Université de Montpellier

Compte rendu - TP1 et TP2

HMIN317 - Moteur de Jeux
Année universitaire 2018/2019

Prise en main de QtCreator et Game Loop

Auteur :
Roï Shviro

Lien GitHub :
https://github.com/Celthim/Moteur_jeux

TP1 - Prise en main de QtCreator

Question 1

La classe MainWidget a pour rôle la gestion de l'affichage et la composition de la fenêtre. On y retrouve ainsi les fonctions d'initialisation et de paramétrage de OpenGL ainsi que les appels de fonctions de dessin tels que drawCubeGeometry.

La classe GeometryEngine gère tout l'aspect de création de dessin et de géométrie dans l'espace, on y retrouve ainsi des fonctions tels que initPlaneGeometry en charge de la création des vertex et des indices dans la modélisation d'un terrain plat.

fshader (Fragment Shader) permet de fragmenter les textures de manière à les adapter au terrain.

vshader a pour rôle de gérer les positions de chaque élément de l'espace (position des vertex et de la texture).

Question 2

La fonction initCubeGeometry crée tout les vertex nécessaires au dessin du dé et leur attribue une texture par la même occasion. Elle inscrit également les sommets qui serviront au dessin des triangles ainsi que l'ordre à suivre afin que tout les triangles soient dessinés dans le même sens horaire.

La fonction drawCubeGeometry quant à elle procède au dessin du dé en suivant les instructions donnés dans la fonction d'initialisation.

Question 3

En s'inspirant des deux fonctions précédentes on écrit donc deux nouvelles méthodes en charge d'écrire un plan composé de 16*16 sommets.

Figure 1 : Plan 16*16 sommets

Question 4

On applique une randomisation de la hauteur du terrain à l'aide de l'axe Z. Ensuite on ajoute des contrôles à l'aide des events ci-dessous afin de pouvoir naviguer à l'aide des touches directionnelles et zoomer avec la molette.

```
//Dans geometryengine.cpp, initPlaneGeometry
( (float) rand()/RAND_MAX)*(Hmax-Hmin) + Hmin
```

```
//Dans mainwidget.cpp
void MainWidget::keyPressEvent(QKeyEvent *k){
switch(k->key()){
    case Qt::Key_S : projection.translate(0.0f,-1.0f,0.0f);
    break;
    case Qt::Key_Z : projection.translate(0.0f,1.0f,0.0f);
    break;
    case Qt::Key_D : projection.translate(1.0f,0.0f,0.0f);
    break;
    case Qt::Key_Q : projection.translate(-1.0f,0.0f,0.0f);
    break;
}
update();
}

void MainWidget::wheelEvent(QWheelEvent *w){
    QPoint angle = w->angleDelta()/8;
    if(!angle.isNull()){
        projection.translate(0.0f,0.0f,angle.y()*0.1);
    }
    update();
}
```

FIGURE 2 : Plan avec l'axe Z randomisé

TP2 - Game Loop et Timers

Question 1

On reprend le TP précédent et on remplace la randomisation des Z par la lecture d'une HeightMap. Pour cela il suffit de prendre une image en Grayscale que l'on lit à l'aide de QImage et de sa méthode load(). On parcours ensuite l'image concernée en prenant soin de faire correspondre la taille de la heigtmap au nombre de sommets de notre plan. Pour cela on divise la taille de la Hmap par le nombre de sommets suivit d'une multiplication par la coordonnée désirée :

```
    // Dans geometryengine.cpp, initPlaneGeometry
    // size = Nombre de sommets, fourni en paramètre de initPlaneGeometry
    // QImage Hmap = variable de GeometryEngine initialisée dans le constructeur
    for(int i=0; i<(size*size); i++){

        i_img = ((float) ( (i) % size ) ); //Conversion du tableau 1D
        j_img = ((float) ( (i) / size ) ); //En coordonnée 2D du plan

        z = ( (float) qRed(
                           Hmap.pixel( i_img * ( ( (float)Hmap.width()-1)/(size-1) ),
                           j_img*(( (float)Hmap.height()-1)/(size-1) ) )
                          )
            )/256.0;
```

