Real-time doors and windows recognition in OpenCV using SURF for a guiding ROBOT

Object recognition – in computer vision, this is the task of finding a given object (door and windows in our case) in an image or video sequence. Humans recognize a multitude of objects in images with little effort, despite the fact that the image of the objects may vary somewhat in different viewpoints, in many different sizes / scale or even when they are translated or rotated. Objects can even be recognized when they are partially obstructed from view. This task is still a challenge for computer vision systems in general.

Feature-based methods

- A search is used to find feasible matches between object features and image features.
- The primary constraint is that a single position of the object must account for all of the feasible matches.
- Methods that extract features from the objects to be recognized and the images to be searched.
  - surface patches
  - corners
  - linear edges

Invariably all of them employ machine learning, because the computer has to first 'learn' that a particular bunch of pixels with particular properties is a door or window, remember that information, and use it in future to say whether the query image has a door or not.

Training images are the images which the detector uses to learn information. Query images are the images from which the detector, after learning, is supposed to detect object(s).

Generally, our aim in such experiments would be to achieve robust object recognition even when the object in a query image is at a different size or angle than the training images called scale in-variance and rotation in-variance respectively.

1. **Match Template:**

Template matching is a technique for finding areas of an image that match (are similar) to a template image (patch). To identify the matching area, we have to *compare* the template image against the source image by sliding it. By **sliding**, we mean moving the patch one pixel at a time (left to right, up to down). At each location, a metric is calculated so it represents how “good” or “bad” the match at that location is (or how similar the patch is to that particular area of the source image).
For each location of **T** over **I**, you *store* the metric in the *result matrix* **(R)**. Each location (x, y) in **R** contains the match metric.

Disadvantages: This approach consumes huge memory and is relatively slow.

2. **OpenCV objdetect module:**

The standard method in OpenCV to detect objects is to use its ready-made objdetect module, which uses the method of Haar cascade classifiers proposed by Viola and Jones. After training with thousands of positive and negative images for days on end, you can get a classifier file which contains all the training information. Load it into your program and you are ready to detect your object. This method is well-tested, works fast, and is scale invariant.
Disadvantages: Not rotation invariant and requires long training.

There is a method for object detection that is scale and rotation invariant, robust, fast and most importantly, can work with a single training image. We are going to discuss about this in detail and implement it for recognition of doors and windows.

3. **SURF** (Speeded Up Robust Features):

It is a robust local feature detector, first presented by Herbert Bay et al. in 2006, that can be used in computer vision tasks like object recognition or 3D reconstruction. It is partly inspired by the SIFT descriptor. The standard version of SURF is several times faster than SIFT and claimed by its authors to be more robust against different image transformations than SIFT. SURF is based on sums of 2D Haar wavelet responses and makes an efficient use of integral images.

This method

- Finds interest points in the image using Hessian matrices
- Determines the orientation of these points
- Uses basic Haar wavelets in a suitably oriented square region around the interest points to find intensity gradients in the X and Y directions. As the square region is divided into 16 squares for this, and each such sub-square yields 4 features, the SURF descriptor for every interest point is 64 dimensional.

It uses an integer approximation to the determinant of Hessian blob detector, which can be computed extremely quickly with an integral image (3 integer operations). For features, it uses the sum of the Haar wavelet response around the point of interest. Again, these can be computed with the aid of the integral image.

Integral Image

Much of the performance increase in SURF can be attributed to the use of an intermediate image representation known as the “Integral Image". The integral image is computed rapidly from an input image and is used to speed up the calculation of any upright rectangular area. Given an input image I and a point (x; y) the integral image I_P is calculated by the sum of the values between the point and the origin.

Fast-Hessian
Overview: Finds hessian based interest points/regions in a given integral image.
Inputs: An integral image representation of an image.

Processes: Builds determinant of Hessian response map. Performs a non-maximal suppression to localize interest points in a scale-space. Interpolates detected points to sub-pixel accuracy.
Outputs: A vector of accurately localized interest points.

