Optimizing Smartphone Power Consumption through Dynamic Resolution Scaling

Songtao He[1,2], Yunxin Liu[1], Hucheng Zhou[1]

[1]Microsoft Research, Beijing, China
[2]University of Science and Technology of China, Hefei, China

hst@mail.ustc.edu.cn, {yunliu, huzho}@microsoft.com

ABSTRACT

The extremely-high display density of modern smartphones imposes a significant burden on power consumption, yet does not always provide an improved user experience and may even lead to a compromised user experience. As human visually-perceivable ability highly depends on the user-screen distance, a reduced display resolution may still achieve the same user experience when the user-screen distance is large. This provides new power-saving opportunities. In this paper, we present a flexible dynamic resolution scaling system for smartphones. The system adopts an ultrasonic-based approach to accurately detect the user-screen distance at low-power cost and makes scaling decisions automatically for maximum user experience and power saving. App developers or users can also adjust the resolution manually as their needs. Our system is able to work on existing commercial smartphones and support legacy apps, without requiring re-building the ROM or any changes of apps. An end-to-end dynamic resolution scaling system is implemented on the Galaxy S5 LTE-A and Nexus 6 smartphones, and the correctness and effectiveness are evaluated against 30 games and benchmarks. Experimental results show that all the 30 apps can run successfully with per-frame, real-time dynamic resolution scaling. The energy per frame can be reduced by 30.1% on average and up to 60.5% at most when the resolution is halved, for 15 apps. A user study with 10 users indicates that our system remains good user experience, as none of the 10 users could perceive the resolution changes in the user study.

Categories and Subject Descriptors

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

General Terms

Experiments; Measurement; Performance

MobiCom'15, September 07–11, 2015, Paris, France.

DOI: http://dx.doi.org/10.1145/2789168.2790117.

Keywords

Smartphone; GPU; Power Consumption; Display Resolution; Dynamic Resolution Scaling

1. INTRODUCTION

The display resolution of smartphones has become increasingly high. Since the "Retina display" of the iPhone 4 released in 2010 [18], display resolution of smartphones has continued to rise, from 960x640 pixels of the iPhone 4 to 1920x1080 pixels (Full HD) and recently to 2560x1440 pixels (2K). As a result, latest smartphones have an extremely-high display density. For example, the LG G3 [16] has a display of 538 pixels per inch (ppi), while the Samsung Galaxy S5 LTE-A [17] has a display of 577 ppi.

However, this extremely-high display density does not always help bring an improved user experience. The Retina display of the iPhone 4 has a display density of only 326 ppi. The high display density of the LG G3 and the Samsung Galaxy S5 LTE-A is far beyond the human ability of visual acuity, and thus does not lead to visibly-sharper content and interfaces displayed on the screen. Nevertheless, a high-resolution display consumes a large amount of system resources, especially the GPU computation, and thus results in high system power consumption (see Section 2). As energy is a paramount concern on smartphones, it is desirable to study the tradeoff between display density and power consumption.

In this paper, we propose to optimize the high power consumption caused by high-density displays through Dynamic Resolution Scaling (DRS). Based on the viewing distance (i.e., the screen-user distance), DRS dynamically adjusts the user-interface resolution displayed on the screen but ensures that the pixels are always small enough to be individually unobservable by human eyes. The philosophy of DRS is that the system should render the interface with exactly as many details as it is perceptible to a user, in order to maximize both user experience and battery life. When a resolution is too high for a viewing distance, DRS reduces the resolution to save power.

Enabling DRS imposes several requirements. First, DRS must seamlessly adjust the resolution in real-time as the viewing distance changes, without compromising the user experience. Second, DRS must be done systematically and is transparent from applications. Third, DRS must be able to accurately determine the viewing distance of a user in real-time and with minimal energy cost.

To meet the requirements, we develop per-frame DRS technique for seamless and real-time resolution adjusting. We intercept the system graphics pipeline to enable system-wide DRS, without requiring any changes or recompiling from applications, or re-building a new smartphone ROM. We further propose to use ultrasonic sensors to detect the viewing distance of a user, which is accurate and low-power. Based on the detected viewing distance, we utilize existing knowledge of the human visual system to define the required display-pixel density for optimal user experience, and adjust the extravagant resolution accordingly.

We have implemented a DRS system that works on existing commercial smartphones including the Galaxy S5 LTE-A and the Nexus 6. We conduct comprehensive experiments and a user study to evaluate the DRS implementation against 30 real gaming and benchmark applications. Experimental results show that all the 30 applications can run successfully with per-frame, real-time dynamic resolution scaling. The energy per frame can be reduced by 30.1% in average and up to 60.5% at most when the resolution is halved, for 15 applications. Even for other less GPU-intensive applications such as Adobe Reader and web browser, we still reduce the energy per frame for about 10%. A user study with 10 users indicates that our system remains good user experience: none of the 10 users could perceive the resolution changes in the user study although the resolution changed for more than 100 times for each user.

To the best of our knowledge, we are the first to design and implement a DRS system for smartphones. The main contributions of this paper are as the following:

1. We present a per-frame and application-transparent system-level DRS design and implementation.
2. We adopt an ultrasonic-based approach that detects the user-screen distance in real-time and with low power.
3. We conduct experiments and a user study to confirm the effectiveness of our system for maximum user experience and power saving.

The rest of this paper is organized as follows. Section 2 motivates the need of DRS and set our goals. Section 3 introduces how the graphics pipeline works on Android platform as background. Section 4 overviews the architecture design and the key components of DRS. Section 5 and Section 6 presents how to enable DRS and how to determine the viewing distance of users, respectively. Section 7 describes more implementation details and Section 8 reports the evaluation results. Section 9 discusses the limitations and future work, Section 10 surveys the related work and Section 11 concludes.

2. MOTIVATION AND GOALS

In this section, we describe three problems caused by high-resolution displays on smartphones, to motivate our work and set our goals of this paper.

2.1 Motivation

High-resolution displays can be only justified if they can improve the user experience, e.g., allowing sharper images or more content to be legibly displayed. However, they fail to provide an improved user experience on smartphones because the physical size of display is small and thus the extremely-high display density surpasses the human ability of visual acuity. Instead, high-resolution displays consume a large amount of system resources and cause the following three issues on smartphones.

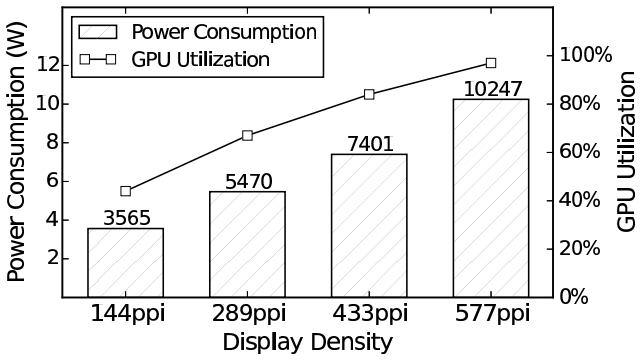


Figure 1: System power and GPU utilization of Galaxy S5 LTE-A in different display resolutions.

High power consumption. High-resolution displays significantly increase the system power mainly because rendering extra pixels imposes very high GPU workload and memory usage. To verify this, we wrote a test program to render a 3D scene with a fixed number of vertexes/triangles at 60 frames per second (fps). The program was specially designed to generate independent arithmetic, special function and memory access workload on GPU. So, the GPU can reach its peak power consumption while processing those workload concurrently. We ran the program on Galaxy S5 LTE-A with different display resolutions and measured the system power (with the screen turned off) and the GPU utilization. The result is shown in Figure 1. The system power increases almost linearly when the display density increases and a higher display resolution consumes significantly more power than a lower display resolution. For example, when the display resolution is 2560x1440 pixels (577 ppi), the system power is 10247 mW, 87.3% higher than the system power when the display resolution is 1280x720 pixels (289 ppi). This power increase is mainly caused by the increased GPU workload. Indeed, to support a high-resolution display, smartphones have to utilize a high-performance and thus power-hungry GPU. For example, the Galaxy S5 LTE-A utilizes an Adreno 420 GPU that dominates the system power of more than 10 W in Figure 1.

