RAVEN : Perception-aware Optimization of Power Consumption for Mobile Games

Chanyou Hwang*
chanyou@nclab.kaist.ac.kr
School of Computing - KAIST

Saumay Pushp*
saumay@nclab.kaist.ac.kr
School of Computing - KAIST

Changyoung Koh
changyoung@nclab.kaist.ac.kr
School of Computing - KAIST

Jungpil Yoon
jdsp@nclab.kaist.ac.kr
School of Computing - KAIST

Yunxin Liu
yunxin.liu@microsoft.com
Microsoft Research

Seungpyo Choi
spchoi@nclab.kaist.ac.kr
School of Computing - KAIST

Junehwa Song
junesong@nclab.kaist.ac.kr
School of Computing - KAIST

ABSTRACT

High-end mobile GPUs are now becoming an integral part of mobile devices. However, a mobile GPU constitutes a major portion of power consumption on the devices, and mobile games top as the most popular class of graphics applications. This paper presents the design and implementation of RAVEN, a novel, on-the-fly frame rate scaling system for mobile gaming applications. RAVEN utilizes human visual perception of graphics change to opportunistically achieve power saving without degrading user experiences. The system develops a light-weight frame comparison technique to measure and predict perception-aware frame similarity. It also builds a low resolution virtual display which clones the device screen for performing similarity measurement at a low-power cost. It is able to work on an existing commercial smartphone and support applications from app stores without any modifications. It has been implemented on Nexus 5X, and its performance has been measured with 13 games. The system effectively reduces the overall power consumption of mobile devices while maintaining satisfactory user experiences. The power consumption is reduced by 21.78% on average and up to 34.74%.

CCS CONCEPTS

• **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; • **Computing methodologies** → *Perception*;

KEYWORDS

Perception-aware processing; Energy-efficient processing; Frame-rate scaling; Mobile games; Experiments; Measurement; Mobile systems

*Co-primary authors, order chosen alphabetically.

MobiCom '17, October 16-20, 2017, Snowbird, UT, USA.

ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/.1145/3117811.3117841

1 INTRODUCTION

Over the past three years, the processing power of mobile GPUs has increased more than double [5][6]. In response, mobile game application developers have begun to capitalize on the modern GPU's processing power to make their game graphics look crispier and more appealing. However, those visually appealing mobile games come at a high energy cost. Each high-resolution frame consumes a large amount of system resources, especially the GPU computation, and results in high system power consumption.

Mobile games top as the most popular class of graphics applications (apps) on smartphones [18] and are frequently the major sources of energy drain. Depending upon the amount of graphics contents inside a game app, the GPU power consumption increases almost linearly. However, few power management studies have been performed for mobile games [23]. In this paper, we explore the potential to optimally use limited computing resources and energy by delving into the computation process from the perspective of human perception, i.e., game player's perception in this case. We specifically look into the rendering process of gaming apps considering perceptual sensitivity of human, and re-organize the process, emphasizing sensitive moments while de-emphasizing the rest.

Upon the above idea, we present RAVEN[1], a novel system for scaling the rate of rendering frames which leverages human visual perception. RAVEN introduces perception-aware scaling of frame-rendering rates (PAS) while a user is engaged in a game-play. In detail, PAS reduces an application's rate of rendering game frames whenever succeeding frames are predicted to be perceptually similar enough. For this, we set a side channel to track the rendered frame sequences which can tailor user's perception with graphics changes in a game-play. In this way, RAVEN opportunistically reduces power consumption of a GPU. Existing methods to measure user's perception of visual changes such as structural similarity (SSIM) [33] are computationally expensive and not adequate for a system to maintain quality user experiences during game-play. We leverage **YUV** color space [12] to develop a simple, but effective way to track user's perception on-the-fly.

There have been efforts to reduce battery consumption in playing games or other apps on mobile systems. User configuration-based

[1] RAVEN is a DC comics character, from whom we would like to signify the ability to 'sense and alter'.

approaches have been adopted in many systems, e.g., Samsung-Game-Tuner [9]. They allow users to pre-set manually parameters related to display resolutions or frame rates. They could lead to energy savings; however, it is difficult to match consistently users' desires for quality experiences which potentially require different configurations depending on game contents such as scene changes or graphics changes during plays. Approaches for user interaction-based rate control have also been attempted [27][34]. They make systems respond with high resolutions or frame rates upon user interactions assuming that users pay higher attention at those moments. However, moments of high visual attention do not always overlap with those of user interactions. In addition, user interactions frequently come as responses to game flows intended in a game design. To provide a truly immersive game-play and energy saving, the system needs to consider visual fidelity, i.e., the game-play should be seamless to the eyes of a user.

There are several challenges in building the RAVEN system. First, it is important to measure the perceptual similarity efficiently. As mentioned, inferring the perceptual similarity with existing methods incur high computational cost. The system should be able to approximate such computation-intensive methods at a very low cost. Second, while saving energy, degradation of user experiences caused by the system should be minimal. Third, mobile gaming apps and GPU drivers are closed source. This asks for building a system that is transparent to the apps, and avoids low-level system optimization.

To overcome the above challenges, we have developed an energy-efficient method to measure perceptual similarity exploiting the susceptibility of human eyes to color difference. The proposed method leverages the difference in the luminance components (in **YUV** color space) between frames. We extensively evaluated the method by comparing it with the well-known SSIM method under various settings, and show that the proposed luminance-based method can efficiently measure perceptual similarity with low computational costs. Also, we build in the RAVEN system a virtual display, cloned from the main screen of the device but with a much lower resolution. The system reads the graphical contents of the small virtual display for the similarity measurement with low computational and energy overhead. In addition, RAVEN realizes the idea of perception-aware rate scaling by interfering and impeding the rendering loop of a game app. For frames that are perceptually similar to the current frame, the loop skips its operation, i.e., rendering of those frames. Instead, a delay for the amount of corresponding frames' periods, is inserted to the loop. Afterwards, it jumps to the next frame to be rendered. In this way, the rate of frame rendering is effectively scaled. The RAVEN system is designed and implemented by customizing an Android framework as Android's system services, which makes itself transparent to apps and avoids the complication of accessing closed source.

We have implemented the RAVEN system on a Nexus 5X smartphone. We conduct comprehensive experiments and a user study to evaluate our implementation with 13 real mobile games. Experimental results show that all 13 games can run successfully with per-frame, real-time perception-aware scaling of frame rendering rates. The energy per game session can be reduced by 21.78% in average and up to 34.74% while maintaining quality user experiences. To the best of our knowledge, RAVEN is the first system designed

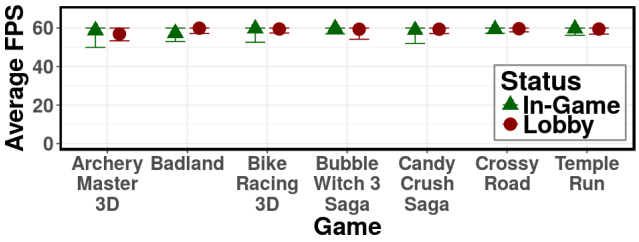


Figure 1: Frame rates in a game-play and while standing in a game lobby.

and implemented to achieve rate scaling and energy savings based on perceptual similarity for mobile games. The main contributions of this paper are:

- We present a novel rate-scaling technique for rendering app frames based on the human visual perception to game's graphics to reduce energy consumption on-the-fly.
- We explore and develop a novel, energy-efficient perceptual similarity measurement method leveraging the differences in luminance values between frames.
- We present the design and implementation of the RAVEN system with its application transparency.
- We adopt a user-tunable approach which provides an on-demand access to tune power drain and user experience.
- We conduct experiments and a user study to confirm the effectiveness of our system for quality user experience and power saving.

The rest of this paper is organized as follows. Section 2 motivates the need of RAVEN and set our goals. It also introduces the graphics processing flow of Android. Section 3 gives an overview of RAVEN's system architecture and defines its key components. Section 4 and Section 5 present how to perform frame comparison for regulating the applications' rates of rendering frames, and our rationale behind building a low-power virtual display. Section 6 describes some details on RAVEN's implementation and Section 7 reports the evaluation results. Section 8 surveys the related work and Section 9 discusses the limitations of our system, while Section 10 concludes the paper.

2 MOTIVATION AND BACKGROUND

Motivation. A high frame rate is beneficial if it provides a good user experience. However, we have observed that a high frame rate may not always be necessary in mobile games. We have played a set of mobile games (see Figure 1) and measured their frame rates. We provide description on the types of the games used in our study in Section 7. We have intercepted `eglSwapBuffer()` function in OpenGL ES/EGL [13] which gives accurate numbers on how fast games generate new frames. In a game-play, the game may enter different states, depending on user interactions and game stages. We focus on two cases: "In-Game" where we play a game actively and graphics contents on the screen change quickly; and "Lobby" where a game is in a session-transition stage or shows setting menus and thus we perceive few or zero changes on the screen.

