VISION FOR ROBOTICS
LAB 3 & 4 - TRACKING A GIVEN OBJECT IN A VIDEO

Truong Giang Vo
Subodh Mishra

1 Object to be Tracked

The figure to be tracked is the white bulb in the Figure 1.1, which is the first frame of the video provided for the lab.

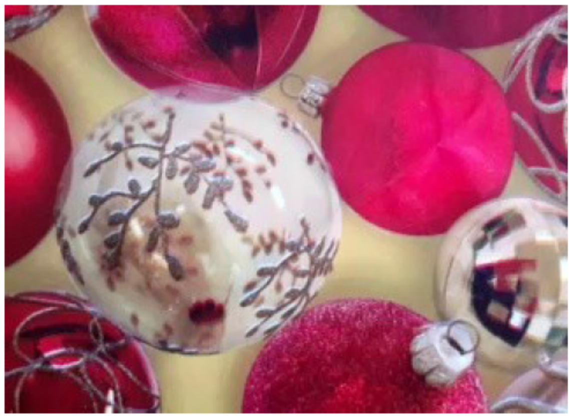

Figure 1.1: The object to be tracked

2 General Theory of Tracking

In this section, the general theory for tracking a feature is presented. For analysis we need two images with the same resolution. The first image is used as the base image and the object is tracked in the second image. The entire process can be broken down to 4 basic steps as follows:
1. Keypoint Detection using suitable detector
2. Calculate the Descriptors associated with each keypoint
3. Matching descriptor vectors using suitable matcher
4. Find homography between the two images to locate the required object in the second image.

2.1 Keypoint Detection

Keypoints are spatial locations, or points in the image that define what is interesting or what stand out in the image. The reason why keypoints are special is because no matter how the image changes, whether the image rotates, shrinks or is subject to distortion, one should be able to find the same keypoints in the image. Intuitively, it can be realized that textureless patches in the image are nearly impossible to localize. Patches with large contrast changes (gradients) are easier to localize, although straight line segments at a single orientation suffer from aperture problem, i.e. it is only possible to align the patches along the direction normal to the edge direction. Patches with gradients in atleast two significantly different orientations are the easiest to localize. The mathematics involved in keypoint detection is as follows:

The change in appearance of a window $w(x, y)$ for a shift $[u, v]$ is

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$

We are interested to see how this function behaves for very small shifts in the neighbourhood of $(u_o, v_0) = (0, 0)$. By taking the Taylor series approximation and ignoring the higher order terms we obtaing the following:

$$E(u,v) \approx \sum_{x,y} w(x,y)[I(x,y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v - I(x,y)]^2$$

$$\implies E(u,v) \approx \sum_{x,y} w(x,y)[(\frac{\partial I}{\partial x})^2 u^2 + 2\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}uv + (\frac{\partial I}{\partial y})^2 v^2]$$

$$\implies E(u,v) \approx \sum_{x,y} w(x,y) \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} (\frac{\partial I}{\partial x})^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & (\frac{\partial I}{\partial y})^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

$$\implies E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} A \begin{bmatrix} u \\ v \end{bmatrix}$$

Here, $A = \begin{bmatrix} \sum_{x,y} w(x,y) \begin{bmatrix} (\frac{\partial I}{\partial x})^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & (\frac{\partial I}{\partial y})^2 \end{bmatrix} \end{bmatrix}$

The eigen values of the 2×2 matrix A help us locate keypoints. For a textureless patch, both the eigen values are very small, for edges one of the eigen values is large and the other is very small and for corners both the eigen values are large.

There are many algorithms for keypoint detection, for e.g. SIFT detector, SURF detector, Harris detector, etc.

2.2 Descriptor Calculation

In most cases the local appearance of features will change in orientation and scale, and sometimes even undergo affine deformation. Extracting a local scale, orientation or affine frame estimate and then using this to resample the patch before forming the feature descriptor is thus usually preferable. Even after compensating for these changes, the local appearance of image patches will usually still vary from image to image. Image descriptors can be made more invariant to such changes and a lot of algorithms have been developed. Scale invariant feature transform (SIFT) is one of the most used algorithm for feature description. SIFT features are formed by computing the gradient at each pixel in a 16×16 window around the detected keypoint. In each 4×4 quadrant, a gradient orientation histogram is formed by adding weighted gradient value to one of the eight orientation histogram bins. So, each 4×4 quadrant has 8 bins, giving us a total $16 \times 8 = 128$ features for each keypoint.

2.3 Feature Matching

Once we have extracted features and their descriptors from two or more images, the next step is to establish some preliminary feature matches between these images. The FLANN (Fast Library for Approximate Nearest Neighbours) based feature descriptor matcher is very common. FLANN is a library for performing fast approximate nearest neighbour searches in high dimensional spaces. It contains a collection of algorithms that works best for nearest neighbour search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. FLANN is written in C++ and contains bindings for the following languages: C, MATLAB and Python.

