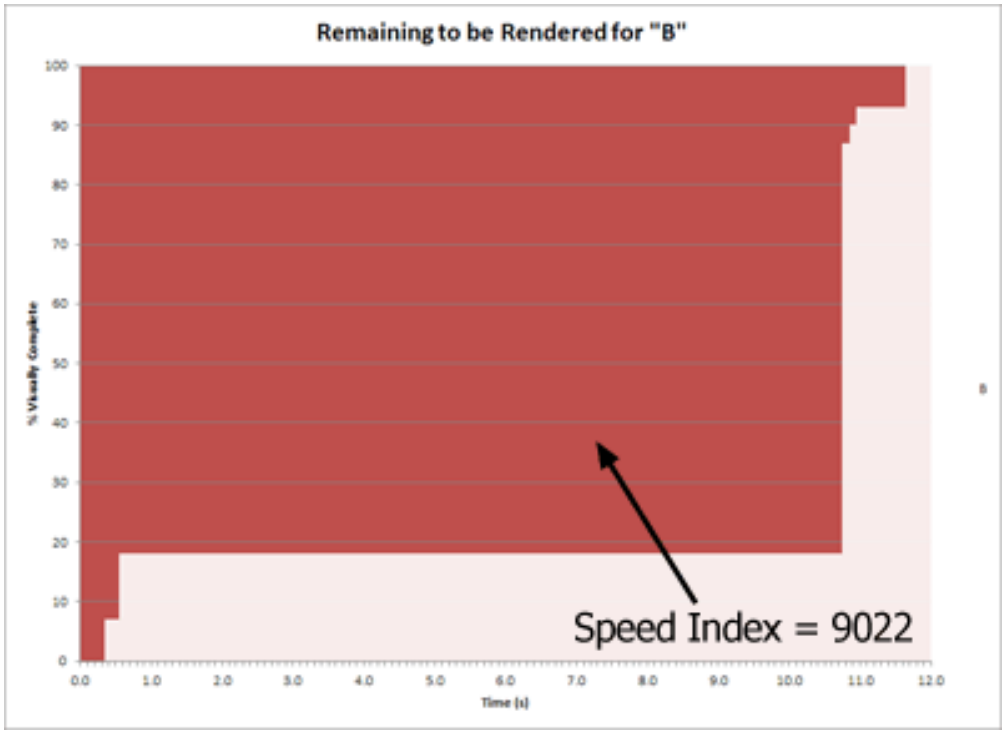
[PUBLIC] Client Side Speed Index Proposal

Owner: csharrison@chromium.org
Last updated: December 22, 2015
Required reading: WebPagetest Documentation
Tracking bug: 288758

Overview

The Speed Index metric tracks completeness of a page load from a rendering point of view, and is defined as the “average time at which visible parts of the page are displayed.” This is represented as the area above the curve in the following figure, where the X-axis is time, and the Y-axis is % visually complete. Note that the integration’s stopping point is when the page load is 100% visually complete. This definition is enough to be intriguing but vague enough to be frustrating. A client side implementation will need to iron out some details.



Objectives

Primary goal: Implement this metric for internal chrome use (e.g. UMA / Telemetry / RAPPOR) to give teams driving page load optimization a better metric for user happiness. This is designed to replace and enhance current state of the art point metrics in Chrome (first paint and first contentful paint metrics). Success will be primarily measured based on correlation with Speed Index measured by video capture.

Secondary goals:

- Expose it as a metric in devtools. This will involve explaining in clear words how the metric is recorded to developers. To do this well, we'll need to explain it without referencing specific implementation details, which brings us to the next goal.
- Provide a path forward for standardizing Speed Index, so developers can use it for RUM. This may involve compromising on implementation details and letting simplicity and ease of satisfying a spec trump accuracy.
- Provide a path forward for a trace-based implementation. Depending on goals of the tracing team, this may turn into many implementations (e.g. layoutSpeedIndex, paintSpeedIndex).

