BAN CƠ YẾU CHÍNH PHỦ

**HỌC VIỆN KỸ THUẬT MẬT MÃ**

**---------------------------------**

****

**TỐI ƯU PHẦN MỀM DI ĐỘNG**

**Đề tài: RENDERSCRIPT**

Giảng viên hướng dẫn: **Lê Bá Cường**

Sinh viên thực hiện:

**Vũ Đình Sâm  
Nguyễn Anh Tuấn**

**Bùi Thị Thanh Loan**

**Đặng Thị Thu Hà**

Hà Nội, 12/2020

Contents

[I. Giới thiệu về RenderScript 5](#_Toc59183489)

[II. OverView 5](#_Toc59183490)

[III. Hello World 8](#_Toc59183491)

[IV. Hello Rendering 13](#_Toc59183492)

[1. Tạo một Rendering script 14](#_Toc59183493)

[2. Tạo một RenderScriptGL Context 15](#_Toc59183494)

[3. Extending RSSurfaceView 16](#_Toc59183495)

[4. Cài đặt Content View 17](#_Toc59183496)

[5. Adding Variables to Script 20](#_Toc59183497)

[V. HelloCompute 24](#_Toc59183498)

[1. Allocations 26](#_Toc59183499)

[2. rsForEach 27](#_Toc59183500)

[3. Performance 32](#_Toc59183501)

[VI. Native Renderscript Apis 34](#_Toc59183502)

[1. rs\_types.rsh 35](#_Toc59183503)

[2. rs\_core.rsh 37](#_Toc59183504)

[3. rs\_cl.rsh 39](#_Toc59183505)

[4. rs\_math.rsh 43](#_Toc59183506)

[5. rs\_graphics.rsh 45](#_Toc59183507)

[6. rs\_time.rsh 46](#_Toc59183508)

[7. rs\_atomic.rsh 47](#_Toc59183509)

[VII. RenderScript vs NDK 48](#_Toc59183510)

[VIII. Summary 49](#_Toc59183511)

[IX. Thực nghiệm 50](#_Toc59183512)

[1. Thực nghiệm ScriptIntrinsic 50](#_Toc59183513)

[2. Thực nghiệm ScriptC 54](#_Toc59183514)

[X. Tổng kết 55](#_Toc59183515)

[Tài liêu tham khảo 56](#_Toc59183516)

Danh mục hình ảnh

[Hình 1. 1 Chế độ xem băng chuyền YouTube 5](#_Toc58539435)

[Hình 1. 2 Ứng dụng mẫu Balls 6](#_Toc58539436)

[Hình 1. 3 Ảnh original 49](#_Toc58539437)

[Hình 1. 4 Ảnh sau khi Blur 50](#_Toc58539438)

[Hình 1. 5 Ảnh sau khi Convolve 51](#_Toc58539439)

[Hình 1. 6 Ảnh sau khi Convolve 52](#_Toc58539440)

Hình 1.7 Thực nghiệm ScripC…………………………………………....54

# Giới thiệu về RenderScript

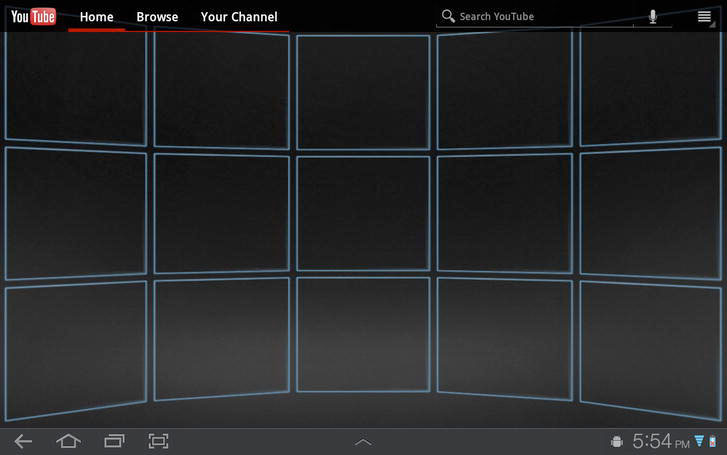
RenderScript đã được giới thiệu trong vesion Honeycomb(API 11), RenderScript là một framework mới được đặt mục tiêu vào tính toán hoạt động và việc kết xuất 3D với hiệu suất cao. Trong khi RenderScript đã được sử dụng trong Android 2.1 (Éclair), Honeycomb là phiên bản android đầu tiên nơi mà RenderScript được cung cấp công khai.

Sau phần tổng quan ngắn về RenderScript, bạn sẽ học cách tạo một script đơn giản và cách sử dụng nó vào ứng dụng Java của mình. Sau đó, bạn sẽ được học cách thực hiện kết xuất hoạt động từ tập lệnh của bạn và cách truy cập dữ liệu tập lệnh từ Java. Chúng ta sẽ đánh giá một trong những ứng dụng mẫu được cung cấp trong Android SDK, HelloCompute và so sánh hai cách triển khai của cùng một ứng dụng, một cách dựa trên Honeycomb APIs và cách còn lại dựa trên Ice Cream Sandwich (Android 4.0) APIs. Để kết luận chương này, chúng ta sẽ đánh giá các tệp tiêu đề RenderScrip cho cả Honeycomb và Ice Cream Sandwich và các chức năng bạn có thể sử dụng trong scripts của bạn.

**LƯU Ý**: Chương này không phải là hướng dẫn đầy đủ về kết xuất 3D. Ví dụ, một người nên có hiểu biết về OpenGL Shading Language (GLSL) cũng như để sử dụng các tính năng nâng cao của RenderScript. Toàn bộ sách dành riêng cho OpenGL, GLSL và việc thành thạo kết xuất 3D có thể mất nhiều năm thực hành.

# OverView

Không có nhiều ứng dụng hiện đang sử dụng RenderScript. Vì một bức tranh có giá trị hàng nghìn từ, hãy xem các ví dụ về nơi RenderScript được sử dụng. Có lẽ ví dụ phổ biến nhất là ứng dụng YouTube và chế độ xem băng chuyền của nó, như trong Hình 9-1. Để rõ ràng, ảnh chụp màn hình được chụp khi video chưa được tải và bạn có thể xem mỗi hình chữ nhật trông như thế nào (mỗi hình chữ nhật sau đó sẽ chứa một hình thu nhỏ video). Một ví dụ khác, Balls, có sẵn dưới dạng code mẫu trong Android SDK và sử dụng thông tin và vật lý của cảm biến để tính toán vị trí của các quả bóng trên màn hình, như trong Hình 9–2.



Hình 1. 1 Chế độ xem băng chuyền YouTube



Hình 1. 2 Ứng dụng mẫu Balls

**LƯU Ý**: Để tạo một dự án từ code mẫu (ví dụ: Balls), hãy tạo một dự án Android mới trong Eclipse và chọn “Tạo dự án từ mẫu hiện có” hoặc chỉ cần tạo một dự án mẫu Android mới (tùy chọn khả dụng với các phiên bản mới hơn của plugin Eclipse ADT). Các mẫu RenderScript chỉ khả dụng khi bạn chọn 3.0 hoặc cao hơn làm mục tiêu. Bạn có thể tải xuống ví dụ về một băng chuyền từ <http://code.google.com/p/android-ui-utils/downloads/list>

RenderScript sử dụng C99 làm ngôn ngữ của nó nhưng không có nghĩa là thay thế hoàn toàn ứng dụng Java của bạn. Thay vào đó, ứng dụng của bạn sẽ sử dụng RenderScript cho một số phần nhất định và Java cho những phần khác. Khi bạn sử dụng RenderScript, các Scrpit của bạn sẽ được biên dịch sang mã bít trên máy chủ Low-Level Virtual Machine (LLVM) (ví dụ: PC Windows của bạn trên đó bạn đã cài đặt Eclipse) và mã bit LLVM sau đó sẽ được biên dịch thành mã gốc trên chính thiết bị Android. Về mặt đó, nó tương tự như mã Java của bạn được biên dịch sang Dalvik bytecode trên máy chủ, mã này có thể được biên dịch thành mã gốc trên thiết bị bằng trình biên dịch Dalvik Just-In-Time.

