Advanced Vision Practical Report 2020

Yichao Liang S1735938, Wim Zwart S1999534

Abstract

This is our report for the coursework for the Advanced Vision course at the University of Edinburgh. In it we describe how from a sequence of 3D point clouds, each showing the same person from a different angle, a single 3-D model of the person's head is generated. We describe the various steps taken for the entire process in Section 2, and analyze, discuss the quality of the 3-D model in Section 3, 4.

1. Task Description

In this task we have been provided with 4 sequences of 15 frames. Each sequence shows a different person sitting on a chair, captured by a stationary Intel RealSense depth sensor. The chair is turned by a small angle from frame to frame, such that the person turns by a full 360° over a complete sequence. Each frame is a point cloud, composing a 640 x 480 dimension RGB image in the x-y plane, and each with a z-coordinate representing the distance to the camera. The task is to generate a single 3-D model, fusing the 15 frames into a single point cloud, and to repeat this for each of the 4 sequences.

2. Algorithms

The algorithms applied for the processing can be summarized into four major steps:

1. Extracting the relevant data point from each point cloud.
2. Estimating the reference frame transformation between consecutive frames.
3. Fusing all of the points into a single 3D coordinate system.
4. Building as complete a 3D model as possible.

which will be discussed in detail in the following four subsection respectively.

2.1. Extract the relevant data from each point cloud

This stage attempts to remove the points which are irrelevant to the "head-reconstruction" task. The irrelevant points correspond to the entire background in the image (i.e. the wall, the whiteboard, etc.), as shown in the first column of Figure 1.

This stage is achieved by applying three filters sequentially.

To keep track of the mapping of the pixel indices between the filtered image and the unfiltered image, a list of integers `xy_mesh` ranging from 0 to 307200[1] is updated as each filter is applied.

Two other filters have also been tried out, but were eventually not used. All five filters are discussed in the sections below.

2.1.1. NaN filter

For some pixels the z-coordinate is undefined. These pixels should be removed for the future tasks. The NaN filter removes the 3D pixels whose z-coordinate is `NaN`[2]. Example images showing the remaining pixels are shown in the second column of Figure 1.

2.1.2. Depth filter

This filter removes the 3D pixels whose `z` coordinate value is greater than a certain threshold[3]. These pixels correspond to the 3-D points of the background that are further away to the camera. After several trials, we found a threshold of 1.5 works best for the given data. The third column in Figure 1 shows the remaining pixels after applying the depth filter. Unfortunately, some pixels of the nose, e.g. in image c of S_1 in Figure 1, are also removed due to the defective depth data.

2.1.3. Background color filter

After the applying these filters, we still have some pixels left that belong to the background, such as the pixels on the left of the right ear of the person in the third column in Figure 1. These points can be easily identified by their color, which is close to the average color of the background. Thus we designed a filter that removes the pixels at the edge of the foreground object with a color similar to the

[1] The number of pixels for each frame of the image

[2] See method filter_nan() in single_head.py

[3] See method in filter_depth() in single_head.py

Figure 1. Pre-processing results on sample frames in each sequence of the data set. Column a) shows the original colour image, column b) shows the same image after the NaN filter has been applied. Note that the pixels that are filtered out have been coloured blue, to maximize contrast with the remaining pixels. Column c) shows the result after the Depth filter has been applied. Finally column d) shows the result after applying the background removal filter

average background color[4]. The average background color `bg_color` is computed by taking the mean RGB value of all pixels that have been filtered out by the preceding filters. Now, a pixel is removed from the remaining pixel list if the $l2$ distance between its color vector and `bg_color` is greater than a preset threshold d_color and if the pixel is on the edge of the remaining foreground object. The edge pixels are determined by whether there are enough pixels within a distance, e.g. a 3x3 pixel block, defined by that pixel, in the filtered image we currently have.

We apply this filter iteratively, until no more pixels are removed from the pixel list of the foreground object. The fourth column in Figure 1 shows the remaining pixels after applying the background color filter.

[4]See method remove_background_color() in single_head.py

We can see from the figures that there are still some unwanted silver colored pixels on the left of the chair, which corresponds to the metal frame of the whiteboard as in the first column in Figure 1. They were not removed because this color differs too much from the background. Two other attempts (discussed in Section 2.1.4 and 2.1.5) were made to replace this background color filter, though failed. As it turns out, the problem of the silver pixels is successfully eliminated in the processing of Section 2.4, when we merge the 15 frames into a complete head and reduce the total number of pixels.

2.1.4. Edge-based filter(Unemployed)

As discussed in Section 2.1.3, we wanted to remove the pixels belonging to the whiteboard frame from our foreground object. The challenge is that their color is quite different from the background. If we would increase the $l2$ distance

threshold w.r.t. the RGB color as used in the background color filter, that could lead to removing parts of the person where color is similar to the background.

Thus we experimented with an Edge-based filter[5] in the light of these problems. We want the filter to find the boundary of the object we are interested, then flood fill it to obtain a binary image, which we can use to obtain the corresponding pixels. However, the edge output of the Canny edge detector isn't always connect, and carelessly dilation would result in the edges merged with irrelevant background objects. Thus we use `findContours(), drawContours()` from OpenCV to try to connect the disconnected edges, and apply these three steps iteratively to obtain as complete an edge as possible while avoiding cases similar to dilation happen (because drawing contours would also make the edge thicker). To avoid unwanted objects in the image, we also had to manually located the range each person are moving around (e.g. for the first person, it's between column 260-480 and row 120-370 of the image) and cropped the outside region out. In the end, we apply the `binary_fill_holes()` and a couple iterations of erosion to obtain the interested region. The unfiltered pixels are shown in Figure 2

As a result, we successfully removed the silver pixels from the whiteboard, but still lost the bright pixels on the nose of the person. But since a) the problem of the silver pixels can be eliminated in the final merging step and b) the assignment does not ask us to crop out the head itself, this filter is not used for the final system. We also didn't remove the edge pixels of the person as that can abandon a lot of useful pixels.

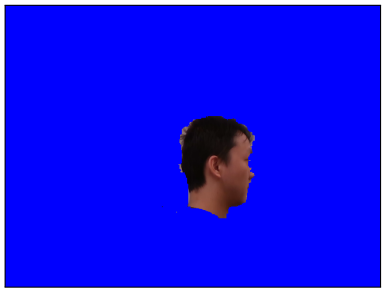


Figure 2. After the Edge-based filter is applied. All the removed pixels so far are colored blue in the image. The frame is the same one as the frame S_1:6/15 in Figure 1

2.1.5. Parzen window filter (Unemployed)

This filter also attempts to remove the flying pixels by replacing the Background Color filter mentioned in Section 2.1.3. These 3D points typically have less neighbouring points than the normal points within a radius by their nature, which is exploited here. The filter[6] works by, given a 3D point `p` and a preset radius `r`, measuring the number of its neighbouring points `num_neighbor` within the sphere defined by `p` as the center and `r` as the radius. And we remove the point `p` if `num_neighbor` is less than a preset threshold. However, this algorithm has a computation complexity $O(DN^2)$ where N is the number of points, D is the dimension of each point vector. We improved it by testing only the points on the edge of the foreground object. The edge points are determined, based on the binary 2D image from the Section 2.1.2 filter, using essentially the same method as described in Section 2.1.3.

After experimenting various hyper parameters, we found that the filter works best with `num_neighbor=7, r =0.005` and applying this iteratively 10 times. The resulting image is shown in Figure 3. The output is not ideal as a) it still leaves quite a few flying pixels in the foreground and b) it starts to remove some foreground pixels that we would like to keep, e.g. the pixels on the chest. Therefore this filter is also not used in the final system.

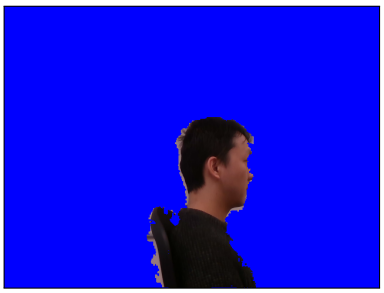


Figure 3. After the Parzen window filter is applied. All the removed pixels so far are colored blue in the image.

2.2. Estimating the Reference Frame Transformation Between Consecutive Frames

The reference frame transformation estimation is performed in four steps. First we calculate SIFT keypoint descriptors for each frame. Then we use these keypoint descriptors to find an initial, coarse estimation of the reference frame transformations between consecutive frames. Subsequently we applied two additional algorithms to further refine this estimation: the Iterative Closest Point algorithm (ICP) and a self-defined algorithm, 6-D Refine.

2.2.1. SIFT keypoint extraction and matching

SIFT keypoints are extracted from 2D images and matched between consecutive frames for the next step of the pipeline. The SIFT keypoints and the matches are computed using the OpenCV library (Bradski, 2000) functions[7]. We experi-

[5] See `edge_based_filter()` in single_head.py

[6] See in `parzen_filter()` in single_head.py

[7] See `get_descriptors()` in SIFT.py

mented with over 100 sets of parameters and a variety of image color spaces, and finally settled on the following parameters: `contrastThreshold=0.02, edgeThreshold =10, sigma=0.5`. The `contrastThreshold` was decreased and the `edgeThreshold` was increased from the default values to make the function generate more keypoints, while the `sigma` was reduced because of the relative high image resolution. These adjustments enabled the function to capture more subtle details. The descriptors located at the edge of the person is removed as a)they are usually weak, b)the existence of some flying pixels, using the same procedure as the "edge pixel detection" except with a larger window.

The keypoint matching is done by a brute force matcher[8]. It works by comparing the $l2$ distance between each 128-dimensional-keypoint descriptor from one image to all the descriptors from another image and returning the one with the closest distance.

On a small tangent, the ratio test on the distances d_1, d_2 to the two nearest descriptors are experimented, with only the matches with $d_1 < 0.9 * d_2$ are kept to improve the matches' quality. The lower the ratio, the more matches are filtered out. But this was left out in the end as the performance improvement was insignificant.

An even simpler filter has been applied and proved to be helpful. We remove the match pairs with a difference d_y in the y-axis higher than 15[9], based on the fact that we know the people are only rotating so that the real corresponding points between frames should not have much vertical variance.

The remaining matches between two frames are then passed on for the next stage of processing.

The results for each of the data sequences from this stage are shown in detail in Figure 10, 11, 12 and 13.

2.2.2. Estimating 3-D transformations with the RANSAC method

With keypoints matches from consecutive frames, we use the SVD of matrix based algorithm (Lorusso et al., 1995) in combination of the Random sample consensus (RANSAC) technique (due to some spurious matches between keypoints) to estimate the best transformation.

The transformation estimation algorithm can be summarised as follows: assuming that two set of N 3-D points m_i and d_i, $i = 1...N$ (corresponding to the matching points from two consecutive frames) are related by $d_i = Rm_i + T + V_i$, where R is a 3x3 rotation matrix, T a 3x1 translation vector and V_i a 3x1 noise vector. Solving for the optimal transformation $\hat{R}, \hat{T}$ such that the least squares error function $\sum_{i=1}^{N} \|d_i - \hat{R}m_i - \hat{T}\|^2$ is minimised. The closed form solution we used works by first obtaining a)a 3x3 correlation matrix $H = \sum_{i=1}^{N} m_c, id_c, i^T$ where each vector set is centered and b)its SVD decomposition matrices $H = U\Sigma V^T$. Then the $\hat{R}, \hat{T}$ that we want is $\hat{R} = VU^T$ and $\hat{T} = \bar{d} - \hat{R}\bar{m}$ (whereby $\bar{d}$, $\bar{m}$ represents the mean vector of the set $\{d_i\}$, $\{m_i\}$). In the special case when the determinant of $\hat{R}$ is -1, the reflection of the rotation is produced, and we can correct it by using $\hat{R} = V'U^T$ instead, where $V' = [v_1, v_2, -v_3]$ and v_i is the i-th column of V.

This method is wrapped inside a RANSAC algorithm[10]. At each iteration, a number of match points are sampled. To find the optimal way of calculating inliers for each RANSAC transformation, we designed and experimented with the following three modes (all calculated after the estimated transformation is applied).

Mode "Matches": In this mode, we define an inlier as a pair of matching keypoints with a $l2$ distance smaller than a pre-defined threshold `maxDist`.

Mode "Coverage": In this mode, an inlier is defined as a cloud point in one frame that has a neighboring point in the other frame within a pre-defined threshold `maxDist`.

Mode "Dynamic": In this mode, the system records both the previous mentioned inlier statistics.

This is computed iterative e.g. 10000 times, and the transformation associated with the most number of inliers is used as the transformation for the whole frame of the cloud points. In the "Dynamic" mode, there are two "best" transformations one based on each calculation of the inliers, and the "coverage" one is only used when the distance between the transformed inliers from "matches" are bigger than a empirically defined threshold.

In the reported result, only the mode "Matches" is used as the performance of the other two modes are not consistently better over the data set.

2.2.3. Fine tuning I - 6D Refine

We observed that sometimes the RANSAC based transformation as described in Section 2.2.2 introduces a significant error. This may still place two adjacent frames reasonably close to each other, but leaves a significant offset and angle between the two. The ICP method can only solve this partially, which can be understood by comparing its operation with the effect of hooking rubber bands to each point in frame A, and to connecting it to its closest neighbour in Frame B. Apart from the desired effect (frame A is pulled towards frame B), this also introduces two unwanted side effects: First of all parts of the frame A that should not

[8] See `get_matches()` in SIFT.py

[9] An value found empirically

[10] See `estimate_transform()` in multi_head.py

overlap with frame B are still tied with rubber bands to frame B. The effect this has on the rest position towards which frame A is pulled is exacerbated by the fact that especially these rubber bands from non overlapping sections are stretched out most, therefore applying a relative big "force" away from the ideal position. Secondly, if two frames are already overlapping over a significant area, but still need to be shifted w.r.t to each other, ICP will not help, as the rubber bands will now be tied between sections of the frames that are shifted w.r.t. to each other, introducing a force that will counter any correction of the offset.

To overcome these issues with ICP we thought of an alternative algorithm, which we called the the 6D-refine algorithm. The objective of this algorithm is not not minimize certain distances, but rather to maximise the number of pairs of points between two frames that are close to each other. SO more formally we are trying to maximize the number of pairs - $\{a_i, b_i\}$, where a_i and b_i are each a point in frames A and B, such that for each pair $\|a_i - b_i\|$ is small. This is done by changing (wiggling) the position and orientation of frame B w.r.t to frame A. The position of frame B is varied over its 6 degree-of-freedom: 3 angles ϕ, ρ and θ and 3 Cartesian axes, x, y and z, hence its name: 6D refine.

The 6D refine works in two phases: In the *first* phase[11], Frame B is step wise rotated over $\pm 25^\circ$ for each angle and step wise translated over 10 cm for each axis, all w.r.t to the starting position. (So the algorithm does not sample the full 6 dimensional space, which would be computationally prohibitive). This allows the algorithm to scan the solution space coarsely, and identify positions/orientations of Frame B that increase the number of points of frame B that are close to Frame A.

In the *second* phase[12], frame B is moved to look for the local optimum of n for each of the 6 parameters. Each parameters ϕ, ρ, θ, x, y and z is individually increased or decreased until further increasing or decreasing does not further improve n. And then this local optimization is repeated until n does not change anymore between subsequent sweeps. The maximum distance between adjacent points in the second phase (0.02) is set to a smaller value than in the first phase (0.01).

The effect of 6D refine between two frames is shown in the Figure 4.

2.2.4. Fine tuning II - Iterative Closest Point (ICP) algorithm

After the previous transformations from Section 2.2.2, the $l2$ distance between matching keypoints are low, but this is not guaranteed for the other actual-corresponding points between these two frames. The ICP algorithm is used to compute another fine tuning transformation, on top of the current one, to refine the registration. The ICP works by first set a number N which is less than the number of points in either frames, sample N points from the first frame, then for each point sampled, find the closest point in the other frame in terms of Euclidean distance. This results in two sets of points of the same size from the two frames of cloud points. Then a transformation is estimated to minimize the sum of the distances combined using exactly the same method as the algorithm discussed in Section 2.2.2 except without RANSAC.

However, the ICP does not work as good as we expected in our situation, because we suspect there are non-corresponding points on the edge of the frames (e.g. the right edge of the frame on the right and the left edge of the frame on the left) will shift the frames to a worse registration in some cases. Thus we decide not to include it in our final work. Some remedies are discussed in Section 4.

