**http://developer.android.com/guide/topics/renderscript/compute.html**

**Renderscript Computation**

-RS는 고성능 계산 API제공함

-RS는 어플리케이션이 자동으로 병렬 컴퓨팅 할수 있게 함 (각 코어를 이용하여)

-CPU, GPU, DSP등의 프로세스를 지원함

-Image Processing, mathemetical modeling, 그리고 많은 수학 계산을 필요로 하는 앱에 적당함

-프로세서 타입에 맞춰 코딩하지 않아도 된다. RS는 실행될 때 컴파일되니깐

**Renderscript System OverView**

-RS는 네이티브 레벨에서 동작하고 Android VM과 커뮤니케이팅 해야 함, 그래서 RS어플리케이션의 set up 방법은 기존 VM 어플리케이션과 다름.

-RS를 사용하는 어플은 전통적인 안드로이드 어플과 기본적으로 같음. 하지만 필요한 부분에서만 RS를 쓸 수 있음. 이렇게 할 수 있는건 RS가 플랫폼 독립적이기 때문임

-RS 시스템은 control and slave architecture를 채택했다. (low level RS 코드는 VM에서 돌아가는higher level 안드로이드 시스템에 의해 컨트롤 된다.)

-VM은 memory management, bind memory에 대한 컨트롤을 유지하기 때문에 RS코드는 VM에 접근할 수 있다. (VM이 RS에도 Memory를 바인딩 하니까)

-Android framework는 RS에 비동기 call을 하고, call은 메시지 큐에 있다가 실행된다. Figure 1.

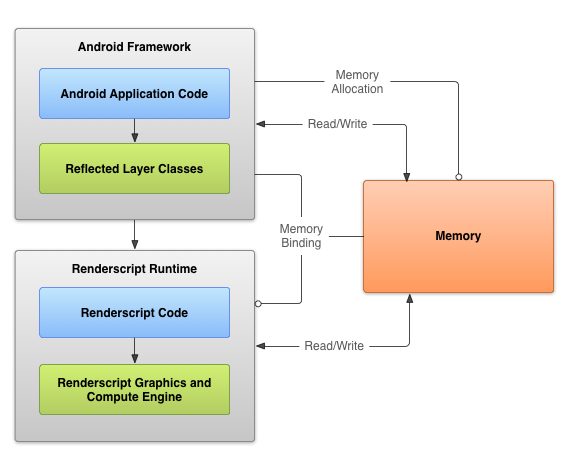


Figure 1. Renderscript system overview

-Rs를 사용할 때, 세개의 API 레이어가 있는데 얘네가 Android framework코드와 RS runtime과의 커뮤니케이션을 가능하게 해준다.  
-RS runtime API는 어플리케이션에서 필요로 하는 computation을 가능하게 해준다  
-reflected layer API들은 RS runtime code로부터 만들어진 클래스들이다. 이건 기본적으로 RS코드의 wrapper이며, Android framework가 RS runtime과 상호작용하게 해준다. 빌더에 의해 만들어진다. 얘네들 덕분에 NDK와 같은 JNI 글루코드를 작성하지 않아도 된다.  
-Android framework layer가 RS rumtime에 접근하기 위해 relected layer를 콜한다

- 편한점은…  
-이식성(portability): Rs는 여러 다른 프로세서에서 동작할 수 있도록 디자인 되어 편함  
-성능(Performance): Rs는 매우 좋은 성능을 보여주는데, 여러 코어들을 이용해 병렬처리하기 때문임  
-사용성(Userbility): JNI글루코드 같은거 쓸 필요없이 개발되었다.

-단점은..  
-개발의 복잡성: 어렵당  
-디버깅 가시성: main CPU가 아닌 다른 프로세서에서 동작할 수 있기 때문에 디버깅 어려움

**Filterscript**

이미지 프로세싱에 관한 거라 넘어감

**Creating a Rendersript**

-RS는 rsForeach() 메서드를 통해서 가용한 프로세서 수를 조절할 수 있다. (혹은 framework 레벨의 forEach\_root()메서드), 이 메서드는 자동으로 일을 분산시켜 가용한 프로세스에서 동작하게 한다.

-RS를 구현하는 것은 .rs 파일을 만드는 것과, Android framework 레벨에서 이를 호출하는 것을 포함한다. (호출은 forEach\_root(), rsForeach() 메서드로) 아래 다이어그램 참고

