi le résultat suivant :

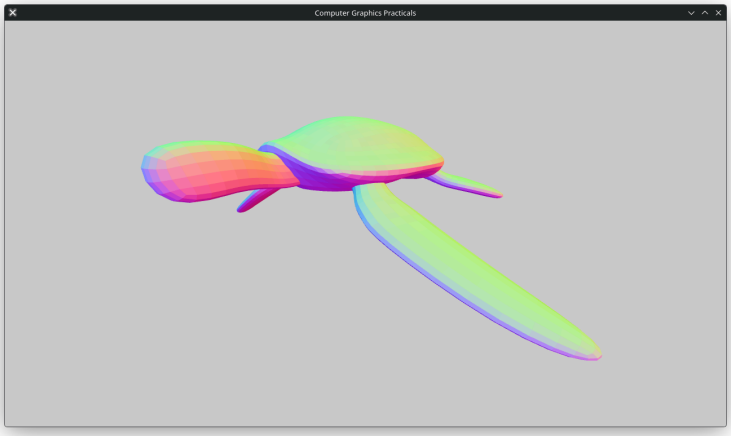

Figure 2 : Arbre récursif hiérarchique

V. CONCLUSION

Dans ce TP, nous avons vu comment utiliser la modélisation hiérarchique pour créer des objects complexes à partir de composants plus simples. Nous avons utilisé cette technique pour construire un arbre récursivement et pour importer notre propre modèle conçu avec Blender. La modélisation hiérarchique sera essentielle pour l'animation, que nous aborderons dans le prochain TP et que nous utiliserons dans notre projet.