Alexandre
Louveau
Master 2 - IMAGINA

# Moteur de jeux

# Rapport TP 1 & 2

# TP1 :

## Question 1 :

La classe MainWidget sert à gérer l'affichage et les évènements du programme.

La classe GeometryEngine sert à créer le cube et le tracer.

Le fichier fshader.glsl, le fragment shader, va servir à appliquer la bonne couleur au pixel (en se basant sur la texture fournie).

Le fichier vshader, le vertex shader, a calculer la position des vertices par rapport à la fenêtre.

## Question 2 :

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

Cette fonction créée les sommets du cube et range dans un tableau les indices des sommets représentant les faces dans l'ordre dans lequel elle seront tracées, puis transfert les sommets vers VBO 0 et les indices vers VBO 1.

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

Cette fonction va récupérer la localisation des sommets puis des textures puis tracer les triangles à partir indice de sommet dans VBO 1.

## Question 3 :

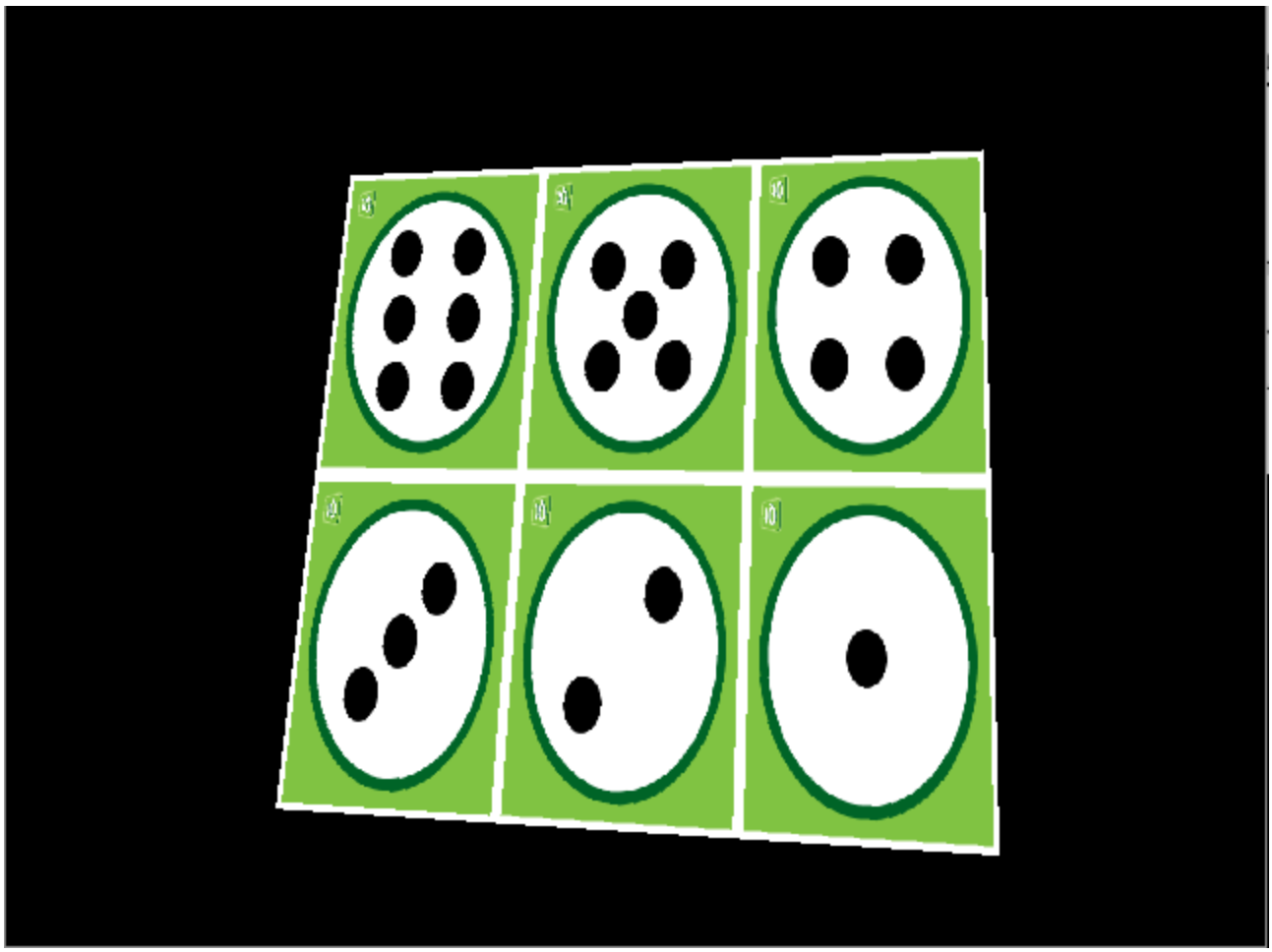
Pour faire une surface plane on l'a sur 2 dimensions (z = 0), on créé ensuite nos sommet et on indexe les sommet des face comme pour le cubes.

```
void GeometryEngine::initPlaneGeometry()
{
   VertexData vertices[] = {
        // Vertex data for face 0
        {QVector3D(-1.5f, -1.5f,  0.0f), QVector2D(0.00f, 1.00f)}, // v0
        {QVector3D(-0.5f, -1.5f,  0.0f), QVector2D(0.33f, 1.00f)}, // v1
        {QVector3D( 0.5f, -1.5f,  0.0f), QVector2D(0.66f, 1.00f)}, // v2
        {QVector3D( 1.5f, -1.5f,  0.0f), QVector2D(1.00f, 1.00f)}, // v3
```

```
        // Vertex data for face 1
        {QVector3D(-1.5f, -0.5f,  0.0f), QVector2D(0.00f, 0.66f)}, // v4
        {QVector3D(-0.5f, -0.5f,  0.0f), QVector2D(0.33f, 0.66f)}, // v5
        {QVector3D( 0.5f, -0.5f,  0.0f), QVector2D(0.66f, 0.66f)}, // v6
        {QVector3D( 1.5f, -0.5f,  0.0f), QVector2D(1.00f, 0.66f)}, // v7

        // Vertex data for face 2
        {QVector3D(-1.5f,  0.5f,  0.0f), QVector2D(0.00f, 0.33f)}, // v8
        {QVector3D(-0.5f,  0.5f,  0.0f), QVector2D(0.33f, 0.33f)}, // v9
        {QVector3D( 0.5f,  0.5f,  0.0f), QVector2D(0.66f, 0.33f)}, // v10
        {QVector3D( 1.5f,  0.5f,  0.0f), QVector2D(1.00f, 0.33f)}, // v11

        // Vertex data for face 3
        {QVector3D(-1.5f,  1.5f,  0.0f), QVector2D(0.00f, 0.00f)}, // v12