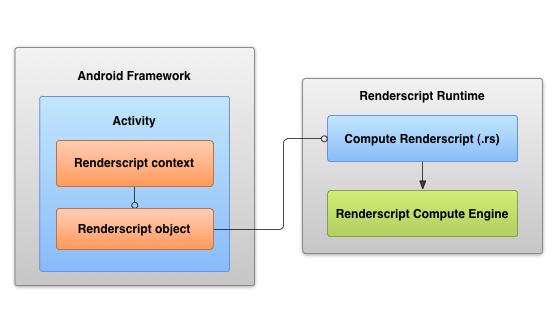


Figure 1. Renderscript overview

-> 다음은 예제임

**Creating the Renderscript file**

-Rs 코드는 <project\_root>/src/ 디렉토리에 있는 .rs, .rsh 파일에 작성되어 있다. 이 코드는 계산 로직과 변수, 포인터 선언을 포함하고 있다. 모든 .rs파일은 일반적으로 다음과 같은 아이템을 포함한다.  
-pragma 선언(#pragma rs java\_package\_name(package.name)) : .java 파생 파일의 페키지 이름을 선언한다.  
-pragma 선언(#pragma version(1)) : 사용할 RS버전. 지금은 1 밖에 없음.  
-root 메서드 : RS의 main entry point. 기본 root() 메서드는 void이고 다음 인자들을 받는다.  
 -RS의 인풋, 아웃풋으로 쓰일 할당된 메모리의 포인터. API13이전까지는 두 포인터 모두   
 필요하고, 14부터는 둘중에 하나만 있거나, 둘다 있으면 된다.  
 (다음은 옵셔널한 것)  
 -사용자가 정의한 데이터에 대한 포인터,  
 -사용자가 정의한 데이터의 사이즈  
 -API 16이 시작되면서, 디폴트 root 메서드 말고 직접 정의할 수 있다. 게다가, 여러 개의

root 메서드를 정의할 수도 있다. 이걸 하기 위해서는 \_\_attribute\_\_((kernel)) 속성

을 사용한다. 예를 들어 두개의 uin32\_t를 받아 uchar4를 리턴하는 root() 메서드는 다음과

같다.

uchar4 \_\_attribute\_\_((kernel)) root(uint32\_t x, uint32\_t y) {

...

}

-init() 메서드 (optional) : root()가 실행되기 전 필요한 초기화를 할 수 있다. RS가 시작할 때 한번만 불린다. 다른 어떤 RS실행 부분보다 먼저 실행됨.  
-사용하기를 원하는 variables, pointers, structures

아래는 mono.rs 파일의 예시이다.

*#pragma version(1)*

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

*//multipliers to convert a RGB colors to black and white*

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

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

*//unpack a color to a float4*

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

*//take the dot product of the color and the multiplier*

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

*//repack the float to a color*

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

*}*

**Setting floating point precision**

-컴퓨터 알고리즘에 의해 요구되는 floating point 정확도를 정의할 수 있다. 디폴트 정확도보다 낮은 정확도가 요구될 때 사용하기 좋다. Script에 아래 pragma들을 이용해서 정의할 수 있다.

- #pragma rs\_fp\_full(아무것도 정의 안 할 때 디폴트): 스탠다드 정확도  
- #pragma rs\_fp\_relaxed : 좀 구린거  
- #pragma rs\_fp\_imprecise : 더 구린거

**Script intrinsics**

-RS는 set of script intrinsics을 지원하는데 이것은 미리 구현된 필터링 프리미티브다. 코딩을 줄여준다. 그리고 가능한 좋은 성능을 낼 수 있게 해준다.

-Blends, Blur, Color matrix, 3x3 convolve, 5x5 vonvolve, per-channel lookup table, converting an Android YUV buffer to RGB

**Calling the Renderscript code**

-Android framework 코드에서 RS를 call 하려면, (ScriptC\_script\_name) 클래스를 instant로 만들어서 RS 오브젝트를 만들면 된다. 이 클래스는 forEach\_root()메서드를 포함하는데, rsForEach를 파생할 수 있게 해준다. RS runtime level에서 invoke하고 있다면 같은 파라미터를 주게 될거다. 이 테크닉은 어플이 초정밀하게 수학 계산을 하지 않아도 되게끔 해준다. 샘플을 보고 Android application이 어떻게 RS를 이용하는지 봅시다.

Android Framework level에서 RS를 call 하려면:

1. Android framework 코드에 Rs에서 필요로 하는 메모리를 할당한다. API 레벨 13이전에는 인풋 아웃풋 모두 할당해야 되고 그 이후는 둘중에 하나만 하거나 둘다 해야됨.
2. ScriptC\_script\_name 클래스를 인스턴스화한다.
3. forEach\_root()를 호출한다.

