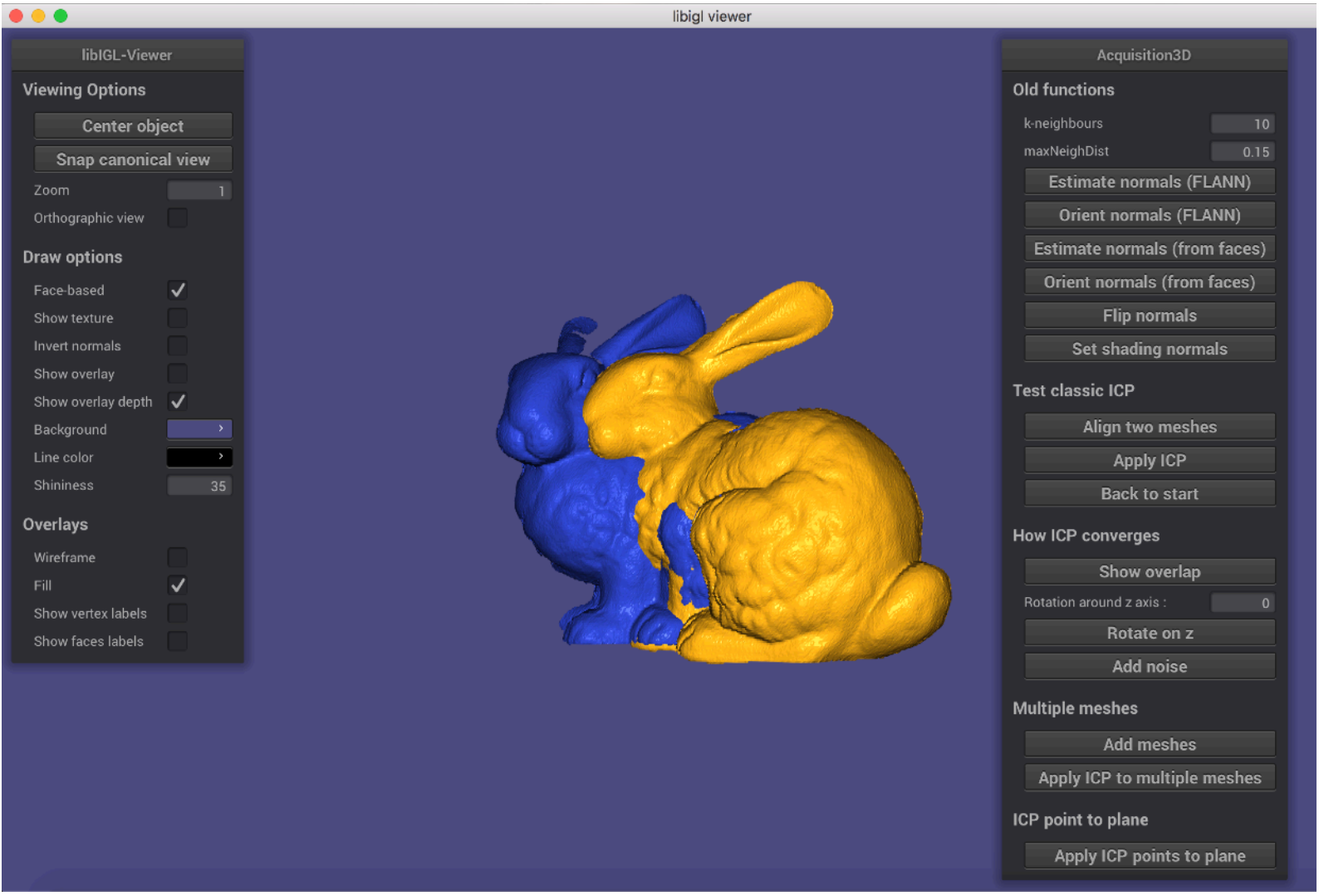
# Assigment 1 : ICP algorithms

COMPGV18/COMPM080

**Task 1 :**

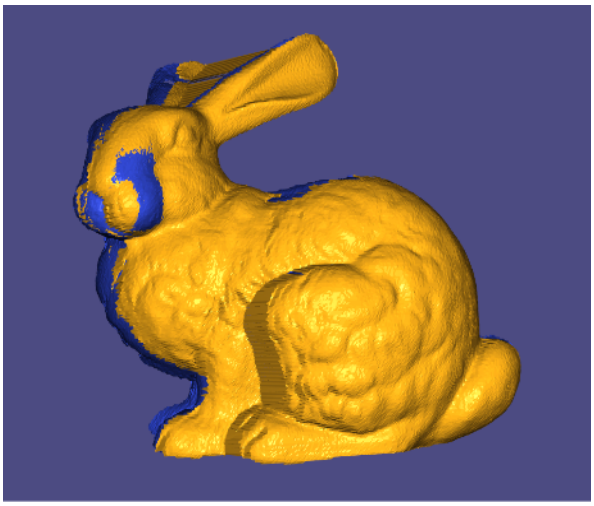
For this task, I used the given framework and I modified the code in it. I also changed the interface to be able to apply my new functions.
Here is the new interface :