2.3. Fuse all of the points into a single 3D coordinate system

For each pair of consecutive frames i, j we stored the sets of resulting SIFT matches in a separate data structure, "Link"[13] which describes the connection between any pair of consecutive frames. This data structure allowed us to pre-compute the SIFT matches using RANSAC as described in Section 2.2.1, and to only apply the actual transformation $\mathbf{T}_{i,j}$ based on the calculated matches when fusing the frames together. In this way we could use the same SIFT matches, but choose freely in which order we would assemble the entire 3D model.

When performing the final fusion of the individual frames, we used 3 different methods, each using a different order for assembling the individual frames:

Method A: Starting with frame 1 and moves all the way through to frame 15.

Method B: Starting at the back (Frame 8) and then stitching the frames from either side until reaching the front.

Method C: In this method we first examine the quality of the SIFT matches between each pair of adjacent frames. The quality is based on an estimate of the transformation error. We calculate the mean squared transformation error as:

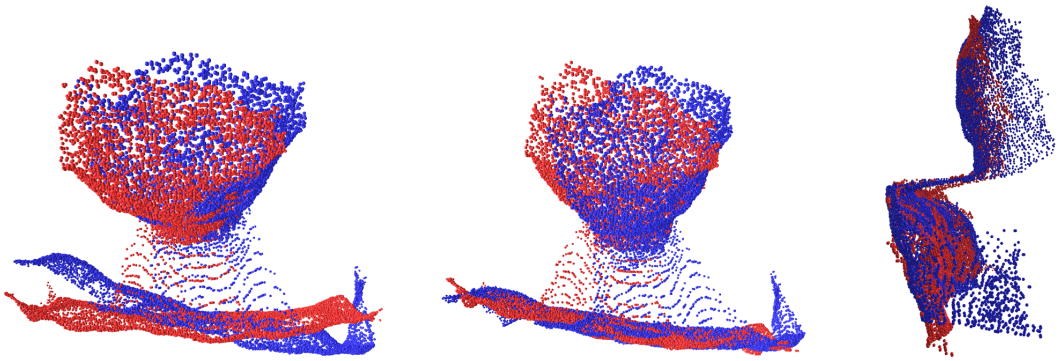
$$MSE_T = \frac{1}{N-6} \sum_{i=1}^{N} \|d_i - \hat{R} m_i - \hat{T}\|^2$$

This is a measure for the accuracy of the transform. The

[11] See `refine_over_range()` in refine.py

[12] See `refine_local()` in refine.py

[13] See `class Link():` in link.py



(a) (b) (c)

Figure 4. The effect of the 6D-Refine algorithm on two subsequent frames in sequence 1: 4a before 6D-Refine, 4b and 4c after 6D-Refine seen from the top and the left-hand side. The point clouds have been coloured and sparsified to clearly show how well shapes match

number 6 reflects Bessel's correction to account for the degrees of freedom in the transformations we considered. Note that this error metric does not reflect the expected error in individual transformations, but rather the contribution in the the expected error of individual transformations from the error in the estimation of the translation itself. This makes this metric suitable for comparing the quality of a set of SIFT matches.

We used this error metric to determine the order in which we stitch new frames to the set of already fused frames. We start with the pair i, j that has the smallest MSE_T and then subsequently add the left- or right-hand adjacent frame to the set of fused frames, depending on which one has the smallest value for MSE_T. This ensures that we do not use the transformation associated with the pair with the worst MSE_T, which significantly improves the quality of the overall set of fused frames.

2.4. Building as complete a 3D model as possible

After all the frames have been transformed to one system, there are many layers of points corresponding to the same region of the head (e.g. the person's left eye appears in more 5 frames). We apply a scanning algorithm to reduce the points to one single mesh. This mesh is generated from a set of radials, each radiating out from an axis through the center of the torso. The position of this axis is marked in the images in Figure 7 by the green and red dot, just above and below the center of the torso. The radials are now each defined by their height y and angle ϕ. The y-values range from -0.35 to 0.20 and are separated by a 0.003 interval. The ϕ values range from 0° to 360° in steps of 2°. For each radial we project the points that are within a small distance (0.01) from the radial, onto the radial. For these points we calculate the mean RGB color and median distance to the center axis. This results in a new dot on the radial. We now obtain a regular mesh of points.

As a result, the total number of points of the model are reduced without losing much details of the head.

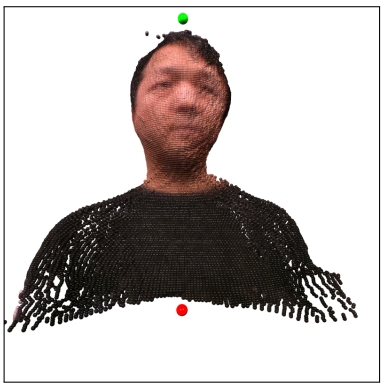


Figure 5. The resulting mesh for sequence 1.

3. Performance on The Test Data

The performance of the four stages of the algorithms introduced above are each discussed in detail in the following four subsections.

3.1. Data Extraction

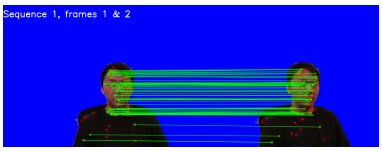
As briefly mentioned before, Figure 1 shows the results after each of the 3 key steps during pre-processing: the NAN filter, the depth filter and the background color filter.

In the end, a large proportion of a flying pixels belonging to the background are removed. However, there are still some

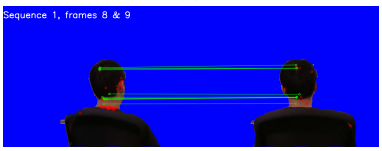
flying pixels with color similar to the background remaining, e.g. from the metal frame of the whiteboard, and we can already see that there are pixels from the person starting to be removed, e.g. the pixels on the sweater of the second row, due to a)its color being similar to the background and b)a NAN pixel on the sweater which allows the near by pixels to pass the "edge pixel test". Thus we cannot further increase the threshold d_{color}. The effect of these irrelevant pixels to the following processing is insignificant because of the removal of the SIFT descriptors near the edge of the person and our final pixel reduction method.

3.2. SIFT keypoint extraction and matching

We show two examples of key point matching in Figure 6. The keypoints that were matched by the RANSAC algorithm are shown in green, the unused ones in red. We can see that there are many more matches between adjacent frames for the two frames showing the front, as there are for the ones showing the back.



(a)



(b)

Figure 6. Examples of keypoint matching

All remaining SIFT matches can be found in appendix Figure 10, 11, 12 and 13 in the appendix show the detected SIFT keypoints for each of the 4 sequences.

Table 1, 2, 3 and 4 show the quality of the of the obtained matches. The number of matches shows the number of potential matches between keypoints. The number of inliers shows the number of matches that were selected by the RANSAC algorithm. $\sqrt{MSE_T}$ shows the root of the expected mean squared error in the transformation, as defined in Section 2.3. The underlined values are $\sqrt{MSE_T}$ are the best performing ones, whereas the **bold** ones are the worst values.

Seq:	1		
Pair	Matches	Inliers	$\sqrt{MSE_T}$
1-2	248	91	0.0061
2-3	264	107	0.0053
3-4	237	89	0.0083
4-5	241	96	0.0073
5-6	172	56	0.0084
6-7	117	42	0.0059
7-8	68	26	0.0114
8-9	29	10	0.0157
9-10	14	6	**NaN**
10-11	33	16	0.0205
11-12	69	21	0.0084
12-13	88	28	0.0084
13-14	127	52	0.0086
14-15	165	57	0.0119
15-1	271	116	0.0045

Table 1. Quality of SIFT transformations in terms of number of inliers and $\sqrt{MSE_T}$ for Sequence 1

Seq:	2		
Pair	Matches	Inliers	$\sqrt{MSE_T}$
1-2	921	334	0.0027
2-3	763	302	0.0036
3-4	680	268	0.0037
4-5	639	217	0.0041
5-6	585	233	0.0033
6-7	295	111	0.0048
7-8	154	55	0.0032
8-9	137	50	0.0029
9-10	162	61	0.0034
10-11	150	57	0.0031
11-12	84	32	**0.0076**
12-13	230	97	0.0072
13-14	704	282	0.004
14-15	022	384	0.002
15-1	980	371	0.0025

Table 2. Quality of SIFT transformations in terms of number of inliers and $\sqrt{MSE_T}$ for Sequence 2

3.3. Data Point Fusion

We explored 5 methods to determine the order in which frames are fused, in combination with additional post-processing to compensate for possible remaining transformation errors: Method A and method B (see Section 2.3, each without additional processing steps, Method A + ICP, in which the set of transformations is followed by ICP (see Section 2.2.4 method A followed by + 6D refine (see Section 2.2.3), and finally we also used method C (Section 2.3).

Seq:	3		
Pair	Matches	Inliers	$\sqrt{MSE_T}$
1-2	270	109	0.0023
2-3	283	103	0.0023
3-4	184	67	0.0058
4-5	149	62	0.0038
5-6	109	52	0.0028
6-7	91	35	0.0034
7-8	47	17	0.0081
8-9	21	8	**0.0195**
9-10	35	12	0.0057
10-11	52	21	0.0088
11-12	81	28	0.0041
12-13	111	52	0.0037
13-14	55	17	0.0146
14-15	156	57	0.0053
15-1	235	101	0.0025

Table 3. Quality of SIFT transformations in terms of number of inliers and $\sqrt{MSE_T}$ for Sequence 3

Seq:	4		
Pair	Matches	Inliers	$\sqrt{MSE_T}$
1-2	165	67	0.0011
2-3	145	60	0.0025
3-4	102	37	0.0061
4-5	83	34	0.0069
5-6	78	24	0.0091
6-7	86	35	0.0034
7-8	59	21	0.0047
8-9	23	6	**NaN**
9-10	20	6	**NaN**
10-11	27	7	0.0219
11-12	68	31	0.005
12-13	86	29	0.0059
13-14	82	26	0.007
14-15	136	49	0.0056
15-1	122	50	0.0054

Table 4. Quality of SIFT transformations in terms of number of inliers and $\sqrt{MSE_T}$ for Sequence 4

In order to judge the quality of the overall fusion process we manually annotated the position of the center of the left eye in each of the frames (if the left eye is visible). Using this data we calculated for each of the 5 methods the average distance from the mean positions of center of the eye markers. The results are given in Table 5

As can be seen, Method A and B yield exactly the same result. This can be understood by only considering the error each transformation introduces: As long as for each neighbouring pair $\{i, j\}$, both method A and B use the same set of SIFT matches, the resulting transformation $T_{i,j}$ introduces the exact same error. How the errors accumulate only depends on which pairs of adjacent frames are used. For method A and B these pairs are identical. IN both methods only pair $\{15, 1\}$ is not used. We see that ICP and 6D-Refine can improve these results. Especially the 6D-Refine method shows a good performance. Finally we have method C. Method C works really well when only looking at the mean deviation from the left eye markers, as the the transformations between the frames at the front are of better quality in terms of MSE_T, as compared to the ones on the back. This can be explained from the fact that the features on the face (nose, mouth, eyes etc) are very distinct, and provide good keypoints.

Seq:	A	B	A + ICP	A + 6D-Refine	C
1	0.047	0.047	0.032	0.020	0.0056
2	0.088	0.088	0.090	0.044	0.0023
3	0.035	0.035	0.029	0.031	0.0030
4	0.053	0.053	0.070	0.016	0.0018

Table 5. Mean deviation from left eye markers

The final results based on method C are shown in Figure 7

For the data point fusion we only use method C to generate the full set of fused points. This method ensures a better quality of the overall fused as we do not use the weakest link between individual frames. The resulting point clouds are shown in Figure 7.

3.4. Data Point Reduction

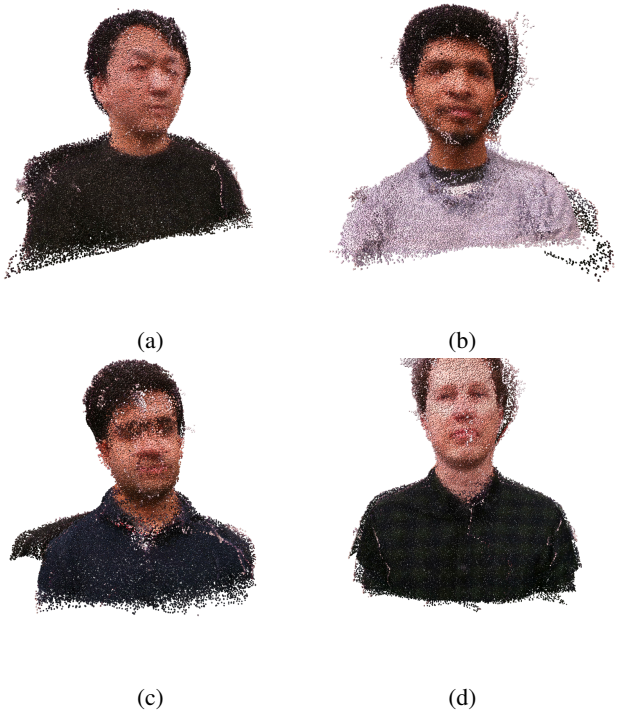
Finally we merge the data set into a single mesh, as described in Section 2.3. In the images we see that the points on the mesh are further apart if the distance from the core is greater. The resulting meshes for all heads area shown in Figure 8.

4. Discussion

There are some particular successes, failures and potential remedies in this work that we conclude to the following 3 points.

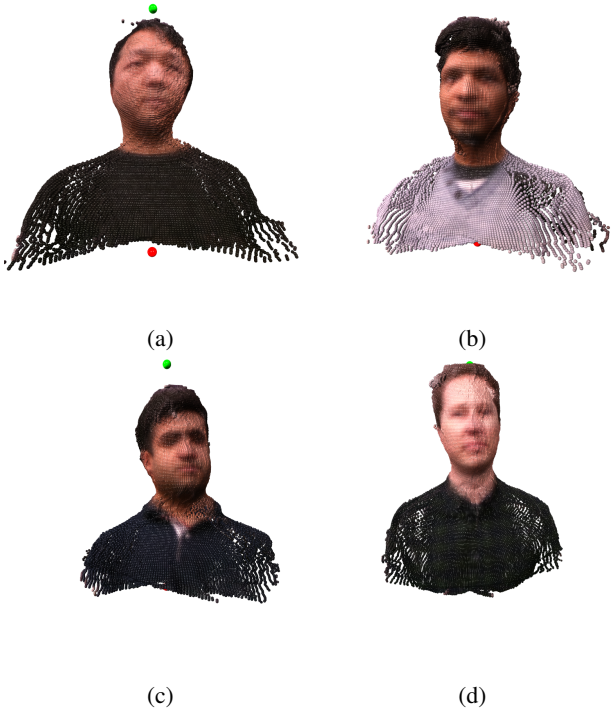
4.1. Hyper-parameter optimization

There are numerous hyper-parameters in the system, e.g. the contrast threshold, the edge threshold for SIFT, the sampling proportion for RANSAC, the ratio for the ratio test for matches, different modes to calculate the inliers, whether to apply ICP, etc, that are difficult to choose that can generalize



(a) (b) (c) (d)

Figure 7. Fused images of the individual sequences



(a) (b) (c) (d)

Figure 8. Mesh-rendered images of the individual sequences

to the entire data set. A set of parameters that work well with one sequence might lead to inconsistent performance for another. This naturally leads us to looking for an general objective function; whereby it is defined in terms of the parameters mentioned above and outputs a numeral value as an indication of the transformation estimation quality. This allows us to optimize all the parameters with respect to individual "links". Such a function should take into account of both the quality of matches points and the percentage of coverage (as calculated in the mode "coverage" in Section 2.2). Such a function is not tested systematically on the data set due to our time and computing power constraints, and it can be investigated in future works.

4.2. Fusing all points into a single 3D system

The overall process of fusing the heads depends strongly on the quality of the SIFT matches, which varies from frame to frame and is in itself strongly depending on hyper-parameter values. We did find that the MSE_T metric provides a good measure of judging the quality of the found transformation. This led us to using the MSE_T of the transformation to decide on the order in which the frames are stitched together, which we referred to as method C. Method C provided a considerable improvement over method A and B(, which are effectively the same).