-다음은 HelloCompute 예제다. Bitmap을 흑백으로 만들어주는 거임. createScript()메서드가 위에서 설명한 스텝을 수행하고 있음. 이 메서드는 RS인 mono.rs를 호출한담에 메모리 할당한걸 넘겨주고 비트맵을 저장하고 결과적으로 아웃풋을 낸다.

package com.example.android.rs.hellocompute;

import android.app.Activity;

import android.os.Bundle;

import android.graphics.BitmapFactory;

import android.graphics.Bitmap;

import android.renderscript.RenderScript;

import android.renderscript.Allocation;

import android.widget.ImageView;

public class HelloCompute extends Activity {

private Bitmap mBitmapIn;

private Bitmap mBitmapOut;

private RenderScript mRS;

private Allocation mInAllocation;

private Allocation mOutAllocation;

private ScriptC\_mono mScript;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

mBitmapIn = loadBitmap(R.drawable.data);

mBitmapOut = Bitmap.createBitmap(mBitmapIn.getWidth(), mBitmapIn.getHeight(),

mBitmapIn.getConfig());

ImageView in = (ImageView) findViewById(R.id.displayin);

in.setImageBitmap(mBitmapIn);

ImageView out = (ImageView) findViewById(R.id.displayout);

out.setImageBitmap(mBitmapOut);

createScript();

}

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);

}

private Bitmap loadBitmap(int resource) {

final BitmapFactory.Options options = new BitmapFactory.Options();

options.inPreferredConfig = Bitmap.Config.ARGB\_8888;

return BitmapFactory.decodeResource(getResources(), resource, options);

}

다른 RS 파일에서 RS를 콜 하려면

1. Android framework 코드에서 RS가 필요로 하는 메모리 할당.
2. Call rsForEach()

rs\_script script;

rs\_allocation in\_allocation;

rs\_allocation out\_allocation;

UserData\_t data;

...

rsForEach(script, in\_allocation, out\_allocation, &data, sizeof(data));

예에서, 포인터 넘기는거랑, 사이즈 넘기는 거는 옵셔널, 하지만 유즈풀

**Script groups**

-RS script를 그룹화 하고 한꺼번에 실행시킬 수 있다.

-그룹화 할려면 ScriptGroup.Builder 클래스를 사용한다.

**Important:** The script group must be a direct acyclic graph for this feature to work.

<http://developer.android.com/guide/topics/renderscript/advanced.html>

**Advanced Renderscript**

-RS를 이용하는 어플리케이션이 Android VM에서 실행되기 때문에, 익숙한 모든 framework API에 접근한다. Framework와 RS runtime의 상호작용을 가능하게 하기 위해서, 중간 계층의 코드가 존재한다. 이 중간 계층의 코드는 두 레벨 간의 커뮤니케이션과 메모리 관리를 위해서 존재하기도 한다. 이 문서는 이 두 개의 서로 다른 계층의 코드(Android VM, RS runtime)에 대해서 어떻게 메모리가 공유되는지 디테일하게 다룬다.

**Renderscript Runtime Layer**

-RS 코드는 잘 정의된 런타임 레이어에서 컴파일 되고 실행된다. RS API는 프로세서의 코어 양에 맞춰 엄청 빠르게 계산할 수 있게 해준다.

(Note: NDK 안의 C 코드는 CPU에서 도는데, RS는 서로 다른 타입의 프로세서에서도 돌 수 있도록 만들어진 것이기 때문에, RS는 C 라이브러리에 접근하지 못한다.)

-RS 코드는 src 디렉토리의 .rs, .rsh 파일에 작성될 것이다. 코드는 llvm 컴파일러(Abdriud build의하나의 파트로 동작하는)에 의해 중간수준의 bytecode로 컴파일 된다. 어플리케이션이 디바이디바이 실행될 때, 디바이스에 존재하는 또다른 llvm 컴파일러에 의해 바이트 코드가 머신코드로 컴파일 된다. 머신코드는 디바이스에 최적화 되고, 다음번의 RS 사용은 어플리케이션이 다시 바이트 코드로 컴파일 되지 않아도 되게끔 된다.

RS 런타임 라이브러리의 주요 요소들은 다음을 포함한다.

1. 메모리 할당
2. Scalar, vector타입을 포함하는 large collection of math functions
3. 전환 루틴 (Conversion routines for primitive data types and vectors, and more)
4. Vector type과 같은 Rs 시스템을 서포트 하는 여러 데이터 타입과 구조체
5. Logging function

**Reflected Layer**

- reflected layer는 클래스의 집합으로, Android 빌드 툴에 의해 만들어진 것이다. 이는 Android framework에서 RS로 접근하는 것을 허용하기 위해 만들어졌다. Reflected layer는 또한 여러 메서드와 생성자를 지원하는데, 이것은 Rs 코드에 정의된 포인터에 메모리를 할당하거나 관련된 일을 할 수 있도록 하는 것이다. 다음 리스트는 reflected 된 주요 컴포넌트이다.

1. 모든 .rs 파일은 class name으로 만들어진다.   
   project\_root/gen/package/name/ScriptC\_renderscript\_filename 은 .rs파일의 .java 버전이다. 이 파일을 통해서 Android framework에서 RS로 접근 할 수 있다. 이 클래스는 다음의 아이템들을 포함한다.  
   - Non static functions,  
   - Non static, global RS 변수, 각 변수에 대한 접근자, (const로 선언된 건 set메서드가 안생긴다.)  
   - Global pointers
2. Struct 는 하나의 클래스 네임으로 reflected 된다.  
   -project\_root/gen/package/name/ScriptField\_struct\_name, 이 클래스는 struct의 배열을 나타낸다. 이 struct의 인스턴스에 메모리를 할당 할 수 있다.

