

Edit Manage Stats



Ingo Steinke

Posted on Sep 6

Inspecting the wrong elements in the browser

#webdev #browser #productivity

The browsers' inspection tool is one of the most helpful tool for testing, validating and sometimes even conceiving and planning frontend features.

"Inspect Element", "Inspector", F12, Ctrl+B, Ctrl+Shift+Q, Ctrl+Shift+C, Cmd+Shift+C, "start element selection", "pick an element from the page", or "select an element in the page to inspect it" – names and details differ, but the basic concept is always the same in any modern browser.

Web development Detective Stories

Before I share a kind of web development detective story for your [edutainment](#), here is another practical use of browsers dev tools and a quick peek into their history.

Removing unwanted parts of a website

Some people use developer tools to remove unwanted parts of third party websites like mandatory sign up overlays, utilizing the fact that we can view but also modify and remove website elements in our browser.

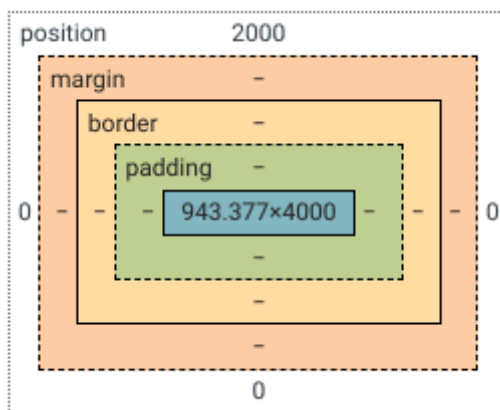
It depends, and there are some edge cases where you can't. Some websites might observe elements in JavaScript and put them back in place after you removed them. But that's a very edgy edge case. Usually, some simple strategies can help us find the element that we are looking for.

Layering 🍷

If there is more than one element at the same location, we might not be able to click on the desired one.

Thinking about how we put multiple elements in the same place or make them overlay will help us here:

Position 🎯



Elements might be displaced from their original [position](#) in the document flow. A typical case is so called absolute-relative positioning, where a parent element is assigned a `position: relative` and its child either `position: absolute` or anything else, like `sticky`, `relative`, apart from the implicit default `static` position.

The following code makes a child element start at the same position as its parent container:

```
.parent {  
  position: relative;  
}  
  
.child {  
  position: absolute;  
  top: 0;  
}
```

(Negative) Margins ⬆️⬆️⬆️

A similar effect can be achieved without explicit positioning by setting a negative top margin on the child element or a subsequent sibling.

Both kinds of overlay are often easy to detect, if the elements can be found not far from each other in the DOM. But they don't have to be.

We can position an overlay element relative to the page and set something like `top: 50%` or add a nested flex box to center the overlay vertically. So it will appear to hover over any part of the page that we are currently looking at, while the DOM element might be defined before the closing `</body>` tag or anywhere else in the DOM.

Overflow 🌊

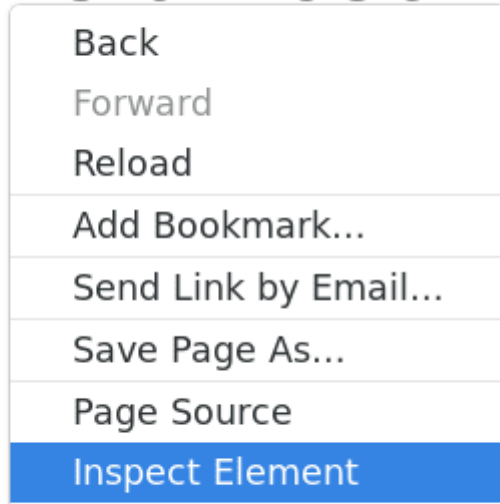
Content can overflow its container, which has been chosen as the default behavior on the premise that it's more important to ensure accessibility and visibility than pixel-perfect design, or to prevent data loss, as Rachel Andrew explained in her Smashing Magazine post about: [Overflow And Data Loss In CSS](#).

That's what's shown in the iconic "CSS is awesome" meme.