Mã gốc sau đó sẽ được lưu vào bộ nhớ đệm trên thiết bị của bạn, vì vậy các lần thực thi tiếp theo của script sẽ nhanh hơn vì script đã được biên dịch sang mã gốc. Khi quá trình biên soạn mã bit thành mã gốc xảy ra trên thiết bị, mã gốc thu được có thể được tối ưu hóa cho thiết bị hoặc thậm chí là mô-đun phần cứng (ví dụ: GPU) và bạn với tư cách là nhà phát triển sẽ không phải lo lắng về các kiến ​​trúc và tính năng khác nhau. Đây là một sự khác biệt lớn với NDK, ví dụ trong đó code của bạn phải kiểm tra xem NEON có khả dụng hay không vfa sẽ đơn giản hóa quy trình làm việc của bạn một cách đáng kể.

**LƯU Ý**: Để biết thêm thông tin về LLVM, hãy truy cập http://llvm.org. Bạn có thể tải xuống và cài đặt các công cụ LLVM khác nhau từ đó. LLVM không phải là công nghệ dành riêng cho Android. Trên thực tế, nó được sử dụng bởi Apple’s Xcode 4.

Trước khi nói chi tiết hơn về những gì bạn có thể làm trong RenderScript, chúng tôi sẽ hướng dẫn cách tạo một kịch bản RenderScript cơ bản trong một dự án. Vì trình giả lập không hỗ trợ tất cả các tính năng cần thiết của RenderScript nên bạn sẽ cần một thiết bị Android để sử dụng RenderScript. Bạn sẽ thấy rằng thường thì một kịch bản RenderScript được gọi đơn giản là RenderScript.

# Hello World

Như mọi khi, khi được giới thiệu với một framework mới, bản năng đầu tiên của bạn là nói “Hello” thế giới từ tập lệnh RenderScript. Để tạo một tập lệnh, chỉ cần tạo một tệp trong dự án Android của bạn và gọi nó là helloworld.rs. Listing 9–1 cho thấy cách chúng ta có thể tạo một hàm đơn giản sẽ xuất ra chuỗi “Hello, World” bằng cách sử dụng một hàm gỡ lỗi RenderScrip đơn giản.

Listing 9–1. “Hello, World” Script

#pragma version(1)

#pragma rs java\_package\_name(com.apress.proandroid.ch9)

void hello\_world() {

rsDebug("Hello, World", 0); // 0 là bởi vì chúng ta không thể chỉ gửi một //chuỗi đến debug output...

}

Dòng đầu tiên khai báo phiên bản RenderScript mà tập lệnh của bạn sử dụng. Dòng thứ 2 khai báo tên gói của phản ánh Java của tập lệnh này. Khi bạn xây dựng dự án Android của mình, nhiều thứ sẽ xảy ra và các công cụ xây dựng sẽ tạo ra ba tệp sau:

* ScriptC\_helloworld.java (in gen/ directory)
* helloworld.d (in gen/ directory)
* helloworld.bc (in res/raw/ directory)

Listing 9-2 hiển thị nội dung của tệp ScriptC\_helloword.java được tạo tự động. Bạn có thể tìm thấy tệp đó trong thư mục gen của dự án, chính là thư mục mà R.java được tạo. Tệp này là một phần của lớp phản ánh của tập lệnh của bạn và chứa định nghĩa của lớp sẽ cho phép bạn gọi các hàm từ tập lệnh của mình và giao tiếp với tập lệnh của bạn nói chung. Trong cùng thư mục đó, bạn cũng sẽ tìm thấy helloworld.d, chỉ liệt kê các phần phụ thuộc cho tệp helloworld.bc (nghĩa là, helloworld.bc sẽ phải được tạo lại nếu bất kỳ phần phụ thuộc nào được liệt kê trong helloworld.d được sửa đổi).

Listing 9–2. ScriptC\_helloworld.java

/\*

\* Copyright (C) 2011 The Android Open Source Project

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

/\*

\* This file is auto-generated. DO NOT MODIFY!

\* The source RenderScript file: E:\ws\Chapter

9\src\com\apress\proandroid\ch9\helloworld.rs

\*/

package com.apress.proandroid.ch9;

import android.renderscript.\*;

import android.content.res.Resources;

/\*\*

\* @hide

\*/

public class ScriptC\_helloworld extends ScriptC {

// Constructor

public ScriptC\_helloworld(RenderScript rs, Resources resources, int id){

super(rs, resources, id);

}

private final static int mExportFuncIdx\_hello\_world = 0;

public void invoke\_hello\_world() {

invoke(mExportFuncIdx\_hello\_world);

}

}

Ba điều nổi bật trong tệp được tạo tự động này:

* Tên gói là tên được xác định trong tập lệnh với #pragma
* Công cụ xây dựng
* Hàm invoke\_hello\_world

Các phần của tệp được tạo tự động này, mặc dù chúng quan trọng nhưng không liên quan đến bạn. Ví dụ: hằng số mExportFuncIdx\_hello\_world riêng trong Listing 9–2 đề cập đến chỉ mục của một hàm trong tập lệnh, nhưng giá trị thực của nó không quan trọng đối với bạn hoặc ứng dụng của bạn. Để sử dụng tập lệnh, bạn sẽ phải tạo một phiên bản của ScriptC\_helloworld trong code Java của mình, sau đó sẽ cho phép bạn gọi hàm hello\_world của script thông qua phương thức invoke\_hello\_world được phản ánh, như được hiển thị trong Listing 9–3.

Listing 9–3. Calling RenderScript Function

public class Chapter9Activity extends Activity {

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

HelloWorldRenderScript();

}

private void HelloWorldRenderScript() {

RenderScript rs = RenderScript.create(this); // cần một ngữ cảnh làm tham số

// script được tạo bằng bitcode helloworld trong res/raw/helloworld.bc

ScriptC\_helloworld helloworldScript = new ScriptC\_helloworld(rs, getResources(),

R.raw.helloworld);

// bây giờ chúng ta có thể gọi hàm hello\_world của script bằng phương thức phản ánh

helloworldScript.invoke\_hello\_world();

}

}

Việc thực thi code này sẽ dẫn đến “Hello, World 0 0x0” được hiển thị trong logcat với thẻ “RenderScript”.

**LƯU Ý:** Sử dụng phối cảnh DDMS trong Eclipse để quan sát rằng các luồng liên quan đến RenderScript hiện đang chạy trong ứng dụng (với tên RSMessageThread). Bạn cũng có thể thấy kết quả sau trong logcat: “RS Launching thread(s), số CPU được báo cáo là 2” (nếu thiết bị của bạn có 2 lõi). Khi bạn tìm hiểu về RenderScript, hãy quan sát các kết quả đầu ra được tạo bằng thẻ “RenderScript”.

Mặc dù file helloworld.bc rất quan trọng vì nó chứa script bitcode, nhưng nội dung thực tế của nó không có giá trị lớn đối với bạn. Tất cả những gì bạn nên quan tâm là thực tế rằng mã bit LLVM là bản đại diện độc lập với nền tảng cho tập lệnh của bạn. Điều đó đang được nói, có thể tháo rời tệp này và chuyển đổi nó sang ngôn ngữ hợp ngữ LLVM mà con người có thể đọc được bằng cách sử dụng trình tháo gỡ LLVM llvm-dis (có sẵn như một phần của bộ LLVM). Listing 9–4 hiển thị phiên bản hợp ngữ LLVM của helloworld.bc

Listing 9–4. LLVM Assembly Language (helloworld.bc)

; ModuleID = '/home/herve/helloworld.bc'

target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-

f64:64:64-v64:64:64-v128:128:128-a0:0:64-n32"

target triple = "armv7-none-linux-gnueabi"

@.str = private constant [13 x i8] c"Hello, World\00"

define void @hello\_world() nounwind {

tail call void @\_Z7rsDebugPKci(i8\* getelementptr inbounds ([13 x i8]\* @.str, i32 0,

i32 0), i32 0) nounwind

ret void

}

declare void @\_Z7rsDebugPKci(i8\*, i32)

!#pragma = !{!0, !1}

!#rs\_export\_func = !{!2}

!0 = metadata !{metadata !"version", metadata !"1"}

!1 = metadata !{metadata !"java\_package\_name", metadata !"com.apress.proandroid.ch9"}

!2 = metadata !{metadata !"hello\_world"}

Ví dụ này không thực hiện bất kỳ kết xuất nào bằng RenderScript. Trên thực tế, bố cục của hoạt động vẫn được xác định bằng cách sử dụng XML và nội dung hoạt động vẫn được đặt từ một tài nguyên bố cục đã được thổi phồng khi setContentView () được gọi. Bây giờ bạn đã biết cách tạo một kịch bản cơ bản, đã đến lúc xem cách kết xuất có thể được thực hiện với RenderScript.