The SURF detector is based on the determinant of the Hessian matrix. In order to motivate the use of the Hessian, we consider a continuous function of two variables such that the value of the function at (x; y) is given by f(x; y). The Hessian matrix, H, is the matrix of partial derivates of the function f.

$$H(f(x,y)) = \begin{vmatrix} \partial^2 f/\partial^2 x & \partial^2 f/\partial x \partial y \\ \partial^2 f/\partial x \partial y & \partial^2 f/\partial^2 y \end{vmatrix}$$

If the determinant is negative then the eigenvalues have divergent signs and hence the point is not a local extremum; if it is positive then either both eigenvalues are positive or both are negative and in either case the point is classified as an extremum. Working from this we can calculate the determinant of the Hessian for each pixel in the image and use the value to find interest points.

SURF Descriptor:
Overview: Extracts descriptor components for a given set of detected interest points.
Inputs: An integral image representation of an image, vector of interest points.
Processes: Calculates Haar wavelet responses. Calculates dominant orientation of an interest point. Extracts 64-dimensional descriptor vector based on sums of wavelet responses.
Outputs: A vector of `SURF described' interest points.

The SURF descriptor describes how the pixel intensities are distributed within a scale dependent neighborhood of each interest point detected by the Fast-Hessian. This approach is similar to that of SIFT [4] but integral images used in conjunction with filters known as Haar wavelets are used in order to increase robustness and decrease computation time. Haar wavelets are simple filters which can be used to find gradients in the x and y directions.

Extraction of the descriptor can be divided into two distinct tasks. First each interest point is assigned a reproducible orientation before a scale dependent window is constructed in which a 64-dimensional vector is extracted. It is important that all calculations for the descriptor are based on measurements relative to the detected scale in order to achieve scale invariant results.

Interest Point:
Overview: Stores data associated with each individual interest point.
Inputs: Interest Point data.
Processes: Accessor / Mutator Methods for data.
Outputs: None

Matching Strategy

A 64 dimensional descriptor for every key point sounds cool, but is actually useless without a method of deciding whether a query descriptor matches a training descriptor or not. For matching, we use the K nearest neighbour search. A KNN search basically computes the 'distance' between a query descriptor and all of the training descriptors, and returns the K pairs with lowest distance. Here we will keep K=2.

Hence we now have 2 pairs of matches for each query descriptor. So basically the KNN search gave us 2 matches to every query descriptor. What is important is how we decide which of all these matches are 'good matches' after all. The strategy we employ is (remember that each match has a 'distance' associated with it) -

For every query descriptor,
if distance(match1) < 0.8 * distance(match2), match1 is a good match
otherwise discard both match1 and match2 as false matches.

Real-time implementation

- Take an image of the door or window you want to detect and extract SURF descriptors for it. It is important for this image to contain only the object and to be free from any harsh lighting.
- Now do the same for every frame coming from your camera.
- Employ the matching strategy to match descriptors from every frame with the descriptors of the object and find out the 'good matches'.
- Create a window with the object image on one side and the video playing on the other side. Draw good matches between the two.
- To get a bounding box around a detected object, using the good matches, find a homography that transforms points from the object image to the video frame. Using this homography, transform the 4 corners of the object image. Consider these 4 transformed points as vertices and draw a box in the video frame.

