Assignment 1: Hello World

Handout date: 01/03/2019
Submission date: 15/03/2019, 08:00

In this exercise you will

- Familiarize yourself with `libigl` and the provided mesh viewer.
- Get acquainted with some basic mesh programming by evaluating surface normals, computing mesh connectivity and isolating connected components.
- Implement a simple mesh subdivision scheme.

1. Setup

Please go to https://github.com/eth-igl/GP2019-Assignments an carefully follow the instructions.

2. First steps with libigl

The first task is to familiarize yourself with some of the basic code infrastructure provided by `libigl`.

2.1. **Eigen.** `libigl` uses the `Eigen` library for all of its matrix computations. In `libigl`, a mesh is typically represented by a set of two `Eigen` arrays, V and F. V is a float or double array (dimension #V $\times$ 3 where #V is the number of vertices) that contains the positions of the vertices of the mesh, where the i-th row of V contains the coordinates of the i-th vertex. F is an integer array (dimension #faces $\times$ 3 where #F is the number of faces) which contains the descriptions of the triangles in the mesh. The i-th row of F contains the indices of the vertices in V that form the i-th face, ordered counter-clockwise.

Check out the "Getting Started" page of `Eigen` as well as the Quick Reference page to acquaint yourselves with the basic matrix operations supported.

Note that you don't need to install Eigen manually since a reasonably up-to-date version is included as a submodule in `libigl`.

2.2. **Compiling and Running the `libigl` Tutorials.** First, build the libigl tutorials:
`cd libigl/tutorial; mkdir build; cd build; cmake ..; make -j2`.
This can take a while, so be sure to set the "-j" flag to run on as many cores as possible.

Each tutorial is named with format `XXX_TUTORIAL_NAME`, where `XXX` is the number ID of the tutorial. After the build completes, each tutorial executable can be run, e.g.: `./XXX_TUTORIAL_NAME_bin`

The source code for the corresponding tutorial is located in
`libigl/tutorial/XXX_TUTORIAL_NAME/main.cpp`

Experiment with the basic functionality of `libigl` and the included mesh viewer by running at least the first 6 tutorials and inspecting the corresponding source code.

3. Neighborhood Computations

For this task, you will use `libigl` to perform basic neighborhood computations on a mesh. Computing the neighbors of a mesh face or vertex is required for most mesh processing operations, as you will see later in the class. You need to fill in the appropriate sections (inside the keyboard callback, keys '1' to '2') of `src/main.cpp` to compute the neighborhood relations using `libigl`. In order to use a function from `libigl` (e.g. the function to compute per-face normals), you must include the relevant header file at the top of your `main.cpp` file (e.g. `#include <igl/per_face_normals.h>`) and call it later in your code (`igl::per_face_normals(V,F,FN)`).

3.1. **Vertex-to-Face Relations.** Given V and F, generate an adjacency list which contains, for each vertex, a list of faces adjacent to it. The ordering of the faces incident on a vertex does not matter. Your program should print out the vertex-to-face relations in text form when key '1' is pressed.

Relevant `libigl` *functions:* `igl::vertex_triangle_adjacency`.

3.2. **Vertex-to-Vertex Relations.** Given V and F, generate an adjacency list which contains, for each vertex, a list of vertices connected with it. Two vertices are connected if there exists an edge between them, i.e., if the two vertices appear in the same row of F. The ordering of the vertices in each list does not matter. Your program should print out the vertex-to-vertex relations in text form when key '2' is pressed.

Relevant `libigl` *functions:* `igl::adjacency_list`.

3.3. **Visualizing the Neighborhood Relations.** Check your results by comparing them to the built-in relations calculated by the mesh viewer. You can do this by clicking on the checkboxes "Show vertex labels" and "Show faces labels" in the viewer window.

Required output of this section:

- A text dump of the content of the two data structures for the provided mesh "plane.off".

4. Shading

For this task, you will experiment with the different ways of shading discrete surfaces already implemented in `libigl`. Fill in the appropriate source code sections (inside the keyboard callback, keys '3' to '5') to display the mesh with the appropriate shading.

4.1. **Per-face Shading.** The simplest shading technique is flat shading, where each polygon of an object is colored based on the angle between the polygon's surface normal and the direction of the light source, their respective colors, and the intensity of the light source. With flat shading, all vertices of a polygon are colored identically. Your program should compute the appropriate shading normals and shade the input mesh with flat shading when the key '3' is pressed.

Relevant `libigl` *functions:* `igl::per_face_normals`. Call `viewer.data.set_normals(·)` to set the shading in the viewer to use the normals you computed.

FIGURE 1. Flat Shading

4.2. **Per-vertex Shading.** Flat shading may produce visual artifacts, due to the color discontinuity between neighboring faces. Specular highlights may be rendered poorly with flat shading. When per-vertex shading is used, per-vertex normals are computed for each vertex by averaging the normals of the surrounding faces. Your program should compute the appropriate shading normals and shade the input mesh with per-vertex shading when the key '4' is pressed.

