University of Padua
Computer Vision
Term Project
Orkhan Bayramli
24/06/2021

Introduction

In this project, we will be detecting the objects in the sceneries. In order to do that, we will be using SIFT (Scale-invariant feature transform) algorithm to extract the keypoints and descriptors, and FLANN (Fast Library for Approximate Nearest Neighbors) to match the keypoints of object and scenery from OpenCV library, v.4.5.2. The project is implemented in XCode, macOS. The report will explain the algorithms and their implementation in a detailed way.

Explanation

First, we start with loading the images. The code is below:

```
const char* keys =
"{ help h |                  | Print help message. }"
"{ input1 | dataset3/obj2.png | Path to input image 1. }"
"{ input2 | dataset3/scene2.png | Path to input image 2. }";

//-- Step 0: Load images as Mat arrays and convert their colorspace to
grayscale
CommandLineParser parser( argc, argv, keys );

img_obj_colored = imread( samples::findFile( parser.get<String>("input1")));
img_scene_colored = imread( samples::findFile( parser.get<String>("input2")));

/// Convert to grayscale
cvtColor(img_obj_colored, img_object, COLOR_BGR2GRAY);
cvtColor(img_scene_colored, img_scene, COLOR_BGR2GRAY);

/// Sanity check
if ( img_object.empty() || img_scene.empty() )
{
    cout << "Could not open or find the image!\n" << endl;
    parser.printMessage();
    return -1;
}
//--Step 0: end
```

We store the paths to the images in a constant string called **keys**. With the help of **CommandLineParser,** the corresponding images are retrieved from the paths and stored in the **Mat** arrays from OpenCV.

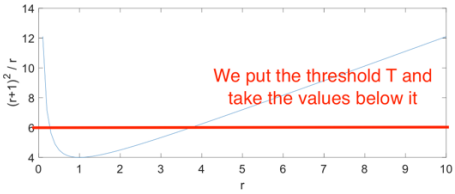
cvtColor() then convert the images to the grayscale ones and store in the Mat arrays. Sanity check is applied to prevent us from using the empty or unloaded images. **“-1”** is returned to signify that an error occurred.

```
//-- Step 1: Detect the keypoints using SURF Detector, compute the descriptors
Ptr<SIFT> detector = SIFT::create(0, 3, 0.015, 10, 1.6);

detector->detectAndCompute( img_object, noArray(), keypoints_object,
descriptors_object );
detector->detectAndCompute( img_scene, noArray(), keypoints_scene,
descriptors_scene );
showKeypoints();
//-- Step 1: end
```

In order to detect and compute the keypoints and descriptors of an image, the SIFT algorithm is used. The constructor *create()* uses the default recommended values for SIFT. They are as following:

- *nfeatures = 0*, the number of keypoints to be extracted from the image. 0 means there is no limit
- *nOctaveLayers = 3*, the number of layers in each octave. 3 as it is suggested in Lowe's paper. The total number of layers will be 6 = (3+1) + 2
- *contrastThreshold = 0.09,* the gradients below that will be discarded (low contrast as a low gradient signifies the weak change). The paper suggests 0.03 but OpenCV suggests 0.09. However, because of the dataset 4, I used the value of 0.015 which gave me better results.
- *edgeThreshold = 10,* $\frac{Tr(H)^2}{Det(H)} = \frac{(r+1)^2}{r} < T$, the value of T.

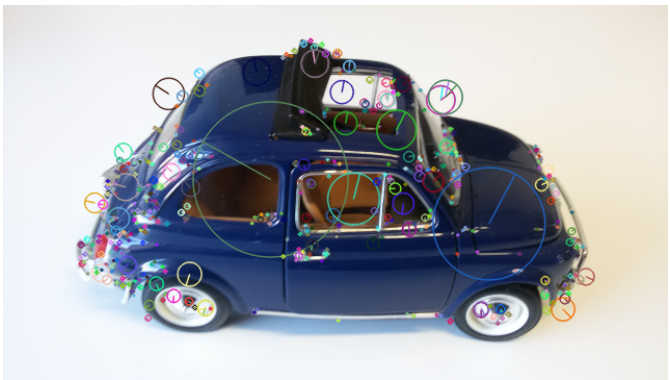


the larger the edgeThreshold, the less features are filtered out (more features are retained). In this formula we basically try to see how strong the change in gradients is by computing their eigenvalues and eigenvectors. Larger the eigenvalues, stronger the response to the changes. It is stronger in the corner as we move by x and y directions ,we will have steep changes since two edges are almost orthogonal in the corners.

