Développement d'applications interactives

Compte-rendu TP5

Texture 3D

Louis Jean

Master 2 IMAGINE
Université de Montpellier
N° étudiant : `21914083`

11 octobre 2024

Table des matières

1 Introduction

Ce TP a pour but de nous introduire au concept de rendu volumique via les textures 3D sous Qt.

2 Connexion des signaux

2.1 Mise à jour des positions des plans de coupes avec les sliders

```
connect(madDockWidget, &TextureDockWidget::xValueChanged, viewer, &TextureViewer::setXCut);
connect(madDockWidget, &TextureDockWidget::yValueChanged, viewer, &TextureViewer::setYCut);
connect(madDockWidget, &TextureDockWidget::zValueChanged, viewer, &TextureViewer::setZCut);
```

2.2 Modification de la direction de visibilité

```
connect(madDockWidget, &TextureDockWidget::xInvert, viewer, &TextureViewer::invertXCut);
connect(madDockWidget, &TextureDockWidget::yInvert, viewer, &TextureViewer::invertYCut);
connect(madDockWidget, &TextureDockWidget::zInvert, viewer, &TextureViewer::invertZCut);
```

2.3 Affichage des plans de coupe

```
connect(madDockWidget, &TextureDockWidget::xDisplay, viewer, &TextureViewer::setXCutDisplay);
connect(madDockWidget, &TextureDockWidget::yDisplay, viewer, &TextureViewer::setYCutDisplay);
connect(madDockWidget, &TextureDockWidget::zDisplay, viewer, &TextureViewer::setZCutDisplay);
```

Après avoir réalisé ces étapes, on peut observer les 3 plans de coupe et les déplacer.

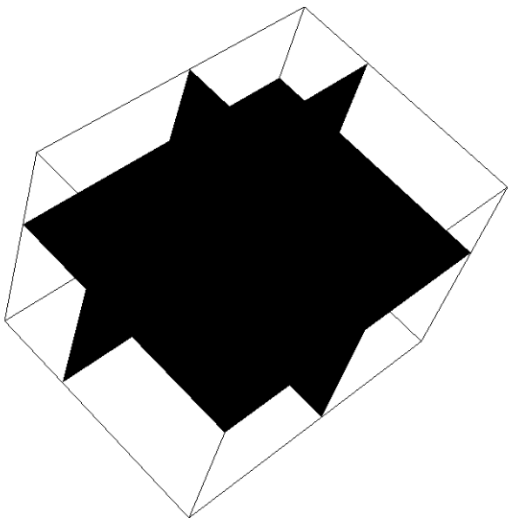


Figure 1: Rendu des 3 plans de coupe dans le cube

3 Création et envoi de la texture

Ici, on s'assure que la texture soit bien remplie pour chaque pixel sur chaque canal. Pour cela, on remplit le tableau `rgbTexture`.

```
rgbTexture = new unsigned char[n[0]*n[1]*n[2]*4];
    for(int i = 0; i < labelsToColor.size(); ++i) {
        int value = data[i];
        QColor color = labelsToColor[value];
        rgbTexture[4*i] = color.red();
        rgbTexture[4*i+1] = color.green();
        rgbTexture[4*i+2] = color.blue();
        rgbTexture[4*i+3] = color.alpha();
    }
```

Ensuite, on initialise la texture et ses paramètres, comme en OpenGL classique (à noter qu'ici on utilise bien une texture 3D).

```
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA, n[0], n[1], n[2], 0, GL_RGBA, GL_UNSIGNED_BYTE,
↪  rgbTexture);
```

4 Affichage de la texture sur les plans de coupe

4.1 Envoi des uniform aux shaders, activation et binding de la texture

Il faut ensuite envoyer toutes les variables uniform nécessaires aux shaders, à savoir les limites de la boîte englobante, les positions des plans le long des axes et leurs directions. Enfin, on peut activer l'unité de texture et lier notre texture précédemment créée.

```
// xMax, yMax, zMax
glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID, "xMax"), xMax);
glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID, "yMax"), yMax);
glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID, "zMax"), zMax);

// xCutPosition, yCutPosition, zCutPosition
glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID, "xCutPosition"),
↪  xCutPosition);    glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID,
↪  "yCutPosition"), yCutPosition);
glFunctions->glUniform1f(glFunctions->glGetUniformLocation(programID, "zCutPosition"),
↪  zCutPosition);

// xCutDirection, yCutDirection, zCutDirection
glFunctions->glUniform1i(glFunctions->glGetUniformLocation(programID, "xCutDirection"),
↪  xCutDirection);
glFunctions->glUniform1i(glFunctions->glGetUniformLocation(programID, "yCutDirection"),
↪  yCutDirection);
```

```
glFunctions->glUniform1i(glFunctions->glGetUniformLocation(programID, "zCutDirection"),
↪  zCutDirection);

// Activation et binding de la texture
glFunctions->glActiveTexture(GL_TEXTURE0);
glFunctions->glBindTexture(GL_TEXTURE_3D, textureId);
```