It is interesting to see how using method A (or B) results in fused heads that seem to have two faces in the front. It is clear that because we stitch together the frames ending at the front side, that the errors will all accumulate between frames 1 and 15, appearing as a with two faces. We can see from the bottom view that the 6D-refine method was able to ensure all the surfaces align well, however we do not cover the full 360°. This can be explained from the fact that all transformation functions we use try to find as many matches between the individual frames as possible. This leads to greater overlap between the individual frames, more than there should have been. This then leads to making a smaller rotation than the full 360°.

(a) (b)

Figure 9. Method A resulting in two faces at the front, front and bottom view

As described in Section 2.2.2 we also implemented an alternative RANSAC mechanism, in which we did not try to minimize the transformation error MSE_T, but rather tried

to maximize the "coverage", where the coverage is the fraction of points that has a neighbour in the other frame that is within a certain maximum distance. A potential advantage is that it is quite resilient against errors in the SIFT matches, as it does not judge the quality of the found transformation based on SIFT matches, but rather on proximity between all potential point pair. However, it is not as accurate as a good SIFT match itself, because we set the maximum distance between point pairs to a relatively large number for robustness, which in itself set a lower bound on the achievable accuracy. Secondly it still requires finding a set of SIFT matches that yields a transformation with high coverage. In that sense it still depends on good SIFT matches. This method worked sometimes better and sometimes less good than the standard RANSAC method. For this reason we decided not to use the "coverage" method.

Then finally the 6D refine method proved capable of correcting some significant transformation errors. Especially the frames with the backs of the persons had less good SIFT matches and therefore a greater MSE_T. Here we saw that the 6D-refine method proved beneficial as it corrected some significant transformation errors. The first part of Section 2.2.3 provides insight in the underlying mechanism that made this method better than ICP. Figure 4 shows a practical example of the 6D refine algorithm at work.

4.3. Reducing the total number of points

The mesh generation method as described in Section 2.4. It works reasonably well, and provides the additional advantage that it not only defines a set of points, but also provides an unambiguous definition of a surface of the person. However, we do realize two downsides: firstly, the point density becomes a linear function of the distance to the center of the torso. This leads to loss of resolution around the shoulders. Secondly, by setting a certain maximum distance to the radial and using the average color of all points within that distance as the color of the resulting point in the mesh, the resulting colors of the head become slightly blurred. Interestingly we found that for determining the projected position of the resulting points on the radial, the median provided a smoother surface than the mean. This can be explained from the fact that the median by natures is robust to outliers. We also see that the mesh face is smaller than what we see when rendering the fused point sets. This is because what we see from the latter is only the frames that form the outside of the torso, whereas our mesh follows the contours of the frames in the middle.

References

Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Lorusso, A., Eggert, D. W., and Fisher, R. B. A comparison of four algorithms for estimating 3-d rigid transformations. In *BMVC*, 1995.

A. SIFT Matches

The SIFT descriptor matches between all the consecutive frames in the data set.

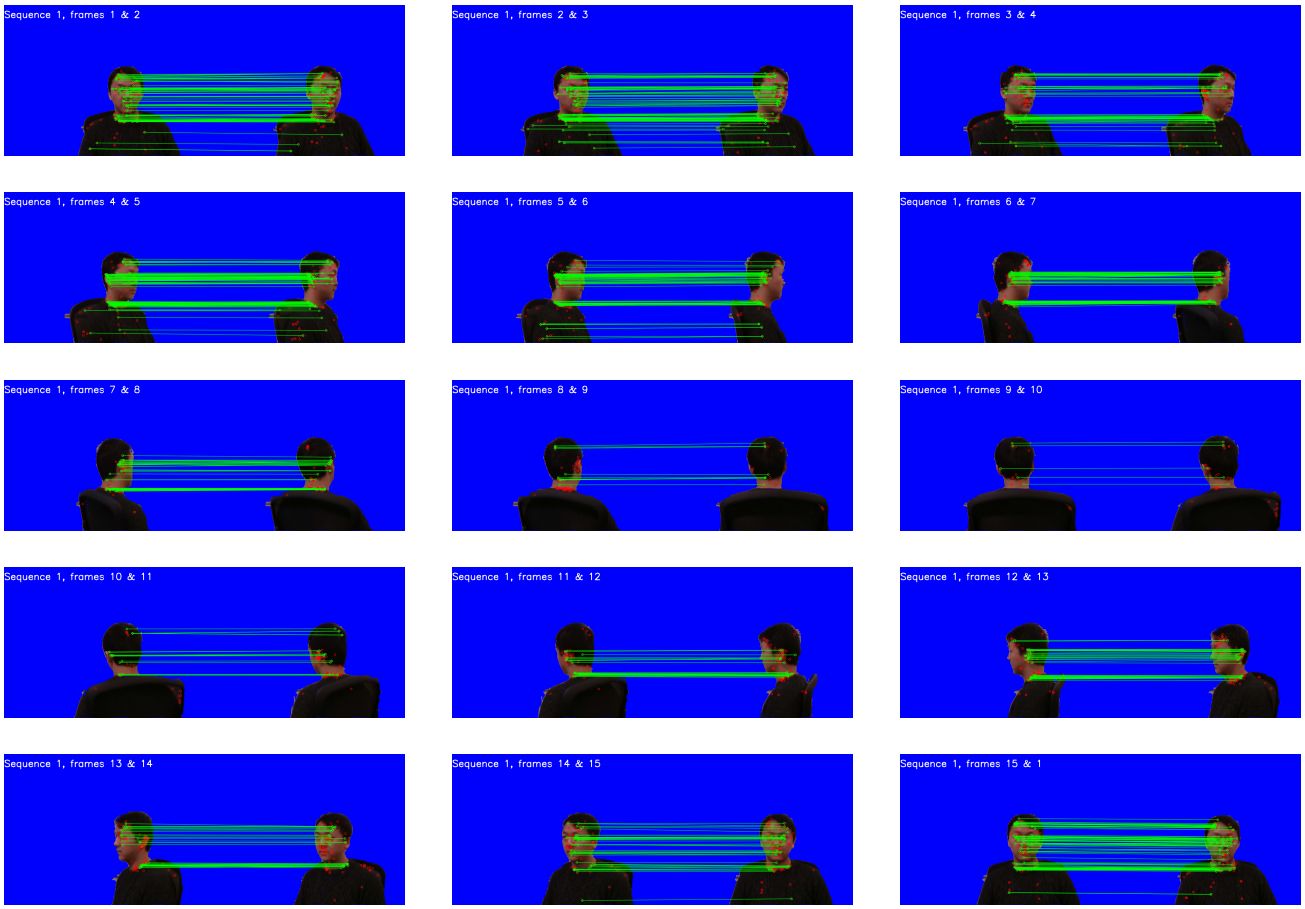


Figure 10. SIFT Points for all possible pairs of adjacent frames for Sequence 1. Green ones are the ones that match, red ones are the ones that do not match

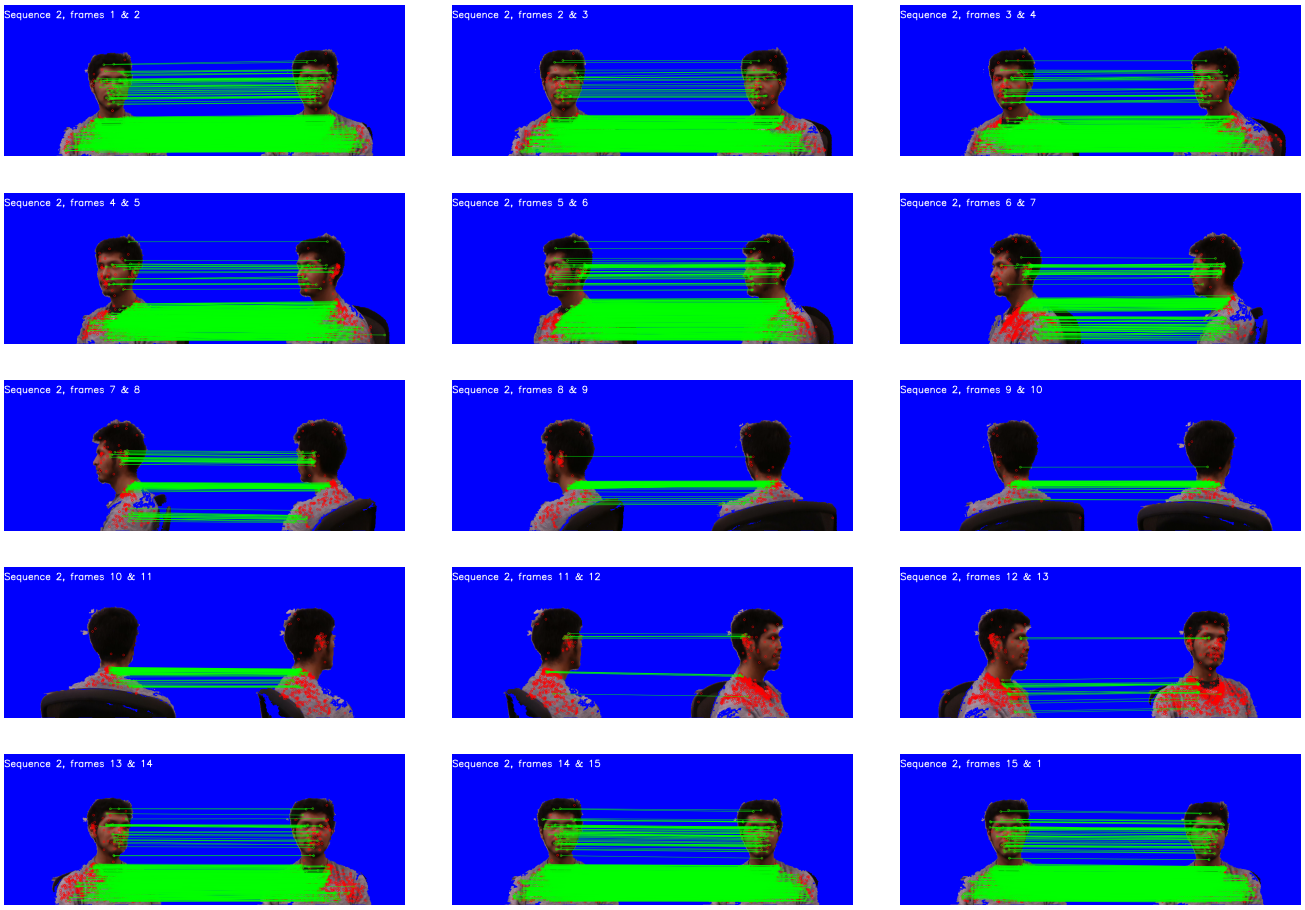


Figure 11. SIFT Points for all possible pairs of adjacent frames for Sequence 2. Green ones are the ones that match, red ones are the ones that do not match

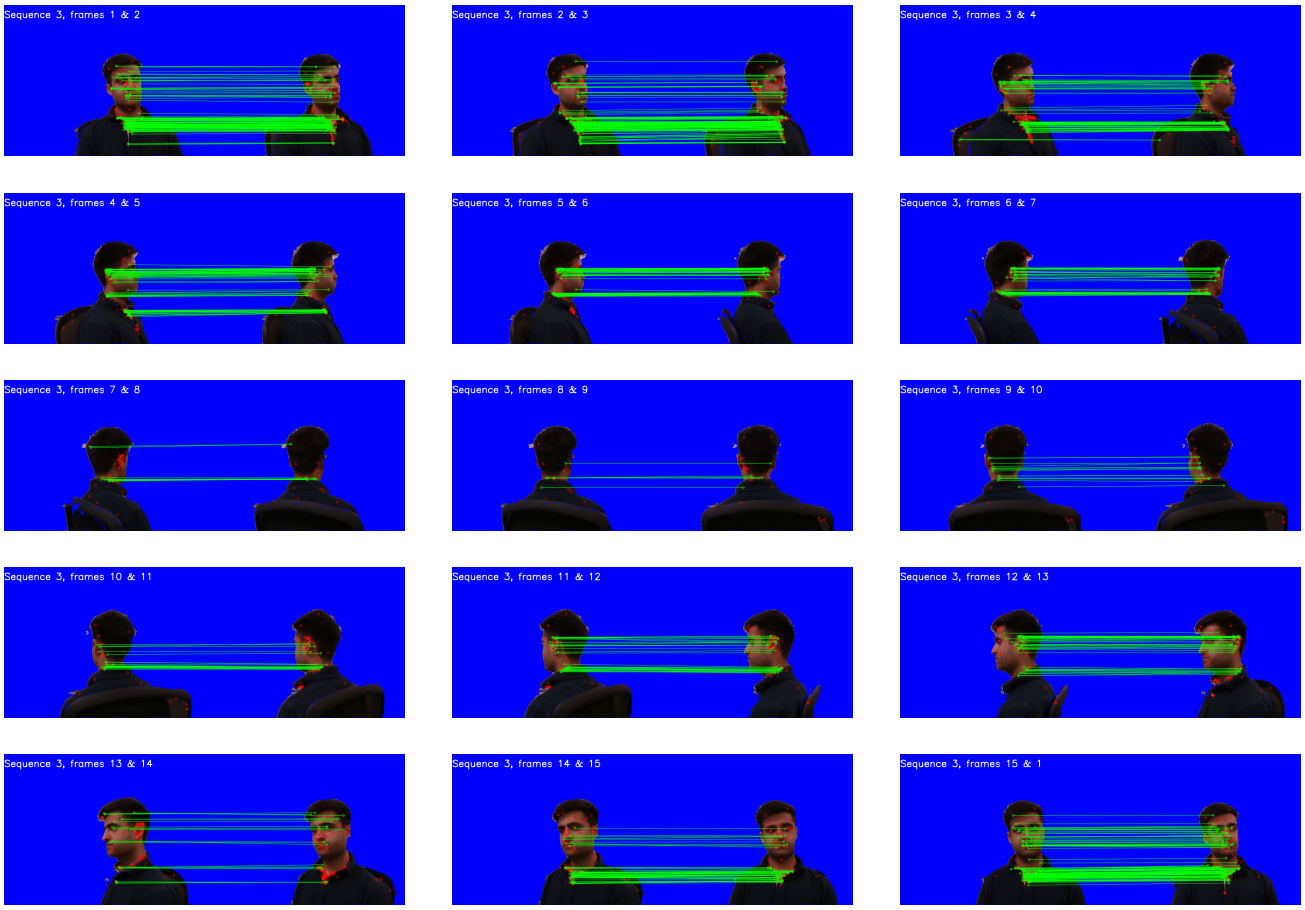


Figure 12. SIFT Points for all possible pairs of adjacent frames for Sequence 3. Green ones are the ones that match, red ones are the ones that do not match

Figure 13. SIFT Points for all possible pairs of adjacent frames for Sequence 4. Green ones are the ones that match, red ones are the ones that do not match

B. Frame Fusion Results

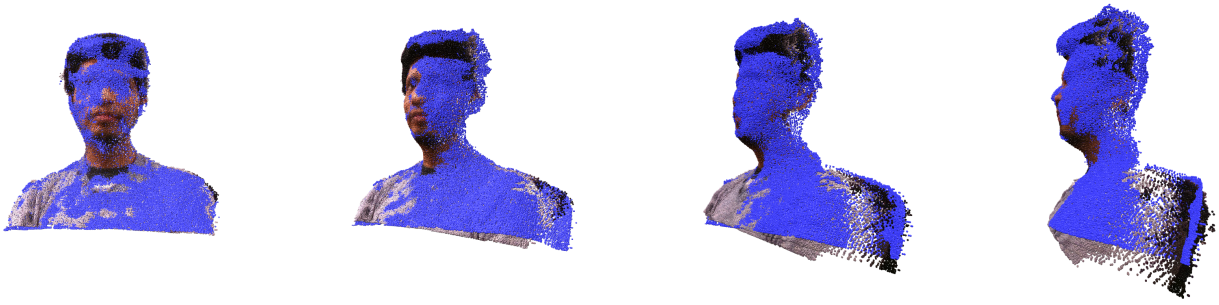
The full merge set of the cloud point of each sequence with our method A. The newly added frame is colored blue in each figure, with the full head in the end.