- *sigma = 1.6,* Gaussian smoothing with the standard deviation of sigma which is 1.6 by default.

Next, we call the *detectAndCompute()* function of SIFT class by providing it with parameters such that **InputArray** – an image to extract the keypoints and their descriptors. We can also provide it with a mask however I have skipped it with passing *noArray()* which returns an empty array.

In the next picture we can see the keypoints of an image:



The circles indicate how strong the gradient is which eventually means how good the feature is. An arrow in the circle indicates the direction of gradient. We can see there are lots of features on the back of the car but almost nothing on the doors as they are more homogenous areas compared to other parts.

```
//-- Step 2: Matching descriptor vectors with a FLANN based matcher
Ptr<DescriptorMatcher> matcher =
DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
std::vector< std::vector<DMatch> > knn_matches;
matcher->knnMatch( descriptors_object, descriptors_scene, knn_matches, 3 );
//-- Step 2: end
```

Now, here is the final step of SIFT algorithm implementation. We need to match the keypoints of two images which are object and scenery. We are using FLANN based descriptor matcher. The count of best matches has been set to 3 which means that the matches equal and below then 3 will be taken into account.

```
//-- Filter matches using the Lowe's ratio test
const float ratio_thresh = 0.75f;
std::vector<DMatch> good_matches;
for (size_t i = 0; i < knn_matches.size(); i++)
{
    if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
    {
        good_matches.push_back(knn_matches[i][0]);
    }
}
```

As result, we will get many matches but not all of them are good. We need to filter the good ones and store them in a separate array. **knn_matches** is a vector of **DMatch** vectors. We have storing two **DMatch** objects in a single index. The first one is the match in object and the second one is the match in scenery. Lowe suggests applying the threshold of 0.75 and take the ones below that.

```
    // We have the (x,y) locations of obj and scene keypoints stored in obj and
scene, respectively.
    // We get the Homography matrix which will help us to detect the
coordinations of object in the
    // scenery.
    Mat H = findHomography(obj, scene, RANSAC);

    //-- Get the corners from the image_1 ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = Point2f(0, 0);
    obj_corners[1] = Point2f( (float)img_object.cols, 0 );
    obj_corners[2] = Point2f( (float)img_object.cols, (float)img_object.rows );
    obj_corners[3] = Point2f( 0, (float)img_object.rows );
    std::vector<Point2f> scene_corners(4);

    // get the locations of obj corners in the scene by using homography. the
corner locations
    // are stored in the scene_corners vector.
    perspectiveTransform( obj_corners, scene_corners, H);

    //-- Draw lines between the corners (the mapped object in the scene -
image_2 )
    line( img_matches, scene_corners[0] + Point2f((float)img_object.cols, 0),
          scene_corners[1] + Point2f((float)img_object.cols, 0), Scalar(0, 255,
0), 4 );
    line( img_matches, scene_corners[1] + Point2f((float)img_object.cols, 0),
          scene_corners[2] + Point2f((float)img_object.cols, 0), Scalar( 0, 255,
0), 4 );
    line( img_matches, scene_corners[2] + Point2f((float)img_object.cols, 0),
          scene_corners[3] + Point2f((float)img_object.cols, 0), Scalar( 0, 255,
0), 4 );
    line( img_matches, scene_corners[3] + Point2f((float)img_object.cols, 0),
          scene_corners[0] + Point2f((float)img_object.cols, 0), Scalar( 0, 255,
0), 4 );
    //-- Show detected matches
    imshow("Good Matches & Object detection", img_matches );
    waitKey();
    return 0;
```

As we now have the keypoints, we utilize the RANSAC algorithm and find the homography matrix H to correctly identify the corners of object in scenery. *perspectiveTransform()* fills the **scene_corners** vector with coordinates of four corners of an object in the scenery. At last, the lines between these corners are drawn.

Example