Acquisition and Processing of 3D Geometry
Assignment 2 – Laplacian filtering
Report

Benjamin Barral

0. Questions attempted and comments on my implementation

I attempted all the tasks required in this coursework, and achieved them to a reasonable extent.
I coded this coursework in C++, using LibIGL.
I used the library Spectra for eigen value decompositions (spectral reconstruction, task 3).

Figure 1 shows the interface I implemented. It allowed me to run and simulate all of the tasks at once, for one mesh.
I ran my simulations on all of the meshes except for the dragon.

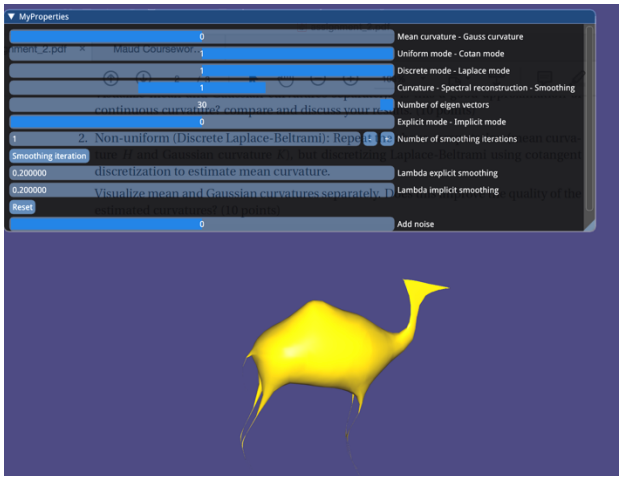


Figure 1 : The LibIGL interface I built for this coursework

I. Discrete curvature and spectral meshing

1. Uniform Laplace

a. Implementation

I implemented the discretization of the Laplace operator using both a per-vertex equation, and the uniform Laplace matrix.
I used the `igl::adjacency_list` method for the vertex connectivity.
I then computed the mean curvature using : $H = \frac{\|\Delta_S x\|}{2}$.
The limitation is that such defined mean curvatures can only be positive, whereas in theory the sign of the curvature should be given by the orientation of the normal. Unfortunately, no information has been provided for the mesh normals. I am aware that I could have used algorithms to estimate point cloud normals, such as using Principal Component Analysis on the covariance matrix, but I did not make such an effort. Therefore, I could only interpret the magnitudes of the mean curvatures in my results, and not the signs.

For the Gaussian curvature, I used the "angle deficit" formula with area normalization.
I used barycentric area normalization : I computed the areas of the triangles and divided them by 3.
I visualized both mean and Gauss curvature fields using the `igl::jet` function, with normalization for

color.

b. Results

I found that the uniform version of both mean and gauss curvature was quite limited.
For mean curvature, the main artefact observed was the very high curvatures on large triangles that were not in curvy areas (*figure 2*, right), due to the fact that distant neighboring pixels are adding a large impact to the average of positions.
On the camel example (*figure 3*), the legs and the body have big triangles – but are relatively planar – and end up with high curvature values, whereas the toes and ears are made of small triangles (due to the presence of details) and end up with relatively small values. This is very visible on the cow's body as well (*figure 4*).
On the bunny example, this artefact is obvious on the pitted bottom of the body (*figure 2*), where the triangles are huge.

