Comparison and Analysis of the Three Programming Models in Google Android

Xi Qian
Intel Corporation
xi.qian@intel.com

Guangyu Zhu
Intel Corporation
greg.zhu@intel.com

Xiao-Feng Li
Intel Corporation
xiao-feng.li@intel.com

Abstract

Smartphone and tablet are becoming more like personal computer. It is important to understand the pros and cons of different programming models in these kinds of mobile devices. In order to fully understand their implications to the platform architecture and their technical correlation, we develop variants of the same representative Android application in all of the three sets of APIs provided in Google Android, i.e., the Java SDK, C++ NDK, and the new powerful Renderscript. Based on the same application, we conduct detailed apple-to-apple analysis, with focus on hands-on programming convenience, runtime behavior, and technical correlation of the different programming models. We find that the current programming models provided in Android can be improved. We propose a unified solution that we expect to satisfy the requirement of both programmability and performance as a programming model, and also maintain the applications' security and portability when deployed in mobile devices.

Keywords Programming language, programming model, mobile platform, managed runtime

1. Introduction

Smartphone and tablet are becoming more like personal computer. They provide abundant and powerful APIs for the programmers to develop all kinds of attractive applications [1]. For example, Apple iPhone has SDK providing C-class programming language support [2]. Google provides Android SDK supporting Java programming API [3].

Different programming models require different system implementations to support both the application development in host machine, and the application execution in client device. More importantly, different programming models have different implications in programming convenience, execution efficiency, and system requirement. To deliver the best user experience to end users, it is important for the system software designers to understand these implications.

To investigate the different programming models, it is desirable to compare them apple-to-apple with other non-essential factors fixed. Android provides a good environment for such a study, since Android has three programming models in the same system: Java support in SDK [3], C++ support in NDK [4], and Renderscript support [5]. All the three programming models can be used to develop applications of similar functionalities, while the developers could expect some differences in variants of the same application developed in different models.

In order to fully understand the pros and cons of each programming model, their implications to the platform architecture, their technical correlation and trend, we develop variants of the same representative Android application in all of the three APIs. Then we conduct detailed apple-to-apple analysis based on the same application. In our analysis, we find that, each of the three models has its respective advantages and disadvantages. There is not a single model that can satisfy all the requirements from mobile applications for programmability, portability, security and performance.

1. The SDK variant is quite easy to develop, but it gets pretty low performance compared to its NDK and Renderscript counterparts, even after careful optimizations in the application code.
2. The NDK variant can achieve much better performance than the SDK variant. NDK's portability across different microarchitectures is an issue.
3. Renderscript gets the best performance across devices. Its memory allocation model complicates the programming, and makes the porting of legacy code difficult.

The major contributions of this work are the followings:

1. Hands-on comparison of the programming convenience and characteristics of the three Android programming models;
2. Deep analysis in the runtime behavior of the same applications in different programming models;
3. Investigation in the pros and cons of the different models, and their correlations. Based on the study, we propose a unified model.

The rest of the text is organized as follows. We discuss related work in section 2. Then we introduce our experiment setup in the study in section 3. We compare the Android programming models in various aspects in section 4 through section 7, including the differences in working flow, execution model, performance, development and deployment. Based on the investigation, we propose a unified programming model in section 8. We summarize our work in section 9.

2. Related Work

The number of applications available to a platform has been an important indicator for mobile systems since Apple launched their App Store. Platform vendors always try to attract developers to their platforms. There are many factors that can impact a developer's choice, such as the market share, platform stability, development cost, etc. According to D. Gavalas and D. Economou [1], the programming model of a platform significantly impacts the application quality and development cost.

Different programming languages require different effort to develop same application. Prechelt [6] studies 80 implementations of a phonecode program in seven languages. The result shows that designing and writing the program in script languages usually takes no more than half as much time as writing in C, C++ and Java. Besides the language diffrence, API support difference also impacts programming effort. For example, Gavalas and Economou [1] find that Android applications are easier to develop than Java ME, though both

use Java language, due to Android API's improved compatibility with the Java SE API. Familarity to a language definitely impacts a developer's choice as well. According to TIOBE Programming Community Index [7], Java and C stay at the top two positions for quite a period. C#, C++ and Object-C take the following three positions.

Given the limited resource of mobile systems, different programming models have implications on different performance and power efficiency. Some research [8] on desktop and server environment claims that the performance gap between Java and other native languages such as C and FORTRAN is very small, while some other research suggests different result. At least for specific domain [9], it is believed to have significant performance difference between Java and C/C++. More recent study on Android platform [10] shows that applications written in C/C++ do achieve better performance than those in java. Further study shows that the NDK application performance can be further improved by optimizing the JNI mechanism [11].

3. Experiment setup

In order to compare the three programming models, we choose to develop representative applications in all the models, and run them in actual Android devices. Throughout the process, we get hands-on experience in application development, operation and execution.

The application we present in this investigation is Balls [5]. The application behaves like a real Android game. It gets input from sensors (orientation and touch), conducts physics computations and draws resulted graphs on the screen. Balls simulate the movement of several hundred of bodies according to the gravity to the ground and repulsion among them. It is like the well-known scientific problem N-Body. The bodies in Balls have initial states when the application is launched and then move around autonomously. Finger can touch a body to force it move straightly thus disturb other bodies' movement through the repulsion. Figure 1 shows a screen snapshot of Balls application in execution.

