READ MODE · ANSWER FIRST

Browser Rendering Pipeline + INP

An interactive, visual explanation of Browser Rendering Pipeline + INP, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

Browser Rendering Pipeline + INP becomes easier to reason about when every stage is connected as one system.

1. Show the parser as a left-to-right transformation from bytes to tokens to DOM nodes, with blocking script execution as the main branch.
2. Teach why CSS is render-blocking by showing sheet fetches and rule merging into a single CSSOM panel.
3. Make the distinction between DOM and render tree explicit: DOM contains everything, render tree keeps only paintable boxes.
4. Show layout as a geometry pass over a dirty subtree, not as an abstract black box.
5. Separate paint from composite and show why transform-only motion is cheaper than layout-affecting motion.
6. Tie the full rendering pipeline back to user pain: long tasks delay event handling, heavy handlers extend work, and paint/composite...

STAGE 1 OF 8

Setup

Welcome. Every time you open a web page, the browser runs an assembly line that turns code into pixels. This explainer walks through that pipeline one stage at a time, so you can see exactly where your page spends its time.

The DOM is the tree structure the browser builds from your HTML. Think of it like a family tree where every tag becomes a node with parents, children, and siblings.

The CSSOM is the same idea, but for CSS. The browser reads your stylesheets and builds a separate tree of rules that describe how things should look.

The render tree combines the DOM and CSSOM, keeping only the boxes that actually need to be painted. Hidden elements get pruned before any geometry work starts.

Layout computes where each box sits and how big it is. Compositing then layers painted tiles into the final frame. INP measures how long a user waits between clicking and seeing the result.

Over the next six stages we will trace the full path: HTML parsing, CSS parsing, style resolution, layout, paint and compositing, and finally interaction delay. Let us start with how bytes become a DOM tree.

STAGE 2 OF 8

HTML -> DOM

02 HTML -> DOM

The browser just received raw HTML bytes from the network. Right now the stream panel is full of data, but there is no structure yet. Think of it like receiving a letter you have not opened.

The parser reads those bytes and starts emitting tokens, one tag at a time. Watch the token slots light up below the parser node as each tag is recognized.

Each open tag becomes a real DOM node. You can see the DOM tree panel growing as the parser attaches nodes. Try switching the Document control to "App shell" to see a different tree shape appear.

When the parser hits a blocking script, everything stops. The main thread must execute that JavaScript before the parser can continue. Toggle the Script control off to skip the pause entirely and watch parsing sail through.

Once the script finishes, the parser resumes and the remaining child nodes attach to the DOM. Notice how the tree fills in from where it left off.

The DOM tree is now complete. Every HTML tag has a node, and the parser is done. Next, the browser needs to figure out how everything should look, so we move on to CSS parsing.

STAGE 3 OF 8

CSS -> CSSOM

03 CSS -> CSSOM

The DOM tree from the previous stage is ready, but the browser still has no idea how anything should look. Stylesheets have not been discovered yet, so the CSSOM panel is empty.

As the HTML scanner reads through the document, it finds <link> tags pointing to external stylesheets. It hands those URLs to the loader so they can be fetched in parallel.

The browser fires off network requests for each stylesheet. Use the Sheets stepper to add a third sheet and watch an extra request appear. More sheets mean more waiting before rendering can start.

Each stylesheet gets parsed into a list of rules. Selectors like ".hero" tell the browser which DOM nodes the declarations apply to. Switch the Cascade control to "Nested" to see deeper selector matching work.

All the parsed rules merge into a single CSSOM. Once that merge finishes, the "render blocked" badge flips to "ready for style." That unblocking moment is why CSS performance matters.

The CSSOM is complete. We now have both structure (the DOM) and appearance (the CSSOM). Next, the browser will combine them into a render tree that contains only the boxes it actually needs to paint.

STAGE 4 OF 8

Style + Render Tree

04 STYLE + RENDER TREE

We now have both the DOM and the CSSOM from the previous two stages. The style engine has not started yet, so the render tree panel on the right is still empty.

The style engine pulls elements from the DOM and rules from the CSSOM, then begins matching selectors to nodes. Watch the messages flow into the engine circle.

Selector matching walks through every node and figures out which CSS rules apply. Switch Selector Cost to "High" and notice the match counter jump from 420 to 1.8k. Deeply nested selectors like ".promo .title" force the engine to walk further up the tree.

Not every DOM node becomes a box. Toggle Hidden UI on and you will see a "promo [display:none]" row highlighted in the DOM panel. The engine prunes that node so layout never wastes time on it.

The render tree now holds only the paintable boxes with their final computed styles. Each row in the render tree panel shows a node mapped to a box.

Style resolution is done. The render tree is a lean list of boxes ready for geometry. Next, the layout engine will turn those boxes into pixel positions on the screen.

STAGE 5 OF 8

Layout

05 LAYOUT

The render tree from the previous stage is ready, but the browser has not computed any geometry yet. Each box needs a position and a size before anything can appear on screen.

