Computer Vision 2 - Assignment 1

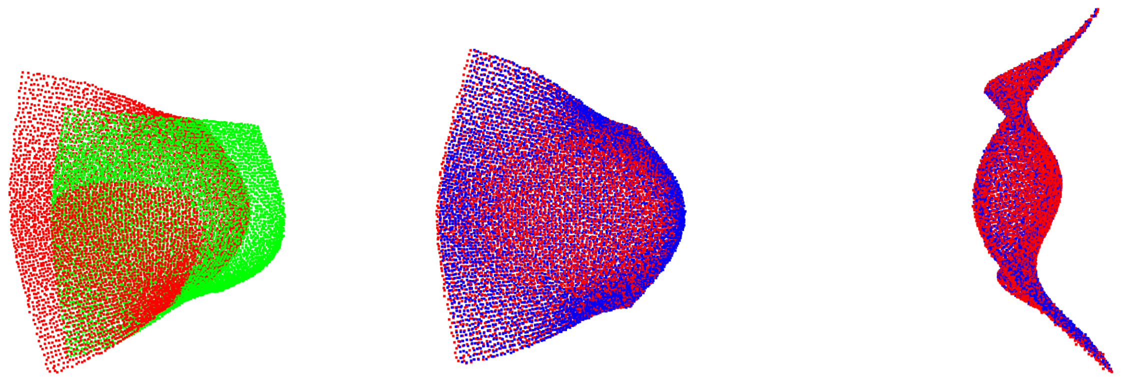
April 2022

Introduction

This report will look at Iterative Closes Point (ICP), improving speed and quality of ICP and Global registration of ICP. In the Iterative Closest Point, the ICP algorithm will be implemented. When this is done, the algorithm will be improved by using different sampling methods, by using kd-tree and using z-buffer. Afterwards when the algorithm is improved, it will be used on real data with noise instead of synthetic data. After these additional questions will be answered and a self-evaluation will be done.

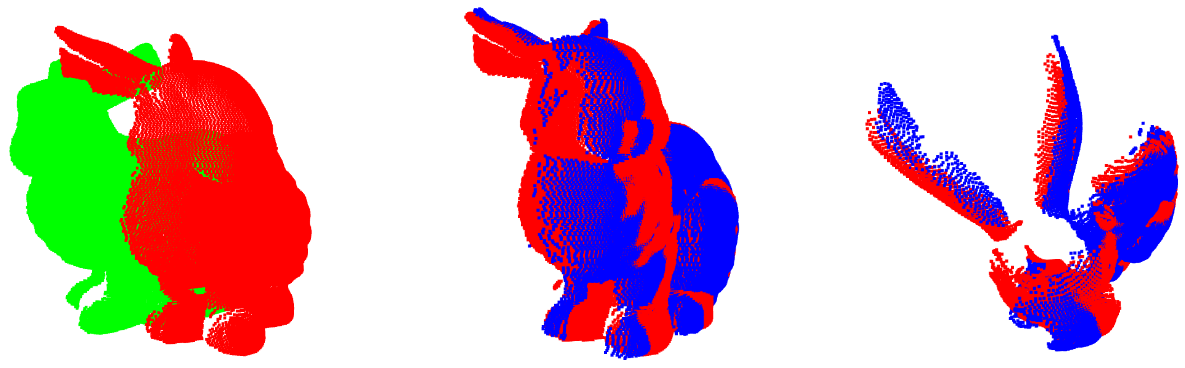
1 Implementation of the ICP algorithm

The ICP algorithm was implemented in Python. The default Euclidean matching process, as well as the alternative matching strategies were vectorized as much as possible. The same holds for computing the root mean squared error (RMS) between matched transformed source points and target points, which served as the quantity ought to be minimized.



(a) (b) (c)

Figure 1: Results of the default ICP algorithm executed with wave point clouds. Green: source cloud. Blue: transformed source cloud. Red: target cloud.



(a) (b) (c)

Figure 2: Results of the default ICP algorithm executed with bunny point clouds. Green: source cloud. Blue: transformed source cloud. Red: target cloud.

