國立臺灣大學電機資訊學院資訊工程學系

碩士論文

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

Master Thesis

Remote Renderscript: 基於Android ICS框架
Remote Renderscript: Based On Android ICS Framework

徐偉期
Hsu Wei-Chi

指導教授：廖世偉 博士
Advisor: Liao Shih-Wei, Ph.D.

中華民國 101 年 7 月
July, 2012

致謝

經過幾個多月的努力，終於完成了這篇碩士論文。

首先，我要感謝我的指導教授廖世偉博士。雖然平常因為工作的繁忙而時常無法當面給予指導，但透過線上討論和信件的往來還是令我從中學習很多。老師常常告訴我們一個觀念，就是凡事親自動手做，在做的過程中學習，而不要流於紙上談兵。

再來要感謝楊佳玲教授、梁伯嵩委員、陳呈瑋委員、陳官辰委員四位口試委員在口試時對論文的指點，得以讓論文的完成度更加提升。另外要特別感謝曜瑋學長和證諺同學給予的啟發和指點，在論文完成過程中受到很大的幫助。還有感謝一些實驗室的同學和學弟們，如德育、logan、nowar等等，能讓我在二年碩士生活可以不斷向他們學習，也讓實驗室生活多采多姿。

最後感謝我的家人在求學路上對我的支持。未來希望在資訊工程領域中不枉所學，對社會有貢獻。

中文摘要

螢幕分享系統通常會受到網路頻寬限制而影響效能。 如果我們可以找到一個減少傳輸資料量的新方法，效能將可以顯著地提升。 而新的解決辦法就是Android的Renderscript架構。 Android在2011年11月的時後釋出了ICS的原始碼，其中包含了使用FifoSocket做為進程間溝通的Renderscript資料庫。 我們將它做了一些架構上的延伸以及使用網際網路套接字來達到螢幕分享的功能。

本篇論文實作了以下三個部份：

1. 我們使用Renderscript實做了一個遠端分享螢幕的系統。
2. 我們將FifoSocket版本的Renderscript資料庫成功移植回Android ICS。
3. 我們能從本地端初始化遠端機器上的程式。

而這將帶來四大好處：

1. 我們減少了資料的傳輸量，從點陣圖等級降低成指令等級。
2. 我們利用了遠端機器上的硬體提昇運算效能。
3. 我們制定了格式，可以支援所有現存的Renderscript應用程式。
4. 我們可以利用遠端初始化去操作任何使用Android系統的裝置。

關鍵字： 螢幕分享系統、Android Renderscript、進程間溝通、網際網路套接字

Abstract

The efficiency of screen sharing systems often limited by the network bandwidth. If we could find a new solution to reduce the amount of the transport data, the performance may improve significantly. The new solution is Android Renderscript framework.

Android released ICS source code in November 2011, including the Renderscript library using FifoSocket for Inter-process communication. We do some extensions and use Internet socket to reach a screen sharing system.

In this thesis, we implement the following three parts. First, we implement a screen sharing system by Android Renderscript framework. Second, we apply Renderscript library using FifoSocket to Android ICS. Last, we can initialize the remote device from local device. These works bring us four benefits:

1. We reduce the amount of the transport data from bitmap level to command level.
2. We take advantage of computing on the remote device.
3. We formulate a specific format for original Renderscript applications.
4. We could control any remote devices with Android system by Remote Initialization.

key words: Screen sharing system, Android Renderscript, Inter-process communication, Internet socket

Contents

口試委員會審定書 i

致謝 ii

中文摘要 iii

Abstract iv

1 Introduction 1

2 Related Work 3
- 2.1 VNC : A Graphical Sharing System . . . 3
- 2.2 Remote Renderscript in Android Gingerbread . . . 4

3 Renderscript Overview 5
- 3.1 Renderscript Overview . . . 5
- 3.2 Renderscript Library . . . 6
  - 3.2.1 rsContext . . . 6
  - 3.2.2 rsLocklessFifo . . . 7
  - 3.2.3 rsThreadIO . . . 7
- 3.3 Renderscript Command . . . 8
- 3.4 Renderscript Application : RsHelloWorld and RsFountain . . . 8
- 3.5 Compiler Toolchain . . . 9

4 Apply New Renderscript Library to Android ICS 10
- 4.1 Replace LocklessFifo by FifoSocket . . . 10
- 4.2 The Send Buffer . . . 11

5 Remote Renderscript 13
- 5.1 Remote Renderscript in Android ICS . . . 13
- 5.2 Remote Initialization . . . 14

6 Implementation 16
- 6.1 Back-porting to Android ICS . . . 16
- 6.2 Transport Layer . . . 16
- 6.3 Socket Send and Receive . . . 17
- 6.4 Renderscript Runtime API . . . 19
- 6.5 Blocking Problem . . . 21
- 6.6 Remote Initialization . . . 21

7 Conclusions and Future Work **24**

Bibliography **26**

List of Figures

1.1 Renderscript thread model 2

2.1 How VNC works 3

3.1 The generation of *libRS.so* 6
3.2 Renderscript framework in Android ICS 7

4.1 Renderscript framework in Android Jelly Bean 11
4.2 Renderscript command buffer encoding 12

5.1 Remote Renderscript using FifoSocket in Android ICS 14
5.2 Remote Initialization design 15

6.1 ScriptSetVarI buffer encoding 20
6.2 ScriptInvokeV buffer encoding 21

List of Tables

Chapter 1

Introduction

As the smart phone devices are popular in recent years, more and more applications are developed for the devices, and one of the important technology is screen sharing.[1] The traditional way is to send the framebuffer from the local side to the remote side, and the most famous system is VNC. However, the way VNC protocol does demands on a lot of bandwidth, and also a lot of communication overheads. But in today's, the growth of network speed is far less then the growth of hardware. Even more is the fact that some devices are using not only CPUs but also GPUs to enhance the parallel computing tasks. As a result, network bandwidth will be the bottleneck on the screen sharing topics. We have to solve this problems. Fortunately, Android Renderscript framework can help us.

Android released Renderscript framework in 3.0 Honeycomb, and improved the architecture in the latest Android 4.0 Ice Cream Sandwich (AKA ICS). Nowadays, Renderscript has become a well-developed technology. Renderscript is a new option for computing and rendering, providing a new API (Application Programming Interface) for Android developers. The three main advantages of Renderscript are portability, performance and usability. In this thesis, we are focus on the rendering part.

