國立臺灣大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science and Informantion Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

Remote RenderScript: 利用遠端引擎上之圖形硬體提昇效能表現

Remote RenderScript: Leveraging the Graphics Hardware on the Remote Engine

歐曜瑋

Ou, Yao-Wei

指導教授：廖世偉博士

Advisor: Shih-Wei Liao, Ph.D.

中華民國 100 年 5 月

May, 2011

國立臺灣大學
資訊工程研究所

碩士論文

Remote RenderScript: 利用遠端引擎上之圖形硬體提昇效能表現

歐曜瑋 撰

100
5

致謝

在這碩士兩年來，最先要感謝的就是我的指導教授廖世偉老師。雖然工作繁重，但總是會盡可能地抽空給予我們協助，對我而言，指導教授不像是大家所俗稱的老闆，反而比較像是工作上的夥伴、工作外的朋友。除了課業上的指導外，也關心我們的生涯發展，謝謝老師。

接著要感謝的就是在碩一給我極大鼓勵與幫助的張家榮學長，不論我問什麼問題，總是不厭其煩地耐心指導我。也因為有著老師與學長為榜樣，讓我在碩士生涯中，永遠有個需要努力追求的目標。

還要謝謝三位口試委員：蘇雅韻教授、楊佳玲教授、以及黃世勳教授能撥空參加我的口試，也因為有三位口試委員以及指導教授的指點之下，才得以完成這篇論文。

最後，感謝父母、家人的全力支持，讓我衣食無憂地將心思全力投注在研究上；感謝實驗室的同學、學弟們，研究課題上的討論讓我學習到許多，而課業外因有著你們的陪伴也讓我研究所生活多采多姿。

結束了十八年的學涯後，期許自己能為我們的社會、國家貢獻一些力量。

中文摘要

為了追求 (1) 效能; (2) 可移植性; (3) 使用性，Google 在 Android 3.0 的版本中推出了 RenderScript。我們延伸其功能，使之可遠端化。具體一點的說法，我們透過網路接口傳送 RenderScript 命令到遠端圖形引擎上。請注意，在此所提到的遠端引擎可以是在同一裝置或是遠端裝置之上。

我們的貢獻分為兩個部分：

首先，我們讓 RenderScript 現有的先進先出佇列可以連網。我們建構了一個轉運層以實現網路功能抽象化。

第二，透過把指令傳送到遠端引擎之上，我們可以重新在該引擎上執行所接收而來的命令。視不同型態的遠端引擎而定，重新執行命令的動作可能會需要把 RenderScript 的命令對應到 egl、glx、agl、或是 wgl。其分別對應於不同的平台，依序為 Android, Linux, Mac OS, Windows。在本論文中，我們將敘述對應到 egl 的實作細節。

Abstract

Google introduces RenderScript as part of Android 3.0 for the following reasons: (1) Performance; (2) Portability; and (3) Usability. We extend it to make it remote-able. Specifically, we send RenderScript commands through a socket to a remote graphics engine. We use the term, remote engine, to refer to the engine on the same device or on a remote host.

Our contribution is two-fold. First, we make the existing FIFO queues in RenderScript network-ready. We build the transport layer to facilitate the networking abstraction. Second, we replay the commands on the remote engine by materializing the commands on top of a remote engine. Depending on the type of remote engines, the replay may need to map RenderScript commands to egl, glx, agl, or wgl. They are for Android, Linux, Mac OS, or Windows, respectively. We demonstrate the egl-mapped implementation in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing adoption of smartphones [2], more and more software developers are interested in building mobile applications for smartphones [4]. Among the various types of mobile applications, game is the most popular. According to [3] and [5], by 2010, about 58 percent of the applications in the biggest mobile application store (i.e., Apple's App Store) are games.

However, on Google's Android platform, existing tools for game development are still not perfect. For instance, developers need to deal with the low-level details of JNI (Java Native Interface) calls such as method signature when using Android NDK (Native Development Kit) ; are not able to use GLSL (OpenGL Shading Language) for mathematical computation; and are more difficult to build an application with satisfying performance by using Android SDK (Software Development Kit).

More specifically on the limitation, Java interacts poorly with most current 3D APIs: (1) The interface of Java and native GL is not ideal; (2) Lots of copies or translations needed. Current 2D application performance is not what we would like: (1) Walking through the view hierarchy is slow; (2) Applications need to be called each time we render , e.g., *onDraw()*.; (3) Skia is a poor fit for GL acceleration; (4) We are not in a position to scale to multiple threads; and more importantly, (5) we need 3D apps, Skia performance on larger size screen is unacceptable.

To overcome above limitations, Google introduces a framework, RenderScript [7], in Android SDK 3.0 named Honeycomb. (For simplification, we use RS and Render-

SCRIPT interchangably in this thesis.)

RENDERSCRIPT aims to provide high performance 3D rendering and mathematical computation at the native level, which targets for the following goals: (listed from most to least important) [8]

- **Portability** — Application code needs to be able to run across all devices with different hardware, and be able to fully utilize the capability of various hardwares. For example, ARM currently comes in several variants: with and without VFP, with and without NEON, and with various register counts. Beyond ARM, application code should be able to run on other CPU architectures such as x86, or even run on a GPU or a DSP.

- **Performance** — The second goal is to get as much performance as possible under the requirement of portability.

- **Usability** — The third goal is to simplify development as much as possible by offloading the heavy lifting of interfacing with a native graphics API while retaining full application control. Where possible we automate steps to avoid hand-written glue code and other developer-side busy work.

There are two kinds of RENDERSCRIPTs: compute and graphics. A compute RENDERSCRIPT does not do any graphics rendering while a graphics RENDERSCRIPT does.

A RENDERSCRIPT application comprises both Java codes and native codes. Similar to other Android applications, the Java code in a RENDERSCRIPT application uses Android SDK API. The native code, or the RS script in other words, is actually written in a C99-standard C language with extensions such as vector operations, function overloading, and rsForEach() parallelization. Note that a RENDERSCRIPT application contains at least one RS script, you write and save it to a .rs file in your project. The communication between Java codes and native codes is through JNI function calls. Simply stated, there are **two layers and one bridge in between**:

- Android *System Layer* (**Java Framwork**, i.e., *framework.jar*) : That is, traditional framework APIs, which include the RENDERSCRIPT APIs in `android.renderscript`.

System Layer handles things such as the Activity lifecycle management of your Android application and communicates with the native RENDERSCRIPT code via *Bridge*.

- *Native Layer* (Codes stored in the **native .rs files**) : *Native Layer* is responsible for doing intensive computing and graphics rendering tasks. *Native Layer* returns the result to *System Layer* through *Bridge*.

- *Bridge* (**JNI**, i.e., *librs_jni.so*, and **reflected Java classes**) : *Bridge* deals with the communication between the above two layers. The Android build tools automatically generate the classes for *Bridge* during the build process. RENDERSCRIPT uses lockless FIFO (First In and First Out) command queue to communicate between layers. Figure 1.1 shows the flow of communication.

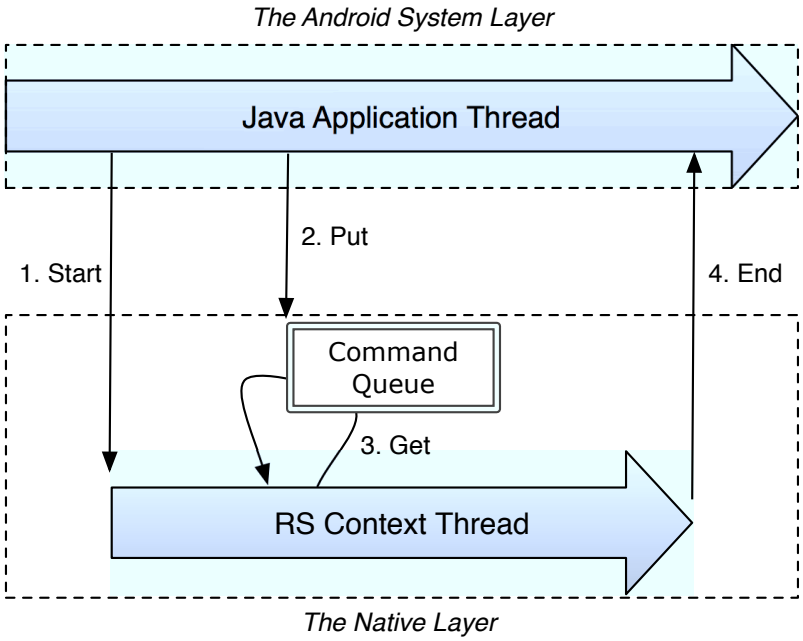

Figure 1.1: Communication between Java and RS code

As Figure 1.1 shows, minimal work is done in the application thread, and no native drawing calls are directly from Java application codes. (i.e., All drawing calls are invoked from *Native Layer*) The RS native thread does heavy lifting: screen can be refreshed without a callback to the Java application thread.

Step 1. When a RENDERSCRIPT application is launched, *System Layer* creates a Java application thread. After being initialized, the Java application thread notifies *Native Layer* to create an RS context thread and establish the command queue via *Bridge*.

Step 2. Both computing and rendering are done at *Native Layer* and UI events are handled in the *System Layer*. Once the Java application thread receive an event, it puts the commands for computation or rendering into the FIFO command queue.

Step 3. If all of the conditions are met, RS context thread will keep polling the command queue until getting a command. While getting a command, the RS context thread executes the command in *Native Layer*. (See Chapter 3.8.1)

Step 4. As the RS context thread receives a destory signal, the control is handed over to *System Layer*.

We will discuss the flow in more details in Chapter 3.

Although the RENDERSCRIPT framework is a major improvement in development of fancy visual effects and other computing intensive application, developers may find it difficult to develop Renderscript applications due to the absence of a debugger or an emulator. Although Google does provide an emulator for Android Honeycomb, it is only starting to support OpenGL ES 2.0, which is required to run RENDERSCRIPT applications.

For reducing the complexity of development and debugging, we need a mechanism which can replay and execute RENDERSCRIPT commands on the remote engine. To accomplish the objective, we propose the REMOTE RENDERSCRIPT architecture in this thesis. The main idea of the REMOTE RENDERSCRIPT architecture consists of two steps. Based on the architecture showed in Figure 1.1, first, we extend RENDERSCRIPT to make it remote-able. To acheive this, we build a *Transport Layer* on the top of *Native Layer*. *Transport Layer* takes care of sending commands of a local RENDERSCRIPT system to a remote RENDERSCRIPT system. Figure 1.2 below illustrates that.