Compromised user experience. The high GPU workload imposed by a high-resolution setting may saturate the GPU and thus lead to a reduced frame rate. For example, we ran the GFX Benchmark [1] on the Galaxy S5 LTE-A with the native resolution (2560x1440 pixels), the frame rate of the Manhattan scene is only 11.5 fps, far lower than the default frame rate of 60 fps on Android. As shown in Figure 2, even if we reduce the resolution to 1280x720 pixels, the frame rate only increases to 30.2 fps. When the frame rate is too low (e.g., less than 30 fps), users may perceive visible delay on the display. Thus, this high overhead of display may significantly impact apps performance and harm the user experience.

Overheating. The high power consumption caused by a high-resolution display also generates more heat energy. To prevent overheating, smartphone systems will force to re-

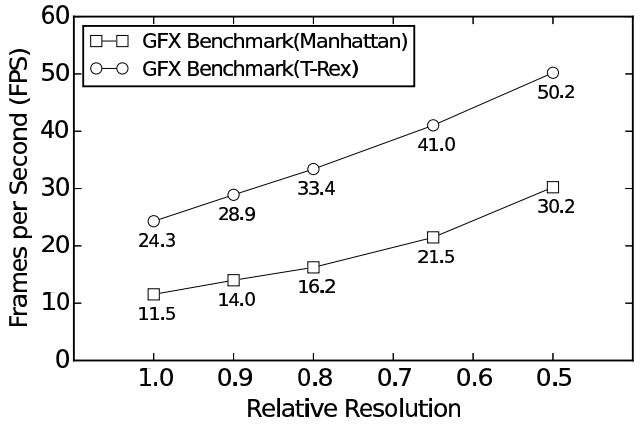


Figure 2: GFX benchmark frame rate in different display resolutions.

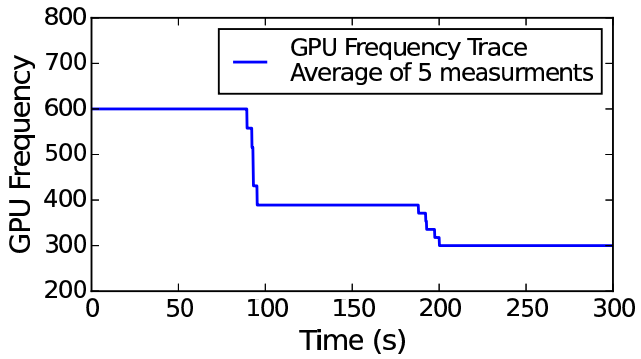


Figure 3: GPU frequency in running the Ridge Racer Slipstream game on Galaxy S5 LTE-A.

duce the operating frequencies of the CPU and GPU when their temperatures exceed predefined thresholds. Such frequency throttling would happen aggressively. Figure 3 shows the GPU frequency in running the *Ridge Racer Slipstream* game [4] on the Galaxy S5 LTE-A. Within 5 minutes, the GPU frequency were quickly throttled from 600 MHz to 300 MHz. As a consequence, the gaming performance is significantly downgraded resulting in an unacceptable user experience.

To alleviate or eliminate these issues caused by extravagant display resolutions, it is desirable for users to flexibly change the display resolution of their smartphones when needed. However, existing commercial smartphones including iPhones, Windows Phones, and Android phones, always run at the native display resolution and provide no way for app developers or users to change the display resolution.

2.2 Goals

In this paper, we seek to enable users to change the display resolution of their smartphones on-the-fly. The proposed DRS system should work on existing commercial smartphones and support all legacy applications without requiring any changes or recompiling from applications, or re-building a new ROM. The system should also be flexible enough for various usage scenarios, and we below describe three example scenarios.

Automatic power optimization. The DRS system runs transparently in the background, automatically infers the viewing distance, and decides the best display density for maximum user experience but with minimal system power consumption.

Run high-performance applications. Sometimes users may want to run a high-performance application, e.g., the latest version of a 3D game, but their last-generation old phones cannot afford. With the DRS system, users can force to reduce the rendering resolution and run it smoothly.

Extend battery life aggressively. Users may also use our system to reduce the display resolution of their phones in a proactive way. This would be particularly useful when the battery is low but the user still wants to use the phone for a longer time, e.g., to continue enjoying the game or composing a long email. In this case, the user could tolerate the compromised user experience to save power and extend the battery life.

It is worth to note that we do not change the physical resolution of smartphone displays. As a software solution, we cannot change the number of hardware pixels of a smartphone display. What we change is the resolution of the user interface (i.e., the final image generated to the system frame buffer) to be displayed on the screen. After we reduce the resolution, one pixel of the user-interface image is displayed by multiple physical pixels of the display hardware. As the physical pixels are very small, the users may still perceive a smooth user experience. Consequently, we save power by reducing the GPU computations in rendering the user interface at a lower resolution, rather than by reducing the power consumption of the display hardware. Unless otherwise stated, in this paper, we refer "display resolution" as the resolution of the content to be displayed on the screen.

On smartphones, GPU is mainly used for 3D and 2D graphics acceleration. It is not only heavily used by high-end 3D games but also widely used for rendering 2D user interfaces in non-gaming apps and system UI. With the support of OpenCL [11] on smartphones, GPU can also be used for general-purpose computing (GPGPU) such as accelerating various machine-learning algorithms. In this paper, we mainly focus on GPU-intensive gaming apps but we will show that non-gaming apps can also benefit from our DRS system to reduce their power consumption.

3. ANDROID GRAPHICS FUNDAMENTALS

In this section, we describe the background information on Android graphics architecture and pipeline on top of which we design our DRS system.

3.1 Android Graphics Overview

Figure 4 shows an overview of Android graphics architecture focusing on the data flow. On Android, OpenGL ES/EGL[1] is used for GPU-accelerated graphics. Applications call the OpenGL ES API [31] to use the GPU for graphics processing and to draw their User Interface (UI). The processing result is put into a graphics buffer in a structure

[1]OpenGL ES (Open Graphics Library for Embedded Systems) is the standard for embedded accelerated graphics [29]. OpenGL ES relies on the EGL (Embedded-System Graphics Library) API [28].

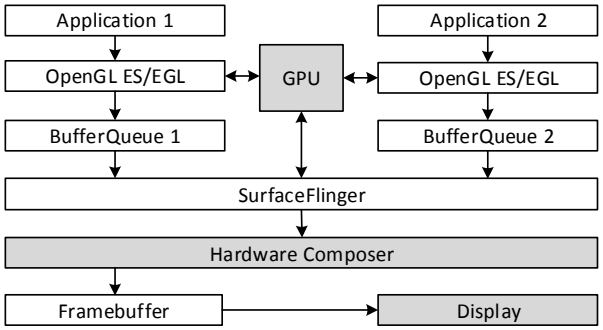


Figure 4: Android Graphics Overview.

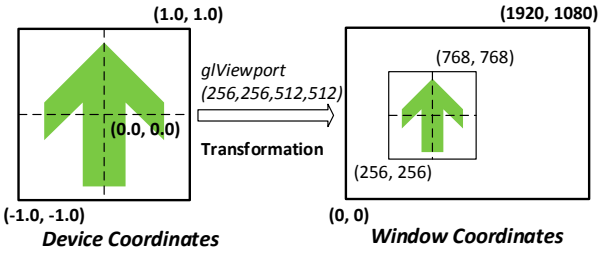


Figure 5: Transformation from normalized device coordinates to window coordinates.

called *BufferQueue*. Each application has its own BufferQueue that has three graphics buffers by default. The system UI process also has a dedicated BufferQueue for drawing the system UI elements such as the "navigation bar" at the bottom of the screen and the "status bar" at the top of the screen.

Android uses a system service called *SurfaceFlinger* to coordinate all the graphics layers from the running applications and the system UI process. SurfaceFlinger knows the layout of all the UI windows in the system by working with the *WindowManager* service. Smartphone displays typically refresh at a rate of 60 fps and for each frame period the system generates a VSYNC signal. When a VSYNC signal arrives, SurfaceFlinger collects all the graphics buffers for visible layers and asks the *Hardware Composer* to composite all visible layers together. Usually the Hardware Composer will do the composition itself and generates the final graphics data into the system *Framebuffer* for displaying on the screen. Sometimes the Hardware Composer may determine that the most efficient way to composite the buffers is using the GPU and thus asks SurfaceFlinger to call OpenGL ES API to use the GPU for buffer composition. In this case, SurfaceFlinger also generates GPU workload. After SurfaceFlinger consumes a graphics buffer in a BufferQueue, the corresponding application is notified for drawing a new frame.

