SHAPE MODELING AND GEOMETRY PROCESSING

ASSIGNMENT 1 - LIBIGL "HELLO WORLD"

In this exercise you will

- Familiarize yourself with `libigl` and the provided mesh viewer
- Get acquainted with some basic mesh programming, by computing topological relations and isolating connected components.
- Implement a simple mesh subdivision scheme.
- Implement a simple mesh editing operation.

Setup

Please go to https://github.com/HaifaGraphics/geometry-processing-assignments and carefully follow the instructions.

1. First steps with libigl

The first task of the assignment is getting familiar with some of the basic code infrastructure provided in `libigl`.

1.1. **Eigen.** `libigl` uses the Eigen library for all 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 will contain the indices of the vertices in V that form the i-th face, sorted counter-clockwise.

Have a look at the "Getting Started" page of `Eigen` as well as the Quick Reference page to acquaintain yourselves with the basic matrix operations supported.

1.2. **libigl.** Throughout this course you will use `libigl`, a geometry processing library. Have a look at the libigil tutorials. Each tutorial has a name of the form `./XXX_TUTORIAL_NAME` where `XXX` is the number ID of the tutorial. The executables for all tutorials will then be located inside the directory `TOPDIR/libigl/tutorial/build` and can be run e.g. by using: `cd TOPDIR/libigl/tutorial/build; ./XXX_TUTORIAL_NAME_bin`

The source code for the corresponding tutorial is located in

`TOPDIR/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 having a look at the corresponding source code.

2. 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. For this task, you need to fill in the appropriate sections (inside the keyboard callback, keys '1' to '2') of the source code provided in the `main.cpp` file of the provided project to compute the neighborhood relations using `libigl`. In order to use the a library function (e.g. the function to compute per-face normals), you need to 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)`).

2.1. **Vertex-to-Face relations.** Given V and F, generate an adjacency list which contains, for each vertex, the 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`.

2.2. **Vertex-to-Vertex relations.** Given V and F, generate an adjacency list which contains, for each vertex, the 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 the list does not matter. Your program should print out the vertex-to-face relations in text form when key '2' is pressed.

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

2.3. **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 textual dump of the content of the two data structures for the provided mesh "cube.off" .

3. Connected Components

Using the neighborhood connectivity, it is possible to separate a mesh into separated connected components, where each mesh face only belongs to a single component. Fill in the appropriate source code sections (inside the keyboard callback, key '3') of the project to display the mesh with faces of the various connected components colored differently for each component. For coloring the components you can use the jet colormap provided with `libigl`, 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.

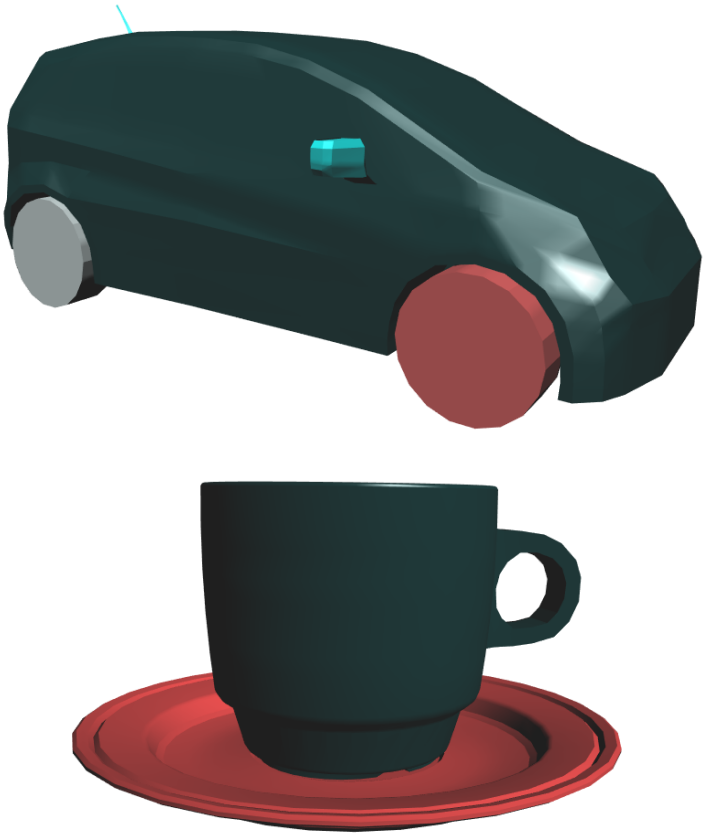

Figure 1. Connected Components of meshes

Required output of this section:

- Screenshots of the provided meshes with their connected components colored differently.