Figure 1 presents the results. It shows that no matter the states of the games, they always ran at a high frame rate of 60 FPS. Even in the

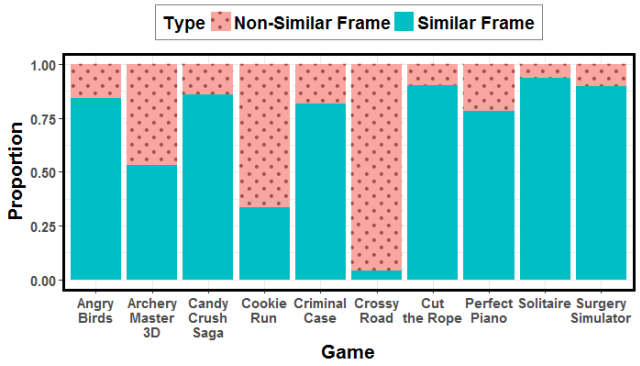


Figure 2: Proportions of similar and non-similar adjacent frames in different gaming apps.

"Lobby" case where the screen showed relatively-static contents, they still kept generating 60 new frames per second. The same observation was also reported in a previous study [34].

We have further studied similarity between consecutive frames in playing games. We have employed SSIM [33] to calculate the similarity between frames. A SSIM index is a number between 0 and 1, and a value higher than 0.97 is considered as a strong level of similarity in mobile games [22]. We have chosen 0.975 as the threshold to decide whether two frames are similar or not. Figure 2 shows how many frames are similar in playing the selected games. On average, we have recorded 15,035 frames for each game. That is, the average game-play time was 4.2 minutes. Up to 93.5% of the total frames generated by the games are indeed very similar. This strong similarity existed not only for "Lobby" frames but also "In-Game" frames.

The observations show that 1) the game apps maintain high frame rates irrespective of their game states, and 2) large portions of consecutive frames of the game apps show strong similarity. These findings give an opportunity to save energy without compromising user experiences by regulating redundant frame renderings. Specifically, when consecutive frames are highly similar, we may bypass rendering the frames and reduce the power consumption. Thanks to strong frame similarity, the user experiences will not be compromised much. These findings motivate us to seek for a system to optimize power consumption of mobile games using perception-aware frame rate scaling.

Goals. We set the following goals and requirements in designing a target system: 1) Automatically change the rates of rendering application frames on-the-fly without requiring any user efforts; 2) Be able to run on commercial smartphones without requiring any special hardware or re-building a device driver; 3) Support legacy apps without requiring any app-side modifications; 4) Effectively achieve significant power savings.

2.1 Graphics Processing on Android

We briefly describe some background information on Android graphics operations and OpenGL ES/EGL [13] related to our design. Android applications use OpenGL ES API [17] to leverage GPU acceleration for rendering their graphics contents and to draw their

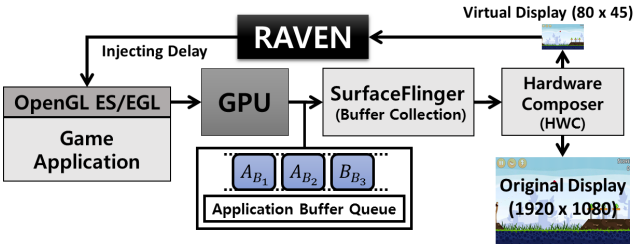


Figure 3: Flow of graphics processing on a Nexus 5X running Android 6.0.1. OpenGL ES/EGL is paddled by RAVEN, and the frame-rendering process of GPU is regulated.

User Interface (UI) objects. Rendered results are stored in a **BufferQueue**[3], which consists of graphics buffers. Each app has its own, separate **BufferQueue**. The Android system uses a system service called **SurfaceFlinger**[10] to coordinate all rendered graphics contents of foreground apps and the Android UI subsystem. On a smartphone, the display driver typically refreshes the screen at 60 Hz and generates a **VSYNC** (Vertical Synchronization) signal for each frame period. Upon receiving a **VSYNC** signal, **SurfaceFlinger** collects the graphics buffers (from multiple apps) and sends them to **Hardware Composer**. Hardware Composer composes the final graphics contents which will be displayed on a device screen. A detailed flow of Android graphics processing is shown in Figure 3.

3 SYSTEM OVERVIEW

The idea of RAVEN can be described in high level with its major components and their interactions. The system consists of three major components which collectively perform scaling the rates of game-frame rendering (Figure 4): **FrameDiff Tracker** (F-Tracker), **Rate Regulator** (R-Regulator), and **Rate Injector** (R-Injector). The system works in a pipelined fashion. First, F-Tracker measures perceptual similarity between two recent frames at a low energy cost. Then, R-Regulator predicts the level of similarity between the current and the next frame(s). If the next frames are similar to the current one, it notifies R-Injector to limit frame-rendering rates by injecting some delay in a rendering loop and skip graphics processing for unnecessary frame(s). Presently, RAVEN is able to skip up-to the maximum of three frames, and thus, inflict frame-rate drop down to 15 FPS. The continuous regulation of the rates results in opportunistic energy savings. Figures 5 and 7 show three different system states of current implementation, i.e., No-skipping, 1-Skipping, and 3-Skipping states, and their transitions corresponding to different rates. Note that the system is not limited to the above and could be generalized to include other skipping states.

F-Tracker measures perceptual similarity between frames on-the-fly. To compute the perceptual similarity efficiently, the system takes a twofold strategy. First, RAVEN leverages a light-weight, low-resolution virtual display supported by Android system. Constructing a virtual display is lighter weighted than reading a display frame buffer as the latter involves a higher resolution (equivalent to the native resolution of the screen). Second, to measure perceptual similarity between frames, RAVEN utilizes an approximation of SSIM based on the **Y**(luminance) component in the **YUV** color space. SSIM is a well-known index to measure perceptual similarity, but computationally expensive and hard to compute for every

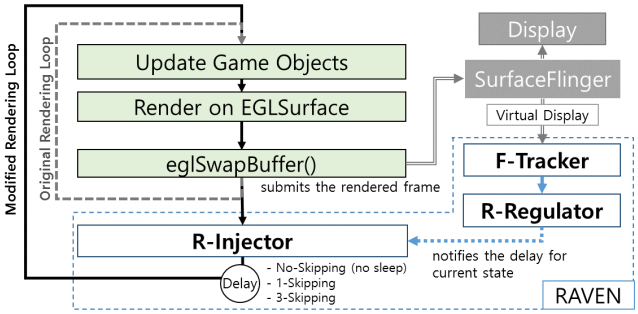


Figure 4: Modified rendering loop for delay injection and rate scaling. White-colored blocks are RAVEN's components.

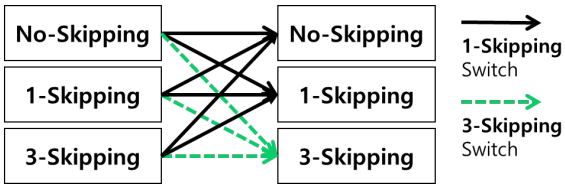


Figure 5: Different system states and their transitions in RAVEN: 1-Skipping skips a frame and results in 30 FPS, and 3-Skipping does three and results in 15 FPS.

frame at 60 FPS. In Section 4.1, we show the design rationale and the performance of our approximation method.

The system forces apps to skip perceptually similar frames by applying a proper frame rendering rate. Consider the following three frames (Section 4): the current frame (f_N), the frame which is expected to arrive after 16.7ms (f_{N+1}) , and the one, expected after 50ms (f_{N+3}). `R-Regulator` predicts the similarity scores of (f_N, f_{N+1}) and (f_N, f_{N+3}), and compares them with thresholds τ_1 and τ_3 (used as scaling factors), respectively. Each threshold stands for the minimum required similarity level for skipping f_{N+1} or f_{N+1}~f_{N+3}. If the predicted scores exceed their threshold values, the system applies the corresponding rate scaling by skipping the frames.

For rate scaling, RAVEN interferes and modifies the rendering loop of a gaming app (see Figure 4). Upon skipping decision of some frames, it inserts into the loop a delay for the duration of the corresponding frames' periods and have the system sleep for the duration without performing the rendering operations. The loop awakens after the delay and jumps to the next frame to render. This ensures the rate of frame rendering to be effectively scaled.

RAVEN is designed and implemented by customizing an Android framework, i.e., `FrameDiff Tracker` and `Rate Regulator` as Android's system services as well as `R-Injector` by adjusting the system function, `eglSwapBuffer`. The rendering loop in a mobile game runs in a separate thread from the main thread of the game. The operating system ensures that the rendering loop processes each frame to attain the maximum frame rate of 60 FPS. The RAVEN system successfully hooks into the rendering loop as above by customizing an Android framework, more specifically `eglSwapBuffer()` function is called in the rendering loop and swaps the graphical content inside `EGLSurface`'s buffer (an off-screen buffer

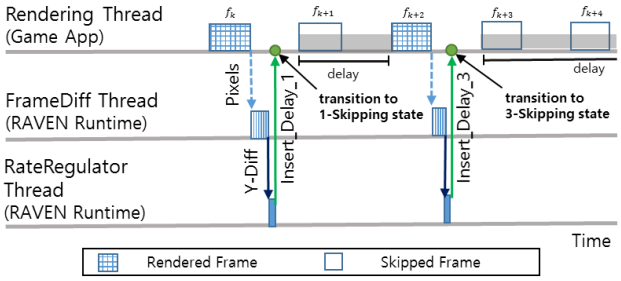


Figure 6: Flow of operations at a thread level.

which stores SurfaceView, SurfaceTexture etc. of an app). We modify the `eglSwapBuffer()` function in Android's EGL API and inject delay in this function through `R-Injector`. Thereafter, the operation of the loop is regulated with the inserted delay, and enforces the desired frame rate.

