Blink's Rendering Pipeline

Eric Seidel, Emil A. Eklund, James Robinson, Elliott Sprehn
[please add your name if you'd like folks to bug you about what's up in rendering]

Status as of June 10, 2013: Published to blink-dev, under discussion

Objective

Complicated web applications and large pages can feel slow in Blink, particularly on mobile. Many of these issues can be traced to long/frequent blocks of time spent on the main thread calculating style or layout in Blink.

This document seeks to lay out both a long-term vision for how better/faster Rendering in Blink might come to look as well as short-term steps for getting there.

Problems in Rendering

- blocks JavaScript + user interaction, by executing synchronously on the main thread
- executes 100x more often than necessary (e.g. crbug.com/236063)
- exposes its internals all over the engine (no api) hard to change w/o breaking callers
- shares large/mutable data structures preventing parallel execution
- operates on the entire document at once (often more than required)
- uses coarse-grained phases (clients execute a whole layout/style/attach phase for the answer to much finer-grained questions)

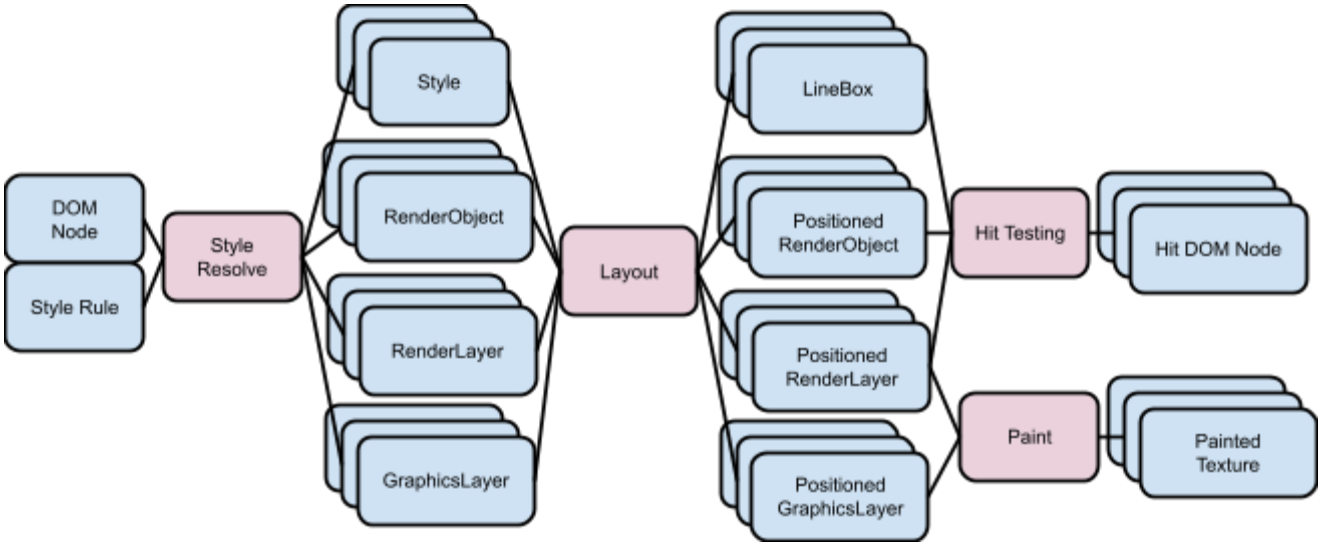
Goal

Our primary goal is performance. Right now our telemetry-based top25-sites "loading_benchmark" shows rendering accounting for on average 16% of main-thread time (and sometimes much more than that). If we drive that to 1.6% of main thread time via this effort, I'll consider this effort a success (and likely move onto other problems).

As part of this effort, I believe we should take this opportunity to clarify our rendering architecture to enable not only this performance improvement, but pave the way for improvements for years to come.