(a) (b) (c) (d)

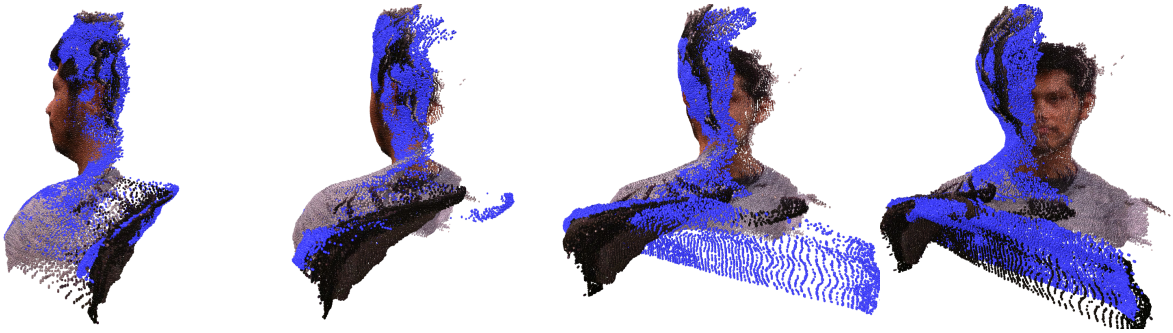
(e) (f) (g) (h)

(i) (j) (k) (l)

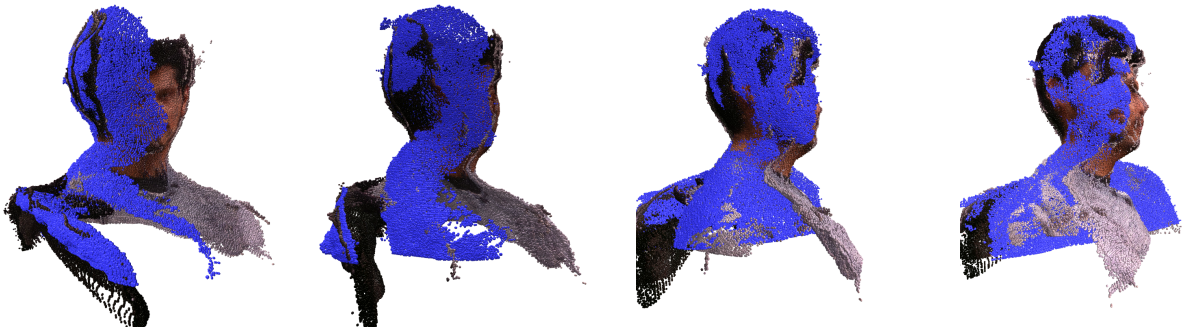
(m) (n)



(a) (b) (c) (d)

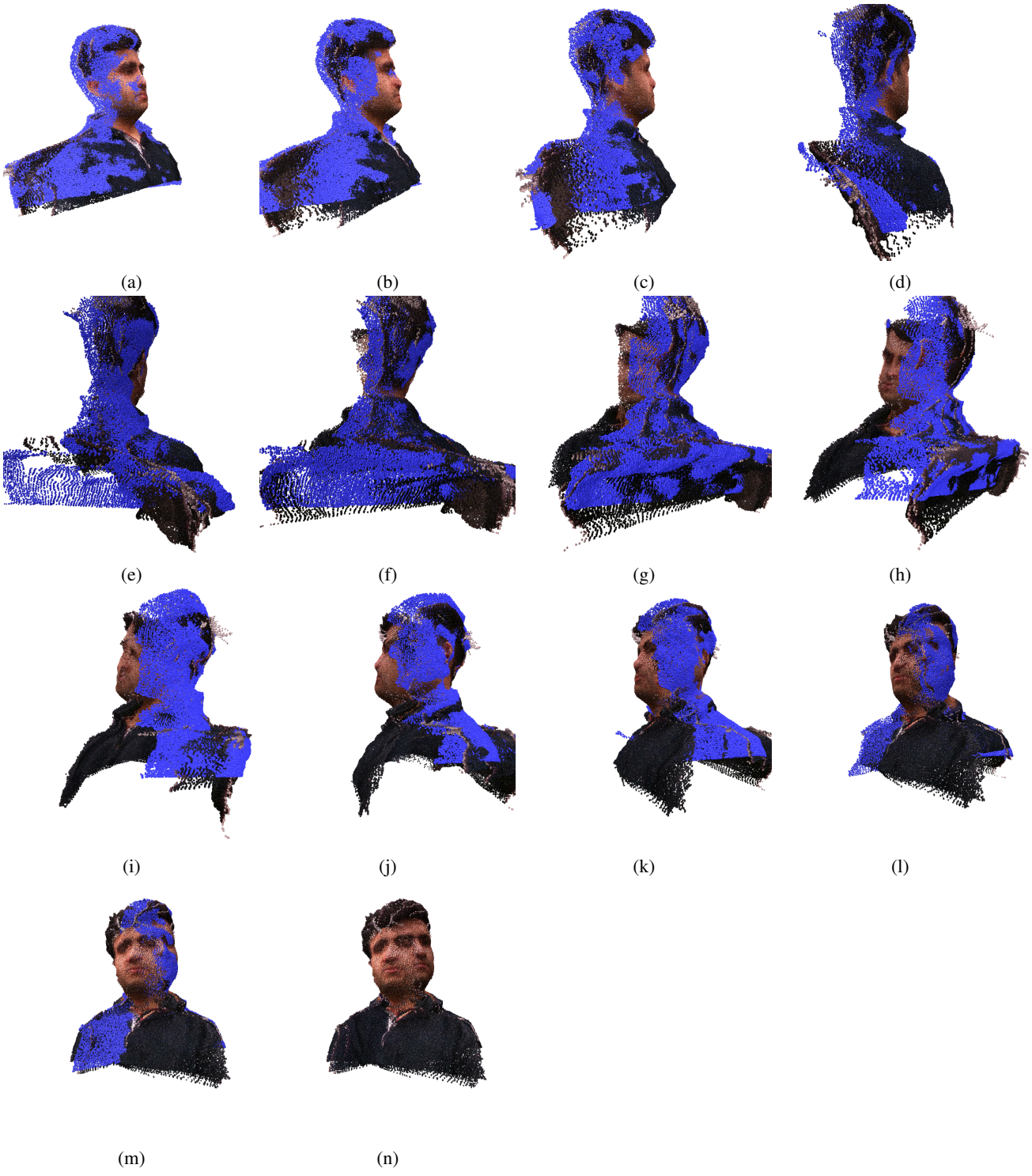


(e) (f) (g) (h)



(i) (j) (k) (l)

(m) (n)



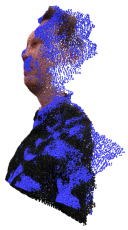
(a) (b) (c) (d)

(e) (f) (g) (h)

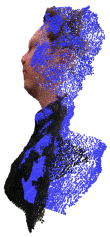
(i) (j) (k) (l)

(m) (n)

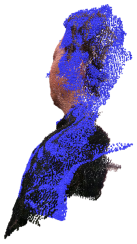
(a)



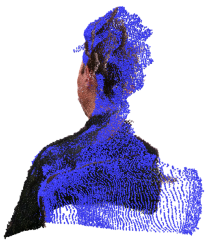
(b)



(c)



(d)



(e)

(f)

(g)

(h)

(i)

(j)

(k)

(l)

(m)

(n)

C. Code base

The code for this coursework is subdivide over the following nine files:

- **av_cw.py** Top Level Code for AV Coursework. It merges individual point clouds into a single MulitHead object. Produces images of the individual merging steps and calculates performance metrics.
- **single_head.py** Defining the SingleHead class, used for pre-processing on individual frames. It's main discussed in Section 2.1 of the report.
- **multi_head.py** Defining the MulitHead class used for joining a set of SingleHeads objects into one complete head. It imports functions from SIFT.py, icp.py, refine.py (mentioned below) to generate as complete a head as possible.
- **link.py** Defines the Link object, which contains all required variables to store the results of SIFT matches between two subsequent frames.
- **SIFT.py** for extracting keypoints from images and estimating the 3D transformation.
- **icp.py** for performing ICP transformation as discussed in Section 2.2.4.
- **refine.py** for performing the 6D refine method, as discussed in Section 2.2.3.
- **gui.py** for displaying 3D objects such as the 3D model of the head.
- **procrustes.py**. Performing the Procrustes analysis without the RANSAC, as discussed in Section 2.2.2. We borrowed this function from the internet so it is not listed in the appendix.

These files are shown below:

av_cw.py:

```
'''
Top Level Code for AV Coursework
Merges individual point clouds into a single mulithead object
Produces images of the individual merging steps and calculates performance metrics
'''
from single_head import SingleHead
from multi_head import MultiHead
import numpy as np

# Seqence number in the data, range from 1 to 4.
Sequence = 1

# Frame number in the data, range from 1 to 15.
for frame_idx in range(1, 16):  # loop through all frames
    # Read the data from the file, define by Sequence and frame_idx and store the data in
    a SingeHead object
    head = SingleHead.read_from_file(Sequence,
                                     frame_idx)
    print(frame_idx)
    # apply all filters to remove the unwanted cloud points.
    head.apply_all_filters()
    # save the processed head
    head.save()

# create a list of all the heads, from the save SingleHead objects
list_of_all_heads = [SingleHead.load_from_pickle(Sequence, i) for i in
                     range(1, 16)]

# create a MultiHead object from the list:
mhead = MultiHead.create_from_heads(list_of_all_heads)
mhead.save()
# calculate teh SIFT points for each SingleHead in the MultiHead object:
```

```
mhead.calc_all_sift_keypoints()
# calculate the SIFT transform for each pair of adjacent heads, each pair of adjacnt SingleHeads shares a Link Object:
mhead.calc_all_sift_transforms()

# Set merge method to either A-C
method = 'A'

if method == 'A':
    # corresponds to Method A in the documentation/report.
    mhead.Method_A(sift_transform_method="matches", icp=True, refine_range=False, refine_local=False)
elif method == 'B':
    # Method B
    mhead.Method_B(sift_transform_method="matches", icp=True, refine_range=False, refine_local=False)
elif method == 'C':
    # Method C
    mhead.Method_C(sift_transform_method="matches", icp=True, refine_range=False, refine_local=False)

# create a series of png images for the spheres
mhead.create_png_series()

# calculate the mean eye distance:
mean_eye_dist = mhead.left_eye_deviation()

# Save the MultiHead object for later re-use:
mhead.save()
```

single_head.py:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import os
import struct
from vpython import *
import pickle
import cv2
from scipy.ndimage.morphology import binary_fill_holes
from SIFT import *
from tqdm.notebook import tqdm
from sklearn.neighbors import NearestNeighbors


def float_2_rgb(num):
    packed = struct.pack('!f', num)
    return [c for c in packed][1:]


class SingleHead():
    '''
    Class for manipulating individual frames of the head.
    '''
    def __init__(self, data_path=None):
        self.data_path = data_path
        self.background_color = [0,0,1]
        self.keypoints_clr = [1, 0, 0]
        self.visible = True

    @classmethod
    def read_from_file(cls, sequence_id, frame_id):
        '''
        Read from a .pcd file and extract its xyz and rgb values as
```

```
        two seperate lists, stored in xyz, rgb class variables.
        params:
        sequence_id (int): The person's number, from 1 to 4.
        frame_id (int): The frame number, from 1 to 15.
        depth (float): threshold parameter for the depth filter.
        '''
        # compose the data path
        data_path = f"./Data/{sequence_id}"
        this = cls(data_path)
        file_name = os.path.join(data_path, f"{sequence_id}_{frame_id}.pcd")

        # read the .pcd file
        pc = np.genfromtxt(file_name, skip_header=13)

        # extract the xyz, rgb info
        this.xyz = pc[:, 0:3]
        this.rgb = np.asarray([float_2_rgb(num) for num in pc[:, 3]])/256
        this.xyz_unfiltered= pc[:, 0:3]
        this.rgb_unfiltered= np.asarray([float_2_rgb(num) for num in pc[:, 3]])/256
        this.sequence_id=sequence_id
        this.frame_id=frame_id
        this.img_coord_from_xyz()

        return this

    @classmethod
    def load_from_pickle(cls, sequence_id, frame_id):
            return cls.load(f"pickled_head/head{sequence_id}_{frame_id}.p")

    @classmethod
    def load(cls, data_file='head.p'):
        '''
        :param data_file:  file to load from, default name is the default file used for
    saving
        :return: object of  threeD_head class
        '''
        try:
            with open(data_file, 'rb') as file_object:
                raw_data = file_object.read()
            return pickle.loads(raw_data)
        except:
            raise FileExistsError (f'{data_file} could not be found, create {data_file} by
     using .save() first ')

    def apply_all_filters(self, depth=1.5):
        '''
        perform thresholding in depth axis, remove the nan pixels and the flying pixels.
        Then center the pixels, create vpython spheres
        '''
        self.reset_filters()
        self.filter_nan()
        self.filter_depth(depth)
        self.remove_background_color()
        # two experimental filters that are not applied in the real production.
        # self.edge_based_filter()
        # self.parzen_filter()
        self.center()

    def color_eye(self,row,column):
        '''
        Color the left eye point in red
        '''
        self.left_eye_ind = row * 640 + column
        image = self.twoD_image.copy()
        image = image.reshape(-1,3)
```

```
        image[self.left_eye_ind] = [1,0,0]
        image = image.reshape(480,640,3)
        plt.imshow(image); plt.show()
        plt.imsave("left_eye_mark.png",image)

    def img_coord_from_xyz(self):
        '''
        Convert a list of rbg values to a 2D image
        params:
            rgb (list[int,int,int]): a list of rgb values
        return:
            the rgb image in a 480, 640 format
        '''
        image = np.reshape(self.rgb,(480,640,3))
        self.xy_mesh=np.arange(640*480)
        self.twoD_image= image
        return image

    def get_filtered_image(self,filename=None):
        '''
        return the 2d image after all filters have been applied
        :return: returns image filterd by all filter operations, black where filters have
    been applied
        '''
        twoD_image = self.twoD_image.copy().reshape(-1, 3)
        if not self.background_color is None:
            img = np.ones((480 * 640, 3)) * self.background_color
        else:
            img = np.zeros((480 * 640, 3))
        for v in self.xy_mesh:
            img[v] = twoD_image[v]

        return img.reshape((480, 640, 3))

    def save_filtered_image(self,filename=None):
        '''
        save the 2d image after all filters have been applied
        :return: None
        '''
        twoD_image = self.twoD_image.copy().reshape(-1, 3)
        if not self.background_color is None:
            img = np.ones((480 * 640, 3)) * self.background_color
        else:
            img = np.zeros((480 * 640, 3))
        for v in self.xy_mesh:
            img[v] = twoD_image[v]
        # save image to head_2d_image
        image_dir = "head_2d_image"
        save_path = os.path.join(image_dir,"head_{}_{}{}.png".format(self.sequence_id,\
        self.frame_id,filename))
        img = img.reshape((480,640,3))
        plt.imsave(save_path,img)

    def reset_filters(self):
        '''
        Resets all the filters and create a xy_mesh variable storing the indices of the
    original/unfiltered pixels, which becomes
        useful when some pixels are filtered out later (when the length of xy_mesh is less
     than 640*480). So e.g. we can say after several
        filters that the first element in the xy_mesh corresponds to the pixel 105731 in
    the original image.
        '''
        self.xy_mesh = np.arange(640 * 480)
        self.xyz=self.xyz_unfiltered
        self.rgb=self.rgb_unfiltered
```

