Using CameraEnhance API for SuperResolution

by Neil Del Gallego

About this document

This document is a guide on how to use the CameraEnhance API provided for developers. It contains many useful tools and classes that makes it easy for developers to easily use image processing techniques to generate a processed image for Android.

Prerequisites

It is assumed that you have properly setup the CameraEnhance project using Eclipse and has properly imported the OpenCV library. You have also successfuly build the project into an APK file and has run it successfully on to an Android device for the first time.

Setting up the Camera

The application automatically starts the camera for you. Tapping the camera icon on the top-left requests for your device front-camera, then back. Should you need to manually configure the camera, there is the **CameraManager** class. It is a singleton class that can be accessed using the *getInstance()* method.

Here are the following methods of importance:

- `requestCamera(): returns android.Camera`
  - `Requests for the device camera. The back-camera of the device is returned by default. Returns an android.Camera instance.`

- `requestFrontCamera(): returns android.Camera`
  - `Requests for the device front camera (if existing). This will throw an Exception if there is no front camera on the device. Returns an android.Camera instance.`

- `requestBackCamera(): returns android.Camera`
  - `Requests for the device back camera. Returns an android.Camera instance.`

- `swapCamera(): returns android.Camera`
  - `Swaps between the front and the device back camera. This will thrown an exception if the front camera is not existing. Returns an android.Camera instance.`

- `getActualCameraSize(): returns android.Camera.Size`
  - `Returns the actual camera size set. Note that this size refers to the size of the original.jpg output.`

- `getShutterSize(): returns android.Camera.Size`
  - `Returns the shutter size. Note that this size refers to the size of n.jpg output where n is the frame sequence.`

Configuring camera settings

You can configure the following parameters for the camera. As of this writing, the following can be configured:

- `shutterDelay – the delay between capture of frame sequences in milliseconds.`
- `imageLimit – the number of frame sequences to capture.`
- `defaultWidth – the specified width of the camera.`
- `defaultHeight – the specified height of the camera.`

To configure the settings of your camera accordingly on your program, do the following steps.

1. Create a new class that inherits *BaseConfig* class. Override the function *configure()* and put the the required values as seen below.

```
public class MultipleImageConfig extends BaseConfig {

        @Override
        protected void configure() {
        this.imageLimit = 10;
        this.shutterDelay = 100;

        this.defaultWidth = 300;
        this.defaultHeight = 300;
        }

}
```

2. Replace the *BaseConfig* instance inside the *ConfigHandler* constructor. This will load the values you specified above.

```
private ConfigHandler() {
        //change the configuration here
        this.assignedConfig = new MultipleImageConfig();

        Log.d(TAG, "Initialized config. Image limit:
"+this.assignedConfig.getImageLimit()+ " Shutter delay:"
+this.assignedConfig.getShutterDelay() + "Width: "
+this.assignedConfig.getCameraWidth()
                         + " Height: " +this.assignedConfig.getCameraHeight());
        }
```

Save location of images

Images are automatically processed and saved once you tap the capture button in the application. Images are normally stored on your device external storage but this depends on how your Android device is configured. Normally it is stored in your PICTURES external directory with a **DIGIMAP_N** folder, where N represents the current album number of your application**.**

On stock Android devices, the location is in ***/storage/sdcard0/Pictures/DIGIMAP_N.*** Opening that folder should contain the following images ready for processing:

- original.jpg – This is the original image captured by the camera and the very first image before the frame sequences.
- 0.jpg, 1.jpg,...,n.jpg – This represents your frame sequences right after the original.jpg.
- processed.jpg – This is the final output image after all image processing is done. This will only appear if you actually did some image processing work. More on this later.

Loading images into memory

Images can be loaded using a utility class called *ImageDataStorage.* This is a singleton class that can be accessed using *getInstance().* Methods contained herein already has javadoc comments that defines how it should be used.

ImageDataStorage loads your images upon request and stores it into memory as byte[] or opencv.Mat form. In this document, only the opencv.Mat methods are discussed as this is most commonly used for openCV image processing.

- `loadMatFormOfOriginalImage(): returns opencv.Mat`
  - `Loads the original image and creates an opencv.Mat representation. This method assures that the image is only loaded once into memory and reused by succeeding calls.`
- `loadMatOfImageSequence(int sequenceNum): returns opencv.Mat`
  - `Loads the specified frame sequence and creates an opencv.Mat representation. This method assures that the image is only loaded once into memory and reused`

by succeeding calls.

