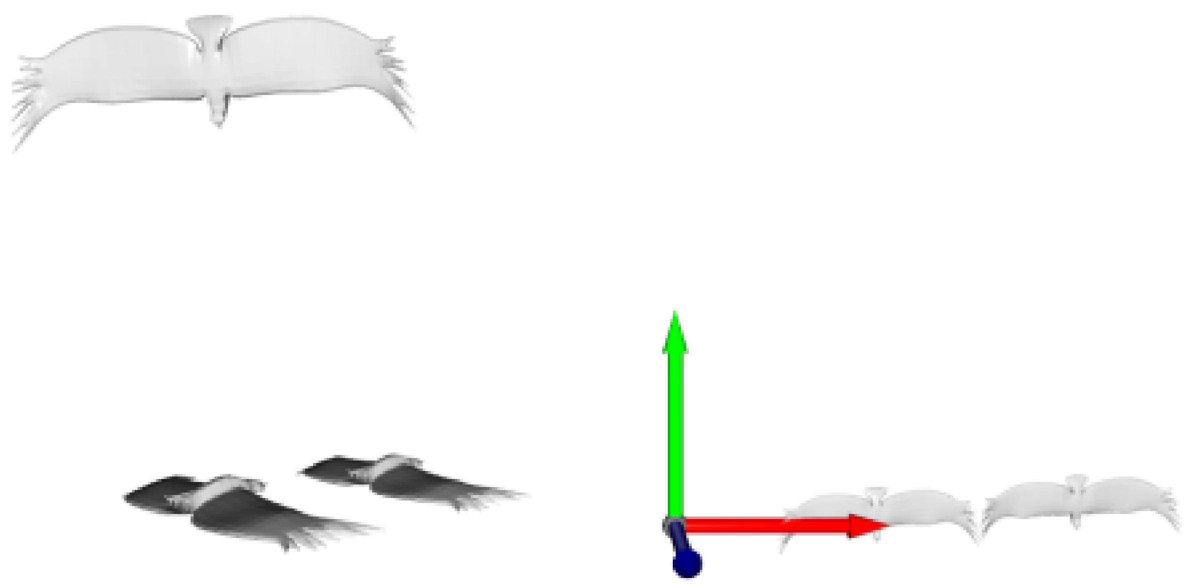
Technical guide step 2 - Preprocessing

As in the previous guide, exercises are included in this guide to help you think about the discussed topics. You don't need to hand in any solutions to these exercises; they are meant only for personal study.

Basic transformations

The provided sample script '`transformations_basic.py`' shows how to perform three basic mesh transformations (scaling, rotation, and translation) on a sample mesh. We also recommend taking a look at open3d's 'Transformation' documentation page, from which these code snippets were taken (Transformation — Open3D latest (664eff5) documentation)



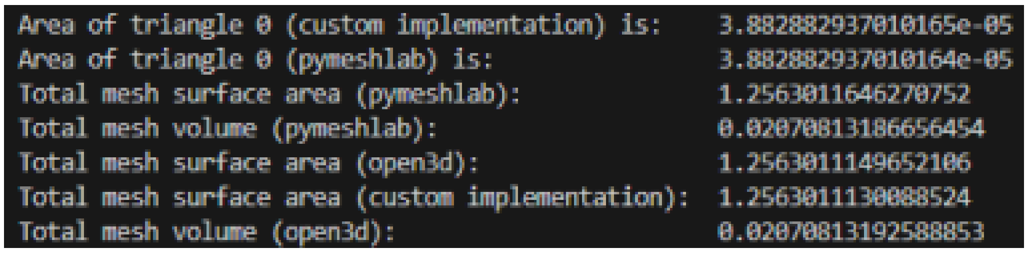
This is the UI output you should get when running 'transformations_basic'.

Exercise 1: Notice that for the rotation and scaling methods, differences in settings for the parameter 'center' have the effect of placing the meshes in different positions. Can you explain why? Tip: the value passed to the 'center' parameter describes a position around which the mesh is rotated, or with respect to which the mesh is scaled, respectively.