```
158
159     def reset_positions(self):
160         '''
161         Resets the xyz position to the untransformed position.
162         '''
163         self.xyz=self.xyz_unfiltered[self.xy_mesh]
164
165     def reset_colors(self):
166         '''
167         Resets the rgb values to the unfiltered, unmarked rgb values
168         '''
169         self.rgb=self.rgb_unfiltered[self.xy_mesh]
170
171     def get_bw_image(self):
172         '''
173         :return: black and white image with all pixels set to white that have not been
    filtered out
174         '''
175         img= np.zeros((480*640,3))
176         for v in self.xy_mesh:
177             img[v]=[1,1,1]
178         return img.reshape(480,640,3)
179
180     def get_bool_image(self):
181         '''
182         :return: a boolean image, True for all pixels thah have not been filtered out
183         '''
184         img = np.zeros(480 * 640) > 0
185         for v in self.xy_mesh:
186             img[v] = True
187         return img.reshape(480, 640)
188
189     def transform(self, tform):
190         '''
191         transform the cloud points:  XYZ*cR + t
192         '''
193         R, c, t = tform['rotation'], tform['scale'], tform['translation']
194         self.xyz = self.xyz.dot(c * R) + t
195         self.keypoints = self.keypoints.dot(c * R) + t
196
197     def transform_homo(self, T):
198         '''
199         transform the image:  XYZ*T
200         '''
201         ones = np.ones((1, self.xyz.shape[0]))
202         self.xyz = np.concatenate((self.xyz, ones.T), axis=1)
203         self.xyz = self.xyz.dot(T)
204         self.xyz = self.xyz[:, :3]
205
206     def paint(self, color):
207         '''
208         paint the entire head into one color
209         '''
210         color=np.asarray(color).reshape(-1)
211         # self.rgb = self.rgb.mean(axis=1).reshape((-1,1)).dot(np.asarray([[1,1,1]])) *
    color
212         self.rgb = self.rgb.mean(axis=1).reshape((-1,1)).dot(np.asarray([[0,0,0]])) +
    color
213
214     def edge_based_filter(self,up=150,down=370,left=260,right=480):
215         '''
216         Take the twoD_image attribute and generate a binary
217         filter based on edge detection and binary fill holes.
218         '''
219         # extract the s layer of the HSV image
```

```
        image = self.twoD_image.copy()
        plt.imsave("head_2d_image/full_{}_{}.png".format(self.sequence_id,self.frame_id),
    image)
        image = cv2.imread("head_2d_image/full_{}_{}.png".format(self.sequence_id,self.
    frame_id))
        edge = cv2.Canny(image,0,250)
        # generating as complete an edge as possible
        for i in range (3):
            _, contours, hierarchy = cv2.findContours(edge,cv2.RETR_TREE,cv2.
    CHAIN_APPROX_NONE)
            image_another = image.copy()
            cv2.drawContours(image_another, contours, -1, (255,0,0), 0)
            edge = cv2.Canny(image_another,0,250)

        plt.imshow(edge);plt.show()
        # manually crop out the person
        edge[479,:]=1
        kernel = np.ones((3,3))
        dilation = cv2.dilate(edge,kernel,iterations =4)
        dilation[:up,:]=0
        dilation[down:,:]=0
        dilation[:,right:]=0
        dilation[:,:left]=0
        # perform flood fill based on the boundry
        im_floodfill = binary_fill_holes(dilation)
        im_floodfill = im_floodfill*1
        im_floodfill = np.uint8(im_floodfill)
        erode = cv2.erode(im_floodfill,kernel,iterations=7)
        # filter out the unwanted pixels
        edge_filter = erode > 0
        edge_filter = np.ravel(edge_filter)
        filter = [edge_filter[i] for i in self.xy_mesh]
        self.xy_mesh = self.xy_mesh[filter]
        self.rgb = self.rgb[filter]
        self.xyz = self.xyz[filter]
        self.save_filtered_image("edge_based")

    def filter_depth(self,depth):
        '''
        :param depth: any pixel with depth greater than this value is removed
        :return:
        '''
        depth_filter = self.xyz[:, 2] < depth
        self.xy_mesh=self.xy_mesh[depth_filter]
        self.rgb=  self.rgb[depth_filter]
        self.xyz=  self.xyz[depth_filter]
        self.save_filtered_image("depth_filter")


    def filter_nan(self):
        '''
        removes all entries where any of the xyz coordinates is nan
        '''
        nan_filter = ~np.isnan(self.xyz).any(axis=1)
        self.xy_mesh=self.xy_mesh[nan_filter]
        self.xyz = self.xyz[nan_filter]
        self.rgb = self.rgb[nan_filter]
        self.save_filtered_image("nan_filter")

    def sparsify(self,sparsity):
        '''
        :param sparsity: the fraction of pixles that is retained
        :return: updates the object
        '''
        l = self.xyz.shape[0]
```

```
        filter = np.random.random((1)) < sparsity
        self.sparse_xy_mesh = self.xy_mesh[filter]

        self.sparse_xyz = self.xyz[filter]
        self.sparse_rgb = self.rgb[filter]

    def center(self):
        '''
        centers the object
        :return:
        '''
        self.center_pos= self.xyz.mean(axis=0)
        self.xyz = self.xyz - self.center_pos
        self.xyz_unfiltered = self.xyz_unfiltered - self.center_pos

    def parzen_filter(self,p=7,r=0.005):
        '''
        Remove the "flying pixels" if there are less than p pixels within a distance r from the 3D pixel.
        '''
        filter = np.ones(self.xy_mesh.shape) > 0
        start_cnt = np.sum(filter)
        end_cnt = 0
        remove_count = 0

        # perform removal for 10 iterations
        for _ in range(10):
            filter = np.ones(self.xy_mesh.shape) > 0
            start_cnt = np.sum(filter)
            length = len(filter)
            bool_img = self.get_bool_image()

            NN = NearestNeighbors()
            NN.fit(self.xyz)
            # check the number of pixels in each 3 by 3 window
            for i,(coord,index) in enumerate (zip(self.xyz,self.xy_mesh)):
                y=index//640
                x=index%640
                small_bool = bool_img[max(y-1,0):y + 2, max(x-1,0):x + 2]

                if np.sum(small_bool)<8:
                    # calculate the number of points near coord with a radius of r
                    num_within = len(NN.radius_neighbors([coord],radius=r,return_distance=False)[0])
                    print(num_within)
                    if num_within < p :
                        remove_count+=1
                        filter[i] = False
            # remove unwanted points from attributes.
            self.xy_mesh=self.xy_mesh[filter]
            self.xyz = self.xyz[filter]
            self.rgb = self.rgb[filter]
            end_cnt = np.sum(filter)
        print(remove_count)
        self.save_filtered_image("parzen_window")

    def remove_background_color(self):
        '''
        Updates the filters by removing colors on the edge
        :return:
        '''
        verbose = False
        min_grad=0.08
        # size of the window
        size=3
```

```
        # lower bound and upper bound offset
        lb=size//2 # 1
        ub=size//2+1 # 2
        filter = np.ones(self.xy_mesh.shape) > 0
        # number of pixel points before processing
        start_cnt=np.sum(filter)
        # mean colors
        m1=np.nanmean(self.rgb_unfiltered,axis=0)
        m2=np.nanmean(self.rgb,axis=0)
        self.bg_color = (m1*640*480-m2*start_cnt)/(640*480-start_cnt)
        end_cnt=0

        # hault when there are still pixels being removed from the image
        while end_cnt<start_cnt:
            bool_img = self.get_bool_image()
            filter = np.ones(self.xy_mesh.shape) > 0
            start_cnt = np.sum(filter)
            # loop through all the unfiltered pixels
            for i, index in enumerate (self.xy_mesh):
                # convert to x,y coordinate in the 2D image
                y=index//640
                x=index%640
                verbose =(y==200 and x > 270  and x < 300)
                verbose=False
                if x>0 and y >0 and x < 640-lb and y < 480-lb:
                    # sum the pixels in the window around the current pixel
                    small_bool = bool_img[y-1:y + 2, x-1:x + 2]
                    small_rgb = self.twoD_image[y - lb:y + ub, x - lb:x + ub]
                    ctr_color = small_rgb[lb, lb]

                    # if the pixel is on the edge of the filtered image
                    if np.sum(small_bool)<= 6:
                        if verbose:
                            print(small_bool)
                            print(x, y, ctr_color, self.bg_color)
                            print(np.linalg.norm(ctr_color - self.bg_color))

                        # if the pixel is too similar to the background color, then
removed
                        if np.linalg.norm(ctr_color-self.bg_color) < min_grad:
                            if verbose:
                                print("remove")
                            filter[i] = False
            # compute the number of left over pixel in the image
            end_cnt = np.sum(filter)
            # update the xy_mesh
            self.xy_mesh = self.xy_mesh[filter]
            self.xyz = self.xyz[filter]
            self.rgb = self.rgb[filter]

        self.save_filtered_image("background_color")

    def create_keypoints(self, SIFT_contrastThreshold, SIFT_edgeThreshold, SIFT_sigma):
        '''
        Get the SIFT keypoints and descriptions with the specified parameters and remove
the keypoints
        located at the edge of the foreground object.
        '''
        # fetch the filtered 2D image as img
        img = self.get_filtered_image()
        self.kp, self.des = get_descriptors(img, SIFT_contrastThreshold,
SIFT_edgeThreshold, SIFT_sigma)
        # remove the edges that are on the edge
        diameter = 15  # minimum distance from the edge
        self.kp, self.des = self.remove_edge_points(self.kp, self.des, diameter=diameter)
```

```
    def create_vpython_spheres(self, force_sparce=False):
        '''
        creates the spheres for either a frame or consecutive frames that can be used by vpython
        '''
        if force_sparce:
            sparce_xyz = self.sparse_xyz
            sparce_rgb = self.sparse_rgb
        else:
            sparce_xyz = self.xyz
            sparce_rgb = self.rgb
        radius = np.ones(sparce_xyz.shape[0]) * 0.0015
        rad = 0.001
        # generate vpython spheres
        self.spheres = [{'pos': vec(sparce_xyz[i, 0], -sparce_xyz[i, 1], -sparce_xyz[i, 2]), 'radius': 0.0015,
                         'color': (vec(sparce_rgb[i, 0], sparce_rgb[i, 1], sparce_rgb[i, 2]))} for i in
                        range(sparce_xyz.shape[0])]
        data_file = f"pickled_head/head_spheres.p"
        pickle.dump(self.spheres, open(data_file, 'wb'))

    def save(self, file_name=None):
        '''
        :param file_name: file name for storing the file
        '''
        if not os.path.isdir("pickled_head"):
            os.mkdir("pickled_head")
        if file_name is None:
            file_name = f"pickled_head/head{self.sequence_id}_{self.frame_id}.p"
        pickle.dump(self, open(file_name, 'wb'))

    def remove_edge_points(self, kps, des, diameter):
        '''
        Works on input sift keypoint and  descriptors and filter out the ones that are on the edge.
        '''
        coords = np.round([kp.pt for kp in kps]).astype("int")
        bw = self.get_bool_image()
        offset = diameter // 2
        # sum up the number of pixels in the squared within the proximity of descriptor points.
        filter = [np.sum(bw[coord[1]-offset : coord[1]-offset+diameter,
                            coord[0]-offset : coord[0]-offset+diameter]) == diameter ** 2 for coord in coords]
        return [kp for i, kp in enumerate(kps) if filter[i]], np.asarray([d for i, d in enumerate(des) if filter[i]])
```

multi_head.py:

```
import pickle
import icp
from vpython import *
from SIFT import *
from tqdm.autonotebook import tqdm
from refine import *
from link import Link
import numpy as np
from math import isnan
import pandas as pd


class MultiHead():
    def __init__(self):
```

```
        self.heads = []
        self.links = []
        self.frame_sequence = []

    @classmethod
    def joined_heads(cls, head1, head2):
        '''
        Class method that initiate MultiHead object, with head1, head2 being two SingleHead objects.
        '''
        this = cls()
        for head in [head1, head2]:
            this.append(head)
        return this

    @classmethod
    def load_from_pickle(cls, sequence_id, name=None):
        '''
        :param data_file:  file to load from, default name is the default file used for saving
        :return: object of  MultiHead class
        '''
        if name is None:
            print(f"Loading Sequence {sequence_id}...", end="")
            data_file = f"pickled_head/mhead{sequence_id}.p"
        else:
            data_file = f"pickled_head/{name}.p"
            print(f"Loading {data_file}...", end="")
        try:
            with open(data_file, 'rb') as file_object:
                raw_data = file_object.read()
            this = pickle.loads(raw_data)
        except:
            raise FileExistsError(f'{data_file} could not be found, create {data_file} by using .save() first ')
        # load the keypoint information from pickle
        for head in this.heads:
            if hasattr(head, 'kp'):
                head.kp = [cv2.KeyPoint(x=point[0][0], y=point[0][1], _size=point[1], _angle=point[2],
                                        _response=point[3], _octave=point[4], _class_id=point[5]) for point in head.kp]
        # load the link information from pickle
        for link in this.links:
            if hasattr(link, 'matches'):
                link.matches = [cv2.DMatch(_distance=match[0], _imgIdx=match[1], _queryIdx=match[2], _trainIdx=match[3])
                                for match in link.matches]
        print("done")
        return this

    @classmethod
    def create_from_heads(cls, list_of_heads, first=0, last=14):
        '''
        :param list_of_heads: A list of SingleHead objects
        :param first: frame id of the first head
        :param last: frame id of the last head
        :return: an MultiHead object
        '''
        print("creating mhead object from a list of SingleHead objects")
        list_of_heads[first].reset_positions()
        list_of_heads[first].reset_colors()
        list_of_heads[first + 1].reset_positions()
        list_of_heads[first + 1].reset_colors()
        this = MultiHead.joined_heads(list_of_heads[first], list_of_heads[first + 1])
```

```
        this.links.append(Link(left=list_of_heads[first + 1].frame_id, right=list_of_heads[first].frame_id))
        for i in range(first + 2, last + 1):
            this.links.append(Link(left=list_of_heads[i].frame_id, right=list_of_heads[i - 1].frame_id))
            list_of_heads[i].reset_positions()
            list_of_heads[i].reset_colors()
            this.append(list_of_heads[i])
            if i == last:
                this.links.append(Link(left=list_of_heads[first].frame_id, right=list_of_heads[i].frame_id))
        return this

    def left_eye_deviation(self):
        '''
        Compute the error deviation from the prelabeled points.
        param: sequence_id (int) the person's number
        Return: the mean and the deviations of the left eyes in the 3D model for the person with sequence_id.
        '''
        all_eye_ind = [
            [172576, 172581, None, None, None, None, None, None, None, None, None, None, 169336, 170000, 169380],
            [153974, 150143, 150124, None, None, None, None, None, None, None, None, None, None, 147576, 150775],
            [152103, 152745, None, None, None, None, None, None, None, None, None, None, 150113, 149516, 151469],
            [116190, 116191, 114265, None, None, None, None, None, None, None, None, None, None, 110410, 111688]]
        my_eye_ind = all_eye_ind[self.heads[0].sequence_id - 1]

        eye_coord = []
        for frame, ind in enumerate(my_eye_ind):
            if ind:
                ind_xy = np.argwhere(self.heads[frame].xy_mesh == ind)
                eye_coord.append(self.heads[frame].xyz[ind_xy][0][0])
        eye_coord = np.array(eye_coord)
        mean_coord = np.mean(eye_coord, axis=0)
        sub_mean = eye_coord - mean_coord
        distances = np.linalg.norm(sub_mean, axis=1)
        print(f"mean distance: {np.mean(distances)}")
        return np.mean(distances)

    def calc_all_sift_keypoints(self, SIFT_contrastThreshold=0.02, SIFT_edgeThreshold=8, SIFT_sigma=0.5):
        '''
        Calculate the keypoints for all the SingleHead object in self.heads.
        '''
        print(SIFT_contrastThreshold,SIFT_edgeThreshold)
        for head in self.heads:
            head.create_keypoints(SIFT_contrastThreshold, SIFT_edgeThreshold, SIFT_sigma)

    def calc_all_sift_transforms(self):
        '''
        calculates SIFT transforms for each of the heads
        :return:
        '''
        for link in self.links:
            # link.reset()
            self.ransac_from_link(link)

    def append(self, head):
        '''
        Append another head to the list of heads
        '''
```

