HMIN317: Moteurs de jeux

Rendu TP4 : Movements d'objets

(Détection des collisions)

Tianning MA

M2 IMAGINA

03/12/2020

Table de matière

1. Introduction

Ce compte rendu est dédié au TP4 mouvements d'objet au cours de Moteur des jeux.

Toutes les questions sont répondues.

Vous trouvez également l'adresse du git pour ce tp : https://github.com/matianning/Moteurs_de_jeux

2. Fonctionnement des touches

Touche	Fonctionnement		
P	Afficher les triangles (PolygoneMode(GL_Lines))		
UP	Mouvement de l'objet (sphère)	Z	Mouvement de caméra (monde)
Left		S	
DOWN		Q	
Right		D	

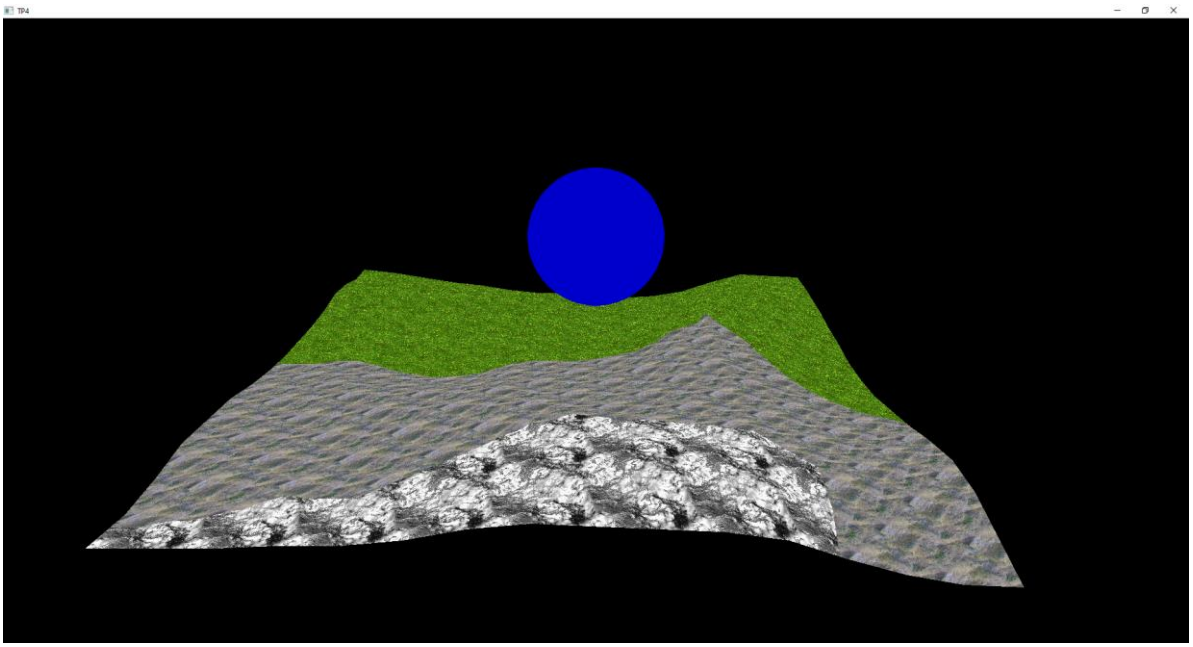
3. Rendu et explication des exercices

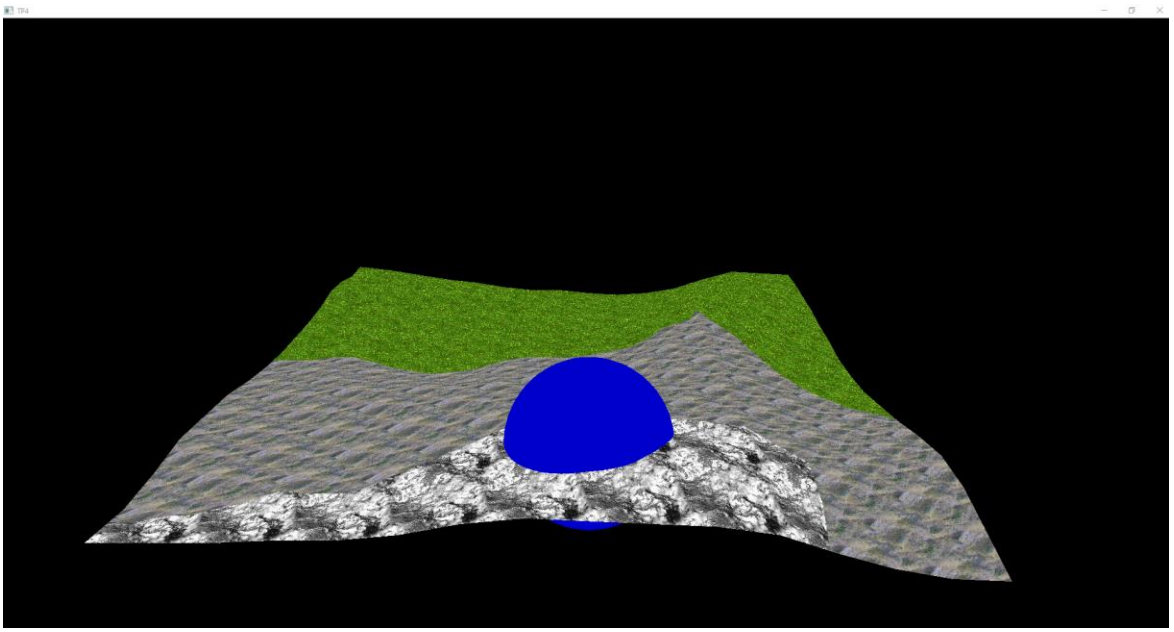
3.1 Objet non traversant

Etape 1 : chargement et déplacement d'un objet dans la scène

Pour réaliser cette partie, j'ai repris le TP1 (Affichage du terrain par heightmap) et travaillé dessus. Tout d'abord, j'ai crée des classes pour avoir un graph de scène afin de appliquer les transformations sur les objets plus facilement.

Comme montrée par l'image ci-dessous, on a un terrain texturé et une sphère bleue. Le terrain est en 16 * 16 et qui est généré par une carte de hauteur, et la sphère via le chargement du modèle OBJ et qui peuvent être déplacé par les touches de direction sur clavier.