- The number of connected components and the size of every one of them (measured in number of faces) for all the provided models.

4. A simple subdivision scheme

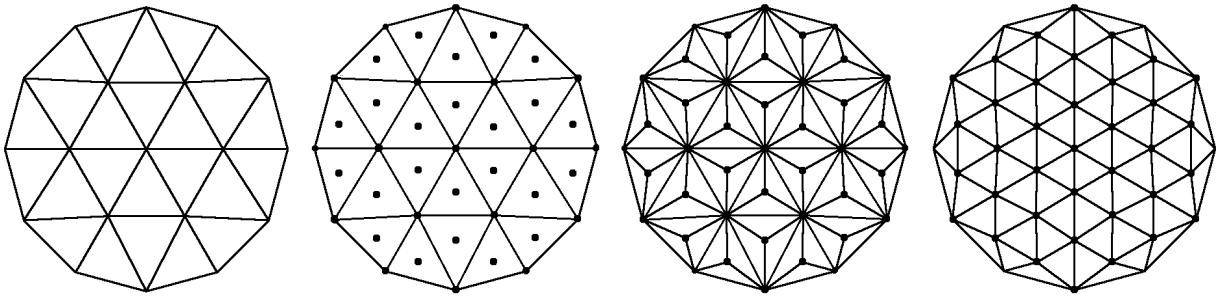

FIGURE 2. $\sqrt{3}$ Subdivision. From left to right: original mesh, added vertices at the midpoints of the faces (step 1), connecting the new points to the original mesh (step 1), flipping the original edges to obtain a new set of faces (step 3). Step 2 involves shifting the original vertices and is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

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

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

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 moved vertices corresponding to the original vertices) remain but the original edges of the mesh connecting points in M to each other are flipped. See Figure 2.

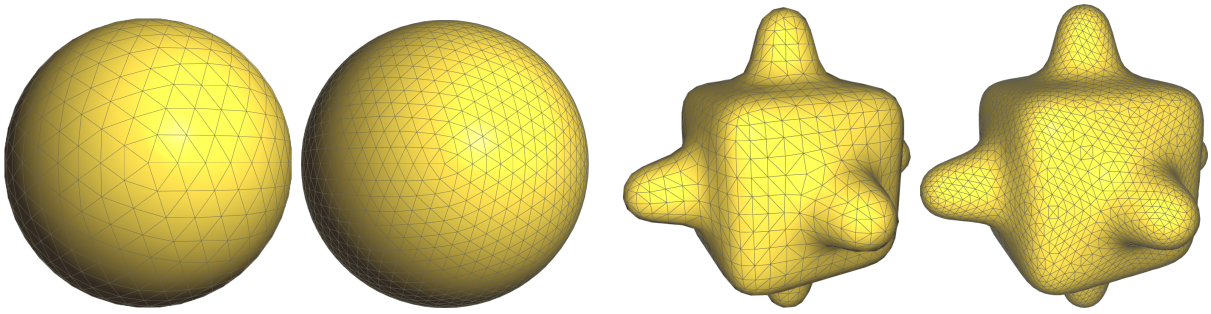

FIGURE 3. Example results of one step of $\sqrt{3}$ Subdivision.

Fill in the appropriate source code sections (inside the keyboard callback, key '4') of the project so that hitting key '4' 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.

5. MESH EXTRUDE - A SIMPLE MESH EDITING OPERATION

In this task you will implement a basic mesh editing operation. The code supplied already has a built-in support for a few operations: selecting faces, translating faces, and selecting/translating a single vertex (See Figure 5).

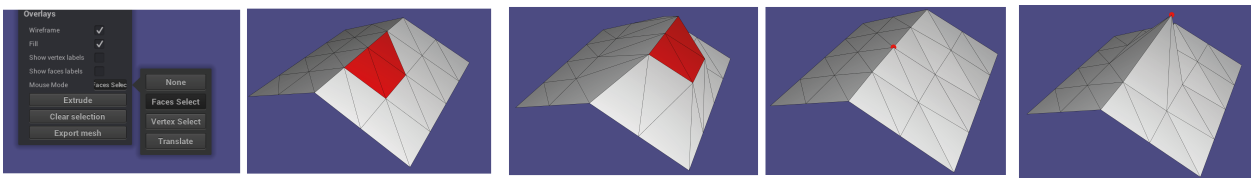

FIGURE 4. Editing operations supported in the code. From left to right: The editor GUI, face selection, face translation, vertex selection, vertex translation.