```
        head.reset_positions()
        head.visible = False
        head.center()
        self.heads.append(head)

    def head_id_from_frame_id(self, frame_id):
        '''
        Map from actual frame id to the index of this list of heads.
        '''
        for i, head in enumerate(self.heads):
            if head.frame_id == frame_id:
                return i
        raise ValueError

    def get_next_unpositioned_link(self, sift_transform_method="coverage"):
        '''
        Return the next link to merge based the error of the links.
        '''
        if not sift_transform_method in ["coverage", "matches", "dynamic"]:
            raise ValueError
        best_error = 1
        best_coverage = -1
        best_link_index = None
        any_head_positioned = False
        for head in self.heads:
            if head.visible:
                any_head_positioned = True

        for i, link in enumerate(self.links):
            head_left = self.heads[self.head_id_from_frame_id(link.left)]
            head_right = self.heads[self.head_id_from_frame_id(link.right)]
            # if this is the best error so far and either (one of the 2 heads is visible,
    or none of all the heads is visible)
            possibly_this_link = ((head_left.visible and not head_right.visible) or (
                    (not head_left.visible) and head_right.visible) or not
    any_head_positioned)
            if ((link.err_matches < best_error and sift_transform_method == "matches") or
    (
                    link.pct_coverage > best_coverage and sift_transform_method == "
    coverage")) and possibly_this_link:
                best_error = link.err_matches
                best_coverage = link.pct_coverage
                best_link_index = i
        return best_link_index, best_error

    def reset_all_head_positions(self):
        '''
        Reset the head positions and colors to initial values
        '''
        self.frame_sequence=[]
        for head in self.heads:
            head.reset_positions()
            head.reset_colors()
            head.visible = False


    def reset_all_head_colors(self):
        '''
        Reset the head color to initial values
        '''
        for head in self.heads:
            head.reset_colors()

    def ransac_from_link(self, link):
        '''
```

```
        Computes the matching keypoints and estimate the Procrustes transformation,
        then stores the tranformation result as link attributes.
        '''
        # fetch the frame id.
        head1 = self.heads[self.head_id_from_frame_id(link.right)]
        head2 = self.heads[self.head_id_from_frame_id(link.left)]
        # if the maches has not yet been computed
        if not hasattr(link, "matches"):
            matches = get_matches(head1, head2)
            sample_matches_cvg, pct_coverage, sample_matches_mchs, err_matches, matches = estimate_transform(head1,
                                head2,
                                matches)
            # store the results
            link.add_ransac_results(sample_matches_cvg, pct_coverage, sample_matches_mchs, err_matches, matches)
        return link

    def sift_transform_from_link(self, link, right_to_left, sift_transform_method="matches"):
        '''
        Perform transformation estimation with the selected transformation method
        '''
        # obtain the frame id
        if not sift_transform_method in ["coverage", "matches", "dynamic"]:
            raise ValueError
        head1 = self.heads[self.head_id_from_frame_id(link.right)]
        head2 = self.heads[self.head_id_from_frame_id(link.left)]

        # perform ransac and procrustes to find the best matches for the transformation
        link = self.ransac_from_link(link)
        xyz1, xyz2, matches = get_xyz_from_matches(head1, head2, link.matches)

        # use different set of match points based on the transform_method which can be ["coverage", "matches", "dynamic"].
        if sift_transform_method == "dynamic":
            if isnan(link.err_matches): # if the error value using match point inliers is Nan
                spl_mchs = link.sample_matches_cvg
            elif link.err_matches < 0.01: # if the error value is less that 0.01
                spl_mchs = link.sample_matches_mchs
            else:
                spl_mchs = link.sample_matches_cvg
        # use the coverage error measure
        elif sift_transform_method == "coverage":
            spl_mchs = link.sample_matches_cvg
        # use the matches error measure
        elif sift_transform_method == "matches":
            spl_mchs = link.sample_matches_mchs
        head1.keypoints = xyz1[spl_mchs]
        head2.keypoints = xyz2[spl_mchs]

        # set color for drawings
        head1.keypoints_clr = [1, 0, 0]
        head2.keypoints_clr = [0, 1, 0]

        # perform the transformation
        if right_to_left:
            d, Z, tform12 = procrustes(xyz2[spl_mchs], xyz1[spl_mchs])
            t = tform12['rotation']
            head1.transform(tform12)
        else:
            d, Z, tform21 = procrustes(xyz1[spl_mchs], xyz2[spl_mchs])
```

```
            t = tform21['rotation']
            head2.transform(tform21)
        # generate the images indicating the transformation
        draw_matches(head1, head2, link.matches, spl_mchs)
        return link

    def icp_transform_from_link(self, link, right_to_left):
        '''
        :param link: the Link object between the two heads that are to be ICP-ed
        :param right_to_left: in which direction is the ICP happening? Left to right or rightto left
        :return: link, while correct head has been transformed
        '''
        if right_to_left:
            self.icp_transform(link.left, link.right,
                               max_iterations=1)
        else:
            self.icp_transform(link.right, link.left,
                               max_iterations=1)
        return link

    def refine_transform_from_link(self, link, right_to_left, angle_over_range=False, cartesian_over_range=False,
                                   filter=None):
        '''
        calls the refine_6D fucntions, passing on all parameters, selecting whether head A moves to head B or the other way round
        '''
        if right_to_left:
            filter, score = refine_6D(self, A=link.left, B=link.right, angle_over_range=angle_over_range,
                                      pos_over_range=cartesian_over_range, filter=filter)
        else:
            filter, score = refine_6D(self, A=link.right, B=link.left, angle_over_range=angle_over_range,
                                      pos_over_range=cartesian_over_range, filter=filter)
        return filter, score

    def all_transforms_from_link(self, link, sift_transform_method="matches", icp=False, refine_range=False, refine_local=False):
        '''
        :param link: The link for which all transforms are to be performed
        :return: the link itslef, unmodified.
        Identifies which head is to be transformed ( the one that is not visible)
        Transforms the head that is not visible and makes it visble (headx.visible = True)
        '''
        head1 = self.heads[self.head_id_from_frame_id(link.right)]
        head2 = self.heads[self.head_id_from_frame_id(link.left)]

        # add the head to the frame sequence
        if link.right not in self.frame_sequence:
            self.frame_sequence.append(link.right)
        if link.left not in self.frame_sequence:
            self.frame_sequence.append(link.left)

        # determine the merge direction
        right_to_left = head2.visible and not head1.visible
        if right_to_left:
            self.last_head_id = self.head_id_from_frame_id(link.right)
        else:
            self.last_head_id = self.head_id_from_frame_id(link.left)

        # perform the basic transformation based on the calculated SIFT matches:
        self.sift_transform_from_link(link, right_to_left, sift_transform_method=sift_transform_method)
```

```
307
308         # perform the Refine operation with Zwart algorithm, if it's set True
309         filter = None
310         if refine_range:
311             filter, score = self.refine_transform_from_link(link, right_to_left,
    angle_over_range=True,
312                                                             cartesian_over_range=True,
313                                                             filter=filter)
314             filter, score = self.refine_transform_from_link(link, right_to_left,
    angle_over_range=True,
315                                                             cartesian_over_range=True,
316                                                             filter=filter)
317         else:
318             score = 1000  # just very high
319         if refine_local:
320             last_score = 0
321             before_last_score = 0
322             while score > last_score or score > before_last_score:
323                 before_last_score = last_score
324                 last_score = score
325                 filter, score = self.refine_transform_from_link(link, right_to_left,
    filter=filter)
326             self.icp_transform_from_link(link, right_to_left)
327
328         # perform the ICP transformation, if it's set True
329         if icp:
330             self.icp_transform_from_link(link, right_to_left)
331         head1.visible = True
332         head2.visible = True
333         return link
334
335     def icp_transform(self, frame1, frame2, r=0.05, max_iterations=1):
336         '''
337         param:
338         r (float): sampling rate for head1
339         file_name (string): file name of combined spheres
340         '''
341         # perform one iteration of icp algorithm
342         head1 = self.heads[self.head_id_from_frame_id(frame1)]
343         head2 = self.heads[self.head_id_from_frame_id(frame2)]
344
345         # sample both array to the same size
346         n_sample = int(head1.xyz.shape[0] * r)
347         n_1 = head1.xyz.shape[0]
348         n_2 = head2.xyz.shape[0]
349         sample_1 = np.random.choice(np.arange(n_1), n_sample)
350         sample_2 = np.random.choice(np.arange(n_2), n_sample)
351         T, distance, ite = icp.icp(head1.xyz[sample_1], head2.xyz[sample_2],
    max_iterations=max_iterations)
352
353         # transform head2
354         head2.transform_homo(T)
355         return
356
357     def create_spheres(self, sparcity=1.0, name=None):
358         self.spheres = []
359         for head in self.heads:
360             if head.visible:
361                 if sparcity < 1:
362                     head.sparsify(sparcity)
363                     head.create_vpython_spheres(force_sparce=True)
364                 else:
365                     head.create_vpython_spheres(force_sparce=False)
366                 self.spheres += head.spheres
367         pickle.dump((self.spheres, name), open("pickled_head/head_spheres.p", 'wb'))
```

```
368
369     def save(self):
370         # save keypoints for each SingleHead
371         for head in self.heads:
372             if hasattr(head, 'kp'):
373                 head.kp = [(point.pt, point.size, point.angle, point.response, point.
    octave,
374                             point.class_id) for point in head.kp]
375         # save the information for Links, e.g. machtes
376         for link in self.links:
377             if hasattr(link, 'matches'):
378                 link.matches = [(match.distance, match.imgIdx, match.queryIdx, match.
    trainIdx) for match in
379                                 link.matches]
380
381         pickle.dump(self, open(f"pickled_head/mhead{self.heads[0].sequence_id}.p", 'wb'))
382         for head in self.heads:
383             if hasattr(head, 'kp'):
384                 head.kp = [cv2.KeyPoint(x=point[0][0], y=point[0][1], _size=point[1],
    _angle=point[2],
385                                         _response=point[3], _octave=point[4], _class_id=
    point[5]) for point in head.kp]
386
387         for link in self.links:
388             if hasattr(link, 'matches'):
389                 link.matches = [cv2.DMatch(_distance=match[0], _imgIdx=match[1], _queryIdx
    =match[2], _trainIdx=match[3])
390                                 for match in link.matches]
391
392         print("Saving Completed")
393
394     def create_png_of_spheres(self, sparcity, name, alpha=0):
395         import subprocess
396         self.create_spheres(sparcity=sparcity, name=name)
397         subprocess.call(["python", "gui.py", "save_only", "alpha", f"{int(alpha)}"])
398
399     def show_spheres(self, sparcity, name=None, alpha=0):
400         import subprocess
401         self.create_spheres(sparcity=sparcity, name=name)
402         subprocess.run(["python", "gui.py", "alpha", f"{int(alpha)}"])
403
404     def create_png_series(self, name="", sparcity=0.5):
405         '''
406         Generate the image sequences for the merge set
407         '''
408         self.reset_all_head_colors()
409         for head in self.heads:
410             head.visible = False
411         sequence = self.heads[0].sequence_id
412         print(self.frame_sequence)
413         for i, head_idx in enumerate (self.frame_sequence):
414             head=self.heads[head_idx-1]
415             head.visible = True
416             self.reset_all_head_colors()
417             head.paint([.2, .2, 1])
418
419             if sequence in [2, 4]:
420                 alpha = -(head_idx-self.frame_sequence[0]) * 360 / 15
421             else:
422                 alpha = (head_idx-self.frame_sequence[0]) * 360 / 15
423             if i != 0:
424                 file_name=f"Seq_{sequence}_{i}_{name}".replace("__","_")
425                 self.create_png_of_spheres(sparcity=sparcity, name=file_name, alpha=alpha)
426         self.reset_all_head_colors()
427         file_name = f"Seq_{sequence}_all_{name}".replace("__", "_")
```

```
        self.create_png_of_spheres(sparcity=sparcity, name=file_name, alpha=alpha)

    def create_mesh(self, name):
        '''
        The full head pixel reduction algorithm as described in Building as complete a 3D head as possible
        '''
        def n2v(p):
            return vec(p[0], p[1], p[2])
        def prj(p):
            return vec(p[0], -p[1], -p[2])
        self.objects = []
        O = np.asarray([0, 0, 0.05])

        # set the vertically varying values
        y_range = (-0.35, 0.20)
        o = {'type': "point", 'pos': prj(O + np.asarray([0, 1, 0]) * y_range[0]), 'radius': "0.01",
             'color': vec(0, 1, 0)}
        self.objects.append(o)
        o = {'type': "point", 'pos': prj(O + np.asarray([0, 1, 0]) * y_range[1]), 'radius': "0.01",
             'color': vec(1, 0, 0)}
        self.objects.append(o)

        y_step = 0.003
        big_angle_step = 20
        small_angle_step = 2

        colors = np.vstack([this_head.rgb for this_head in self.heads if this_head.visible])
        points = np.vstack([this_head.xyz for this_head in self.heads if this_head.visible])

        filter_s = np.logical_and(points[:, 1] > y_range[0], points[:, 1] < y_range[1])
        points_s = points[filter_s]
        colors_s = colors[filter_s]
        y_range = np.arange(y_range[0], y_range[1], step=y_step)

        with tqdm(total=y_range.size) as progressbar:
            for y in y_range:
                O_for_y = np.asarray([0, y, 0]) + O
                filter_ss = np.logical_and(points_s[:, 1] > y - 2 * y_step, points_s[:, 1] < y + 2 * y_step)
                points_ss = points_s[filter_ss]
                colors_ss = colors_s[filter_ss]
                vec_from_O_ss = points_ss - O_for_y
                angles_ss = np.mod(np.angle(vec_from_O_ss[:, 0] + 1j * vec_from_O_ss[:, 2]), 2 * np.pi)
                # revolve through all the angles in a plane
                for theta_0 in np.pi * np.arange(0., 360, step=big_angle_step) / 180:
                    filter_sss = np.logical_and(
                        angles_ss > theta_0 - np.pi * small_angle_step / 180,
                        angles_ss < theta_0 + (big_angle_step + small_angle_step) * np.pi / 180)
                    points_sss = points_ss[filter_sss]
                    colors_sss = colors_ss[filter_sss]
                    vec_from_O_sss = vec_from_O_ss[filter_sss]
                    if points_sss.size > 0:
                        for theta_1 in np.pi * np.arange(0, big_angle_step, step=small_angle_step) / 180:
                            theta = theta_0 + theta_1
                            U = np.cos(theta), 0, np.sin(theta)
                            filter1 = np.linalg.norm(vec_from_O_sss - np.inner(U, vec_from_O_sss).reshape((-1, 1)) * U,
```

```
                                                    axis=1) < 0.003
                            filter2 = np.inner(U, vec_from_O_sss) > 0
                            filter = np.logical_and(filter1, filter2)
                            filtered_points = points_sss[filter]
                            if len(filtered_points) > 0:
                                angles_sss = angles_ss[filter_sss]
                                mean = np.mean(filtered_points, axis=0)
                                mean = np.inner(U, mean - O_for_y) * np.asarray(U) +
    O_for_y
                                mean_col = np.mean(colors_sss[filter], axis=0)
                                self.objects.append(
                                    {'type': "point", 'pos': prj(mean), 'radius': "0.003",
     'color': n2v(mean_col)})
                progressbar.set_description(f"Scanning{np.arange(0, big_angle_step, step=
    small_angle_step).size}")
                progressbar.update(1)
        # save the object inside a pickle
        pickle.dump((self.objects,name) , open(f"pickled_head/head_mesh.p", 'wb'))
        print("completed")


    def Method_A(self, sift_transform_method="matches", icp=True, refine_range=True,
    refine_local=True, verbose=True):
        '''
        Build up a full head starting from the right front and revolve in one direction.
        '''
        for link_idx in range(14):  # iterate through the links between heads
            # calculate and perform all transformations for each link:
            if verbose:
                self.links[link_idx].print_short()
            self.all_transforms_from_link(self.links[link_idx], sift_transform_method=
    sift_transform_method, icp=icp, refine_range=refine_range, refine_local=refine_local)

    def Method_B(self, sift_transform_method="matches", icp=True, refine_range=True,
    refine_local=True, verbose=True):
        '''
        Build a full head starting from the back and build up from both directions to the
    front.
        '''
        for link_idx in range(6,14):  # iterate through the links between heads
            if verbose:
                self.links[link_idx].print_short()
            self.all_transforms_from_link(self.links[link_idx], sift_transform_method=
    sift_transform_method, icp=icp, refine_range=refine_range,
                                          refine_local=refine_local)
        for link_idx in range(5, -1, -1):
            if verbose:
                self.links[link_idx].print_short()
            self.all_transforms_from_link(self.links[link_idx], sift_transform_method=
    sift_transform_method, icp=icp, refine_range=refine_range, refine_local=refine_local)

    def Method_C(self, sift_transform_method="matches", icp=True, refine_range=True,
    refine_local=True, verbose=True):
        '''
        Build up a full head by starting from the link with the lowest error and leave the
     one with the worst in the end.
        '''
        only_first_n = 15
        self.reset_all_head_positions()
        link_idx, err = self.get_next_unpositioned_link(sift_transform_method=
    sift_transform_method)
        positioned_head_count = 0
        while (not link_idx is None) and (positioned_head_count < only_first_n or
    only_first_n == -1):# iterate through the links between heads:
            if verbose:
```

