TP3 Moteur

Mateusz Birembaut

March 2025

Code du TP 3 (Release v3.0.0) car plus de 50 Mo

Contents

1 Interface ImGui docker et ImGuiFileDialog 2

2 Ressource Manager 3

3 Transform 4

4 Graphe de Scène 5

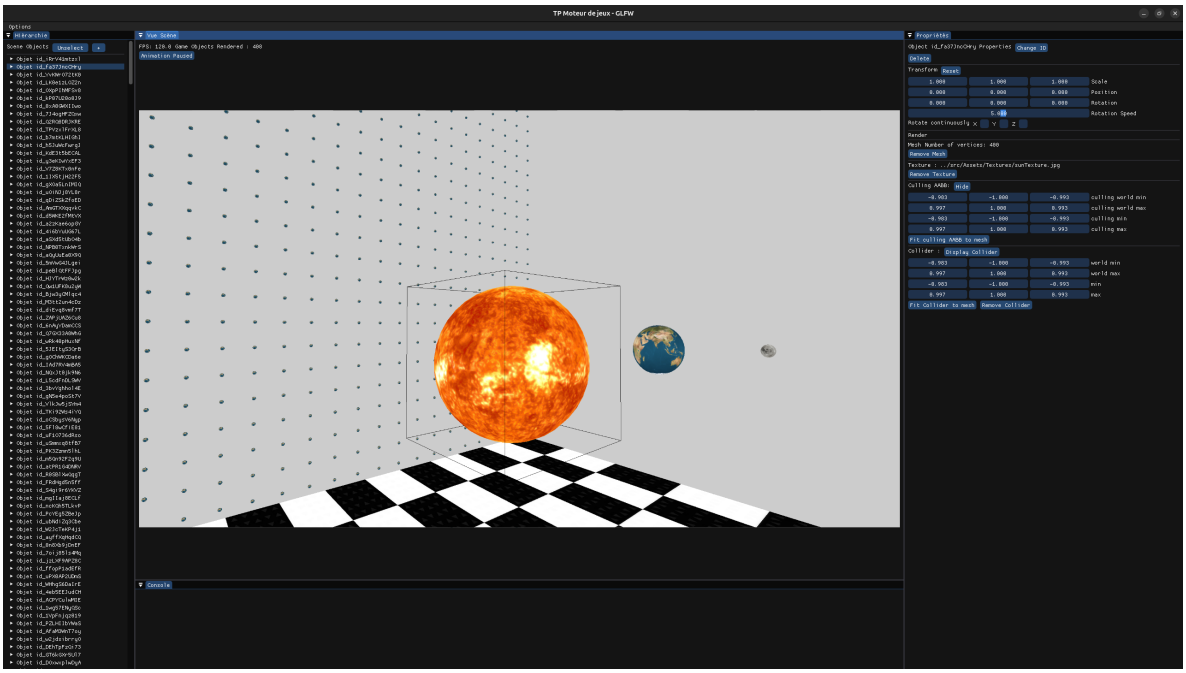
5 Frustum Culling 6

6 Partitionnement de l'espace 7

7 Autres optimisations 8

1 Interface ImGui docker et ImGuiFileDialog

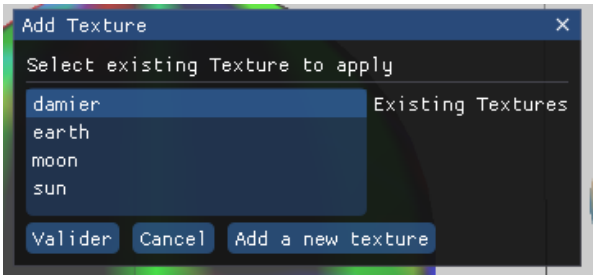
- Onglet "Hiérarchie" avec graphe de scène avec tous les games objects et la possibilité de voir leurs parents / fils.
- Onglet "Vue Scène" avec la vue de la caméra, le nombre de fps et d'objets rendus et bouton pour pauser les animations.
- Onglet "Console" avec des messages en fonction des boutons / actions.
- Onglet "Propriétés" avec toutes les propriétes d'un objet lorsqu'on en sélectionne un dans l'onglet "Hiérarchie", possibilité d'ajouter un mesh / texture / collider à un gameObject.