In this example, we prevented the text from breaking into multiple lines with `white-space: nowrap` and restricted the border box width to `100px`. We could cut the text by adding `overflow: hidden`, but otherwise it will overflow and possibly overlap unrelated elements.

CSS unrelated content is awesome



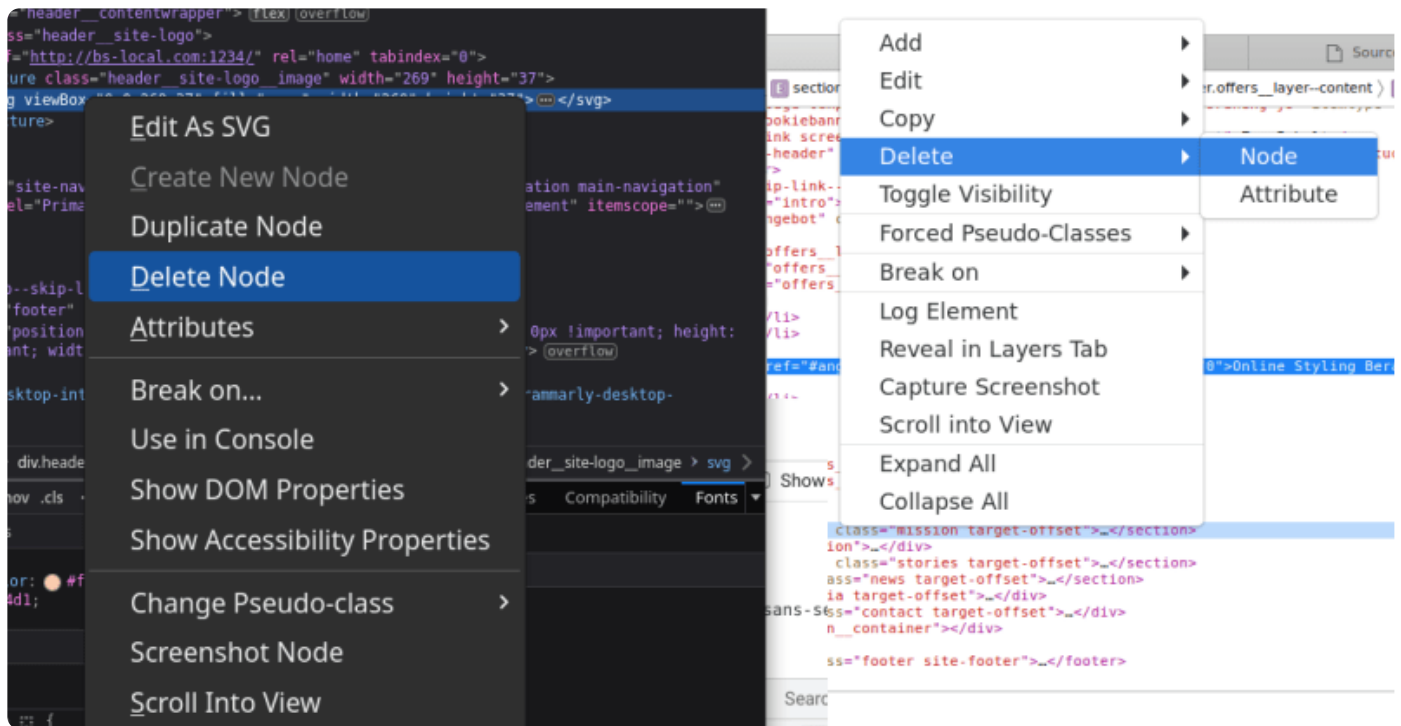
When we point to the word "is" and click "inspect element", which element would we actually inspect?

The browser has no way to know what we intent to select, the word "is" or the word "unrelated". Or maybe we only want to know about the background color behind both, which might be set on a parent container or the document body.

Deleting unrelated elements 🗑️

Like the proverbial sculptor who said to form a statue of an elephant (or [David](#)) by removing anything that does not belong to (David) the elephant, we can remove unhelpful elements until there is nothing left but a minimal version of the page including the element that we have been looking for.

