Tell Your Graphics Stack That the Display Is Circular

Hongyu Miao
Purdue ECE
miaoh@purdue.edu

Felix Xiaozhu Lin
Purdue ECE
xzl@purdue.edu

ABSTRACT

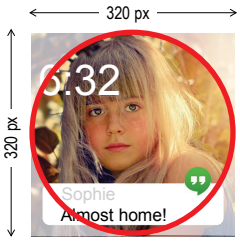
Computer displays have been mostly rectangular since they were analog. Recently, smart watches running Android Wear have started to embrace circular displays. However, the graphics stack – from user interface (UI) libraries to GPU to display controller – is kept oblivious to the display shape for engineering ease and compatibility; it still produces contents for a virtual square region that circumscribes the actual circular display. To understand the implications on resource usage, we have tested eleven Android Wear apps on a cutting edge wearable device and examined the key layers of Android Wear's graphics stack. We have found that while no significant amount of CPU/GPU operations are wasted, the obliviousness incurs excessive memory and display interface traffic, and thus leads to efficiency loss.

To minimize such waste, we advocate for a new software layer at the OpenGL interface while keeping the other layers oblivious. Following the idea, we propose a pilot solution that intercepts the OpenGL commands and rewrites the GPU shader programs on-the-fly. Through running a handcrafted app, we show a reduction in the GPU memory read by up to 22.4%. Overall, our experience suggests that it is both desirable and tractable to adapt the existing graphics stack for circular displays.

1. INTRODUCTION

Computer displays have been rectangular for a long time. This has been recently changed as wearable computers enter people's daily lives: on these computers, the display's role of content presentation is gradually giving way to the aesthetic value or human factor consideration. In embracing non-rectangular displays, smart watches are among the pioneers. Since Moto 360 sparked the trend in 2014, more than ten smart watch models have featured circular displays. The trend is burgeoning.

Compared to the fast-evolving display, the graphics stack that backs the display is mostly retaining the legacy implementation for engineering ease. Figure 1 shows the case of Android Wear 5.1 "Lollipop" atop the LG watch R [9], a cutting-edge smart watch featuring a round display. Despite the circular display, an app is responsible for generating contents for a virtual 320×320 square area that circumscribes the actual display as shown in Figure 1(a); in addition, all the drawing API available to apps is based on rectangular regions of configurable sizes. To ensure the UI integrity, the app developers are expected to be aware of the display shape and not to place the important UI elements off the circular boundary [5]. Underneath the apps, the graphics stack – from UI libraries to GPU to display controller – is oblivious to the circular shape of the display; they mechanically transform the app's UI contents to the pixels targeting the square area. The excessive pixels, as shaded in Figure 1(a), are only discarded at the display panel – the lowest layer of the graphics stack.



(a) The produced (in square) vs. displayed contents (in circle)

CPU: 4x Coretx-A7 @ 1.2 GHz	
GPU: Adreno 305 @ ~400 MHz	
Memory: 512 MB	
Display: 1.3'' round OLED. Diameter 320px	
p: Percentage of hidden pixels	21.5%
f: Frames rate	60 fps
S: Frame size	400 KB

(b) Device hardware specs

Figure 1: The LG watch R, a circular smart watch, running the Android Wear Lollipop

Clearly, there exists a mismatch between a circular display and the graphics stack. While this mismatch is acquiesced by today's wearable OS as-is, we seek a clear understanding of it by asking the following two questions.

1. How many resources are wasted in producing the contents that are ultimately discarded due to the circular display shape?
2. If the waste is non-trivial, how should an existing graphics stack adapt accordingly?

To answer these questions, we have examined a set of eleven typical Android Wear apps; we run them on a LG watch R and analyze the graphics stack's key layers. Through examining empirical evidence collected on the device, we have quantified the wasted resources: CPU/GPU opera-

HotMobile '16, February 26-27, 2016, St. Augustine, FL, USA

DOI: http://dx.doi.org/10.1145/2873587.2873603

tions, memory bandwidth/capacity, and interconnect traffic. Our findings are twofold:

- While the circular display shape does not affect an app's UI hierarchy much, it often discards a substantial portion (up to 21.5%) of background textures and rendered images.
- Accordingly, while no significant amount of CPU or GPU operations are wasted, the excessive memory and interconnect traffic – as high as 25 MBps – have led to a noticeable efficiency loss.

