3DP HW3

Juri Farruku

ID: 2157856

1 Task 1

In order to complete the third assignment of the course, we had to complete some steps of a typical ICP refinement process.
As first task, in order to retrieve an initial and robust transformation between the two given Point Clouds and since no initial correspondences were given, a function which computes descriptors for the 3D structures and matches them was made. Since for the assignment there were no time constraints of any kind, i chose to manually implement and compute Point Feature Histogram, since it is very robust and popular.

In order to retrieve descriptors for keypoints of the given point clouds, two for loops, which are symmetrical, were implemented: one for the source cloud and one for the target. Given this symmetry, in this section only the source one will be discussed. As first step the cloud points are downsampled, thanks to the `VoxelDownSample` function, and in this case a voxel size of 0.007 was chosen. Once obtained the new cloud point, keypoints were extracted through `ComputeISSKeypoints` and then a `KDTreeFlann` was constructed upon them. The last step is done in order to find the neighbor of each keypoints in the next steps. Through the `EstimateNormals` function, each keypoint normal was computed, and for each angular feature used in PFH, which are **alpha**, **phi** and **theta**, a vector of size **bins** was initialized to zero (in this case bins is set to 7). Also, a vector containing vectors of doubles is made for the source keypoints: indeed each element of these vectors is associated to the descriptor, which is made by different doubles, of the aforementioned element.

After all these operations, the main for loop is implemented: for each keypoint obtained in the previous steps, we retrieve its associated 3D point and normal and through the function `SearchKNN` we find its neighboring point, whose indices and distances are stored in `source_neighbor_indices` and `source_neighbor_distances`. Then we loop through all neighbors actually retrieved from the previous function, and thanks to the Darboux frame, we compute alpha, phi and theta. These values are then discretized in the number of bins set before, in order to compute the actual histograms, which are finally normalized, dividing each of their elements by the count of the neighbors retrieved for the point being processed.

In order to compute the final descriptor for the keypoint of the iteration, the 3 histograms are concatenated and stored in the vector of descriptors at the index associated with the keypoint.

Once descriptors for both the source and target are computed, we begin the matching phase by setting a maximum distance as a threshold between the descriptors and instantiating an element of the type `CorrespondenceSet` to store correspondences between the two point clouds.
We iterate through all source keypoints and for each iteration we get the associated descriptor stored in `source_keypoint_descriptor[i]` and then iterate through all target's keypoint descriptors in order to retrieve a correspondence for the source point, if any exist. This is simply done by computing the Euclidean distance between the 2 aforementioned and checking whether the latest distance computed is the best retrieved so far. When all target descriptors have been tried, in the end the best distance found is compared with the actual threshold set before: if the best distance is less or equal than it, then an actual correspondence between source keypoint i and best possible target keypoint of index stored in **best** is set, otherwise no.

Finally, the initial transformation is obtained using the function **RegistrationRANSACBasedOnCorrespondence**, thanks to the correspondences obtained through the matching of descriptors.

2 Task 2

From now on, the described tasks are used to implement a complete ICP process. We start implementing the function `find_closest_point`, which is used to obtain correspondences between the source point cloud and the target one. This is done by building a **KDTreeFlann** using the target point cloud and then creating a for loop which iterates on all source points, and using the function **SearchKNN** we find the nearest neighbor of the processed point. The retrieved point and its distance are stored in `retrieved_neighbor_index` and `distances`. We then compare the distance obtained with the threshold given as input to the function: if the distance is less than or equal to the threshold given, we save the index of source point and the index of the target point respectively in `final_source_indices` and `final_target_indices`. In the end, we calculate the RMSE between the 2 point clouds' correspondences and return a tuple made by: `final_source_indices`, `final_target_indices` and RMSE previously computed.

3 Task 3

The next step is now obtaining the matrices R and t between the point clouds completing function `get_svd_icp_transformation`, which gets in input the source indices correspondences and the target's one. As first step, we iterate

through all the correspondences' indices and we store the points in `source_corr` and `target_corr` and add these values, respectively, to `s_centroid` and `t_centroid`. We finally obtain the centroids by dividing the aforementioned values by the number of correspondences. In order to retrieve the matrix **W** we iterate through all correspondences and multiply `target_corr[i]` and the transpose of `source_corr[i]`. The **Singular Value Decomposition** of W is obtained through the function `svd` and then matrices **U** and **V** are obtained respectively using `matrixU` and `matrixV`. Finally, rotation matrix **R** is calculated as U multiplied by the transpose of V. We compute the determinant of R and check if it is less than zero: in the positive case, in order to handle reflection cases, we compute R as U * **D** * transpose of V, where D is a diagonal matrix with values 1, 1 and -1. Finally, the translational vector **t** is obtained in closed form as the difference of the target centroid and the source centroid multiplied by the matrix R. These 2 values are then returned as transformation.

