Assignment #2

CS 200, Fall 2015

Due Tuesday, September 15

Mesh interface class

The file `Mesh.h` contains the following class interface (base class with pure virtual functions)

```
struct Mesh {
  struct Face {
    int index1, index2, index3;
    Face(int i=-1, int j=-1, int k=-1) : index1(i), index2(j), index3(k) {}
  };

  struct Edge {
    int index1, index2;
    Edge(int i=-1, int j=-1) : index1(i), index2(j) {}
  };

  virtual ~Mesh(void) {}
  virtual int VertexCount(void) = 0;
  virtual Point GetVertex(int i) = 0;
  virtual Vector Dimensions(void) = 0;
  virtual Point Center(void) = 0;
  virtual int FaceCount(void) = 0;
  virtual Face GetFace(int i) = 0;
  virtual int EdgeCount(void) = 0;
  virtual Edge GetEdge(int i) = 0;
};
```

(the file `Affine.h` from the first assignment is included). Actual triangular meshes are created by deriving (publicly) from this class and implementing the pure virtual functions, which are described below.

Interface Details

`~Mesh()` — (destructor) called when the mesh is destroyed. Unless your mesh makes use of dynamically allocated memory, you need not implement this function.

`VertexCount()` — returns the number of vertices in the vertex array of the mesh.

`GetVertex(i)` — returns vertex i from the vertex array of the mesh. The vertices are given in object coordinates. You may assume that i is a valid index into this list; i.e., between 0 and one less than the value returned by `VertexCount`.

`Dimensions()` — returns the vector $\langle \Delta x, \Delta y \rangle$ that gives the dimensions of the (tight) axis–aligned bounding box that contains the mesh.

`Center` — returns the center (C_x, C_y) of the axis–aligned bounding box of the object. Note that any vertex point (x, y, z) of the mesh satisfies

$$C_x - \tfrac{1}{2}\Delta x \le x \le C_x + \tfrac{1}{2}\Delta x \quad \text{and} \quad C_y - \tfrac{1}{2}\Delta y \le y \le C_y + \tfrac{1}{2}\Delta y$$

`FaceCount()` — returns the number of triangular faces in the face list of the mesh.

`GetFace(i)` — returns face i from the face list of the mesh. You may assume that i is a valid index into this list.

`EdgeCount()` – returns the number of edges in the edge list of the mesh.

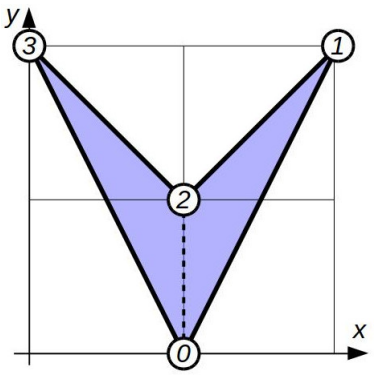
`GetEdge(i)` — returns edge i from the edge list of the mesh. You may assume that i is a valid index into this list.

An Example Mesh

As an example, suppose that our object is the quadrilateral shown in blue below. In the figure, the dashed line between vertices 0 and 2 indicates how the quadrilateral is to be triangulated. We would subclass the `Mesh` class as follows.

```
class QuadMesh : public Mesh {
  public:

    int VertexCount(void);
    Point GetVertex(int i);
    Vector Dimensions(void);
    Point Center(void);
    int FaceCount(void);
    Face GetFace(int i);
    int EdgeCount(void);
    Edge GetEdge(int i);
  private:
    static const Point vertices[4];
    static const Face faces[2];
    static const Edge edges[4];
};
```



The array `vertices` gives the vertex list of the triangular mesh that describes our object, and would be defined as:

```
const Point QuadMesh::vertices[4]
  = { Point(1,0), Point(2,2), Point(1,1),  Point(0,2) };
```