Relevant `libigl` *functions:* `igl::per_vertex_normals`. Call `viewer.data.set_normals(·)` to set the shading in the viewer to use the normals you computed.

Compare the result must be compared with the one obtained with flat shading.

FIGURE 2. Per-Vertex Shading

4.3. **Per-corner Shading.** On models with sharp feature lines, averaging the per-face normals on the feature, as done for per-vertex shading, may result in blurred rendering. It is possible to avoid this limitation and to render crisp sharp features by using per-corner normals. In this case, a different normal is assigned to each face corner; this implies that every vertex will get a (possibly different) normal for every adjacent face. A threshold parameter is used to decide when an edge belongs to a sharp feature. The threshold is applied to the angle between the two corner normals: if it is less than the threshold value, the normals must be averaged, otherwise they are kept untouched. Your program should

compute the appropriate shading normals (with a threshold of your choice) and shade the input mesh with per-vertex shading when the key '5' is pressed.

Relevant `libigl` *functions:* `igl::per_corner_normals`. Call `viewer.data.set_normals(·)` to set the shading in the viewer to use the normals you computed.

Compare the result must be compared with the one obtained with flat and per-vertex shading. Experiment with the threshold value.

FIGURE 3. Per-Corner Shading

Required output of this section:

- Screenshots of the provided meshes shaded with flat, per-vertex, and per-corner normals.

5. CONNECTED COMPONENTS

Using neighborhood connectivity, it is possible to partition a mesh into connected components, where each mesh face belongs only to a single component. Fill in the appropriate source code sections (inside the keyboard callback, key '6') to display the mesh with each face colored to indicate the component it belongs to (coloring each component distinctly). You can use the jet colormap provided with `libigl` to assign colors to the components, or you can implement your own colormap.

Relevant `libigl` *functions:* `igl::facet_components, igl::jet`. Call `viewer.data.set_colors(·)` to set the displayed colors to the per-face colors you computed.

Required output of this section:

- Screenshots of the provided meshes with each connected component colored differently.
- The number of connected components and the size of each component (measured in number of faces) for all the provided models.

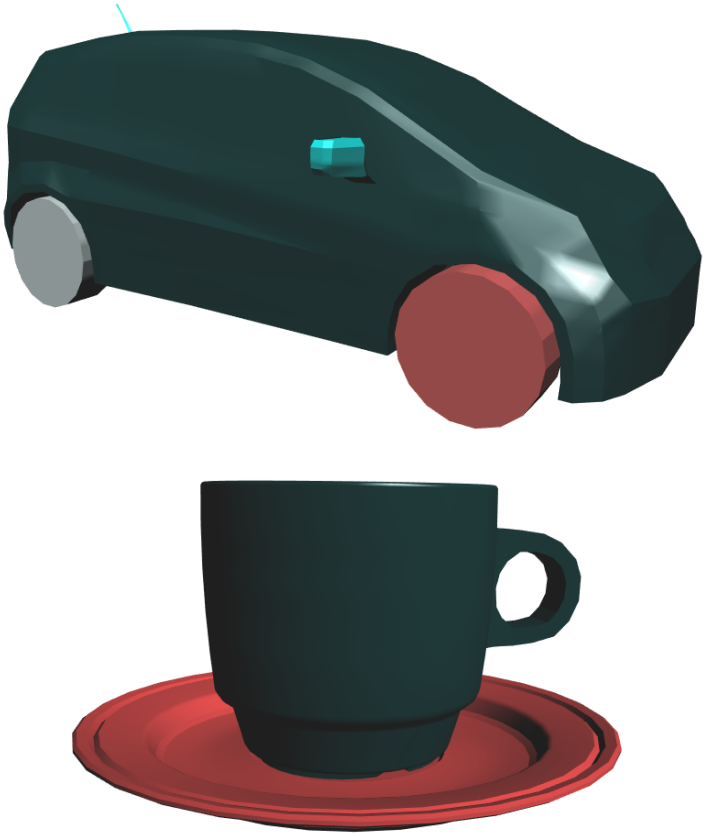

FIGURE 4. Connected components visualized by coloring each component distinctly.

6. A SIMPLE SUBDIVISION SCHEME

For this task, you will implement the subdivision scheme described in [1] (https://www.graphics.rwth-aachen.de/media/papers/sqrt31.pdf) to iteratively create finer meshes from a given coarse mesh. According to the paper, given a mesh (V,F), the $\sqrt{3}$-subdivision scheme creates a new mesh (V',F') by applying the following rules:

(1) Add a new vertex at location $\mathbf{m}_f$ for each face $f \in F$ of the original mesh. The new vertex will be located at the midpoint of the face. Append the newly created vertices $M = \{\mathbf{m}_f\}$ to V to create a new set of vertices $V'' = [V; M]$. Add three new faces for each face f in order by connecting $\mathbf{m_f}$ with edges to the original 3 vertices of the face; we call the set of this newly created faces F''. Replace the old set of faces F with F''.