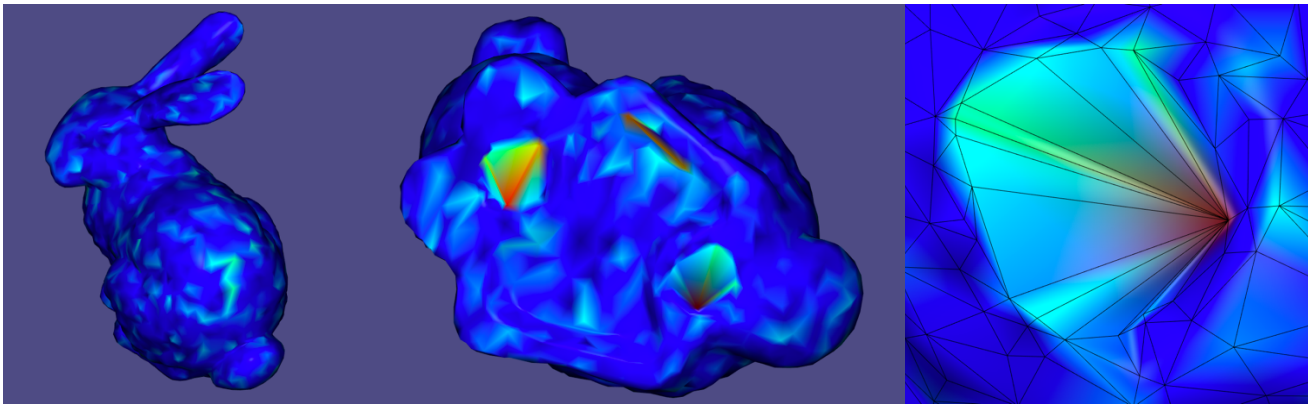


Figure 2 : Results of uniform mean curvature on the bunny

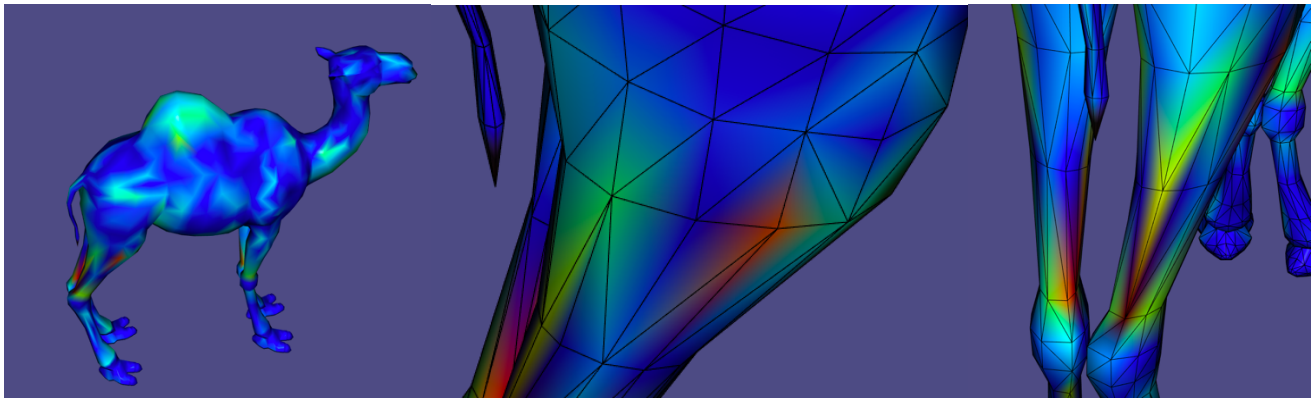


Figure 3 : Results of uniform mean curvature on the camel

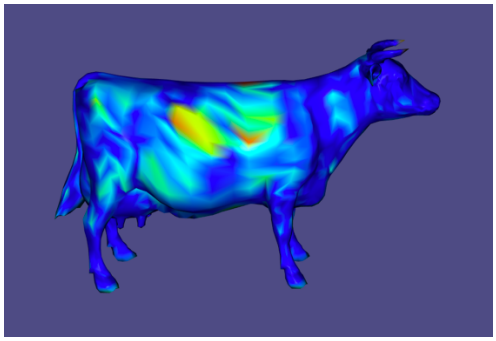


Figure 4 : Results of uniform mean curvature on the cow

I found the discretization of Gauss curvature to be a bit more accurate.
First of all, using area normalization prevents from the previously described artefact.
Secondly, in the continuous case, Gauss curvature is defined by the product of the minimal and maximal

curvatures. The example of the camel (*figure 6*) shows a relatively decent estimation, with gauss curvatures being relatively uniformly low on the whole mesh, because parts like the body or the legs are (approximately) planar in one direction (almost cylindrical); except for the toe tips which are more spherical, and rightly end up brighter. Similarly, on the cow mesh, the whole body has a rather uniform low curvature, and the high curvatures are seen on the horns and the udders (*figure 7)*.
Nonetheless, as discussed in the next section, area normalization has artefacts as well, which explain why the gauss curvature visualizations are quite dark (due to extreme outliers, such as a point on the tail of the cow, see *figure 7, right*).

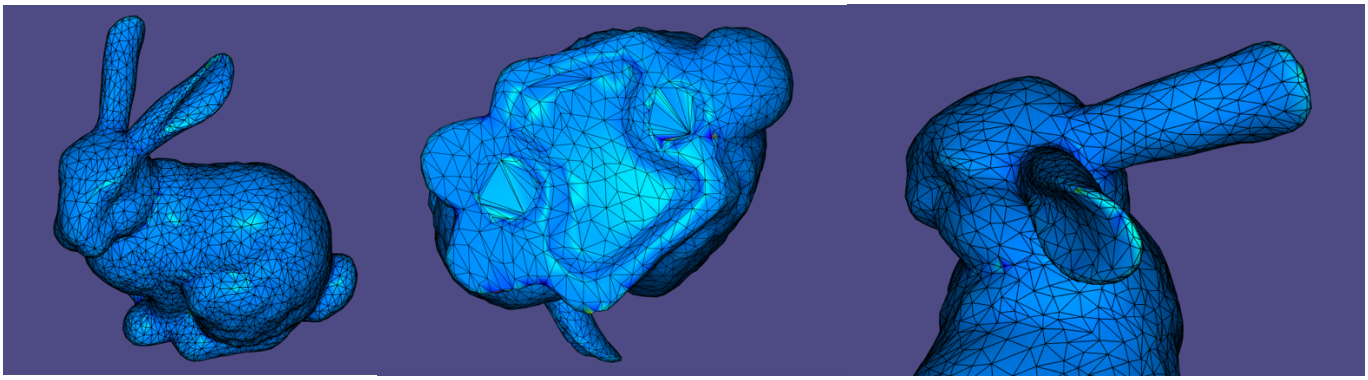


Figure 5 : Results of Gauss curvature on the bunny
Uniform, high on the ear. Bottom less erratic than mean curvature (figure 1)

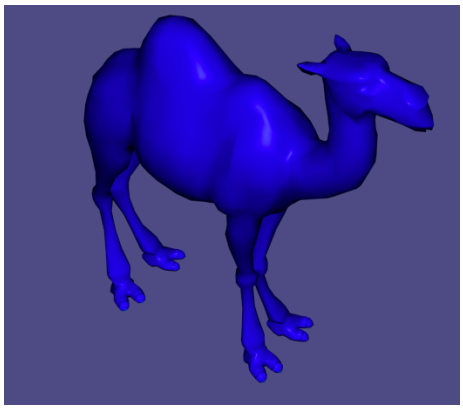


Figure 6 : Result of Gauss curvature on the camel
Comment : very uniform except for a small variation on the toes