A change marks the "content" node dirty. That dirty flag sends an invalidation message into the layout engine. Watch the warning color highlight the affected row in the render tree panel.

The layout engine decides how far that dirty flag spreads. Switch Scope to "Full page" and you will see every row in the render tree light up, because a full reflow must remeasure all boxes.

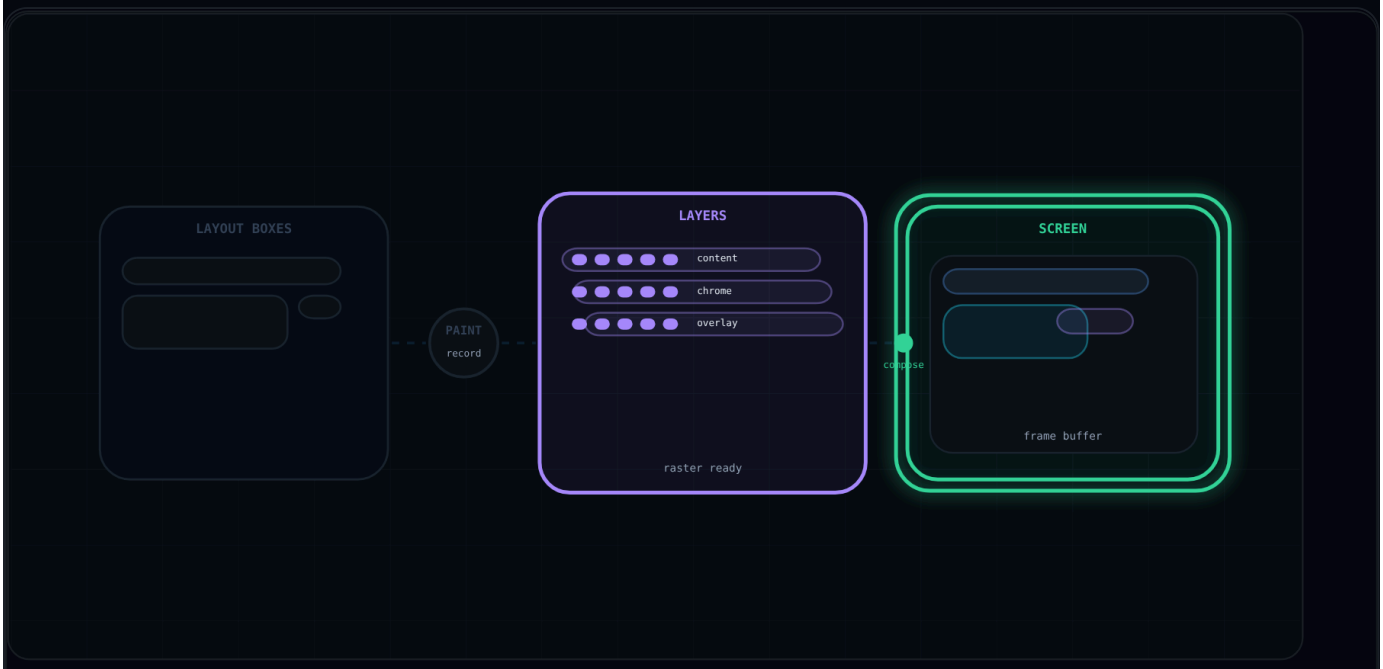
Boxes are measured against the viewport. Switch Viewport to "Mobile" and notice the boxes in the viewport panel stack vertically instead of sitting side by side. Narrower screens force more line breaks.

The engine writes final pixel positions and dimensions back to each affected box. You can see the completed layout reflected in the viewport panel.

Geometry is now stable. The browser knows where every box sits and how big it is. Next, we move on to paint, where the browser records the draw commands and composites the final frame.

STAGE 6 OF 8

Paint + Composite



06 PAINT + COMPOSITE

Layout gave us pixel positions in the previous stage. Now the browser needs to record what to draw and fill in the actual pixels. The layout boxes panel on the left shows the geometry, but the screen is still blank.

The paint step walks through the layout boxes in visual order and emits draw commands. You can see the paint node light up as it records instructions like "fill this rectangle with blue."

Raster takes those paint commands and fills tiles for each layer. Toggle the Overlay control to add a sticky overlay, and you will see a third layer appear in the layers panel. More layers give the compositor more flexibility.

The compositor stacks those rasterized layers in the correct order and writes them into the frame buffer. The screen panel now shows the assembled result.

This is where the Animation control matters. Switch it to "Layout" and you will see a repaint message travel back through paint, because a layout change forces new draw commands. With "Transform" selected, the compositor can reuse the existing painted layers and just move them.

The frame is ready and the pixels are on screen. Compositing is the last rendering step. Now we will look at what happens when a real user interacts with the page, and how all of these stages add up to interaction delay.

STAGE 7 OF 8

Interaction Delay + INP

07 INTERACTION DELAY + INP

The rendering pipeline from the previous five stages produced a painted frame. Now the page is sitting idle, waiting for the user to do something. This is where interaction delay begins.

