Moteur de jeux

Compte-rendu TP3

Graphe de scène

Louis Jean
Master 1 IMAGINE
Université de Montpellier
N° étudiant : 21914083

7 mars 2024

Table des matières

1 Introduction

L'objectif de ce travail est de développer une compréhension approfondie des graphes de scènes et de leur impact sur la structuration et l'optimisation des environnements 3D complexes.

N.B : Pour ce TP, j'ai repris la base de code que nous utilisons pour le projet de TER (OceanGL), et ai itéré dessus en ajoutant de nombreuses classes, comme **Mesh, Sphere, Texture,** ... Pour build le projet puis l'exécuter :

```
mkdir build
cd build
cmake ..
./main
```

Aussi, lors de ce TP, j'ai rencontré un bug in-com-pré-hen-sible : mon code ne fonctionnait que si avant chaque exécution, je modifiais un des deux shaders (par exemple simplement ajouter un espace et faire Ctrl+S). Autrement, j'avais une erreur *bad_alloc* à l'exécution. J'ai perdu énormément de temps sur ce problème. Pour remédier à cela, j'ai du faire un script **run_main.sh**, qui modifie automatiquement le fragment shader avant chaque exécution. N'hésitez pas à vous en servir si vous rencontrez le même problème. Je pense que ce problème était dû à la mise en cache des shaders compilés.

2 Mise en place du graphe de scène

Pour matérialiser le graphe de scène, j'ai choisi d'écrire une classe **SceneNode**, qui correspond à un nœud de ce dernier.

```
class SceneNode {
    public:
        // Public attributes
        SceneNode* parent = nullptr;
        Mesh* mesh;
        Transform transform;
        std::vector<SceneNode*> children;
        // Constructors
        SceneNode();
        SceneNode(Mesh *mesh);
        // Destructor
        ~SceneNode();
        // Public methods
        void set_parent(SceneNode *parent);
        void add_child(SceneNode *child);
        glm::mat4 get_world_transform();
        void draw();
};
```

Figure 1: Classe SceneNode

Pour initialiser l'affichage du graphe de scène, on appelle **draw()** sur la racine, qui se chargera de dessiner tous les fils.

```
void SceneNode::draw() {
    if(mesh) {
        glm::mat4 model = get_world_transform();
        mesh->shader.useShader();
        mesh->shader.setBindMatrix4fv("model", 1, 0, glm::
    value_ptr(model));
        mesh->draw();
    }
    for(SceneNode* child : children) {
        child->draw();
    }
}
```

Figure 2: Méthode draw() de SceneNode

3 Implémentation des transformations

Ma classe **Transform** permet de réaliser des translations, rotations et mises à l'échelle.

```
class Transform {
    public:
        // Public attributes
        glm::vec3 translation;
        glm::vec3 rotation;
        glm::vec3 scale;
        // Constructors
        Transform();
        Transform(glm::vec3 translation, glm::vec3 rotation,
   glm::vec3 scale);
        // Public methods
        glm::mat4 get_matrix();
};
```

Figure 3: Classe Transform

On met à jour la transformation selon les valeurs de *translation,rotation* et *scale*.

```
glm::mat4 Transform::get_matrix() {
    glm::mat4 mat = glm::mat4(1.0f);
    mat = glm::translate(mat,this->translation);
    mat = glm::rotate(mat, glm::radians(rotation.x), glm::
   vec3(1.0f,0.0f,0.0f));
    mat = glm::rotate(mat, glm::radians(rotation.y), glm::
   vec3(0.0f,1.0f,0.0f));
    mat = glm::rotate(mat, glm::radians(rotation.z), glm::
   vec3(0.0f,0.0f,1.0f));
    mat = glm::scale(mat,scale);
    return mat;
}
```

Figure 4: Implémentation de get_matrix()

La transformation est propagée grâce à la méthode **get_world_transform()** de la classe **SceneNode**.

```
glm::mat4 SceneNode::get_world_transform() {
    if(parent) {
        return parent->get_world_transform() * transform.
    get_matrix();
    }
    else {
        return transform.get_matrix();
    }
}
```

Figure 5: Implémentation de get_world_transform()

4 Création du système solaire

Pour créer le système solaire, j'ai créé une classe **Sphere** qui crée une sphère de manière explicite pour faire les planètes, car je n'ai pas réussi à charger des objets via Assimp.

```
// Sun
Mesh* sun_mesh = new Sphere(glm::vec3(0.0f,0.0f,0.0f),1.0f
    ,50,50);
sun_mesh->add_texture(sun_texture);
sun_mesh->bind_shader(shader);
sun_mesh->setup_mesh();
SceneNode* sun = new SceneNode(sun_mesh);
float sun_rotation_speed = 10.0f;
// Earth
Mesh* earth_mesh = new Sphere(glm::vec3(0.0f,0.0f,0.0f),0.5f
    ,50,50);
earth_mesh->add_texture(earth_texture);
earth_mesh->bind_shader(shader);
earth_mesh->setup_mesh();
SceneNode* earth = new SceneNode(earth_mesh);
float earth_rotation_speed = 20.0f;
float earth_to_sun_distance = 3.0f;
earth->transform.translation = glm::vec3(
    earth_to_sun_distance,0.0f,0.0f);
earth->transform.rotation.x = 23.44f;
sun->add_child(earth);
// Moon
Mesh* moon_mesh = new Sphere(glm::vec3(0.0f,0.0f,0.0f),0.25f
    ,50,50);
moon_mesh->add_texture(moon_texture);
moon_mesh->bind_shader(shader);
moon_mesh->setup_mesh();
SceneNode* moon = new SceneNode(moon_mesh);
float moon_rotation_speed = 50.0f;
float moon_to_earth_distance = 1.0f;
moon->transform.translation = glm::vec3(
    moon_to_earth_distance,0.0f,0.0f);
moon->transform.rotation.x = 6.68f;
earth->add_child(moon);
```

Figure 6: Implémentation de get_world_transform()

Le Soleil est parent de la Terre, qui elle est parent de la Lune.

