Computer Vision, Project - Year 2019/20

Davide Ghion, Matr. 1225841

Padova, 1 February, 2021

1 Introduction

The aim of this report is to explain how the template matching problem, presented by the project, has been developed. In particular, it will be illustrate the steps that made possible to find the final algorithm.
Recalling the problem, it's required to develop a system capable of automatically estimating in an image the best match from a series of views (*i.e.*, models or templates) of an object and trying to localize. Each view corresponds to a position of the object with respect to the camera. The objects taken in account are three: can, duck and driller. Some images are provided to test the algorithm, indeed for each object there are 10 test images and 250 templates. The latter are divided into models and masks, the first one is a colored3D view reconstructed from the object and the second are distinguish in the views the object from the background.

Figura 1: Example of model view and mask

2 Problem solution

The main algorithm is a template matching that is helped with a similarity check to overcome possible outliers, indeed after an upload step of all images (tests, models and masks) *w.r.t.* the object, it's called the template matching algorithm that try to localize the object and his pose checking one by one the models.
When a model is localized in a point then it's saved a cropped part of the test image, because it's used to check and understand if the template matching give a bad result.
When all controls ends it's extracted the 10 best models that are recognized into the test image.

The basic idea of the algorithm is:

1. Load model's and mask's images $model_i, mask_i, i = [1, 250]$.
2. Load test's images $T_j, j = [1, 10]$.
3. Run the template matching algorithm using the model and image $A(model_i, T_j) \forall i, \forall j$ saving the score $s_{j,i}$ and the cropped image $c_{j,i}$.
4. Sort all the results by $s_{j,i} \forall i$.
5. Check similarity between $c_{j,i}$ and $model_i$ with $i = 1...30$ and recompute a new score.
6. Extract by the new score, the 10 best models.
7. Show the image T_j and the boxes of the ten images of the object model with the highest matching scores. (ONLY FOR DEBUG PURPOSE)
8. Save the results in a log file: *results.txt*.

2.1 Template matching

This section represent the first part of the algorithm, it's used the OpenCV function **matchTemplate** to search for matches between a model and a test image.
In order to guarantee that the rules of the project are respected, before applying the real template matching algorithm it's needed to remove any possibility to work with colored images indeed direct color comparison is forbidden.
Hence the first operation is the conversion of the images in grayscale, then it's computed the edge map through **Canny edge detector**.

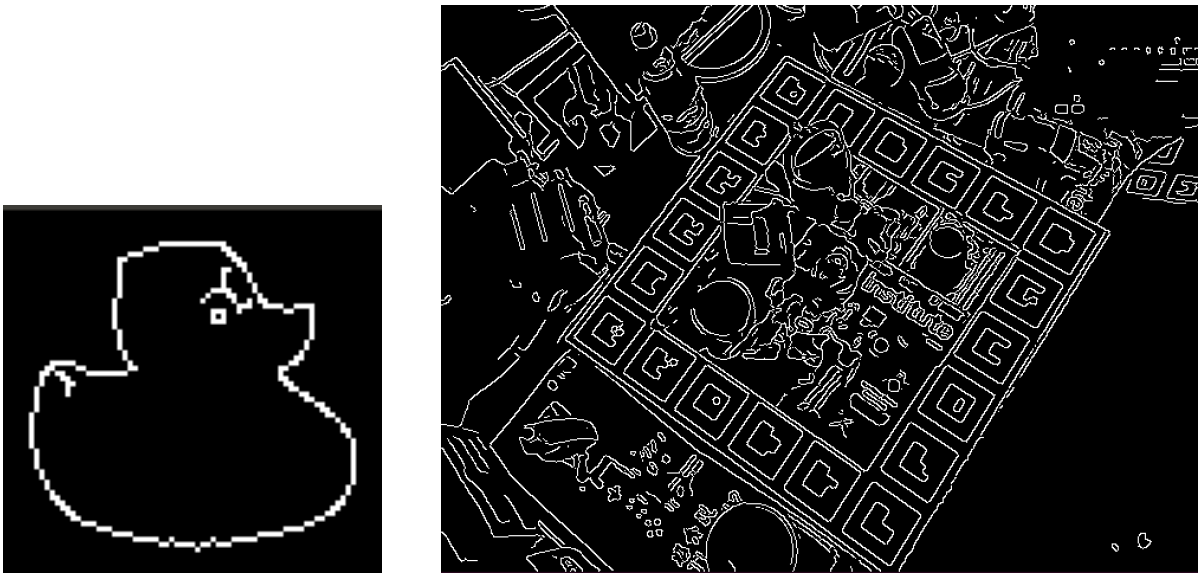
For the model image the *Canny* is applied directly indeed there isn't problem of illumination or rotation changes, after a tuning phase it's found best thresholds $T_{low} = 0$ and $T_{high} = 100$ and kernel size 3×3.

