2021/2022 IACV Project
F10. Template matching in depth images

Alessandro Polidori, Nicolò Stagnoli

February 13, 2022

1 Project Goal

Template matching is a very practical technique for finding all the occurrences of a known target in a given query image. Deep-learning techniques are not very flexible, requiring re-training for new templates. Traditional computer vision techniques (e.g. SIFT + RANSAC) are more practical but require an underlying model (2D-Homography) describing the transformation between the template and its occurrences in the query image. When there are several occurrences of the same template in the scene, template matching becomes a robust multi-model fitting problem, which requires to disambiguate among multiple spurious matches. The goal of this project is to expand a template detection pipeline based on images only, to RGB-D data.

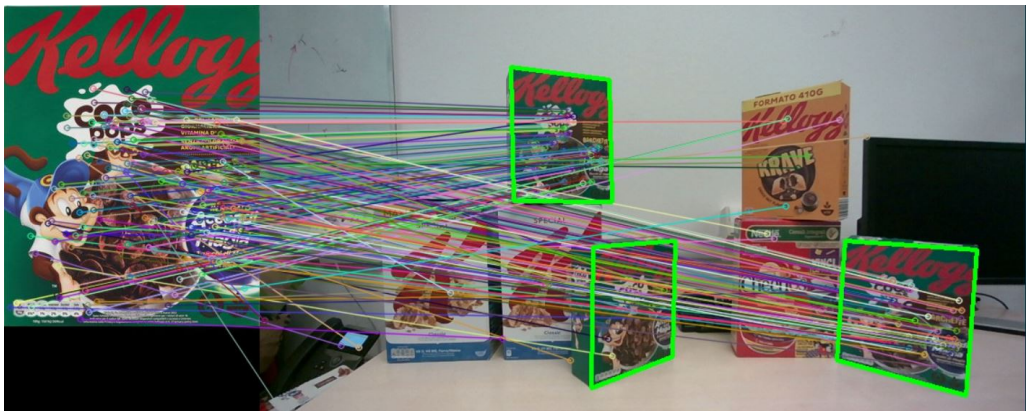

Figure 1: Template matching

2 State of the Art

In order to perform template matching, the first step is to detect features (keypoint detection + descriptor computation) from the scene and from the template images and to match them. This is usually done via well known algorithms like SIFT (1999), SURF(2006) and ORB(2011). ORB is the fastest algorithm while SIFT performs the best in the most scenarios. In special cases, when the angle of rotation is proportional to 90 degrees, ORB and SURF outperform SIFT and in the noisy images, ORB and SIFT show almost similar performances. So, once the features get detected and matched, in an ideal scenario only 4 matches would be enough to fit the 2D-Homography mapping the template to the scene's plane. The reality is that in any real world case, several bad matches (outliers) will be present. After a coarse cleaning of the worse matches, for instance with a ratio test, a robust model fitting technique is still needed.

2.1 Robust model estimation

In the context of model fitting (in our case an Homography) with the presence of outliers, least squares approach is not a viable option. The squared loss would give too much "decision weight" to the outliers. In theoretical statistics, investigation of robustness started in the early 1960s, and the first robust estimator, the M-estimator, was introduced by Huber in 1964. The basic approach is to find a different loss function (bisquare,cauchy,huber...) which penalizes less large residual values.

The first appearence of RANSAC was in 1981, in a paper published by Fischler e Bolles. It is a random-sampling based technique. Given a dataset whose data elements contain both inliers and outliers, RANSAC uses the voting scheme to find the optimal fitting result. It is a very robust method (handles up to 50% outliers).
Ransac steps to fit a model to a dataset S:

1. Determine a test model from n random data points x1,x2,x3,...,xn, where n is the minimum number of data points required to estimate the model
2. Check how well each individual datapoint in S fits with the test model (datapoints within a distance t constitute a set of inliers)
3. If the set of inliers is the largest set of inliers encountered so far, we keep this test model
4. Repeat 1-3 steps until N tries or until a certain percentage of inliers is found
5. Fit the final model using all the inliers

