Computer Engineering Master Degree

Computer Vision

Report about Project

Raffaele Di Nardo Di Maio 1204879

September 11, 2020

1 Method used

1.1 Project details

The goal of the experience was to find in an image the best match from a set of views of an object that we are trying to localize. The steps followed by the implemented program, to reach the goal, are the following:

1. **Computation of γ-transform**
   the used gamma values are all greater than 1. This transform is applied to increase the contrast and stretch very high pixel values in an image. This operation is performed only on views because of their very high brightness.

2. **Computation of Canny detection**
   Canny detector is applied to the test images and also to the γ-transformed views.

3. **Matching phase**
   this phase is performed using two different approaches, depending on the specification of the optional argument `-dist` (see Section 2.2). These approaches are:

   - **Template Matching with distance transform**
     The distance transform is applied to each test image with detected edges. Then the function `cv::matchTemplate()` is applied to the result of the transform using, as templates, the views on which Canny was applied. The metric used is the *Correlation Coefficient (TM_CCOEF* in opencv*)*. Then the max value of an element in the resulting matrix is kept and its position is used as the location of the template for the corresponding match. With this approach I take only the ten best matches for each image, as specified in the assignment, by looking only at the best matches of each view in that image. The score value used to establish the best matches is the value contained in the max entry of the result matrix.
   - **Template Matching with histogram refinement**
     Template matching is performed on the test image (on which Canny was applied) and using view images processed with Canny as templates, looking at TM_CCOEF as in the previous approach. To improve the results obtained, I don't take only the 10 best matches for a test image but the 50 best results and then I apply a histogram-based refinement algorithm. For each one of the 50 matches I take a sub-window, related to the position of the match and with size equal to the related view, and I filter it with the corresponding mask. This RGB sub-window of the original test image is then converted to the HSV space and the histogram of its Hue channel is computed. The resulting histogram is then compared with the Hue histogram of the original view image using `cv::compareHist()`. The final score assigned to a match is the composition of the sliding window score and the histogram comparison score:

     $$final\_score = \alpha * match\_score + \beta * hist\_score$$

     where $match\_score$ is the score obtained by applying only `cv::matchTemplate()` and $hist\_score$ is the score obtained through `cv::compareHist()`. Hence from the initial set of 50 best matches obtained through template matching, based on sliding window, I only take the 10 best matches w.r.t. new final score.

4. **Generation of result text files and result images**
   for each of the 10 best matches, the edges of its mask are highlighted in red and printed on the position evaluated before over the image. Each result image has a name with this format:

   `datasetNum_matchNum.jpg`

   where `datasetNum` is the name of the dataset folder (it can be *can*, *driller* or *duck*) followed by the number of the corresponding test image and `matchNum` is an integer number in $[0, 9]$ that identifies the quality of the score of a match in the set of 10 best matches (`0` for higher score, `9` for lower score).
   The edges of the view that matches in the test image are highlighted in red. For each dataset the matches are written in the corresponding results text file with the format specified in the assignment.

1.2 Experimental results

Looking at the details of the previously analysed steps of the implementation, I explain the reasons and the results obtained with the previous operations:

1. **Computation of γ-transform**
   The γ-transform was applied after different trials because of the presence of too many weak edges and because of the colours of the views. Instead of using this transform, I tried to equalize the images but the results were worse than those obtained without any single-pixel operations. After several trials, the final values of γ in γ-transform applied to each set of views are the following:

	can	**driller**	**duck**
γ	1.4	1.3	1.4

Table 1: γ values for γ-transform applied on views of each dataset.

2. **Computation of Canny detection**
   For each dataset I defined two different high thresholds of the Canny detector for each dataset: the first for the views and the second for the test images. Then, looking at the results obtained, I assigned to the low thresholds of Canny detector the values of the high thresholds decreased by a specific quantity: 30 for views and 40 for the test images. The values for high thresholds are the following:

	can	**driller**	**duck**
test images	150.0	150.0	100.0
views	100.0	160.0	100.0

Table 2: High thresholds for the Canny detector.

   The high thresholds of the *duck* dataset are both smaller than the others because more details are needed to correctly detect the shape of the wing and the beak of the object.

3. **Matching phase**
   In this phase for each image of each dataset, the two methods obtain both all the best feasible matches. In several cases the correct match isn't the match with highest score (e.g. Figure 1). For both approaches, the matching was more simple on the `can` dataset and very complex for images of the `driller` and `duck` datasets.
   The main problem of these methods based on histogram refinement was to find the best thresholds for Canny detection of `driller` and `duck` because of many problems based on their shapes. In particular for `driller` matching, too small values for the high Canny threshold help me to identify better the object in some images but increase the loss of correct matches on other ones. The same happens with too big values for the high Canny threshold. The reason comes from the level of contrast of different images and the smoothness of some of them.
   In particular, the method with the distance transform is more efficient in terms of memory occupation because it stores less results. As explained before, the histogram comparison performed during the Refinement phase only involved the Hue channel of the HSV colour space. I did not work with the RGB colour space because the test image objects and their corresponding models differed in luminosity and saturation. I tried another approach to estimate the final score from the histogram comparison without combining it with the previous one. I tried to use only the histogram comparison score to discover, from the set of 50 best matches, the new 10 best ones. The two approaches implemented were:

- take only matches with the histogram comparison score greater than a specified threshold;
- take the 10 matches with the highest histogram comparison score

For both methods, I noticed that having higher histogram comparison scores doesn't mean that there will not be false positives. For this reason I decided to give a higher weight to the previous Template Matching score and combine it with the histogram comparison score in the following way:

$$final\_score = 100 * match\_score + hist\_score$$

The coefficients were computed after trying several combinations of them.

