Digital 3D Geometry Processing
Exercise 3 – Differential Geometry and Mesh Data Structure

Handout date: 11.10.2016

Submission deadline: 20.10.2016, 23:00 h

What to hand in

A .zip compressed file renamed to `Exercisen-GroupMemberNames.zip` where n is the number of the current exercise sheet. It should contain:

- Hand in **only** the files you changed (headers and source). It is up to you to make sure that all files that you have changed are in the zip.
- A `readme.txt` file containing a description on how you solved each exercise (use the same numbers and titles) and the encountered problems.
- Other files that are required by your `readme.txt` file. For example, if you mention some screenshot images in `readme.txt`, these images need to be submitted too.
- For the theory exercise, put your solutions in the same .zip you submit for the code. Both PDF files (generated with LaTex, Word, ...) or scanned handwritten solutions are acceptable for this exercise.
- Submit your solutions to Moodle before the submission deadline. Late submissions will receive 0 points!

1 Theory Exercise

King Archimedes wants to renovate his palace. The most striking structure is a spherical half-dome of 20m in diameter that covers the great hall. The king wants to cover this dome in a layer of pure gold. He has decided to split the work into two parts, each one covering a vertical slice of the dome of the same height (see Figure 1). For each part he hires different people and gives them 700kg of gold. The task is to cover the surface of one vertical slice with a layer of gold of 0.1mm thickness. The amount of gold that is left over is the salary for doing the job. Which slice should you pick if you want to make the most profit? Explain your answer.

How does your answer change when you have n slices instead of just two?

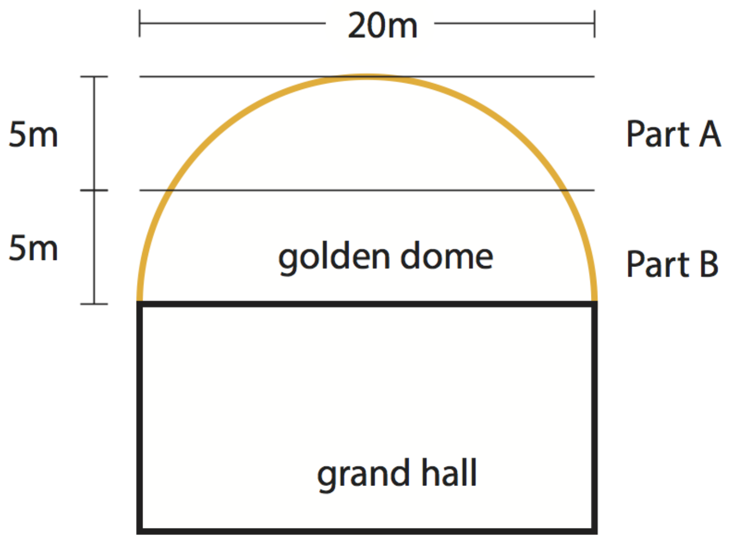

Figure 1: Sketch of King Archimedes dome.

2 Coding Exercise

The aim of the this exercise sheet 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 (Qt Creator for example), make sure to run the `surfaces` 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.

2.1 Vertex valence of a triangle mesh

The valence $v(\mathbf{x}_i)$ of a vertex $\mathbf{x}_i$ in a triangle mesh is the number of vertices in the 1-ring neighborhood $\mathcal{N}(\mathbf{x}_i)$ of $\mathbf{x}_i$, i.e., the vertices from $\mathcal{N}(\mathbf{x}_i)$ are connected with an edge to $\mathbf{x}_i$.

The application entry point (main function) is located in file `surfaces.cpp`. In this part, you need to do the following:

- Implement the `computeValence()` function in file `surfaces.cpp`. It should compute the valences of all vertices of the mesh, and store them as a vertex property `v:valence` in the vector `vertex_valence`. After implementing the `computeValence()` function, you will be able to see the coloring of vertices according to their valence, as shown in Figure 2. You can see the implementation of the color-coding in the file `viewer.h` but you don't need to edit anything there.

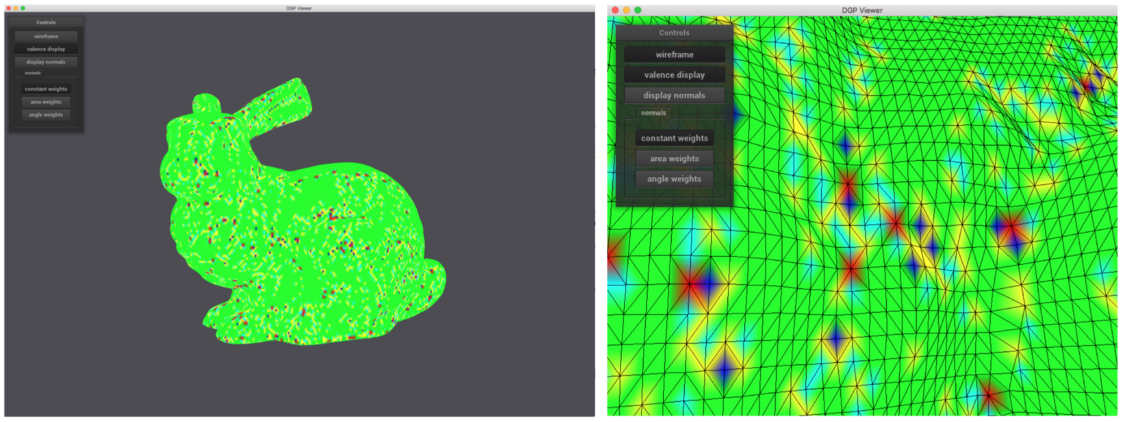

Figure 2: Color visualization of the vertex valences.