Then, I used 2 mains meshes to test my algorithms : the first fixed cloud was the bunny000.ply that I parsed into .off (using meshlab). And I tried to align the second bunny045.off to it.
All my functions are in the « decoratedCloud.cpp » file because most of them apply directly to meshes so I coded them as member functions.
Moreover, the structure of my updated framework works on the cloudManager class. Indeed, I created a cloudManager object to be able to store my original meshes and the transformed ones and the fusion of both to visualize. This way I can use all the meshes in the viewer more easily.

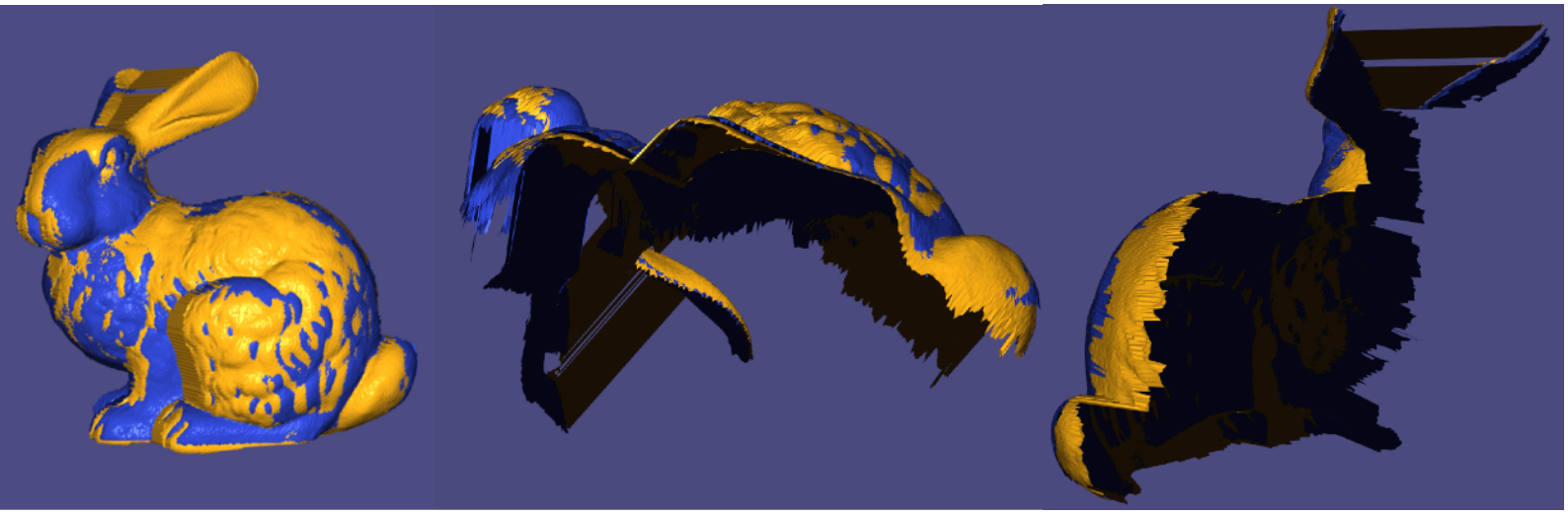
First I implemented a function to be able to rotate and translate the meshes with a rotation matrix sized 3*3 and a vector of translation sized 3*1. It's the «transformation(Matrix3d R, Matrix<double, 3, 1> T) » function. I added a button « align two meshes » to nearly aligned them and be able to apply my ICP algorithm (only the yellow bunny is moving)

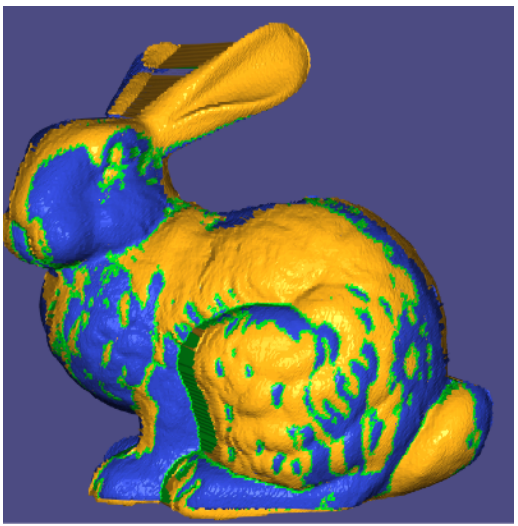
Here is my result :



Then, I implemented point-to-point ICP to merge and to better align them. To do that, I used the ANN library which will find the closest correspondance point between the fixed mesh and our moving mesh. With those correspondances, I can find the rotation and translation matrices. Indeed, by computing the centered point clouds (points-barycentricPoint) then, computing the covariance matrix between the 2 point clouds and using the SVD decomposition, I can find the rotation matrix and deduce the translation vector from it.
To be able to see the overlapping parts, I used the function « intersect_other » of the CGAL librairy to show faces which overlap. The result of this function is seen with the button « show overlap ».