The findings suggest the direction towards adapting a graphics stack to a circular display: while revising the UI libraries is disruptive and unlikely to be profitable, making the textures and rendered surfaces aware of the screen shape is likely rewarding.

To this end, we explore the design space for introducing the awareness of display shape to the graphics stack; we find that interposing OpenGL, the low-level interface between apps and the GPU driver, is a viable approach. Hence, we propose to build a new software layer that rewrites the OpenGL commands and shader programs on-the-fly to fit the circular display, while keeping all other layers oblivious and unmodified. A simple prototype following this idea has reduced the GPU memory read by 21.5% and reduced the GPU cycles by up to 12%.

We have made two major contributions in this paper:

1. Through examining a set of typical wearable apps, we have quantified the resource waste due to the graphics stack's obliviousness to the circular display.
2. Towards eliminating the resource waste, we have shown that interposing the OpenGL interface is a promising approach: we demonstrate a pilot solution that rewrites the shader program and hence skips unnecessary GPU texture loading.

2. BACKGROUND & MOTIVATION

We next describe our test device, overview Android Wear's graphics stack, which inherits its overall structure from Android for smartphone, and discuss the existing system support for circular displays.

Test device. Our test device is the LG Watch R [9], one of the most popular smart watches. As summarized in Figure 1(b), the device features an LG 4237 OLED panel with a diameter of 320 pixels; it embraces a Qualcomm's APQ8026 SoC engineered towards low power. By running the STREAM benchmark [12] with all four cores at their highest frequency, we have measured the memory bandwidth as 1.8 GBps, which is much lower than smartphone memory bandwidth (often 5–10 GBps). Our prior work [10] has reported the system-level power consumption of Watch R.

Android graphics stack. In a nutshell, a graphics stack's role is to translate the app UIs to pixels shown on the display. On modern mobile devices, this procedure is heavily hardware-accelerated: multiple hardware components – CPU, GPU, and display controller – collaborate to repeat this procedure periodically, aiming at keeping pace with the display refresh rate, often 60 Hz. In the collaboration, these hardware components communicate mainly through shared buffers residing in the off-chip DRAM. Each buffer often contains information for a *rectangular* region on the display.

The graphics stack produces a frame of image in three stages from the top to bottom as shown in Figure 2. *Drawing* is done by CPU: each app transforms its hierarchy of UI elements, i.e. "view"s, to display lists, an intermediate representation of the final GPU commands. For every frame, the app translates its current display lists to OpenGL commands and sends them to the GPU driver for execution.

Rendering is done by GPU: controlled by the driver, the GPU executes the CPU-generated commands in its pipeline. It first performs vertex/primitive processing by running shader programs on the input vertices and assembling the vertices into triangles. The GPU then performs rasterization: it breaks down the triangles into fragments, runs shader programs on the fragments, and samples textures to determine each fragment's color. It finally performs pixel processing, writing back the produced pixels to the DRAM.

Composition and display are done by compositor and the display controller: once a surface is rendered by the GPU, the owner app passes it to an OS daemon called SurfaceFlinger. SurfaceFlinger blends surfaces passed from multiple apps by invoking hardware composer, which often resides in the hardware display controller. After composition, the display controller transmits the final frame to display panel for presentation.

Existing system support for circular displays. As described in Section 1, we have observed that the Android Wear OS is mostly oblivious to the circular shape of a display; as illustrated in Figure 1, it functions by assuming that the physical display is a square that circumscribes the actual circular display.

Multiple pieces of evidence support our observation: *i)* in examining the OS's UI framework source (which is fortunately open), we always see that drawable regions are specified as rectangles; *ii)* in peeking the test device's rendered surfaces or framebuffers by using Android's `GLtracer`, we always get the circumscribed square images; *iii)* the kernel's device tree specifies the dimensions of the display panel as 320×320 (shown below); this specification is the display controller driver's only knowledge about the panel.

```
//apq8026-lenok-panel.dtsi:
qcom,mdss-dsi-panel-name = "LG4237 320P OLED
    command mode dsi panel";
qcom,mdss-dsi-panel-width = <320>;
qcom,mdss-dsi-panel-height = <320>;
```

Listing 1: An exerpt from the Linux kernel (3.10) device tree for Watch R's circular display panel

The app developers, on the other hand, are expected to make their app UI layout "shape-aware" [5]. In particular, the developers are asked to place important UI elements in a square area called "window insets" that *inscribes* the circular display, to prevent these UI elements from being clipped by the circular display edge.

