RenderScript Basic Tutorial for Android* OS

User's Guide

Compute Code Builder - Samples

Document Number: 330186-002US
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>2</td>
</tr>
<tr>
<td>Legal Information</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>What is RenderScript?</td>
<td>4</td>
</tr>
<tr>
<td>Running and Controlling the Tutorial</td>
<td>4</td>
</tr>
<tr>
<td>RenderScript Java* APIs</td>
<td>5</td>
</tr>
<tr>
<td>RenderScript Kernels</td>
<td>7</td>
</tr>
<tr>
<td>Tutorial Application Structure</td>
<td>8</td>
</tr>
<tr>
<td>Separating RenderScript Kernel and UI Thread Executions</td>
<td>8</td>
</tr>
<tr>
<td>Android Application Lifecycle Events</td>
<td>8</td>
</tr>
<tr>
<td>Limitations</td>
<td>9</td>
</tr>
<tr>
<td>Notes on Threading</td>
<td>9</td>
</tr>
<tr>
<td>References</td>
<td>10</td>
</tr>
</tbody>
</table>
Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number/.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel, Intel logo, Intel Core, VTune, Xeon are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission from Khronos.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Copyright © 2014 Intel Corporation. All rights reserved.

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

RenderScript Basic Tutorial for Android OS provides guidelines on using the RenderScript in application development for Android OS version 4.2.2 and higher.

The focus of the tutorial is on the RenderScript, its kernel language, and Java API. Since the tutorial uses only Java, even though Native (C/C++) analogs for RenderScript API exist, and since RenderScript kernels are actually Just-In-Time as well, the resulting application binary can be executed on any device running the supported version of the Android OS.

The tutorial application is a minimalistic GUI activity that displays the computation results and statistics. It implements a simple full-screen image processing algorithm, with which you can interact through screen touches. After launching the tutorial application, a built-in image is displayed, and no post-processing happens until user interaction begins.

When you touch the screen, an animated growing circle centered at coordinates of the touch appears, and all pixels that are inside the circle become black and white. The circle continues to grow each frame until the complete picture becomes black and white.

Since the tutorial uses only Java, even though Native (C/C++) analogs for RenderScript API exist, and since RenderScript kernels are actually Just-In-Time as well, the resulting application binary can be executed on any device running the supported version of the Android OS.

Look into README.txt for more details on the tutorial file structure and build/run steps.

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 communication and memory management between the two levels of code is facilitated automatically, so you do not need to code much for things like data conversions, marshaling, and so on. The RenderScript runtime also takes care of the efficient scheduling and parallelizing of work on the right processor available on a device.

The tutorial covers the important concepts you should understand to begin with the RenderScript.

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/AndroidBasicRenderScript.apk.

Finally, if you have neither IDE nor even Android SDK, nor ADB, you may 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 an image, which is being updated with the results of applying the RenderScript kernel to each frame. On the right-top corner you can find the current FPS and corresponding frame time (tagged “Frame”) and the time taken by the RenderScript kernel itself (tagged “Effect”). The compute API itself is also shown at the left topmost corner (“RenderScript”).

The effect or processing that applies to the input image originally loaded from the application resources is computing the “circle of black-and-white” with the dedicated RenderScript kernel. The effect becomes apparently visible upon a screen touch. Specifically, the origin of the circle is the point of touch, so when you touch the screen the new circle appears and expands over time. For each input pixel, the RenderScript kernel determines if the pixel is inside or outside the circle. It then outputs the outer pixels untouched, while converting the inner pixels to the black-and-white for the output and also draws a thin circle border.

The **TakePhoto** button enables you to replace the default image for processing with the picture taken from the camera of your device. For details and the general machinery of this functionality including the update of the input Allocation with new picture from the camera, refer to the tutorial code. The code that handles the camera is intentionally simplified (for example it does not handle camera orientation) to keep focus on the core tutorial topic which is RenderScript.

**RenderScript Java* APIs**

Android OS offers a Java* API (android.renderscript class package) for managing the RenderScript resources and kernel execution.
Basic steps to use RenderScript from the Java code:

- Creating a RenderScript context via `RenderScript.create()` that returns the new RenderScript context.
  
  - `RenderScript.create()` is a static method that accepts instance of the `android.content.context`, that is the interface to the application environment including the target SDK version.
  
  - The tutorial has only a single RenderScript that spans the general `Activity` lifetime:
    ```java
    RenderScript rs = RenderScript.create(this);
    ```

- Creating scripts via reflected Java class for user-defined scripts.
  
  - Named as `ScriptC_filename`, where `filename` is a name of the script (`*.rs`) file.
  
