Assignment 1: Hello World

In this exercise you will

- Familiarize yourself with `igl` and the provided viewer `meshplot`.
- 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. First steps with `igl`

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

1.1. **Data structures.** In `igl`, a mesh is typically represented with a set of two `numpy` arrays, V and F. V is a float or double array (dimension #V x 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 x 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.

Note that you may need to install `numpy` manually with `conda install numpy`.

1.2. **Installing `igl` and Running the Tutorials.** Follow the steps in the General Instructions to install `igl`. You can find an extensive set of tutorials at tutorial repository. Many tutorials require understanding subjects in geometry processing that we will see in class.

2. Neighborhood Computations

For this task, you will use `igl` to perform basic neighborhood computations on a mesh. Computing the neighbors of a mesh face or vertex is required for most mesh processing operations. In order to use any function from `igl` (e.g. the function to compute per-face normals), you must just import the package with `import igl`.

2.1. **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.

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

2.2. **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 adjacent 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.

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

2.3. **Visualizing the Neighborhood Relations.** Check your results by comparing them to raw data inside F.

Required output of this section:

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

3. Shading

For this task, you will experiment with the different ways of shading discrete surfaces already implemented in `igl`. Display the mesh with the appropriate shading.

Use `meshplot.plot(v,f, n=n)` to set the shading in the viewer to use the normals `n` you computed. Note that notation `n=n` means that the internal parameter `n` of `meshplot` (left end side) must be set at the value of the variable `n` that you have computed (right end side). The latter shall be different for each of the different shading options listed below.

Note: `meshplot` supports only per vertex normals. To visualize the different types of shading (in particular, the flat and the per-corner shading) you will need to "explode" the mesh. That is, separate all faces and duplicate vertices, as in a polygon soup data structure. For instance, if the mesh has faces f=[[0, 1, 2], [1, 3, 2]] (with vertices 1 and 2 shared among the two faces), exploded_f=[[0, 1, 2], [3, 4, 5]] with the vertices 3 and 5 being a copy of vertices 1 and 2. Note that, on the contrary, `igl` will give you per-vertex, per-face, and per-vertex-per-face normals, depending on the selected option. So you will need to compute and store an index mapping from the input mesh to the exploded one, and apply the proper mapping each time. **Warning:** The solution involves just few lines of code, but it is a non-trivial exercise of manipulation of the data structures. Be careful on the way you explode the mesh and maintain the mapping between the original and the exploded meshes.

Figure 1. Flat Shading

3.1. **Flat 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.

Relevant `igl` *functions:* `igl.per_face_normals`.

3.2. **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.

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

FIGURE 2. Per-Vertex Shading

3.3. **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.

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

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

Required output of this section:

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

4. 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

FIGURE 3. Per-Corner Shading

to display the mesh with each face colored to indicate the component it belongs to (coloring each component distinctly).

Relevant `igl` *functions:* `igl.face_components`. Call `meshplot.plot(v,f,c=c)`, where `c` contains the computed labels to display colors to the per-face colors you computed.

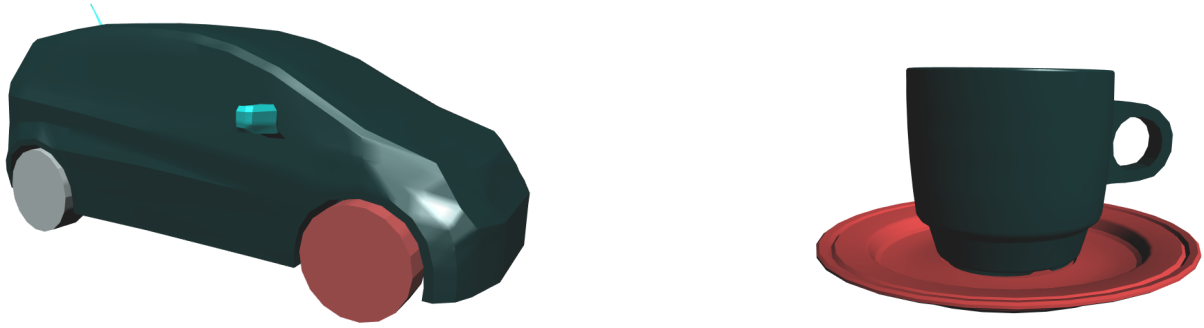

FIGURE 4. Connected components visualized by coloring each component distinctly.

