Inside look at modern web browser (part 3)

Published on Thursday, September 20, 2018 • Updated on Tuesday, August 18, 2020

Mariko Kosaka
Mariko is a drawsplainer

Table of contents

Inner workings of a Renderer Process

This is part 3 of 4 part blog series looking at how browsers work. Previously, we covered multi-process architecture and navigation flow. In this post, we are going to look at what happens inside of the renderer process.

Renderer process touches many aspects of web performance. Since there is a lot happening inside of the renderer process, this post is only a general overview. If you'd like to dig deeper, the Performance section of Web Fundamentals has many more resources.

Renderer processes handle web contents

The renderer process is responsible for everything that happens inside of a tab. In a renderer process, the main thread handles most of the code you send to the user. Sometimes parts of your JavaScript is handled by worker threads if you use a web worker or a service worker. Compositor and raster threads are also run inside of a renderer processes to render a page efficiently and smoothly.

The renderer process's core job is to turn HTML, CSS, and JavaScript into a web page that the user can interact with.

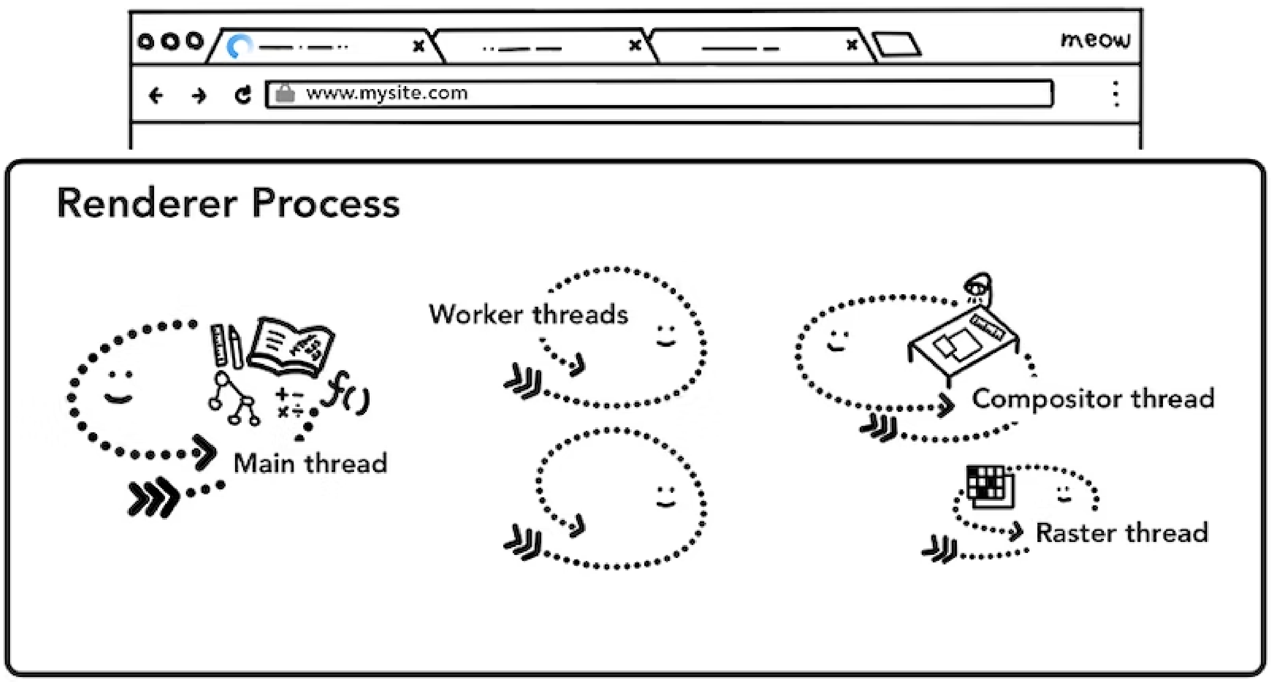


Figure 1: Renderer process with a main thread, worker threads, a compositor thread, and a raster thread inside

Parsing

Construction of a DOM

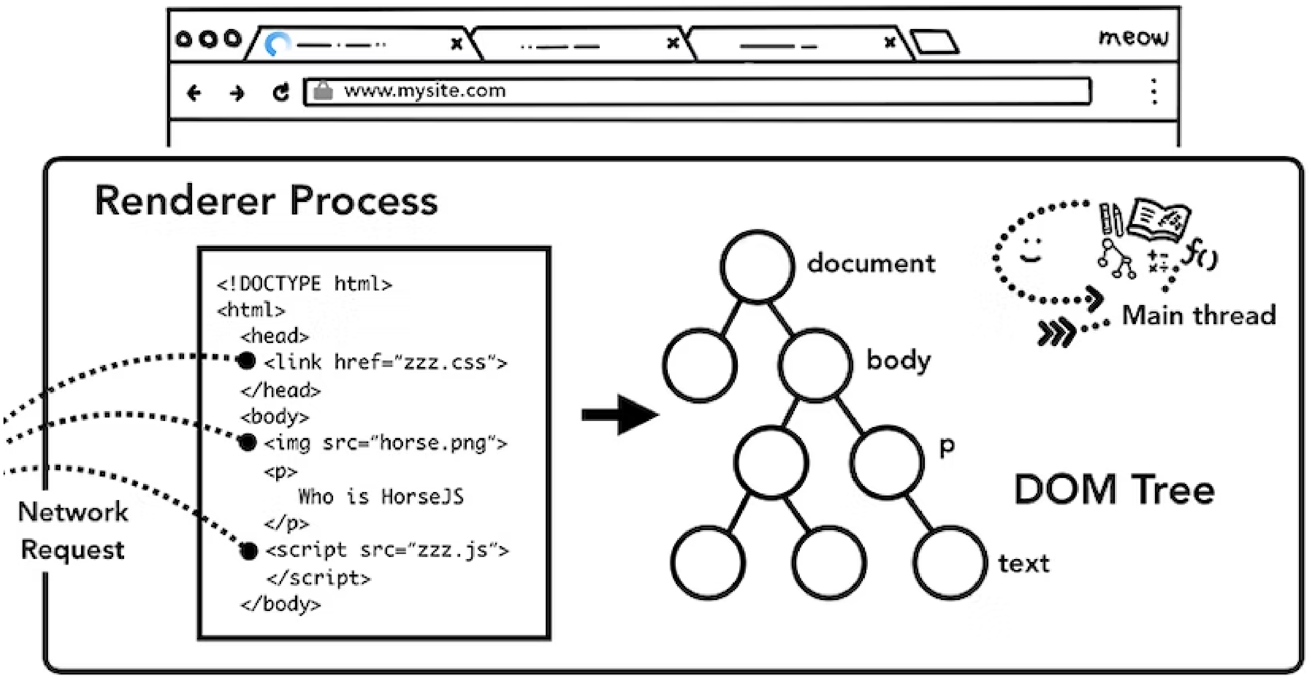
When the renderer process receives a commit message for a navigation and starts to receive HTML data, the main thread begins to parse the text string (HTML) and turn it into a **D**ocument **O**bject **M**odel (**DOM**).