3. HOW MANY RESOURCES ARE WASTED?

We next quantify the wasted resources due to the graphics stack's obliviousness to the display shape. In the discussion,

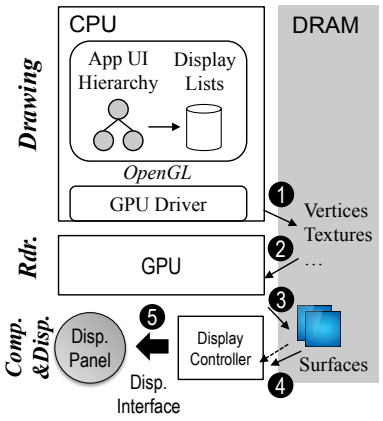


Figure 2: An overview of the graphics stack

Apps	# of UI Views			Drawing Time				Rdr. Time
	Hidden	Clipped	Total	Shader compile†	Shader link†	Texture upload†	Other cmds	
Google keep	×	×	×	8.6	1.3	4.4	2.9	4.3
Attopedia	0	9	10	8.2	1.2	25.0	2.4	4.5
Hole19	0	5	8	30.4	1.1	4.9	4.1	2.6
WearbottleSpinner	0	4	5	18.0	3.2	116.2	2.1	3.0
GridViewPager	0	6	9	23.9	4.4	2.0	2.0	2.8
*Runtastic**	0	14	17	-	-	-	-	3.9
*ReminderByTime**	0	13	14	-	-	-	-	3.8
*Fit**	0	13	16	-	-	-	-	3.3
*Weatherlive**	0	14	17	-	-	-	-	4.6
*Instaweather**	0	13	16	-	-	-	-	3.8
*Hangout**	0	13	16	-	-	-	-	3.7

* Wearable app "Cards" † Mostly occur upon surface creation
- Unable to determine: GLTracer unable to launch app Cards × Unable to determine: app crash
Hidden/Clipped: how many views in the UI are hidden/clipped Rdr. Time: the time of rendering an UI
Note: shader compile, shader link, and texture upload are the most expensive actions in UI drawing

Table 1: A list of studied wearable apps. All time values in ms.

Drawing (CPU)	Memory traffic: 225 KB per app bkgnd Memory capacity: minor	❶
Rendering (GPU)	Memory traffic: 10 MBps Memory capacity: 1 MB	❷ ❸
Composition (Disp. Ctrl & panel)	Memory traffic: 10 MBps Display interface traffic: 5 MBps Memory capacity: < 100 KB	❹ ❺

Table 2: A summary of the estimated resource waste, which is more than 3× of the inefficiency addressed in a prior wearable work [7]. Note that the wasted CPU/GPU operations are insignificant.

we use "oracle" to refer to an imaginary, ideal graphics stack tailored to the circular display: it wastes zero resource on pixels falling out of the display edge.

App study. In order to understand the UI structure of wearable apps, we studied eleven wearable apps that are popular in the app store as summarized in Table 1. These apps represent typical interactive use of smart watches and all feature image background for visual appeal. Focusing on graphics, we use Android's `Hierarchy Viewer` to examine the complexity of the UI hierarchies and how individual UI elements intersect with the circular display edge. We further measure the time spent in producing typical frames of the apps: we use `GLTracer` to measure the drawing time and use `dumpsys` to measure the CPU-perceived rendering time.

Based on the app study, we make three observations. First, a wearable app tends to have a simple UI hierarchy with a small number of views, as shown in the "Total" column of Table 1. Second, in such a hierarchy, although the views are often clipped by the circular display edge, they are rarely hidden, which is shown in the "Clipped" and "Hidden" columns of Table 1. We attribute this to the developer's consciousness of the display shape. Third, in initializing a UI surface, drawing is often expensive, as shown in the "Drawing" columns of Table 1, while in other frames it is cheap. The rendering cost is steadily moderate, as shown in the "Rdr. Time" column of Table 1.

Motivated by these observations, we next examine the graphics stack's major layers as shown in Figure 2. For each layer, we discuss the key operations and estimate how a circular display may affect the associated resource usage. Table 2 summarizes the resource waste.