Renderscript is invoked by JAVA, using Android SDK, and will create a context thread at initial time. The context thread is the thread that helps the JAVA main thread. Each Renderscript application will certainly creates a context thread. The two threads are communicate with Renderscript commands. When JAVA main thread detects a behavior, such as touch on screen, it will send the command to the context thread. The commands may include (X,Y) coordinate and pressure, etc. After sending commands, context thread will decoding the commands and replay it on the graphic view, such as display a light point.

According to the structure, we can make the assumption: If we can launch JAVA main thread on local device, and create a context thread on remote device, that will be so cool. We can imagine that, when we do some behavior on local device, and send Renderscript commands to remote device by Internet socket; simultaneously, we create a context thread

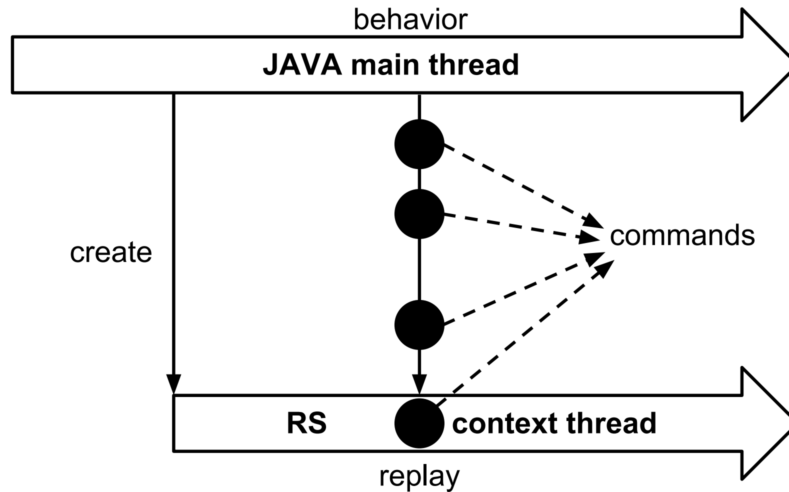


Figure 1.1: Renderscript thread model

on remote device to fetch commands and execute them. Therefore, we get the ability to control the screen of remote device from local device, and this is the screen sharing we want in the beginning. Above all things, the data we send is not the large framebuffer but just Renderscript commands. We reduce the amount of transmitted data from bitmap level to command level. In the meantime, we can take advantage of remote hardware if needed.

The rest of the thesis is organized as follows. Chapter 2 talks about some related works. Chapter 3 introduces the whole Renderscript we should know in details. Chapter 4 mentions the new version of Renderscript library compared with the old version. Chapter 5 explains how to design Remote Renderscript using new version of Renderscript library in Android ICS. Chapter 6 shows implementations in details. Chapter 7 presents conclusions and future works.

Chapter 2

Related Work

2.1 VNC : A Graphical Sharing System

When it comes to talking about screen sharing system, we have to mention VNC. The full name of VNC is Virtual Network Computing, and is the most basic graphical desktop sharing system. An original VNC system consists of a server end, a client end and an RFB (Remote FrameBuffer) protocol.[3] The principle of the RFB protocol is very simple. When server end and client end connect together through the Internet, the client can control the server, and the server sends small rectangles of the framebuffer to the client. Thus it can reach screen sharing.

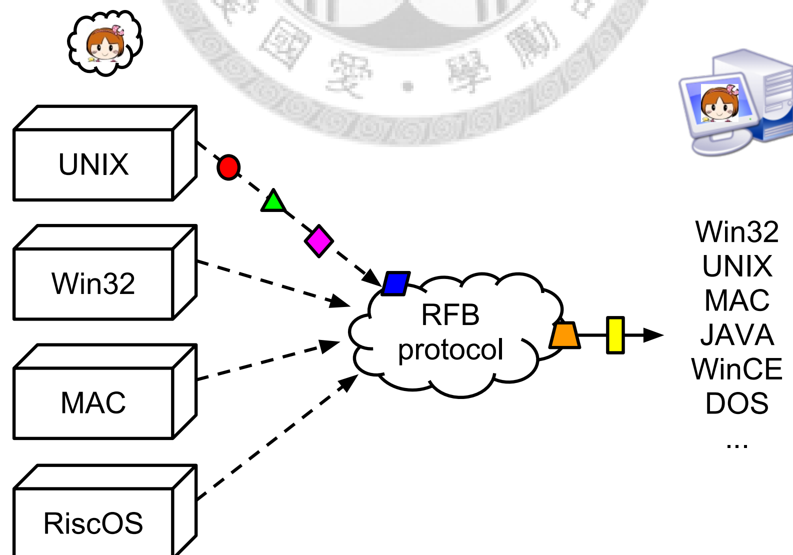


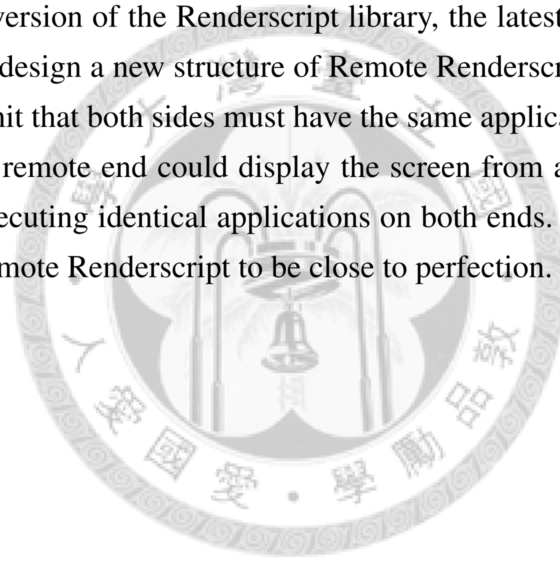
Figure 2.1: How VNC works

However, what seems to be the problem is that the data of the framebuffer is huge and cause communication overheads. Hence, several optimizations have been presented to improve performance of VNC, such as various framebuffer encoding ways. RealVNC

is the most common VNC software, and is popular beacuse it is free and platform-independent.[2] TightVNC uses efficient tight encoding with optional JPEG compression to improve performance over low bandwidth connections.[4] UltraVNC adds an encryption plugin to secure the client and server connection, and also supports file transfer.[5]

2.2 Remote Renderscript in Android Gingerbread

The premier version of Remote Renderscript is published by Yao-Wei in May 2011.[6] At that time, his approach is porting the Renderscript library in Android Honeycomb back to the Android Gingerbread. Using Google Nexus S as testing devices, Fountain applications can simultaneously display the same views on two side of the screen successfully. However, in the past year, we have found that there are many parts to improve. In comparison with the old version of the Renderscript library, the latest version has changed a lot, and we have to redesign a new structure of Remote Renderscript. On the other hand, we must break the limit that both sides must have the same applications. What we expect to achieve is that the remote end could display the screen from any Renderscript applications, instead of executing identical applications on both ends. Through the following works, we believe Remote Renderscript to be close to perfection.