(a) Match with the highest score on test1.jpg.

(b) Correct match on test1.jpg.

(c) Match with the highest score on test3.jpg.

(d) Correct match on test3.jpg.

Figure 1: Comparison between the highest score match and the correct match on two images of the `driller` dataset.

2 Code

2.1 Performance

The program can analyse only one dataset or all the datasets in parallel. The management of this behaviour was done using threads. Each thread performs the template matching on a dataset and the terminal window, for log printing, is the only resource managed using mutual exclusion.
Referencing to Section 1.1, I'm going to discuss the little tricks that I used to improved efficiency of the phases of the algorithm:

1. **Computation of γ-transform**
   At the instantiation of the GammaTransform object, I create a vector of size equal to the number of pixel levels to reduce the computational complexity. In position i of this vector there is the value of the γ-transform for the input level i. In this way all the following computations of gamma transform on an image require only $O(M * N * channels)$ accesses in that vector where $M * N$ is the size of the image and $channels$ is its number of channels.

2. **Computation of Canny detection**
   The computation of Canny detection for each view image is performed before the Matching phase and the results are stored. In this way then we need to compute the edges only once for each test image but no more for each view image.

3. **Matching phase**
   To improve the efficiency of the buffer containing the best matches for each image, I use a sorted vector (by increasing score of matches). Every time that I obtain the best position/score of a mask in the image, I try to insert it in the buffer following this approach:

Algorithm 1: Insertion

```
1 if buffer is full then
2     if match.score > buffer[0].score then
3         buffer.insert_ordered(element);
4 else
5     buffer.insert_ordered(element)
```

   The insertion is based on the search of the position in the buffer that maintains sorted the vector even if the new element is inserted. In my program was implemented using a `std::vector`. In this way if the vector was already composed by the maximum number of elements, I find the position for the new match and I insert it. Then I immediately remove the first element of the buffer, that has the minimum score w.r.t. all the others. The two template matching approaches use buffers of different sizes, as explained in the following lines:

   - **Template Matching with distance transform**
     for each test image a buffer of max size equal to 10 is created. The best match of each view is inserted in the buffer with the score obtained from `cv::matchTemplate()`.
   - **Template Matching with histogram refinement**
     for each test image a buffer of max size equal to 50 is created. The best match of each view is inserted in the buffer with the score obtained from `cv::matchTemplate()`. Then a new buffer of max size equal to 10 is created. After reading all the best matches from the first buffer, I insert them in the second buffer using the *final_score* previously described.

4. **Generation of result text files and result images**
   If the user decides to analyse all the datasets in parallel, each thread creates the text result files and writes in each of them. The same happens for the creation of the images with 10 best detected matches. The use of threads was intended to write in parallel these files without loosing too much time. All the results were written in reverse order w.r.t. the buffer of results. This is because the buffer is sorted w.r.t. increasing score value but the detected object are printed from higher to lower best score values.

2.2 Program execution

The program needs to receive the following command line arguments with the specified format:

```
-i input_path [-r results_path] [-o output_path] [-h] [-dist]
```

-i input_path *Mandatory argument*
`input_path` is the existing folder that contains the sub-folders of the three datasets (`can/`, `driller/` and `duck/`). Each one of these sub-folders must also contain, as specified in the assignment of the project, the two sub-folders `models/` and `test_images/`.

-r results_path *Optional argument*
`results_path` is the existing folder that will contain the text files describing the ten best matches for each image (`can_results.txt`, `driller_results.txt` and `duck_results.txt`).
If this argument isn't specified by the user, the text files are stored by "default" path `../../../`

-o output_path *Optional argument*
`output_path` is the existing path in which the program will save all the test_images, modified by printing the edges (detected through Canny) of the views corresponding to the 10 best matches.
These are written in red on the image and the name of the related model is shown in the top-left.
corner of the image.
If this argument isn't specified, the result images with highlighted matches are not going to be stored on disk.

-h *Optional argument*
if this argument is specified, even if there are other arguments, the program exits printing the description of possible command line arguments.

-dist *Optional argument*
if the argument is specified, the distance transform method will be used in the computation of the best matches, otherwise the histogram refinement method will be used instead.

The parser that processes the command line arguments also detects if any of them is inserted twice and if "-i", "-r", "-o" are typed without the specified paths. An example of a possible command on the terminal on Windows is the following:

```
./Project.exe -i ../../../dat -r ../../../results -o ../../../results -dist
```

After running the program, a menu page will be displayed and the user can interact with it, selecting if he wants to apply Template Matching on a single dataset or in parallel to all the datasets (see Figure 2 and 3). The user can also exit from the program using this menu that will be displayed again at the end of the desired computation. To select an option, the user needs to type the number related to the action that he wants to perform.

