Programming Assignment #2

CS 250, Summer 2020

Due Friday, May 22

3D Triangular Mesh Interface

The following gives the contents of the file `Mesh.h`, which declares an interface class for the 3D triangular meshes that we will use in this course. Actual meshes will be derived from this class.

```
namespace cs250 {
  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) const = 0;
    virtual const glm::vec4* vertexArray(void) const = 0;
    virtual glm::vec4 dimensions(void) const = 0;
    virtual glm::vec4 center(void) const = 0;
    virtual int faceCount(void) const = 0;
    virtual const Face* faceArray(void) const = 0;
    virtual int edgeCount(void) const = 0;
    virtual const Edge* edgeArray(void) const = 0;
  };

  struct NormalMesh : Mesh {
    virtual const glm::vec4* normalArray(void) const = 0;
  };

  NormalMesh* createFlatShadedMesh(const Mesh &m);
}
```

The `Face` nested class is used to represent the triangular faces of the mesh. The values of `index1`, `index2`, and `index3` give the indices of the triangle's vertices in the mesh's

vertex table. Similarly, the `Edge` nested class is used to represent the edges of the mesh. The values of `index1` and `index2` specify the indices of the edge's endpoints in the vertex table. A (nonabstract) subclass of the `Mesh` class or `NormalMesh` class should implement the functions in the interface, which are detailed below.

`~Mesh()` — (destructor). If a derived class does not make use of dynamically allocated memory, an explicit destructor need not be provided.

`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 $\langle \Delta x, \Delta y, \Delta z \rangle$ that gives the dimensions of the (tight) axis–aligned bounding box in object space that contains the mesh.

`center()` — returns the center (C_x, C_y, C_z) of the axis–aligned bounding box in object space that contains the mesh. That is, vertices of the mesh have coordinates (x, y, z) where

$$C_x - \tfrac{1}{2}\Delta x \le x \le C_x + \tfrac{1}{2}\Delta x, C_y - \tfrac{1}{2}\Delta y \le y \le C_y + \tfrac{1}{2}\Delta y, C_z - \tfrac{1}{2}\Delta z \le z \le C_z + \tfrac{1}{2}\Delta z$$

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

`faceArray()` — returns a pointer to the face array of the mesh. Faces should be counter-clockwise oriented. See the *Orientation convention* description below.

`edgeCount()` — returns the number of edges in the mesh.

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

`normalArray()` — returns a pointer to an array of outwardly pointing surface normals for the mesh. The normal array has the same number of elements as the vertex array: the vertex at index i is assigned the normal at the same index i. The normal vectors are assumed to be nonzero, but not assumed to be unit.

`createFlatShadedMesh(m)` — returns a pointer to a subclass of the `NormalMesh` class that represents the same geometric figure as the passed–in mesh m, but with surface normals that guarantee that each individual face of the mesh is rendered with constant intensity (flat shading). The function caller is responsible for destroying the returned object. E.g.,

```
NormalMesh *nm = createFlatShadedMesh(CubeMesh());
// ... use mesh ...
delete nm;
```

See the programming task #2 description for more details on what this function does.

Orientation convention. The triangular faces of the mesh m should be oriented in a counter-clockwise fashion. That is if i is a valid index into the face array, then the vertices V_1, V_2, V_3 defined by

```
const Mesh::Face &F = m.faceArray()[i];
Point V1 = m.vertexArray()[F.index1],
      V2 = m.vertexArray()[F.index2],
      V3 = m.vertexArray()[F.index3];
```

are such that the vector $(V_2 - V_1) \times (V_3 - V_1)$ is an outwardly facing surface normal for the face F of the mesh.

Programming Task #1

I will provide you with the header file `CubeMesh.h`, which declares the following class for a triangular mesh of the standard cube:

```
namespace cs250 {
  class CubeMesh : public Mesh {
    public:
      int vertexCount(void) const override;
      const glm::vec4* vertexArray(void) const override;
      glm::vec4 dimensions(void) const override;
      glm::vec4 center(void) const override;
      int faceCount(void) const override;
      const Face* faceArray(void) const override;
      int edgeCount(void) const override;
      const Edge* edgeArray(void) const override;
    private:
      static const glm::vec4 vertices[8];
      static const Face faces[12];
      static const Edge edges[12];
  };
}
```

(the `Mesh.h` header file has been included). You are to implement implement the member functions and initialize the static constant data members. Remember that the standard cube is the axis–aligned cube with sides of length 2, and centered at the origin.

Your submission for this part of the assignment should consist of a single file named `CubeMesh.cpp`. You may include only the `CubeMesh.h` and `Affine.h` header files (note that the `Mesh.h` header file will automatically be included as a result).