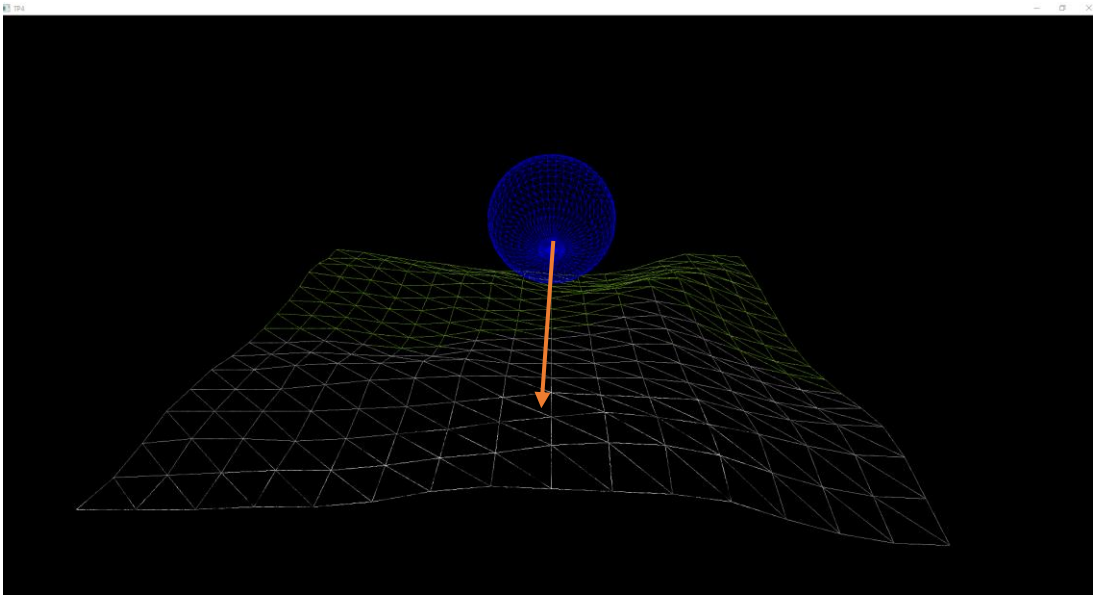




Etape 2 : Déterminer le triangle où se trouve la sphère

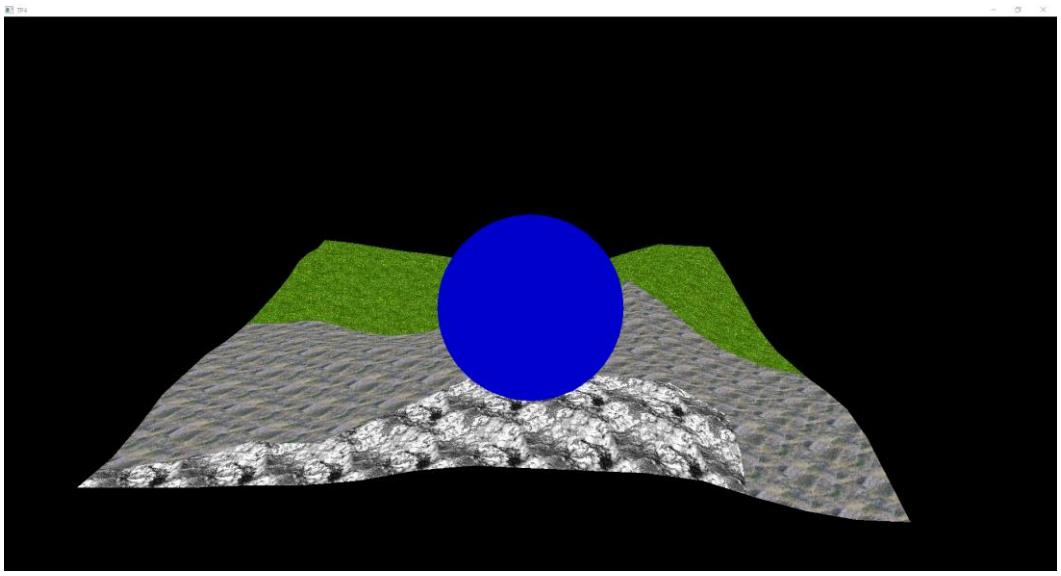
Pour trouver le triangle où se trouve la collision avec l'objet :

Tout d'abord, j'ai calculé le centre barycentre de l'objet selon ses positions de vertex. Puis, j'ai fait projeté ce point sur le plan. Ensuite, en parcourant tous les triangles du plan, je pourrais déterminer à quel triangle (selon la distance minimum entre les points de triangles et le point projeté) où se trouve l'objet actuellement, puis trouver la hauteur exacte du terrain.



Etape 3 : Adjuster la hauteur de sphère avec un offset

Pour fixer la hauteur de sphère afin que la sphère puisse se déplacer au dessus de terrain, j'ai ajouté une valeur « offset_Sphere » pour translater la sphère intégralement vers le haut (offset_Sphere $\approx$ radius de sphère)