  - Since the tutorial comes with a `process.rs`, the Java code to create the script appears as:
    ```java
    ScriptC_process script = new ScriptC_process(rs);
    ```

- Creating `Allocation` that are memory abstractions to be passed to the script kernel as input and output.
  
  - Each `Allocation` contains a fixed amount of data of the type, specified when created with `createTyped(RenderScript, Type)`, or, for example, an `Allocation` can contain bitmap data when created with `createFromBitmap(RenderScript, Bitmap)`.
  
  - The tutorial has an "input" (read-only) `Allocation` created from the original bitmap, and another (write-only) `Allocation` for the kernel outputs.

- Executing the kernel. The reflected Java class for the tutorial script has an auto-generated method to issue the "process" kernel, named `forEach_process()`. The method accepts two `Allocation` (input and output). In this tutorial, the kernel executes over the entire input `Allocation`. Also prior to kernel execution, the values for the script global variables are set via methods from the same "reflected" class `ScriptC_process`. Methods are automatically created using the naming scheme `set_globalname()`, where `globalname` is the name of the variable in the script. This tutorial utilized a few script global variables:
  
  - `xTouchApply` and `yTouchApply` for the touch coordinates (to define the center of the circle)
    ```java
    script.set_xTouchApply(xTouchApply);
    script.set_yTouchApply(yTouchApply);
    ```
  
  - `radiusHi` and `radiusLo` for the circle inner/outer radii that are computed on the host using the current timestamp
    ```java
    script.set_radiusHi(radiusHi);
    script.set_radiusLo(radiusLo);
    ```

- Copying data out of the output `Allocation`.
  
  - This is needed to expose the data to the Java and, eventually, to the screen. This tutorial application copies from `Allocation` back to the Bitmap:
    ```java
    AllocationOut.copyTo(outputBitmap);
    ```

The last two steps (executing the kernel and copying the result back to Java) happen each frame, followed by a "worker" thread posting a Runnable, which is "call to action", to the UI thread, so it accepts the updated bitmap and invalidates the respective view (to display the result):

```java
outputImageView.setImageBitmap(outputBitmap);
outputImageView.invalidate();
```
RenderScript Kernels

RenderScript kernels reside in RS files in the regular source directory. Each RS file constitutes a script. Every script has a set of global variables and kernels.

The script can include several compute kernels. Essentially, each kernel is a parallel function that is executed for each Element within the input Allocation. Also there is special case of serial RenderScript functions (called invokables) that can be called from Java code with arbitrary arguments.

Let’s look at the typical RenderScript script and an example kernel declaration:

```c
#pragma version(1)
#pragma rs java_package_name(com.intel.sample.androidbasicrs)

//script globals
int radiusHi;
...

//script global constants
const float4 gWhite = {1.f, 1.f, 1.f, 1.f};
...

//kernel declaration
uchar4 __attribute__((kernel)) root(const uchar4 in, uint32_t x, uint32_t y)
{
//kernel body
  float4 f4 = rsUnpackColor8888(in);
  ... 
  uchar4 out;
  ...
  return out;
}
```

Here are a few important caveats about this code:

- A pragma in the first line declares the version of the RenderScript kernel language used in this script with 1 being the only valid value for today;
- Second pragma declaration - #pragma rs java_package_name - declares the package name for the Java class that reflects this script.

This particular declaration of the kernel `root` is specific for API Level 17 (Android 4.2.2):

- Notice the __attribute__((kernel)) applied to the function prototype. This tags the kernel in contrast to the regular (serial) invokable function of the script.
- The first in argument is somewhat special. For example, it has a type, automatically matching the input Allocation passed to the kernel. Similarly, the return value of the kernel is automatically stored to the corresponding location in the output Allocation. It is automatically checked that the Element types of the input and output Allocations match the kernel declaration.
- The kernel is executed across the entire input Allocation (which can be altered via an additional LaunchOptions parameters for the script.forEach_root() function) so its body is issued once per each Element in the input Allocation. Thus there is one output Allocation entry per input Allocation value.
- A kernel is passed with the coordinates of the current execution: x, y (and z in the case of 3-d iteration space arguments). These arguments are optional for use inside a kernel, so this tutorial application ignores them. For example of kernel that does use x and y refer below.

