IGR202 Practical Assignment

Geometry Processing (3 x 3 hours)

Build upon the work that you have done during the Rendering assignment: you will have to add a set of Geometry Processing functionalities to your prototype, namely 1) mesh filtering and 2) mesh simplification. Refer to the previous assignment for help in compiling / executing your code.

0) General advices

To add the functionalities you will implement during this practical, you can follow these steps, which were used to add the Loop subdivision to the prototype:

- Add a function in the **Mesh** class (for ex, void **doSomething()**).
- Add a function in the **Scene** structure (**main.cpp,** for ex, void **doSomethingToTheCenterMesh()**).
  In this practical, you will change the vertex positions (for the filtering part) and the mesh connectivity – number of vertices, indices of the triangles (for the simplification part). You will probably need to update the VBOs when you do so, you can (for example) follow this function prototype:
  void **doSomethingToTheCenterMesh()** {

```
  centerMesh->doSomething(); //  centerMesh is probably rhino in your application
  centerMesh->recomputePerVertexNormals();
  centerMesh->recomputePerVertexTextureCoordinates();
  centerMesh->init();
}
```

- Bind a key callback to call your new function when pressing a key (in the **keyCallback** function, in **main.cpp**)

1) Filtering

The filtering operation can be decomposed in several steps:

- for every vertex, compute its barycenter (Eq1)
- for every vertex, compute the linear interpolation between its current position and its barycenter (Eq2)
- for every vertex, update its normal.

Eq1: $b_i = \sum_{j \in Neighb(i)} w_{ij} v_j / \sum_{j \in Neighb(i)} w_{ij}$ Eq2: $v_i = (1-s) v_i + s b_i$

Important Tips:

- Typically, it is recommended to compute the weights once and for all when loading the mesh (especially for the cotangent weights, which can be degenerate for degenerate triangles – what are the angles at the corners of a triangle whose vertices coincide??):
- You may want to store your weights in a structure like this one:
  std::vector< std::map< unsigned int , float > > vertex_vertex_weights; // w_ij in the course
  The reason for using a std::map is, that your weights are **sparse**: only neighboring vertices can influence each other.
  DO NOT ALLOCATE A NxN WEIGHT MATRIX, OR YOU WILL NOT BE ABLE TO PROCESS MESHES OF EVEN MODERATE SIZE!!!

Tasks:

1. ***Implement two variants for the weights:*** *(use key '1' to change mode, cout the mode in the console)*

- uniform weights.
- Laplacian cotangent weights.

2. ***Implement two variants for the flow:*** *(use key '2' to change mode, cout the mode in the console)*
   - flow in the direction of the barycenter (unconstrained),
   - *constrain the displacement to be aligned with the vertex normal (constrained)*
3. ***Compare the results when using a displacement factor s = {0.1 , 0.5 , 1.0}*** *(use key '3' to change mode, cout the value of s in the console)*

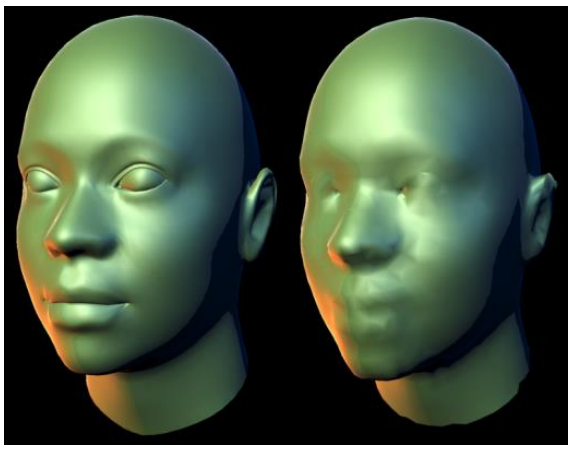
You can (for example) implement the main functions in the **Mesh** class in the following manner:

```
void Mesh::computeUniformWeights();
void Mesh::computeCotangentWeights();
void Mesh::filter( int weightsType , bool normalConstrained , float s );
```

2) Simplification

A) BASICS: Implement the grid-based simplification method. The main principle is:

- Calculate a cube C encompassing the mesh M.
  Expand it slightly to prevent numerical accuracy issues.
- Create G, a uniform grid of resolution resolution inside C.
- For each vertex v of the mesh, add its position
  and its normal to the cell representative vertex of the cell of G that contains v. Count the number of per-cell vertices.
- For each triangle t of the mesh, re-index its three vertices on the representative vertices of their respective cells if the three cells are different; eliminate the t otherwise.
- Divide the position of each representative by the number of vertices in the cell, normalize the normal vector of the representative vertex.



You can (for example) implement the main functions in the **Mesh** class in the following manner:

```
void Mesh::gridSimplification( float scaleFactor , int resolution );
```

B) EXTENDED: Implement the octree-based simplification method.

The octree-based approach follows the uniform grid – based approach in spirit.
You will need to construct a sparse octree structure, which can follow this baseline:

```
struct Octree {
        std::vector< unsigned int > pointIndices; // ONLY FOR LEAF NODES!
        Octree * children[8];
        void buildRecursive( std::vector< glm::vec3 > const & inputPoints );
        void buildBoundingBox( std::vector< glm::vec3 > const & inputPoints );
        // ...
};
```

You can (for example) implement the main functions in the **Mesh** class in the following manner:

```
void Mesh::octreeAdaptiveSimplification( int numberOfPointsInLeaves , int maxDepth );
```