Although for the test images there another situation (*e.g.* the photos are not illuminated in the same way), so it's needed another way to set up the edge detector: **Otsu's method**.
This method is a global thresholding that minimizes intra-class variance and depends on the intensity histogram, so before the call of *Canny* the image is smoothed with a Gaussian blur (no ringing in the image) and compute the *Otsu's* threshold. Then the two thresholds for the computation of the edge map are derived from the latter:

$$T_{low} = T_{Otsu} * 0.6 \qquad T_{high} = T_{Otsu} * 0.8 \tag{1}$$

Now it's possible to use the *matchTemplate* function:

$$matchTemplate(edge_{T_j}, edge_{m_i}, result, method)$$



(a) Edge map duck model

(b) Edge map test image

Figura 2: Example of *Canny* output

To identify the matching area, the template image is compared against the source image by sliding it, at each location a metric is calculated so it represents how "good" or "bad" the match at that location is, this metric is defined by the *method*'s field, it's used `TM_CCOEFF`:

$$R(x,y) = \sum_{x',y'} T'(x',y') \cdot I'(x+x', y+y') \tag{2}$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} T(x'',y'')$$
$$I'(x+x', y+y') = I(x+x', y+y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} I'(x+x'', y+y'')$$

Notice that R is the result matrix, using the method above, the higher values represent better matches indeed it's localized the minimum and maximum values in the result matrix by using **minMaxLoc** function:

$$minMaxLoc(result, minVal, maxVal, minLoc, maxLoc, mask)$$

All the informations about the comparison are stored in a *MatchInfo* object (see **Section 3.1**) , such as the score and the point where the model is found. Furthermore it's also saved sub-window cropped from the test image, that correspond to the result image of the template matching.

Figura 3: From the left the crop of the result, from the right corresponding model

2.2 Similarity check

This section represent the second part of the algorithm, indeed to be in this step it's needed to have processed all the models in the test image T_j. It means that all the models have a score, that belong from the template matching, and the vector of *MatchInfo* objects are sorted by the score descending.
Hence the aim of this control, is to find and remove possible outliers, in order to do this it's compared the best 30 cropped result of the template matching (with the best score) with the corresponding models, then compute a new score and again sort them.
Before compute the metric between the two images, to have a good comparison, the background of the cropped result must be removed and it is easily done by filtering it with the corresponding mask image.

Figura 4: Filtering of the cropped result with the corresponding mask

Now it's computed a measure of similarity using the **Structural Similarity Index (SSIM)**, this will return a similarity index averaged over all channels of the image. This value is between zero and one, where one corresponds to perfect fit.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{3}$$

Where respectively $\mu_{x,y}$, $\sigma^2_{x,y}$ are the mean and the variance of x, y, σ_{xy} is the covariance of x and y and $c_{1,2}$ are two variables that stabilize the division with weak denominator having information

about the dynamic range of the pixel-values, in this algorithm they have fixed values: $c_1 = 6.5025$ and $c_2 = 58.5225$.
To understand how this index works it's referred to the OpenCV tutorial: *Video Input with OpenCV and similarity measurement*, moreover the argument is described more in-depth in *"Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, Apr. 2004."* article, where it's explained how the formula (3) is derived.