At the thread level, three threads play important roles in the operation of RAVEN: game app's rendering thread, RAVEN's `FrameDiff Tracker` thread, and `Rate Regulator` thread. Figure 6 presents how these threads communicate with each other to work together. At runtime, the rendering thread generates the pixels of the current frame. The `FrameDiff` thread reads the down-scaled pixels through a virtual display in a synchronous way, and compares **Y**–values of consecutive frames and sends the result to the `Rate Regulator` thread. Finally, the `Rate Regulator` thread calculates the appropriate delay to adjust the frame rendering rates, and then send the delay value to the rendering thread. The successful implementation of RAVEN based on the customization of an Android framework as above leads the system to be transparent to apps, enabling it to effectively support commercial gaming apps from app stores without any modifications.

4 PERCEPTUAL SIMILARITY AND REGULATION OF FRAME RENDERING RATES

The idea of regulating rendering rates without degrading user experiences is to skip rendering frames whenever adjacent frames are perceptually similar. However, without actually rendering a frame, we do not know how similar it is with the previous one, and thus, cannot decide whether to render it or not. Therefore, RAVEN compares the recently rendered frames, measures their perceptual similarity, and uses the result to decide whether to skip future one(s). Figure 7 depicts the prediction scheme as well as the rendering process on a sequence of frames.

The discussion in this section focuses on the case of developing two different skipping states capitalizing on four frames; however, the idea of perception-aware optimization (or rate scaling) could easily be generalized in diverse ways. Importantly, the similarity among consecutive frames is prevalent, and there is a broad spectrum of potential to utilize such similarity. Particularly, it would be feasible to develop many different levels of frame skipping transitions and rate regulations.

Consider the four frames, f_{N-k}, f_N, f_{N+1}, and f_{N+3}, where f_{N-k} is the previously rendered frame, f_N is the currently rendered one, and f_{N+1} and f_{N+3} are the future frames to be rendered.

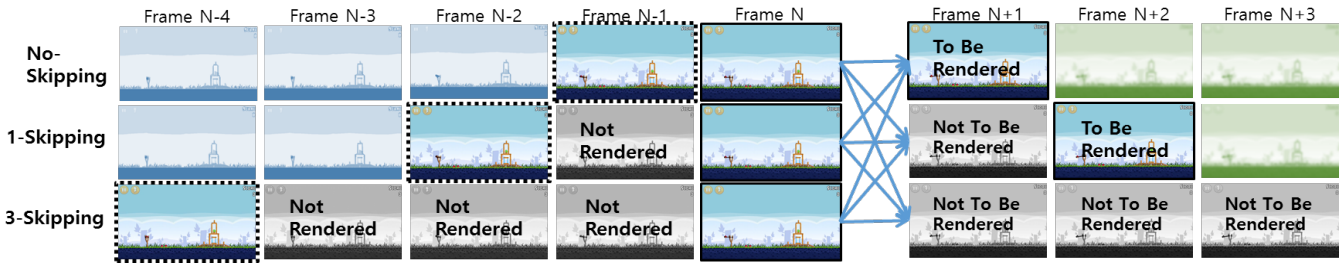


Figure 7: Process of rendering a sequence of frames in RAVEN (Best viewed in color).

Without RAVEN, these frames will be sequentially rendered, however, with RAVEN, the system first predicts the perceptual similarity scores of the future frames, i.e., f_{N+1} and f_{N+3} from those of the previously rendered ones. If the result of prediction for f_{N+1} is sufficiently similar to f_N, the system skips the rendering of f_{N+1}. If f_{N+3} is also predicted to be similar, the system skips f_{N+1}, f_{N+2}, and f_{N+3}. The user-perceived experience is maintained as if the frame rate is still 60 FPS, even though the system has reduced the rates to 30 FPS or 15 FPS.

4.1 Building a Perceptual Similarity Score

To build a perceptual similarity score for frame comparison, we leverage ***Y–Difference*** (or ***Y–Diff***). **Y**–Diff is the difference in **Y**(luminance) values of two images in the **YUV** color space. It takes into account the human perception associated with perceiving brightness as human vision is sensitive to brightness changes. In the human visual system, a majority of visual information is conveyed by patterns of contrasts from its brightness changes[35]. Furthermore, the luminance is a major input for the human perception of motion[30]. Since people perceive a sequence of graphics changes as a motion, consecutive images are perceptually similar if people do not recognize any motions from the image frames. From this perspective, **Y**–Diff could be suitable for measuring perceptual similarity. Also, as we discuss later in Sections 4.2 and 5, **Y**–Diff is a good, energy-efficient approximation of SSIM.

Our primary interest in using **Y**–Diff is on its low overhead. Its computation requires $O(N)$ comparisons (N is the number of pixels) as well as a single matrix multiplication to transform a color space, from **RGB** to **YUV**. The overhead is low enough to compute it for every frame while matching the rate of 60 FPS. SSIM requires a relatively heavy computational cost. Regarding the limited computing resource of mobile devices, computing SSIM for every frame is impractical.

Why not grid-based comparison with RGB? A straightforward approach in measuring graphics difference between image frames would be to compare them in RGB space. However, as shown in Figure 8, when it comes to measuring structural similarities, SSIM shows a linear relationship with **Y**–Diffs while not with RGB distances.

Why not SSIM scores on a low-resolution virtual display of RAVEN? Computing SSIM indices of consecutive frames on a mobile environment is expensive from the view of both computation and energy costs. We measured the cost of calculating SSIM indices for every pair of frames on a Nexus 5X smartphone. Comparing frames

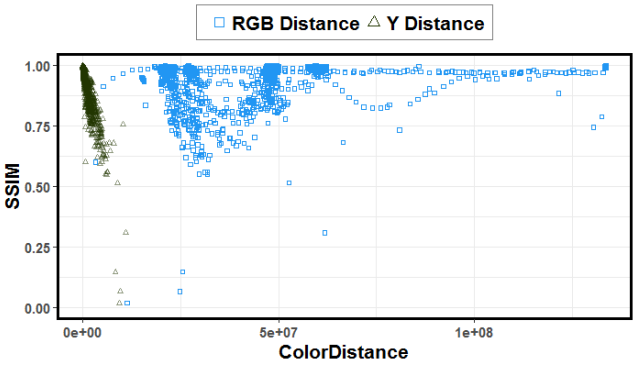


Figure 8: Comparison of RGB distances and Y distance with respect to SSIM.

with a low resolution of 192×108 pixels continuously took power drain of 332mW, and computation time of 87ms. This result shows that we could only calculate SSIM indices of up to 10 frames per second on single-thread settings, and the extra power consumption makes it impractical to save power.

Would it be reasonable to focus on luminance component? Chrominance components (U and V) may show dramatic changes while luminance does not change significantly. In such cases, the system's performance would get affected if only **Y**–Diff is used. To see if such cases occur frequently, we analyzed the videos recorded while playing Cookie Run and extracted **Y**–Diff, **U**–Diff, and **V**–Diff of consecutive frames. We observed that for the pairs of frames with small **Y**–Diff values (within the low 10%), 99.95% of them had also small **U**–Diff and 99.9%, small **V**–Diff values (and belonged to the low 20%). From the observation, it is not highly likely that abrupt changes occur in chrominance values when luminance show gradual changes. Therefore, it would be reasonable to focus on luminance in estimating perceptual similarity for energy-efficient processing.

4.1.1 *Using **Y**–Diff for perceptual similarity.*

We look into the performance of the approximation by **Y**–Diff in comparison to SSIM. For this, we calculate the correlation between the two.

Performing a fair comparison between **Y**–Diff and SSIM is difficult in the case of gaming apps. Game-plays are not reproducible because of their non-deterministic nature. Therefore, we take a record and replay based approach. We first recorded the screen using a screen recording application. Then, we used the OpenCV

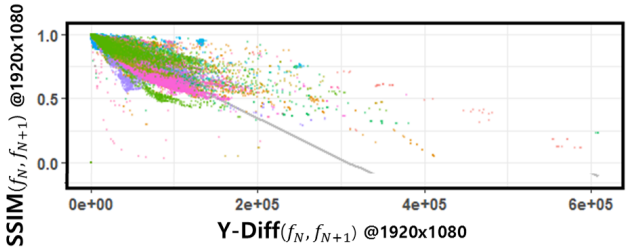


Figure 9: Correlation between Y−Diff and SSIM (Best viewed in color).

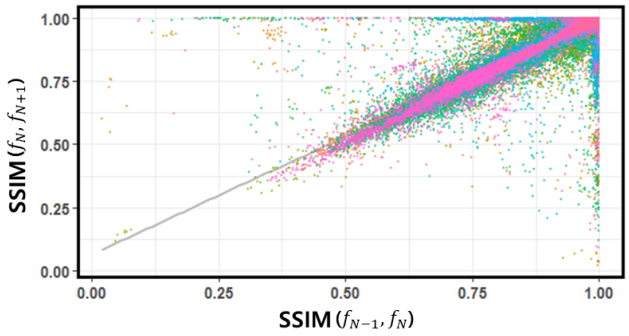


Figure 10: (Best viewed in color) Correlation between SSIM(f_{N-1}, f_N) and SSIM(f_N, f_{N+1}) (Pearson's correlation coefficient: 0.93). The correlation between SSIM(f_{N-1}, f_N) and SSIM(f_N, f_{N+3}) has also been explored (Pearson's correlation coefficient: 0.90), but the graph is omitted to save space.

library [8], and extracted each frame of a game-play. Thereafter, we iteratively calculated the **Y**−Diff and SSIM values for every frame f_N with respect to its previous four frames. In total, we analyzed 16 recorded plays of 13 games.