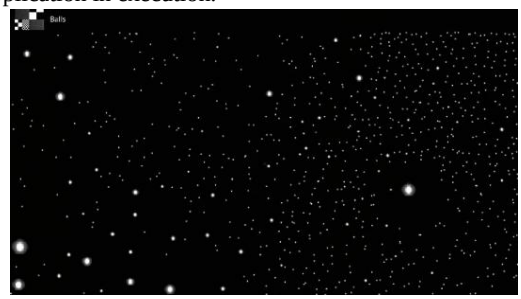

Figure 1. A screen snapshot of Balls application in execution

As common Android games, there are two main logical components in Balls. One is the computes part, which conduct the physics computation based on the movement of the bodies; the other is the graphics part, which draws the bodies on the screen.

The major differences in the programming models can be investigated according to how the two logical components – computes and graphics - are processed. Next we go through the differences one by one in details.

4. Working Flow Comparison

4.1 Android SDK working flow

When Google first released Android in year 2007, it provided only one programming model with Android SDK. It uses Java as the application programming language. Java has its natural advantages. First, it has the largest developer community than any other languages [7]. Second, Java is portable across different platforms as long as its runtime engine is available. Third, with its proven type-safety and verification mechanism, Java has language-level security, which is important for mobile devices that are mostly private.

The source code developed in Android SDK is compiled to bytecode on host machine and packaged into an application. Users can download the application from Google Android Market and then install into Android devices. When the application is launched, Android execution engine DalvikVM can interpret the bytecode or use a JIT-compiler to compile the bytecode into machine instructions and then execute them. The process is shown in Figure 2.

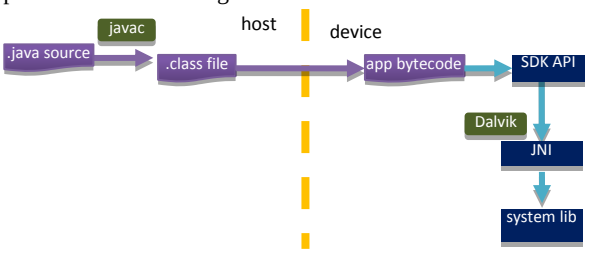

Figure 2. Android SDK working flow

Android SDK programming model is very much like the traditional J2SE programming model, except that Android SDK does not provide J2SE-compatible API but provides complete framework APIs for applications to work with the mobile system. The "Write once, run anywhere" feature of Java is also valid for Android SDK that the programs developed in it can theoretically run across different Android devices.

4.2 Android NDK working flow

Android NDK was first provided in June 2009 for developers to build library in C/C++. NDK provides a few advantages over SDK. First, Android SDK did not have OpenGL ES2.0 support in the early releases, which is critical for graphic performance. Second, developers have already accumulated lots of code in C-class languages, including those for Apple iPhone. It is unlikely for them to rewrite all their code in Android SDK. Android NDK allows developers to easily port the legacy code to Android devices. Third, in certain segments like mathematics computation, Java-programmed application still has gap to their C/C++ counterpart.

Android NDK does not provide complete programming API for Android application. Code developed in Android NDK is compiled to target machine code and packaged into apk. It has to be used with Java code explicitly or implicitly in order to run in Android device. When the application runs on mobile devices, the native code is loaded and executed through JNI (Java native interface). NDK working flow is shown in Figure 3.

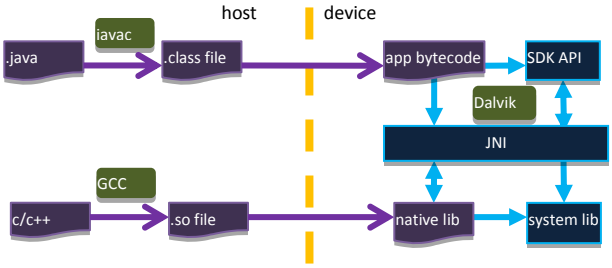

Figure 3. Android NDK working flow

The first release of Android NDK provides rather limited API.

Starting from revision 5, it supports more framework APIs to access resources. It is even possible to have native-only Android application without explicitly writing Java code, though that seriously limits the available functionalities to the application compared to that written in SDK, because most of the Android features are provided through SDK API such as services and content providers, etc. Using JNI can be a solution to access the SDK API from native code, but that just calls for unnecessary troubles.

To make a NDK application run across devices of different CPU architectures, the developer has to build different versions of the native library for the targeted ABIs (application binary interface). The developer can choose to package all the compiled versions into one application, which is called "fat binary". When installed into a device, only one version of the native library in the fat binary is used.

4.3 Android Renderscript working flow

Renderscript is a new programming model introduced since Android 3.0. It tries to solve the performance problem of SDK with Java and the portability problem of NDK with C/C++.

Renderscript chooses C99 as the base programming language, and introduces certain additional programming guidelines. The key idea is to compile Renderscript program into an intermediate representation that is close to the target architecture. Application is packaged and distributed with the intermediate representation, and the runtime engine in the device compiles the intermediate representation into machine instructions. In this way, the developer can use the flexibility in the C-like language for data manipulation, and the developed application is portable across Android devices. In other words, the position of Renderscript in Android is much like Android NDK, while its runtime philosophy is similar to Android SDK. As common Android NDK applications, Renderscript applications cannot run alone without Android SDK code.

The source code of Renderscript is compiled by C99 frontend compiler Slang into two targets: LLVM bitcode as the intermediate representation of the program, and reflection Java classes as the glue layer between the Android SDK Java code and the Renderscript code. The reflection Java code is used by the SDK code to invoke Renderscript function, manage Renderscript memory allocation and write Renderscript variables.