Although deleting elements is named and styled inconsistently across browser developer tools, it is usually found in a context menu when right-clicking on an element in the inspector and sometimes you can also select the DOM element and press the delete key.



(Partially) hiding elements 🤖

So we can delete elements quickly, but what happens if we want to undo deletion? When we were able to use the delete key, there is a good chance that we can undo our action by pressing `strg + z`.

All of this can be done more quickly than focusing on the element's styles and typing `display: none` (or the shorter `d / Tab / n / Tab` making use of auto-completion). Alternatively, `visibility: hidden` will keep their position and effectively behaves the same way like setting a zero opacity.

We could even keep an element and make it semi-transparent by reducing its opacity only partially:

```
opacity: 0.25;
```

Hint: `opacity` can be typed as `op / Tab` using auto-completion.

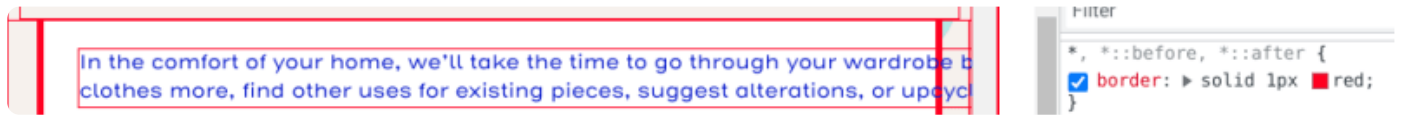
We can also add the classic frontend debugging technique of adding borders in different colors to distinguish elements and see their dimension.

To apply this to any element on the current page, we can add a new style rule using wildcards.

```
*, *::before, *::after {  
  border: solid 1px red;  
}
```

}

This can make it easier and faster to find elements that behave in an unintended way, like this one overflowing beyond the intended container width, at a glance instead of clicking through elements one by one.



Feeling like Schrödinger's Cat 🧪🐱

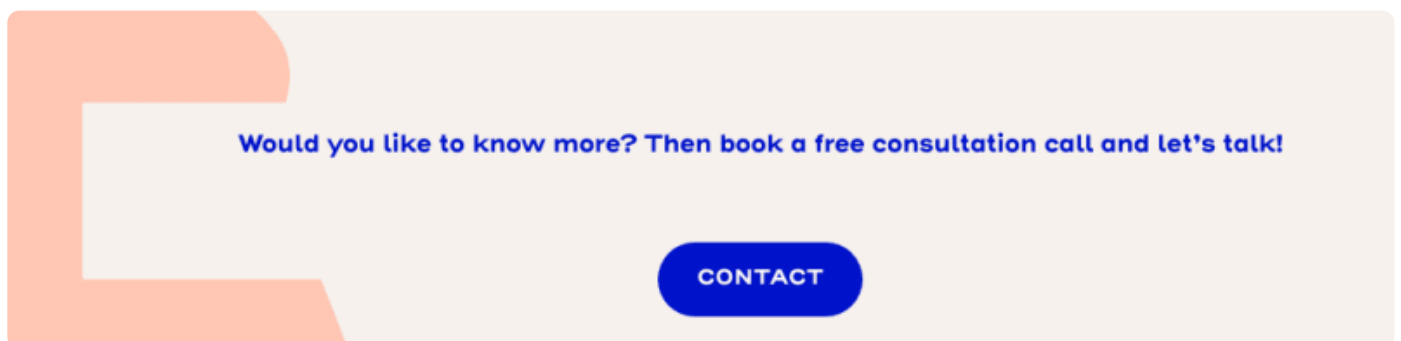
Inspecting elements this way, you might eventually experience something like [Heisenberg's uncertainty principle](#) or [Schrödinger's Cat](#), a thought experiment about the effect that **the act of investigation can alter the state of the subject** that you are investigating.

In frontend web development, this can manifest as an inability to reproduce an error in an investigation context, like when an erratic overflow behavior stops to occur the moment that we apply diagnostic border styles to the elements involved or remove an adjacent element.

Still, experimenting with our page elements in a visual way can provide insights in a more intuitive and playful fashion than mere logical deduction and debugging techniques used in non-visual programming contexts.

