Optimizing Power Consumption of Mobile Games

Yu Yan[1,2], Songtao He[1,3], Yunxin Liu[1], and Longbo Huang[2]

[1]Microsoft Research, Beijing, China

[2]Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China

[3]University of Science and Technology of China, Hefei, China

Abstract

In this paper we aim to optimize power consumption of mobile games without compromising user experience. We study the behavior of 40 mobile games on a smartphone and identify two power-inefficient issues: 1) fixed high frame rate that consumes a high power but brings negligible extra benefits to user experience when the screen content does not change rapidly or stays nearly static, and 2) high overdraw rate—the same pixels are drawn for multiple times and thus wastes energy. We report the measurement results of our study and explore possible solutions to mitigate these two issues. In particular, for the first issue, we have implemented a prototype to enable dynamic frame rate scaling that is able to reduce the frame rate to save power based on how fast the game content changes. A lower frame rate is used when the game content does not change fast and thus user-perceived experience is retained. Preliminary experimental results show that our approach is promising.

Categories and Subject Descriptors

I.3.4 [**Computer Graphics**]: Graphics Utilities—*Software support*

General Terms

Experiments; Measurement; Performance

Keywords

Smartphone; GPU; Power Consumption; Mobile Game

1. INTRODUCTION

The flourish of mobile game industry in recent years benefits a lot from the increasingly powerful GPU carried by smartphones. Smartphones are now able to display images more livingly and quickly, but can also be very power consuming for the sake of the heavy workload brought to GPU. Mochocki et al. [8] show that based on some assumptions on the growth rate of resolution and frame rate, the power demand on graphics processing of future mobile devices will not be satisfied if only relying on the development of battery technology. Therefore, the research in improving mobile power efficiency is extremely necessary, especially for the mobile games given that there are rare power management studies in this field [5].

HotPower'15, October 04-07, 2015, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/2818613.2818746

In this paper, we seek for opportunities to reduce power consumption of mobile games. In particular, we aim to not compromise user experience. To this end, we study the behavior of 40 typical Android games from all kinds of game categories. Based on the measurement results from the study, we identify two issues that lead to severe power inefficiency. The first one is that a fixed high frame rate is always used to render game content regardless of game states. Such a high frame rate is unnecessary when the game content does not change rapidly or stays nearly static. Thus, there are opportunities to reduce the frame rate based on how fast the game content changes to save power, without compromising the user experience. The second issue is high overdraw rate: the same pixels are re-drawn for multiple times, wasting energy but bringing zero user-perceivable benefits. Consequently, reducing overdraw provides more power-optimization opportunities without compromising user experience.

We report our measurement results and describe the two power-inefficient issues in Section 2. We then investigate how to mitigate the two issues in Section 3. In particular, we have built a prototype to enable dynamic frame rate scaling (DFRS) [9] that reduces the frame rate when the game content does not change rapidly to save power. Preliminary experimental results show that significant power savings can be achieved when games are idle. We survey related work in Section 4 and we conclude and discuss future work in Section 5.

2. POWER-INEFFICIENT ISSUES

We choose 40 popular Android games from 12 different categories, such as action, puzzle, racing, etc., to study their behavior. We focus on two metrics 1) frame rate: frames per second (FPS) at which the game keeps rendering; 2) average overdraw rate: the average times that all pixels are painted over *again* (so the average overdraw can be 0, meaning that all pixels are painted only once).

We pay special attention to the frame rate in the game idle state that is the state that a game will typically enter if the user does not touch the screen for a while, say 3 seconds. We also care about the drawing order of multiple graphics layers in an attempt to study its influence on the overdraw. For example, the "back to front" order represents that the game draws the background first, then objects far away, and the front most objects last.

Category	Game Name	Game Idle State	FPS	Draw Order	Avg. Overdraw
Action	Temple Run 2	Dynamic	60	Back to Front	1.25x
Strategy	3D Chess Game	Nearly Static	60	Back to Front	2.16x
Arcade	Angry Birds	Nearly Static	60	Back to Front	3.18x
Casual	Plane Parking 3D	Nearly Static	60	Front to Back	3.09x
Racing	Angry Birds Go!	Dynamic	30	Back to Front	74.55x
Cards	Solitaire	Totally Static	60	Back to Front	61.18x

Table 1: Example Measurement Results of Various Game Categories.

