Workshops in Creative Computing
Computer Vision
Lab 2: Image Features and Planar Object Detection

Parag K Mital

February 26, 2013

Introduction

By now, you will have had 1 week to explore using `ofxOpenCV`, an addon for openFrameworks wrapping some of OpenCV's basic functionalities, and also had an introduction to using OpenCV directly by getting the pointer to the OpenCV object directly from the ofxOpenCV object (e.g.: `IplImage * myCvImage = myOfxCvImage.getCvImage()`). If you need a reminder, check the labsheet from week 1. This week, we will explore some of the more advanced OpenCV functionality surrounding image features.

Part 1 will be in building the base of this system: detecting and visualizing features. Features are a very basic detail of a visual scene that should be repeatably and robustly detected. As well, they are able to take an image comprised of 320x240x3 pixels, and reduce it to a much smaller amount of data that describes only the "rich" aspects of a scene, i.e. features.

Feature detection provides the building blocks of many much more advanced algorithms.

- Finding particular objects (Object Tracking/Detection)
- Learning the 3D geometry of a scene (Structure from Motion)
- Detecting motion (Sparse Optical Flow)
- Automatically aligning images into a panorama (Stitching/Montage)
- Sourcing image databases to create rich 3D maps of the world (Photomontage)

Part 2 will be in exploring just the first of these possibilities, object detection. We will use an existing library that uses this base system for the purpose of storing a set of those features selected by the user, and detecting any new set of features to the stored set using a powerful library for indexing and matching large databases of features: fast library for approximate nearest neighbors (Muja and Lowe, 2009). We will not be making full use of this library, but its power is in high-numbers, as this library is developed for quickly finding a single instance not from just 1 frame of a video, but from millions of images. This is a classic problem in Image Retrieval, and it is the same problem that Google tries to solve in its Image Search or in Google Goggles: how do I quickly find the nearest image from a library of millions to an input example image? We will work with a system that solves a good portion of that problem.

Part 1

Use either a video or camera input and display the original image. Next, copy the pixels from the camera image into an `ofxCvColorImage`, convert this to `HSV` space, and then grab the "`V`" channel into a `ofxCvGrayscaleImage` (we did all of this last week). We are going to display all the features detected from this image.

First we need a feature detector to find the features in our image. We can find information about OpenCV's feature detectors here: `http://opencv.willowgarage.com/documentation/cpp/common_interfaces_for_feature_detection_and_descriptor_extraction.html?highlight=featuredetector`. OpenCV has a number of feature detectors (about 20), so rather than creating a class for every kind of feature detector, and