**Functions**

-function은 스스로 script class로 reflected 된다. (project\_root/gen/package/name/ScriptC\_renderscript\_filename 안에), 예를 들어, 다음과 같은 함수를 RS코드에 선언한다면,

*void touch(float x, float y, float pressure, int id) {*

*if (id >= 10) {*

*return;*

*}*

*touchPos[id].x = x;*

*touchPos[id].y = y;*

*touchPressure[id] = pressure;*

*}*

다음과 같은 코드가 만들어진다.

*public void invoke\_touch(float x, float y, float pressure, int id) {*

*FieldPacker touch\_fp = new FieldPacker(16);*

*touch\_fp.addF32(x);*

*touch\_fp.addF32(y);*

*touch\_fp.addF32(pressure);*

*touch\_fp.addI32(id);*

*invoke(mExportFuncIdx\_touch, touch\_fp);*

*}*

-함수는 리턴값이 있으면 안된다. 왜냐하면 Rs 시스템은 비동기식으로 디자인되었기 때문이다. Android framework에서 Rs 코드를 호출하면, 일단 큐에 들어갔다가 실행될 수 있을 때 실행된다. 이런 제약이 있기 때문에 Rs 시스템은 계속되는 인터럽트를 피할 수 있고, 효율성을 높일 수 있다. 만약 리턴값이 허용된다면, 호출은 리턴값이 리턴될때까지 block 될거다.

-만약 RS 코드로부터 어떤 값을 Android framework로 전달받아야 된다면, **rsSentToCline()** 함수를 사용해야 한다.

**Variables**

-지원되는 타입의 변수들은 project\_root/gen/package/name/ScriptC\_renderscript\_filename안에 reflected 된다. Set, get 함수가 생성된다. 다음과 같이 RS 코드에 변수 선언을 하면

uint32\_t unsignedInteger = 1;

다음 코드가 만들어진다.

private long mExportVar\_unsignedInteger;

public void set\_unsignedInteger(long v){

mExportVar\_unsignedInteger = v;

setVar(mExportVarIdx\_unsignedInteger, v);

}

public long get\_unsignedInteger(){

return mExportVar\_unsignedInteger;

}

**Structs**

-Struct는 <project\_root>/gen/com/example/renderscript/ScriptField\_struct\_name에 클래스로 reflected된다. 이 클래스는 특정 개수의 struct에 메모리를 할당할 수 있는 struct array를 나타낸다. 예를 들어 아래와 같이 struct를 선언하면

typedef struct Point {

float2 position;

float size;

} Point\_t;

다음 코드가 ScriptField\_Point.java 파일에 생성된다.

package com.example.android.rs.hellocompute;

import android.renderscript.\*;

import android.content.res.Resources;

/\*\*

\* @hide

\*/

public class ScriptField\_Point extends android.renderscript.Script.FieldBase {

static public class Item {

public static final int sizeof = 12;

Float2 position;

float size;

Item() {

position = new Float2();

}

}

private Item mItemArray[];

private FieldPacker mIOBuffer;

public static Element createElement(RenderScript rs) {

Element.Builder eb = new Element.Builder(rs);

eb.add(Element.F32\_2(rs), "position");

eb.add(Element.F32(rs), "size");

return eb.create();

}

public ScriptField\_Point(RenderScript rs, int count) {

mItemArray = null;

mIOBuffer = null;

mElement = createElement(rs);

init(rs, count);

}

public ScriptField\_Point(RenderScript rs, int count, int usages) {

mItemArray = null;

mIOBuffer = null;

mElement = createElement(rs);

init(rs, count, usages);

}

private void copyToArray(Item i, int index) {

if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof \* getType().getX()/\* count

\*/);

mIOBuffer.reset(index \* Item.sizeof);

mIOBuffer.addF32(i.position);

mIOBuffer.addF32(i.size);

}

public void set(Item i, int index, boolean copyNow) {

if (mItemArray == null) mItemArray = new Item[getType().getX() /\* count \*/];

mItemArray[index] = i;

if (copyNow) {

copyToArray(i, index);

mAllocation.setFromFieldPacker(index, mIOBuffer);

}

}

public Item get(int index) {

if (mItemArray == null) return null;

return mItemArray[index];

}

public void set\_position(int index, Float2 v, boolean copyNow) {

if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof \* getType().getX()/\* count \*/);

if (mItemArray == null) mItemArray = new Item[getType().getX() /\* count \*/];

if (mItemArray[index] == null) mItemArray[index] = new Item();

mItemArray[index].position = v;

if (copyNow) {

mIOBuffer.reset(index \* Item.sizeof);

mIOBuffer.addF32(v);

FieldPacker fp = new FieldPacker(8);

fp.addF32(v);

mAllocation.setFromFieldPacker(index, 0, fp);