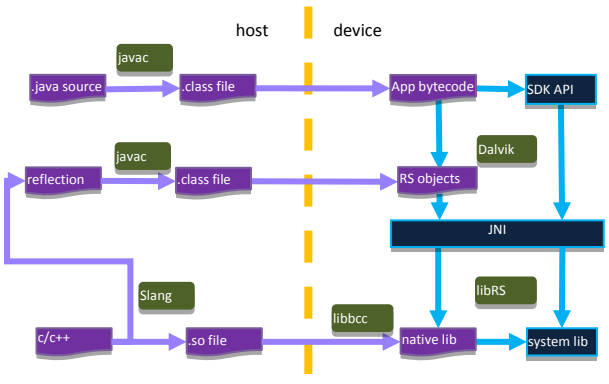

Figure 4. Android Renderscript working flow

On the first execution of Renderscript application in the device, the LLVM bitcode backend compiler libbcc compiles the bitcode into machine instructions and caches them. Later executions reuse the cached version unless the application is modified, thus trigger another compilation. Android Renderscript working flow is illustrated in Figure 4.

Renderscript provides API for computes and graphics. The computes part is a subset of C99 while it adds vector type so as to facilitate the array or matrix computations. The graphics part is roughly a wrapper of OpenGL ES2.0.

5. Execution Model Comparison

5.1 Renderscript Execution model

With Renderscript programming model, an application can basically be partitioned into two levels: the higher level SDK-developed code, and the lower level Renderscript-developed code.

The SDK-developed higher level code takes care of supporting functions to the application such as resource management, activity life-cycle and windowing system. It provides the RSSurfaceView as the drawing context to the application to draw upon.

The low level Renderscript code implements the major features of the application, including both the computes part and the graphics part. They are triggered or invoked by the higher level Java code through JNI with the reflection Java classes and libRS native engine staying aside of the JNI border. The entire rendering process is managed by a system built-in RS Proc thread so as not to block the main activity's response to the device user. When the renderer finishes the physics computation, it invokes the graphics part to draw the bodies on the RSSurfaceView context through OpenGL ES.

Renderscript provides rsForEach() API so that the render can distribute the physics computation to multiple helper threads. Figure 5 shows the execution model of Renderscript.

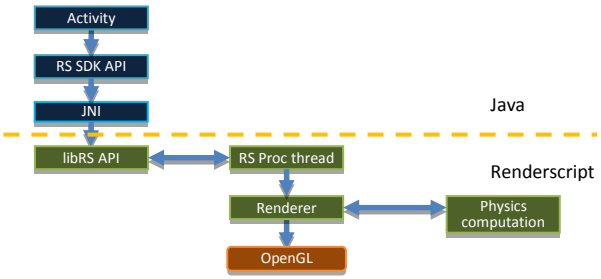

Figure 5. Android Renderscript execution model

5.2 NDK execution model

Balls implementation in NDK is very similar to that of Renderscript. It also has two levels: The higher level SDK-developed code, and the lower level NDK-developed code.

Same as Renderscript execution model, the SDK-developed higher level code in NDK also takes care of supporting functions to the application such as resource management, activity life-cycle and windowing system. It provides the GLSurfaceView as the drawing context to the application to draw upon.

The lower-level NDK-developed code is no different from other native applications: It almost controls everything it wants, as long as that is supported by NDK API. Android NDK provides a libc-like library bionic and a pthread implementation. NDK has access to OpenGL ES for graphics. In this way, it is easy for the lower level NDK code to implement both the computes part and the graphics part.

The major differences between NDK and Renderscript are two things. First, NDK model is much cleaner than the Renderscript model, just as a traditional Java application plus its native library. There is no special glue layer between Java and native code. The native code has its full control of memory allocations based on JNI or malloc. It does not depend on other code to manage its memory allocations. Second, with NDK model, the application higher level code creates a

GLSurfaceView as the graphic context. GLSurfaceView has its built-in asynchronous threading mechanism, so NDK model does not require the native code to have a separate thread (i.e., RS Proc thread in Renderscript) to manage the rendering process.

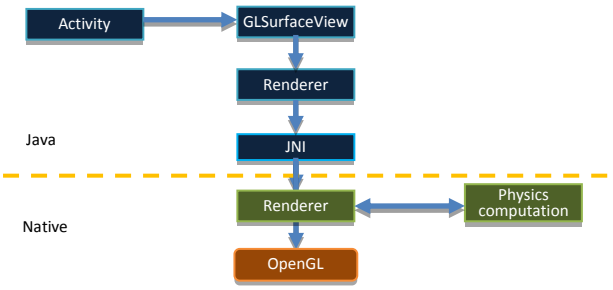

Figure 6. Android NDK execution model

In order to use native renderer implementation, the renderer class in Java code is only a simple wrapper of the native renderer. The native renderer conducts the physics computation and draws balls on screen for each frame. The execution model of Android NDK is shown in Figure 6.

In actual coding for Balls, we implement two NDK versions. The single-thread version conducts physics computation on the same thread as the native renderer. And the multi-thread version implements a thread pool with pthread API and dynamically distributes the physics computation to the threads in the pool.

5.3 SDK execution model

We implement the SDK version by moving all the native part in NDK code to Java. We still use GLSurfaceView for the graphics context since that is the class for OpenGL graphics. The Java Renderer now is no longer a wrapper, but the real entity performing the physics computation. Since Android SDK provides full OpenGL ES2.0 support, we only need to invoke the SDK API to access OpenGL support for graphics. The Android SDK execution model is shown in Figure 7.

As in the NDK version, GLSurfaceView already has built-in support for a separate thread to execute the entire rendering process, so as to avoid blocking the main activity thread's response to the device user. When the computation tasks are too heavy and inherently parallel, it is desirable to execute them in multiple threads, instead of using the same GLSurfaceView thread. So we have two SDK versions of Balls. The single-thread version performs the physics computation in the context of the renderer. The multi-thread version computes the physics in multiple threads by using Android built-in thread pool executor class. The renderer distributes the computes tasks to the pool dynamically.

