HMIN317: Moteurs de jeux

Rendu TP3 : Graphe de scène

Tianning MA

M2 IMAGINA

11/11/2020

Table de matière

1. Introduction

Ce compte rendu est dédié au TP3 Graphe de scène en Qt et en OpenGL ES 3.0.

L'adresse du git est : https://github.com/matianning/Moteurs_de_jeux/tree/master/TP3

2. Fonctionnement des touches et souris

Touche	Fonctionnement
P	Polygone_line / Polygone_Fill
Souris	Drag : Rotation

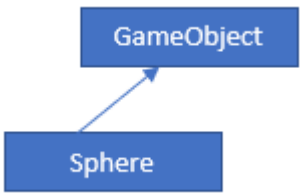
3. Rendu et explication des exercices

Partie 1 Gestionnaire de scène

Tout d'abord, pour avoir plusieurs objets dans la scène, j'ai ajouté un vector de GameObject (qui sera expliqué par la suite) dans le GeometryEngine :

```
std::vector<GameObject *> gameobjects;
```

Pour réaliser le graphe scène, j'ai utilisé la notion de polymorphisme.



GameObject représente chaque élément (objet) dans la scène

Sphere représente une sphere et qui sert particulièrement pour ce tp (système solaire)

Dans le vector de GameObject, grâce au polymorphisme, on pourra ajouter tous les sous-class de GameObject dans le graph de scène.

Dans le vector de components, on pourra spécifier les différents components (composants). Cette attribut n'est pas encore utile pour ce tp, mais sera utile pour la suite de moteur du jeu.

Ensuite, l'attribut transform représente la transformation qui sera appliqué à ce GameObject.

Pour les méthodes :

addChild() et addComponent() est essentiel pour le graphe de scene, pour définir la relation des différents objets dans la scènes et ajouter différents composants à ce objet.

```
class GameObject : protected QOpenGLFunctions_3_1{

protected :
    std::vector<GameObject *> children;
    std::vector<GameComponent> components;
    Transform transform;

public :

    GameObject(){children = std::vector<GameObject*>();
                 components = std::vector<GameComponent>();
                 transform = Transform();}

    virtual void init();   //Initilisation de l'objet
    virtual void update();  //mettre à jour (eg transformation) de l'objet
    virtual void render(QOpenGLShaderProgram *program);  //dessiner (à nouveau)

    void g_translate(const QVector3D & t);
    void g_rotate(const QVector3D & r);
    void g_scale(const QVector3D & scale);

    void addChild(GameObject * child){children.push_back(child);}
    void addComponent(GameComponent component){components.push_back(component);}
    Transform getTransform(){return transform;}

    int indexSize = 0;

};
```

```
void GameObject::init(){

    for(GameComponent component : components){
        component.init();
    }

    for(GameObject * child : children){
        child->init();
    }
}

void GameObject::update(){

    for(GameComponent component : components){
        component.update();
    }

    for(GameObject * child : children){
        child->update();
    }
}

void GameObject::render(QOpenGLShaderProgram *program){

    for(GameComponent component : components){
        component.render();
    }

    if(children.size()!=0){
        for(GameObject * child : children){
            child->render(program);
        }
    }

}
```

Dans chaque instance de GameObject, on va parcourir la liste de children et components pour effectuer les méthodes suivantes pour que les parents et les enfants soient cohérents

Par exemple :

dans la méthode de Init() de GameObject, on va parcourir la liste de children et appeller children[i].init()

Les objectives de ces méthodes sont comme suivants :

Init() : qui sera spécifiée dans les objets (sous-classe) et qui sert à lire les fichiers de maillage (eg . obj) et les sauvegarder dans un tableau de vertex et un tableau d'indice

Update() : qui sera aussi spécifiée dans les sous-classe et qui sert à appliquer les transformations (pour le moment) : multiplier les positions lues dans les fichiers maillages avec la matrice de transformation.