3 Project description

3.1 Class *MatchInfo*

This class is designed to easy up the share of information between the different parts of the algorithm, for example it's used on the exiting of the template matching in order to store all the informations about the results found.

```
class MatchInfo {
        public:
                // Variables definition
                Point coords;
                double score;
                int templNumber;
                Mat matchedImage;

                // Getter and setter
                void setMatchedImage(Mat);
                void setCoords(Point);
                void setScore(double);
                Mat getMatchedImage();
                Point getCoords();
                double getScore();
};
```

1. `coords` the spatial coordinates of the template matching result
2. `score` the value of the score matrix $R(x, y)$
3. `templNumber` the corresponding number of the model image
4. `matchedImage` the cropped image of where model is identified

3.2 Project structure

The entire project is written in C++, it maintains the structure suggested from the problem text indeed:

```
.
├── ./build
│   └── ./build/CMakeFiles
├── ./images
│   └── ./images/pose_estimation_dataset
├── ./include
└── ./src
    ├── ./src/main.cpp
    └── ./src/results_writer.cpp
```

Figura 5: Structure of the project

1. `src` folder contains all the sources code, in this case there are only two C++ files: *main.cpp* and *results_writer.cpp* where the first one contains all the code of the algorithm and the second is the file provided to write the results.

2. `include` folder contains the header file *result_writer.h*.

3. `build` folder contains all the files and folders generated by the cmake in order to compile the entire project

4. `image` folder contains all the images used by the algorithm (test, models and masks), but the position is relative indeed on program execution it can be possible to specify the path of the *pose_estimation_dataset* folder.

3.3 Program execution

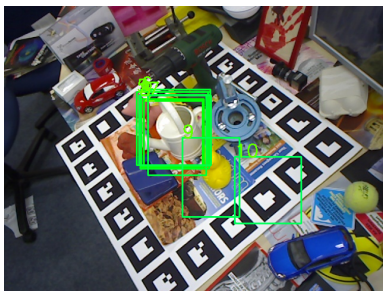
As said above there is the possibility to specify the path folder that contains all the images, this input parameter is called *imagesPath* indeed to execute correctly the program the, absolute or relative, path to the folder must be declared like this:

```
./project --imagesPath=/path/to/pose_estimation_dataset
```

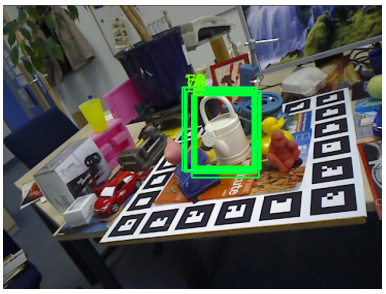
Notice that if you are interested to view the resulting images with bounding box then you need to set the global variable `debug = true` otherwise the program doesn't stop the execution and analyze all the test images writing the results on the text file associated.
Another important configuration is the vector of strings called `objects` that contains the name of the objects (*i.e* can, driller and duck) then you can specify the objects, and the order, of the objects that the algorithm take in account.

4 Results obtained

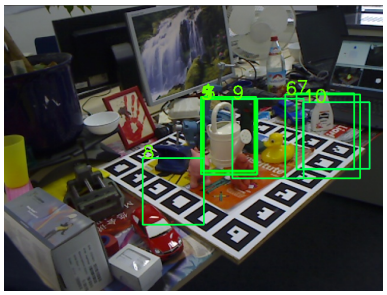
In this section there showed all the results obtained from the algorithm. In some of the test images the objects are not correctly localized the main reasons that cause this problem is the similarity with other objects and the poor quality of the edge map.
These problems affect mainly the duck and the driller, indeed the matches are very sparse, for example the duck is really similar to the white cup (it can be visible in the figures *7.d*, *7.e* and *7.h*). In the other hand, the can is very visible thank to the specific shape of the object, also the internal edges helps to the localization, at the end, the solution proposed works well with objects that have peculiar shapes.