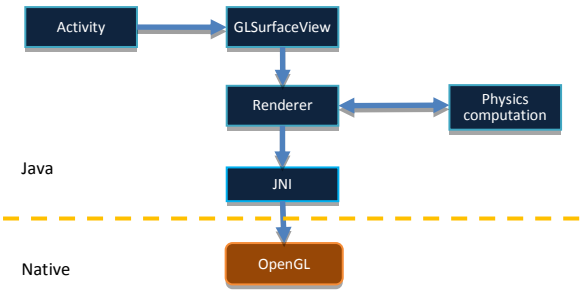

Figure 7. Android SDK execution model

6. Performance Difference and Analysis

6.1 Performance metric

We evaluate the three variants of Balls on an Android tablet with Honeycomb 3.2 OS. We mainly measure the FPS (frames per second) value of the application as the major performance metric. We choose FPS as the major metric because Android is mainly used for user interactions. The user experience of Android is largely decided by the smoothness and responsiveness of the graphic user interface that is visible to end user. For Balls, FPS is the best metric to reflect user experience.

The maximal FPS that is achievable in a platform is decided by the platform. Common nowadays devices set their maximal platform FPS around 60, because that is believed to be perfect for common people's visual cognitive ability, i.e., higher value may not be perceived by more smooth by common people, hence not worth the hardware investment.

The device in our experiment sets 60 as the maximal FPS value. That means, the device has 1/60 second (16.7ms) time for the computation of one frame. When the computation for one frame is faster than 16.7ms, the device still gives 60 FPS since that is the maximal. When the computation for one frame is longer than 16.7ms, say T, the device gives FPS value of 1/T.

As we describe earlier, Balls has mainly two logical components. One is computes and the other is graphics.

The computation amount in Balls' computes part is mainly decided by the number of bodies, which is close to a linear relation. We can increase or decrease the computation amount by simply adding or reducing the number of bodies in the application. The tablet we use has dual-core CPU processor hence dual-core parallelism support.

The computation amount in Balls' graphics part is mainly with OpenGL drawing. The tablet in our experiment has GPU (graphic processing unit) hardware that can offload the OpenGL computations from CPU. GPU and CPU can run in parallel if the software supports. In current Android design, the screen display mechanism uses two or more buffers so that the computation/composition part and the graphics drawing part can be executed in parallel to certain extent. In this way, the major computation of the graphics part is not on the critical path of CPU computation. The remaining computation part of the graphics part on CPU mainly consists of OpenGL API access and the driver invocation. Similar to the computes part, the graphics part is also related to the number of bodies, since the numbers of bodies decide the number of vertexes. At the same time, larger number of bodies incurs higher consumption of memory bandwidth.

6.2 Initial performance

Figure 8 shows the initial performance of Balls in three programming models. The initial version does not explicitly develop multiple worker threads or include any design optimizations, so the SDK and NDK versions are single threaded, while the RS version has built-in RsForEach primitive that provides implicit multiple worker threads. We can see that all of them can get the maximal FPS value 60 with a certain number of bodies. For description simplicity, we use term "saturation point" refers to the maximal number of bodies when the application can sustain 60 FPS. This is an important data to understand the trend of the behavior. The saturation points for the initial SDK, NDK and RS (Renderscript) versions are 50, 500 and 700 bodies respectively. The SDK version drops its performance drastically with more bodies, while the NDK and RS versions have almost linear performance degradation with more bodies.

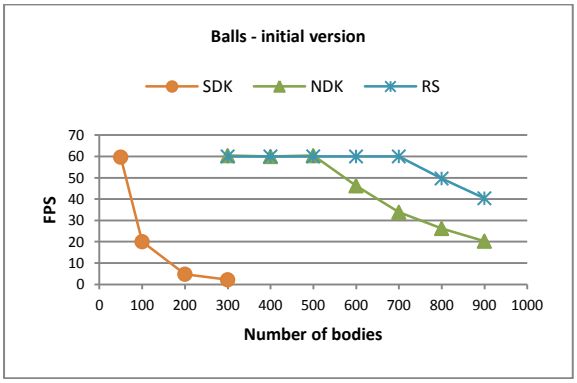

Figure 8. Performance of Balls with initial version.

The major difference between NDK and RS is due to the threading model. NDK is single-threaded, while RS is implicitly multi-threaded. Next we have a look at the major difference between SDK and NDK, when both are single-threaded.

6.3 Impact of garbage collection (GC)

To find out the root cause of the low performance of the SDK version Balls, we collect the time ratio spent in Dalvik execution engine. We partition the time into two categories. One is for physics computation; the other is for runtime garbage collection. As shown in Figure 9, we find almost half of the Dalvik time is spent in GC.

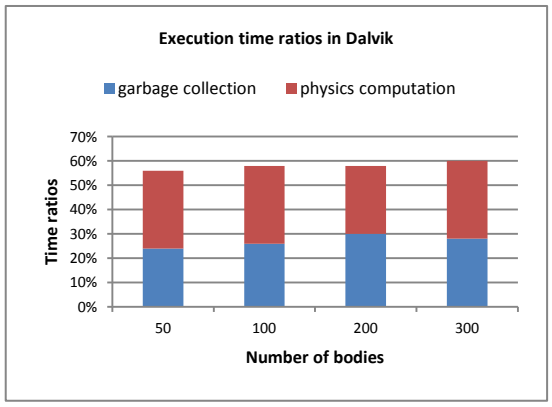

Figure 9. Time partitioning in the initial SDK version of Balls.