Pour calculer les UV, on normalise simplement les coordonnées du sommet en question, en divisant chacune par les bornes de la boîte englobante, tout ceci dans le vertex shader.

```
textCoord.x = position.x / xMax;
textCoord.y = position.y / yMax;
textCoord.z = position.z / zMax;
```

4.2 Complétion de la fonction `ComputeVisibility`

Pour savoir si un fragment est visible depuis la caméra, on utilise le produit scalaire entre le vecteur qui part du plan de coupe vers le point considéré et le vecteur de direction du plan en question. Si ce produit scalaire est négatif, alors le fragment correspondant n'est pas visible par la caméra, et on peut le discard pour ne pas avoir à l'afficher. Voici comment j'ai fait ceci dans la fonction du fragment shader.

```
bool ComputeVisibility(vec3 point){
        vec3 xDir = vec3(1.*xCutDirection,0.,0.);
        vec3 yDir = vec3(0.,1.*yCutDirection,0.);
        vec3 zDir = vec3(0.,0.,1.*zCutDirection);

        vec3 xCut = vec3(xCutPosition, 0.,0.);
        vec3 yCut = vec3(0., yCutPosition,0.);
        vec3 zCut = vec3(0., 0.,zCutPosition);

        //TODO compute visibility

        vec3 xVec = point - xCut;
        vec3 yVec = point - yCut;
        vec3 zVec = point - zCut;

        float xVisibility = dot(xVec,xDir);
        float yVisibility = dot(yVec,yDir);
        float zVisibility = dot(zVec,zDir);

        if(xVisibility < 0.0 || yVisibility < 0.0 || zVisibility < 0.0) {
                return false;
        }

        return true;
}
```

N.B : la décision de discard le fragment se fait dans le `main`, en fonction du résultat de la fonction.

En dessinant ensuite la boîte englobante, on obtient un cube que l'on peut agrandir / rétrécir selon les plans, et l'on peut voir la texture apparaître. Mission accomplie !

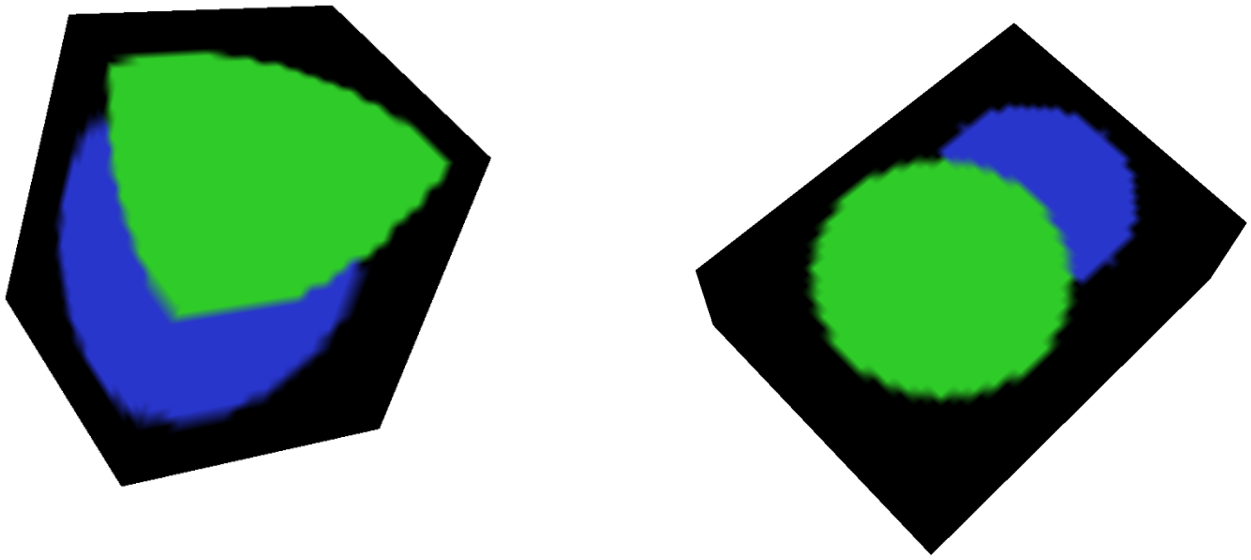


Figure 2: Illustration de la texture 3D

5 Affichage de la texture sur un maillage quelconque

Pour cette partie, je me suis d'abord inspiré de l'action `Load 3D Image`, pour créer un item `Load mesh` dans le menu déroulant `File`.

```
QAction * actionLoadMesh = new QAction("Load mesh", this);
menuFile->addAction(actionLoadMesh);
```

J'ai ensuite ajouté un slot `openMesh`, qui s'inspire grandement du slot `open3DImage`, et qui appelle la fonction `openOffMesh` de `TextureViewer`. J'ai connecté ce slot au signal déclenché lors du clic sur l'action `actionLoadMesh`.

```
connect(actionLoadMesh, SIGNAL(triggered()), this, SLOT(openMesh()));
```

J'ai aussi modifié `openOffMesh` de sorte à ce que les normales du maillage chargé soient calculées, avant de me rendre compte que dans ce programme, cela ne servait à rien puisqu'elles ne sont pas exploitées.