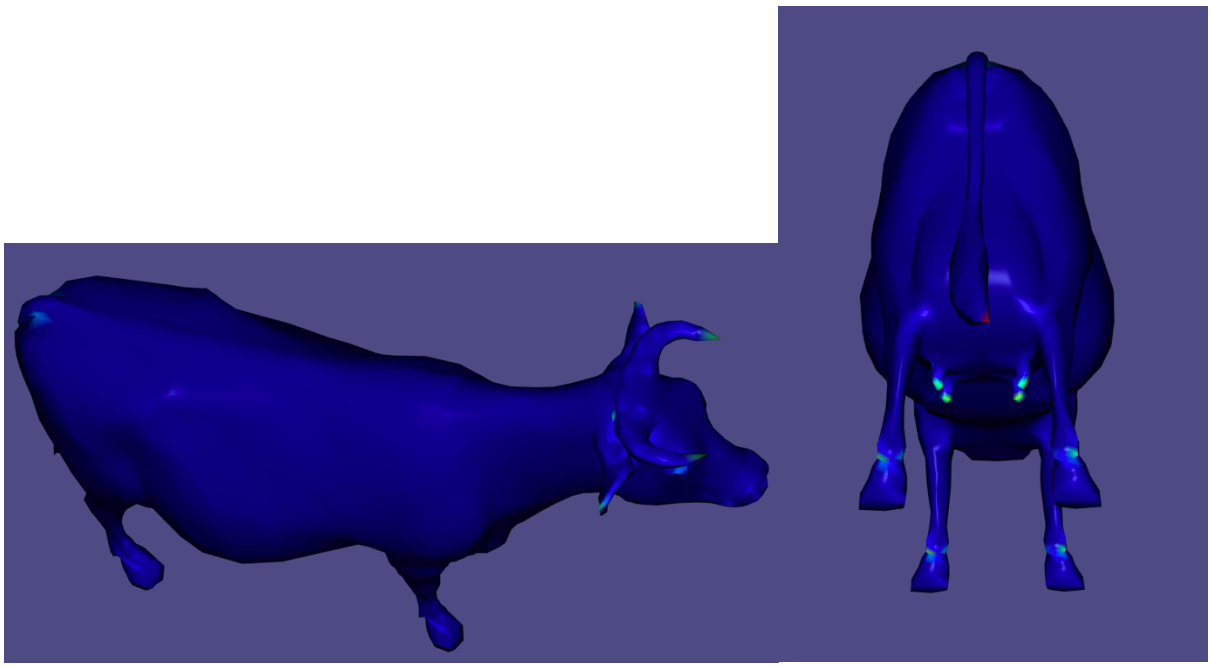


Figure 7 : Results of gauss curvature on the cow
Horns and udders
Outlier on the tail

2. Cotan Laplace

a. Implementation

To compute the Laplace-Beltrami ("cotan" version) matrix, I constructed a vertex-face adjacency vector (of type `std::vector< std::vector<int> >`, the k^{th} element has a list of the indices of the faces adjacent to the k^{th} vertex).
I then do a simple double loop on vertices and adjacent faces.
The pseudo-code is the following :

```
C and M : empty sparse matrices
For each vertex v (of index i)
   areaSum = 0
   For each adjacent face f
      Let i1 and i2 be the indices of the two vertices in f which are not v.
      Let u1 = V(i1) and u2 = V(i2)
      Compute the angles a2 = angle(u1->v,u1->u2) and a1 = angle(u2->v,u2->u1)
      C(i,i1) +=  cot(a1) ;   C(i,i2) +=  cot(a2);   C(i,i) -=  ( cot(a1)+ cot(a2) )
      areaSum += area(v,u1,u2)s
    end
    M(i,i) = 2 * areaSum
end
```

Like for Gauss curvature, I used barycentric area normalization.
The code for the matrix construction is in the function `computeLaplaceBeltramiMatrixCotan(…)`.

b. Results

Overall, the results provided by the cotan discretization are more accurate (i.e. more in keeping with the continuous definition of mean curvature) than the uniform one.
Figure 8 shows results for mean curvature with cotan, where the uniform version had failed (section 1.b.).
Typical examples are the camel toes, the cow udders and the bunny ear edges, which all rightly look bright now.

Nonetheless, I observed some isolated artefacts generated by the cotan discretization.
The first one appears with extremely stretched triangles, with the distribution of angles being close to {0,0,180°}, all of whose cotangents are diverging to +- infinity (see *figure 9*, *first row*).
The other artefact is on points with high valences, where the sum of neighboring areas end up being large and therefore bringing the curvature down : see the cow's horns (*figure 10, second row*), which look brighter on the pixels neighbor to the actual peak.

