Measuring and Improving Smartphone QoE Using the Screen as a Sensor

Scott Haseley and Geoffrey Challen

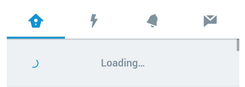
University at Buffalo - Department of Computer Science and Engineering

Improving Smartphone QoE

Smartphone users face a variety of issues while using their devices that affect their quality of experience (QoE). Slow or unresponsive apps, janky scrolling, long latencies, and poorly designed layouts all contribute to poor QoE.

By its very nature QoE is largely subjective, but there are many aspects of QoE that can be objectively measured. To the degree possible, the OS should try to improve user experience by measuring aspects of QoE, and react in ways to improve them.

A common source of poor QoE comes from long latencies in response to user interactions, such as clicking a button or refreshing app content. The OS should respond by doing anything it can to prioritize that process. However, how does the OS even know the user is waiting?

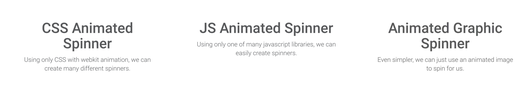


The Visual Language of Waiting

When apps and websites make users wait indefinitely, they provide visual feedback - such as small spinning circles - that users interpret as an indication to wait.

When users see these indicators, they expect to wait. But as the length of the wait increases, so does the likelihood of poor QoE.

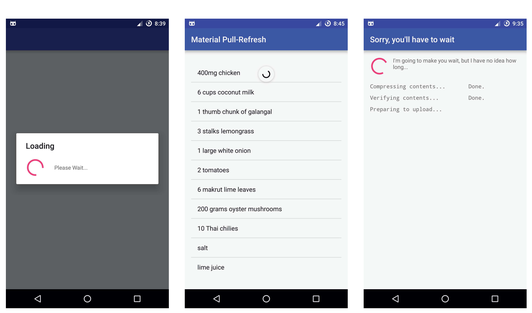
Because there is no shortage of ways that app developers can create these waiting indicators, it is a challenging problem to know when they are displayed across a large, diverse set of apps and websites.



On the web, there are a variety of ways that developers can show spinners.

From left to right:
1) CSS with webkit animation,
2) using a 3rd party JavaScript library, and
3) simply displaying a GIF

There are yet other methods, such as using an HTML canvas element to directly draw graphics.



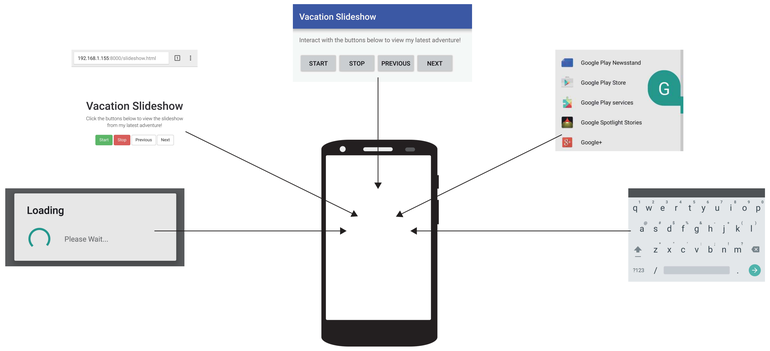
On Android, there are also many ways to show waiting indicators.

From left to right:
1) a ProgressDialog component
2) Google's material design SwipeRefreshLayout, and
3) an indeterminate ProgressBar

On Android, there are other methods as well, such as drawing using the Canvas API, or drawing directly using OpenGL.