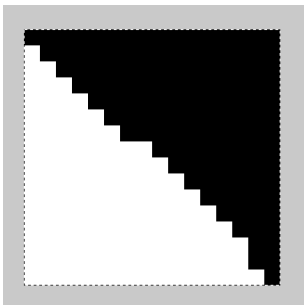

FIGURE 1.3 : Hmap très basique

FIGURE 1.4 : Hmap appliquée au plan

Bonus : J'utilise ensuite un gradient de nuances de gris que j'ai généré dans le but d'appliquer en même temps que le Z la texture correspondante.
J'ai également ajouté deux paramètres W et H à ma fonction afin de rendre paramètrable la hauteur et la largeur de la projection.

FIGURE 1.5 : Gradient de nuances de gris

```
    //Dans geometryengine.cpp, initPlaneGeometry
    // x = XOrigin = 0.0f , y = YOrigin = 0.0f
    QVector3D( x + (i_img /(size-1) )*W , y + (j_img /(size-1))*H , z*5.0 )
    QVector2D( z, 0.0f )
```

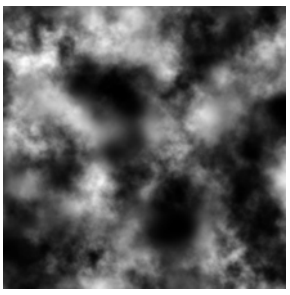

FIGURE 1.6 : Hmap plus élaborée

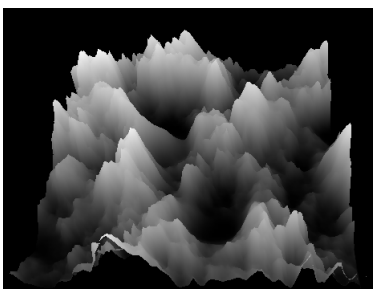

FIGURE 1.7 : Hmap appliquée au plan

Question 2

On désire maintenant faire tourner notre terrain sur lui même et le regarder sous un angle de 45°. Pour cela il faut déjà commencer par mettre le centre du terrain en 0,0. Pour ma part, ayant fixé la taille de ma projection à l'aide des variables W et H il me suffit de bouger l'origine (fixée par les variables x et y) au centre de celles-ci.

```
    //Dans geometryengine.cpp, initPlaneGeometry
    float x = -W/2.0f, y = -H/2.0f;
```

À présent il nous faut modifier l'angle de vue sur le terrain.

```
    //Dans mainwidget.cpp, resizeGL
    QVector3D v = QVector3D(1.0f,0.0f,0.0f);
    rotation = QQuaternion::fromAxisAndAngle(v,-45.0);
```

Maintenant on veut appliquer notre rotation au terrain. Il suffit d'utiliser notre timerEvent au sein de mainwidget afin d'appliquer une rotation constante selon les axes Y et Z en simultané puisque l'on a appliqué un angle de 45° à notre projection.

```
    // mainwidget.cpp, timerEvent
    // vector
    QVector3D n = QVector3D(0.0, 1.0, 1.0).normalized();

    // Calculate new rotation axis
    rotationAxis = (rotationAxis + n).normalized();

    // Update rotation
    rotation = QQuaternion::fromAxisAndAngle(rotationAxis, rotationSpeed) * rotation;

    // Request an update
    update();
```