The DOM is a browser's internal representation of the page as well as the data structure and API that web developer can interact with via JavaScript.

Parsing an HTML document into a DOM is defined by the HTML Standard. You may have noticed that feeding HTML to a browser never throws an error. For example, missing closing `</p>` tag is a valid HTML. Erroneous markup like `Hi! <b>I'm <i>Chrome</b>!</i>` (b tag is closed before i tag) is treated as if you wrote `Hi! <b>I'm <i>Chrome</i></b><i>!</i>`. This is because the HTML specification is designed to handle those errors gracefully. If you are curious how these things are done, you can read on "An introduction to error handling and strange cases in the parser" section of the HTML spec.

Subresource loading

A website usually uses external resources like images, CSS, and JavaScript. Those files need to be loaded from network or cache. The main thread *could* request them one by one as they find them while parsing to build a DOM, but in order to speed up, "preload scanner" is run concurrently. If there are things like `<img>` or `<link>` in the HTML document, preload scanner peeks at tokens generated by HTML parser and sends requests to the network thread in the browser process.



```
<!DOCTYPE html>
<html>
  <head>
  <link href="zzz.css">
  </head>
  <body>
  <img src="horse.png">
    <p>
      Who is HorseJS
    </p>
  <script src="zzz.js">
    </script>
  </body>
```

Figure 2: The main thread parsing HTML and building a DOM tree

JavaScript can block the parsing

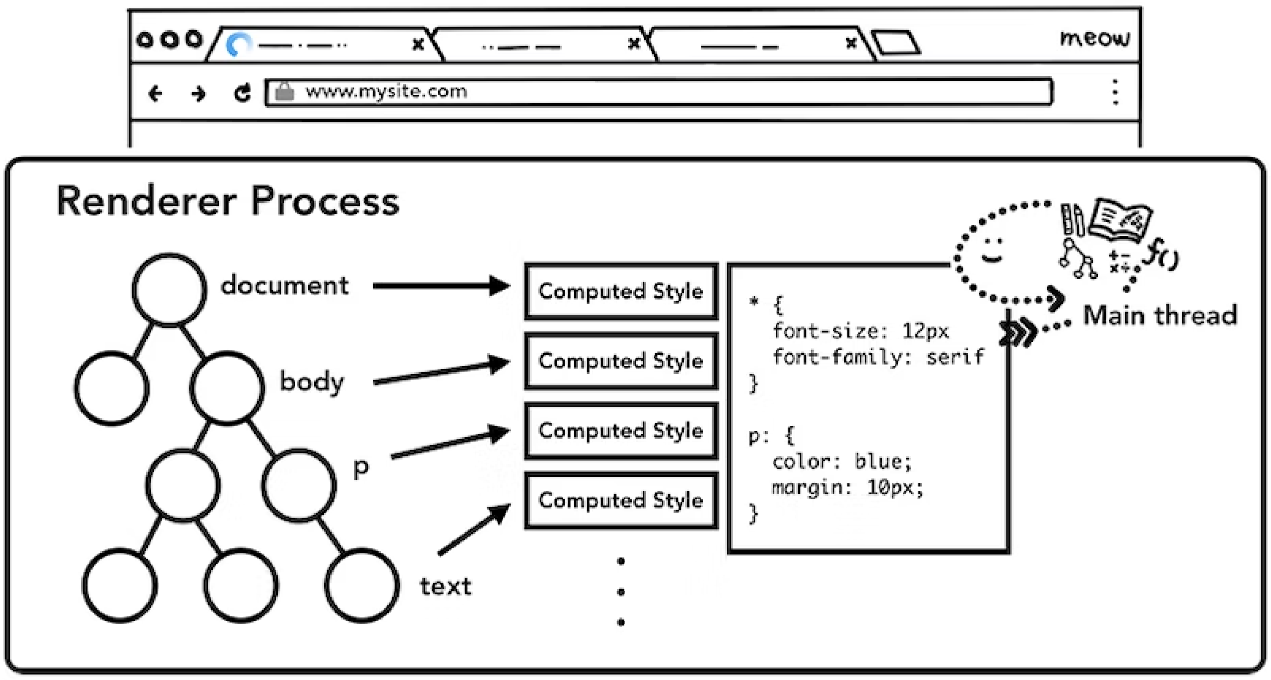
When the HTML parser finds a `<script>` tag, it pauses the parsing of the HTML document and has to load, parse, and execute the JavaScript code. Why? because JavaScript can change the shape of the document using things like `document.write()` which changes the entire DOM structure (overview of the parsing model in the HTML spec has a nice diagram). This is why the HTML parser has to wait for JavaScript to run before it can resume parsing of the HTML document. If you are curious about what happens in JavaScript execution, the V8 team has talks and blog posts on this.

Hint to browser how you want to load resources

There are many ways web developers can send hints to the browser in order to load resources nicely. If your JavaScript does not use `document.write()`, you can add `async` or `defer` attribute to the `<script>` tag. The browser then loads and runs the JavaScript code asynchronously and does not block the parsing. You may also use JavaScript module if that's suitable. `<link rel="preload">` is a way to inform browser that the resource is definitely needed for current navigation and you would like to download as soon as possible. You can read more on this at Resource Prioritization – Getting the Browser to Help You.

Style calculation

Having a DOM is not enough to know what the page would look like because we can style page elements in CSS. The main thread parses CSS and determines the computed style for each DOM node. This is information about what kind of style is applied to each element based on CSS selectors. You can see this information in the `computed` section of DevTools.



```
* {
  font-size: 12px
  font-family: serif
}

p: {
  color: blue;
  margin: 10px;
}
```

Figure 3: The main thread parsing CSS to add computed style

Even if you do not provide any CSS, each DOM node has a computed style. `<h1>` tag is displayed bigger than `<h2>` tag and margins are defined for each element. This is because the browser has a default style sheet. If you want to know what Chrome's default CSS is like, you can see the source code here.

