COMPTE-RENDU TP1

HMIN317 – Moteur de jeux

Résumé

Création d'un terrain triangulé à partir d'une heightmap.

David Lonni

Master 2 IMAGINA

1) Fonctionnalités

- Possibilité de passer le chemin de la *heightmap* au constructeur de la classe terrain afin qu'il génère avec les bonnes dimensions et altitude (correspondant aux données de l'image) les *vertices* du maillage,

- Possibilité de se déplacer dans la scène avec la caméra grâce à la souris et au clavier, à la manière d'un jeu de type FPS. Le curseur sera également masqué afin de ne pas gêner lors des déplacements,
  Déplacements : En avant (E), en arrière (D), à gauche (S), à droite (F), en haut (Flèche du haut), en bas (Flèche du bas).
  Clic gauche de la souris enfoncé pour regarder et se diriger autour de soi.

- Les *vertices* du terrain sont colorés en fonction de leur altitude, vert si c'est bas, marron si c'est plus élevé et blanc pour les sommets des montagnes,

- Possibilité de basculer en mode d'affichage *fil de fer* du terrain en appuyant sur la touche « W », puis à nouveau pour repasser en mode *normal*,

- Et enfin, la touche « Echap » permet à l'utilisateur de fermer l'application.

2) Démarche de développement

Après avoir réussi à lancer le projet affichant un triangle coloré, j'ai pu commencer la création d'une surface carrée triangulée.

Pour cela, j'ai créé une classe Terrain qui contiendra les fonctionnalités de la fenêtre et de la génération du terrain en 3D. Le constructeur de cette classe prend en paramètre un **QString** contenant le chemin vers la *heightmap* que l'on veut appliquer au terrain et un **QGuiApplication** qui nous servira par la suite pour fermer l'application.

Après cela, j'ai pu générer un terrain triangulé d'une taille en longueur et largeur correspondant à la taille en pixel de ma *heightmap* (ici 240x240).

J'ai changé une première fois de structure de données concernant les *vertices* du terrain. J'avais alors un tableau global de **GLFloat** contenant tous les points du maillage, et plusieurs autres tableaux crées à la volé permettant d'afficher « bande par bande » le terrain avec un **GL_LINE_STRIP** dans un « glDrawArrays » qui est une visualisation par *VertexArrays*. Puis j'ai lu les valeurs des pixels de la *heightmap* que j'ai stocké dans un tableau afin de les appliquer aux *vertices* de mon terrain selon l'axe Y, celui-ci étant généré selon l'axe XZ.

Suite à cela, j'ai ajouté les mouvements de la caméra avec les touches du clavier. Avec également avec la possibilité de regarder autour de soi et de déplacer la caméra en cliquant et déplaçant la souris comme dans un jeu de type FPS. Le curseur de la souris disparait lorsqu'on enfonce le clic gauche et réapparait lorsqu'on le relâche afin de ne pas gêner lors des déplacements.

J'ai ensuite opté pour une nouvelle structure de données, afin de passer avec une visualisation VBO. J'ai alors un tableau de *vertices*, qui est maintenant un tableau de **QVector3D**, et un tableau d'indices qui contient des **GLushort**.

Ainsi, je crée mes points de la gauche vers la droite et de haut en bas, puis je fais la liaison entre ces points afin de former des **GL_TRIANGLES** avec le tableau d'indices. J'alloue ensuite la place nécessaire pour stocker ces informations dans des buffers et j'envoie le tout dans un « glDrawElements ».

Pour finir, afin d'ajouter des couleurs à mon terrain, j'ai créé une structure « **VertexData** » contenant la position d'un *vertice* toujours avec un **QVector3D**, mais ajoutant cette fois-ci un autre **QVector3D** permettant de gérer leur couleur. Ainsi, chaque *vertice* peut avoir une couleur propre à son altitude, **vert** si < 85, **marron** si < 170 et **blanc** sinon.

3) Structure de données

Pour réaliser ce Terrain et l'afficher, j'ai utilisé une structure utilisant les *VBOs*.

J'ai un tableau de *vertices* de type **VertexData**, qui est une structure que j'ai créée, se présentant ainsi :

```
struct VertexData {
        QVector3D  position ;
        QVector3D  couleur ;
} ;
```

Chaque *vertice* aura dont une position et une couleur.

J'ai ensuite un tableau d'indices qui est de type **GLushort** qui permettra de récupérer les indices des points du tableau de *vertices* qui formeront les triangles du maillage.

Pour les afficher, j'ai créé deux buffers, *arrayBuf* et *indexBuf*, correspondant respectivement au buffer des *vertices* et au buffer des indices.

J'alloue la taille nécessaire pour ces buffers avant de les passer à l'appel de glDrawElements.

4) Parties bonus

- Qt utilisant par défaut OpenGL ES 2.0, je ne suis pas parvenu à ajouter une lumière ni de brouillard. En effet, certaines fonctions d'OpenGL de base, dont celles pour afficher une source lumineuse et du brouillard (glFog()) me donnent une erreur du type :

« Undefined reference to … »

- Afin de gérer l'affichage infini d'un terrain, j'aurai pu paramétrer le nombre de point que contenait le maillage, en choisissant de prendre 1 point sur 2 par exemple pour les terrains plus éloignés. Puis de faire en sorte que chaque dernier point d'un terrain corresponde au premier point du suivant et ainsi créer une continuité entre eux.

- Pour pouvoir gérer les collisions entre la caméra et le terrain, il faudrait tout d'abord définir un rayon autour de la caméra qui permettrait de déterminer une boite englobante autour de celle-ci. Ensuite, à chaque mouvement de la caméra, on cherche le triangle du maillage le plus proche de celle-ci, en considérant ce dernier comme un plan. Puis on projette un point partant de la position de la caméra sur ce plan. Si la distance entre le point obtenu et la position de la caméra est inférieure au rayon de la sphère englobante définie, alors il y a collision.