Figure 9 shows the relation between SSIM and **Y**−Diff values. Each color in the figure represents a game app. SSIM and **Y**−Diff values are strongly correlated and shows a linear relationship (Pearson's correlation coefficient of -0.926). We thus approximate SSIM with **Y**−Diff by linear regression. We denote the approximation as *Estimated Perceptual Similarity (EPS)*.

4.2 Estimating Frame Rendering Rates

Predicting perceptual similarity: To make decisions on whether to skip a future frame(s), we first look into the perceptual similarity of the current frame with the previously rendered one, and then predict the similarity with future one(s) to be rendered.

Figure 10 shows the relationship between the similarity scores of consecutive game frames from the 13 games. They show a linear relationship with a Pearson's correlation coefficient value of 0.93. This fact motivates us to use linear regression for predicting similarity scores between two frames. That is, the similarity score with a future frame in-line is predicted from that with a previous one. Let EPS(f_N, f_{N+l}) represent the prediction of perceptual similarity between frames f_N and f_{N+l}. It is basically predicted from **Y**−Diff

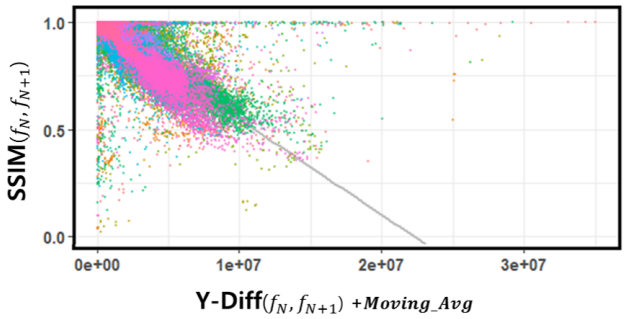


Figure 11: Correlation between SSIM(f_N, f_{N+1}) and the Y−Diff(f_{N-1}, f_N) + 'moving average' of Y−Diff (Best viewed in color).

of previous frames. Currently, the system implements two different levels of rate reduction, 30 FPS and 15 FPS, corresponding to 1-skipping and 3-skipping, respectively, and utilizes EPS(f_N, f_{N+1}) and EPS(f_N, f_{N+3}). To improve the performance of similarity prediction, it leverages the moving average of previous **Y**−Diff values to reflect their past history.

Figure 11 shows the correlation between the SSIM and the predicted similarity scores using both **Y**−Diff and the moving average of **Y**−Diff with the window size 10. The resulting equation of regression is:

$$\text{EPS}(f_N, f_{N+l}) = 1 - c_{kl_1} \times \text{D}_\text{Y}(f_{N-k}, f_N) - c_{kl_2} \times ma_w. \quad (1)$$

In the equation, the index k of the previous frame f_{N-k} depends on the current state, i.e., $k = 1$, 2 and 4 if the state is No-Skipping, 1-Skipping, and 3-Skipping, respectively. Also, there are two possible l values, i.e., 1 or 3, to predict EPS(f_N, f_{N+1}) or EPS(f_N, f_{N+3}). $\text{D}_\text{Y}(f_i, f_j)$ is the **Y**-Diff between f_i and f_j. ma_w is the moving average of the recent **Y**−Diff scores between consecutive frames over window size w. c_{kl_1} and c_{kl_2} are regression coefficients to predict EPS(f_N, f_{N+l}) with $\text{D}_\text{Y}(f_{N-k}, f_N)$. These values are obtained from the videos of the selected games.

The window size of moving average affects the performance of the regression. We compared R^2 (*R-squared*) values while varying the window sizes to 1, 2, 5, and 10 frames. R^2 is a measure of goodness of fit to the regression model. For each size, the R^2 values are obtained as 0.417, 0.467, 0.559, and 0.622, respectively. This result shows that the R^2 value increases with the window size. In the current study, the size is set to 10 frames, and finding the optimal size is left as a future work.

Determining state transitions: The system determines the state to transit, i.e., No-Skipping, 1-Skipping, or 3-Skipping, from the predictions of similarity scores, i.e., EPS(f_N, f_{N+1}) and EPS(f_N, f_{N+3}). It uses separate threshold values (or *scaling factors*), τ_1 and τ_3, to look for the opportunity to skip either one frame or three consecutive frames, respectively. The system compares EPS(f_N, f_{N+1}) and EPS(f_N, f_{N+3}) with respect to τ_1 and τ_3. If EPS(f_N, f_{N+1}) is not greater than τ_1, no frame skipping takes place, and the system shifts to No-Skipping state. If EPS(f_N, f_{N+1}) is greater than τ_1, it further checks EPS(f_N, f_{N+3}) with τ_3. If EPS(f_N, f_{N+3}) is also greater than

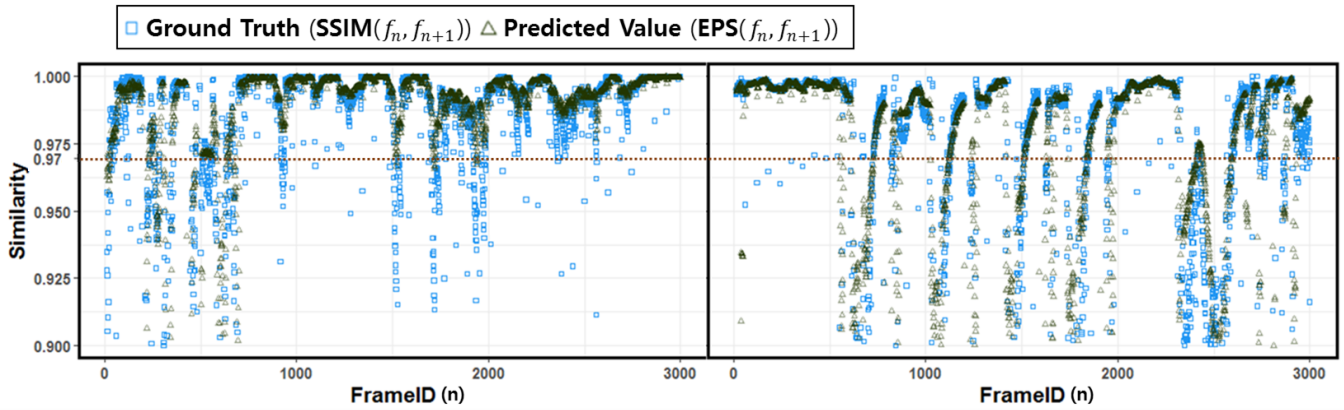


Figure 12: Comparison between the ground truths and the predicted values of SSIM(f_N, f_{N+1}) in Candy Crush Soda Saga (Left) and Archery Master 3D (Right). In total, ~50,000 frames were recorded. For brevity, predictions up-to 3000 frames are shown.

τ_3, it skips 3 frames and moves to 3-Skipping. Otherwise, it moves to 1-Skipping while skipping only one frame.

Determining scaling factors: A raw SSIM score of greater than 0.97 indicates a strong level of similarity between two images in mobile games [22]. Since our concept of predicting perceptual similarity thrives on the scale of actual SSIM scores, we look for threshold values between 0.97 to 1.

Figure 12 shows the comparison between the ground truths, i.e., SSIM(f_N,f_{N+1}) and the predicted similarity scores, i.e., EPS(f_N, f_{N+1}) over two game-play scenarios. We simulated game-play scenarios with an oracle (Section 7.1), which replays identically the recorded game-plays of gaming apps. The predictions were more accurate within the potential range of threshold values (> 0.97) than below it. This implies that the selection of threshold values in the range would result in inclusion of the areas in which EPS potentially makes correct skipping decisions; that is, it is likely that decisions to skip frames would be made (e.g., EPS(f_N, f_{N+1}) > τ_1) when the SSIM values are in reality greater than 0.97.

To maintain quality user experiences, the determination of threshold values should reduce potential of wrong skipping (or state transition) decisions, i.e., skipping decisions should not be made when no skipping should occur in reality. This means that the decision on the similarity assessed by EPS should be consistent with that made by SSIM:

$$\text{EPS}(f_N, f_{N+1}) > \tau_1 \;\; \textit{only if} \;\; \text{SSIM}(f_N, f_{N+1}) > 0.97$$
$$\text{EPS}(f_N, f_{N+3}) > \tau_3 \;\; \textit{only if} \;\; \text{SSIM}(f_N, f_{N+3}) > 0.97$$

The errors in making state transition decisions could cause significant loss in user experience. We experimented with different threshold values in our lab environment in the range of 0.97 to 1 and figured out τ_1 and τ_3 which best satisfy the above conditions. The system also provides users with a UI to customize the selection of the scaling factors on their own as shown in Figure 15.

