Boosting UI Rendering in Android Applications

Subrota Kumar Mondal*, Yu Pei‡, Hong Ning Dai*, Jyoti Prakash Sahoo†
*Faculty of Information Technology, Macau University of Science and Technology, Macau, China.
‡Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China.
†Department of CSIT, Institute of Technical Education and Research (ITER), Bhubaneswar, India.
Email: {skmondal, hndai}@must.edu.mo, csypei@comp.polyu.edu.hk, jyotiprakashsahoo@soa.ac.in

***Abstract*—The Android operating system captures over 86% mobile OS market share and a large number of software developers are keen on developing applications for the Android platform. Many Android applications, however, suffer from the problem of slow UI rendering, thereby losing their competitive edge. To be able to address this problem, the developers first need to understand the underlying reasons. In this paper, we present an empirical study on reasons for slow UI rendering on the Android platform, with its focus on the impact of (poor) layout implementation on UI rendering. We also propose a taxonomy of existing techniques that might help tackle the problem and strategies for efficient layout implementation. Results from applying the strategies to sample applications demonstrate that they can help enhance the efficiency of UI rendering.**

***Index Terms*—Layout, UI Rendering Issues, Boosting UI Rendering.**

I. Introduction

Android is the dominant mobile OS with over 86% market share [1]. Further, analysts believe that the use of Android will continue to grow. As such, software developers are more keen on developing applications for the Android platform. On the other hand, developers confront with numerous challenges in developing Android apps. In addition, different Android OS versions bring-in intricacies to the developers. Besides, there are more than 24,000 distinct device models [2] running the OS. The diverse features of every individual device introduce nightmare to the developers. Moreover, resources such as processing power, memory, and battery power are limited on smartphones. Thus, performance of smartphone apps are crucial factors on user experience.

We find that slow UI rendering is one of the prime factors in degrading the performance of an app. Aspect such as, this paper strictly focuses on identifying and rectifying *rendering issues* for enhancing performance of applications. We know that "performing heavy processing in the main thread" causes slow UI rendering, but note that there are some other important factors which play significant role in assessing the performance of an app. Layouts are one of them. They are pivotal components of Android applications that precisely affect the user experience. A poorly implemented layout may lead to significant memory consumption, so the performance of the app may degrade [2]. As such in this study, our goal is to find out the efficient layout implementation schemes for simply boosting UI rendering (note that in this paper, *improving layout performance* or *boosting UI rendering* are used synonymously).

The rest of this paper is organized as follows: Section II first gives a brief overview of a layout initialization. It further demonstrates commonly used layouts types in Android and their usage scenarios. Section III presents an analysis for identifying the sources of rendering issues. It further helps us disclose the significance of layouts' performance improvement in solving the rendering issues. Section IV discusses about employing the different solution strategies for improving layout performance in our applications. We conclude this paper in Section V.

TABLE I: Comparative analysis of layouts of its kind.

Layout Type	Use cases	Alternatives
Linear Layout	Simple vertical or horizontal alignment of elements without any nesting.	Relative or Constraint Layout is better choice for nested elements.
Relative Layout	Flexible, used for custom layout designing.	Constraint layout is a better alternative.
Constraint Layout	Flexible, used for building large and complex layout with a flat view.	It is the best option for custom layout designing.
Frame Layout	Has a single child view. Used as a container or parent layout.	Coordinator layout can be used, but not a complete alternative.
Coordinator Layout	Used as a toplevel application decor for a specific interaction with child.	Much like FrameLayout but its behavior attribute makes it different.
List View Layout	Groups items and displays them in vertical scrollable list, e.g., Contacts, Messages.	RecyclerView layout is a better alternative.
Grid View Layout	Renders elements in a 2D scrolling grid, e.g., Default Gallery, Tools, Themes.	CardView with RecyclerView is a better alternative.
Recycler View Layout	Rendering frequently changing data sets, e.g., online video streaming app.	It is the Advanced and flexible version of ListView.

II. Layouts in Android

Usually, layouts form the visual structure for UIs, e.g., the UI for an activity or app widget. In general, the key component for building UI is *View* object, which takes a rectangular area on the screen and helps us draw and handle events. The another component is *ViewGroup* - it supplies invisible container which itself can contain View instances. Usually, a layout is the integration of Views and ViewGroups [3].

A. Android Layout Types

In Android, we have various types of layout, which we can use in applications to render diverse view, look, feel and especially to meet the specific demand. The most commonly used layouts are **Linear Layout** and **Relative Layout**. There are some other popularly used layouts such as **Constraint Layout**, **Frame Layout**, **Coordinator Layout**, **List View Layout**, **Grid View Layout**, **Recycler View Layout**, **CardView Layout**, and so on. We find that layout designing should be flat, i.e., there should not be any nesting in layout design and implementation.

