HMIN317: Moteurs de jeux

Rendu TP1 &TP2 : Prise en main

Tianning MA

M2 IMAGINA

01/10/2020

Table de matière

1. Introduction

Ce compte rendu est dédié au TP1 et TP2 prise en main de Qt Creator, Git et OpenGL ES 3.0.

Toutes les questions sont répondues.

Pour la question bonus, j'ai réussi à réaliser une partie. Par contre, je n'ai toujours pas trouvé le moyen pour afficher un text dans la fenetre. (J'ai essayé Qpainter, mais cela ne marche pas maleheuresment). En plus, les fonctions comme renderText(), ne supporte pas cette version de qt.

2. Fonctionnement des touches

Touche	Fonctionnement		
Z	Déplacement de caméra vers z+	Q	Déplacement de caméra vers x-
S	Déplacement de caméra vers z-	D	Déplacement de caméra vers x+
C	Changement de mode de la caméra (libre / orbital)	ESPACE	Afficher les triangles (PolygoneMode(GL_Lines))
UP	Augmenter la vitesse de rotation dans le mode orbital	DOWN	Diminuer la vitesse de rotation dans le mode orbital

3. Rendu et explication des exercices

Question 1

Expliquer Le fonctionnement les méthodes de dessin et de transformation appliquées aux objets. Quelles sont les mécanismes et fonctions permettant de transmettre à l'application les mises à jour à partir des entrées utilisateur ?

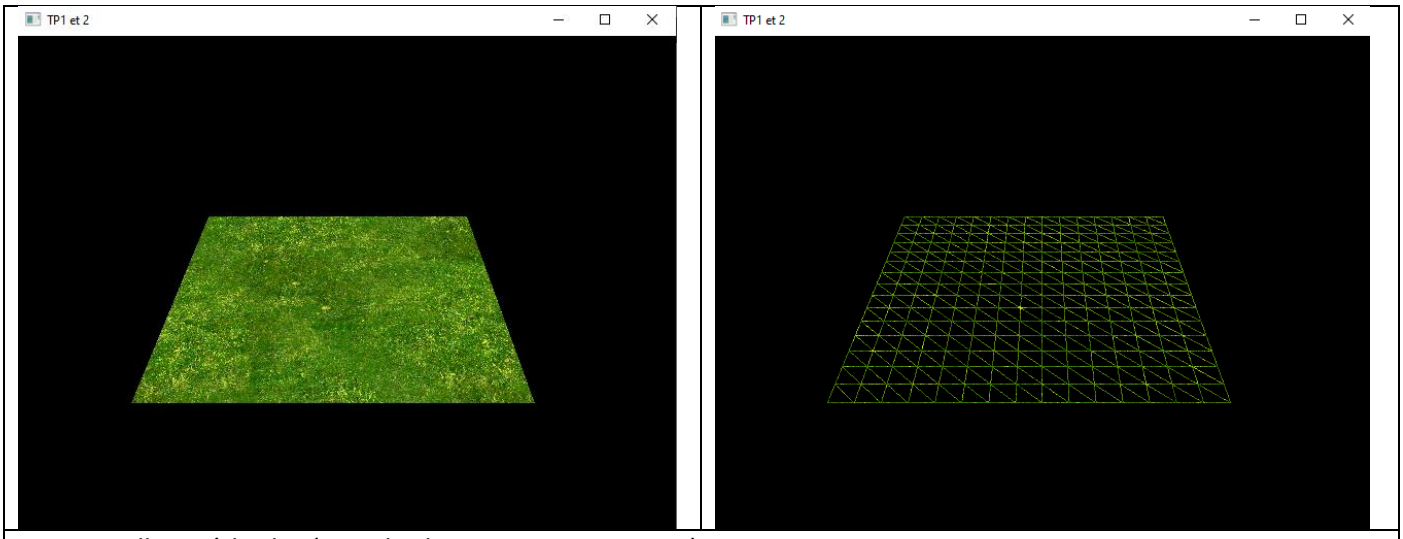
La méthode *paintGL()* dans la classe MainWidget permet de dessiner les contenus définis(de la classe geometryengine) dans la fenêtre principale. Les transformations se font par la transformation matricielle (dans la méthode paintGL) sur la matrice model en utilisant les fonctions comme *rotate*, *translate* etc.