Here are my result :





In green we can see the faces which are intersected. I also print the number of intersected faces, in this case 11,9% of the faces are interested after 7 iterations. I'll use that to test how my ICP algorithm converges with sampled meshes or rotated one after.
To reject bad correspondances in my ICP algorithm I computed the dot product between the normals of 2 corresponding points. If this dot product is bellow 0.98, the points are rejected (the angle between the normals is too important to be considered as a good match). Points are also rejected if the distance between them is too large. The program prints the pourcentage of how much points are selected as good matches over the number of points in the cloud. Logically, as the meshes get closer, this pourcentage increases to reach around 30% when meshes are closed. I think this number should be higher but as computing normals isn't easy it could lead to some errors and explain this low pourcentage. However, even with the result the algorithm seems to work well.

This part used to be quite fast to compute but, when adding the CGAL librairy, functions took much longer to show results even if this librairy is not called. I wasn't able to solve this problem.
We can see that some misalignments remain, on the legs of the bunny for example, but more iteration doesn't solve this issue. I think as most points have a good positions, the misaligned part doesn't count enough to make the bunny moves. I think this issue could be solved by adding a weight to a corresponding pair.

Here is how we should compute the rotation and translation matrices with a weight in the energy function :
We want to minimize this energy function :

$$E(R,T) = \sum_i \left\| w_i (Rp_i + t - q_i)^2 \right\|$$

To start, we take the derivative with respect to T :

$$\frac{d(E(R,T))}{dt} = \sum_i 2w_i(Rp_i + t - q_i) = 0$$

And set it to zero, which gives (as 2 can't be equal to zero) :

$$\frac{d(E(R,T))}{dt} = \sum_i w_i(Rp_i + t - q_i) = 0$$

Then ,

$$\frac{d(E(R,T))}{dt} = R\sum_i w_i p_i + nt - \sum_i w_i q_i = 0$$

By setting $\bar{p} = \frac{\sum_i w_i p_i}{n}$ $\bar{q} = \frac{\sum_i w_i q_i}{n}$ we obtain : $t = \bar{q} - R\bar{p}$

Then, we need to find R. We can remplace T by its value :

$$E(R,T) = \sum_i \|w_i(Rp_i + \bar{q} - R\bar{p} - q_i)^2\|$$

As it doesn't depend on T anymore we have :

$$E(R) = \sum_i \|w_i(R(p_i - \bar{p}) - (q_i - \bar{q}))^2\|$$

We can set :

$$\tilde{p_i} = p_i - \bar{p}$$
$$\tilde{q_i} = q_i - \bar{q}$$

To obtain :

$$E(R) = \sum_i \|w_i(R\tilde{p_i} - \tilde{q_i})^2\|$$

Which can be written as

$$E(R) = \sum_i w_i(R\tilde{p_i} - \tilde{q_i})^T(R\tilde{p_i} - \tilde{q_i})$$

By developing :

$$E(R) = \sum_i w_i\tilde{p_i}^T\tilde{p_i} - 2w_i\tilde{q_i}^TR\tilde{p_i} + w_i\tilde{q_i}^T\tilde{q_i}$$

We want to minimize this energy, which is equivalent to maximize the middle term (the only one depending on R) :

$$min(E(R)) = max(\sum_i w_i\tilde{q_i}^TR\tilde{p_i})$$

This expression can be rewritten as : $$min(E(R)) = max(tr(Q^TRP))$$

With :

$$Q = \begin{bmatrix} w_1q_{1x} & w_2q_{2x} & \dots & w_nq_{nx} \\ w_1q_{1y} & w_2q_{2y} & \dots & w_nq_{ny} \\ w_1q_{1z} & w_2q_{2z} & \dots & w_nq_{nz} \end{bmatrix} \qquad P = \begin{bmatrix} p_{1x} & p_{2x} & \dots & p_{nx} \\ p_{1y} & p_{2y} & \dots & p_{ny} \\ p_{1z} & p_{2z} & \dots & p_{nz} \end{bmatrix}$$

