Digital 3D Geometry Processing
Exercise 5 – Surface Normals, Curvature

Handout date: Mon, 14.10.2019

Submission deadline: Thu, 24.10.2019, 23:00 h

What to hand in

- From your modified source code, please upload **only** the file **viewer.cpp**, not in a subfolder, and not compressed.
- We run an automated script to compile and test your homework, so it is necessary for us to find the file **viewer.cpp** directly in your uploaded files.
- Optionally, if there are comments or problems in your code, you can submit those as comments in the Moodle submission or simply as comments in your code (`\\Like this`).
- Submit your solutions to Moodle before the submission deadline. Late submissions will receive 0 points!

Goal

The aim of the this exercise is to make yourself familiar with the provided mesh processing framework and `Surface_mesh` implementation of a halfedge data structure. The framework is a cross-platform C++ project managed with cmake. If you are building the framework using an IDE (Visual Studop for example), make sure to run the `curvature` target and not the `bin2c` which might be selected by default. Once the application is started, the mesh will be processed and an OpenGL based graphical user interface shows the resulting mesh. The simple GUI allows you to navigate the loaded mesh with the following mouse controls:

- Left-MouseMove: rotate view;
- Right-MouseMove: zoom in and out;
- Middle-MouseMove or Ctrl+Left-MouseMove: drag the mesh.

Use the buttons on the left side of the GUI to activate different rendering modes.

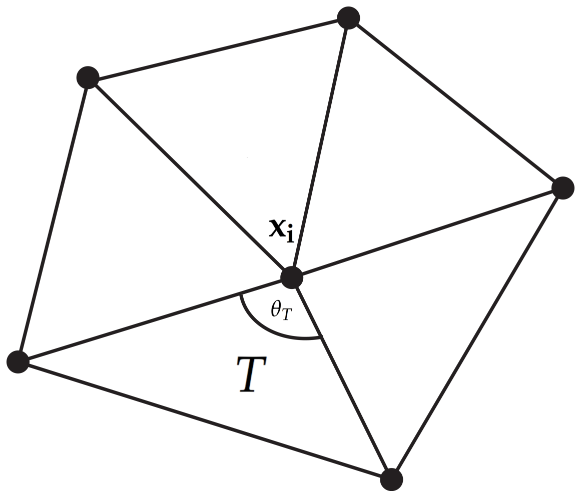


Figure 1: Incident triangle angle for normal weights.

1 Computing Vertex Normals

50 points
Normal vectors for individual triangles $T = (\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ can be computed as the normalized cross-product of two triangle edges:

$$\boldsymbol{n}(T) = \frac{(\mathbf{x_j} - \mathbf{x_i}) \times (\mathbf{x_k} - \mathbf{x_i})}{\left\|(\mathbf{x_j} - \mathbf{x_i}) \times (\mathbf{x_k} - \mathbf{x_i})\right\|}. \tag{1}$$

Computing vertex normals as spatial averages of normal vectors in a local one-ring neighborhood leads to a normalized weighted average of the (constant) normal vectors of incident triangles:

$$\boldsymbol{n}(x_i) = \frac{\sum_{T \in \mathcal{N}_1(x_i)} \alpha_T \, \boldsymbol{n}(T)}{\left\|\sum_{T \in \mathcal{N}_1(x_i)} \alpha_T \, \boldsymbol{n}(T)\right\|} \tag{2}$$

where α_T are weights. In this exercise you will compute vertex normals with three most frequently used types of weights.

- Consider the weights are constant $\alpha_T = 1$. Implement the `computeNormalsWith-ConstantWeights()` function in file `viewer.cpp`. You need to compute normals for all vertices and store them in `v_cste_weights_n` vector.
- Let the weighting be based on triangle area, i.e., $\alpha_T = |T|$. Exploit the relationship between vector cross-product and triangle area to simplify the implementation. Implement the `computeNormalsByAreaWeights()` function in file `viewer.cpp`. You need to compute normals for all vertices and store them in `v_area_weights_n` vector.
- Consider weighting by incident triangle angles $\alpha_T = \theta_T$ (see Figure 1). The involved trigonometric functions make this method computationally more expensive, but it gives superior results in general. Implement the `computeNormalsWithAngle-Weights()` function in file `viewer.cpp`. You need to compute normals for all vertices and store them in `v_angle_weights_n` vector.