# Hello Rendering

Rendering với RenderScript phức tạp hơn một chút do với việc chỉ gọi một hàm script, và yêu cầu thêm một chút công việc trước khi bạn có thể nhìn thấy bất cứ thứ gì trên màn hình. Bạn sẽ cần đến:

* Tạo một tập lệnh thực hiện rendering
* Tạo một đối tượng context RenderScriptGL
* Tạo một lớp extends RSSurfaceView
* Đặt RSSurfaceView của bạn làm nội dụng cho activity của bạn

## Tạo một Rendering script

Tập lệnh rendering đầu tiên của chúng tôi sẽ cực kì đơn giản vì nó chỉ thay đổi nền màu đến một số màu ngẫu nhiên. Tập lệnh được hiển thị trong Listing 9-5 và nằm trong một tệp có tên là hellorendering.rs. Tên của tệp renderScript rất quan trọng vì chúng xác định tên tài nguyên (trong trường hợp đó là R.raw.hellorendering, được định nghĩa đơn giản là một số nguyên trong R.java).

#include "rs\_graphics.rsh"

// Được gọi tự động khi script được tạo

void init() {

// làm những gì bạn cần vào đây...

}

int root() {

float red = rsRand(1.0f);

float green = rsRand(1.0f);

float blue = rsRand(1.0f);

// Xóa màu nền bằng màu ngẫu nhiên

rsgClearColor(red, green, blue, 1.0f); // alpha is 1.0f, i.e. opaque

// 50 frames per second = 20 milliseconds per frame

return 20;

}

Hai dòng đầu tiên của tệp giống như trong Liệt kê 9–1. Dòng sau bao gồm tệp tiêu đề rs\_graphics.rsh, trong đó rsgClearColor () được định nghĩa. Tệp tiêu đề được tự động bao gồm ngoại trừ rs\_graphics.rsh, vì vậy bạn sẽ phải bao gồm tệp đó một cách rõ ràng khi bạn sử dụng các hàm RenderScript đồ họa như rsgClearColor () hoặc rsgBindFont ().

Hàm init () ở đây chỉ dành cho mục đích cung cấp thông tin vì việc triển khai nó trống. Hàm init () là tùy chọn và nếu có, sẽ được gọi tự động khi tập lệnh được tạo. Điều này cho phép bạn thực hiện bất kỳ khởi tạo nào trước khi các quy trình khác được thực thi. Một tập lệnh không rendering, giống như tập lệnh được hiển thị trong Liệt kê 9–1, cũng có thể có hàm init ().

Hàm root() là nơi rendring thực sự xảy ra. Việc triển khai ở đây rất đơn giản vì tất cả những gì nó làm là xóa phông nền bằng một màu ngẫu nhiên. Tuy nhiên, điều thú vị là giá trị trả về cảu hàm nó chỉ định tần xuất hiển thị sẽ xảy ra. Trong trường hợp cụ thể này, chings trả về 20, có nhĩa khung hình phải được cập nhập mỗi 20 mili giây (nghĩa là tốc độ khung nhìn là 50 khung nhìn trên mỗi giây)

**Lưu ý:** hàm init() và root() là các hàm dành riêng và không dẫn đến các phương thức invoke\_init() và invoke\_root() được tạo trong lớp reflected (ScriptC\_hellorendering.java trong

trường hợp này).

## Tạo một RenderScriptGL Context

Bây giờ tập lệnh rendering đã hoàn tất, bạn sẽ cần một đối tượng RenderScriptGL context. Ở đây đối tượng RenderScriptGL context đã được tạo khi RSSurface view của phưng thức surfaceChanged() được gọi. Việc triển khai HelloRenderingView được hiển thị trong Listing 9–6.

Listing 9–6: HelloRenderingView.java

public class HelloRenderingView extends RSSurfaceView {

public HelloRenderingView(Context context) {

super(context);

}

private RenderScriptGL mRS;

private HelloRenderingRS mRender; // helper class

public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {

super.surfaceChanged(holder, format, w, h);

if (mRS == null) {

RenderScriptGL.SurfaceConfig sc = new RenderScriptGL.SurfaceConfig();

mRS = createRenderScriptGL(sc);

mRender = new HelloRenderingRS();

mRender.init(mRS, getResources());

}

mRS.setSurface(holder, w, h);

}

@Override

protected void onDetachedFromWindow() {

if(mRS != null) {

mRS = null;

destroyRenderScriptGL();

}}}

Như bạn có thể thấy trong 9–6, lớp HelloRenderingRS helper cũng được sử dụng.

## Extending RSSurfaceView

Việc triển khai lớp này được hiển thị trong Listing 9–7

Listing 9–7. HelloRenderingRS.java

public class HelloRenderingRS {

public HelloRenderingRS() {

}

private Resources mRes;

private RenderScriptGL mRS;

private ScriptC\_hellorendering mScript;

public void init(RenderScriptGL rs, Resources res) {

mRS = rs;

mRes = res;

mScript = new ScriptC\_hellorendering(mRS, mRes, R.raw.hellorendering);

mRS.bindRootScript(mScript);

}

}

Trong lớp này, chúng ta tạo tập lệnh ScriptC\_hellorendering thực và liên kết nó với đối tượng RenderScriptGL. Nếu không có lệnh gọi đến bindRootScript(), sẽ không có gì thực sự nhận được

được hiển thị dưới dạng hàm root () của tập lệnh sẽ không được gọi. Nếu bạn thấy mình đang nhìn chằm chằm vào màn hình đen khi bạn đang mong đợi một số rendring xảy ra, trước tiên hãy kiểm tra xem bạn có quên gọi bindRootScript() không.

## Cài đặt Content View

Bây giờ tất cả các tệp này đều tồn tại, bạn có thể tạo một phiên bản của HelloRenderingView và đặt

Nội dung của hoạt động đối với trường hợp này, như được hiển thị trong Listingn 9–8.

Listing 9–8. Activity

public class Chapter9Activity extends Activity {

static final String TAG = "Chapter9Activity";

private HelloRenderingView mHelloRenderingView;

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

//setContentView(R.layout.main); // we don’t use the XML layout anymore here so

//we comment it out

HelloWorldRenderScript();

mHelloRenderingView = new HelloRenderingView(this);

setContentView(mHelloRenderingView);

}

@Override

protected void onPause() {

super.onPause();

mHelloRenderingView.pause(); // to pause the rendering thread

}

@Override

protected void onResume() {

super.onResume();

mHelloRenderingView.resume(); // to resume the rendering thread

}

private void HelloWorldRenderScript() {

// see Listing 9–3 for implementation

}

}

Mặc dù bây giờ hầu hết tệp này có vẻ nhỏ nhặt đối với bạn, nhưng có một điều trong tệp này đáng lưu ý: RSSurfaceView phải được thông báo khi hoạt động bị tạm dừng và tiếp tục.

Nếu không có các lệnh gọi tới RSSurfaceView.pause () và RSSurfaceView.resume (), chuỗi kết xuất sẽ không nhận biết được trạng thái của hoạt động và sẽ tiếp tục hiển thị mọi thứ ngay cả khi hoạt động không nhất thiết phải hiển thị.

Trong Android 4.0 có 2 tính năng mới quan trọng đã được thêm:

* Ứng dụng của bạn có thể sử dụng allocation làm bộ đệm ngoài màn hình bằng cách đặt cờ Allocation.USAGE\_GRAPHICS\_RENDER\_TARGET khi tạo đối tượng Allocation
* Bạn có thể sử dụng RSTextureView trong layout của mình (cùng với các view khác). Trong khi một RSSurfaceView tạo một cửa sổ riêng biệt, một RSTextureView thì không.

## Adding Variables to Script

Hãy thay đổi Listing 9–5 một chút để tìm hiểu thêm về reflected layer. Cho đến nay, chúng ta chỉ thấy quy trình hello\_world từ tập lệnh được reflected như thế nào trong Listing 9–2. Ở đây, chúng ta sẽ thêm nhiều biến hơn trong tập lệnh, mà chúng ta sẽ gọi là hellorendering2.rs, như được hiển thị trong Listing 9–9.

