Exercise 3: edge based data structures

Task: implement an edge based data structure to the framework, allowing computation of vertex normals. Choose any edge based data structure from the following list:

- Winged edge
- Halfedge
- Directed edge
- Corner table

Following sub-routines should be implemented (use your implementations from exercise 1 if you already have them):

- Computation of a triangle normal
- Initialization of the data structure of choice
- Evaluation of the VF and VV incidence queries, i.e. finding all triangles and vertices that are incident with a given vertex
- Computation of a vertex normal

Initialization of the edge based data structure (2 points)

This step usually consists of two sub-tasks: initializing the data structure that stores an incident edge (corner) for each vertex (for full functionality an initial edge might be also needed for each triangle). The second task is initializing the edge/corner data. These two tasks can be done simultaneously or sequentially, in a function `initDataStructure` of the `TriangleMesh` object.

Initializing the edge data structures

It is important to achieve linear complexity of this step, and therefore it is necessary to use an advanced data structure that delivers answers to key requests in constant time. The problem that is common with all the data structures is finding a neighboring triangle for a given triangle, or more precisely, finding a neighboring triangle for a given edge. In order to do that in constant time, a hashtable based data structure, such as `Dictionary<,>`, is required. The general structure of the initialization procedure is as follows:

- Initialize empty `Dictionary dict<Edge, int> = new Dictionary<Edge, int>()`
- For each triangle *t*
    - For each edge *e* = (*v1*, *v2*) in *t*
        - Check whether dictionary contains edge *e* (use the `ContainsKey(Edge e)` method of the `Dictionary`)
            - In case it does not:
                - Create opposite edge *oe* = (*v2*, *v1*)
                - Add a record (*oe*, *p*) to the dictionary (Use the `Add(Edge, int)` function), where p is the edge/corner index
            - In case it does:

- Find out the index of the neighboring triangle (use the indexer, i.e. `nt = dict[e]`
- Initialize all necessary fields that can be initialized based on the knowledge that *t* is incident with *nt* across the edge *e*
- Optionally, remove the record with key e from the dictionary (use the `Remove(Edge)` function)

Initializing the incident edge structure

Following linear procedure can be used:

- For each edge data structure *e* (winged edge, halfedge, directed edge), involving vertices *v1*, *v2*
  - Set initial[v1] = e
  - Set initial[v2] = e

Data structure for edge

This data structure is required for indexing the dictionary. It should hold indices of the two incident vertices, and implement following functions:

- Equality evaluation – override the `Equals(object other)` method, and implement equality based on the stored indices
- Hash function – override the `GetHashCode()` method and return a reasonable has code, such as *v1* + *v2**1000

Evaluation of the VF incidence query (3 points)

The implementation depends on the chosen data structure, however in general, it consists of following steps:

- Requesting an initial edge (or corner) incident with the given vertex
- Extracting the index of the incident element from the data structure
- Determining the next incident edge (corner) structure in a given orientation (usually counterclockwise)
- Repeating around the vertex until the initial edge structure is reached

It is possible to collect the result in a dynamic data structure, such as `List<int>`, which allows adding items using the `Add(int i)` method. Finally, the data structure can be converted to an array using the `ToArray()` method.

The implementation should be defined as a function of the `TriangleMesh` data structure, accepting one argument, the index of the vertex, and returning an `int[]` of indices of incident triangles. For proper implementation, you will need an initialized internal edge based data structure.

Optional extension (1 point)

Adjust the implementation so that meshes with border are handled as well. Use index -1 for not existing neighbors. Not that following changes must be made:

- Proper initialization of the data structures, i.e. initializing all neighbors to -1, because the default initialization to 0 is a valid, yet incorrect index

- Proper implementation of the VF and VV function, which cannot rely on the full neighborhood anymore. A special case must be treated when a non-existent neighbor is found. The usual treatment is to return to the initial incident edge and perform traversal in the other direction (i.e. usually clockwise)