(2) Move each vertex $\mathbf{v}$ of the old vertices V to a new position $\mathbf{p}$ by averaging $\mathbf{v}$ with the positions of its neighboring vertices in the *original* mesh. If $\mathbf{v}$ has valence n and its neighbors in the original mesh (V,F) are located at $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$, then the update rule is

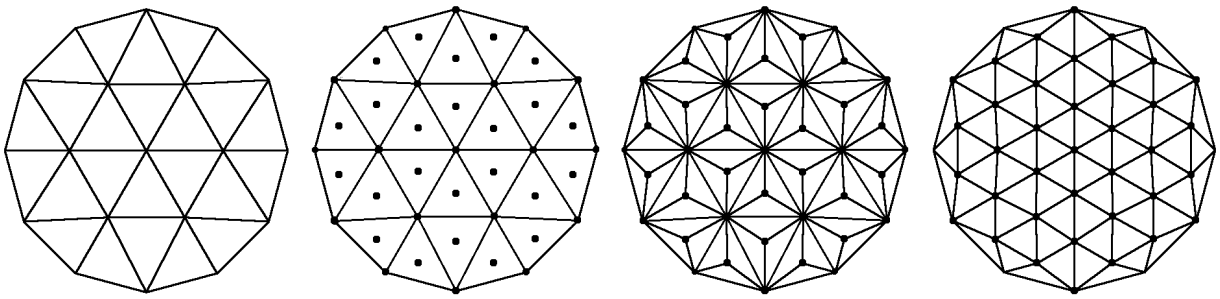

FIGURE 5. $\sqrt{3}$ Subdivision. From left to right: original mesh, added vertices at the midpoints of the faces (step 1), connecting the new points to the original mesh (step 1), flipping the original edges to obtain a new set of faces (step 3). Step 2 involves shifting the original vertices and is not shown.

$$\mathbf{p} = (1 - a_n)\mathbf{v} + \frac{a_n}{n}\sum_{i=0}^{n-1}\mathbf{v}_i$$

where $a_n = \frac{4-2\cos\left(\frac{2\pi}{n}\right)}{9}$. The vertex set of the subdivided mesh is then $V' = [P, M]$, where P is the concatenation of the new positions $\mathbf{p}$ for all vertices.

(3) Replace the F'' with a new set of faces F' such that the edges connecting the newly added points M to P (the relocated original vertices) remain but the original edges of the mesh connecting points in P to each other are flipped. See Figure 5.

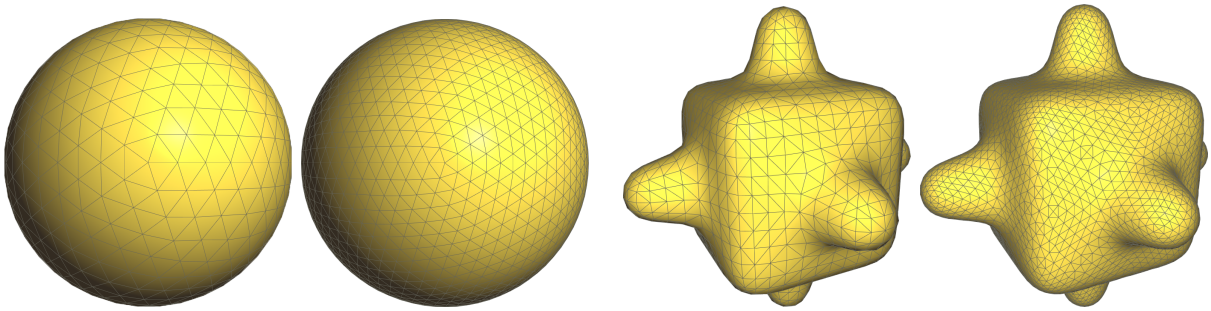

FIGURE 6. Example of one $\sqrt{3}$ subdivision step.

Fill in the appropriate source code sections (inside the keyboard callback, key '7') so that hitting key '7' subdivides the mesh once and displays it in place of the old mesh.

Relevant `libigl` *functions:* Many options possible. Some suggestions: `igl::adjacency_list`, `igl::triangle_triangle_adjacency`, `igl::edge_topology`, `igl::barycenter`. Use `viewer.data.clear()` and `viewer.data.set_mesh(·,·)` to replace the displayed mesh in the viewer.

Required output of this section:

- Screenshots of the subdivided meshes.

References

[1] Leif Kobbelt. Sqrt(3)-subdivision, 2000.