In Table I, we present the use cases and alternatives of the listed layouts. It shows that Constraint layout is the best choice

for custom layout designing and RecyclerView is the advanced and flexible option with scrollable features for frequently changing large data sets.

III. Analyzing and Identifying Rendering Issues

The analysis in this section first discusses about profiling an application which let us know whether the app suffers from rendering issues or not. It follows analyzing Android rendering pipeline for identifying the sources of rendering issues. We conclude this section figuring out the significance of layouts' performance improvement in solving the rendering issues.

A. Android Rendering Pipeline

Rendering is the process of building viewable objects. The rendering pipeline is bifurcated into the components: CPU and GPU [4]. They work together to render the view or draw the image on the screen.

- CPU: It generates the Display List of a view and executes them. The List keeps the info to render a view at GPU.
- GPU: The processed List on the CPU is passed to the GPU for rasterization. If no update in the CPU side, reached from GPU.

B. Sources of Rendering Issues

The sources of rendering issues might be:

- On the CPU side, the most common rendering issues come from using unnecessary layouts and invalidations in the view hierarchy - (1) frequently rebuilding the display list, and (2) wasting time for invalidating unneeded components of the view hierarchy and redrawing unnecessarily [2].
- On the GPU side, we might face an *overdraw* problem in which an app may draw the same pixel more than once within a single frame [5]. This redundant operation should be undoubtedly eliminated. Specifically, these overdrawn pixels do not contribute to what the we see on the screen [5], [6]. On the other hand, it consumes unnecessary GPU time, which may degrade performance.

IV. Improving Layout Performance

In this section, we present the state-of-the-art solution strategies for improving layout performance. Further, we evaluate different representative samples to determine how well we can apply this solution strategies to enhance the performance of Android applications. We find that we can boost the rendering of a UI in a various ways, such as Flattening Layout Hierarchies, Reusing Layout, Loading Views on Demand, Enhancing ListView Scrolling, Optimizing GPU Overdraw, as shown [2].

1) Flattening Layout Hierarchies:
   - Layout Inspection: We can use Hierarchy Viewer [4] to detect unnecessary nested view groups and to compare the time it takes to measure, layout, and draw.
   - Layout Revision: We can revise the nested layout by shallow and deep version, such as, Relative, or Constraint layout.
2) Creating Reusable Layout and Reusing Onward:
   - Reusable Layout Creation: App UI reiterates certain layout constructs, e.g., title bar, progress bar, so reusing them favors to create reusable complex layout. We can use $< merge >$ tag for this - it helps avoid repetitious view group.
   - Reusing by $< include >$ tag: We can emplace the reusable layout by $< include >$ tag.
3) Loading Views on Demand:
   - Defining a ViewStub: Loading the views only when desired helps boost UI rendering. ViewStub helps carry out this without taking part in the layout.
   - Loading the ViewStub: Usually, we can load a defined ViewStub by (1) setting it visible or (2) by calling $inflate()$.
4) Enhancing ListView Scrolling:
   - Using a Background Thread: AsyncTask keeps the main thread free from heavy work.
   - Using View Holder Design Pattern in RecyclerView: It helps reuse recycled items that helps optimize list item inflation, inner view retrieval, and memory consumption for creating new list items.
5) Optimizing GPU Overdraw:
   - On-devic **Debug GPU Overdraw** tool helps visualize overdraw problem.
   - Step-by-step, we can remove overdraws from (i). activity's window, (ii). parent ViewGroup, and (iii). ListView.

V. Conclusion

In this paper, we have studied, characterized, and analyzed a wide range of works related to performance analysis and enhancement of Android Applications. In particular, we have focused on improving layout performance or boosting UI rendering. Aspects such as, this paper has brought out the insight of Android rendering process and revealed the fundamental issues in UI rendering. Further, We have highlighted the state-of-the-art solution strategies to address those issues and evaluated their effectiveness to validate them. We thus hope that the developer community can better explore various potential opportunities to further enhance the performance of Android applications.

References

[1] IDC. Smartphone OS Market Share Q1 2020. URL https://www.idc.com/promo/smartphone-market-share/os.

[2] K-Fox A. Android Training: 'Docand' Reference Series 2/6 Nov 2013 - API19 - Volume 2 | Improving Layout Performance 2013; :574.

[3] K-Fox A. Android API Components Part 1:'Docand'Reference Series 3/6 Nov 2013-API19-Volume 3 | Layouts 2013; :350–357.

[4] K-Fox A. Android Studio:'Docand'Reference Series 6/7 April 2017-API24-Volume 6 | Profile Your Layout with Hierarchy Viewer 2017; :467–471.

[5] Reducing Overdraw 2020. URL https://developer.android.com/topic/performance/rendering/overdraw.html.

[6] Garbe M. Introduction to the Android Graphics Pipeline. *Bahelor Thesis, Informatik, Karlsruhe University of Applied Sciences* March 2014; .