3.1 Drawing

As described in Section 2, drawing is mainly done in the UI libraries and GPU driver. The associated major costs are three: *i)* transforming UI hierarchy to display lists; *ii)* compiling GPU shader code; *iii)* decoding and uploading textures to GPU. We examine these in details below.

App UI library. One app maintains a mapping between its view hierarchy and a set of Display Lists. In order to draw its UI, the app traverses the view hierarchy to update the corresponding Display Lists. When the view hierarchy is initialized or undergoes a significant change, the app has to rebuild substantial Display Lists. This is often expensive due to sophisticated UI measurement and layout [4]. However, for most frames, in particular during UI animation, existing display lists can be incrementally updated and translated to GL commands with low cost.

GPU driver. Once GL commands are ready, the app submits them to the GPU driver, which will operate the GPU accordingly. Since the GPU execution is mostly asynchronous, the cost to CPU is often minor except for two actions: compiling the GPU shader programs just-in-time and uploading textures to the GPU-owned memory region. This is shown in the "Shader compile" and "Texture upload" columns of Table 1. Fortunately, the two expensive actions often only occur upon the creation of UI surfaces; for other frames, the GPU driver execution only takes around 2–4 ms, as shown in the "Other cmds" column of Table 1.

Impact of a circular display. The circular display has little impact on the execution of app UI libraries (cost *i* above). The oracle solution is unlikely to see a reduced cost of transforming UI hierarchy: as few views are hidden by the display edge, the oracle's view hierarchy will not be any simpler. On the contrary, by catering to the circular edge, the oracle may even see an increase in the overhead of UI measurement and layout.

For the execution of the GPU driver, the oracle solution

will compile the same shader programs, paying the same cost *ii*. Its texture handling (cost *iii*), however, may decode smaller images and move fewer bytes in uploading the textures (❶). Assuming the uploaded texture has a size of T, the wasted memory traffic is $T \cdot p$. For a typical 512×512 texture used as a wearable app background, the waste is around 225 KB.

3.2 Rendering

In rendering a UI surface, the GPU processes the CPU-generated commands and data, produces triangles, rasterizes them, and fills the resultant fragments with colors from textures.

GPU execution. As shown in the "Drawing Time" columns of Table 1, the GPU execution is on the critical path of producing every frame. Given the observed simple UI hierarchy, we expect that the triangle count in each frame is low, implying light vertex/primitive processing. Furthermore, we expect much of the execution overhead comes from rasterization, in particular the entailed memory access. Since mobile GPU is backed by the off-chip DRAM, it has been demonstrated that a mobile GPU is often memory-bound and the memory access also becomes efficiency hotspots [3]. Note that due to the lack of hardware cache coherence among CPU, GPU, and display controllers, all memory sharing among them has to go through the external DRAM.

Among the memory access overheads, the following ones are tied to the circular screen shape:

Texture reading (❷): To output pixels, the GPU fragment shader loads texture and determines the pixel colors accordingly. Based on our wearable app study, the main use of textures is as the app background. Such a texture is often hundreds of KB and cannot be held in a mobile GPU's texture cache which is often a few KBs [2]. As a result, the texture access is streaming and most texture data have to be fetched from the DRAM [3]. Note this may not significantly harm the GPU performance much: the DRAM latency is hidden by massive parallelism and various optimizations such as batch fetch. Yet, the resultant excessive memory move leads to an efficiency loss.

Pixel operations (❸): The GPU produces a rendered surface by writing the final pixels back to the DRAM. To do so, it may need to first read in the pixels before writing them back, e.g. for implementing the z-order among multiple objects or a translucent effect.

To estimate resource waste, we make the following simple, conservative assumptions: for each frame, the GPU renders at least one surface, i.e. the one belonging to the on-screen app, and thus accesses the background texture that is uncompressed; in pixel operations, it writes one rendered surface exactly once. Using the notations defined in Figure 1(b), the wasted memory traffic is:

$$2 \cdot S \cdot f \cdot p = 2 \cdot 400KB \cdot 60FPS \cdot 21.5\% = 10MBps$$

Graphic buffers. The wasted graphics buffer capacity is low. The memory cost comes from two parts: the texture buffers and the rendered surfaces for individual apps. Note that each surface is often double- or triple-buffered; that is, each buffer has two to three copies.