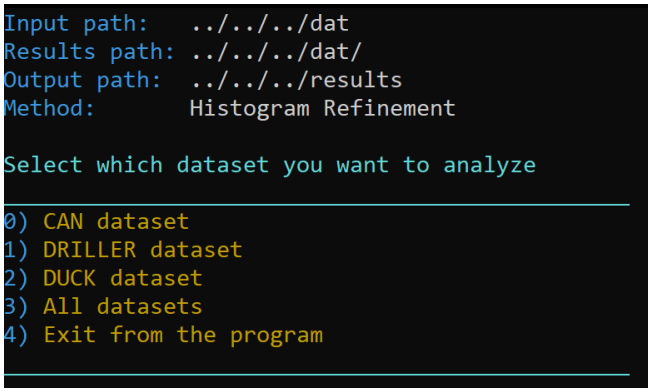

Figure 2: Appearance of the initial Menu.

```
../../../dat/duck
_____________________________________________
Dataset of duck
Number of masks images:   250
Number of views images:   250
Number of test images:    10
_____________________________________________


../../../dat/driller
_____________________________________________
Dataset of driller
Number of masks images:   250
Number of views images:   250
Number of test images:    10
_____________________________________________


../../../dat/can
_____________________________________________
Dataset of can
Number of masks images:   250
Number of views images:   250
Number of test images:    10
_____________________________________________
```

(a) Initial program log with the description of the input.

```
test 0:  mask60 >>> 2.63465     in (299,199)
test 0: mask239 >>> 2.48044     in (290,104)
test 0: mask173 >>> 2.46037     in (282,105)
test 0: mask151 >>> 2.3914      in (241,329)
test 0:  mask73 >>> 2.35711     in (319,21)
test 0: mask192 >>> 2.34525     in (259,39)
test 0:  mask18 >>> 2.34517     in (301,205)
test 0: mask143 >>> 2.34024     in (300,181)
test 0:  mask30 >>> 2.33383     in (191,6)
test 0:  mask45 >>> 2.33006     in (217,261)
test 0: mask145 >>> 3.53062     in (226,144)
test 0: mask147 >>> 3.23787     in (225,144)
test 0:  mask61 >>> 2.94583     in (112,251)
test 0: mask135 >>> 2.8512      in (385,269)
test 0: mask216 >>> 2.78627     in (136,293)
test 0: mask228 >>> 2.64655     in (107,239)
test 0: mask144 >>> 2.6387      in (230,150)
test 0: mask217 >>> 2.55541     in (107,240)
test 0: mask124 >>> 2.54647     in (378,259)
test 0: mask213 >>> 2.50481     in (163,335)
test 0: mask109 >>> 4.52507     in (295,126)
test 0: mask107 >>> 3.20882     in (296,125)
test 0: mask108 >>> 2.89834     in (294,132)
test 0: mask155 >>> 2.86743     in (283,141)
test 0: mask182 >>> 2.73468     in (148,167)
test 0: mask203 >>> 2.72802     in (150,171)
test 0: mask197 >>> 2.66381     in (193,160)
test 0: mask105 >>> 2.66319     in (292,132)
test 0: mask165 >>> 2.65544     in (358,202)
test 0: mask154 >>> 2.64893     in (284,142)
```

(b) Log with image number, matched mask, score and position of the mask.

Figure 3: Logs during execution on all datasets in parallel.

2.3 Code Organisation

The program is composed by 13 files organized, looking to their functionalities, into the following sets:

- **Template matching**
  It's composed by the `TemplateMatching.hpp` and `TemplateMatching.cpp` files that implement the class used to perform the two methods for estimating the best matches.
- **γ-transform**
  It's composed by the `GammaTransform.hpp` and `GammaTransform.cpp` files that implement the class used to perform the γ-transform.
- **Menu**
  It's composed by the `Menu.hpp` and `Menu.cpp` files that implement the static class with parser of the command line arguments and management of menu printing on screen.
- **Main activity**
  It's composed by the `Project.hpp` and `Project.cpp` files that implement the main function and manage threads to perform Template Matching.
- **Canny detection**
  It's composed by the `CannyDetector.hpp` and `CannyDetector.cpp` files that implement the class used to perform Canny Detection on a specific image.
- **Storing of best matches**
  It's composed by the `BestResults.hpp` and `BestResults.cpp` files that implement the classes:
  - `Result`
    it defines the main information to be stored for each result of the matching approaches and implements the methods needed to access these fields.

- `BestResults`
  it constructs the buffer of results and manages its update.

- **Utility constants**
  It's composed by the `Utility.hpp` file and declares some useful variables:
  - Colours used in log printing
  - Colours used in Menu
  - Colours used for printing the mask name and the edges of the matched mask in the result image
  - Gamma values for Gamma Transform
  - Threshold values for Canny Detection
  - Name of sub-folders needed to load images of datasets provided by the user