Computing area and volume

The provided sample script `'compute_face_areas.py'` showcases different methods of computing the area of triangles, as well as mesh volume. PyMeshlab has options to compute either the areas of all triangles (yielding an array with as many entries as there are triangles in the mesh), or the total area of the mesh. Open3d has similar functions. We encourage you to also view the documentation of the functions that are being called in the sample script in the online documentation of PyMeshLab and Open3D. You are likely to find many other useful functions there to obtain other mesh features (please do keep in mind that we may not be able to help you with questions about other functions, as we do not know all the details of the libraries' functionality).

The sample script also contains a custom implementation of a triangle area calculation function. This is intended to help you get a feel for how you can iterate over vertices and triangles, so that you can create feature calculation functions yourself. This might be particularly useful for those of you who are not yet very familiar with computer graphics. Implementing your own mesh feature computation functionality will be an important part of later assignment steps, so taking some time to learn to do a bit of basic mesh processing could be a worthwhile investment.



```
Area of triangle 0 (custom implementation) is:    3.882882937010165e-05
Area of triangle 0 (pymeshlab) is:                3.882882937010164e-05
Total mesh surface area (pymeshlab):              1.2563011646270752
Total mesh volume (pymeshlab):                    0.02070813186656454
Total mesh surface area (open3d):                 1.2563011149652106
Total mesh surface area (custom implementation):  1.2563011130088524
Total mesh volume (open3d):                       0.02070813192588853
```

This is the console output you should get when running `'compute_face_areas.py'` Note that area computations using different libraries/methods yield very similar, but slightly different, results.

Mesh center computation

One way to get the center of a mesh is by taking the mean of all vertex positions.

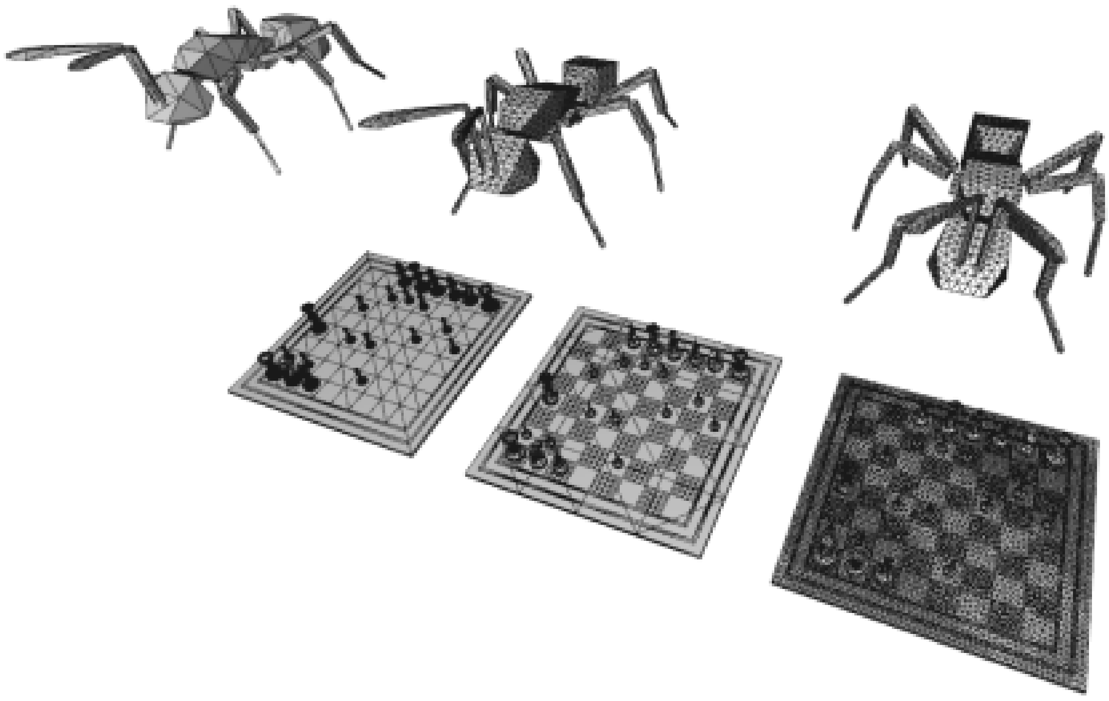
Exercise 2: Instead of using the `get_center()` Open3d function which is used in `transformations_basic.py`, implement your own custom function to get the center of the shape by computing the mean of all vertex positions.