Figures 1 and 2 show the results of executing the ICP algorithm with wave and bunny point clouds. To obtain the results, default Euclidean matching was used and the tolerance value for convergence was set to 1e-4. Due to memory issues, the source and target clouds of the bunny were downsampled by a factor 2 by using every second point in both point clouds. This resulted in a source cloud containing 20,128 points and a target cloud containing 17,668 points. The algorithm converged after 27 iterations with an RMS of 0.029 which took 32 seconds using the wave data. Using the bunny data the algorithm convereged after 34 iterations with an RMS of 0.003 which took 6.5 minutes.

The visualizations in Figures 1b and 1c show that the approximated transformation provides an almost perfect mapping. Figures 4c and especially 4d show that the approximated transformation still results a minor misalignment. This is likely due to the fact that there are regions of the shape that are only covered by either the source or the target cloud. Although not explicitly visible in the Figures, these regions include sides of the head and parts or the ears.

2 Improving Speed & Quality

To improve the runtime of the algorithm as well as the quality of the approximated transformation, several subsampling and matching techniques were implemented and investigated.

2.1 Sampling

Implementations were done to be able to perform uniform, random and multi-resolution subsampling of the point clouds. Uniform sampling is done once after which the same points are used for each iteration whereas random and multi-resolution sampling result in different points being used in each iteration. As the number of points in the wave data did not cause memory issues, the sampling strategies were evaluated using the bunny data.

Figure 3 show the results of the ICP algorithm when applying different sampling techniques to the bunny data. Euclidean matching was used and the tolerance value was set to 1e-5 for uniform and random matching and to 1e-4 for multi-resolution matching. A total of 6000 samples were taken from both the source and target clouds when applying uniform and random sampling. Table 1 shows the iterations, final RMS and runtime in seconds for different sampling methods.

	Iterations	RMS	Time (seconds)
Uniform sampling	44	0.0033	45
Random sampling	40	0.0034	43
Multi-resolution sampling	34	0.0033	279

Table 1: Runtime statistics when using Euclidean matching and sampling

A notable consequence of random sampling is the fact that the approximation process becomes unstable when using a tolerance of 1e-4. Due to the randomness, the RMS decreases less smoothly at each iteration causing the algorithm to stop with a highly sub-optimal approximated transformation. A possible method to prevent this problem could be to impose the constraint that the RMS should be stable during several iterations, rather than the current 2 iterations. We propose a change in convergence criteria rather than maintaining a low tolerance value, as the latter could cause the algorithm to overfit.

Figure 3 shows the result of the ICP algorithm with different sampling methods applied. Overall, the results after sampling are quite similar to those when using half the points as defined in Section 1, though the alignment appears to be slightly better after subsampling the point clouds. This could Furthermore, there appears to be no noticeable difference between the results obtained with uniform and random subsampling. Sampling positively influences the runtime of the algorithm substantially and produces results similar to using at least half the points in the clouds.

