Acquisition and Processing of 3D Geometry

Coursework 2

Jiacheng Liu

April 2nd 2018

In this coursework, there are two major parts are required to be completed, including discrete curvature and Laplacian mesh smoothing. In the aspect of discrete curvature and spectral meshes, we need to compute Mean and Gaussian Curvature of various 3D meshes. Both uniform and cotangent discretization are applied to estimate curvature and the results comparison between these two algorithms needs to be indicated as well. Meanwhile, the algorithm that could realize the reconstruction of meshes based on certain number of eigenvectors of the discrete Laplace-Beltrami operator should be implemented. In the second part related to Laplacian mesh smoothing, we are required to perform smoothing in both explicit and implicit format. In addition, the performance of smoothing algorithms will be tested on meshes with different noise level as well.

The list of questions covered by this coursework could be seen as below,

- **Discrete Curvature and Spectral Meshes**

1. **Uniform Laplace:** Both mean curvature and Gaussian curvature were estimated based on Laplacian operator and uniform discretization. 1/3 of the area sum of neighboring faces was used for area computation.

2. **Non-Uniform Laplace:** Using cotangent discretization to discretizing Laplace-Beltrami for mean curvature estimation.

3. **Mesh Reconstruction:** Reconstruct meshes based on k smallest eigenvectors of discrete Laplace-Beltrami operator compute above. Use k = 5, 10, 30.

- **Laplacian Meshes Smoothing**

4. **Explicit Laplacian Mesh Smoothing:** Use explicit Laplacian mesh smoothing algorithm and indicate good λ for this algorithm. Additionally, note the results for large λ steps.

5. **Implicit Laplacian Mesh Smoothing:** Implement implicit Laplacian mesh smoothing algorithm and compare results with explicit one.

6. **Denosing:** Add different levels of noise on meshes to evaluate performance of above smoothing algorithms.

Part 1: Discrete Curvature and Spectral Meshes

1. Uniform Laplace

The definition of mean curvature and Gaussian curvature could be expressed as below,

$$H = \frac{\kappa_1 + \kappa_2}{2} \quad , \quad K = \kappa_1 \cdot \kappa_2$$

where $\kappa_1 = \max_{\phi} \kappa_n(\phi)$ and $\kappa_2 = \max_{\phi} \kappa_n(\phi)$.

In functions of Laplace Operator, above equations could be expressed as follows alternatively,

$$-2Hn = \Delta_s x \quad , \quad K = \frac{2\pi - \sum_j \theta_j}{A}$$

In order to color code the mesh vertices using mean curvature and Gaussian curvature separately, `igl::jet` function was applied to produce color bar with range from blue to red to indicate changes of curvature coefficients.

1.1. Mean Curvature

As the first step for mean curvature calculation, the uniform Laplacian operator should be constructed. In this algorithm, I defined it as a sparse matrix in double-precision format and it was filled with the distance between current vertex and the average of its neighbors. Each element satisfies following equation,

$$\Delta_{uni} f(v_i) := \frac{1}{|N_1(v_i)|} \sum_{v_j \in N_1(v_i)} (x_j - x_i)$$

During this procedure of Laplacian operator construction, I retrieved and stored all neighbors of each vertex included in certain mesh into vector lists. Each row in Laplacian matrix represents the corresponded vertex in mesh hence each element was filled with corresponded weights, which were -1 for diagonal ones and $\frac{1}{|N_1(v_i)|}$ for indices of neighboring vertices.

After the construction of Laplacian operator, we could compute the mean curvature magnitude by using equation

$$H = \frac{1}{2} ||\Delta_s x||$$

Moreover, the sign of mean curvature needed to be evaluated as well. The sign is positive when the surface gradient and normal have the opposite direction, whereas the sign is negative when the surface gradient and normal have the same direction.

The specific algorithm could be seen as `discreteCurvature::uniform_laplacian()` and `discreteCurvature::mean_curvature()` in `discreteCurvatur.hpp`.

The results of mean curvature for meshes in this part could be seen by pressing *Mean Curvature* button within *Uniform Discretization* block. Additionally, users could select three different meshes for initialization and experiments (Figure 1).

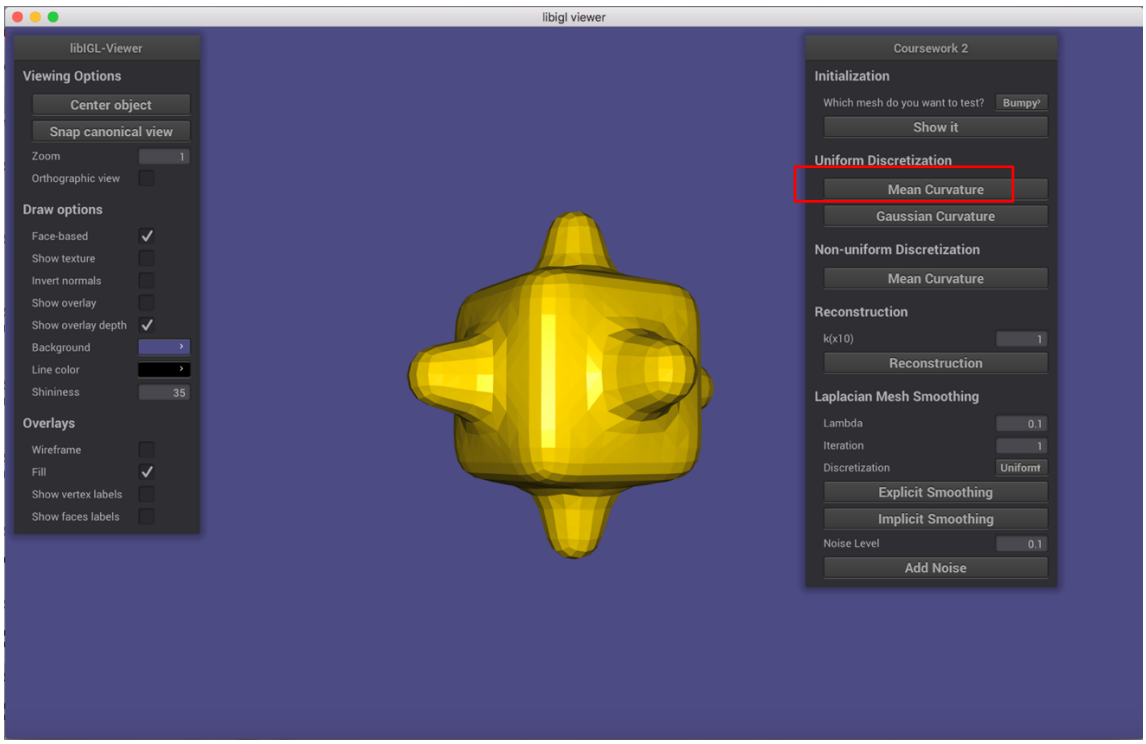


Figure 1: Interface for mean curvature based on uniform discretization

The test results for three meshes could be seen as Figure 2.

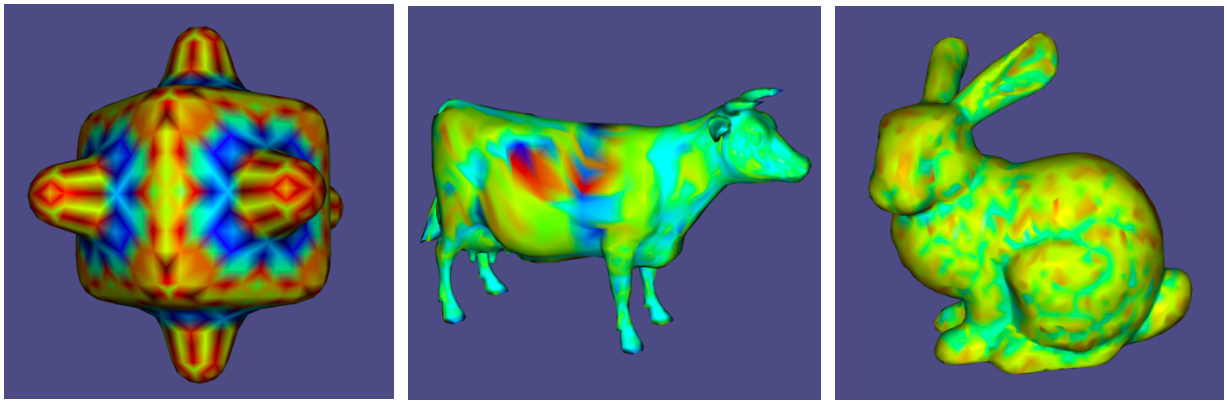


Figure 2: Mean curvature for bumpy.off (left), cow.off (middle) and bunny.off (right)

Above figures illustrate quite noisy results. On comparative planar parts of meshes, we observed some important variations rather than expected smooth results. This means that mean curvature based on uniform Laplacian matrix has quite poor approximation for irregular triangulation, which is resulted from that such algorithm only consider about the connectivity with neighbors. Additionally, such algorithm could give non-zero H for planar meshes and cause tangential drift for mesh smoothing though it is simple and efficient. Therefore, mean curvature is a poor approximation for continuous curvature.