Le mécanisme utilisé est principalement le mécanisme des signaux et des slots. C'est un principe propre à Qt pour gérer les évènements au sein d'une fenêtre. Un signal est un message envoyé par un widget lorsqu'un évènement

se produit, et un slot est la fonction qui va être appelée (ex : la méthode d'une classe). Le signal et le slot se relie grâce à la méthode statique connect().

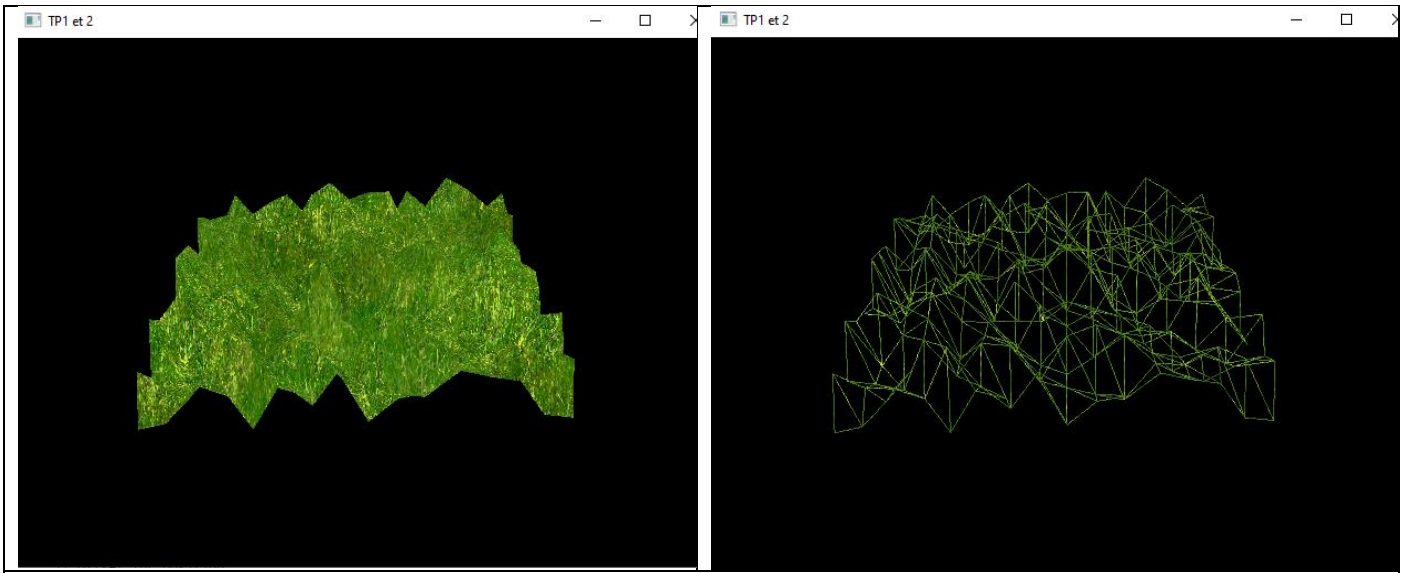
La macro Q_OBJECT est nécessaire dans le header de la classe (dans le tp, on en a dans la classe « mainwidget »), qui consiste à demander le compilateur à accepter les slots. Grâce à ce mécanisme, on pourra réaliser les intéractions (clavier ou souris) avec la fenêtre via les slots (méthodes) comme : mousePressEvent(e), keyPressEvent(e) etc.

Question 2



Les nouvelles méthodes (Dans la classe GeometryEngine):
Void drawPlane(*program);
Void initPlaneGeometry() ;
Ces deux méthodes permettent de générer la géométrie du plan et le dessiner à l'aide de shaders.
Les triangles sont dessinés dans le plan (z=0), puis j'ai modifié la caméra pour garder la surface visible. Le plan est de taille 16 * 16, et enfin, j'ai appliqué la texture « grass » dessus.

