3DP HW2

Juri Farruku

ID: 2157856

1 Task 1

For the second assignment of the course we had to complete 7 main tasks of a Structure from Motion pipeline made by 2 main programs. The first program, which had the task to extract keypoints between images of the given dataset and then to set the matches between them, had 2 of the 7 tasks.
In the first one, the `extractFeatures` method was completed using a SIFT detector which, through the `detectAndCompute` function, extracts keypoints and descriptors of the i image and stores them respectively in `features_[i]` and `descriptors_[i]`. Through the following `for` the keypoint colors are stored in the `feat_colors_[i]`. This is done by accessing the image pixel in the keypoint coordinates, which are stored in `features_[i][j].pt` where the variable j is used to iterate on the array of keypoints.

2 Task 2

To complete the second task in the `exhaustiveMatching` function it was used a BruteForce matcher. Indeed using the function `match` of the matcher on the descriptors of two different images, which in this case are identified by index i and j, i was able to store the matches between the aforementioned images in a suitable vector. In order to extract the **Essential** and **Homography** matrix, the points of the matches were stored in 2 vectors of `cv::Point2f`, pt1 and pt2, where, thanks to trainIdx and queryIdx were populated.
Once these two vectors were obtained, the two matrices were calculated through `findEssentialMat` and `findHomography`, using the given parameters. For each geometric model the inliers were stored in 2 cv::Mat, namely `inliers_essential` and `inliers_homography`. These 2 matrices were used in order to select which was the best geometric model for the 2 images currently being processed, counting the number of inliers of each model. Once the best model was chosen, its inliers matrix was saved in the matrix `Best_Matrix` and then, if the latter had a number of inliers above the minimum required (5 in this case), the 2 processed images had their matches set thanks to the given `setMatches` function.

3 Task 3

The following four tasks were part of the second program named `basic_sfm`, which implemented the SfM pipeline.
In task 3 the matches between the images were already given and as first step the 2 matrices H and E were extracted as said before, with different parameters. The first check to see if the occurring reconstruction could be useful was checking that the number of inliers of E was greater than the number of inliers of H, otherwise the function should return false and restart the reconstruction. Once this phase is passed, thanks to the `recoverPose` function the program tries to obtain the rotation matrix and translation vector, stored in `R` and `t` from the essential matrix E calculated before and checks if the motion of the camera in this case is mainly sideward, accessing the 3 elements of `t` and checking whether the main component is either the x or y one. If this is the case, then the recovered matrices are stored in suitable variables, otherwise the function return false.

4 Task 4

In this task the main objective was to triangulate some given points given 2 different cameras parameters. Since the first 3 camera data structure parameters were the axis-angle representation of its rotation matrix and the following 3 its translation vector, for each camera these parameters were stored as follows:

```
cv::Mat cam0_axis_angle_repr = (cv::Mat_<double>(3,1)
<< cam0_data[0], cam0_data[1], cam0_data[2]);
cv::Mat cam0_translation_vector = (cv::Mat_<double>(3,1)
<< cam0_data[3], cam0_data[4], cam0_data[5]);

cv::Mat cam1_axis_angle_repr = (cv::Mat_<double>(3,1)
<< cam1_data[0], cam1_data[1], cam1_data[2]);
cv::Mat cam1_translation_vector = (cv::Mat_<double>(3,1)
<< cam1_data[3], cam1_data[4], cam1_data[5]);
```

Then the rotation matrices of each camera were obtatin using the `Rodrigues` function of openCV, which takes an axis-angle-represation rotation and gives in output the 3x3 one, or viceversa.

These 2 matrices were then concatenated with their respective translaction vector, obtaining the projection matrix of camera 0 and 1, which were store in `cam0_projection_matrix` and `cam1_projection_matrix`

The 2d point observations were retrieved accessing the `observation` vector, using the vector of maps called `cam_observation` that given the camera index and the 3d point index gives the corresponding 2d point index.

Finally, through the `triangulatePoints` function, the 3D point in homogeneous coordinates was obtained. It was then divided for its fourth component in order to get the non homogeneous coordinates. Here an if which checked the

dimension of the fourth coordinate was placed in order to not divide for very small values.

In the end the cheirality constraint for both cameras was checked, multiplying the point for each of the camera rotation matrix and adding to it the corresponding translaction vector and finally checking whether the Z coordinate was greater than 0

5 Task 5

In task 5 a struct used later to minimize the reprojection error was created through the ceres Solver. This struct uses the 2 observed points given as input, the 6 camera parameters (3 for the axis-angle representation and 3 for the translaction vector) and 3 3D-Point parameters. The first thing done here by the function is to project the 3D point in image plane thanks to the `ceres::AngleAxisRotatePoint` which gives the homogeneous image plane point. Then the non homogeneous point is obtained dividing for the third component of the point and the residuals are computed, in order to check the reprojection error.