By examining the debugging information of SurfaceFlinger, the system daemon managing all surface buffers, we found the whole system often sees ten surface buffers, occupying about 4MB memory. As a result, the oracle solution can save at most 21%, or around 1MB, memory. This is less than 1% of the total device DRAM (512MB). We expect that the saving from the texture buffers is even lower.

3.3 Composition and Display

As shown in Figure 2, SurfaceFlinger sends multiple rendered surfaces to the display controller, which in turn composites the surfaces and directly sends the final image to the display panel over a display interface. Note that this eliminates the legacy notion of "framebuffer". These actions matter to energy efficiency: LPD [7] shows that on a similar wearable device (Samsung Gear S), the memory access and data move consume up to 14 mW or 8% of the system power.

Composition. Based on our experiment with SurfaceFlingers when running various apps, it almost always composites two surfaces: one from the on-screen app and one OS surface, i.e. a black circle around the display's edge for anti-aliasing effect. As compared to smartphone, a watch has no system bar, no action bar, etc. Both surfaces are rectangular, in the same size of the circumscribed square. As a result, using the notations defined in Figure 1(b), the excessive memory traffic flowing into display controller (❹) is given by:

$$2 \cdot S \cdot f \cdot p = 2 \cdot 400KB \cdot 60FPS \cdot 21.5\% = 10MB/sec$$

Beyond the memory move overhead, we estimate that the compute demand by composition is low: the overlay engine has been specialized for bit manipulation, and the number of overlays is as small as two. Hence, the oracle solution can save little processing in composition.

Display interface traffic. Each second, the display controller sends 60 frames to the display panel (❺). Each pixel, on the interface, is still represented by four bytes. Thus, the wasted interface traffic is:

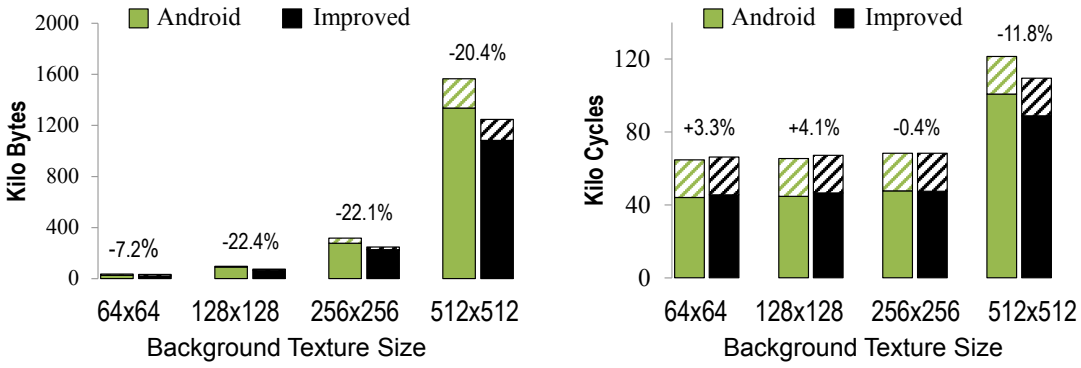
$$S \cdot f \cdot p = 400KB \cdot 60FPS \cdot 21.5\% = 5MB/sec$$

Display panel's internal SRAM. Modern display panels often use command mode [7], which holds the pixels being displayed in an internal SRAM . Although the display panel's specification (§2) presents an illusionary, 320 × 320 pixel array that takes 400 KB, it is unclear to us how its internal SRAM is organized. Ideally, it should only store pixels that will be visible on the circular display (292 KB).

4. HOW SHOULD SOFTWARE ADAPT?

Design space exploration. To reduce the resource waste, it is clear that the graphics stack should be made aware of circular displays. This raises a top design question: to which layer of the stack (shown in Figure 2) should we introduce the awareness? This design question involves a key trade-off between the entailed resource saving and software complexity.

At the very top of the stack, we may overhaul the framework API and make the app code fully aware of the display shape. In drawing their UI elements, apps explicitly confine all UI hierarchies within the circular area; they trim their



(a) Memory read per frame **(b) Total number of cycles per frame**

Figure 3: The measured GPU resource consumption in rendering the benchmark app. In each column, the bottom part: the resource for rendering the background; the top part: that for all the remaining UI elements

shapes and textures so that none overflows. This approach is very similar to the oracle solution discussed in Section 3. It almost eliminates the resource waste, but puts a high burden on the app developers; even worse, wearable apps developed in this fashion are not portable across devices with different shapes of displays.