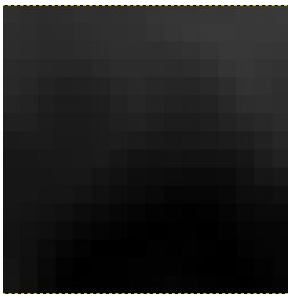
Question 4



Pour cette question, tout d'abord, j'ai modifié l'altitude (coordonnée z) pour chaque vertex par un chiffre aléatoire. Donc, on a obtenu un relief comme l'image ci-dessus. La caméra est toujours fixée pour garder la surface visible. J'ai aussi réalisé le déplacement de la caméra, vous pouvez le tester. Le déplacement se fait par les touches de direction de clavier.

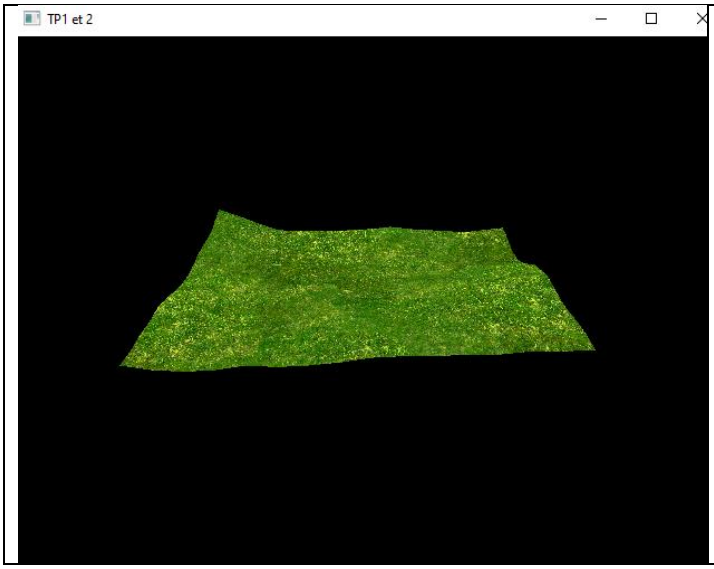
Question 5

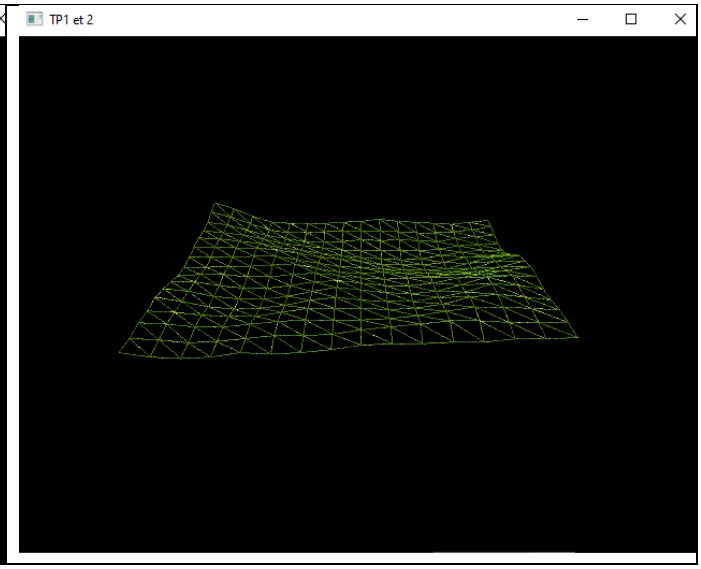
Modification d'altitude par heightmap



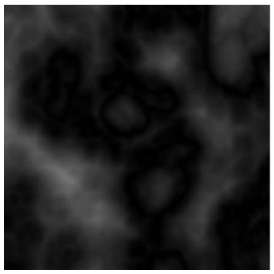
Le Heightmap que j'ai utilisé est comme l'image ci-contre :

c'est une image de taille 16 * 16, rogné de fichier « heightmap – 1024 * 1024 ». (pour convenir la taille de notre terrain 16 * 16).