Programming Task #2

You are to implement the `createFlatShadedMesh` function declared in the `Mesh.h` header file. To do this, you will first need to declare a subclass of the `NormalMesh` class. With each call to `createFlatShadedMesh`, with passed–in mesh $\mathcal{M}$, you will need to create an object $\mathcal{N}$ from your derived class, that contains the following mesh data. The number of faces of the derived mesh should be the same as the passed–in mesh. However, mesh $\mathcal{N}$ should have separate vertices for each face. In fact, if f is the face count of $\mathcal{M}$ (and also that of n), then $\mathcal{N}$ will have exactly $3f$ vertices, regardless of the number of vertices of $\mathcal{M}$. Note that the vertices of $\mathcal{N}$ will necessarily duplicate some of the vertices of $\mathcal{M}$. Finally, for each triangular face of the derived mesh $\mathcal{N}$, the normal vectors for the vertices of that face should all be the same, namely the outwardly pointing normal vector for that face. Here are some more details.

Derived mesh

The `createFlatShadedMesh` function will return a pointer to an object derived from the `NormalMesh` interface class. You can name this class as you see fit. You can also declare the class as you see fit, although at a minimum it must declared all of the functions in the `NormalMesh` interface. As a suggestion, you might declare

```
struct NMesh : cs250::NormalMesh {
  int vertexCount(void) const override;
  const glm::vec4* vertexArray(void) const override;
  glm::vec4 dimensions(void) const override;
  glm::vec4 center(void) const override;
  int faceCount(void) const override;
  const Face* faceArray(void) const override;
  int edgeCount(void) const override;
  const Edge* edgeArray(void) const override;
  const glm::vec4* normalArray(void) const override;

  std::vector<glm::vec4> vertices;
  std::vector<glm::vec4> normals;
  std::vector<Mesh::Face> faces;
  glm::vec4 _dimensions,
            _center;
};
```

Again, you may give the subclass a different name, and you may wish to have different member data field names and types. The above suggestion is the simplest one that I can think of. Note that the above declaration will be put in your *implementation file,* as you are not allowed to modify the `Mesh.h` header file. In particular, the name of the derived class will be hidden from the function caller.

In response to a call to `createFlatShadedMesh`, you will create an object of your derived class

```
NMesh *np = new NMesh();
```

fill in the data fields (`vertices`, `normals`, et cetera), and return a pointer to the new object:

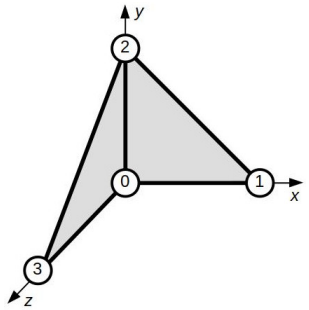
```
return np;
```

Faces, vertices, and normals

To construct the flat–shaded mesh data, bear in mind that we will create a separate triangle for each triangular face of the original mesh, ignoring how those faces are connected geometrically. To illustrate the process, consider the triangular mesh with vertex and face arrays

index	0	1	2	3
vertex value V_{in}	$(0,0,0)$	$(1,0,0)$	$(0,1,0)$	$(0,0,1)$

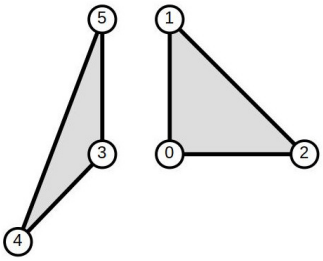
index	0	1
face value F_{in}	$\langle 0,2,1\rangle$	$\langle 0,3,2\rangle$



The corresponding flat–shaded mesh also has two faces. However, it has a total of $3 \cdot 2 = 6$ vertices, with two duplicated vertices (the vertices labeled 0 and 2 in the above diagram). The vertex and face arrays for the flat–shaded mesh are then

index	0	1	2	3	4	5
vertex value V_{out}	$(0,0,0)$	$(0,1,0)$	$(1,0,0)$	$(0,0,0)$	$(0,0,1)$	$(0,1,0)$

index	0	1
face value F_{out}	$\langle 0,1,2\rangle$	$\langle 3,4,5\rangle$



Note that for the i–th face, if the original mesh face is $F_{\text{in}}[i] = \langle f_0, f_1, f_2\rangle$, then for the corresponding face of the flat–shaded mesh, we have the face $F_{\text{out}}[i] = \langle 3i, 3i+1, 3i+2\rangle$ with vertices

$$V_{\text{out}}[3i] = V_{\text{in}}[f_0], \quad V_{\text{out}}[3i+1] = V_{\text{in}}[f_1], \quad V_{\text{out}}[3i+2] = V_{\text{in}}[f_2]$$

For the surface normals, we can use the orientation vector of the face. That is, if $\vec{m}$ is the orientation vector for the i–th face, then we would assign $\vec{m}$ to each of the three vertices of this face.

For the edge array, you can simply use a trivial (empty) array. I.e., `edgeCount` returns 0, and the `edgeArray` returns the null pointer.

Your submission for this part of the assignment should consist of a single file named `FlatShadedMesh.cpp`. You may include the `Mesh.h` and `Affine.h` header files, as well as any standard `C++` header file.