3 Polygon Mesh Processing

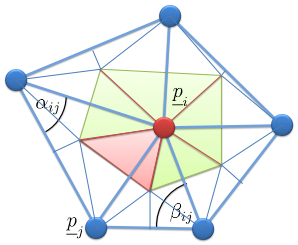
This exercise focuses on processing surfaces represented as polygon meshes.

All exercises and source code is property of the Chair of Computer Graphics and Visualization. Publication and distribution is prohibited.

3.1 Theory

3.1.1 Voronoi Area (2 Points)

The following figure shows the Voronoi region (green) of vertex $\underline{p}_i$. The corners of the region are the centers of circumcircles of the individual triangles. These can be constructed as the intersection of the perpendicular bisectors of the sides.



(a) What happens to the corner of a Voronoi region if the corresponding triangle becomes obtuse? (**0.5 Points**)

(b) The area of the Voronoi region can be calculated with the following formula:

$$A_{\text{voronoi}}(i) = \frac{1}{8} \sum_{j \in N_1(i)} (\cot \alpha_{ij} + \cot \beta_{ij}) \|\underline{p}_i - \underline{p}_j\|^2$$

Derive this formula. (**1.5 Points**)

(Hint: Start by finding a formula for the red area from vertex positions $\underline{p}_i$, $\underline{p}_j$ and the angles α_{ij} und β_{ij}. What causes the cotangent terms?)

3.1.2 Memory requirements of a halfedge data structure (3 Points)

Assume an indexed halfedge data structure that stores for each vertex 16 bytes (position + halfedge index), for each halfedge 16 bytes (next, inverse, origin, face index), and for each edge and face 4 bytes (halfedge index). We know that for triangle meshes without boundary the following ratio of number of vertices, number of edges, and number of faces holds: $(v : e : f) = 1 : 3 : 2$ or $2e = 3f$ and $v - e + f = \chi$. Assuming that χ is usually small (≈ 0) ist, we can estimate $f \approx 2v$ and $e \approx 3v$.

(a) Derive a formula for estimating the memory size (in bytes) for a triangle mesh without boundary **depending on the number of vertices**, using the previously described halfedge datastructure. (**1 Point**)

(b) The ratio $2e = 3f$ is only valid for triangle meshes. What is the ratio $v : e : f$ for quad meshes without boundary? Explain your results. (**1.5 Points**)

(c) Derive an according formula for estimating the memory size of quad meshes without boundary depending on the number of vertices. (**0.5 Points**)

3.2 Practical Part (10 Points)

The practical part uses OpenMesh's halfedge data structure to represent polygon meshes. The data structure can be accessed via the type `HEMesh` (which is actually a typedef of `OpenMesh::PolyMesh_ArrayKernelT<>`). Make yourself familiar with the data structure by looking at the online documentation and the code examples in the appendix.

3.2.1 Mesh Generation (1 Point)

Implement the creation of a unit cube in function `CreateCube` in file `Primitives.cpp`. The file already contains other primitives, which you can use as an orientation. You can choose several primitives in the GUI using the according buttons.

3.2.2 Surface Area and Volume Calculation (2 Points)

After solving the following tasks, you can test the functionality using the respective GUI buttons. These will display a message box with the calculated measures. You could test your implementation with a unit cube (surface area 6, volume 1).

(a) Implement the calculation of surface area of triangle meshes in function `ComputeSurfaceArea` in file `SurfaceArea.cpp`. (**0.5 Points**)

(b) Extend the method such that it can be used for arbitrary polygon meshes. (**1 Point**)

(c) Implement the calculation of the enclosed volume of a closed triangle mesh in function `ComputeVolume` in file `Volume.cpp`. (**0.5 Points**)

3.2.3 Connected Component Analysis (2 Points)

Implement function `ExtractShells` in file `ShellExtraction.cpp`. The function returns the number of connected components. Furthermore, is assigns a shell id to every face. The shell idea is an integer in the range $[0, n)$, where n is the number of connected components. Use the following code to set the shell id for a face f (represented as a face handle):

```
m.property(perFaceShellIndex, f) = shell id;
```

You can solve this task either with region growing or using a union-find data structure (available under `util/UnionFind.h`). In contrast to the lecture, the region growing approach should operate on faces instead of vertices.

Hint: You can check if a face has already been visited via its `shell_id` attribute. This attribute is initialized to -1 for each face at the beginning of the function. (**2 Points**)

You can test your implementation by loading `foot_bones.obj` from the `data` folder. Clicking the "Extract Shells" button, a message box should tell you that the model consists of 26 shells. Furthermore, the shells are colored according to their shell id.

