Camera Video Stream Processing with RenderScript (Android* OS Tutorial)

User's Guide

Compute Code Builder - Samples

Contents

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Introduction

The Camera Video Stream Processing tutorial demonstrates basic approach to processing video stream from a camera with help of the Google RenderScript*.

Specifically, the tutorial uses "preview" feature of the Android camera interface, which relies on the RenderScript to convert YUV data from the camera to RGB. The code sample also uses RenderScript to accelerate the data post-processing: a popular “Old Movie” video effect is implemented as a RenderScript kernel. Finally, the sample uses the copy-free way of displaying the resulting image on the screen.

The tutorial runs on devices with Android* OS 4.3 and higher.

Enabling Camera Support in the Application

Android offers rich API supporting enumerating the available cameras (for example, front/rear), gathering camera capabilities (like resolution), and finally enabling to capture pictures and videos in the applications.

The steps for implementing general camera “pipeline” are as follows:

1. Enumerate cameras and capabilities (for example, available resolutions)

```
//here we use default back-facing camera (best for the video effect)
Camera camera = Camera.open(0);
params = camera.getParameters();
// the old movie effect does not require high resolution source
// 640x480 resolution is enough so choose preview resolution that is close
List<Camera.Size> sizes = params.getSupportedPreviewSizes();
for(int i = 0; i < sizes.size(); ++i)
{
      Camera.Size size = sizes.get(i);
      //any logic to select the preview size
      …
      //notice that when setting preview size you must
      //use entries from getSupportedPreviewSizes(), not arbitrary values
      params.setPreviewSize(size.width,size.height);
}
// preview in NV21 format (which is supported by all cameras, unlike RGB)
params.setPreviewFormat(ImageFormat.NV21);
camera.setParameters(params);
```

2. Setup camera preview callbacks, which enables live image previews from the camera (see Basics of “Camera Preview” Feature for details)

```
//in the OnCreate method:
…
//Assign a view that displays the preview to the user
SurfaceView surView = (SurfaceView) findViewById(R.id.inputSurfaceView);
//the surface callback is needed for assigning the camera preview input
SurfaceHolder surHolder = surView.getHolder();
//our Activity implements SurfaceHolder.Callback, so we use ‘this’
//to handle preview surface changes
surHolder.addCallback(this);
…
//in the surfaceCreated callback
camera.setPreviewCallback(this);
```

3. <Optional step> Implement any kind of post-processing to the preview frame .See section Implementing the “Old Movie” Effect for details.

4. <Optional step> Use MediaRecorder class to capture the resulting video (not a subject for this tutorial)

5. Implement proper resources releasing, for example freeing Camera object upon application exit, for use by other applications.

```
camera.stopPreview();
camera.setPreviewCallback(null);
camera.release();
camera = null;
```

Find more details on the general camera setup steps in the dedicated Android tutorial. You can also navigate directly to the source code of this tutorial and look through comments.

Basics of "Camera Preview" Feature

Camera preview is an important feature for users. The feature provides a capability of taking pictures or video more effectively by enabling to preview what the device camera sees. It is also possible to apply any kind of post-processing to the preview frame before displaying the preview on the screen.

The following example code demonstrates the basic camera preview class. It implements methods of SurfaceHolder.Callback to capture the callback events for creating and destroying the surface.

NOTE: Connect the surface to some valid `SurfaceView`, otherwise the camera does not produce preview frames.

