Compte-rendu TP1

Codes graphiques pour le stockage d'informations

HMIN322 - Codage et Compression Multimedia

BOYER Benoît

M2 IMAGINA - Septembre 2017

Table des matières

0.1 Question 1

0.1.1 A quoi servent les classes MainWidget et GeometryEngine ?

La classe MainWidget sert à gérer l'affichage et les évènements du programme, en effet si on se fie aux foncions présentes dans le .h :

```
protected:
    void mousePressEvent(QMouseEvent *e) override;
    void mouseReleaseEvent(QMouseEvent *e) override;
    void timerEvent(QTimerEvent *e) override;

    void initializeGL() override;
    void resizeGL(int w, int h) override;
    void paintGL() override;

    void initShaders();
    void initTextures();
```

Listing 1 – Fonctions de la classe MainWidget

Les fonctions dans l'ordre permettent :

- De gérer le clic souris
- De gérer le relachement du clic souris
- De gérer le temps (et faire un mouvement continu, comme par exemple attrapper le cube et lui inculquer un mouvement pour qu'il continue et ralentisse sur le temps)
- D'initialiser la fenêtre OpenGL
- De gérer la modification de la taille de la fenêtre
- De redessiner et mettre à jour le contenu de la fenêtre OpenGL
- D'initialiser les shaders
- D'initialiser les textures (donc les charger et de les préparer pour les utiliser plus tard)

La classe GeometryEngine est utilisée pour la géométrie du dé présent :

```
    void drawCubeGeometry(QOpenGLShaderProgram *program);

private:
    void initCubeGeometry();
```

Listing 2 – Fonctions de la classe GeometryEngine

Les fonctions présentes servent à dessiner et à initialiser le cube qui, une fois texturé, sera le dé à la fenêtre.

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

Les fichiers sont dans l'ordre le *fragment shader* et le *vertex shader*. Le fragment shader va servir à appliquer la bonne couleur au pixel (en se basant sur la texture fournie), tandis que le vertex shader va calculer la position des vertices par rapport à la fenêtre.

0.2 Question 2

0.2.1 Expliquer le fonctionnement des deux fonctions de la classe CubeGeometry.

La fonction void initCubeGeometry() permet d'initialiser le cube en créant dans un premier temps les vertices du cube :

```
VertexData vertices[] = {
    // Vertex data for face 0
    {QVector3D(-1.0f, -1.0f,  1.0f), QVector2D(0.0f, 0.0f)}, // v0
    {QVector3D( 1.0f, -1.0f,  1.0f), QVector2D(0.33f, 0.0f)}, // v1
    {QVector3D(-1.0f,  1.0f,  1.0f), QVector2D(0.0f, 0.5f)}, // v2
    {QVector3D( 1.0f,  1.0f,  1.0f), QVector2D(0.33f, 0.5f)}, // v3

    // Vertex data for face 1
    {QVector3D( 1.0f, -1.0f,  1.0f), QVector2D( 0.0f, 0.5f)}, // v4
    {QVector3D( 1.0f, -1.0f, -1.0f), QVector2D(0.33f, 0.5f)}, // v5
    {QVector3D( 1.0f,  1.0f,  1.0f), QVector2D(0.0f, 1.0f)}, // v6
    {QVector3D( 1.0f,  1.0f, -1.0f), QVector2D(0.33f, 1.0f)}, // v7
```

Listing 3 – Création de deux faces du cube

Pour ensuite dans un second temps, recenser les indices pour faire les triangles des faces des cubes :

```
GLushort indices[] = {
     0,  1,  2,  3,  3,     // Face 0 - triangle strip ( v0, v1, v2, v3)
     4,  4,  5,  6,  7,  7, // Face 1 - triangle strip ( v4, v5, v6, v7)
     8,  8,  9, 10, 11, 11, // Face 2 - triangle strip ( v8, v9, v10, v11)
    12, 12, 13, 14, 15, 15, // Face 3 - triangle strip (v12, v13, v14, v15)
    16, 16, 17, 18, 19, 19, // Face 4 - triangle strip (v16, v17, v18, v19)
    20, 20, 21, 22, 23      // Face 5 - triangle strip (v20, v21, v22, v23)
};
```

Listing 4 – Création des indices

Et pour terminer, on transfère les données au GPU via des buffers :

```
1     // Transfer vertex data to VBO 0
2     arrayBuf.bind();
3     arrayBuf.allocate(vertices, 24 * sizeof(VertexData));
4
5     // Transfer index data to VBO 1
6     indexBuf.bind();
7     indexBuf.allocate(indices, 34 * sizeof(GLushort));
```

Listing 5 – Transfert des données via un VBO

Une fois terminé, la fonction void drawCubeGeometry(QOpenGLShaderProgram * program) doit être utilisée pour dessiner le cube.

