Exercise 1: Framework

Task: get acquainted with the template we will be using throughout the semester. In particular, try changing some vertex positions and triangle indices and test coloring of vertices and assigning vectors and normals to vertices.

Mesh Processing framework

During the semester, we will be implementing various algorithms in a framework that allows you to do basic interaction with a mesh, such as rendering, manipulating, displaying wireframe, displaying vertex colors and more. The framework is implemented in C#, which is a simple object oriented language that provides good performance for our purposes.

The framework consists of 4 projects:

- `Client`, represents your application. It has only one class called `Program`, which represents the main procedure being performed. Usually, it loads a mesh from a file, creates an instance of the renderer, assigns the mesh to it and starts the renderer. Other actions, such as mesh processing, will be also called here, before starting the rendering to display the result to the user.
- `Framework`, represents mainly the data structures. There are data staructures for vertices (`Point3D`), triangles (`Triangle`) and meshes (`TriangleMesh`). There is also a class for loading a mesh (`ObjLoader`) and some other supporting classes. When new functionality is required, it will mostly be implemented as new methods of some of these classes
- `SlimDXRenderer` and `SlimDXRenderSystem`, these projects represent the renderer. You will mostly not change anything in these projects, they just provide the functionality needed in order to view the results of your work.

Controlling the renderer

Left mouse	– rotate
Right mouse button	– shift
Y key	– render mode (fill, point, wire)
L key + mouse move	– move light position
C key	– toggle culling

Accessing vertex positions and triangle indices (1 point)

Have a look at the `Program.cs` source file in the Client project. A loader is instantiated and used to load a triangle mesh from a file. This mesh is the in turn passed to an instance of renderer, which is used to show it to the user. You can write your code between the two events, modifying the triangle mesh, or just investigating its properties.

Vertex coordinates can be accessed using the `TriangleMesh.points` field. It is an array of `Point3D` structures, each of which provides the `double X, Y` and `Z` fields. Similarly, triangle indices can be accessed using the `TriangleMesh.triangles` field, which is in array of

`Triangle` structures. This structure in turn provides `int V1, V2` and `V3` fields, which represent vertex indices that form a triangle. These indices can be used for indexing the points field.

For the exercise, try writing out the coordinates of the vertices that form triangle no. 100 to the console. You can use the standard `System.Console.WriteLine` method.

Assigning colors to vertices (1 point)

You can assign any color to any vertex in the triangle mesh. For the exercise, choose some interesting coloring. If you can't think of any, try mapping the X, Y and Z span of vertex positions to R, G and B values. Colors will be represented by the `ColorRGBA` structure. There are two constructors you may use:

```
new ColorRGBA(byte red, byte green, byte blue)
```

- In this case, arguments are expected in range 0-255

```
new ColorRGBA(float red, float green, float blue)
```

- In this case, arguments are expected in range 0.0-1.0

Ideally, you should use the full scale of the colors, i.e. for example map the smallest X value to zero and largest X value to 1.0 in red.

Having an array of vertex colors, assign it to the `Colors` field of the `TriangleMesh` instance. Finally, inform the rendering framework that it should use the color attribute by setting the `ShowColorAttribute` flag of the renderer instance to true. Disable lighting in the renderer to better view the colors.

Visualising per-vertex vectors (1 point)

Similarly, you can create an array of per vertex instances of Point3D, which will be interpreted as vectors originating from each vertex. Try visualizing some quantity, or simply build an array of random vectors with length equal to the number of vertices.

Having the array of vectors computed, assign them to the `Vectors` field of the `TriangleMesh` instance and notify the renderer that vectors should be displayed by setting the `ShowVectorAttribute` of the renderer instance to `true`. You can then display the vectors by pressing N in the renderer.

Computation of a triangle normal (1 point)

Having a triangle *t*, consisting of vertices *v1*, *v2*, *v3*, it is possible to compute its normal vector as

$$n = (v_3 - v_1) \times (v_2 - v_1)$$

For the evaluation of crossproduct, you can use the `CrossProduct(Point3D v2)` method implemented for `Point3D` structures. The normal should also be normalised. In order to do that, you can evaluate the length of a vector using the `Abs()` method defined for a `Point3D` structure.

The implementation should be defined as a function `Point3D TriangleNormal(int t)` of the `TriangleMesh` object, which has access to both triangles (`Triangle[] triangles`) and

vertex positions (`Point3D[] points`). The function should accept one argument, the index of the triangle, and return its normal, represented by a `Point3D` structure.

Computation of a vertex normal (1 point)

Having a vertex *v*, its normal can be estimated as average normal of all triangles incident with that vertex. The normal should be stored into the `Point3D[] normals` field of the `TriangleMesh` structure in order for the framework to use them. Think of some way to evaluate the average. Consider the complexity of your solution.