A mysterious rectangular background 🧛

Coming back to the actual use case that inspired me to write this post: where does this rectangular box shape come from?

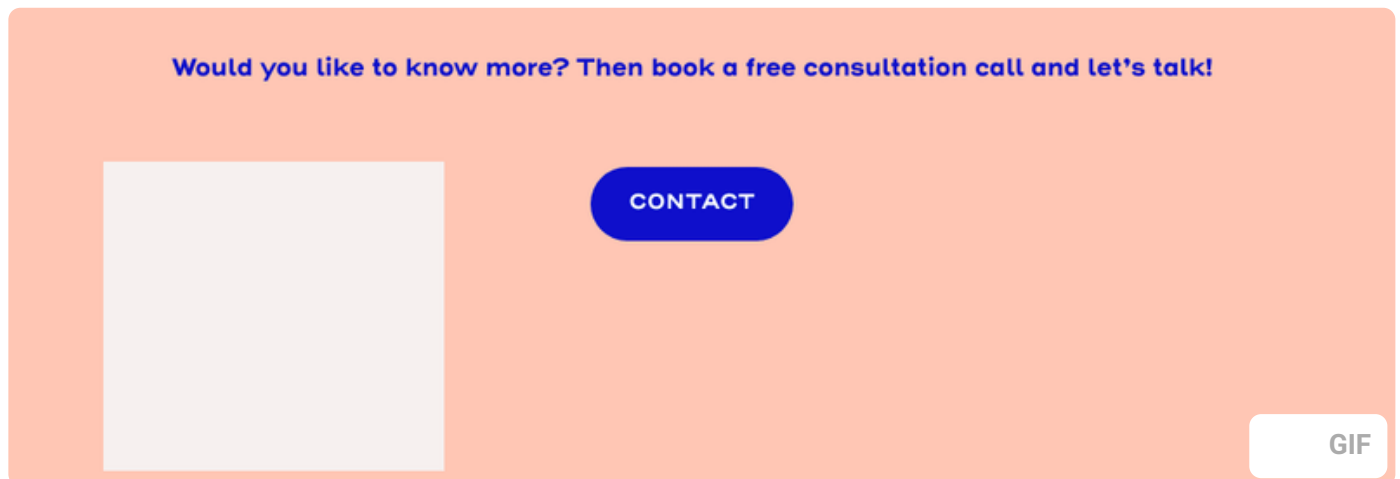


I knew that the apricot shape is defined in a background layer and that some of the foreground elements have explicitly set a background color that matches the default page background. But I was not able to find a matching background setting in any of the few elements.

Coloring the background 🎨

None of the other techniques helped, as this turned out to be a severe example of the uncertainty principle. The mysterious shape was only visible at some rare scrolling positions, and it disappeared as soon as I deleted adjacent elements.

One of my last hopes was another variation of the visual border / background-color approach: I gave the decoration layer an opaque background where the used to be some distinct shapes before. Now it looks like this:



I still haven't traced the origin of that element, but whatever it is, I have learned more about it. Its unexpected shape and its scrolling behavior might be another hint where to find and fix it!

The nearly square shape appears above the decoration layer but gets covered below the subsequent section. A new suspicion formed in my head: this could be some stray hidden child or even pseudo element that should never be visible or present at all. It must be a descendant of the decorated `<main>` element (which holds true for at least 90% of the website) and it must have a lower `z-index` than the element covering it, luckily no greater number than a modest `2`.

Exclusion principle 🙅

We're finally getting close! There should not be many elements left that match our updated criteria.