Chapter 3

Renderscript Overview

3.1 Renderscript Overview

Now let us introduce the whole Renderscript framework.

In Android ICS, the related codes are in the following directories.

JAVA SDK : {Android-src}/frameworks/base/graphics/java/android/renderscript/

These JAVA codes are connect with Android SDK (Software Development Kit). When Android developers use Android SDK to start a Renderscript application, JAVA codes will be called at initial time and create surface view or something else.

Renderscript library : {Android-src}/frameworks/base/libs/rs/

This part is Renderscript runtime library, which are written by C++. These CPP files are core codes for Renderscript runtime. Compiler compiled the whole library into *libRS.so* for target device, and they are the shared library that Renderscript applications can use directly when running. As we know from the above, the key modification will be in this part. We will tell more in the next section.

JNI : {Android-src}/frameworks/base/graphics/jni/

Because of using C++ library from JAVA, we need JNI (JAVA Native Interface) as a bridge to help us. The directory contains a corresponding table between JAVA functions and C++ functions. As long as JAVA side has to make use of Renderscript library, it can just easily check the table.

After compiling the Renderscript and JNI libraries below, some files are generated automatically. To take Google Nexus S (AKA crespo) for example, we will generate the following two directories.

Shared libraries : {Android-src}/out/target/product/crespo/system/lib/

The directory places the Android system shared libraries. The two libraries are called *libRS.so* and *librs_jni.so*. We must push these two libraries into the crespo device.

Runtime API : {Android-src}/out/target/product/crespo/obj/
SHARED_LIBRARIES/libRS_intermediates/

The directory places intermediate files of Renderscript library. The sequence that GCC (GNU Compiler Collection) compiles Renderscript library is as follows. First, *rsg_generator.c* is compiled and executed, and generates some files such as *rsgApi.cpp* and *rsgApiStructs.h*. These files are API functions and structures for runtime. Then, all the files including original Renderscript library files are compiled together into *libRS.so*.

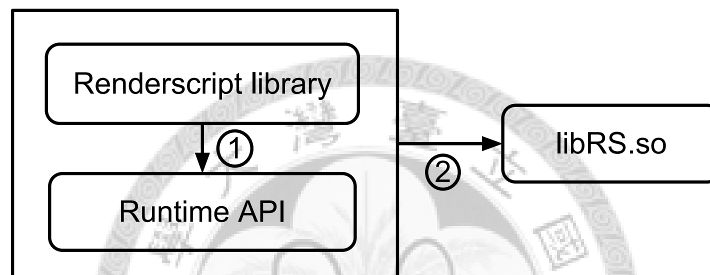


Figure 3.1: The generation of *libRS.so*

3.2 Renderscript Library

In accordance with previous section, we know that Renderscript library is for runtime. From now on, it is abbreviated to a new word "libRS". In libRS, there are three most important files (or classes) that establish the Renderscript overall framework. They are *rsContext.cpp*, *rsFifoSocket.cpp* and *rsThreadIO.cpp*. What we must emphasize is the latest AOSP (Android Open Source Project) has not released rsFifoSocket[1]. Instead, the libRS version is using rsLocklessFifo. Thus, we introduce rsLocklessFifo here.

3.2.1 rsContext

The file is the entry point from JAVA to libRS. When the function *rsContextCreate()* is called by JAVA at initial time, via JNI, rsContext starts to execute. Then a thread is created at once, we named it as "context thread". Since then both the JAVA main thread and the context thread exist at the same time until process terminated.

[1]till June 27th, 2012

3.2.2 rsLocklessFifo

Now that there are two threads in the process simultaneously, we have to deal with IPC (Inter-Process Communication) problems. In Android Honeycomb and Ice Cream Sandwich, FIFO queue is used for bridge of communication. Each thread can access the FIFO queue, hence they can talk with each other. In order to avoid ambiguous, generally, a Renderscript process creates two FIFO queues. One of the queue is written by JAVA main thread and read by context thread, and the other one vice versa.

3.2.3 rsThreadIO

Just as the name implies, the class handles thread I/O, access of FIFO queue. The main method is *coreCommit()*. "Commit" is an action that push data into FIFO queue. As a matter of fact, these data are Renderscript commands. In generally, *coreCommit()* is called by JAVA main thread, and send commands to context thread. It stands to reason that some other rsThreadIO methods affect to FIFO queue, including commit and wait for context thread to response. But in this thesis, we just consider one way rendering, so will not discuss it.

Another main method is *playCoreCommand()*. It is for data, known as Renderscript commands, fetching and executing. In *playCoreCommand()*, there is a while loop keep checking if there are new commands in FIFO queue. If so, then decode the contents of command and execute it. Likewise, *playCoreCommand()* often played by context thread.

According to the aforementioned, now we can illustrate the main Renderscript structure. To sum up, we could create a context thread to help JAVA main thread, we could set up FIFO queue for IPC, and we could access the commands in FIFO queue and then execute them.

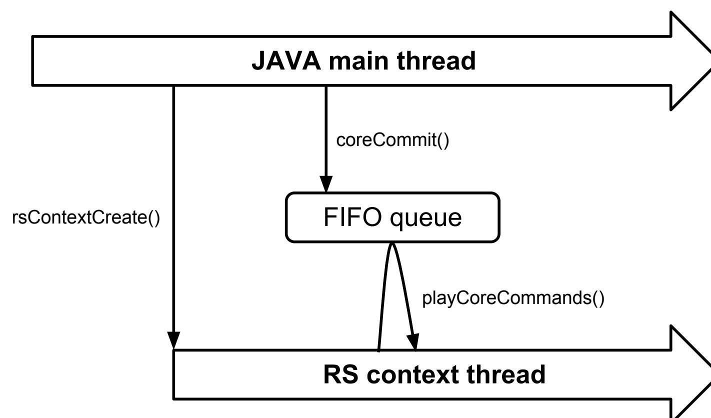


Figure 3.2: Renderscript framework in Android ICS

3.3 Renderscript Command

Actually, Renderscript command refers to Renderscript runtime API. Renderscript defines a set of API, and all the commands are mapping to the corresponding API functions through the table. When we say we play a command, that is to say, to call the low level API function with parameters. These APIs are defined in *rsgApi.cpp*, and are generated automatically in compile time. A complete command is composed of command header and command API structure. In command header, there are two integer variables to store command ID and size of API structure. Command API structure is various which depends on different command, and is defined in *rsgApiStructs.h* file. The variables of the structure is the inputs of API function.