In addition, by zooming in the viewer we could find that the consistency of color code for convex and concave region of meshes, which use warm or color style to indicate the sign of mean curvature. In particular, on regions with local maximum or minimum curvature, such as rather non-planar portions, color code with higher saturation is observed which presented mean curvature with high magnitude.

1.2. Gaussian Curvature

To compute Gaussian curvature, I retrieved each mesh vertex and find its one-ring neighbors for its angle deficit and area sum calculation. Assume the current vertex is V_0 and the other two vertices included in one of V_0's surrounding faces are V_1 and V_2, then the angle θ_j could be expressed as

$$\theta_j = acos\frac{V_0V_1 \cdot V_0V_2}{|V_0V_1| \cdot |V_0V_2|}$$

Next, we could compute the area of surrounding triangle,

$$A_j = \frac{1}{2}|V_0V_1| \cdot |V_0V_2| \cdot \sin(\theta_j)$$

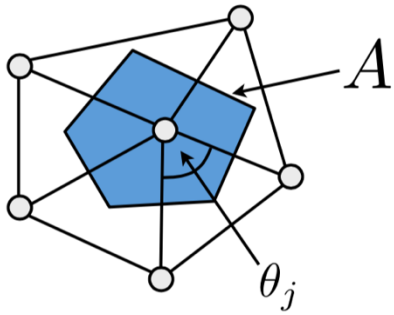


Figure 3: Mesh vertex and its one-ring neighbors

Area factor A used for normalization is obtained by using below equation, which is named as Barycentric Cells. It works by connecting midpoints and triangle barycenters.

$$A = \frac{1}{3}\sum_j A_j$$

The specific codes for this task could be found in `discreteCurvature::Gaussian_curvature()`

in `discreteCurvatur.hpp`.

To estimate Gaussian curvature for prepared meshed, we need to press the button called *Gaussian Curvature* within *Uniform Discretization* block. Results for three meshes could be seen as Figure 4.

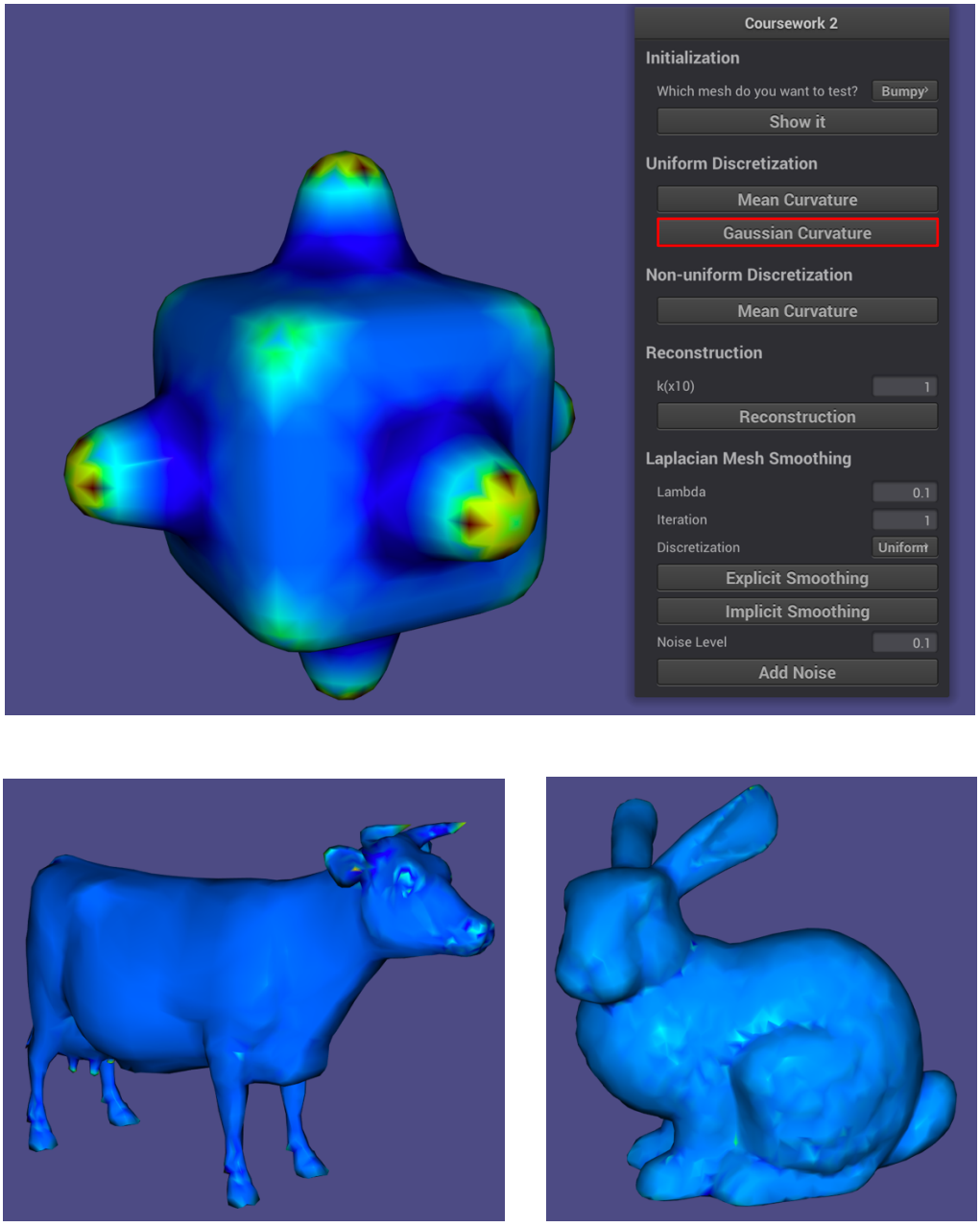


Figure 4: Gaussian Curvature bumpy.off (top), cow.off (left) and bunny.off (right)

It could be found that the results of mesh *bumpy.off* shows the change of Gaussian curvature best, while *cow.off* and *bunny.off* show almost blue color for estimated Gaussian curvature, hence we'll do the analysis based on *bumpy.off* mainly.

Different from mean curvature we computed in last task, colored mesh based on Gaussian curvature is very coherent. Additionally, points on peak and corner regions are shown with brighter color, which represents Gaussian curvature with high magnitude, whereas points on planar and cylindric-like regions are displayed

with almost blue color with slight different levels. This is because that Gaussian curvature is defined as the production of minimum and maximum curvature, only points with both high maximum and minimum curvature are able to have high Gaussian curvature. For example, for points at the edge in *bumpy.off*, the minimum curvature is zeros hence the Gaussian curvature will be low. This explanation could be proved in *cow.off* as we could find that the Gaussian curvature for regions of cow horn, peaks of ears and udder is rather high by observing brighter color. Therefore, discrete Gaussian curvature is not an enough good approximation for continuous curvature, though it is better than mean curvature estimated based on uniform discretization.

2. Non-uniform (Discrete Laplace-Beltrami)

In this task, we were required to estimate mean curvature based on cotangent discretization. The only difference of this part from above part is the way to construct Laplacian filter.