Layout

Now the renderer process knows the structure of a document and styles for each nodes, but that is not enough to render a page. Imagine you are trying to describe a painting to your friend over a phone. "There is a big red circle and a small blue square" is not enough information for your friend to know what exactly the painting would look like.

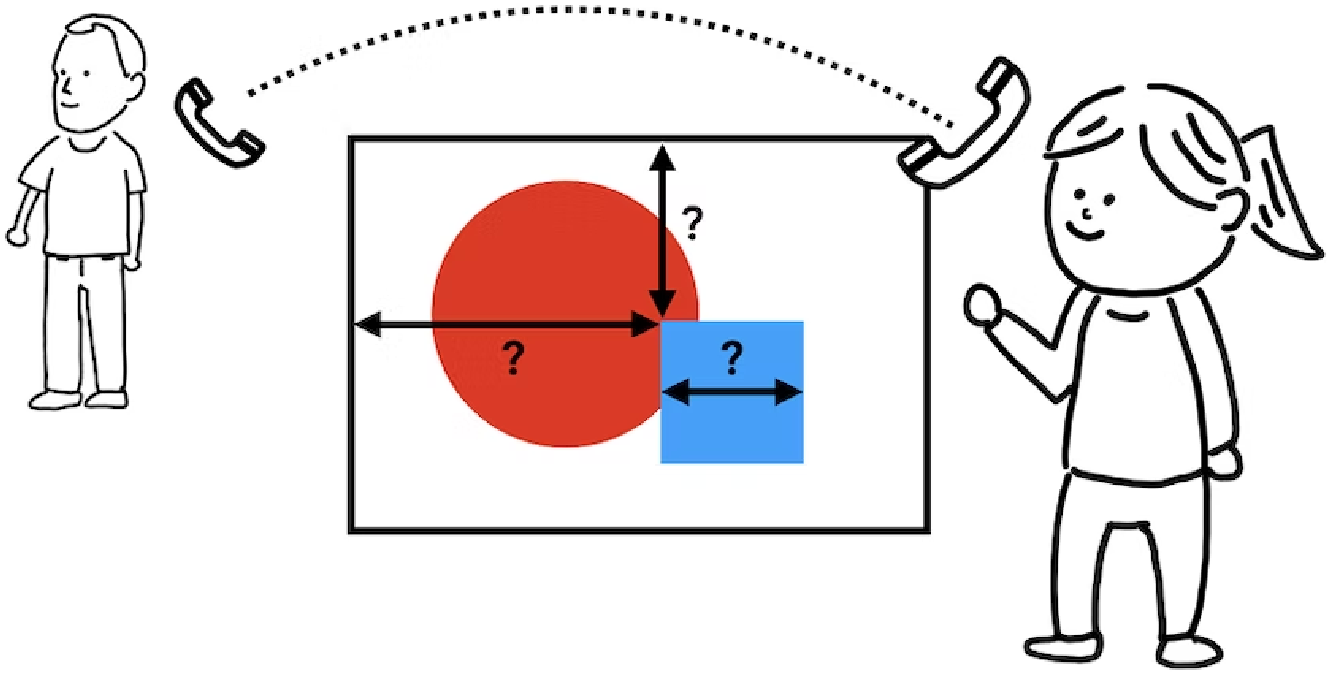


Figure 4: A person standing in front of a painting, phone line connected to the other person

The layout is a process to find the geometry of elements. The main thread walks through the DOM and computed styles and creates the layout tree which has information like x y coordinates and bounding box sizes. Layout tree may be similar structure to the DOM tree, but it only contains information related to what's visible on the page. If `display: none` is applied, that element is not part of the layout tree (however, an element with `visibility: hidden` is in the layout tree). Similarly, if a pseudo class with content like `p::before{content:"Hi!"}` is applied, it is included in the layout tree even though that is not in the DOM.

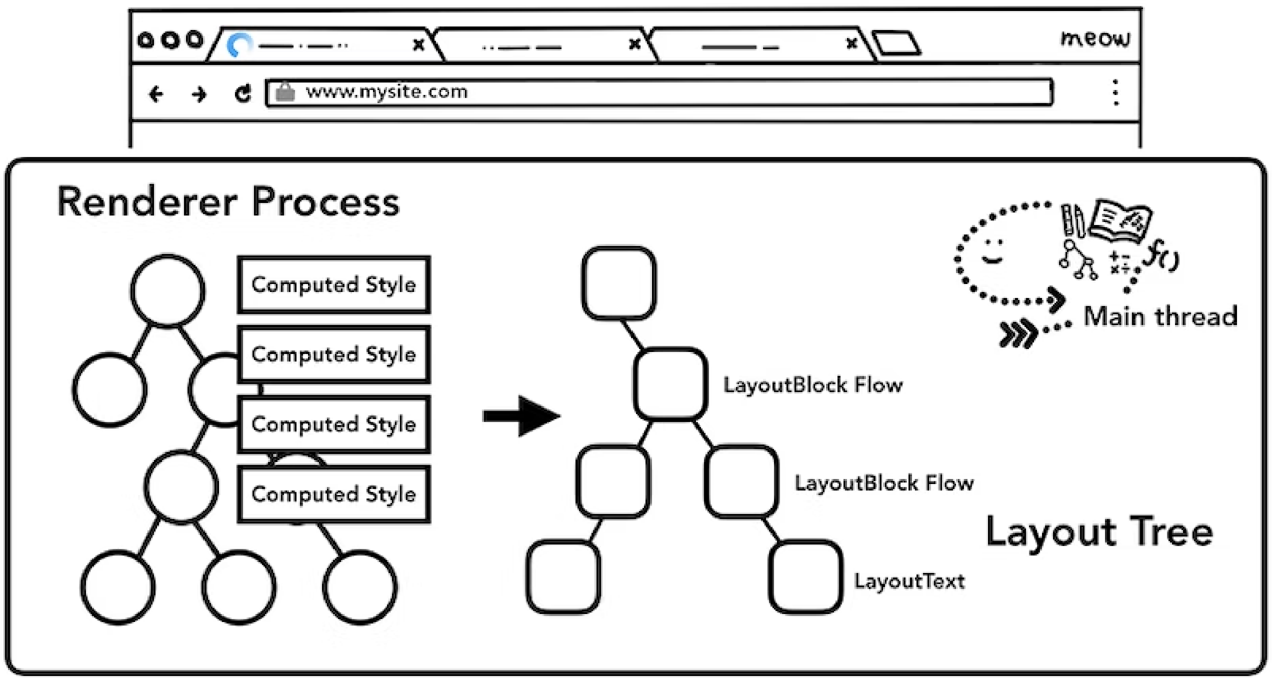


Figure 5: The main thread going over DOM tree with computed styles and producing layout tree