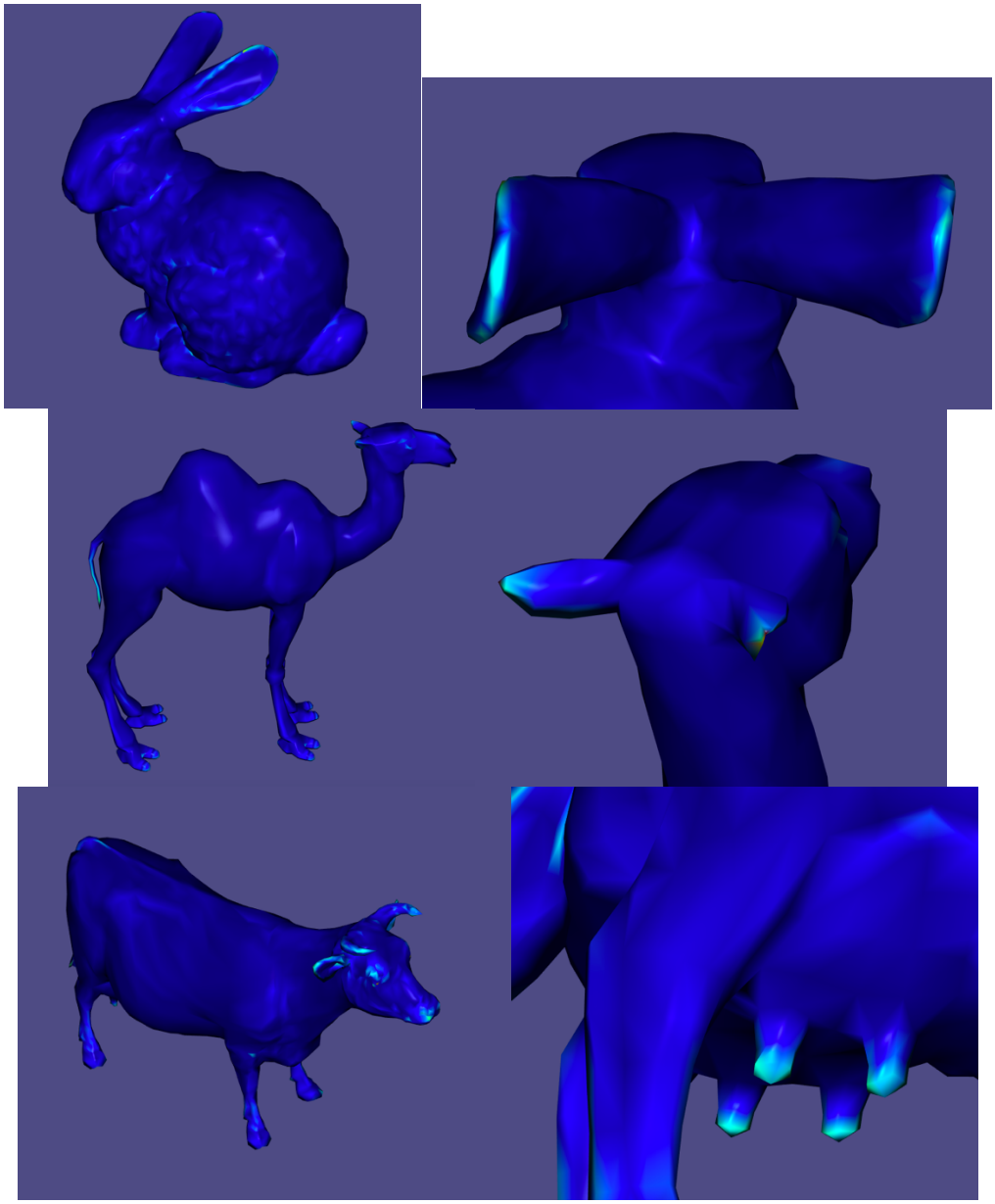


Figure 8 : Results of mean curvature with cotan Laplace-Beltrami

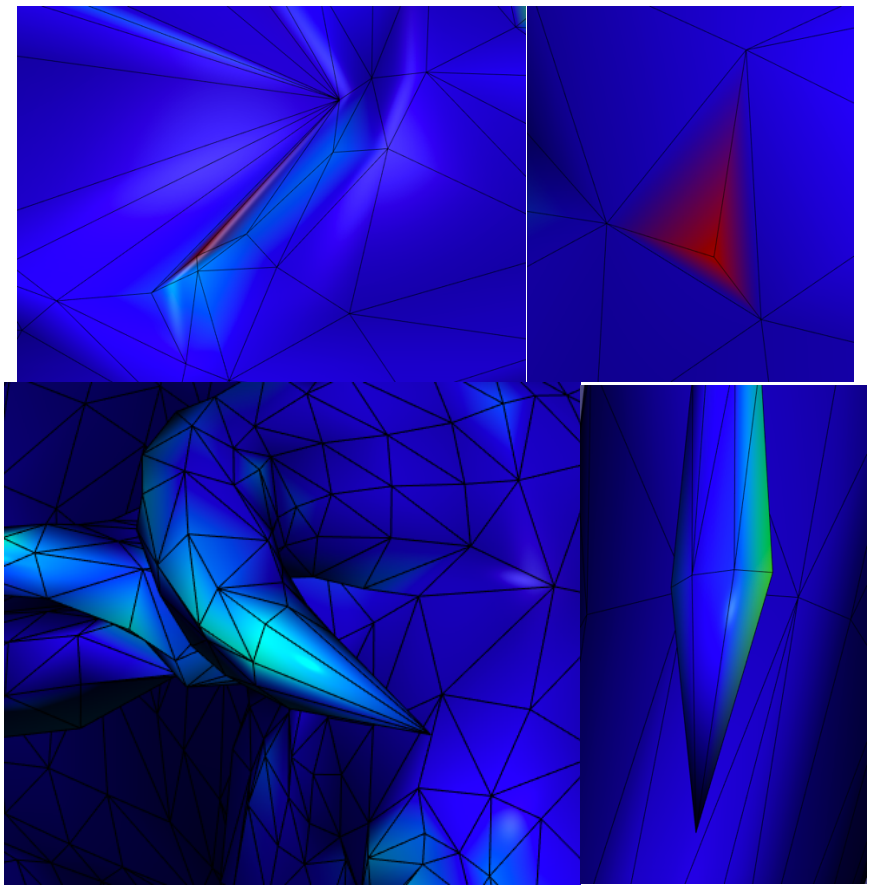


Figure 9 : The artefacts of mean curvature with cotan discretization
Upper left : bunny's legs – Upper right : cow's neck
Bottom left : cow's horns – Bottom right : camel's tail

3. Spectral reconstruction

Figures 10-12 show results of spectral reconstruction with Laplace-Beltrami (cotan) eigen analysis, with different values of k (number of eigen vectors).
I used the "redefinition" of inner product, as provided by the coursework FAQ material.
I found that a value of k greater than 6 usually gave decently intuitive representations of the meshes, and that the representation would naturally get refined as k increased.