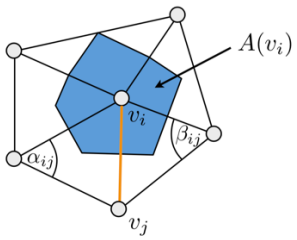


Figure 5: Cotangent Discretization

The Laplacian matrix is defined based on the production of two matrix,

$$L = M^{-1}C$$

where

$$M = diag(2A_j) \quad \text{and} \quad C = \begin{cases} \cot(\alpha_{ij}) + \cot(\beta_{ij}), & where\ i \neq j,\ j \in N_1(v_j) \\ -\sum_{v_j \in N_1(v_j)} (\cot(\alpha_{ij}) + \cot(\beta_{ij})), & where\ i = j \\ 0, & otherwise \end{cases}$$

In order to obtain α_{ij} and β_{ij} used in C matrix, I used function `igl::triangle_triangle_adjacency()` to retrieve adjacent triangles. There are two matrices returned by this approach, which store neighboring triangles and corresponded edges respectively. Angles are calculated by using the equation described in last part and the area factor for normalization is estimated as $1/3$ of neighboring triangle area sum (Barycentric Cells), which is simpler to compute. The normalization methods such as Voronoi Cells and Mixed Cells, which may achieve better approximate though complex to compute, could be considered in the further work.

The specific algorithm was written as `discreteCurvature::nonUniform_laplacian()` in `discreteCurvatur.hpp`.

The mean curvature for meshes based on cotangent discretization could be obtained by pressing button *Mean Curvature* within *Non-uniform Discretization* block. Test results could be seen as below.

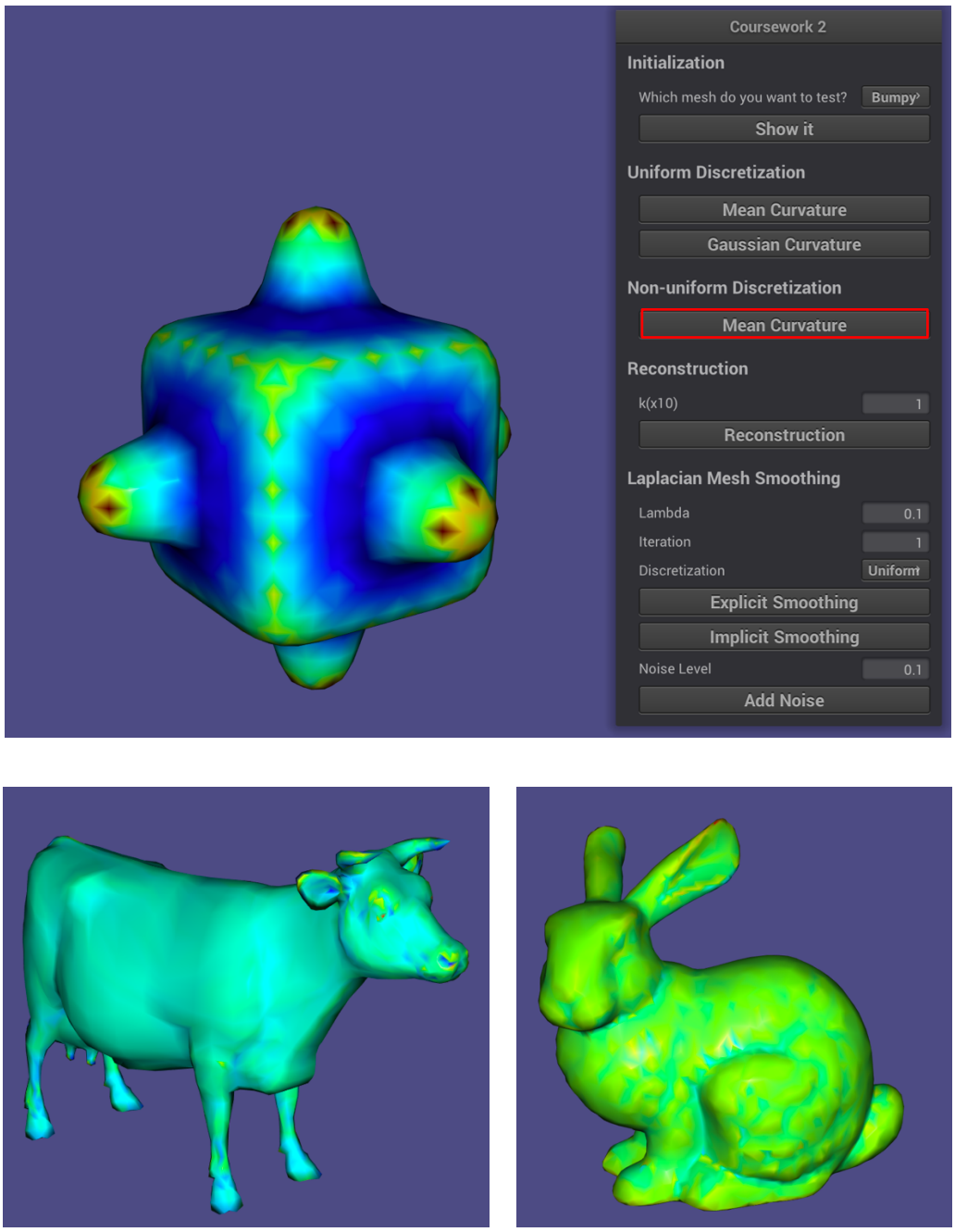


Figure 5: Discrete Mean Curvature for bumpy.off (top), cow.off (left) and bunny.off (right)

From above figure, we could find that results are improved much and appear to be more coherent and consistent than those obtained based on uniform discretization. Indeed, mean curvature is defined as the average of minimum and maximum curvature, hence it will be medium if one of them is high and another is low, while it will be low if both minimum and maximum curvature are low, high if both of them are high. Similar to above part, *bumpy.off* is the mesh that visualize variation of curvature best. The edge and corner of cube and peak of cylinder-like region are displayed with high response, whereas flat surface is shown with low response, especially for connect portion between flat surface and cylinder-like region. Additionally, the artifacts caused by irregular triangulation disappear on meshes as well. Sides of both cow and bunny are coherent and smooth.

Therefore, using cotangent discretization to estimate mean curvature is a better approach than applying uniform discretization for continuous curvature approximation.

3. Mesh Reconstruction

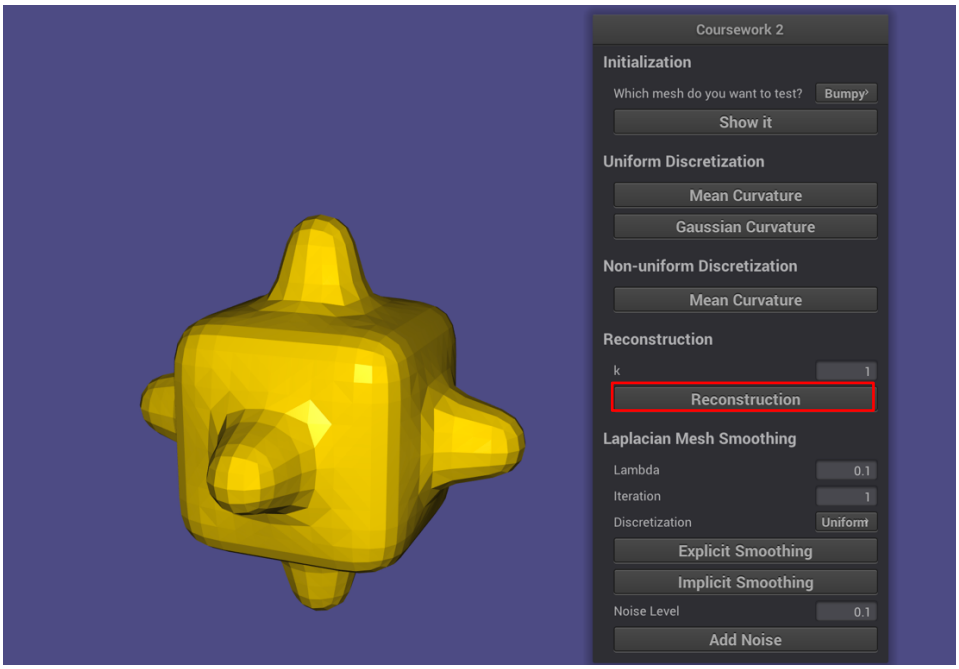
In this part, we are required to reconstruct meshes based on certain number of smallest eigenvectors of discrete Laplace-Beltrami operator. The reconstruction equations could be expressed as below.