Listing 9–9. Changing Background Color (Second Version)

#pragma version(1)

#pragma rs java\_package\_name(com.apress.proandroid.ch9)

#include "rs\_graphics.rsh"

// we removed init() as it was empty anyway

float red = 0.0f; // initialized to 1.0f

float green; // purposely not initialized

static float blue; // purposely not initialized, static

const float alpha = 1.0f; // constant

int root() {

// clear the background color

blue = rsRand(1.0f);

rsgClearColor(red, green, blue, alpha);

// 50 frames per second = 20 milliseconds per frame

return 20;

}

Như bạn có thể thấy, bốn biến đỏ, lục, lam và alpha đều được định nghĩa khác nhau. Mặc dù điều này không nhất thiết phải có ý nghĩa đối với tập lệnh thực tế, nhưng nó cho phép chúng ta xem cách tạo reflected layer. Listing 9–10 hiển thị lớp ScriptC\_hellorendering2.java.

Listing 9–10. ScriptC\_hellorendering2.java

package com.apress.proandroid.ch9;

import android.renderscript.\*;

import android.content.res.Resources;

/\*\*

\* @hide

\*/

public class ScriptC\_hellorendering2 extends ScriptC {

// Constructor

public ScriptC\_hellorendering2(RenderScript rs, Resources resources, int id) {

super(rs, resources, id);

mExportVar\_red = 0f;

mExportVar\_alpha = 1f;

}

private final static int mExportVarIdx\_red = 0;

private float mExportVar\_red;

public void set\_red(float v) {

mExportVar\_red = v;

setVar(mExportVarIdx\_red, v);

}

public float get\_red() {

return mExportVar\_red;

}

private final static int mExportVarIdx\_green = 1;

private float mExportVar\_green;

public void set\_green(float v) {

mExportVar\_green = v;

setVar(mExportVarIdx\_green, v);

}

public float get\_green() {

return mExportVar\_green;

}

private final static int mExportVarIdx\_alpha = 2;

private float mExportVar\_alpha;

public float get\_alpha() {

return mExportVar\_alpha;

}

}

Biến màu đỏ toàn cục đã được định nghĩa trong tập lệnh và được khởi tạo bằng 0. Do đó, lớp reflected xác định một phần tử dấu phẩy động mExportVar\_red private, được khởi tạo bằng 0 trong hàm tạo, cũng như hai phương thức để đặt và nhận giá trị màu đỏ: set\_red () và get\_red ().

Biến toàn cục màu xanh lá cây trong tập lệnh rất giống với biến màu đỏ, ngoại trừ thực tế là nó không được khởi tạo. Do đó, lớp reflected cũng xác định một phần tử dấu phẩy động mExportVar\_green private cũng như hai phương thức, set\_green() và get\_green(). Tuy nhiên, mExportVar\_green không được khởi tạo trong hàm tạo.

Biến màu xanh lam được định nghĩa là biến static, có nghĩa là nó không được xuất ra bên ngoài tập lệnh. Do đó, lớp được refected không xác định bất kỳ phần tử hoặc phương thức nào liên quan đến thành phần màu xanh lam của tập lệnh.

Cuối cùng, phần tử alpha được định nghĩa là một hằng số, và do đó lớp refected chứa phần khởi tạo phần tử mExportVar\_alpha trong phương thức khởi tạo và chỉ định nghĩa một phương thức, get\_alpha (), để truy cập giá trị của phần tử. Vì phần tử alpha là hằng số, nên không cần xác định bất kỳ phương thức set\_alpha () nào trong Java.

**LƯU Ý:** Con trỏ toàn cục được xác định trong tập lệnh sẽ dẫn đến phương thức bind\_pointer\_name () trong lớp reflected thay vì phương thức set\_pointer\_name().

Như bạn có thể thấy, một phương thức get () trong lớp reflected trả về giá trị cuối cùng được đặt trong Java. Ví dụ: get\_green () chỉ trả về ExportVar\_green, chỉ có thể được sửa đổi bằng cách gọi đến set\_green (). Về cơ bản, điều này có nghĩa là nếu tập lệnh sửa đổi giá trị của biến toàn cục màu xanh lá cây, thì thay đổi sẽ không truyền đến lớp Java. Đây là một chi tiết rất quan trọng cần lưu ý.

Bây giờ chúng ta sẽ xem xét một trong những ví dụ đơn giản do Android cung cấp dưới dạng mã mẫu để tìm hiểu thêm về một trong những chức năng RenderScript quan trọng nhất.

# HelloCompute

Ứng dụng mẫu HelloCompute của Android chuyển đổi hình ảnh màu thành đen trắng. Listing 9–11 cho thấy việc triển khai tập lệnh

Listing 9–11. RenderScript mono.rs

/\*

\* Copyright (C) 2011 The Android Open Source Project

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* bạn không thể sử dụng tệp này ngoại trừ việc tuân thủ License.

\* Bạn có thể nhận được một bản sao của License tại

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

#pragma version(1)

#pragma rs java\_package\_name(com.android.example.hellocompute)

rs\_allocation gIn; rs\_allocation gOut; rs\_script gScript;

const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

void root(const uchar4 \*v\_in, uchar4 \*v\_out, const void \*usrData, uint32\_t x, uint32\_t y) {

float4 f4 = rsUnpackColor8888(\*v\_in);

float3 mono = dot(f4.rgb, gMonoMult);

\*v\_out = rsPackColorTo8888(mono); }

void filter() {

rsForEach(gScript, gIn, gOut, 0); // first: script; second: input allocation; third:

output allocation }

Hàm root () chuyển đổi một pixel thành đen trắng. Nó làm như vậy trong bốn bước:

1. Giá trị ARGB (\* v\_in) được chuyển đổi thành vectơ dấu phẩy động.
2. root () thực hiện tích số chấm của f4.rgb và gMonoMult, trả về một giá trị dấu phẩy động duy nhất (độ chói).
3. Cả ba mục trong vectơ đơn sắc đều được cho cùng một giá trị (kết quả của tích số chấm, tức là độ chói).
4. Vectơ đơn sắc được chuyển đổi thành giá trị ARGB được sao chép vào \* v\_out.

**LƯU Ý:** Các giá trị gMonoMult được xác định bởi tiêu chuẩn NTSC.

Để tập lệnh chuyển đổi toàn bộ bức ảnh sang đen trắng, nó phải được yêu cầu thực thi hàm root () trên mỗi pixel. Đây là những gì hàm filter () đang thực hiện bằng cách gọi hàm rsForEach (). Hàm rsForEach () thực thi tập lệnh (tham số đầu tiên) cho từng cặp phần tử trong phân bổ đầu vào và đầu ra (tham số thứ hai và thứ ba). Tham số thứ tư, usrData, được đặt thành 0 ở đây và bị bỏ qua bởi hàm root ().

# Allocations

Vì rsForEach () sử dụng allocations làm tham số, allocations phải được tạo trong Java và sau đó phải được chuyển cho tập lệnh. Đây là những gì createScript () trong HelloCompute.java thực hiện trong số những thứ khác, như được hiển thị trong Liệt kê 9–12.

Listing 9–12. Allocations

private RenderScript mRS;

private Allocation mInAllocation; // input allocation

private Allocation mOutAllocation; // output allocation

private ScriptC\_mono mScript;

...

private void createScript() {

mRS = RenderScript.create(this);

mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn,

Allocation.MipmapControl.MIPMAP\_NONE,

Allocation.USAGE\_SCRIPT);

mOutAllocation = Allocation.createTyped(mRS, mInAllocation.getType());

mScript = new ScriptC\_mono(mRS, getResources(), R.raw.mono);

mScript.set\_gIn(mInAllocation);

mScript.set\_gOut(mOutAllocation);

mScript.set\_gScript(mScript);

mScript.invoke\_filter();

mOutAllocation.copyTo(mBitmapOutRS);

}

Trong ví dụ trên, allocations của bộ nhớ đầu vào được tạo từ một bitmap và allocations sẽ được sử dụng bởi tập lệnh. Các cách sử dụng có thể khác bao gồm nguồn kết cấu (USAGE\_GRAPHICS\_TEXTURE) và lưới đồ họa (USAGE\_GRAPHICS\_VERTEX). Bạn có thể xem lại các phương pháp khác nhau được xác định trong class Allocations để tạo allocations. Allocations có thể là một chiều (ví dụ: mảng), hai chiều (ví dụ: bitmap) hoặc ba chiều.