Each API function involves three main steps:

- Step 1. Set the variables of command header.
- Step 2. Assign values to API structure.
- Step 3. Commit the command into local FIFO queue.

3.4 Renderscript Application : RsHelloWorld and RsFountain

Now let us start to recognize what a real Renderscript application looks like. Here we take RsHelloWorld and RsFountain for example, cause both of them are from AOSP standard Renderscript sample applications, and they are graphic samples. Now, HelloWorld is short of RsHelloWorld, and Fountain is short of RsFountain. When HelloWorld is executed, it will show the string "Hello World" on the view where finger prints. When Fountain is executed, it will display a fountain with random color where finger prints, and fall down until disappeared. The two applications directories are:

- {Android-src}/development/samples/RenderScript/HelloWorld/
- {Android-src}/development/samples/RenderScript/Fountain/

Each Renderscript graphic application has three JAVA files and one RS file. To take HelloWorld for example, there will be *HelloWorld.java*, *HelloWorldView.java*, *HelloWorldRS.java* and *helloworld.rs*.

HelloWorld.java : Handle Android application lifecycle. For example, *onCreate()* is called at initial time, creating view or binding data and, etc.

HelloWorldView.java : Create an environment for rendering. Owning to graphic application, we need a view to display the result. Separate actions to the view are also written here.

HelloWorldRS.java : Control the low level libRS. Any usage of libRS entry point is here, such as various initialization and script for runtime.

helloworld.rs : Equivalent to kernel in other script language, which is written by C99 standard[2].

In Fountain, the files are named as *Fountain.java*, *FountainView.java*, *FountainRS.java* and *fountain.rs*.

3.5 Compiler Toolchain

Renderscript applications use slang (AKA llvm-rs-cc) and libbcc for compile tools. Slang is for front-end, and libbcc is for back-end. Now we specifically introduce them.

Slang's former is Clang. Like Clang, it helps to compile RS file to BC file, for example, from *helloworld.rs* to *helloworld.bc*. BC file contains LLVM bitcode, LLVM compiler IR. Slang will also reflect one or more JAVA files, and combine with original Renderscript JAVA files together into an APK (Android Package) file. In HelloWorld, the reflected file is *ScriptC_helloworld.java*; while Fountain defined a Point structure, the reflected files is *ScriptC_fountain.java*, in addition, *ScriptField_Point.java*. Files with prefix ScriptC_* are used to create a new ScriptC object, and we will explain more in section 6-6. Related directories are here.

- {Android-src}/out/target/common/obj/APPS/{App-name}_intermediates/ src/com/example/android/rs/
- {Android-src}/out/target/common/obj/APPS/{App-name}_intermediates/ src/renderscript/res/raw/

Libbcc interprets LLVM bitcode to machine code at runtime. In this manner, it could run on different devices, and that is to say its portability.

[2] an extend version of C language

Chapter 4

Apply New Renderscript Library to Android ICS

Jelly Bean, the next version of Android system, does some significant updates to Renderscript library. In order to continue the usability of present works, there is no doubt that we should based on the new version of libRS framework. In the first place, let us compare the new version to the old one.

4.1 Replace LocklessFifo by FifoSocket

In the new version of libRS, rsLocklessFifo class is no longer exists. Instead, rsFifoSocket class is added in. JAVA main thread and context thread communicate by UNIX domain sockets for IPC. Renderscript calls *socketpair()* function at initial time. *socketpair()* creates a pair of connected sockets and assigns a two-element integer array to two threads. This array is equivalent to UNIX socket ID, like different windows, individual deals with two sides of information. In the same way to avoid ambiguous issue, we create *socketpair()* twice at the same time, which are known as mToCore and mToClient. The direction of mToCore is from JAVA side to context side, and mToClient is vice versa. Currently, *socketpair()* can use just AF_UNIX domain, and not yet to support AF_INET domain.

One of the benefits of using *socketpair()* is that there are several ready-made functions to use. On the contrary, FIFO has to do some redundant works such as develop format and others. Another benefit is scalability. As long as *socketpair()* supports Internet sockets, we can easily implement Remote Renderscript without modify too many codes. Regretful to say, now we have to create another Internet socket for Remote Renderscript. We will mention it later.

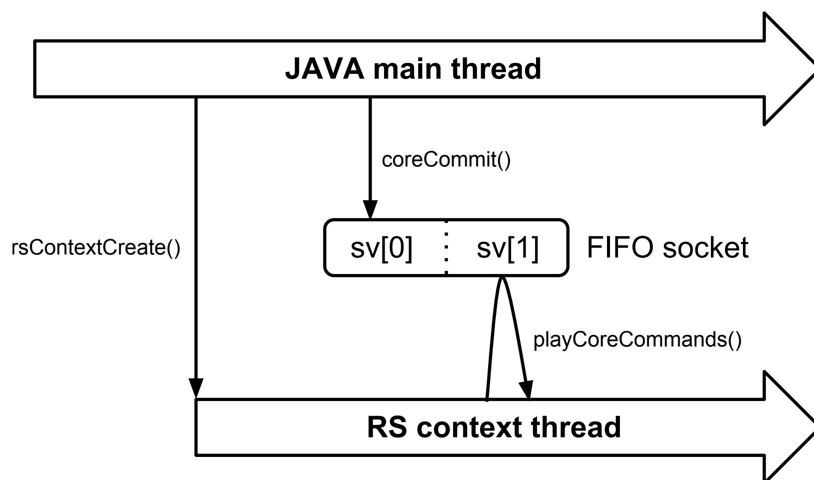


Figure 4.1: Renderscript framework in Android Jelly Bean

4.2 The Send Buffer

Because socket has a feature of being transmitted in buffer units, we have to place all the data together with a specific sequence, then send it at once. In the new version of libRS, it provides unsigned character array "mSendBuffer" for the send buffer. The mSendBuffer is encoded by CoreCmdHeader structure and command API structure.

CoreCmdHeader structure : Includes two integer variables named cmdID and bytes, and are unique header information for each command. "cmdID" stands for command identification, and the range of the number is between 1 and 65. "bytes" contains the size of the command API structure and the size of the data that pointer variables in API structure points to. CoreCmdHeader structure is used in the *coreHeader()* method in ThreadIO class. The receiving end could recognize which kinds of API structure and size it will receive by header information. For example, to check the API structure table and integrity, and the buffer size that is required.