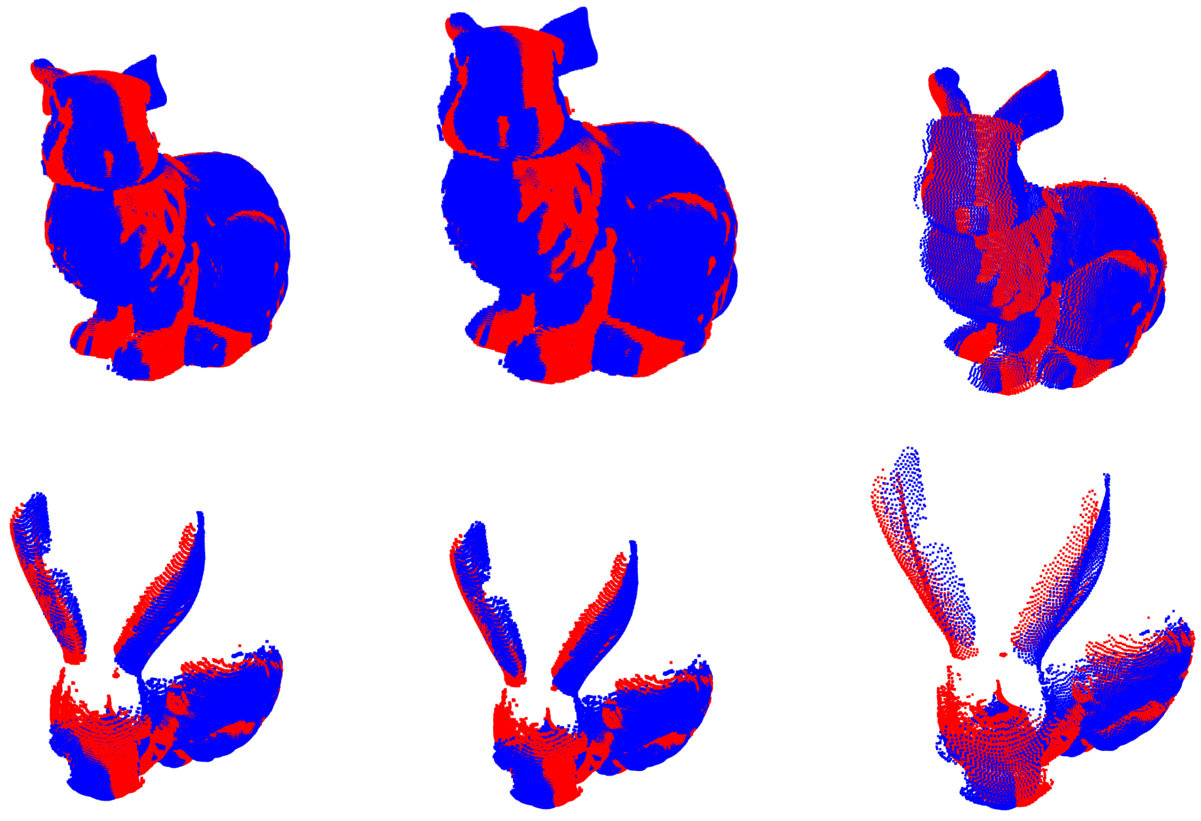

(a) Uniform sampling (b) Random sampling (c) Multi-resolution sampling

Figure 3: Results of the default ICP algorithm executed with bunny point clouds and different smapling techniques. Blue: transformed source cloud. Red: target cloud.