## rsForEach

Hàm rsForEach tồn tại ở ba cách khác nhau:

* rsForEach(rs\_script script, rs\_allocation input, rs\_allocation output)

(chỉ dành cho Android 4.0 trở lên)

* rsForEach(rs\_script script, rs\_allocation input, rs\_allocation output, const void \* usrData)
* rsForEach(rs\_script script, rs\_allocation input, rs\_allocation output, const void \* usrData, const rs\_script\_call\_t\*)

Như chúng ta đã thấy trong Listing 9–11, tham số đầu tiên là một tập lệnh. Nó sẽ xác định hàm root () nào để thực thi. Tham số thứ hai và thứ ba chỉ định allocations input và output. Tham số thứ tư, nếu có, là một số dữ liệu người dùng riêng tư có thể được sử dụng bởi tập lệnh. Android sẽ không sử dụng giá trị này và nó có thể được đặt thành 0 hoặc bất kỳ giá trị nào khác khi không được sử dụng bởi hàm root (). Tham số thứ năm, khi có mặt, sẽ chỉ định cách thực thi hàm root (). Listing 9–13 hiển thị các định nghĩa của cấu trúc rs\_script\_call\_t.

Listing 9–13. rs\_script\_call\_t

enum rs\_for\_each\_strategy

{RS\_FOR\_EACH\_STRATEGY\_SERIAL, RS\_FOR\_EACH\_STRATEGY\_DONT\_CARE, RS\_FOR\_EACH\_STRATEGY\_DST\_LINEAR, RS\_FOR\_EACH\_STRATEGY\_TILE\_SMALL, RS\_FOR\_EACH\_STRATEGY\_TILE\_MEDIUM, RS\_FOR\_EACH\_STRATEGY\_TILE\_LARGE

};

typedef struct rs\_script\_call {

enum rs\_for\_each\_strategy strategy;

uint32\_t xStart;

uint32\_t xEnd;

uint32\_t yStart;

uint32\_t yEnd; uint32\_t zStart;

uint32\_t zEnd; uint32\_t arrayStart;

uint32\_t arrayEnd;

} rs\_script\_call\_t;

Bạn có thể định cấu hình chiến lược cũng như các thông số bắt đầu và kết thúc (cho tất cả các thứ nguyên của allocations). Chiến lược chỉ là một gợi ý và không có gì đảm bảo việc triển khai sẽ tuân theo mệnh lệnh.

Để thuận tiện, Android 4.0 (Ice Cream Sandwich) xác định phương thức Script.forEach (). Mặc dù bạn không nên gọi phương thức này trực tiếp trong mã của mình, nhưng nó đang được sử dụng trong mã được phản ánh. Nhờ phương pháp mới này, tập lệnh mono.rs có thể được đơn giản hóa và Liệt kê 9–14 cho thấy việc triển khai tập lệnh khi mục tiêu xây dựng được đặt thành Android 4.0.

**Listing 9–14.** RenderScript mono.rs (Android 4.0 Version)

#pragma version(1)  
#pragma rs java\_package\_name(com.example.android.rs.hellocompute)

const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

void root(const uchar4 \*v\_in, uchar4 \*v\_out) {

float4 f4 = rsUnpackColor8888(\*v\_in);

float3 mono = dot(f4.rgb, gMonoMult);

\*v\_out = rsPackColorTo8888(mono);

}

Như bạn có thể thấy, hàm filter () đã bị loại bỏ và các biến gIn, gOut và gScript cũng vậy. Để hiểu tại sao có thể loại bỏ chức năng này trong ứng dụng mẫu Android 4.0, người ta phải nhìn vào lớp phản chiếu được tạo ra khi dự án được xây dựng. Listing 9–15 hiển thị lớp phản chiếu mới.

Listing 9–15. ScriptC\_mono.java (Android 4.0 Version)

package com.example.android.rs.hellocompute;

import android.renderscript.\*;

import android.content.res.Resources;

/\*\*

\* @hide

\*/

public class ScriptC\_mono extends ScriptC {

// Constructor

public ScriptC\_mono(RenderScript rs, Resources resources, int id) {

super(rs, resources, id);

\_\_U8\_4 = Element.U8\_4(rs);

}

private Element \_\_U8\_4;

private final static int mExportForEachIdx\_root = 0;

public void forEach\_root(Allocation ain, Allocation aout) {

// check ain

if (!ain.getType().getElement().isCompatible(\_\_U8\_4)) {

throw new RSRuntimeException("Type mismatch with U8\_4!");

}

// check aout

if (!aout.getType().getElement().isCompatible(\_\_U8\_4)) {

throw new RSRuntimeException("Type mismatch with U8\_4!");

}

// Verify dimensions

Type tIn = ain.getType();

Type tOut = aout.getType();

if ((tIn.getCount() != tOut.getCount()) ||

(tIn.getX() != tOut.getX()) ||

(tIn.getY() != tOut.getY()) ||

(tIn.getZ() != tOut.getZ()) ||

(tIn.hasFaces() != tOut.hasFaces()) ||

(tIn.hasMipmaps() != tOut.hasMipmaps())) {

throw new RSRuntimeException("Dimension mismatch between input and output parameters!");

}

forEach(mExportForEachIdx\_root, ain, aout, null);

}

}

Rõ ràng là phương thức invoke\_filter () không xuất hiện trong lớp được phản ánh vì filter () không được định nghĩa trong script nữa. Vì các biến gIn, gOut và gScript đã bị loại bỏ, tất cả các phương thức set / get trong lớp được phản ánh cũng biến mất. Phương thức quan trọng bây giờ là forEach\_root (), được gọi trong createScript () trong HelloCompute.java như được hiển thị trong Listing 9–16 (phiên bản Honeycomb được hiển thị trong Listing 9–12).

Listing 9–16. createScript() Method (Android 4.0 Version)

private void createScript() {

mRS = RenderScript.create(this);

mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn, Allocation.MipmapControl.MIPMAP\_NONE,

Allocation.USAGE\_SCRIPT); mOutAllocation = Allocation.createTyped(mRS, mInAllocation.getType());

mScript = new ScriptC\_mono(mRS, getResources(), R.raw.mono);

mScript.forEach\_root(mInAllocation, mOutAllocation);

mOutAllocation.copyTo(mBitmapOut);

}

**LƯU Ý**: Việc phân bổ phải tương thích (ví dụ: cùng kích thước ở tất cả các thứ nguyên).

Như bạn có thể thấy, createScript () bây giờ chỉ cần gọi phương thức forEach\_root () từ lớp được phản ánh. Vì phương thức Script.forEach () không tồn tại trong Honeycomb, bạn có thể muốn tránh sử dụng phương thức đó và thay vào đó sử dụng cách hoạt động của Honeycomb ban đầu. Mặc dù phức tạp hơn nhưng nó cũng cho phép ứng dụng của bạn tương thích với các thiết bị Android 3.x.

## Performance

Vì RenderScript được nhắm mục tiêu vào các hoạt động tính toán và kết xuất 3D hiệu suất cao, điều quan trọng là phải so sánh hiệu suất của một tập lệnh với đối tác Java của nó để cung cấp cho bạn ý tưởng về mức lợi nhuận bạn có thể đạt được với RenderScript. Với mục đích này, một triển khai Java đơn giản của bộ lọc đen trắng được cung cấp trong Listing 9–17.

Listing 9–17. Java Filter

int w = mBitmapIn.getWidth();

int h = mBitmapIn.getHeight();

int size = w \* h;

int[] pixels = new int[size];

mBitmapIn.getPixels(pixels, 0, w, 0, 0, w, h); // we get all the pixels (as 32-bit integer values)

for (int i = 0; i < size; i++) {

int c = pixels[i]; // 0xAARRGGBB

// we extract red, green and blue components (each one in [0, 255] range) int r = (c >> 16) & 0xFF;

intg=(c>>8) &0xFF;

intb= c &0xFF;

// approximation of the formula using integer arithmetic r \*= 76;

g \*= 151;

b \*= 29;

int y = (r + g + b) >> 8; // luminance

pixels[i] = y | (y << 8) | (y << 16) | (c & 0xFF000000);

}

mBitmapOut.setPixels(pixels, 0, w, 0, 0, w, h); //set the output bitmap’s pixels

Trong khi phiên bản RenderScript của bộ lọc mất khoảng 115 mili giây để hoàn thành trên Galaxy Tab 10.1 (sử dụng tài nguyên bitmap được cung cấp trong ứng dụng mẫu làm đầu vào), phiên bản Java chỉ mất 48 mili giây! Phiên bản không có JIT (android: vmSafeMode = “true” trong tệp kê khai của ứng dụng) mất khoảng 130 mili giây để hoàn thành. Kết quả nghe có vẻ đáng ngạc nhiên nhưng người ta phải xem xét chi phí tạo tập lệnh, allocations và hoạt động dấu phẩy động. Một tập lệnh phức tạp hơn có thể hoạt động tốt hơn đối tác Java của nó và có thể được hưởng lợi trong tương lai từ các đơn vị xử lý mạnh mẽ hơn. Ví dụ: trong tương lai, một tập lệnh có thể chạy trên GPU thay vì CPU, có thể tận dụng sự hỗ trợ tốt hơn để thực thi song song. Không phải tất cả các tập lệnh sẽ chậm hơn, nhưng ví dụ này cho thấy rằng việc sử dụng RenderScript không phải lúc nào cũng làm cho mã của bạn nhanh hơn.