As tr(AB) = tr(BA), we can say that :

$$min(E(R)) = max(tr(RPQ^T)) = max(tr(RC))$$

With :

$$C = \sum_i w_i(p_i - \bar{p})(q_i - \bar{q})^T$$

It exist a SVD decomposition of C such as $C = U\Sigma V^T$

So, we can write the trace this way, using the same trick :

$$tr(RU\Sigma V^T) = tr(\Sigma V^T RU) = tr(\Sigma M)$$

by defining M (orthogonal matrix by construction as V and U are orthogonal and R as well as it's a rotation matrix).
This way, we are trying to find M such as finding :

$$max(\sum_i \sigma_i m_{ii})$$

As the sigmas are positive, it can be given only with M=I.
So, we arrive to

$$V^T RU = M = I$$

And,

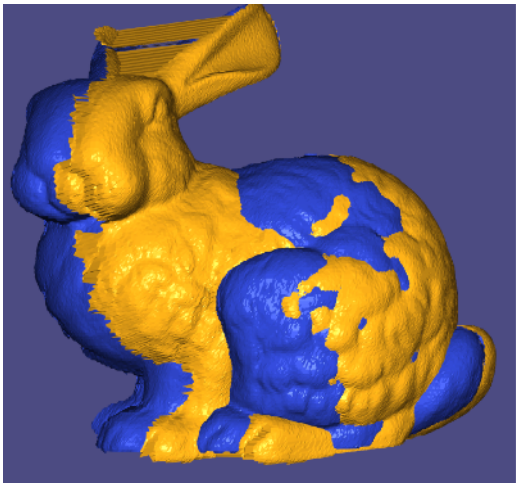
$$R = VU^T$$

So, the way we compute C is different but we also use the SVD decomposition of C to find R.

**Task 2 :**

In this task, I mostly try to show the effect of increasing misalignment by computing an interface to rotate the meshes around the z axis (y in the framework). Indeed, we can chose the wanted rotation and apply it with the button « rotate on z ». After gathering the result we can get back to the original position with the button « back to start » to start over with a different value of rotation. I implemented a function « install(float theta, float x, float y, float z) » which takes the parameters of a rotation around z and a translation to apply it to a mesh. The same think could have been done with the other axis but I though the z one was the most interesting.

Example with a rotation of 10° around z for the yellow mesh



To evaluate the convergence behavior of ICP, I applied several rotation and look the number of intersecting faces after 10 iterations.

Here is my result :

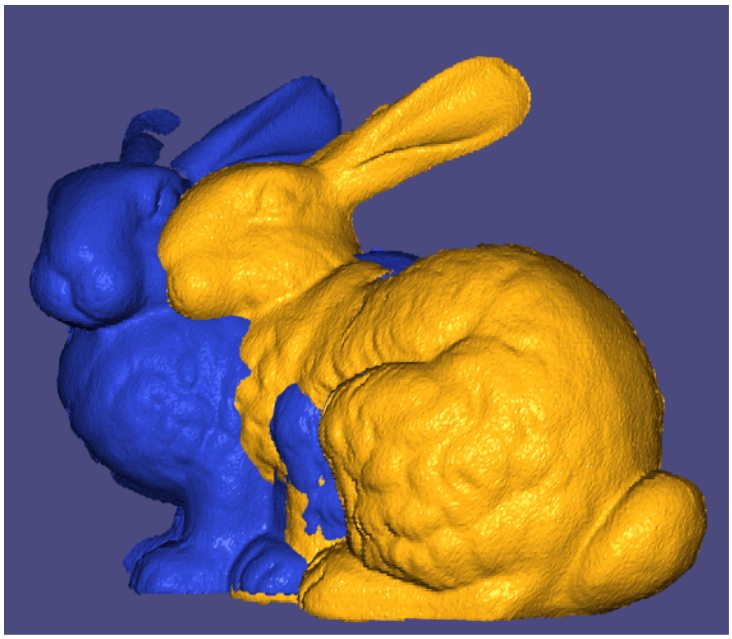
| | -5° | 5° | -10° | 10° | -20° | 20° |
|---|---|---|---|---|---|---|
| Intersecting faces (%) | 8.9 % | 5.8 % | 4.0 % | 3.9 % | 2.1 % | 1.9 % |

We can see that more the angle increases, less the algorithm successes and more iterations doesn't change that. Indeed, the energy function is in a « local minimum » and can't get out of it if the meshes aren't well aligned at the beginning.
Moreover, if the conditions to accept corresponding points in the ICP algorithm are hard to achieve (for example the dot product below 0.98), not enough points will satisfy it to recover from the rotation.
It's why it's important to start the algorithm with 2 nearly aligned meshes.