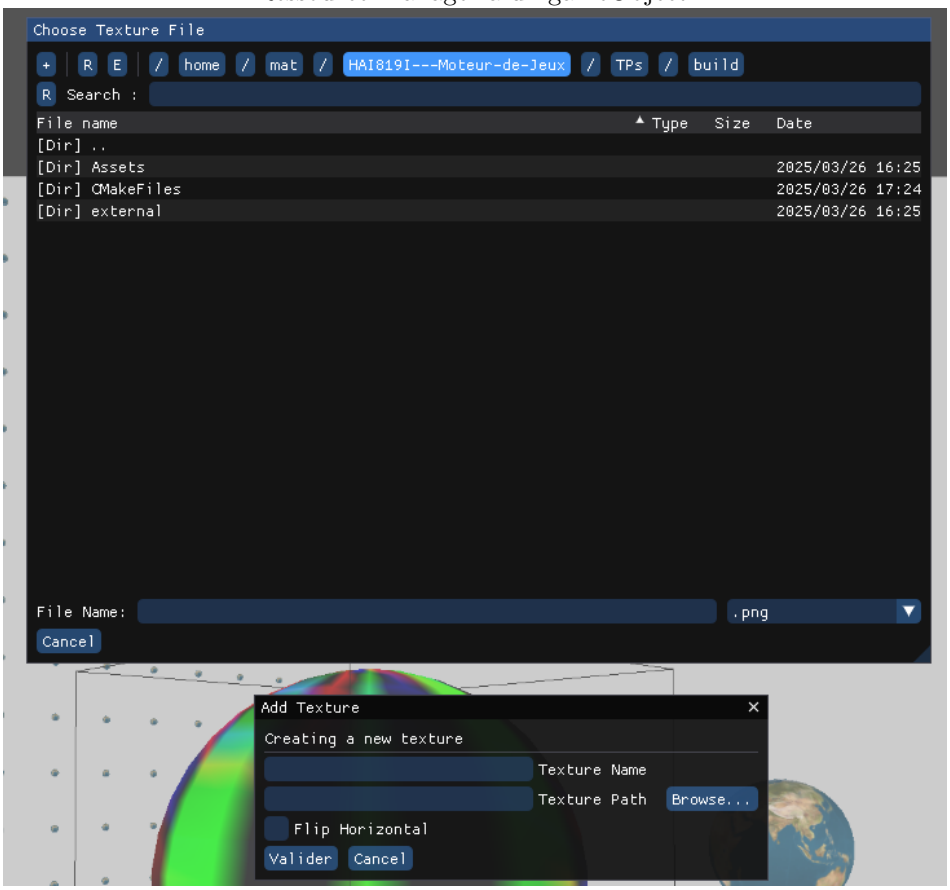
Interface

2 Ressource Manager

Création d'une classe Ressource Manager qui stocke les textures / mesh ajoutés et évite de stocker plusieurs fois le même mesh / texture. Utile pour quand on ajoute un gameObject qui par défaut est vide.



Ajout d'une texture existante stocké dans Ressource Manager à un gameObject.