Command API structure : Begin with RS_CMD_*. The number of the variables demands on different commands. What we must emphasize is that the structure initial address has to be the extremity of CoreCmdHeader structure. In this way, we could commit a continuous memory space. All these things will be done in the Renderscript command API. Furthermore, some of the structure variables are pointers, and points to data in another memory space. Similarity, we copy the data to the end of command API structure.

In summary, a complete Renderscript command buffer is encoded as the figure.

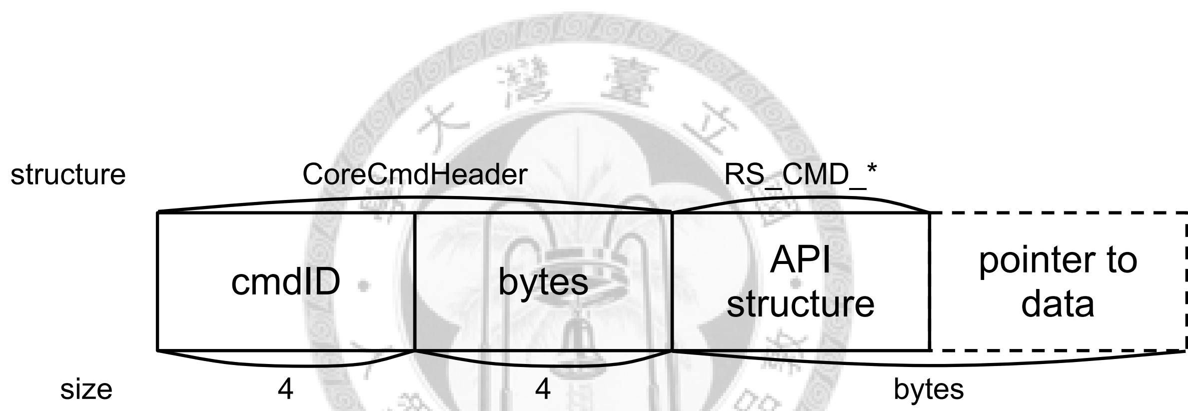


Figure 4.2: Renderscript command buffer encoding

Chapter 5

Remote Renderscript

Now we understand the integral Renderscript framework. So, what works should we do for Remote Renderscript? The contributions of this thesis are the following three points:

1. Back porting FifoSocket libRS to Android ICS.

2. Implement Remote Renderscript in Android ICS.

3. Remote Initialization.

We select HelloWorld and Fountain to be the testing applications. As mentioned earlier, these applications are standard graphic examples in AOSP. Our target device is Google Nexus S (AKA crespo). To results, Remote Renderscript works well in both applications; moreover, Remote Initialization is feasible in HelloWorld, and will support Fountain in the future.

5.1 Remote Renderscript in Android ICS

Reference to Yao-Wei's work, we design the structure for the new Renderscript as figure below.

What we must do is:

1. Get ready Renderscript on both local and remote devices.

2. Send the commands from local device to remote one.

3. Receive commands on remote device and execute them.

The main aim of the Remote Renderscript design is for screen sharing, that is, to display the same screen on different devices. Thus there should be libRS in both devices

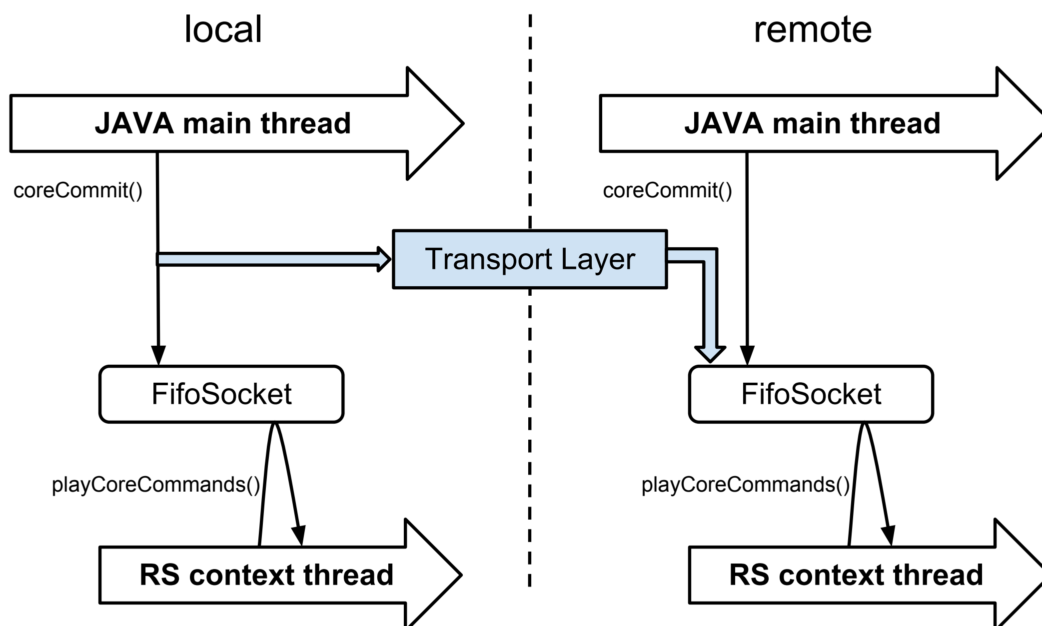


Figure 5.1: Remote Renderscript using FifoSocket in Android ICS

(for runtime). In addition, only JAVA can drive libRS, so we need JAVA applications to drive it, and JNI is necessary certainly. When we prepare the same JAVA application on two devices, before start up, we have to decide whether it is sender or receiver. The former is called server, and the latter is called client. The chief reason why it is named is due to the things it does at application initial time. For instance, server side will execute *listen()* function and wait for *connect()* function on the client side. So in implementation, local side is equals to server, and remote side is equals to client.

Next step, we have to deal with the issue of sending commands. Because in HelloWorld and Fountain runtime, JAVA commits commands into FifoSocket is single command repeatedly (ScriptSetVarI and ScriptInvokeV). In this matter, every time we commit, we can send the commands to the client by Internet sockets at the same time.

Last but no least, client has to receive these commands. As long as they are decoded successfully, client will also commit into its local FifoSocket. Thereafter, context thread in remote device will fetch the commands and show us the screen.

However, in the new version of libRS, we have some difficulties to overcome. Chapter 6 illustrates the solutions and detail implementations.

