Université de Montpellier

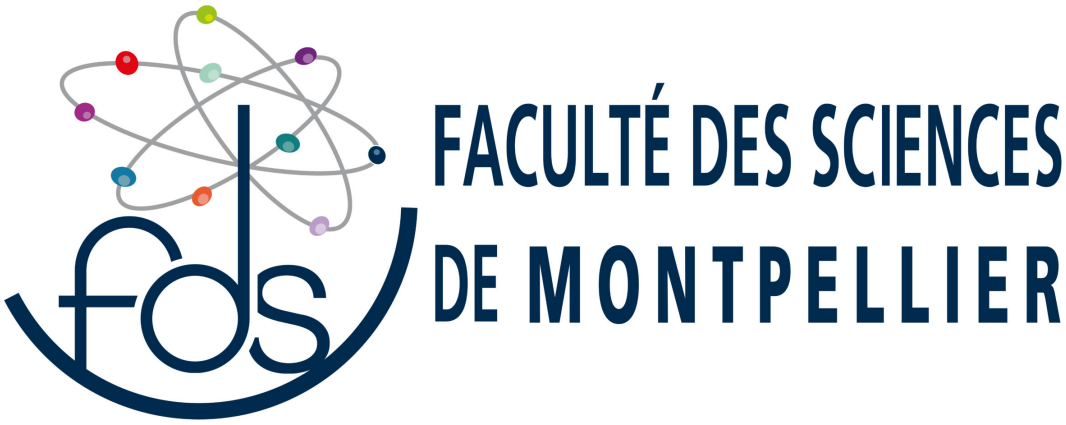
M1 IMAGINE - Moteur de jeu TPn°1/2
Compte Rendu - Géneration d'un plan, manipulation de camera, texture et heightmap

---

Etudiant :
Guillaume Bataille

Encadrant :
Noura Faraj

Le git de moteur de jeu ici : Cliquez ici



0.1 Contexte et objectif

Lors des différents cours, nous avons vu l'interêt de structurer les objets de nos scènes afin de répercuter des transformations de certains objets sur d'autres.

L'objectif de ces TPs est donc de :
- Créer un graphe de scène
- Créer de quoi appliquer des transformations sur un mesh
- Tester le tout sur un exemple concret

Contrôles :
Z et S : zoom in/out
Fleches directionnelles : Bouger la camera
O : Active le mode de vue Orbite (Avec I et K pour accélerer/ralentir la rotation)
P : Active le mode de vue Stationnaire
W : Active le mode de rendu wireframe (Line)
X : Active le mode de rendu Fill
+ et - : Augmente la résolution du plane

Sommaire

0.1 Contexte et objectif 1

1 Graphe de scène **3**

2 Transformation **4**

3 Mini système solaire **5**

4 Conclusion **7**

5 **8**

1. Graphe de scène

Afin de pouvoir gérer de façon hierarchique et recursive nos meshs et leurs transformations. Nous avons décidé de créer un graphe de scène.
Nous choisissons pour cela de faire un arbre.
L'objectif de cet arbre est de stocker à chaque noeud un mesh et des propriétés qui lui sont associés, comme ses tableaux index vertices, indices, sa matrice de transformation etc.. Mais le plus important est que chaque noeud possède un noeud parent et une liste de noeud enfant.
Avec cela, nous avons une forme de hierarchie dans nos meshes.
En effet, on se dit que chaque enfant va hériter des transformations parentes via cette méthode :

```
1   glm::mat4 getFullTransform() const
2      {
3          if (parent_ == nullptr)
4          {
5              return transform_;
6          }
7          else
8          {
9              return parent_->getFullTransform() * transform_;
10         }
11     }
```

Notre graphe de scène a donc :
- Un constructeur
- Un destructeur
- De quoi lui ajouter des enfants
- De quoi gérer ses transformations locales
- De quoi récupérer la transformation globale
- De quoi dessiner le mesh dans le repère global

2. Transformation

Pour pouvoir gérer cette structure graphe de scène, il faut pouvoir attribuer a chaque noeud une transformation qui lui est propre. Cela permet, entre autre, d'appeller la fonction vue précédement pour récupérer toutes les transformations des noeuds supérieur. C'est pour cela qu'a chaque noeud, j'ai une matrice 4 x 4 "transform_" qui me permet de tracker les transformations subies par le noeud courant.
Elle est initialisé a la matrice identité lors de la création du noeud et voici les opérations qu'elle peut subir :

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

    void scale(const glm::vec3 &scale)
    {
        transform_ = glm::scale(transform_, scale);
    }

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

```

Et en appellant la méthode "getFullTransform()", on obtient la transformation dans le repère global (monde). C'est cette matrice que l'on envoie dans un buffer au GPU pour que dans le vertex shader on puisse l'appliquer sur nos différents éléments du graphe de scène.