        {QVector3D(-0.5f,  1.5f,  0.0f), QVector2D(0.33f, 0.00f)}, // v13
        {QVector3D( 0.5f,  1.5f,  0.0f), QVector2D(0.66f, 0.00f)}, // v14
        {QVector3D( 1.5f,  1.5f,  0.0f), QVector2D(1.00f, 0.00f)}, // v15
    };
    // Indices for drawing cube faces using triangle strips.
    // Triangle strips can be connected by duplicating indices
    // between the strips. If connecting strips have opposite
    // vertex order then last index of the first strip and first
    // index of the second strip needs to be duplicated. If
    // connecting strips have same vertex order then only last
    // index of the first strip needs to be duplicated.
    GLushort indices[] = {
         0,  4,  1,  5,  2,  6,  3,  7,  7,    // Face 0 - triangle strip (v0, v4, v1, v5, v2, v6, v3, v7)
         4,  4,  8,  5,  9,  6, 10,  7, 11, 11, // Face 1 - triangle strip (v4, v8, v5 v9, v6, v10, v7,
v11)
         8,  8, 12,  9, 13, 10, 14, 11, 15, 15  // Face 2 - triangle strip (v8, v12, v9, v13, v10, v14,
v11, v15)
    };

    // Transfer vertex data to VBO 0
    arrayBuf.bind();
    arrayBuf.allocate(vertices, 16 * sizeof(VertexData));

    // Transfer index data to VBO 1
    indexBuf.bind();
    indexBuf.allocate(indices, 29 * sizeof(GLushort));
}


void GeometryEngine::drawPlaneGeometry(QOpenGLShaderProgram *program)
{
    // Tell OpenGL which VBOs to use
    arrayBuf.bind();
    indexBuf.bind();

    // Offset for position
    quintptr offset = 0;
```

```
    // Tell OpenGL programmable pipeline how to locate vertex position data
    int vertexLocation = program->attributeLocation("a_position");
    program->enableAttributeArray(vertexLocation);
    program->setAttributeBuffer(vertexLocation, GL_FLOAT, offset, 3,
sizeof(VertexData));

    // Offset for texture coordinate
    offset += sizeof(QVector3D);

    // Tell OpenGL programmable pipeline how to locate vertex texture coordinate data
    int texcoordLocation = program->attributeLocation("a_texcoord");
    program->enableAttributeArray(texcoordLocation);
    program->setAttributeBuffer(texcoordLocation, GL_FLOAT, offset, 2,
sizeof(VertexData));

    // Draw cube geometry using indices from VBO 1
    glDrawElements(GL_TRIANGLE_STRIP, 29, GL_UNSIGNED_SHORT, 0);
}
```

affichage de la surface plane :



afin de créer une surface de 16 * 16 sommets on créer la fonction :