Source Code:

```
#include <stdio.h>
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/nonfree/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/calib3d/calib3d.hpp"

using namespace cv;
using namespace std;

int main()
{
   Mat object = imread( "C:\\Users\\Kivvi\\Documents\\Visual Studio
2010\\Projects\\test7\\test7\\door.jpg", CV_LOAD_IMAGE_GRAYSCALE );

   if( !object.data )
   {
      std::cout<< "Error reading object " << std::endl;
      return -1;
   }

   //Detect the keypoints using SURF Detector
   int minHessian = 500;
   double tt = (double)cvGetTickCount();

   SurfFeatureDetector detector( minHessian );
   std::vector<KeyPoint> kp_object;
   detector.detect( object, kp_object );

   //Calculate descriptors (feature vectors)
   SurfDescriptorExtractor extractor;
   Mat des_object;
   extractor.compute( object, kp_object, des_object );
   FlannBasedMatcher matcher;
   VideoCapture cap(0);
   namedWindow("Good Matches");
   std::vector<Point2f> obj_corners(4);

   //Get the corners from the object
```

```
    obj_corners[0] = cvPoint(0,0);
    obj_corners[1] = cvPoint( object.cols, 0 );
    obj_corners[2] = cvPoint( object.cols, object.rows );
    obj_corners[3] = cvPoint( 0, object.rows );

    Mat frame;
    cap >> frame;
    Mat des_image, img_matches;

    std::vector<KeyPoint> kp_image;
    std::vector<vector<DMatch > > matches;
    std::vector<DMatch > good_matches;
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;
    std::vector<Point2f> scene_corners(4);
    Mat H;
    Mat image;

    cvtColor(frame, image, CV_RGB2GRAY);

    detector.detect( image, kp_image );
    extractor.compute( image, kp_image, des_image );

    matcher.knnMatch(des_object, des_image, matches, 2);
          tt = (double)cvGetTickCount() - tt;
          printf( "Extraction time = %gms\n", tt/(cvGetTickFrequency()*1000.));

    for(int i = 0; i < min(des_image.rows-1,(int) matches.size()); i++)
    {
       if((matches[i][0].distance < 0.6*(matches[i][1].distance)) && ((int)
matches[i].size()<=2 && (int) matches[i].size()>0))
       {
           good_matches.push_back(matches[i][0]);
       }
    }

    // Draw only "good" matches


drawMatches( object, kp_object, image, kp_image, good_matches, img_matches,
Scalar::all(-1), Scalar::all(-1), vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );
```

```
    if (good_matches.size() >= 4)
    {
        cout << "Object Found" << endl;
        for( int i = 0; i < good_matches.size(); i++ )
        {
            //Get the keypoints from the good matches
            obj.push_back( kp_object[ good_matches[i].queryIdx ].pt );
            scene.push_back( kp_image[ good_matches[i].trainIdx ].pt );
        }

        H = findHomography( obj, scene, CV_RANSAC );

        perspectiveTransform( obj_corners, scene_corners, H);

        //Draw lines between the corners (the mapped object in the scene image )
        line( img_matches, scene_corners[0] + Point2f( object.cols, 0), scene_corners[1] +
Point2f( object.cols, 0), Scalar(0, 255, 0), 4 );
        line( img_matches, scene_corners[1] + Point2f( object.cols, 0), scene_corners[2] +
Point2f( object.cols, 0), Scalar( 0, 255, 0), 4 );
        line( img_matches, scene_corners[2] + Point2f( object.cols, 0), scene_corners[3] +
Point2f( object.cols, 0), Scalar( 0, 255, 0), 4 );
        line( img_matches, scene_corners[3] + Point2f( object.cols, 0), scene_corners[0] +
Point2f( object.cols, 0), Scalar( 0, 255, 0), 4 );
    }
    else
          cout<< "Object Not Found" << endl;
    //Show detected matches
    imshow( "Good Matches", img_matches );
    getchar();

    return 0;
}
```

References:

- OpenCV Tutorials: http://dasl.mem.drexel.edu/~noahKuntz/openCVTut6.html http://docs.opencv.org/trunk/modules/nonfree/doc/feature_detection.html#surf
- Open SURF documentations: http://www.chrisevansdev.com/computer-vision-opensurf.html
- First publication of Speeded Up Robust Features (2006): http://www.vision.ee.ethz.ch/~surf/eccv06.pdf
- Website of SURF: Speeded Up Robust Features: http://www.vision.ee.ethz.ch/~surf

Output:

Recognition of Vending machine for Good matches.
Input query image is displayed in the left side of the Good matches window.
The video frame captured is displayed on the right side with detected key points and markings around the detected image.

For now, if a match is found, we are displaying “Object Found” and the extraction time (in ns) in the output window. But this can be used as input for the guiding robot to make decisions.