```
// Dans le main du vertexshader
  if(isMesh) // Si on a affaire a un mesh, c'est ça propre matrice de transformation qui est utilisé
        {
        gl_Position = project_mat * view_mat * transform_ * vec4(pos,1);
        }
        else // Sinon, pour l'instant y'a que le plan donc on le traite comme un plan
        {
        float height = texture(texture0,textureCoordinates).r;
        pos.y += height;
        gl_Position =view_mat * transform_mat_plane * vec4(pos,1);
        TexCoord = textureCoordinates;
        gl_Position = project_mat * gl_Position;
        }
```

3. Mini système solaire

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

```
// Set rotation and orbit speeds
    float sunRotationSpeed = 0.1f;
    float earthRotationSpeed = 0.6f;

    // Create node of my scenegraph
    Mesh *scene1 = new Mesh("");
    Mesh *sun = new Mesh("../mesh/off/sphere.off");
    Mesh *earth = new Mesh("../mesh/off/sphere.off");
    Mesh *moon = new Mesh("../mesh/off/sphere.off");

    // Bind my scenegraph together
    scene1->addChild(sun);
    sun->addChild(earth);
    earth->addChild(moon);

    // Set sun transform
    sun->translate(glm::vec3(0., 1, 0.));
    sun->scale(glm::vec3(0.2));
    sun->rotate(sunRotationSpeed * glfwGetTime(), glm::vec3(0., 1, 0.));

    // Set earth transform
    earth->translate(glm::vec3(-3, 0, 0));
    earth->scale(glm::vec3(0.5));
    earth->rotate(earthRotationSpeed * glfwGetTime(), glm::vec3(0., 1, 0.));
    earth->rotate(glm::radians(23.0f), glm::vec3(1.0f, 0.0f, 0.0f)); // Inclinaison terre

    // Set moon transform
    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, 0.0f)); // Inclinaison lune
```

Nous avons testé la liaison et la repercussion des transformations parentes sur les enfants et cela semble bien fonctionner. Bouger la terre affecte bien la lune et la terre sans modifier le soleil.

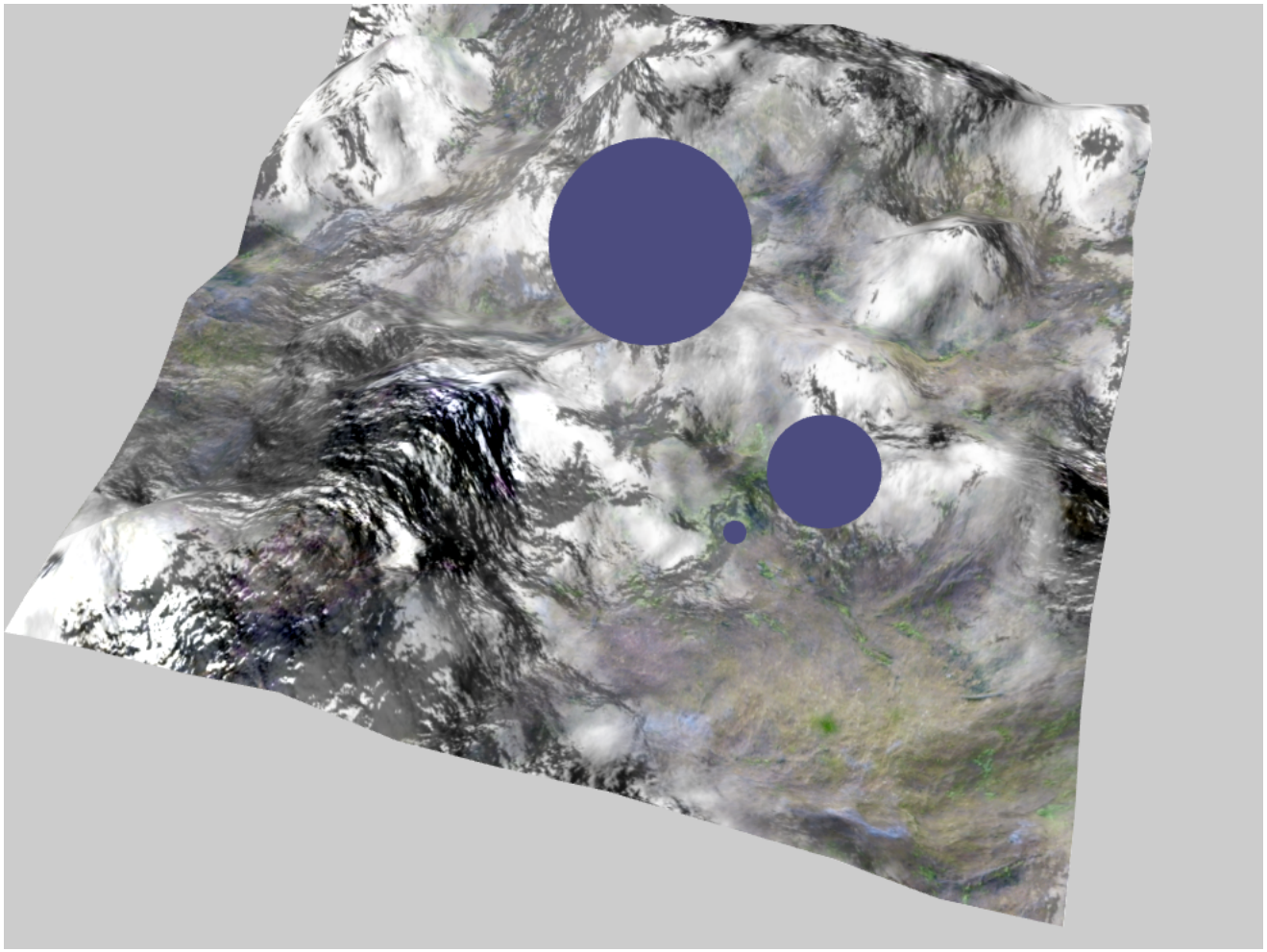


FIGURE 3.1 – Plane du tp précédent + Système solaire

Le screenshot ne rends pas hommage aux axes de rotation inclinés des spheres mais ils sont bien implémentés.

4. Conclusion

Dans ce tp, nous avons instaurer une structure de scenegraph afin de hierarchiser nos scènes. Et nous avons aussi ré-utilisé les transformations vu le tp3 de programmation 3D du semestre précédent.

5.