# Native Renderscript Apis

Mã RenderScript của bạn có quyền truy cập vào một bộ API giới hạn vì nó không thể chỉ gọi các API từ NDK hoặc thư viện C. Các API RenderScript được định nghĩa trong sáu tệp tiêu đề, nằm trong platform/android-xx/renderscript/include của SDK (ví dụ: trong đó xx là cấp API, ví dụ: 13):

* rs\_core.rsh
* rs\_types.rsh
* rs\_cl.rsh
* rs\_math.rsh
* rs\_graphics.rsh
* rs\_time.rsh

Trong Ice Cream Sandwich, các API RenderScript được định nghĩa trong 12 tệp tiêu đề. Ngoài sáu tệp tiêu đề ở trên, sáu tệp sau đã được thêm vào:

* rs\_allocation.rsh
* rs\_atomic.rsh
* rs\_debug.rsh
* rs\_matrix.rsh
* rs\_object.rsh
* rs\_quaternion.rsh

Một lần nữa, vì tài liệu trực tuyến cho RenderScript hiện tại khá ít ỏi, bạn nên xem lại các tệp này. Một trong những điểm khác biệt lớn giữa Honeycomb và Ice Cream Sandwich là việc bổ sung các nhận xét trong các tệp tiêu đề này. Do đó, bạn nên tham khảo các tệp tiêu đề RenderScript của Android 4.0 bất cứ khi nào có thể vì bạn sẽ tìm thấy nhiều thông tin trong đó hơn là trong các tệp tiêu đề Honeycomb.

## rs\_types.rsh

* Tệp tiêu đề này xác định các kiểu cơ bản được sử dụng trong RenderScript (ngoài các kiểu tiêu chuẩn C99 char, short, int, long và float):
* int8\_t
* int16\_t
* int32\_t
* int64\_t
* uint8\_t
* uint16\_t
* uint32\_t
* uint64\_t
* uchar (defined as uint8\_t)
* ushort (defined as uint16\_t)
* uint (defined as uint32\_t)
* ulong (defined as uint64\_t)

Ngoài các loại cơ bản này, tệp này còn xác định các loại vectơ:

* float2
* float3
* float4
* double2 (Android 4.0 and above)
* double3 (Android 4.0 and above)
* double4 (Android 4.0 and above)
* char2
* char3
* char4
* uchar2
* uchar3
* uchar4
* short2
* short3
* short4
* ushort2
* ushort3
* ushort4
* int2
* int3
* int4
* uint2
* uint3
* uint4

Vì khuôn khổ 3D sẽ không hoàn chỉnh nếu không có ma trận, một số loại ma trận cũng được xác định:

* rs\_matrix2x2 (square matrix of order 2)
* rs\_matrix3x3 (square matrix of order 3)
* rs\_matrix4x4 (square matrix of order 4)

**LƯU Ý:** Trong khi các loại vectơ được xác định cho cả vectơ dấu phẩy động và số nguyên (ví dụ: float2, ushort3 và int4), các loại ma trận chỉ sử dụng float.

Honeycomb MR1 (API cấp 12) đã giới thiệu rs\_quaternion (được định nghĩa là float4). Các API bổ sung cũng được giới thiệu để thao tác với loại mới này.

Như bạn có thể thấy, số lượng các loại tương đối nhiều, tuy nhiên mỗi loại này có thể được xem như một đại lượng vô hướng, một vectơ hoặc một ma trận.

Các kiểu sau đây cũng được định nghĩa trong rs\_types.rsh và là các xử lý mờ đối với các đối tác Java của chúng:

* rs\_element
* rs\_type
* rs\_allocation
* rs\_sampler
* rs\_script
* rs\_mesh
* rs\_program\_fragment
* rs\_program\_vertex
* rs\_program\_raster
* rs\_program\_store
* rs\_font

Tất cả các loại này cũng được hiển thị trong lớp Java trong gói android.renderscript:

* Byte2
* Byte3
* Byte4
* Double2 (Android 4.0 and above)
* Double3 (Android 4.0 and above)
* Double4 (Android 4.0 and above)
* Float2
* Float3
* Float4
* Int2
* Int3
* Int4
* Long2
* Long3
* Long4
* Matrix2f
* Matrix3f
* Matrix4f
* Short2
* Short3
* Short4
* Element
* Type
* Allocation
* Sampler
* Script
* Mesh
* ProgramFragment
* ProgramVertex
* ProgramRaster
* ProgramStore
* Font

## rs\_core.rsh

Tệp tiêu đề này xác định các chức năng sau trong Honeycomb:

* rsDebug
* rsPackColorTo8888
* rsUnpackColor8888
* rsMatrixSet
* rsMatrixGet
* rsMatrixLoadIdentity
* rsMatrixLoad
* rsMatrixLoadRotate
* rsMatrixLoadScale
* rsMatrixLoadTranslate
* rsMatrixLoadMultiply
* rsMatrixMultiply
* rsMatrixRotate
* rsMatrixScale
* rsMatrixTranslate
* rsMatrixLoadOrtho
* rsMatrixLoadFrustum
* rsMatrixLoadPerspective
* rsMatrixInverse
* rsMatrixInverseTranspose
* rsMatrixTranspose
* rsClamp

Các hàm ma trận đã được chuyển đến tệp tiêu đề chuyên dụng trong Android 4.0: rs\_matrix.rsh.

Khi quaternion type được giới thiệu trong Honeycomb MR1 (Android 3.1), các chức năng mới cũng được thêm vào rs\_core.rsh:

* rsQuaternionSet
* rsQuaternionMultiply
* rsQuaternionAdd
* rsQuaternionLoadRotateUnit
* rsQuaternionLoadRotate
* rsQuaternionConjugate
* rsQuaternionDot
* rsQuaternionNormalize
* rsQuaternionSlerp
* rsQuaternionGetMatrixUnit

Tệp tiêu đề này đã trải qua một lần biến hình trong Android 4.0. Trên thực tế, phiên bản Honeycomb của tệp này ở khắp nơi, chứa ma trận, quaternion, gỡ lỗi và một số hàm toán học. Tệp đã được tổ chức lại đáng kể trong Android 4.0 để chỉ xác định các chức năng sau:

* rsSendToClient
* rsSendToClientBlocking
* rsForEach

Ma trận, quaternion và các hàm gỡ lỗi đã được chuyển đến các tệp tương ứng mới của chúng (rs\_matrix.rsh, rs\_quaternion.rsh và rs\_debug.rsh) trong khi các hàm toán học được chuyển tự nhiên sang rs\_math.rsh. Vì các tệp tiêu đề được tự động đưa vào, thực tế là các hàm được chuyển từ tệp tiêu đề này sang tệp tiêu đề khác sẽ không ảnh hưởng đến tập lệnh của bạn.

## rs\_cl.rsh

Tệp tiêu đề này thực sự hơi khó đọc vì nó sử dụng nhiều macro để xác định các hàm. Nó chứa các API main“computer” và nếu bạn chơi trò lôi thôi thay vì tham gia các lớp học toán, bạn sẽ bị sốc.