3.2 Graphics Pipeline and Coordinates

Graphics processing consists of a sequence of steps called graphics pipeline. A modern graphics pipeline may be very complex and usually has more than ten stages. These stages may be grouped into three high-level stages: vertex processing, rasterization, and pixel processing.

In OpenGL ES, complex geometry scenes are represented by basic vertexes and the relationships among them. For example, three vertexes form a triangle face. The vertex-processing stage processes vertices, typically performing operations such as transformations and skinning [26]. Vertex processing is done in a square, uniform coordinate system called *normalized device coordinate space*. In this space, the lower left corner corresponds to (-1, -1) and the upper right corner corresponds to (1, 1), as shown on the left side in Figure 5. After the vertex-processing stage, each vertex has its own normalized device coordinates.

Then, the rasterization stage solves the relationships (lines, triangles etc.) among these vertexes and maps these lines and triangles from the continuous device-coordinate space to a discrete window-pixel space as shown on the right side in Figure 5. In OpenGL ES, this coordinate mapping is configured by calling the *glViewport()* function that takes four parameters to determine the affine transformation. As an example shown in Figure 5, *glViewport(256, 256, 512, 512)* maps the points (-1, -1) and (1, 1) in the normalized device coordinates to the points (256, 256) and (256+512, 256+512) in the window coordinates.

Finally, the pixel-processing stage generates per-pixel data such as colors and depths for each pixel. This is done in the window-coordinate space. Since OpenGL ES 2.0, vertex-processing and pixel-processing become programmable. Developers can define the functions of these two stages by using shader programs. Typically, shader programs are written using a specific programming language, *OpenGL ES Shader Language* [30]. The source code can be compiled and linked at runtime through OpenGL ES API calls.

For a given scene, if the display resolution increases, there will be more pixels to compute, and thus the GPU workload of the two pixel-based stages of rasterization and pixel-processing will increase accordingly. The GPU workload of the vertex-processing stage will remain unchanged because vertex-processing is done in the device-coordinate space which is independent from the display resolution. Consequently, DRS saves power by reducing the GPU workload generated in the rasterization stage and the pixel-processing stage.

According to Figure 4, to reduce the resolution of display content, a simple approach is to let SurfaceFlinger scale down the pixel resolution of the graphics buffers in the BufferQueues. However, this approach cannot save power because the GPU workload required to generate those graphics buffers remains the same and thus consumes the same amount of power. Therefore, we have to seek for a new approach and next we describe how our approach works.

4. SYSTEM ARCHITECTURE

Enabling DRS on existing smartphones imposes a couple of challenges. First, we must reduce the number of pixels in all the pixel-related computations from the complex procedures of rasterization and pixel-processing involving many OpenGL function calls. Second, OpenGL libraries are close source and thus we cannot revise and re-compile the source code to add new functionalities. And we cannot require rebuilding a smartphone ROM or making application-specific modifications. Third, we must design a lightweight approach to decide the user-screen distance for automatic resolution scaling.

To address the above challenges, we employ binary-rewriting techniques that are able to hook and intercept the function

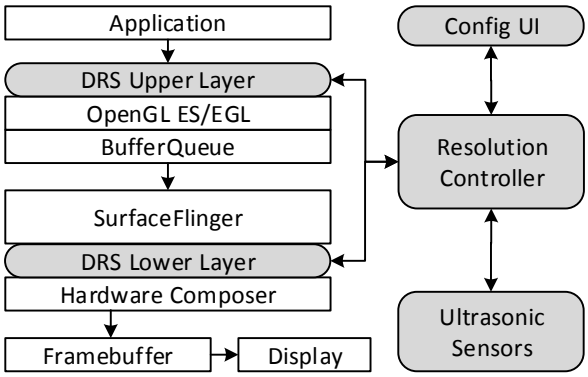


Figure 6: DRS System Architecture.

calls in existing binaries and add new functionalities. As a result, we can enable DRS on commercial smartphones without requiring any source-code level changes. Due to the complexity of OpenGL rendering context, it is non-trivial to intercept the right function calls to make DRS work correctly. In Section 5 we describe the details. To detect the user-screen distance, we design a lower-power ultrasonic based approach and we provide more details in Section 6.

Figure 6 shows the architecture of our DRS system. To enable resolution scaling, we add two new layers into the existing system. The first layer is the *DRS Upper Layer* that sits between the application layer and the OpenGL ES/EGL layer. It intercepts the necessary OpenGL ES function calls and EGL function calls to ensure that the graphics rendering is done with a proper display resolution. It applies a scaling factor to the parameters of the necessary OpenGL ES/EGL function calls to transform the default display resolution to a targeted one.

The second layer is the *DRS Lower Layer* that locates between the SurfaceFlinger layer and the Hardware Composer. This layer intercepts the function calls passed to the Hardware Composer to ensure that the composition is done with a proper display resolution. This second layer is needed because after the DRS Upper Layer scales down the resolution, we must scale the resolution up so that the rendered content can be correctly displayed on the screen in the native display resolution. These two DRS layers synchronize with each other to make sure they use the same targeted display resolution for the same graphics buffer in the BufferQueue. This is necessary because if a user changes the targeted display resolution to a new value, the DRS Upper Layer will start to use a new scaling factor to generate graphics buffers into the BufferQueue. The DRS Lower Layer needs to make sure that the older scaling factor is used for previously-generated graphics buffers and the new scaling factor is applied only to newly-generated graphics buffer during the composition.

To decide a proper targeted display resolution, we add ultrasonic sensors into the system for instant and accurate viewing-distance detection. Based on the measured viewing-distance, the *Resolution Controller* calculates the best display resolution for maximum user experience and maximum power optimization, and indicates the two DRS layers to do scaling transformation for the calculated display resolution. In addition, our DRS system also provide a UI for users to

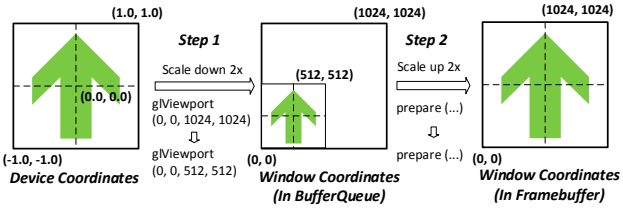


Figure 7: Scaling process for a default render target.

configure the system behaviors, e.g., enabling automatic resolution scaling based on the viewing-distance or setting to a fixed display resolution.

5. DYNAMIC RESOLUTION SCALING

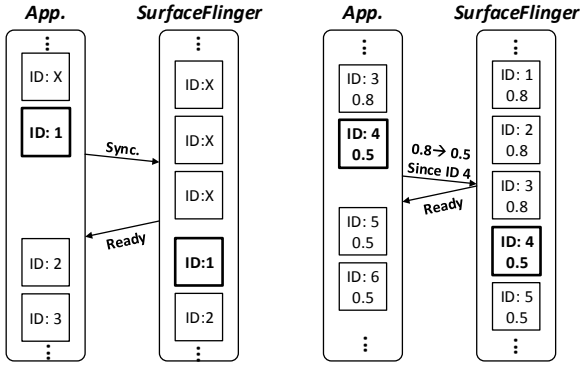
The key of DRS is to control the pixel-resolution of the graphics results rendered by the GPU. In OpenGL ES, the GPU-rendered graphics results are stored in a memory buffer called *render target*. There are two types of render targets: default render target and user-defined render target. A default render target is a graphics buffer in a BufferQueue as shown in Figure 4. The size of a default render target is typically equal to the native display resolution of the device screen, as its content will be drawn on the device screen. In applications that cannot support the native display resolution, they may use a smaller default render target. In this case, the graphics buffers will be scaled up, e.g., by 2x, in the composition stage of SurfaceFlinger, to fill up the whole device screen.

User-defined render targets are usually used for off-screen content rendering, to calculate the intermediate graphics results that will not be directly drawn on the device screen. Instead, the content of a user-defined render target may be used as the input to other render targets, e.g., to a default render target for displaying on the device screen. User-defined render targets are widely used in many graphics algorithms such as high dynamic range [20], shadows [21] and depth of field [47].