5.2 Remote Initialization

If Remote Renderscript using FifoSocket in Android ICS is feasible, and it is feasible indeed, what could we do more to improve it? Here we put forward hypothesis. In Renderscript framework, not only handles commands receiving and fetching in runtime, every

initial setup is completed through commands. When Renderscript applications start-up and create a context thread, we use some commands that are specialized for initialization to setup. Meanwhile, we send these commands to remote device for its initial information. In this way, remote device does not need to spontaneously execute initial commands, instead, it can just wait for local device. Figure below shows the modified architecture. Experimental results shows that it is feasible in HelloWorld. We can receive initial messages in remote device which send from local device, and decode and then execute them.

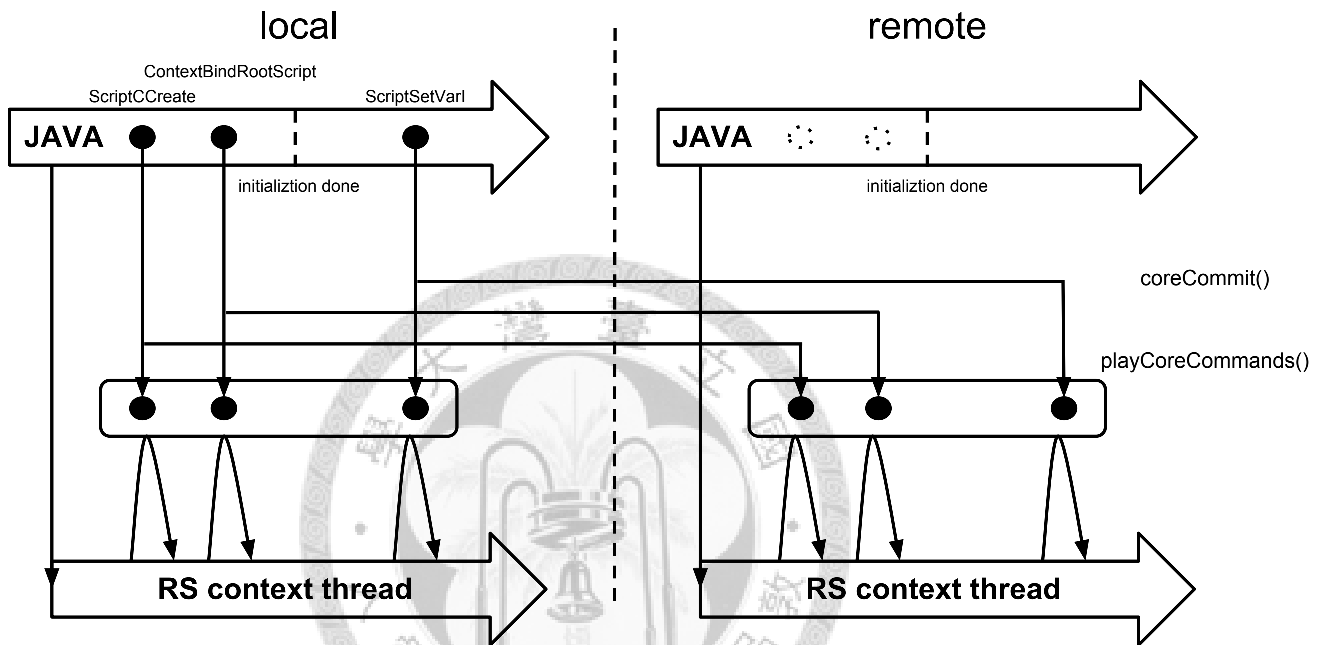


Figure 5.2: Remote Initialization design

Remote Initialization brought us many benefits. In remote device, we do not need LLVM bitcode anymore, instead, all of them is obtained from Internet sockets. Therefore, it saves many spaces.

Chapter 6

Implementation

6.1 Back-porting to Android ICS

Porting the new version of libRS is not just substitute the original folder and do "mm" command (Android build tool) to build a shared library and then push into smart device. Frankly speaking, even "mm" will cause problems. Owing to brand new classes, we have to keep those are needed in HelloWorld and Fountain, and comment others.

The long and the short of it, we do the following steps.

1. Comment some codes in libRS to make it compiled successfully. In this way, HelloWorld works well but Fountain crashes. The reason is that one of JNI native method called nProgramFragmentCreate does not update yet.

2. We need the new version of JNI to compile to *librs_jni.so*. Before that, we have to compile another new library called androidfw to *androidfw.so*.

3. After pushing the three shared libraries, we cannot boot crespo yet. The problem is because of parts of JNINativeMethod. We must comment unrecognized methods and modify the parameters.

4. Up to this point, crespo device is bootable, but Fountain still crash. The solution is apply old version of nProgramFragmentCreate function and assign needed parameters. From now on, Fountain works well in crespo.

6.2 Transport Layer

Because of its easy to use, we select UNIX socket API for IPC. The way we do is to create a new thread to handle the socket tasks. The best time to create socket thread is after creating the context thread, since we can guarantee it is produced with Renderscript

application. We use Android properties to distinguish whether it is server socket thread or client socket thread. Providing commands like “adb shell setprop rs.remote.server true”, we can set the properties directly.

6.3 Socket Send and Receive

After creating server and client threads by adb setprop, we have to implement socket *send()* and *recv()*. *Send()* is inappropriate to be written in server thread, because our approach is send the command after each commitment. In other words, it should be written in Renderscript runtime API. In contrast, *recv()* must to be implemented in client thread. We use a while loop to continuously waits commands from the server side, and decodes and executes them as soon as receive some, and then waits for another one.

The related code is as follows.