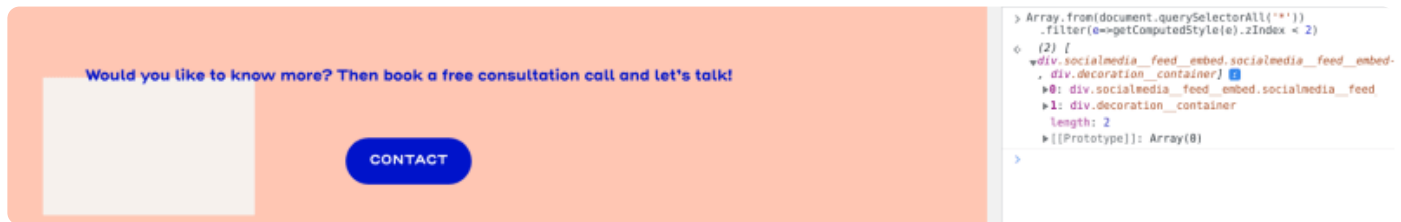
Query selectors, filters, and computed styles 🧩

Can we use a CSS selector or its JavaScript / DOM API equivalent, a [query selector](#), to single out all elements that have a certain `z-index` (very few)?

No, but we can iterate over all suspicious elements and filter them by their computed styles. [Find element with specified z-index without JQuery](#):


```
Array.from(document.querySelectorAll('*'))  
  .filter(e=>getComputedStyle(e).zIndex < 2)
```

We can even put this in our developer tools' JavaScript console.



The results look promising although I would have expected to find much more elements that match my criteria. But by setting a `z-index: 2` to most top-level elements, I must have done the same to their descendants implicitly.

We could check the number of elements that do have a computed z-index of 2, to verify our assumptions and apparatus before proceeding with our quest, or we could skip that and inspect those two final suspects, decorate them with debug borders — and find out that none of them has anything to do with the mysterious shape!

Innocent until proven guilty

While our methods are helpful, we have used them based on a false conclusion. The fact that an element gets covered below another does not imply that it must have a lower z-index. It might have the same one when it comes before the covering element in the DOM.

But how can we filter elements that precede a known other element? Probably there is a smart and elegant query for that as well, but I would rather have another visual look at the DOM tree and ignore anything starting with the covering element.




Further experiments show that the phenomenon is somehow connected to a carousel container section that provides an element carousel in a simple and accessible fashion based on CSS scroll behavior.

A composite image featuring a man in a video call, a large quote mark, and a GIF of a man in a suit.

Cross-browser validation

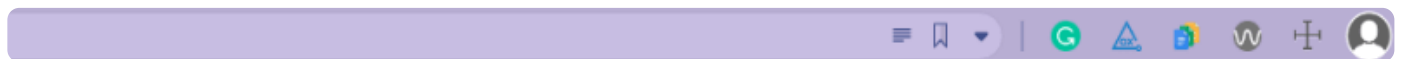
To avoid wasting time or making mistakes, we must wrap up what we did to achieve and isolate the phenomenon:

- 
- The screenshot shows the homepage of 'kleiderordnung'. The navigation bar includes 'SERVICES', 'MISSION' (highlighted in orange), 'STORIES', 'NEWS', 'CONTACT' (highlighted in blue), and 'DE/EN'. The main content area features a large blue rectangle with the text 'Would you like to know more? Then book a free consultation call and let's talk!' and a blue 'CONTACT' button. The right side of the image is partially obscured by a red heart shape.

(Hint: note the different UI themes used to distinguish my open browser windows more easily.)

Okay, so it looks like it's a browser issue. But even if it is browser-related and can't be reproduced in all but one single browser engine — no matter if it's latest Chrome or an obscure Safari release that only your customer's CEO uses regularly — I don't want any mysterious square shapes to appear uncontrollably where they shouldn't!

Maybe it's not about my browser's rendering engine (Vivaldi is based on Chromium after all) but rather my specific setup, including invasive extensions like axe, WAVE, Dimensions or Grammarly?



Deactivating browser extensions

Extensions, also known as plugins might be developed with a helpful intention, but they can still do harm in the form of messing up my (and my customers') document element tree.