Tệp tiêu đề này xác định các hàm toán học sau:

* acos arc cosine
* acosh inverse hyperbolic cosine
* acospi acos(x) / pi
* asin arc sine
* asinh inverse hyperbolic sine
* asinpi asin(x) / pi
* atan arc tangent
* atan2 arc tangent with 2 parameters
* atanh hyperbolic arc tangent
* atanpi atan(x) / pi
* atan2pi atan2(x,y) / pi
* cbrt cubic root
* ceil round up value
* copysign x with its sign changed to sign of y
* cos cosine
* cosh hyperbolic cosine
* cospi cos(pi \* x)
* erf error function encountered in integrating the normal distribution
* erfc complimentary error function
* exp ex
* exp2 2x
* exp10 10x
* expm1 ex – 1.0
* fabs absolute value of floating-point value
* fdim positive difference between x and y
* floor largest integer less than or equal to x
* fma (x \* y) + z, rounded
* fmax maximum of 2 floating-point values
* fmin minimum of 2 floating-point values
* fmod modulus
* fract fractional value in x (floor(x) also returned)
* frexp extract mantissa and exponent
* hypot square root of x2 + y2
* ilogb exponent as integer value
* ldexp x \* 2y
* lgamma natural logarithm of the absolute value of the gamma
* log natural logarithm (base e)
* log10 common logarithm (base 10)
* log2 logarithm base 2
* log1p natural logarithm of 1+x
* logb exponent of x, which is the integral part of logr|x|
* mad approximation of (x \* y) + z (when speed is preferred
* over accuracy)
* modf decompose floating-point number into integral and fractional parts
* nextafter next representable floating-point value following x in the direction of y
* pow xy
* pown xy (y is integer)
* powr xy (x greater than or equal to zero)
* remainder remainder
* remquo remainder and quotient
* rint round to nearest integer (in floating-point format)
* rootn x1/y
* round nearest integer value (in floating-point format)
* sqrt square root
* rsqrt reciprocal of square root (i.e. 1.0 / sqrt(x))
* sin sine
* sincos sine and cosine
* sinh hyperbolic sine
* sinpi sin(pi \* x)
* tan tangent
* tanh hyperbolic tangent
* tanpi tan(pi \* x)
* tgamma gamma function
* trunc truncate floating-point value

Nó cũng định nghĩa một số hàm số nguyên sau:

* abs absolute value
* clz number of leading 0-bits
* min minimum of two integers
* max maximum of two integers

Một số chức năng phổ biến khác cũng được định nghĩa:

* clamp clamp amount to range given by low and high
* degrees convert radians to degrees
* mix start + (stop – start)\*amount (linear blend)
* radians convert degrees to radians
* step 0.0 if v is less than edge, else 1.0
* smoothstep 0.0 if v edge0, 1.0 if v edge1, else smooth Hermite interpolation
* sign sign (-1.0, -0.0, +0.0 or 1.0)

Cuối cùng nhưng không kém phần quan trọng, tệp tiêu đề này xác định một số hàm hình học:

* cross cross product
* dot dot product
* length vector length
* distance distance between 2 points
* normalize normalize vector (length will be 1.0)

Nếu bạn đã quen với OpenCL, bạn nên nhận ra các API này. Trên thực tế, bản thân tệp tiêu đề Honeycomb rs\_cl.rsh đề cập đến tài liệu OpenCL vì nó chứa các nhận xét đề cập đến các phần 6.11.2, 6.11.3, 6.11.4 và 6.11.5, là các phần trong OpenCL 1.1 thông số kỹ thuật bao gồm các hàm toán học, số nguyên, thông dụng và hình học tương ứng. Listing 9–18 hiển thị những nhận xét này từ rs\_cl.rsh. Bạn cũng có thể tham khảo tài liệu OpenCL để biết thêm thông tin, được cung cấp miễn phí trên trang web Khronos ([www.khronos.org/opencl](http://www.khronos.org/opencl)).

Listing 9–18. Comments Referring To OpenCL In rs\_cl.rsh

#ifndef \_\_RS\_CL\_RSH\_\_

#define \_\_RS\_CL\_RSH\_\_

...

// Float ops, 6.11.2

...

extern float \_\_attribute\_\_((overloadable)) trunc(float);

FN\_FUNC\_FN(trunc)

// Int ops (partial), 6.11.3

...

IN\_FUNC\_IN\_IN\_BODY(max, (v1 > v2 ? v1 : v2))

FN\_FUNC\_FN\_F(max)

// 6.11.4

\_RS\_RUNTIME float \_\_attribute\_\_((overloadable)) clamp(float amount, float low, float

high);

...

// 6.11.5

\_RS\_RUNTIME float3 \_\_attribute\_\_((overloadable)) cross(float3 lhs, float3 rhs);

...

#endif

Tệp tiêu đề này cũng xác định một số hàm chuyển đổi như convert\_int3, hàm này chuyển đổi một vectơ gồm ba phần tử thành một vectơ gồm ba số nguyên.

## rs\_math.rsh

Tệp tiêu đề này xác định các chức năng sau trong Honeycomb:

* rsSetObject
* rsClearObject
* rsIsObject
* rsGetAllocation
* rsAllocationGetDimX
* rsAllocationGetDimY
* rsAllocationGetDimZ
* rsAllocationGetDimLOD
* rsAllocationGetDimFaces
* rsGetElementAt
* rsRand
* rsFrac
* rsSendToClient
* rsSendToClientBlocking
* rsForEach

Như chúng ta đã thấy trước đó, rsForEach () là một trong những hàm quan trọng nhất.

Các hàm rsSendToClient () và rsSendToClientBlocking () (được chuyển đến rs\_core.rsh trong Android 4.0) đáng được nhắc đến vì chúng cho phép một tập lệnh gửi dữ liệu đến lớp Java. Để lớp Java có thể nhận dữ liệu, ứng dụng của bạn sẽ phải đăng ký một trình xử lý thông báo, như được hiển thị trong Listing 9–19.

**Listing 9–19.** Message Handler

RenderScript rs = RenderScript.create(this); // needs a Context as parameter

rs.setMessageHandler(new RenderScript.RSMessageHandler() {

@Override

public void run() {

super.run();

Log.d(TAG, String.valueOf(this.mID) + " " + mData + ", length:" + mLength);

}

});

Các trường mID, mData và mLength của trình xử lý sẽ chứa thông tin được tập lệnh chuyển qua. Nếu tập lệnh của bạn gửi dữ liệu đến lớp Java nhưng không có trình xử lý nào được đăng ký thì một ngoại lệ sẽ được đưa ra.

Trong Android 4.0, rs\_math.rsh cũng được sửa đổi để tập trung hoàn toàn vào các hàm toán học:

* rsRand
* rsFrac
* rsClamp
* rsExtractFrustumPlanes (new in Android 4.0)
* rsIsSphereInFrustum (new in Android 4.0)
* rsPackColorTo8888
* rsUnpackColor8888

Các hàm cấp phát đã được chuyển sang một tệp mới, rs\_allocation.rsh, ngoài các hàm được giới thiệu trong Honeycomb thì định nghĩa hai hàm mới sau:

* rsAllocationCopy1Drange
* rsAllocationCopy2Drange

Các chức năng đối tượng cũng được chuyển sang tệp tiêu đề mới: rs\_object.rsh.

## rs\_graphics.rsh

Tệp tiêu đề này xác định các chức năng sau:

* rsgBindProgramFragment
* rsgBindProgramStore
* rsgBindProgramVertex
* rsgBindProgramRaster
* rsgBindSampler
* rsgBindTexture
* rsgProgramVertexLoadProjectionMatrix
* rsgProgramVertexLoadModelMatrix
* rsgProgramVertexLoadTextureMatrix
* rsgProgramVertexGetProjectionMatrix
* rsgProgramFragmentConstantColor
* rsgGetWidth
* rsgGetHeight
* rsgAllocationsSyncAll (new overloaded function added in Android 4.0)
* rsgDrawRect
* rsgDrawQuad
* rsgDrawQuadTexCoords
* rsgDrawSpriteScreenspace
* rsgDrawMesh
* rsgClearColor
* rsgClearDepth
* rsgDrawText
* rsgBindFont
* rsgFontColor
* rsgMeasureText
* rsgMeshComputeBoundingBox

Trong Android 4.0, các chức năng sau đã được thêm vào rs\_graphics.rsh:

* rsgBindColorTarget
* rsgClearColorTarget
* rsgBindDepthTarget
* rsgClearDepthTarget
* rsgClearAllRenderTargets
* rsgFinish

Vì trọng tâm của chương này không phải là giúp bạn trở thành chuyên gia trong lĩnh vực kết xuất 3D, chúng tôi sẽ không xem xét chi tiết các chức năng này. Nếu bạn đã quen với OpenGL thì bạn sẽ có thể dễ dàng sử dụng các chức năng này. Nếu chưa, thì trước tiên bạn nên làm quen với các thuật ngữ 3D như phân mảnh, đỉnh và lưới trước khi đi sâu vào RenderScript.

