Reducing Power Consumption in Graphic Intensive Android Applications

Amit Singhai, Joy Bose
Web Solutions
Samsung R&D Institute India- Bangalore
Bangalore, India
{a.singhai , joy.bose}@samsung.com

***Abstract*— With the ubiquity and increased usage of mobile devices with higher processing capability, power consumption in such devices is an important issue. In this paper we propose a software based approach to reduce the power consumption of an Android application which is having higher graphics or rendering requirements. Graphics intensive applications such as games, internet browser and video player contribute most in draining the battery. Therefore, we concentrate most on such applications for reducing the power consumption. Our approach involves removing the surface view and directly utilizing the application's main window surface. We conduct an experiment to measure the power consumption when running an application with and without using surface view, and show that we get an improvement in the consumption. We also discuss about advantages and problems of the proposed method along with some possible solutions.**

***Index Terms*— android, power consumption, current measurement, surface view, window manager, surface flinger**

I. INTRODUCTION

Mobile computing devices such as smart phones and tablets are embedded systems mostly powered by battery. Power consumption in such devices is an important issue, with the advances in processing capability of such devices and restrictions on the battery size and weight.

Certain applications such as network I/O intensive applications consume lot of power due to transmitting and receiving signals, whereas graphics intensive applications like games consume lot of power due to large GPU and CPU utilization.

In this paper we propose a method which results in lesser power consumption for graphic intensive applications on mobile devices, such as browser, games, and video player. We consider this power saving approach in the context of the Android operating system, a popular operating system for mobile devices. Our method is to reduce the CPU/GPU utilization in maintaining and merging two surfaces. Although in this paper we consider only Android, the same principle can be theoretically applicable for other mobile operating systems such as iOS, Tizen, Windows and others.

The rest of the paper is organized as follows: section 2 looks at related work, section 3 talks of applications using surface view and the power consumed by such applications. Section 4 looks at the proposed method to optimize power consumption. Section 5 discusses experiments to measure the improvement in power consumption using our approach. Section 6 talks of the pros and cons of using surface view, and section 7 provides some methods to overcome the disadvantages. Section 8 concludes the paper.

II. RELATED WORK

In order to minimize the power consumption, one way is to minimize the utilization of hardware components. Turning off sensors which are not being used at the user level is one method to achieve this. Hardware manufacturers especially in embedded systems domain are coming up with the latest processors and DSPs which consume less battery and can also operate on low voltages. To understand utilization of power in various hardware components, the work by Aaron Carroll [5] is very useful.

Another approach to optimize the power consumption is software based. Programmers can optimize the power consumed by using software techniques, efficient design and algorithms.

Thiagarajan et al [6] study energy consumption in mobile browsers, but this too does not focus on the Graphics side. An NVidia whitepaper [7] details techniques to enable high end graphics in mobile devices. However, it is also concentrated on the hardware side and do not talk of how power consumption can be optimized by using programming techniques.

Generally speaking, the software aspect of power optimization in graphics is not well studied. In this paper, we focus on software techniques to reduce the power consumption for Android devices on the graphics and rendering side.

III. APPLICATIONS USING SURFACE VIEW

In the Android OS, only the UI thread can update the user interface. This means that worker threads cannot update the views. For graphics related application like games, updating the view in a separate thread is desirable.

To solve this problem, Android provides a special view called Surface View which has its own surface.

An example activity in Android is shown in fig. 1. It has one surface view along with other standard android widgets e.g. "Text View". The surface view requests a transparent hole

978-1-4799-3635-9/14/$31.00 ©2014 IEEE

in the window in which it has been added [1]. This transparent hole allows user to see the surface (which belongs to surface view) behind the window. Since surface view has its own surface, a separate thread can update it. It is particularly useful in games and other applications where rendering is done by native code.

