Nicolas Luciani
7/03/2023
Master 1 IMAGINE Université de Montpellier

HAI819I : Moteurs de jeux
TP 3
Compte-rendu

---

1 Introduction

L'objectif de ces TPs est de se familiariser avec la création et l'utilisation d'un graphe de scène.

2 Modèles, Entités

Comme nous l'avons vu, un graphe de scène peut être représenté sous la forme d'un arbre, donc chacun des noeud peut être un élément de la scène. Ces éléments, nous les appellerons entités. Un modèle est défini comme un ensemble de maillages. Un maillage est à présent chargé de générer les tampons à envoyer au GPU (VBO, VAO et textures), mais aussi de "se dessiner". Un modèle encapsule ces maillages et leurs services. Un entité est alors une extension d'un modèle, on lui a cependant ajouté deux éléments important qui nous permet de les considérer comme des noeuds du graphe de scène.

- Une instance de la classe Transform
- Une liste d'entités enfants

2.1 Modèle

On a d'abord la classe modèle. Il faut remarquer que la méthode permettant de charger un maillage étant privée, un modèle ne contient pour l'instant qu'un seul maillage. C'est une fonctionnalité qui viendra potentiellement à évoluer dans le futur.

```
1  class Model
2  {
3  public:
4          std::vector<Mesh> meshes;
5
6          Model(const char* path, unsigned int type){loadModel(path, type);}
7
8          void Draw(GLuint shaderID)
9          {
10                 for (size_t i = 0; i < meshes.size(); ++i)
11                 {
12                         meshes[i].draw(shaderID);
13                 }
14         }
15 private:
16         void loadModel(const char* path, unsigned int type)
```

```
        {
                std::vector<unsigned short> indices; //Triangles concaténés dans une liste
            std::vector<glm::vec3> indexed_vertices;
            std::vector<glm::vec2> uvs;
            std::vector<glm::vec3> normals;
             std::vector<std::vector<unsigned short> > triangles;
             Mesh mesh;
            //Chargement du fichier de maillage
            switch(type)
            {
                    case OBJ:
                            loadOBJ(path, indexed_vertices, uvs, normals);
                            indexVBO(indexed_vertices, uvs, normals, indices,
       indexed_vertices, uvs, normals);
                            mesh = Mesh(indexed_vertices, normals, indices, uvs);
                            break;
                    case OFF:
                            loadOFF(path, indexed_vertices, indices, triangles);
                            mesh = Mesh(indexed_vertices, indices, triangles);
                            break;
                    case SQUARE:
                            mesh = static_cast<Mesh>(Square());
                            break;
                    default:
                            mesh = static_cast<Mesh>(Sphere(1, {0.f, 0.f, 0.f}));
                            break;
            }
            meshes.push_back(mesh);
        }
};
```

2.2 Transform

Définissons alors la class Transform telle que nous avons pu la voir en cours. En complétant avec les tutoriels OpenGL, nous avons implémenté la classe suivante.

```
class Transform
{
protected:
    //Local
    glm::vec3 m_pos = { 0.0f, 0.0f, 0.0f };
    glm::vec3 m_eulerRot = { 0.0f, 0.0f, 0.0f }; //In degrees
    glm::vec3 m_scale = { 1.0f, 1.0f, 1.0f };

    //Global space information concatenate in matrix
    glm::mat4 m_modelMatrix = glm::mat4(1.0f);

    // This flag is used to know if the transform has been modified
    bool m_hasMoved = true;

```

```
protected:
    glm::mat4 getLocalModelMatrix()
    {
        const glm::mat4 transformX = glm::rotate(glm::mat4(1.0f),
                    glm::radians(m_eulerRot.x),
                    glm::vec3(1.0f, 0.0f, 0.0f));
        const glm::mat4 transformY = glm::rotate(glm::mat4(1.0f),
                    glm::radians(m_eulerRot.y),
                    glm::vec3(0.0f, 1.0f, 0.0f));
        const glm::mat4 transformZ = glm::rotate(glm::mat4(1.0f),
                    glm::radians(m_eulerRot.z),
                    glm::vec3(0.0f, 0.0f, 1.0f));

        const glm::mat4 rotationMatrix = transformY * transformX * transformZ;

        return glm::translate(glm::mat4(1.0f), m_pos) *
                    rotationMatrix *
                    glm::scale(glm::mat4(1.0f), m_scale);
    }
public:

    void computeModelMatrix()
    {
        m_modelMatrix = getLocalModelMatrix();
        m_hasMoved = false;
    }

    void computeModelMatrix(const glm::mat4& parentGlobalModelMatrix)
    {
        m_modelMatrix = parentGlobalModelMatrix * getLocalModelMatrix();
        m_hasMoved = false;
    }

   /* Getters and Setters */
   //...
};
```

