COMPTE-RENDU

GMIN317 – Moteur de Jeux

Contenu

– DØcouverte d'OpenGL avec le Framework Qt

– CrØation de surfaces maillØes

– RØcupØration d'informations dans une heightmap

SØbastien Beugnon
M2 IMAGINA 2015-2016

1. FonctionnalitØs

- DØplacement de la camØra en mode *Free Fly*, l'utilisateur se dirigera dans la direction qu'il pointe en utilisant les touches de direction Z (forward),Q(left) , S(backward), D(right) et en maintenant le clic gauche de sa souris.

- Changer entre le mode *Wireframe* (Affichage des arŒtes) et le mode *Fill* (Affichage des faces) l'aide de la touche W.

- Passage du chemin de la *heightmap* directement dans la commande : ./tp1 heightmap-1.png

- Passage des dimensions de la *heightmap* dans la reprØsentation (de grandes valeurs Ølargissent la taille d'une cellule et applati) dans la commande :

    ./tp1 heightmap-1.png 2000 2000

- Passage d'une valeur *magnitude* permettant de pondØrer les valeurs en Y de la *heightmap* en commande :

    ./tp1 heightmap-1.png 50.0f 50.0f 50 (en % pour 0.50 de magnitude)

2. DØmarche de dØveloppement

Au dØbut du dØveloppement, une recherche dans la documentation de *Qt* s'est imposØafin, partir du modŁle de dØpart, retrouvØles fonctions *OpenGL* qui ont ØtØencapsulØdans la couche du *framework Qt.*

Ensuite, j'ai dØvelopp Ø une classe que j'ai nommØ *DirtyPlane*, cette derniŁre permet de gØnØrer une grille de points *QVector3D* prenant en paramŁtres le nombre de sommets en x et en z ainsi que la largeur et la hauteur de l'espace oø les points seront contenus. Cette classe contient des accesseurs pour modifier la position (x,y,z) des points contenus dans la grille et une mØthode gØnØrant un vecteur de *GLfloat* de trois fois le nombre de points sous la forme. (x1,y1,z1,x2,y2,z2, …,…) Ce vecteur contient seulement des *vertices* formant des triangles dont les points sont enregistrØs de maniŁre ce qu'il soit lu dans le sens anti-horaire afin que les faces dites *Front* des triangles se trouvent du mŒme c t Øet ces mŒmes points sont rØpØtØs autant de fois que nØcessaire. Cette structure permet de rØaliser rapidement des appels successifs *glDrawArrays* en mode *triangles_strip* sans s'inquiØter de passer un index des faces. Cependant comme son nom l'indique cette structure n'est pas propre car elle consomme de la mØmoire pour rØpØter certains points et rend plus difficile les opØrations topologiques et l'application des couleurs.

AprŁs le dØveloppement de ceci, j'ai ajoutØla classe *HeightMapper* qui prend en paramŁtre le nom d'un fichier image qui servira de *heightmap* en vØrifiant que si celle-ci ne charge pas alors on arrŒte l'aplication. Lorsque l'image est ouverte, on gØnŁre une instance *DirtyPlane* avec comme taille le nombre de pixel en hauteur et en largeur de l'image, on modifie les valeurs en Y des points de cette instance avec la valeur rouge (ou bleu, ou vert) des pixels associØs. Il suffit alors d'utiliser la mØthode dØfinie pour dessiner les objets DirtyPlane et on a le rendu d'une *heightmap*.

Afin de pouvoir faire des manipulations intØressantes, j'ai rajoutØune valeur en pourcentage appelØe *magnitude* qui pondŁre les valeurs en Y des *heightmaps* ; ainsi que la possiblitØd'Øtendre la reprØsentation de ces derniŁres en agrandissant la taille des cellules de la surface. Puis j'ai rajoutØle contr le de ces manipulations au dØmarrage de l'application en options comme le nom du fichier servant de *heightmaps.*

Ensuite j'ai cherchØcomment reproduire les fonctions de la bibliothŁque *Glut* gr ce aux mØthodes de *Qt*, je pensais notamment la fonction :

gluLookAt(eyeX,eyeY,eyeZ,targetX,targetY,targetZ,upX,upY,upZ) permettant de dØfinir une camØra avec sa position initiale (eye), la direction de son regard (target) et la direction vers laquelle se trouve le haut de la camØra (up)

Dans la bibliothŁque *Qt*, il suffit d'appeler la mØthode *lookAt* sur une matrice ; il a alors fallu juste rØcupØrer les calculs mathØmatiques pour la conversion des entrØes claviers et souris en mouvements de camØra ou de position.