Comme montrée l'image ci-dessus, on constate qu'après avoir ajouté la détection des collisions, la sphère se déplace au dessus du terrain avec une hauteur fixe selon où elle se trouve.

(Vous trouverez la vidéo de démonstration ou build du code pour la visualisation plus claire).

Réflexion :

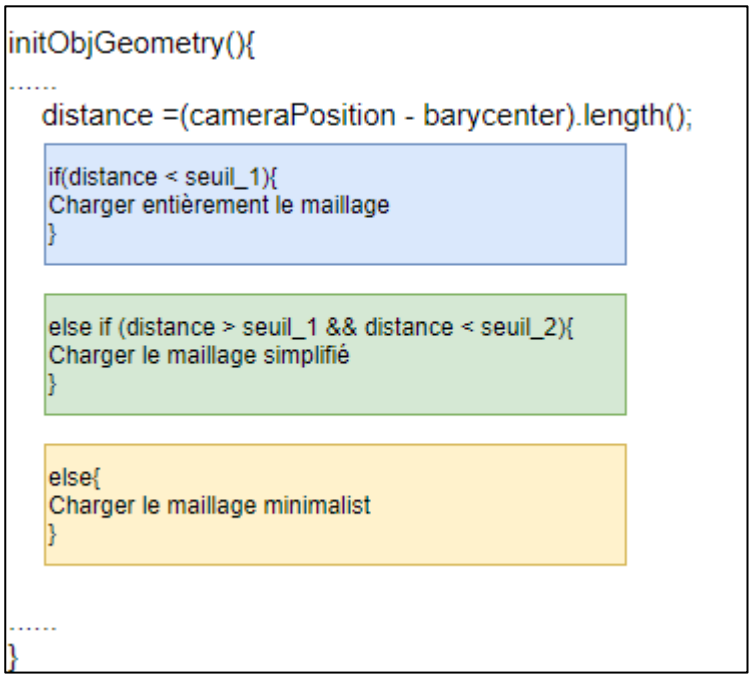
La méthode que j'ai implémenté consiste à calculer la distance minimale entre le point projeté (sur le plan principal) et les sommets des triangles. Mais cela pourrait poser des problèmes quand les hauteurs de terrain ont beaucoup de variations ou de valeur importante. De plus, le terrain est généré par STRIP qui est plus compliqué à parcourir par rapport à TRIANGE. Peut-être qu'il serait mieux de générer le terrain par TRIANGLES pour cela.