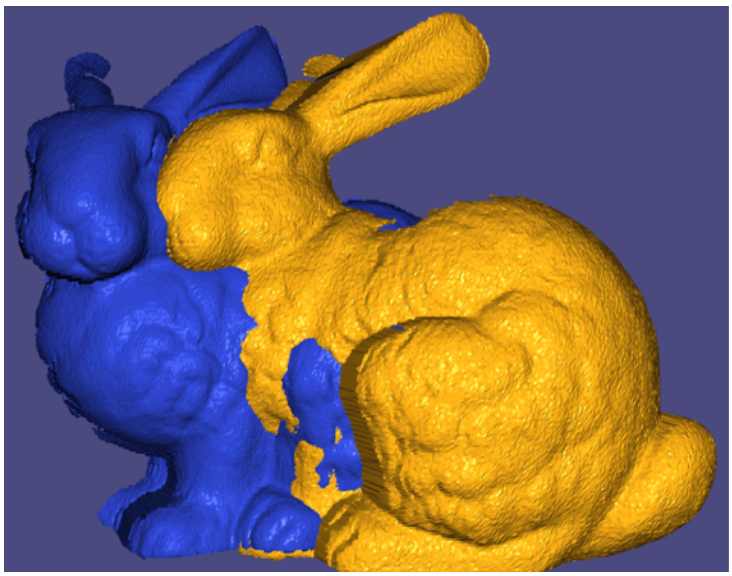
**Task 3 :**

To simulate a noisy model, I added a gaussian noise to each components of every vertex. I took the zero mean and a 0.08% of the bounding box size in the current axis as the standard deviation. For example for the x component of the vertices :
new_x = old_x + 0.0008*(boundingBox_X)
And the same for the y and z components.

Here is the new noisy cloud I obtained after adding noise once :



And after adding noise 5 times :



To compare the result with noisy and non noisy meshes, we are going to count how many iteration are needed for the algorithm to converge. I considered the algorithm converged when the number of points used to compute the new position doesn't vary a lot (stays in the same floor(pourcentage) 4 times in a row).

After adding the noise, I recomputed the normals for the noisy mesh. So, the more noisy is the mesh, the harder is it to compute good normal for the ICP algorithm.

| | **No noise** | **1 perturbations** | **2 perturbations** | **3 perturbations** | **4** | **5** |
|---|---|---|---|---|---|---|
| Number of iterations of the algorithm | 7 | 7 | 9 | 10 | 15 | 14 |

So, as we can see in the table, it takes an increasing number of time for the algorithm to converge even if it's possible with small perturbations.

**Task 4 :**

For this task, I used a number of points given in the main (nbP) to sample the moving mesh. Indeed, I always used the entire fixed points cloud to compute the kd-tree used by the ANN librairy. Doing that, I'm sure that a point in the moving mesh can find a correspondance (right ou false). However, I can sample the moving mesh to speed up my computation.
Here, I will use the number of intersecting faces after 7 iterations to report the accuracy of the sampled (knowing that the mesh has 40097 vertices).

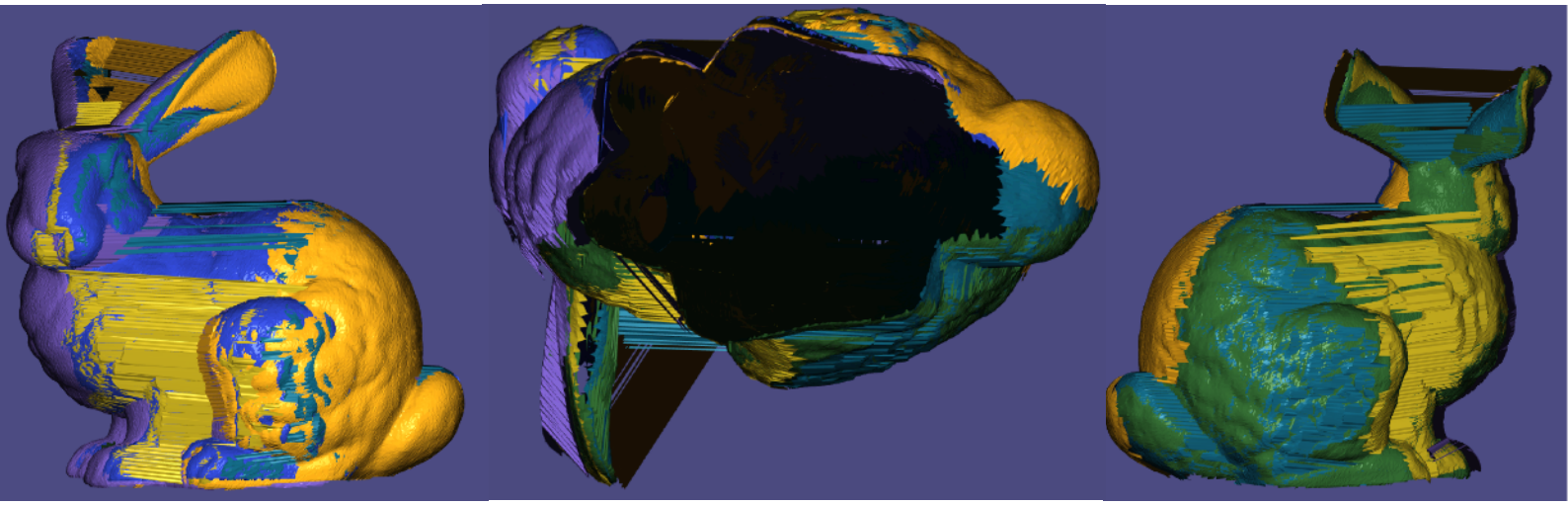
| | **500 (1.24%)** | **1000 (2.49%)** | **5000 (12.46%)** | **10000 (24.93%)** | **25000 (62.3%)** | **35000 (87.3%)** |
|---|---|---|---|---|---|---|
| Pourcentage of intersecting faces after 7 iterations | 7.54 % | 7.2 % | 8.04 % | 8.07 % | 7.95 % | 8.4 % |