We find the root cause of heavy garbage collections is the common Java programming convention, where an arithmetic operation usually creates a new object and return it as the operation result, as shown in the code snippet below for the add operation of type Float2. Float2 holds two float variables x, y.

```
public Float2 add (Float2 b) {
        return new Float2 (x + b.x, y + b.y);
}
```

To reduce the GC time, we change the Float2 add operation into an accumulation instead of creating new an object, as shown below.

```
public void add_by (Float2 b) {
        x = x + b.x;
        y = y + b.y;
}
```

With this change, the GC overhead is significantly reduced and the FPS value of Balls is improved. Actually, we effectively eliminate any collection occurrence throughout Balls measurement. As shown in Figure 10, with same number of bodies (same X-axis value), the FPS value is improved a lot. For example, with 300 bodies, the original version has only FPS value 2, while the optimized version has FPS 60. The saturation point is changed from 50 to 300 bodies. This means that, within 1/60 second time, the original version can only compute 50 bodies, while the optimized version can process 300 bodies. The degradation of the optimized SDK version is also much flatter than before. In following text, when we refer to SDK version, we mean the one with the optimization.

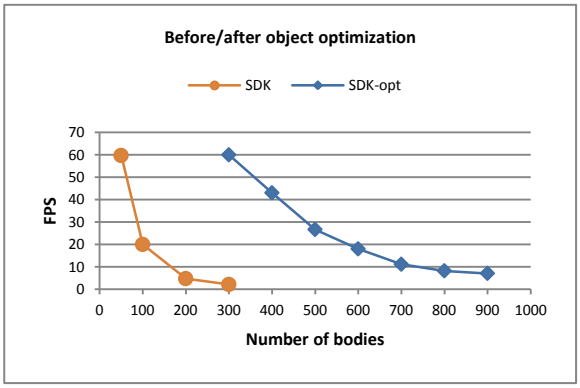

Figure 10. Performance improvement after object optimization for GC.

6.4 Impact of multiple worker threads

We also develop the multiple worker threads variants for SDK and NDK versions. For SDK version, we use the Java thread pool executor class. For NDK version, we use the pthread API in bionic library. Figure 11 shows how the performance is changed with multiple worker threads (MT).

With multiple worker threads, both SDK and NDK versions have obvious performance improvement. The SDK MT version pushes the saturation point from the original 300 to current 400 bodies. The NDK MT version pushes from 500 to 700 bodies.

This experiment means two things:

1. The multiple worker threads can effectively improve Android application's performance with the dual-core hardware support.
2. The application's performance can be improved by either reducing the computation amount of one body, or improving the throughput of the system while keeping the computation amount unchanged.

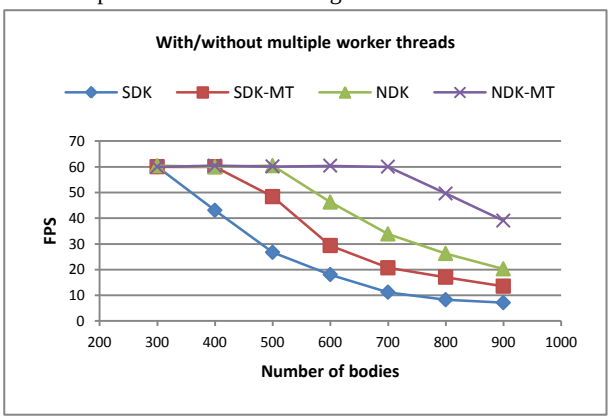

Figure 11. Performance improvement with multiple worker threads

Note, Android system is inherently designed with multiple-process and multiple-thread support. An application without multiple worker thread still could benefit from dual-core hardware, but that is out of the scope of this work.

6.5 Runtime design differences

Figure 12 shows the best achieved performance with the three programming models. We find that the NDK and RS versions are quite close in performance, while the SDK version lags far behind. The major differences are as follows. Next subsection has more data discussions.

1. The SDK version uses Java classes for mathematics, depending on runtime engine for data layout and memory management. NDK and RS versions use C-class language that has direct data layout and manipulation.
2. The SDK version uses just-in-time compiler to generate machine instructions at runtime. NDK version uses offline compiler in the host development machine to prebuild the binary. RS version uses first-execution-time compilation to generate the machine instructions, and the binary is cached for later invocations.
3. The SDK version passes the data back and forth between the physics computation and OpenGL drawing across JNI interface. NDK and RS versions keep the data mainly within the native layer.

These design differences lead to the performance differences between the SDK version and the other two versions. From runtime design point of view, all of them implement the drawing on GLSurfaceView. SDK mainly does that in Java code, while the other two versions do in native code. This conclusion sounds confusing because RS version is not native code. But from performance point of view, RS version is not far from native code in terms of data manipulation and OpenGL access.

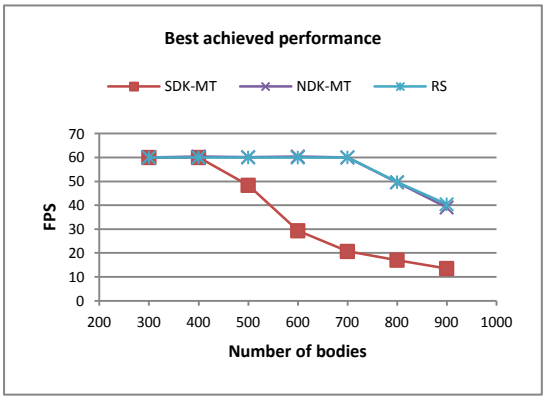

Figure 12. Best Balls' performance with the three models

The major runtime difference between RS and NDK is that, RS is not direct binary code in the device. This difference is not significant enough to be visible in Balls' performance. This as a side fact proves that the advanced compiler optimizations are not always critical for mobile applications.