	Game Idle State			FPS		Draw Order		Avg. Overdraw		
	Totally Static	Nearly Static	Dynamic	30	60	B2F	F2B	Min	Max	Mean
Obs	8	21	11	5	35	39	1	0.3x	157.9x	15.5x

Table 2: Statistics Results of the 40 Games. B2F means back to front and F2B means front to back.

We conduct the study on a Nexus 5 smartphone. Due to space limit, we show a subset of results in Table 1 but the entire statistical result in Table 2.

2.1 Fixed High Frame Rate

The study result indicates that all the 40 Android games refresh at a fixed frame rate, with 12.5% (or 5) at 30 FPS and 87.5% (or 35) at 60 FPS, no matter what the game state is. However, in some game states where the screen does not change rapidly, a high frame rate (e.g., 60 FPS) can be a waste of power with no extra benefits to user experience.

Statistics also suggest that 72.5% (or 29) of the games will enter a game state with frames totally (20% or 8) or nearly (52.5% or 21) static if there is no touch event from users for a while. At this time, the game is typically playing a series of pictures repeatedly, waiting for players to operate. Due to the slight changes in the frames, there are less demands for a high frame rate, making the high frame rate unnecessary. If a system could switch between different frame rates efficiently whenever it is necessary, the mobile power efficiency is expected to be improved significantly [11, 9].

Therefore, motivated by this optimization opportunity, we develop a preliminary DFRS system and present the details in Section 3.

2.2 High Average Overdraw

Overdraw is the issue that the same pixels are drawn for multiple times. It is usually caused by rendering multiple graphics layers. Our study result indicates that the issue is indeed significant in mobile games. The mean of average overdraw of the 40 games is as high as 15.5x, and the maximum is even up to 157.9x. It means that the same pixels are re-drawn over and over again for many times. As the last drawing may make all the previous drawings invisible to users, the high overdraw rate leads to a large amount of energy waste. A good rule of thumb for the target maximum overdraw is 2x [6], but only 47.5% (or 19) of the 40 games can achieve this 2x target.

This high-overdraw issue is caused by the improper draw order of *back to front*. This issue can be avoided by changing to the opposite draw order of *front to back* because modern GPUs support the optimization of not drawing the parts of a layer overlapped by previously drawn layers. However, even though this programming practice is well-known, there are amazingly 97.5% (or 39) of the 40 games that do not follow the right draw order of front to back, leading to unnecessary and significant overdraw issue.

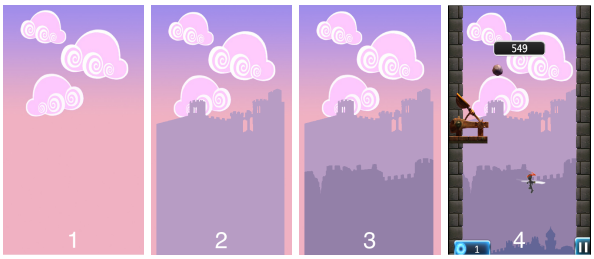

Figure 1: NinJump: an example of improper draw order with extremely high overdraw.

Figure 1, as an example, is a popular Android game called NinJump, whose average overdraw is as high as 157.9x. We have traced the OpenGL ES calls and observed that this game drew objects almost in a completely wrong order: entirely from back to front. From the parameters of the calls of `glDrawElements()`, we found that the game drew 1,002 triangles from picture 1 to picture 2, and 582 triangles from picture 2 to picture 3, respectively, in Figure 1. These triangles were all painted over the top of others, and thus caused the very high average overdraw.

Another example is shown in Figure 2, showing the improper drawing habit of game developers: draw the background first, then characters, and the widgets (buttons, clock, scoreboard, etc.) at the last. Even though the drawing order is natural and easy to program, it results in unnecessary high power consumption. We discuss how to mitigate this issue in Section 3.

