3D Geometry Processing
Exercise 1

Fall 2013

Hand in: 3.10.2013, 16:00

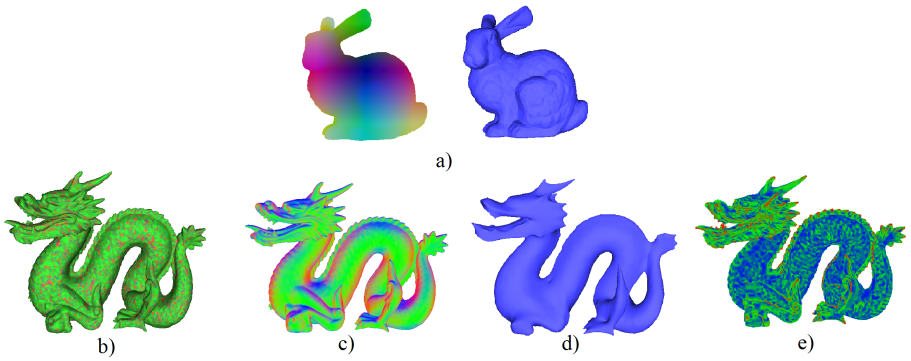

Figure 1: a) Running Example.java b) Valence visualization (green: 6, red: >6, blue: <6). c) Per vertex normals, directly interpreted as colors. d) Smoothed dragon. e) Mean curvature

1 Getting Started

We provide base code on top of which you will build various geometry processing algorithms. This week's exercise should allow you to accommodate yourself with the code provided. First of all set up your programming environment. Note that the code was developed on a windows machine and only superficially tested on Linux, there is no guarantee there won't be unexpected issues on MAC/Linux.

1. If you don't have the java JDK and Eclipse installed yet, install both.
2. Install the m2eclipse plug-in as described at `http://eclipse.org/m2e/download/`.
3. Download the exercise1.zip package provided on the Ilias exercise page and unzip it to a folder that does not lie in your eclipse workspace folder. Import the unzipped project as a Maven project. The eclipse project should be marked by an M as its dependencies are managed by Maven.
4. Right-click the project and select Maven->Update Project... . This should resolve all errors and download the referenced jar files.
5. Run Example.java. The bunnies in Fig. 1 should be displayed.

2 Getting Started: Vizualize It!

During the exercise sessions you will visualize various data structures and results. We provide code which hides the details of using OpenGL with Java while providing you the flexibility to display any data structure you want with any shader you fancy. This exercise should help you get aquainted with the code provided.

OpenGL runs small programs (called shaders) on the graphics card to render images. While geometrical data might be represented in arbitrary ways in our applications, OpenGL needs the data to be carefully laid out in arrays to pass it to a shader. In our framework, the task of getting from some representation to the OpenGL presentation is handled by wrapper-classes that extend the class GLDisplayable. Such a wrapper has to lay out the data in arrays and provide hints on how the data is to be interpreted by the shaders, such as to which shader variables the data should be bound.

In the base code, one such wrapper is implemented for the wireframe data structure. Your task is to implement a wrapper for the provided half-edge structure.

1. Study the classes assignment1.Example and glWrapper.GLWireframeMesh, as well as the shader files used.

2. The classes HalfedgeStructure, HalfEdge, Vertex and Face implement a half-edge structure; the method HalfedgeStructure.init(WireFrameMesh) can be used to build a halfedge structure out of a wireframe mesh. Write a class GLHalfedgeStructure, which extends the class GLDisplayable and casts the half-edge structure to a GL-compatible format in a similar way as the GLWireframeMesh class does for wireframe meshes. With your wrapper, display the half-edge structure using the shaders used in the Example class.

(2 points)

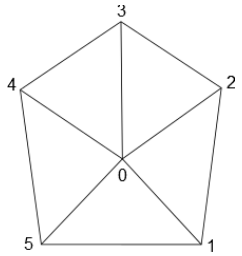

Figure 2: The mesh stored in oneNeighborhood.obj

3 Half-Edge Structure

The half-edge structure in the base code does not yet provide any methods that truly exploit the adjacency information stored in the half-edge structure. In this exercise you will implement iterators over neighborhoods and then implement three geometry processing algorithms on top of the half-edge structure.