(Code: sendCMD, while loop in clientProc)

```
bool FifoSocket::sendCMD(uint32_t cmdID, size_t dataLen, void *
   cmd, size_t data_length) {
    const int ARG_SIZE = sizeof(uint32_t) + sizeof(size_t) +
       dataLen + data_length;
    ALOGE("ARG_SIZE:␣%d", ARG_SIZE);
    char buffer[ARG_SIZE];
    bzero(buffer, ARG_SIZE);

    //buffer encoding : ScriptSetVarI [cmdID][dataLen][*cmd]
    // ScriptInvokeV [cmdID][dataLen][*cmd][*(cmd+1)]
    reinterpret_cast<uint32_t *>(&buffer)[0] = cmdID;
    reinterpret_cast<size_t *>(&buffer)[1] = dataLen;
    memcpy(buffer + sizeof(uint32_t) + sizeof(size_t), cmd,
       dataLen + data_length);

    //send buffer
    int n = send(sockfd, buffer, ARG_SIZE, 0);

    return true;
}

while (1) {
    //receive coreHeader from server : [cmdID][dataLen]
    const int TOP_SIZE = sizeof(uint32_t) + sizeof(size_t);
    char buffer_top[TOP_SIZE];
```

```
bzero(buffer_top, TOP_SIZE);

int n = recv(toCore->sockfd, buffer_top, TOP_SIZE, 0);
uint32_t cmdID = reinterpret_cast<uint32_t *>(&buffer_top)
    [0];
size_t dataLen = reinterpret_cast<size_t *>(&buffer_top)[1];
ALOGE("cmdID:␣%d,␣dataLen:␣%d", cmdID, dataLen);

if (cmdID == RS_CMD_ID_ScriptSetVarI) {//48
     const int BOT_SIZE = dataLen;
     char buffer_bot[BOT_SIZE];
     bzero(buffer_bot, BOT_SIZE);

     n = recv(toCore->sockfd, buffer_bot, BOT_SIZE, 0);

     RS_CMD_ScriptSetVarI *cmd = (RS_CMD_ScriptSetVarI *)(io
         ->coreHeader(cmdID, dataLen));
     memcpy(cmd, buffer_bot, dataLen);
     ALOGE("cmd->value:␣%d", cmd->value);
     cmd->s = (RsScript *)(io->mToCoreRet);//using local
         RsScript
}

else if (cmdID == RS_CMD_ID_ScriptInvokeV) {//46
     const int BOT_SIZE = dataLen + 20;
     char buffer_bot[BOT_SIZE];
     bzero(buffer_bot, BOT_SIZE);

     n = recv(toCore->sockfd, buffer_bot, BOT_SIZE, 0);

     RS_CMD_ScriptInvokeV *cmd = (RS_CMD_ScriptInvokeV *)(io
         ->coreHeader(cmdID, dataLen + 20));
     memcpy(cmd, buffer_bot, dataLen + 20);
     cmd->s = (RsScript *)(io->mToCoreRet);//using local
         RsScript
}

else {
     ALOGE("clientProc␣receive␣cmdID␣error.");
}
```

```
}
```

6.4 Renderscript Runtime API

To support HelloWorld and Fountain, we recognized that the corresponding Renderscript runtime API are ScriptSetVarI and ScriptInvokeV. They look like this.

(Code: LF_ScriptSetVarI, LF_ScriptInvokeV)

```
static void LF_ScriptSetVarI (RsContext rsc, RsScript s,
   uint32_t slot, int value)
{
    ThreadIO *io = &((Context *)rsc)->mIO;
    const uint32_t size = sizeof(RS_CMD_ScriptSetVarI);
    ALOGE("add␣command␣ScriptSetVarI\n");
    RS_CMD_ScriptSetVarI *cmd = static_cast<RS_CMD_ScriptSetVarI
        *>
        (io->coreHeader(RS_CMD_ID_ScriptSetVarI, size));
    cmd->s = s;
    cmd->slot = slot;
    cmd->value = value;
    io->coreCommit();
};

static void LF_ScriptInvokeV (RsContext rsc, RsScript s,
   uint32_t slot, const void * data, size_t data_length)
{
    ThreadIO *io = &((Context *)rsc)->mIO;
    const uint32_t size = sizeof(RS_CMD_ScriptInvokeV);
    uint32_t dataSize = 0;
    dataSize += data_length;
    ALOGE("add␣command␣ScriptInvokeV\n");
    RS_CMD_ScriptInvokeV *cmd = NULL;
    cmd = static_cast<RS_CMD_ScriptInvokeV *>(io->coreHeader(
       RS_CMD_ID_ScriptInvokeV, dataSize + size));
    uint8_t *payload = (uint8_t *)&cmd[1];
    cmd->s = s;
    cmd->slot = slot;

    memcpy(payload, data, data_length);
    cmd->data = (const void *)(payload - ((uint8_t *)&cmd[1]));
```

```
    payload += data_length;

    cmd->data_length = data_length;
    io->coreCommit();
};
```

Which API structures are defined here.

(Code: struct RS_CMD_ScriptSetVarI, struct RS_CMD_ScriptInvokeV)

```
#define RS_CMD_ID_ScriptSetVarI 48
struct RS_CMD_ScriptSetVarI_rec {
    RsScript s;
    uint32_t slot;
    int value;
};

#define RS_CMD_ID_ScriptInvokeV 46
struct RS_CMD_ScriptInvokeV_rec {
    RsScript s;
    uint32_t slot;
    const void * data;
    size_t data_length;
};
```

In the same way FifoSocket does, we have to put all the data together into a buffer. Lucky to say, in the new version of libRS, most of the API structures with pointer variables copy the data and write into a continuous memory space. ScriptInvokeV is a good example. It contains a pointer variable named ”data”, and points to another memory address with data. All the things we have to do is calculate the command size, including the header, API structure and pointer to data, and then send to remote side. In implementation, we add *coreSend()* function after each *coreCommit()* function. Since Renderscript runtime API is generated automatically, we must modify the *rsg_generator.c* file.

The figures show the encoding buffers of ScriptSetVarI and ScriptInvokeV.

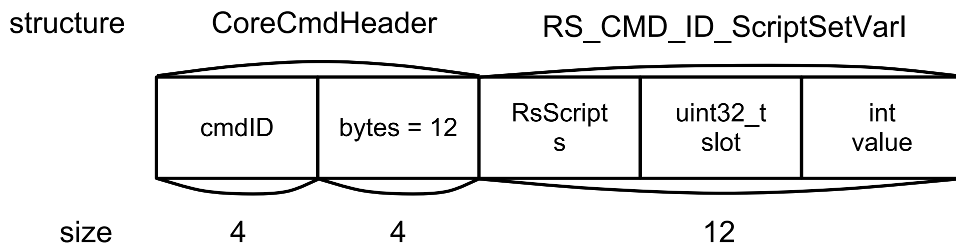


Figure 6.1: ScriptSetVarI buffer encoding

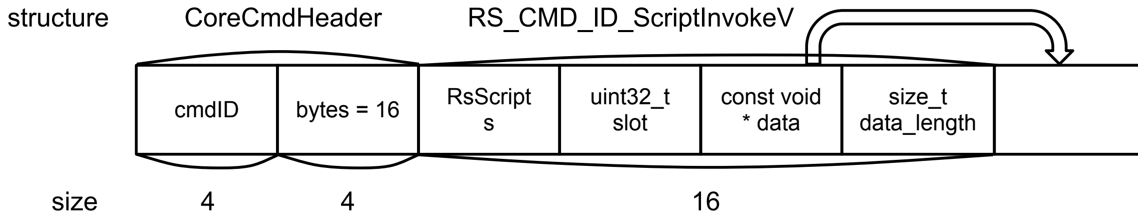


Figure 6.2: ScriptInvokeV buffer encoding

6.5 Blocking Problem

We have found that there is a problem of Fountain view display on crespo with Android ICS using the new version of libRS. This is because, in libRS with FifoSocket, the context thread provides *recv()* function to wait for commands. Whenever no new commands is coming, context thread would continue wait and be blocked. Then, the context thread cannot calculate new coordinates and refresh the surface view. HelloWorld does not cause the problem since it just show the string. As good luck would have it, Android Jelly Bean overcomes the problem. In implementation, it provides "#ifndef ANDROID_RS_SERIALIZE" in *rsContext.cpp* file to slove it.

