Moteurs de jeux - Compte Rendu TP1/TP2

Nicolas CELLIER

https://github.com/NicoCellier/Moteurs-de-jeux

TP1

Question 1 :

La classe MainWidget sert à gérer l'affichage de la fenêtre (shaders, textures, dimensions) ainsi que tous les evenements souris qui permettent la rotation.

La classe GeometryEngine sert à gérer le maillage du cube (initialisation et affichage).

Le fichier fshader.glsl sert à gérer les shaders des fragments.

Le fichier vshader.glsl sert à gérer les shaders des vertices.

Question 2 :

void GeometryEngine::initCubeGeometry()

Initialise le cube avec le tableau vertices, qui contient les coordonnées des sommets dans l'espace et les coordonnées des textures dans l'image 2D cube.png, ainsi que le tableau indices, qui contient les indices des sommets de chaque triangle strip du maillage. Ces données sont ensuite stockées dans des VBO (Vertex Buffer Objects).

void GeometryEngine::drawCubeGeometry(QOpenGLShaderProgram *program)

Permet de récupérer les VBO initialisés précedemment dans initCubeGeometry, puis indique à la pipeline OpenGL comment récupérer les données des vertexs ainsi que des textures. Enfin, glDrawElements est appelé pour dessiner le cube.

Question 3 :

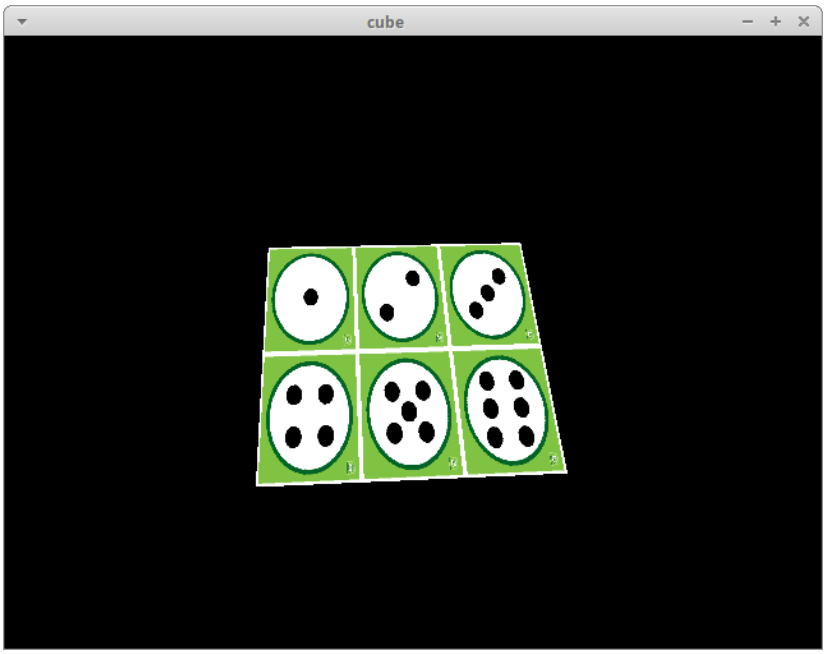
initPlaneGeometry() :

```
void GeometryEngine::initPlaneGeometry() {

    int nbV = 16 ; // nb VertexByRow or Column

    // Init vertices
    VertexData *vertices = new VertexData[nbV*nbV] ;
    float x,y,z, a ;
    x = -1.0f ;
    y = -1.0f ;
    z = 0.0f ;
    a = 2.0f/nbV ;
    for (int i = 0; i < nbV; ++i) {
        x = -1.0f + i * a ;
        z = i%2*(1.0f/10) ;
        for (int j = 0; j < nbV; ++j) {
            y = -1.0f + j * a ;
            vertices[i*nbV+j] = {QVector3D(x, y, z), QVector2D((1.0f/(nbV-1))*i, (1.0f/(nbV-1))*j)} ;
        }
    }

    // init indices
    int nbIndices = (nbV*2+1)*2 +(nbV*2+2)*(nbV-3) ;
    GLushort *indices = new GLushort[nbIndices];
    int cpt = 0 ;
    int i, j ;
    i = 0 ;
    j = 0 ;
    // do a triangle strip for each line
    for (i = 0; i < 15; ++i) {
        if (i!=0) {
            indices[cpt] = i*16 + j ;
            cpt ++ ;
        }
        for (j = 0; j < 16; ++j) {
            indices[cpt] = i*16 + j ;
            cpt ++ ;
            indices[cpt] = (i+1)*16 + j;
            cpt++;
        }
        if (i!=15) {
            indices[cpt] = (i+1)*16 + j;
            cpt++;
        }
    }

    // Transfer vertex data to VBO 0
    arrayBuf.bind();
    arrayBuf.allocate(vertices, nbV*nbV * sizeof(VertexData));

    // Transfer index data to VBO 1
    indexBuf.bind();
    indexBuf.allocate(indices, nbIndices * sizeof(GLushort));

}
```