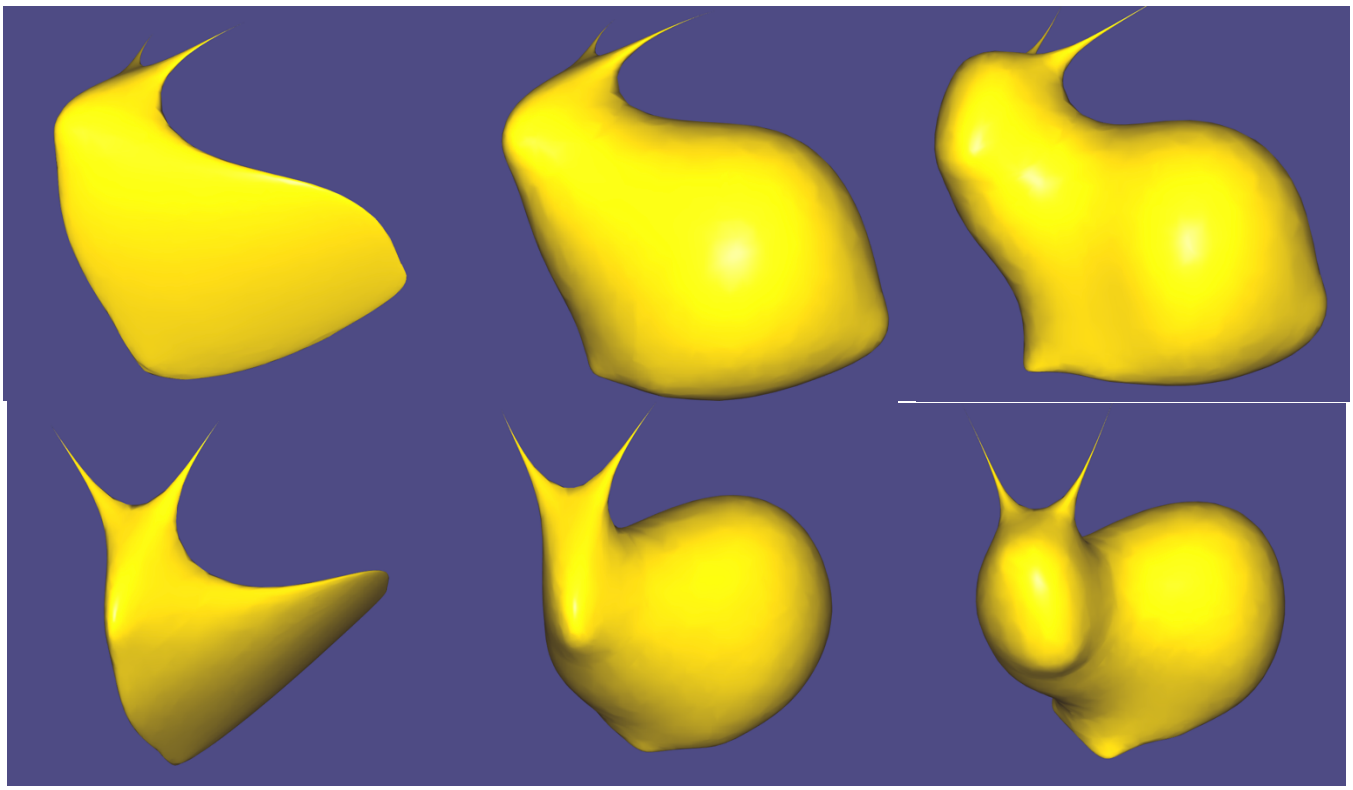


Figure 10 : Spectral reconstruction of the bunny
Left : k = 5 – Middle : k = 10 – Right : k = 30
Bottom vs Top : different views

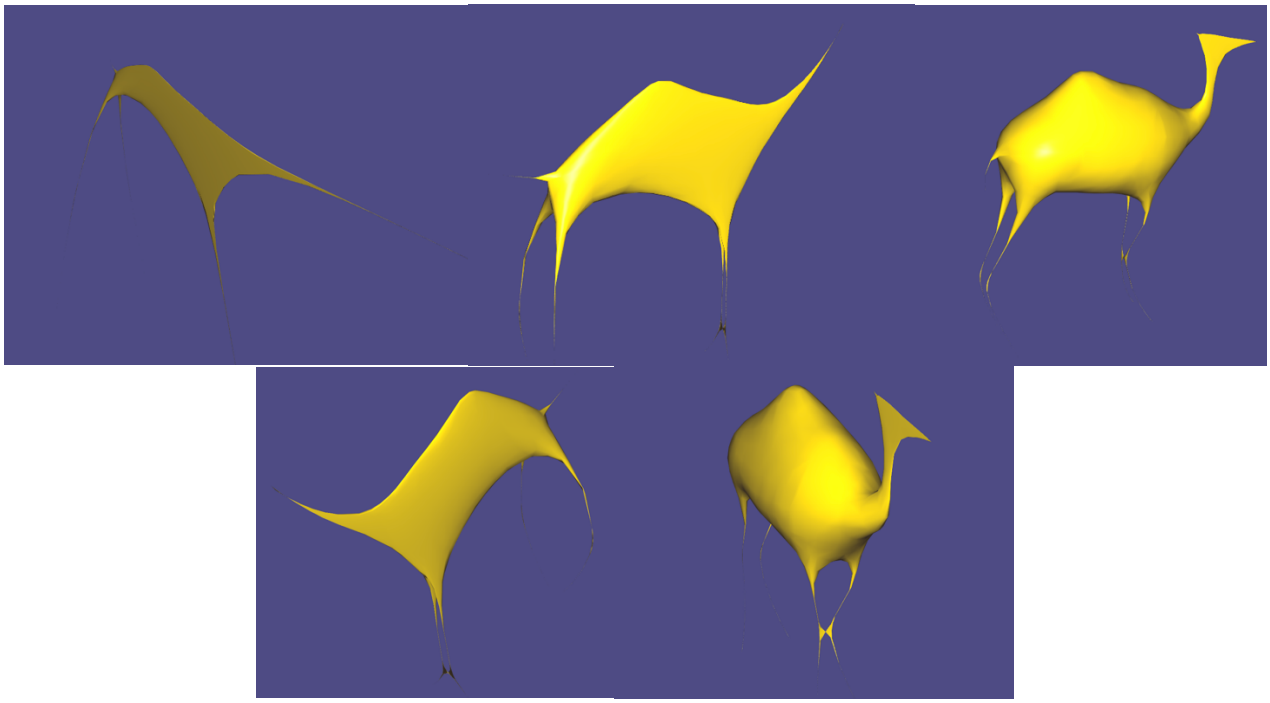


Figure 11 : Spectral reconstruction of the camel
Top : Left : k = 5 – Middle : k = 10 – Right : k = 30
Bottom : other views of k = 10 and k = 30

