Technical tips: Step 1

1. Report structure

There are several aspects to take care about here. They all stem from the fact that the final product is to be a *technical report*, similar in its size, structure, and style to any white paper or scientific article in practice. This implies that the report should be self-contained and structured so that any interested reader (having the required technical background) can understand *and* replicate your work just by reading the report. Several aspects flow from this:

- do not structure the report as a **per-week progress**. Rather, structure it so that every section covers a step of the assignment. Also, list explicitly the aims/goals of each step at the start of the respective section. This way, the reader knows exactly what will be covered in that section, without having to look elsewhere for the assignment description. Split large sections (for steps having several sub-steps) in subsections, as needed.
- **general sections**: to the above per-step sections, you should add a general introduction (describing the overall aims and scope of the document and a short overview of what comes in the next sections, as in any technical report), and an overall conclusion section. You can, if you find it appropriate, add some other general sections such as Design/Implementation Choices, where you describe your choice of tools, frameworks, programming languages, and the like.
- **figures, tables, equations**: when you use such elements, number them all and refer to them in the main text by their numbers. This makes it easy for you to refer later on in the report to any such element, and also to the lecturer to refer to them when providing feedback. Also, add page numbers so the lecturer can easily refer to a specific page when providing feedback. When using figures and tables, add a caption (text below that visual) that briefly explains what is shown there. As a general rule, any figure or table should have a simple and self-contained explanation in its caption. This allows one to quickly skim through the report and understand what such visuals are about, before diving into the detailed explanations in the main text.
- **references**: Add references (listed in a bibliography at the end) to all elements which are not your own creation. These include algorithms, methods, software toolkits, libraries, and frameworks. For the software artefacts, specify explicitly the version number and URL you got them from. This helps the reader in precisely seeing what you have used in your work and thus makes your work replicable.
- **source code**: You do NOT need to list the source code in the report. The main reason is that the source code is provided separately. You may, however, list very short fragments of source code, e.g. class names and method names, if these help you explain easier and more compactly certain parts of your design - for example, if you used a specific class and functionality thereof from a given package, and you want to comment on how you set the parameters of that function. Listing long code fragments makes the report unnecessarily long without bringing in additional clarity. Related to this: You do not need to provide source code with each progress iteration. Source code will change a lot over the iterations, so having the lecturer assess it continuously is neither practical nor useful. The source code will only be assessed at the end when you provide its final version.

2. Level of reporting

It is crucial that you understand what is the right level of detail for reporting. There are two aspects related to this: A good report tells (1) what was done and how that was done; and (2) shows proof that the results are correct and complete. We elaborate on these aspects below:

(1) What was done: For every step, you have to explain in detail what that step aims to accomplish. In order to do this, you have to introduce notations in the report early on. The best way to do this is to introduce them right in or before step 1. Of course, additional notations may be introduced later on. However, a large number thereof can be already introduced and explained at the start of the report, e.g. mesh, mesh cells, cell vertices, data attributes (defined per mesh, per cell, or per vertex). Once introduced, these notations should be used consistently throughout the entire document. This allows one to easily and precisely see what you did at every step of the process.

Separately, you need to explain how you accomplished all the technical operations you implemented (e.g., normalization, remeshing, feature extraction, computing distance functions, evaluating quality metrics). This means that every such operation must be explained both in plain text but also by means of a formula. The latter is especially important when operations are more involved. For example, for pose normalization, it is not sufficient to say "We rotate all shapes so their longest axis coincides with the X coordinate axis". Rather, you must complement this textual explanation by the exact and full formulas needed to define and compute a shape's longest axis and how you perform the rotation to align this axis with the X axis.

(2) Proof of completeness and correctness: For every computational step you add to your pipeline, you have to provide proof (evidence) that that step was designed and implemented correctly. In a very few cases, this can be done by inserting an actual mathematical proof. In most cases, however, this is not possible. Hence, you have to provide empirical (practical) evidence. I recommend two forms of this proof.

- **visual evidence**: For a given processing step, show a few selected shapes before and after the respective step. By comparing the two, the reader can get a good impression on how the processing works. Of course, this is not formal proof that your processing works correctly in all cases, but is a start in that sense. Example: For remeshing, you can show a shape (rendered with wireframe-atop-surface) before, respectively after, the remeshing. By visually comparing the two, one can see how remising actually worked.
- **numerical evidence**: For a given processing step, decide which characteristic that step should change. Then, compute the respective characteristic on all shapes before processing and after processing. Finally, compare aggregates of the characteristic before with those after. This comparison should immediately show that your

processing worked correctly - and in case it did not, it is an excellent tool for debugging cases when things did not work OK.

Consider again remeshing as an example. What is the GOAL thereof? To create shapes that have, overall, a similar number of elements (say, faces) and a low variability of sizes (say, face areas) over the same shape. To test that remeshing worked OK, you can thus

- define the two **characteristics** of interest of a shape: number of faces and face areas
- compute **aggregates** of each of these over the entire shape set **before remeshing**: A simple aggregate consists of a triplet (min, max, average). A more insightful aggregate is a histogram (giving e.g. the number of faces having areas in a given interval). If a shape has a wide variability of areas, its (min, max, average) values will be far apart; and its histogram of areas will look quite flat. Conversely, if a shape has faces of roughly the same areas, its (min, max, average) values will be very close to each other; and its histogram will be nearly zero everywhere except having a very large and narrow peak (around the value of the common face area).
- compute the same aggregates **after remeshing** (e.g. (min, max, average) or histograms)
- **compare** the aggregates before with the ones after remeshing. If the remeshing accomplished its goal of making all faces of roughly the same area, you should see this immediately in the characteristic computed after remeshing (e.g., pointy histogram).