$$x := [x_1, \dots x_n] \quad y := [y_1, \dots y_n] \quad z := [z_1, \dots z_n]$$

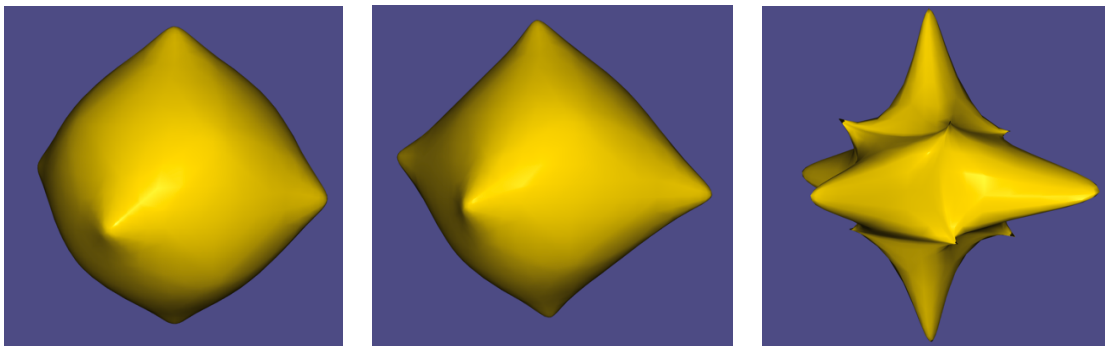
$$x \leftarrow \sum_{i=1}^{k} (x^T e_i) e_i \quad y \leftarrow \sum_{i=1}^{k} (y^T e_i) e_i \quad z \leftarrow \sum_{i=1}^{k} (z^T e_i) e_i$$

Where e_i represents the i th smallest eigenvector of the discrete Laplace-Beltrami operator. To get eigenvectors of computed Laplacian operator, I used SPECTRA library in C++ to compute certain number of smallest eigenvectors of sparse matrix. The algorithm for mesh reconstruction was written as `discreteCurvature::eigen_reconstruction()` in `discreteCurvatur.hpp`.

The reconstruction algorithm could be run by clicking the button *Reconstruction*. Additionally, the value of *k* could be modified. The reconstruction results with k=5, 10 and 30 could be seen as below.



(a) Original mesh



(b) k=5 (c) k=10 (d) k=30

Figure 6: Reconstruction results for 'bumpy.off'

From above results, we could find that the reconstructed mesh becomes closer to the original one with the increase of k. The outline and structure of results is becoming obvious. However, the mesh seems to be ‘stretched’ too much when k=30 though it contains most of important features of the original one. Additionally, it is clear that 30 smallest eigenvectors are not enough to reconstruct this mesh with enough details and the possible number may be over than 100.

Part 2: Laplacian Mesh Smoothing

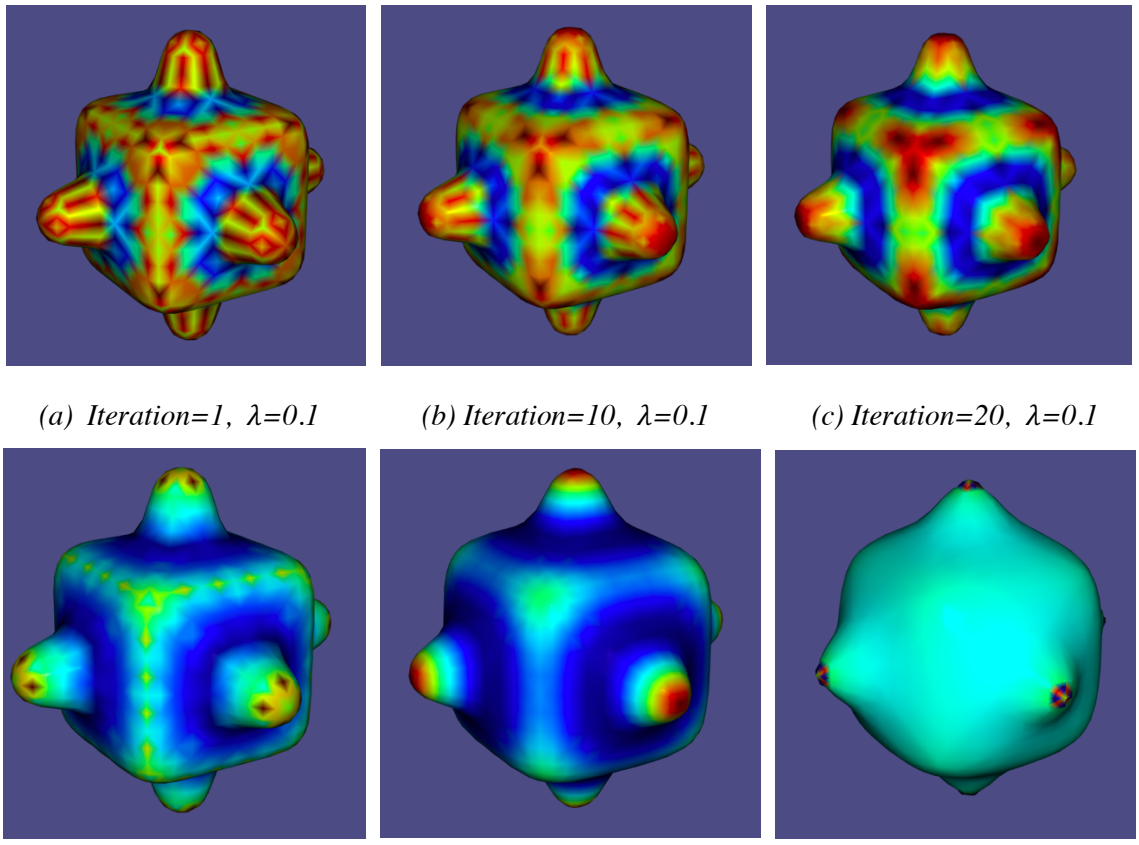
4. Explicit Laplacian Mesh Smoothing

By using the Laplacian operator, we have already computed, mesh smoothing algorithm could be implemented. The explicit Laplacian mesh smoothing algorithm update vertex based on previous version directly,

$$P^{t+1} = (I + \lambda L)P^t$$

where λ represents a kind of step size. In the explicit integration, λ should be small enough to ensure stability. In this task, I used both uniform and cotangent discretization to construct Laplacian operator for smoothing experiment to observe the differences. The specific algorithm could be seen as `Smoothing::explicit_smoothing()` in `Smoothing.hpp`.

The smoothing results could be shown by clicking the button *Explicit Smoothing* within *Laplacian Mesh Smoothing* block. Moreover, the certain value of λ step size and iteration count could be modified. The type of discretization could be selected as well through the list named *Discretization*. The test results could be seen as below.