- releaseMatOfOriginalImage(): void
  - Releases allocation of the original image to free up memory. Thus, calling loadMatFormOfOriginalImage() again after this method would fetch it from the storage.
- releaseMatOfImageSequence(int sequenceNum): void
  - Releases allocation of the frame sequence. Calling loadMatOfImageSequence(int sequenceNum) after this method would fetch the frame sequence from storage again.
- releaseAll(): void
  - Attempts to release all allocation used. Call this when you no longer need any more images for processing. Calling any load methods would fetch it from storage again.

Saving of Images

It is completely up to you when you want to save the image (this includes saving the final output as well!). It is also sometimes necessary to save images during processing to serve as checkpoints and to debug if operations actually went well for the images. To save images, use the *ImageSaver* class with the following methods:

- encodeAndSaveAsProcessed(Mat matrixToSave): void
  - Saves the specified opencv.Mat and encodes it as **processed.jpg**. Call this method for the final matrix you consider as the final output.
- encodeAndSave(Mat matrixToSave, String fileName)
  - Saves the specified opencv.Mat and encodes it with the specified filename in .JPG. Call this method for specific matrices you want to see as images.

Implementing your own image processing technique

Before you attempt to create your own algorithm to process images, you must understand the API behind it. There is the *IImageProcessor* interface that serves as a template for all image processing operations. Here are the following methods involved.

- Preprocess() - function that handles preprocessing of images such as removal of noise, enhancing quality, etc. Put pre-processing operations here and initialization such as allocation of opencv.Mat.
- Process() - function that handles actual processing of images. Put your super-resolution operations and other heavy loads here in this function.
- Postprocess() - function that handles post processing of images. Put final touches here and do not forget to include the save method for your image here.

The following methods are event functions called on start and on finish of major steps above. This can be useful for UI updates or prompts to the user.

- onPreProcessStarted()
- onPreProcessFinished()
- onProcessStarted()
- onProcessFinished()
- onPostProcessStarted()
- onPostProcessFinished()

Refer to Figure 1 for the image processing lifecycle. The *IimageProcessor* methods are executed on a separate thread.

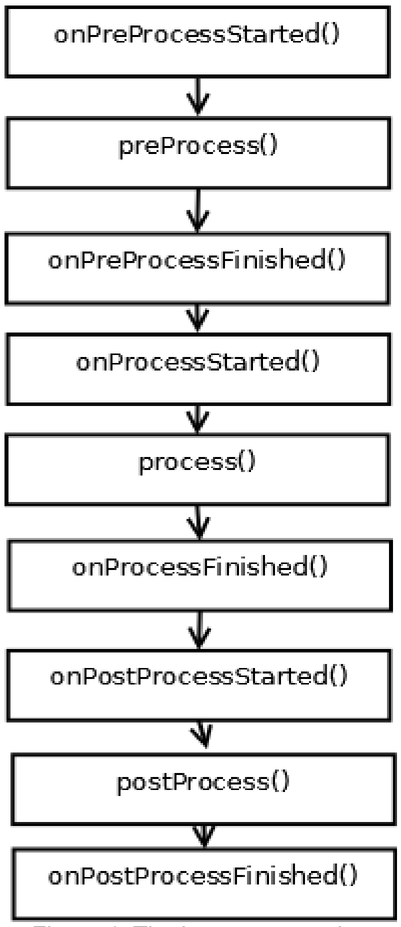

Figure 1: The image processing lifecycle

To implement your own image processing technique, provided you have understood the lifecycle above, do the following:

1. Create a class that implements *IimageProcessor* like the code snippet. Fill-in your required

```
public class ImageProcessor01 implements IImageProcessor {
        private final static String TAG = "CameraEnhance_ImageProcessor01";

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#Preprocess()
         */
        @Override
        public void Preprocess() {
        Log.d(TAG, "Preprocessing");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#Process()
         */
        @Override
        public void Process() {
        Log.d(TAG, "Processing");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#PostProcess()
         */
        @Override
        public void PostProcess() {
        Log.d(TAG, "Postprocessing");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onPreProcessStarted()
         */
        @Override
        public void onPreProcessStarted() {
        Log.d(TAG, "Pre process started");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onPreProcessFinished()
         */
        @Override
        public void onPreProcessFinished() {
        Log.d(TAG, "Pre process finished");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onProcessStarted()
         */
        @Override
        public void onProcessStarted() {
        Log.d(TAG, "Process started");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onProcessFinished()
         */
        @Override
        public void onProcessFinished() {
        Log.d(TAG, "Process finished");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onPostProcessStarted()
         */
        @Override
        public void onPostProcessStarted() {
        Log.d(TAG, "Post process started");
        }

        /* (non-Javadoc)
         * @see com.neildg.cameraenhance.processing.IImageProcessor#onPostProcessFinished()
         */
        @Override
        public void onPostProcessFinished() {
        Log.d(TAG, "Post process finished");
        }
```

operations.