Hence, we can see that taking 10% of the mesh or 60% of it doesn't change much the result and even taking 90% doesn't improve it a lot. So, better take around 15% (6000 points) to obtain good results in a small computational time.

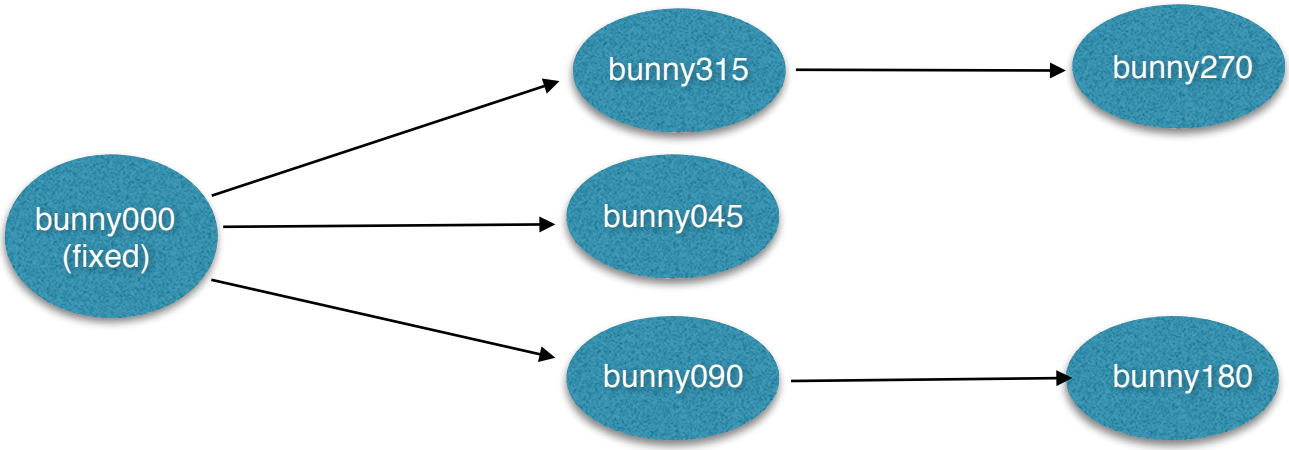
I will use this value from now on.

**Task 5 :**

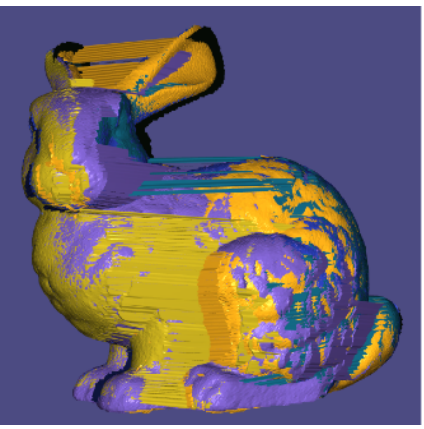
To do this task I first placed the 6 meshes near to each other. We can visualize that with the « add multiple meshes » button.
Here is my result :