Finally, the Activity-derived class also implements the `Camera.PreviewCallback` to get actual raw previews in plain byte arrays. Another option is to operate on `SurfaceTexture` instead. Refer to the setPreviewTexture, but remember that operating on `SurfaceTexture` is much easier with OpenGL, not RenderScript.

```
public class MainActivity extends Activity
        implements Camera.PreviewCallback, SurfaceHolder.Callback
{
    // Single ImageView that is used to output the resulting bitmap
    private ImageView outputImageView;
    …

    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        …

        setContentView(R.layout.activity_main);
        //ImageView, which is used to display the resulting image
        outputImageView = (ImageView)findViewById(R.id.outputImageView);
        …

        //get preview surface for camera preview and set callback for surface
        //actually the layout is specified the way
        // the inputSurfaceView is completely overlaid by the outputSurfaceView
        //   so inputSurfaceView is rather fake view for unprocessed preview frames
        //   (which do not want to display, yet it is required to setup the callback)
        SurfaceView surView = (SurfaceView) findViewById(R.id.inputSurfaceView);
        SurfaceHolder surHolder = surView.getHolder();
        surHolder.addCallback(this);
    }

    public void onPreviewFrame(byte[] arg0, Camera arg1)
    {
        // actual data from the Camera arrives here
```

```
        // since the processing is async, camera continues to produce frames
        // so we skip subsequent previews until processing of the current is done
        if (RenderScriptIsWorking…)
            return;
       RenderScriptIsWorking = true;
      …
      //copying bytes from the buffer
      //(to let camera proceeding with next frames upon return from onPreviewFrame)
      allocationYUV.copyFrom(arg0);
      //issue an async task to process the frame in (other than UI) thread
      //it will also cause ImageView update, and reset the RenderScriptIsWorking flag
      new ProcessData().execute();

      …
    }

    public void surfaceChanged(SurfaceHolder holder,int format,int width,int height){
      //do nothing as we actually do not use the surface
      //still need the stub to complete the camera callbacks
    }

    public void surfaceCreated(SurfaceHolder holder) {
      //re-init the camera upon any surface change
        try {
            camera = Camera.open(0);
            camera.setParameters(params);
            camera.setPreviewCallback(this);
            camera.setPreviewDisplay(holder);
            camera.startPreview();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
      //release the camera upon surface destroy
        camera.stopPreview();
        camera.setPreviewCallback(null);
        camera.release();
        camera = null;
    }
…
}
```

Let's inspect the code that sets the camera callbacks in details:

```
            camera.setPreviewCallback(this);
            camera.setPreviewDisplay(holder);
            camera.startPreview();
```

This is minimal APIs calls to start displaying the unprocessed preview frames. Camera implicitly communicates with `SurfaceView` using the surfaces. All you need is to retrieve a `SurfaceHolder` object ('holder' above) and provide this object to the camera via `setPreviewDisplay()`. Then camera also needs an object that implements callbacks on the surface changes, to be set with `setPreviewCallback()`, in the case of this tutorial, the `Activity` provides the necessary callbacks, so the code uses 'this'. Upon that you can start getting preview frames with `startPreview()`.

The only trick is when you want to display modified images **only**. The easiest way (employed in this tutorial) is to add another `View` that renders the processed images, while completely hiding the first `View`. You need the first view just to get unmodified preview images in surfaces (hence it is of `SurfaceView` type, below). These roles are exactly the purposes of the `inputImageView` and `outputImageView`. Notice that per layout, the second view completely overlaid the first one:

```
    <SurfaceView
```

```
        android:id="@+id/inputSurfaceView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
    <ImageView
        android:id="@+id/outputImageView"
        android:background="#FF000000"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
```

Unless your application is already OpenGL-enabled, you can use this API to process the input preview frames, otherwise the RenderScript is the best match to implement the kernels. The easiest way to get zero-copy displaying of the image processing results, is to rely on bitmap-to-allocation sharing. In turn, bitmaps are best displayed with `ImageView`, hence the type of `outputImageView` above. More details on this in the Multi-Stage Post-processing with RenderScript section below.

What is RenderScript?

RenderScript is a framework designed by Google* for performing data-parallel computation in Android-based devices. RenderScript kernels and intrinsics may be accelerated by an integrated GPU. Thanks to tight integration of the RenderScript with the Android OS through Java APIs, you can perform data-parallel computations from Java applications in the efficient way. The RenderScript runtime also takes care of the efficient scheduling and parallelizing of work on the right processor available on a device. Refer to the previous tutorial in the series for the general introduction on the RenderScript. Finally RenderScript offers a set of built-in kernels (intrinsics) for populr tasks like converting YUV to RGB, blur, etc.

In this tutorial RenderScript is used to implement the popular “Old Movie” video effect.

Multi-Stage Post-processing with RenderScript

First step in the application logic is converting the camera input (preview) frames from YUV to RGB using `ScriptIntrinsicYuvToRGB` intrinsic (built-in) class. Then the image is blurred with another intrinsic function, `ScriptIntrinsicBlur`. After that, the simple setup function is called per frame (to advance time step, and so on) and finally the post-processing kernel itself.