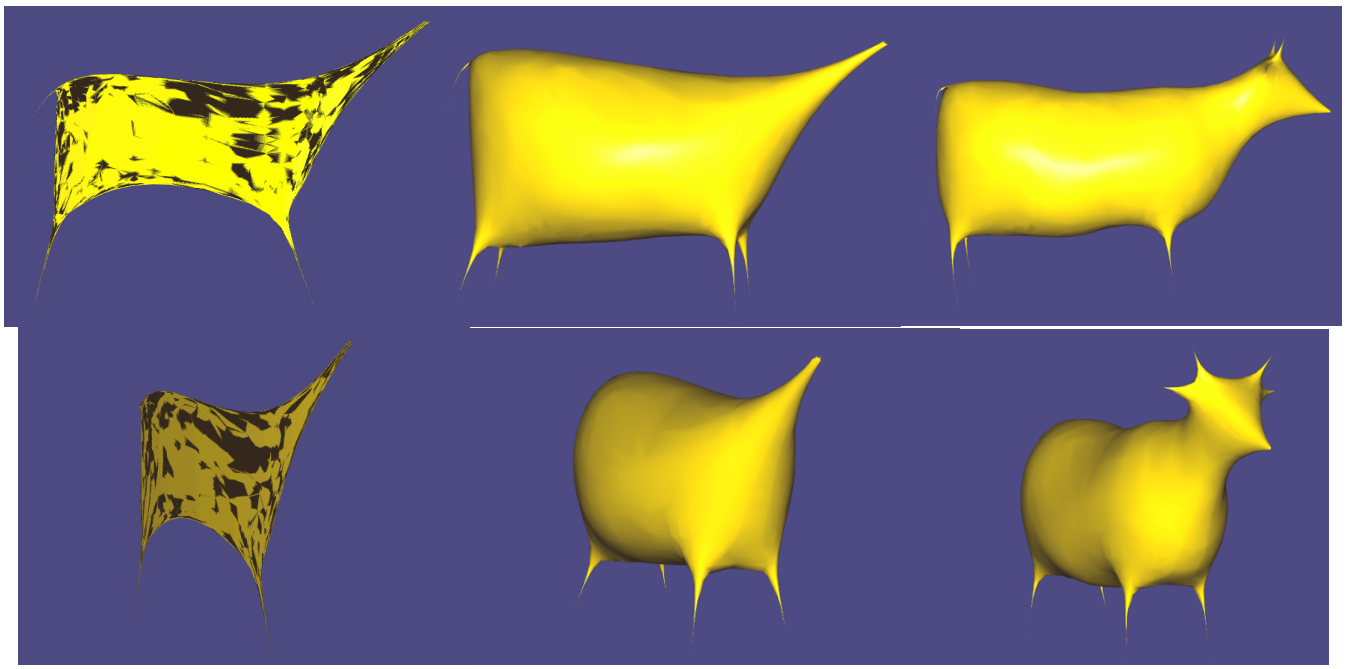


Figure 12 : Spectral reconstruction of the cow
Left : k = 5 – Middle : k = 10 – Right : k = 30
Bottom vs Top : different views

I also tried the spectral reconstruction using the Uniform Laplace (regular dot product product this time). I noticed the results were less "coherent" than with the cotan Laplace-Beltrami matrix (*figure 13* : notice the "weird" bunny lower part and legs)

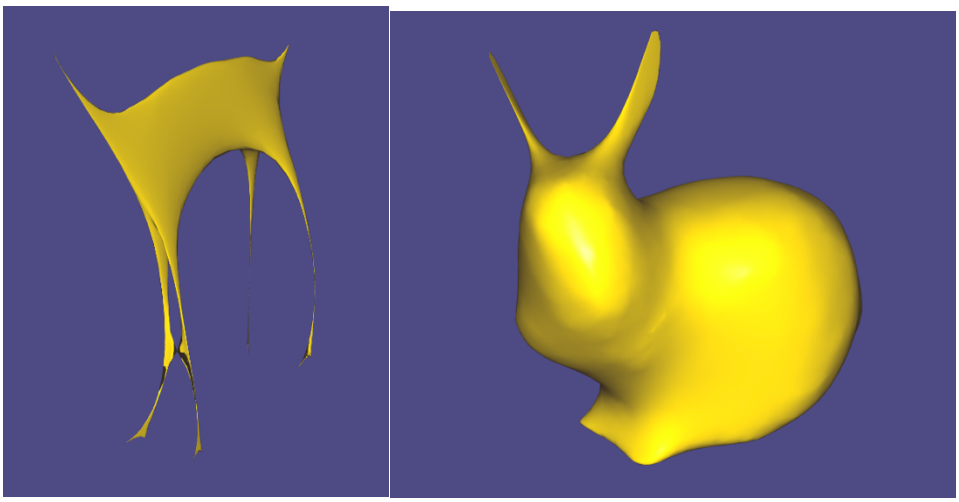


Figure 13 : Results of spectral reconstruction with Uniform Laplace
Left : Camel : k = 10
Right : Bunny : k = 30

II. Smoothing

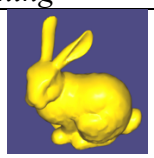
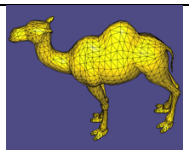
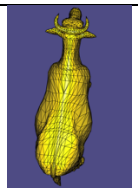
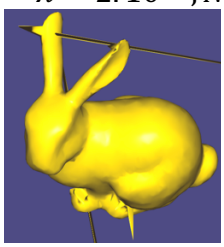
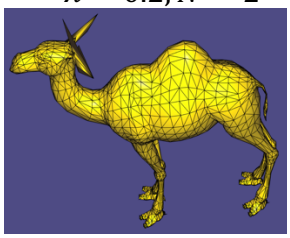
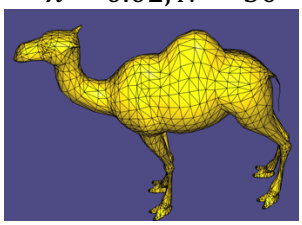
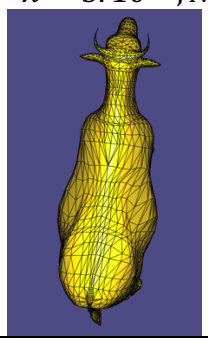
I performed Laplacian smoothing using the Laplace-Beltrami matrix (cotan version).

1. Explicit Laplacian smoothing

If the step size is too large, then the mesh gets extremely distorted after potentially one iteration, especially on the points with large Laplacians (i.e. the points that showed high curvatures in the previous section). This is because the method is unstable (c.f. *Figure 14*, first row of Results).

The limit step size depends on how smooth the mesh is in the first place.
The next chart shows my results.
I found that the bunny was the only mesh for which smoothing made a significant difference (because it is the noisiest out of the three).
For the cow and the camel: smoothing has the effect of making the body look "skinnier" – especially the horns, the legs and the udders get thinner.

Figure 14 : results of explicit smoothing

	Bunny 	Camel 	Cow 
Lambda limit for stability	10^{-7}	0.02	5.10^{-7}
Results (λ : step size N : number of iterations)	$\lambda = 2.10^{-6}, N = 3$ 	$\lambda = 0.2, N = 2$ 	$\lambda = 2.10^{-5}, N = 3$ 
	$\lambda = 2.10^{-7}, N = 80$ 	$\lambda = 0.02, N = 50$ 	$\lambda = 5.10^{-7}, N = 300$ 