2.2 Matching

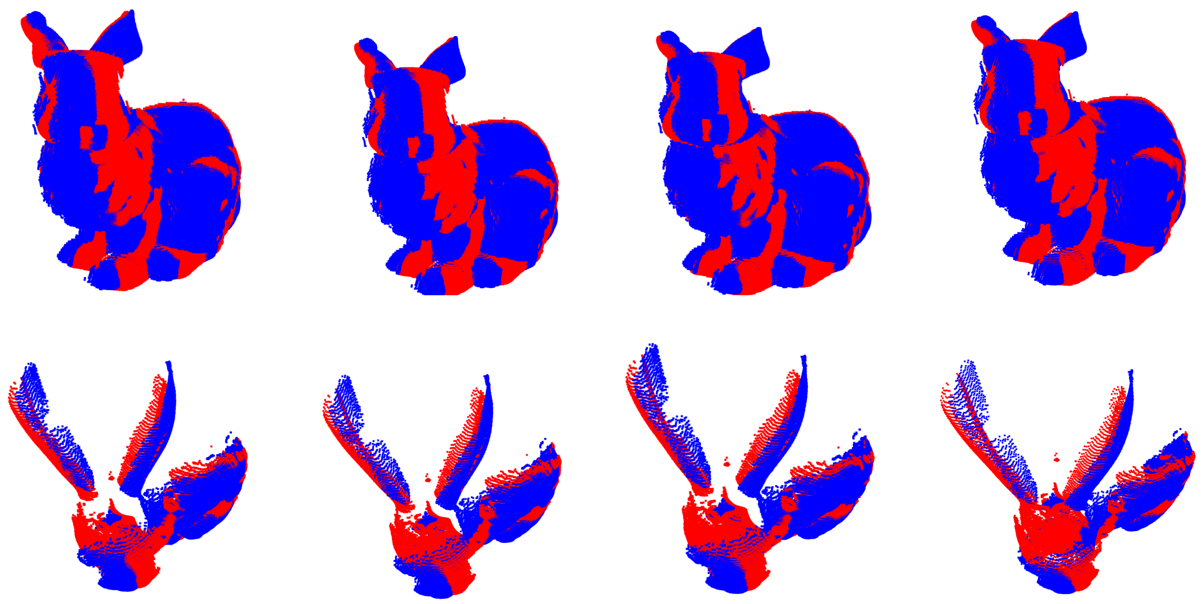
2.2.1 KD-Tree

KD-Tree matching was implemented using scikit-learn. Upon matching a KD-Tree is made using the points in the target cloud. After this the closest point, in terms of Euclidean distance, is queried for each of the points in the source cloud to create point correspondences.

Using KD-Tree matching drastically improves the runtime of the algorithm. Figure 4 shows results of running the ICP algorithm with KD-Tree matching. A tolerance of 1e-4 was used for each sampling method except for random sampling, for which the tolerance was set to 1e-5. Table 2 shows the number of iterations, final RMS and runtimes. The results of the bunny data are all almost the same. They all work good. The biggest difference is the time they take as can be seen in Table 2. This is due to the uniform and random sampling being a lot smaller than the full samples or the multi-resolution sampling.

	Iterations	RMS	Time (seconds)
All samples	36	0.0032	9
Uniform sampling	33	0.0037	0.87
Random sampling	38	0.0038	0.84
Multi-resolution sampling	35	0.0033	5

Table 2: Runtime statistics when using KD-Tree matching and sampling



(a) All samples (b) Uniform sampling (c) Random sampling (d) Multi-resolution sampling

Figure 4: Results of the ICP algorithm with KD-Tree executed with bunny point clouds and different smapling techniques. Blue: transformed source cloud. Red: target cloud.

2.2.2 Z-Buffer

The Z-Buffer matching method, as defined in [1], was manually implemented. The referential was defined to be the same as the coordinate system as the target cloud, using its z-axis as the direction of projection. Therefore the minimal enclosing box was defined with x and y coordinates. Both the source and target referentials were divided into 20x20 cells. These values were derived experimentally and result in an appropriate approximation of the transformation. Cells without a candidate point were omitted upon matching the points in the referentials. When finding matches in the target referential for each source point, points in cells within a region of 7x7 centered around a cell were used.