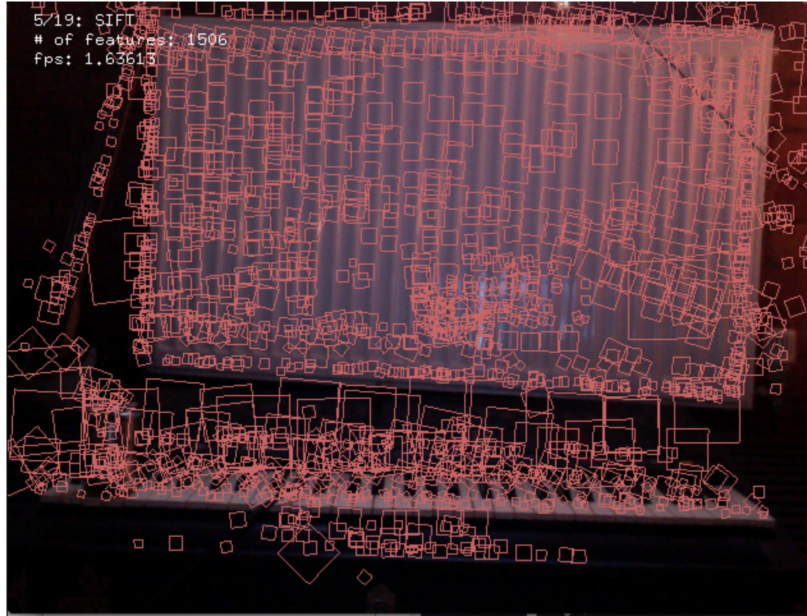

Figure 1: Detecting and displaying figures using SIFT

having the user remember the class names of each of these feature detectors, OpenCV provides an "Abstract Class", also called the "Base Class". We mentioned the idea of "Base Classes" when we looked at the source code for "`ofxCvImage`". `ofxCvImage` is the base class for `ofxCvColorImage` and `ofxCvGrayscaleImage`. Base classes declare functions which any derived class (e.g. `ofxCvColorImage`) have to implement (e.g. `allocate(int width, int height)`). This is known as Inheritance in C++. One way to describe it is, "derived class is a type of base class", e.g.: `ofxCvColorImage` is a type of `ofxCvImage`.

Since `FeatureDetector` is an Abstract class, we cannot create an instance of it. Though, when we want our program to have all the functionality of the possible derived classes, we need to use the Abstract class. So how do we use the functionality of an Abstract class if we don't have an instance to it? We use a pointer to it. Luckily, we don't have to even think about it when we use OpenCV's Pointer class: `Ptr`. This class does all the allocation/deallocation for us by keeping track of anyone that is using the object. To create a pointer to the `FeatureDetector` then, we just say: `cv::Ptr<FeatureDetector> feature_detector`.

Here is the header file describing the base class `FeatureDetector`:

```
/*
 * Abstract base class for 2D image feature detectors.
 */
class CV_EXPORTS FeatureDetector
{
public:
    virtual ~FeatureDetector();

    /*
     * Detect keypoints in an image.
     * image        The image.
     * keypoints    The detected keypoints.
     * mask         Mask specifying where to look for keypoints (optional). Must be a char
     *              matrix with non-zero values in the region of interest.
     */
    void detect( const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask=Mat() )
        const;

    /*
     * Detect keypoints in an image set.
     * images       Image collection.
```

```
21       * keypoints     Collection of keypoints detected in an input images. keypoints[i] is a
             set of keypoints detected in an images[i].
22       * masks         Masks for image set. masks[i] is a mask for images[i].
23       */
24      void detect( const vector<Mat>& images, vector<vector<KeyPoint> >& keypoints, const
            vector<Mat>& masks=vector<Mat>() ) const;
25
26      // Read detector object from a file node.
27      virtual void read( const FileNode& );
28      // Read detector object from a file node.
29      virtual void write( FileStorage& ) const;
30
31      // Create feature detector by detector name.
32      static Ptr<FeatureDetector> create( const string& detectorType );
33
34  protected:
35    virtual void detectImpl( const Mat& image, vector<KeyPoint>& keypoints, const Mat& mask=
          Mat() ) const = 0;
36      /*
37       * Remove keypoints that are not in the mask.
38       * Helper function, useful when wrapping a library call for keypoint detection that
39       * does not support a mask argument.
40       */
41      static void removeInvalidPoints( const Mat& mask, vector<KeyPoint>& keypoints );
42  };
```

In our `setup()` function, we can instantiate the pointer by using the function `static Ptr<FeatureDetector> create( const string& detectorType );`. This function requires a string as an argument, which will define which Feature Detector gets instantiated. The documentation for which feature detectors OpenCV implements isn't the best, so I will list the strings you should try: `SURF`, `SIFT`, `STAR`, `FAST`, `GFTT`, `MSER`, and `HARRIS` (there are more but you can find out on your own).

Knowing which strings we can use, the way we call this in our code is by writing: `feature_detector = FeatureDetector::create(''SIFT'');` if we wanted to try SIFT.

We are also going to need storage for all of our features, which OpenCV calls `KeyPoint`. You can look up the documentation of this structure, or view the source code for it to get a better idea of what it contains. In the code fragment above, we see the `FeatureDetector` class has a method called `detect`. It takes as the first argument, an image of type `Mat`, and a `vector` of `KeyPoint`, where it will store the resulting features it detects. Use this method in your `update()` to find the features in the image.