Après les cours et les tutoriels en ligne, j'ai aussi essayé d'implémenter la détection de la collision entre un bounding sphere et un plan. Mais au moment de rédaction de ce rapport, je n'ai pas encore le résultat pertinent pour cela.

3.2 Gestionnaire de niveau de détails

L'objectif de cette partie est de charger le maillage de différent résolution selon la distance entre le maillage et la position de camera. Donc, tout d'abord j'ai mis une variable Qvector3D CameraPosition dans la classe GameObject pour sauvegarder la position de camera au moment de l'initialisation des objets (maillages).

Donc, à chaque moment quand on met à jours et appeller la fonction initObjGeometry(), je peux tout d'abord calculer la distance entre le barycentre de l'objet et la position de camera, puis selon la distance, j'affficherai le maillage de différente résolution correspondant.



Au niveau d'implémentation :

```
void GameObject::init(){
    switch (type) {
    case objectType::SPHERE :
        //************Gestionnaire de détail*****************
        distance_objet_camera = distance(cameraPosition, center);
        if(distance_objet_camera < seuil1){ //Très proche
            initObjGeometry("sphere.obj");
        }
        else if (distance_objet_camera >= seuil1 && distance_objet_camera < seuil2){ //Distance
            initSimplifiedObjGeometry("sphere.obj");
        }
        else{   //loin de camera
            initMinimalistObjGeometry("sphere.obj");
        }
        break;
    case objectType::PLANE :
        initPlaneGeometry();
        break;
    default : break;
    }
}
```

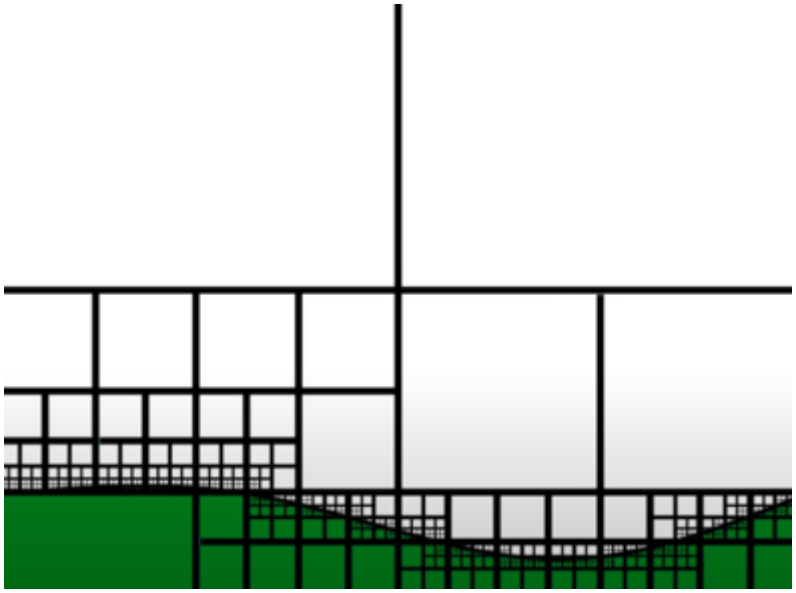
Remarque : Pour cette partie, je n'ai pas préparé le maillage simplifié, donc cette partie (la gestion de niveau de détail) n'a pas encore la visualisation.

3.3 Bonus : Quadtree

Utilisation d'un Quadtree pour la détection des collisions du terrain.

(Pour cette partie, je n'ai pas réussi à implémenter la structure, voici ma compréhension et réflexion sur cela.)

La structure d'un quadtree permet de détecter la collisions plus efficacement. A partir de la racine (la boite englobante d'ensemble de la scène), on divise les grilles une fois la cellule courante touche le terrain. On itère cette opération jusqu'à la taille de grille est suffisant (supérieur à taille d'un voxel)



Ref : https://www.youtube.com/watch?v=jxbDYxm-pXg&ab_channel=MrHeyheyhey27

Pour la structure de quadtree, après réflexion, je pense qu'il faut au moins les opération nécéssaires ci-dessous :

- Insérer un élément dans le quadtree
- Supprimer un élément dans le quadtree
- Tester si un noeud a l'intersection avec le terrain
- Update du quadtree

En comparaison avec la méthode utilisée précédement, la structure quadtree pourrait détecter la collision plus efficacement et de manière plus précise sur l'endroit exacte des collisions.