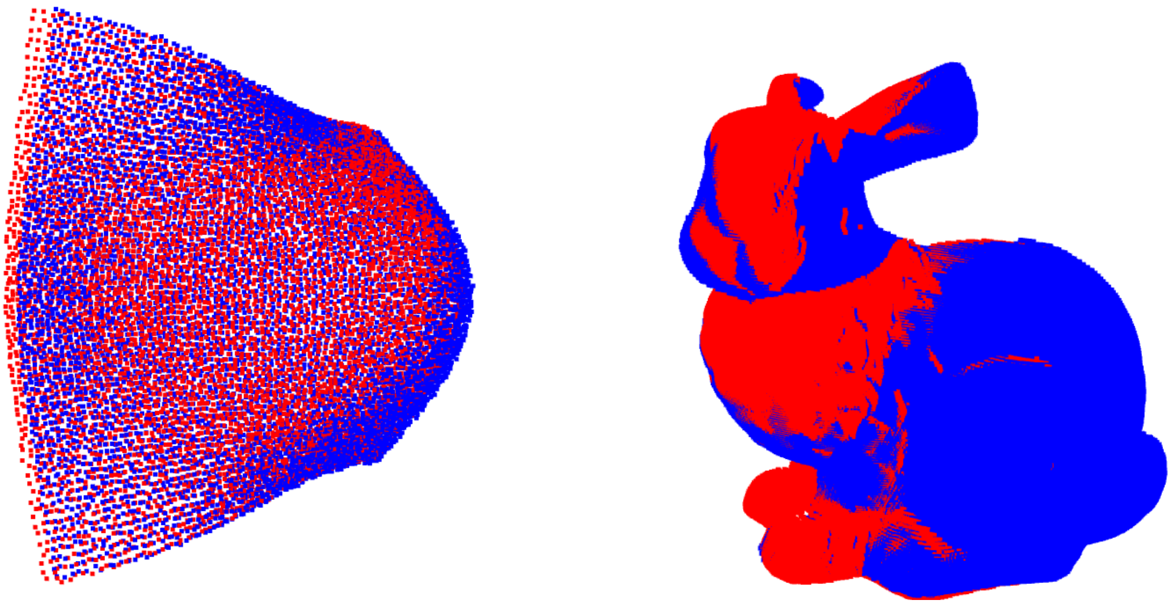


Figure 5: Results of the ICP algorithm using Z-Buffer matching. Blue: transformed source cloud. Red: target cloud.

Figure 5 shows the results obtained with Z-Buffer matching using the wave and bunny datasets. A tolerance of 1e-5 was used for both datasets. Approximating the transformation of the wave data terminated after 31 iterations with an RMS of 0.082 which took 3 seconds. The algorithm executed with the bunny data also converged after 31 iterations but this time with an RMS of 0.006. This took about 10 seconds.

2.2.3 Z-Buffer: Additional questions

There are a few downsides of using a mono-z-buffer, for example the storage requirements are higher. And since depth storing is not done, a location in the z-buffer may have to be changed many times depending on the number of surfaces representing the scene. Moreover, it's a time-consuming process, since it needs to scan and convert every single polygon. Lastly, the space involved is very large, at least it requires X*Y size of the buffers.
Since depth storing is not done, the location needs to be changed many times. To solve this problem a depth buffer can be used. A depth buffer is a piece of memory that stores, for each pixel of the image to be rendered, how far it is from the camera. If another polygon is drawn on that same pixel, the renderer will check whether the distance of the new pixel is closer than the current pixel. If so, the new pixel is drawn and the value of the depth buffer will be adjusted. If the pixel is further away, the old pixel is kept. In this way, the polygon closest to the camera are always visible, even if they are gridded in a completely different order.
The z-buffer did not work fully on the wave dummy data, this can be seen in Figure **??**. The source and the target do not match exactly, but they are close. The z-buffer did work very well on the bunny dummy data, this can be seen in Figure **??**. The source and the target match very well.
The choice of the z-buffer referential is important to the quality of the registration. Choosing a wrong direction for the projection results in very little spreading of the points. This results in less overlap of the source and the target. The change we would implement to the z-buffer is to define a system to calculate the transformation of the target to the referential.

3 Global Registration

Finally, we will attempt to use the ICP algorithm on real-life data. A dataset of 100 images of a student from all angles is provided along with point-cloud data. We will match the images in order using different step sizes between frames. We find that making a full 3d image from 360 degrees proves difficult.

3.1 Estimate pose at N frames and merge in the end

The first implementation simply calculates the transformation between adjacent frames for step sizes of 1,2,4 and 10. These transformations are stored in a list and when all required matrices are collected they are applied to map all point-clouds to a single shared coordinate system which hopefully matches them up to form a person. In figure 6 we can see the result of the different step sizes on the resulting 3d image for the first half of the frames. A tolerance of at least 1e-6 is necessary to produce a clean looking image. When comparing the stepsize it can be seen that the algorithm clearly struggles with matching pointclouds with little overlap. Stepsize 1 shows only slight errors while having everything be in the right orientation and position. Even from stepsize 2 we can see completely rotated torsos intersecting the main image and the nose and ear being matched causing a two faced person. This is also the case for the higher stepsizes where several arms can be seen intersecting. In the end, using a stepsize of 1 produces sufficient results.