Going one layer lower, we may keep the apps oblivious but make the UI libraries aware. The apps still believe that they lay their UI hierarchies on a square display; the UI libraries keep track of the intersections between a hierarchy and the display boundary and therefore avoid producing display lists that fall out of the boundary. This approach removes developer's burden but complicates the UI libraries (which are already complicated! [4]) by exposing the display shape to them.

Going all the way down to the bottom layer, we may introduce the awareness right before all rendered surfaces are composited, making the display controller skip all out-of-boundary regions in the surfaces. This approach is similar to LPD [7]. It requires zero effort from the developers of CPU and GPU programs, but only reduces waste at ❸ and ❹; the excessive GPU-induced memory traffic (❷) has already been wasted.

Pilot solution: OpenGL interposition. In the graphics stack, we have observed that the OpenGL interface is ideal for bringing the display shape awareness: on one hand, its abstraction is low enough for granting us great control of the rendering action; on the other hand, it is still atop the GPU where the major waste occurs. To this end, we advocate for interposing the OpenGL commands and GPU shader programs before they are sent to the GPU driver. We keep the rest of the graphics stack oblivious and unchanged.

Shader program rewriting. We next showcase that a simple rewrite action of a shader program can effectively reduce GPU's memory traffic. The idea is simple: we manually rewrite the GPU fragment shader so that it checks each fragment's coordinates before coloring it; if a fragment falls out of the circular display area, the shader skips coloring it.

To evaluate the benefit of shader rewriting, we build a small benchmark app for the test wearable device; the app invokes the OpenGL API and employs GPU shaders to draw its pictorial background. We vary the size of the background's texture during benchmarking. Following the idea above, we manually rewrite its fragment shader and profile the GPU resource usage with and without the rewrite. The profiling, however, faces a platform limitation: Qualcomm's GPU profiler only works with handheld devices but not wearables at the time of writing (Oct. 2015). As a workaround, we use Nexus 5 – a Qualcomm-powered smartphone – as an emulation platform: we compile the benchmark wearable app for Nexus 5 while keeping its source code, UI structure, and UI dimension exactly identical. Note that we do emulation on real hardware instead of simulation. Nexus 5 and Watch R both feature Qualcomm's Adreno GPUs from the same generation (28 nm) and only differ in the amount of physical compute resources.

As shown in Figure 3(a), our profiling results show that the rewritten shader reduces the memory read by up to 22.4%. This reduction and the resultant efficiency gain are notable: *i)* memory read is in general known as the major efficiency bottleneck in mobile graphics [2]; *ii)* the energy efficiency of a wearable device is known to be sensitive to memory traffic [7]; *iii)* the efficiency benefit applies to every produced frame, even when the UI is not updated by the app and remains still; *iv)* the gain is achieved through very low engineering effort, which is in contrast to far more sophisticated techniques, e.g., texture compression, towards the same goal.

The execution overhead introduced by the rewritten shader is low, as shown in Figure 3(b). With small to medium background textures ($\leq$128×128), the coordinates check inserted in the shader increases the total GPU cycles by less than 5%; with larger textures, the performance benefit from memory read reduction overshadows the execution overhead, leading to an up to 12% reduction in GPU cycles.

Expected Power Saving. We estimate that the pilot solution can save non-trivial power by comparing it with prior wearable study: LPD [7] saves 2.7 mW of system power by reducing the DRAM-to-display traffic by 7 MBps; assuming that the saved power is roughly proportional to the reduced traffic, our reduced DRAM traffic (❷❸) will lead to 3.9 mW of system power reduction. It is worth noting that *i)* such an amount of power matters as wearable battery is tiny; *ii)* further power saving of 5.8 mW may be harvested by novel display controller hardware that avoids the traffic waste shown in Table 2 (❹❺); *iii)* our power saving is orthogonal to LPD [7], which avoids refreshing a display region that is not re-drawn.

5. RELATED WORK

Wearables are less understood as compared to smartphones. Our own work [10] has characterized its major system aspects such as power and CPU usage; Min *et al.* [13] studies the battery usage of smart watches. However, none has studied the unique aspects of a wearable's display.