Render() : qui servira à bind les buffers et transmettre les positions au shader de vertex.

Les fonctions concernant les transformations :

g_translate : consiste à mettre à jours la position de transform dans chaque objet

g_rotate : consisite à mettre à jours la rotation de transform dans chaque objet

g_scale : consiste à mettre à jours la mise en échelle dans chaque objet

Puis au moment de update() de chaque objet, on appelle la fonction apply() dans transform de chaque objet, pour calculer la matrice de transformation finale pour ensuite calculer le position de chaque vertex après les transformations.

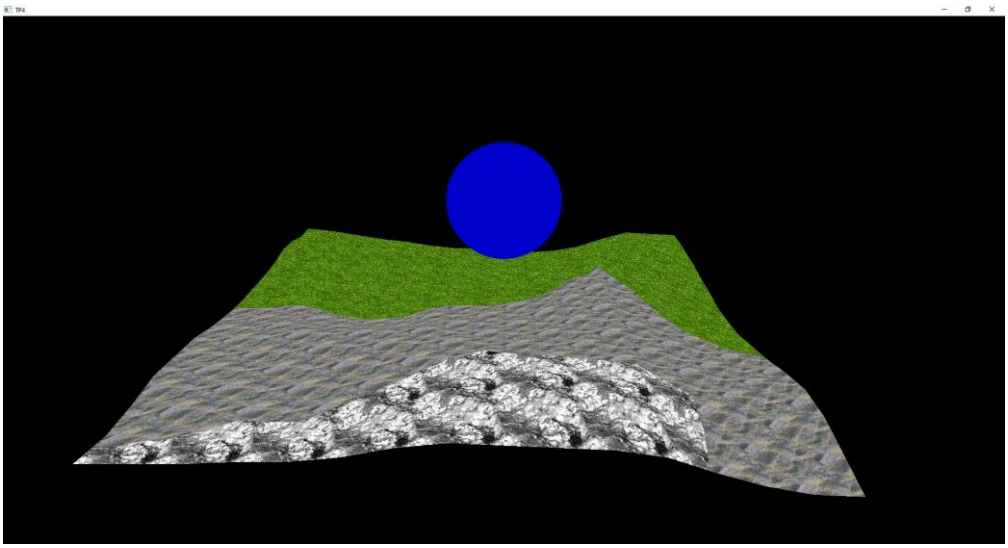
```
void GameObject::g_translate(const QVector3D & t){
    //std::cout<<"translate test"<<std::endl;
    this->transform.setPosition(t);

    for(GameObject * child : children){
        if(children.size()!=0){
            child->g_translate(t);
        }
        else{
            child->transform.setPosition(t);
            child->transform.apply();
        }

    }
    this->transform.apply();
}
```

Attention, ces transformations vont être appliquée aussi pour tous les fils de cet objet. Dans chaque méthode de ces transformation, on va faire itérer le vector de children, pour appliquer la même transformation sur tous les objets d'enfant afin de réaliser des transformations hiérarchiques et cohérentes.

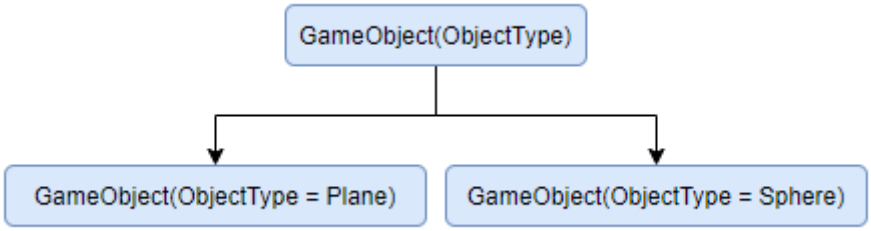
Application sur TP précédent (HeightMap)