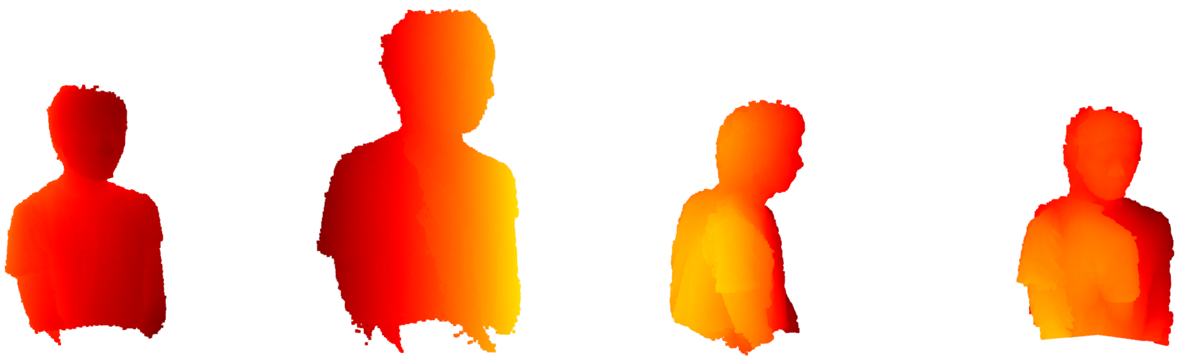

(a) Stepsize 1 (b) Stepsize 2 (c) Stepsize 4 (d) Stepsize 10

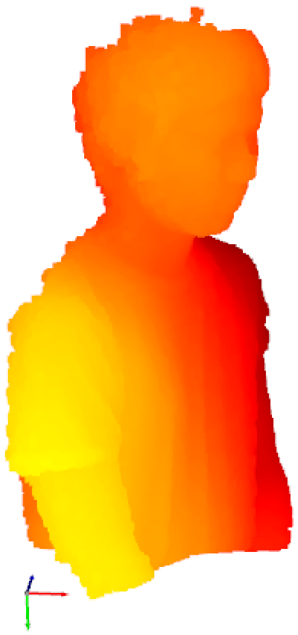
Figure 6: Results matching over different step sizes

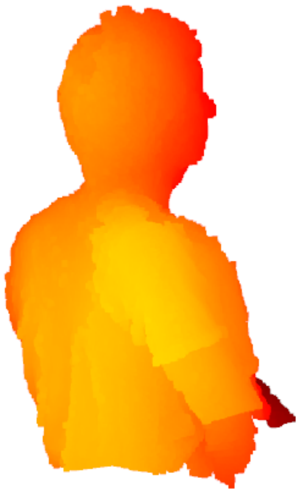
3.2 Estimate pose and merge at N frames

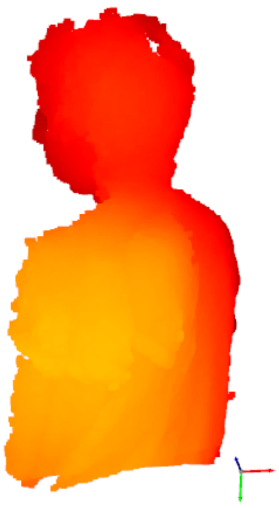
The second implementation changes the order of the algorithm. Instead of computing the transformation between adjacent separately and then merging with the resulting transformations we will now merge the point clouds on the go and map that merged cloud to the next frame. This technique will only be applied to step sizes of 4 and 10.