Open Questions and Challenges

User Interaction

In a lab setting, Speed Index calculations involve a phone quietly loading a page without any user interaction. At scale and in the wild, this will not be the case. It is the norm to use and scroll a site before it has finished loading, which makes questions of visual completeness much more complicated. There are a few possibilities, each one differs in what area they measure visual progress from.

1. Only consider the original viewport even when the user scrolls.
2. Always consider the entire document.
3. Expand the area of interest as the user scrolls. This means that no scrolling is equivalent to (1) and scrolling to the bottom of the document before any progress is made is equivalent to (2).
4. Fail the calculation if the user scrolls.

For Chrome, (4) is the only option feasible for a paint-based Speed Index. As far as I know, paint events only occur for regions of the document proximal to the current viewport.
In all cases, we may need a mechanism to report to the developer / API consumer that the calculation was inaccurate. This is especially important as work on prioritization may allow users to change how the page loads with how they scroll (i.e. prioritizing visible content).

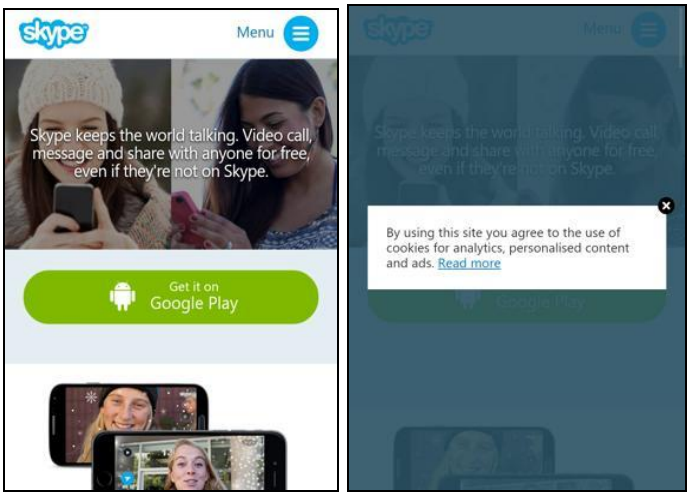
Minimum required for v1: Fail the calculation if the user scrolls, pinch zooms, or backgrounds the page. Later implementations may do (1) or (3). In general, mark a boolean if we are not confident with results.
Possible extension: Users could possibly be able to adjust this at will depending on their needs. For example:

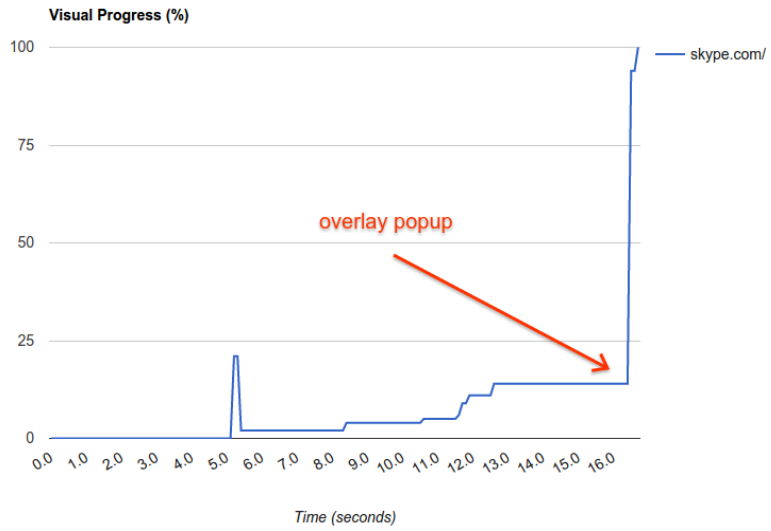
```
// Integrates % visual complete over the given viewing rectangle from navigationStart
// to onLoad.
double speedIndex(IntRect clipRect);
```

Would this mean the browser is responsible for maintaining the full list of completeness events for later filtering? Should we make this decision for them?

