Compte Rendu

TP 3 Moteur de Jeux

Ayoub GOUSSEM

21 mai 2024

Lien vers le répertoire Git : Cliquez ici

1 Graphe de scène

Pour gérer de manière hiérarchique et récursive les meshs ainsi que leurs transformations, un graphe de scène sous forme d'arbre est créé.

Cet arbre a pour objectif de stocker, dans chaque nœud, un maillage et les propriétés associées, telles que les tableaux de sommets, les indices, la matrice de transformation, etc. Un élément crucial de cette structure est que chaque nœud possède un nœud parent ainsi qu'une liste de nœuds enfants.

Grâce à cette organisation, une hiérarchie entre les maillages est établie. En effet, chaque nœud enfant hérite des transformations de son nœud parent via cette méthode.

```
glm::mat4 getGlobalTransform() const
{
    if (parent == nullptr)
    {
        return localTransformMatrix;
    }
    else
    {
        return parent->getGlobalTransform() * localTransformMatrix;
    }
}
```

2 Transformation

Pour pouvoir gérer cette structure de graphe de scène, chaque nœud doit avoir une transformation qui lui est propre. Cela permet, entre autres, d'appeler la fonction précédemment vue pour récupérer toutes les transformations des nœuds supérieurs. Ainsi, chaque nœud possède une matrice 4 x 4 appelée "transform_" qui permet de suivre les transformations appliquées au nœud courant.

Cette matrice est initialisée à la matrice identité lors de la création du nœud. Les opérations qu'elle peut subir sont les suivantes :

```
void translate(const glm::vec3 &translation)
{
    localTransformMatrix = glm::translate(localTransformMatrix, translation);
}

void scale(const glm::vec3 &scale)
{
```

```
    localTransformMatrix = glm::scale(localTransformMatrix, scale);
}

void rotate(float angle, const glm::vec3 &axis)
{
    localTransformMatrix = glm::rotate(localTransformMatrix, angle, axis);
}
```

En appelant la méthode "getFullTransform()", la transformation dans le repère monde est obtenue. Cette matrice est ensuite envoyée dans un buffer au GPU afin qu'elle puisse être appliquée sur les différents éléments du graphe de scène dans le vertex shader.

```
if(isMesh)
{
    gl_Position = project_mat * view_mat * localTransformMatrix * vec4(pos,1);
} else
{
    float height = texture(texture0,textureCoordinates).r;
    pos.y += height;
    TexCoord = textureCoordinates;

    gl_Position =  project_mat * view_mat * model_mat * vec4(pos,1);
}
```

3 Système solaire

Afin de tester tout cela, voici la création de notre scène :

```
Mesh *genScene1()
{
    Mesh *sun = new Mesh("../mesh/off/sphere.off", "sun");
    Mesh *earth = new Mesh("../mesh/off/sphere.off", "earth");
    Mesh *moon = new Mesh("../mesh/off/sphere.off", "moon");

    sun->addChild(earth);
    earth->addChild(moon);

    sun->translate(glm::vec3(0., 1.3, 0.));
    sun->scale(glm::vec3(0.3));
    sun->rotate(glfwGetTime()*0.5, glm::vec3(0., 1, 0.2));

    earth->translate(glm::vec3(-4, 0, 0));
    earth->scale(glm::vec3(0.5));
    earth->rotate(glfwGetTime() * 0.25f, glm::vec3(0., 1, 0.2));
    earth->rotate(glm::radians(23.0f), glm::vec3(1.0f, 0.0f, 0.0f));

    moon->translate(glm::vec3(2, 0, 0));
    moon->scale(glm::vec3(0.2));
    moon->rotate(glm::radians(6.0f), glm::vec3(0.0f, 1.0f, 1.0f));
```

```
22      return sun;
23  }
```

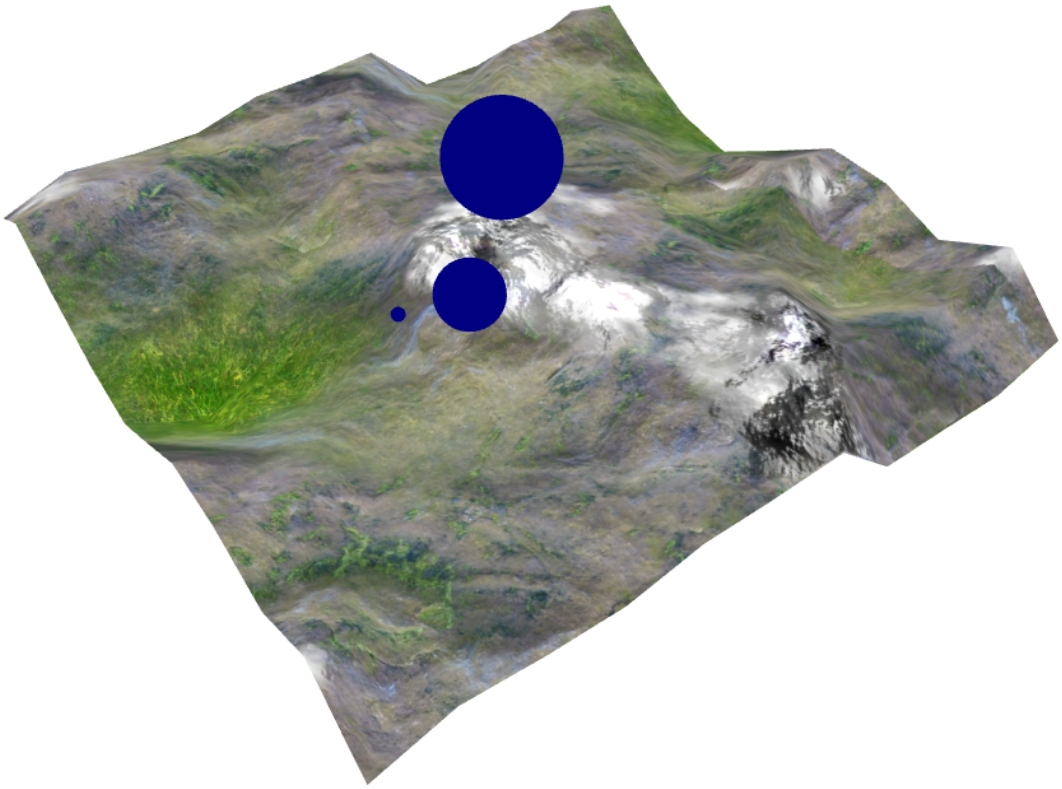

FIGURE 1 – Plan avec système solaire

4 Commandes clavier

— z,q,s,d,a,e : déplacement libre de la caméra
— 3, 4 : augmenter / diminuer la résolution du plan
— p : activer le mode orbite libre
— o : activer le mode orbite automatique
— r,f : accélérer / ralentir la rotation de l'orbite
— w, x : mode de rendu, polygon / wired