To achieve maximum power saving, our system support DRS for both default render targets and user-defined render targets. For a given render target, we intercept all the OpenGL ES/EGL API calls that manipulate the render target for correct resolution scaling. Next we describe the details.

5.1 DRS for Default Render Target

We design a two-step DRS scheme for default render targets, each step running in one of the two DRS layers, respectively. Assume that the native display resolution of a smartphone is 1024x1024 pixels and we want to scale down the resolution by 2x to 512x512 pixels (i.e., the scaling factor is 0.5), Figure 7 shows the scaling process for a default render target. In Step 1, we intercept all the *glViewport()* function calls on the default render target and apply the scaling factor of 0.5 to their parameters. That is, we convert the function calls of *glViewport(0, 0, 1024, 1024)* to *glViewport(0, 0, 512, 512)*. As a result, the content of the default render target is scaled down to 512x512 pixels as shown in the middle of Figure 7. Then the GPU will do rasterization and pixel processing using this small pixel block rather than



(a) Sync. of frame ID (b) Sync. of scaling factor

Figure 8: Synchronizing frame ID and scaling factor between application and SurfaceFlinger.

the original big one of 1024x1024. Consequently, the GPU workload and thus its power consumption are reduced.

In Step 2, we intercept the *prepare()* function calls during the composition stage in SurfaceFlinger. For each frame, SurfaceFlinger uses the *prepare()* function to tell the Hardware Composer how to do the composition. By changing the parameters of the *prepare()* function call, we tell the Hardware Composer to scale up the reduced pixel block back to the original size so that it can be correctly displayed on the device screen.

The scaling process in Figure 7 is simplified for easy illustration. More functions must be intercepted to ensure that all the function calls have a consistent OpenGL ES rendering context. In Section 7, we list the functions that we intercepted in our implementation.

Synchronization. The two steps must agree on the same scaling factor to ensure that the final graphics UI is correctly displayed on the device screen. As they run in different processes, Step 1 in the application process and Step 2 in the SurfaceFlinger process, a synchronization scheme is needed. A simple approach is to extend the data structure of graphics buffers in BufferQueue to add a new field for the scaling factor. However, this approach is not feasible without changing and recompiling the source code of the Android graphics system, because the graphics buffers are totally managed by the Android graphics system and transparent from applications. Thus, we design a new synchronization scheme as shown in Figure 8.

The key idea of our scheme is to assign a unique ID to each graphics buffer in a BufferQueue. We call it *frame ID*. As we cannot change the data structure of a graphics buffer to add a frame ID, the application and SurfaceFlinger maintain their own frame IDs separately. For a graphics buffer, the same frame ID must be used by both the application and SurfaceFlinger. To achieve it, the application initiates a process to synchronize frame ID with SurfaceFlinger, as shown in Figure 8a. The application holds its current graphics buffer (dequeued from the BufferQueue) without adding it back into the BufferQueue. The application assigns a frame ID of 1 to the graphics buffer and sends a message to SurfaceFlinger and waits for a response. SurfaceFlinger continues to consume the graphics buffers in the BufferQueue of the application as normal. After all the graphics buffers in the BufferQueue are consumed, SurfaceFlinger sends a *ready* message back to the application. Then the application resumes to enqueue its graphics buffer into the BufferQueue as normal and SurfaceFlinger assigns the same frame ID of 1 to the first graphics buffer in the BufferQueue after the ready message. Consequently, the application and SurfaceFlinger are able to assign the same frame ID to all the graphics buffers afterwards. As three graphics buffers are used in a BufferQueue, with a frame rate of 60 fps, the application waits for SurfaceFlinger to consume at most two graphics buffers, i.e., up to 33 ms. Such a small latency is hardly invisible to users, and we only need to do this synchronization process once for each application launch. Therefore, this frame-ID synchronization introduces negligible impact on the user experience.

Figure 8b shows what happens when the scaling factor is changed. The application simply applies the new scaling factor to its current graphics buffer and sends a message to tell SurfaceFlinger from which frame the scaling factor is changes, and waits for a response. For the example in Figure 8b, the scaling factor is changed from 0.8 to 0.5 starting from frame 4. As SurfaceFlinger uses the same frame ID, it can reply a ready message immediately and use the new scaling factor starting from frame 4. This process introduces less than 1 ms extra latency. Thus, we are able to enable per-frame, real-time resolution scaling.

5.2 DRS for User-Defined Render Target

Enabling DRS for a user-defined render target is more complex than the case of a default render target. Similar to a default graphics buffer in a BufferQueue that is managed by the Android graphics system, a user-defined graphics buffer is completely managed by OpenGL ES libraries and we cannot directly change its content. Different from a default graphics buffer, a user-define graphics buffer is used in a much more complex way. User-defined render targets are typically used to create a variety of texture effects that can be reused later, known as *render to texture* technique [46]. However, we cannot change the texture parameters such as the texture size for resolution scaling without re-allocating the memory of the graphics buffer. Furthermore, when a user-defined render target is used as the input of another rendering process, OpenGL ES system uses the relative coordinates to locate points in a texture, which causes another transformation from the relative coordinate to the absolute-pixel coordinate. Maintaining those coordinate transformations in all the involved function calls can be very complex.

Therefore, we take a different approach to make DRS for user-defined render targets easy. Instead of using the graphics buffer managed by OpenGL ES system, we allocate a new graphic buffer for a user-defined render target and use the new buffer for the whole graphics-rendering pipeline. As we can totally control the new buffer, we can easily set the size and other necessary parameters for correct resolution scaling. When the content of the render target is used by other render targets, we replace the original graphics buffer managed by OpenGL ES system with our new buffer. This approach causes extra memory overhead. However, allocating a new buffer happens only when resolution scaling is needed and the buffer is smaller than the default one for the lower display resolution. Furthermore, latest smartphones

Original Exp.	→	New Exp.
gl_FragCoord.x	→	(gl_FragCoord.x * ScalingFactor_Rec)
gl_FragCoord.y	→	(gl_FragCoord.y * ScaleFactor_Rec)
gl_FragCoord.xy	→	(gl_FragCoord.xy * ScalingFactor_Rec)

Table 1: Expression replacement in pixel-shader programs.

have a large memory size and thus this memory overhead may not be significant.

As user-defined render targets are for off-screen rendering, Step 2 in Figure 7 and thus the synchronization scheme in Figure 8 are not required in DRS for user-defined render targets.

5.3 Shader Program Issue

As aforementioned in Section 3, developers may use shader programs that are compiled and shipped to the GPU by OpenGL ES system at runtime. A shader program may obtain the position of a point and use the returned position values for further graphics processing. If the position of the point is returned in relative values to the render target size, it does not cause any problem. This is true for vertex processing. However, for pixel processing, the returned values are in the absolute-pixel coordinates. If we do resolution scaling, the returned values will be wrong, making the rendering result incorrect. This happens for both default render targets and user-defined render targets.

To address this issue, we scan all the pixel-shader programs at the initialization stage. In OpenGL ES system, the position of a point in window-pixel coordinates are stored in a built-in variable *gl_FragCoord*. We replace all the *gl_FragCoord* expressions in a pixel-shader program by multiplying the reciprocal of the scaling factor (*ScaleFactor_Rec*) used in our resolution scaling, as shown in Table 1. We set *ScaleFactor_Rec* as an input variable of the shader program, thus we can change its value on the fly, without re-compiling the shader program. Consequently, we can guarantee the correctness of the pixel processing stage by assigning proper values to the *ScaleFactor_Rec*, while display resolution changing.

6. DETERMINE DISPLAY RESOLUTION

In this section, we describe how to determine the best display resolution for maximum user experience and maximum power saving. We utilize existing knowledge of the human visual system to define the maximum observable display density based on the user-screen distance.