Use your custom function to find the center of m1342.obj. You could display the center by rendering a sphere at the center position. Does your custom function give the actual center of the geometry? Why is your custom function giving that position?

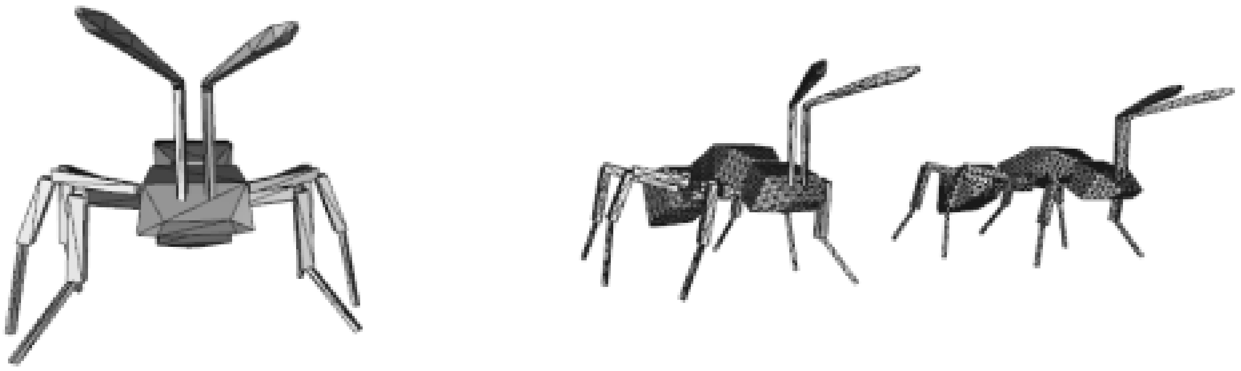
Exercise 3: Write an improved function to get the center of a shape.
Hint: The `compute_face_areas.py` file might be useful to do this. For more info on this file check out the previous page of this document.

Mesh refinement and -decimation

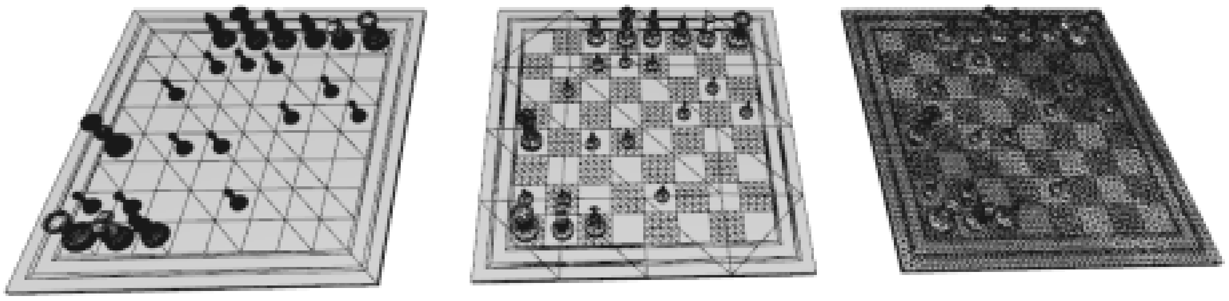
The sample script '`remeshing.py`' provides examples of mesh refinement and mesh decimation using pymeshlab. First, low-resolution and high-resolution meshes with few, respectively many, triangles are loaded. Then, the number of triangles in each mesh is increased, respectively decreased, by calling a pymeshlab remeshing function. The visual result is shown below.



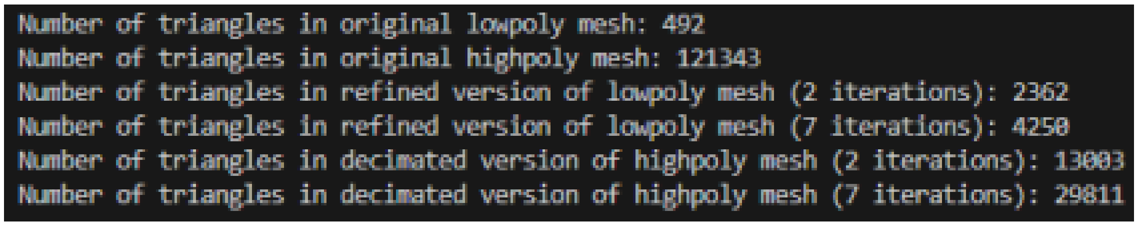
Explanation of the visual output:



Left: original low-res mesh from the database, **middle and right**: result of refining the input model to target edge length 0.02 using 2 iterations (middle) and 7 iterations (right)



Left: original high-res mesh from the database, **middle and right**: result of decimating the input model to target edge length 0.02 using 2 iterations (middle) and 7 iterations (right)



```
Number of triangles in original lowpoly mesh: 492
Number of triangles in original highpoly mesh: 121343
Number of triangles in refined version of lowpoly mesh (2 iterations): 2362
Number of triangles in refined version of lowpoly mesh (7 iterations): 4250
Number of triangles in decimated version of highpoly mesh (2 iterations): 13003
Number of triangles in decimated version of highpoly mesh (7 iterations): 29811
```

Console output showing how the number of triangles changes due to remeshing.

Notice the large contrast in terms of vertex density within the original high-resolution mesh; the board is comparatively large but consists of only several dozen triangles, whereas the chess pieces are each quite high-resolution despite their relatively small size). What differences can you see between the remeshed meshes that were remeshed with 7 iterations versus the ones which were remeshed with 2 iterations? Tip: the difference has to do not only with the number of vertices but also their distribution over the surface of the mesh.

Exercise 4: Can you think of why a 3d artist might intentionally add more vertices and triangles to some parts of a mesh and leave other parts fairly low-resolution?

An important parameter of the remeshing function is 'targetlen', which specifies a target for the edge length of the remeshed object. Smaller values for targetlen should result in meshes with more triangles.

Exercise 5: Can you explain the relation between target edge length and the number of triangles in the remeshed mesh?

Another important parameter is 'iterations'. The remeshing algorithm used by this function (isotropic explicit remeshing) is an iterative one. As you might expect, increasing the number of iterations will yield more high-quality meshes (in the sense that the result will have average edge lengths closer to the value specified for the parameter 'targetlen', and the variance of edge lengths will also be lower).

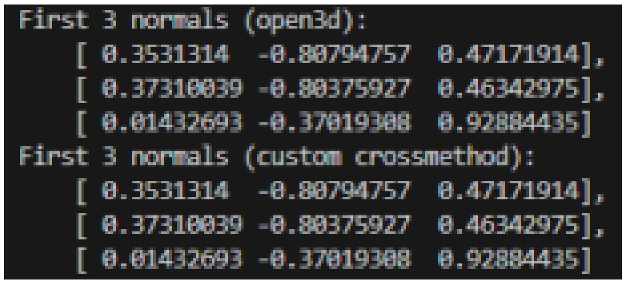
Unsurprisingly, more iterations also means more time is needed to generate the remeshed result. So, in practice, a balance must be found between remeshing quality and performance. Finding this balance is up to you.

Exercise 6: Imagine that we have two 3d meshes, one with a small surface area and another with a large surface area. We remesh both meshes using pymeshlab as shown in the sample script, in both cases using the same values for 'targetlen' and 'iterations'. Do you think the remeshed meshes will have the same number of triangles? If not: Which of the two input meshes do you think will lead to a remeshed mesh with more triangles? Why?

FYI: Open3d also has some mesh refinement and decimation functionality, but they may not produce results on par with those of pymeshlab (although we encourage you to try both and see which works best). Open3d's refinement- and decimation functionality is described here [Mesh — Open3D master (b7f9f3a) documentation](), under 'subdivision' and 'simplification', respectively.

Compute triangle normals

The sample script '`compute_face_normals.py`' shows two ways of computing triangle normals. The first method uses the built-in open3d method 'mesh.compute_triangle_normals()', and the other is a custom implementation. The custom method computes triangle normal by taking the cross product of two edges of the triangle in question.