Dans la boucle de rendu, on peut transformer les planètes comme on le veut.

```
// Render loop
while (!glfwWindowShouldClose(window.get_window())) {
    ...
    // Solar system
    float sun_angle = sun_rotation_speed * delta_time;
    sun->transform.rotation.y += sun_rotation_speed *
   delta_time;
    earth->transform.rotation.y += earth_rotation_speed *
   delta_time;
    earth->transform.translation.x = earth_to_sun_distance;
    moon->transform.translation.x = moon_to_earth_distance;
    moon->transform.rotation.y += moon_rotation_speed *
   delta_time;
    sun->draw();
    ...
}
```

Figure 7: Boucle de rendu pour les planètes

Au niveau du rendu final, voici ce que l'on peut observer.

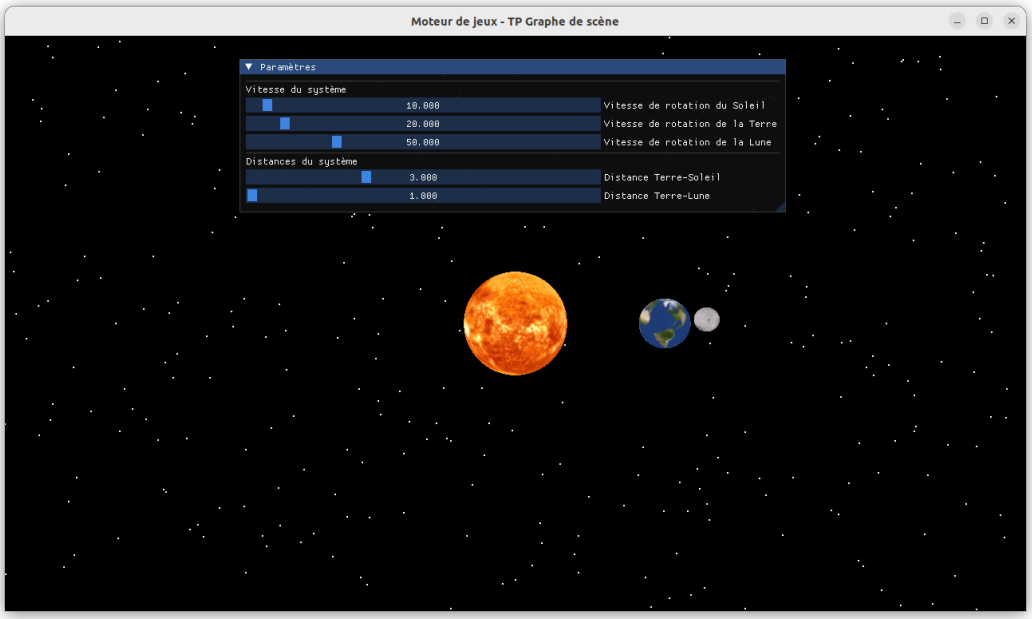

Figure 8: Rendu final du système solaire

N.B : Je me suis amusé à rajouter quelques étoiles, des textures et ImGui, mais je n'ai pas détaillé tout cela ici car c'était facultatif.

5 Question bonus

5.1 Afficher une scène infinie tout en gardant un rendu temps réel

Pour afficher une scène infinie en gardant un rendu temps réel, il faut grandement optimiser la quantité d'objets à afficher à l'écran. Pour ce faire, on peut implémenter un **octree**, qui sert à partitionner l'espace, permettant de charger/décharger les objets en mémoire en ne sélectionnant que ceux qui sont potentiellement visibles. Il est aussi possible de mettre en place du **LOD (Levels Of Detail)**, qui charge des résolutions différentes des maillages en fonction de leur distance à la caméra. On peut aussi noter l'utilisation d'**imposteurs**, qui sont des images (donc très peu gourmand au niveau des ressources) que l'on peut positionner au loin de manière à donner l'illusion qu'on est face à un objet 3D. Une chose très importante est l'implémentation du **frustum culling**, qui permet d'éviter de rendre des objets qui sont en dehors du champ de vision de la caméra. On peut aussi utiliser le **compute shader**, qui permet de massivement paralléliser des calculs gourmands (moteur physique, éclairage complexe, ...). Enfin, un peu à la manière des jeux vidéos, il est possible de mettre en place **plusieurs niveaux de qualité** avec un framerate cible, à choisir dans les paramètres, pour toujours garder une fluidité correcte.

6 Conclusion

J'ai pris beaucoup de plaisir à réaliser ce TP, notamment à ré-écrire presque entièrement tout le code nécessaire. Cela m'a permis d'être plus à l'aise avec GLFW, glad, glm et ImGui. Le rendu est très sympathique, et j'ai hâte d'itérer sur ce code pour créer mon moteur de jeux.

Je tiens à m'excuser pour tout retard dans la soumission de ce travail.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.