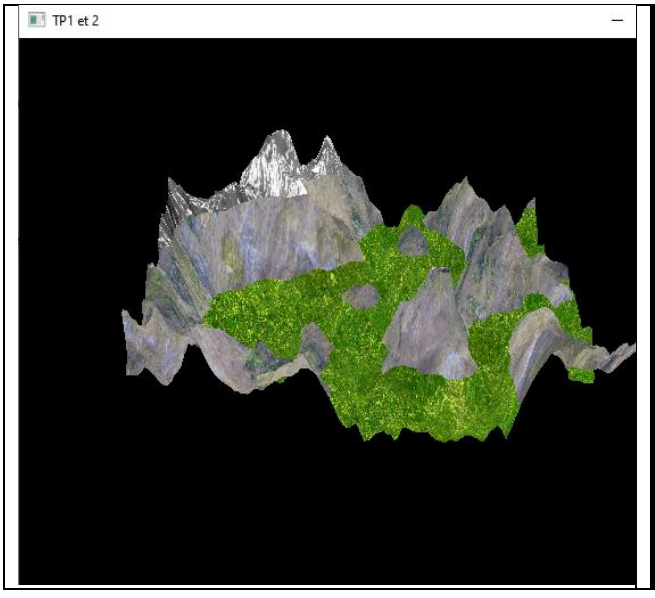


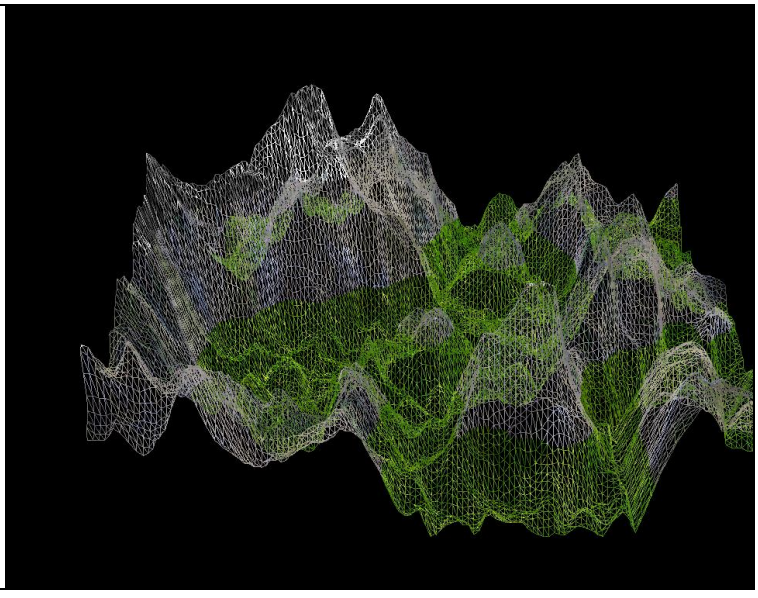
Ensuite, pour avoir un rendu plus idéal, j'ai modifié la taille de terrain en 128*128.



Heightmap de taille 128*128 utilisé comme l'image ci-contre.

En ajoutant plusieurs textures sur ce terrain, voici l'image résultante que j'ai obetenu.





Pour réaliser d'ajouter plusieurs textures, j'ai fait initialiser les textures dans « initTextures() », puis bind et setvalue dans « paintGL() »

```
void MainWidget::initTextures()
{
/*
    texture = new QOpenGLTexture(QImage(":/grass.png").mirrored());
    texture->setMinificationFilter(QOpenGLTexture::Nearest);
    texture->setMagnificationFilter(QOpenGLTexture::Linear);
    texture->setWrapMode(QOpenGLTexture::Repeat);
*/
    texture_grass = new QOpenGLTexture(QImage(":/grass.png").mirrored());
    texture_rock = new QOpenGLTexture(QImage(":/rock.png").mirrored());
    texture_snow = new QOpenGLTexture(QImage(":/snowrocks.png").mirrored());

    texture_grass->setMinificationFilter(QOpenGLTexture::Nearest);
    texture_grass->setMagnificationFilter(QOpenGLTexture::Linear);
    texture_grass->setWrapMode(QOpenGLTexture::Repeat);

    texture_rock->setMinificationFilter(QOpenGLTexture::Nearest);
    texture_rock->setMagnificationFilter(QOpenGLTexture::Linear);
    texture_rock->setWrapMode(QOpenGLTexture::Repeat);

    texture_snow->setMinificationFilter(QOpenGLTexture::Nearest);
    texture_snow->setMagnificationFilter(QOpenGLTexture::Linear);
    texture_snow->setWrapMode(QOpenGLTexture::Repeat);
}
```

