Développement d'applications interactives

Compte-rendu TP1

Prise en main de Qt OpenGL

Louis Jean
Master 2 IMAGINE
Université de Montpellier
N° étudiant : 21914083

11 septembre 2024

Table des matières

1 Introduction

Dans ce premier TP, nous prenons en main les fonctionnalités offertes par la bibliothèque Qt pour OpenGL afin de développer une application interactive, ainsi que l'IDE QtCreator. Cette application permet de manipuler un objet 3D et d'interagir avec celui-ci en utilisant des sliders et la souris. Une base de code est fournie pour aider au démarrage.

2 Sliders

L'une des premières étapes consiste à permettre à l'utilisateur de contrôler la rotation d'un objet 3D à l'aide de sliders. Dans cette partie, nous utilisons des connexions de signaux et de slots pour synchroniser les sliders avec la rotation de l'objet affiché dans un widget OpenGL.

Le code ci-dessous montre comment connecter les sliders pour contrôler la rotation de l'objet sur les trois axes x, y et z. Chaque fois que la valeur d'un slider change, cela déclenche une mise à jour de la rotation correspondante dans le widget OpenGL. Les slots étaient déjà présents dans la base de code.

```
connect(xSlider, SIGNAL(valueChanged(int)), glWidget, SLOT(setXRotation(int)));
connect(ySlider, SIGNAL(valueChanged(int)), glWidget, SLOT(setYRotation(int)));
connect(zSlider, SIGNAL(valueChanged(int)), glWidget, SLOT(setZRotation(int)));
```

Ces connexions permettent d'envoyer les nouvelles valeurs de rotation des sliders vers le widget `glWidget`, où elles seront appliquées pour modifier l'orientation de l'objet dans la scène.

Nous devons également mettre à jour les sliders lorsque l'utilisateur interagit directement avec l'objet à l'aide de la souris. Voici comment les valeurs des sliders sont automatiquement mises à jour en fonction des changements de rotation appliqués dans le widget.

```
connect(glWidget, SIGNAL(changeOnXRotation(int)), xSlider, SLOT(setValue(int)));
connect(glWidget, SIGNAL(changeOnYRotation(int)), ySlider, SLOT(setValue(int)));
connect(glWidget, SIGNAL(changeOnZRotation(int)), zSlider, SLOT(setValue(int)));
```

Pour que le code ci-dessus fonctionne, il faut rajouter des signaux dans `glWidget`, qui sont émis dès lors que le slot permettant la rotation via le slider est appelé.

```
void GLWidget::setXRotation(int angle) {
    qNormalizeAngle(angle);
    if (angle != m_xRot) {
        m_xRot = angle;
        emit changeOnXRotation(angle);
        update();
    }
}

// Pareil pour y et z
```

Ainsi, lorsque l'utilisateur modifie l'objet dans l'interface, les sliders ajustent également leurs positions.

De plus, j'ai ajouté des labels x, y et z sur les sliders pour permettre une meilleure clarté visuelle.

```
QObject* xLabel = new QLabel("X", xSlider);
QObject* yLabel = new QLabel("Y", ySlider);
QObject* zLabel = new QLabel("Z", zSlider);
```

Figure 1: Rendu de l'application après ces étapes

3 Création et affichage d'un cube

Dans cette partie du TP, nous allons créer un cube 3D et l'afficher dans le widget OpenGL. Le cube est défini par ses six faces, chacune composée de deux triangles. Cependant, pour avoir des normales correctes et un éclairage cohérent, j'ai dupliqué les sommets au nombre de 24.