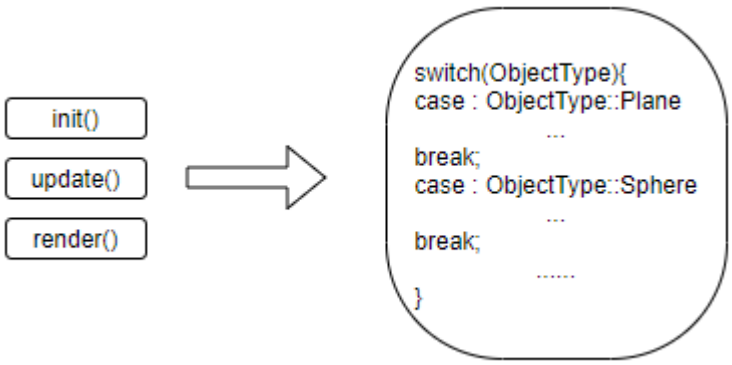
Vous pouvez regarder le code source du TP4 (la détection de collisions et qui est basé sur cela.)

https://github.com/matianning/Moteurs_de_jeux/tree/master/TP4



Pour appliquer le graphe de scène sur TP précédent, j'ai crée un type pour déterminer si c'est le plan ou c'est l'objet que je veux ajouter dans la scène.





Après, dans la classe GameObject, selon les différents type d'objet, on pourra choisir les fonctions correspondantes.

(eg : pour init(), avec le plan, on va appeller la fonction initPlaneGeometry(). Pour une sphère, on appelle la fonction initSphereGeometry().)

Remarque : dans cette scène, on ajoute simplement deux GameObject dans la racine du graphe. Donc les deux GameObject n'ont pas d'enfant.

Donc, dans la classe GeometryEngine, on a un vector pour collecter tous les objets dans la scène, puis on ajoute simplement les deux objets pour réaliser la scène du TP précédent.

```
//*************Création de la scène***************

GameObject plane(objectType::PLANE);
GameObject sphere(objectType::SPHERE);

this->gameObjects.clear();
addObject(plane);
addObject(sphere);
```

Partie 2 Transformation

Pour effectuer les transformations, j'ai crée une classe comme celle-ci :

La classe Transform est composée principalement de 4 attributs essentiels et des methodes :

Attributs :

Vec3 : Position / Rotation / scale

Mat4x4 : Matrice de transformation à appliquer

```h
#ifndef TRANSFORM_H
#define TRANSFORM_H

#include <QMatrix4x4>


class Transform {

public :
    Transform(const QVector3D & position = QVector3D(0.0f,0.0f,0.0f),
              const QVector3D & rotation = QVector3D(0.0f,0.0f,0.0f),
              const QVector3D & scale = QVector3D(1.0f,1.0f,1.0f));

    QMatrix4x4 apply();
    QVector3D & getPosition() { return position; }
    QVector3D & getRotation() { return rotation; }
    QVector3D & getScale() { return scale; }
    QMatrix4x4 & getOriginalMatrix() { return OriginalMatrix; }
    QMatrix4x4 & getAppliedMatrix(){return AppliedMatrix;}


    void setPosition(const QVector3D & pos) { position = pos; }
    void setRotation(const QVector3D & rot) { rotation = rot; }
    void setScale(const QVector3D & scal) { scale = scal; }

private :
    QVector3D position;
    QVector3D rotation;
    QVector3D scale;
    QMatrix4x4 OriginalMatrix;
    QMatrix4x4 AppliedMatrix;

};
```