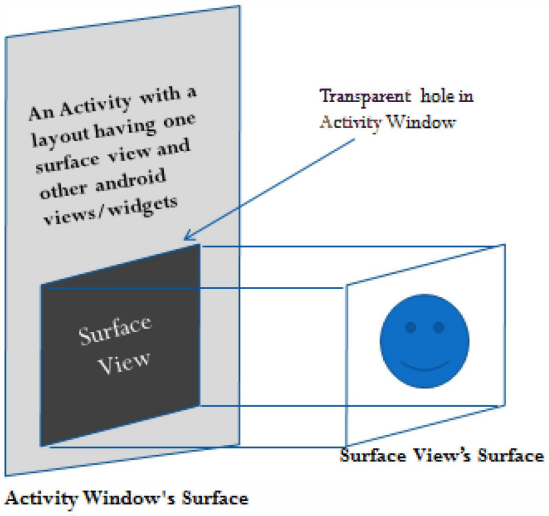

Fig. 1. Mechanism of the Surface View in Android OS.

One drawback of this mechanism is that it consumes more power as two different surfaces are maintained. Also, the upper surface contributes in more processing as the area acquired by surface view has to be made transparent so that the layer below (surface view's own layer) can be visible. These surfaces have to be painted separately and merged at flinger level. However using surface view has its own benefit. It is part of the view hierarchy, so any kind of manipulation and touch event processing are easily possible.

In the following section, we explain how power consumption can be optimized when using Surface View.

IV. Proposed Approach for Optimizing the Power Consumption

In the previous section we introduced Surface View and explained how it consumed more power since two surfaces are maintained and merged. In our approach, we propose to save power consumption by directly drawing on the application's main window surface.

In certain applications where Android specific widgets such as buttons or text edit areas are minimal, it is possible to directly use the surface of the window instead of surface view for rendering or drawing.

This approach has some limitations. If the User Interface has a lot of views which respond to user touch, using the window's surface directly is not recommended. So the proposed approach can be utilized mainly if the application is graphics intensive, and where the number of android widgets shown on main window is negligible. Once the window's surface is taken for showing contents, other views cannot be visible until they are directly added via the window manager. Adding a view via 'WindowManager' has some disadvantages, which are discussed in a later section of this paper.

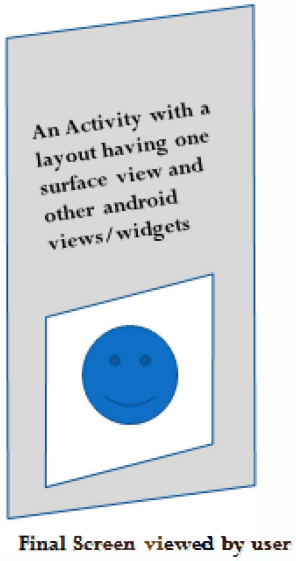

Fig. 2. The composited final screen viewed by the user in Android OS.

In the following section, we describe an experiment to measure the efficacy of the proposed method for optimizing power consumption.

V. Experiment to Measure the Improvement in Power Consumption While Directly Utilizing Application Main Window's Surface

In our approach, the application main window's surface is used to render the content [2]. In this section, we test the effectiveness of this approach in reducing power consumption. We do this by measuring the power consumption in the two mentioned cases (with and without surface view) for the same graphical rendering requirement on the same device. In this way, we find out the improvement by removing the surface view and drawing directly on window's surface. Later we verify the number of surfaces visible at flinger level in our test applications.

A. Experiment to measure the power consumption for content rendering on surface in both the cases (with and without surface view)

We conducted an experiment to find out the power consumption when only drawing is being performed. For this two Android applications have been developed using the publicly available Android SDK. In the first application, random bubbles are drawn on the surface of surface view in a thread. In the second application, random bubbles are drawn with exact size and duration as in first application, but here they are drawn directly on the surface of main window.

For precise measurement all sensors such as accelerometer are disabled on the Android device. For drawing directly on the surface of window, an API provided by Android called Window.takeSurface (SurfaceHolder.Callback2) has been

used. We used a Samsung Galaxy Note3 device and a Power Monitor from Monsoon Solutions [4] for our measurements. The power monitor replaces the battery and logs the power consumed. The user has a choice to start and stop the measurement as they wish. We kept the input voltage constant at 4.0 volts, so measuring the drawn current alone, rather than the power consumed, is sufficient.

We recorded the current consumption for 10 seconds of application run time with stable current output from the power monitor equipment. Measuring the current drawn for a longer time has no appreciable advantage, because current readings are taken at stable output.