Second, we replay the commands on the remote engine by materializing the commands on top of a remote engine. Depending on the type of remote engines, the replay may need

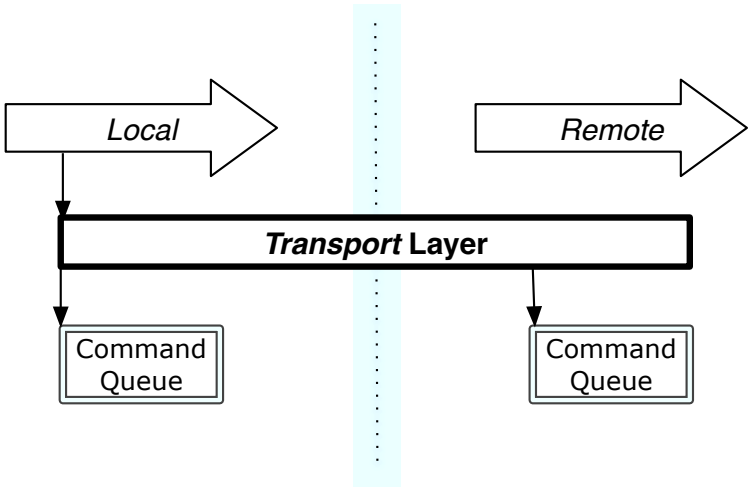

Figure 1.2: Transport Layer

to map RenderScript commands to egl, glx, agl, or wgl[1]. Note that both rendering and computing commands are played in individual engine so that we could leverage the hardwares on the remote engine. The egl-mapped implementation is demonstrated in this thesis. With Remote RenderScript, we could send the commands from emulator and replay it on the remote host. In this scenario, the remote host is local machine. In our implmentaion, extended RenderScript runtime could enable the debugging option by setting the property via ADB (Android Debugging Bridge).

Additionally, Remote RenderScript has the potential to be the foundation of many other applicaiotns in addition to debugging, such as remote controlling, collaborating tools, multi-player games, remote monitoring, and etc.

The rest of the paper is organized as follows. Chapter 2 describes related works. As a background knowledge, Chapter 3 goes through the existing design and implementation of RenderScript system. Chapter 4 illustrates our design of Remote RenderScript. Chapter 5 discusses the egl-mapped implementation for Remote RenderScript. Finally, Chapter 6 concludes the thesis and presents the future works.

[1]They are OpenGL interfaces for Android, Linux, Mac OS, and Windows, respectively.

Chapter 2

Related Work

2.1 Backing-Porting

Two major library in our REMOTE RENDERSCRIPT system: (1) libbcc: LLVM bit-code compiler and (2) libslang: shim + clang (on-the-host). We show the relation and reference of LLVM compiler library in figure 2.1 and the following sections.

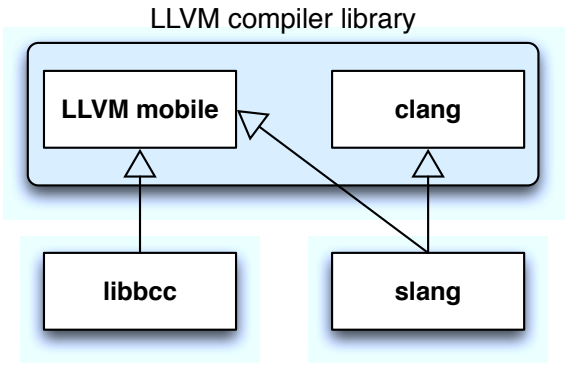

Figure 2.1: The referenece of LLVM compiler library

To deploying REMOTE RENDERSCRIPT, we have to back-port libbcc and libslang from Android Honeycomb to Gingerbread.

Five makefiles are modified for building the whole system:

1. `build/core/config.mk`
2. `build/core/definitions.mk`

3. `build/core/java.mk`

4. `build/core/package.mk`

5. `build/core/prelink-linux-arm.map`

We brief the change as follows:

config.mk sets up standard variables and other configuration like compiler flags.

```
SLANG := $(HOST_OUT_EXECUTABLES)/llvm-rs-cc$(HOST_EXECUTABLE_SUFFIX)
LLVM_RS_LINK := $(HOST_OUT_EXECUTABLES)/llvm-rs-link$(HOST_EXECUTABLE_SUFFIX)

ifeq ($(HOST_OS),darwin)
HOST_GLOBAL_CFLAGS += -arch i386
HOST_GLOBAL_CPPFLAGS += -arch i386
HOST_GLOBAL_LDFLAGS += -arch i386
endif
```

definitions.mk mostly includes standard commands for building various types of targets, which are used by others to construct the final targes.

```
/*************************************************************
/* Find all of the RenderScript files under the named directories.
/*  Meant to be used like:
/*    SRC_FILES := $(call all-renderscript-files-under,src)
/*************************************************************/

define all-renderscript-files-under
$(patsubst ./%,%, \
   +  $(shell cd $(LOCAL_PATH) ; \
   +          find $(1) -name "*.rs" -and -not -name ".*") \
   +  )
endef

/*************************************************************
/* Commands to compile RenderScript
/*************************************************************/

define transform-renderscripts-to-java-and-bc
@echo "RenderScript: $(PRIVATE_MODULE) <= $(PRIVATE_RS_SOURCE_FILES)"
$(hide) rm -rf $(PRIVATE_RS_OUTPUT_DIR)
```

```
$(hide) mkdir -p $(PRIVATE_RS_OUTPUT_DIR)/res/raw
$(hide) mkdir -p $(PRIVATE_RS_OUTPUT_DIR)/src
$(hide) $(SLANG) \
    -o $(PRIVATE_RS_OUTPUT_DIR)/res/raw \
    -p $(PRIVATE_RS_OUTPUT_DIR)/src \
    $(foreach inc,$(PRIVATE_RS_INCLUDES),$(addprefix -I , $(inc))) \
    $(PRIVATE_RS_SOURCE_FILES)
    $(hide) $(LLVM_RS_LINK)    \
    $(PRIVATE_RS_OUTPUT_DIR)/res/raw/*.bc
    $(hide) mkdir -p $(dir $@)
    $(hide) touch $@
```

java.mk takes charge of compiling .java files and .bc files.

```
#############################################################
## .rs files: RenderScript sources to .java files and .bc files
#############################################################
renderscript_sources := $(filter %.rs,$(LOCAL_SRC_FILES))
# Because names of the java files from RenderScript are unknown until the
# .rs file(s) are compiled, we have to depend on a timestamp file.
RenderScript_file_stamp :=
ifneq ($(renderscript_sources),)
renderscript_sources_fullpath := $(addprefix $(LOCAL_PATH)/, $(↩
    renderscript_sources))
RenderScript_file_stamp := $(LOCAL_INTERMEDIATE_SOURCE_DIR)/RenderScript.stamp

# prepend the RenderScript system include path
LOCAL_RENDERSCRIPT_INCLUDES := $(TOPDIR)frameworks/base/libs/rs/scriptc \
    $(LOCAL_RENDERSCRIPT_INCLUDES)

$(RenderScript_file_stamp): PRIVATE_RS_INCLUDES := $(LOCAL_RENDERSCRIPT_INCLUDES)
$(RenderScript_file_stamp): PRIVATE_RS_SOURCE_FILES := $(↩
    renderscript_sources_fullpath)
# By putting the generated java files into $(LOCAL_INTERMEDIATE_SOURCE_DIR), they ↩
    will be
# automatically found by the java compiling function transform-java-to-classes.jar↩
    .
$(RenderScript_file_stamp): PRIVATE_RS_OUTPUT_DIR := $(↩
    LOCAL_INTERMEDIATE_SOURCE_DIR)/renderscript
# TODO: slang support to generate implicit dependency derived from "include" ↩
    directives.
$(RenderScript_file_stamp): $(renderscript_sources_fullpath) $(SLANG)
   $(transform-renderscripts-to-java-and-bc)
```

```
LOCAL_INTERMEDIATE_TARGETS += $(RenderScript_file_stamp)
# Make sure the generated resource will be added to the apk.
LOCAL_RESOURCE_DIR := $(LOCAL_INTERMEDIATE_SOURCE_DIR)/renderscript/res $(↩
    LOCAL_RESOURCE_DIR)
endif

# source files generated from RenderScript must be generated before java compiling
ifneq ($(RenderScript_file_stamp),)
$(full_classes_compiled_jar): $(RenderScript_file_stamp)
endif
```

package.mk includes standard rules for building an application package.

```
$(R_file_stamp): $(all_res_assets) $(full_android_manifest) $(↩
    RenderScript_file_stamp) $(AAPT) | $(ACP)
$(resource_export_package): $(all_res_assets) $(full_android_manifest) $($(↩
    RenderScript_file_stamp)) $(AAPT)
```

prelink-linux-arm.map is a memory-mapping table for prelinking.

```
libbcc.so               0x99000000
```

2.2 slang

1. **Frontend**: Cleverly reuse Clang abstract syntax tree (AST) to reflect information back to Java layer.
2. **Heavey-weight optimizations**: Bcc embeds metadata within bitcode (type, ...) to perform aggressive machine-independent optimizations on host before emitting portable bitcode.
3. All bitcode supplied as a resource within .apk container.

2.3 libbcc

We shrink the size of LLVM by 10 times smaller to fit in a phone so that we could push frontend + heavy-weight optimizations to slang (in build time). We do our own Execution

engine and JIT, while just leveraging LLVM's low-level codegen libraries and have our 3 debugging mechanisms, so we just removed those big dwarf codes.

For a better Performance:

- Fastcc calling convention
- VFPv3
- Use NEON instead of float4
- Wide range of global, scalar optimizations
- 3x speedups over acc

Libbcc design:

- **Method-based JIT**: Reasonable scope for optimization.
- **Ahead-of-Time (AOT) compilation**: Caching of EXE cuts launch time.
- **Delta compilation**: Incremental compilation.
- **Modularity:** Each device will have its own JIT. For ARM devices, just include ARM codegen. Same thing for x86, PowerPC, mips devices.
- **On-device linking**: Specialization by CPU/GPU vendors.
- **Reflection**: Inter-operate across languages.
- **Portability**: Many languages target .bc and .bc targets many hardware.

2.4 LLVM for Android

For reflection support, we have a tailored version of LLVM(Low Level Virtual Machine) for Android. LLVM_mobile is ten times smaller than LLVM and three times faster[1] than acc[2].