6 Task 6

The aforementioned struct is mainly used in task 6, where for each registered observation a block in the ceres Solver is added. First of all the camera and points parameters are stored in 2 suitable variables as follows:

```
double* camera_data = parameters_.data() +
cam_pose_index_[i_obs] * camera_block_size_
double* point_data = parameters_.data()
+ num_cam_poses_ * camera_block_size_ + point_index_[i_obs] * point_block_size_;
```

where `camera_block_size` contains the constant number associated with each camera (same for `point_block_size`) and `num_cam_poses_` contains the number of camera poses

Then the cost function and the loss function are created using respectively `ReprojectionError::Create(observations_[i_obs * 2], observations_[(i_obs * 2) + 1])` and `new ceres::CauchyLoss(2.0 * max_reproj_err_)`. Once the block is created, it is added to the problem using the function `AddResidualBlock` given by ceres.

7 Task 7

The last task required to check whether the reconstruction had diverged in the end, checking the camera's positions and the point's positions with respect to the coordinate system adopted. In order to achieve the aforementioned task 2

variables where instantiated : `flag` and `max_point_dist`, the first used to indicate if the reconstruction had diverged and the latter used to set the maximum distance acceptable.

These 2 operations are followed by 2 different for cycles, one for points and one for cameras and since their structure is similar, only one of them will be discussed in the report. The for cycle iterates through all points and through an if condition checks whether they were used for reconstruction or not. In the first case the distance of the point from the origin of the coordinate system is computed and then confronted with `max_point_dist`, setting the flag to true if necessary.

8 Qualitative results

This pipeline was tested on 2 different datasets given by the professor and 4 made through and IPhone16. Since the `feature_matcher` program takes in input the camera parameters, it was necessary to obtains the phone's matrix parameters through camera calibration.

8.1 Given Datasets

On Dataset 1, which had images of some statues, the reconstruction can be seen through the screenshot 1 of the `.ply` file created by the program.

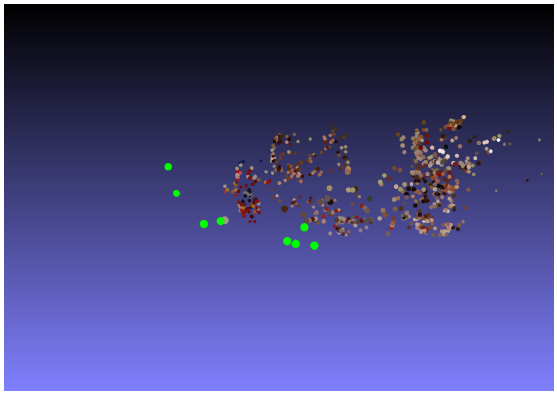

Figure 1: Snapshot of .ply file of statues obtained from the program.

Instead the results obtained on Dataset 2 can be observed in image 2 and image 3.

As noticeable through the images seen, the quality of the second reconstruction is higher, probably since the second dataset was composed by 7 more images than the first one. The program was indeed able to obtain more noticeable keypoints with more images

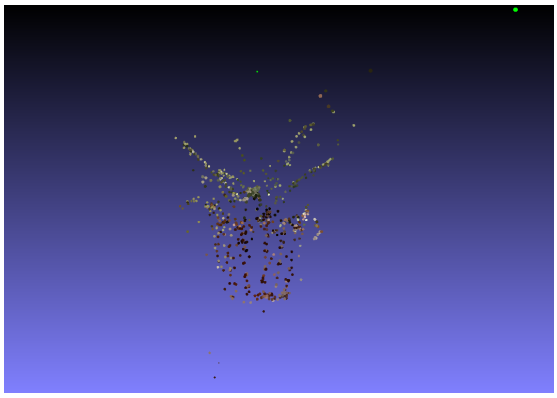

Figure 2: Snapshot of .ply file of a plant obtained from the program.