An issue in RANSAC is the sensibility to the correct choice of noise threshold, defining which datapoints are counted as inliers and which are not. To alleviate this problem, Torr et al. proposed two modified versions: MSAC and MLESAC (maximizes the likelihood that the data was generated from the sample-fitted model). The fundamental idea is to evaluate the quality of the consensus set, and not only the cardinality of it. From 1981 numerous modified versions of RANSAC has been proposed. Part of these improved versions cope with the fact that RANSAC algorithm generates hypotheses by uniformly sampling the input data set. This implicitly means that there is no a priori useful information available (but that's often not the case, as we will show in this project). See, as an example, PROSAC (2005). PROSAC exploits a linear ordering on the set of correspondences

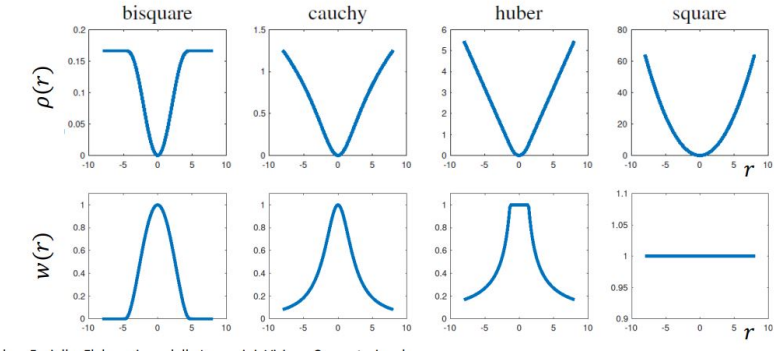

Figure 2: Loss functions and associated weights

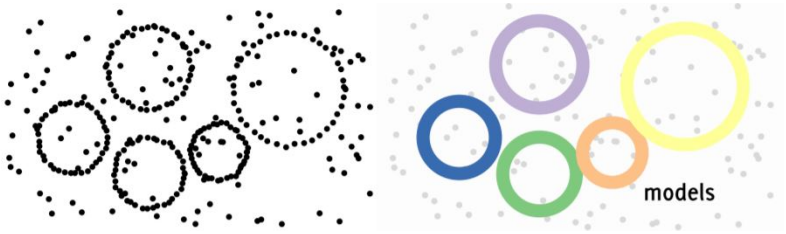

Figure 3: Multimodel fitting

using a similarity function. Unlike RANSAC, which treats all correspondences equally and draws random samples uniformly from the full set, PROSAC samples are drawn from progressively larger sets of top-ranked correspondences.

We want to cite also another popular algorithm, the least median of squares (LMedS), that was introduced by Rousseeuw in 1984. It estimates the parameters by solving a nonlinear minimization problem: finding the smallest value for the median of squared residuals computed for the entire data set. It must be solved by a search in the space of possible estimates generated from the data. Since this space is too large, only a randomly chosen subset of data can be analyzed.

2.2 Robust multi-model estimation

The match of multiple occurrences of the same template in the scene is not trivial. The presence of multiple instances hinders robust estimation, which has to cope with both gross outliers and pseudo-outliers. So it requires a robust multi-model estimation technique. Many algorithms have been proposed. It is possible to identify two main types of approach, connecting the extensive literature on geometric fitting: the analysis of preferences and its dual counterpart, the analysis of consensus. Methods based on consensus sets, like the already mentioned RANSAC and LMedS, try to fit the models choosing randomly a minimum set of datapoints and yield the model with the largest set of inliers. In the context of multi-model fitting some of the most popular consensus-set based solutions are Multi-RANSAC and its variants, Randomized Hough Transform and many optimization algorithms designed for geometric fitting (also based on the maximization of a consensus set). On the other side, preference analysis, introduced by Residual Histogram Analysis, is founded upon the reversal of roles of datapoints and models. The multi-model fitting problem can be considered as a typical example of a chicken-and-egg problem: both the data-to-model assignments and model parameters are unavailable, but given a solution of one sub-problem, the solution of the other can be easily derived. Hence, the preference analysis-based method first generates a large number of hypotheses by sampling a minimum sample set (MSS), and then performs preference analysis on the hypotheses residuals. J-Linkage and T-Linkage are examples of that. Each point is represented with the characteristic function of its preference set, i.e., the set of models that fit the point within a tolerance. Points belonging to the same model will have similar preference sets. In other words, they will cluster in the conceptual space.