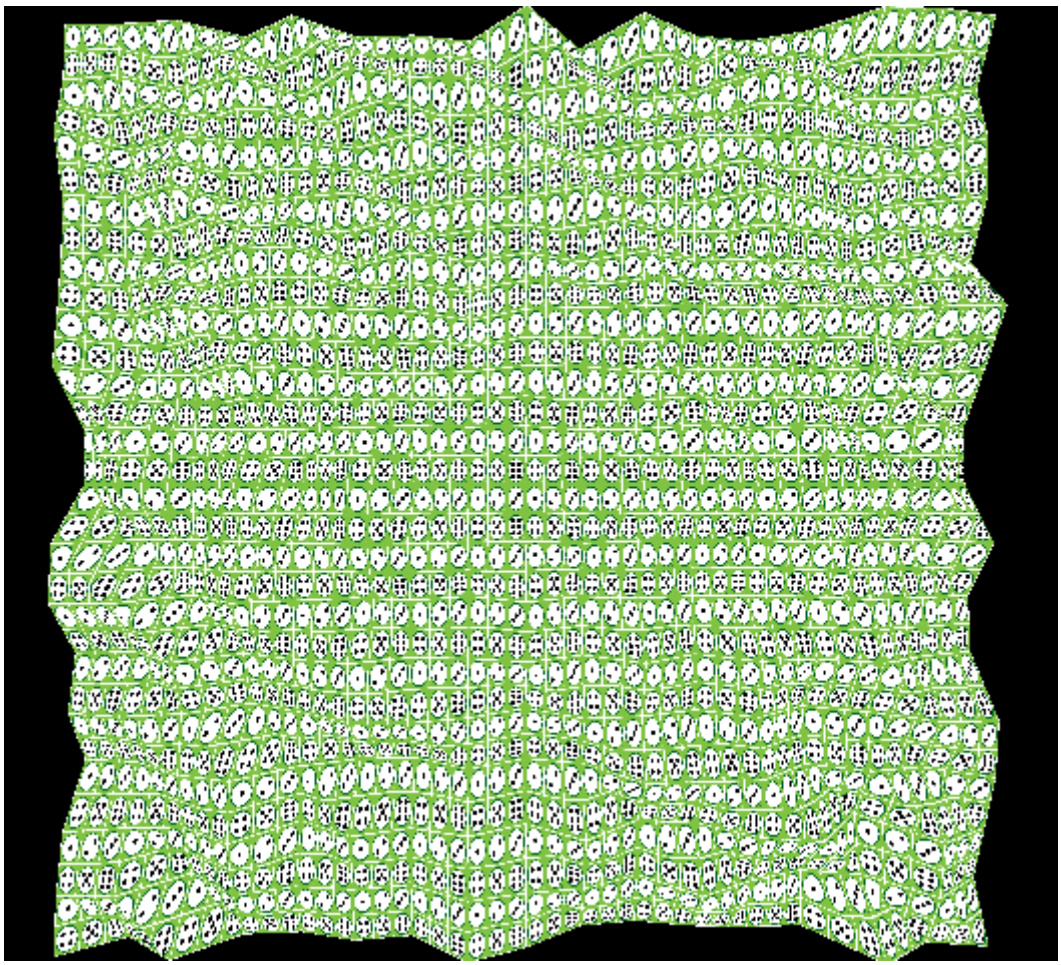
void GeometryEngine::**initPlaneGeometry2**() ; qui va créer les 16 * 16 sommets en partant des coordonnée x =0, y = 0, z = 0 ;