Cette classe consiste à calculer la matrice de 4x4 à partir de position, rotation et scale donnée. Selon ces trois composants, on pourra appliquer directement à la matrice originale pour obtenir une matrice de transformation (qui sera appliquée sur les positions de chaque point de maillage)

```cpp
#include "Transform.h"

Transform::Transform(const QVector3D & pos  , const QVector3D & rot , const QVector3D & scal)
    : position(pos), rotation(rot), scale(scal)
{
    OriginalMatrix.setToIdentity();
    AppliedMatrix.setToIdentity();
}

QMatrix4x4 Transform::apply(){

    AppliedMatrix.translate(position);
    AppliedMatrix.rotate(rotation.x(), QVector3D(1.0f, 0.0f, 0.0f));
    AppliedMatrix.rotate(rotation.y(), QVector3D(0.0f, 1.0f, 0.0f));
    AppliedMatrix.rotate(rotation.z(), QVector3D(0.0f, 0.0f, 1.0f));
    AppliedMatrix.scale(scale);

    return AppliedMatrix;
}
```

Les attributs :

OriginalMatrix consiste à sauvgarder la matrix original de cette transformation (pour l'instant, cette variable est inutile, mais après réflexion, je pense ça serait utile pour faire les transformations inverse pour ne pas refaire le calcul)

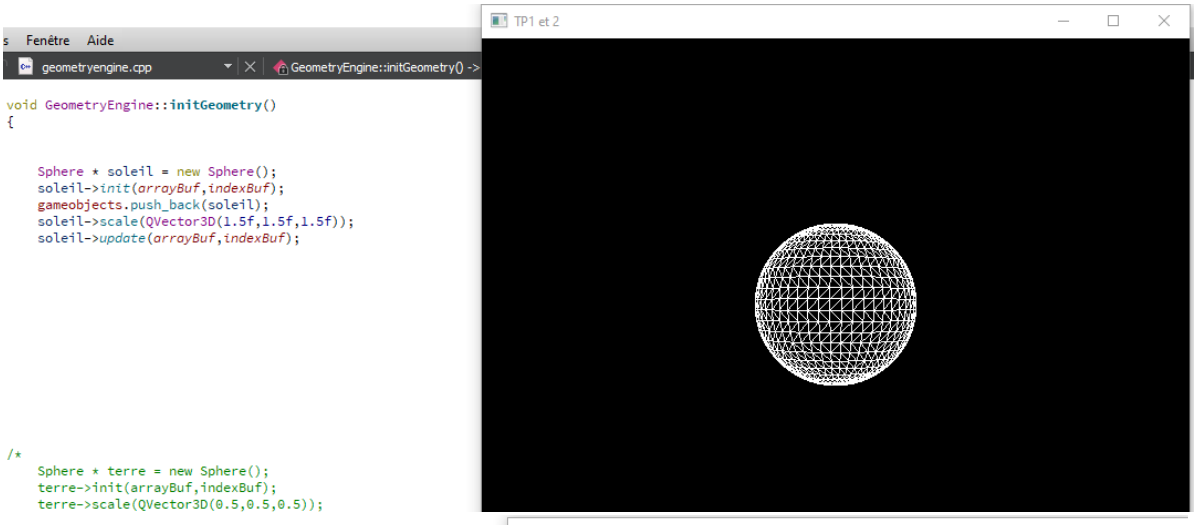
AppliedMatrix consiste à sauvegarder la matrix finale (la matrice calculée après toutes les transformations)