LLVM_mobile is utilized as:

[1] more for math-heavy code
[2] old RENDERSCRIPT compiler

1. **Disassembler** on the phone on the JIT results

2. Android's **Self-Verifying Native JIT** —Three steps to eliminate the human assembler+emulator. First, using Disassembler to debug on assemble code. Second, comparing LLVM assemble code with GCC assemble code for locating where LLVM CodeGen bug is Solution. Third, use Anroid toolchain under prebuilt/ Also users libbcc.so, slang, and modified libElf for compare/verify. It's based on 3 assumptions: (1) Clang/LLVM compile the source and output correct .s; (2) Given a ".s", "as" generates correct binary; (3) ARM instructions can be read as a sequence of unsigned.

3. Standalone **bcc** mocking RS apis —Determine whether a pair of statements is possible to access thread-shared data.

2.5 Properties of Android Bitcode

1. ABI-level portability

2. Basic types, structures and functions

3. ILP32(Int, Long, and Pointer) , no LP64

4. Data layout:

   {i32, i32, i32}: A triple of three i32 values

   {float, i32 (i32) *}: A pair, the second element is a pointer to a function that takes an i32, returning an i32

5. Little endian

6. A portable intermediate representation:

   .bc includes target triple in .bc, so runtime JIT can change it.

   .bc includes calling convention such as "Arm Architecture Procedure Call Standards"

   Bitcode didn't materialize it --> runtime JIT can change it

7. Alignment: We don't use LLVM default alignments for space/time/GPU concerns. Alignment issues can't resort to runtime JIT, because code may have offsetof()

Chapter 3

Design and Implementation of RenderScript System

Three major components in RenderScript:

- **Offline** Compiler: *llvm-rs-cc* (stands for **LLVM RS C C**ompiler)
- **Online** JIT Compiler: *libbcc.so* (stands for **Bi**Code **C**ompiler)
- RenderScript **Runtime**: *libRS.so*

When you build your RS project, Android SDK will use *llvm-rs-cc* to compile your .rs file to .bc file (LLVM bitcode), plus reflection. (We will discuss reflection later in section 3.1.3.) The .bc file will be packaged with Java files and other application resources into a APK file. Once you launch the RS applicaiotn after installation, *bcc* would be invoked to just-in-time compile the bitcode. RenderScript runtime then supports the execution of *Native Layer*.

3.1 RenderScript Directory Structure

Let's take a overview on the directories coverd in this thesis.

- **Java SDK** (*System Layer* side):

  You can refer to the offical document for API usage.

`<FRAMEWORKS_BASE>/graphics/java/android/renderscript`

- **Native Engine** (*Native Layer* side):

  `<FRAMEWORKS\_BASE>/libs/rs`

- **JNI Bridge**:

  `<FRAMEWORKS_BASE>/graphics/jni`

- **RS Graphics Commands and API**:

  `out/target/product/<DEV>/obj/SHARED_LIBRARIES/libRS_intermediates/`

- **Reflected Java Code**:

  `<APP_intermediates>/src/renderscript/res/raw`

  `<APP_intermediates>/src/renderscript/src/com/android/fountain`[1]

- **Compiled bitcode**:

  `<APP_intermediates>/src/renderscript/res/raw`

3.1.1 Native Engine

Native Engine means the RENDERSCRIPT runtime, *libRS.so*, and the source is located on `frameworks/base/libs/rs`. All real RENDERSCRIPT implementation is there, including RS graphics-specific header[2].

Some key features of the native RENDERSCRIPT libraries include:

1. A large collection of math functions with both **scalar** and **vector** typed overloaded versions of many common routines. Operations such as adding, multiplying, dot product, and cross product are available.

2. Conversion routines for primitive data types and vectors, matrix routines, date and time routines, and graphics routines.

[1] Path of <APP_intermediates>: <ANDROID_ROOT>/ out/ target/ common/ obj/ APPS/ APP-NAME_intermediates/

[2] in scriptc/

3. Logging functions

4. Graphics rendering functions

5. Memory allocation request features

6. Data types and structures to support the RENDERSCRIPT system such as Vector types for defining two-, three-, or four-vectors.

3.1.2 Java Client

All application is wrapped in Java code, of course, RS application is not the exception. Java Client takes charge of UI handling, collaborates with other SDK code, and do partial initialization of RS application. Android Developer will not be able to get accesses to *Native Layer*. Only Java SDK code is permitted to use, so even RS object(see sec 3.5) should be allocated by invoking Java SDK.

Two major RENDERSCRIPT context classes of Java Client:

- **RenderScript** is for a compute RS script.

- **RenderScriptGL** is for a graphics RS script.

  RENDERSCRIPT provides a number of graphics APIs for hardware-accelerated 3D rendering. The RENDERSCRIPT graphics APIs include a stateful context, RenderScriptGL that contains the current rendering state. The primary state consists of the objects that are attached to the rendering context, which are the graphics RENDERSCRIPT and the four program types.[3] Graphical scripts have more properties beyond a basic computational script, and they call the 'rsg'-prefixed functions defined in the rs_graphics.rsh header file.

With above two classes, you could bind to the reflected RENDERSCRIPT class, so that the RENDERSCRIPT context knows what its corresponding native RENDERSCRIPT is. If you have a graphics RENDERSCRIPT context, you can also specify a variety of Programs

[3]The four program types mirror a traditional graphical rendering pipeline and are: (1)Vertex; (2)Fragment; (3)Store; and (4)Raster.

(stages in the graphics pipeline) to tweek how your graphics are rendered. A graphics RENDERSCRIPT context also needs a surface to render on, *RSSurfaceView*, which gets passed into RSSurfaceView's constructor.

The Android system APIs are broken into a few main groups [1]:

- **Allocation APIs** are used internally by the system for memory allocation. They are used by the classes that are generated by the build tools:
- **Data Types** are used by the classes that are generated by the build tools. They are the reflected counterparts of the native data types that are defined by the native RS APIs and used by your RS code.
- **Graphics APIs** are specific to graphics RENDERSCRIPT and support a typical rendering pipeline.

Table 3.1: Major gourps of APIs

Allocation	Allocation, Element, Type, Script
Data Types	Byte2, Byte3, and Byte4 Float2, Float3, Float4 Int2, Int3, Int4 Long2, Long3, Long4 Matrix2f, Matrix3f, Matrix4f Short2, Short3, Short4
Graphics	Mesh ProgramFragment, ProgramRaster ProgramStore, ProgramVertex RSSurfaceView

3.1.3 Bridge in between

Bridge provides the entry points into the native code, enabling the Android system to give high level commands like, "rotate the view" or "filter the bitmap" to *Client*, which does the heavy lifting. To accomplish this, you need to create logic to hook together all of these ones so that they can correctly communicate.

libRS.so is written in C++, so the way to communicate beteen **Core** and **Client** is using *JNI*. This mapping is done in libjni.so. The mapping function table is as below:

```
static JNINativeMethod methods[] = {
    ...
{"rsnContextBindRootScript",      "(II)V", (void*)nContextBindRootScript },
{"rsnContextBindProgramStore",    "(II)V", (void*)nContextBindProgramStore },
{"rsnContextBindProgramFragment", "(II)V", (void*)nContextBindProgramFragment },
{"rsnContextBindProgramVertex",   "(II)V", (void*)nContextBindProgramVertex },
{"rsnContextBindProgramRaster",   "(II)V", (void*)nContextBindProgramRaster },
    ...
}
```

Besides JNI, reflected Java classes work as *Bridge*.

Reflection generates the class named *ScriptC_FILENAME.java* from a RENDERSCRIPT file that has the .rs file extension. *ScriptC_fountain.java* which is the reflective version of the RENDERSCRIPT and contains the entry points into the fountain.rs native code. This class does not appear until you run a build.

Any non-static, global RENDERSCRIPT variables are reflected into *ScriptC_FILENAME.java*. Accessor methods are generated, so *Client* can access the values. The get method comes with a one-way communication restriction. *Client* always caches the last value that is set and returns that during a call to a get method. If the native RENDERSCRIPT code changes the value, the change does **not** propagate back to *Client*. If the global variables are initialized in the native RENDERSCRIPT code, those values are used to initialize the corresponding values in *Client*. If global variables are marked as `const`, then a set method is not generated.

Structs are reflected into their own classes, one for each struct, into a class named *ScriptField_STRUCT.java* of type `Script.FieldBase`.

Global pointers have a special property. They provide attachment points where *Client* can attach allocations. If the global pointer is a user defined structure type, it must be a type that is legal for reflection (primitives or RENDERSCRIPT data types). *Client* can call the reflected class to allocate memory and optionally populate data, then attach it to the RENDERSCRIPT. For arrays of basic types, the procedure is similar, except a reflected class is not needed. RENDERSCRIPT should not directly set the exported global pointers.

3.1.4 Subtle Function Name

- **android_renderscript_RenderScript.cpp**(It's auto-generated):

```
{"rsnScriptCCreate", "(ILjava/lang/String;)I", (void*)nScriptCCreate }'
Javas native method                  Native method but has JNI
```

- BTW, in the same file we have:

```
static jint nScriptCCreate(JNIEnv *_env, jobject _this,
{
  RsContext con, jstring resName)
  LOG_API("nScriptCCreate, con(%p)", con);
  const char* resNameUTF = _env->GetStringUTFChars(resName, NULL);
  return (jint)rsScriptCCreate(con, resNameUTF);
  // Note: rsScriptCCreate is created based on rs.spec.
  // It will just call rsi_ScriptCCreate
}
```

- RenderScript.java:

```
native int rsnScriptCCreate(int con, String val);
synchronized int /*Better name: NScriptCCreate*/ nScriptCCreate(String {
  // Terrible naming. This "nScriptCCreate" is different from
  // the (void*)nScriptCCreate above. This "nScriptCCreate" is invoked // ↩
      ScriptC.java
  return rsnScriptCCreate(mContext, val);
}
```

3.2 RenderScript Commands

RS Commands to JNI: They are generated at Build Time. Figure 3.1 shows how it works.

spec.l and *spec.h* are files for **rsg_generator**[4] to scan *rs.spec* so that it could generate RS Commands related implementation, header files, etc. (Output path see sec.3.1)

3.2.1 API Prefix

The star marks in the following table usually follow the rules [Class name][Action]. Take "rsi_ScriptSetClearDepth()" as example, [Class name] is "Script" and [Action] is

[4] g stands for "g"raphics.

