Fumey David

Compte Rendu TP1/TP2

TP1

Question 1

A quoi servent les classes **MainWidget** et **GeometryEngine** ?

Classe **MainWidget** :

Son rôle principal est de permettre la manipulation de notre objet courant, en repérant les event souris. Grâce à ça, on peut transposer notre objet, ainsi que lui appliquer une rotation. On peut également le redimensionner, grâce à la fonction resizeGL.

Mais ce n'est pas tout. Cette classe gère également tout ce qui concerne l'affichage.

La fonction void MainWidget::**initTextures**() va permettre de charger les textures que nous allons employer.
Ensuite, la fonction void MainWidget::***paintGL***() va lier ces textures à notre objet.
La gestion des shaders va permettre, dans un premier temps, de récupérer les coordonnées de nos sommets, puis ensuite, va, pour chaque pixel de notre forme définie précédemment, lui attribuer une couleur (tirée de la texture que nous lui avons lié).

Classe **GeometryEngine** :

Cette classe a pour but la création de notre objet, à partir de tableaux de vertex, tout en gérant le maillage de la figure.
Elle définit donc les vertex de chacune des faces, ainsi que les indices permettant le maillage de nos faces.
Elle va ensuite lier ces vertex à un buffer VBO, idem pour les indices.
Ensuite, avec OpenGL, elle va utiliser ces buffers, et va programmer la pipeline d'OpenGL, pour permettre l'utilisation de ces buffers, qui fourniront les positions des vertex, ainsi que les coordonnées des textures associées.
Finalement, l'appel à glDrawElements provoquera l'affichage de l'objet.

A quoi servent les fichiers fshader.glsl et vshader.glsl ?

Vshader a pour but de transmettre à la carte graphique les coordonnées que nous avons définis préalablement. Cela va donc récupérer toutes les coordonnées, et former des formes géométriques, à l'aide de ces sommets.

Fshader va utiliser ces formes géométriques, et, pour chacun des pixels de chacune des formes, va appliquer la couleur de la texture qui lui est associé.

Question 2

Expliquer le fonctionnement des deux méthodes void GeometryEngine::**initCubeGeometry**()

et
void GeometryEngine::**drawCubeGeometry**(QOpenGLShaderProgram *program)

void GeometryEngine::**initCubeGeometry**() :

Cette méthode définit les vertex (arêtes) de chacune des faces de notre cube. Il nous faut donc 4 vertex par faces, et nous avons 6 faces. Comme indiqué, il faut préciser les 6 faces, car leur texture n'est pas identique.
Ensuite, nous allons initialiser le buffer des vertex, de la taille voulue, et le remplir avec nos valeurs.

Nous faisons de même pour le buffer d'indices.

void GeometryEngine::**drawCubeGeometry**(QOpenGLShaderProgram *program) :

Dans cette méthode, nous allons tout d'abord indiquer au GPU quels buffers sont les buffers courants, à l'aide de bind()

Ensuite, nous définissons l'offset.
Pour le premier buffer, l'offset est égal à 0, car nous voulons affihcer toutes les valeurs de notre buffer.
Nous récupérons donc l'ID de notre buffer, à l'aide d'attributeLocation.
Ensuite, nous l'affichons, avec setAttributeBuffer, et les paramètres adéquats :
Le buffer commence à « vertexLocation », est composé de float. Nous commençons à « offset » (ici 0), chaque élément sera composé de 3 valeurs (x,y,z), et le buffer a pour taille sizeof(VertexData)

Ensuite, nous allons incrémenter notre offset. Dans notre tableau, qui contient deux attributs(QVector3D, et QVector2D), nous désirons n'afficher que la deuxième valeur (correspondant à la texture).

L'offset sera alors égal à la taille de notre premier attribut, indiquant donc au GPU de s'occuper uniquement de l'affichage du deuxième attribut.

TP2

Comment est contrôlée la mise à jour du terrain dans la classe **MainWidget** ? A quoi sert la classe QTimer ? Comment fonctionne-t-elle ?

La fonction void MainWidget::**initShaders**() s'occupe de mettre en place tous les shaders, et de les lier à notre programme.
Même principe pour la fonction void MainWidget::**initTextures**(), qui va charger les textures voulues.
La méthode void MainWidget::***initializeGL***() permet l'initialisation de notre modèle et de toutes nos fonctions OpenGL.
La fonction void MainWidget::***resizeGL***(int w, int h) permet la gestion de la Vue (caméra)

Enfin, la fonction void MainWidget::***paintGL***() va utiliser toutes les variables calculées dans les fonctions précédentes, pour s'occuper de l'affichage

La dernière ligne de cette fonction : timer.start(fps, this); permet la gestion du rafraîchissement de notre image. En d'autres termes, la variable « fps » correspond au timeout entre deux rafraîchissements, en millisecondes.