Rendering Today

Our Rendering system today looks something like this:



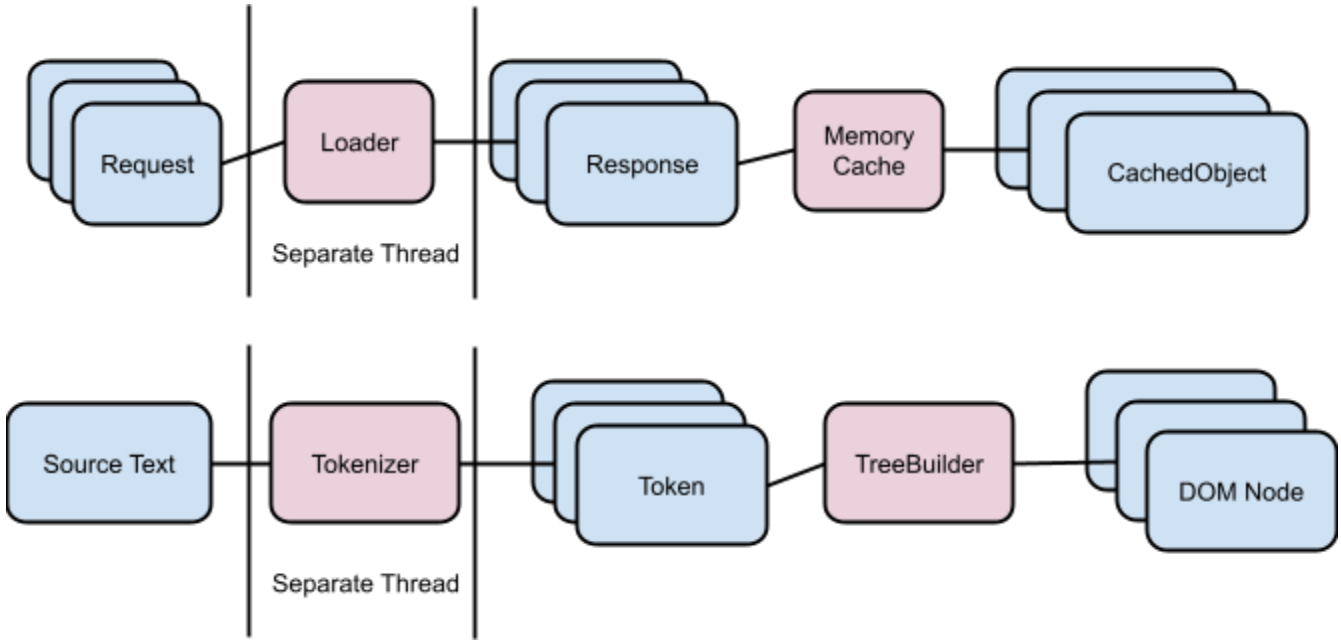
It's a bit awkward to draw today's rendering as a pipeline (as above). Currently "phases" or rendering operate on shared mutable data structures, so it's likely more useful to think of current rendering as a set of passes over the same data-structure, but I've drawn it such for easier comparison with below.

See Phases Of Rendering or the WebKit Technical Articles for more details on current architecture.

Making a Pipeline

Ideally Blink's rendering code should operate with many *explicit* and *independent* phases (similar to other graphics pipelines). Each phase should consume the output from the previous phase and produces output for the next. These outputs could be independently cached.