Les sommets sont générés de la manière suivante :

```
while(face_courante < faces) {
      //Départ
      if(x == 0 && y == 0) {
         vertices[sommet_courant + 0] = {QVector3D(x    , y    , random(max_z)),
QVector2D(0.0f, 0.0f)};
         vertices[sommet_courant + 1] = {QVector3D(x + 1, y    , random(max_z)),
QVector2D(1.0f, 0.0f)};
         vertices[sommet_courant + 2] = {QVector3D(x    , y + 1, random(max_z)),
QVector2D(0.0f, 1.0f)};
         vertices[sommet_courant + 3] = {QVector3D(x + 1, y + 1, random(max_z)),
QVector2D(1.0f, 1.0f)};
      } else if ( x != 0 && y == 0 ) { //Première ligne
         vertices[sommet_courant + 0] = {QVector3D(vertices[sommet_courant -
3].position),                                    QVector2D(0.0f, 0.0f)};
         vertices[sommet_courant + 1] = {QVector3D(x + 1, y    , random(max_z)),
QVector2D(1.0f, 0.0f)};
         vertices[sommet_courant + 2] = {QVector3D(vertices[sommet_courant -
1].position),                                    QVector2D(0.0f, 1.0f)};
         vertices[sommet_courant + 3] = {QVector3D(x + 1, y + 1, random(max_z)),
QVector2D(1.0f, 1.0f)};
      } else if ( x == 0 && y != 0) { //Première colonne
         vertices[sommet_courant + 0] = {QVector3D(vertices[( (int) y - 1) * sommets * 4 + 2
+ 4 * (int) x].position),     QVector2D(0.0f, 0.0f)};
         vertices[sommet_courant + 1] = {QVector3D(vertices[( (int) y - 1) * sommets * 4 + 3
+ 4 * (int) x].position),     QVector2D(1.0f, 0.0f)};
         vertices[sommet_courant + 2] = {QVector3D(x    , y + 1, random(max_z)),
QVector2D(0.0f, 1.0f)};
         vertices[sommet_courant + 3] = {QVector3D(x + 1, y + 1, random(max_z)),
QVector2D(1.0f, 1.0f)};
      } else { //Le reste
         vertices[sommet_courant + 0] = {QVector3D(vertices[sommet_courant -
3].position),                                    QVector2D(0.0f, 0.0f)};
         vertices[sommet_courant + 1] = {QVector3D(vertices[( (int) y - 1) * sommets * 4 + 3
+ 4 * (int) x].position),     QVector2D(1.0f, 0.0f)};
         vertices[sommet_courant + 2] = {QVector3D(vertices[sommet_courant -
1].position),                                    QVector2D(0.0f, 1.0f)};
         vertices[sommet_courant + 3] = {QVector3D(x + 1, y + 1, random(max_z)),
QVector2D(1.0f, 1.0f)};
      }

      x++;

      if( x == sommets ) {
         x = 0; y++;
      }
```

```
        sommet_courant += 4; face_courante++;
}
```

Image surface 16 * 16 sommets avec relief :



Une face représente une fois toute la texture du dès.

## Question 4 :

Modification de l’altitude (z) afin de réaliser un relief :

Afin de déplacer la caméra à hauteur fixe et de pouvoir se déplacer à l'aide des touches on créer la fonction void MainWidget::***keyPressEvent***(QKeyEvent *e) :