2.3 Entité

Une entité est donc définie comme suit.

```
lass Entity : public Model
{
public:
    std::list<std::unique_ptr<Entity>> children;
    Entity* parent = nullptr;
    Transform transform;
    Entity(const char* path, unsigned int type) : Model(path, type)
    {}
    void addChild(const char* path, unsigned int type)
```

```
    {
        children.emplace_back(std::make_unique<Entity>(path, type));
        children.back()->parent = this;
    }
    //Update transform if it was changed
    void updateSelfAndChild()
    {
        if (transform.hasMoved())
        {
            forceUpdateSelfAndChild();
            return;
        }
        for (auto&& child : children) child->updateSelfAndChild();
    }
    //Force update of transform even if local space don't change
    void forceUpdateSelfAndChild()
    {
        if (parent) transform.computeModelMatrix(parent->transform.getModelMatrix());
        else transform.computeModelMatrix();
        for (auto&& child : children) child->forceUpdateSelfAndChild();
    }
};
```

On voit alors que le graphe de scène n'est pas une classe à part entière mais s'inclut dans la définition d'une entité. Son parcours se fait en parcourant récursivement les enfants d'une entités. Nous pouvons modifier la classe main et créer une entité, lui ajouter un enfant, et appliquer une transformation sur cette entité. Il faudra ensuite appliquer la bonne transformation aux enfants créés.

3 Applications et résultat

Nous avons donc ajouté les lignes suivantes à notre classe principale.

```
/* Before the Main Loop */
Entity sun("../data/models/planet.obj", 0);
sun.addChild("../data/models/planet.obj", 0);
Entity* earth = sun.children.back().get();
earth->addChild("../data/models/planet.obj", 0);
Entity* moon = earth->children.back().get();
moon->transform.setScale({0.3,0.3,0.3});
moon->transform.setLocalPosition({17.0, 0.5, 0.5});
earth->transform.setScale({0.5, 0.5, 0.5});
earth->transform.setLocalPosition({16.0, 0.0, 0.0});
earth->transform.setEulerRot({0.f, 0.f, 23.0f});
sun.updateSelfAndChild();
//...

/* In the main Loop */
Entity * entity = &sun;
while(entity != nullptr)
{
```

```
    glUniformMatrix4fv(modelID, 1, false, glm::value_ptr(entity->transform.getModelMatrix()));
    entity->Draw(programID);
    entity = entity->children.back().get();
}
earth->transform.setEulerRot({0.f, earth->transform.getEulerRot().y + (50.0f * deltaTime), 0.f});
sun.transform.setEulerRot({0.f, sun.transform.getEulerRot().y + (20.0f * deltaTime), 0.f});
sun.updateSelfAndChild();
//...
```

On obtient alors les résultats suivants. On peut cependant remarquer que les textures semblent avoir un problème sur ces modèles. Cela provient du modèle ou du chargement de celui-ci, nous avons estimé que cela pouvait être corrigé ultérieurement.