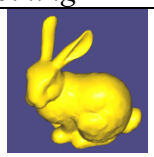
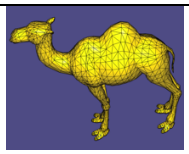
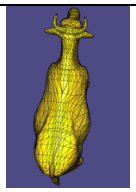
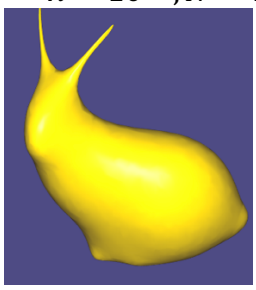
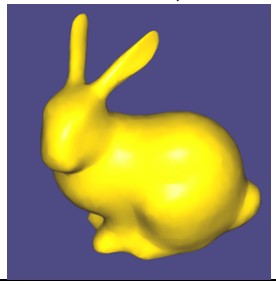
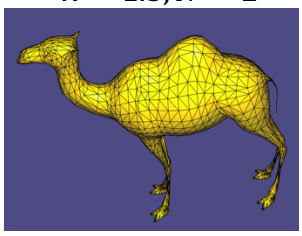
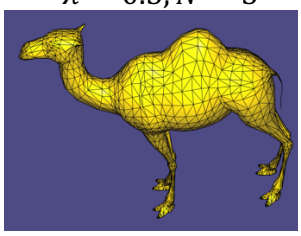
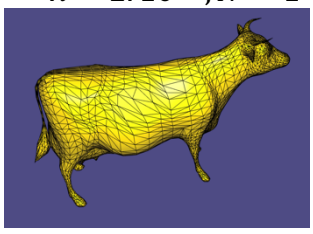
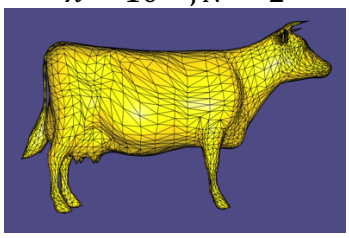
2. Implicit Laplacian smoothing

I used the `BiCGSTAB` solver from the Eigen library, which implements a "bi conjugate gradient stabilized solver for sparse square problems". Bi conjugate gradient was suggested by Desbrun et al.

This method is more stable than the 'explicit' smoothing, since no matter what the step size is, the mesh never 'blows up'. Setting too high a lambda for implicit smoothing in fact has the effect of "oversmoothing" the mesh (see *figure 15*, first row in "Results")
Some of the results remind the "low frequency" meshes obtained with spectral reconstruction and a small amount of eigen vectors.
Even if this method is stable, due to the last observation, I experimented on finding the step sizes that were optimal for the mesh to be smoothed adequately.
I noticed that this method is more tolerant on the step size (usually one or two orders of magnitude greater than for explicit smoothing), and that it converges a lot faster : about 100 iterations were needed for explicit smoothing, whereas only 2 to 3 can lead to very convincing results with implicit smoothing.
For the camel, I could even set a step size greater than 1 and get decent results after one iteration.

See *figure 15* for my results on the optimal step size for implicit smoothing.

Figure 15 : results of implicit smoothing

	Bunny 	Camel 	Cow 
Lambda limit for good results	10^{-5}	0.5	10^{-5}
Results (λ : step size N : number of iterations)	$\lambda = 10^{-3}, N = 1$  $\lambda = 10^{-5}, N = 3$ 	$\lambda = 1.5, N = 2$  $\lambda = 0.5, N = 3$ 	$\lambda = 2.10^{-4}, N = 1$  $\lambda = 10^{-5}, N = 2$ 

3. Denoising

I generated zero-mean Gaussian noise on the vertex positions, setting the gaussian variance (sigma) as a ratio of the mesh's bounding box (the code is in the "`addNoise()`" function).
I tried increasing values of the ratio (therefore increasing the amount of noise).

I used the implicit smoothing, and the optimal step size found before for each mesh.
I found that the implicit method smooths quite accurately the noisy mesh, even with extreme amounts of noise (c.f. *figure 17*).
Figure 16 reports the number of iterations needed to get a decently smooth result, depending on the amount of noise added.
I kept on adding noise up until ratio = 0.1 (i.e. 10% of the bounding box dimensions) which looks quite chaotic (*figure 17*), which still lead to decently recognizable meshes after a few number of iterations.

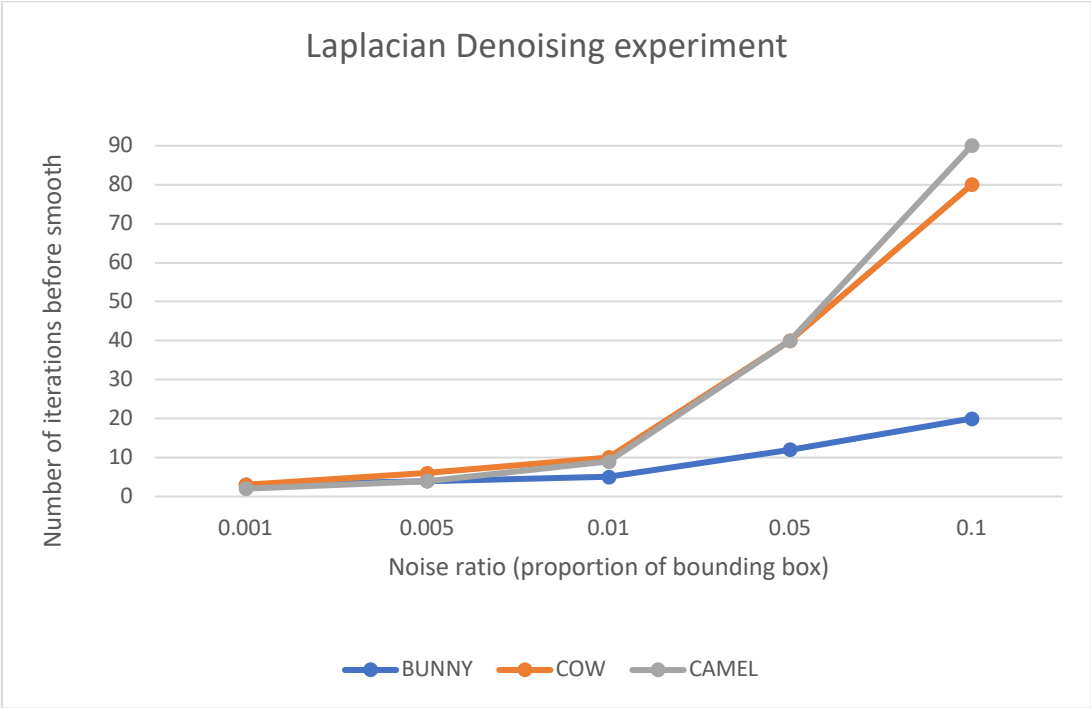
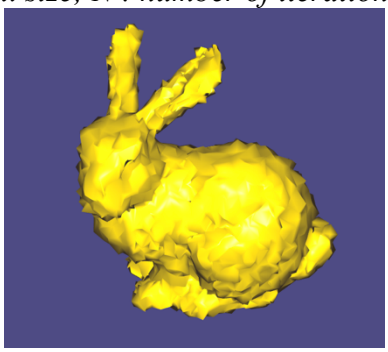
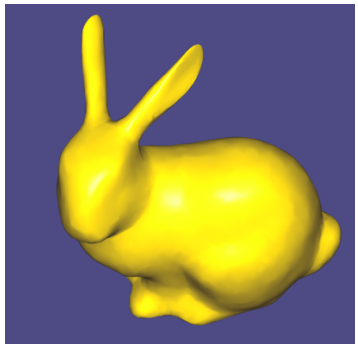
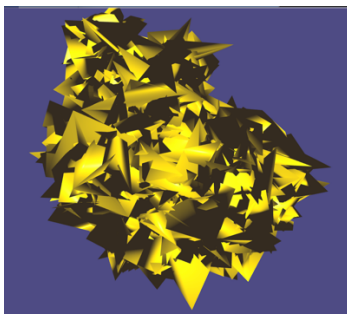
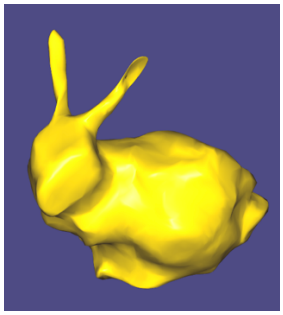
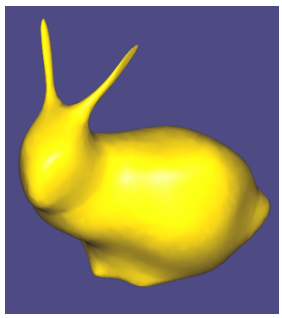
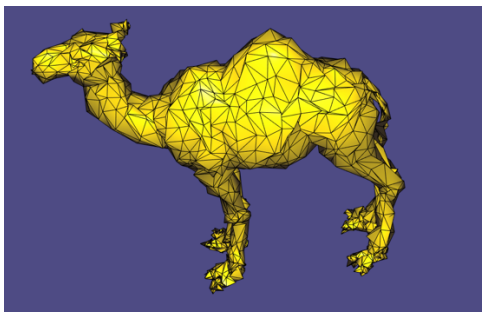
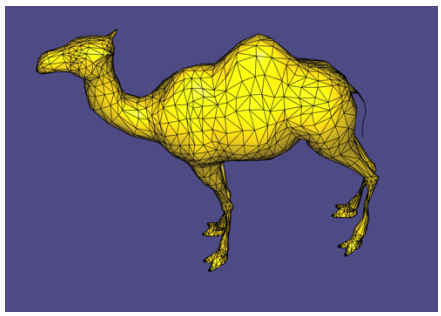


