

TP 1

Au début de ce TP/TD, vous recevrez une archive zip contenant une base de code. Ce code permet d'afficher un **maillage triangulaire** à l'aide d'OpenGL.

1. Vous trouverez la description du code dans le sujet.
2. Vous devez faire évoluer ce code au fur et à mesure du TP, pour répondre aux questions.
3. Vous devez déposer votre code, fichier image de résultat et une phrase sur l'état de votre TP sur le moodle.

1 Base de code

Téléchargez l'archive sur le moodle <https://moodle.umontpellier.fr/course/view.php?id=22845>. Pour compiler le code et l'exécuter :

```
$ make  
$ ./tp
```

1.1 Interactions utilisateur

```
1 void key (unsigned char keyPressed, int x, int y)
```

La fonction **key** permet de d'interpréter les entrée clavier utilisateur. Les options de visualisation activée par des touches sont les suivantes, en appuyant sur la touche :

- **n** : activation/désactivation de l'affichage des normales,
- **1** : activation/désactivation de l'affichage du modèle d'entrée sur lequel vous effectuez les calculs de normale,
- **2** : activation/désactivation de l'affichage du modèle transformé,
- **s** : changement entre l'affichage avec les normales de face et de sommet (maillage plus lisse),
- **w** : activation/désactivation de l'affichage en fil de fer,
- **b** : activation/désactivation de l'affichage des repères,
- **f** : activation/désactivation du mode plein écran.

Vous pouvez interagir avec le modèle avec la souris :

- Bouton du milieu appuyé : zoomer ou reculer la caméra,
- Clic gauche appuyé : faire tourner le modèle.

1.2 Rendu de maillages

Le fichier **tp.cpp** contient une méthode **draw** : c'est une fonction appelée pour rafraîchir l'affichage dès que nécessaire. Elle permet de définir les éléments à afficher, leurs couleurs (**glColor3f(r,g,b)** en float de 0 à 1 donnant la couleur **RGB = (r*255, g*255, b*255)**). La fonction **drawMesh()** permet d'afficher un maillage triangulaire. Celle-ci fait appel à deux fonctions contenant du code **OpenGL** basique pour afficher des maillages. La première permet un affichage en utilisant les normales au triangles :

```

1  void drawTriangleMesh( Mesh const & i_mesh ) {
2
3      glBegin(GL_TRIANGLES) ;
4      //Iterer sur les triangles
5      for(unsigned int tIt = 0 ; tIt < i_mesh.triangles.size() ; ++tIt) {
6          //Récupération des positions des 3 sommets du triangle pour l'affichage
7          //Vertices --> liste indexée de sommets
8          //i_mesh.triangles[tIt][i] --> indice du sommet vi du triangle dans la liste
9      ↪ de sommet
10         //pi --> position du sommet vi du triangle
11
12         //Normal au triangle
13         Vec3 n = i_mesh.triangle_normals[tIt] ;
14
15         glNormal3f( n[0] , n[1] , n[2] ) ;
16
17         glVertex3f( p0[0] , p0[1] , p0[2] ) ;
18         glVertex3f( p1[0] , p1[1] , p1[2] ) ;
19         glVertex3f( p2[0] , p2[1] , p2[2] ) ;
20     }
21     glEnd() ;
22 }

```

La première permet un affichage en utilisant les normales aux sommets :

```

1  void drawSmoothTriangleMesh( Mesh const & i_mesh ) {
2
3      glBegin(GL_TRIANGLES) ;
4      //Iterer sur les triangles
5      for(unsigned int tIt = 0 ; tIt < i_mesh.triangles.size() ; ++tIt) {
6          //Récupération des positions des 3 sommets du triangle pour l'affichage
7          //Vertices --> liste indexée de sommets
8          //i_mesh.triangles[tIt][i] --> indice du sommet vi du triangle dans la liste
9      ↪ de sommet
10         //pi --> position du sommet vi du triangle
11         //ni --> normal du sommet vi du triangle pour un affichage lisse
12         Vec3 p0 = i_mesh.vertices[i_mesh.triangles[tIt][0]] ;
13         Vec3 n0 = i_mesh.normals[i_mesh.triangles[tIt][0]] ;
14
15         Vec3 p1 = i_mesh.vertices[i_mesh.triangles[tIt][1]] ;
16         Vec3 n1 = i_mesh.normals[i_mesh.triangles[tIt][1]] ;
17
18         Vec3 p2 = i_mesh.vertices[i_mesh.triangles[tIt][2]] ;
19         Vec3 n2 = i_mesh.normals[i_mesh.triangles[tIt][2]] ;
20
21         //Passage des positions et normales à OpenGL
22         glNormal3f( n0[0] , n0[1] , n0[2] ) ;
23         glVertex3f( p0[0] , p0[1] , p0[2] ) ;
24         glNormal3f( n1[0] , n1[1] , n1[2] ) ;
25         glVertex3f( p1[0] , p1[1] , p1[2] ) ;
26         glNormal3f( n2[0] , n2[1] , n2[2] ) ;
27         glVertex3f( p2[0] , p2[1] , p2[2] ) ;
28     }
29     glEnd() ;
30 }

```

Remarque 1. Vous pouvez vous inspirer de ce code pour le parcours du maillage nécessaire

au calcul des normales. Vous remarquerez qu'il y a plusieurs fonctions commençant par un appel à **draw**, elles permettent l'affichage de vecteurs, repères et champs de vecteurs tels que les normales.