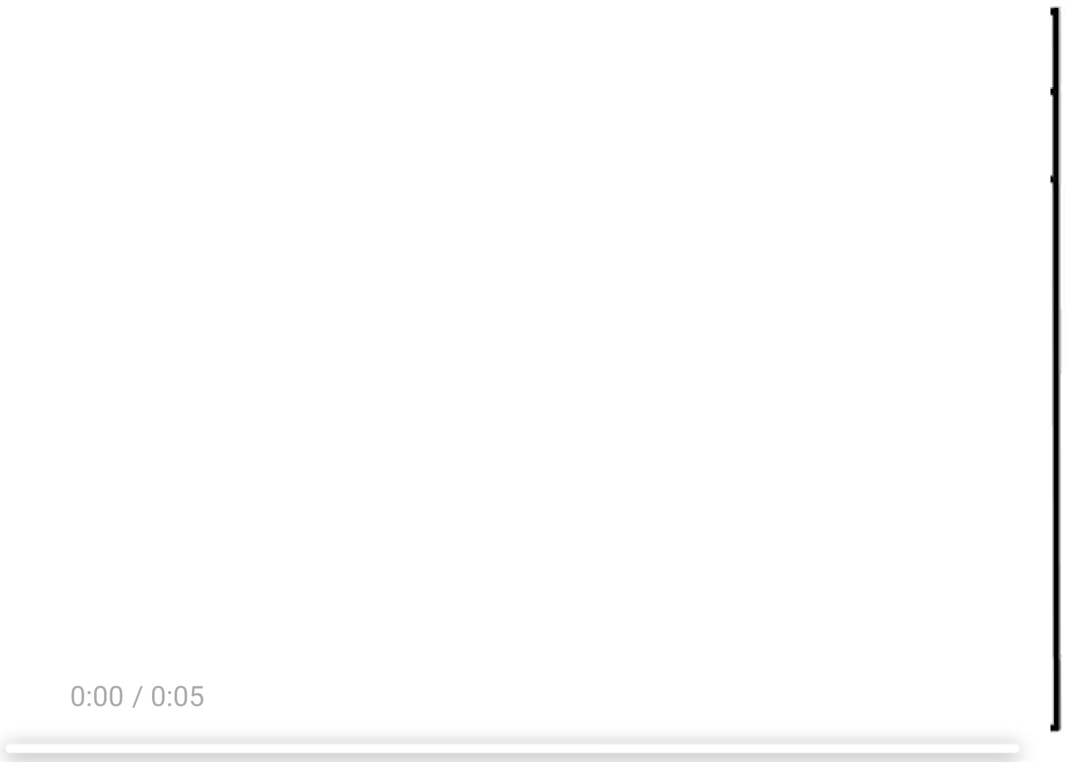


Figure 6: Box layout for a paragraph moving due to line break change

Determining the Layout of a page is a challenging task. Even the simplest page layout like a block flow from top to bottom has to consider how big the font is and where to line break them because those affect the size and shape of a paragraph; which then affects where the following paragraph needs to be.

CSS can make element float to one side, mask overflow item, and change writing directions. You can imagine, this layout stage has a mighty task. In Chrome, a whole team of engineers works on the layout. If you want to see details of their work, few talks from BlinkOn Conference are recorded and quite interesting to watch.

Paint

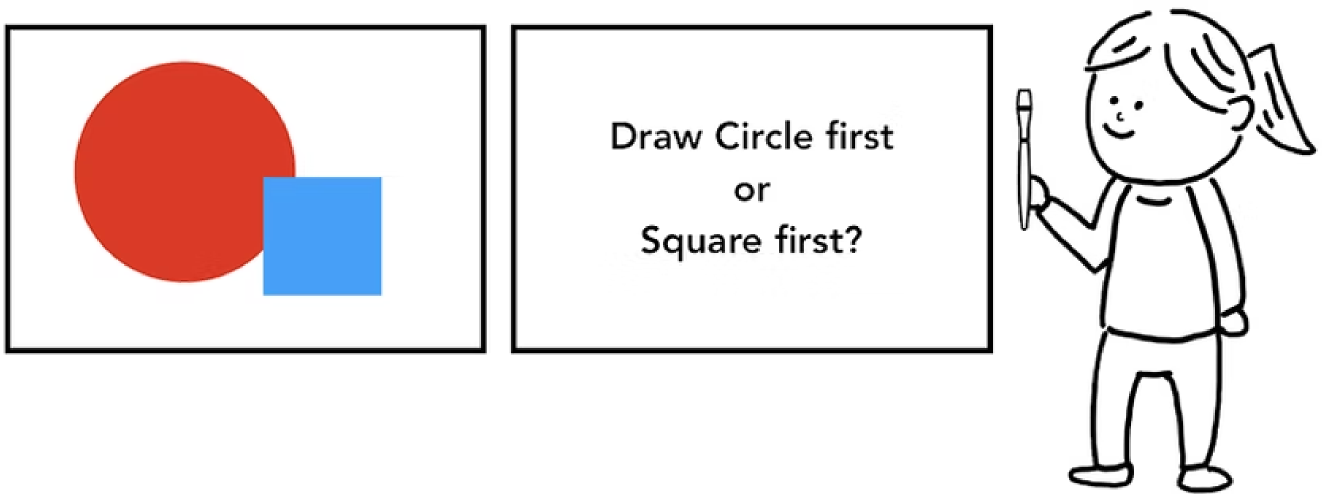


Figure 7: A person in front of a canvas holding paintbrush, wondering if they should draw a circle first or square first

Having a DOM, style, and layout is still not enough to render a page. Let's say you are trying to reproduce a painting. You know the size, shape, and location of elements, but you still have to judge in what order you paint them.

For example, `z-index` might be set for certain elements, in that case painting in order of elements written in the HTML will result in incorrect rendering.

