Workshops in Creative Computing 2
Computer Vision Module
Lab 1

Parag K Mital

February 20, 2013

Introduction

This lab will orient you with using some features of openFrameworks' ofx-OpenCv addon, which builds wrappers for OpenCV, and also get you started with using OpenCV itself using some of its most basic functions. OpenCV is the industry standard library for OpenCV applications and is developed by Intel and supported by WillowGarage. As it is free for use under the open-source BSD license, you can use it for research and commercial use. The OpenCV project started in 1999, and is still in active development today, with updates daily. It is a MASSIVE library, and has tons of libraries dedicated to:

- 2D and 3D feature toolkits
- Egomotion estimation
- Facial recognition system
- Gesture recognition
- Humancomputer interaction (HCI)
- Mobile robotics
- Motion understanding
- Object identification
- Segmentation and Recognition
- Stereopsis Stereo vision: depth perception from 2 cameras
- Structure from motion (SFM)
- Motion tracking
- Boosting
- Decision tree learning
- Gradient boosting trees
- Expectation-maximization algorithm
- k-nearest neighbor algorithm
- Naive Bayes classifier
- Artificial neural networks

- Random forest
- Support vector machine (SVM)
- GPU enabled functions
- GUI Building
- Preliminary Android and iPhone support
- Integrates with Point Cloud Library (PCL), and the Robotics Operating System (ROS)

and lots and lots more.

Some links you will very likely need:

- OpenCV 2.1+ Documentation: `http://opencv.willowgarage.com/documentation/cpp/`[1].
- General OpenCV knowledge can be found here: `http://opencv.willowgarage.com`
- OpenCV Wiki is located here: `http://opencv.willowgarage.com/wiki/FullOpenCVWiki`.

Part 1 - Luminance and Colorspaces

The first part will get you familiar with some of the ofxOpenCv objects. These are very nicely written "wrappers" of some commonly used OpenCV functions (OpenCV is a library included by openFrameworks). As well, they interface with openFramework's ofTexture class, allowing you to draw the OpenCV images using OpenGL (another library included by openFrameworks). We'll investigate some ways of changing the image's colorspaces using the ofxCv classes and also use OpenGL blending.

To begin, copy the openFrameworks OpenCV example project's entire directory to a new directory at the same hierarchy. The example project's folder "opencvExample" is 3 levels above the main openFrameworks directory. Make sure when you copy the "opencvExample" directory to another location, it is still 3 levels above the main openFrameworks directory. This ensures all the project settings still make sense (as they are include files relative to it's own current path).

Once you have copied the example directory to a new location, open the project (either xcode, visual studio, codeblocks, emacs, etc...) and edit `testApp.h` to look like:

```
#pragma once

#include "ofMain.h"
#include "ofxCvMain.h"

// this allows us to use the new OpenCV API
```

[1]If you are an advanced user and want to use the newest OpenCV's API, look here: `http://opencv.itseez.com/` (Make sure you ignore the Python documentation)

```
// Make sure you add this line if you are going to use the new API
    !!!
using namespace cv;

class testApp : public ofBaseApp{

public:

    // redeclaration of functions (declared in base class)
    void setup();
    void update();
    void draw();

    void keyPressed(int key);

};
```

and `testApp.cpp` to look like:

```
#include "testApp.h"

// here we "define" the methods we "declared" in the "testApp.h"
    file

// i get called once
void testApp::setup(){

}

// i get called in a loop that runs until the program ends
void testApp::update(){

}

// i also get called in a loop that runs until the program ends
void testApp::draw(){

}

// i am a callback that gets called whenever the keyboard is
    pressed
void testApp::keyPressed(int key){

}
```

This is a basic openFrameworks project which includes all of the OpenCV and ofxOpenCv header files. We will work with this template in the following lab exercises. It is recommended that you keep this blank template, and copy the entire project to a new directory (keeping the same directory hierarchy) when you are working on a new OpenCV-based project.

With a fresh project loaded, load either a Quicktime movie file or live web-camera and display the output (if you are unsure of how, look at the code from the examples that come with openFrameworks).

1. In your header file, create both a ofxGrayscaleImage and an ofxCvColorImage. If you need more, make sure you declare them in your header

file. In your setup() function, make sure you allocate the images you've declared to be the same size as your movie/camera.

2. Using '=', you can convert an object of type ofxCvColorImage to an object of type ofxCvGrayscaleImage. In your update() function, convert the incoming image to a grayscale image. In your draw() function, display both the original color image and next to it the converted grayscale image. (hint: use the classes ofxCvColorImage and ofxCvGrayscaleImage, and the member functions setFromPixels(...), '=', and draw(...)); you can also change the openFrameworks window output size using ofSetWindowShape(...)).

3. Create three more ofxCvGrayscale objects. Get the R, G, and B color channels of the original color image and use these to draw the contents of the red, green, and blue channels as grayscale images (hint: look for a function in the ofxCvColorImage header file which converts a color image into three planar images)

4. Use ofSetColor(...) before drawing each image to display the 3 grayscale images you've just created as red, green, and blue tinted images, respectively.

