Moteurs de jeux
-
Compte rendu de TP1 - TP2

Desmarais Yann Master 1 IMAGINA

October 9, 2018

Ce compte rendu a pour but de résumer et expliquer les travaux effectués lors du TP1 et TP2 du cours de Moteurs de jeux de seconde année de Master Informatique IMAGINA de la faculté de science de Montpellier.

TP1

Question 1

La classe **MainWidget** correspond aux évènements souris pour contrôler l'objet principal et gérer l'affichage de sa texture via le vertex et le fragment shader.

La classe **GeometryEngine** correspond à la structure géométrique de l'objet que l'on veut afficher. Pour l'instant elle contient des fonctions d'affichage d'un cube (**initCubeGeometry**, **drawCubeGeometry**) mais il possible de construire d'autre type d'objets géométriques par la suite.

Les fichiers **fshader.glsl** et **vshader.glsl** correspondent aux fragment et vertex shaders. Le vertex shader gère l'éclairage et la position des sommets alors que le fragment shader gère la couleur, transparence et profondeur de chaque pixel.

Question 2

initCubeGeometry : Cette méthode va initialiser la géométrie du cube ainsi que ses coordonnées de texture. Deux tableaux sont créés :

- Un tableau de sommets correspondant aux coordonnées des triangles à tracer pour construire le cube. Ces coordonnées sont répétées pour avoir les coordonnées de texture (dans l'image cube.png).
- Un tableau d'index qui sera utilisé par les primitives d'OpenGl afin de tracer la géométrie. Ces indices correspondent au différents sommets du tableau précédent.

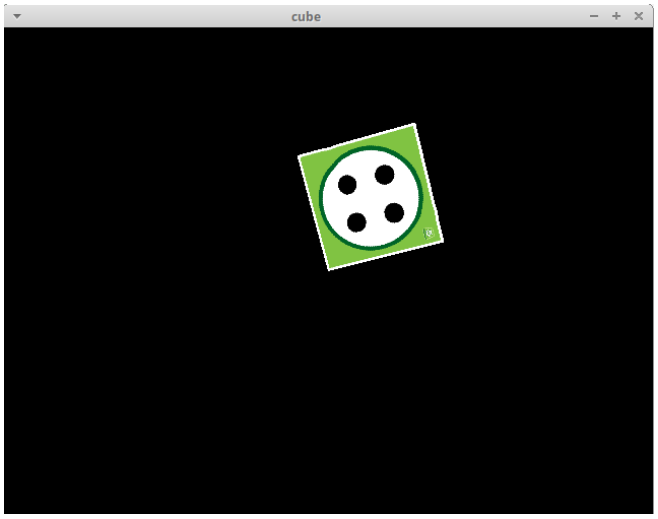
Enfin les deux tableaux sont transférés dans deux VBO créés au préalable.

drawCubeGeometry : Dans cette méthode on donne toutes les informations à OpenGL pour préparer le tracé du cube : bind des VBO, position des vertex et emplacement de la texture à utiliser. Ensuite on lance la construction de la géométrie à l'aide de la primitive **glDrawElements**.

Question 3 : Plane geometry

Nous allons créer des méthodes inspirées de **initCubeGeometry** et **drawCubeGeometry** pour tracer des plans de n x n vertices : **initPlaneGeometry(int size)** et **drawPlaneGeometry**.

De plus les coordonnées de texture dans le tableau de vertexData son modifiées pour obtenir l'une des face du dé au choix.



Question 4 : Modification d'altitude

Pour cette partie nous avons modifié en dur l'altitude du terrain pour former des pics, creux et plis.

De plus nous avons ajouté des contrôles clavier pour se déplacer autour et zoomer sur l'objet en rédéfinissant le **keyPressedEvent** dans le mainwidget.

Problèmes rencontrés & Bonus

Les problèmes principaux rencontrés lors de ce premier TP sont surtout des difficultés liées aux index de la géométrie du plan lors de son initialisation. Trouver les bons indices de manière générale et appliquer la bonne formule pour construire le tableau d'indice fut la partie la plus complexe pour moi.