Par exemple, si fps = 100, cela veut dire que notre image sera redessinée une fois toutes les 100 millisecondes, soit 10 frames / secondes.

Modifier votre programme principal pour afficher votre terrain dans quatre fenêtres différentes, avec des fréquences de mise à jour différentes(1 FPS, 10 FPS, 100 FPS, 1000 FPS).
Qu'observez-vous?

Nous pouvons voir qu'en fonction du taux de rafraîchissement choisi, la rotation de l'objet se fait de manière plus ou moins fluide.
Plus le taux de rafraîchissement est élevé, plus la rotation paraîtra naturelle, et non saccadée.

Utiliser les flèches UP et DOWN pour modifier les vitesses de rotations de votre terrain (vitesse identique dans toutes les fenêtres).
Qu'observez-vous?

Du fait du taux de rafraîchissement, si l'on augmente suffisamment la vitesse de rotation, alors nous pourrons voir l'objet se « téléporter » d'une position à une autre. Au contraire, sur les fenêtres avec des fps élevés, l'accélération paraît naturelle et continue.

Si l'on ralentit suffisamment la rotation, alors l'objet cessera, évidemment, sa rotation, et deviendra totalement immobile.

Problèmes rencontrés

J'ai décidé de créer mon tableau d'indices en me basant sur l'exemple du cube, donc en TRIANGLE_STRIP.
Pour ça, il a donc fallu adopter une certaine logique pour dédoubler de manière correcte les indices adéquats.
Pour ça, j'ai donc décidé de ne dédoubler que les indices ne se trouvant pas au bord de l'objet, pour éviter que les triangles ne se relient, d'un bord à l'autre.

La création de l'objet et l'application de la texture se sont bien déroulés.
Mais lros de l'application de la carte des hauteurs, j'avais un soucis.

La carte n'était pas correctement prise en compte, et je pouvais voir l'apparition d'artefacts.
J'ai donc pensé que mon tableau d'indices était erroné. J'ai recommencé le tableau, en adoptant non pas le TRIANGLE_STRIP, mais juste TRIANGLES, qui me paraissait plus facile, et donc plus sûr.

J'ai appliqué cette méthode, mais le problème subsistait.
Ce n'est que plus tard, à l'aide de Mme. Faraj, que nous nous sommes rendus compte que mon image de la carte de hauteurs n'était pas prise en compte.

Après correction de ce soucis, tout marchait comme voulu.

L'affichage du terrain sous un angle de 45 degrés m'a posé quelques soucis.
J'avais modifié les valeurs :
const qreal zNear = 3.0, zFar = 7.0, fov = 45.0;
pour pouvoir mieux observer mon objet lors du premier TP.

Cela a posé un problème, car les valeurs que j'avais rentré n'étaient pas très cohérentes. Alors, lorsque j'ai utilisé matrix.lookAt pour modifier la vue, les résultats étaient inattendus.

Cela m'a pris un peu de temps pour bien comprendre comment fonctionnait la fonction, puis j'ai remis des valeurs classiques à zNear, zFar, fov, et après ceci, l'inclinaison de la vue fonctionnait bien.

Pour les vitesses de rotation, il est spécifié que la vitesse doit être la même dans chaque fenêtre. Lorsque j'applique une accélération, ou un ralentissement, à l'une des fenêtres, cela ne s'applique qu'à la fenêtre courante.
Je n'ai malheureusement pas résolu ce problème.

Pour la partie bonus sur la lumière, j'ai tenté des ajouts dans les fichiers vshaders.glsl et fshaders.glsl, mais cela n'a rien donné de concret.
A la compilation , j'avais droit à ces messages d'erreurs :

QOpenGLShaderProgram::attributeLocation(a_position): shader program is not linked

QOpenGLShaderProgram:attributeLocation(a_texcoord): shader program is not linked
QOpenGLShaderProgram::uniformLocation(ambiant_color): shader program is not linked
QOpenGLShaderProgram::uniformLocation(light_position): shader program is not linked
QOpenGLShaderProgram::uniformLocation(mvp_matrix): shader program is not linked
QOpenGLShaderProgram::uniformLocation(texture): shader program is not linked

Malgré des tentatives pour comprendre et résoudre ce problème, rien n'a été concluant.