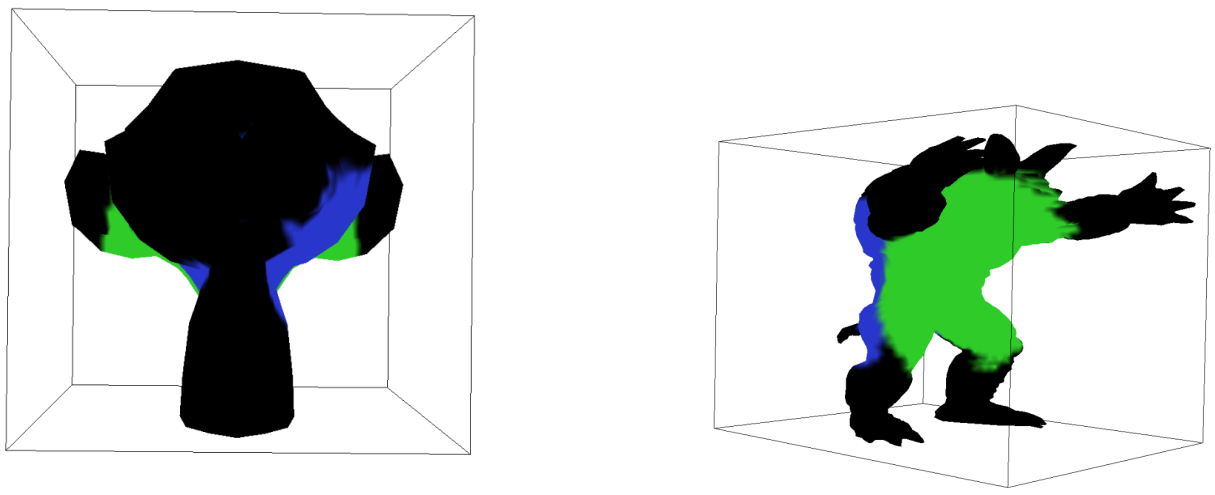
J'ai ensuite normalisé les positions du mesh suivant xMax, yMax et zMax, pour qu'il soit redimensionné selon la taille de la boîte englobante, directement dans `drawMesh`, fonction dont je me suis servi pour afficher le mesh.

```
qglviewer::Vec BBMin{std::numeric_limits<float>::infinity(),
↪  std::numeric_limits<float>::infinity(), std::numeric_limits<float>::infinity()};
qglviewer::Vec BBMax{-std::numeric_limits<float>::infinity(),
↪  -std::numeric_limits<float>::infinity(), -std::numeric_limits<float>::infinity()};

for (const auto& triangle : triangles) {
    for (size_t i = 0; i < 3; ++i) {
        const qglviewer::Vec& vertex = vertices[triangle[i]];
        BBMin = {std::min(BBMin.x, vertex.x), std::min(BBMin.y, vertex.y), std::min(BBMin.z,
        ↪  vertex.z)};
        BBMax = {std::max(BBMax.x, vertex.x), std::max(BBMax.y, vertex.y), std::max(BBMax.z,
        ↪  vertex.z)};
    }
}

for (const auto& triangle : triangles) {
    for (size_t i = 0; i < 3; ++i) {
        const qglviewer::Vec& vertex = vertices[triangle[i]];
        glVertex3f(texture->getXMax() * (vertex.x - BBMin.x) / (BBMax.x - BBMin.x),
        ↪  texture->getYMax() * (vertex.y - BBMin.y) / (BBMax.y - BBMin.y), texture->getZMax()
        ↪  * (vertex.z - BBMin.z) / (BBMax.z - BBMin.z));
        const qglviewer::Vec& normal = normals[triangle[i]];
        glNormal3f(normal.x, normal.y, normal.z);
    }
}
```

Et voici les rendus que j'ai obtenu pour différents maillages.



(a) monkey.off (b) arma.off

Figure 3: Illustration de la texture 3D sur des mesh

6 Conclusion

Ce TP nous a permis de mettre en œuvre une texture 3D sur des volumes et des maillages, tout en maîtrisant la gestion des shaders, des uniforms et des interactions utilisateur. Je pense que l'application de textures 3D sur des maillages ouvre la porte à de nombreuses possibilités. J'ai trouvé ce TP très bien guidé, soutenu par une bonne base de code, et agréable à réaliser.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.