The results are displayed in table 1 for three iterations. As we can see, we get an approximately 7 mA advantage in the power consumption in our approach, as compared to the power consumption of the application with the surface view approach.

TABLE I. MEASUREMENT OF THE AVERAGE CURRENT WITH AND WITHOUT SURFACE VIEW

Iteration number	Application running time (in seconds)	Average current with Surface View (mA)	Average current without Surface View (mA)
1	10	189	182.7
2	10	189	182.12
3	10	188.5	182.3

A second set of measurements has been taken on a Chromium based browser on the same Android device. The browser uses heavy graphics and rendering capabilities to draw html content on the screen. The Chromium based browser uses surface view for drawing the content.

TABLE II. MEASUREMENT OF THE AVERAGE CURRENT WITH AND WITHOUT SURFACE VIEW IN CHROMIUM BASED BROWSER

Scenario for power measurement	Average current with Surface View (mA)	Average current without Surface View (mA)	Improvement with our approach
Pinch & zooming of content	924.89	922.02	2.8 mA
Browsing same web site over wi-fi	604.73	543.84	60.0 mA

We modified the browser code so that it utilizes widow surface for drawing the content instead of surface view. With two Application packages (.apk) prepared with and without surface for same browser code base we were able to take power readings and thus compare the results in both the user scenarios. The readings have been taken 3 times for each use case and the average results have been taken into consideration.

The results are shown in table 2. As we can see, our approach results in an improvement in the power consumption while using heavier applications such as a web browser.

B. Experiment to find the visible surfaces for the two cases (with and without surface view)

To find out various visible surfaces or layers in our application, we use an Android SDK provided tool called "dumpsys". We use this tool for capturing surface flinger dump logs using shell command "adb shell dumpsys SurfaceFlinger"[3]. Following are dumpsys output for Surface Flinger in both cases.

- *Extract of Surface flinger output with surface view*

```
Visible layers (count = 6)

+ Layer 0xb8057b90 (SurfaceView) id=64
    Region transparentRegion (this=0xb8057d80, count=1)
    [ 0,  0,  0,  0]
    Region visibleRegion (this=0xb8057b98, count=1)
    [ 0,  75, 1080, 1920]

+Layer0xb802ac68
(com.example.withsurfaceview/com.example.withsurfaceview.MainActivity)
id=63
    Region transparentRegion (this=0xb802ae58, count=1)
    [ 0, 121, 1080, 1920]
    Region visibleRegion (this=0xb802ac70, count=1)
    [ 0,  75, 1080, 1920]
```

The output clearly shows two layers. First is Surface View as it has its own surface, second is application's main Activity which can be identified by the package name provided in AndroidManifest.xml file. The transparent area in Main Activity corresponds to size of the Surface View. These readings are in line with the fact that Surface View requests transparency.

- *Extract of Surface flinger output without surface view*

```
Visible layers (count = 5)

+Layer0xb805a398
(com.example.withoutsurfaceview/com.example.withoutsurfaceview.MainActi
vity) id=69
    Region transparentRegion (this=0xb805a588, count=1)
    [ 0,  0,  0,  0]
    Region visibleRegion (this=0xb805a3a0, count=1)
    [ 0,  75, 1080, 1920]
```

The surface flinger output shown above clearly states that there is only one surface present, which corresponds to the main window's surface.

VI. PROS AND CONS OF REMOVING SURFACE VIEW

As mentioned earlier, our proposed approach of using window's surface directly in order to optimize the power consumption of graphics intensive applications is good for applications which require full screen drawing. As we have confirmed from the experiments, our approach uses less power when compared to the surface view approach for the same application.

However, there are certain disadvantages to using our approach. As taking window's surface bypasses the view hierarchy, it is difficult for the application to add some Android widgets such as buttons, text edit or other touch responding widgets.

A typical hierarchy has been shown in fig 3. Fig 4 shows the view hierarchy after capturing the window's surface for drawing. As shown in the figure, the view hierarchy is actually cut and view root no longer receives any user events. This is one of the disadvantages of this approach.