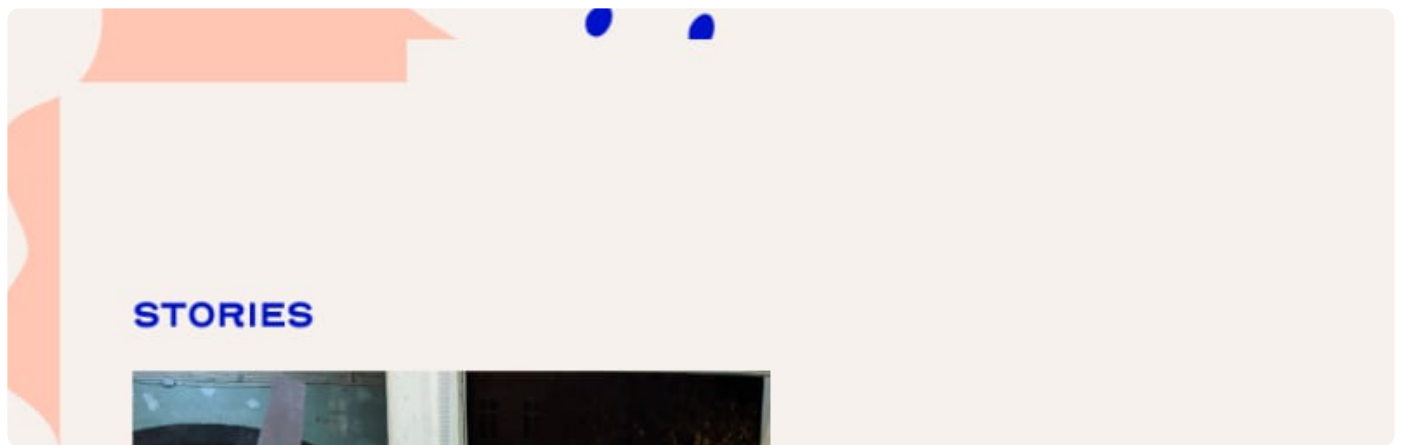
Let's have a look at what's inside my rendered DOM that I never explicitly included in the website.

```
<!DOCTYPE html>
<html lang="en-US" style="--header-height: 122px; --header-height--stuck: 122px;">
  <head>
    <body class="home page-template-default page page-id-65 theme-1234 js" itemtype="https://schema.org/WebPage" itemscope data-new-gr-c-s-check-loaded="14.1119.0" data-gr-ext-installed>
      <div id="cnp1z-cookiebanner-container"></div>
      <a class="skip-link screen-reader-text" href="#wp--skip-link--target">Skip to content</a>
      <header id="site-header" class="header header--site-header has-inline-mobile-toggle stuck" aria-label="Site" itemtype="https://schema.org/WPHeader" itemscope></header>
      <main id="wp--skip-link--target"></main>
      <footer id="footer" class="footer site-footer"></footer>
    </body>
    <grammarly-desktop-integration data-grammarly-shadow-root="true"></grammarly-desktop-integration>
  </html>
```

It might or might not be relevant to my detective work, but that's one element that I should have deleted upfront. Or, more easily, I should have deactivated my browser extensions unless I actually need them for a specific purpose.

Deactivating all browser extension and restarting my browser did not make the problem go away, but it was still important to try.

Restarting my browser reset my ad-hoc changes to the DOM and closed the developer tools, so I noticed another phenomenon: there is not one, but two mysterious shapes when the window is wide enough! I can also open my developer tools below the website to continue debugging without losing that view.



The second shape goes away when I set a solid background color, which is why I didn't notice it before. Maybe the whole phenomenon still has something to do with my decoration layer. 🤔

Another new suspicion: there might be something (formally) wrong with my background definition, using the multiple background syntax to add various shapes in various sizes and positions. What could possibly go wrong? It looked maintainable and readable at least in my source code.

```
.decoration__container {
  background-image:
    var(--background-image-pita),
    var(--background-image-salmon),
    var(--background-image-lilac),
    var(--background-image-bluedots),
    var(--background-image-apricot),
    var(--background-image-salmon2);
  background-position:
    100% 0%,          /* top right pita shape */
    -20% 800px,       /* top left salon shape */
    110% 900px,       /* middle right lilac shape */
    20% 1999px,       /* blue dots */
    0% 1800px,        /* lower left apricot shape */
    120% 1900px;      /* lower right salmon shape */
  background-size:
    31.25rem auto, /* 500px auto, */
    40% 40%,
    40% 40%,
    6.25rem auto,  /* 100px auto, */
    30% 30%,
    30% 30%;
  background-repeat:
    no-repeat,
    no-repeat,
```

```
no-repeat,  
no-repeat,  
no-repeat,  
no-repeat;
```