2.4 Finding homography

Any two images of the same planar surface in space are related by a homography (assuming a pinhole camera model). Homography helps in computation of camera motion rotation and translation between two images. Once camera rotation and translation have been extracted from an estimated homography matrix, this information may be used to insert models of 3D objects into an image or video, so that they are rendered with the correct perspective and appear to have been part of the original scene. The mathematical explanation is as follows:
Since we are working in homogeneous coordinates, the relationship between two corresponding points x and $x^{'}$ can be re-written as :

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The perspective transform matrix H is a 3×3 matrix .

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix}$$

Dividing the first row of equation by third row and also dividing the second row of equation by the third row gives the following equations:

$$-h_1x - h_2y - h_3 + (h_7x + h_8y + h_9)u = 0$$
$$-h_4x - h_5y - h_6 + (h_7x + h_8y + h_9)v = 0$$

These equations can be written in matrix form as follows:

$$A_ih = 0$$

where, $A_i = \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & ux & uy & u \\ 0 & 0 & 0 & -x & -y & -1 & vx & vy & v \end{bmatrix}$

and $h = \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \end{bmatrix}^T$

Since each point correspondence provides 2 equations, 4 correspondences are sufficient to solve for the 8 degree of freedom of H. The restriction is no 3 points can be collinear. Four 2×9 A_i matrices (one per point correspondence) can be stacked one over the other to form a single 8×9 matrix A. Since this is a homogeneous system with less equations as compared to variables, to have a non trivial solution, the matrix A will have rank 8. This means out of the 9 unknown elements of h, one can be a free variable. Hence we can determine the other 8 easily by arbitrary choosing this free variable. In many cases we may be able to use more than 4 correspondences to ensure a more robust solution. However many point correspondences are used, if all of them are exact then A will still have rank 8 and there will be a single homogeneous solution. In practice, there will be some uncertainty, the points will be inexact and there will not be an exact solution. The problem then becomes to solve for a vector h that minimizes a suitable cost function.

3 OpenCv Instructions and Data Structures used

For this lab, the following data structures from OpenCv are used; `Mat`, `KeyPoint`,`DMatch` and `Point2f`. Besides these, the important functions used are `detect()`, `compute()`, `match()`,`drawMatches()`,`findHomography()` and `perspectiveTransform()`.

Data Structures

1. `Mat`: Mat is basically a class with two data parts: the matrix header (containing information such as the size of the matrix, the method used for storing, at which address is the matrix stored, and so on) and a pointer to the matrix containing the pixel values (taking any dimensionality depending on the method chosen for storing) . The matrix header size is constant, however the size of the matrix itself may vary from image to image and usually is larger by orders of magnitude.

2. `KeyPoint`: Data structure for salient point detectors. The class instance stores a keypoint, i.e. a point feature found by one of many available keypoint detectors, such as Harris corner detector. The keypoint is characterized by the 2D position, scale (proportional to the diameter of the neighborhood that needs to be taken into account), orientation and some other parameters. The keypoint neighborhood is then analyzed by another algorithm that builds a descriptor (usually represented as a feature vector). The keypoints representing the same object in different images can then be matched using some matching method.

3. `DMatch`:Class for matching keypoint descriptors: query descriptor index, train descriptor index, train image index, and distance between descriptors. Query index and train index say which descriptors have been matched and distance says how good the match is.

4. `Point2f`: Stores the coordinates of a 2D surface. In this example, this data structure stores keypoints that show some good level of matching between the two images.

Functions

1. `detect()`: Detects keypoints in an image (type `MAT`) and stores it in a variable of type `KeyPoint`.

2. `compute()`:Computes the descriptors for a set of keypoints detected in an image and stores it in a variable of type `MAT`.

3. `match()`: Finds the best match for each descriptor from a query set and stores it in a variable of type `DMatch`

4. `findHomography()`: Finds a perspective transformation between two planes and stores it in a variable of type `MAT`.

5. `perspectiveTransform()`: In the present example, this function takes three arguments, the first two are vectors of type `Point2f` and the last one is the perspective transformation matrix `H` of type `MAT`. If we write `perspectiveTransform(O1,O2,H)` where `O1` is a vector of known points in a 2D plane,`O2` is an empty vector of type `Point2f` and H is a transformation from second vector to the first vector,it signifies the relationship: `O2=H.O1`.

4 Tracking Cases

The SURF feature detector is used to track the white bulb in the figure for the following two cases:

- By matching the object in current frame to the object in the first frame of the video.
- By matching the object in current frame to the object in the previous frame of the video.

4.1 Matching current frame to first frame

The source code is written in the file named `3_LoadVideo.cpp`. The stepwise description of the process:

1. The 1^{st} frame of the video is grabbed by creating an object (let it be `capture`) of class `VideoCapture` and then invoking the function `grab()` through this object.

