Exercise 2: Face based data structure

In this exercise, we will implement the indexed face list with neighbours in order to evaluate the VF query, needed (not only) by the vertex normal computation.

Initialization of the data structure (2 points)

This step usually consists of two sub-tasks: initializing the data structure that stores neighboring triangles for each triangle, and storing an incident face for each vertex. These two tasks can be done simultaneously or sequentially, in a function `initDataStructure` of the `TriangleMesh` object.

Initializing the data structures

The indices of neighboring triangles can be stored in an `array int[] neighbors` of length 3T, T being the number of triangles. The neighbors of i-th triangle will be stored at positions 3*i, 3*i+1 and 3*i+2. The triangle that lies opposite to the first vertex will be stored at position 3*i, the one opposite to second vertex at 3*i+1 and the one opposite to third vertex at 3*i+2

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 index to the array neighbors, where we are missing the information about the neighbor
      - In case it does:
        - Find out the index of the neighboring triangle (use the indexer, i.e. `nt = dict[e]/3`
        - 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 face structure

Following linear procedure can be used:

- For each triangle t = (v1, v2, v3)
    - Set initial[v1] = t
    - Set initial[v2] = t
    - Set initial[v3] = t

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 (2 points)

The VF query provides for a given vertex the indices of all its incident faces. At this point, we assume that the *indexed face list with neighbours* data structure is available, and therefore we can use the following procedure:

- Request a single initial incident face *f* and add it to the result
- Request the neighbors of the incident face
- Depending on the position of *v* in *f*, choose the next incident face in the counterclockwise fashion
- Continue in that fashion until you reach the initial neighbor.

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 function, you will need an initialized internal face-based data structure.

Extension (2 points)

The function of the presented algorithm depends on several assumptions made about the mesh. First, the mesh must be closed. This assumption can be lifted, but it requires updating the data structure (for example setting the neighbor index to -1 where needed) and also the VF function (we would have to address the situation when a border is encountered, and in that case, we would have to also add the neighbors in the opposite direction). Still, the mesh must be topologically manifold, i.e. each vertex must be only incident with a single fan of triangles. Can you come up with any way how this could be verified?