To assess the frequency of false skipping decisions, we looked into the cases when a given threshold value is lower than predicted EPS (i.e., a skip decision occurs) but higher than actual SSIM (i.e., not to be skipped in reality). As for 1-Skipping transitions, the false decisions accounted for 4.1% of the total 1-Skipping transitions when τ_1 is set to 0.97. In the case of 3-Skipping with τ_3 set to 0.97, 27% of the total 3-Skipping transitions were false. To reduce the number of the false decisions, we increased the threshold values. To be conservative with user experiences, the 90th percentile of the EPS which makes the false skipping with threshold values set to 0.97 is taken as the revised values. For example, τ_3 is set to 0.9945, which is the 90th percentile of the EPS(f_N, f_{N+3}) such that SSIM(f_N, f_{N+3}) $\leq$ 0.97 and EPS(f_N, f_{N+3}) > 0.97. The ratio of false transitions with this new value is reduced to 2.7%

5 CLONING THE PRIMARY DISPLAY

In this section, we describe why and how `RAVEN` clones the primary display of an Android device into a small virtual display.

The processes of graphics and display on Android could be viewed from two perspectives, i.e., that of apps and that of system display. In its rendering loop, a gaming app draws its game frame on `EGLSurface` using OpenGL ES. From the system's view, each frame is stored as `Surface` in a graphics memory area. The display process is initiated by `SurfaceFlinger`. It takes the `Surface` from each app or UI object as a display `layer`. When a `VSYNC` signal arrives, `SurfaceFlinger` arranges display layers from multiple applications or UI objects for the primary display output. Usually, `Hardware Composer` composes all the collected graphics buffers (or layers) and sends the result to the primary display.

A simple approach to read a rendered game frame is to intercept and read it from an app's `EGLSurface` upon its generation. However, this approach has a limitation; an access to `EGLSurface` occurs in a blocking fashion, and would cause the whole rendering pipeline to be delayed. Moreover, it contains a frame in its full-resolution (in the case of Nexus 5X, 1920×1080), which is heavy to process for 60 times per second. According to our preliminary experiment, reading a single frame from an `EGLSurface` takes 0.49 seconds on average. Thus, this approach could be performed for less than 3 frames per second, and is not feasible to match usual frame rates.

A different approach would be to read it from a (display) frame buffer (e.g., `/dev/graphics/fb0`). It is well-known and has been used in other works [27]. However, recent Android OS versions do not allocate display frame buffers[10].

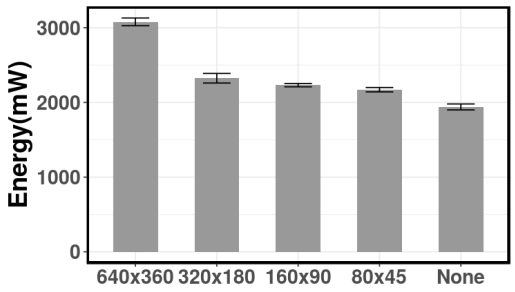
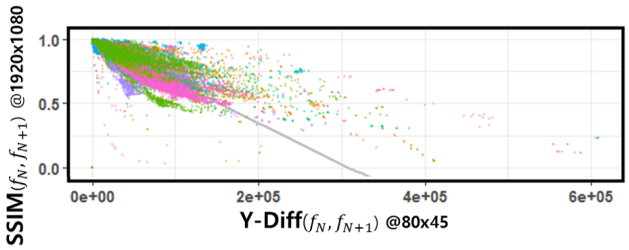
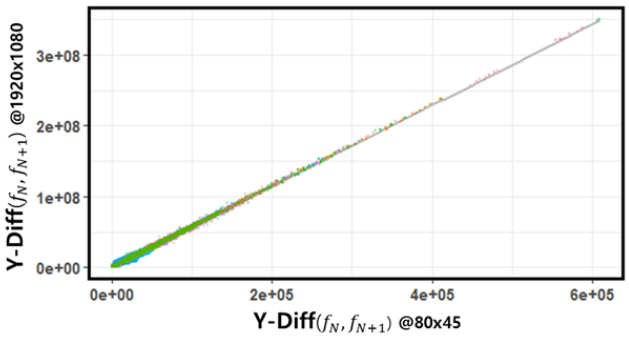


Figure 13: Total system power consumption according to the resolution of a virtual display (width×height; *None* implies no virtual display), measured with Candy Crush Saga.



(a) Correlation between SSIM at 1080p and Y−Diff at 45p.



(b) Correlation between Y−Diff at 1080p and Y−Diff at 45p.

Figure 14: 80×45 resolution and 1920×1080 resolution are nearly identical in (a) SSIM and (b) Y−Diff (Best viewed in color).

RAVEN leverages the virtual display of Android. A virtual display could be allocated and maintained by `SurfaceFlinger` and used to side-track the screen data for various purposes, e.g., screen sharing or capturing. Upon generation, HWC fills up the virtual display while it composes the display for the physical screen. While allocated separately in graphics memory, the overhead to compose its content is small because it piggybacks the process of physical screen composition. As mentioned earlier, HWC also reduces the buffer-composition overhead.

The overhead of a virtual display is primarily proportional to the resolution of the display. In Figure 13, we show the total system power while running `FrameDiff Tracker` with different virtual display resolutions. It indicates that using the resolution of 80×45 can effectively reduce the energy overhead. From extensive trials

Type	Component	LoC
Java based System Service	Rate Regulator	570
Native C/C++ based System Service	Frame Difference Tracker	847
OpenGL/ES	Rate Injector	161

Table 1: RAVEN's code breakdown for AOSP modification.

with different resolutions, we figured out that the resolution of 80×45 for the virtual display is optimal for attaining minimum energy overhead while preserving the performance. The **Y**−Diff results with 80×45 and 1920×1080 are nearly identical with the selected game-play data, scoring a 0.9989 Pearson's correlation coefficient, as shown in Figure 14.

6 IMPLEMENTATION

We have implemented a prototype of RAVEN as a system service by customizing Android (AOSP) 6.0.1 upon Google Nexus 5X. The implementation is in C++ and Java, consisting of 1,578 lines of code in total as shown in Table 1.

Rate Injector: To inject a delay into the rendering loop of a game app, `eglSwapBuffer()` function in OpenGL/ES is modified, so that it lets the rendering loop sleep and adjust the frame rendering rates. For this, a `sleep()` function call is inserted right after submitting the rendered frame, stored in `EGLSurface`, to `BufferQueue`. (See Figure 4, Section 3.) The sleep duration is decided by considering the time for the next rendering (after skipping renderings), T_r, decided by `Rate Regulator` and the estimation of a rendering duration D_r, i.e., T_r - D_r. The rendering duration can be estimated by capturing the system times at the end of a sleep and at the completion of the next round rendering.

The three major components are implemented in different processes, i.e., `Rate Injector` as a part of the game app process, and `Rate Regulator` as well as `FrameDiff Tracker` as parts of system services, requiring a medium for inter-process communication. Current implementation uses a UNIX Local Socket.

FrameDiff Tracker: As mentioned, `SurfaceFlinger` service is used to clone the primary display, which offers the concept of a virtual display [10]. In reality, `SurfaceComposerClient` is used, which acts as an interface for `SurfaceFlinger` since `FrameDiff Tracker` and `SurfaceFlinger` are in different processes. `SurfaceComposerClient` calls `createDisplay()` function to create a virtual display as a newly allocated Surface.

To access thus constructed virtual display, `FrameDiff Tracker` also creates a new `BufferQueue` to connect to the `Surface`, i.e., the virtual display. Thereafter, `FrameDiff Tracker` can extract the cloned frames from the `Surface` successively. The RGB values in the frames are converted into **Y**−values.

`FrameDiff Tracker` utilizing the virtual display faces a slight delay in tracking the most recently rendered frame. Upon rendering a frame, it needs to be composed by the `SurfaceFlinger`, and thus, would not yet be available to `FrameDiff Tracker`. The rendering time of a frame usually overlaps with the display composition time of its previous one. We currently use the most recently composed frame from the virtual display for similarity prediction. This could

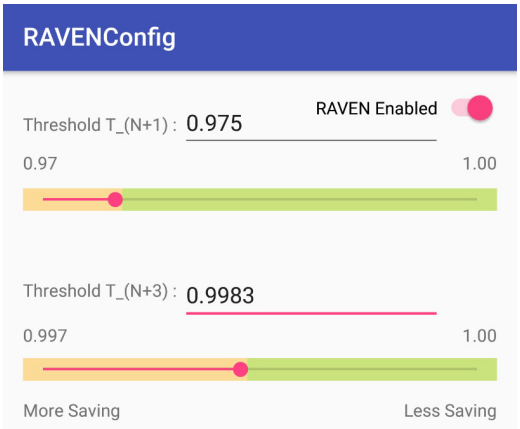


Figure 15: Screen-shot of RAVENConfig, an app to scale threshold values. If τ_3 is moved leftward, frame rates from GPU will lean more towards 15 FPS. Users are also provided with a switch to enable or disable RAVEN to tune the rates or keep the original rates following developer's intention.

incur some errors but within an acceptable range. Our measurements show that the errors were negligible. A better approach would be to slightly tune the regression model in Equation 1.