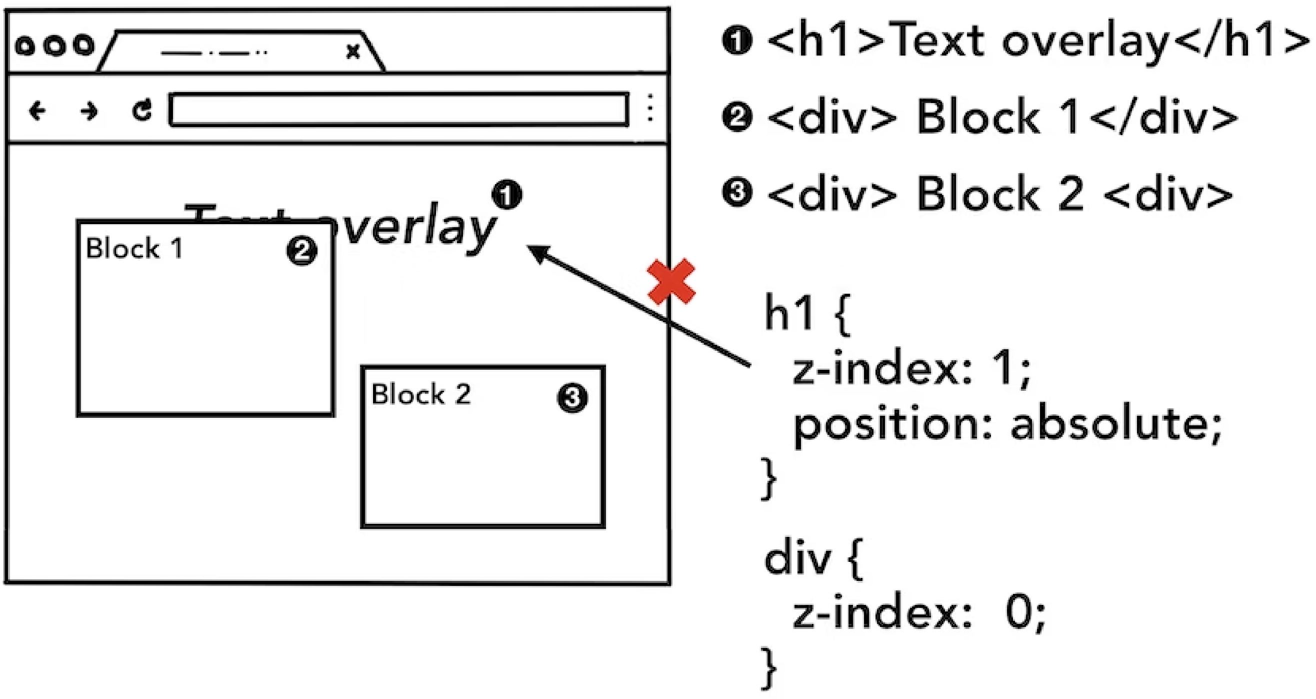


Figure 8: Page elements appearing in order of an HTML markup, resulting in wrong rendered image because z-index was not taken into account

At this paint step, the main thread walks the layout tree to create paint records. Paint record is a note of painting process like "background first, then text, then rectangle". If you have drawn on `<canvas>` element using JavaScript, this process might be familiar to you.

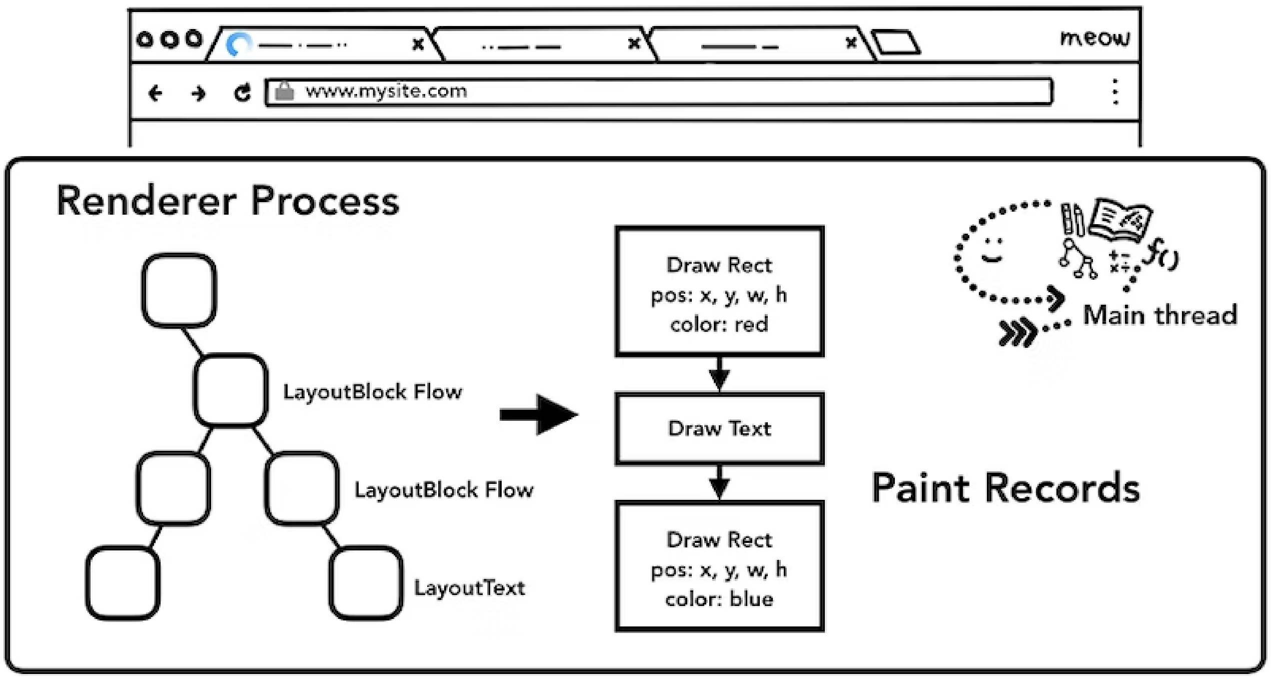
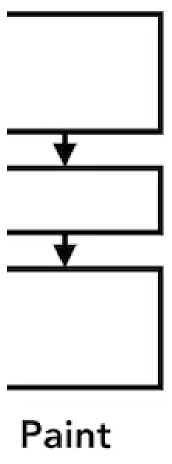


Figure 9: The main thread walking through layout tree and producing paint records

Updating rendering pipeline is costly



0:00 / 0:04

Figure 10: DOM+Style, Layout, and Paint trees in order it is generated

The most important thing to grasp in rendering pipeline is that at each step the result of the previous operation is used to create new data. For example, if something changes in the layout tree, then the Paint order needs to be regenerated for affected parts of the document.

If you are animating elements, the browser has to run these operations in between every frame. Most of our displays refresh the screen 60 times a second (60 fps); animation will appear smooth to human eyes when you are moving things across the screen at every frame. However, if the animation misses the frames in between, then the page will appear "janky".