The Stopping Point

What does it mean for a page to be visually complete? This is a thorn in the side of Speed Index: the end state is not quite defined. In a lab setting with video capture (the traditional method of measurement), this can be calculated somewhat by whether video frames are changing, with additional data about loading state coming in from devtools. This fails for sites with site image carousels or animation. When stopping point finally occurs by some mix of signals, the final image may be much different from the one judged by human perception.





Overlays will confuse paint-based Speed Index calculations

On the client, the situation is even harder. Browsers can't instrument themselves efficiently to take a video capture on all page loads, so they must estimate a stopping point for visual completeness. Traditionally, the approach was to just use the load event as a stopping point for integration. This approach solves the majority of cases, but misses things like javascript based rendering called after onload. Another approach would be to define a new metric "Visually Complete," and use that as a stopping point. This adds another implementation and spec hurdle to the problem.

One solution would be to define Speed Index relative to bounds measures. This places the burden of generating bounds to the user or embedder:

```
// Integrates % visual complete over the given time range, assuming at time
// |startTime| the page is 0% complete, and at |endTime| it is 100% complete.
double speedIndex(double startTime, double endTime);
// Or, with added clipping rect:
double speedIndex(double startTime, double endTime, IntRect clipRect);
```

This approach has its own flaws. Is the browser required to keep track of visual progress for an unbounded amount of time? There may be room for some additional, reasonable stopping point (such as onload or later), that could be overridden by a user supplying |endTime|.

Visual Progress

How can the browser determine visual progress? Which layer is the most accurate? Which layer is the simplest?

The algorithm linked to in the WPT documentation requires every visual progress event to have an associated **timestamp** and **rectangle**. In our implementation, we require

- An event's timestamp reasonably coincides with a perceptual event occurring on the screen.
- An event's rectangle accurately describes where on the physical screen the event is occurring. Its area should be proportional to how much the event increases the visual progress of the load.

Chrome Implementations

There are a few places in Chrome itself to collect visual progress data to calculate this metric. There is not a clear solution at this point. Contenders are enumerated below.

Paint Layer

A rough draft implementation CL used invalidation rects passing through the GraphicsLayer. The technique achieved a .8 - .9 correlation coefficient with Speed Index calculated via video capture. This implementation used the following tricks:

- Starting point is responseStart, as many invalidations would occur before any bytes are received.
- Stopping point is the load event.
- If a repeat invalidation rect occurs, count its area as 1 / # repeats. This is supposed to account for progressive images but in practice a lot of repeat invalidations occur during the course of a page load. This technique requires keeping persisting a HashMap of size O(unique invalidation rects) for the course of the page load.

No testing was done with user interaction.

Ideas for a later implementation:

- Collect all invalidation rects after a completed paint phase.
- Correlate invalidation rects with corresponding SkPictures
- Get approximate op counts to judge how contentful the paint was
    - Possibly this idea could reduce the need to count duplicate rects

Problems**:**

- Cannot track visual progress for offscreen events.
- Possibly too implementation dependent for a unified spec.
- Repeated paints lead to problems with image carousels and video.

Layout Layer

The layout layer gives some flexibility to the implementation of Speed Index. No draft implementations have been written, so it's hard to directly compare to a paint-based approach. On the surface, instrumenting layout gives us the following.

Pros:

- Allows calculating the metric by only considering layouts that occurred in a certain rectangle.
- Is more fine grained than paint (which require instrumenting invalidation rects rather than paint rects)
- Could allow for tracking contentful layouts (e.g. layouts involving text)?
- "Solves" the problem of repeated paints by not re-laying out an animating object.
- Easier to spec?

Cons:

- Not strictly measuring user perception (though neither are invalidation rects).
- Not yet proven to be accurate.
- "Solves" the problem of repeated paints by not considering a progressive image load as multiple events indicating visual progress.