2. In the *ProcessorDispatcher* class constructor, attach your image processor task.

```
private ProcessorDispatcher() {
        NotificationCenter.getInstance().addObserver(Notifications.ON_IMAGE_PROCESSING_STARTED, this);
        NotificationCenter.getInstance().addObserver(Notifications.ON_IMAGE_PROCESSING_FINISHED, this);

        //IMPORTANT: define image processing task here.
        this.attachImageProcessor(new ImageProcessor01());
        //this.attachImageProcessor(new IterativeUpSampleProcessor());
        }
```

Using NotificationCenter for decoupling code

The class *NotificationCenter* is a singleton instance that is very useful for keeping your classes decoupled. This is based on the observer pattern wherein certain functions of your class (the *observer)* "listens" to events and performs a behavior once the event that it is listening to is thrown by the *notifier* class. To summarize, the following steps are performed.

1. Class A is marked as an observer, listening to an event ON_FINISHED.
2. Class A instance is added to the *NotificationCenter.*
3. Class B is instantiated.
4. Class B performs *doSomething().*
5. Class B tells *NotificationCenter* to fire the event ON_FINISHED.
6. Class A receives the event and does something with it.

To make your classes ready for these kind of setup, follow the steps below:

1. The class that should be an observer should implement the *NotificationListener* interface.

2. Add specific methods that listens to specific events. Define your event name as string in the *Notifications* class. Retrieve needed data from the *Parameters* instance (more on this later).

```
public class ProcessorDispatcher implements NotificationListener {
   public void onNotify(String notificationString, Parameters params) {
        if(notificationString == Notifications.ON_IMAGE_PROCESSING_STARTED) {
                this.startProcessing();
        }
        else if(notificationString == Notifications.ON_IMAGE_PROCESSING_FINISHED) {
                this.stopProcessing();
        }
   }
}
```

3. On your class constructor ***(or any part of the class as long as this step is performed before you throw the event!),*** add the class instance as an observer to *NotificationCenter.*

```
private ProcessorDispatcher() {
        NotificationCenter.getInstance().addObserver(Notifications.ON_IMAGE_PROCESSING_STARTED, this);
        NotificationCenter.getInstance().addObserver(Notifications.ON_IMAGE_PROCESSING_FINISHED, this);
}
```

4. On the class that will throw an event (the notifier), put any needed data on a *Parameters* instance. Define the keys properly as you would need this for retrieving that data later. Throw the event as needed.

```
public class TestNotify {
        public final static String MY_ARRAY_KEY = "MY_ARRAY_KEY";

        private int[] testData;

        public TestNotify() {
        this.testData = new int[]{1,2,3,4,5};

        Parameters params = new Parameters();
        params.putExtra(MY_ARRAY_KEY, this.testData);

        //with parameters
        NotificationCenter.getInstance().postNotification(Notifications.ON_IMAGE_PROCESSING_STARTED, params);
        //no parameters
        NotificationCenter.getInstance().postNotification(Notifications.ON_IMAGE_PROCESSING_STARTED);

        }
}
```

5. Should you need additional data on your observer, this can be retrieved on the *Parameters* instance by using functions with the *get* prefix as seen below.

```
if(notificationString == Notifications.ON_IMAGE_PROCESSING_STARTED) {
                int[] retrievedData = (int[]) params.getObjectExtra(TestNotify.MY_ARRAY_KEY, null);
                this.startProcessing();
        }
```

It is up for the developers to be aware of their data types and if typecasting should be necessary. Do not abuse this as it also produces spaghetti code on the class level. This technique is only useful if a large component of your system (a group of classes) needs to throw an event to another component that is really completely different in essence.

For example, it is useful to use *NotificationCenter* to notify the UI handler component once the image processing component is complete; while it is redundant to throw an event in which only the image processing component listens to it.