When you have found your keypoints, loop through the `vector` and display each `KeyPoint` on top of the original color image using `ofRect(...)`. The relevant parts of the header file for the `KeyPoint` class are copied below:

```
1  class KeyPoint
2  {
3  public:
4
5      ...
6
7      Point2f pt; // coordinates of the keypoints
8      float size; // diameter of the meaningfull keypoint neighborhood
9      float angle; // computed orientation of the keypoint (-1 if not applicable)
10
11     ...
12
13 };
```

We can see that each `KeyPoint` has a `Point2f` object called `pt`, with fields `x` and `y` (i.e. we can find the x,y location of every `KeyPoint` using `pt.x` and `pt.y`). Further, each `KeyPoint` also has an `angle` and `size`.

Part 2

Make sure you have downloaded the additional files required for this lab from the website. The zip file include a header and source file for a class called `pkmImageFeatureDetector` which wraps a lot of the functionality necessary for detecting features, describing them, approximately matching them, and describing a homography between them. Homographies are very useful concepts in Computer Vision that describe an affine transformation. We will use a homography in order to detect the position and rotation of an object.

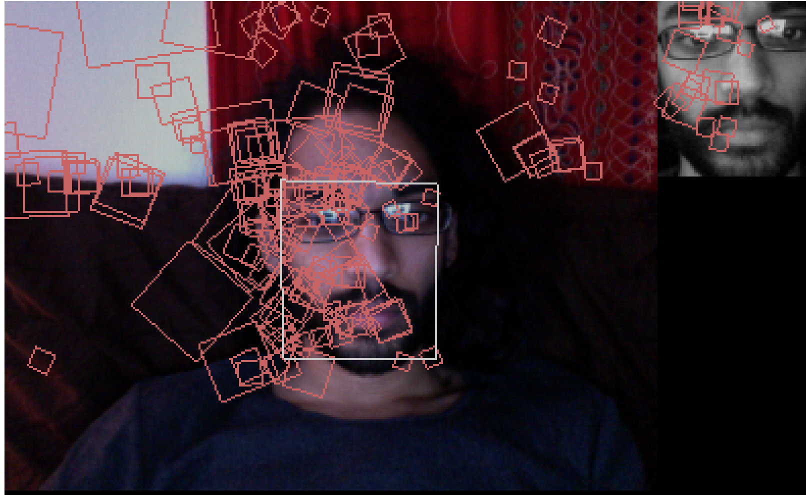

Figure 2: Detection of an object

To begin, we need to describe an object that we can proceed to detect. We start by collecting the pixels, converting to those pixels to luminance, and then finding the features which describe those luminance pixels. I have done this by allowing the user to draw a box on the camera image, and storing the subimage. Note that the subimage should be a grayscale image, ideally of the luminance in the image. You can store a subimage using "Regions of Interest" (ROI). This defines an area on an image, for which any further operations on that image will operate. We set a ROI, copy the image, and reset the ROI. Now the copied image is not the entire image, but only the ROI.

```
1        // set the region of interest on the color image
2        color_img.setROI(roi_x, roi_y, roi_width, roi_height);
3
4        // allocate images which will store the color and luminance images of the ROI
5        color_roi_img.allocate(roi_width, roi_height);
6        gray_roi_img.allocate(roi_width, roi_height);
7
8        // get the ROI
9        color_roi_img = color_img;
10
11       // convert to HSV
12       color_roi_img.convertRgbToHsv();
13
14       // get the luminance
15       color_roi_img.convertToGrayscalePlanarImage(gray_roi_img, 2);
16
17       // reset the ROI to the full image
18       color_img.resetROI();
```

It is up to you how you obtain the ROI, and I leave it as an open task that you can work in groups in to figure out. Once you have a luminance sub-image, you are ready to start using the provided `pkmImageFeatureDetector` class.

Create an object of type `pkmImageFeatureDetector` (e.g. `detector`) in your `testApp` header file. Once you get the ROI, you can set the template image, that is, the object to search for, using `detector.setTemplateImage(...)`.

On every iteration of an incoming camera frame, search for the template image by using `detector.setSearchImage(...)` and following it, calling `detector.update()`.

On draw, you can see the result of the homography by accessing the variable `detector.dst_corners`, which is a 4 element array of `Point2f`, and contains fields `x` and `y`. This array defines the 4 points of the resulting found object. Using these 4 points, draw a bounding box on top of the search image.