3. MITIGATING THE POWER ISSUES

In this section we describe our preliminary efforts on mitigating the two power-inefficient issues.

3.1 Dynamic Frame Rate Scaling (DFRS)

Motivated by our study, we propose to enable DFRS to address the fixed-high-frame-rate issue. Instead of always using a fixed high frame rate to render the content of a game, DFRS dynamically adjusts the rendering frame rate based on the states of the game. When the game content changes slowly or becomes static, DFRS reduces the frame rate to save power.

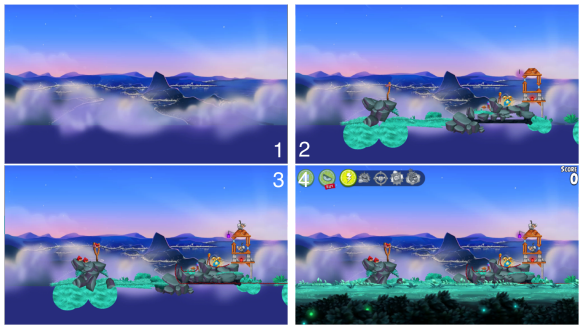

Figure 2: Angry Bird: another example of improper draw order, to show the improper draw habit of developers.

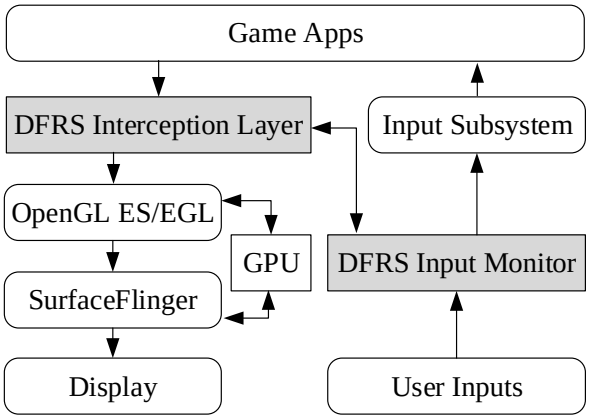

Figure 3: DFRS system architecture on Android.

Implementation. We have implemented a preliminary DFRS system on Android. Our implementation works as a system-level service and is transparent from individual game apps. Thus, it does not require any changes from games and is able to support any existing games.

As shown in Figure 3, our implementation has two main components. The first one is the *DFRS Interception Layer*, lying between game apps and the OpenGL ES/EGL layer. It hijacks the OpenGL ES API calls through binary rewriting to control the rendering frame rate. To reduce the frame rate, it skips the rendering of certain frames. For example, to reduce the frame rate from the default 60 FPS to 30 FPS, this DFRS Interception Layer skips one frame for every two frames. Skipping a frame is done by intercepting all the OpenGL ES draw functions (in the form of `glDraw*()`). We force each `glDraw*()` call to return, without generating any GPU drawing workload. As a result, we reduce GPU computation and thus save power.

It is worth noting that skipping frames may also be done by simply ignoring the VSYNC signals generated by the operating system. We do not take this simple approach because it is hard to be extended to support flexible control of frame rendering. For example, instead of completely skipping a frame, we may render the frame at a low resolution for a better trade-off between power saving and user experience: we save less power but provide better user experience. This kind of flexibility can be achieved by our API-interception-

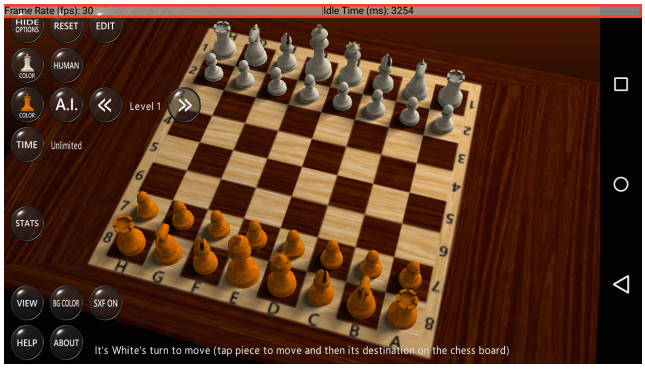

Figure 4: In the red rectangle on the top, it shows the current rendering frame rate and how long time the Chess game has been in idle.