5. You can blend images with OpenGL using openFrameworks. An example codefragment for blending images is shown below:

```
        ofEnableAlphaBlending();
        ofEnableBlendMode(OF_BLENDMODE_ADD);

        image1.draw(0,0);
        image2.draw(0,0);
        image3.draw(0,0);

        ofDisableAlphaBlending();
```

   Use blending to blend the three red, green, and blue images.

6. Convert the ofxCvColorImage from the RGB to Hue-Saturation-Value (HSV) colorspace (hint: look for a function in the ofxCvColorImage header file). Display each of the H, S, and V channels as a grayscale image. Then blend the H, S, and V images together and notice how it differs from blending R, G, and B images. Why doesn't it work like RGB?

7. Compare just the 'V' (Value) image from your HSV conversion to the grayscale image you created using '=' (in the beginning of the lab). What is different about these two images?

Part 2 - Motion Detection

Having some basics of how to start an OpenCV based openFrameworks project and work with ofxCv* objects, we will now work with storing images and finding differences between them for a simple measure of motion and foreground/background. The same algorithm will let us either detect changes between successive frames, or find the difference between a stored "background" image and the current image to find "foreground" objects. As we are going to use some more advanced techniques, we will begin using the OpenCV API directly, and not the openFrameworks wrapper ofxCv objects. It is very painful at first, and requires a lot of practice. Take it one step at a time, and breathe.

Start with a new templated project (copying the one you created in the beginning of Part 1).

1. Begin by getting either a grayscale or "Value" image of an incoming movie or web-camera, and displaying it (you've already done this in Part 1).

2. Copy this image into another grayscale object and calculate and display the absolute difference of the previous frame's image (using absDiff(...)). This means you'll need another grayscale image for storing the previous frame's grayscale image. Pseudo-code for doing this is below:

```
  // set the color image (opencv container) to the camera image

  // convert to grayscale

  // compute the absolute difference with the previous frame's grayscale image

    // store the current image for the next iteration (which becomes the previous image)
```

3. We are now going to find the average pixel value of the difference image we just made as a simple measure of the magnitude of "motion". To help us, we will use a function to sum every pixel in an image which is part of OpenCV, but unfortunately not wrapped by ofxCv. In order to use any of OpenCV functions directly, we need to use obtain the OpenCV image container: IplImage *cvImage, which is part of every ofxCv object. To get this image pointer, we use the function "getCvImage()", which returns an IplImage *. e.g.:

```
IplImage *myCvImage = myOfxCvGrayscaleImage.getCvImage();
```

Now we can use the IplImage pointer with any of the functions defined in OpenCV's massive library version 2.1+. Since we will use the new OpenCV functions, included in 2.1+, you will have to use a "Mat" container like so: IplImage is still supported by the new OpenCV, though was abandoned for C++ style objects in OpenCV 2.1+. You can convert a IplImage to the new Mat format very simply using:

```
Mat myOpenCvMat(myCvImage);
```

Unfortunately, openFrameworks v0.07 doesn't wrap any of the new API, but it does include it's libraries allowing you to use it. Remember to add `using namespace std;` after including ofxCvMain.h, if you are using the new API.

You will want to find a function which takes either the mean or sum, in order to find the average of all pixels in the grayscale image. The way I would go about it is to go to the new OpenCV's API reference page, and search for "mean". The following reference comes up: `http://opencv.willowgarage.com/documentation/cpp/core_operations_on_arrays.html?highlight=mean#mean`. Using the new API, the function "mean(...)" returns a Scalar. You can search for Scalar on the API reference page to find out how it works, but basically it stores 4-values, where each value is indexed by the plane it operates on. A 3-channel image has 3 planes, 1 for each channel, e.g. R, G, B. A grayscale image has 1 channel. When you call:

```
Scalar myScalar = mean(myOpenCvMat);
```

you can access any of the 4 channels simply by using square brackets:

```
double motion = myScalar[0];
```

Thus for our single-channel grayscale image encoding the difference between two frames, we have found a mean value that we will interpret as the motion in the frame.

4. Display the motion value as text in the window, or print it to the console. If you want to have a bit more fun, use this value as a "signal", and use it to control something interesting such as the speed of a video, the frequency of a synthesizer, or the speed of a sound sample. For instance, have a look at ofVideoPlayer's or ofSoundPlayer's setSpeed(...) function.

   More advanced models of motion look at the motion at feature points and can classify activity using trained models of the motion patterns. Dense optical flow calculates the motion at every pixel of an image. The first optical flow algorithms appeared in 1981, 31 years ago, forming the problem of detecting motion as an image alignment problem. If you are interested in playing with motion more, search the OpenCV documentation for "calcOpticalFlowPyrLK".

Part 3 - Interactive Colorspaces & Background Subtraction

(a) Use the motion signal to offset the 3 R,G,B images you displayed in Part 1. How you do this is up to you. One way would look like a full color image when there is no motion, and R, G, B planes separated more relative to higher motion.

(b) Building on Part 2, use the keyboard callback to control when a frame is stored as a background image instead of storing it every update. For instance, when a user presses the spacebar, the "previousImage" gets updated. Otherwise, it doesn't update the previousImage. The ability to select when the image gets stored allows you to interpret that frame as "background". The idea is to take a "snapshot" of a live camera image when no one is in the scene, storing that image as background. When someone enters, the difference between the stored image, and the current live image can be used to define blobs representing the people in the scene.