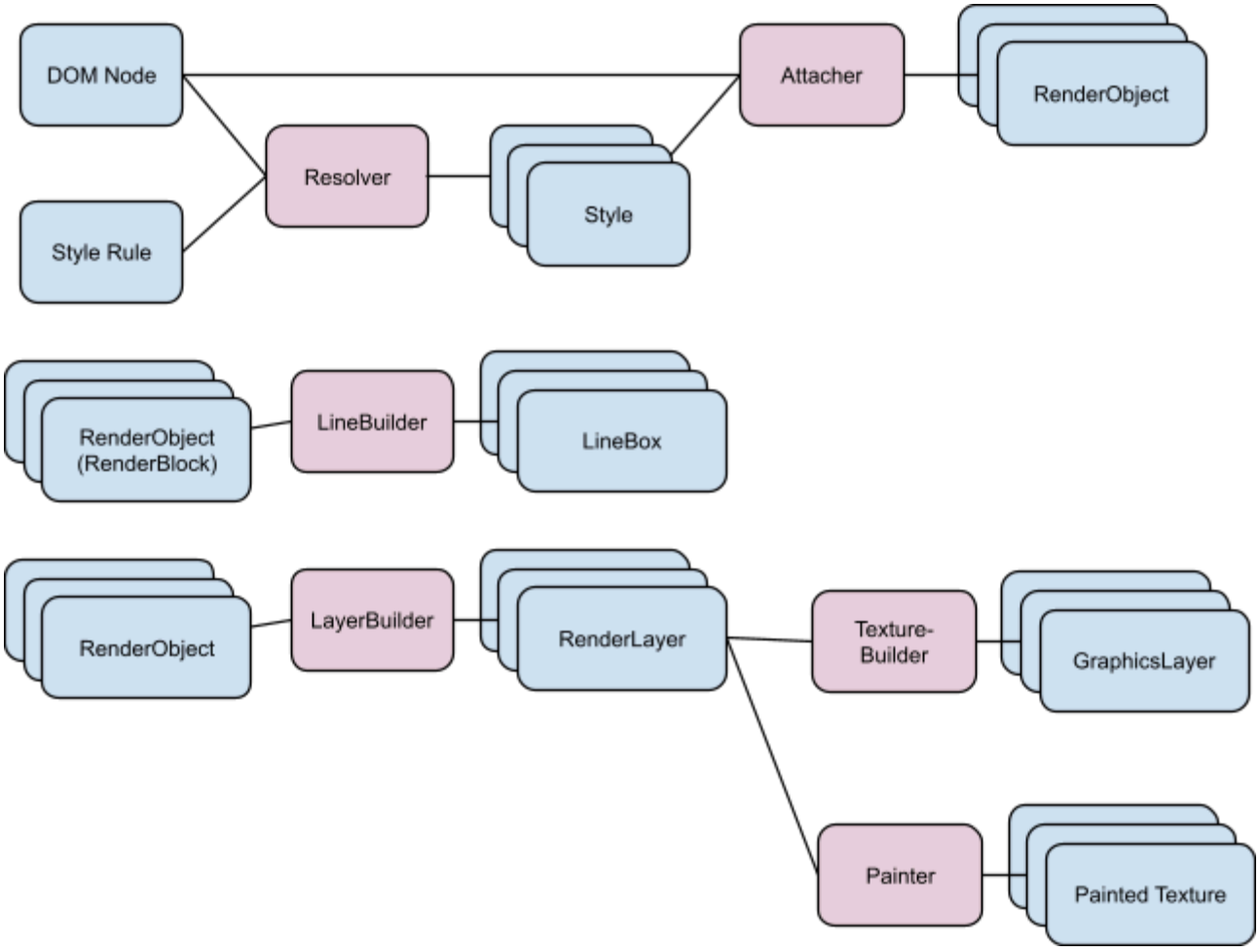
We've had success with a similar pipeline-based approach in both the Loader and HTML Parser:



(Before decoupling HTML Tokenization from the main thread, it accounted for approximately 5% of total time. Once we moved it to a separate thread we saw as much as 10% gains on page cyclers. As noted above, style resolve currently accounts for 16% of main-thread time on Telemetry's loading_benchmark, it's possible that we will see larger than 16% gains if successful at moving Style off of the main thread.)

We can similarly re-imagine the Rendering in Blink independent phases.

A hypothetical (and incomplete) Rendering phase diagram:



The major difference in that diagram is breaking existing phases into many pieces. Unfortunately, as drawn, the outputs of those phases need to modify their inputs (e.g. RenderLayers set on the input RenderObjects). However, breaking phases into smaller pieces can allow us to control when these mutations occur.

Initially a careful phase-based approach could allow us to execute the style and layout concurrently. Longer-term (as we scale to handle even larger web applications) we may add the ability to shard large style/layout jobs across many copies of the same phase.

Getting from here to there

Our biggest obstacle between today and the dreams of this document is modularization. The Rendering system in Blink needs to be abstracted from other systems, and the phases within rendering needs abstraction between each other.