But it looks like a mess in my dev tools!

```
var(--background-image-lilac),  
var(--background-image-bluedots),  
var(--background-image-apricot),  
var(--background-image-salmon2);  
background-position:  
100% 0%, /* top right pita shape */  
-20% 800px, /* top left salon shape */  
110% 900px; /* middle right lilac shape */
```

```
z-index: 1;  
background-repeat: > no-repeat,no-repeat,no-repeat,no-repeat,no-repeat,no-repeat;  
background-image: url(data:image/svg+xml,%3Csvg width='880' height='1191' viewBox='0 0 880 1191' fill='none' xmlns='h-  
'>url(data:image/svg+xml,%3Csvg width='1011' height='841' viewBox='0 0 1011 841' fill='none' xmlns='h-'),url(  
data:image/svg+xml,%3Csvg width='1025' height='905' viewBox='0 0 1025 905' fill='none' xmlns='h-'),url(  
data:image/svg+xml,%3Csvg width='92' height='105' viewBox='0 0 92 105' fill='none' xmlns='http-'),url(  
data:image/svg+xml,%3Csvg width='778' height='762' viewBox='0 0 778 762' fill='none' xmlns='http-'),url(  
data:image/svg+xml,%3Csvg width='1892' height='885' viewBox='0 0 1892 885' fill='none' xmlns='h-');  
background-position: > 100% 0%, -20% 800px, 110% 900px, 20% 1999px, 0% 1800px, 120% 1900px;  
background-size: 31.25rem auto, 40% 48%, 40% 40%, 6.25rem auto, 30% 30%, 30% 30%;
```

There are not warning signs or strike-through in the browser, and stylelint did not complain either. But there is still a possibility that the browser misunderstood my code.

Questioning my current strategy (again) 🙄

Does it make sense to follow this path of examination, or am I just desperately clinging on to what seems one of the last options before I give up.

Do I have any chance to get help from a fellow developer?

I haven't been able to provide a reproducible minimal example, not even inside my own personal default browser. At least this time I can surely tell what I already tried.

Consulting my pillow 🛌

An underrated strategy: call it a day and sleep over it. Go for a walk if it's only midday, or switch tasks and work on something else for the rest of the day.

New ideas often come easier when we stop staring at the same task for too many hours.

Playing for time or luck 🕒 🎰

If this was for a customer's project and it still was my own Linux browser and none of their management's Safari or Edge clients, I would play for time or hope to stay lucky, follow the [80/20 Pareto rule](#) and stop wasting time on a bug that maybe nobody will see except for me.

Sharing experience and knowledge 🙌

Talking about economy, I think I have already wasted too much time investigating a single strange phenomenon. But I can still turn it into some kind of [Productive Procrastination](#) by yielding a long-term value eventually.

I did not feel ready to share my problem in a way that other might come up with a solution quickly, but I can still prepare to share, order my thoughts by trying to explain my story in an understandable and possibly entertaining way, despite its lengthy details.

Talking about it

Talking about educational entertainment, a case like this one might even make a good story to tell as a conference talk or on YouTube.

Hoping for a solution, but can't reproduce instead

So, after I took a walk, had some sleep, spent the next days working on completely different issues and **updated my browser** to the latest version, the mysterious problematic element eventually disappeared!

¬\('ツ)/¬

which I feel is kind of bad, as I always prefer "today I learned" to "can't reproduce" because without knowing what happened, how can I guarantee that it won't happen again?

If you have a lot of spare time and feel like a detective, you could

- a) find out which website I have been working on
- b) check out git tag 2.8.2 or any other probable commit
- c) guess which Vivaldi browser version I used before upgrading
- d) check Vivaldi's and/or Chromium's release notes
- e) verify that you can reproduce the issue
- f) and that you can't after upgrading.

But on the other hand, especially considering a situation in a paid project where we strive for effectiveness rather than perfection, it's fair enough to close some issues exactly like that or even accept some minor bugs that only ever occur in rare edge cases.