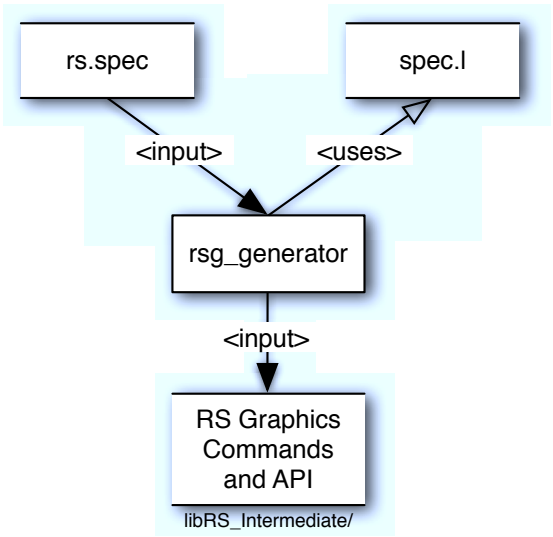

Figure 3.1: RS graphics API generator

"SetClearDepth". Thus, its implementation would be found under frameworks/base/libs/rs (libRS.so, since its prefix is rsi_) in rsScript.cpp (by [Class name].)

Table 3.2: RS API Prefix Table

Component	Prefix	Comment
libRS.so		File names are prefixed with rs*
	rsi_*()[5]	RS API real implementation [6]
	rsa_*()	RS allocation API
RS Command	rs*()[7]	the commands, *rsgApi.cpp*
	RS_CMD_*	parameter data structure for the command
	RS_CMD_ID_*	command ID [8]
	rsp_*	playback function [9]
JNI	n*()	**n** stands for native

We illustrate the usage of each kind of prefix in Figure 3.2. Why not call rsi_ScriptSetVarF(...)? It's because that only ID and a pointer could be pulled out of a command. To call rsi_ScriptSetVarF(...), we need exact arguments. Hence we do a casting with known structure information in *rsgApiStruct.h*.

[5]For example, rsi_ScriptCreate(), which will call *ScriptCState::runCompiler()*

[6]The rs*() finally routes here

[7]an integer constant

[8]rs*()->...->rsp_*()->rsi_*(), *rsgApiReplay.cpp*

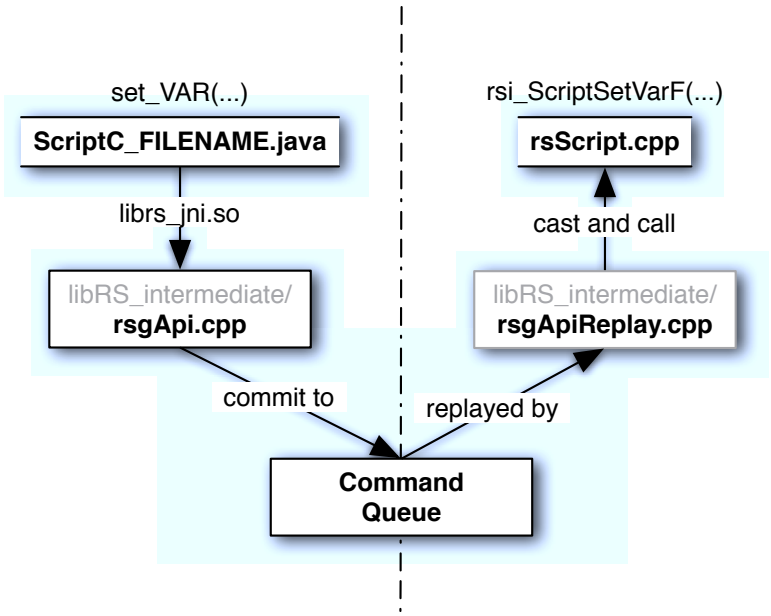

Figure 3.2: A command set_VAR(value) called from *System Layer* and executed in *Native Layer*.

3.3 The Fountain RS Application

For an example of RENDERSCRIPT in action, see the 3D carousel view in the Android 3.0 versions of Google Books and YouTube or install the RENDERSCRIPT sample applications that are shipped with the SDK.

In this thesis, we take *Fountain* for instance. *Fountain* acts just like its name. If any touches on screen, *Fountain* spraies out points in direct proportion of the pressure.

In *Fountain*, each file has its own distinct use. The following files comprise the main parts of the sample and demonstrate in detail how the sample works:

- **Fountain.java** --- The main Activity for the application. This class is present to provide Activity lifecycle management. It mainly delegates work to FountainView, which is the RENDERSCRIPT surface that the sample actually draws on.

- **FountainView.java** --- The RENDERSCRIPT surface that the graphics render on. If you are using RENDERSCRIPT for graphics rendering, you must have a surface to render on. If you are using it for computatational operations only, then you do not need this.

- **FountainRS** --- The class that calls the native RENDERSCRIPT code through high level entry points that are generated by the Android build tools.

- **fountain.rs** --- The RENDERSCRIPT native code that draws the text on the screen.

The <project_root>/gen directory contains the reflected layer classes that are generated by the Android build tools. You will notice a ScriptC_fountain class, which is the reflective version of the RENDERSCRIPT and contains the entry points into the fountain.rs native code. This file does not appear until you run a build.

ScriptC_fountain This class is generated by the Android build tools and is the reflected version of the fountain.rs RENDERSCRIPT. It provides a a high level entry point into the fountain.rs native code by defining the corresponding methods that you can call from the traditional framework APIs.

fountain.bc bitcode This file is the intermediate, platform-independent bitcode that gets compiled on the device when the RENDERSCRIPT application runs. It is generated by the Android build tools and is packaged with the .apk file and subsequently compiled on the device at runtime. This file is located in the <project_root>/res/raw/ directory and is named rs_FILENAME.bc. You need to bind these files to your RENDERSCRIPT context before call any RENDERSCRIPT code from your Android application. You can reference them in your code with R.id.rs_FILENAME.

FountainView class This class represents the Surface View that the RENDERSCRIPT graphics are drawn on. It does some administrative tasks in the ensureRenderScript() method that sets up the RENDERSCRIPT system. This method creates a RenderScriptGL object, which represents the context of the RENDERSCRIPT and creates a default surface to draw on (you can set the surface properties such as alpha and bit depth in the RenderScriptGL.SurfaceConfig class). When a RenderScriptGL is instantiated, this class calls the HelloRS class and creates the instance of the actual RENDERSCRIPT graphics renderer. This class also handles the important lifecycle events and relays touch events

to the RENDERSCRIPT renderer. When a user touches the screen, it calls the renderer, FountainRS and asks it to draw the text on the screen at the new location.

FountainRS class This class represents the RENDERSCRIPT renderer for the FountainView Surface View. It interacts with the native RENDERSCRIPT code that is defined in fountain.rs through the interfaces exposed by ScriptC_fountain. To be able to call the native code, it creates an instance of the RENDERSCRIPT reflected class, ScriptC_fountain. The reflected RENDERSCRIPT object binds the RENDERSCRIPT bitcode (R.raw.fountain) and the RENDERSCRIPT context, RenderScriptGL, so the context knows to use the right RENDERSCRIPT to render its surface.

3.3.1 Root Script

Root Script means a script with root function: The main working function of the graphics RENDERSCRIPT is the code that is defined in the root() function. The root() function is called each time the surface goes through a frame refresh.

root() This function is the default worker function for this RENDERSCRIPT file. For graphics RENDERSCRIPT applications, like this one, the RENDERSCRIPT system expects this function to render the frame that is going to be displayed. It is called every time the frame refreshes. The root() function for the fountain.rs script moves all pixels down to simulate the dropping **every 1ms**.

```
1  int root() {
2      float dt = min(rsGetDt(), 0.1f);
3      rsgClearColor(0.f, 0.f, 0.f, 1.f);
4      const float height = rsgGetHeight();
5      const int size = rsAllocationGetDimX(rsGetAllocation(point));
6      float dy2 = dt * (10.f);
7      Point_t * p = point;
8
9      for (int ct=0; ct < size; ct++) {
10         p->delta.y += dy2;
11         p->position += p->delta;
12         if ((p->position.y > height) && (p->delta.y > 0)) {
13             p->delta.y *= -0.3f;
```

```
14            }
15            p++;
16        }
17
18        rsgDrawMesh(partMesh);
19        return 1;
20    }
```

init() This function is called once for each instance of this RENDERSCRIPT file that is loaded on the device, before the script is accessed in any other way by the RENDERSCRIPT system. The init() is ideal for doing one time setup after the machine code is loaded such as initializing complex constant tables. The init() function for the fountain.rs script sets the initial location of the text that is rendered to the screen:

3.3.2 old-version, libacc

As we mentioned in the begining of this chapter, there are three major components in RENDERSCRIPT system. In the earlier version of Android (v2.0 - v2.3.3), only two component in RENDERSCRIPT system: (1) online compiler libacc; and (2) RENDERSCRIPT runtime.

libacc is a small C compiler, not like libbcc which compiles bitcode, it compiles normal native C code. Figure 3.3 shows the old workflow. We could found that the old workflow is difficult to use and debug. Not until running the application can we find the compile error.

3.3.3 new-version, libbcc

To overcome those shortages, Google replaces libacc with *libbcc* and *llvm-rs-cc*. Both *libbcc* and *llvm-rs-cc* are derived from LLVM (Low Level Virtual Machine) compiler system. (We will discuss more on Chapter 2.)

Figure 3.4 shows the new workflow in Android v3.0.

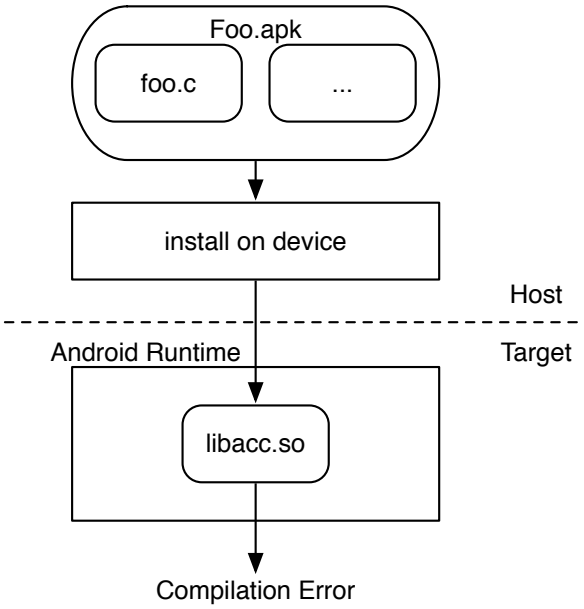

Figure 3.3: workflow of libacc

3.4 Core Design Choices

Those three goals lead to several design trade-offs. It's these trade-offs that separate RENDERSCRIPT from the existing approaches on the device, such as Dalvik or the NDK. They should be thought of as different tools intended to solve different problems.

