Technical tips: Step 2

1. Report structure (continued)

Besides what Tech Tips 1 said, there are a few extra elements that are important when reporting:

- Formulas: Basically every mathematical processing you do on your data should be 'captured' by a formula. This is needed for exact understanding and replication. Without a formula, a phrase in the text can be interpreted in many ways, leading to partial and/or incorrect understanding potential. Secondly, formulas force one to be exact and also introduce and use notations for all the involved quantities (areas, vertices, angles, lengths, etc). To give just an example: The pose normalization can be intuitively described using PCA as a 'rotation'. But how to implement that rotation exactly? One needs to give the respective formula.
- Writing: Since you work in a group, *proofreading* should not be optional. For example, in a group of two, person X writes, then Y proofreads and corrects; roles are reversed in the next iteration.

2. Remeshing

There are several (subtle and less subtle) misconceptions about what remeshing aims to do and how to do it:

(1) Remeshing aims to create densely-sampled meshes: Partially true. Of course, a very coarsely sampled mesh is not good, as it contains too few surface samples to allow a good computation of shape descriptors. However, a very densely sampled mesh (say, 100K vertices or more) will cause serious computational bottlenecks. Moreover, for a fair testing of your algorithm, you should not use meshes having a widely varying sampling density. Meaning, you don't need only refining coarse meshes, you also need decimating too fine ones.

(2) Remeshing using average values: Another frequent error is to consider the average vertex count V_{avg} (or face count, or face area, whichever sampling-related property you want) and to use remeshing to bring every actual vertex count V in the database close to this average. This is wrong, since it supposes that V_{avg} is the *target value* that you want to achieve. This is not necessarily so! Consider for instance a shape database where there are a lot of simple shapes having, say, 10..20 vertices. Then, V_{avg} is around 15. So, remeshing will force all shapes to have 15 vertices on average – which is clearly too low a value for feature extraction. The correct way is to remesh so as to bring the average vertex count close to a user-given *target value* V_{target} (say, $V_{target} = 1000$). Or in simple words:

You don't remesh to bring sampling resolution close to the database's average. You remesh for bringing the average of the sampling resolution close to a desired target value.

3. Order of normalization steps

You must do translation, size, pose, orientation, and mesh-resolution normalization (5 steps). In which order should you do them? Does it matter?

It *partially* does: Any transformation T1 should be done *after* a transformation T2 if T2 can affect properties that T1 also affects.

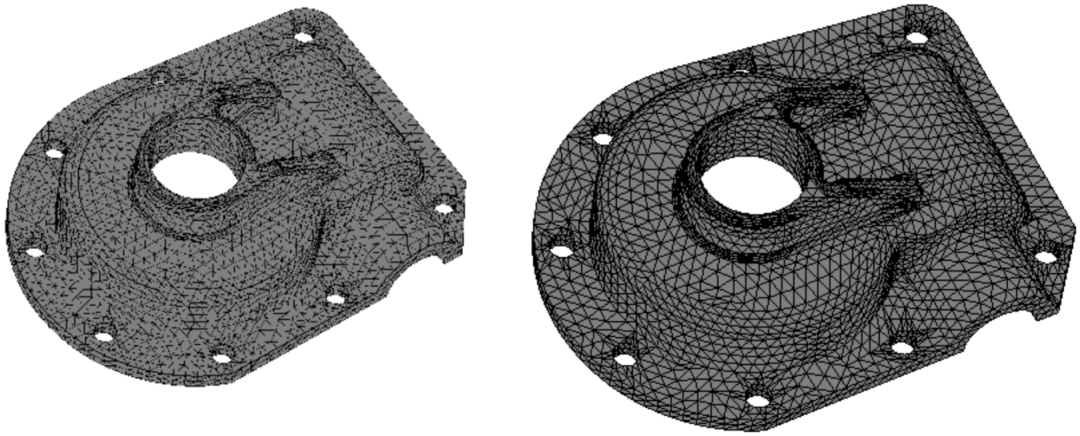
Proposed order:

- **translation:** do first, since then, computing the pose normalisation is much more easily (in the shape covariance matrix description, you anyway need to subtract the barycentre from the coordinates);
- **scaling:** it doesn't matter much when you do that, it only changes size and none of the other transformations does change size;
- **remeshing:** you could do it at any point, since it doesn't change any of the size/position/orientation. There's no perfect place for it: If you do it at the end, for the very large meshes you will decimate, you pass them through all other transformations (compute a lot) only to decimate them next. Conversely: if you remesh at the beginning, for the subdivided meshes you'll do more scaling/orientation/translation computations than if you did the subdivision at the end of the pipeline. I still advise to do remeshing at the end, since then you can debug your other transformations more easily, simply by comparing what they give with the original shapes (as they appear in the database).