1. Implement the following methods; you can look at the already implemented method Face.iteratorFV() to get started.

   (a) Face.iteratorFE(Face): return an iterator which iterates over the edges adjacent to the given Face.

   (b) Vertex.iteratorVE(Vertex), Vertex.iteratorVV(Vertex), Vertex.iteratorVF(Vertex): return an iterator, which iterates over the adjacent edges, vertices or faces of the given vertex respectively.

   (c) Demonstrate your methods by creating a half-edge structure from the mesh "oneNeighborhood.obj" (Fig. 2), and printing the adjacent faces/vertices/half-edges of selected vertices and faces by using your iterators.

   (2 Points)

2. Compute and visualize the valence of every vertex. The valence of a vertex is the number of incident edges. One-dimensional numerical data associated to vertices can be stored using the class HEData1D. Adapt your GLHalfedgeStructure class such that the data stored in a HEData1D class is sent to OpenGL and write shaders similar to the trimesh_flat shaders, that accept and display the additional data. **(1 Point)**

3. Implement a basic smoothing algorithm. In every iteration, iterate over all vertices and compute for each vertex a new vertex position by averaging the positions of the vertices in the one ring neighborhood. At the end of every iteration update all vertex positions with the newly computed positions. Three dimensional data associated to vertices can be stored using the class HEData3D. **(1 Point)**

4. Implement a basic algorithm to compute a normal vector for every vertex. At every vertex sum up the normals of the adjacent faces, weighted by the incident angle, and normalize the result. Find and implement a way to visualize the computed normals. **(1 Point)**

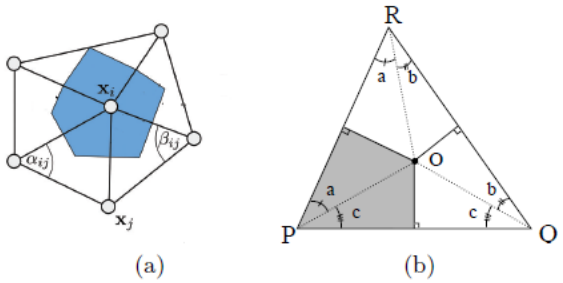

Figure 3: (a) A mixed area (blue) and other quantities for curvature computations. (b) Quantities for mixed area computation on non-obtuse faces

4 Differential Geometry

1. Compute and visualize the curvature at vertex positions using the cotangent laplacian with mixed area weights. The per vertex curvature is given by

$$K(v_i) = \frac{1}{2\mathcal{A}_{mixed}} \sum_{\mathcal{N}_1(i)} (cot(\alpha_{ij}) + cot(\beta_{ij}))(x_i - x_j)$$

where the sum goes over the one ring neighborhood $\mathcal{N}_1(i)$ of the vertex i. The mixed area $\mathcal{A}_{mixed}$ is computed by the following sum over the adjacent faces of the vertex i:

$$\mathcal{A}_{mixed} = \sum_{\mathcal{F}_1(i)} area_i(F)$$

$$area_i(F) = \begin{cases} \frac{1}{8}(||PR||^2 cot\angle Q + ||PQ||^2 cot\angle R) & \text{if F non-obtuse} \\ area(F)/2 & \text{if } F \text{ obtuse at vertex i} \\ area(F)/4 & \text{else.} \end{cases}$$

Instead of visualizing the curvature by mapping it linearly to some colors, visualize $log(1+curvature/C)$ with some appropriate C. This enhances the visibility of variations of the curvature when it is close to zero. (3 Points)

Prepare your Demo, Hand in your Code

To get credit for your hard work you will need to do the following:

1. Put all the classes you altered in a zip file and commit the zip file via ilias, before the deadline.

2. Prepare a short demo for each of the exercises; you will have to demonstrate that your code works after the exercise hour, either on a CGG Pool machine or on your own laptop. If you want to work on a pool machine contact the assistant to get an account.