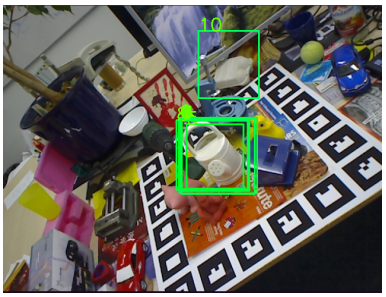
(a)



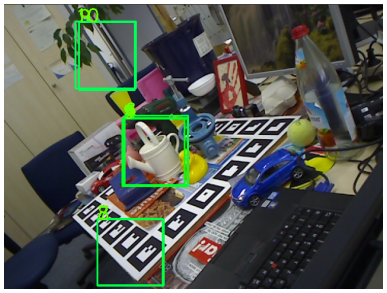
(b)



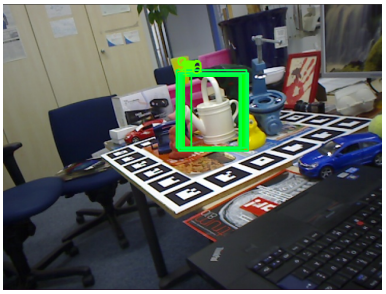
(c)



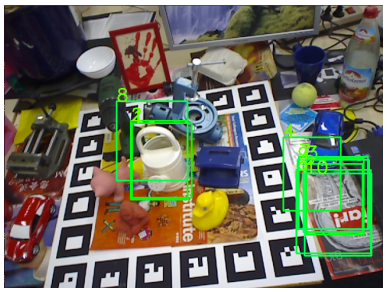
(d)



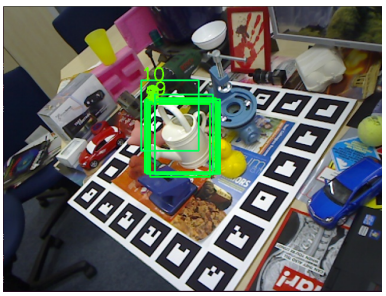
(e)



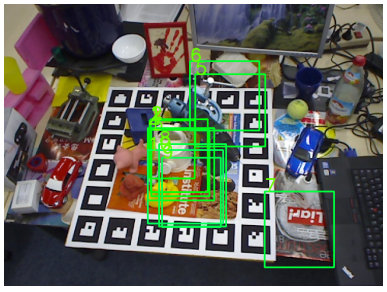
(f)



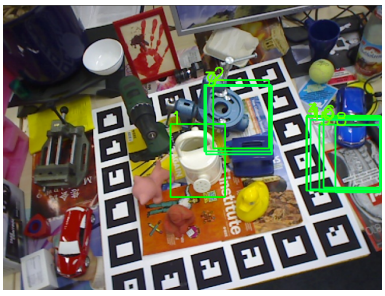
(g)



(h)

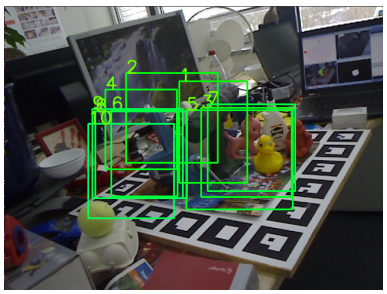


(i)

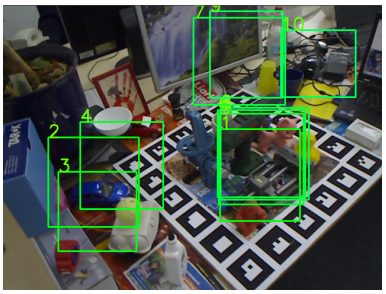


(j)