2 Calcul géométrique et implémentation de classes

Compléter la classe **Vec3** qui contient les fonctions essentielles pour le calcul de base : assignation, somme, soustraction, multiplication et division par un scalaire, produit scalaire, produit vectoriel, norme...

```
1   class Vec3 {
2   private :
3       float mVals[3] ;
4   public :
5       Vec3() {}
6       Vec3( float x , float y , float z ) {
7           mVals[0] = x ; mVals[1] = y ; mVals[2] = z ;
8       }
9       float & operator [] (unsigned int c) { return mVals[c] ; }
10      float operator [] (unsigned int c) const { return mVals[c] ; }
11      void operator = (Vec3 const & other) {
12          mVals[0] = other[0] ; mVals[1] = other[1] ; mVals[2] = other[2] ;
13      }
14      float squareLength() const {
15          return mVals[0]*mVals[0] + mVals[1]*mVals[1] + mVals[2]*mVals[2] ;
16      }
17      float length() const { return sqrt( squareLength() ) ; }
18      void normalize() { float L = length() ; mVals[0] /= L ; mVals[1] /= L ; mVals[2] /= L ;
19      ↪ }
20      static float dot( Vec3 const & a , Vec3 const & b ) {
21          //TODO
22      }
23      static Vec3 cross( Vec3 const & a , Vec3 const & b ) {
24          //TODO
25      }
26      void operator += (Vec3 const & other) {
27          mVals[0] += other[0] ;
28          mVals[1] += other[1] ;
29          mVals[2] += other[2] ;
30      }
31      void operator -= (Vec3 const & other) {
32          mVals[0] -= other[0] ;
33          mVals[1] -= other[1] ;
34          mVals[2] -= other[2] ;
35      }
36      void operator *= (float s) {
37          mVals[0] *= s ;
38          mVals[1] *= s ;
39          mVals[2] *= s ;
40      }
41      void operator /= (float s) {
42          mVals[0] /= s ;
43          mVals[1] /= s ;
44          mVals[2] /= s ;
45      }
```

```
} ;
```

1. Compléter les fonctions de calcul du produit scalaire `dot` et du produit vectoriel `cross`
2. Compléter la fonctions de `Mat3` permettant d'effectuer un produit matriciel
3. Compléter la fonction de `Mat3` permettant d'effectuer la multiplication d'appliquer une matrice de transformation à un point : `Mat3*Vec3`.

Remarque 2. 1. On a revu les notions associées au produit scalaire, notamment :

- Le produit scalaire entre le vecteur **a** et le vecteur **b** donne le produit des longueurs multiplié par le cosinus de l'angle formé par ces vecteurs.
 - Le produit scalaire entre un vecteur **a** et un vecteur unitaire **d** vaut la longueur (attention au signe !) de **a** projeté sur la (demi-)droite dirigée par **d**.
2. On a revu les notions associées au produit vectoriel, notamment :
- Le produit vectoriel entre le vecteur **a** et le vecteur **b** est orthogonal à ces deux vecteurs (et sa direction est donnée par la “règle de la main droite”.
 - Sa norme vaut le produit des longueurs multiplié par le sinus de l'angle formé par ces vecteurs.

3 Application

1. Pour tester vos calculs, compléter les fonctions de calcul de normales aux faces triangulaires `computeTrianglesNormals()`. Les normales calculées seront celles du modèle vert. Vous pourrez comparer vos résultats avec les normales du maillage gris `transformed_mesh`.
2. Calculez ensuite les normales par sommet en faisant la moyenne des normales des triangles incidents. N'oubliez pas de normaliser.
3. Remplacer

```
1   for( unsigned int i = 0 ; i < mesh.vertices.size() ; ++i ) {
2       transformed_mesh.vertices.push_back( mesh.vertices[i] +
↪   mesh_transformation.translation ) ;
3   }
```

par

```
1   for( unsigned int i = 0 ; i < mesh.vertices.size() ; ++i ) {
2       transformed_mesh.vertices.push_back(
↪   mesh_transformation.rotation*mesh.vertices[i]
3       + mesh_transformation.translation ) ;
4   }
```

4. Essayer différentes transformations en mettant à jour `mesh_transformation.rotation` et `mesh_transformation.translation`. Créer une matrice `scale` de mise à l'échelle non-uniforme et essayer `mesh_transformation.rotation = mesh_transformation.rotation*scale`. Regarder les normales, que constatez-vous ?