Pour ce faire, j'ai créé une classe `Mesh`, qui contient deux vecteurs : l'un pour stocker les positions des sommets et leurs attributs (ici, normales), et l'autre pour stocker une liste de faces indexées. Cette classe contient également deux `QOpenGLBuffer` (VBOs) pour stocker les données et un `QOpenGLVertexArrayObject` (VAO) pour regrouper les états du rendu.

```
class Mesh
{
public:
    Mesh();
    ~Mesh();
    void buildCube();
    void initializeBuffers(QOpenGLShaderProgram* program);
    void render();
private:
    QVector<GLfloat> m_data;
    QVector<GLuint> m_indexes;
    QOpenGLBuffer m_vertexBuffer;
    QOpenGLBuffer m_indexBuffer;
    QOpenGLVertexArrayObject m_vao;
};
```

La méthode `initializeBuffers` est responsable de l'initialisation des buffers OpenGL. Elle commence par créer le VAO pour stocker les états de rendu, puis lie et alloue les VBOs avec les positions des sommets, les normales et les indices des faces.

```
void Mesh::initializeBuffers(QOpenGLShaderProgram *program) {
    m_vao.create();
    m_vao.bind();
    m_vertexBuffer.create();
    m_vertexBuffer.bind();
    m_vertexBuffer.allocate(m_data.constData(), m_data.size() * sizeof(GLfloat));
    program->enableAttributeArray(0); // Position
    program->setAttributeBuffer(0, GL_FLOAT, 0, 3, 7 * sizeof(GLfloat));
    program->enableAttributeArray(1); // Normale
    program->setAttributeBuffer(1, GL_FLOAT, 4 * sizeof(GLfloat), 3, 7 * sizeof(
    GLfloat));
    m_indexBuffer.create();
    m_indexBuffer.bind();
    m_indexBuffer.allocate(m_indexes.constData(), m_indexes.size() * sizeof(
    GLuint));
    m_vao.release();
}
```

Une fois les buffers initialisés, la méthode `render` utilise ces buffers pour dessiner le cube en appelant `glDrawElements`, qui utilise les indices stockés pour former les triangles du cube.

Les méthodes `buildCube` et `initializeBuffers` sont appelées dans la fonction `initializeGL` de `glWidget`, tandis que `render` est utilisée dans `paintGL` (toujours dans `glWidget`), tout ceci grâce à l'ajout d'un attribut `m_mesh` dans `glWidget`.

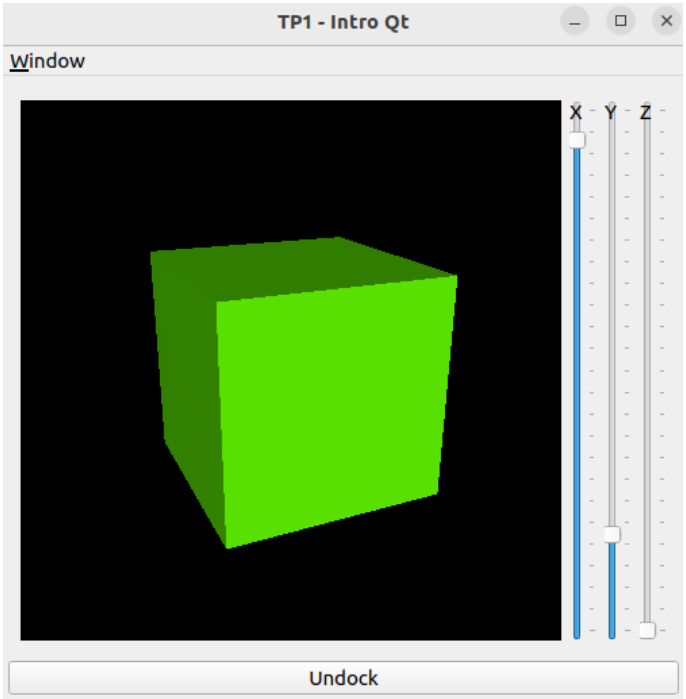


Figure 2: Rendu du cube créé

4 Chargement et affichage d'un maillage

Dans cette dernière partie du TP, nous allons charger un fichier de maillage 3D externe et l'afficher dans notre scène OpenGL.