Possible paths to better modulation may include:

- Adding an API layer for Rendering and attempting to move Accessibility and Editing (two large consumers of Rendering) onto that.
- Break up large classes, files, methods and phases of rendering into smaller pieces. Not only do external clients reach into rendering at unexpected times, but rendering classes, phases, reach across object boundaries in ways which prevent further factoring.
- Fix current rendering-access points -- notably Node::renderer() and RenderObject::style() --- to ASSERT that rendering is fully-up-to-date before returning. This may help us ferret out callers which are making timing expectations of the rendering tree outside of the rendering subsystem. We likely will need to add “stale” equivalents of these methods (staleRenderer, staleStyle) which are used internally to rendering.

A second obstacle is making the individual phases interruptible/resumable. Right now style recalc and layout always operate on the entire document regardless of how long it takes. (Creating a partial-tree version of style-recalc would be enough to speed up some clients such as getComputedStyle.)

Another third task is to break our two monolithic phases (style, layout) into smaller pieces. Right now we have hackish “post attach callbacks” and actions such as “focus update” which instead of being hacked into Document::recalcStyle() could be viewed as entirely separate phases through a generalized state-machine.

Finally, real parallelism is blocked by making individual phases idempotent and their outputs immutable. Until Style Recalc and Layout stop sharing the same output structure (the rendering tree) we can’t make them run in parallel with one another. Style Recalc is probably the best place to start here, as it would be possible to separate the creation of the styles from assigning them into the tree.

An Alternate Approach: Just Double Buffer

This document focuses on updating each of the individual phases in Rendering to be idempotent and parallelization ready. That’s probably useful in the long-term regardless (for architectural cleanliness, if nothing else), but may not be our fastest path to victory. Alternatively we could keep all of the existing (shared) data structures, but instead keep two copies and atomically switch between them like a graphics system might switch between buffers. I’m not advocating this approach, but it should be consider as a possible way to trade memory for a quicker parallel-rendering solution.

Testing

- Telemetry can be used to test what % of main-thread time is spent in style/layout.

Ideally testing should be done on Android, the top_25 tests suite is a good place to start:
`tools/perf/run_multipage_benchmarks --browser=android-content-shell loading_benchmark tools/perf/page_sets/top_25.json`

- `run-perf-tests Parser/html5-full-render.html`
  has also proven useful for testing time spent in the style-system.
- Running some of the top 1000 sites (with the inspector, possibly in telemetry) and counting the number of style resolves/layouts is another useful metric.

Near-term Tasks

- Documentation -- No one understands the current Rendering system fully. One approach to fixing it may be to start documenting it (better than the drawings in this document).
- ASSERTs -- Besides the renderer()/style() asserts mentioned above, a good place to start may be adding ASSERTs through existing phases of Rendering. Ensuring that the phases do what we expect they do (and in the order we expect) is a necessary prerequisite to breaking them apart further. An example would be to ASSERT that we attach() in the order expected: https://bugs.webkit.org/show_bug.cgi?id=112855
- Break up StyleResolver -- There are many classes we could simplify in Rendering, but StyleResolver is perhaps the worst. It contains data of document lifetime, per-element resolve lifetime, and per-document resolve lifetime. Splitting these off blocks any future attempt at parallelizing style resolve: https://bugs.webkit.org/show_bug.cgi?id=89879, https://code.google.com/p/chromium/issues/detail?id=237268
- ~~lazyAttach -- The "attach" phase is currently partially a sub-phase of style recalc, and partially an explicit action during parsing. lazyAttach is a replacement approach which just marks a node as needing style resolve instead of synchronously creating a renderer for it. We need to finish deploying this to delete the "attach" complexity of our engine. Elliott is working on this currently:~~
  ~~https://bugs.webkit.org/show_bug.cgi?id=111494~~
  ~~https://bugs.webkit.org/show_bug.cgi?id=113097~~
  ~~https://bugs.webkit.org/show_bug.cgi?id=37688~~
- Remove the loader from Rendering and Rendering from the Loader -- This huge layering violation is in place to support iframe loads being sensitive to display: none. This causes unnecessary layouts (https://bugs.webkit.org/show_bug.cgi?id=70009) and introduces complexity into Rendering. Removing this wart is on the path to isolating Rendering from the rest of Blink before we can change it's internals.
- There are also micro-benchmark style optimizations available for style/rendering which can be found using:
  run-perf-tests Parser/html5-full-render.html --profile
  These are less important than removing whole style recalcs, or layouts, but could

provide immediate gains across many sites.
- Create a new core/rendering/api directory, move only the necessary set of headers there necessary to keep the rest of core building, and then slowly develop a smaller set of exposed headers. This has no performance gain, only code complexity gain, and gives us a list of headers to burn down to 0 as we constrain Rendering's api exposure in preparation for re-writing Rendering's guts.