4 Task 4

As last task, we implement the function `execute_icp_registration`, which implements the real ICP loop. We first apply the initial transformation retrieved through the descriptors of task 1 and then do a for loop that iterates through the number of iterations given as input. In each iteration, we first use the function `find_closest_point` in order to obtain the indices of the correspondences and the RMSE between these two. We then obtain the transformation between these two sets of points through the function `get_svd_icp_transformation`: the transformation is then applied to the source point cloud and checks the convergence criteria for the RMSE retrieved by the aforementioned function.

5 Results

As said before, in order to retrieve solid descriptors, since there were no time constraints and the assignment had no applications for real-time programs, i decided to use PFH, thus obtaining solid results with a little more of time needed.
The results of different datasets can be viewed in Figure 1 for the dragon dataset, Figure 2 for the vase dataset and Figure 3 for the bunny one. It can be seen that the program obtains good results, since the clouds in the aforementioned images are completely aligned.

The visual results are corroborated by the numerical data. In table 1 we can see that the values of RMSE after description-based alignment are low, since the method adopted to extract descriptors is very solid and robust. This shows that the method is implemented correctly and we can also notice that the ICP refinement works also well, since the final RMSEs that we obtain are lower than the initial ones (in the Dragon and Vase datasets the final RMSEs are significantly lower than their initial values). Inspecting the times, we can

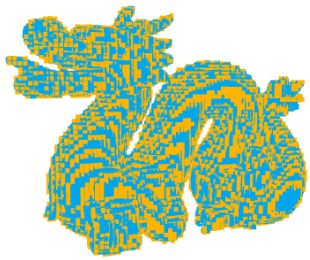

Figure 1: Snapshot of aligned clouds at the end of the ICP registration process on the Dragon dataset.

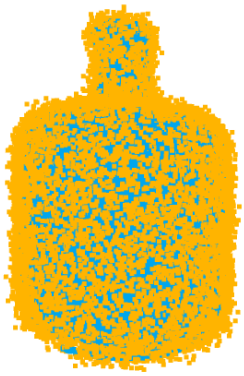

Figure 2: Snapshot of aligned clouds at the end of the ICP registration process on the Vase dataset.

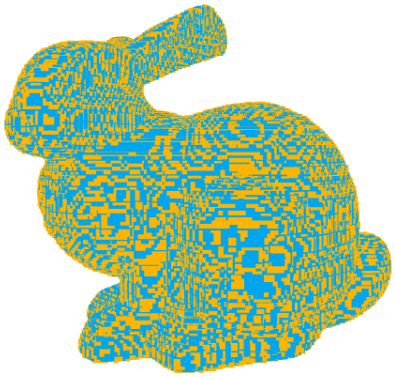

Figure 3: Snapshot of aligned clouds at the end of the ICP registration process on the Bunny dataset.

Table 1: Table containing values of RMSE at the beginning and in the end, and times for each dataset

Data Set	**Dragon**	**Bunny**	**Vase**
RMSE after descriptors alignment	0.100133	0.00413456	0.107546
Final RMSE	0.00564117	0.00341362	0.0164676
Descriptor-based registration time	2386.43 ms	12010.9 ms	25327.3 ms
ICP Refinement Time	2209.42 ms	545.885 ms	7099.53 ms

notice that both the ICP and descriptors extraction times are dependent of the number of points and the density of the point clouds: indeed, the vase and bunny datasets are composed of more 3D points than the dragon one, requiring more time in order to be processed. The computational complexity could be reduced by downsampling the clouds even more than what is done in the assignment, obviously checking that values of the RMSE do not get extremely worse.

Since the shape of the Vase Point Cloud is symmetrical, it caused problem when trying to align the two different point clouds. This problem was mitigated by not using the **phi** feature, not compromising the other datasets. It can be also noticed that even without the phi feature and using only two angular features, the final results on the datasets are all stable and correct.