The Android camera

Developing Android and native applications using the camera

Mobile Programming
2^e année SN, Parcours Multimédia - Informatique

Sessions 4-6

Version: v0.3.2019
(master @ 8db4eb8 2020-02-12)

Objective

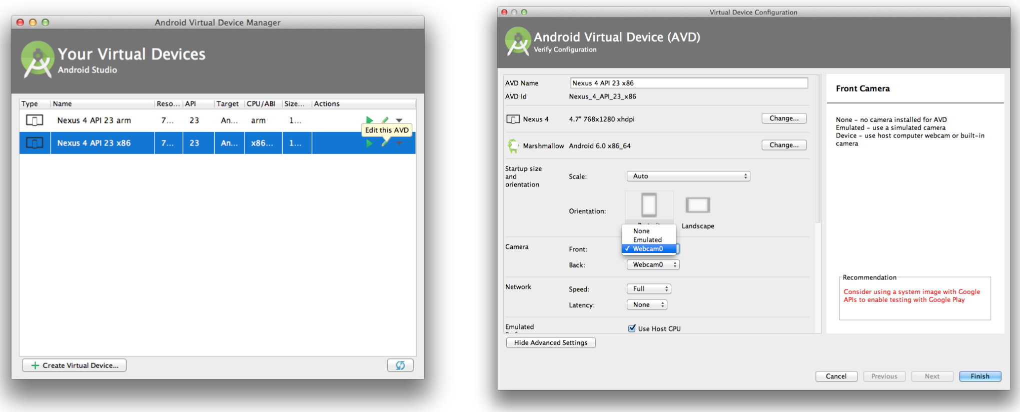
In these TP sessions, we will be focus on the multimedia applications using the Android Camera interface. In particular we will see how to develop a simple application that requires the use of the `Camera` interface to capture and display and image. The Android Camera interface is useful for most common application, although it provides a limited support for developing applications that have to process the camera stream in real time. In the second exercise we will see how we can develop a more complex camera application able to process the camera stream in real-time using the OpenCV libraries by means of the Java Native Interface (JNI).

Contents

1 Setting the camera on your AVD

For these lab session a webcam will be provided to each of you. The Android emulator can use the camera to emulate the camera of a mobile device. In order to set up the emulator to use the webcam you have to:

- plug the camera in one of the USB port of your PC;
- in Android Studio, the AVD manager (Tools > Android > AVD Manager);


Android Virtual Device Manager

Your Virtual Devices
Android Studio

Type	Name	Reso…	API	Target	CPU/ABI	Size…	Actions
	Nexus 4 API 23 arm	7…	23	An…	arm	1…	
	Nexus 4 API 23 x86	7…	23	An…	x86…	1…	

Virtual Device Configuration

Android Virtual Device (AVD)
Verify Configuration

AVD Name: Nexus 4 API 23 x86
AVD Id: Nexus_4_API_23_x86
Nexus 4: 4.7" 768x1280 xhdpi
Marshmallow: Android 6.0 x86_64
Startup size and orientation — Scale: Auto
Orientation: Landscape
Camera — Front: None / Emulated / ✓ Webcam0
Back: Webcam0
Network — Speed: Full
Latency: None
Emulated: ✓ Use Host GPU

Front Camera

None – no camera installed for AVD
Emulated – use a simulated camera
Device – use host computer webcam or built-in camera

Recommendation

Consider using a system image with Google APIs to enable testing with Google Play

Figure 1: Setting up the virtual device to work with the webcam.

- select the device you created for the other sessions and click on the green stencil icon on the left to edit its settings (see Figure 1);
- in the Camera, as "Front Camera" and "Back Camera" select `Webcam0`. Click "OK" and your virtual device is all set (see Figure 1);
- you can start the virtual device (you may need a "Cold boot") and check the camera is actually working by running the default `Camera` application: you should be able to see the live preview of the camera and to take pictures as you would do on your smartphone.

1.1 References

Here are some useful resources that may help you to complete the TP:

- A reminder of the main components of an Android application
  `https://developer.android.com/guide/components/fundamentals.html`
- Some explained examples of applications with specific components
  `https://developer.android.com/guide/components/fundamentals.html`
- The official documentation of the Android API.
  `https://developer.android.com/reference/packages.html`

Note that in this document the clickable hyperlink to the class API documentation is given for some classes and some class methods (*e.g.* `findViewById()`).