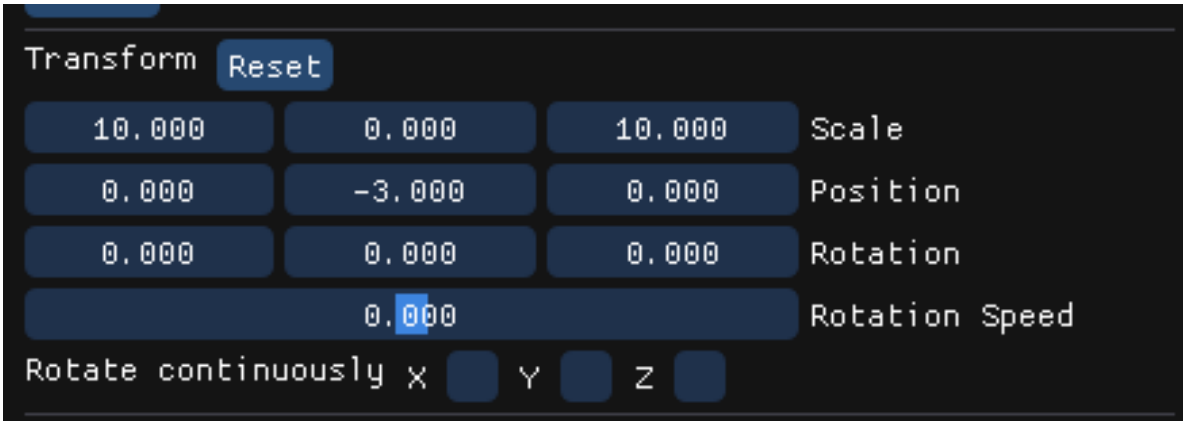
Ajout d'une nouvelle texture qui sera stockée dans Ressource Manager.

On peut faire la même chose Avec les Mesh et charger un OBJ ou PLY quand on ajoute ou crée un nouveau mesh.

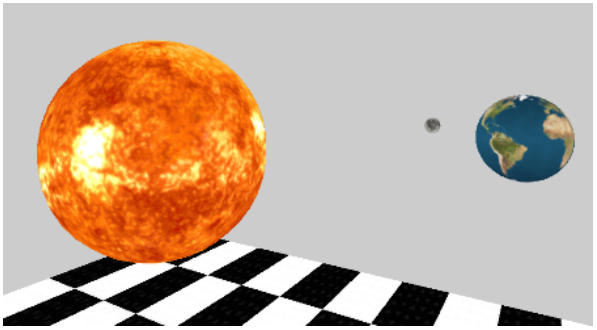
3 Transform

Tous les objets auront une classe Transform qui stockera les informations suivantes :

- Vec3 Translation : place l'objet dans le monde.
- Vec3 Scale : gère la taille.
- Vec3 eulerRot : les angles de rotations.
- Bool "isDirty" pour savoir si l'objet a bougé =¿ Permet de ne pas recalculer la matrice modèle si l'objet n'a pas bougé.
- Mat4 modelMatrix : qui stocke la matrice modèle.



Propriétés modifiables.



Terre tourne autour du Soleil et Lune tourne autour de la Terre 1

Terre tourne autour du Soleil et Lune tourne autour de la Terre 2

4 Graphe de Scène

Tous les gameObjects de la scène sont stockés dans une liste à la racine. Chaque objet peut avoir des pointeurs vers ses enfants, et chaque gameObject possède également un pointeur vers son parent (les objets à la racine ont nullptr comme parent).

Pour mettre à jour les gameObjects, la fonction updateAll du graphe de scène est appelée. Cette fonction parcourt tous les gameObjects et vérifie s'ils nécessitent une mise à jour (par exemple, s'ils sont marqués comme isDirty ou s'ils tournent continuellement autour d'un axe, comme dans l'exemple du TP3). Si une mise à jour est nécessaire, elle est effectuée.

Pour dessiner les objets, la fonction drawAll est utilisée. Elle parcourt tous les gameObjects et, si le partitionnement est activé, elle ne teste que ceux qui sont potentiellement dans notre champ de vision. Ensuite, si un objet est réellement dans notre champ de vision, il est dessiné en utilisant le frustum culling.

5 Frustum Culling

Pour avoir du space culling et ne pas recalculer la matrice vue projection, je la stocke dans ma classe caméra. Je donne une AABB à tous les objets de la scène pour savoir si l'AABB est dans la frustum de la caméra.