Observe the difference in the rendering when the normals are computed with three different versions for weights (see Figure 2 for example).

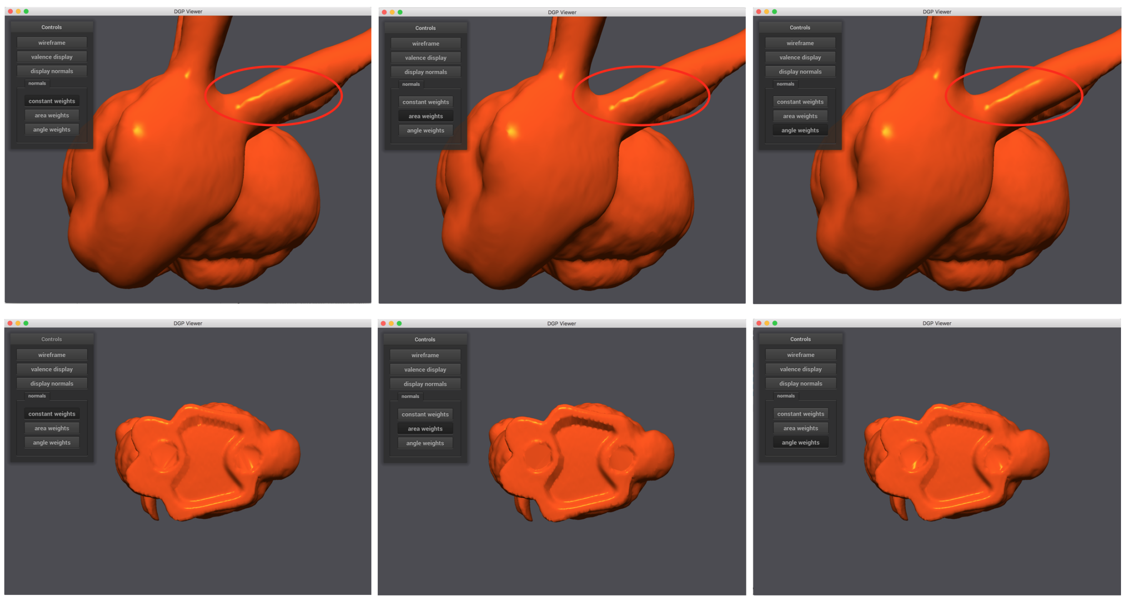


Figure 2: Difference in rendering when computing the normals with different weights.

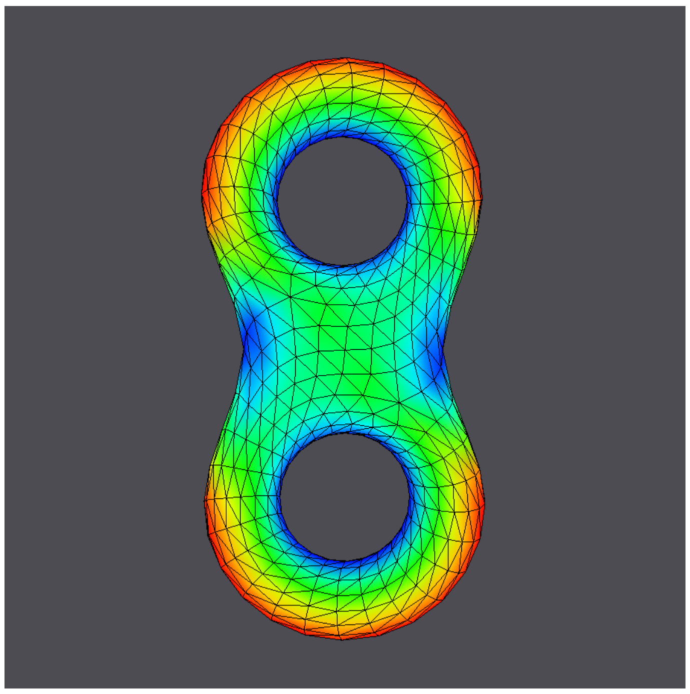


Figure 3: The uniform Laplacian operator at each vertex.