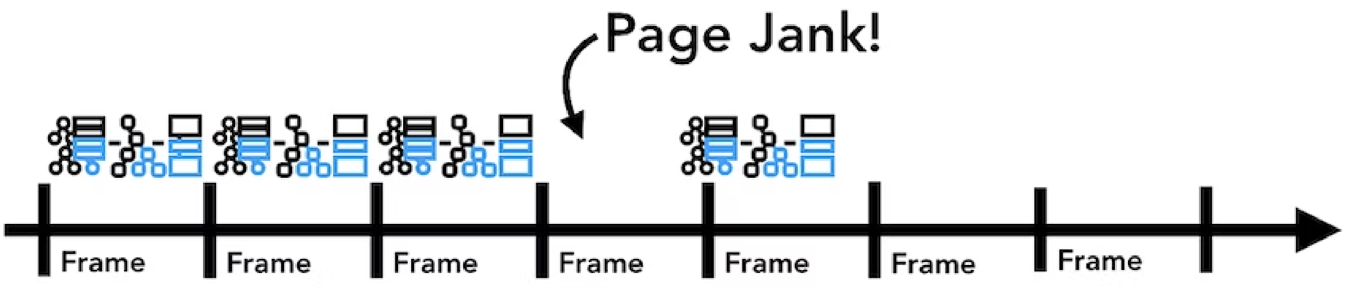


Figure 11: Animation frames on a timeline

Even if your rendering operations are keeping up with screen refresh, these calculations are running on the main thread, which means it could be blocked when your application is running JavaScript.

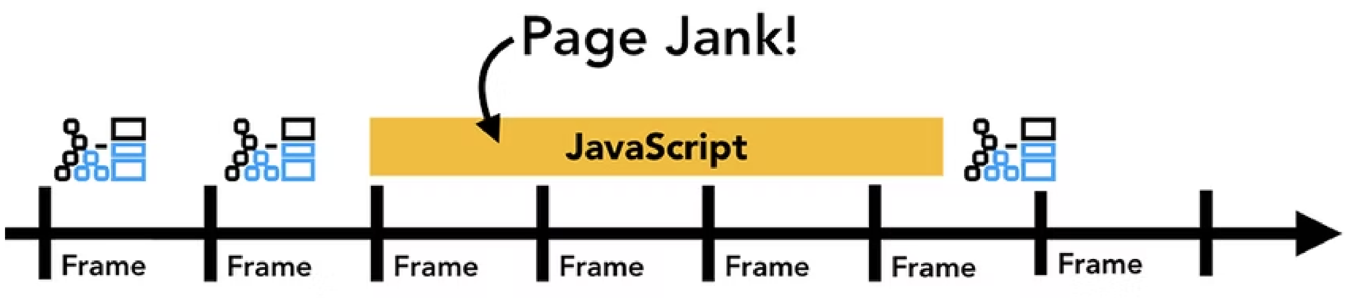


Figure 12: Animation frames on a timeline, but one frame is blocked by JavaScript

You can divide JavaScript operation into small chunks and schedule to run at every frame using `requestAnimationFrame()`. For more on this topic, please see [Optimize JavaScript Execution](). You might also run your [JavaScript in Web Workers]() to avoid blocking the main thread.

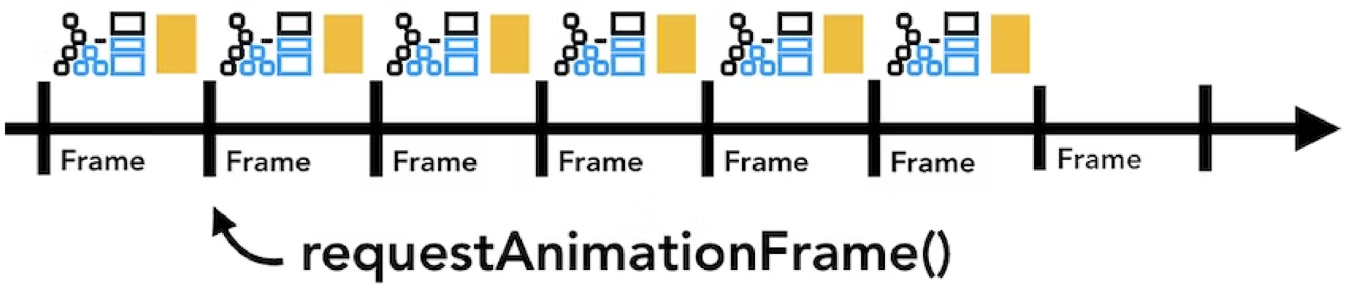


Figure 13: Smaller chunks of JavaScript running on a timeline with animation frame

Compositing

How would you draw a page?

Figure 14: Animation of naive rastering process

Now that the browser knows the structure of the document, the style of each element, the geometry of the page, and the paint order, how does it draw a page? Turning this information into pixels on the screen is called rasterizing.

Perhaps a naive way to handle this would be to raster parts inside of the viewport. If a user scrolls the page, then move the rastered frame, and fill in the missing parts by rastering more. This is how Chrome handled rasterizing when it was first released. However, the modern browser runs a more sophisticated process called compositing.