RAVENConfig: We implemented RAVENConfig, an app for personalized configuration. Figure 15 shows the layout of the RAVENConfig. It allows users to tune the RAVEN parameters according to their own subjective perception on the degree of experience degradation caused by the system. The users are allowed to change the threshold values (τ_1 and τ_3) to inject different rates of game-frame renderings to their game-play. For instance, tuning τ_3 below 0.997, will result in a more frequent usage of the 15 FPS mode, which may affect the user experience in return of higher energy savings. In Section 7, we evaluate the impact of tuning the threshold values with two cases, denoted as PAS and PAS++. PAS++ acts more aggressively to save energy, i.e., it enforces 3-skipping more often than in PAS. PAS sets τ_1 to 0.9975 and τ_3 to 0.9993. PAS++ uses 0.975 and 0.9983 for $\tau 1$ and τ_3.

7 EVALUATION

We evaluated the performance of RAVEN in terms of video quality scores, amount of skipped frames, energy savings in different settings (i.e., PAS and PAS++), and system overheads. We also explored RAVEN's impact on user experiences with two rounds of user studies.

Game Apps used for Experiments. We selected 13 commercial game apps with various graphical characteristics for the evaluation. They are grouped into three categories based on their graphics design and game mechanics: **(A)** Static, **(B)** Dynamic, and **(C)** Hybrid group. The static group includes puzzles and board games (e.g. Solitaire), which are mainly turn-based games that utilize simple graphical effects for visual feedback. The dynamic group consists of games that have *continuous graphical changes* (e.g. racing games). The games that belong to the hybrid group mostly have turn-based flows with clear separation between two phases: the input and the response phase. The input phase tends to be graphically static while displaying basic UI objects. The response phase incorporates dynamic, graphically intense visualizations without user interactions.

	PAS		PAS++		30 FPS		
Game	VMAF	SSIM	VMAF	SSIM	VMAF	SSIM	Type
Perfect Piano	99.98%	99.87%	99.47%	98.65%	99.77%	98.89%	A
Solitaire	99.94%	99.92%	99.68%	99.71%	99.06%	99.57%	
Cookie Run	99.92%	95.60%	99.89%	95.45%	90.35%	94.30%	B
Crossy Road	99.96%	98.24%	99.65%	98.00%	78.87%	90.29%	
Race the Traffic Moto	99.76%	98.21%	99.42%	97.91%	87.81%	93.33%	
Subway Surfers	99.94%	99.71%	99.70%	99.45%	79.46%	87.50%	
Angry Birds	99.80%	99.40%	98.85%	99.08%	97.31%	98.44%	C
Archery Master 3D	99.94%	98.23%	99.01%	97.40%	95.12%	95.60%	
Candy Crush Saga	99.93%	99.09%	99.56%	98.85%	99.09%	98.74%	
Candy Crush Soda Saga	99.97%	99.11%	99.46%	98.44%	98.50%	97.93%	
Criminal Case	99.71%	99.33%	99.32%	99.07%	97.74%	98.40%	
Cut the Rope	99.86%	99.12%	99.81%	98.51%	98.87%	98.57%	
Surgery Simulator	99.96%	99.63%	99.70%	99.24%	99.41%	99.14%	

Table 2: Video quality scores in percentage, higher is better. A, B, and C represents respective game types.

7.1 Objective Quality Assessment

To measure the number of skipped frames and to understand its effect on visual quality across different settings, identical game-play data should be provided for each measurement. Since gaming apps are non-deterministic in their nature, we first recorded game-plays at 60 FPS (for three minutes each) and then extracted each frame as an image from the recorded game-play videos. Thereafter, we applied the perception-aware rate scaling logic on the extracted image sequences, and re-encoded the resulting image sequences as separate videos for the assessment. For comparison, we also prepared constant 30-FPS scenarios; for this, we generated videos by re-encoding the image sequences, constructed by skipping alternate images from the original sequences.

We evaluated the performance of RAVEN using objective video quality metrics. The metrics give video quality scores which collectively indicate perceptual similarity across a sequence of frames from a video. We measured video quality scores for two different frame rendering rate scaling policies (PAS and PAS++) as well as for the constant 30-FPS setting. We used two objective video quality metrics: SSIM, and VMAF[11][2].

Table 2 presents the results of the video quality assessment. The results show that PAS and PAS++ achieve high video-quality scores, which means they do not make significant perceptual differences, compared to the original 60 FPS game-play videos. Also, in each case, PAS and PAS++ produce better results than 30-FPS. In the case of the games in the Static and the Hybrid groups like Criminal Case, Cut The Rope, Candy Crush Saga and Angry bird, 30-FPS also performs decently. On the other hand, for the games in the Dynamic group like Crossy Road and Cookie Run, both PAS and PAS++ scored notably higher, compared to 30-FPS. This indicates that 30-FPS would cause more negative impacts on human visual perception. Figure 16 shows the proportion of the different skipping states that the RAVEN system applied in the re-composed videos.

[2]Unlike SSIM which can give per image based similarity score, VMAF is a newly introduced state-of-the-art video quality metric which gives video quality score across a video stream. It shows a better correlation with device specific Mean Opinion Score (MOS).

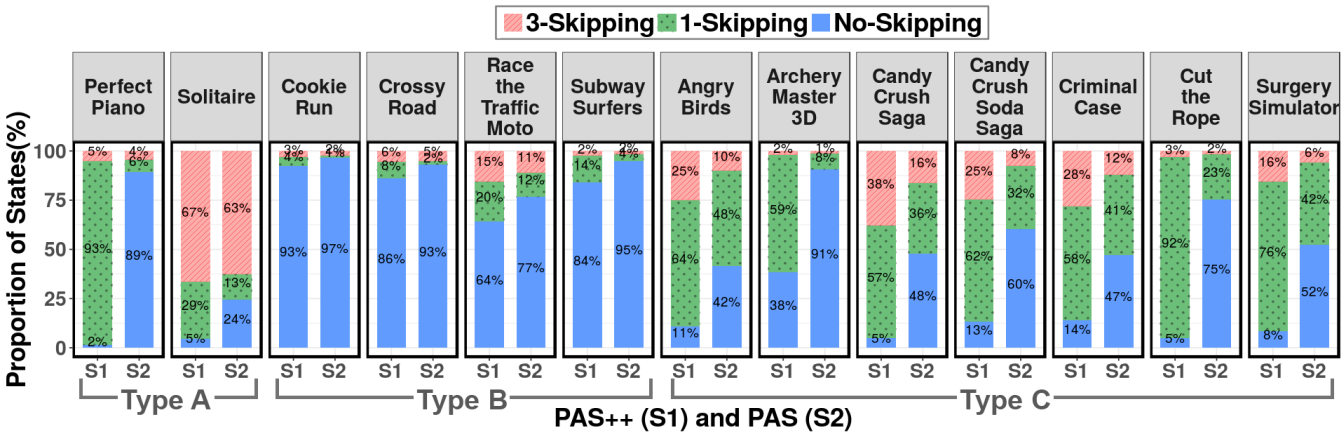


Figure 16: Comparison of PAS and PAS++ in terms of frame skipping (Best viewed in color).

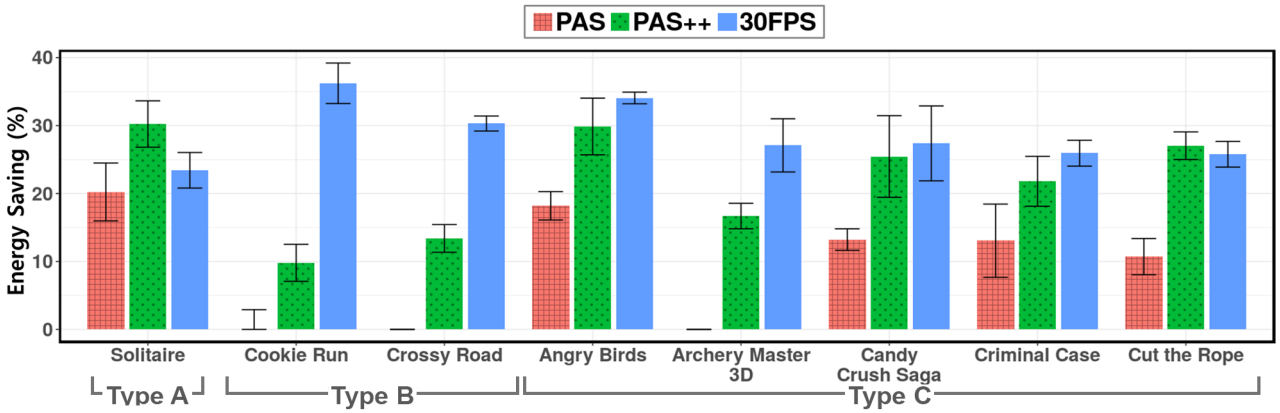


Figure 17: Percentage of energy savings while playing each game for 3 minutes (averaged on five repeated measurements).

In 7 out of the 12 cases, PAS++ skipped as many frames as 30-FPS setting while achieving better video quality scores.

7.2 Energy Saving

We measured energy consumption on Nexus 5X running our customized Android (AOSP) 6.0.1 with a Monsoon Power Monitor[7]. We configured the device brightness to 30% and activated the airplane mode except for Cookie Run which requires a network connection. To prevent CPU/GPU throttling caused by overheating, we ensured that the devices were cooled before each measurement.