```
                self.links[link_idx].print_short()
            self.all_transforms_from_link(self.links[link_idx], sift_transform_method=sift_transform_method, icp=icp,
                                          refine_range=refine_range, refine_local=refine_local)
            link_idx, err = self.get_next_unpositioned_link(sift_transform_method=sift_transform_method)
            positioned_head_count = max(positioned_head_count + 1, 2)


    def create_dataframe(self):
        '''
        Generate the dataFrame for quantitative analysis
        '''
        self.df = pd.DataFrame(
            columns=['Right', 'Left', 'Err', 'Matches', 'Inliers for Matches', 'Inliers for Coverage', 'Coverage'])
        for link in self.links:
            self.df = self.df.append({'Right': int(link.right),
                                      'Left': int(link.left),
                                      'Err': link.err_matches,
                                      'Matches': len(link.matches),
                                      'Inliers for Matches': sum(link.kp_sample_matches),
                                      'Inliers for Coverage': sum(link.inliers_all_points),
                                      'Coverage': link.coverage_all_points},
                                     ignore_index=True)
        return self.df

    def create_df_for_report(self):
        '''
        Populate the dataFrame with data for quantitative analysis
        '''
        self.create_dataframe()
        df = self.df.copy()
        df[["Right", "Left", "Inliers for Matches", "Inliers for Coverage", "Matches"]] = df[
            ["Right", "Left", "Inliers for Matches", "Inliers for Coverage", "Matches"]].astype("int")
        df['Pair'] = df.Left.map(str) + " - " + df.Right.map(str)
        df = df.drop(columns=['Right', 'Left', 'Inliers for Coverage'])
        df['Coverage'] = df['Coverage'] * 100
        df['Coverage'] = df.Coverage.map('{:,.2f}%'.format)
        df['Err'] = df.Err.map('{:,.4f}'.format)
        df = df.rename(columns={"Inliers for Matches": "Inliers"})
        df = df[['Pair', 'Matches', 'Inliers', 'Err', 'Coverage']]
        print(df.to_latex(index=False))
        return df

    def create_res_for_report(self):
        '''
        Add another two information column to the table.
        '''
        res = self.results.copy()
        res = res.rename(columns={'sift_transform_method': 'method'})
        res = res.drop(columns=['refine_range', 'refine_local'])
        print(res.to_latex(index=False))
        return res

    def try_methods(self, refine_range=False, refine_local=False):
        '''
        Automatic generation of experiments result comparing different methods.
        '''
        self.results = pd.DataFrame(
            columns=['method', 'sift_transform_method', 'icp', 'refine_range', '
```

```
    refine_local', 'mean_dist'])

        for method in ['A', 'C']:
            for sift_transform_method in ["coverage", "matches", "dynamic"]:
                # for sift_transform_method in ["matches"]:
                for icp in [True, False]:
                    if method == 'A':
                        # Method A
                        self.Method_A(sift_transform_method=sift_transform_method, icp=icp
    , refine_range=refine_range, refine_local=refine_local, verbose=False)
                    elif method == 'B':
                        # Method B
                        self.Method_B(sift_transform_method=sift_transform_method, icp=icp
    , refine_range=refine_range, refine_local=refine_local, verbose=False)
                    elif method == 'C':
                        # Method C
                        self.Method_C(sift_transform_method=sift_transform_method, icp=icp
    , refine_range=refine_range, refine_local=refine_local, verbose=False)
                    self.results = self.results.append({'method': method,
                                    'sift_transform_method': sift_transform_method,
                                    'icp': icp,
                                    'refine_range': refine_range,
                                    'refine_local': refine_local,
                                    'mean_dist': self.left_eye_deviation()},
                                   ignore_index=True)
        return self.results
```

link.py:

```
import numpy as np
'''
The Link object contains all required variables to store the results of SIFT matches
    between two subsequent frames.

It is defined by:
The IDs of two Single Heads, self.left and self.right
the result of the RANAC operation based on the SIFT matches:
    The matches object, self.macthes: indicating which
    the inliers object, self.inliers: a boolean filter to select the calcualted matches
    from self.matches
    the resulting transformation, tform and the error metric, indicating how good the
    match is.
'''

class Link():
    def __init__(self, left, right):
        '''
        left and right are the frame id of the heads
        '''
        self.left = left
        self.right = right

    def add_matches(self, matches):
        self.matches = matches

    def reset(self):
        if hasattr(self, "sample_matches_cvg"):
            del self.sample_matches_cvg
        if hasattr(self, "pct_coverage"):
            del self.pct_coverage
        if hasattr(self, "sample_matches_mchs"):
            del self.sample_matches_mchs
        if hasattr(self, "err_matches"):
            del self.err_matches
        if hasattr(self, "matches"):
```

```
            del self.matches

    def add_ransac_results(self, sample_matches_cvg, pct_coverage, sample_matches_mchs, err_matches, matches):
        '''
        Store the results from the RANSAC computation to the object.
        '''
        # the sample matches with the best coverage amount all points
        self.sample_matches_cvg = sample_matches_cvg
        self.pct_coverage = pct_coverage
        # the sample matches with the best inlier matches numbers
        self.sample_matches_mchs = sample_matches_mchs
        self.err_matches = err_matches
        self.matches = matches

    def print(self):
        print("left:", self.left, "\tright:", self.right)
        if hasattr(self, "matches"):
            print(f"# Matches {len(self.matches)}")
        if hasattr(self, "sample_matches_mchs"):
            print(f"# Inliers {len(self.sample_matches_mchs)}")
        if hasattr(self, "pct_coverage"):
            print(f"Coverage {100 * self.pct_coverage:.0f}%")
        if hasattr(self, "err_matches"):
            print(f"Err Matches {self.err_matches:.4f}%")

    def print_short(self):
        print(
            f"{self.left}-{self.right}, Count={np.sum(self.sample_matches_mchs)}, Err Matches={self.err_matches:.4f}, Cov={100 * self.pct_coverage:.1f}")
```

SIFT.py:

```
import cv2
import matplotlib.pyplot as plt
from tqdm.autonotebook import tqdm
from procrustes import *
from icp import nearest_neighbor
import os


def get_matches(head1, head2):
    # find K nearest matches ( in terms of descriptors
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(head1.des, head2.des, k=2)
    # unfoled the list
    # matches = [val for sublist in matches for val in sublist]

    # using one nearest neighbor
    good_without_list = [sublist[0] for sublist in matches]

    # use the ratio test
    # good_without_list=[]
    # for m, n in matches:
    #     if m.distance < 0.9 * n.distance:
    #     # good.append([m])
    #         good_without_list.append(m)
    return good_without_list

def remove_height_variation_from_matches(head1, head2, matches):
    max_variation_y_dimension = 15
    matches = [m for m in matches if
               (abs(head1.kp[m.queryIdx].pt[1] - head2.kp[m.trainIdx].pt[1]) < max_variation_y_dimension)]
    return matches
```

```
32
33 def get_xyz_from_matches(head1, head2, matches):
34     '''
35     Get the sets of 3D points of the respective heads, corresponding to the matches.
36     :param head1: a head object.
37     :param head2: a head object.
38     :param matches: a list of DMatch objects.
39     :return: xyz1/2 the sets of 3d points of the respective heads, corresponding to the
       matches.
40     '''
41     matches = remove_height_variation_from_matches(head1, head2, matches)
42
43     pts1 = np.float32([head1.kp[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
44     xy1 = np.round(pts1).astype("int").reshape(-1, 2)
45     xyindex1 = xy1[:, 1] * 640 + xy1[:, 0]
46     indices1 = [np.argwhere(head1.xy_mesh == ind) for ind in xyindex1]
47     filter1 = [len(ind) > 0 for ind in indices1]
48
49     pts2 = np.float32([head2.kp[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
50     xy2 = np.round(pts2).astype("int").reshape(-1, 2)
51     xyindex2 = xy2[:, 1] * 640 + xy2[:, 0]
52     indices2 = [np.argwhere(head2.xy_mesh == ind) for ind in xyindex2]
53     filter2 = [len(ind) > 0 for ind in indices2]
54
55     filter = np.asarray(filter1) & np.asarray(filter2)
56     indices1 = np.asarray(indices1)[filter]
57     indices2 = np.asarray(indices2)[filter]
58     xyz1 = np.asarray([head1.xyz[ind[0][0]] for ind in indices1])
59     xyz2 = np.asarray([head2.xyz[ind[0][0]] for ind in indices2])
60     return xyz1, xyz2, matches
61
62
63 def estimate_transform(head1, head2, matches):
64     '''
65     Estimate the transform from a set of match objects and two head objects using the
       RANSAC
66     technique and the Procrutes algorithm.
67     param head1 (SingleHead): one head as the anchor.
68     param head2 (SingleHead): another head to transform from.
69     param matches (list(DMatch)): a list of match points from two heads.
70     '''
71     kp_xyz1, kp_xyz2, matches = get_xyz_from_matches(head1, head2, matches)
72
73     # RANSAC parameters
74     max_dist_matches = 0.007
75     max_dist_coverage = 0.01
76
77     no_iterations = 5000 # number of ransac iterations.
78     no_iterations_all_points = 200 # number of ransac that does coverage calculation.
79     min_num_matches = 6 # minimum number of matches for procrustes
80     sample_thresh = 0.6 # threshold for sampling match matches
81     best_count_coverage = 0 # record the best number of point covered by a transformation
82     best_count_matches = 0 # record the best number of maches covered by a tranformation
83     best_pct_coverage = -1 # the coverage value that we want to maximize
84     best_err_matches = 1000  # the inlier matches that we want to maximize
85     best_sample_matches_cvg = []
86     best_sample_matches_mchs = []
87
88     # store the points from both frames for coverage calculation
89     l = head2.xyz.shape[0]
90     filter2 = np.random.random((l)) < 0.1
91     xyz1 = head1.xyz
92     xyz2 = head2.xyz[filter2]
93     temp_best_tform = None
94     with tqdm(total=no_iterations) as progressbar:
```

```
        # total RANSAC iterations
        for j in range(no_iterations):
            kp_sample = np.random.rand(kp_xyz2.shape[0]) > sample_thresh  # get random sample set
            progressbar.update(1)  # tqdm
            temp_best_count = -1
            if np.sum(kp_sample) >= min_num_matches:  # enough points to unambiguously define transformation?
                _, _, tform = procrustes(kp_xyz1[kp_sample], kp_xyz2[kp_sample])
                R, c, t = tform['rotation'], tform['scale'], tform['translation']
                dist = np.linalg.norm(kp_xyz2.dot(c * R) + t - kp_xyz1, axis=1)
                inliers = dist < max_dist_matches

                # if there are more inliers for this transformation.
                if (np.sum(inliers) > best_count_matches):
                    best_sample_matches_mchs = kp_sample.copy()
                    best_count_matches = np.sum(inliers)
                    err = np.sqrt(np.var(dist) / (np.sum(kp_sample) - min_num_matches))
                    best_err_matches = err
                    # update the progress bar
                    progressbar.set_description(
                        f"Head {head1.frame_id} & {head2.frame_id} :cnt:{best_count_matches:.0f} err:{best_err_matches:.4f} cov:{100 * best_pct_coverage :.2f}%")

                # number of ransac that does coverage calculation
                if j < no_iterations_all_points:
                    R, c, t = tform['rotation'], tform['scale'], tform['translation']
                    xyz2_trans = xyz2.dot(c * R) + t
                    distances, indices = nearest_neighbor(xyz2_trans, xyz1)
                    count_all_points = np.sum(distances < max_dist_coverage)
                    # see if the new transformation has a better coverage
                    if count_all_points > best_count_coverage:
                        best_count_coverage = count_all_points
                        best_pct_coverage = best_count_coverage / xyz2.shape[0]
                        best_sample_matches_cvg = kp_sample.copy()
                        progressbar.set_description(
                            f"Head {head1.frame_id} & {head2.frame_id} :cnt:{best_count_matches:.0f} err:{best_err_matches:.4f} cov:{100 * best_pct_coverage :.2f}%")

    return best_sample_matches_cvg, best_pct_coverage, best_sample_matches_mchs, best_err_matches, matches

def draw_matches(head1, head2, matches, inliers):
    '''
    Generate the images with the match points between two frames
    '''
    images_dir="images"
    if not os.path.isdir(images_dir):
        os.mkdir(images_dir)
    # set the background color
    head1.background_color=np.array([0,0,0.99])
    head2.background_color=np.array([0,0,0.99])
    img1 = cv2.cvtColor((head1.get_filtered_image() * 256).astype("uint8"), cv2.COLOR_BGR2RGB)
    img2 = cv2.cvtColor((head2.get_filtered_image() * 256).astype("uint8"), cv2.COLOR_BGR2RGB)
    # parameters for drawing
    draw_params = dict(matchColor=(0, 255, 0),
                       singlePointColor=(0, 0, 255),
                       matchesMask=inliers.ravel().tolist(),
                       flags=0)
    # generat the images along with the matches indication
    img3 = cv2.drawMatches(img1, head1.kp, img2, head2.kp, matches1to2=matches,
```

```
                              outImg=None, **draw_params)
    # label parameters
    font = cv2.FONT_HERSHEY_SIMPLEX
    bottomLeftCornerOfText = (0, 40)
    fontScale = 1
    fontColor = (255, 255, 255)
    lineType = 2
    # add text label onto the image
    cv2.putText(img3, f"Sequence {head1.sequence_id}, frames {head1.frame_id} & {head2.
    frame_id}",
                bottomLeftCornerOfText,
                font,
                fontScale,
                fontColor,
                lineType)
    # save the image
    file_name=f"Seq_{head1.sequence_id}_frames_SIFT_{head1.frame_id}_{head2.frame_id}.png"
    full_path= os.path.join(images_dir, file_name)
    cv2.imwrite(full_path, img3)


def get_descriptors(img, SIFT_contrastThreshold, SIFT_edgeThreshold, SIFT_sigma):
    '''
    Get a set of SIFT parameters given a color image in an array and three SIFT parameters
    .
    param:
    img (array): colored image
    The others are all SIFT parameters
    '''
    # scale pixel values the image
    img = img * 256
    img = cv2.cvtColor(img.astype("uint8"), cv2.COLOR_BGR2RGB)
    sift = cv2.xfeatures2d.SIFT_create(contrastThreshold=SIFT_contrastThreshold,
    edgeThreshold=SIFT_edgeThreshold,
                                       sigma=SIFT_sigma)
    kp, des = sift.detectAndCompute(img, None)
    return kp, des
```

icp.py:

```
# From : https://github.com/ClayFlannigan/icp
import numpy as np
from sklearn.neighbors import NearestNeighbors


def best_fit_transform(A, B):
    '''
    Calculates the least-squares best-fit transform that maps corresponding points A to B
    in m spatial dimensions
    Input:
      A: Nxm numpy array of corresponding points
      B: Nxm numpy array of corresponding points
    Returns:
      T: (m+1)x(m+1) homogeneous transformation matrix that maps A on to B
      R: mxm rotation matrix
      t: mx1 translation vector
    '''

    assert A.shape == B.shape

    # get number of dimensions
    m = A.shape[1]

    # translate points to their centroids
    centroid_A = np.mean(A, axis=0)
```

```
    centroid_B = np.mean(B, axis=0)
    AA = A - centroid_A
    BB = B - centroid_B

    # rotation matrix
    H = np.dot(AA.T, BB)
    U, S, Vt = np.linalg.svd(H)
    R = np.dot(Vt.T, U.T)

    # special reflection case
    if np.linalg.det(R) < 0:
       Vt[m-1,:] *= -1
       R = np.dot(Vt.T, U.T)

    # translation
    t = centroid_B.T - np.dot(R,centroid_A.T)

    # homogeneous transformation
    T = np.identity(m+1)
    T[:m, :m] = R
    T[:m, m] = t

    return T, R, t


def nearest_neighbor(src, dst):
    '''
    Find the nearest (Euclidean) neighbor in dst for each point in src
    Input:
        src: Nxm array of points
        dst: Nxm array of points
    Output:
        distances: Euclidean distances of the nearest neighbor
        indices: dst indices of the nearest neighbor
    '''