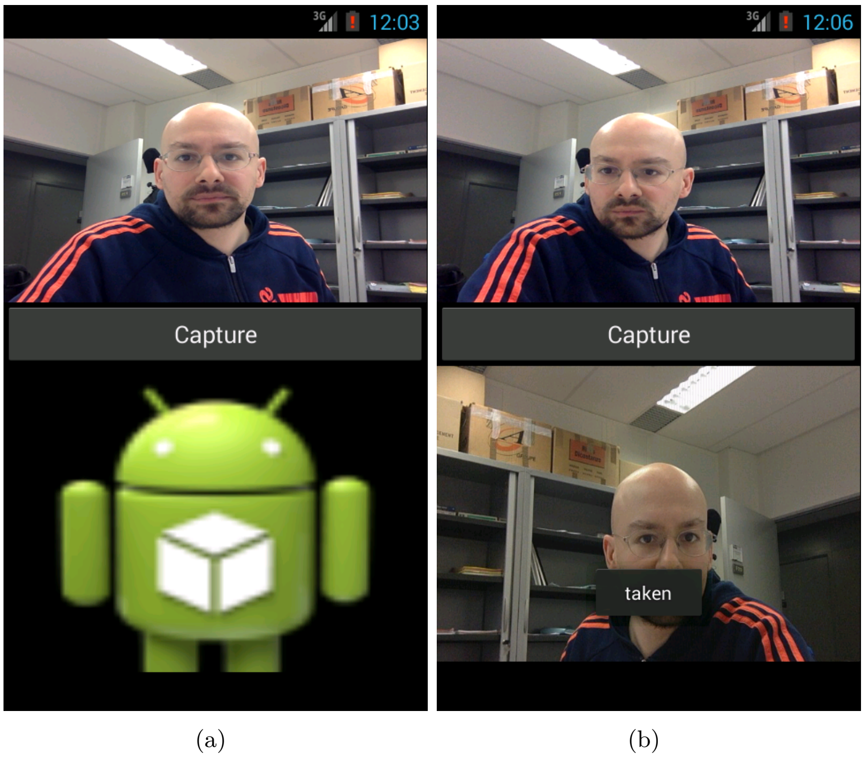


Figure 2: Example of the application to develop. The GUI is composed of 3 main parts (a), a widget to preview the live stream of the camera, a button to capture the image and another widget to display the picture taken. Whenever the picture is taken, a pop-up message (`Toast`) is shown (b).

2 Camera Shot

In this first exercise, you have to develop an application that mimic the default `Camera` application you can find on an Android device. In particular you have to develop an application similar to the one displayed in Figure 2. This camera application is a simplified version and allows the user to see the live preview of the camera, take a shot and see the taken picture while the live camera feed is displayed.

The application should consist of 3 main UI widgets:

- a `FrameLayout` that contains a surface displaying the camera feed;
- a `Button` to take the picture;
- an `ImageView` to display a snapshot of the taken picture.

Optionally, the taken picture can be saved on the sd-card of the virtual device.

Note After creating the new project, you need to change the targeted API version of the application, otherwise you will need to deal with the new way of managing the Android permissions introduced since Android 6 (API 23)[1]. In order to do so, in the Project View expand the Gra-

[1] If you are interested, more information about permission management after Android 6 can be found here.

dle script and open `build.gradle (Module: app)`: there, set the `targetSdkVersion` to 22 (instead of the default value 26).[2]

2.1 Implementation hints

In order to implement the application you can follow the steps that have been introduced in class:

- **Create a Preview Class**. Create a camera preview class that extends `SurfaceView` and implements the `SurfaceHolder.Callback` interface. This class previews the live images from the camera.
  - Implement the relevant callbacks of the `SurfaceHolder.Callback`; for simplicity we can assume that the application only work with the portrait orientation.
- **Create the UI layout**. Once you have the camera preview class, create a view layout that incorporates the preview and the user interface controls you want. Remember, you can use a `FrameLayout` as container in which you can insert the camera preview class.
- **Detect and Access Camera** - In the main activity of your application, create the code to check for the existence of cameras and request access to the front camera (`Camera.CameraInfo`). Get the camera and create a new instance of the preview class you implemented passing the camera object. Remember that the camera has to be acquired when the activity gain focus and it has to be released before the activity is stopped: override the proper activity lifecycle methods.
- **Setup Listeners for Capture** - Wire the button to start image capture in response to user actions, such as pressing a button and set the proper callback for the camera (`void Camera.takePicture()`)
  - When the picture is taken (`Camera.PictureCallback`) display it in the image view widget and save the image in the public directory for pictures on the sd-card (*e.g.* use `getExternalStoragePublicDirectory()` with `Environment.DIRECTORY_PICTURES`).
- Remember to set the proper permission(s) for the application!
- It is likely that whenever you take a picture it does not show immediatly in the "Gallery" or "Photo" application because Android updates the database of the images by means of a `MediaScanner` that is run once in a while. In order to force the update you can try to add this piece of code just after saving the image to the file:

```
File imgFile = getOutputMediaFile(MEDIA_TYPE_IMAGE);

// save the data buffer into imgFile
...

// this is to force media scanner to refresh the pictures for the gallery
MediaScannerConnection.scanFile(this, new String[]{imgFile.toString()}, null, null);
```

[2]Alternatively, you can go in Settings > Apps > CameraShot and grant manually all the permissions.

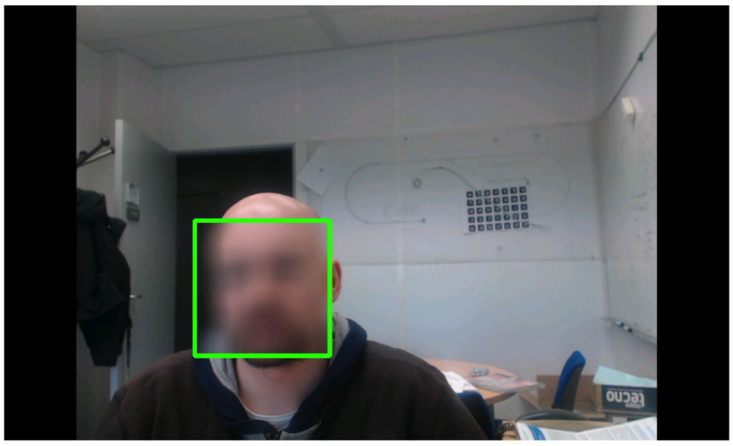


Figure 3: The defacer application to develop: the application detects the face(s) in the image (green rectangle) and it obfuscates them.

3 Defacer

In this exercise you will develop a custom camera application able to automatically detect faces in the live feed and obfuscate them with image processing techniques (Figure 3). In order to have better performances in processing the images, the application will rely on native methods and the OpenCV library to manage the camera and process the image A skeleton of a code is available to download from Moodle. Carefully follow the instructions to import the given code in Android Studio and install the OpenCV library on your virtual device.

3.1 Setting up the environment

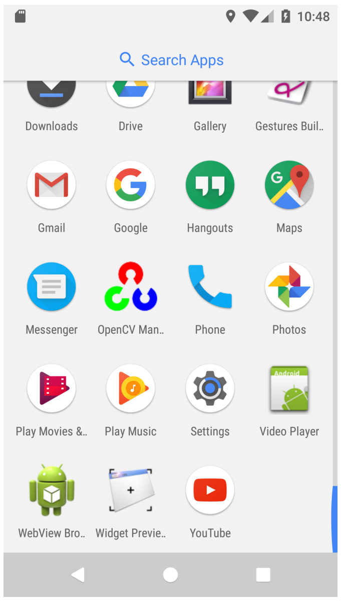
Follow these steps to get started with the code that has been provided:

1. Start your Android emulator. Once it has finished booting you can install the OpenCV `Manager` and the libraries. In order to do that you will use the `adb` tool (Android Debug Bridge). This is a command line tool that lets you communicate with an emulator instance or connected Android-powered device: *e.g.* it allows to install applications on the device by passing the `apk` archive of the application. Android Studio uses it in a transparent way every time you run an application to install it and communicate with the device (the logging).
   In order to install the OpenCV `Manager` and the library you can run the following commands from the command line of your terminal:

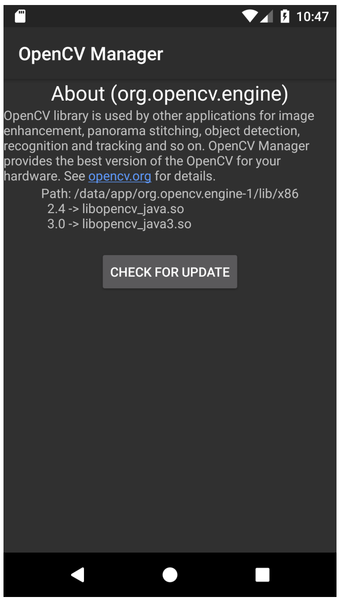
   ```
   adb install /mnt/n7fs/ens/tp_gasparini/OpenCV-3.4.0-android-sdk/apk/OpenCV_3.4.0_Manager_3.40_x86.apk
   ```