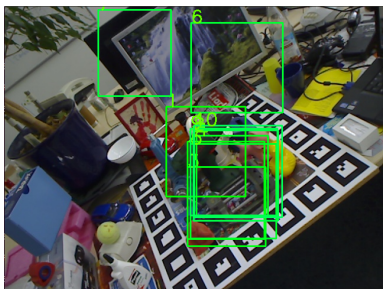
Figura 6: Result with the can



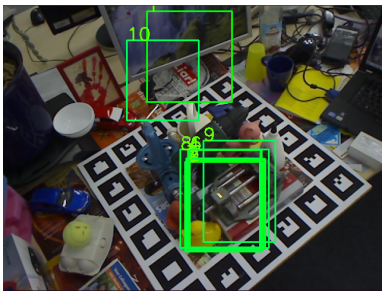
(a)



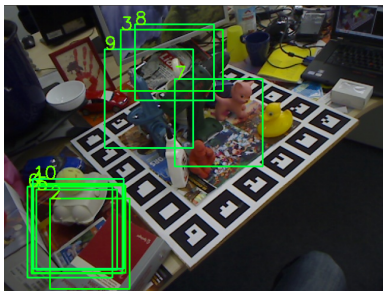
(b)



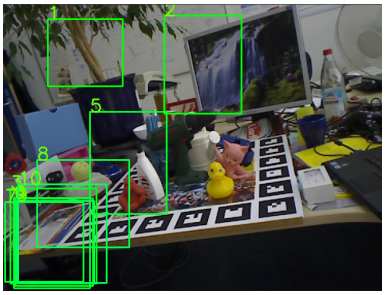
(c)



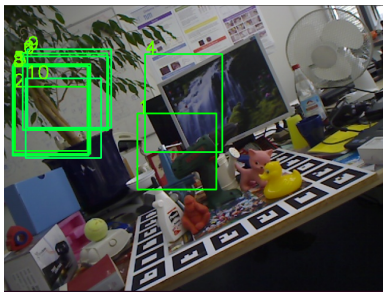
(d)



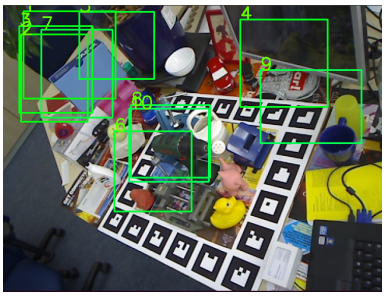
(e)



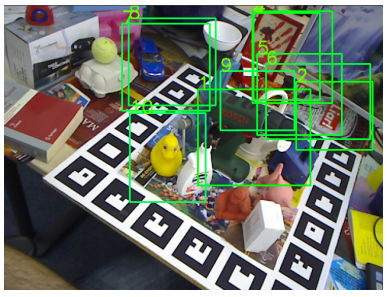
(f)



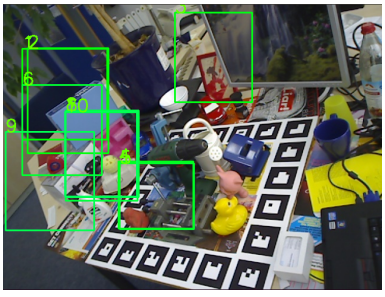
(g)



(h)

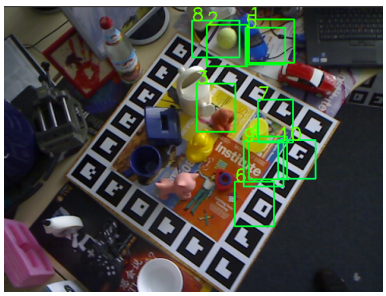


(i)

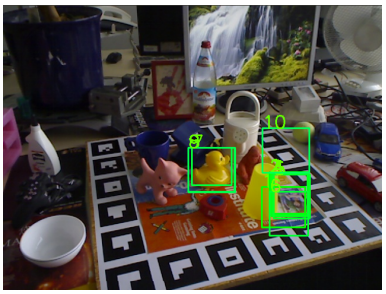


(j)