Performance runtime thread-launch management

Portability NEON, fragmentation. To achieve this, the Android build tools compile your RENDERSCRIPT .rs file to intermediate bitcode and package it inside your application's .apk file. On the device, the bitcode is compiled (just-in-time) to machine code that is further optimized for the device that it is running on. This eliminates the need to target a specific architecture during the development process. The compiled code on the device is cached, so subsequent uses of the RENDERSCRIPT enabled application do not recompile the intermediate code.

RENDERSCRIPT code is compiled and executed in a compact and well defined runtime, which has access to a limited amount of functions. RENDERSCRIPT cannot use the NDK or standard C functions, because these functions are assumed to be running on a

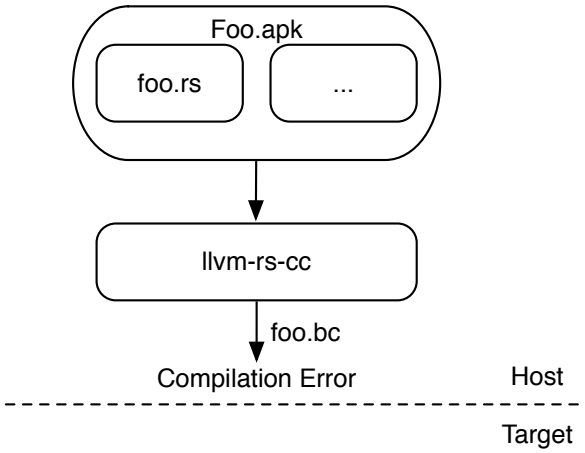

Figure 3.4: workflow of libbcc

standard CPU. The RENDERSCRIPT runtime chooses the best processor to execute the code, which may not be the CPU, so it cannot guarantee support for standard C libraries. What RENDERSCRIPT does offer is an API that supports intensive computation with an extensive collection of math APIs.

Usability workflow, C99 comfortable with developing in C (C99 standard)

Usability was a major driver in RENDERSCRIPT design. Most existing compute and graphics platforms require elaborate glue logic to tie the high performance code back to the core application code. This code is very bug prone and usually painful to write. The static analysis we do in *llvm-rs-cc* is helpful in solving this issue. Each user script generates a Dalvik "glue" class. Names for the glue class and its accessors are derived from the contents of the script. This greatly simplifies the use of the scripts from Dalvik.

3.5 RS Objects

The usage of RS objects is for graphics operation of the pipeline in figure 3.5. We only introduces ten essential objects in this section. For full description, please refer to [1].

Element An Element is the most basic element of a memory type. An element represents one cell of a memory allocation. An element can have two forms: Basic or Complex. They

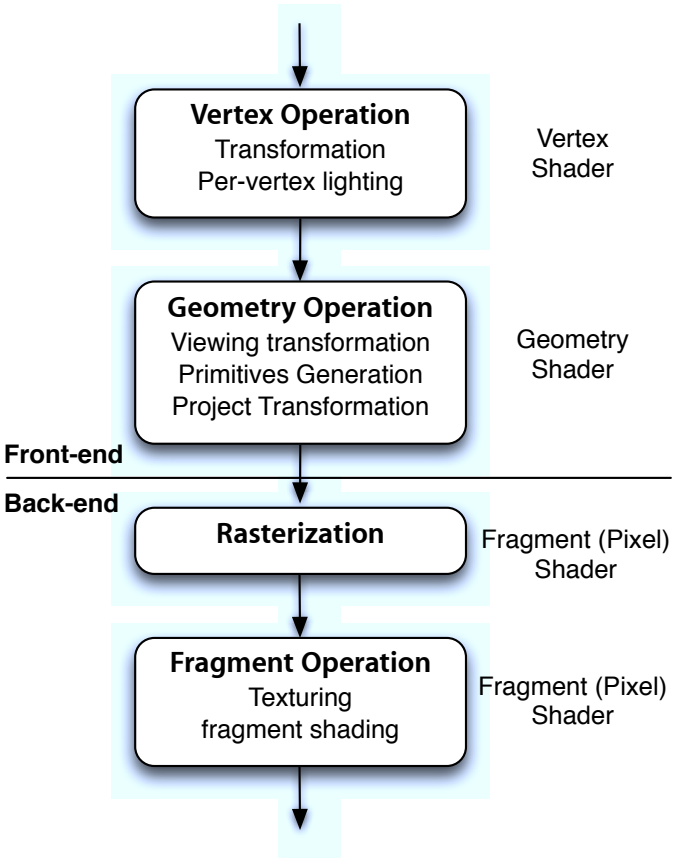

Figure 3.5: The typical graphic pipeline

are typically created from C structures in your RENDERSCRIPT code during the reflection process. Elements cannot contain pointers or nested arrays. The other common source of elements is bitmap formats. A basic element contains a single component of data of any valid RENDERSCRIPT data type. Examples of basic element data types include a single float value, a float4 vector, or a single RGB-565 color.

Complex elements contain a list of sub-elements and names that is basically a reflection of a C struct. You access the sub-elements by name from a script or vertex program. The most basic primitive type determines the data alignment of the structure. For example, a float4 vector is alligned to sizeof(float) and not sizeof(float4). The ordering of the elements in memory are the order in which they were added, with each component aligned as necessary.

In short, Element contains basic types and other Elements. Each entry is a type/Element and name pair. Element is very similar to C structure typedef, except no array

support.

Type A collection of elements assigned a dimensional structure. For example, R8G8B8 with height = 512, width = 256.

A Type is an allocation template that consists of an element and one or more dimensions. It describes the layout of the memory but does not allocate storage for the data that it describes. A Type consists of five dimensions: X, Y, Z, LOD (level of detail), and Faces (of a cube map). You can assign the X,Y,Z dimensions to any positive integer value within the constraints of available memory. A single dimension allocation has an X dimension of greater than zero while the Y and Z dimensions are zero to indicate not present. For example, an allocation of x=10, y=1 is considered two dimensional and x=10, y=0 is considered one dimensional. The LOD and Faces dimensions are booleans to indicate present or not present.

Allocation An Allocation provides the memory for applications, simply a Type with storage allocated to hold its contents. An Allocation allocates memory based on a description of the memory that is represented by a Type. The type describes an array of elements that represent the memory to be allocated. Allocations are the primary way data moves into and out of scripts.

Memory is user-synchronized and it's possible for allocations to exist in multiple memory spaces concurrently. For example, if you make a call to the graphics card to load a bitmap, you give it the bitmap to load from in the system memory. After that call returns, the graphics memory contains its own copy of the bitmap so you can choose whether or not to maintain the bitmap in the system memory. If the RENDERSCRIPT system modifies an allocation that is used by other targets, it must call syncAll() to push the updates to the memory. Otherwise, the results are undefined.

Allocation data is uploaded in one of two primary ways: type checked and type unchecked. For simple arrays there are copyFrom() functions that take an array from the Android system and copy it to the native layer memory store. Both type checked and unchecked copies are provided. The unchecked variants allow the Android system to copy over arrays of

structures because it does not support structures. For example, if there is an allocation that is an array n floats, you can copy the data contained in a float[n] array or a byte[n*4] array. Script

Vertex, Fragment The RENDERSCRIPT vertex program, also known as a vertex shader, describes the stage in the graphics pipeline responsible for manipulating geometric data in a user-defined way. The object is constructed by providing RENDERSCRIPT with the following data:

- An Element describing its varying inputs or attributes
- GLSL shader string that defines the body of the program
- a Type that describes the layout of an Allocation containing constant or uniform inputs

The RENDERSCRIPT fragment program, also known as the fragment shader, is responsible for manipulating pixel data in a user-defined way. It's constructed from a GLSL shader string containing the program body, textures inputs, and a Type object describing the constants used by the program. Like the vertex programs, when an allocation with constant input values is bound to the shader, its values are sent to the graphics program automatically. Note that the values inside the allocation are not explicitly tracked. If they change between two draw calls using the same program object, notify the runtime of that change by calling rsgAllocationSyncAll so it could send the new values to hardware. Communication between the vertex and fragment programs is handled internally in the GLSL code. For example, if the fragment program is expecting a varying input called varTex0, the GLSL code inside the program vertex must provide it.

Raster Contains the state used to rasterize primitives from Vertex programs and feed the Fragment program engine. Simply point, lines, triangles for now. Program raster is primarily used to specify whether point sprites are enabled and to control the culling mode. By default back faces are culled.

Store Program which encapsulates the state (and possibly later programmable operations) that occur when a fragment is written to a buffer. It contains DepthTest, StencilTest, Blend, Masks, etc.

The RENDERSCRIPT ProgramStore contains a set of parameters that control how the graphics hardware writes to the framebuffer. It could be used to enable and disable depth writes and testing, setup various blending modes for effects like transparency and define write masks for color components.

Script Scripts are small self-contained chunks of code that performans tasks on the programs behalf. They are compiled from C99 to llvm bitcode on the host during the build process.Several extensions to the C99 language are supported. For example, Vector types (float4, uchar4, float2, ...) and function overloading.

- Scripts run in a well defined environment.
- They can only access a short list of functions.
- They have a small fixed size stack.
- They cannot directly share data.
- They cannot allocate data.
- They may call other scripts.
- They may change the state of the rendering context.
- They may be run concurrently with other scripts.
- They may be run concurrently with themselves.

RENDERSCRIPT scripts do much of the work in the native layer. Scripts are processed at build time. From this processing two types of Java files are generated. **A ScriptC_FILENAME.java** is generated for each script. This contains code for:

- Initializing the script

- Get and Set of basic global variables
- Binding of allocation
- Invocation of functions

0 or more ScriptField_typename.java files. These are generated for any user structures exported at pointers. Provides an RS Element matching the structure type in the file.

- Error if an element cannot be created.
- Provides packing and unpacking support to transport data from java to the RS environment.
- Script data is broken into two categories.
- Basic globals not declared as static
- non pointer
- basic types
- generate get methods in the java class. Initialized to same values as script.
- If not const, generates a set method in the java class.
- Java class caches the value from set. Sets are propagated from java to RS, however, RS updates are not propagated back to java.
- Data updates always occur while the script is otherwise idle.
- Writing to globals will restrict the script to single threaded execution.
- Pointer globals, non static.
- Can be basic or user type.
- User types limited to the supported set of possible elements, mostly means no arrays.

- generate get methods in the java class initialized to null.
- generates a bind method. Bind point takes either an allocation for basic types or matching ScriptField_typename object for user defined types.
- Data updates always occur while the script is otherwise idle.
- Only method for writing data visible beyond the script.