}

}

public void set\_size(int index, float v, boolean copyNow) {

if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof \* getType().getX()/\* count \*/);

if (mItemArray == null) mItemArray = new Item[getType().getX() /\* count \*/];

if (mItemArray[index] == null) mItemArray[index] = new Item();

mItemArray[index].size = v;

if (copyNow) {

mIOBuffer.reset(index \* Item.sizeof + 8);

mIOBuffer.addF32(v);

FieldPacker fp = new FieldPacker(4);

fp.addF32(v);

mAllocation.setFromFieldPacker(index, 1, fp);

}

}

public Float2 get\_position(int index) {

if (mItemArray == null) return null;

return mItemArray[index].position;

}

public float get\_size(int index) {

if (mItemArray == null) return 0;

return mItemArray[index].size;

}

public void copyAll() {

for (int ct = 0; ct < mItemArray.length; ct++) copyToArray(mItemArray[ct], ct);

mAllocation.setFromFieldPacker(0, mIOBuffer);

}

public void resize(int newSize) {

if (mItemArray != null) {

int oldSize = mItemArray.length;

int copySize = Math.min(oldSize, newSize);

if (newSize == oldSize) return;

Item ni[] = new Item[newSize];

System.arraycopy(mItemArray, 0, ni, 0, copySize);

mItemArray = ni;

}

mAllocation.resize(newSize);

if (mIOBuffer != null) mIOBuffer = new FieldPacker(Item.sizeof \* getType().getX()/\* count \*/);

}

-이 코드가 만들어지니까, 필요한 구조체에 메모리를 할당하기도 편리하고, 메모리에 있는 구조체와 상호작용할 수도 있는 것이다. 각각의 구조체 클래스는 다음 메서드과 생성자를 정의한다.  
 - 오버로딩 된 생성자 : 이 생성자로 메모리 할당을 할 수 있다. ScriptField\_struct\_name(Render  
 Script rs, int count) 생성자로 메모리를 할당하기를 원하는 구조체들을 정의할 수 있다.  
 ScriptFiled\_struct\_name(RenderScript rs, int count, int usages) 생성자는 추가된 매개변수

Usages가 있는데, 할당할 메모리 공간을 지정할 수 있다. 네가지 타입이 있다.

-1. USAGE\_SCRIPT: 디폴트, script memory space를 할당함

-2. USAGE\_GRAPHICS\_TEXTURE: GUP의 texture memory space를 할당함

-3. USAGE\_GRAPHICS\_VERTEX: GPU의 vertex memory space를 할당함

-4. USAGE\_GRAPHICS\_CONSTANTS: 여러 프로그램 객체가 사용하는 GPU의 constant memory

Space를 할당함

Or 연산을 통해 여러 개의 메모리 공간을 정의할 수 있다. 이렇게 하면 Renderscript runtime

에 특정 memory space에 있는 데이터에 접근하고 싶다고 알릴 수 있다. 다음 예는 커스텀

데이터 타입을 script와 vertex memory space에 메모리 할당하는 것이다.

ScriptField\_Point touchPoints = new ScriptField\_Point(myRenderscript, 2,

Allocation.USAGE\_SCRIPT | Allocation.USAGE\_GRAPHICS\_VERTEX);

- Static nested class, ‘Item’은 구조체의 인스턴스를 객체 형태로 생성하게 해준다. 이 nested

Class는 매우 유용하다. 객체를 조종하고 난 후, set(Item I, int index, Boolean copyNow) 메서드

를 부르고, item을 세팅해서, 객체를 할당된 메모리에 푸시할 수 있다. RS runtime은 자동으로

새로 쓰여진 메모리에 접근할 수 있다.

- 접근자(get, set)는 모두 index 매개변수를 가진다. (구조체 배열의

인덱스) 각 Setter 메서드는 또한 copyNow 매개변수를 가지는데 이것은 즉시 이 메모리가 RS

Runtime에 동기화 할 것인지를 나타낸다. 동기화 되지 않은 메모리와 동기화 하려면 copyAll()

을 호출하면 된다.

- createElement() 메서드는 구조체를 생성한다. 하나 혹은 여러 개의 element에 메모리를 할당

할 때 사용된다.

- resize() 메서드는 C에서 realloc()처럼 동작한다. 이미 할당된 메모리를 확장하도록 해주고, 이

전에 생성된 값을 유지하게 해준다.

- copyAll() 메서드는 framework 레벨의 메모리와 RS runtime을 동기화 시킨다. set함수를 호출

하면, copyNow라는 Boolean 타입의 매개변수가 있다. 이 변수를 true로 하면 메서드를 부르는

시점에 동기화가 된다. False로 하면 copyAll()메서드를 호출하면, 아직 동기화 되지 않은 모든

프로퍼티가 동기화 된다.