What is compositing

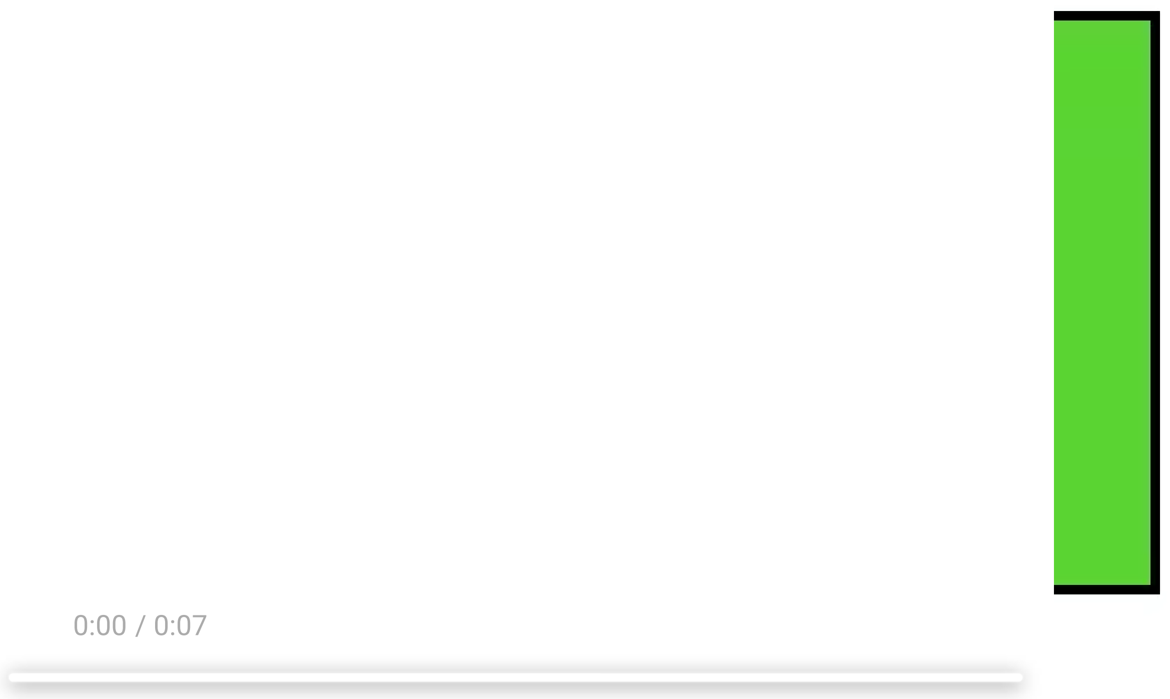


Figure 15: Animation of compositing process

Compositing is a technique to separate parts of a page into layers, rasterize them separately, and composite as a page in a separate thread called compositor thread. If scroll happens, since layers are already rasterized, all it has to do is to composite a new frame. Animation can be achieved in the same way by moving layers and composite a new frame.

You can see how your website is divided into layers in DevTools using Layers panel.

Dividing into layers

In order to find out which elements need to be in which layers, the main thread walks through the layout tree to create the layer tree (this part is called "Update Layer Tree" in the DevTools performance panel). If certain parts of a page that should be separate layer (like slide-in side menu) is not getting one, then you can hint to the browser by using `will-change` attribute in CSS.

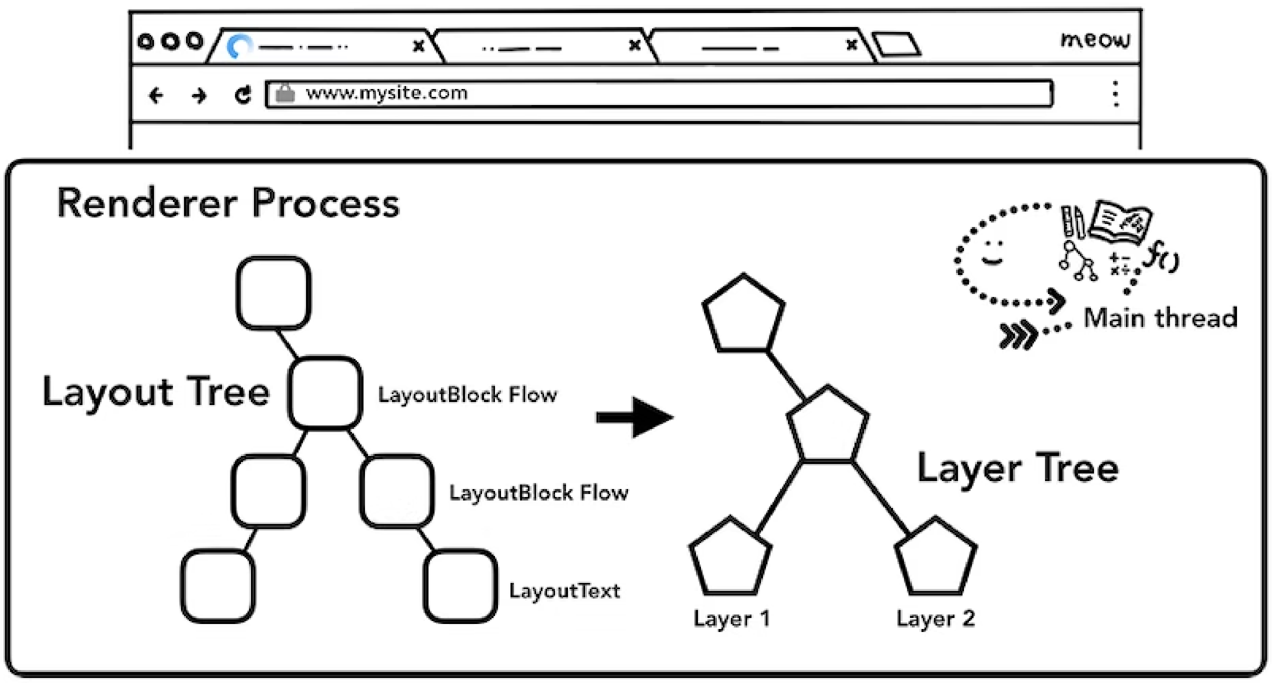


Figure 16: The main thread walking through layout tree producing layer tree

You might be tempted to give layers to every element, but compositing across an excess number of layers could result in slower operation than rasterizing small parts of a page every frame, so it is crucial that you measure rendering performance of your application. For more about on topic, see Stick to Compositor-Only Properties and Manage Layer Count.

Raster and composite off of the main thread

Once the layer tree is created and paint orders are determined, the main thread commits that information to the compositor thread. The compositor thread then rasterizes each layer. A layer could be large like the entire length of a page, so the compositor thread divides them into tiles and sends each tile off to raster threads. Raster threads rasterize each tile and store them in GPU memory.