A more classical and simple consensus based method, maybe the most natural extension of RANSAC in the multi-model domain, is Sequential RANSAC. When an iteration of RANSAC finally finds the best model, all the inliers are taken away and the following iteration begins.

Sequential RANSAC is still a strong first choice since it is O(N), and generally effective; J-Linkage and T-Linkage, for instance, work better but are O(N^2).

3 Adding depth

For this project, our goal was to improve upon the starting rgb-only based pipeline, exploiting depth data.

3.1 Starting pipeline

With only rgb data available, the starting pipeline was composed by these steps:

1. extract features from the template using SIFT/SURF/ORB
2. extract features from the scene

Figure 4: Intel realsense depth camera

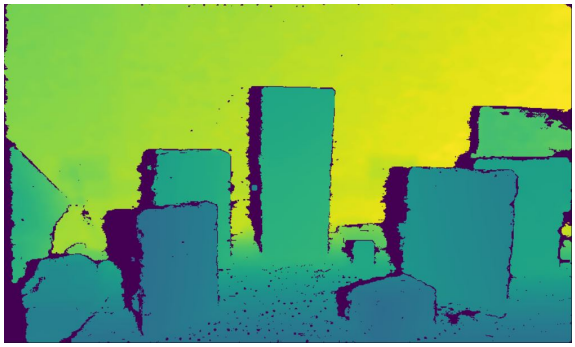

Figure 5: Example of depth image

3. match all the features in the image with the template. We used FLANN based matcher (Fast Library for Approximated Nearest Neighbors).
4. remove ambiguous matches by ratio test
5. fit homographies via sequential ransac
6. draw the bounding box of the template over each instance in the scene image by using the estimated homography

3.2 Depth Map Acquisition

For the acquisition of depth data we exploited an Intel Realsense camera D455 and the python library pyrealsense2. It's important to mention that in order to obtain perfectly corresponding rgb and depth frames, color and depth streams must be aligned first using the align() method. We saved the rgb frame using openCV imWrite, while for the depth data we created a file in which we stored a matrix containing the depth value corresponding to each pixel.

Figure 6: Example of rgb image

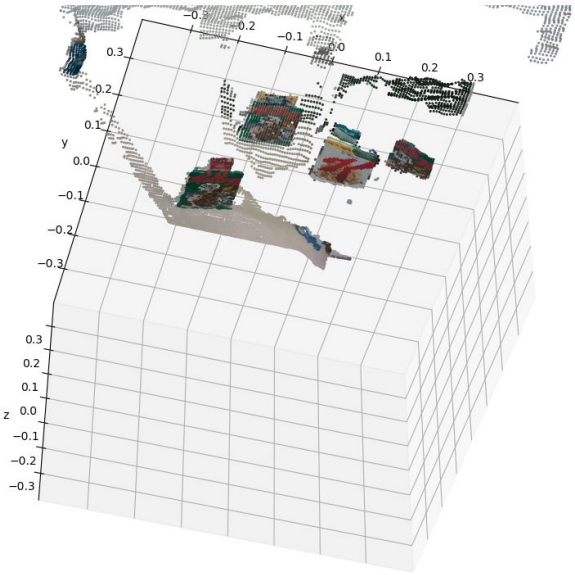

Figure 7: Visualization of a pointcloud

3.3 Building the Pointcloud

The point cloud we built is a mapping between each pixel of the scene image and the 3 dimensional coordinates of the corresponding point in the real world. To build this mapping, we set the Z value as the depth value we measured, while for X and Y:

$$X = Z * u/fx$$
$$Y = Z * v/fy$$

where the pixel corresponding to (u, v) = (0, 0) is in the center of the image, fx and fy are the focal lenghts of the camera. We obtained fx and fy from the field of views values in this way:

$$fx = 1/tan((FovX/2) * \pi/180)$$
$$fy = 1/tan((FovY/2) * \pi/180)$$

Where FovX and FovY are expressed in degrees. In our case, using the realsense D455 with 640x480 resolution:

$$FovX = 92$$
$$FovY = 65$$

3.4 First approach: scene's features co-planarity check

The techniques we are going to present are based on the rejection of the implicit assumption underlying the complete randomness of ransac sampling.The assumption is that there is no regularity in the data we can exploit to have better chances to find a good fit. Our first take to improve the pipeline leveraging 3D data was based on the assumption that all the 4 scene features that lead to the best fit of the homographies must lay on the same plane (they are all on a box's face). So, as we couldn't rely on the opencv "plug and play" RANSAC anymore, we decided to find a simple RANSAC implementation in order to be able to modify it (and hopefully enhance it). We have chosen the implementation from Jordan Hughes (UC Santa Barbara), from the GitHub repository named "HomographyEstimation". We also added a non-degeneracy test to get rid of transformations that do not preserve convexity:

$$(h_{31}x + h_{32}y + h_{33})/det(H) > 0$$

So at this point we were ready to add the co-planarity check. We decided to add another method called customFindHomographyPlane3D, that takes the point_cloud as an extra parameter. In this version, RANSAC tries to fit the 4 matches only when the corresponding 4 scene's feature are on

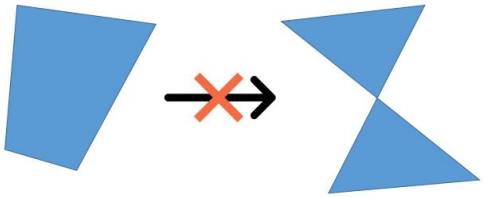

Figure 8: Example of non convex transformation

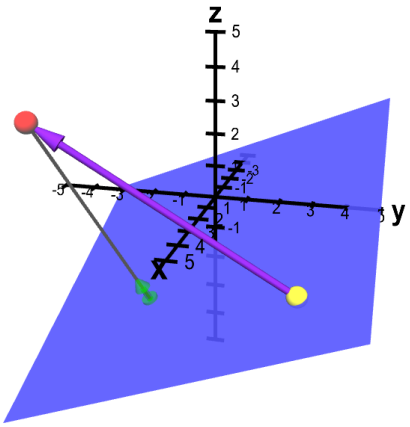

Figure 9: Visualization of point-plane distance

the same 3D plane. To verify if the points in space are on the same plane, we simply find the plane passing through 3 of the points and then we check the point-plane distance between the last point and the plane. If this value is above a certain threshold, then the 4 points are not co-planar.

3.5 Second approach: non-uniform sampling based on 3D distances

The second technique (customFindHomographyNormalSampling3D) is based on non-uniform sampling of the datapoints. The assumption is that the 4 best scene features are somewhat close in 3D space, so it is better to sample closer points than totally random ones. In particular, we sample the first match randomly, we take his scene's feature's 3D position and we sample 3 real numbers using three normal distribution centered in x0,y0,z0 (the first feature's position) and with a certain standard deviation (in the order of tens of centimeters). Once we have sampled this new 3D point we find the closest scene's feature. This "normal-sampling" is repeated for the remaining 2 features.

3.6 Third approach: non-uniform sampling based on a K-d tree

The last technique (customFindHomography3DTree) is also based on non-uniform sampling. In customFindHomographyNormalSampling3D, to find the other 3 matches the algorithm needs to search 3 times the closest feature to the sampled points. This time, in order to exploit the 3D positions without adding an high "fixed cost" in terms of computation time we used a suited data structure: a K-d tree (in this case a 3-d tree). So, before running ransac we build the tree and then we query randomly from it the first feature. The other 3 features are randomly selected inside the pool of 15-20 (this is a parameter that can be tuned) closest ones, quickly available thanks to this particular data structure.