**Pointers**

-포인터는 Project\_root/gen/package/name/ScriptC\_rendersciprt\_filenamed에 위치하는 script class로 reflected 된다. 구조체나 지원되는 RS 타입의 포인터를 정의할 수 있다. 그러나 구조체는 포인터나 nested array를 가질 수 없다. 예를 들어, 아래와 같은 구조체와, int32에 대한 포인터를 생성하면,

typedef struct **Point** {

float2 position;

float size;

} Point\_t;

Point\_t \*touchPoints;

int32\_t \*intPointer;

다음과 같은 코드가 만들어진다.

private ScriptField\_Point mExportVar\_touchPoints;

public void bind\_touchPoints(ScriptField\_Point v) {

mExportVar\_touchPoints = v;

if (v == null) bindAllocation(null, mExportVarIdx\_touchPoints);

else bindAllocation(v.getAllocation(), mExportVarIdx\_touchPoints);

}

public ScriptField\_Point get\_touchPoints() {

return mExportVar\_touchPoints;

}

private Allocation mExportVar\_intPointer;

public void bind\_intPointer(Allocation v) {

mExportVar\_intPointer = v;

if (v == null) bindAllocation(null, mExportVarIdx\_intPointer);

else bindAllocation(v, mExportVarIdx\_intPointer);

}

public Allocation get\_intPointer() {

return mExportVar\_intPointer;

}

- get() 메서드와 bind\_pointer\_name(set()대신임) 메서드가 생긴다. Android VM에 할당되어 있는 메모리를 RS runtime에 bind 할 수 있게 해준다. (.rs 파일에서는 메모리 할당을 할 수 없다.)

**Memory Allocation APIs**

- RS을 사용하는 어플리케이션은 Android VM에서 돌아간다. 하지만 실제 RS 코드는, native에서 실행되며, Android VM에 할당되어 있는 메모리에 접근할 수 있어야 한다. 이것을 가능하게 하기 위해서, VM에 할당되어 있는 메모리를 RS runtime에 붙일 수(attach)있어야 한다. 이 프로세스를 바인딩이라고 부른다. 바인딩은 RS runtime이 필요로 하지만 명확하게 할당할 수 없는 메모리를 이용하여 실행될 수 있도록 해준다. 결과는 C에서 malloc을 불렀을 때랑 기본적으로 같다. 여기에 추가로 좋은 점은, Android VM은 RS runtime 레이어와 메모리를 공유하는 것 뿐 아니라 garbage collector도 수행한다. 바인딩은 동적으로 메모리 할당할 때만 필요하다. 정적으로 할당된 메모리는 RS 코드가 컴파일 될 때 자동으로 생긴다.

-이 메모리 할당 시스템을 지원하기 위해서, API가 있다. API는 Android VM이 메모리를 할당하게하고, malloc과 유사한 기능을 제공한다. 이 클래스들은 기본적으로 메모리가 할당되는 과정을 명시한다. 이해하기 위해서 유사한 malloc을 보자

array = (int \*)malloc(sizeof(int)\*10);

-malloc은 두 파트로 나눌 수 있다. :할당될 메모리의 크기 (sizeof(int)), 그리고 개수 (10). Android framework도 비슷한 기능을 하는 클래스를 제공한다.

-**‘Element’ 클래스**는 malloc의 (sizeof(int)) 부분을 나타내며, 하나의 float 값이나 구조체와 같은 메모리 할당의 한 Cell을 의미한다. **‘Type’ 클래스**는 ‘Element’와 몇 개의 elements를 할당할지 (예에서는 10)을 캡슐화 한다. Type을 Element의 배열이라고 생각하면 된다. **‘Allocation’ 클래스**는 주어진 type에 기초하여 실제 메모리 할당을 하고, 할당된 메모리를 나타낸다.

-대부분의 상황에서, 이 메모리 할당 API들을 직접 호출할 일은 없다. Reflected layer class들이 이 API들을 이용하는 코드를 자동으로 만든다. 메모리를 할당하기 위해서 개발자가 할 일은, reflected layer 클래스에 정의된 생성자를 호출하는 것이다. 그러면 RS에 메모리 할당의 결과가 바인딩 된다. 이 클래스들을 직접 사용하고 싶은 경우가 있다. (예를 들면, 리소스로부터의 비트맵을 로딩할때, primitive type의 구조체에 메모리를 할당하고 싶을 때) 이런 경우 어떻게 하는지는 아래 ‘Allocating and binding memory to the Renderscript’ 섹션에서 볼 수 있다. 아래 표는 세 개의 메모리 매니지먼트 클래스의 디테일이다.