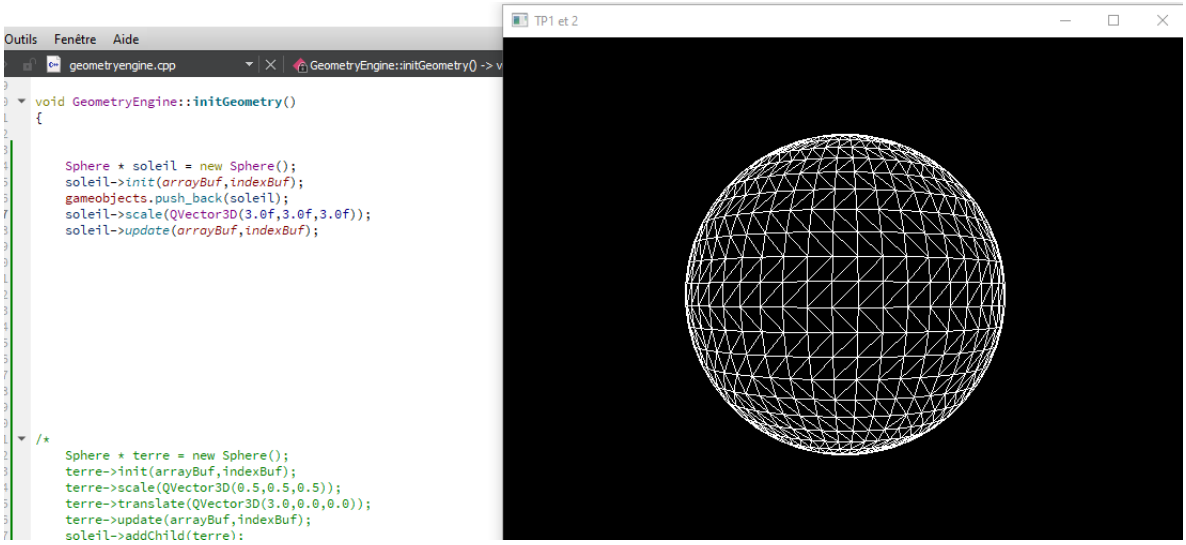
La fonction apply() est le coeur de cette classe et qui retourne la martrice AppliedMatrix comme résultat final de transformation. Cette matrice-là servira à calculer la position d'objet finale dans la scène.

Voici quelleque images pour vous montrer la classe transformation fonctionne.

(Pour asscocier la transform à un objet de la scène, j'ai mis également une attribut Transform dans la classe de GameObject.)