6.1 Human Visual Acuity and Display Density

Human visual acuity is typically measured by utilizing a Snellen or Landolt C chart, on which multiple, similar characters of set sizes are printed. Users are then asked to stand a set distance from the chart and identify the smallest characters that are legible to them. In both cases, a human adult is considered to have normal vision when they are able to separate contours that are approximately 1.75 mm apart on the chart when standing 20 feet away [42]. In order to connect the vision ability and resolvable pixels conveniently, we use angular resolving acuity to define the visual acuity. The angular size of an object can be described using the equation

$$\delta = 2\ tan^{-1}\Big(\frac{d}{2D}\Big) \tag{1}$$

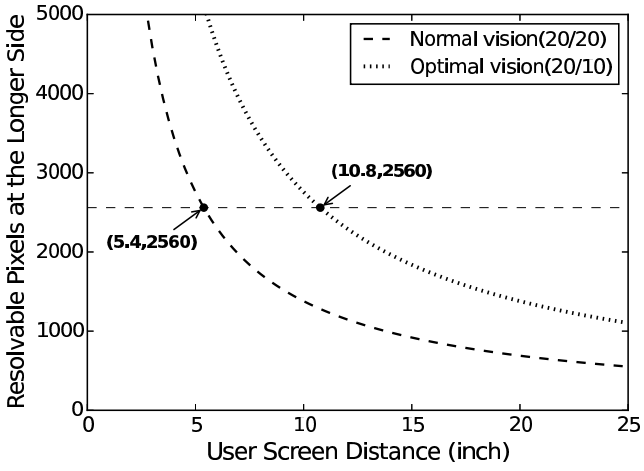


Figure 9: Relationship among resolvable pixel number, user-screen distance and user visual acuity.

where d is the actual size of an object, D is its distance from the observer(both measured with the same unit), and δ is the angular size of the object in radians. Based on Equation 1, normal vision[2] can be represented with an angular resolving acuity of $\delta_{normal} = 2.9 \times 10^{-4}$ radians, and $\delta_{optimal} = 1.45 \times 10^{-4}$ radians for optimal vision[3].

For a specific device, we consider the number of pixels at the longer side of the display as resolvable pixel number when the pixel density of it can just meet the users' visual acuity. Obviously, resolvable pixel number will changed due to different user-screen distance and different user visual acuity. The relationship among resolvable pixel number, user-screen distance and user visual acuity can be approximately described using equation

$$N = \frac{L}{2D\ tan\Big(\frac{\delta}{2}\Big)} \tag{2}$$

where N is the resolvable pixel number, L is the length of the longer side of the display, D is the user screen distance and δ is the angular resolving acuity of the user. For example, Figure 9 illustrates the variation of resolvable pixels number with user screen distance for users with normal vision and optimal vision on the Galaxy S5 LTE-A.

The Galaxy S5 LTE-A utilizes a 5.1-inch display with a resolution of 2560x1440 pixels. It has 2560 pixels on the longer side (4.44 inches). As shown in Figure 9, this high display density surpasses the visual acuity of users with normal vision when the user-screen distance beyond 5.4 inches, and 10.8 inches for users with optimal vision.

6.2 User-Screen Distance Detection

We adopt an ultrasonic-based approach for low-power user-screen distance detection. We utilize an ultrasonic ranging module HC-SR04 [5] that has an ultrasonic transmitter and an ultrasonic receiver. The module measures the distance by

[2]Normal vision is commonly referred as 20/20 vision (vulgar fraction expression) or 1.0 (decimal number expression).
[3]Optimal vision is commonly referred as 20/10 vision (vulgar fraction expression) or 2.0 (decimal number expression).

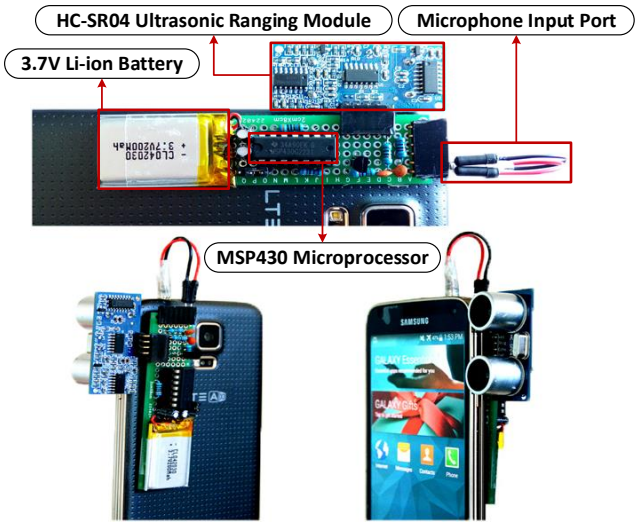


Figure 10: Ultrasonic-based hardware prototype for user-screen distance detection.

sending a 40 KHz ultrasonic signal and measures the waiting time until a reflected signal is detected by the ultrasonic receiver. The measured distances are very accurate, with a small error of only 3 mm, which is enough for our system.

Figure 10 shows our hardware prototype. We utilize an ultra-low-power microprocessor (MSP430G2231) to control the HC-SR04 module. The microprocessor simply encodes and sends the distance values measured by the HC-SR04 module to the smartphone through the microphone input headset port of the smartphone. At the smartphone side, the Resolution Controller in Figure 6 samples and records the microphone input data at 44.1 KHz rate, and then decodes the input data to get a measured distance value. Based on the distance value, the Resolution Controller can determine the desirable display resolution according to Equation 2.

This ultrasonic-based approach consumes a very low power less than 6 mW. For comparison, a camera-based approach needs more than 100 mW [37], even without considering the extra power consumption of the distance-estimation algorithm. Furthermore, this ultrasonic based approach is conservative: its measured distance is never larger than the real distance. This feature is desirable for our DRS system as we want to do conservative resolution scaling for maximum user experience.

7. IMPLEMENTATION

We have implemented our DRS prototype system on the Galaxy S5 LTE-A [17] running a Samsung-customized OS based Android 4.4.2. Like many other Android-based smartphones, the OS is not fully open source. In particular, the OpenGL ES libraries are vendor specific and not open source. Thus, we cannot modify the source code and re-build the ROM. Instead, we utilize binary-rewriting techniques to intercept the API calls.

API interception. As shown in Figure 11, there are three different ways for an application to call an OpenGL ES API function: link to the wrapper library and thus directly call the function (*path 1*); use the *eglGetProcAddress* function in EGL library to get a pointer to the function (*path 2*); and use the system *dlopen* and *dlsym* functions to dynamically load the OpenGL ES library (*path 3*). We handle all these three cases. For *path 1*, we modify the system *linker* program that is used to start application. In launching an application, our new linker program scans all the intercepted API calls and replaces the original functions using our own functions. We implement all our interception functions in a separate library. For *path 2* and *path 3*, we intercept the *eglGetProcAddress*, *dlopen*, and *dlsym* functions and change the function address returned by them to re-direct to our own interception function.

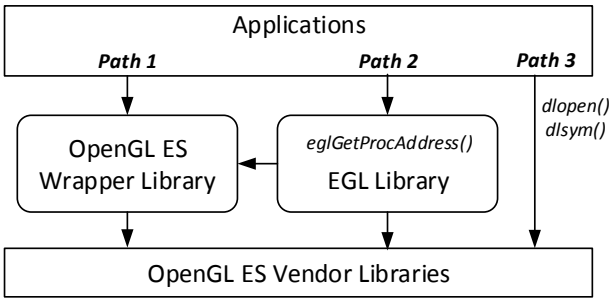


Figure 11: Three paths for applications to call OpenGL ES API functions.

EGL Library	
eglSwapBuffer	eglGetProcAddress
OpenGL ES Library	
glViewport	glScissor
glClear	glBindFramebuffer
glTexStorage2D	glTexImage2D
glBindTexture	glFramebufferTexture2D
glShaderSource	glAttachShader
glUseProgram	
HW Composer Driver Library and HAL Interface	
prepare	hw_get_module_by_class

Table 2: Intercepted functions.

SurfaceFlinger call the *prepare* function in the hardware composer driver through Android's hardware abstract layer (HAL) interface, which is packaged in the *libhardware.so* shared library. Thus, we first intercept the interface function (*hw_get_module_by_class*) in that library. Then, based on the intercepted interface function, we intercept the *prepare* function in the hardware composer driver (hwcomposer.apq8084.so).

To implement the two DRS layers, in total we intercept 15 functions in the libraries of OpenGL ES/EGL, hardware composer driver and HAL interface, as shown in Table 2. The two DRS layers use *Unix Domain Socket* for inter process communication (IPC), to implement the frame synchronization.

