Geometrie Algorithms Fall, 2016

TP 1

Problem. You will have to do three things:
1. write basic geometric functions in classes to manipulate geometric primitives like points and vectors.
2. write myMesh.read function to read a mesh in .obj format into a Doubly-Connected Edge List.
3. write some functions in the class myMesh to manipulate a mesh.

1. Basic Geometric Functions.

You are given the following classes for storing and using basic geometric primitives:

1. **myPoint3D**:

```
class myPoint3D
{
public:
  double X, Y, Z;
};
```

   You have to fill-in the following functions in the class `myPoint3D`:

   (a) `double dist(myPoint3D p1)`. Computes the distance between the current point and $p1$.

   (b) `double dist(myPoint3D *p1, myPoint3D *p2)`. Computes the distance from the current point, and the line segment defined by the points $p1$ and $p2$.

2. **myVector3D**:

```
class myVector3D
{
public:
  double dX, dY, dZ;
};
```

   You have to fill-in the following functions in the class `myVector3D`:

   (a) `double operator*(myVector3D & v1)`. Returns the dot product between the current vector and $v1$.

2. Reading a mesh into a Halfedge structure

You are given the following classes for storing a mesh into a Halfedge structure:

1. **myVertex**: stores information for a vertex of the mesh.

```
class myVertex
{
public:
    myPoint3D *point;
    myHalfedge *originof;
    myVector3D *normal;
};
```

   (a) `computeNormal()`. Computes the normal of the vertex by averaging the normals of the faces around that vertex, and storing that in `*normal`.

2. **myHalfedge**: stores information for a halfedge of the mesh.

```
class myHalfedge
{
public:
      myVertex *source;
      myFace *adjacent_face;
      myHalfedge *next;
      myHalfedge *prev;
      myHalfedge *twin;
};
```

3. **myFace**: stores information for a face of the mesh.

```
class myFace
{
public:
    myHalfedge *adjacent_halfedge;
    myVector3D *normal;
};
```

   (a) `void computeNormal()`. Computes the normal vector of the face, and stores in the member variable `*normal`.

4. **myMesh**: stores information for a Mesh.

```
class myMesh
{
public:
      std::vector<myVertex *> vertices;
      std::vector<myHalfedge *> halfedges;
      std::vector<myFace *> faces;
      std::string name;
};
```

   (a) `readFile(std::string filename)`. Reads the .obj file `filename`, and stores it in a halfedge structure. This should be straightforward, except one difficulty of efficiently finding the twin of each halfedge.

Computing twins. An efficient way to compute twins while reading a mesh is to use the `c++` structure

`map<pair<int,int>, myHalfedge *>`,

which stores for each pair of integers (representing the indices of the two vertices in the `vertices` array) the halfedge between them. Use this to remember the halfedges that have already been created so that for each new halfedge, you can find out its twin halfedge (if already created) from this map, and then link them up. For example, if you have made the variable

`map<pair<int,int>, myHalfedge *> table`,

then to store a `myHalfedge *e` at location (a, b), you can do:

`table[ make_pair(a,b) ] = e;`

To locate the `myHalfedge *` variable for a particular location (a, b), you can do:

```
map<pair<int,int>, myHalfedge *>::iterator it = table.find(make_pair(a,b)) ;
if ( it == table.end() )
{
     This means there was no myHalfedge * present at location (a,b).
}
else
{
     It was found. The variable it->second is of type myHalfedge *,
     and is the halfedge present at location (a,b).
}
```

(b) `void computeNormals()`. Computes the normals for each face of the mesh, and then each vertex of the mesh.

(c) `void triangulate()`. Triangulate the mesh (so each face of the mesh will be a triangle).

You can find a nice document explaining the structure of a .obj file here: `http://www.cs.clemson.edu/~dhouse/courses/405/docs/brief-obj-file-format.html`.

It is very important to remember the faces must be stored in the anti-clockwise order. As that will determine how the normals are computed. Plus you need the "twins" of every edge to be in opposite direction.

3. Other code to write

Write other code in the `main.cpp` file so that in this TP, you get the following popup menu options working:

1. `Draw -> Vertex-shading/Face-shading`
2. `Draw -> Mesh`
3. `Draw -> Wireframe`
4. `Draw -> Vertices`
5. `Draw -> Normals`
6. `Draw -> Silhouette`
7. `Triangulate`
8. `Select -> Closest Vertex`
9. `Select -> Closest Edge`
10. `Select -> Closest Face`
11. `Select -> Clear`

4. Order in which to do the TP

I would recommend doing all the steps of this TP in the following order:

1. `myMesh::readFile(std::string filename)`. After this, you should see the Mesh drawn on the screen in black. It is in black because normals are needed for coloring, which have not yet been computed. The following menu items should be working (or made to work):
   - `Draw -> Mesh`
   - `Draw -> Wireframe`
   - `Draw -> Vertices`
2. `myFace::computeNormal()`, then `myVertex::computeNormal()`, and then `myMesh::computeNormals()`. After these functions, you should see the colored mesh. Then the following menu items should be made to work:
   - `Draw -> Vertex-shading/Face-shading`
   - `Draw -> Normals`
   - `Draw -> Silhouette` (you will have to write the code in the `display` function.)
3. `myMesh::Triangulate()`. Menu items that should work:
   - `Triangulate`
4. `double myPoint3D::dist(myPoint3D p1)`, `double myPoint3D::dist(myPoint3D *p1, myPoint3D *p2)`, `double operator*(myVector3D & v1)`. Then you should make the following menu items work:
   - `Select -> Closest Vertex`
   - `Select -> Closest Edge`
   - `Select -> Closest Face`
   - `Select -> Clear`