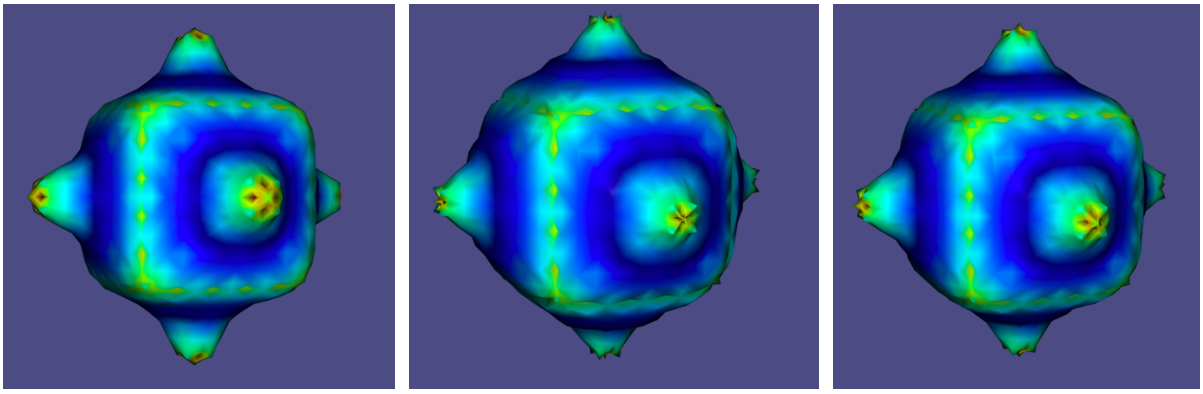
(a) Iteration=1, λ=0.1 *(b) Iteration=10, λ=0.1* *(c) Iteration=20, λ=0.1*

(d) Iteration=1, λ=0.01. (e) Iteration=10, λ=0.01 (f) Iteration=20, λ=0.01

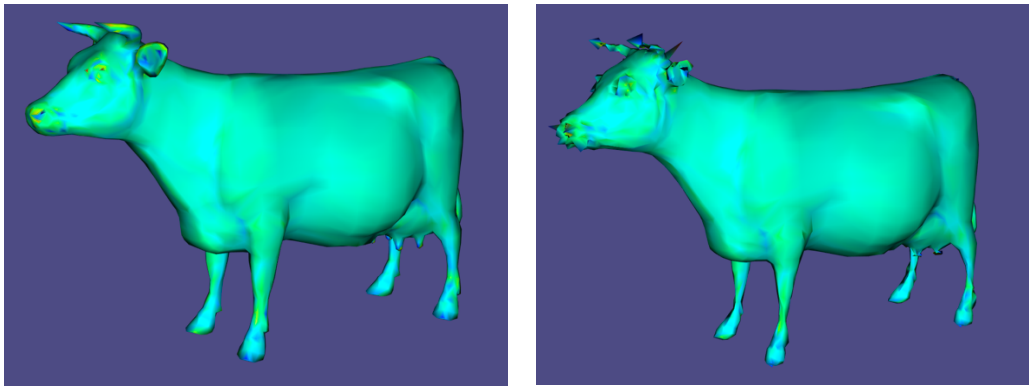
Figure 7: Explicit smoothing results based on uniform (first row) and cotangent discretization (second row) for "bumpy.off"

On the one hand, we notice that the step size of λ need to be higher for smoothing algorithm based on uniform discretization than that based on cotangent discretization. This could be explained that the neighboring weights are bigger in cotangent discretization than that in uniform discretization, hence vertices those been applied by smoothing algorithm with cotangent discretization have fast movement speed with smaller λ step size. On the other hand, smoothing algorithm with cotangent discretization has stronger smoothing ability than the one with uniform discretization. This is because uniform discretization only considers connectivity between neighbors, whereas cotangent discretization also thinks about angles in addition to connectivity.

Additionally, from Figure 7(f) we observe that there are some small artifacts appear at peaks of cylinder-like regions. These artifacts are easier to appear when λ step size is large. There are some examples shown below. In this part, only smoothing algorithm with cotangent discretization will be focused.



(a) λ=0.1. (b) λ=0.15 (c) λ=0.2



(d) λ=0.00001 (e) λ=0.0001

Figure 8: Explicit smoothing results with single iteration

We could find that the both number and size of artifacts increase with the rise of λ step size. The bigger λ step size, the mesh diverges more quickly and more portions overlap with each other to cause artifacts. Therefore, for explicit Laplacian smoothing algorithm, λ step size should be small enough to reduce artifacts.

The 'perfect' λ step size for prepared meshed could be seen as Table 1. In this case, I defined perfect size which could ensure certain could be smoothed with cotangent discretization without artifacts for at least 10 iterations. It could be found that *cow.off* and *bunny.off* has much smaller 'perfect' λ step size than *bumpy.off*. This is because the size of these two meshes is much smaller than that of *bumpy.off* and λ step size is different

based on actual condition of corresponded mesh.

Table 1: 'Perfect' λ step size for meshes

Meshes	**Bumpy.off**	**Cow.off**	**Bunny.off**
$\boldsymbol{\lambda}$	0.01	1×10^{-6}	5×10^{-7}

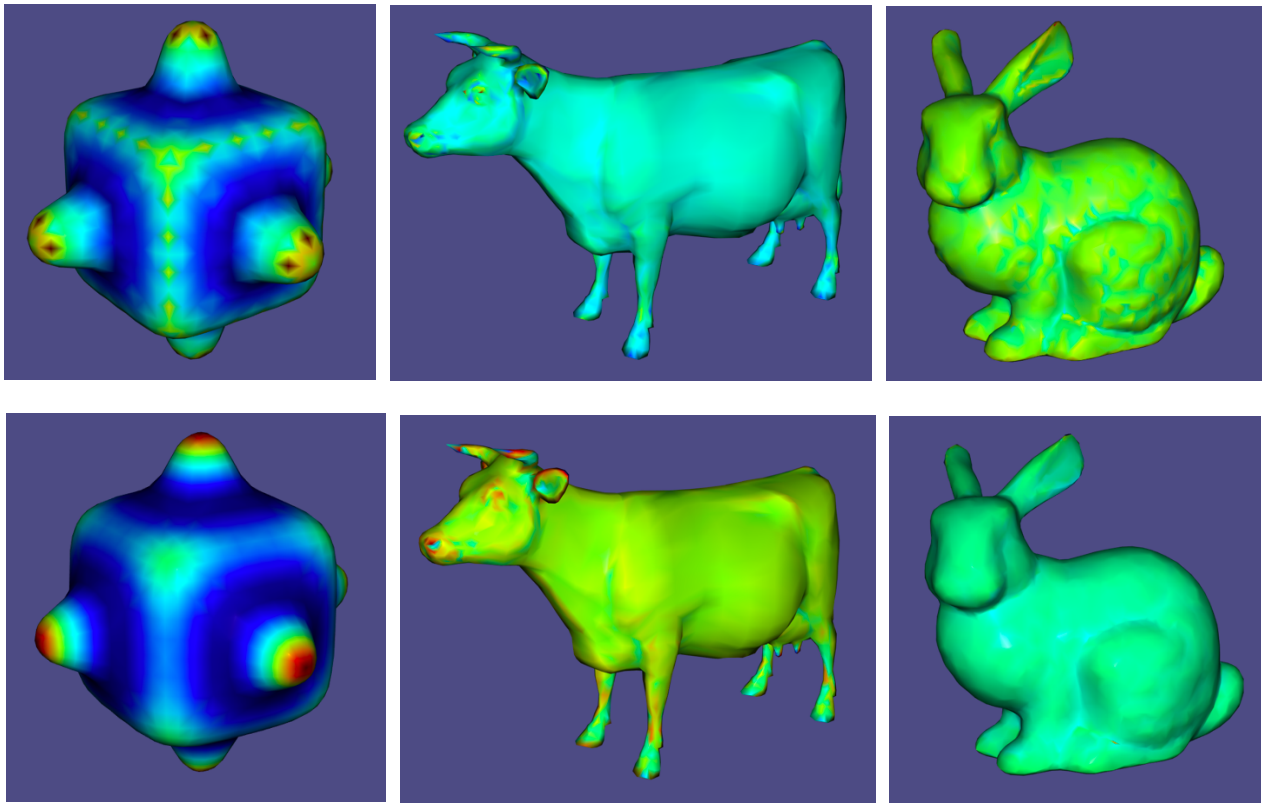


Figure 9: Meshes before (first row) and after (second row) smoothing for 10 iterations

5. Implicit Laplacian Mesh Smoothing

The equation for implicit Laplacian mesh smoothing algorithm could be expressed as below,