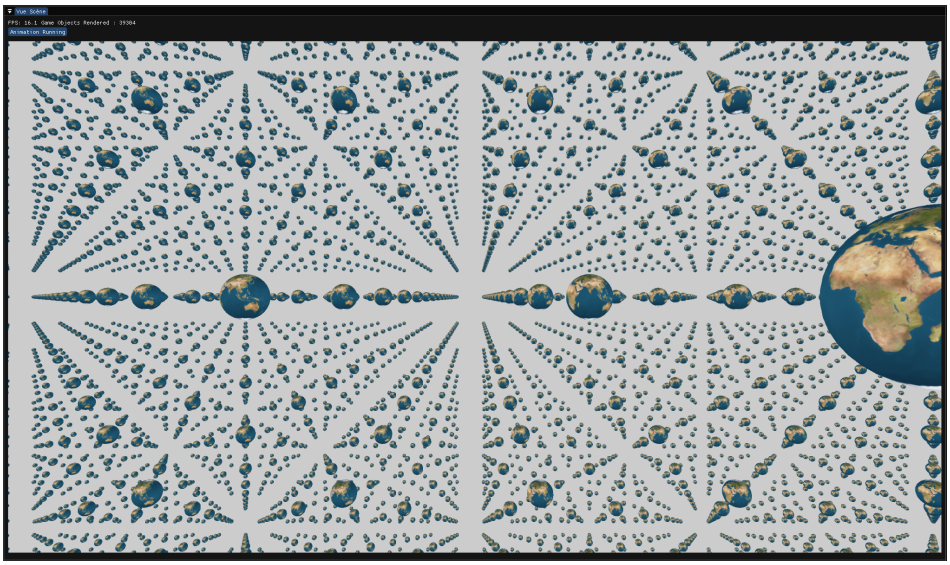
Déroulement du space culling :

- Récupération des plans formant le frustum de la caméra à partir de la matrice vue projection
- Pour chaque plan, on fait le produit scalaire entre la normale du plan et le point de l'AABB à tester*. Si la projection du point sur la normale est inférieur à 0 (produit scalaire), l'objet n'est pas dans le frustum.

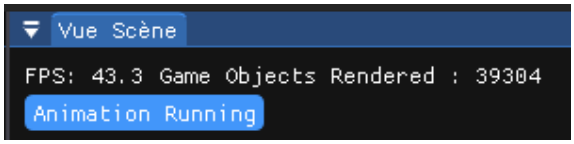
* Pour déterminer le point à tester, on récupère la normale du plan. En fonction de sa direction sur chaque axe, on récupère le point le plus adapté.

Ex : sur une normale : $(0.5, 0.5, 0.5)$, sur x la normale est $>= 0$, notre point à tester aura pour coordonnées sur x le x max de l'AABB. Puis, on répète l'opération sur les deux autres composantes.

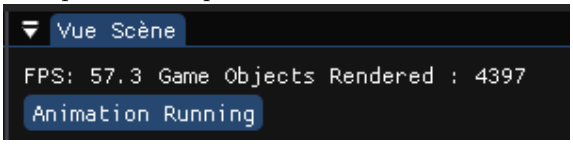
À la fin, on aura récupéré le point : $(max.x, max.y, max.z)$ ce qui est bien le coin vers lequel la normale va et donc le point le plus susceptible d'être dans le Frustum.



Scène de test avec 39 304 Sphères statiques.