paintGL() :

```
texture_grass->bind(1);
texture_rock->bind(2);
texture_snow->bind(3);

program.setUniformValue("texture_grass", 1);
program.setUniformValue("texture_rock", 2);
program.setUniformValue("texture_snow", 3);
```

Avec vertex shader, on passe la position de vertex dans le fragment shader.

et dans *<u>le fragment shader</u>* :

```
uniform sampler2D texture_grass;
uniform sampler2D texture_rock;
uniform sampler2D texture_snow;

in vec2 v_texcoord;
in vec3 v_position;
//! [0]
void main()
{
    // Set fragment color from texture
    //gl_FragColor = texture2D(texture, v_texcoord);

    if(v_position.z <=0.1f)
        gl_FragColor = texture2D(texture_grass, v_texcoord);
    else if (v_position[2]>0.1f && v_position[2]<=0.3f)
        gl_FragColor = texture2D(texture_rock, v_texcoord);
    else
        gl_FragColor = texture2D(texture_snow, v_texcoord);
```

selon l'altitude, on modifie les textures qu'on veut appliquer sur cette position.

Question 6

Proposer un autre mode d'affichage pour regarder le trrain sous un angle de 45 degrés et le faire tourner autour de son origine avec une vitesse constante à l'aide d'un timer.

*<u>Dans MainWidget ::keyPressEvent(*e) :</u>*

```
case Qt::Key::Key_C :
    if(mode_libre) mode_libre = false;
    else mode_libre = true;
    break;
```

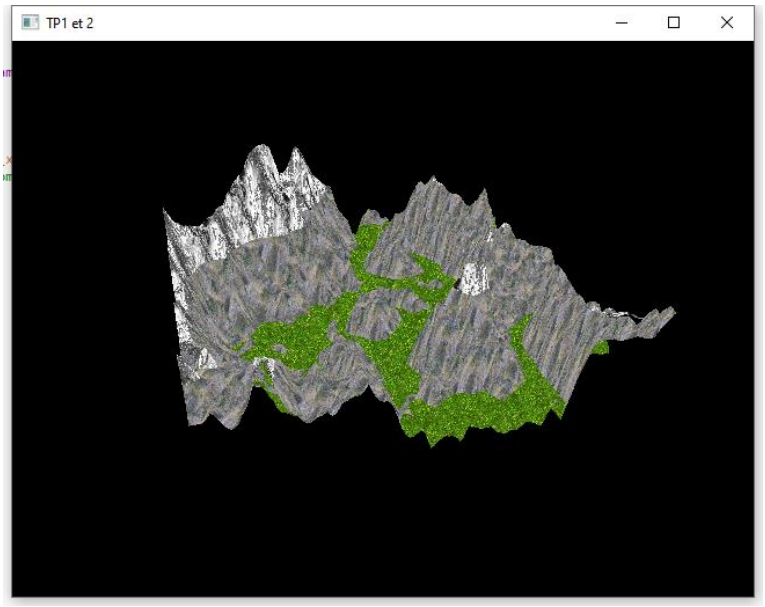
Grâce à la variable « mode_libre », la touche C permet de alterner les 2 modes de camera (libre et orbital)

La variable « ratio » dans la classe « GeometryEngine » permet d'augmenter / diminuer l'échelle du terrain (modification d'échelle du terrain dans l'affichage)