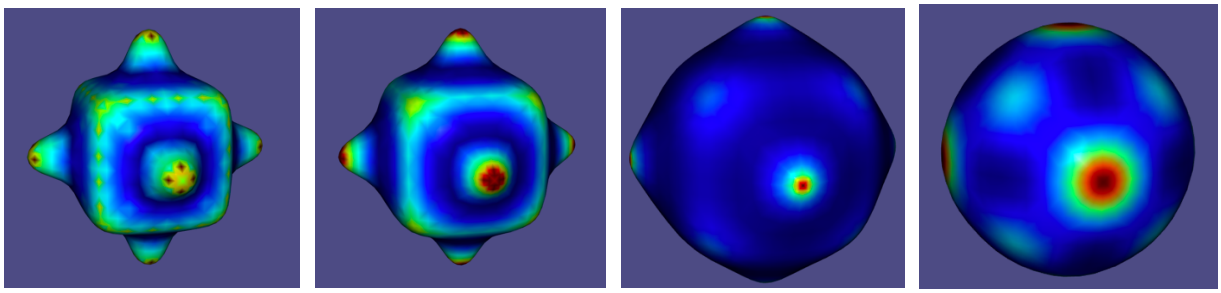
$$(I - \lambda L)P^{t+1} = P^t$$

Compared to explicit integration, implicit integration is unconditionally stable and we do not need to consider about large λ problem. To get the answer of the equation, we need to solve a linear system. However, Laplacian operator is not symmetric in cotangent discretization hence the calculation is quite expensive. To improve the computation efficiency, we can write L as $M^{-1}C$, where M is symmetric. Then the above equation could be rewritten as

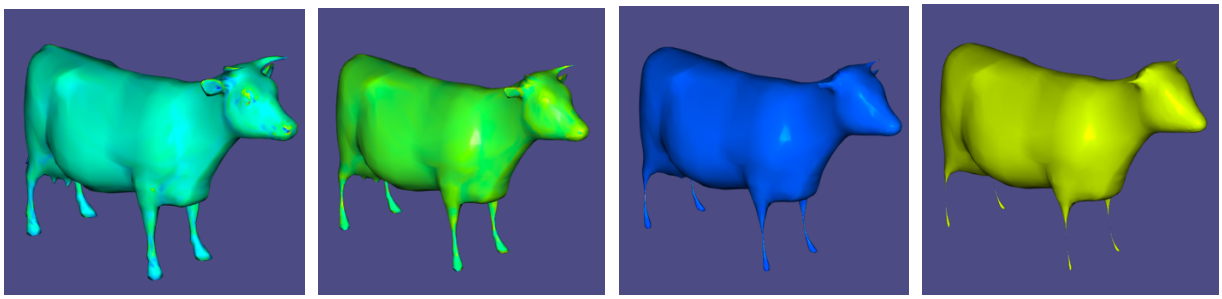
$$(M - \lambda ML)P^{t+1} = MP^t$$

In this algorithm, I used `Eigen::SimplicialCholesky()` to solve sparse symmetric positive definite system.

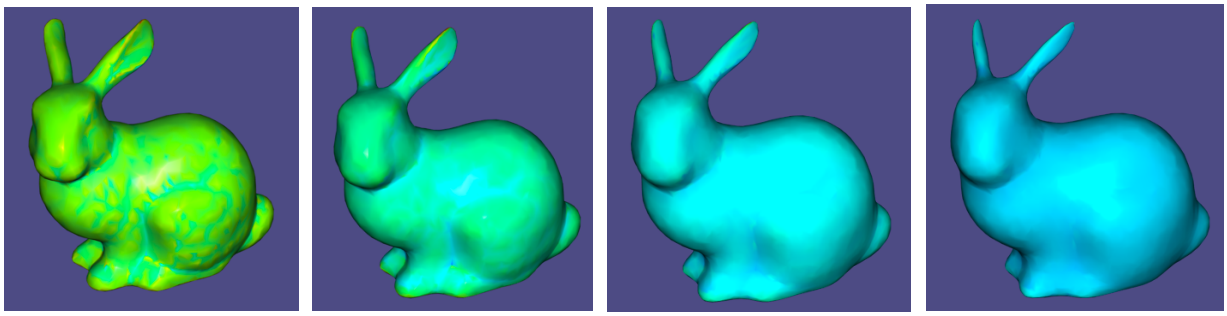
This algorithm is written as `Smoothing::implicit_smooth()` in `Smoothing.hpp`. Users are allowed to use implicit smoothing algorithm on meshes by clicking button *Implicit Smoothing* within *Laplacian Mesh Smoothing* block. Some results are shown as follows.



(a) Iteration=1. (b) Iteration=2. (c) Iteration=5. (d) Iteration=10



(e) Iteration=1. (f) Iteration=2. (g) Iteration=3. (h) Iteration=4



(i) Iteration=1. (j) Iteration=2. (k) Iteration=3. (l) Iteration=4

Figure 10: Implicit smoothing results with λ*=1 (first row),* λ*=0.001 (second row) and* $\lambda = 1 \times 10^{-5}$*(last row)*

With larger λ step sizes, meshes become smoother with fewer iterations and without any artifacts compared with explicit Laplacian smoothing we did in last task. Mesh *bumpy.off* almost becomes a sphere after 10 iterations. For meshes *cow.off* and *bunny.off* which have more complex structure, the implicit Laplacian smoothing algorithm also works successfully. From Figure 10, we can find that details and textures reduce with the increase of iteration with fast speed. As one of results for smoothing, the volume of meshes also reduces slightly. For instance, we can see that cow legs are stretched and narrowed increasingly with the rise of iteration. In conclusion, implicit approach has following three advantages over explicit approach: being more stable, allowing large λ step size and requires fewer iterations.

6. Mesh Denoising

To test the performance of implemented explicit and implicit smoothing algorithms, I programmed a function named `Smoothing::add_noise()` to add synthetic noise on selected mesh. This function could be called by pressing button *Add Noise* and the noise level could be adjusted by modifying values in corresponded test filed. In this section, only smoothing algorithm with cotangent discretization will be tested.

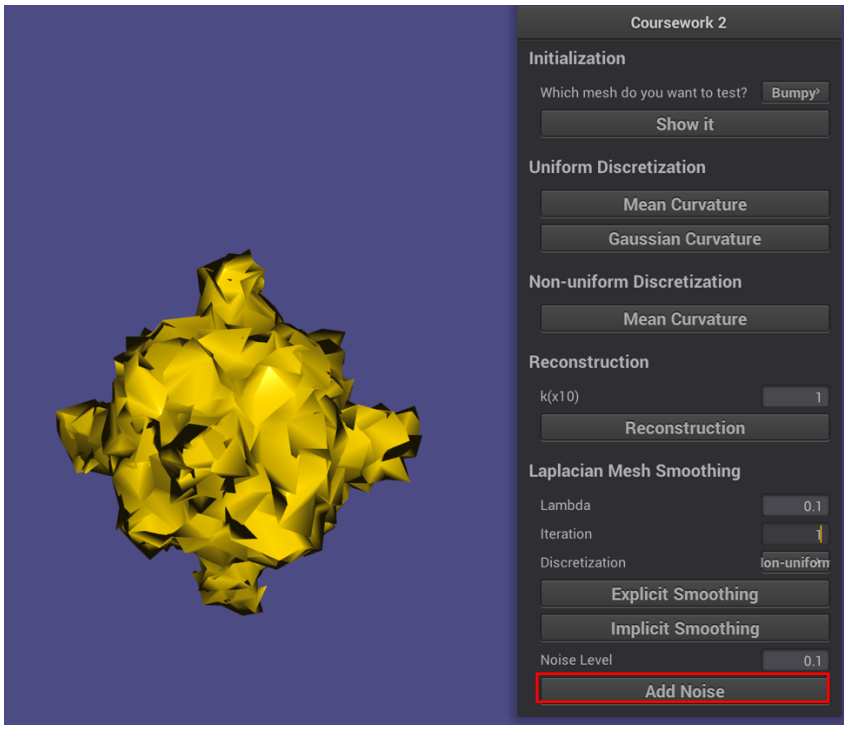
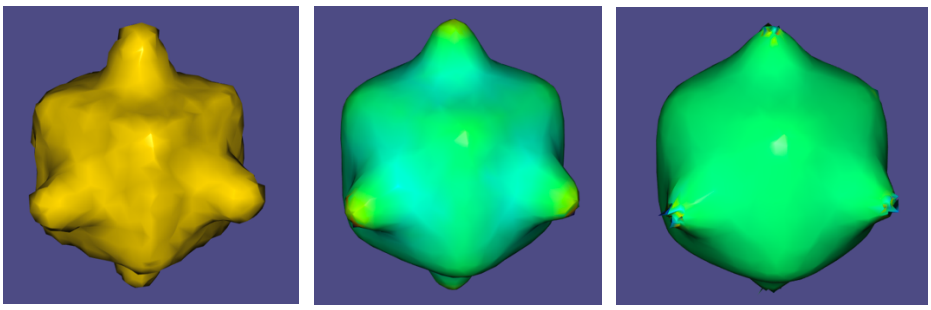


Figure 11: Interface for Denoising

6.1. Denoising with Explicit Laplacian Smoothing Algorithm

To evaluate the denoising ability of explicit smoothing algorithm, few amount of noise ($f = 0.02$) was applied on meshes. Results with small λ steps after several iterations could be seen as Figure 12.