Mesh A collection of allocations that represent vertex data (positions, normals, texture coordinates) and index data such as triangles and lines. Vertex data can be interleaved within one allocation, provided separately as multiple allocation objects, or done as a combination of the above. The layout of these allocations will be extracted from their Elements. When a vertex channel name matches an input in the vertex program, RENDERSCRIPT automatically connects the two. Moreover, even allocations that cannot be directly mapped to graphics hardware can be stored as part of the mesh. Such allocations can be used as a working area for vertex-related computation and will be ignored by the hardware. Parts of the mesh could be rendered with either explicit index sets or primitive types.

Font This class gives you a way to draw hardware accelerated text. Internally, the glyphs are rendered using the Freetype library, and an internal cache of rendered glyph bitmaps is maintained. Each font object represents a combination of a typeface and point sizes. Multiple font objects can be created to represent faces such as bold and italic and to create different font sizes. During creation, the framework determines the device screen's DPI to ensure proper sizing across multiple configurations.

Font rendering can impact performance. Even though though the state changes are transparent to the user, they are happening internally. It is more efficient to render large batches of text in sequence, and it is also more efficient to render multiple characters at once instead of one by one.

Font color and transparency are not part of the font object and can be freely modified in the script to suit the your needs. Font colors work as a state machine, and every new

call to draw text will use the last color set in the script.

3.6 Memory Allocation for RenderScript

1. Allocation = Blocks of memory for a Type

2. Type = Element + ArrayDims

3. Element = struct + name

Element, Type, Allocation, and Script. These classes are mainly used by the reflected classes that are generated from your native RENDERSCRIPT code. They allocate and manage memory for your RENDERSCRIPT on the Android system side. You normally do not need to call these classes directly.

3.6.1 Reference Counting

Because of the constraints of *Native Layer*, you cannot do any dynamic memory allocation in your RENDERSCRIPT .rs file. *Native Layer* can request memory from *System Layer*, which allocates memory for you and does reference counting to figure out when to free the memory. A memory allocation is taken care of by the Allocation class and memory is **requested in your** RENDERSCRIPT **code** with the the rs_allocation type. All references to RENDERSCRIPT objects are counted, so when your RENDERSCRIPT native code or system code no longer references a particular Allocation, it destroys itself. Alternatively, you can call destroy() from *System Layer*, which decreases the reference to the Allocation. If no references exist after the decrease, the Allocation destroys itself. The Android system object, which at this point is just an empty shell, is eventually garbage collected.

For example, you could manage your objects by rsSetObject() and rsClearObject() [6].

3.7 Bridge

The "physical" bridge between client side and core side is a FIFO. JNI receives the client's requests and calls the corresponded RS command to manipulate the object such as Context and Allocation at the Core side.

Key diagram:

- *rsgAPI.cpp*: Push commands into the FIFO
- *rsgApiReplay.cpp*: Pop commands out of FIFO

Details: Each files below contains a number that instructs rsg_generator about the type of output (and the output will save to the filename with .rsg removed):

`frameworks/base/libs/rs/rsgApi.cpp.rsg`

`frameworks/base/libs/rs/rsgApiFuncDecl.h.rsg`

`frameworks/base/libs/rs/rsgApiReplay.cpp.rsg`

`frameworks/base/libs/rs/rsgApiStructs.h.rsg`

3.7.1 Command Queue

In RENDERSCRIPT, there are two kind of command queue.

- *mToClient*: Commands for passing message to *System Layer*
- *mToCore*: Commands to be executed in *Native Layer*

Note that both of them are lockless, which means atomic operation is used instead of pthread lock.

After the compilation with *llvm-rs-cc*, four files are generated:

`rsgApi.cpp`

`rsgApiFuncDecl.h`

`rsgApiReplay.cpp`

`rsgApiStructs.h`

RENDERSCRIPT runtime commits commands through *rsgApi.cpp*, take `rsContextSetSurface` for example:

```
void rsContextSetSurface (RsContext rsc, uint32_t width, uint32_t height, ANativeWindow * sur)
{
    ThreadIO *io = &((Context *)rsc)->mIO;
    RS_CMD_ContextSetSurface *cmd = static_cast<RS_CMD_ContextSetSurface *>(io->mToCore.reserve(sizeof(RS_CMD_ContextSetSurface)));
    uint32_t size = sizeof(RS_CMD_ContextSetSurface);
    cmd->width = width;
    cmd->height = height;
    cmd->sur = sur;
    // commit to the LocklessFIFO command queue
    io->mToCore.commitSync(RS_CMD_ID_ContextSetSurface, size);
};
```

The command queue keeps polling and pull the command to execute once there is any command in queue:

```
bool ThreadIO::playCoreCommands(Context *con, bool waitForCommand) {
    bool ret = false;
    while (!mToCore.isEmpty() || waitForCommand) {
        uint32_t cmdID = 0;
        uint32_t cmdSize = 0;
        ret = true;
        if (con->props.mLogTimes) {
            con->timerSet(Context::RS_TIMER_IDLE);
        }
        const void * data = mToCore.get(&cmdID, &cmdSize);
        if (!cmdSize) {
            // exception occured, probably shutdown.
            return false;
        }
        if (con->props.mLogTimes) {
            con->timerSet(Context::RS_TIMER_INTERNAL);
        }
        waitForCommand = false;
        con->timerPrint();
        if (cmdID >= (sizeof(gPlaybackFuncs) / sizeof(void *))) {
            rsAssert(cmdID < (sizeof(gPlaybackFuncs) / sizeof(void *)));
            LOGE("playCoreCommands error con %p, cmd %i", con, cmdID);
            mToCore.printDebugData();
        }
        // execute the command
        gPlaybackFuncs[cmdID](con, data);
        mToCore.next();
    }
```

```
29
30      return ret;
31  }
```

Then it calls the function in graphics playback function array, which is rsp prefixed:

```
1  void rsp_ContextSetSurface(Context *con, const void *vp)
2  {
3      const RS_CMD_ContextSetSurface *cmd = static_cast<const ↩
           RS_CMD_ContextSetSurface *>(vp);
4      rsi_ContextSetSurface(con,
5              cmd->width,
6              cmd->height,
7              cmd->sur);
8  };
```

Finally, it calls the real implementation, which is rsi prefixed:

```
1  void rsi_ContextSetSurface(Context *rsc, uint32_t w, uint32_t h, ANativeWindow *↩
       sur) {
2      rsc->setSurface(w, h, sur);
3  }
```

3.8 Run-time Thread-launchment in RS

Three major threads are in RENDERSCRIPT.

1. **Context Thread** is the main thread in *Native Layer*
2. **Helper Thread**s are created when *rsForEach()* is invoked and the number of CPU is in direct proportion to the helper thread.
3. **Message Thread** is used for *Native Layer* to deal with *System Layer*.

3.8.1 Three Condition to Execute

1. A surface is prepared to be drawn on.
2. The root script[10] is compiled and binded.
3. The RS script is for graphics rendering.

[10] A RS script with a root function.

Chapter 4

Remote RenderScript

RenderScript is great but still not perfect. The disadvantage of the RenderScript system is that it adds complexity to the debugging processes. Debugging visibility can be limited, because the RenderScript system can execute on processors other than the main CPU (such as the GPU), so if this occurs, debugging becomes more difficult. Testing ability is also limited, by now, RenderScript applicaiton could only run on a device rather than the emulator on Android SDK 3.0. The reason is the emulator has not supported OpenGL ES 2.0 yet, which is required for running the RS application.

Consequently, we propose Remote RenderScript for two main reasons:

- Reducing the complexity of the development and debugging processes
- Punching through and leveraging the hardwares of remote engine for outperforming QEMU-emulated ARM

The main idea of the proposed design is that to replay the RS command on another platform which supports OpenGL ES 2.0. We can do the following steps to achieve this: (1) Modify RenderScript to make it able to send RS command to another RS context thread, 2. Port RenderScript to the platform which supports OpenGL, and make it able to get RS commands from other RS system and execute them. (see Section 4.2)

4.1 Debugging and Leveraging

To reduce the complexity of development and debugging processes, we propose REMOTE RENDERSCRIPT as a debugging framework. For example, assemed that we run a RS applicaiton in the emulator of a x86 destop PC. Although the result could not be successfully displayed, we bypass the command to the remote engine. In this case, the remote engine is still the same with the computer on which the emulator runs.

By materializing the commands on top of a remote engine, remote engine does the heavy lift of computation and graphics operation and finally outputs the result on another window (i.e., a surface to draw on). Simply put, a desktop runs two program, one is the emulator and the other one is RS debugger. So-called RS debugger is actually a RS command receiver. The difference between the programs is the execution environment, which runs on x86 and QEMU-emulated ARM respectively.

We could learn that the performance of RS application running on the top of RS debugger must be better than the other one, because it leverages CPU and graphics hardware. Even the emulator supports OpenGL ES 2.0 in the future, REMOTE RENDERSCRIPT still has its uniqueness with respect to the debugging.

4.2 Command Sending

Chaper 3 introduces both command queue and RS command. And now we go through how they function step by step.

```
1  rsc->mRunning = true;
2  bool mDraw = true;
3  while (!rsc->mExit) {
4      mDraw |= rsc->mIO.playCoreCommands(rsc, !mDraw);
5      mDraw &= (rsc->mRootScript.get() != NULL);
6      mDraw &= (rsc->mWndSurface != NULL);
7
8      uint32_t targetTime = 0;
9      if (mDraw && rsc->mIsGraphicsContext) {
10         targetTime = rsc->runRootScript();
11
12         if (rsc->props.mLogVisual) {
```

```
13              rsc->displayDebugStats();
14          }
15
16          mDraw = targetTime && !rsc->mPaused;
17          rsc->timerSet(RS_TIMER_CLEAR_SWAP);
18          eglSwapBuffers(rsc->mEGL.mDisplay, rsc->mEGL.mSurface);
19          rsc->timerFrame();
20          rsc->timerSet(RS_TIMER_INTERNAL);
21          rsc->timerPrint();
22          rsc->timerReset();
23      }
24      if (targetTime > 1) {
25          int32_t t = (targetTime - (int32_t)(rsc->mTimeMSLastScript + rsc->↩
                mTimeMSLastSwap)) * 1000;
26          if (t > 0) {
27              usleep(t);
28          }
29      }
30  }
```

We found that *Native Layer* keep polling command queue until it gets a command to execute. For simple one-way screen sharing, a straightforward thinking is send the command to the remote engine along with the committing to the local device.