Dans un premier temps, on va indiquer à OpenGL quels VBO (*vertex shaders*) utiliser :

```
1     // Tell OpenGL which VBOs to use
2     arrayBuf.bind();
3     indexBuf.bind();
```

Listing 6 – Selection des VBO

Ensuite, on indique au pointeur le début du VBO, puis le lieu des données :

```
1     // Offset for position
2     quintptr offset = 0;
3
4     // Tell OpenGL programmable pipeline how to locate vertex
      position data
5     int vertexLocation = program->attributeLocation("a_position"
      );
6     program->enableAttributeArray(vertexLocation);
7     program->setAttributeBuffer(vertexLocation, GL_FLOAT, offset
      , 3, sizeof(VertexData));
```

Listing 7 – Indication aux vshader des données

Ensuite, on indique où se trouve les coordonnées pour les textures dans la mémoire, puis on la donne au buffer :

```
1     // Offset for texture coordinate
2     offset += sizeof(QVector3D);
3
4     // Tell OpenGL programmable pipeline how to locate vertex
      texture coordinate data
5     int texcoordLocation = program->attributeLocation("
      a_texcoord");
6     program->enableAttributeArray(texcoordLocation);
7     program->setAttributeBuffer(texcoordLocation, GL_FLOAT,
      offset, 2, sizeof(VertexData));
```

Listing 8 – Indication des coordonnées de texture

Et pour finir, on dessine le cube.

```
1     // Draw cube geometry using indices from VBO 1
2     glDrawElements(GL_TRIANGLE_STRIP, 34, GL_UNSIGNED_SHORT, 0);
```

Listing 9 – Dessin du cube

0.3 Question 3

0.3.1 Création d'une surface plane

Pour créer une surface plane, il faut créer une surface en deux dimentions, dans ce cas il faut mettre un des axes à 0, ici ça sera l'axe z. La création de la surface plane se fera dans la fonction void GeometryEngine::initPlaneGeometry(). Dans un premier temps, on va recenser les vertices :

```
1     VertexData vertices[] = {
2             // Vertex data for face 0
3             {QVector3D(-1.5f, -1.5f,  0.0f), QVector2D(0.00f,
      1.00f)}, // v0
4             {QVector3D(-0.5f, -1.5f,  0.0f), QVector2D(0.33f,
      1.00f)}, // v1
5             {QVector3D( 0.5f, -1.5f,  0.0f), QVector2D(0.66f,
      1.00f)}, // v2
6             {QVector3D( 1.5f, -1.5f,  0.0f), QVector2D(1.00f,
      1.00f)}, // v3
7
8             // Vertex data for face 1
9             {QVector3D(-1.5f, -0.5f,  0.0f), QVector2D(0.00f,
      0.66f)}, // v4
10            {QVector3D(-0.5f, -0.5f, -1.0f), QVector2D(0.33f,
      0.66f)}, // v5
11            {QVector3D( 0.5f, -0.5f, -2.0f), QVector2D(0.66f,
      0.66f)}, // v6
12            {QVector3D( 1.5f, -0.5f,  0.0f), QVector2D(1.00f,
      0.66f)}, // v7
13
14            // Vertex data for face 2
15            {QVector3D(-1.5f,  0.5f,  0.0f), QVector2D(0.00f,
      0.33f)}, // v8
16            {QVector3D(-0.5f,  0.5f,  0.0f), QVector2D(0.33f,
      0.33f)}, // v9
17            {QVector3D( 0.5f,  0.5f, -1.0f), QVector2D(0.66f,
      0.33f)}, // v10
18            {QVector3D( 1.5f,  0.5f,  0.0f), QVector2D(1.00f,
      0.33f)}, // v11
19
20            // Vertex data for face 3
21            {QVector3D(-1.5f,  1.5f,  0.0f), QVector2D(0.00f,
      0.00f)}, // v12
22            {QVector3D(-0.5f,  1.5f,  0.0f), QVector2D(0.33f,
      0.00f)}, // v13
23            {QVector3D( 0.5f,  1.5f,  0.0f), QVector2D(0.66f,
      0.00f)}, // v14
24            {QVector3D( 1.5f,  1.5f,  0.0f), QVector2D(1.00f,
      0.00f)}, // v15
25    };
```

Listing 10 – Création des points

Puis on enverra les données, comme effectué pour le cube.