3.2.4 Smoothing of Vertex Positions (1 Point)

In this task, you are going to implement iterative smoothing using the uniform Laplacian and explicit integration. The update rule for vertex positions is:

$$\underline{p}_{\text{new}} \leftarrow \underline{p}_{\text{old}} + \lambda \Delta \underline{p}_{\text{old}}$$

Implement the smoothing functionality in `SmoothUniformLaplacian` in file `Smoothing.cpp`. The function takes λ and the number of iterations as parameters (you can set them via the GUI). Be aware that by applying the update rule to a vertex, you are changing the one-rings of its immediate neighbors (which might not have been updated yet). For testing purposes, you can load a model and add noise with the corresponding GUI button. If your implementation is correct, the button `Laplacian Smoothing` should allow you to de-noise the model.

3.2.5 Triangle Stripification (4 Points)

Implement the extraction of triangle strips using the greedy algorithm presented in the lecture in function `ExtractTriStrips` in `Stripification.cpp`. The function returns – similar to `ExtractShells` – the number of strips, and it should assign a strip id to each face. The function takes the number of trials as parameter, which you can set via the GUI.

Make sure that a strip always starts with parity 0 to make the tessellation consistent with OpenGL rendering.

Hint: Take a look at the data structure `sample_set` in file `sample_set.h`. It allows to efficiently add, remove, and randomly sample elements (using a uniform distribution).

The button `Extract Triangle Strips` starts the calculation for the loaded model. To test your implementation, you can use a torus. Using a sufficiently large number of trials (e.g., 20), your implementation should find 20 triangle strips of length 40. The torus should be colored in a regular pattern.

Submission

Zip all source code files in the folder `exercise2` into a zip archive and upload it to Opal.

3.2.6 Bonus Tasks (max. +5 Points)

- Add a function `is_edge_flip_ok` that checks for a given edge `e` if it is valid to flip an edge using topological and geometric criteria. Implement a visualization in wireframe mode that highlights edges that must not be flipped. (**4 Points**)
- Implement a mesh optimizer that aims at maximizing the number of regular vertices (inner vertices with valence 6, border vertices with valence 4) using flip operations. (**3 Points**)
- Implement an OpenGL visualization that shows the halfedge links (next, previous, inverse), e.g., using arrows in a clear way. (**3 Points**)
- Implement a method that calculates the dual of an input mesh. The dual mesh contains a vertex for every face in the primal mesh. To vertices in the dual mesh are connected if the corresponding faces in the primal mesh are adjacent. (**2 Points**)
- Implement smoothing using the cotangent discretization of the Laplacian instead of the uniform Laplacian. (**2 Punkte**)

A OpenMesh Halfedge Data Structure

You can find the online documentation of OpenMesh under

`http://www.openmesh.org/media/Documentations/OpenMesh-7.0-Documentation`

The following pages present a few code examples.

All operations on the halfedge structure are represented using element handles. Possible handle types are:

- `OpenMesh::VertexHandle`
- `OpenMesh::HalfedgeHandle`
- `OpenMesh::EdgeHandle`
- `OpenMesh::FaceHandle`

Sometimes, it is necessary to use element indices instead of element handles. In this case, the `.idx()` method of the handles can be used. To convert an index back to a handle, you can use, e.g., `mesh.vertex_handle(idx)`.

Use the following methods of `HEMesh` to get linked elements:

- `mesh.halfedge_handle(vertexHandle)`: an outgoing halfedge of a given vertex
- `mesh.halfedge_handle(faceHandle)`: a halfedge incident to a facet
- `mesh.from_vertex_handle(halfedgeHandle)`: the source vertex of a halfedge
- `mesh.to_vertex_handle(halfedgeHandle)`: the target vertex of a halfedge
- `mesh.next_halfedge_handle(halfedgeHandle)`: next halfedge

- `mesh.prev_halfedge_handle(halfedgeHandle)`: previous halfedge
- `mesh.opposite_halfedge_handle(halfedgeHandle)`: inverse halfedge
- ...

A.1 Access to Vertex Attributes

- Vertex position: `mesh.point(vertexHandle)`
- Texture coordinate: `mesh.set_texcoord2D(vertexHandle, texCoord)`
- Custom attribute: `mesh.property(propertyHandle, elementHandle)`

The attributes are usually represented as OpenMesh vectors. Use the utility function from `util/OpenMeshUtils.h` to convert between OpenMesh vectors and Eigen vectors.

A.2 Iterators

All iterators are available as simple iterators (`begin` und `end`) and as C++ 11 range objects. Using the range objects is quite simple. E.g., the following code iterates all neighboring vertices of a vertex `v`:

```
for(OpenMesh::VertexHandle neighbor : mesh.vv_range(v))
    //do something with neighbor
    ...
```

The iterators are named according to the following scheme: The first letter denotes the reference element (in this case, we want to iterate elements corresponding to a vertex); the second part denotes the iterated elements (in this case, we iterate vertices, i.e. `vv`). There are many more iterators, e.g. `voh_range` (outgoing halfedges of a vertex), `fv_range` (vertices of a face), or `ff_range` (neighbor faces of a face).