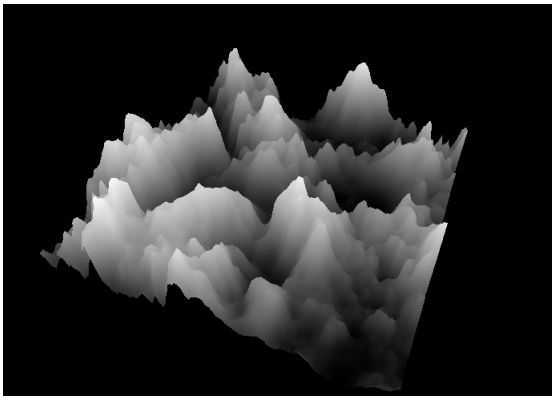

FIGURE 1.8 : Terrain en rotation 1

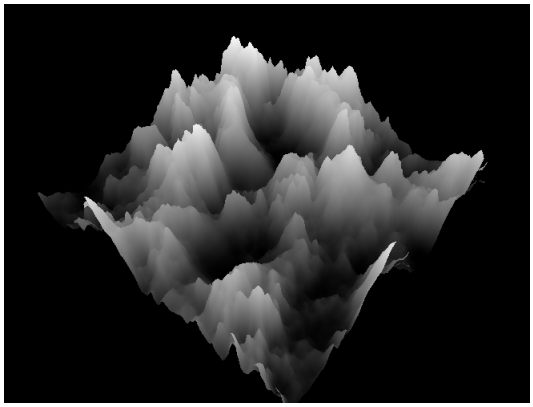

FIGURE 1.9 : Terrain en rotation 2

Question 3

La mise à jour du terrain est contrôlée dans la classe MainWidget à l'aide de l'instance de QTimer **timer**. Celle ci permet à l'aide de sa méthode start d'initialiser le timer et de fixer le délai en miliseconde d'appel du TimerEvent.

```
    //Dans mainwidget.cpp, initializeGL
    timer.start(12, this);
```

On modifie la classe MainWidget et son constructeur de manière à pouvoir fournir les FPS en paramètre. On modifie également l'appel à timer.start de sorte à convertir ces frames par secondes en délai en chaque appel de TimerEvent.

```
    //Dans mainwidget.cpp
    MainWidget::MainWidget(int FPSin, QWidget *parent) :
    QOpenGLWidget(parent),
    geometries(0),
    texture(0),
    angularSpeed(0),
    rotationSpeed(1),
    FPS(FPSin)
    {
    }
```

```
    //Dans mainwidget.cpp, initializeGL
    timer.start(1000/FPS, this);
```

On rajoute un contrôle de la vitesse de rotation à l'aide de la variable rotationSpeed avec les touches UP et DOWN dans keyPressEvent.

```
    //Dans mainwidget.cpp, keyPressEvent
        case Qt::Key_Up : rotationSpeed++;
        break;
        case Qt::Key_Down : rotationSpeed--;
        break;
```

Et à présent on utilise cela pour créer 4 instances de MainWidget avec des FPS différents pour observer la différence de FPS.

```
    //Dans main.cpp
    MainWidget widget(1);
    MainWidget widget1(10);
    MainWidget widget2(100);
    MainWidget widget3(1000);
    widget.show();
    widget1.show();
    widget2.show();
    widget3.show();
```

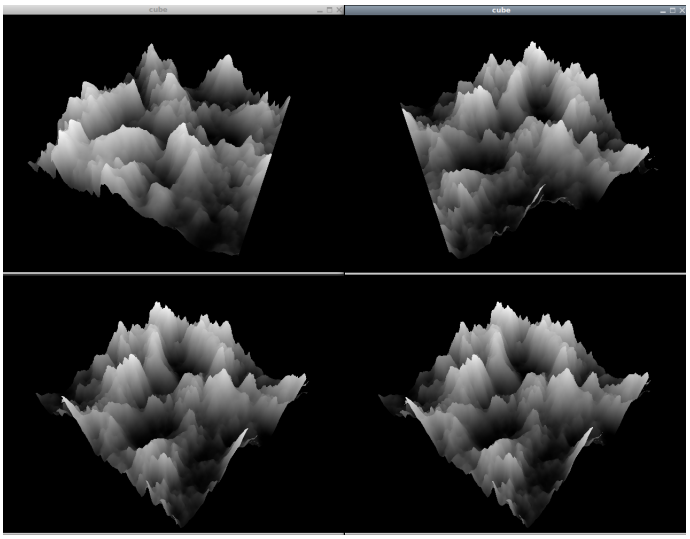

FIGURE 2.10 : Les 4 fenêtres avec différents FPS