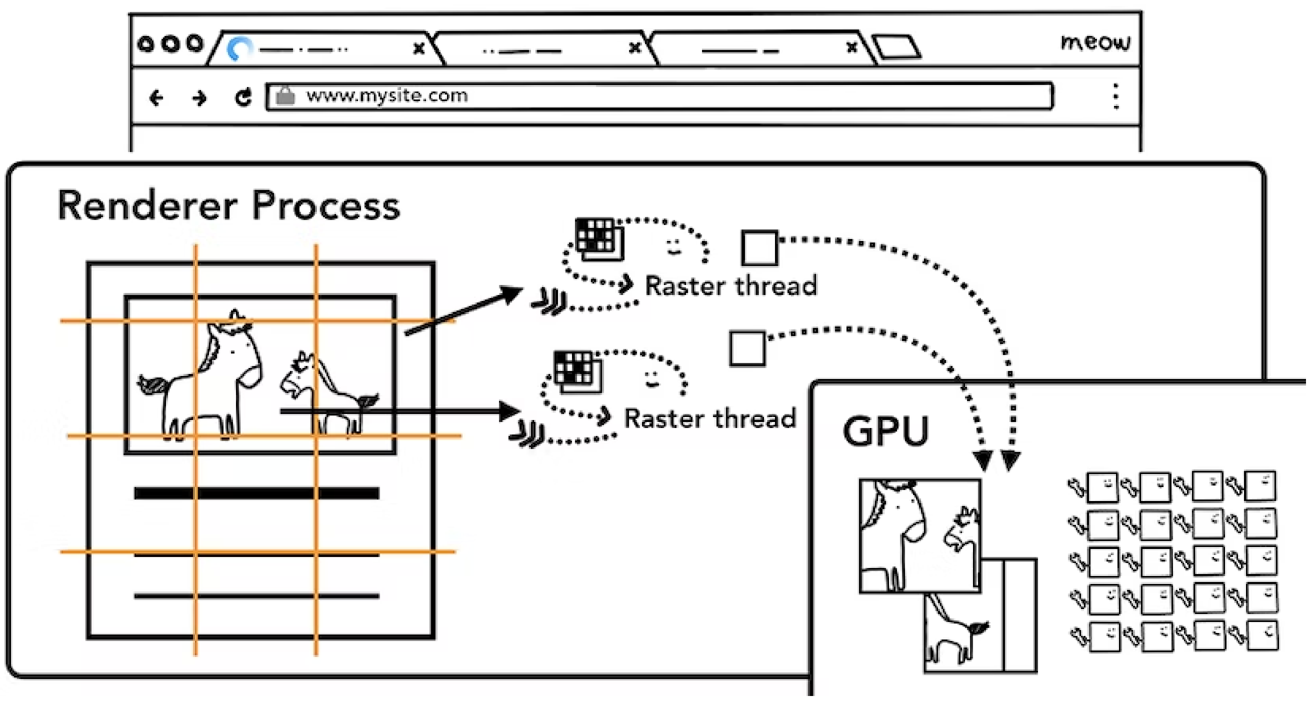


Figure 17: Raster threads creating the bitmap of tiles and sending to GPU

The compositor thread can prioritize different raster threads so that things within the viewport (or nearby) can be rastered first. A layer also has multiple tilings for different resolutions to handle things like zoom-in action.

Once tiles are rastered, compositor thread gathers tile information called **draw quads** to create a **compositor frame**.

Draw quads	Contains information such as the tile's location in memory and where in the page to draw the tile taking in consideration of the page compositing.
Compositor frame	A collection of draw quads that represents a frame of a page.

A compositor frame is then submitted to the browser process via IPC. At this point, another compositor frame could be added from UI thread for the browser UI change or from other renderer processes for extensions. These compositor frames are sent to the GPU to display it on a screen. If a scroll event comes in, compositor thread creates another compositor frame to be sent to the GPU.

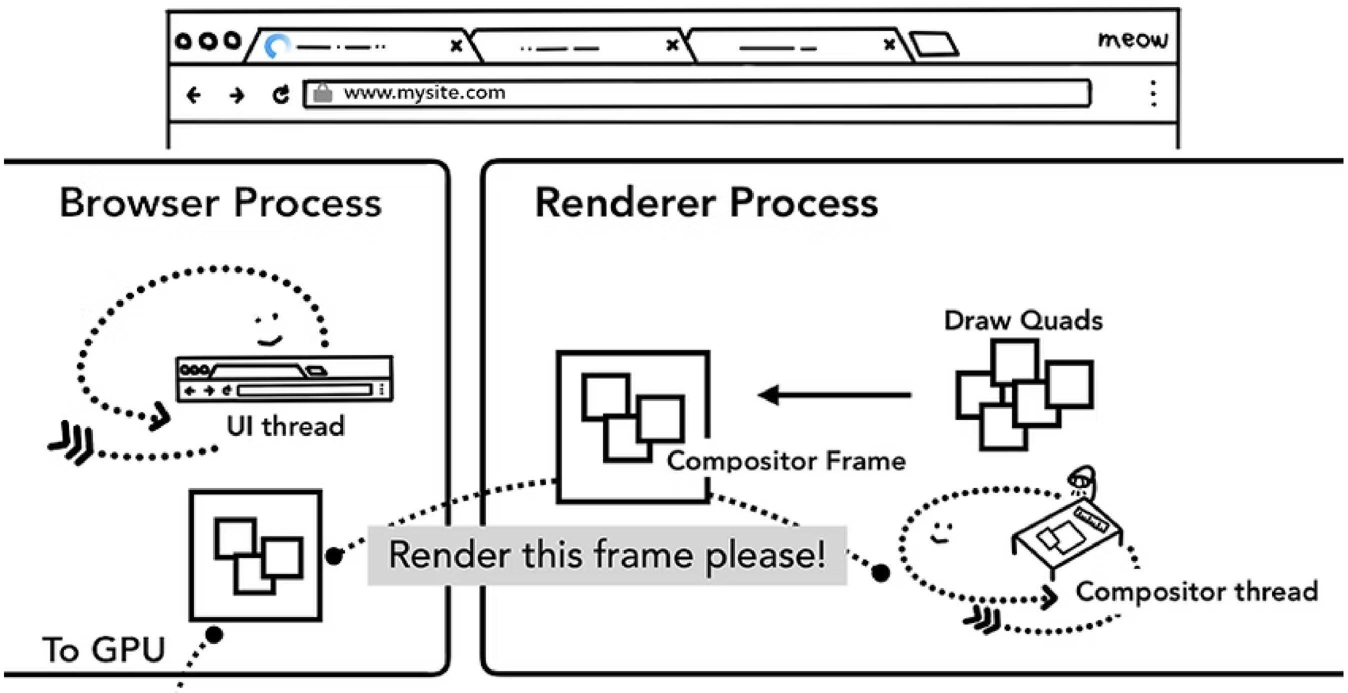


Figure 18: Compositor thread creating compositing frame. Frame is sent to the browser process then to GPU

The benefit of compositing is that it is done without involving the main thread. Compositor thread does not need to wait on style calculation or JavaScript execution. This is why compositing only animations are considered the best for smooth performance. If layout or paint needs to be calculated again then the main thread has to be involved.

Wrap Up

In this post, we looked at rendering pipeline from parsing to compositing. Hopefully, you are now empowered to read more about performance optimization of a website.

In the next and last post of this series, we'll look at the compositor thread in more details and see what happens when user input like `mouse move` and `click` comes in.

Did you enjoy the post? If you have any questions or suggestions for the future post, I'd love to hear from you in the comment section below or @kosamari on Twitter.

Next: Input is coming to the compositor

Last updated: Tuesday, August 18, 2020 Improve article

Follow us

Contribute

File a bug

View source

Related content

web.dev

Case studies

Podcasts

Connect

Twitter

YouTube

GitHub

Google Developers

Chrome Firebase All products Privacy Terms

Choose language

ENGLISH (en)