We evaluated and compared the energy consumption of four settings: Stock is the baseline setting that uses the stock Android, which does not change frame rendering rates. 30-FPS, PAS++, and PAS are the same as described above. The evaluation covers 8 game apps, including at least one game from each game category to reflect various gaming behaviors. For each game app, we tried to repeat the same play (or interaction) sequence for measurement. We measured the energy consumption for three minutes, and recorded the average after repeating the session five times.

Figure 17 shows the proportion of energy savings in each setting compared to Stock. PAS does not save substantial amount of energy in the games that show continuous movements of characters or changes of scenes such as Cookie Run, Archery Master 3D, and Crossy Road. Except those games, PAS saves at least half the amount of energy saved by 30-FPS. PAS++ saves at least 10% more energy than Stock. Notably, PAS++ saves 5% more energy than 30-FPS in Solitaire since there are many perceptually similar frames as seen in Figure 2 (Section 2).

7.3 System Overhead

Energy overhead: We measured the energy overhead of RAVEN under the same experiment configurations as in Section 7.2 while playing Candy Crush Saga. The measurements were performed for each major component of RAVEN, namely `FrameDiff Tracker`, `Rate Regulator`, and `Rate Injector`. In total, the RAVEN system consumes 173 mW in average. This accounts for only 8% of total device power consumption (2186 mW) while playing the game. 96% of total energy overhead (165.5 mW) is taken by `FrameDiff Tracker` for managing a virtual display and calculating $\mathbf{Y}$–Diffs from it. The other components consume negligible amounts of energy (3.7 mW by `Rate Regulator` and 3.9 mW by `Rate Injector`), since they are engaged in light-weight computations.

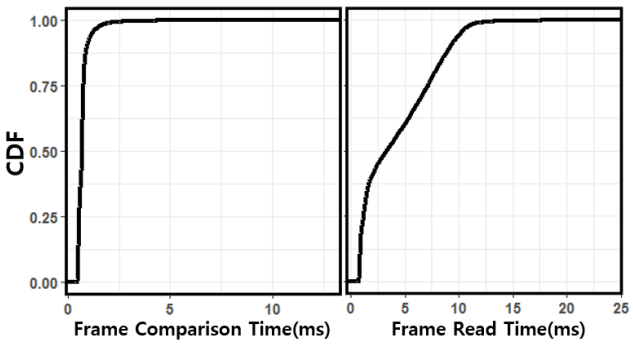


Figure 18: Times taken for frame read and comparison in the games Candy Crush Saga, and Cut the Rope (combined).

Application overhead: Since RAVEN requires modifications in `eglSwapBuffer()` function, it could incur extra overhead in the rendering thread of gaming apps. We measured the processing time of RAVEN's code block in the `eglSwapBuffer()` function. The code block takes only 4.88μs on average. Thus, RAVEN does not impose significant overheads to the rendering threads of game apps.

Frame reading and comparison overhead: The bottleneck of RAVEN's processing pipeline lies in reading frames and calculating **Y**–Diffs by comparing them. We measured the time taken to read each frame and to compute **Y**–Diff between two frames for all gaming apps. Figure 18 shows the results with two games.

7.4 User Study

To explore the impact of RAVEN on user experiences, we conducted a user study with 12 participants (7 males and 5 females are recruited, all were in the ages from 20 to 30). To assess the game-play experience, we used Double Stimulus Impairment Scale(DSIS) and Double Stimulus Continuous Quality Scale (DSCQS), which are widely used as specifications for quality assessment of systems [2][27].

DSIS uses a discrete five-point impairment scale in two variants, i.e., stimulus A and B. Both stimuli are sequentially presented to each participant and the participant is asked to rate the quality of each stimulus. The participants are informed of the device setting. Unlike DSIS, DSCQS aims at assessing the differences in perceived visual qualities among different device settings without informing participants the setting of each task. Also, the order of the assessment tasks is randomly arranged. This could mitigate potential contextual biases (e.g. effects caused by the names of settings). The participants score the perceived quality of each test setting on a [0-100] scale.

DSIS: The participants were asked to play three games: 1) Cookie Run, 2) Solitaire, and 3) Candy Crush Saga for 2 minutes each. The participants are informed of RAVEN with PAS++ before the test. Figure 19(a) shows the impairment rating result. Most of the participants scored either "imperceptible" or "perceptible but not annoying" on all three games. However, some scored "slightly annoying" with Cookie Run. Those participants observed slight stuttering from repetitive movements of a character. The character stayed at the same position and made short, repetitive movements, which made even slight changes in the frame rendering rate noticeable to the participants.

DSCQS: The participants were asked to play the same three games. They played them for four sessions, three minutes each, with different FPS manipulation settings including 1) `Stock` (60-FPS fixed), 2) `30-FPS` fixed, 3) PAS, and 4) PAS++. During each session, they are allowed to play the game freely without game content-related instructions. Figures 19(b) and (c) show the results. The participants could not discriminate the tested settings against `Stock`. The difference in the scores between `Stock` and PAS++ was 4.5 points in average, which could be considered as negligible.

The results above show that the RAVEN system provides similar levels of user experience compared to the reference system, i.e., `Stock`. However, considering the diversity of mobile game apps and their player groups, we plan to evaluate the system more extensively over a broader set of games and participant groups.

8 RELATED WORKS

`dJay` [25] utilized the concept of predictive SSIM-based approach to dynamically tune client GPU rendering workloads in order to, first, ensure all clients get satisfactory frame rates, and second, provide the best possible graphics quality across clients. The system builds a *utility* optimizer in the cloud gaming server which defines *cost* in terms of GPU time and *benefits* in terms of graphics visual quality. On the other hand, RAVEN is the first system which performs rendering rate scaling on a commodity smartphone while maintaining satisfactory user experiences.

Apogee [31] made a low-overhead pre-fetcher that adapts to address patterns found in graphics memory. This approach reduces the degree of multi-threading and thus, resulting in higher energy efficiency. The whole process involves highly complicated processing and needs access to GPU driver source code.

There was an effort to utilize the knowledge on display contents on a screen [27][26]. The work used the RGB differences of the contents and controlled the refresh rates from the display driver. The approach incurs considerable deterioration in quality without a touch-boosting technique, which enforces the refresh rates to the maximum upon a touch event. On the contrary, RAVEN leverages human perception by measuring perceptual similarity, and thus, provides good performance without any extra hints like touch.

Vatjus-Anttila et al.[32] built a GPU power model based on three render complexity characteristics 1) number of triangles, 2) render batches, and 3) addressed texels. It mostly focuses on relating the power consumption to hardware events by observing hardware performance counters of the GPU. Such techniques require low-level hardware modifications and also cannot cope with different graphics designs of gaming apps. There have been several other works on GPU power modeling [28][29][36], but most of them targeted the concept of dynamic frequency scaling. Though these techniques could be applicable on smartphones, they require vendor-side modifications of GPU drivers.

The power consumption of display increases with its pixel brightness [19][21] [24]. In recent years, vendors are exploring methods to reduce power consumption by darkening the least important contents on the screen [20]. On the similar note, Chen et al.[21]

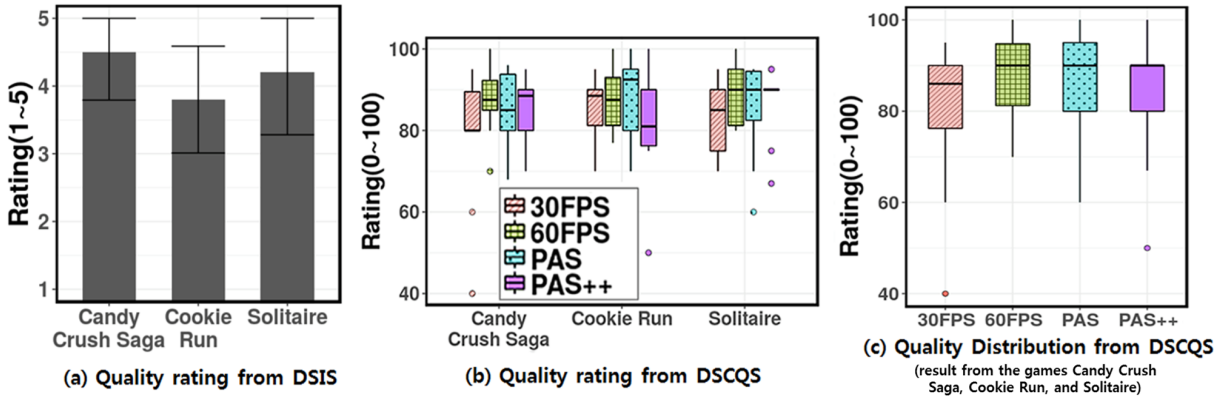


Figure 19: Results from the user study.

introduced a software technique which could perform local dimming for the screen areas covered by users' fingers to save more power, without compromising their visual experiences. Anand et al.[19] and Dong et al.[24] utilized the brightness index of pixels to save significant amounts of power while preserving image qualities. Though these works are motivational, they are subjected to external factors, e.g., lighting conditions. To the contrary, RAVEN's approach is independent of such external factors. Moreover, the works focused more on power consumption rather than retaining the quality of user experience.