Performance et nombre de "Draw" effectué sans Frustum



Performance et nommbre de "Draw" effectué avec Frustum Activé

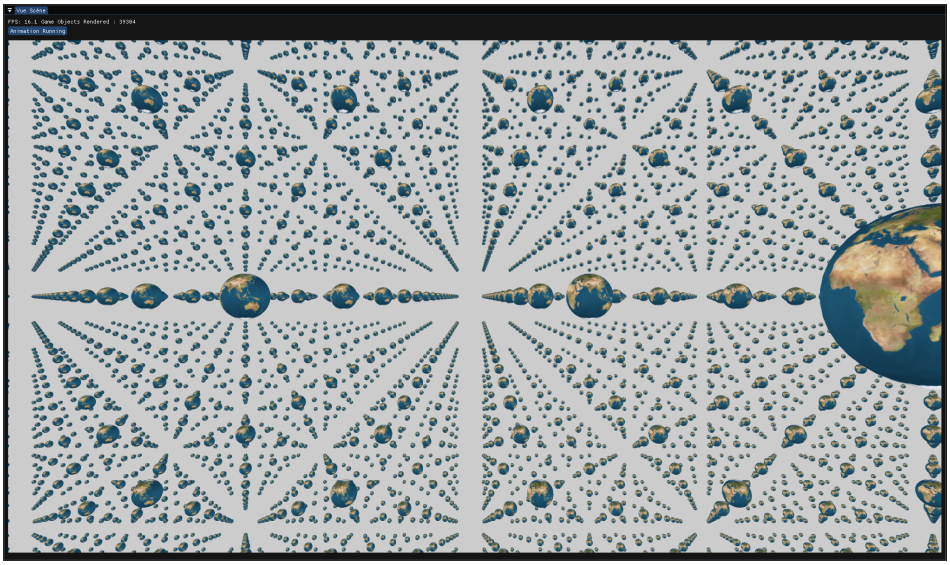
On appelle 10x fois moins la fonction draw donc on envoie beaucoup moins de données au GPU, on double les performances. Cependant, on effectue le test avec tous nos objets dans la scène. Dans la section d'après, j'ai implémenté un partitionnement de l'espace afin de réduire le nombre de tests.

6 Partitionnement de l'espace

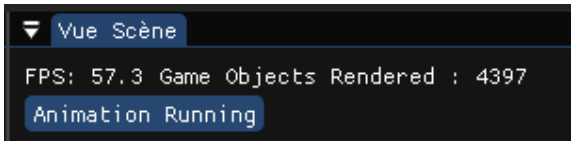
Pour partitionner l'espace et réduire le nombre de tests de frustum, j'ai implémenté un Octree. Pour l'instant l'octree est généré au lancement avec une taille et une profondeur prédéfini dans le constructeur.

Ce qui n'est pas encore géré :

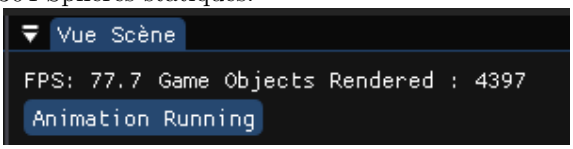
- Si dans le moteur, on déplace un objet, il ne change jamais de nœud.



Scène de test avec 39 304 Sphères statiques.



Performance et nombre de "Draw" effectué avec Frustum Activé



Performance et nombre de "Draw" effectué avec Frustum Activé et le partitionnement.

7 Autres optimisations

Pour optimiser davantage, on pourrait faire les choses suivantes :

- Implémenter l'occlusion culling avec un "Hierarchical z-buffer visibility" et l'octree que j'ai déjà implémenté.
- En fonction de la distance avec la caméra, modifier le LOD des mesh et/ou des imposteurs
- Toujours en fonction de la distance, on pourrait update les objets tous les x frames au lieu de mettre à jour toutes les frames, ce serait moins précis, mais comme les objets seraient lointains.
- Si on a le même objects présent de nombreuse fois avec seulement la transformation qui diffère, on peut implémenter l'instanciation, qui avec un appel rend tous les objets.
- Si on a le même objet, mais avec des triangles différents présent plusieurs fois (comme un terrain composé de plusieurs "parcelles" par exemple), on peut faire bu batching pour la aussi réduire le nombre de draw.