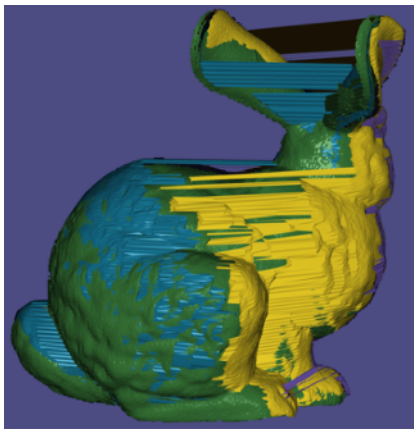


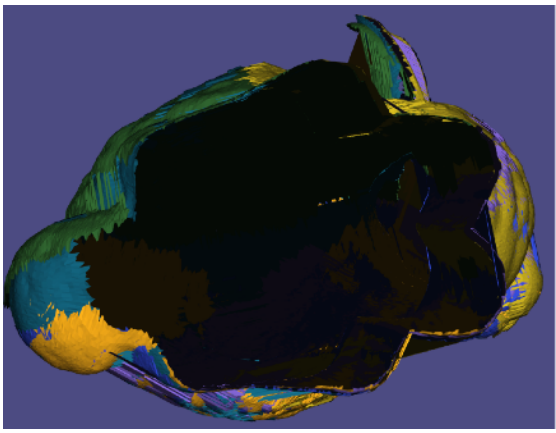
Then, I tried to align them using ICP. To do so, I used a fixed mesh, bunny000 and I placed 3 meshes with respect to this mesh. However, for the 2 others there weren't enough corresponding region to align them with bunny000. So, I computed 2 others kd-tree to place the last 2 with respect to one off the moving meshes.
So, at each call to « apply ICP to multiple meshes », here is how the meshes are moved with respect to one another :

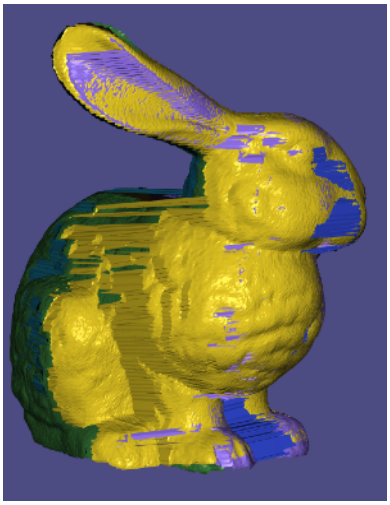



Here is my result after 7 iterations :









We can see that meshes have merged but some errors remain at the ears of the bunny for example.
This method takes a long time to compute as I need to build 2 more kd-trees from the ANN librairy and apply my ICP algorithm to 5 meshes. However, I didn't know how to do it without adding new reference meshes, which have been moved before.
I also tried to build a kd-tree from all the first 4 meshes (bunny000, bunny045, bunny090 and bunny315) and place the last 2 with respect to this one but it didn't really change the result or improve the computational time so I used the previous technique.

Moreover, with this method, the meshes are moving 1 after the other only with respect to 1 other mesh. I would have liked to find a solution so the meshes could move together with respect to all the other meshes.

**Task 6 :**

I already assumed I know the normals and I computed them after each transformations using the *cloud.setNormals(acq::recalcNormals())* function. This way, I was able to select good matching points in the first ICP algorithm. When computing my combined mesh for visualization purpose as the sum of 2 meshes, I add the normals together to be able to have normals on every vertices of my visualization.

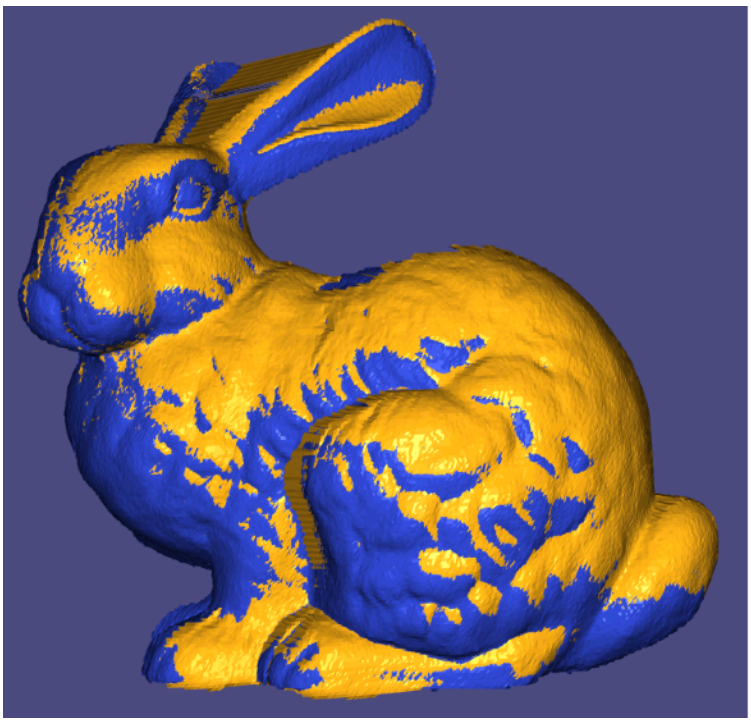
Hence, for each function, I applied it both meshes and sum the result for the visualization. However, for some reasons, the functions provided in the framework (to orient normals from FLANN or faces) didn't work anymore. The function to calculate neighbours from faces or with flann gives back a null pointer or made the program crashed. I think it might be a conflict of librairy between ANN, nanoflann and CGAL, however I haven't got time to look into it.