Determine display resolution. We use a predefined lookup table to determine the display resolution according to the user-screen distance. For minimizing the overhead, we use a set of discrete scaling factors rather than doing continuous scaling. Table 3 shows the scaling factors used in our implementation. When the distance is larger than 30 inches, we resume to the native display resolution for conservative scaling. Such a large distance may indicate that

Scaling Factors		1.0	0.9	0.8	0.65	0.5	1.0
PPI		577	519	462	375	289	577
Range(inch) Optimal Vision	Start	0.0	12.0	13.5	16.6	21.5	30.0
	End	12.0	13.5	16.6	21.5	30.0	$+\infty$
Range(inch) Normal Vision	Start	0.0	6.0	6.7	8.3	10.8	30.0
	End	6.0	6.7	8.3	10.8	30.0	$+\infty$

Table 3: Scaling factors vs. distance range.

the user is out of the sensor's detection range (±15 degrees), which rarely happens in practice though.

We configure the ultrasonic sensor to detect the distance for three times per second (3 Hz) and the Resolution Controller reads the distance value from the sensor at a 2x sampling rate of 6Hz. We smooth the detected distance values using a one-second time window for stable resolution scaling.

Lines of code. In total our implementation consists of 4,280 lines of code, including the DRS Upper Layer and Lower Layer, the Resolution Controller and its UI, the new linker program, and the code running on the MSP430 microprocessor.

Implementation on the Nexus 6. To study whether our implementation can work on other smartphones, we have experimented on a Nexus 6 smartphone that uses the same Adreno 420 GPU as the Galaxy S5 LTE-A but runs Android 5.0. Our implementation developed for the Galaxy S5 LTE-A can seamlessly run on the Nexus 6, without requiring any source-code changes, thanks to the common API layer of OpenGL. We expect that our implementation may easily work on more other Android smartphones.

8. EVALUATION

We evaluate our prototype system using a Galaxy S5 LTE-A smartphone, in terms of the application coverage, power savings in different display resolutions, system overhead, and user feedback from a user study.

8.1 Application Coverage

To study how many OpenGL ES applications our DRS system can support, we install 30 GPU-intensive applications (apps) on the S5 phone, including various 3D games and graphics benchmarks. We choose these apps for their large size and complex graphics computations. Thus, they are supposed to be the hardest apps to support. 12 out of the 30 apps have an installation package size larger than 500 MB. Two of them are extremely large, with an installation package size larger than 1,500 MB. Our DRS system is able to support all these 30 apps. That is, they run as normal with our DRS system enabled, demonstrating that our system has a very good app coverage.

8.2 Power Saving

From the 30 apps in the coverage test, we use 15 apps including 14 games and a benchmark (names listed in Table 4) to evaluate how much power can be saved in different display resolutions. We select at least one game from each game category and thus we think the games we use are representative for various gaming behaviors. For each game, we adopt a specific repeatable scene as the test case. For the *GFXBench* benchmark, we test two different scenes: *Manhattan* and *T-Rex*. Thus, in total we have 16 test cases.

We run the test cases on the S5 phone and use a Monsoon Power Monitor [3] to measure the system power. We turn

Application(Scene) Name	Scale Factor			
	0.9	0.8	0.65	0.5
Games				
Badland	95.1%	91.2%	86.7%	83.5%
Hitman Sniper	94.0%	85.5%	80.6%	73.0%
Iron man 3	94.5%	90.1%	83.4%	75.9%
Leo's Fotune	95.4%	92.8%	88.7%	84.3%
Minecraft PE	94.5%	90.4%	82.6%	76.1%
NFS Most Wanted	94.3%	89.6%	79.7%	72.3%
Over Kill 3	95.7%	92.0%	88.3%	82.5%
Ridge Racer Slipstream	94.5%	85.0%	76.8%	65.9%
Riptide GP2	95.3%	89.0%	77.5%	68.7%
Shine Runner	95.9%	90.1%	81.2%	74.2%
Smash Hit	93.9%	89.5%	81.1%	72.8%
Temple Run Brave	93.7%	85.6%	78.9%	71.2%
Tiny Troopers 2	94.4%	90.0%	82.7%	76.1%
Warships	95.3%	90.7%	84.3%	76.4%
Graphics Benchmark				
GFX Bench Manhattan	84.2%	70.4%	56.7%	39.5%
GFX Bench T-Rex	85.9%	73.1%	57.6%	48.0%

Table 4: Normalized energy per frame of games and benchmarks in different scaling factors.

the phone into airplane mode, disable unnecessary hardware components such as GPS and camera, and set the backlight brightness to 50%. We lock the GPU frequency to 500 MHz to avoid the inference of DVFS of GPU. We cool down the phone before each test to make sure that the GPU can keep working at 500 MHz for at least 60 seconds. We repeat each test for three times and report the average results.

We adopt the total system energy per frame (EPF) as the metric to evaluate the power saving of our prototype system. In order to conveniently compare the different results, we normalize the results to the case of the native display resolution. Table 4 shows the normalized EPF in different scaling factors. The scaling factors are normalized to the native display resolution, i.e., the scaling factor is 1.0 for the full resolution. When we reduce the display resolution by half (i.e., the scaling factor is 0.5, reducing the display resolution from 2560x1440 pixels to 1280x720 pixels), on average for the 16 test cases, we can reduce the EPF by 30.1%, ranging from 15.7% (in *Leo's Fortune*) to 60.5% (in *GFX-Manhattan*). For the 14 games, they always run at a fixed frame rate no matter what value the scaling factor is. Thus, the same amount of saving on power consumption can be achieved in practice (24.9% if we only count the 14 games).

For the two GFXBench cases, as the benchmark always tries to use up all the GPU processing capability, their power consumptions remain almost the same in all the scaling factors. However, as we mentioned in Section 2, the resolution can extremely influence the frame rate. For a smaller scaling factor, they may run at a higher frame rate and thus provide a better user experience.

Indeed, the display resolution of 1280x720 pixels (a.k.a 720p) is still pretty high and widely used on many mainstream smartphones. If we reduce the display resolution further, we may save more power. Furthermore, as smartphone makers are continuing the arms race on even higher display resolutions (e.g., 4K, or 3840x2160 pixels), more power-optimization opportunities can be provided by our DRS system.

Benefits from DRS for user-defined render targets. Figure 12 shows how much extra benefit can be achieved by applying DRS to user-defined graphics buffers, for the two

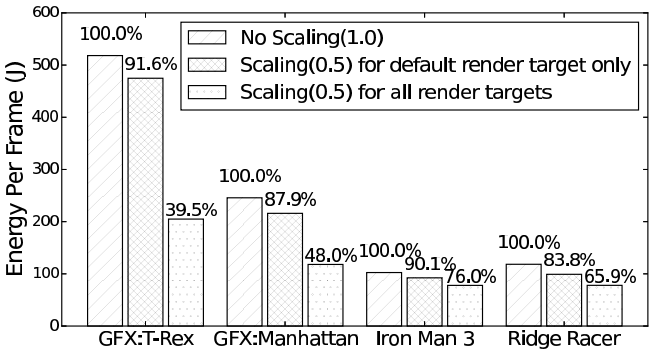


Figure 12: Benefit from scaling for user-defined render target.

Application Name	Scale Factor = 0.5
Adobe PDF Reader	90.6%
Dolphin Browser	92.4%
The Weather Channel	86.3%

Table 5: Normalized energy per frame of three non-gaming applications with halved resolution.

benchmark cases and two games (*Iron Man 3* and *Ridge Racer Slipstream*), when the scaling factor is 0.5. On average, it improves the power/energy optimization room by an extra 31.0%. The extra improvement for GFX-Manhattan is as high as 52.1% as it extensively utilize user-defined graphics buffers as the render targets. The extra saving for the two apps are relatively small but still more than 14.1%. Thus, user-defined render targets must be considered in scaling the display resolution.

Power saving of DRS for non-gaming apps. Although in this paper we focus on GPU-intensive apps, we also evaluate how much power DRS can save for non-gaming apps that are less GPU intensive than games. To this end, we measure the power consumption of three popular non-gaming apps, including *Adobe PDF Reader*, *Dolphin Browser*, and *The Weather Channel*. We use monkeyrunner[2] to generate the same sets of touch events for repeatable experiments including browsing a opened PDF file, scrolling the web page of www.yahoo.com, and updating the weather information.