    # assert src.shape == dst.shape

    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True)
    return distances.ravel(), indices.ravel()


def icp(A, B, init_pose=None, max_iterations=1, tolerance=0.0001, distance_threshold=0.04)
    :
    '''
    The Iterative Closest Point method: finds best-fit transform that maps points A on to
    points B
    Input:
        A: Nxm numpy array of source mD points
        B: Nxm numpy array of destination mD point
        init_pose: (m+1)x(m+1) homogeneous transformation
        max_iterations: exit algorithm after max_iterations
        tolerance: convergence criteria
    Output:
        T: final homogeneous transformation that maps A on to B
        distances: Euclidean distances (errors) of the nearest neighbor
        i: number of iterations to converge
    '''
    verbose = False
    if verbose:
        print("start icp")
    assert A.shape == B.shape

```

```
    # get number of dimensions
    m = A.shape[1]

    # make points homogeneous, copy them to maintain the originals
    src = np.ones((m + 1, A.shape[0]))
    dst = np.ones((m + 1, B.shape[0]))
    src[:m, :] = np.copy(A.T)
    dst[:m, :] = np.copy(B.T)

    # apply the initial pose estimation
    if init_pose is not None:
        src = np.dot(init_pose, src)

    prev_error = -1

    for i in range(max_iterations):
        # find the nearest neighbors between the current source and destination points
        distances, indices = nearest_neighbor(src[:m, :].T, dst[:m, :].T)
        filter = distances < 0.02
        if verbose:
            print(distances.shape, indices.shape, src.shape)
            print("step: ", i, "before: ", np.mean(distances))
        # compute the transformation between the current source and nearest destination
    points
        T, _, _ = best_fit_transform(src[:m, :].T[filter], dst[:m, indices].T[filter])
        if verbose:
            print("found the best fit transform", i)
        # update the current source
        src = np.dot(T, src)

        # check error
        mean_error = np.mean(distances)
        if np.abs(prev_error - mean_error) < tolerance and prev_error > 0:
            break
        prev_error = mean_error

    # calculate final transformation
    T,_,_ = best_fit_transform(A, src[:m,:].T)

    return T, distances, i
```

refine.py:

```
from icp import nearest_neighbor
import numpy as np
from tqdm.autonotebook import tqdm


def calc_R(phi, axis):
    '''
    :param phi: angle, in degrees
    :param axis: axis for rotation 3D array
    :return:  rotation matrix, as function of angle and selected axis.
    '''
    phi_rad = phi * np.pi / 180
    if (np.array(axis) == np.array([1, 0, 0])).all():
        return np.array([[1, 0, 0], [0, np.cos(phi_rad), -np.sin(phi_rad)], [0, np.sin(
    phi_rad), np.cos(phi_rad)]])
    elif (np.array(axis) == np.array([0, 1, 0])).all():
        return np.array([[np.cos(phi_rad), 0, np.sin(phi_rad)], [0, 1, 0], [- np.sin(
    phi_rad), 0, np.cos(phi_rad)]])
    elif (np.array(axis) == np.array([0, 0, 1])).all():
        return np.array([[np.cos(phi_rad), -np.sin(phi_rad), 0], [np.sin(phi_rad), np.cos(
    phi_rad), 0], [0, 0, 1]])
    raise ValueError("invalid axis")
```

```
20
21
22 def refine_over_range(mhead, A, B, range, step, axis, filter, angle, max_distance=0.005):
23     '''
24     :param mhead: Multi Head Object
25     :param A: Index of Head A
26     :param B: Index of Head B
27     :param range: vary the axis over this range
28     :param step: step size
29     :param axis: axis (can be carthsesian or angles)
30     :param filter: subset of  points of Head B that  are actually used
31     :param angle: angular(True) or  carthesian(False)
32     :param max_distance: max distance between points on head B to head A
33     :return: (optimum translation or rotation,fraction of point in range, applied filter )
34     '''
35
36     max_distance_between_points = max_distance
37     best_count = -1
38
39     head1 = mhead.heads[mhead.head_id_from_frame_id(A)]
40     head2 = mhead.heads[mhead.head_id_from_frame_id(B)]
41     l = head2.xyz.shape[0]
42
43     if filter is None:
44         filter2 = np.random.random((l)) < 0.1  # only consider 10% of the points, to
       reduce computational complexity
45     else:
46         filter2 = filter  # if filter is provided, re-use existing filter, to keep
       optimization consistent
47
48     for value in np.arange(-range, range, step):
49         if angle:
50             CoG = head2.xyz.mean(axis=0)
51             R = calc_R(value, axis)
52             head2.xyz = np.dot(head2.xyz - CoG, R) + CoG
53         else:
54             head2.xyz = head2.xyz + value
55         xyz1 = head1.xyz
56         xyz2 = head2.xyz[filter2]
57         distances, indices = nearest_neighbor(xyz2, xyz1)
58         count = np.sum(distances < max_distance_between_points)
59         if count > best_count:
60             best_value = value
61             best_count = count
62         if angle:
63             R = calc_R(-value, axis)
64             head2.xyz = np.dot(head2.xyz - CoG, R) + CoG
65         else:
66             head2.xyz = head2.xyz - value
67     if angle:
68         R = calc_R(best_value, axis)
69         head2.xyz = np.dot(head2.xyz - CoG, R) + CoG
70     else:
71         head2.xyz = head2.xyz + best_value
72     return best_value * np.array(axis), best_count / np.size(
73         xyz2), filter2  # move the head to the best position for the given
74
75
76 def refine_local(mhead, A, B, step, axis, angle, filter=None, max_distance=0.01):
77     '''
78     :param mhead: Multi Head Object
79     :param A: Index of Head A
80     :param B: Index of Head B
81     :param step: step size
82     :param axis: axis (can be carthsesian or angles)
```

```
    :param angle: angular(True) or  carthesian(False)
    :param filter: subset of  points of Head B that  are actually used
    :param max_distance: max distance between points on head B to head A
    :return: (optimum translation or rotation,fraction of point in range, applied filter )

    algorithm looks for local optimum: it scans over the given axis (angular or carthesian
    ) and look for local optimum by either increasing or decreasing the middle position,
    tracking values at the boundaries
    '''
    max_distance_between_points = max_distance
    head1 = mhead.heads[mhead.head_id_from_frame_id(A)]
    head2 = mhead.heads[mhead.head_id_from_frame_id(B)]
    l = head2.xyz.shape[0]
    if filter is None:
        filter2 = np.random.random((l)) < 0.1  # only consider 10% of the points, to
    reduce computational complexity
    else:
        filter2 = filter  # if filter is provided, re-use existing filter, to keep
    optimization consistent
    lower_count = 0  # track at thee positions lower, middle and higher
    higher_count = 0
    middle_count = 0
    value = 0
    init = 3
    bounce_count = 0
    move_down = True
    while lower_count >= middle_count or higher_count >= middle_count or init > 0:
        if lower_count > middle_count or init > 0:
            s = -step
            move_down = True
        elif higher_count > middle_count:
            s = step
            move_down = False
        elif lower_count == middle_count:
            s = -step
            if move_down == False:
                bounce_count += 1
            move_down = True
        elif higher_count == middle_count:
            s = step
            move_down = False
        else:
            break
        if bounce_count == 2:
            break
        init -= 1
        value = value + s

        # apply transformation, looking either at higher or lower position. aS we come
    from the middle position, we need to take a double step (2 * s)
        if angle:
            CoG = head2.xyz.mean(axis=0)  # calculate Center of Gravity, i.e. the middle
    of the object
            R = calc_R(2 * s, axis)
            head2.xyz = np.dot(head2.xyz - CoG, R) + CoG  # turn with respect to center of
     gravity
        else:
            t = np.array(axis) * 2 * s
            head2.xyz = head2.xyz + t
        xyz1 = head1.xyz
        xyz2 = head2.xyz[filter2]
        # calculate distances
        distances, indices = nearest_neighbor(xyz2, xyz1)
        # count how many distance are within range
        count = np.sum(distances < max_distance_between_points)
```

```
        # move back to middle position
        if angle:

            R = calc_R(-s, axis)
            head2.xyz = np.dot(head2.xyz - CoG, R) + CoG
        else:
            t = np.array(axis) * -1 * s
            head2.xyz = head2.xyz + t
        if move_down:
            higher_count = middle_count
            middle_count = lower_count
            lower_count = count
        elif not move_down:
            lower_count = middle_count
            middle_count = higher_count
            higher_count = count
    return value * np.array(axis), middle_count / np.size(xyz2), filter2

def refine_6D(mhead, A, B, angle_over_range=False, pos_over_range=False, filter=None):
    '''
    :param mhead: Multi Head Object
    :param A: Index of Head A
    :param B: Index of Head B
    :param angle_over_range: Whether we refine the angle locally or over a range
    :param pos_over_range: Whether we refine the position locally or over a range
    :param filter: subset of  points of Head B that  are actually used
    :return: the filter and the score (fraction of points of Head B that are within wide/
    fine max distance of a point on head A) after 6D refinement

    Transforms Head B over 3 axes and 3 angles to maximise number of overlapping points
    with Head A

    '''
    t = np.zeros(3)

    # hyperparameters:
    # for range scanning
    phi_range = 25  # degrees
    phi_raster = 1  # degrees
    s_range = 0.1  # meters
    s_raster = 0.02  # meters
    max_distance_range = 0.02  # maximum distance between points during range scanning
    # for local  scanning:
    s_wide = 0.005  # wide steps for the cartesian axes
    s_fine = 0.002  # fine steps for the cartesian axes
    phi_wide = 2  # wide steps for the angle
    phi_fine = 0.4  # fine steps for the angle
    max_distance_local = 0.01  # maximum distance between points during local scanning

    if angle_over_range:  # phase 1, scanning angle over a range
        with tqdm(total=3) as progressbar:
            for axis in [[0, 1, 0], [1, 0, 0], [0, 0, 1]]:
                x, score, filter = refine_over_range(mhead, A, B, range=phi_range, step=
    phi_raster, axis=axis,
                                                     filter=filter,
                                                     angle=True, max_distance=
    max_distance_range)
                t = t + x
                progressbar.set_description(
                    f"scan {A} to {B} phi:{t[0]: 3.1f},{t[1]: 3.1f},{t[2]: 3.1f} Scr={100
    * score:2.2f}%")
                progressbar.update(1)
    else:  # phase 2, iterative local scanning of the angle
        with tqdm(total=3) as progressbar:
            for s in [phi_wide, phi_fine]:
```

```
                for axis in [[0, 1, 0], [1, 0, 0], [0, 0, 1]]:
                    x, score, filter = refine_local(mhead, A, B, s, axis, angle=True, filter=filter,
                                                    max_distance=max_distance_local)
                    t = t + x
                    progressbar.set_description(
                        f"refine {A} to {B} phi:{t[0]: 3.1f},{t[1]: 3.1f},{t[2]: 3.1f} Scr={100 * score:2.2f}%")
                    progressbar.update(1)
    t = np.zeros(3)

    if pos_over_range:  # phase 1, scanning position over a range
        with tqdm(total=3) as progressbar:
            for axis in [[1, 0, 0], [0, 1, 0], [0, 0, 1]]:
                x, score, filter = refine_over_range(mhead, A, B, range=s_range, step=s_raster, axis=axis,
                                                     filter=filter,
                                                     angle=False, max_distance=max_distance_range)
                t = t + 1000 * x
                progressbar.set_description(
                    f"scan {A} to {B} t  :{t[0]: 3.1f},{t[1]: 3.1f},{t[2]: 3.1f} Scr={100 * score:2.2f}%")
                progressbar.update(1)
    else:  # phase 2, scanning position locally
        with tqdm(total=6) as progressbar:
            for s in [s_wide, s_fine]:
                for axis in [[1, 0, 0], [0, 1, 0], [0, 0, 1]]:
                    x, score, filter = refine_local(mhead, A, B, s, axis, angle=False, filter=filter,
                                                    max_distance=max_distance_local)
                    t = t + 1000 * x
                    progressbar.set_description(
                        f"refine {A} to {B} t  :{t[0]: 3.1f},{t[1]: 3.1f},{t[2]: 3.1f} Scr={100 * score:2.2f}%")
                    progressbar.update(1)

    return filter, score
```

gui.py:

```
'''
GUI for rendering 3D head objects
'''

import sys

from vpython import *
from vpython.no_notebook import stop_server
import pickle
import time
import numpy as np

def Savebutton():
    # take a screen shot of the 3d object
    global name
    if name is None:
        file_name = f"mesh.png"
    else:
        file_name = f"{name}.png"
    scene.capture(file_name)


def Readbutton():
    # read 3-d object from the file
```

```
    global name
    if 'l' in globals():
        global l
        for i in range(len(l)):
            o = l.pop()
            o.visible = False
            del o

    if 'c' in globals():
        global c
        c.visible = False
        del c

    data_file = 'pickled_head/head_spheres.p'
    try:
        with open(data_file, 'rb') as file_object:
            raw_data = file_object.read()
        (spheres,name) = pickle.loads(raw_data)
    except:
        raise FileExistsError(f'{data_file} could not be found, create {data_file} by
    using .save() first ')

    c = points(pos=spheres, size_units='world')
    print(len(spheres))

def ReadMeshbutton():
    # read the other 3-D object which should correspond to the mesh of the head
    global name
    if 'c' in globals():
        global c
        c.visible = False
        del c

    if 'l' in globals():

        for i in range(len(l)):
            o = l.pop()
            o.visible = False
            del o

    data_file = 'pickled_head/head_mesh.p'
    try:
        with open(data_file, 'rb') as file_object:
            raw_data = file_object.read()
        (mesh,name) = pickle.loads(raw_data)
    except:
        raise FileExistsError(f'{data_file} could not be found, create {data_file} by
    using .save() first ')

    for o in mesh:
        if o['type'] == "pyr":
            p = pyramid(pos=o['pos'])
            l.append(p)
        elif o['type'] == "point":
            col = o['color']
            r = float(o['radius'])
            p = sphere(pos=o['pos'], color=col, radius=r)
            l.append(p)

# displaying the 3-d object
args = sys.argv

scene.width = scene.height = 800
scene.background = color.white
scene.range = 0.5
```

```
88
89 button(text='Save', bind=Savebutton)
90 button(text='Read', bind=Readbutton)
91 button(text='Mesh', bind=ReadMeshbutton)
92
93 scene.append_to_caption("""<br>Right button drag or Ctrl-drag to rotate "camera" to view
       scene.
94 Middle button or Alt-drag to drag up or down to zoom in or out.
95   On a two-button mouse, middle is left + right.
96 Touch screen: pinch/extend to zoom, swipe or two-finger rotate.""")
97
98 data_file = 'pickled_head/head_spheres.p'
99
100 name = None
101
102 try:
103     with open(data_file, 'rb') as file_object:
104         raw_data = file_object.read()
105 except:
106     raise FileExistsError(f'{data_file} could not be found, create {data_file} by using .
       save() first ')
107
108 (spheres, name) = pickle.loads(raw_data)
109 print(len(spheres))
110 c = points(pos=spheres, size_units='world')
111 rate(20)
112 l = []
113
114 print(args)
115
116 if "alpha" in args:
117     alpha = int(args[args.index("alpha") + 1])
118
119 else:
120     alpha = 0
121
122 scene.light = [
123     distant_light(direction=vector(np.sin(-(45 + alpha) * np.pi / 180), 0.3, np.cos(-(45 +
        alpha) * np.pi / 180)),
124                   color=color.gray(0.3)),
125     distant_light(direction=vector(np.sin(-(+45 + alpha) * np.pi / 180), 0.3, np.cos(-(-45
        + alpha) * np.pi / 180)),
126                   color=color.white)]
127
128 print(alpha)
129 scene.forward = vec(np.sin(alpha * np.pi / 180), 0, -np.cos(alpha * np.pi / 180))
130
131 if "save_only" in args:
132     time.sleep(len(spheres) // 10000)
133     Savebutton()
134     time.sleep(len(spheres) // 30000)
135     stop_server()
```