For the subsequent Android API Levels, the kernel declarations are getting somewhat relaxed. Still, it is important to understand that a kernel has a single input and single output Allocation. If more Allocations are needed, those objects should be bound to the rs_allocation script globals.
Binding an Allocation via script global also enables to access arbitrary Elements of the allocation via rsGetElementAt_type() and rsSetElementAt_type(). This is a useful alternative to the plain Element-in/Element-out approach the default kernel declaration offers. For example, if you need to gather the input Elements neighboring to the current, you define the input Allocation as global. Consider the example for API Level 19:

```java
...
rs_allocation in;

void root(uchar4 *out, uint32_t x, uint32_t y)
{
    float4 f4_center = rsUnpackColor8888(rsGetElementAt(in, x, y));
    float4 f4_1      = rsUnpackColor8888(rsGetElementAt(in, x+1,y));
    float4 f4_2      = rsUnpackColor8888(rsGetElementAt(in, x  ,y+1));
    ...
}
```

### Tutorial Application Structure

The tutorial comprises a single Activity (sub-classed in the MainActivity.java) that interacts with the user. The Activity takes care of creating a full-screen window, in which takes place the UI, and which consists of the following elements:

- TextViews to print the performance statistics on the screen.
- A single ImageView to output the result of the computations that the RenderScript produces.

The visual structure of the GUI elements and the associated logic, like the relative layout, is declared in XML, see activity_main.xml in the \res\layout folder.

### Separating RenderScript Kernel and UI Thread Executions

One particularly important concept in the mobile domain is "Keeping Your Application Responsive". Android applications typically run in a single thread, which is the "UI thread" by default. The thread handles UI events. Any high-latency operations within this thread should be minimized, in order to keep the application GUI responsive.

Thus the first important tip is to utilize the rich set of task and threading primitives in the Android API to execute any potentially long-running operations, such as computationally intensive RenderScript kernels via some sort of asynchronous requests.

To express the producer-consumer relations between the UI and the RenderScript in this tutorial, which is the UI thread displaying or "consuming" bitmaps that RenderScript kernel updates or "produces", it uses a scheme with a dedicated "worker" thread. This thread computes the updated images and notifies the UI thread upon completion, so the UI thread updates the screen with the result.

### Android Application Lifecycle Events

From the purposes of this tutorial, there are few important methods that the Activity needs to implement:

- onCreate is where the activity is initialized. Most importantly, beyond actually setting Views with resources defining the UI, the general RenderScript context initialization is taking place here.
- onStart/onStop () where the application becomes visible and invisible to the user. Most importantly this is the place where "worker" thread is paused/resumed.
OnWindowsFocusChanged(boolean) which is called (with 'true' as a parameter) when all of the UI have been successfully loaded and created properly.

- In this function it is safe to initialize bitmaps for double-buffering, because here you already know your widgets dimensions.
- This is also a place where a dedicated "worker" thread is spawned, since the memory resources are already initialized at this point.

- OnDestroy, the final call to the application to release resources and enable the "worker" thread to exit.

Limitations

The focus of this tutorial is on the RenderScript APIs, not advanced topics like Android application lifetime caveats, so for simplicity this tutorial uses screen orientation fixed to the 'landscape'. So images you taken with camera (refer to the GUI section) might be rotated.

Using bitmaps without more efficient surface handling is also for simplicity. In real performance/power oriented application one may want to use OpenGL surfaces instead.

Notes on Threading

Unlike potential alternatives based on any notion of a "async task issued by the UI thread", which would essentially inverse the producer-consumer relations, the worker thread is not popping up each time the UI thread needs an updated image. Instead, it produces the images and waits for the UI thread to consume them. This is implemented through a couple of conditional variables.

The lifetime of the worker thread spans the lifetime of the Activity itself, so this ensures no expensive thread creation or deletion happens behind the scene. Similarly, the double-buffering scheme prevents the intensive resource manipulation for the output Bitmaps.

Consider the loop the worker thread executes simplified to the pseudo-code for brevity:

```java
public void run()
{
    // Check if the application is exiting, so you don’t need frames anymore.
    while(!isShuttingDown)
    {
        // Conditional that allows the thread to sleep when
        // the application is not in focus.
        isGoing.block();
        // Conditional that prevents output to the bitmap the UI thread
        // displays now.
        isRendering.block();
        {
            // Check if you need to update the input image (after
            // default photo replaced image captured from the camera).
            // Swap bitmaps used for processing and displaying.
            // Actual call to the kernel
            step();
            // Block the thread from computing next
            // frame to the same bitmap.
            isRendering.close();
            // issue the Bitmap/View update on behalf of
```
// UI thread via post
outputImageView.post
{
    outputImageView.setImageBitmap(outputBitmap);
    outputImageView.invalidate();
    // allow the worker thread to process the next frame
    isRendering.open();
};

References

- Intel® SDK for OpenCL™ Applications – User’s Guide
- Intel SDK for OpenCL Applications – Optimization Guide