To make the process asynchronous (per general Android guidelines to avoid expensive operations in the main UI thread), the tutorial implements a simple class that extends basic AsyncTask functions with what is needed:

```
private class ProcessData extends AsyncTask<byte[], Void, Boolean>
{
      protected Boolean doInBackground(byte[]... args)
      {
            …
            // Run the scripts
            // 1) conversion to RGB
            intrinsicYuvToRGB.setInput(allocationYUV);
            intrinsicYuvToRGB.forEach(allocationIn);
            // 2) apply Blur
            intrinsicBlur.setInput(allocationIn);
            intrinsicBlur.forEach(allocationBlur);

            // 3)update filter state
            script.invoke_update_state();
            // 4)set filter args and run the OldMovie filter
            script.forEach_filter(allocationBlur, allocationOut);

            //5) wait for completion
            rs.finish();

            //6)propagate the results back to the bitmap
```

```
        allocationOut.syncAll(Allocation.USAGE_SHARED);
        …
}
```

Notice that the very first input allocation (`allocationYUV`) gets the bytes directly from the camera preview frames. Copying is required to keep the process asynchronous, which, for example, enables camera to proceed without blocking until the processing is done:

```
//copying bytes from the buffer (to let camera to proceed with next frames)
allocationYUV.copyFrom(arg0);
```

In contrast, the final (output) allocation, which is the `allocationOut` is connected to the bitmap upon creation to enable zero-copy updates, refer to the `OnCreate` method of the sample `Activity`:

```
// Create an allocation (which is memory abstraction in the Renderscript)
// that corresponds to the outputBitmap
allocationOut = Allocation.createFromBitmap(rs,outputBitmap);
```

Finally, the bitmap is displayed using the regular `ImageView`. Refer to the following method of the `ProcessData` class:

```
  protected void onPostExecute(Boolean result) {
        …
         outputImageView.setImageBitmap(outputBitmap);
         outputImageView.invalidate();
         RenderScriptIsWorking = false;
    }
```

Implementing the "Old Movie" Effect

"Old movie", also known as "film look" is the popular video effect that applies film-style lighting, noise, camera motion (jittering), converting to black-and-white, specific framing, simulated grading over time, and other effects that are associated with aged films.

This tutorial comprises the basic implementation of "Old Movie" effect, including:

- Stains
- (Vertical) Scratches
- Hairs and irregular scratches
- Camera jittering
- Intensity and color variations
- Semi-transparent vignette

To simplify the code, the tutorial keeps only one random and vertical scratch, one stain, and so on at a time. Still, all the effects rely on random number generation to evolve over time in the visually convincing way.

See the code in the `process.rs` for details on the particular implementation.

Running and Controlling the Tutorial

If you compile the tutorial using the Eclipse* IDE, you should be able to run the tutorial on the device directly from the IDE. Refer to Android documentation for details on building and running from Eclipse with ADT. Alternatively, you can use the ADB utility from the Android SDK platform-tools to install the pre-built executable yourself from `bin/CameraRenderScript.apk`.

Finally, if you have neither IDE nor even Android SDK, nor ADB, you can copy the pre-built APK to the device over regular USB and install it directly on the device using any file manager for Android OS.

After successful installation, the tutorial application becomes available on the device with the following icon:

Upon launching, the tutorial starts in the full-screen mode and displays a preview from the default device camera (back-facing) with a video effect applied to each frame. On the left-top corner you can find the current overall application FPS and also a separate FPS value (in brackets) for performance of the RenderScript portion of the app.

The **VideoEffect On / Off** slider enables you to turn the video effect off, so that the application displays the unmodified camera output.

References

- Intel® SDK for OpenCL™ Applications – User's Guide
- Intel SDK for OpenCL Applications – Optimization Guide
- Android* Development Tools at http://developer.android.com/tools/sdk/eclipse-adt.html
- Android NDK at http://developer.android.com/tools/sdk/ndk/index.html
- RenderScript at http://developer.android.com/guide/topics/renderscript/compute.html