4. Rendering wireframe-atop-solid

Several people implement just a solid (shaded) rendering mode and a wireframe mode. This is not optimal. The solid mode is, of course, good for actually seeing the shape (and debugging normal orientations, see next). The wireframe mode is a poor man's attempt to show the mesh itself. It is very limited since, being 'transparent', you cannot easily see individual faces (they overlap other faces in the wireframe).

Wireframe-atop-solid combines the advantages of the two rendering modes. A naïve implementation of that simply draws the mesh twice, once as solid polygons and then as wireframe (with hidden surface removal). This is problematic: Since edges (in the wireframe) share the same 3D coordinates as the underlying faces, you get lines which are interrupted and actually 'flicker' as you rotate the view. The snapshot below (left) demonstrates this.



This is very easy to fix in OpenGL! In your drawing initialization code, that is, just before drawing the actual 3D geometry, add the following two calls:

```
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 2);
```

Then, draw the shape twice – once with solid faces, and once as wireframe. You should obtain the result shown in the figure above (right). A detailed explanation of what these two calls do is given here:

https://www.cs.rit.edu/~ncs/Courses/570/UserGuide/OpenGLonWin-14.html

5. Wrong normal orientations (?)

Many real-world shape databases contain so-called *inconsistently oriented faces.* These are faces whose vertices are not listed in the same order *as we look at the shape from the same position* (either from outside or from the inside). Recall that face normals are computed as the cross-product of two consecutive edges on a face. Hence, if seen from the outside, a face whose vertices are numbered in counterclockwise order will have an outside pointing normal. However, a face whose vertices are numbered in clockwise order will have an inwards pointing normal.

Such normal can cause problems for various aspects, such as shading but, more seriously, mesh analysis and descriptor computation.

How to deal with this? Split the problem:

(1) First check consistency: Are all faces oriented in the same way? You can check this simply by computing the normal yourself (as above) and rendering the shape. However, this doesn't scale if you have hundreds of shapes. A better way to check consistency programmatically is to visit all faces and note the order in which vertices of each edge are listed. Typically, such an edge E will be shared by two faces F1 and F2. If E is listed as (v1,v2) in F1 and as (v2,v1) in F2, then all is fine. If however E is listed in the same way in both F1 and F2, i.e. (v1,v2) in both or (v2,v1) in both, F1 and F2 have *opposite orientations*.

(2) Next check global orientation: Imagine that you have a shape with all faces listed so that normals are computed all pointing inwards. This is a global inconsistency, easy to fix: Just multiply all normal by -1.

(3) Try to fix consistency: If the test (1) told that faces are inconsistently oriented over the same surface, we need to flip *some of them*. Which set we do flip is not very important as long as they all become consistently oriented in the end.

Fully fixing consistency for any 3D surface is extremely hard (for the curious, see http://jcgt.org/published/0003/04/02/paper.pdf and papers cited there). However, some simple heuristics exist in particular cases:

- If you can reliably find a point **x inside** the shape: Let **v-x** be the position vector of some surface point (e.g. face center) with respect to **x**. Let **n** be the (possibly incorrectly oriented) normal of a face using vertex **x**. Then, the correctly oriented normal **n'** should be making an angle less than 90 degrees with **v-x**. Thus, **n'** is the vector of **{n, -n}** whose angle with **v-x** is the smallest. You can estimate that angle by the dot product of the normalized version of **v-x** and n, respectively **n'**.
  For convex shapes, x can be easily found as the shape's barycenter. For concave shapes, the problem is much more complicated.
- If your mesh is **clean** (no self-intersecting faces, i.e., it is a manifold like surface), another heuristic helps: Pick any starting face f, mark it visited. Decide that its orientation (order of listing of its vertices) is correct. Then, find all faces f_i that f shares one edge e_i with. For each f_i, check the order of the vertices along e_i: If these come in the same order as in f, reverse the order of the vertices in the entire face f_i, else leave them as they are and mark f_i visited. Repeat the process in breadth-first search over the surface until all faces have been visited. Now all faces have a consistent orientation. Note that you may still need to globally flip this orientation (see point (2) above).