2. The grabbed frame is retrieved by invoking the function `retrieve()` and stored in a variable of type `MAT`. The function call is done by using the object `capture`.

3. The grabbed frame is converted to GRAYSCALE by using function `cvtColor()`.

4. A feature detector named `detector` of type `SurfFeatureDetector` is created.

5. Two vectors of type `KeyPoint` are created, `keypoints_1` for the 1^{st} frame and `keypoints_2` for the r^{th} frame, which will be grabbed and retrieved inside a loop.

6. The `detect()` function is invoked using the object `detector` created in step 4. Keypoint detection is done for the 1^{st} frame.

7. A feature descriptor named `extractor` of type `SurfDescriptorExtractor` is created.

8. Two variables of type `MAT` are created. The first is named `descriptors_1` and the second is `descriptors_2`. These are used for feature description of 1^{st} and r^{th} image respectively.

9. The `compute()` function is invoked using the object `extractor` created in step 7. The descriptors for the keypoints of 1^{st} image are obtained.

10. While Loop Started

11. The current frame of the video is grabbed, retrieved and converted to GRAYSCALE.

12. The keypoints are obtained for the current frame and stored in the variable `keypoints_2` created in step 5.

13. The descriptors are calculated for the keypoints obtained in previous step and stored in variable named `descriptors_2` created in step 8.

14. A `FlannBasedMatcher` object named `matcher` is created.

15. A vector of `DMatch` named `matches` is created.

16. The object `matcher` is used to invoke function `match()`. This function obtains the closest match to each descriptor in `descriptors_1` and stores its information in the variable `matches` created in step 15.

17. Out of all the matches present in the variable `matches` we chose the matches that satisfy certain criteria (in this example we chose only those matches where distance in between matched points is below certain value). These chosen matches are stored in a vector of type `DMatch` named `good_matches`.

18. We draw these matches using function `drawMatches()`.

19. Two vectors of type `Point2f` named `obj` and `scene` are created.

20. `obj` contains all the keypoints that are in the query index of `good_matches` and `scene` contains all the keypoints that are in the train index of `good_matches`.

21. The perspective transformation matrix `H` is calculated between `obj` and `scene` using the function `findHomography()`. This function can use different algorithms available in `OpenCv`. The default algorithm is the least squares algorithm, though one can choose robust methods like RANSAC or LMEDS.

22. After `H` is obtained, the corners of the rectangle in the current video frame is obtained by use of function `perspectiveTransfprm()`.

23. The rectangle is drawn in the current frame by using the function `line()`.

24. The function `imshow()` is used to display good matches and the detected object in the new frame.

25. End of while.

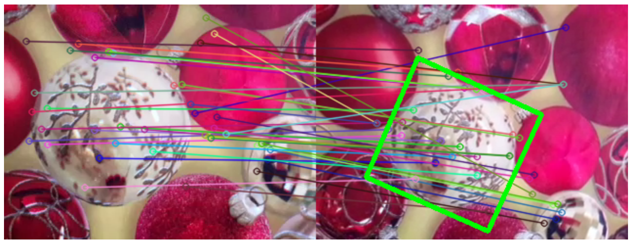

Figure 4.1: Matching current frame to first frame

4.2 Matching current frame to previous frame

A few changes should be made in the previous code to obtain this case. The source code for this case is `2_LoadVideo.cpp`. In this case, the current frame is compared with the previous frame (not the first frame). Therefore, at the end of the while loop, the current frame(say `frame`) is stored in a variable (say `frame0`) that contained the previous frame for the present iteration. In the next iteration the new frame (`frame`) obtained is compared to `frame0`. The corners of the rectangle in the `frame0` also changes over time. So, at the end of the loop, the new rectangle corners obtained by perspective transform must be stored and used as the rectangle corners of `frame0` in the next iteration.

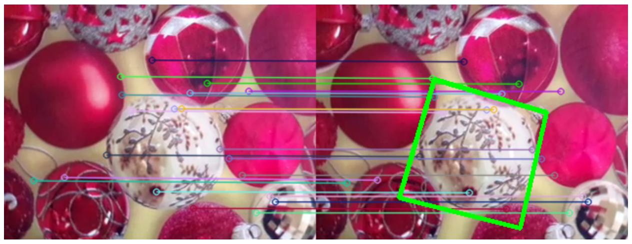

Figure 4.2: Matching current frame to previous frame

5 Executing the Code

The code for the first case can be executed by going to the ubuntu terminal and changing the directory to the build directory and typing `cmake ..` and pressing enter and then typing `make` and pressing enter . Then the code can be executed by typing `./Load_video ../../video1.mp4`.
The code for the second case can be executed by going to the ubuntu terminal and changing the directory to the build directory and typing `cmake ..` and pressing enter and then typing `make` and pressing enter . Then the code can be executed by typing `./Loadvideo ../../video1.mp4`.