The Screen as a Narrow Waist

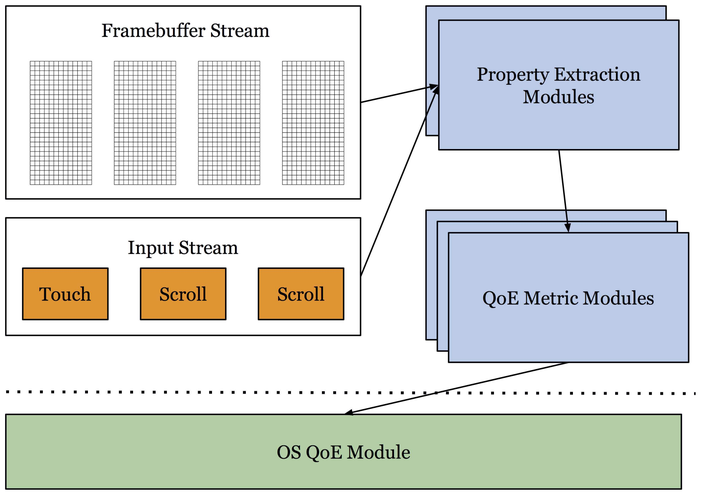
The screen forms a natural narrow waist at which to capture many QoE-related issues. Poor responsiveness, janky scrolling, laggy typing, and long latencies are all perceived by users through interactions with their screens.



Independent of implementation, the screen input and output captures a common language of interactions with smartphones.

QoEye Architecture

Our proposed system, QoEye, is a system that measures QoE at the pixel level, and exposes that information to the OS to help improve QoE.



The system takes two input streams: the stream of graphics framebuffers generated when the screen is drawn, and the stream of touch input events generated as a user interacts with the device.

Property extraction modules consume the screen I/O streams and compute high-level metrics and properties. Examples include frame rate and frame diff percent.

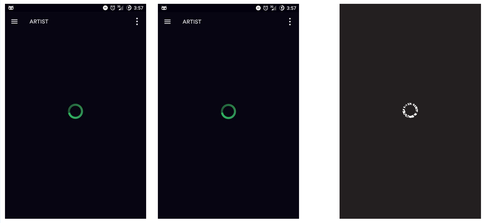
The high-level properties are fed into various QoE metric modules that compute objective measures of QoE aspects, such as active waiting and responsiveness.

The QoE metrics are used as input by an OS QoE module that can use that information to make decisions aimed at improving QoE.

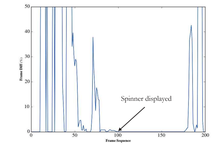
Efficiently Measuring QoE

Efficiency is paramount for QoEye since a typical screen refresh rate is 60 Hz. At modern smartphone resolutions, each frame can easily eclipse 14 MB of data.

While computing metrics such as frame rate are naturally efficient for QoEye, computing frame diffs, as needed by several modules, requires optimization.

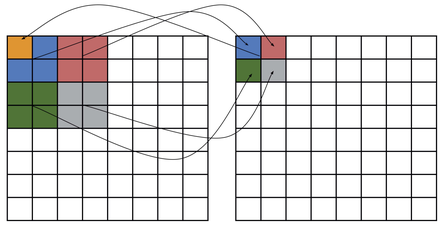


The first two images are consecutive frames of a spinner shown by the Spotify app on Android. The third image is the pixelwise diff image, with a pixel shown as white if there is a difference, black otherwise.



A plot showing frame diffs over time. The long valley corresponds to when a spinner was on screen.

On Android, our frame diff algorithm takes advantage of the data already being GPU-resident, and the massive parallelization available on GPUs. To compute diffs between two frames, we first scale down the images to a reasonable size, and perform the diff using a custom OpenGL shader.



To compute the % diff, we use a classic GPGPU reduction algorithm implemented using OpenGL shaders. The algorithm ping-pongs between two textures, averaging neighboring pixels in groups of 4, and writing the result to the alternate texture in a single texel per group. Computation is done on the GPU, and we read back a single value.

Responding to QoE Metrics

Once the system computes QoE metrics, this information must be used in a meaningful way. When possible, the OS should use this data to inform its own decisions. However, we also envision this information being used in a broader system that enables QoE to be improved across the network.

The OS can use QoE metrics to inform resource scheduling decisions. Threads, disk access, and network should be prioritized based on their impact on QoE.

Access points can also benefit from QoE metrics provided by QoEye. Traffic prioritization and bandwidth partitioning decisions can be made with user QoE in mind. For example, data that is destined for a user that is actively waiting should be prioritized over background traffic.

Similar to access points, cloud servers can benefit from knowledge of how their decisions might affect QoE. Internally, tasks can be prioritized in such a way that users actively waiting or interacting with their devices are prioritized.

University at Buffalo

Blue.

https://www.bluegroup.systems/projects/qoe
shaseley@buffalo.edu
challen@buffalo.edu