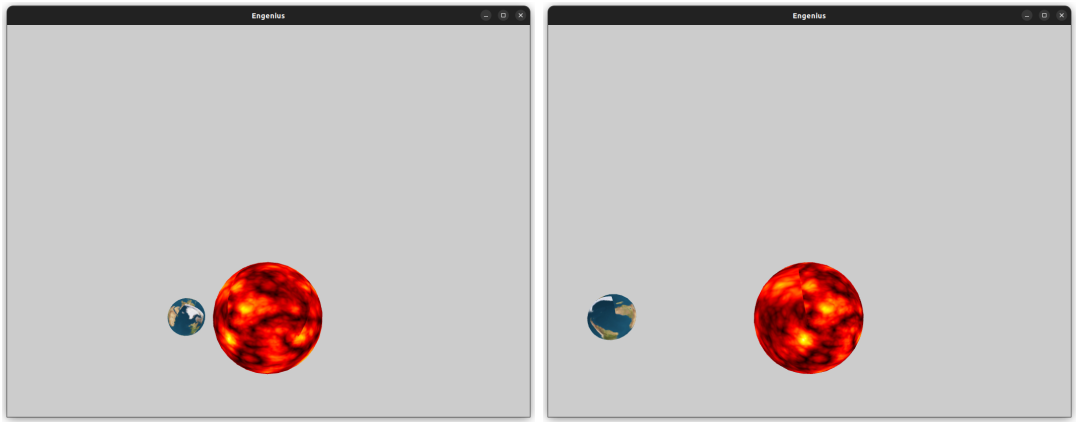

Figure 1: Une terre orbitant autour du soleil

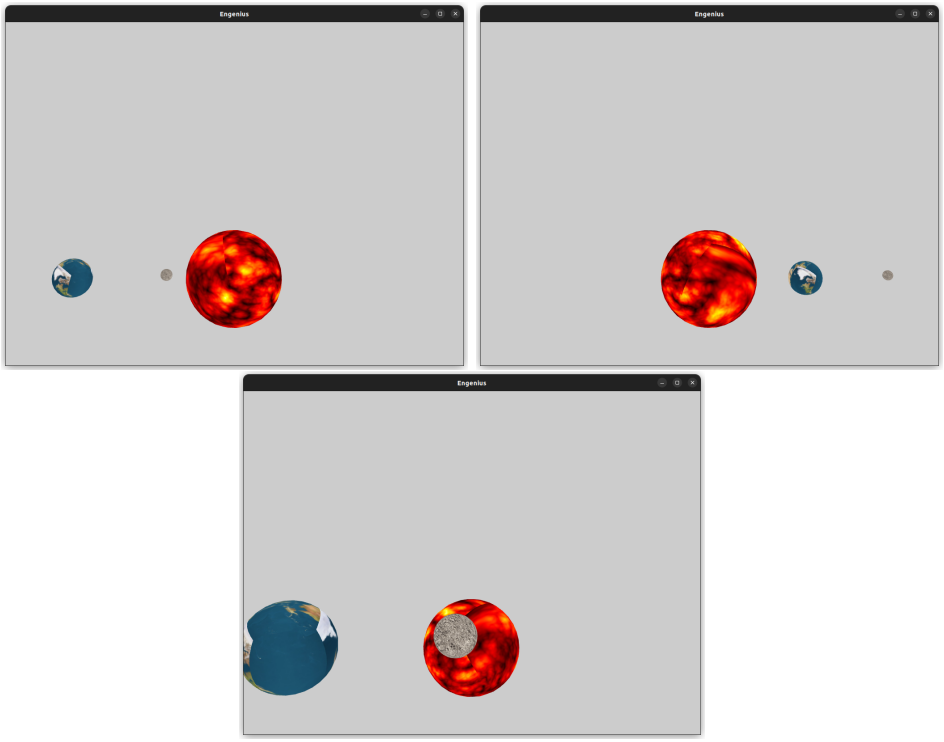

Figure 2: Ajout d'une lune orbitant autour de la Terre

4 Optimisations

4.1 Scène Infinie

Pour représenter une scène infinie, il faudrait repenser les structures de données utilisées et la taille des instances. Il peut potentiellement aussi y avoir des fuites mémoires dans la classe principale (même si des pointeurs dynamiques sont utilisés au niveau de l'entité).

4.2 Rendu temps réel

Pour l'instant, avec le peu d'objet dont la scène dispose, le rendu n'est pas très long. Pour autant avec des objets plus complexes et plus nombreux cela peut poser problème. On peut alors penser à se servir d'enveloppes englobantes pour savoir quels objets sont visibles ou non et s'il y a besoin de les rendre. De plus, il nous serait possible d'ajouter du LOD pour éviter de consommer trop de ressources et garder un rendu temps réel fluide.