There is however a feature that is not fully implemented: triangular faces extrusion. The process is best explained in Figure 5.

Triangular faces extrusion takes a mesh with vertices V, faces F, and a set of selected faces and returns a new mesh with vertices V' and faces F' using the following steps:

(1) Verify that the set of faces are connected by edges.

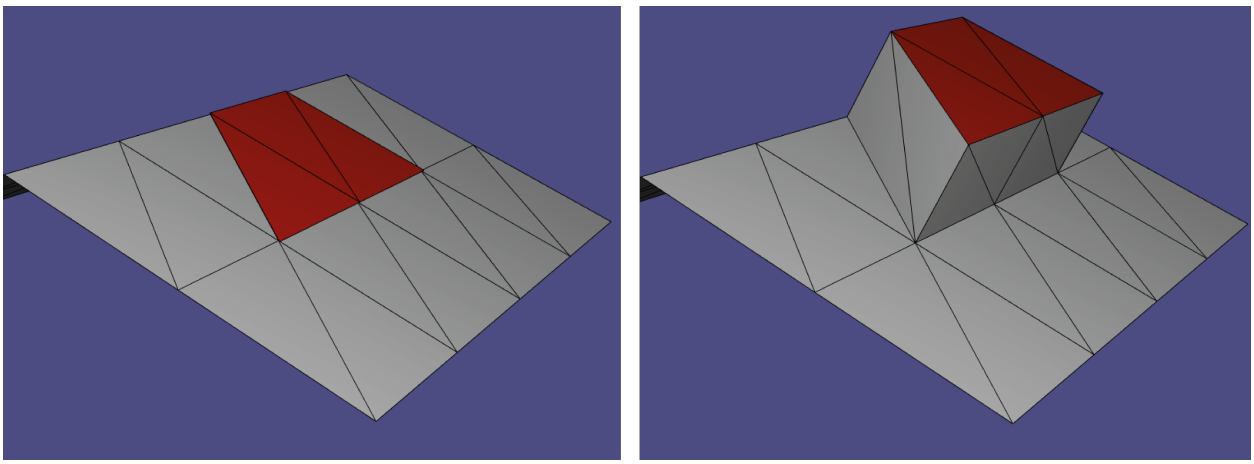

FIGURE 5. Faces extrusion. Left: The user select a set of connected faces. Right: The faces are extruded by their boundary. The boundary vertices/edges are the vertices/edges separating the red selected faces and the unselected grey faces. The boundary vertices are duplicated, the old vertices are slightly translated upwards, and 2 new faces are created for every boundary edge of the selected faces. These faces are connecting the old boundary vertices with the new duplicated ones.

(2) Create a new set of vertices V', which contains the old set of vertices as well as new vertices that are just duplicates of the boundary of the selected faces.
(3) Compute T: The direction of the average normal of the selected faces.
(4) Offset the old vertices in direction T.
(5) Create a new set of faces F'. F' contain all the faces F contained, as well as new faces: 2 for each boundary edge surrounding the selected faces which connect the old boundary vertices into the new duplicated ones. Furthermore, faces touching the boundary vertices that were not selected should be updated to contain the new duplicated vertices, instead of the old ones that were offseted by T in step (4).
(6) Make sure the new set of V',F' is a manifold.

The code provided to you already offers a partial implementation of `extrude()` with steps (1)-(4). Your task is to implement tasks (5)-(6).

Relevant `libigl` *functions:* `igl::vertex_triangle_adjacency`, `igl::is_edge_manifold`, `igl::is_vertex_manifold`. Also consider using `std::set` to store unique lists of indices, and `std::set_difference` to get the list of faces that should be updated.

To call the extrude routine, first select a few faces (connected by edges) and then press the 'extrude' button in the gui.
Lastly, here are a few tips:

(1) First test your extrude implentation on a single selected face.

(2) Make sure your created faces are correctly oriented. If they are not, they will appear dark in the viewer as the normal will be flipped. Given a face defined with vertices v_1, v_2, v_3, one can flip its orientation by changing the order of its vertices to v_1, v_3, v_2.
(3) After implementing the extrude function, please uncomment the lines: V = Vout; F = Fout; for your changes to take affect.

Required output of this section:

- Screenshots of two different extrusion operation on the 'cube.off' model. Make sure to include a 'before' and 'after' extrude picture, and that at least one of the extrude operations will be on 3 faces or more.
- A mesh you designed starting from 'cube.off' and using the GUI supplied editing operations, as well as your implemented subdivision and extrude operations. You can save your result using the 'Export mesh' button. Save it under 'design.off', and also take a screenshot of it from a few angles.

References

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