Table 5 shows the results when the resolution is halved, compared to the full-resolution case. As expected, the savings are much smaller than the ones of games, only 10.2% on average. This is because that these apps generate much less GPU workload, compared to the games and graphics benchmarks. However, 10.2% of saving is still valuable, given that energy is a very scarce resource on smartphones.

In all the experiments we have conducted, we take a conservative approach, assuming that the user has the optimal vision (i.e., 2.0 in decimal number expression). If a user has the normal vision (i.e., 1.0 in decimal number expression), our DRS system may further save more power.

8.3 System Overhead

Synchronization latency. As shown in Section 5, when the scaling factor is changed, the DRS Upper Layer needs to notify the DRS Lower Layer and block the graphics rendering for a while due to the communication delay. To quantify the delay, we manually generate a large number of resolution-scaling signals into our DRS system and measure the synchronization latency. We run the test for 100 seconds. The extra latency caused by each resolution-scaling ranges from 0.41 ms to 0.94 ms. It is small, compared to the frame duration of 16.7 ms. Furthermore, in our implementation, we change the screen resolution at most for three times per second. Thus, such a small latency is negligible.

Memory overhead. Our system allocate extra memory for handling user-defined render targets. To quantify the memory overhead, we measure the extra memory used in the four games/benchmarks in Figure 12. The memory overhead depends on how many user-defined render targets are used. The *GFX:Manhattan* test has a high memory overhead of 238.8 MB because it is a benchmark program using user-defined render targets extensively. However, the memory overheads of the two real games are not large: 59.7 MB for *Iron Man 3* and 33.6 MB for *Ridge Racer Slipstream*, respectively. The *GFX:T-Rex* test also has a small memory overhead of 59.7 MB.

Energy overhead. The extra hardware we used for user-screen distance detection consumes extra energy. However, the energy overhead is small. We use the Monsoon Power Monitor [3] to measure the additional hardware's power consumption. Due to the power consumption of the additional hardware is relatively stable and repeatable, we simply regard the average power consumption of a long time (e.g., 60 seconds) as the power consumption of the additional hardware. For a distance-detecting rate of 3 Hz, we measure that the total power consumption of the ultrasonic sensor and the MSP430 microprocessor is only 5-6 mW, depending on how large the user-screen distance is. We imagine that in the future, smartphones may integrate an internal ultrasonic sensor to further reduce the power consumption of user-screen detection, as the MSP430 is not needed anymore.

8.4 User Study

To study the impact of DRS on user experience, we conducted a user study with 10 users. They are all young students and all of them have at least normal vision.

In the study, we let each user play two games, one is *Smash Hit* and the other is *Temple Run Brave*. For each game, the users play it for 10 minutes. During the 10 minutes, we randomly select the first 5 minutes or the last 5 minutes to enable our DRS system without notifying the users. We encourage the users to play the games in any way they want, changing their postures freely and trying different user-screen distances during the test. After the test, we ask the users whether they felt any differences between the first 5 minutes and the last 5 minutes. None of the 10 users could tell the difference when our DRS system was turned on or off, even though the resolution has been changed for 136 times on average for each user. This result demonstrates that our DRS system is able to remain good user experience.

9. DISCUSSION

In this section, we discuss the limitations of our current implementation and further work.

9.1 Limitations

Our implementation is limited in several aspects. First, it only supports one app for resolution scaling. While for the most of the time, smartphones typically runs only one front-

ground app, some latest smartphones such as the LG G3 [16] allow running two apps simultaneously and show their UIs side-by-side on the screen. Resolution scaling for multiple apps can be supported by extending our current implementation so that SurfaceFlinger can connect to multiple apps and do the frame synchronization with all of them. Consider that multiple apps share the display simultaneously is a rare case, the current system is already useful for many users.

Our implementation is also limited that it is only developed and tested on the Galaxy S5 LTE-A and the Nexus 6. It is not clear whether it works for other types of Android smartphones. Despite our API-interception based design is general, more efforts are needed to study how to implement the design and test it on more Android smartphones. Furthermore, it may be worthy to study how to enable DRS for other mobile OSes such as iOS and Windows Phone. It may not be hard to implement DRS in iOS, since iOS uses the same OpenGL ES as Android that is mostly platform neutral. However, Windows Phone utilizes the Direct X [9] as the graphic API, which needs extra engineering efforts.

Another limitation is that we only test the performance of our DRS prototype on Adreno 420 GPU. It is not clear how much energy we can save on other types of GPUs, without porting our implementation on them. Mobile GPU's architectures vary significantly from vendors to vendors. In terms of the graphic pipelines, ARM's Mali GPU [7] adopts TBR (Tiled Based Render) pipeline, PowerVR GPU [12] and Nvidia's Tegra series GPUs [13] adopt TBDR (Tailed Based Deferred Render) pipeline, and Qualcomm's Adreno GPU [6] adopts a flex render pipeline that can support both TBR and TBDR. In terms of the shader organization form, PowerVR GPU, Mali GPU, and Adreno GPU adopt unified shader [14], while Tegra series GPUs adopt non-unified shader. Luckily, although the architectures of different GPUs are very different, they all share the same character that the GPU workload is proportional to the number of pixels. Meanwhile, several power models for mobile GPU [34, 33, 45] have already pointed out that the GPU power consumption and GPU workload approximately have the linear relationship. Thus we believe our DRS system can achieve similar performance on other GPUs as well as it does on Adreno 420.

In addition, we add extra hardware for user-instance detection and the ultrasonic sensor is not very small. It may be an issue how to integrate an ultrasonic sensor into a smartphone. Fortunately, the emerging miniaturization techniques have already brought micrometer-level ultrasonic emitters [22] and detectors [36] into reality. It is even possible to put an ultrasonic detector into human body [39] for medical treatment. Therefore, we expect that it is not hard to integrate an ultrasonic sensor into new-generation smartphones, due to the small size and low power.

9.2 Future Work

We plan to extend our current implementation to support resolution scaling for multiple apps, and port our implementation onto more Android smartphones for more testing. In addition, we also plan to consider other opportunities to further reduce the power consumption of the GPU. Below we describe some ideas.

Scene moving speed. Existing studies and our own user study show that human visual-perceivable ability depends on the moving speed of objects. If the objects in a scene move fast, users are not as sensitive on the display resolution as they are for a slowly-changing scene. This suggests another angle besides user-screen distance for resolution scaling without compromising the user experience. We may detect the moving speed of a scene and apply more aggressive resolution scaling to save more power without compromising the user experience.

Frame rate. Another factor tightly related to GPU workload is frame rate [25, 45, 43]. The higher frame rate, the more GPU workload and thus the more power consumption. Thus, we may consider frame rate scaling for more power saving. The frame rate scaling also depends on the user-screen distance. It would be interesting to see how the right dynamic frame rate scaling strategy is designed given that we have already a costless approach to detect user-screen distance.

GPU frequency. We only qualitatively conclude that resolution down-scaling can reduce the GPU workload, but do not consider how it may affect GPU frequency. As a lower workload may trigger the GPU frequency down-scaling, it may be interesting to study how our DRS can work together with GPU DVFS for more power saving.

10. RELATED WORK

Dynamic Resolution Scaling. Dynamic resolution scaling is not a new technique, which has been already discussed and recommended to application developers to achieve better image quality and performance optimization [8]. Also this technique has already been used in real games, e.g., Witcher 3 [15], to achieve the tradeoff between resolution and framerate. However, these techniques are implemented by application developers or implemented as plugin for specific applications [10]. In this paper, we reuse the term of dynamic resolution scaling but our work is different from previous application-specific approaches. Specifically, we design a system-level dynamic resolution scaling to optimize the power consumption of smartphones. Our DRS system can achieve per-frame resolution scaling for legacy applications without any modifications of their source code and introduce minimal system overhead.

Dalton *et al.* [24] proposed to use camera to detect the presence of users to turn on or off the display to save power. This work can be considered as an extreme case of DRS but it does not provide a full DRS system and the camera-based approach is less useful for smartphones due to the high power consumption.