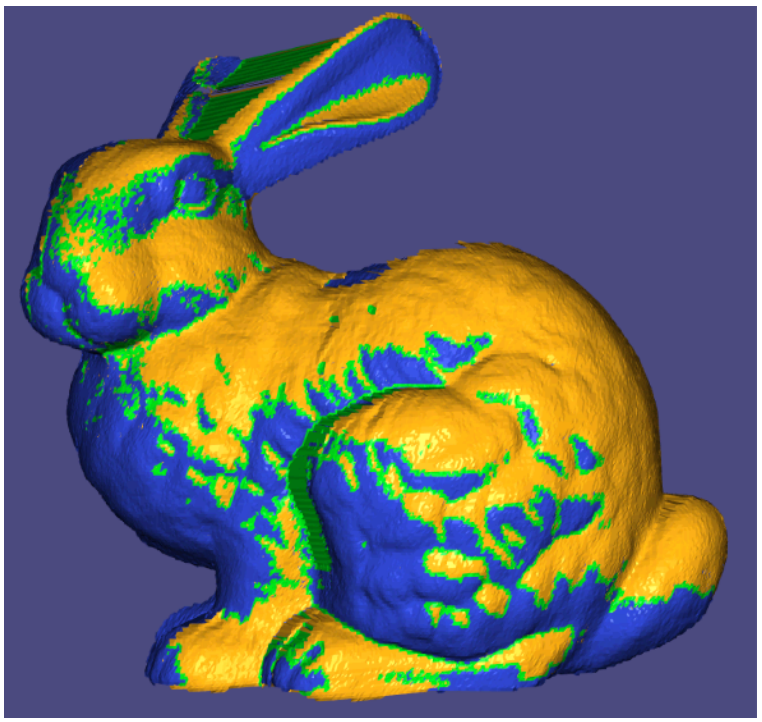
With the normals computed using flann (so, without any corrections of orientation), I could compute the ICP point-to-plane algorithm (in decoratedCoud.cpp). To do so, I solve a linear system Ax = b using x = A.colPivHouseholderQr().solve(b) function in Eigen.

The A matrix is sized N*6 with N the number of good corresponding points (with normals pointing in the same direction). I filled it with the cross product between the coordinates of the matching points and the normal of the moving mesh point. The b vector is N*1 and is filled with the dot product between the normal of the moving mesh point and the difference between both coordinate points. This way, mesh can slice using the corresponding plane and not only the points.

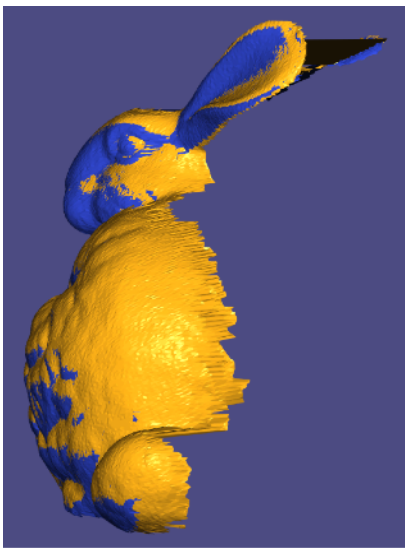
I kept a condition of good matches but less strict : the dot product now needs to be above 0.5. I also computed the normals using 15 k-neighbours and a maxNeighDist at 0.1.

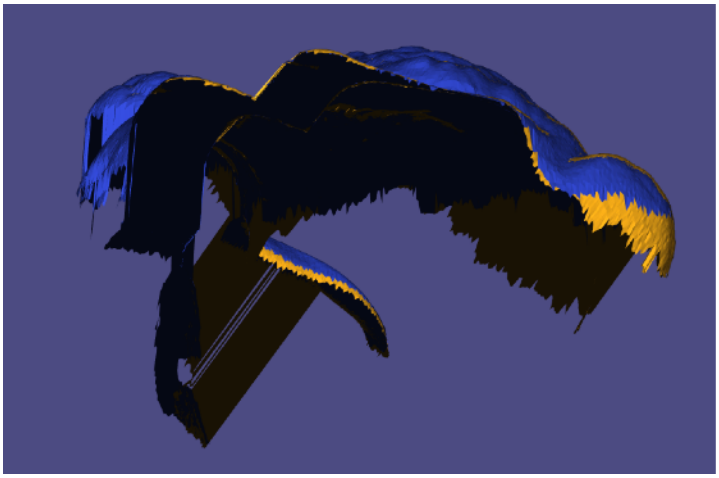
So, for the ICP points to plane, here is my result after 10 iterations :





It gives a pourcentage of intersections of 14% and converges after 10 iterations.





By comparing visually (and with the pourcentage of intersections) both algorithms I think we can say this one gives better results even if the normals aren't correctly computed.

Computing the weighted point-to-plane might be the most efficient one.