2.1 Uniform Laplace Operator

10 points
The uniform Laplace operator approximates the Laplacian of the discretized surface using the centroid of the one-ring neighborhood. For a vertex v let us denote the N neighbor vertices with v_i. Applying the uniform Laplacian to the functions holding the vertex positions amounts to computing

$$L_U(v) = \frac{1}{N}\sum_{i}^{|N|}(v_i - v)$$

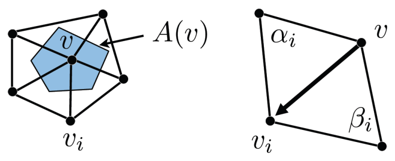
Implement the uniform Laplace operator in the function `calc_uniform_laplacian()` in the `viewer.cpp` file. Store the length of the computed vector $L_U(v)$ in vertex property `v_uniLaplace[v]`. Store the minimal Laplacian value in the `min_uniLaplace` and the maximal Laplacian value in `max_uniLaplace`. To display the per-vertex Laplacian operator click on the `Curvature -> Uniform Laplacian` button. The minimal and maximal Laplacian value will be displayed on the standard output. You should get the result similar to Figure 3.

2.2 Laplace-Beltrami Curvature

20 points
The discretization of the uniform Laplacian does not depend on vertex coordinates and therefore does not take into account the geometry of the mesh. To obtain a mean curvature approximation we need to introduce weights that depend on the geometry. We will use the *cotan*-weights introduced in the lecture, which leads to the following approximation when applying the Laplace-Beltrami operator to the functions holding the vertex coordinates:

$$L_B(v) = \frac{1}{2A}\sum_{i}^{|N|}(\cot\alpha_i + \cot\beta_i)(v_i - v)$$

See the lecture slides and the picture on the right for an explanation about this formula.
Study the `calc_weights()` function to understand how and which weights are computed. Note that here, we use the barycentric cells, so the Area A associated to each vertex is the sum of the areas of the surrounding triangles, devided by 3.



Use these weight values (`vweight` and `eweight`) to implement the mean curvature approximation using the Laplace-Beltrami operator. The length of the vector $L_B(v)$ computed via the formula above, divided by two, gives an approximation of the mean curvature. In the function `calc_mean_curvature()`, fill in the `v_curvature` property with the mean curvature approximation values. Additionally, store the minimal curvature value in `min_mean_curvature` and the maximal curvature value in `max_mean_curvature`.
To display the per-vertex mean curvature values click on the `Curvature -> Laplace-Beltrami` button. The minimal and maximal curvature value will be displayed on the standard output. You should get the result similar to Figure 4.

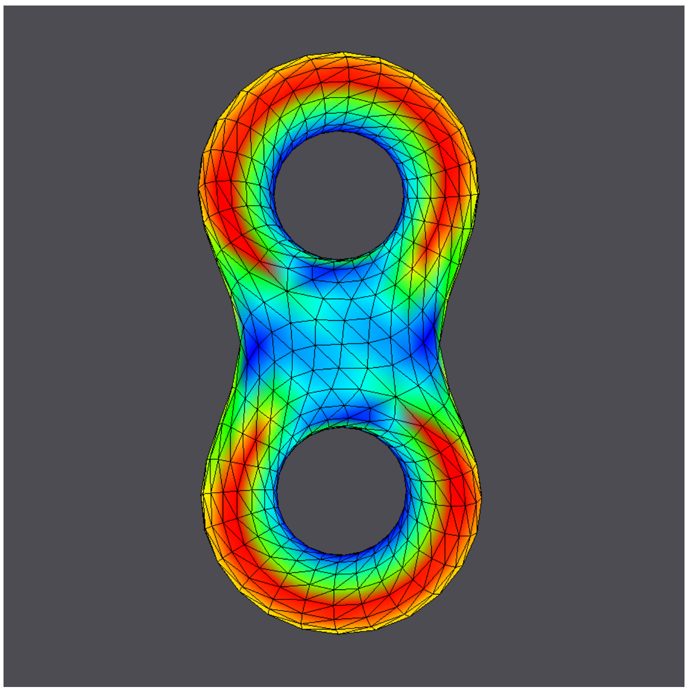
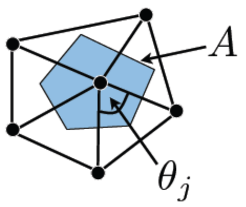