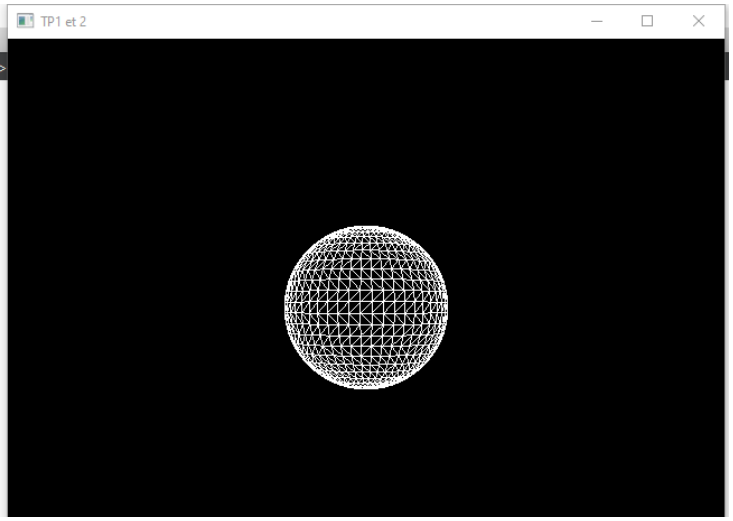
SCALE



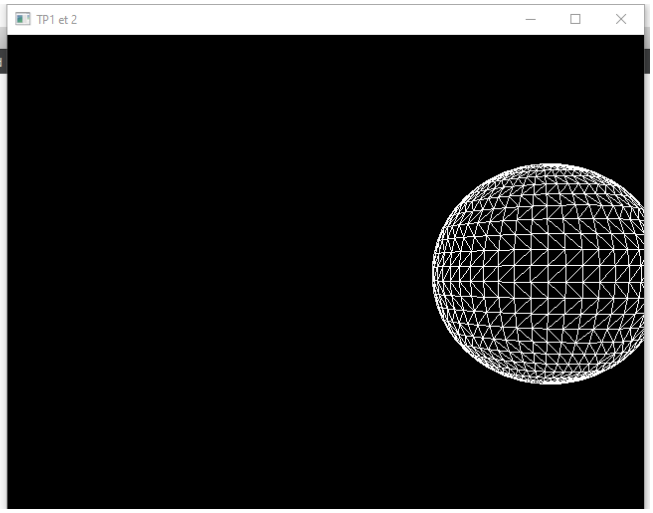


Translate

```
Fenêtre  Aide
geometryengine.cpp | GeometryEngine::initGeometry() ->

void GeometryEngine::initGeometry()
{

    Sphere * soleil = new Sphere();
    soleil->init(arrayBuf,indexBuf);
    gameobjects.push_back(soleil);
    soleil->scale(QVector3D(1.5f,1.5f,1.5f));
    soleil->update(arrayBuf,indexBuf);

/*
    Sphere * terre = new Sphere();
    terre->init(arrayBuf,indexBuf);
    terre->scale(QVector3D(0.5,0.5,0.5));
```



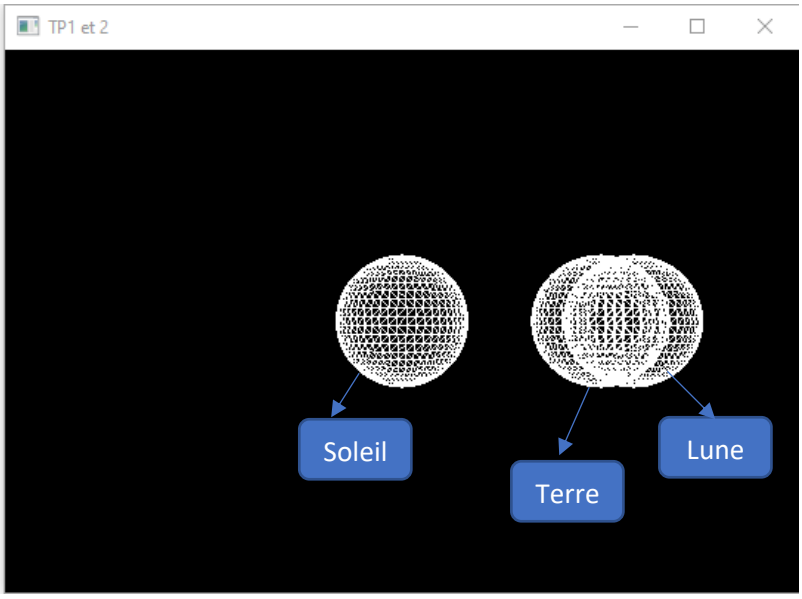

```
Outils  Fenêtre  Aide
geometryengine.cpp | GeometryEngine::initGeometry() -> void

void GeometryEngine::initGeometry()
{

    Sphere * soleil = new Sphere();
    soleil->init(arrayBuf,indexBuf);
    gameobjects.push_back(soleil);
    soleil->scale(QVector3D(1.5f,1.5f,1.5f));
    soleil->translate(QVector3D(3.0f,0.0f,0.0f));
    soleil->update(arrayBuf,indexBuf);

/*
    Sphere * terre = new Sphere();
    terre->init(arrayBuf,indexBuf);
    terre->scale(QVector3D(0.5,0.5,0.5));
    terre->translate(QVector3D(3.0,0.0,0.0));
    terre->update(arrayBuf,indexBuf);
```



Rotation (n'est pas évident pour la sphère. Donc je n'ai pas fait la capture d'écran)

Transformation par héritage :

```
}
//! [0]

void GeometryEngine::initGeometry()
{

    GameObject * soleil = new Sphere();
    GameObject * terre = new Sphere();
    GameObject * lune = new Sphere();

    lune->init();

    lune->g_translate(QVector3D(1.0,0.0,0.0));

    terre->init();
    terre->addChild(lune);
    terre->g_translate(QVector3D(3.0,0.0,0.0));

    soleil->init();
    soleil->addChild(terre);
    soleil->g_scale(QVector3D(2.0f,2.0f,2.0f));
    soleil->update();

    gameobjects.push_back(soleil);
```