Premièrement, j'ai ajouté un bouton dans la fenêtre afin de permettre l'ouverture de l'explorateur de fichiers.

```
meshBtn = new QPushButton("Open mesh", this);
connect(meshBtn, &QPushButton::clicked, this, &Window::openFileExplorer);
mainLayout->addWidget(meshBtn);
```

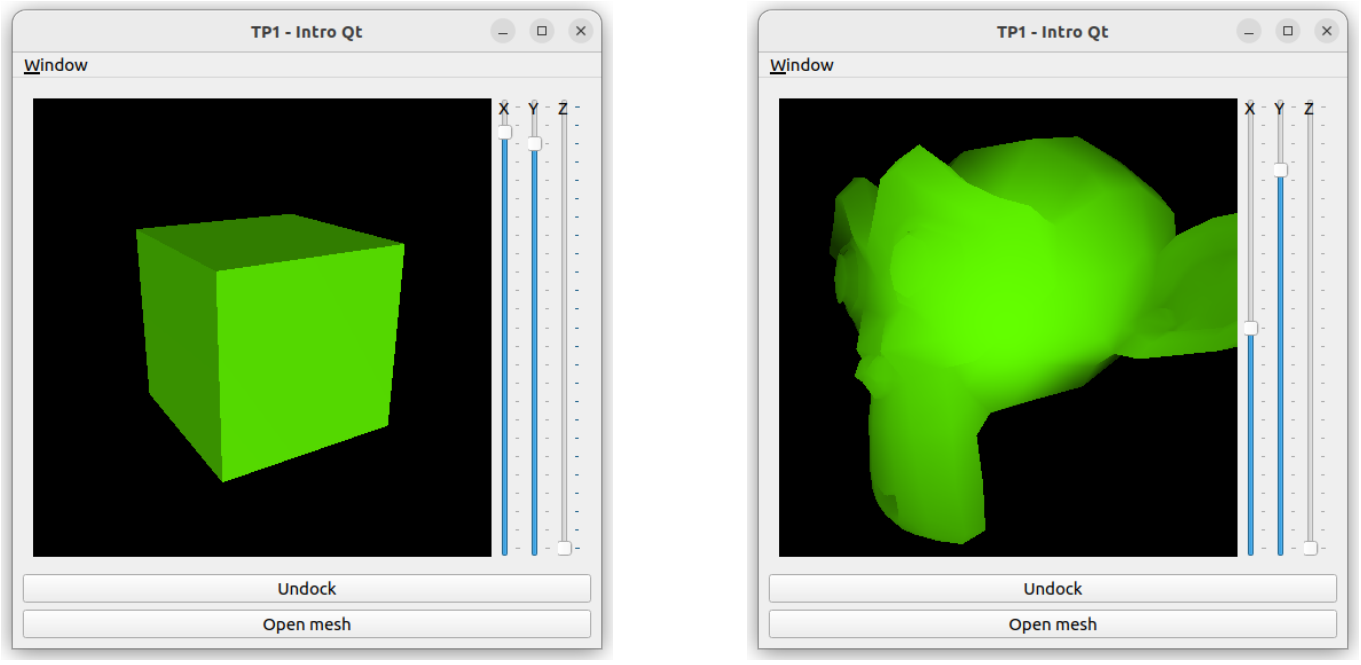
Ce bouton fait appel à une fonction, qui ouvre l'explorateur de fichiers.

```
void Window::openFileExplorer() {
    QString fileName = QFileDialog::getOpenFileName(this);
    glWidget->loadMeshFromFile(fileName);
}
```

Ensuite, le nom du fichier sélectionné est passé en argument à la fonction `loadMeshFromFile` de la classe `glWidget`, qui appelera `loadMeshFromFile` de la classe `Mesh`.

```
void GLWidget::loadMeshFromFile(QString fileName) {
    m_mesh.loadMeshFromFile(fileName.toStdString());
    m_mesh.initializeBuffers(m_program);
}
```

`loadMeshFromFile` de la classe `Mesh` s'inspire fortement d'une fonction de chargement de maillages au format `.off` présente dans les TP de l'UE Programmation 3D. Elle gère les positions et les faces, mais pas les normales (j'ai donc mis les normales aux valeurs des positions pour tout de même avoir un rendu plus agréable).



(a) Bouton pour charger un maillage

(b) Maillage de Suzanne affiché

Figure 3: Bouton et maillage

5 Conclusion

Ce TP m'a permis de prendre en main les concepts fondamentaux du développement d'une application interactive en 3D avec Qt OpenGL. Cette base solide pourra être enrichie au cours des prochains TP.