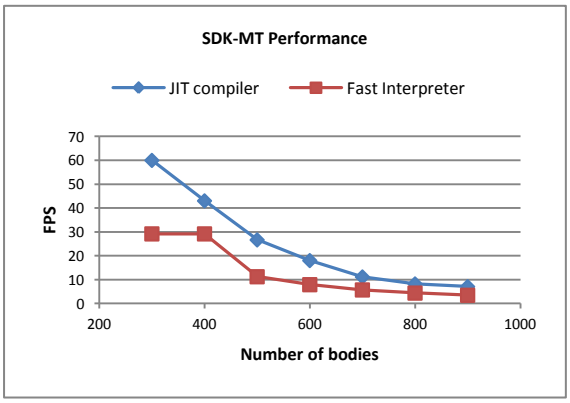

Figure 13. Performance comparison between faster interpreter and JIT compiler

Although the performance difference between RS and NDK is minimal, we still can see that in the figure. The reason for RS has a very subtle advantage is that, RS has built-in vector type support, which makes the large volume of vector processing fast. It is actually easy for NDK to implements vector class.

Although our data shows that advanced compiler optimizations may not be critical for mobile applications, we want to understand if that is true between compiler and interpreter. Dalvik has a fast interpreter that is written in assembly code for target architecture. We compare the performance between the fast interpreter and the JIT compiler in Figure 13. For every configuration, the compiler run always gets better performance than the interpreter run, though the gap becomes smaller with more bodies.

6.6 Computation intensity

To really understand the runtime behavior of Balls under different programming models, we measure the time spent by each version in drawing one frame and the CPU utilization ratios, shown in Table 5 - Table 4. This reflects the computation intensity on the dual-core CPU, which also gives us an impression about the room of further optimization.

The frame time in a configuration is the reciprocal of the FPS value achieved by that configuration. For example, when the configuration achieves 60FPS, the frame time is 16.7ms. We partition the time of one frame into computation part and the rest. We only show the computation part in tables below since the rest time can be simply deducted.

CPU utilization refers to how much time the CPU is actively working. If the CPU utilization is 50%, it means only half of the processor resource is used. In other words, half of the processor resource is not used, or idle. If it is a dual-core processor, possibly there is one core unused.

Since the device display system only refreshes the screen at a maximal 60 FPS frequency, that means, if the processor can finish a frame's computation within 1/60 second (16.7ms), the application can achieve the maximal FPS. There are two cases regarding the frame computation time.

In case one, if the application's frame computation time T is less than 16.7ms, the application can get 60 FPS, but no more, because 60 FPS is the platform maximal. Then within each frame, there is a period of (16.7 – T) when the processor does not contribute or is idle. The shorter T is, the longer the idle time is, and the smaller the CPU utilization is. In this situation, although the FPS is no longer increased, the device power is less due to the longer CPU idle time.

Table 1. Computation intensity of SDK-ST Balls

SDK-ST version of Balls							
Number of Balls	300	400	500	600	700	800	900
FPS value	60	43	27	18	11	8	7
CPU utilization	37%	48%	49%	49%	49%	49%	49%
Frame-time (ms)	16.7	23.2	37.5	55.4	89.4	121.2	140.5
Computation (ms)	12.4	22.2	36.2	54.0	87.4	118.9	138.0

Table 2. Computation intensity of SDK-MT Balls

SDK-MT version of Balls							
Number of Balls	300	400	500	600	700	800	900
FPS value	60	60	48	29	21	17	14
CPU utilization	46%	77%	88%	90%	90%	92%	92%
Frame-time (ms)	16.7	16.7	20.7	34.1	48.4	58.8	74.0
Computation (ms)	8.2	12.3	19.5	32.7	46.8	56.9	71.9

Table 3. Computation intensity of NDK-ST Balls

NDK-ST version of Balls							
Number of Balls	300	400	500	600	700	800	900
FPS value	60	60	60	46	34	26	20
CPU utilization	25%	33%	45%	49%	49%	49%	49%
Frame-time(ms)	16.6	16.7	16.6	21.6	29.5	38.1	49.3
Computation(ms)	8.6	11.0	15.2	21.3	29.3	37.9	49.2

Table 4. Computation intensity of NDK-MT Balls

NDK-MT version of Balls							
Number of Balls	300	400	500	600	700	800	900
FPS value	60	60	60	60	60	50	39
CPU utilization	34%	43%	53%	70%	86%	92%	92%
Frame-time(ms)	16.7	16.6	16.6	16.6	16.7	20.2	25.6
Computation(ms)	6.0	8.0	9.7	11.2	15.7	20.0	25.4

Table 5. Computation intensity of Renderscript Balls

Renderscript version of Balls							
Number of Balls	300	400	500	600	700	800	900
FPS value	60	60	60	60	60	50	40
CPU utilization	34%	44%	54%	68%	88%	94%	96%
Frame-time(ms)	16.7	16.7	16.7	16.7	16.7	20.1	24.8
Computation(ms)	6.6	7.4	9.0	12.2	15.7	19.8	24.3

In case two, if the application's frame computation time T is more than 16.7ms, the application cannot get the perfect maximal FPS value, but gets 1/T FPS. In this situation, the application computes slower than the display refresh rate. It has to keep busy in frame computation. In a single-core processor, the CPU utilization should be close to 100%. Otherwise, if the CPU utilization is obviously lower than 100%, there must be thread synchronization or I/O blocking optimizations. In case of dual-core processor, to achieve 100% CPU utilization needs to have no less than two parallel working threads. If there is only one active working thread, the maximal CPU utilization it can achieve in a dual-core platform is 50%.