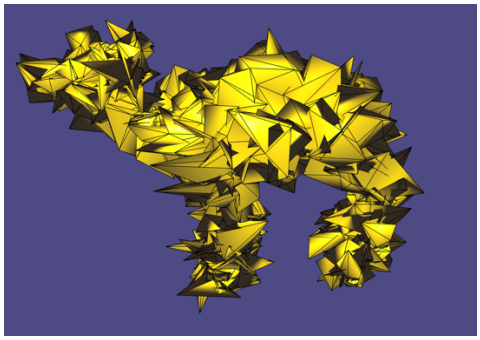
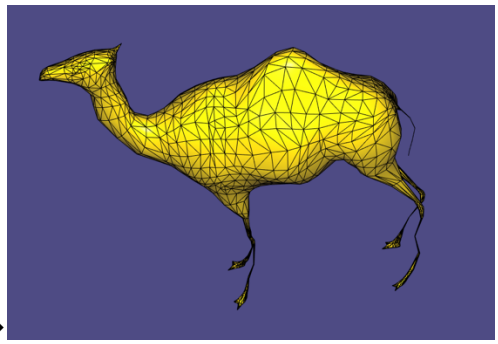
Figure 16 : Chart showing the results of my tests on Laplacian denoising

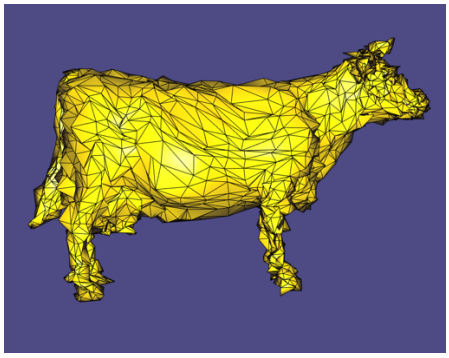
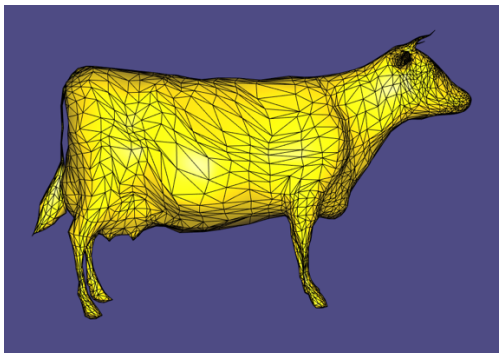
Figure 17 : The results of denoising for large amounts of noise (R : the noise ratio: proportion of the bounding box size; N : number of iterations)

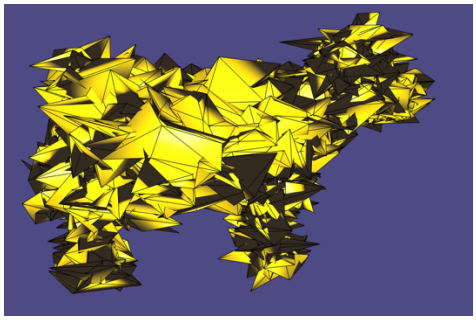
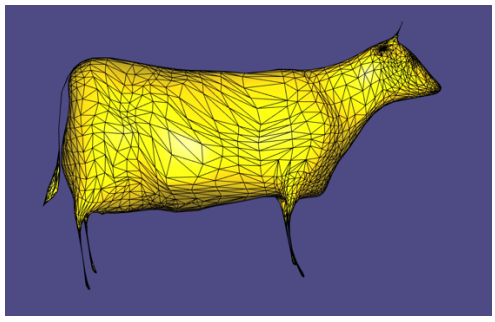
R = 0.01  → N = 5 →  .

R = 0.1  → N = 8 →  → N = 20 →  .

R = 0.01  → N = 9 →  .

R = 0.05  → N = 40 →  .

R = 0.01  → N = 10 →  .

R = 0.05  → N = 40 →  .