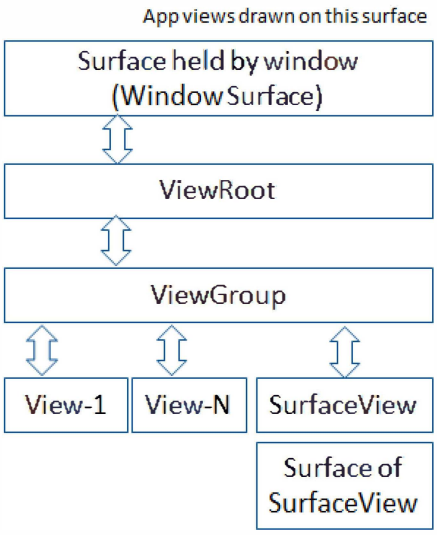

Fig. 3. The view hierarchy in a typical Android application.

Window Manager is responsible for maintaining all the surfaces. Since only one surface is being utilized for both rendering and Android views, the management of these views would be difficult, in comparison to the surface view approach. This is another disadvantage of our approach. In the following section, we mention a few methods to overcome the disadvantages while using our proposed approach.

VII. POSSIBLE WAYS FOR RECEIVING USER INPUTS IN CHILD VIEWS WHERE VIEW HIERARCHY HAS BEEN BYPASSED.

Since view management is difficult in our approach, at the same time the number of views is limited, we provide some methods to overcome the limitation of view management.

Once the window's surface is taken, the following actions can be done to add the remaining widgets:

- Add the widgets in separate window such as dialogs. In case the application has very few standard Android widgets, then it is good to create a separate layout and inflate in a separate dialog. Such a dialog can be shown and hidden as per the need.
- Add the widgets directly via window manager. If views/layouts are added directly via window manager the touch events should be delegated from main activity of application to such views for responding to user actions, due to the fact that the view hierarchy is bypassed and programmer has to take care of these events. This delegation will require more processing and thus slightly more power consumption especially for faster gestures such as pinch zoom and scroll. So before adopting a particular solution, it is recommended that the application requirements are analyzed thoroughly along with its user interface.

VIII. CONCLUSION AND FUTURE WORK

We performed an analysis of our approach to reduce the power consumption of graphics intensive Android applications by reducing an additional layer or surface introduced by Surface View. Our motive was to enable researchers to find out ways using only software to reduce the power consumption of an application. Our solution was limited to the applications which have higher graphics rendering requirements in comparison with the applications which have more static standard Android views and widgets.

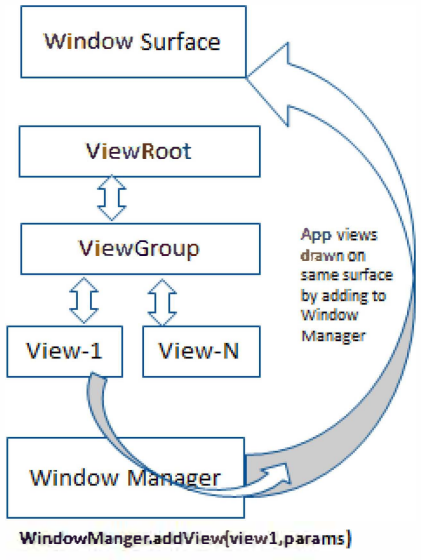

Fig. 4. Showing a view via window manager.

The power is consumed in hardware components which are actuated and utilized by software. In future, we will study ways to reduce the power consumption by optimization of hardware components as well.

ACKNOWLEDGMENT

The authors would like to thanks Nagaraju Yendeti from applications team, Harshala and Darshan Venu from testing team, of Samsung Research Institute Bangalore, India for providing support in this research.

REFERENCES

[1] http://developer.android.com/reference/android/view/SurfaceView.html

[2] http://developer.android.com/reference/android/view/Window.html

[3] http://developer.android.com/tools/index.html

[4] Power Monitor equipment manual www.msoon.com

[5] A. Carroll, G. Heiser, "An analysis of power consumption in a smart phone", Proc. USENIX Annual Technical Conference, 2010

[6] N. Thiagarajan, G. Aggarwal et al, "Who killed my battery?: analyzing mobile browser energy consumption", Proc. WWW' 12, 21st International Conference on World Wide Web, 2012

[7] "Bringing High-End Graphics to Handheld Devices", NVIDIA Whitepaper, 2011