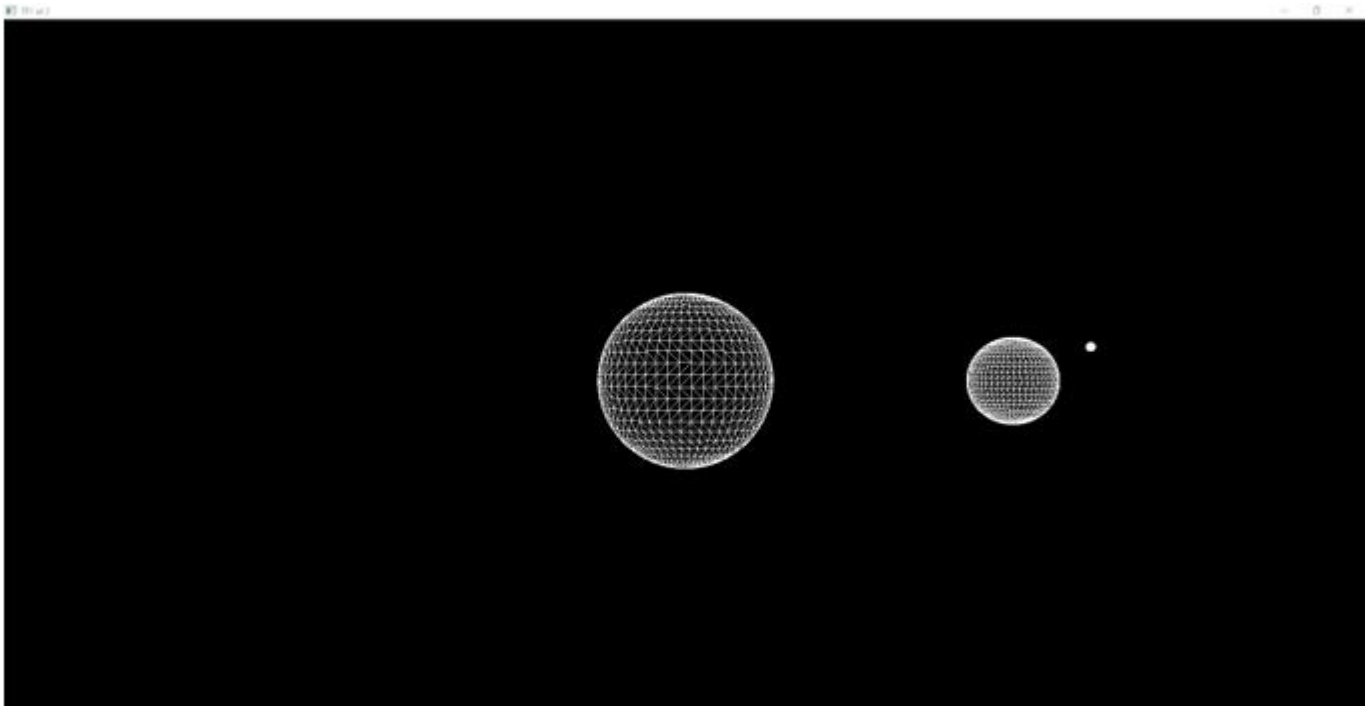


L'image ci-dessus est juste un exemple pour montrer que la transformation par héritage fonctionne. En effet, tout d'abord, on voit que le scale est appliqué au « soleil » qui a un enfant « terre » et qui a aussi un enfant « lune ». et La transformation scale est bien appliqué au soleil et son enfant (terre) et aussi son enfant d'enfant (lune).

Ensuite, on voit que la translation (x+1) de lune est bien appliqué à la base de translation de son parent (terre) qui est translaté en axe x +3.