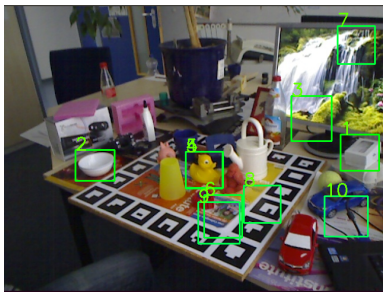
Figura 7: Result with the driller



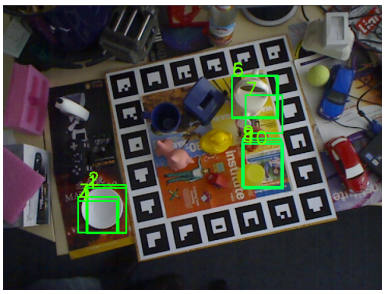
(a)



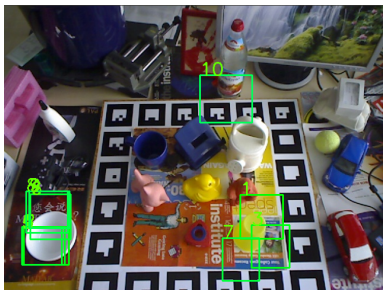
(b)



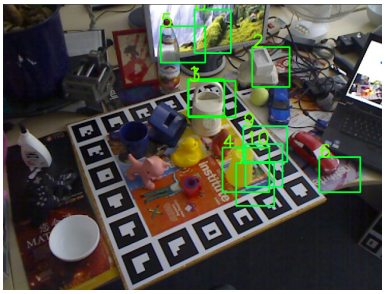
(c)



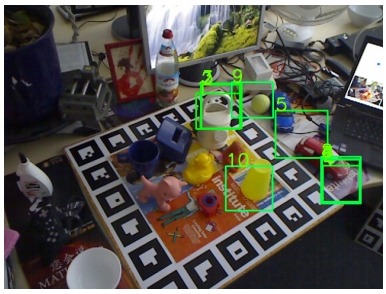
(d)



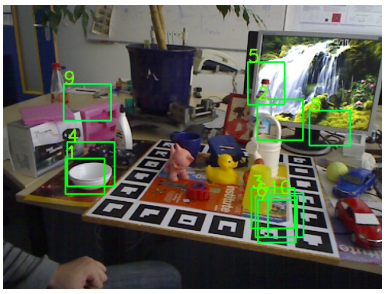
(e)



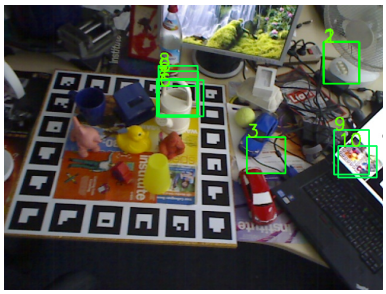
(f)



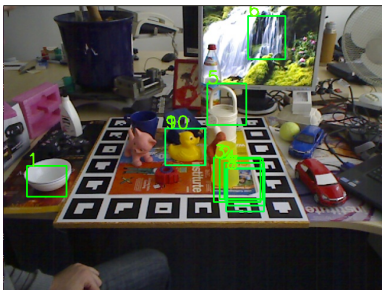
(g)



(h)



(i)



(j)

Figura 8: Result with the duck

5 Possible improvements

Clearly, a first important improvement is the remove of any constraints, then being able to use the color for comparison is very easy to get the a better idea on what the template matching has found. Furthermore to improve the algorithm, I tried to add a feature matching algorithm (*i.e. : SIFT, SURF* or *ORB*) to get more details as possible and compare the template matching result with the model. The problem is that when template matching fails finding outlier also the feature matching does, the score computed from the latter may can beat a good result, this implies that it's not impossible to discriminate a good result from a bad one.

5.1 Another solution

Another possibility is to try a segmentation method in order to do image segmentation, for example the **Watershed Algorithm** used into the test image, then having a set of regions (of single objects) segmented it can be used the **Generalized Hough Transform** in order to detect the presence of the mask into the test image.