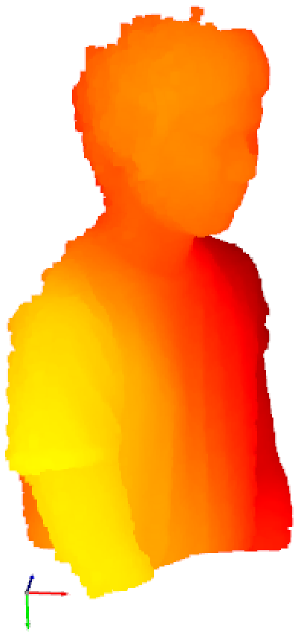
The first results show that the algorithm does not produce a clean 3d image. One of the issues is that the algorithm attempts to match the back and front of the person to the same plane, instead of creating a 3d shape with a front and backside. Because the front and back are both relatively flat, there is likely a high amount of matches for this part which may offset the errors around the border of the image. This is amplified in the *merge as you go* algorithm because there is a much greater area to match on. Some of the decrease in performance may also be due to the step size limitation.

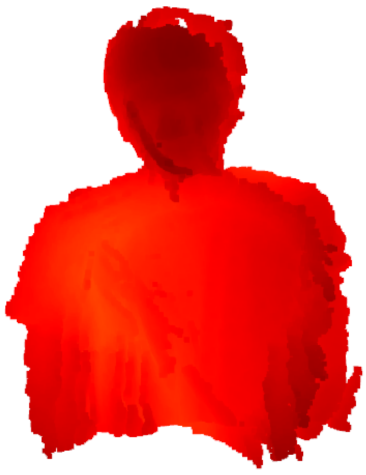
Figure 7 shows that for some sequences of images the algorithm is successful in creating a good match. It shows both incrementally increasing the amount of frames, as well as taking the subset that is added separately. It shows clearly that in the subsets of frames itself there aren't the same big errors as in incrementally increasing set. The algorithm seems to be limited in the amount of frames it can combine correctly.


(a) matching frames 1-20


(b) matching frames 20-50


(c) matching frames 50-75


(d) matching frames 1-20


(e) matching frames 1-50

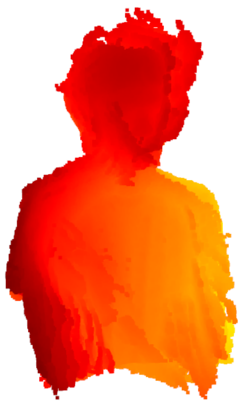
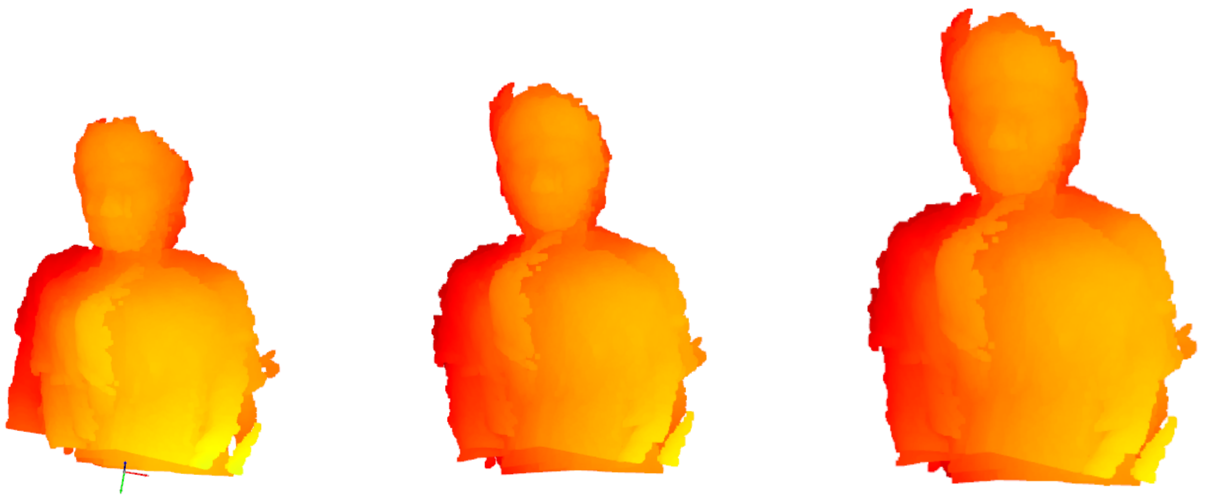

(f) matching frames 1-70

Figure 7: Top: Subsets, Bottom: Incremental

One of the things that caused this issue was when target point-clouds had a relatively small sample size. The algorithm could not match up the source image to small target clouds. This is partly because the source image had to be downsampled and partly because there weren't as many target points to map to. The resulting error then ruined subsequent matches as well. The RMS for these sets turned out much higher than the larger sets.