The user taps a button. That tap creates an event that needs to reach the handler. Watch the message travel from the input node toward the main thread queue.

If the main thread is already busy with long tasks, your event sits in the queue. Use the Backlog stepper to add more tasks and watch the queue panel fill up. Each extra task adds roughly 58ms of waiting.

Once the queue clears, the handler finally runs. Toggle Yielding on and notice the handler node turns green instead of red. Yielding breaks the handler into smaller chunks so the browser can render sooner.

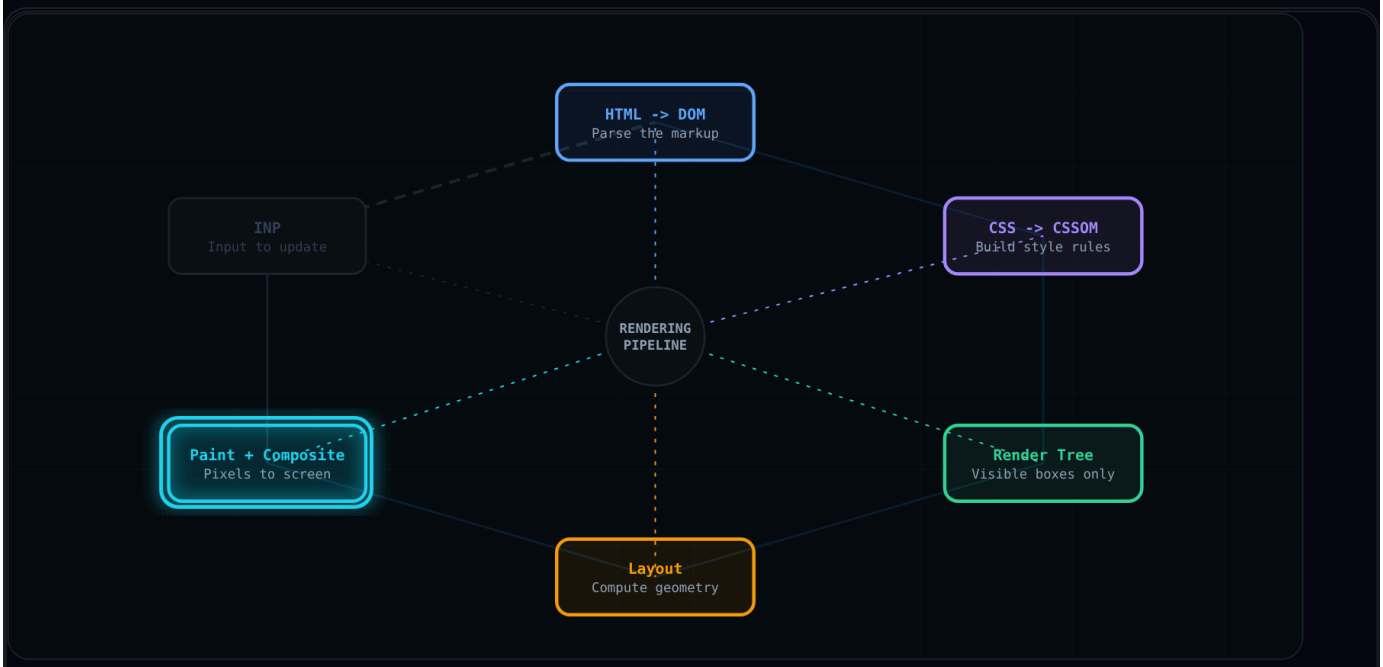
After the handler mutates state, the browser runs through the familiar cycle of style, layout, and paint to produce the next frame. You can see those three steps light up inside the frame panel.

The compositor presents the updated pixels to the screen. The INP bar above now shows how the total time splits across delay, processing, and rendering.

INP measures the entire path from tap to visible update. Every millisecond spent in parsing, style, layout, and paint from the earlier stages adds up here. That is the full rendering pipeline, from HTML bytes to user response time. Let us wrap up with a recap of the whole journey.

STAGE 8 OF 8

Recap



08 RECAP

It all starts with HTML. The parser turns raw bytes into a DOM tree, which is the structural backbone every later stage depends on. Without a DOM, nothing else can happen.

CSS rules get their own tree. Remember how stylesheets block rendering? That is because the CSSOM must be complete before the browser can figure out what anything looks like.

The render tree merges DOM and CSSOM, but only keeps the boxes that will actually appear on screen. Hidden elements never make it past this gate.

Layout reads the render tree and computes real pixel positions. The viewport size matters here, which is why a page can look different on a phone than on a laptop.

Paint records draw commands, raster fills tiles, and the compositor layers everything into the frame buffer. Transform-only changes can skip the expensive repaint step entirely.

INP ties it all together from the user perspective. Every millisecond in parsing, style, layout, and paint adds up to the delay between a click and the visible response.

The full pipeline is a loop: user interactions trigger new style and layout work, which flows right back through paint and compositing. Optimizing any single stage makes the whole cycle faster.