GPU power optimization. There have been related works on GPU power modeling [32, 35, 33], power consumption characters [38] and optimization mainly for desktop/server, which mainly resort to dynamic frequency scaling [41, 35, 19, 27, 48]. These techniques are generally applicable to smartphones. Usually, GPU frequency scaling is controlled in the vendor-provided drivers, and it is hard to be configured by applications even at the operating system level. In this paper, we optimize GPU power from another angle via dynamic resolution scaling.

Analysis on the PC gaming experience is presented [23, 40] when display resolution and refresh rate is changed. It is reported that lower frame rates may lead to system power reduction [25, 45, 43]. In this paper, we further give the detailed quantitative analysis among GPU power and display resolution, and present specific techniques on dynamic resolution scaling without changing the app source code.

Tradeoff between power saving and user experience. Techniques such as frequency scaling usually do not consider the user experience, either poor user experience with aggressive frequency scaling or poor power saving with conservative frequency scaling. Huge opportunities exist if we take into consideration the tradeoff between power saving and user experience. Previous analysis only discusses the relationship between power saving and display resolution scaling [23, 40] or frame rate scaling [25, 45, 43], however, they did not discuss the relationship between user experience and resolution or frame rate scaling. In this paper, quantitative analysis between resolution and user-screen distance is presented with user experience preserved, and the analysis framework can also guide app developers or users to decide the right tradeoff if they tolerate certain degree of downgraded user experience.

Nixon *et al.* [44] reported that the display on some modern devices already exceeds human perceptive capabilities and studied the relationship between GPU power and display resolution on the Nexus 4 and 5. They share the same motivation with us. However, they did not provide any end-to-end implementation and evaluation.

11. CONCLUSION

In this paper, we have built the first end-to-end DRS system for smartphones. By intercepting the function calls of OpenGL ES/EGL, our system requires no changes from applications and is able to work on existing smartphones without re-building a new ROM. We design an ultrasonic-based, low-power approach for user-screen distance detection. Based on the measured user-screen distance, our system can enable automatic resolution scaling for maximum user experience and power saving. Experimental results and user feedback demonstrate that our system is able to support a large set of real applications, provide per-frame and real-time resolution scaling with low system overhead, and retain good user experience.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers and shepherd for their insightful and constructive comments that helped us improve this paper. We also thank Kent W. Nixon for his help on the early investigations and experiments of this work.

12. REFERENCES

[1] GFXBench 3.0, Unified cross-platform 3D graphics benchmark . http://gfxbench.com/.

[2] Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[3] Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[4] RidgeRacerSlipstream. http://www.ign.com/prime/promo/ridge-racer-slipstream-free.

[5] Ultrasonic Ranging Module HC - SR04. http://www.micropik.com/PDF/HCSR04.pdf.

[6] Adreno GPU. https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu, 2015.

[7] ARM Mali Graphics. http://www.arm.com/products/multimedia/mali-graphics-hardware/, 2015.

[8] Dynamic Resolution Rendering. https://software.intel.com/en-us/articles/dynamic-resolution-rendering-article, 2015.

[9] Graphics and Gaming. https://msdn.microsoft.com/en-us/library/windows/desktop/ee663279(v=vs.85).aspx, 2015.

[10] Hialgo Boost. http://www.hialgo.com/TechnologyBOOST.html, 2015.

[11] OpenCL - the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/, 2015.

[12] PowerVR GPU. http://www.imgtec.com/powervr/graphics.asp, 2015.

[13] Tegra GPU. http://www.nvidia.com/object/tegra-4-processor.html, 2015.

[14] Unified Shader Model. https://en.wikipedia.org/wiki/Unified_shader_model, 2015.

[15] Witcher 3 uses dynamic resolution scaling on Xbox One to hit 1080p. http://www.extremetech.com/gaming/205487-witcher-3-uses-dynamic-resolution-scaling-on-xbox-one-to-hit-1080p, 2015.

[16] LG G3. http://www.lg.com/us/mobile-phones/g3, Feb. 2015.

[17] Samsung Galaxy S5 LTE-A. http://www.samsung.com/us/news/23327, Feb. 2015.

[18] Apple iPhone. http://www.apple.com/iphone/features/retina-display.html, Jun. 2010.

[19] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato. Power and performance analysis of gpu-accelerated systems. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems, ser. HotPower*, volume 12, pages 10–10, 2012.

[20] A. O. Akyüz. High dynamic range imaging pipeline on the gpu. *Journal of Real-Time Image Processing*, 10(2):273–287, 2012.

[21] L. Bavoil. Advanced soft shadow mapping techniques. In *Presentation at the game developers conference*, volume 2008, page 11, 2008.

[22] L. Belsito, E. Vannacci, F. Mancarella, M. Ferri, G. P. Veronese, E. Biagi, and A. Roncaglia. Fabrication of fiber-optic broadband ultrasound emitters by micro-opto-mechanical technology. *Journal of Micromechanics and Microengineering*, 24(8):085003, 2014.

[23] M. Claypool and K. Claypool. Perspectives, frame rates and resolutions: it's all in the game. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 42–49. ACM, 2009.

[24] A. B. Dalton and C. S. Ellis. Sensing user intention and context for energy management. In *HotOS*, pages 151–156, 2003.

[25] M. Dong and L. Zhong. Power modeling and optimization for oled displays. *Mobile Computing, IEEE Transactions on*, 11(9):1587–1599, 2012.

[26] R. Fernando. Chapter 3 and 4. In *GPU gems*. Addison-Wesley Professional, 2004.

[27] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 826–833. IEEE, 2013.

[28] K. Group. Khronos Native Platform Graphics Interface (EGL Version 1.4 - February 11, 2013). `https://www.khronos.org/registry/egl/specs/eglspec.1.4.20130211.pdf`, Feb. 2013.

[29] K. Group. OpenGL ES – The Standard for Embedded Accelerated 3D Graphics. https://www.khronos.org/opengles/, Jan. 2013.

[30] K. Group. The OpenGL ES Shading Language (Specification). `https://www.khronos.org/files/opengles_shading_language.pdf`, May. 2009.

[31] K. Group. OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification). `https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`, Nov. 2010.

[32] S. Hong and H. Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[33] T. Jin, S. He, and Y. Liu. Towards accurate gpu power modeling for smartphones. In *Proceedings of the 2nd Workshop on Mobile Gaming*, pages 7–11. ACM, 2015.

[34] Y. G. Kim, M. Kim, J. M. Kim, M. Sung, and S. W. Chung. A novel gpu power model for accurate smartphone power breakdown. *ETRI Journal*, 37(1):157–164, 2015.

[35] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41(3):487–498, 2013.

[36] H. Li, B. Dong, Z. Zhang, H. F. Zhang, and C. Sun. A transparent broadband ultrasonic detector based on an optical micro-ring resonator for photoacoustic microscopy. *Scientific reports*, 4, 2014.

[37] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 69–82. ACM, 2013.

[38] X. Ma, Z. Deng, M. Dong, and L. Zhong. Characterizing the performance and power consumption of 3d mobile games. *Computer*, (4):76–82, 2013.

[39] T. Maleki, N. Cao, S. H. Song, C. Kao, S.-C. A. Ko, and B. Ziaie. An ultrasonically powered implantable micro-oxygen generator (imog). *Biomedical Engineering, IEEE Transactions on*, 58(11):3104–3111, 2011.

[40] J. D. McCarthy, M. A. Sasse, and D. Miras. Sharp or smooth?: comparing the effects of quantization vs. frame rate for streamed video. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 535–542. ACM, 2004.

[41] X. Mei, L. S. Yung, K. Zhao, and X. Chu. A measurement study of gpu dvfs on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, page 10. ACM, 2013.

[42] E. Messina. Standards for visual acuity. *National Institute for Standards and Technology*, 2006.

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

[44] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen. 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*, pages 5–5. USENIX Association, 2014.

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

[46] C. Wynn. Opengl render-to-texture. *GDC. NVIDIA Corporation*, 2002.

[47] T.-T. Yu. Depth of field implementation with opengl. *Journal of Computing Sciences in Colleges*, 20(1):136–146, 2004.

[48] Y. Zhu, A. Srikanth, J. Leng, and V. J. Reddi. Exploiting webpage characteristics for energy-efficient mobile web browsing. *Computer Architecture Letters*, 13(1):33–36, 2014.