Several attempts to improve the result were made by setting the tolerance to be lower. The results below show that this only provides small improvements for high computational cost. This can also be seen in the results of the 10 step version.



(a) tolerance: 1e-4 (b) tolerance: 1e-6 (c) tolerance: 1e-7

Figure 8: Results matching with step size 10 over different tolerance values

Figure 8 shows the effect of decreasing the tolerance. Though the larger errors are not resolve there is a slight improvement in the details of the image. The higher tolerance image shows several noses in the face, which points to smaller errors in the rotation of the image. This is resolved in the lower tolerance images.

When running the algorithm with a tolerance of 1e-7 the last steps took up to 150 iterations to converge. Again, this is likely because the merged point-cloud already contained errors from previous steps. These errors are likely to be bigger because of the big step size. There is less overlap between the two frames which leads to less accurate matching. Another factor is that in creating a 3d image the figure has to match up at the end, when overlap with the starting image is reached after going around. Any errors made in the previous iterations may severely hinder the algorithm from finding a good match. This is less the case when matching only adjacent images.

In the end the algorithm does produce better results for the high step sizes. Overall the resulting image is smoother and although the same intersecting bodies can be seen in both, this is more intense in the 3.1 implementation.

4 Additional Questions

1. ICP has a small convergence basin and therewith a slow convergence speed, because of its linear convergence rate[2].

   Moreover, it has a high sensitivity to outliers. This comes from a problem in the alignment step, whic hestimates a rigid model based on the least-squares optimization. This is not a robust cost, so if many source points have no true correspondences in the target set, the ICP will converge to a bad local optimum. Due to this bad local optimum ICP is sensitive to outliers and partial overlaps, which are a common feature of real datasets[2].

2. The ICP algorithm can be improved by making a new ICP-based approach with better registration accuracy, wider convergence basin, higher convergence speed and better robustness to outliers and partial overlaps. This will help the ICP algorithm improve in terms of efficiency and accuracy. This can be done by using a symmetric point-to-plane distance metric, which minimizes the point-to-surface error based on normals of bothe the target and source points, which has a wider convergence basin and higher convergence speed. It has a wider convergence basin and higher convergence speed becasue its functional zero-set is locally-second-order surfaces while the one of point-to-plane metric is only local plane patches. Moreover, using robust loss instead of the least-squares cost of the ICP algorithm can improve the robustness of the outliers and partial overlaps. The least-squares cost is a good choice for observations with Gaussain noise, but its not a robust loss and very sensitive to outliers. Therefore a robust loss is adviced. Together you get a robust symmetric ICP method[2].

5 Self-Evaluation

The introduction and conclusion are written by Bunyamin. The first exercise is done by Gijs. The second exercise is done by Gijs and partially by Bunyamin. The third exercise is done by Levi. The fourth exercise is done by Bunyamin. Every team member worked on the report.

Conclusion

In the end we learned how Iterative Closest Point (ICP) works. We implemented the ICP algorithm. After this we improved the speed and quality of the algorithm by improving the point cloud sampling, using kd-tree and using the z-buffer. It has been shown that subsampling point clouds and using the kd-tree and z-buffer matching techniques drastically improve the execution time of the algorithm. Finally, we estimated the camera poses of frames from real data.

References

[1] Raouf Benjemaa and Francis Schmitt. Fast global registration of 3d sampled surfaces using a multi-z-buffer technique. *Image and Vision Computing*, 17(2):113–123, 1999.

[2] Jiayuan Li, Qingwu Hu, Yongjun Zhang, and Mingyao Ai. Robust symmetric iterative closest point. *ISPRS Journal of Photogrammetry and Remote Sensing*, 185:219–231, 2022.