En ce qui concerne les bonus je ne les ai pas implémentés mais je pense qu'il doit être possible de modifier la lumière et la teinte du terrain créé à l'aide des vertex et fragment shader.

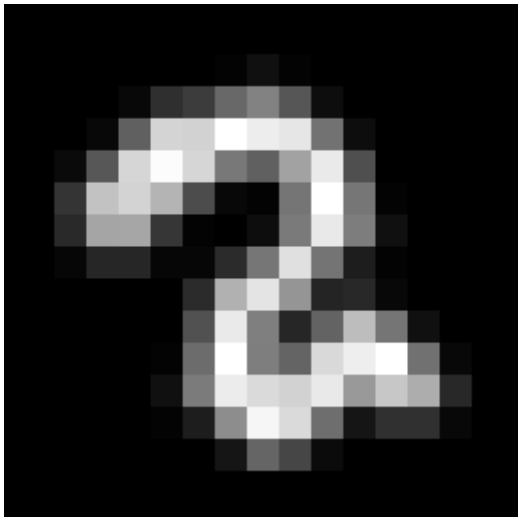
TP2

Question 1 : Height map

Par rapport à la dernière question du TP1 nous allons créer un relief sur notre plan mais cette fois-ci à l'aide d'une carte de relief qui sera lue par le programme et reproduite sur notre terrain.

Pour cela j'ai écrit 2 méthodes différentes :

- **initPlaneHeightMapTextFileGeometry** : méthode qui lis une height map contenue dans un fichier texte avec toutes les altitudes écrites directement comme une chaîne d'entiers.
- **initPlaneHeightMapImgFileGeometry** : une méthode qui va lire directement n'importe quel fichier image avec les fonctions de **QImage** de QT et la traduire en carte d'altitude en prenant la valeur de la composante rouge si l'image est en couleur le niveau de gris sinon.



(a) Premier test de height map

(b) Nombreux niveaux différents

(a) Height map plus homogène

(b) Résultat du deuxième test

Voici les résultats que l'on a obtenu avec ces différentes carte d'altitude au format png avec la méhode **initPlaneHeightMapImgFileGeometry**.

Question 2 : Orientation et Rotation

Pour gérer la rotation initiale de la camera on utilise la fonction **fromAxisAndAngle** sur l'axe X afin d'appliquer à la rotation un angle de 45 degrés :

rotation = Quaternion::fromAxisAndAngle(initialRotationVector, -45.0f) ;

Où initialRotationVector est un vecteur 3D en 1, 0, 0. Pour gérer la rotation continue c'est un peu le même principe sauf que l'on va mettre à jours la rotation dans le timerEvent pour la faire évoluer avec le temps. Dans ce cas on défini un vecteur (0.0, 1.0, 1.0) et on applique une rotation à chaque instant en fonction de la vitesse angulaire :

rotation = QQuaternion::fromAxisAndAngle(rotationVector, angularSpeed) * rotation;

Il faut bien sûr retirer la friction qui était appliquée quand la rotation était gérée par des inputs de la souris. Il faut également mettre à jour à chaque changement de la rotation et définir le centre du terrain construit dans l'initialisation de la géométrie.

Question 3 : Gestion des frames

La mise à jour du terrain est faites à chaque appel de la méthode **update** dans la redéfinition du handler **timerEvent**.

La classe **QTimer** sert à la gestion des timer dans QT. Ici il sert à update l'affichage à intervalles réguliers.

On ajoute un nombre de frame par seconde au constructeur de MainWidget et on instancie 4 versions de cette classe avec des valeurs de 1, 10, 100 et 1000 fps. On constate que les deux fenêtre avec 1000 et 100 fps ont une vitesse de rotation qui semble similaire à l'oeuil nu.

On ajoute égallement dans la redfinition du handler **keyPressedEvent** la gestion de l'augmentation des fps sur les touches contrôle et shift (UP et DOWN étant déjà utilisée pour le déplacement de caméra).