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.

Base de code

Installation

Téléchargez l'archive sur le moodle du cours. Pour compiler le code et l'exécuter :

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

Interactions utilisateur

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

La fonction `key` permet de d'interpréter les entrées clavier utilisateur. Les options de visualisation activées 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` : changement du mode d'affichage (fil de fer/éclairé/non-éclairé/fil de fer + éclairé)
- `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.

Rendu de maillages

Le fichier `tp.cpp` contient une méthode `draw` : c'est une fonction appelée pour rafraichir 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 :

```
    void drawTriangleMesh( Mesh const & i_mesh ) {

        glBegin(GL_TRIANGLES) ;
        //Iterer sur les triangles
        for(unsigned int tIt = 0  ; tIt < i_mesh.triangles.size() ; ++tIt) {
            //Récupération des positions des 3 sommets du triangle pour l'affichage
            //Vertices --> liste indexée de sommets
            //i_mesh.triangles[tIt][i] --> indice du sommet vi du triangle dans la liste
↪   de sommet
            //pi --> position du sommet vi du triangle
            Vec3 p0 = i_mesh.vertices[i_mesh.triangles[tIt][0]] ;
            Vec3 p1 = i_mesh.vertices[i_mesh.triangles[tIt][1]] ;
            Vec3 p2 = i_mesh.vertices[i_mesh.triangles[tIt][2]] ;

            //Normal au triangle
            Vec3 n = i_mesh.triangle_normals[tIt] ;

            glNormal3f( n[0] , n[1] , n[2] ) ;

            glVertex3f( p0[0] , p0[1] , p0[2] ) ;
            glVertex3f( p1[0] , p1[1] , p1[2] ) ;
            glVertex3f( p2[0] , p2[1] , p2[2] ) ;
        }
        glEnd() ;
    }
```

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

```
    void drawSmoothTriangleMesh( Mesh const & i_mesh ) {

        glBegin(GL_TRIANGLES) ;
        //Iterer sur les triangles
        for(unsigned int tIt = 0  ; tIt < i_mesh.triangles.size() ; ++tIt) {
            //Récupération des positions des 3 sommets du triangle pour l'affichage
            //Vertices --> liste indexée de sommets
            //i_mesh.triangles[tIt][i] --> indice du sommet vi du triangle dans la liste
↪   de sommet
            //pi --> position du sommet vi du triangle
            //ni --> normal du sommet vi du triangle pour un affichage lisse
            Vec3 p0 = i_mesh.vertices[i_mesh.triangles[tIt][0]] ;
            Vec3 n0 = i_mesh.normals[i_mesh.triangles[tIt][0]] ;

            Vec3 p1 = i_mesh.vertices[i_mesh.triangles[tIt][1]] ;
            Vec3 n1 = i_mesh.normals[i_mesh.triangles[tIt][1]] ;

            Vec3 p2 = i_mesh.vertices[i_mesh.triangles[tIt][2]] ;
            Vec3 n2 = i_mesh.normals[i_mesh.triangles[tIt][2]] ;

            //Passage des positions et normales à OpenGL
            glNormal3f( n0[0] , n0[1] , n0[2] ) ;
            glVertex3f( p0[0] , p0[1] , p0[2] ) ;
            glNormal3f( n1[0] , n1[1] , n1[2] ) ;
            glVertex3f( p1[0] , p1[1] , p1[2] ) ;
            glNormal3f( n2[0] , n2[1] , n2[2] ) ;
            glVertex3f( p2[0] , p2[1] , p2[2] ) ;
    }
    glEnd() ;
```

```
    }
```

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.

1 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...

```
    class Vec3 {
private :
    float mVals[3] ;
public :
    Vec3() {}
    Vec3( float x , float y , float z ) {
       mVals[0] = x ; mVals[1] = y ; mVals[2] = z ;
    }
    float & operator [] (unsigned int c) { return mVals[c] ; }
    float operator [] (unsigned int c) const { return mVals[c] ; }
    void operator = (Vec3 const & other) {
       mVals[0] = other[0]  ; mVals[1] = other[1] ; mVals[2] = other[2] ;
    }
    float squareLength() const {
       return mVals[0]*mVals[0] + mVals[1]*mVals[1] + mVals[2]*mVals[2] ;
    }
    float length() const { return sqrt( squareLength() ) ; }
    void normalize() { float L = length() ; mVals[0] /= L ; mVals[1] /= L ; mVals[2] /= L ;
↪   }
    //Calculer le produit scalaire entre 2 vecteurs
    static float dot(Vec3 const &a, Vec3 const &b) {
        //TODO : Fonction à compléter
        float res ; //Faire le calcul ici et assigner le resultat à res
        return res ;
    }
    //Calculer le produit vectoriel entre 2 vecteurs
    static Vec3 cross(Vec3 const &a, Vec3 const &b) {
        //TODO : Fonction à compléter
        Vec3 res ; //Faire le calcul ici et assigner le resultat à res
        return res ;
    }
    void operator += (Vec3 const & other) {
        mVals[0] += other[0] ;
        mVals[1] += other[1] ;
        mVals[2] += other[2] ;
    }
    void operator -= (Vec3 const & other) {
        mVals[0] -= other[0] ;
        mVals[1] -= other[1] ;
        mVals[2] -= other[2] ;
    }
```

```
41      void operator *= (float s) {
42          mVals[0] *= s ;
43          mVals[1] *= s ;
44          mVals[2] *= s ;
45      }
46      void operator /= (float s) {
47          mVals[0] /= s ;
48          mVals[1] /= s ;
49          mVals[2] /= s ;
50      }
51  } ;
```

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

2 Application

2.1. Pour tester vos calculs, compléter les fonctions de calcul de normales aux faces triangulaires `computeTrianglesNormals()`. Les normales calculées seront celles du modèle d'entrée (vert). Vous pourrez comparer vos résultats avec les normales du maillage gris `transformed_mesh`.
2.2. Calculez ensuite les normales par sommet en faisant la moyenne des normales des triangles incidents. N'oubliez pas de normaliser.
2.3. Essayer differentes transformation en mettant à jour `Mat3 transformation` et `Vec3 translation`. Créer une matrice de rotation `scale` et une matrice `scale` de mise à l'échelle non-uniforme et essayer `transformation = rotation*scale`. Regarder les normales, que constatez vous ?