Assignment #2

CS 200, Fall 2017

Due Wednesday, September 20

Mesh interface class

I will provide you with the file `Mesh.h`, which contains the following interface class (base class with pure virtual functions) declaration

```
struct Mesh {
  struct Face {
    unsigned index1, index2, index3;
    Face(unsigned i=0, unsigned j=0, unsigned k=0)
        : index1(i), index2(j), index3(k) {}
  };

  struct Edge {
    unsigned index1, index2;
    Edge(unsigned i=0, unsigned j=0)
        : index1(i), index2(j) {}
  };

  virtual ~Mesh(void) {}
  virtual int vertexCount(void) = 0;
  virtual const Point* vertexArray(void) = 0;
  virtual Vector dimensions(void) = 0;
  virtual Point center(void) = 0;
  virtual int faceCount(void) = 0;
  virtual const Face* faceArray(void) = 0;
  virtual int edgeCount(void) = 0;
  virtual const Edge* edgeArray(void) = 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.

`vertexArray()` — returns a pointer to the vertex array of the mesh. The vertices are given in object coordinates.

`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

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

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

`faceArray()` — returns a pointer to the face list of the mesh.

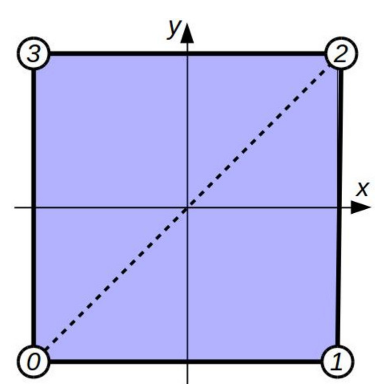
`edgeCount()` – returns the number of edges in the edge list of the mesh.

`edgeArray()` — returns a pointer to the edge list of the mesh.

An Example Mesh

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

```
class SquareMesh : public Mesh {
  public:
    int vertexCount(void);
    const Point* vertexArray(void);
    Vector dimensions(void);
    Point center(void);
    int faceCount(void);
    const Face* faceArray(void);
    int edgeCount(void);
    const Edge* edgeArray(void);
  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 SquareMesh::vertices[4]
  = { Point(-1,-1), Point(1,-1), Point(1,1),  Point(-1,1) };
```

(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 SquareMesh::faces[2]
  = { Face(0,1,2),  Face(0,2,3) };

const Mesh::Edge SquareMesh::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 `vertexArray`, `faceArray`, and `edgeArray`, return pointers to the desired array. E.g., we would define the function `faceArray` as

```
const Mesh::Face* SquareMesh::faceArray(void) {
  return faces;
}
```

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 is the square itself, which has width and height 2 centered at the point $(0,0)$. Thus the function `Dimensions` would both return the vector $\langle 2,2\rangle$, while the function `Center` would return the point $(0,0)$.

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 `SquareMesh.h` and `SquareMesh.cpp` which give the full declaration and implementation of the `SquareMesh` 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`, as well as the standard C++ header file `vector`. Your derived class <u>must</u> be named `MyMesh`, and should work without modification with the test driver `MyMeshTest.cpp` that I will provide.

Programming Task #2

I will provide you with a header file named `Render.h`, which gives the interface to a simple 2D rendering class. You are to implement this interface using the code from the `cs200opengl_demo.cpp` program discussed in class. The (public and private) interface is:

```
class Render {
  public:
    Render(void);
    ~Render(void);
    void clearFrame(const Hcoord& c);
    void setModelToWorld(const Affine &M);
    void loadMesh(Mesh &m);
    void unloadMesh(void);
    void displayEdges(const Hcoord& c);
    void displayFaces(const Hcoord& c);
  private:
    GLint ucolor,
          utransform,
          aposition;
    GLuint program,
           vertex_buffer,
           edge_buffer,
           face_buffer;
    int mesh_edge_count,
        mesh_face_count;
};
```

`Render()` — (constructor) creates/initializes a 2D rendering object. Specifically, the constructor will need to perform several initialization tasks. (1) Compile the fragment and vertex shaders. You are required to use the fragment shader

```
#version 130
uniform vec3 color;
out vec4 frag_color;
void main(void) {
  frag_color = vec4(color,1);
}
```

and the vertex shader

```
#version 130
attribute vec3 position;
uniform mat3 transform;
void main() {
```

```
    vec3 v = transform * position;
    gl_Position = vec4(v.x,v.y,0,1);
  }
```

You may not modify the code for either shader. (2) Link the compiled shaders into a shader program, and set the value of the private datum `program`. If either the fragment or vertex shader fails to compile, or if the shader program fails to link, the standard `runtime_error` exception should be thrown (do **not** print any text messages to the standard output here or in any of the member functions). (3) Set the values of the private data members `ucolor`, `utransform`, and `aposition` by getting the locations of the shader uniform variables `color` and `transform`, and the attribute variable `position`. (4) Enable the use of the `position` variable.

`~Render()` — (destructor) destroys the 2D rendering object. You should delete all resources allocated in the constructor.

`clearFrame(c)` — clears the frame buffer with the color c. Here we are using a floating point RGB color model, with all values in the range $[0, 1]$.

`setModelToWorld(M)` — sets the value of `transform` in the vertex shader to M. The transformation M specifies how the vertices of the current mesh should map to normalized device coordinates.

`loadMesh(m)` — uploads the mesh data (both the vertex array, the edge list, and the face list) to the GPU. The mesh m becomes the current mesh.

`unloadMesh()` — removes the previously loaded mesh from GPU memory.

`displayEdges(c)` — draws the boundary lines of the currently loaded triangular mesh to the frame buffer using the solid color c.

`displayFaces(c)` — draws the faces of the currently loaded triangular mesh to the frame buffer using the solid color c.

Remark. OpenGL architecture is modeled on a state machine. For example, the call `glBindBuffer` is used set a state: the current GPU buffer to use. A subsequent call to `glBufferData` transfers data from the CPU onto into the currently selected GPU buffer. A graphics application may use multiple shader programs. The call `glUseProgram` is used to select the a shader program to use, and a call such as `glUniform3f` sets the value of a uniform variable of the *currently selected shader program.* When implementing the member functions of the Render class, be aware that a different shader program than the one created in the constructor may currently be selected.

Your submission for this part of the assignment consists of a single source file, which should be named `Render.cpp`. In addition to `Render.h`, you may include only the `stdexcept` standard header file. Note that `Render.h` includes the header files `GL/glew.h`, `GL/gl.h`, `Mesh.h`, and `Affine.h`.