6.6 Remote Initialization

To implement the Remote Initialization, we have to take care of the APIs called ScriptCCreate and ContextBindRootScript. They look like this.

(Code: LF_ScriptCCreate, LF_ContextBindRootScript)

```
static RsScript LF_ScriptCCreate (RsContext rsc, const char *
   resName, size_t resName_length, const char * cacheDir, size_t
    cacheDir_length, const char * text, size_t text_length)
{
    ThreadIO *io = &((Context *)rsc)->mIO;
    const uint32_t size = sizeof(RS_CMD_ScriptCCreate);
    ALOGE("add␣command␣ScriptCCreate\n");
    RS_CMD_ScriptCCreate *cmd = static_cast<RS_CMD_ScriptCCreate
        *>
        (io->coreHeader(RS_CMD_ID_ScriptCCreate, size));
    cmd->resName = resName;
    cmd->resName_length = resName_length;
    cmd->cacheDir = cacheDir;
    cmd->cacheDir_length = cacheDir_length;
    cmd->text = text;
```

```
    cmd->text_length = text_length;
    io->coreCommit();

    RsScript ret;
    io->coreGetReturn(&ret, sizeof(ret));
    io->mToCoreRet = (intptr_t)ret;
    return ret;
};

static void LF_ContextBindRootScript (RsContext rsc, RsScript
   sampler)
{
    ThreadIO *io = &((Context *)rsc)->mIO;
    const uint32_t size = sizeof(RS_CMD_ContextBindRootScript);
    ALOGE("add␣command␣ContextBindRootScript\n");
    RS_CMD_ContextBindRootScript *cmd = static_cast<
       RS_CMD_ContextBindRootScript *>
        (io->coreHeader(RS_CMD_ID_ContextBindRootScript, size));
    cmd->sampler = sampler;
    io->coreCommit();
};
```

Which API structures are defined here.

(Code: struct RS_CMD_ScriptCCreate, struct RS_CMD_ContextBindRootScript)

```
#define RS_CMD_ID_ScriptCCreate 54
struct RS_CMD_ScriptCCreate_rec {
    const char * resName;
    size_t resName_length;
    const char * cacheDir;
    size_t cacheDir_length;
    const char * text;
    size_t text_length;
};

#define RS_CMD_ID_ContextBindRootScript 15
struct RS_CMD_ContextBindRootScript_rec {
    RsScript sampler;
};
```

Like other API commands for Remote Renderscript, server device sends to client device after every commitments. Fortunately, the parameter that ContextBindRootScript

needs is exactly the return value that ScriptCCreate returns. So, in the client part implementation, we do not need to receive the commands again and again from server; instead, we just catch the return value from ScriptCCreate and then let it to be the input value of ContextBindRootScript. Furthermore, there is a pointer variable called "text" in ScriptCCreate. The pointer points to the LLVM bitcode that compiled from RS file, and will pass to the remote side. As a result, remote device needs not to generate its own bitcode.

As to the JAVA application in remote device, we still have to create a context thread of libRS. After creating the thread, we do nothing but wait for initial commands. It is important to note that because remote device does not have its own ScriptC_* class object, it shows the purely image from local device, and cannot directly control the remote device.

The works we did are implemented successfully in HelloWorld. But in Fountain, although it needs extra commands and complex initial command sequence, we can achieve it in the future.

Chapter 7

Conclusions and Future Work

According to the implementations introduced in Chapter 6, we could successfully implement Remote Renderscript in Android ICS. To sum up, the values created by this research are as follows.

Reduce the communication overhead:

For the amount of data transmitted, Remote Renderscript depends on the size of the command buffer. Size of command are 12 bytes in ScriptSetVarI and 44 bytes in Script-InvokeV, and is far less than the framebuffer size. Although ScriptCCreate command may become complicate because of the huge LLVM bitcode size, it does not matter since it is initial command and would be executed only once. Owning to this fact we can find that if there are huge commands that need to be send continuously, we must keep an eye on them.

Use remote hardware to improve performance:

This is because, the data we send are not the rendering results after computing but just commands before computing. That is to say, we send the command information to the remote device, and then the hardware on the remote device computes according to the script. Since the scripts (or compiled LLVM bitcode) in both side is identical, screen should be substantially the same.

Formulate a specific format for original Renderscript application:

We did not do many modifications but Renderscript library. With the Renderscript applications growing vigorously, we can implement Remote Renderscript as finding out the corresponding command APIs. For Android developers, they do not need to consider whether or not it is remote-able. On the contrary, they can write normal Renderscript applications as usual.

Remote Receiver comes to life:

In the future, we hope to make Remote Initialization realistically. Remote Receiver is such an idea. It is an Renderscript application which exists in remote end. The meaning of the “receiver” is to receive all the various Renderscript initial commands, and transform into initial status for all kinds of applications. To take Fountain for example, besides the original initial commands mentioned before, some other objects have to be created (e.g., ProgramFragment and Mesh). In order to deal with these type of commands, we should additionally create the corresponding objects. While Remote Receiver really comes to life, every Android devices can be our remote ends. The scenario may like this. In the beginning we use smart phone to control the smart TV in living room to watch a movie. After moving into the car, we continuously project the movie screen on the Android monitor in the car, even if there is nothing in the monitor but a Remote Receiver.

Bibliography

[1] Comparison of remote desktop software http://en.wikipedia.org/wiki/Comparison_of_remote_desktop_software.

[2] RealVNC http://www.realvnc.com.

[3] The RFB Protocol http://www.realvnc.com/docs/rfbproto.pdf.

[4] TightVNC http://tightvnc.com/intro.html.

[5] UltraVNC http://www.uvnc.com.

[6] O. Yao-Wei. Remote Renderscript: Leveraging the graphics hardware on the remote engine. Master's thesis, National Taiwan University, 2011.