Pour réaliser le fonctionnement de la caméra, le code est surtout la fonction « paintGL » de la classe « mainwidget »

```
// Calculate model view transformation
QMatrix4x4 matrixMVP;
QMatrix4x4 view, model;

model.setToIdentity();
if(this->mode_libre){
    model.translate(mouvement_x,mouvement_y,mouvement_z);   //movement de la caméra (mode libre)
}

if(!this->mode_libre){
    model.rotate(mouvement_rotation,QVector3D(0.0f,1.0f,0.0f)); //rotation automatique dans le mode orbital
}

model.rotate(-90,QVector3D(1.0,0.0,0.0));   //pour dresser le plan
model.translate(-0.5 * this->geometries->ratio,-0.5* this->geometries->ratio,0.0);            //mettre le plan au milieu de la scène

if(!this->mode_libre){
    view.lookAt(QVector3D(0,1.0 * this->geometries->ratio,1.5* this->geometries->ratio), QVector3D(0,0,0.0), QVector3D(0.0,1.0,0.0));
}
else{
    view.lookAt(QVector3D(0,1.0 * this->geometries->ratio/5 ,1.5 * this->geometries->ratio/5), QVector3D(0,0,0.0), QVector3D(0.0,1.0,0.0));
}

matrixMVP = this->projection * view * model;

program.setUniformValue("mvp_matrix", matrixMVP);
```

Mode orbital (par défaut):



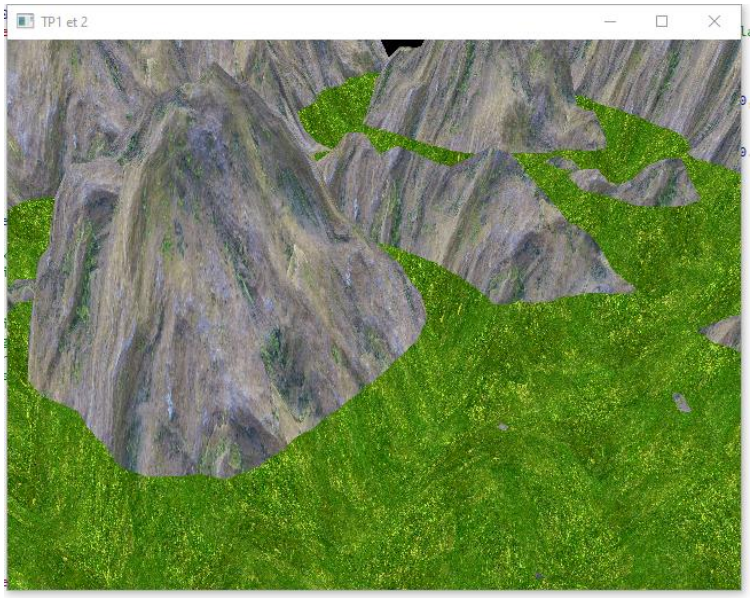
Création de timer dans le constructeur :

```
MainWidget::MainWidget(QWidget *parent) :
    QOpenGLWidget(parent),
    geometries(0),
    texture(0),
    angularSpeed(0)
{
        QTimer *timer = new QTimer(this);
        connect(timer, SIGNAL(timeout()), this, SLOT(updateAnimation()));
        timer->start(100);
}
```

Asscocier au slot updateAnimation

```
void MainWidget::updateAnimation()
{
    //mouvement_rotation += 2.0f;
    timer_rotation+=1.0f;
    this->update();
}
```

Mode libre : (déplacement possible)



On modifie les différentes transformations pour la caméra selon les modes différents.

Question 7

Dans la fonction « paintGL » :

```
frameCount++;

QTime new_time = QTime::currentTime();
if (last_time.msecsTo(new_time) >= 1000)
{
// sauvegarder le FPS dans last_count et on réinitialise
    last_count = frameCount;
    frameCount = 0;
    last_time = QTime::currentTime();
}
```

On utilise Qtime pour mesurer le temps, et à chaque seconde, on dénombre « frameCount » puis sauvegarder ce FPS dans la variable « last_count » pour l'affichage.

```
float vitesse_rotation = 1.0f;
```

dans la classe « mainwidget », la variable vitesse_rotation consiste à modifier la vitesse de rotation dans le mode orbital. (avec la touche UP et DOWN)

```
void MainWidget::updateAnimation()
{
    timer_rotation+=vitesse_rotation;
    this->update();
}
```