In tables, the column for the saturation point of each version is shown in dark color. We can see in NDK-ST and SDK-ST versions that, the CPU utilization ratios are always lower than 50%. After the saturation points, the ratios become 48%-49%. This clearly indicates the single worker thread nature of the two versions. With NDK-MT and SDK-MT versions, the CPU utilization ratios beyond the saturation points are in the range of 88%-92%. As a contrast, the RS version has CPU utilization ratio in range of 94%-98% after the saturation point. That means, the parallelism we exploit with the thread pool in Java and pthread is not as sufficient as the RS built-in support.

We also observe that the computation time beyond the saturation points. The computation time in all the versions is almost the same as the frame time. That means, as long as the CPU is working, it is working in the frame computation. It does not waste CPU resource in other thing. If we compare the computation time between the single-thread and multithread version pairs, we can find that, SDK-MT's computation time is about half of SDK-ST's, and NDK-MT's computation is also about half of NDK-ST's. This is not surprising, because the physics computation is quite parallel in nature. It also indicates that the thread pool implementations in SDK and NDK versions have good scalability.

7. Differences in Development and Deployment

Above we analyze the differences in the three programming models from a system designer's point of view. In this section, we have a comparison from an application developer's point of view.

Firstly, we summarize the programming API differences we experience with Balls development in Table 6. We classify the API differences into five categories: user interaction and resource management; graphics (i.e., OpenGL support); computes (i.e., vector support); worker threads pool; and library extensibility.

From the table, we can see that the major drawback of SDK development is the lack of vector type support. We have to use Java class to simulate. Except that, all the rest API support with SDK is complete and convenient. The vector support is actually not only an API issue, but also a performance issue, as our data show in early section.

Table 6. Programming API differences

	User interaction and Resource management	Graphics (OpenGL)	Computes (Vectors)	Worker thread pool	Library extensibility
SDK	Full support	Full support	No	Full support	Java support
NDK	No	Full support	Can support	Can support	C++ support
RS	No	Most support	Built-in	Built-in	No

NDK and RS do not have support for user interaction and resource management. They have to work with Java code to have those functionalities. NDK and RS can get similar performance in our experiment, while NDK needs extra development efforts, because it does not have built-in vector type and worker thread supports.

Next we compare the memory management API in the three programming models in Table 7. We classify the API differences into three categories: memory allocation; memory release; and data sync between GPU and CPU.

Table 7. Memory management API differences

	Memory Allocation	Memory Release	Data sync between CPU and GPU
SDK	Automatic	Automatic	Manual
NDK	Manual	Manual	Manual
RS	Manual in Java layer	Automatic	RS Sync API

With NDK, one has to do everything manually with the full control of memory. This is tedious and error-prone as proved by tons of literatures. SDK has garbage collection support so the allocation and release is done automatically. Neither NDK nor SDK has the data synchronization support between CPU and GPU, which has to be done manually.

RS is quite different in its memory management API. It has the design philosophy that RS application could run on heterogeneous architecture. For example, the same RS code can be compiled to run on CPU or GPU based on its runtime strategy. So RS API is designed with heterogeneity support in mind. This is not the case for SDK (Java) and NDK (C++). RS design philosophy is reflected in its memory management API. First, RS provides API for data synchronization between CPU and GPU. Second, RS does not give the freedom of memory allocation in RS native code because that would bind the memory to CPU or GPU statically. RS only allows memory allocation in Java layer, which is independent of the underlying architecture. Third, to correspond to the Java layer memory allocation, RS provides automatic memory release with reference counting. Given the fact that the memory allocation and assignment requests are both raised in the main thread as an asynchronous command to RS rendering thread, it is difficult for developer to change allocation during physics computation.

In application deployment stage after it is developed, the installation package size is important for user experience. The

default NDK development tool chain builds native binary for every supported ABI target and package all of them into the final application. That increases the size of application package APK (Android application package). At the same time, Java bytecode is usually much more succinct than the generated machine instructions. As a result, the size of NDK Balls is 2.4 - 2.7 times of that of SDK and RS Balls, as shown in Table 8. As a workaround for portability, "fat-binary" is not really desirable.

Table 8. APK size of different versions of Balls

	SDK	NDK	RS
Package size(KB)	25	60	22

8. Unified Programming Model

Based on the comparison and analysis above, we see that each of the three models SDK, NDK and Renderscript has its respective advantages and disadvantages. There are a few common pursuits in all the Android programming models.

A. The model should be easy to use for developers.

This goal includes two aspects: the language and the APIs. In language aspect, Java in SDK, C++ in NDK, and C99 in RS are all familiar to the developer community after years of evolution. In API aspect, people expect automatic memory management and library extensibility. Extensibility is important to port current physics engines to Android. Only SDK can satisfy both.

B. The model should provide device-portable solution.

SDK is portable by nature with Java language and runtime support. NDK is not portable by nature with precompiled binary unless fat-binary is provided as a work-around. RS compiles the C99 code into LLVM bitcode, which makes it portable in Android devices that have RS runtime support.

C. Applications developed in the model should have reasonable security warranty.

Security is always critical for software. It is even more critical for software in mobile devices, because they usually carry the users' sensitive information. Java with its strong typing and verification mechanism provides high level of security at language level. C++ does not have similar concepts. RS does have security advantage over C++ with its strict memory management. But RS does not put strong restrictions on pointer usage. For example, programmer can simply assign a random number to a pointer without encountering build-time failure. On the other hand, at system level, Android smartly leverages the underlying Linux's support in process isolation and user access control, which effectively prevents a problematic application from making system-wide destructive impact. But that cannot solve all the potential issues that a casual user could meet, if the user is not cautious enough in granting privilege permissions.