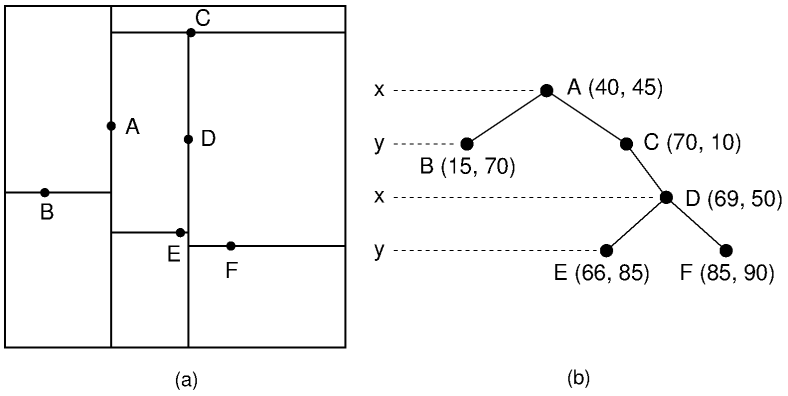

Figure 10: K-d Tree

4 Experiments

In this section we report the experimental activity and the conclusions we have drawn. After the first experiments we observed that to really appreciate the improvements given by this techniques and obtain better results than vanilla ransac in terms of speed and quality of box detection, it is convenient to consider high outlier-ratio settings. That's why in all the following tests we used a ratio test value of 0.87.

4.1 customFindHomography

The first set of experiments were meant to test the new RANSAC implementation taken from Jordan Hughes and modified to be integrated in our codebase. This implementation was written in python2 (we needed python3 code) and used a dfferent format for the data passed to the findHomography method, so we had to make some changes. Before testing our 3D solutions, we needed to make sure that our new vanilla RANSAC led to coherent results wrt the openCV one.

We decided to reimplement the customFindHomography method for each of our solutions so that you only have to change one method inside the main file template_matching_1.py . We also added a version of the pipeline (template_match_multiple_templates.py) which finds all the occurrences of all the templates contained in a directory (but nothing new at a theoretical level).

Figure 11: RANSAC comparison

Figure 12: Multiple templates

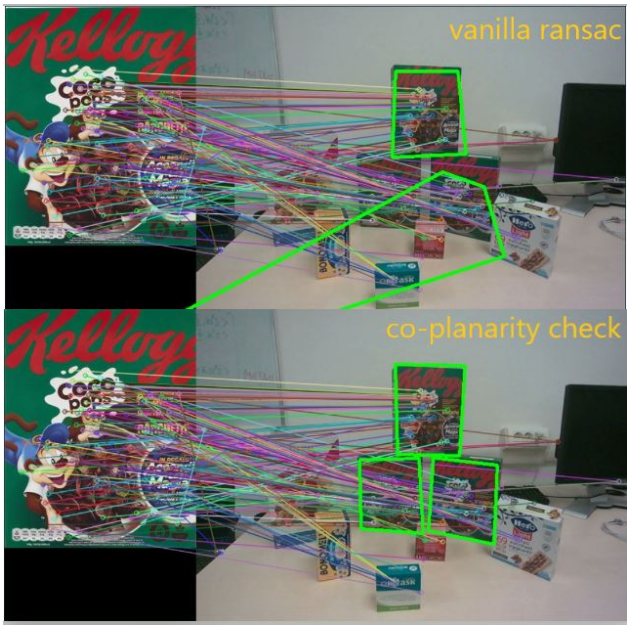

Figure 13: vanilla / co-planarity robustness comparison

4.2 customFindHomographyPlane3D

The first solution we tested was the "co-planarity check", implemented through the method customFindHomographyPlane3D. As we explained in the previous section, this solution keeps sampling randomly the 4 matches until all 4 scene's features pass the co-planarity check. The underlying assumption is that if there is no co-planarity it is better to skip the fitting process, as the fitted homography would be sub-optimal. So we started testing this solution in different settings (you can find all the pics inside the 3D folder) and with different parameters. The tweakable parameters (and this is valid for all three solutions) are:

- ratio of the ratio test (from 0 to 1, an higher value let more features survive the test)
- geometric distance error value (maximum tolerance to consider a fitted homography "good")
- maximum number of RANSAC steps
- customFindHomography threshold parameter (this is the percentage of inliers needed to stop RANSAC before the maximum number of steps)

Parameters used for all the experiments on this first version and the other two, to preserve coherency:

- ratio test value = 0.87
- geometric distance = 4
- maximum number of RANSAC steps = 300
- customFindHomography threshold parameter = 0.4

Looking at the results we concluded that this version is more robust in a high-outliers setting in terms of detection quality. It is also visibly faster when the scene's features groups are segregated in space. See avanti_dietro and avanti_dietro2 tabs in the next chapter: the first box, closer to the camera wrt the others, gets detected faster.

4.3 customFindHomographyNormalSampling3D

The second solution we tested is the one based on normal distributions. Other parameters used:

- std deviation = 0.15

After testing on all the different scene dispositions available (all the sub-directories under the 3D one) we found that the results are consistent with our starting hypothesis. As in the previous method, performances in terms of computation times get boosted only when the groups of scene's features are spatially segregated. In this case, in all the other settings performances gets jeopardized by the high complexity of the search algorithm (everytime it samples scene features, it carries out 3 linear searches). As in the previous case, the effectiveness of our solution is most obvious in a context with high outliers ratio.

4.4 customFindHomography3DTree

To fix the normal sampling sub-optimal complexity at the root we designed the K-d tree version. Our experiments confirmed that this particular data structure optimizes the search of close points in space, as it provides, in almost every case, better results in terms of computation time for the detection of all the boxes. In the last two settings, where the first two methods show their best, the K-d tree version has worse performances (but still better than vanilla ransac). Nevertheless, this is the version that provides the most consistent performances considering the totality of the tests.

4.5 Final conclusions

All the 3 methods provide more robust results in datasets with high percentage of outliers (see Tables 5 and 6 in the next chapter). Moreover, the "co-planarity check" and the "normal-sampling" ones have really low computational times if scene's feature are spatially segregated (see Tables 7 and 8). The K-d tree version is consistently fast, regardless of the scene's characteristics.

5 Test data

Time of 9999.99s means that the box has not been detected.

checkpoint	vanilla	plane	sampling	tree
First Box	20.15s	5.94s	13.54s	5.77s
Second Box	27.12s	18.58s	28.46s	10.23s
Third Box	38.86s	25.59s	35.02s	14.16s

Table 1: avg computation times on 3D\1 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	6.97s	5.94s	10.23s	5.66s
Second Box	12.05s	10.72s	21.84s	10.30s
Third Box	25.81s	14.56s	28.12s	14.89s

Table 2: avg computation times on 3D\2 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	6.72s	6.75s	12.68s	7.67s
Second Box	12.57s	11.86s	19.27s	14.34s

Table 3: avg computation times on 3D\3 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	7.83s	6.73s	14.71s	6.68s
Second Box	13.14s	12.02s	27.53s	14.01s

Table 4: avg computation times on 3D\4 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	0.20s	0.07s	0.81s	3.34s
Second Box	13.46s	5.93s	8.44s	6.76s
Third Box	9999.99s	18.29s	14.82	11.46s

Table 5: avg computation times on 3D\5 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	7.52s	9.61s	14.31s	9.82s
Second Box	49.33s	14.26s	21.87s	15.45s
Third Box	9999.99s	18.06s	45.65s	19.52s

Table 6: avg computation times on 3D\6 test scene

checkpoint	vanilla	plane	sampling	tree
First Box	3.31	1.23s	0.59s	1.28s
Second Box	6.57s	4.11s	8.73s	4.29s

Table 7: avg computation times on 3D\avanti_dietro test scene

checkpoint	vanilla	plane	sampling	tree
First Box	3.45s	0.93s	0.92s	2.19s
Second Box	6.82s	5.04s	7.31s	7.43s
Third Box	9.87s	9.04s	12.21s	11.39s

Table 8: avg computation times on 3D\avanti_dietro2 test scene