|  |  |
| --- | --- |
| Android object type | description |
| Element | 메모리 할당의 one cell을 나타내며 basic, complex 두 가지가 있다.  Basic element는 가능한 타입의 데이터 하나를 포함한다. Basic element data type의 예는 single float value, a float4 vector, single RGB-565 color가 있다.  Complex element는 basic element의 리스트를 포함하며, RS 코드에서 정의한 구조체로부터 생성된다. 예를 들어, Allocation은 메모리안에 순서대로 정렬된 여러 개의 구조체들을 포함한다. 각각의 구조체는 그 구조체의 각 데이터 타입보다는 하나의 element로 취급된다. |
| Type | Type은 메모리 할당의 형식으로, element의 요소로 되어있다. 메모리의 레이아웃을 나타내는데(Element의 배열), type이 의미하는 데이터에 메모리를 할당하지는 않는다.  Type은 다섯개의 관점을 가진다: X, Y, Z, LOD(level of detail), Faces(of a cube map). 어떤 integer 값이든 가용한 메모리 제한 없이 X,Y, Z 디멘션을 추가할 수 있다. X dimension을 0이상의 값으로 갖는 single dimention allocation은 Y, Z가 없음을 표시하기 위해 0dl 된다. 예를 들어 x=10, y=1로 할당한 것은 two dimension 인 것이고, x=10, y=0은 one dimension이다. LOD와 faces dimension은 present, not present를 나타내는 Boolean 이다. |
| Allocation | Allocation은 Type으로 표시된 메모리 디스크립션을 기초로 하여 어플리케이션에 메모리를 제공한다. 할당되는 메모리는 여러 메모리 공간에 동시에 존재할 수 있다. 어떤 한 곳에서 메모리가 수정되면, 메모리의 동기화를 명확히 해야 한다. 그렇게 해야 다른 공간에 있는 것들도 업데이트 된다.  Allocation data는 두 가지 기본적인 방법 중 하나로 업로드 된다. : type checked and type unchecked. copyFrom() 메서드는 Android System에서 array를 가져다가 native layer memory store에 복사한다. The unchecked variants는 구조체를 지원하지 않기 때문에 Android system이 구조체의 배열을 복사하도록 허락한다. 예를 들어, n floats 배열의 Allocation이 있다면, float[n], byte[n\*4] 배열은 복사될 수 있다. |

**Working with Memory**

- RS에 정의된 Non-static, global 변수들은 컴파일 타임에 메모리에 쓰여진다. 개발자는 이 변수들에 필요한 메모리 할당을 Android framework level에서 하지 않아도, 이 변수들을 직접 RS 코드에서 이용할 수 있다. Android framework layer 또한 이 변수들에 접근할 수 있다. (reflected layer classes에 만들어진 accessor를 통해서) 만약 이 변수들이 RS runtime layer에서 초기화 되면, 이 값들은 Android framework layer와 일치하는 값으로 초기화 된다. 글로벌 변수가 const 이면, set 메서드는 만들어지지 않는다.

Note: 만약 rs\_program\_fragment, rs\_allocation과 같은 포인터를 포함하는 특정한 RS 구조체를 사용한다면, Android framework class와 부합하는 오브젝트를 반드시 포함해야 하고 set 메서드를 통해서 RS runtime에 메모리를 바인딩 해야 한다. 이 구조체들을 Rs runtime layer에서 직접 조작할 수 없다. 이 제한은 pointer를 포함한 user define 구조체에는 해당하지 않는다. 왜냐하면 이 구조체들은 reflected class에 전달되지 않기 때문이다. 포인터를 포함하는 non-static, global를 정의하려고 하면 컴파일러가 에러를 표시한다.

-RS가 포인터를 서포트 하지만, 개발자는 Android framework 코드에서 메모리 할당을 확실히 해야 한다. .rs 파일에서 글로벌 포인터를 정의하면, 적절한 reflected layer class를 통해 메모리 할당을 하고 그 메모리를 native RS layer에 바인딩 해야 한다. Android framework layer에서도 RS layer와 마찬가지로 이 메모리들과 상호작용 할 수 있다. RS layer는 대부분의 레이어에서 변수들을 다루기 위한 편리를 제공한다.

**Allocating and binding dynamic memory to the RenderScript**

- dynamic memory를 할당하기 위해서, 가장 일반적인 방법으로, Script.FieldBase클래스의 생성자를 호출해야 한다. 다른 방법은 Allocation을 직접 만드는 것이다. 이것은 primitive type 포인터 같은 데서 필요로 한다. 심플함을 위해서는 가능하면 Script.FieldBase 클래스를 사용해야 한다. Memory allocation을 얻은 뒤에는 포인터의 reflected bind 메서드를 호출하여 Rs runtime에 메모리를 할당한다.

- 다음 예는 primitive타입 포인터(intPointer)와 구조체의 포인터(touchPoints)에 메모리 할당하는 것이다. Rs에 바인딩도 한다.

private RenderScript myRenderscript;   
private ScriptC\_example script;   
private Resources resources;   
   
public void init(RenderScript rs, Resources res) {   
    myRenderscript = rs;   
    resources = res;   
   
    //allocate memory for the struct pointer, calling the constructor   
    ScriptField\_Point touchPoints = new ScriptField\_Point(myRenderscript, 2);   
   
    //Create an element manually and allocate memory for the int pointer   
    intPointer = Allocation.createSized(myRenderscript, Element.I32(myRenderscript), 2);   
   
    //create an instance of the Renderscript, pointing it to the bytecode resource   
    mScript = new ScriptC\_example(myRenderscript, resources, R.raw.example);   
   
    //bind the struct and int pointers to the Renderscript   
    mScript.bind\_touchPoints(touchPoints);   
    script.bind\_intPointer(intPointer);   
   
   ...   
}

**Reading and writing to memory**

- Renderscript runtime, Android framework layer 둘 다에서 정적, 동적으로 할당된 메모리를 읽고 쓰고 할 수 있다.