0.4 Question 4

0.4.1 Modifier l'altitude des sommets pour effectuer un relief

Il suffit juste de modifier la valeur z des sommets sur le terrain plat, ainsi en modifiant quelques valeurs nous pouvons obtenir un terrain comme le suivant :

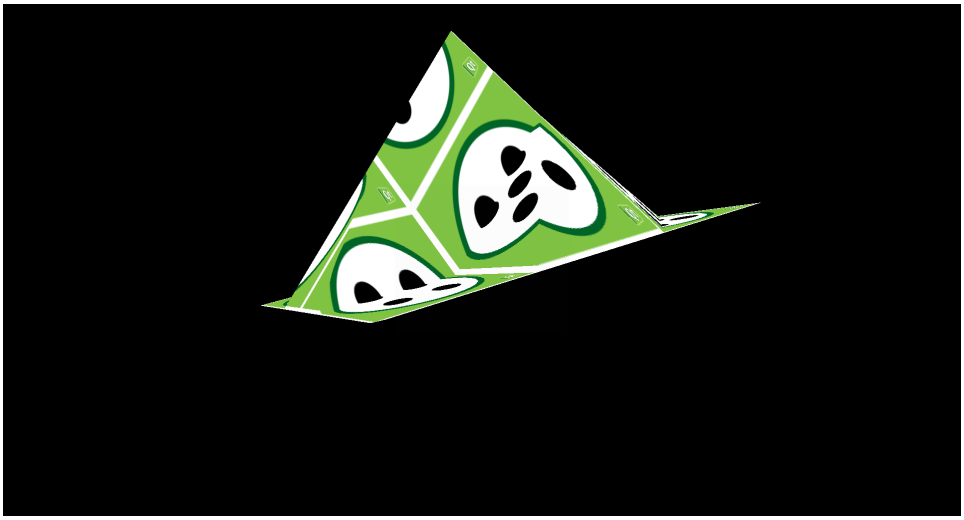

FIGURE 1 – Exemple d'un terrain avec relief

0.4.2 Déplacer la caméra à hauteur fixe

Pour déplacer la caméra à hauteur fixe, on va devoir déplacer la caméra uniquement sur les axes x et y, en incrémentant la valeur voulue selon la touche pressée.
Dans un premier temps, nous allons devoir recréer la fonction keyPressEvent (QKeyEvent *e) pour récupérer l'entrée clavier, et plus précisément la touche appuyée.
On va récuperer la touche appuyée avec e->key(), auquel on fera une comparaison avec la touche voulue :

```
1 void MainWidget::keyPressEvent(QKeyEvent *e) {
2     if (e->key() == Qt::Key_Escape)
3         std::exit(0);
```

Listing 11 – Exemple où on regarde si la touche Échap est celle pressée

Une fois qu'on peut récupérer les entrées utilisateurs, l'étape suivante est de pouvoir mettre les coordonnées en dynamique, pour cela on va modifier une ligne de la fonction void MainWidget::paintGL() :

```
1     QMatrix4x4 matrix;
2     matrix.translate(posx, posy, -5.0);
```

Listing 12 – La translation en x et y ont désormais une variable globale

Ensuite, dans la fonction keyPressEvent(QKeyEvent *e), on va simplement incrémenter la globale selon l'entrée clavier :

```
1     //Reception des inputs
2     float _x = (float)(e->key() == Qt::Key_Right || e->key() ==
      Qt::Key_D) - (float)(e->key() == Qt::Key_Left || e->key() ==
      Qt::Key_Q);
3     float _y = (float)(e->key() == Qt::Key_Up    || e->key() ==
      Qt::Key_Z) - (float)(e->key() == Qt::Key_Down || e->key() ==
      Qt::Key_S);
4
5     posx += _x/10.f;
6     posy += _y/10.f;
7
8     update(); //Il faut mettre a jour la scene !
```

Listing 13 – L'incrémentation de la position selon l'entrée utilisateur

Suite à cette fonction, on peut déplacer la caméra en utilisant les flèches du clavier ou les touches ZQSD.

0.5 Bonus

A partir de cette question, tout le code implémenté et cité ne figure pas dans le programme final.

0.5.1 Jouer avec la lumière

Pour la lumière, le premier élément serait d'activer l'éclairage, puis d'allumer une lumière précise :

```
1 glEnable(GL_LIGHTNING); //Activation de l'eclairage
2 glEnable(GL_LIGHT0); //Activation de la lumiere 1
```

Listing 14 – Activation de la lumière

Ensuite, il faudra appliquer un matériel plus poussé sur le plan, c'est à dire définir la couleur en spéculaire, ambiante et diffuse :