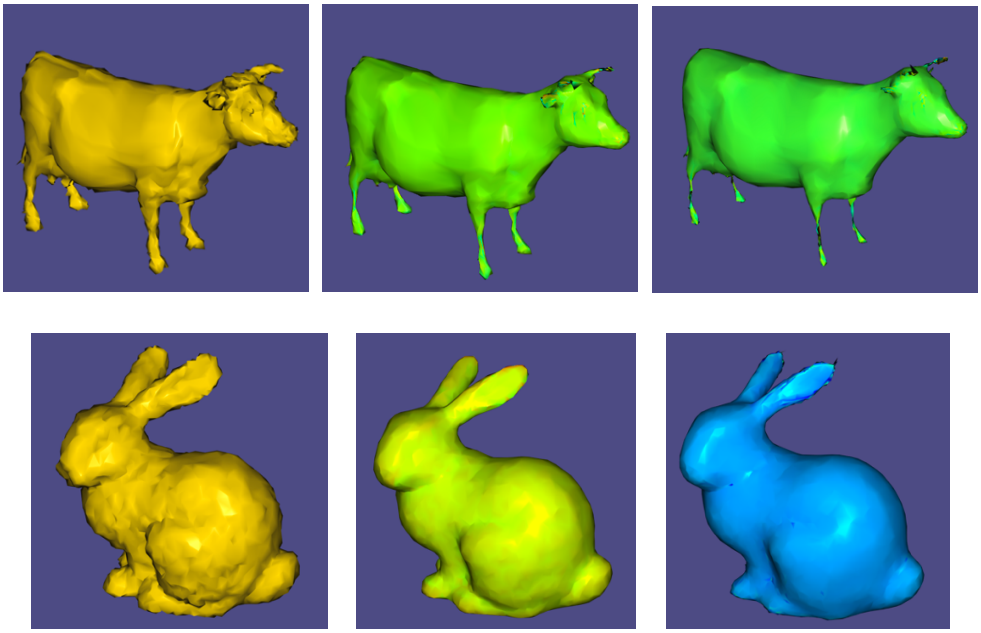
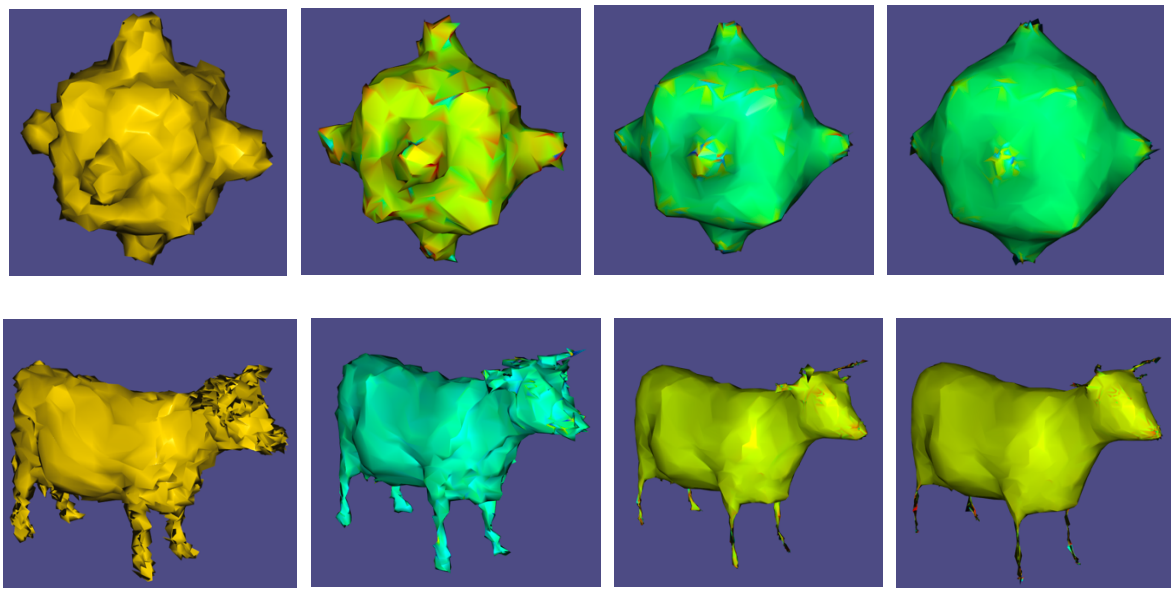


Figure 12: Denoised results for explicit smoothing algorithm with f=0.03

From above figure, we can observe that some artifacts appear though most parts of mesh have been smoothened. With the increase of iteration number, both amount and size of artifacts grows at some peaks of meshes and we can find out that many irregular triangles appear at these portions. This is one of significant problems that reduce the accuracy of explicit denoising.

Then I added noises with higher noise level ($f = 0.05$) and the final results could be seen as below. It could be observed that stronger artifacts appear with the rise of iteration number. Therefore, explicit Laplacian smoothing algorithm works when λ step size is small and iteration number is not too high; otherwise, it will fail. Meanwhile, with the increased strength of added noise, the λ step size should be increased as well to achieve similar smoothing effect as it does in some noisy meshes with low noise level.



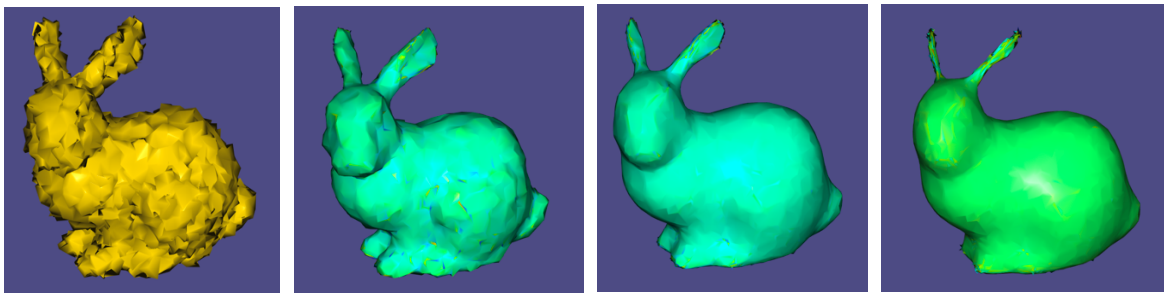


Figure 13: Denoised results for explicit smoothing algorithm with f=0.05

6.2. Denoising with Implicit Laplacian Smoothing Algorithm

In this part, we carried out similar evaluation to measure denoising ability of implicit Laplacian smoothing algorithm. Firstly, noises with $f = 0.05$ was allied on meshes and the results that smoothened by implicit Laplacian smoothing algorithm could be seen as follows.