- 정적으로 할당된 메모리는 RS runtime level에서 one-way communication 제한이 있다. Rs 코드가 변수의 값을 변경할 때, 효율을 목적으로 Android framework layer로 communicate 되지 않는다. Android framewokr에서 마지막으로 set 된 값은 get 메서드가 불릴 대마다 리턴된다. 그러나, Android framework 코드가 변수를 수정하면, 그 변화는 RS runtime에 자동적으로 communicate 될수 있다. 만약 Rs runtime으로부터 Android framework layer에 데이터를 보내고 싶으면 이 제약을 극복하기 위해 rsSendToClient() 메서드를 사용해야 한다. (\*\*정적 할당에서, RS에서의 수정은 Client로 바로 적용되지 않고, Client에서의 수정은 Rs 자동적으로 적용됨. Rs에서의 수정을 Client에 적용하기 위해서는 rsSendToClient() 메서드를 사용해야 함)

-동적으로 할당된 메모리의 경우, 만약 memory allocation을 연결된 포인터로 수정하면, RS runtime layer에서의 모든 변화는 Android framework layer로 전파된다. Android framework layer에서의 오브젝트 수정은 즉시 그 수정사항을 RS runtime layer로 전파한다.

**Reading and writing to global variables**

- global variable의 reading writing은 간단하다. Android framework level에서는 accessor 메서드를 이용하면 되고, RS code에서는 직접 set하면 된다. RS code에서의 변경사항은 Android framework layer로 전파되지 않는 다는 것을 꼭 기억하자

예를 들어, rsFile.rs에 다음과 같은 구조체가 정의되어 있을 때,

typedef struct Point {   
    int x;   
    int y;   
} Point\_t;   
   
Point\_t point;

rsFile.rs에서는 아래와 같이 직접적으로 값을 세팅할 수 있다. 이 값들은 Android framework level로 전달되지 않는다.

point.x = 1;   
point.y = 1;

Android framework layer에서는 값을 아래와 같이 지정한다. 이 값들은 RS runtime level로 전파된다.

ScriptC\_rsfile mScript;   
   
...   
   
Item i = new ScriptField\_Point.Item();   
i.x = 1;   
i.y = 1;   
mScript.set\_point(i);

Rs 코드에서 값을 읽는 것은 다음과 같이 한다.

rsDebug("Printing out a Point", point.x, point.y);

Android framework layer에서 값을 읽는 것을 다음 코드와 같이 한다. 꼭 기억해야 할 것은, 한번은 Android Framework level에서 값을 세팅해야 값을 읽을 때 리턴된다는 것이다. RS runtime level에서만 값을 세팅했었다면 null pointer exception이 발생될 것이다.

Log.i("TAGNAME", "Printing out a Point: " + mScript.get\_point().x + " " + mScript.get\_point().y);   
System.out.println(point.get\_x() + " " + point.get\_y());

**Reading and writing global pointers**

- 메모리가 Android framework level에서 할당되고, RS runtime으로 바인딩 되었다고 생각해 보자, 그러면 Android framework level에서 메모리를 읽고 쓰기 위해 get, set 메서드를 사용할 수 있다.

RS runtime layer에서는 메모리를 읽고 쓰기 위해 보통 pointer를 쓴다. 그리고 그 변화는 정적 메모리 할당에서와 달리, Android framework layer로 바로 전해진다.

예를 들어, rsfile.rs에 다음과 같은 구조체가 있으면,

typedef struct Point {   
    int x;   
    int y;   
} Point\_t;   
   
Point\_t \*point;

메모리 할당은 Android framework layer에서 했다면, 구조체 안의 값에 접근할 수 있다. 포인터 변수를 통해 구조체를 변화 시킨건 자동적으로 Android framework layer에서도 available하다.

point[index].x = 1;   
point[index].y = 1;

Android framework layer에서 포인터의 값을 읽고 쓸 수 있다.

ScriptField\_Point p = new ScriptField\_Point(mRS, 1);   
    Item i = new ScriptField\_Point.Item();   
    i.x=100;   
    i.y = 100;   
    p.set(i, 0, true);   
    mScript.bind\_point(p);   
   
    points.get\_x(0);            //read x and y from index 0   
    points.get\_x(0);

메모리가 한번 바운딩 되면, 값을 변경시킬 때마다 RS runtime으로 메모리 바인딩을 하지 않아도 된다.