3. Structures de donnØes

Les *Vertex Arrays Objects* (VAOs) dØfinis initialement ont ØtØ rØutilisØ pour le rendu final, pour les remplir une classe a ØtØ crØØe du nom de *DirtyPlane* sous la forme :

```
class DirtyPlane {
        QVector<QVector<QVector3D> > grid;
        DirtyPlane(int width, int height, GLfloat model_width, GLfloat model_height);
        long exportation_size();
        GLfloat* export();
};
```

Cette classe ne consiste qu' produire une grille de manipulation puis exporter (ou *build*) cette grille sous la forme d'un vecteur de points pouvant Œtre rØpØtØ Cette structure possŁde des limites ; en plus, de prendre de l'espaces mØmoires, sa construction particuliŁre (l'ordre dans lesquelles les points ont ØtØ disposØs) rend plus difficile l'utilisation de couleurs et l'absence d'index ne permet pas de rØaliser aisØment une topologie. C'est pourquoi j'ai commencØ travailler sur une autre structure beaucoup plus complŁte permettant de gØnØrer les vecteurs de vertices (uniques), d'indexation des faces et des couleurs.

```
class SuperbPlane {
        QVector<QVector<QVector3D> > grid;
        QVector<QVector<QVector3D> > color_grid;
        SuperbPlane(int width, int height, GLfloat model_width, GLfloat model_height);
        long exportation_vertices_size();
        GLfloat* export_vertices();
        GLfloat* export_colors();
        long exportation_indexes_size();
        GLfloat* export_indexes();
};
```

Cette classe *SuperbPlane* est un *Builder* qui permettra alors de dØfinir la position des points de la grille mais aussi leur couleur. Il permettra aussi la production d'un indexage des sommets de chaque face prØparerant l'utilisation prochaines des *Vertex Buffer Objets*. (VBOs)

4. Bonus

- Gestion des collisions camØra/terrains

Tout d'abord il faut dØfinir une boite/une capsule ou bien une sphŁre (on choisira ce cas) reprØsentant la *Hitbox* (ou *Collider*) de la camØra. On cherche la ou les faces les plus proches de cette derniŁre gr ce aux points et le rayon de la sphŁre. (en utilisant un *KDTree*, par exemple) Puis comme une face (ou maille) est forcØment plane ce qui peut ne pas Œtre le cas du maillage. On doit alors faire le projetØdu centre de la Hitbox sur

- Texturer le terrain en utilisant des couleurs

Comme la structure que j'ai choisi sans indexation des sommets et rØpØtition de ces derniers dans le vecteur, cela a rendu l'application des couleurs compliquØes sur les points compliquØe ; C'est pourquoi j'ai commencØ travailler sur une deuxiŁme structure (SuperbPlane) gØnØrant un indexage des faces de la grille me permettant de gØrer les couleurs. point par point. J'avais essayØ d'utiliser les *struct* en *C* car il existe un moyen de stocker intelligemments les donnØes des vertices mais comme je ne me suis pas intØressØaux *VBOs*, j'ai dØcidØde ne pas les utiliser.

- Jouer de la lumiŁre

Pour travailler sur la lumiŁre, on doit activer GL_LIGHTNING et les GL_LIGHT1. Il suffit ensuite de dØfinir sa position, et ses paramŁtres AMBIENT, DIFFUSE et SPECULAR sans oublier de dØfinir avec le glMaterial le matØriaux sur chaque face (Front & Back). Pour la gestion de l'obscuritØ(ou des ombres), il faut s'intØresser toujours aux lumiŁres mais surtout aux fonctions d'attØnuation. (constants, linØaire, quadratique) Je n'ai pas rØussi faire fonctionner la lumiŁre jusqu' obtenir un rØsultat intØressant. Je crois qu'il s'agit d'un problŁme avec ma version d'OpenGL qui provient de mon driver libre Mesa.

- Afficher un brouillard

Pour que le brouillard est un rendu visible, il faut que la lumiŁre fonctionne.

- CrØer un rendu terrain infini

Ayant dØj travaillØsur un sujet similaire, de la crØation procØdurale pseudo-alØatoire, je ne connais quelques outils intØressants permettant de calculer des *heightmaps* en plein fonctionnement notamment le *Bruit de Perlin* (Perlin's Noise) qui a l'avantage de possØder des paramŁtres pour controler le niveau de dØtails des variations rØalistes en 2D, 3D et mŒme 4D. Il suffit juste de dØfinir plusieurs rayons autour de la camØra correspondant diffØrents *LOD* (Level Of Details) dans lesquels le bruit sera appelØ diffØrents niveaux d'interpolation. (linØaire, cubique, logarithme) Cette mØthode permet de crØer des variations de hauteur trŁs dØtaillØes proximitØet trŁs grossiŁres aux endroits Øloignøs. Gros avantage du Bruit de Perlin est qu'il est gradient du fait de ses interpolations donc il produit une texture rØaliste pour les terrains.