drawPlaneGeometry() :

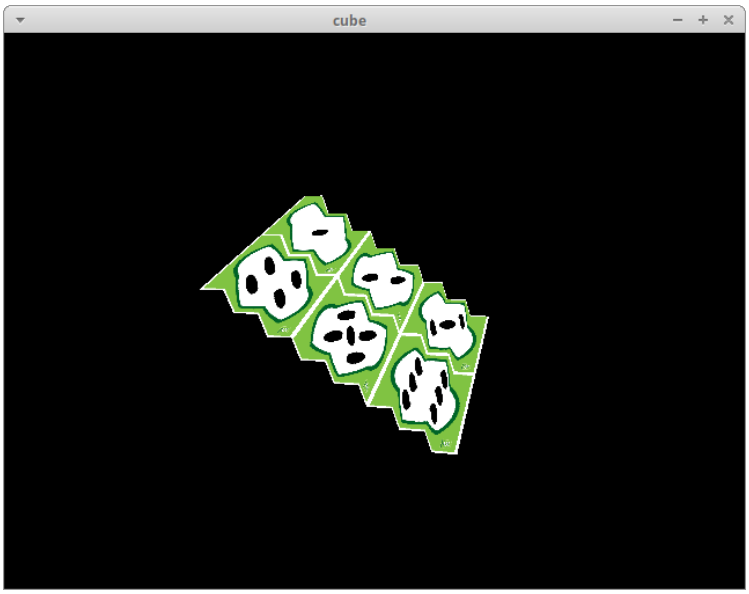
```
237 void GeometryEngine::drawPlaneGeometry(QOpenGLShaderProgram *program)
238 {
239     // Tell OpenGL which VBOs to use
240     arrayBuf.bind();
241     indexBuf.bind();
242
243     // Offset for position
244     quintptr offset = 0;
245
246     // Tell OpenGL programmable pipeline how to locate vertex position data
247     int vertexLocation = program->attributeLocation("a_position");
248     program->enableAttributeArray(vertexLocation);
249     program->setAttributeBuffer(vertexLocation, GL_FLOAT, offset, 3, sizeof(VertexData));
250
251     // Offset for texture coordinate
252     offset += sizeof(QVector3D);
253
254     // Tell OpenGL programmable pipeline how to locate vertex texture coordinate data
255     int texcoordLocation = program->attributeLocation("a_texcoord");
256     program->enableAttributeArray(texcoordLocation);
257     program->setAttributeBuffer(texcoordLocation, GL_FLOAT, offset, 2, sizeof(VertexData));
258
259     // Draw cube geometry using indices from VBO 1
260     glDrawElements(GL_TRIANGLE_STRIP, 33*2+34*13, GL_UNSIGNED_SHORT, 0);
261 }
262
```

Resultat :



Question 4 :