2.2 Computing Vertex Normals

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 `computeNormalsWithConstantWeights()` function in file `surfaces.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., $\alpha_T = |T|$. This method is particularly efficient to compute, since the area-weighted face normals are just the (unnormalized) cross-product of two triangle edges. Implement the `computeNormalsByAreaWeights()` function in file `surfaces.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 3). The involved trigonometric functions make this method computationally more expensive, but it gives superior results in general. Implement the `computeNormalsWithAngleWeights()` function in file `surfaces.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 4 for example).

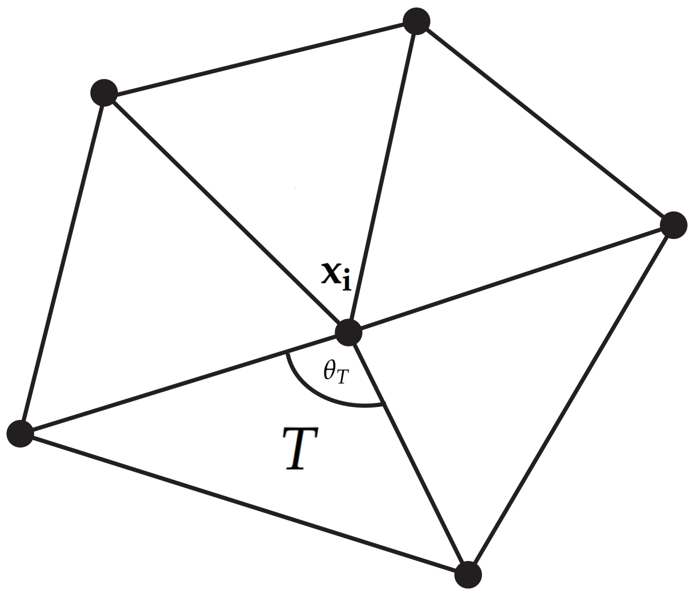

Figure 3: Incident triangle angle for normal weights.

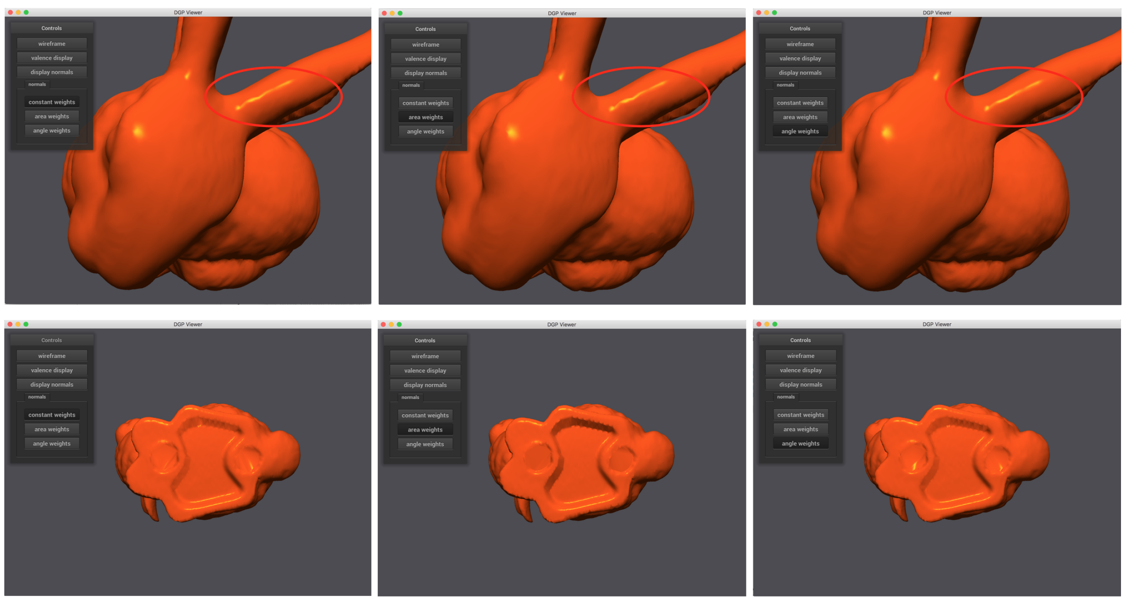

Figure 4: Difference in rendering when computing the normals with different weights.