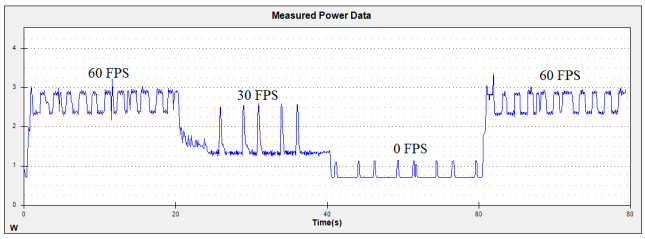

Figure 5: Power trace of a 3D Chess game with different frame rates.

based approach but not by the ignoring-VSYNC approach.

The second main component of our DFRS system is the *DFRS Input Monitor*. It hooks to the Android input subsystem to monitor all the touch events from a user. As a result, we can decide how long time a game has been in idle (i.e., no touch inputs received from the user). According to the findings of our study in Section 2, when a game becomes idle, we may reduce the frame rate of the game to save power. To do it, the DFRS Input Monitor sends a message to ask the DFRS Interception Layer to reduce the frame rate. To ensure good user experience, when a new user touch input is detected, the DFRS Input Monitor sends another message to notify the DFRS Interception Layer to increase the frame rate back.

Despite that we only focus on the game-idle case in this paper, the DFRS Interception Layer is general enough to support other cases. For example, if we develop a new component to detect how fast the content of a game changes, we may reuse the same DFRS Interception Layer to control the frame rate according to the output of such a new component.

In addition, we also provide a UI component to show the current rendering frame rate of a game and how long time the game has been in an idle state. As shown in Figure 4, the UI overlays on top of the UI of the game.

Preliminary experimental results. We conducted experiments on a Nexus 5 smartphone for a preliminary evaluation on our implementation. We used a Monsoon power monitor to measure the system power consumption of the phone.

Game Name	Power (mW)		Saving (%)
	60 FPS	30 FPS	
3D Chess Game	3066.3	1357.3	55.7
Bike Racing 3D	3138.3	1760.4	43.9
Archery Master 3D	2606.4	1606.5	38.4
Crazy Driver 3D	2596.8	1672.6	35.6
FarmVille 2	2500.7	1676.7	32.9
Subway Surfers	2011.8	1392.2	30.8
Cut the Rope 2	1709.6	1224.5	28.4
My Talking Tom	1877.4	1373.2	26.9
Temple Run 2	1761.5	1324.2	24.8
Angry Birds	1370.6	1150.8	16.0

Table 3: System power saving of 10 games when the frame rate is reduced from 60 FPS to 30 FPS.

Figure 5 shows a power trace in playing a 3D Chess game when the frame rate is changed from 60 FPS to 30 FPS then to 0 FPS and then back to 60 FPS. It shows that the system power consumption is significantly reduced when the frame rate is changed from 60 FPS to 30 FPS, due to the reduced GPU workload. However, when the frame rate is further reduced to 0 FPS, less extra power saving is achieved. This is because that at 30 FPS, the GPU only contributes a small part of the total system power consumption. Even though we reduce the GPU computation to zero, the power consumption of other hardware components such as CPU and screen are still there.

Table 3 shows how much the average system power consumption can be reduced for 10 popular Android games, when the frame rate is reduced from 60 FPS to 30 FPS. On average, the power saving is 33.3%, ranging from 16.0% to 55.7%. The power saving of 3D games is more significant, as high as 43.4% on average. In particular, for a game like the 3D Chess game, the game may stay in idle for a large percent of the total play time because users usually spend much time on thinking how to make the next move. Thus, the total energy saving in playing such a game can be significant, even though we only reduce the frame rate during game idle time.

In addition, we conducted a user study with 20 experienced Android game players. We chose several games, randomly ran them with and without our DFRS system enabled, and asked the subjects to determine in which case DFRS was enabled. The result shows that 90% of the subjects could not tell the difference.