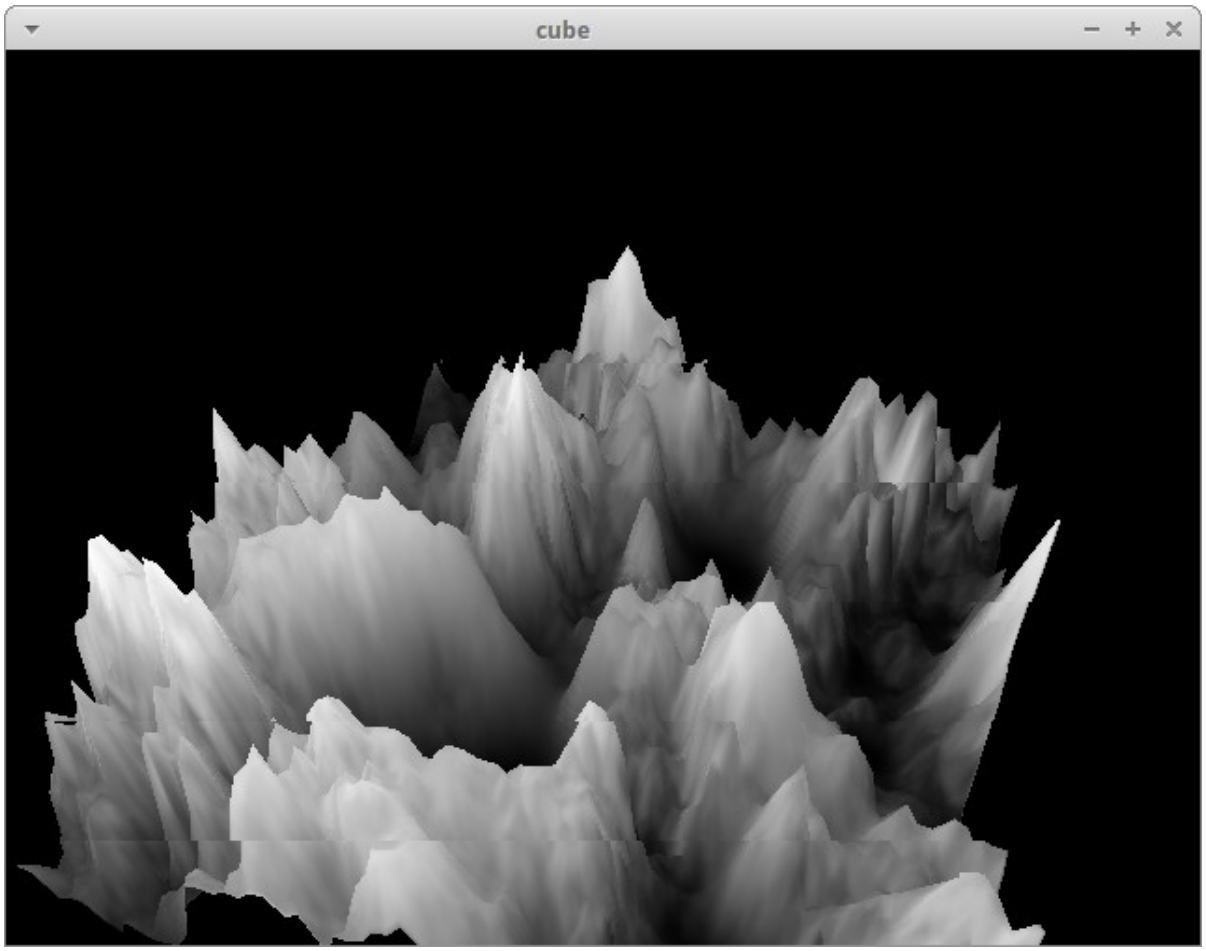
```
103 void MainWidget::keyPressEvent(QKeyEvent *e)
104 {
105     float camSpeed = 0.1f ;
106     switch(e->key()){
107         case(Qt::Key_Left) :
108             cam.setX(cam.x()+camSpeed);
109             break ;
110         case(Qt::Key_Right) :
111             cam.setX(cam.x()-camSpeed);
112             break ;
113         case(Qt::Key_Up) :
114             cam.setY(cam.y()+camSpeed);
115             break ;
116         case(Qt::Key_Down) :
117             cam.setY(cam.y()-camSpeed);
118             break ;
119         case(Qt::Key_Plus) :
120             cam.setZ(cam.z()+camSpeed);
121             break ;
122         case(Qt::Key_Minus) :
123             cam.setZ(cam.z()-camSpeed);
124             break ;
125     }
126     update();
127 }
128
129 void MainWidget::wheelEvent(QWheelEvent *e)
130 {
131     float camSpeed = 0.1f ;
132     if (e->delta()>0) {
133         cam.setZ(cam.z()+camSpeed);
134     } else {
135         cam.setZ(cam.z()-camSpeed);
136     }
137     update();
138 }
```