Note that you need to aggregate the characteristics of interest over all data for which you want to check that your processing went OK. That is:

- if you want to check that remeshing went OK for a **single shape**, you compute and compare e.g. face-area histograms only for that shape (before vs after remeshing it)
- if you want to check that remeshing went OK for **all shapes** in a database, you compute and compare e.g. face-area histograms for all shapes jointly (before vs after remeshing all of them)

The same principle applies to basically ALL your normalizations. The only difference is which characteristics you choose to compute to 'test' that a given normalization worked OK. To determine this, think what the normalization should do. Examples:

- **number of vertices** normalization: If you want that all shapes have roughly N vertices, then the characteristic to check is the absolute difference between the actual number of vertices and N.
- **position** normalization: If you want that all shapes are translated so their barycenter matches the origin, then the characteristic to check is distance between origin and barycenter.

- **size** normalization: If you want that all shapes are scaled so they match a given bounding box, then the characteristic to check is the difference of sizes of the actual bounding box vs the desired bounding box.
- **pose** normalization: If you want that all shapes are rotated so their longest axis is along the X axis, then the characteristic to check is the angle between these two axes.

Recall, again: For every processing step you add to your pipeline, compute and present evidence in the report that that step worked OK on your database. This not only makes the report complete, but helps you in debugging (you spot potential errors early on and can fix them right away).

3. Generic tools

As you execute the assignment, you will find that a number of tools will appear to be useful in many different steps. Hence, it is good to think upfront about implementing them in a generic way. Once you do that, you can then reuse them very easily in subsequent steps. Examples include:

- **Dataset structure:** Your meshes actually come with additional data attributes, such as face areas, vertex normals, vertex curvature, local thickness at a vertex, and many more. Following Module 2, these are nothing but per-vertex and per-cell data attributes (scalars and vectors). Hence, it pays off to design a generic dataset implementation (e.g. class) that can accommodate a variable number of such attributes.

  Designing a simple and flexible dataset structure is not trivial. The challenge here is that you have multiple shapes (in a collection), each having multiple versions (e.g. before and after remeshing) and multiple attributes (e.g. area, normals, curvature). A good compromise between simplicity and flexibility is to have a two-level structure: You have a shape class (in the object-oriented sense) defined in your program. This class holds the mesh structure (vertices, cells) and a small, fixed, number of data attributes (e.g. one scalar and one vector attribute, defined for all vertices). Separately, you have a spreadsheet (table) in which each row is a shape and each column is the aggregated value of an attribute. You use the shape class to load specific shapes in your program and compute specific attributes. Then, if you want to make these attributes persistent (to use them later on), you save their per-shape aggregated values to the respective row (for that shape) in the database. Later on, when you want to further process a shape, you can load its geometry from the respective OFF or PLY file, and load its data attributes from the respective row in the database. This way, you don't need to compute everything from scratch for all shapes in the database - a lot of attributes are saved in the respective table (which can itself be saved as a simple CSV text file). This design is especially useful when you will compute shape descriptors, which are quite intricate and slow. Saving them in such a table makes a lot of sense. Finally, this helps easy debugging: You can have multiple versions of the attributes table, e.g. if you compute these with various parameter settings and want to compare them.

- **Statistics**: As explained above, computing various statistics (min-max-average, histograms) is useful in many parts of your pipeline. Hence, it pays off to think about implementing generic components for that. For instance, you implement a histogram tool that, given one data attribute (e.g. area) defined over a set of items (e.g. shapes, faces of shapes, etc), and a number of parameters (e.g. number of bins), computes the histogram of that attribute over that set. Similarly, it is useful to have a generic tool that visualizes (draws) any such histogram. Other examples of useful tools here are histogram normalization and computing the difference of two histograms.

4. Dealing with outliers

Your input are real-world 3D shape databases which are *complex*, *variable*, and overall *messy*. As such, it is very likely that you will encounter some shapes which are hard to deal with - e.g., you cannot easily normalize them, or they pose problems when computing certain descriptors. How to deal with such *outliers*? The answer has two parts:

- **Detect the outliers:** For this, see the characteristics method outlined above: Each processing step in your pipeline should compute a characteristic that you can (should) check afterwards to test if that step was computed correctly. This will immediately show you the shapes which, for some reason, failed computation. Identifying them is thus the first step.
- **Handle the outliers:** If these outliers are few, you can try to manually fix them by repeating the processing step(s) with various manually-set parameters, e.g., use the PMP viewer or GeomView to manually remesh the shapes. If this fails (or you do not have enough time to do this), you can simply exclude these shapes from the database. The important thing here is that you do not exclude too MANY shapes. To answer this, check the types (classes) of shapes in the database: After exclusion, you should still have a balanced database, i.e., similar numbers of shapes in each class, so that the computation of retrieval metrics, later on, makes sense. If, however, these outliers are many (thus, they actually are not outliers), this means that something is wrong with your current processing step, so you should revisit/fix it before proceeding further.