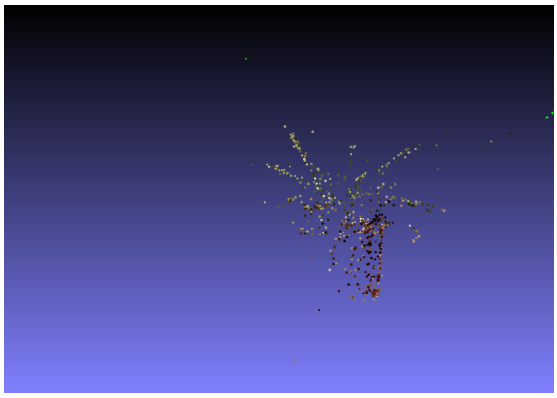

Figure 3: Snapshot of .ply file of a plant obtained from the program.

8.2 Datasets taken with phone

With IPhone16, 4 different datasets were taken.
The results of these datasets can be seen in 4 for the first one, in 5 the second one, in 6 the third one and the last one in 7

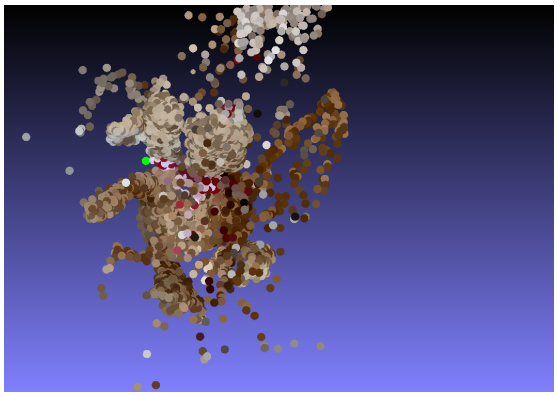

Figure 4: Snapshot of .ply file of a deer puppet obtained from the program.

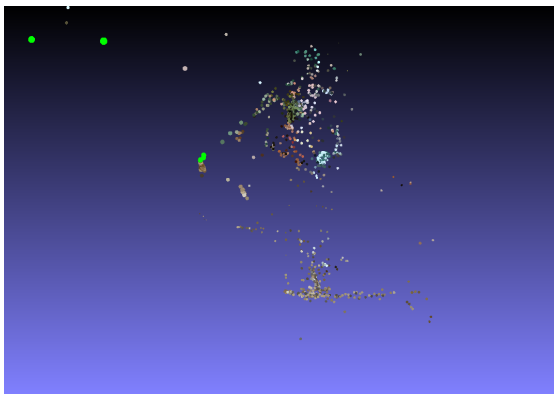

Figure 5: Snapshot of .ply file of a bottle of tea obtained from the program.

Through these screenshots several differences can be spot. For example in the image obtained from the puppet we have a high density of points in the body, which was in high contrast with the background thanks to its color, meanwhile its antlers, being more similar to the background, have significantly main points. The same applies for the bottle of tea: in fact the label, which has a very bright pink color, is well detected meanwhile the plastic part, since it has no peculiar color has less points, on the contrary the floor has more point since it is not a smooth unicolor surface. The pen holder has many different points detected thanks to its color but the probably best obtained image is the can's

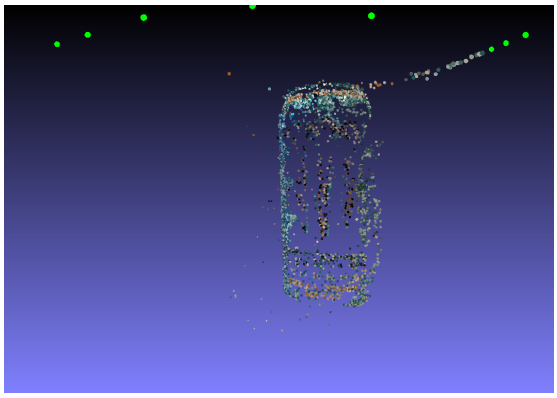

Figure 6: Snapshot of .ply file of a can obtained from the program.

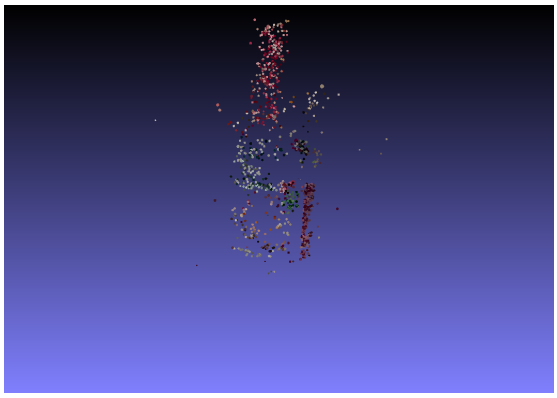

Figure 7: Snapshot of .ply file of a pen holder obtained from the program.

one. Since the can itself has a high number of details and very brights colors, its reconstruction is of high quality, making the object easy recognizable.