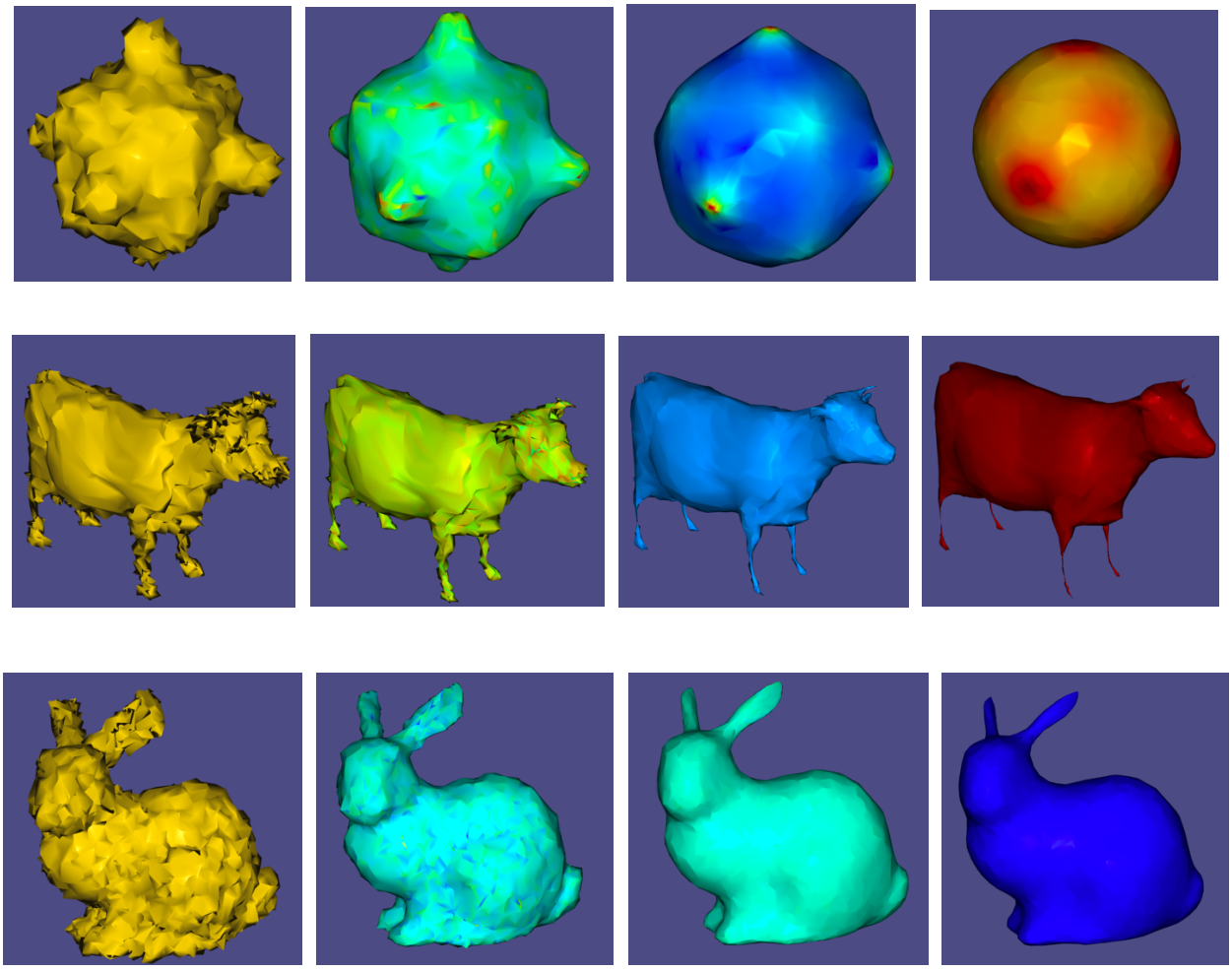


Figure 14: Denoised results for implicit smoothing algorithm with $f = 0.05$

From above figure, we could find out that implicit smoothing algorithm had stronger denoising ability than

explicit one. There is no artifact appear in denoised meshes. However, too strong smoothing is easier to shrink or even eliminate edges and peaks. For example, figures on the first row of Figure 15 illustrate that the bumpy mesh is smoothened into nearly a sphere within a few iterations.

Next, we added stronger noises with $f = 0.1$ on meshes and denoised results are shown as Figure 15. It could be found that implicit algorithm still works for large amount of noises. However, more details of meshes also lost during smoothing. For example, the lower parts of cow legs disappear much with the increased iteration. This problem could be improved by reducing the size of λ step to decrease the smoothing rate within single iteration.

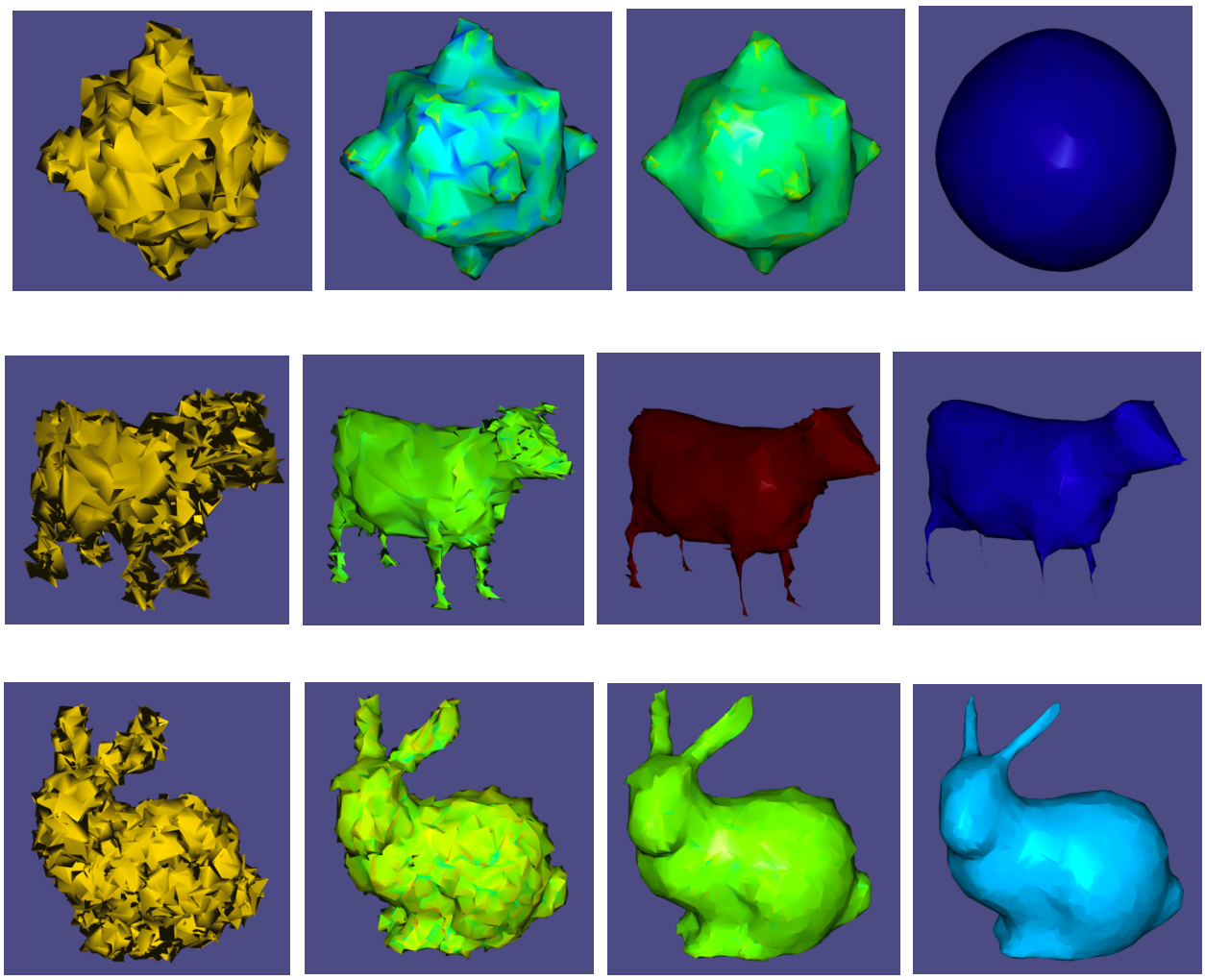


Figure 15: Figure 14: Denoised results for implicit smoothing algorithm with $f = 0.1$