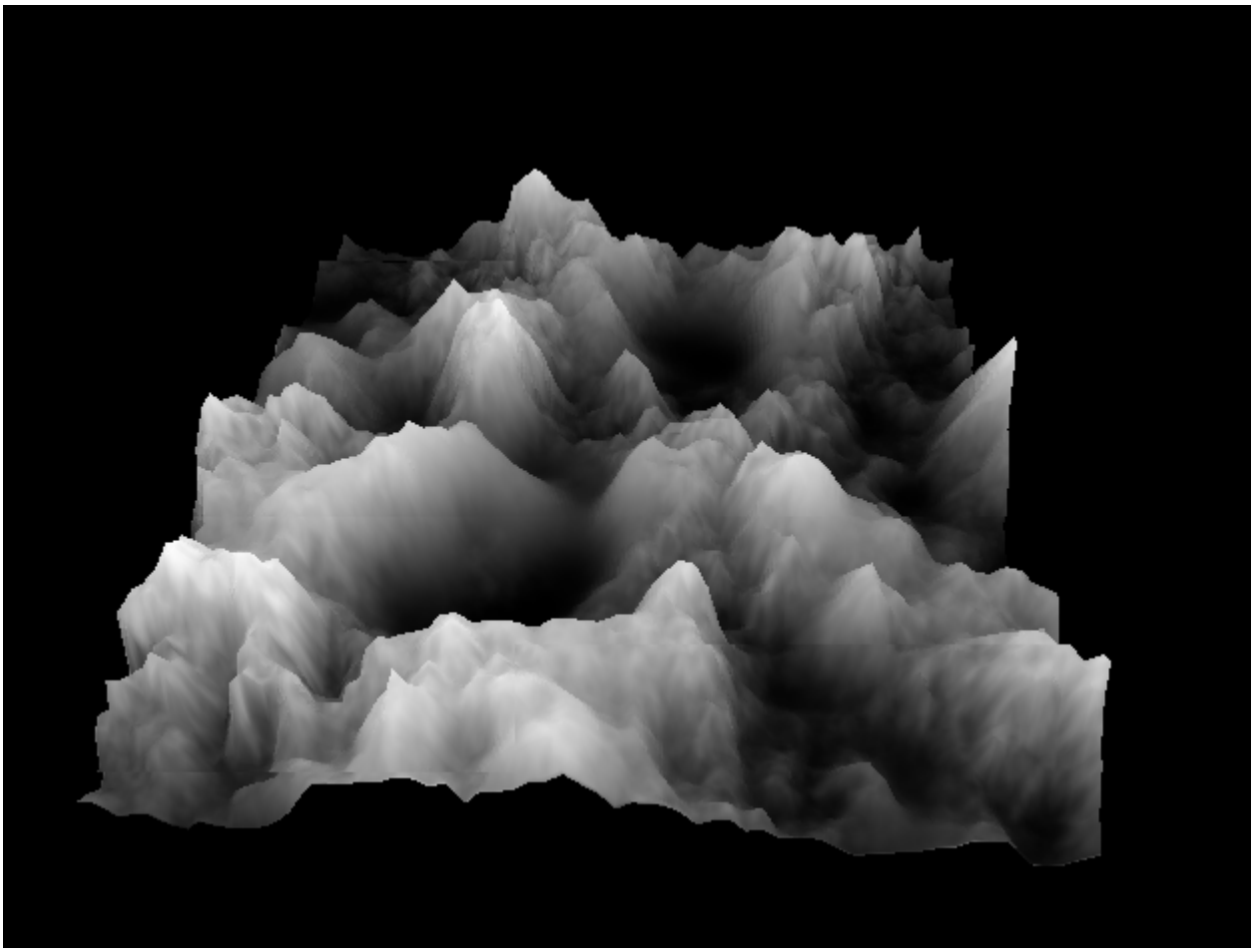
```
void MainWidget::keyPressEvent(QKeyEvent *e) {
   switch (e->key()) {
   case Qt::Key_Up:
   case Qt::Key_Z:
      posY += 1.f/10.f;
      break;
   case Qt::Key_Down:
   case Qt::Key_S:
      posY -= 1.f/10.f;
      break;
   case Qt::Key_Left:
   case Qt::Key_Q:
      posX -= 1.f/10.f;
      break;
   case Qt::Key_Right:
   case Qt::Key_D:
      posX += 1.f/10.f;
      break;
   case Qt::Key_Plus:
   case Qt::Key_A:
      posZ -= 1.f/10.f;
      break;
```

```
    case Qt::Key_Minus:
    case Qt::Key_E:
      posZ += 1.f/10.f;
      break;
    case Qt::Key_Escape:
      std::exit(EXIT_SUCCESS);
    default:
      break;
    }

    update(); //Il faut mettre a jour la scene !

}
```

# TP1 :

## Question 1 :

Rendu avec la heighmap → heightmap-1.png :



## Question 2 :

on initialise
angularSpeed(1),
rotationAxis(0, 0, 1)

Puis on désactive le fait que L'angularSpeed diminue à chaque itération.

## Question 3 :

La mise à jour du terrain est contrôlé dans la fonction :
void MainWidget::***timerEvent***(QTimerEvent *)

La classe QTimer sert à gérer des évènements en fonction du temps, on fourni la durée et le signal à produire à la fin du timer.

Afin de modifier les vitesse de rotation de toutes es fenêtre en même temps on défini une variable : static float speedChange = 1.0;

Et on fait varier sa valeur à l'aide de la fonction : void ***keyPressEvent***(QKeyEvent *e) ;

```
void MainWidget::keyPressEvent(QKeyEvent *e) {
   switch (e->key()) {
   case Qt::Key_Up:
      speedChange += 0.1;
      break;
   case Qt::Key_Down:
      speedChange -= 0.1;
      break;
   case Qt::Key_Escape:
      std::exit(EXIT_SUCCESS);
   default:
      break;
   }
}
```

Et dans la fonction void MainWidget::***timerEvent***(QTimerEvent *)
on ajoute l'instruction : angularSpeed = speedChange;

Observation : Plus les fps sont élevés, plus la rotation est fluide et rapide.