## rs\_time.rsh

Tệp này định nghĩa như sau:

* rsTime
* rsLocalTime
* rsUptimeMillis
* rsUptimeNanos
* rsGetDt

Đặc biệt quan tâm là hàm rsGetDt (), trả về số giây (dưới dạng giá trị dấu phẩy động) kể từ khi nó được gọi lần cuối. Listing 9–20 cho thấy cách điều này thường được sử dụng trong một tập lệnh.

Listing 9–20. Calling rsGetDt()

#pragma version(1)

#pragma rs java\_package\_name(com.yourpackagehere)

typedef struct \_\_attribute\_\_((packed, aligned(4))) MyObject {

float x;

float other\_property;

} MyObject\_t;

MyObject\_t \*object; // Java layer would have to call bind\_object

int root() {

float dt = rsGetDt();

float dx = dt \* 10.f; // 10 pixels per second

object->x += dx; // new x position of object

// do something else here, for example to draw the object

return 20;

}

**LƯU Ý:** Mặc dù hàm root () trả về 20 (nghĩa là 20 mili giây giữa các khung hình hoặc 50 khung hình mỗi giây), tốc độ khung hình không được đảm bảo. Đây là lý do tại sao mã của bạn nên sử dụng rsGetDt () thay vì giả định thời lượng giữa hai khung là giá trị root () trả về.

## rs\_atomic.rsh

Tính năng mới trong Android 4.0, tệp này xác định các chức năng chưa được xác định trong Honeycomb:

* rsAtomicInc
* rsAtomicDec
* rsAtomicAdd
* rsAtomicSub
* rsAtomicAnd
* rsAtomicOr
* rsAtomicXor
* rsAtomicMin
* rsAtomicMax
* rsAtomicCas

Như tên của chúng cho thấy, các hàm này thực hiện các hoạt động khác nhau về mặt nguyên tử.

# RenderScript vs NDK

Cả RenderScript và NDK đều ở đây để cải thiện hiệu suất của các ứng dụng. Mặc dù họ có chung một mục tiêu, nhưng bạn nên suy nghĩ cẩn thận về việc sử dụng cái này so với cái kia. Mỗi thứ đều có những ưu điểm và nhược điểm riêng, và việc không suy nghĩ thấu đáo mọi thứ có thể khiến bạn bị mắc kẹt trong quá trình phát triển hoặc tìm ra cách khó mà một thứ gì đó cực kỳ khó thực hiện.

Những ưu điểm của RenderScript có thể được tóm tắt như sau:

* Không phụ thuộc vào nền tảng (ví dụ: ARM, MIPS, Intel)
* Thực hiện song song dễ dàng
* Có thể sử dụng CPU, GPU hoặc các đơn vị xử lý khác

Mặt khác, RenderScript có một số nhược điểm:

* Chỉ dành cho Android (không thể sử dụng tập lệnh với các nền tảng khác như iOS)
* Đường cong học tập
* Số lượng API có hạn

Kết quả hiệu suất RenderScript thực tế khó có được vì chúng sẽ phụ thuộc nhiều vào phiên bản Android và thiết bị phần cứng. Một số tập lệnh có thể chạy nhanh hơn nhiều so với đối tác Java hoặc NDK của chúng trong khi những tập lệnh khác thì không. Vì mã kế thừa hoặc vì nó đơn giản có ý nghĩa trong ứng dụng của bạn, bạn thực sự có thể sử dụng cả RenderScript và NDK trong cùng một ứng dụng.

# Summary

RenderScript đã được cải thiện trong Android 4.0 và sẽ tận dụng nhiều tính năng phần cứng hơn trong tương lai. Mặc dù ARM rõ ràng là kiến trúc thống trị trong các thiết bị di động Android hiện nay và việc sử dụng NDK do đó có thể được chứng minh, thực tế là RenderScript không phụ thuộc vào nền tảng có thể là một lợi ích to lớn vì cuối cùng nó có thể giảm đáng kể chi phí bảo trì của bạn. Nhìn chung, RenderScript cho thấy một số hứa hẹn và có khả năng trở thành một phần quan trọng trong các ứng dụng của bạn trong tương lai nếu bạn cần đạt đến mức hiệu suất không thể đạt được chỉ với Java.

# Thực nghiệm

## Thực nghiệm ScriptIntrinsic

Các RenderScript được tích hợp sẵn để thực hiện các hoạt động được xác định rõ ở đây là xử lý hình ảnh.

Thực nghiệm có các chế độ chính là: Blur mode, Convolve Mode và Color Matrix Mode.



Hình 1. 3 Ảnh original

Sau khi di chuyển thanh seekbar bức ảnh sẽ bị mờ đi theo việc di chuyển seekbar nhiều hay ít.



Hình 1. 4 Ảnh sau khi Blur

Convolve Mode sẽ làm cho bức ảnh trở nên sắc nét hơn tùy theo việc người dùng muốn ảnh sắc nét như thế nào.



Hình 1. 5 Ảnh sau khi Convolve

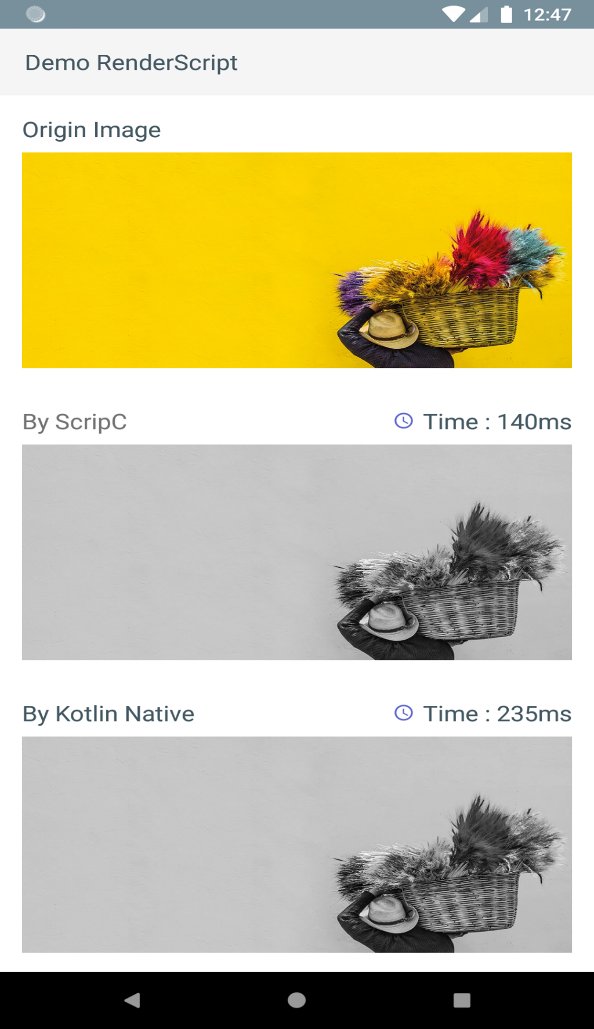
Color Matrix Mode áp dụng ma trận màu để chuyển đổi màu của bức ảnh.



Hình 1. 6 Ảnh sau khi Convolve

## Thực nghiệm ScriptC

Ứng dụng tạo ScripC tạo chỉnh sửa hình ảnh thành đen trắng thời gian chỉnh sửa hết 140ms nhanh hơn khi thực hiện cùng chức năng mà khi code băng Native



Hình 1.7 Thực nghiệm ScripC

# Tổng kết

Qua nội dung đã tìm hiểu ở trên, chúng ta có thể thấy được những lợi ích của RenderScript so với khi code native.

RenderScript có thể được ứng dụng trong 3D Rendering, Image Processing, Computational Photography và Computer Vision.

Đến đây chúng ta đã biết cách tạo một RenderScipt và các thành phần để có thể tạo RenderScript.

Giới thiệu về các api trong RenderScript.

# Tài liêu tham khảo

[1] Hervé Guihot. *Giáo trình:* *Pro Android Apps Performance Optimization*.

[2] https://developer.android.com/guide/topics/renderscript/compute