D. Applications should have reasonable performance for both computes and graphics.

Performance requirement for computes need vector type support and worker thread pool. Only RS has both. Performance requirement for graphics need OpenGL ES. RS does not have full support while SDK and NDK have. It is also important to avoid lots of data movement back and forth across JNI or between CPU and GPU. Since graphic data has to finally stay in OpenGL domain in native layer, it is desirable to have a solution to either keep the data in native layer or avoid the data copying across JNI. Current SDK does not provide the solution.

The issues in different models are summarized in Table 9.

We argue that, some of the issues can be resolved easily or not so easily, while some other issues can be really difficult to resolve. The graphic data management issue with SDK is hard to be eliminated without changing the JNI implementation. Library extensibility in RS is not going to be as convenient as Java and C++.

Table 9. Issues in current Android programming models

	Programmability	portability	Security	Performance
SDK	no issue	no issue	strong	vector processing, data management
NDK	memory management	fat binary	weak	vector processing, thread pool
RS	memory allocation, library extensibility	no issue	weak	full OpenGL support

In this regard, we propose a new programming model that helps to remove the critical issues in current models by combining of the existing models. It introduces vector type in Java layer, and defines runtime to eliminate the data copying across JNI as RS does. The new model compiles NDK C++ code into intermediate representation as RS does. It also introduces vector type and implements worker thread pool API in NDK. In this way, developers do not need to write separate C99 code with RS API but keep the benefits of RS.

The challenge in the new model is mainly with security. To keep the C++ language means to keep its security risk as well. The risk can be largely reduced by limiting the usage of pointer, in both data access and function access. Also the API should be limited to secure ones especially for those memory related operation such as string manipulation functions.

The limited C++ is very similar to subset of GO language [12]. GO language keeps pointer for performance while removes pointer arithmetic. It distinguishes pointer from normal integer with tag bits. This enables accurate garbage collection. Besides this, GO provides extendable stack which makes stack overflow almost impossible. With such strict memory management, GO language is promising in the safety perspective. GO language inherits performance designs from C language, such as static type, simple structure layout, etc. Thus it is possible to achieve similar performance as C. As a practical language, it has various features for ease of use, such as interface, closure, go statement for concurrent execution and channel type for synchronization. Considering the better safety, GO language could be a good alternative of C++ in NDK.

Finally in the runtime engine implementation, the new model unifies the intermediate representation for both Java and C++ (or GO) code, and no longer packages binary in APK. Then one runtime engine is enough to compile all the application code. The compilation infrastructure can be flexible to support adaptive compilation, user-specified compilation, ahead-of-time compilation etc.

9. Conclusion

In this work, we investigate the programming models in Google Android, i.e., SDK with Java, NDK with C++ and Renderscript with C99. To deliver the best user experience to the end users, it is important for the system software designers to understand these implications and relations. We develop variants of the same representative Android application Balls in all of the three models. Based on the same application, we conduct detailed apple-to-apple analysis of the models. We find that, each of the three models its respective advantages and disadvantages. There is not a single model can satisfy all the requirements from mobile applications for programmability, portability, and performance. We propose a unified programming model that helps to remove the critical issues in current models by combining of the existing models.

Bibliography

[1] D. Gavalas and D. Economou, "Development Platforms for Mobile Applications: Status and Trends," *Software, IEEE,* vol. 28, no. 1, pp. 77-86, Jan.-Feb. 2011.

[2] "iOS Dev Center," Apple Inc., [Online]. Available: https://developer.apple.com/devcenter/ios/index.action. [Accessed 21 Feburary 2012].

[3] "Android SDK," Google Inc., [Online]. Available: http://developer.android.com/sdk/index.html. [Accessed 21 Feburary 2012].

[4] "Android NDK," Google Inc., [Online]. Available: http://developer.android.com/sdk/ndk/index.html. [Accessed 21 Feburary 2012].

[5] T. Bray, "Introducing Renderscript," 09 February 2011 . [Online]. Available: http://android-developers.blogspot.com/2011/02/introducing-renderscript.html. [Accessed 21 Feburary 2012].

[6] L. Prechelt, "An empirical comparison of seven programming languages," *Computer,* vol. 33, no. 10, pp. 23-29, Oct 2000.

[7] "TIOBE Programming Community Index for February 2012," [Online]. Available: http://www.tiobe.com. [Accessed 10 February 2012].

[8] J. M. Bull, L. A. Smith, L. Pottage and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, New York, NY, USA, 2001.

[9] M. Fourment and M. R. Gillings, "A Comparison of common programming languages used in bioinformatics," *BMC bioinformatics,* vol. 9, no. 1, pp. 82-91, 2008.

[10] C.-M. Lin, J.-H. Lin, C.-R. Dow and C.-M. Wen, "Benchmark Dalvik and Native Code for Android System," in *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, Shenzhen, Guangdong, China , 2011.

[11] Y.-H. Lee, P. Chandrian and B. Li, "Efficient Java Native Interface for Android Based Mobile Devices," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, Changsha, Hunan Province, P. R. China, 2011.

[12] "The Go Programming Language," Google Inc., [Online]. Available: http://golang.org/. [Accessed 1 5 2012].

[13] "The Computer Language Benchmarks Game," [Online]. Available: http://shootout.alioth.debian.org. [Accessed 10 February 2012].

[14] T.-M. Grønli, J. Hansen and G. Ghinea, "Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments," in *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, Samos, Greece, 2010.

[15] "Renderscript," Google Inc., [Online]. Available: http://developer.android.com/guide/topics/renderscript/index.html. [Accessed 21 Feburary 2012].

[16] "The Go Programming Language," GOOGLE, [Online]. Available: http://golang.org/. [Accessed 29 4 2012].