Much work has characterized mobile GPUs. For example, GraalBench [1] is a 3D benchmark suite for low-end phones and Ma *et al.* [11] has characterized the power consumption of mobile games. While their methods are inspiring, none has studied non-rectangular displays nor has examined the hardware-accelerated wearable UI.

Minimizing waste of mobile GPU resources has been a hot topic. LPD [7] reduces the memory and display interface traffic by only compositing the recently changed UI regions. Android comes with various tools to bust GPU overdraw problems [6]. DRS [8] reduces the GPU compute demand by scaling down the display resolution on-demand. Our goal is orthogonal to them: we aim to avoid producing UI regions that will be hidden by circular displays.

Memory access by mobile GPU, in particular texture memory, is a known bottleneck. Targeting power efficiency, PFR [3] improves the texture access locality by rendering two adjacent frames in parallel. While we share the goal of efficiency in GPU memory access, our approach avoids unnecessary texture memory access altogether.

6. CONCLUDING REMARKS

Limitations. Our work does not provide solution for all the discovered issues. The proposed shader rewrite technique does not reduce the identified resource waste in composition and display. To address this issue, hardware support from the display controller and panel is needed.

Our study is also limited by multiple factors. First, the immature profiling support for wearable GPU forces us to use a smartphone for emulation. The resultant measurement, while shedding lights, does not directly map to that of a wearable GPU. For instance, we have observed that the wearable GPU often causes long system-level delay, up to hundreds of ms, as the user is navigating among UI surfaces. While this implies optimization opportunity, lacking profiling support, we are unable to examine what the GPU is busy with. Second, multiple popular wearable apps crash while they are being debugged or profiled, hindering a better understanding of their internals. These even include Google's official apps and official profiler. Third, the fine-grained power model of wearables is not yet well-known, making it hard to estimate the efficiency gain in actual use. With more efforts invested by industry and academia, we expect these limitations to disappear.

High-resolution, non-rectangular display. Our study addresses the smart watch display, on which the number of pixels is small, $20-40\times$ smaller than that of a typical smartphone display. Down the road, as displays in various form factors become pervasive, we may see larger non-rectangular displays with massive pixels, e.g., a "smart" oval mirror. The implications are two: based on the discussion in Section 3, the resource waste is likely higher, making it more compelling to adapt the graphics stack; addressing the waste may warrant an introduction of the complexity to higher layers in the graphics stack, e.g., redesigning the API.

Conclusions. Our work is a first look at the implication of circular displays on system software design. We have discovered that the existing graphics stack is wasting substantial resources for contents that will never been shown due to the screen shape. To this end, we have quantified the resource waste on the LG watch R. We advocate for interposing the OpenGL commands and GPU shaders to adjust rendering activities to the display boundary. By demonstrating a benchmark app, we have shown that this approach is promising.

Acknowledgement

This work was supported in part by NSF Award #1464357. The authors thank the anonymous reviewers for their useful feedbacks. The photo in Figure 1(a) is in CC0 public domain, and is by courtesy of Pixabay.

7. REFERENCES

[1] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha. Graalbench: A 3d graphics benchmark suite for mobile phones. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004.

[2] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[3] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013.

[4] Google. Android graphics: System-level architecture. https://source.android.com/devices/graphics/architecture.html, 2014.

[5] Google. Android wear – defining layouts, 2015.

[6] R. Guy. Android performance case study, 2012.

[7] M. Ham, I. Dae, and C. Choi. Lpd: low power display mechanism for mobile and wearable devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, 2015.

[8] S. He, Y. Liu, and H. Zhou. Optimizing smartphone power consumption through dynamic resolution scaling. In *Proc. ACM MobiCom*, 2015.

[9] LG USA. Design comes full circle. http://www.lg.com/us/smart-watches/lg-W110-g-watch-r, 2014.

[10] R. Liu, L. Jiang, N. Jiang, and F. X. Lin. Anatomizing system activities on interactive wearable devices. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015.

[11] X. Ma, Z. Deng, M. Dong, and L. Zhong. Characterizing the performance and power consumption of 3d mobile games. *Computer*, 46(4):76–82, 2013.

[12] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[13] C. Min, S. Kang, C. Yoo, J. Cha, S. Choi, Y. Oh, and J. Song. Exploring current practices for battery use and management of smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, 2015.