Figure 4: The Laplace-Beltrami approximation of the mean curvature at each vertex.

2.3 Gaussian Curvature

20 points
In the lecture you have been presented an easy way to approximate the Gaussian curvature on a triangle mesh. The formula uses the sum of the angles around a vertex and the same associated area which is used in the Laplace-Beltrami operator:

$$K = (2\pi - \sum_j \theta_j)/A$$



Implement the `calc_gauss_curvature()` function in the `viewer.cpp` file so that it stores the Gaussian curvature approximations in the `v_gauss_curvature` vertex property. Note that the `v_weight` property already stores $\frac{1}{2A}$ value for every vertex, so you do not need to calculate A again. Store the minimal curvature value in `min_gauss_curvature` and the maximal curvature value in `max_gauss_curvature`. For the figure eight mesh you should get a Gaussian curvature approximation like on Figure 5.

The blue color corresponds to the minimal value and the red color corresponds to the maximal value of the current mesh. Explore the curvature of different given meshes. In addition you are given a small sphere and a 10 times bigger sphere. Observe what happens with the Uniform Laplacian and the Laplace-Beltrami operator on the spheres of different sizes (*hint: check on the maximal and minimal values*).

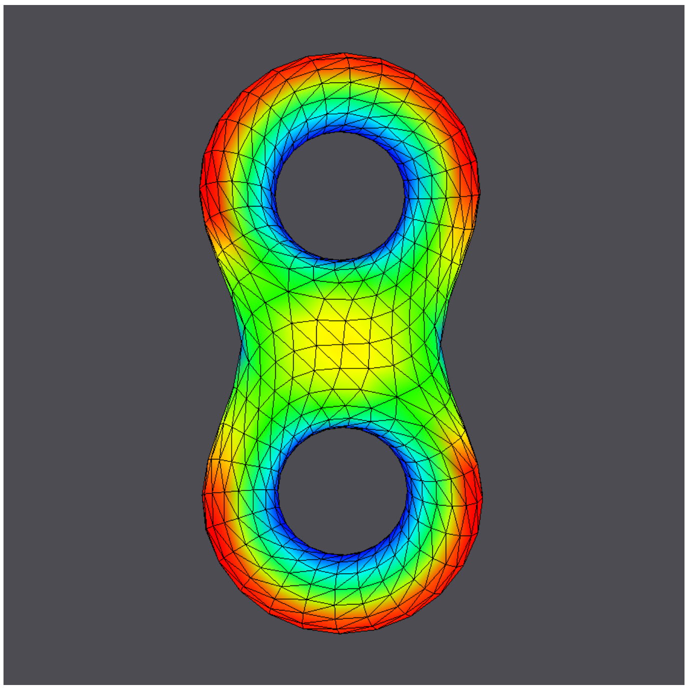


Figure 5: Approximation of the Gaussian curvature at each vertex.

Note: As for the last exercises, we will use automated testing to evaluate your code. You should make sure that you pass the tests provided by the executable `curvature_test`. Note that this only evaluates correctness for a few isolated cases. In particular we test all three normals and curvature values computed for a single vertex in the Bunny mesh, as well as minumum and maximum curvature values. Due to numerical inaccuracies during the computation, your results might vary slightly. We're going to manually check failing test cases and will only mark answers as wrong that clearly deviate from the solution.
The test relies on the fact that you changed no code, except for the functions we asked you to implement above.