   If the operation succeed (and it should!) you can now go on your virtual device and among the installed application you should see the OpenCV `Manager` (Figure 4.a). If you open it, it will display the OpenCV version installed on your device (Figure 4.b).

2. Now you can import the code that you have downloaded from Moodle: "File > Open..." and select the path of the folder containing the code.



(a)



About (org.opencv.engine)

OpenCV library is used by other applications for image enhancement, panorama stitching, object detection, recognition and tracking and so on. OpenCV Manager provides the best version of the OpenCV for your hardware. See opencv.org for details.

Path: /data/app/org.opencv.engine-1/lib/x86
2.4 -> libopencv_java.so
3.0 -> libopencv_java3.so

(b)

Figure 4: Once you run the adb command the OpenCV Manager is installed (a): if you open it you can find the installed version(s) of the OpenCV library.

The reference for the Java API of OpenCV can be found at the Javadoc for OpenCV4Android, while the C++ reference can be found online at OpenCV API Reference.

3.2 The code

The code you have been given has already most of the components already implemented. You are required to fill only some parts of it with few instructions, both in the Java and JNI (C++) part. Normally the parts that you have to implement are marked by this kind of comment

```
//***************************************
// detect the faces and obfuscate them
//***************************************
```

The goal of this exercise is to show the relative complexity of an Android application using native libraries. So, beside just completing the code, you should also take some moments to see how the application is structured and understand the general workflow.

The application is composed of 2 classes:

- `DefacerActivity` which displays the camera preview and which will call the methods to detect the faces and process them;
- `DetectionBaseTracker` is the class containing the native methods for detecting faces from a given image. Under the hood, this class manages an object of the OpenCV (C++) class `DetectionBasedTracker` that implements a general image detector. This means that it can be used to detect and track faces but also any other object: the object to detect is specified by a classifier that contains "the rules" for detecting a given object in an image, and which is passed to the constructor of the class in the form of a XML file. This file is part of the resources of the application and it is stored in `res/raw` of the project.
  You can also note that the class provides a public API of Java methods, each of them

calling a corresponding private native methods. This allows to mask the development details of the class and its underneath structure. The class provides a constructor and some methods that allows to set the parameters of the detector (`setMinFaceSize()`) and control its workflow (`start()`, `stop()`, `release()`). For this exercise you don't have to use them as the initialization and the parameter settings are already managed inside `DefacerActivity`, as we will see in the next section. Instead, you will use the method `detect()` to detect the faces and later on you will add another method to process the detected faces. As usual, the corresponding C/C++ implementation of the native methods is in the source file of the `jni` folder along with the header file generated automatically with `javah`.

3.2.1 `DefacerActivity`

Let's now have a closer look to the main activity.

- You can recognize the code for requesting the bind to the OpenCV using the OpenCV Manager: the `onResume()` method calls the `OpenCVLoader.initAsync()` to get the binding to the installed version of the OpenCV. Also, the Loader Callback is implemented as an anonymous class handled by the class attribute `mLoaderCallback` which implements the abstract class `BaseLoaderCallback`: when the connection to the Manager is established the native library is loaded (`System.loadLibrary("detection_based_tracker")`) and then, inside the `try...catch` code the detector is initialized by loading the classifier file from the resources.

- the `onCreate()` is used, as usual, to get the handle to the GUI. In this application the only UI element is the OpenCV widget for the camera pre-visualization `CameraBridgeViewBase`, which, as the `CameraPreview` class you implemented in the first exercise, allows to display the images from the camera. You should note that the Android `Camera` is not used in this case as the `CameraBridgeViewBase` innerly uses an OpenCV native implementation of the camera. The `mOpenCvCameraView` object need to specify the ID of the camera (`setCameraIndex`) and a callback listener (`setCvCameraViewListener()`) that implements the OpenCV interface `CvCameraViewListener2`: this interface allows to implements methods that manage the life-cycle of the camera, and in particular the callback that is triggered whenever a new frame is available so that it can be processed. `DefacerActivity` implements this interface by overriding: `onCameraViewStarted()`, `onCameraViewStopped()` and `onCameraFrame()`.