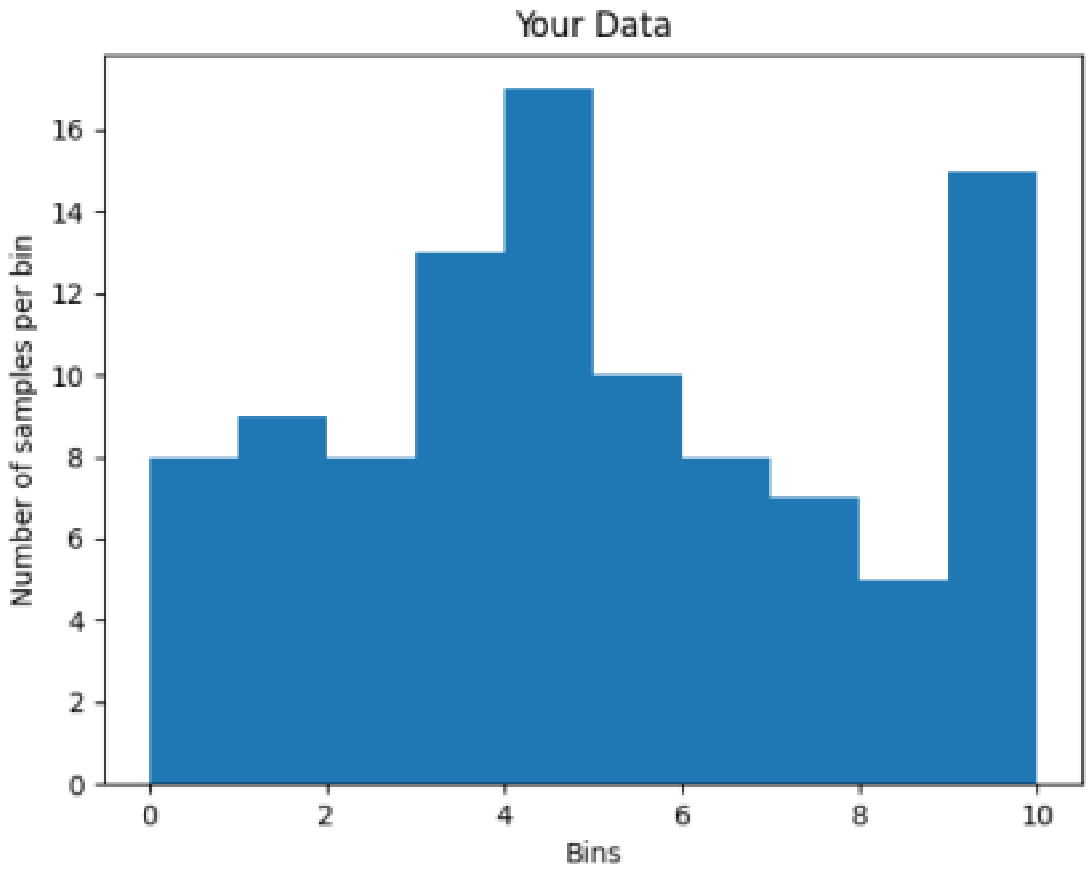


```
First 3 normals (open3d):
    [ 0.3531314  -0.80794757  0.47171914],
    [ 0.37310039 -0.80375927  0.46342975],
    [ 0.01432693 -0.37019308  0.92884435]
First 3 normals (custom crossmethod):
    [ 0.3531314  -0.80794757  0.47171914],
    [ 0.37310039 -0.80375927  0.46342975],
    [ 0.01432693 -0.37019308  0.92884435]
```

Console output of the sample script '`compute_face_normals.py`'

Computing and plotting a histogram

The sample script ‘`histogram.py`’ contains a simple example of a histogram computation using numpy’s histogram function, which is then plotted using matplotlib. The ‘np.histogram’ function simply takes the list of observations in the form of a numpy array, along with the number of bins. By default, the sample script sets the number of bins equal to the square root of the number of samples.



Exercise 7: Try setting the variable ‘no_bins’ to different values and plotting the results. Can you see advantages and/or disadvantages in having more or less bins? Do you agree that the recommended number of bins provides the most visually readable and informative result? If so, why do you think the square root of the number of samples is such a good rule of thumb? (an answer does not necessarily need to come in the form of a mathematical proof, but can also be a well-explained intuition).

Exercise 8: Create a Gaussian distribution centered at 0 and with standard deviation equal to 3 Sample that distribution with 10000 values. Then, plot the histogram of the resulting sample set. Do you see indeed a nice Gaussian?