TP2

Question 1 :

```
QImage img = QImage("../TP1/heightmap-1.png");
int nbV = 64 ;

VertexData *vertices = new VertexData[nbV*nbV] ;
float x,y,z, a ;
int coordX, coordY ;
x = -1.0f ;
y = -1.0f ;
a = 2.0f/nbV ;
for (int i = 0; i < nbV; ++i) {
    x = -1.0f + i * a ;
    for (int j = 0; j < nbV; ++j) {
        y = -1.0f + j * a ;
        coordX = img.width() * (float) i/nbV ;
        coordY = img.height() * (float) j/nbV ;
        z = img.pixelColor(coordX, coordY).black() / 256.0f;
        vertices[i*nbV+j] = {QVector3D(x, y, z), QVector2D((1.0f/(nbV-1))*i, (1.0f/(nbV-1))*j)} ;
    }
}
```



Question 2 :

```
//QVector3D n = QVector3D(diff.y(), diff.x(), 0.0).normalized();
QVector3D n = QVector3D(0.0,0.0,1.0).normalized();
```

```
MainWidget::MainWidget(QWidget *parent) :
    QOpenGLWidget(parent),
    geometries(0),
    texture(0),
    angularSpeed(1),
    cam(0.0,0.0,-5.0),
    rotationAxis(0.0,0.0,1.0)
{
}
```

Question 3 :

La mise à jour du terrain est gérée par la fonction update() appelée par QTimer event.

Le QTimer event est, quand à lui, appelé toutes les 12 millisecondes.

```
#ifndef QT_NO_OPENGL
    MainWidget widget(1000);
    widget.show();
    MainWidget widget2(100);
    widget2.show();
    MainWidget widget3(10);
    widget3.show();
    MainWidget widget4(1);
    widget4.show();
```

On remarque que la vitesse de rotation diffère en fonction du nombre de fps.

```
MainWidget widget(1000);
widget.show();
MainWidget widget2(100);
widget2.show();
MainWidget widget3(10);
widget3.show();
MainWidget widget4(1);
widget4.show();

QObject::connect(&widget, SIGNAL(angularSpeedChanged(int)), &widget2, SLOT(setAngularSpeed(int)));
QObject::connect(&widget, SIGNAL(angularSpeedChanged(int)), &widget3, SLOT(setAngularSpeed(int)));
QObject::connect(&widget, SIGNAL(angularSpeedChanged(int)), &widget4, SLOT(setAngularSpeed(int)));

QObject::connect(&widget2, SIGNAL(angularSpeedChanged(int)), &widget, SLOT(setAngularSpeed(int)));
QObject::connect(&widget2, SIGNAL(angularSpeedChanged(int)), &widget3, SLOT(setAngularSpeed(int)));
QObject::connect(&widget2, SIGNAL(angularSpeedChanged(int)), &widget4, SLOT(setAngularSpeed(int)));

QObject::connect(&widget3, SIGNAL(angularSpeedChanged(int)), &widget, SLOT(setAngularSpeed(int)));
QObject::connect(&widget3, SIGNAL(angularSpeedChanged(int)), &widget2, SLOT(setAngularSpeed(int)));
QObject::connect(&widget3, SIGNAL(angularSpeedChanged(int)), &widget4, SLOT(setAngularSpeed(int)));

QObject::connect(&widget4, SIGNAL(angularSpeedChanged(int)), &widget, SLOT(setAngularSpeed(int)));
QObject::connect(&widget4, SIGNAL(angularSpeedChanged(int)), &widget2, SLOT(setAngularSpeed(int)));
QObject::connect(&widget4, SIGNAL(angularSpeedChanged(int)), &widget3, SLOT(setAngularSpeed(int)));
```

```
void MainWidget::setAngularSpeed(int value)
{
    if (value != angularSpeed) {
        angularSpeed = value ;
        emit angularSpeedChanged(value);
    }
}
```

Problèmes rencontrés et solutions trouvées :

J’ai eu du mal à prendre en main le code existant ainsi que les fonctions inhérentes à Qt et avec lesquelles je ne suis pas familier. Cependant, en utilisant la dicumentation en en fouillant bien, j’ai pu trouver des solutions.