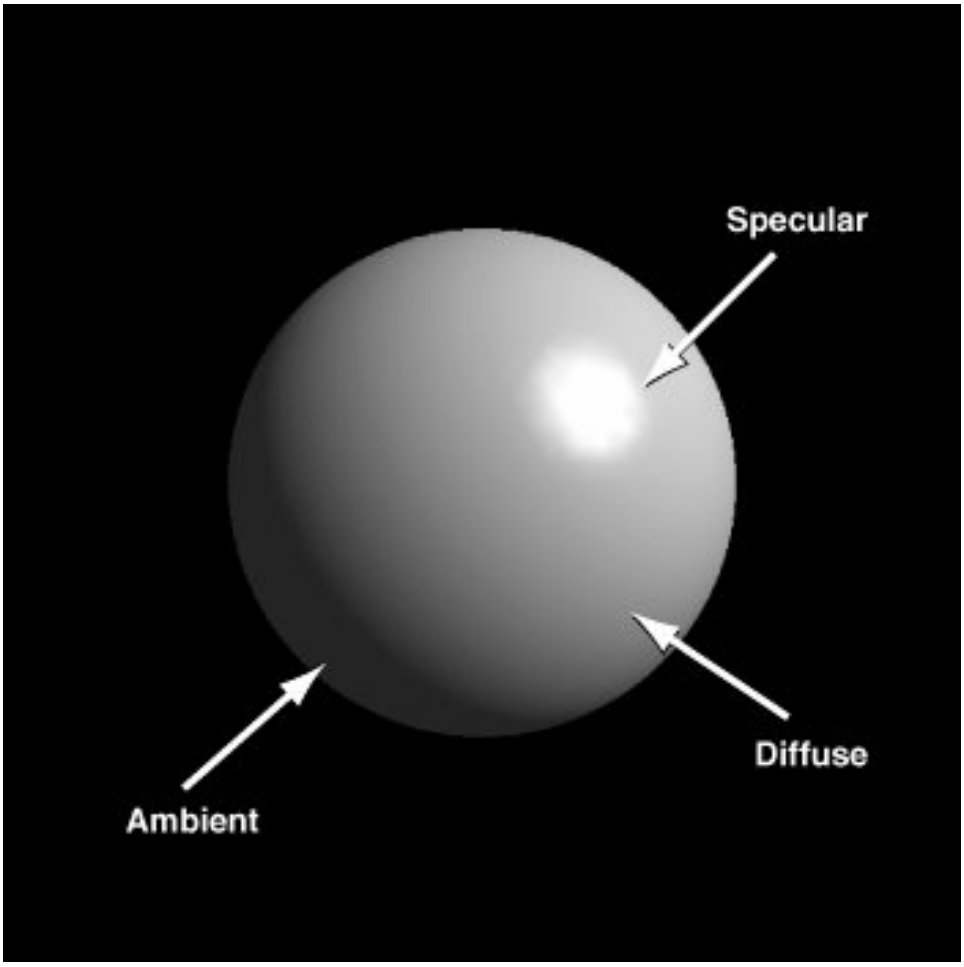

FIGURE 2 – Placement des matériaux

```
GLfloat Lum0Amb[4] = {.2, .2, .2, 1.}; //Materiau ambiant
GLfloat Lum0Dif[4] = {1., 1., 1., 1.}; //Materiau diffuse
GLfloat Lum0Spe[4] = {1., 1., 1., 1.}; //Materiau speculaire
GLfloat Lum0Pos[4] = {0., 0., 5., 1.}; //Position de la lumiere

glLightfv(GL_LIGHT0, GL_AMBIENT,  Lum0Amb}; //Application de la
    couleur ambiante
glLightfv(GL_LIGHT0, GL_DIFFUSE,  Lum0Dif}; //Application de la
    couleur diffuse
glLightfv(GL_LIGHT0, GL_SPECULAR, Lum0Spe}; //Application de la
    couleur speculaire
glLightfv(GL_LIGHT0, GL_POSITION, Lum0Pos}; //Deplacement de la
    lumiere la position donnee
```

Listing 15 – Configuration de la lumière

Cette lumière est placée en hauteur et possède une lumière de type ponctuelle, nous avons dans ce cas un éclairage de type soleil. Pour les lumières localisées, il faudra déplacer la source lumineuse (qui peut être supplémentaire à celle déjà existante, en sachant qu'OpenGL gère huit lumières) et la définir comme une lumière de type directionelle (comme une lampe de chevet).

0.5.2 Texturer le terrain en utilisant des couleurs

Pour colorer le terrain (ou du moins certains points), il faudrait déclarer un glColor3f(Rouge, Vert, Bleu) si on utilisait des glBegin().
Dans le cas avec les VBO, il faudrait créer un buffer contenant les couleurs de chaque sommet dans la fonction d'initialisation de la géométrie, puis de les appliquer lors du dessin de l'élément.