Donc, la transformation par héritage est bien fonctionnée.

Finalement pour le système solaire :



Après certaines transformations appliquée respectivement à soleil / terre / lune, j'obtient le résultat comme l'image ci-dessus.

Problème Rencontré et solution :

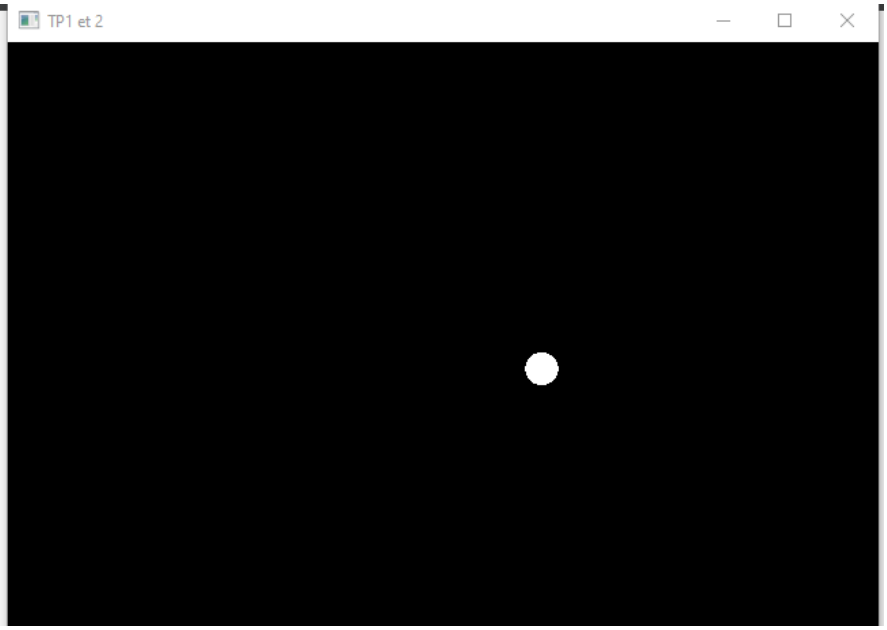
Problème lié à la gestion de buffers :

Le plus grand problème que j'ai rencontré est sur la fonction drawGeometry() :

En effet, si j'ajoute les objets dans la scène, seul le dernier élément sera dessiné dans la scène. (comme l'image montrée ci-dessous : Le soleil et la terre sont ajoutés dans la scène, mais que la terre est dessiné dans la fenêtre)

A cause de manque de temps, je n'ai pas encore réussi à résoudre ce problème. Je vais continuer à chercher la solution pour la suite et le projet de cette matière.

```
GeometryEngine::~GeometryEngine()
{

        arrayBuf.destroy();
        indexBuf.destroy();


}
//! [0]

void GeometryEngine::initGeometry()
{


    Sphere * soleil = new Sphere();
    soleil->init(arrayBuf,indexBuf);
    gameobjects.push_back(soleil);
    soleil->scale(QVector3D(1.5f,1.5f,1.5f));
    soleil->update(arrayBuf,indexBuf);

    Sphere * terre = new Sphere();
    terre->init(arrayBuf,indexBuf);
    terre->scale(QVector3D(0.5,0.5,0.5));
    terre->translate(QVector3D(3.0,0.0,0.0));
    terre->update(arrayBuf,indexBuf);
    gameobjects.push_back(terre);


}
```



Pour résoudre ce problème, j'ai décidé de mettre les buffers internes dans chaque GameObject. Puisque les fonctions bind() pour les buffers doivent être utilisé dans le même context que le create, je pense que c'est ici qui poserait des problèmes. Après beaucoup de recherches et plusieurs essaies, donc j'ai mis le vertex buffer et l'index buffer dans chaque gameObject, et dans ses objets, ils s'occupent eux-même le stockage des vertexs et indices. Cela a bien résolu le problème.