Software plug-ins have been provided for desktop computers in recent years [1][15][4][16], which attempt to optimize the rendering power of a GPU. Such plug-ins transfer the information on their graphics objects to the video cards (GPUs) and adopt Dynamic Resolution Rendering [14]. Upon complex graphics contents, they exert slow changes in resolution while trying to maintain high frame rates.

9 DISCUSSION AND FUTURE WORK

As shown in Figure 17, there are cases where the PAS do not lead to energy saving, e.g., Archery Master 3D. Such situations may occur when a game scene continuously changes throughout the play, and RAVEN does not skip frames. Those situations are not prevalent as the design as well as the play of a game vary across its different stages or sessions. While playing Archery Master 3D, we found that there are cases in which RAVEN saved about ~ 15% of energy with the PAS setting. This shows that RAVEN is designed to understand the importance of having a good game-play experience, and does not try to greedily skip frames to achieve higher energy savings.

The current implementation has been built and tested on a Nexus 5X smartphone. We think that it would work on other modern smartphones running latest versions of Android OS. As for GPUs, RAVEN has been tested with an Adreno 418 GPU. More work needs to be performed to see how much energy it saves on other types of GPUs.

`Hardware Composer(HWC)` is a specialized hardware component, generating the final frames for display on a screen. An idea to save more energy would be to extend HWC to include parts or all of perceptual similarity computation. This approach seems feasible since it touches every frame to be displayed. With such extension, tasks of `FrameDiff Tracker` can be offloaded to the HWC, which incur 96% of total energy overhead.

We plan to continue to improve the current design and implementation to explore ways to further optimize power consumption. A potential direction would be to develop a technique for resolution scaling, which could be used in combination with the current approach. Also interesting is to build a tool, based on RAVEN, to assist developers in optimizing power consumption of their gaming apps.

10 CONCLUSION

This paper presents RAVEN, a system for mobile gaming apps to opportunistically save power consumption without degrading user experiences. It introduces a new, on-the-fly perception-aware rate scaling technique, which utilizes human visual perception of graphics changes, and helps reduce power consumption by smartphone GPUs. It develops a light-weight frame comparison technique to measure and predict perception-aware frame similarity. It also builds a low resolution virtual display which clones the device screen for performing similarity measurement at a low-power cost. It is able to work on an existing commercial smartphone and support applications from app stores without any modifications. The experiments in the paper show that RAVEN saves up-to 35% of total device energy consumption without causing any substantial user experience degradation while playing mobile games.

11 ACKNOWLEDGMENT

We are thankful to our shepherd Prof. Robert LiKamWa and the anonymous reviewers for their valuable comments and suggestions that helped bring the paper to its current form. We would also like to thank Joo Ho Lee for all the valuable discussions during the writing of this paper. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2017R1A2B3010504) and the Ministry of Education (2016R1D1A1B03930311).

REFERENCES

[1] 2017. Boost. http://www.hialgo.com/TechnologyBOOST.html. (2017). [Online; accessed Aug-1-2017].

[2] 2017. BT.500 : Methodology for the subjective assessment of the quality of television pictures. https://www.itu.int/rec/R-REC-BT.500-13-201201-I/en. (2017). [Online; accessed Aug-1-2017].

[3] 2017. BufferQueue and gralloc. https://source.android.com/devices/graphics/arch-bq-gralloc. (2017). [Online; accessed 7-July-2017].

[4] 2017. LucidLogix. http://www.lucidlogix.com/powerxtend/overview/. (2017). [Online; accessed Aug-1-2017].

[5] 2017. Mali-G71 ARM GPUs. https://www.arm.com/products/multimedia/mali-gpu/high-performance/mali-g71.php. (2017). [Online; accessed Aug-1-2017].

[6] 2017. Mali-T760 ARM GPUs. https://www.arm.com/products/multimedia/mali-gpu/high-performance/mali-t760.php. (2017). [Online; accessed Aug-1-2017].

[7] 2017. Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor. (2017). [Online; accessed Aug-1-2017].

[8] 2017. Open CV. http://docs.opencv.org/2.4.13/. (2017). [Online; accessed Aug-1-2017].

[9] 2017. Samsung Game Tuner. https://play.google.com/store/apps/details?id=com.samsung.android.gametuner.thin&hl=en. (2017). [Online; accessed Aug-1-2017].

[10] 2017. SurfaceFlinger and Hardware Composer. https://source.android.com/devices/graphics/arch-sf-hwc. (2017). [Online; accessed 7-July-2017].

[11] 2017. Toward A Practical Perceptual Video Quality Metric . http://techblog.netflix.com/2016/06/toward-practical-perceptual-video.html. (2017). [Online; accessed Aug-1-2017].

[12] 2017. YUV - Wikipedia. https://en.wikipedia.org/wiki/YUV. (2017). [Online; accessed Aug-1-2017].

[13] February 11, 2013. Khronos Native Platform Graphics Interface. https://www.khronos.org/registry/EGL/specs/. (February 11, 2013). [Online; accessed Aug-1-2017].

[14] July 13, 2011. Dynamic Resolution Rendering Article. https://software.intel.com/en-us/articles/dynamic-resolution-rendering-article. (July 13, 2011). [Online; accessed Aug-1-2017].

[15] March 17, 2013. HiAlgoBoost. http://semiaccurate.com/2013/04/17/hialgo-boost-for-far-cry-3/. (March 17, 2013). [Online; accessed Aug-1-2017].

[16] March 23, 2012. DynamiX, LucidLogix. http://semiaccurate.com/2012/03/23/lucid-releases-dynamix-software/. (March 23, 2012). [Online; accessed Aug-1-2017].

[17] May 12, 2009. OpenGL ES Common Profile Specification. https://www.khronos.org/files/opengles_shading_language.pdf. (May 12, 2009). [Online; accessed Aug-1-2017].

[18] October 29, 2012. The Truth About Cats and Dogs: Smartphone vs Tablet Usage Differences. http://flurrymobile.tumblr.com/post/113379683050/the-truth-about-cats-and-dogs-smartphone-vs. (October 29, 2012). [Online; accessed Aug-1-2017].

[19] Bhojan Anand, Karthik Thirugnanam, Jeena Sebastian, Pravein G. Kannan, Akhihebbal L. Ananda, Mun Choon Chan, and Rajesh Krishna Balan. 2011. Adaptive Display Power Management for Mobile Games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, New York, NY, USA, 57–70. https://doi.org/10.1145/1999995.2000002

[20] J. Betts-LaCroix. 2010. Selective dimming of oled displays. (June 17 2010). https://www.google.com/patents/US20100149223 US Patent App. 12/538,846.

[21] Xiang Chen, Kent W. Nixon, Hucheng Zhou, Yunxin Liu, and Yiran Chen. 2014. FingerShadow: An OLED Power Optimization Based on Smartphone Touch Interactions. In *Proceedings of the 6th USENIX Conference on Power-Aware Computing and Systems (HotPower'14)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=2696568.2696574

[22] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*. ACM, New York, NY, USA, 121–135. https://doi.org/10.1145/2742647.2742657

[23] B. Dietrich and S. Chakraborty. 2014. Forget the battery, let's play games!. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. 1–8. https://doi.org/10.1109/ESTIMedia.2014.6962338

[24] Mian Dong and Lin Zhong. 2011. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/1999995.2000004

[25] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. 2015. dJay: Enabling High-density Multi-tenancy for Cloud Gaming Servers with Dynamic Cost-benefit GPU Load Balancing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 58–70. https://doi.org/10.1145/2806777.2806942

[26] Dongwon Kim, Nohyun Jung, and Hojung Cha. 2014. Content-centric Display Energy Management for Mobile Devices. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, USA, Article 41, 6 pages. https://doi.org/10.1145/2593069.2593113

[27] D. Kim, N. Jung, Y. Chon, and H. Cha. 2016. Content-Centric Energy Management of Mobile Displays. *IEEE Transactions on Mobile Computing* 15, 8 (Aug 2016), 1925–1938. https://doi.org/10.1109/TMC.2015.2467393

[28] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. S. Kim. 2011. Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 111–120. https://doi.org/10.1109/PACT.2011.17

[29] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[30] Margaret S Livingstone. 2002. *Vision and Art: The Biology of Seeing*.

[31] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 73–82. http://dl.acm.org/citation.cfm?id=2523721.2523735

[32] J. M. Vatjus-Anttila, T. Koskela, and S. Hickey. 2013. Power Consumption Model of a Mobile GPU Based on Rendering Complexity. In *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*. 210–215. https://doi.org/10.1109/NGMAST.2013.45

[33] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[34] Yu Yan, Songtao He, Yunxin Liu, and Longbo Huang. 2015. Optimizing Power Consumption of Mobile Games. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower '15)*. ACM, New York, NY, USA, 21–25. https://doi.org/10.1145/2818613.2818746

[35] Semir Zeki. 1993. *A Vision of the Brain*. Blackwell scientific publications.

[36] Y. Zhu, A. Srikanth, J. Leng, and V. J. Reddi. 2014. Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing. *IEEE Computer Architecture Letters* 13, 1 (Jan 2014), 33–36. https://doi.org/10.1109/L-CA.2012.33