Speculative Tasks

These are higher-risk possibly high-reward tasks. Less clear which are on the critical path:

- Flatten recalceStyle/layout -- These are both more complicated than needed, and possibly could be implemented with a loop instead of recursive descent. This may be part of making both interruptible/work-on-partial trees.
- Simplify renderers -- RenderObject subclasses do too much. As a concrete example, RenderBlock could be split into multiple classes (ones which have inline children vs. those which do not). It's also possible that some data could be pushed off of Renderers into the DOM or directly into Skia (our graphics layer). It's even possible that the graphics layer should own (and we should just position/style) much of what we currently call the "rendering tree" in Blink (like how v8 is responsible for the lifetime of bindings objects we just augment them).
- Shard Style Resolve -- Unclear if resolving multiple sub-trees in parallel is worth the complexity, but Style Resolve is mostly sub-tree constrained and thus easily shardable. If our algorithm were made idempotent with regards to the Rendering Tree, we could shard subtree resolves across multiple threads and stitch the resulting styles into the tree at the end of the resolve on the main thread. This is a separable task from the larger pipelining work.
- Partial Style Resolve -- As mentioned above, getComputedStyle often only needs the style for a single element, not a recalc of the entire document. Teaching Style Resolve about this is one piece of making it possible to resolve style in parallel with other tasks.
- Style promises -- jQuery and other libraries often grab style/CSS information eagerly and cache it in JavaScript. It's possible that we should be vending them promise objects instead of bothering with a full style resolve each time.

Future Considerations

Versioned Caching / Partial Updates

It's possible to imagine a world in which there is versioned caching between phases, to allow parallel construction/access to phased data. The simplest version of this model involves only

two versions, and is akin to the double-buffering noted above. (One version is being written, the other is being read from.)

If we view all of the cached data as immutable then we can do much more interesting things with our caches, including versioning.

Take for example, a style change to a single element. That would require construction of new version chain of caches starting at the style phase. We could start by computing the new style for that element, and recomputing the layout for its ancestor tree. But we would not need to relayout the entire tree, and we would only need to synchronously (with a mutex) graft in the new versioned subtree once its asynchronous styling/layout updates completed.

We could transition to the example “two version” world once we have an API onto Rendering, using the wait-for-cache-refresh version everywhere first and slowly moving more and more of the Blink to use the cached-result APIs or asynchronous notification APIs where appropriate.

Application Primitives

JavaScript applications which repeatedly invalidate one of these phases (say updating lots of rows of a table in a loop), could cause pathological behavior of our speculative update mechanisms. We would need heuristics to allow us to update these phase caches in parallel to running JavaScript, while delaying updates to make sure that we are not repeatedly discarding our speculations.

It may be possible to use JavaScript/Application primitives to help guide optimizations relating to updating these versioned subtrees.

Rendering Tree Alternatives

Once Rendering is isolated from the rest of Core by an API, there are many improvements we can make. We can even consider replacing the core data structure “the rendering tree”.

Scene Graph

The Rendering Tree and Layer Trees are O(N) or worse for hit-testing. They also have no (built-in) concept for occlusion mapping. There are more advanced data structures we could add into our rendering pipeline to provide this these memory/performance tradeoffs.

Retained Mode

The Rendering Tree serves (too) many purposes. One of them is to translate position and style information into immediate mode drawing commands. We could be more efficient in painting if

we were used a retained-mode drawing API. This would require caching of (opaque) retained mode objects from Skia on the renderers. Once we have a tree like this, it's not necessarily even clear that we need to keep our own tree. (In the same way that v8 owns the underlying V8 objects and we hang data from Blink off of them, it's not clear that we need to "own" our "rendering tree" and that it's possible that we should just hang data off of the compositor or skia's tree.)

More Information

https://bugs.webkit.org/show_bug.cgi?id=111644 is an old meta-bug from webkit.org