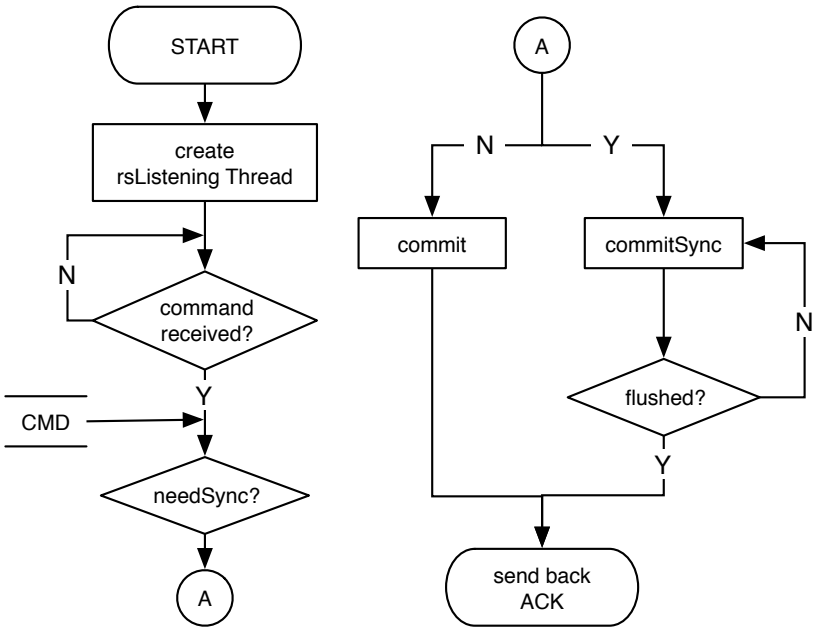

Figure 4.1: Flowchart of rsListening thread

4.3 Network Ready

Follow the thinking, we need a additional transport layers to make RenderScript network-ready, as we said in chapter 1. To facilitate this, we add a thread in addtion to threads we mentions in section 3.4. We name it *RS Listening Thread*, it takes over received commands and commits them to the remote engine. We expose the implmentation details in section 5.6.

4.4 Challenge

With transport layer, we could send the command to another end. But the problem is, where is more appropriate to take send() action? A direct approach is putting send() right after commit(). We verify our idea by implementation, the idea is right in principle but there are some issues not considered.

1. Replicated Command
2. Command Synchronization
3. Allocation of Command Queue
4. The Pointer in Arguments

We describe our approach to solve these issues in the last four section of chapter 5.

Chapter 5

Implementation

In the thesis, we verify our design by executing *Fountain* on REMOTE RENDERSCRIPTand selecting *egl* as a remote engine. We present the implementation details in this chapter, and describe how we solve issues in the last four section.

5.1 Command Encoding

The following figure shows the encoding of command whose command structure is RS_CMD_ScriptInvokeV. The command calls a function declared in *Native Layer*. In Fountain, the function is adding particulars on the screen and is called everytime the screen touched.

The first field of the command is CMD_ID, which is a integer(i.e., *uint32_t*, 4 bytes) representing a CMD type. The CMD_ID table is defined in *rsgApiStructs.h*. The second field is the size used in the command. To get the size, we should know what fields are in the structure of RS_CMD_ScriptInvokeV. According to the figure 5.1, we could know that there are four fields in the structure. RsScript is a pointer to *Script* object.(mentioned in 3.5), slot means the slot number for reflection, and the last two fields stores the informantion of the funciton arguments which including the size and data. In *Fountain*, it inlcudes X-coordinate, Y-coordinate, the pressure andt etc.

Back to the encoding, the third field means the size of argument data. And we use the fourth field for distinguish if the command needs a synchronization, i.e., distinguish

between *send()* and *sendSync()*. The last two fields handle the information of the structure.

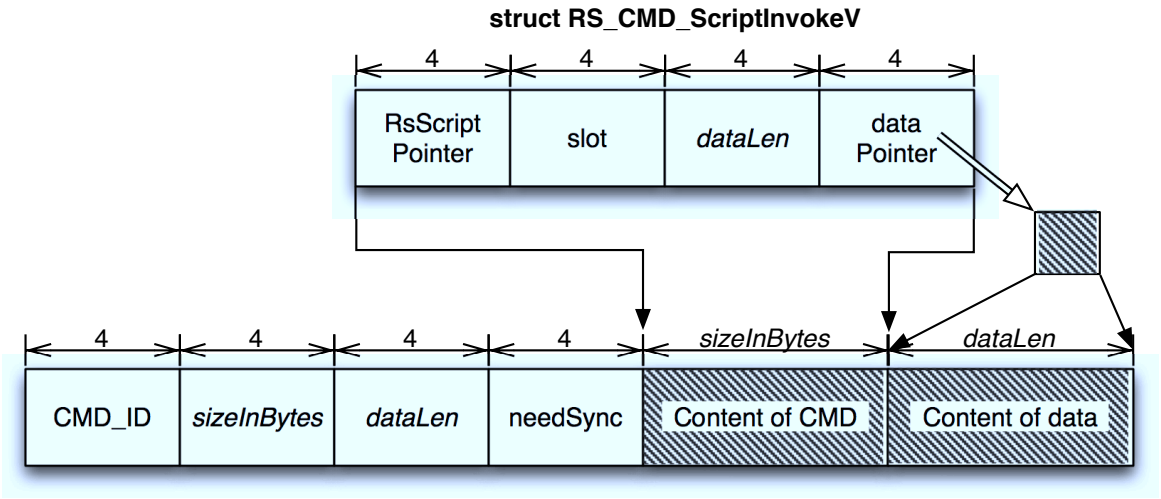

Figure 5.1: RS_CMD_ScriptInvokeV encoding

5.2 Transport Layer

To receive commands, we choose socket to build *Transport Layer* for the the following reasons: (1) We need a lower-level implementation in *Native Layer*; (2) Bi-directional communication; and (3) Better performance of server side.

After the creation of RS context thread, we dynamically create the RS listening thread that depends on Android property. The following code shows our implementation.

```
1  if (props.mRemoteServer) {
2      pthread_t listeningThreadID;
3      LOGV("create rsListeningThread");
4      status = mIO.mToCore.receive(&listeningThreadID, &threadAttr, this);
5      if (status) {
6          LOGE("Failed to start rsListeningThread.");
7          return false;
8      }
9  } else if (props.mRemoteClient){
10     mIO.mToCore.initSocket();
11 }
```

Server or Client? The library of REMOTE RENDERSCRIPT should support not only the client but also the server. A library for server and the other for client doesn't make sense if we want to accomplish the feature of two-way communication. For the reason, we stores the information through Android property system. For example, REMOTE RENDERSCRIPT gets the property from `props.mRemoteServer` and creates the RS listening thread if you type this in the ADB:

```
adb shell setprop remote.rs.server 1
```

The idea of REMOTE RENDERSCRIPT is straightforward, but it's still some issues we should take care. The following four subsection describes the challenge we encounter when extending RENDERSCRIPT to REMOTE RENDERSCRIPT.

5.3 Replicated Command

The key idea of REMOTE RENDERSCRIPT is replaying the commands on the remote engine. Initially, any kind of *commit()* and *commitSync()* commands are followed by *send()* and *sendSync()* respectively.[1] As a result, we found that it caused a error due to those replicated executed commands. For instance, binding the fragment twice might lead to chaos. The commands used for initialization should not be executed on the individual engine. In *Fountain*, they are listed below in order:

`RS_CMD_ID_ContextSetSurface`

`RS_CMD_ID_ProgramFragmentCreate`

`RS_CMD_ID_ContextBindProgramFragment`

`RS_CMD_ID_ElementCreate`

`RS_CMD_ID_ElementCreate2`

`RS_CMD_ID_MeshCreate`

`RS_CMD_ID_MeshBindIndex`

[1] Actually, we use sendCMD() instead of send() to avoid the confusion with socket send().

```
RS_CMD_ID_MeshBindVertex
RS_CMD_ID_MeshInitVertexAttribs
RS_CMD_ID_ScriptCCreate
RS_CMD_ID_ScriptSetVarObj
RS_CMD_ID_ScriptBindAllocation
RS_CMD_ID_ContextBindRootScript
```

5.4 Command Synchornization

As we mentioned in in prior chapter, a part of commands are put into the command queue via *commitSync()*. That means *System Layer* should be blocked until the command is flushed in *Native Layer*. Flushing in RENDERSCRIPT refers to no command in the command queue. We demonstrate the design in Figure 5.2. There are three markers in the figure:

- *mGet* points to the memory address where the next command will be put.
- *mPut* points to the memory address where the next command will be popped.
- *mEnd* points to the memory address of end of the queue.

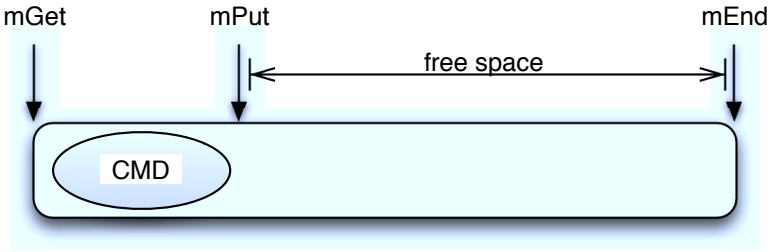

Figure 5.2: Lockless FIFO command queue

Therefore, the command queue is flushed **if** *mGet* **is equal to** *mPut*.

Note that we have a *sendSync()* for REMOTE RENDERSCRIPT, the difference is the client (which sends the command) should be blocked instead of the server.

5.5 Allocation of Command Queue

Another stuff we should take care of the command queue is the allocation. In RENDERSCRIPT implementation, either *mToCore* or *mToClient* are represented as a pointer and passed through functions by pointer. If we still pass the pointer to remote engine, it might cause a segamentation fault. As a result, we should remove it and get the correct *mToCore* pointer.