3.2 Reducing Overdraw

To mitigate the high-overdraw issue, a simple solution is requiring game developers to strictly follow certain programming guidelines. As developers exactly know how many graphics layers their games have and how the layers overlap with each other, they have the best knowledge on how to minimize the overdraw of their games. However, as shown by our study, most games have a high overdraw rate, demonstrating that it is very hard for developers to follow the guidelines in practice, even very basic ones like draw order.

As we cannot trust developers to minimize overdraw, it is desirable to develop a system tool to help mitigate the overdraw issue. Ideally, the tool should work with existing games without requiring any modifications from games. The tool may analyze the graphics-rendering behavior of a game and automatically change its behavior to reduce overdraw. However, it is very challenging to do so without knowing the internal logics of games: 1) it requires deciding how many graphics layers a game has and the depth information of the layers and objects; 2) sometimes it is hard to define the order between different objects and layers due to mutual coverage and because some objects and layers may be transparent; 3) it may be computationally expensive to retrieve the above information and change game behavior accordingly. Those are all open research questions and we plan to investigate how much we could do towards this direction.

4. RELATED WORK

Pathania et al. [11] and Nixon et al. [9] have shown that reducing frame rate leads to significant power savings. Without providing a working system, they are motivational to us. However, blindly lowering frame rate will affect the user experience of game players [4], and thus adjusting frame rate must be done carefully.

Overdraw is also a known issue [7]. Olson et al. [10] report that the average overdraw of 2x to 3x in 3D games is common, but our study shows that the issue becomes more severe in mobile games. Drawing objects in the order of front to back can help mitigate this issue [1] but it is rarely followed by developers in practice. The tile-based rendering [3] has been proposed on the PowerVR chipsets [2]. It reduces overdraw but leads to more external traffic for geometrical data at the same time.

5. CONCLUSION AND FUTURE WORK

In this paper we have studied the behavior of 40 Android games and identified two power-inefficient problems: fixed high frame rate that is unnecessary when games are idle or slow; and high overdraw caused by improper draw order in graphics rendering. Although these two issues are not completely new, we show that they have not been solved in modern mobile games yet. We report the measurement results and explore the possible solutions.

For future work, we will continue to improve our DFRS system, including making the frame rate adaptive to the content-changing speed of games and conducting more comprehensive evaluations. We will also study how to build a system tool to reduce overdraw of mobile games without requiring any modifications from game apps themselves.

6. REFERENCES

[1] Mali GPU. Application Optimization Guide.

[2] VR-PowerVR. 3D Graphical Processing, Nov. 14, 2000.

[3] Antochi, I., Juurlink, B., Vassiliadis, S., and Liuha, P. Memory bandwidth requirements of tile-based rendering. In *Computer Systems: Architectures, Modeling, and Simulation*. Springer, 2004, pp. 323–332.

[4] Claypool, M., Claypool, K., and Damaa, F. The effects of frame rate and resolution on users playing first person shooter games. In *Electronic Imaging 2006* (2006), International Society for Optics and Photonics, pp. 607101–607101.

[5] Dietrich, B., and Chakraborty, S. Forget the battery, let's play games! In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on* (2014), IEEE, pp. 1–8.

[6] Guy, R. Android performance case study. `http://www.curious-creature.com/docs/android-performance-case-study-1.html`.

[7] McCaffrey, J. Exploring mobile vs. desktop opengl performance 24. *OpenGL Insights 337* (2012), 341.

[8] Mochocki, B., Lahiri, K., and Cadambi, S. Power analysis of mobile 3d graphics. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings* (2006), European Design and Automation Association, pp. 502–507.

[9] Nixon, K. W., Chen, X., Zhou, H., Liu, Y., and Chen, Y. Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling. In *Proceedings of the 6th USENIX conference on Power-Aware Computing and Systems* (2014), USENIX Association, pp. 5–5.

[10] Olson, T. J. Hardware 3d graphics acceleration for mobile devices. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on* (2008), IEEE, pp. 5344–5347.

[11] Pathania, A., Jiao, Q., Prakash, A., and Mitra, T. Integrated cpu-gpu power management for 3d mobile games. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014), IEEE, pp. 1–6.