- The callback `onCameraFrame()` is the most important one as it provides the way to access to each new frame of the camera and hence process it. The method just recovers the RGBA image (A is the alpha channel, that models the transparency) and returns it as it is. It is here where you have to use the detector to detect the faces and obfuscate them.

3.3 The assignment

- **Face detection**. The first thing to do is to process the frame to detect the faces.
  - Complete the code inside `onCameraFrame()` in order to process the rgba frame and retrieve the faces.
  - Use the `detect()` method of the tracker object by passing the image and the object that will contain the faces: this is a "matrix of rectangles" `MatOfRect`: each detected

face is represented by a rectangle containing the region of the image where the face has been found. In OpenCV a `Rect` is represented by the x and y coordinates of the top-left corner and by its width and length. `MatOfRect` is just the way a list of rectangles are stored by OpenCV. You can declare a local variable inside `onCameraFrame()` that is passed to the detect method to retrieve the faces.

  – So far there is no visual feedback to show the detected faces. On the other hand you can check the LogCat to see how many faces are detected.

- **Visual feedback**. Now that you have the face detection working you can first try to visualize the rectangle(s) containing the face in the image as shown in Figure 3.

  1. Look at the `detect()` method of the class `DetectionBaseTracker`: you have to create a similar function (let's call it `deface()`) that takes the rgba image and the rectangles containing the faces and call a new native method (let's call it `nativeDeface()`) that will actually draw the rectangles with the native code. The native function takes as parameters the pointers (`jlong`) to the detector, to the image and to the matrix of rectangles (please respect the order of the parameters).
  2. Once you have declared all the new functions, you can update the C/C++ header of the native code: you can open a terminal window in the `app/src/main/jni` folder and run the `javah` utility to generate the header. Remember that `javah` has the following syntax:

     ```
     javah -jni -classpath path -o theNameOfTheHeader.h the.class.name
     ```

     where:

     – `theNameOfTheHeader.h` is the filename to generate, in this case `DetectionBasedTracker_jni.h`

     – `the.class.name` is the full class name (with the package) for which you want to generate the header, in this case `fr.enseeiht.defacer.DetectionBasedTracker`

     – `path` is a list of paths separated by a ":" where to look for classes: in this case you have to provide the path to the `fr.enseeiht.defacer.DetectionBasedTracker`, normally the `build/intermediates/classes/debug/` directory of the `app`, and the path to the OpenCV classes, *i.e.* `/mnt/n7fs/ens/tp_gasparini/OpenCV-2.4.10-android-sdk/sdk/java/bin/classes`.

     You might have a look to the `update` file in `jni` for an example.
  3. Once you have update the header file `DetectionBasedTracker_jni.h` you should see a new prototype for the native function `nativeDeface` called `Java_fr_enseeiht_defacer_DetectionBasedTracker_nativeDeface`
  4. Now you should add this function to the C/C++ source file `DetectionBasedTracker_jni.cpp`: for simplicity this is already been done and you can find the function at the end of the file, you have now to implement the `try` part of the function.
  5. A vector of rectangles `rectFaces` has already been recovered from the parameters of the function as well as the variable containing the image `imgOut`. Now you have to iterate the vector `rectFaces` and for each rectangle draw the rectangle. OpenCV provides the function `rectangle()` to draw a rectangle in a image.
  6. The C/C++ code is compiled automatically by Android Studio every time you save the file.

7. Finally, you just have to properly call the Java method `deface()` inside the main activity after the detection part.

- **Defacing**. Now the only thing left is to obfuscate the faces and for that you just need to complete the code you just wrote.

  - in the native C/C++ implementation of `nativeDefacer`, you can obfuscate the face before drawing the rectangle using the method you want. First you need to get the portion of the image corresponding to the rectangle, or ROI (Region Of Interest). To do so, you can get a portion of the original image into a new `Mat` variable using the `Mat` constructor:

    ```
    Mat::Mat(const Mat& m, const Rect& roi)
    ```

    where `m` is an existing (in this case the `imgOut`) and `roi` is the current rectangle that you are considering. With the new variable you can apply the obfuscate method you prefer, such as a blurring (`blur()`) or just setting all the pixel to a given color (`Mat::setTo()`).