(as indicated in the above figure). Similarly, the arrays `faces` and `edges` give, respectively, the face and edge lists of the mesh, and would be defined as

```
const Mesh::Face QuadMesh::faces[2]
  = { Face(0,1,2), Face(0,2,3) };

const Mesh::Edge QuadMesh::edges[4]
  = { Edge(0,1), Edge(1,2), Edge(2,3), Edge(3,0) };
```

The functions `VertexCount`, `FaceCount`, and `EdgeCount` return the lengths of each of the above arrays. For instance, the function `VertexCount` would simply return the value 4. The functions `GetVertex`, `GetFace`, and `GetEdge`, return the desired vertex, face, and edge from the respective list. E.g., we would define the function `GetFace` as

```
Mesh::Face QuadMesh::GetFace(int i) {
  return faces[i];
}
```

Finally, the functions `Dimensions` and `Center` return the information about the axis–aligned bounding box of the object. In our case, the bounding box of the object, as indicated in the above figure, is the square with width and height 2 centered at the point $(1, 1)$. Thus the function `Dimensions` would both return the vector $\langle 2, 2 \rangle$, while the function `Center` would return the point $(1, 1)$.

Programming Task #1

Create your own triangular mesh. The only requirement is that the mesh that you design should be nontrivial: not a regular polygon, and should have at least 4 triangles. Bonus points will be awarded for artistic merit!

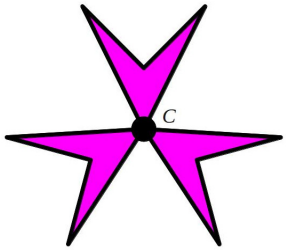
I will provide you with the files `QuadMesh.h` and `QuadMesh.cpp` which give the full declaration and implementation of the `QuadMesh` class. Feel free to use this code as a basis for your own mesh code.

Your submission for this part of the assignment will consist of two files: (1) the interface file `MyMesh.h`, and (2) the implementation file `MyMesh.cpp`. You may only include the header files `Mesh.h`, `MyMesh.h`, and `Affine.h`. Your derived class must be named `MyMesh`, and should work without modification with the test driver `MyMeshTest.cpp` that I will provide.

Programming Task #2

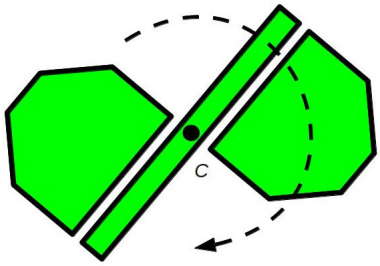
You now have two geometric figures modeled by triangular meshes, namely `QuadMesh` and your own `MyMesh`. You are to write an OpenGL/Glut program to create a window and graphically manipulate these figures. I have uploaded an executable of my version of the program (which uses my own triangular mesh, the DigiPen logo) for reference — your program should perform similar actions. Here are the requirements of this programming task.

- When the user clicks the left mouse button in the window, your program should display three copies of the `QuadMesh` figure.



Each copy is a rotation by 120° of each of the other two copies. The point C is the location of the mouse click. You are allowed to choose the overall size of the displayed figure, but it should be small enough to fit in the window. You are also free to choose the solid color of the figure

- Your program should rotate your `MyMesh` figure clockwise at a constant rate about the point where the user clicks the mouse; i.e., the point C described above.



Again, you may choose the size of the figure displayed on the screen (you may choose the color, as well), although it should fit in the window. The rotation rate should be reasonable, say taking between 1 and 10 seconds to make one rotation.

- OpenGL/Glut may only be used for window creation and management, for user–interaction, and for drawing line segments and triangles: you may not use its matrix and vertex transformation functionality. For drawing line segments and triangles, you may only use the `GL_LINES` and `GL_TRIANGLES` constructs.

For this part of the programming assignment, you should submit a single source file named `cs200_h2.cpp`. You may include the `Affine.h`, `glut.h`, `MyMesh.h`, `QuadMesh.h` header files, as well as any *standard* C++ header file.