5.6 Handle the Pointer

Like the allocation of command queue, almost variable are refered by pointer. So we should take care of it. For example, in *Fountain*, we did that as the below:

```
1  static inline void rsHCAPI_ScriptInvokeV (RsContext rsc, RsScript va, uint32_t ↩
       slot, const void * data, uint32_t sizeBytes) {
2      ThreadIO *io = &((Context *)rsc)->mIO;
3      uint32_t size = sizeof(RS_CMD_ScriptInvokeV);
4      if (sizeBytes < DATA_SYNC_SIZE) {
5          size += (sizeBytes + 3) & ~3;
6      }
7      RS_CMD_ScriptInvokeV *cmd = static_cast<RS_CMD_ScriptInvokeV *>(io->mToCore.↩
           reserve(size));
8      cmd->s = va;
9      cmd->slot = slot;
10     cmd->dataLen = sizeBytes;
11     cmd->data = data;
12     if (sizeBytes < DATA_SYNC_SIZE) {
13         cmd->data = (void *)(cmd+1);
14         memcpy(cmd+1, data, sizeBytes);
15         io->mToCore.commit(RS_CMD_ID_ScriptInvokeV, size);
16         if (&((Context *)rsc)->props.mRemoteClient)
17             io->mToCore.sendCMD(RS_CMD_ID_ScriptInvokeV, size, cmd, cmd->dataLen, ↩
                   0);
18     } else {
19         io->mToCore.commitSync(RS_CMD_ID_ScriptInvokeV, size);
20         io->mToCore.sendSync(RS_CMD_ID_ScriptInvokeV, size);
21     }
22
23 }
```

Chapter 6

Conclusions and Future Work

After the demonstration of *Fountain* in previous chapter, we know that the proposal of REMOTE RENDERSCRIPT system is feasible and has a great extensibility. We summarize the advantages of REMOTE RENDERSCRIPT as follows:

1. **Leveraging the hardwares on the remote engine** —Because RENDERSCRIPT could do the computation and graphics operation, it's better to run on the more powerful hardwares. Unlike the other screen sharing mechanism, we fully exploit the power of the remote engine and relieve the network traffic. Only the RS command is necessary to send.

2. **No migration overhead** —The existing RS applications have almost no code necessary to be added or modified for migrating to REMOTE RENDERSCRIPT. For instance, we migrate *Fountain* without any overhead. Again, we are target for RS debugging framework, so the feature is important for REMOTE RENDERSCRIPT.

3. **Debug for** RENDERSCRIPT — REMOTE RENDERSCRIPT speeds up the debugging progress no matter of using emulator or device. By punching through via REMOTE RENDERSCRIPT and set up the environment, RS developer could have a better user experience on REMOTE RENDERSCRIPT rather than the emulator.

It is just a REMOTE RENDERSCRIPT prototype, there are still many stuff that we need work harder to perfect it in the future:

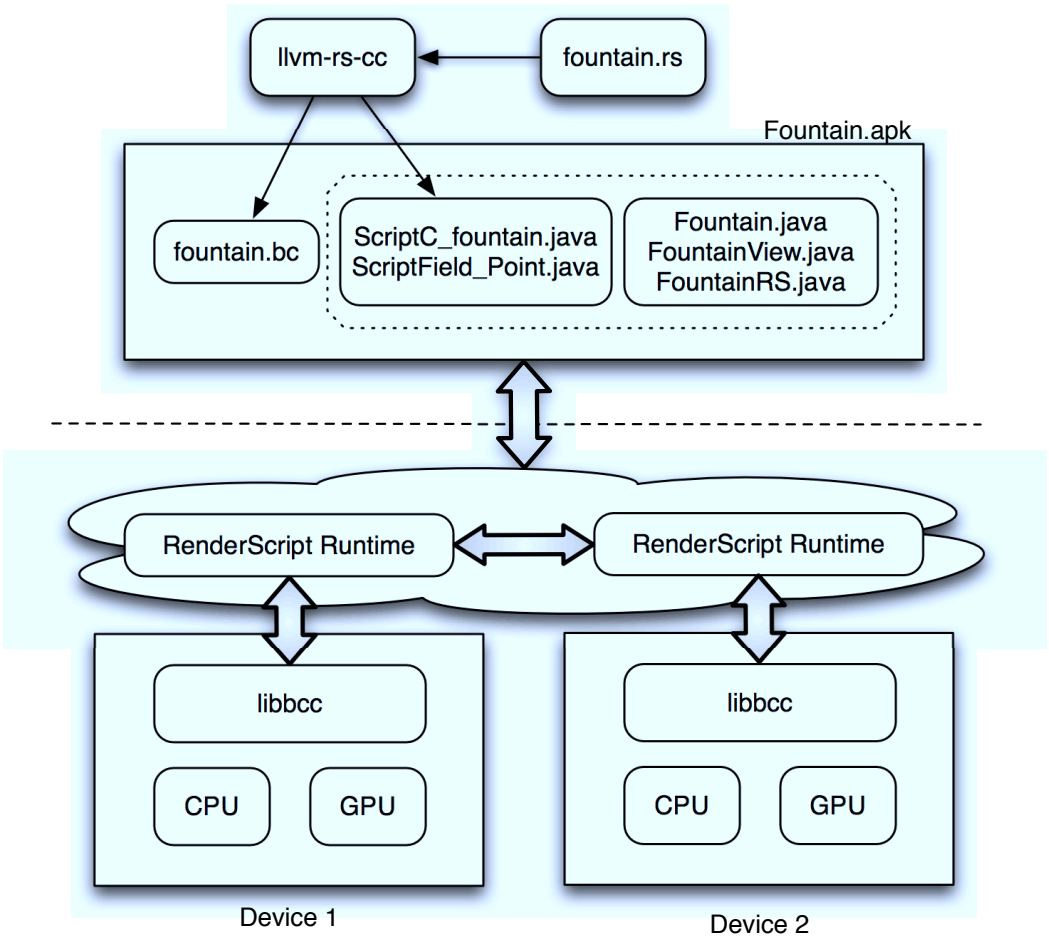

Figure 6.1: The big picture of Remote RenderScript

- **Heterogeneity among Devices** — In the thesis, we just test *Fountain* on the same devices. What if we communicate between two different devices? The size of screen might differ largely, e.g., smartphone and TV. Also, it must need a synchonization of screen size. We compute in individual devices, so we could scale the graphs up losslessly. The other easier approach is simply enlarging the window, but the resolution might be not satisfying.

- **Bi-directional Communication** — One-way synchronization is not enough for non-debugging purposes. Our implementation is one-way by now, that is, one for sending and the other one for receiving only. In the future, we expect to extend it to have the ability of a two-way and one-to-many communication among devices. (As shown in Figure 6.2 (a), and (b) respectively.) In other words, each device could be

not only a client but also a server to facilitate our desirable personal-cloud.

- **Application for Cloud Computing** — As we said in Chapter 1, gaming might are the main purpose of RENDERSCRIPT. Of course, REMOTE RENDERSCRIPT is suitable for game development. For instance, everyone might have one smartphone in the recent future. Game vendor could use REMOTE RENDERSCRIPT to develop a multi-player game instread of tranditional board games, and replace the board with a larger screen device like TV. Our vision is similar with Google Project Tungsten.

- **Security** — Network pairing must be done to set up the environment for REMOTE RENDERSCRIPT. For security concern, auto-pairing is not permitted. So the authentication is necessary in the future network-pairing process to eliminate security concern.

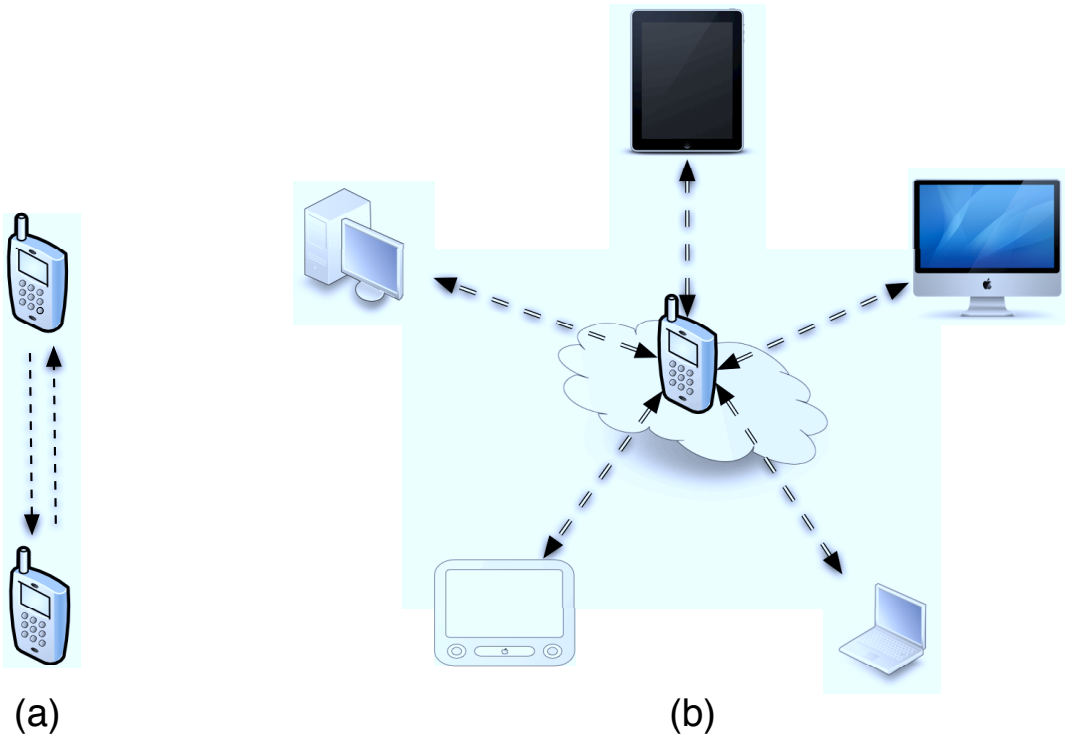

Figure 6.2: Communication among hetrogeneous devices: (a) Bi-Direction (b) One-to-Many

Bibliography

[1] Renderscript package summary http:/Developer//developer.android.com/reference/android/renderscript/package-summary.html.

[2] Competitive Landscape: Mobile Devices, Worldwide, 2Q10. http://www.gartner.com/it/page.jsp?id=1421013, August 2010.

[3] Distimo Mobile World Congress 2010 Presentation - Mobile Application Stores State of Play. http://www.slideshare.net/distimo/distimo-mobile-world-congress-2010-presentation-mobile-application-s from=ss_embed, January 2010.

[4] Android Market attracts game developers, seeks steep growth. http://www.androidapps.com/games/articles/7682-android-market-attracts-game-developers-seeks-steep-growth, April 2011.

[5] Distimo Custom Report. http://report.distimo.com/files/2011/04/Distimo-Custom-Report-Example-April-2011.xlsx, April 2011.

[6] RenderScript native API specification http://code.google.com/p/renderscript-examples/wiki/rs_math, July 2011.

[7] R. J. Sams. Introducing Renderscript. http://android-developers.blogspot.com/2011/02/introducing-renderscript.html, February 2011.

[8] R. J. Sams. Renderscript Part 2. http://android-developers.blogspot.com/2011/03/renderscript.html, March 2011.