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.

5. 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:

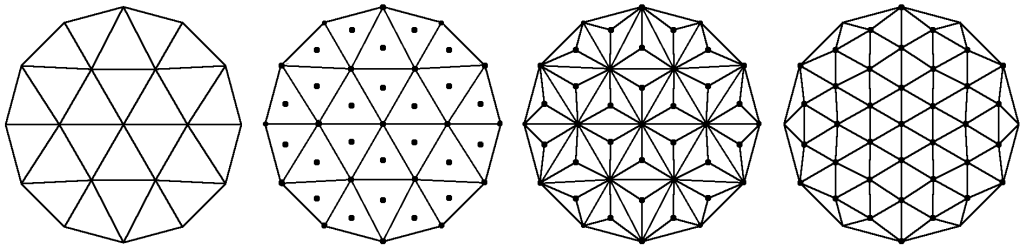

FIGURE 5. $\sqrt{3}$ Subdivision. From left to right: original mesh; added vertices at the midpoints of the faces (step 1); connecting the new points to the original mesh (step 1); flipping the original edges to obtain a new set of faces (step 3). Step 2 involves shifting the original vertices and is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

(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 barycenter 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 (old) vertices. Note that the new vertices are not relocated, they remain at the place set in the previous step.

(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 pairs of points in M are flipped. See Figure 5. **Hint:** You may build F' as follows. Scan the triangles of F'', for each triangle t: in case t is a boundary triangle, just copy t to F'; in case the index of t is larger than the index of its neighbor opposite to the "new" vertex, just skip it; otherwise, fetch its neighbor t' and the (index of) vertex v' of t' opposite to t, then use the (indexes of) vertices of t and v' to generate the two triangles obtained by swapping the edge common to t and t' and add such triangles to F'. There are other options, e.g., by generating the edges of the mesh and flipping each edge that connects two old vertices and is not on the boundary.

Fill in the appropriate source code sections (inside the keyboard so that the mesh is subdivided once and displayed.

Relevant `igl` *functions:* Many options possible. Some suggestions: `adjacency_list`, `edge_topology`, `triangle_triangle_adjacency`, `barycenter`.

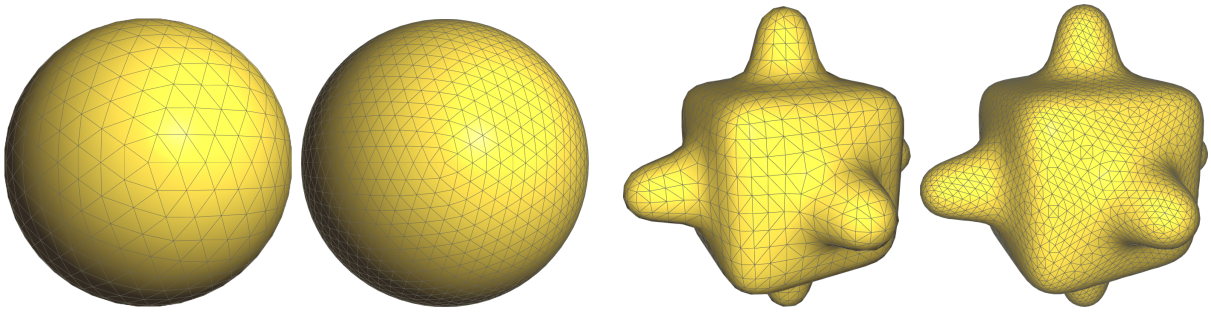

FIGURE 6. Example of one $\sqrt{3}$ subdivision step.

Required output of this section:

- Screenshots of the subdivided meshes.

REFERENCES

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