DOM performance case study

areknawo.com/dom-performance-case-study

I have an interesting question for you - when have you last used truly pure **DOM API** and methods to build a real project? Yeah, can't remember these times either. 😂 But did they even existed at all? Because, you know, you almost always use HTML with the help of DOM API to do some more interactive stuff, but you definitely don't use it as a standalone way of creating your UI. But, with the modern **UI frameworks** and **libraries**, like **React**, **Vue** or **Angular** times have changed and so the way of creating UIs too. So, unless you're using some framework that **compiles your code** to HTML/CSS/JS trio, you're most likely to base your app on some tool that bases on DOM API. 😉 With that said, the level of control that these tools provide us with is incredible. It truly helps to create better, prettier and **faster** experiences. Yeah, speed - that's what we'll be looking into today.

As you might know or heard somewhere, any interaction with DOM is **costly**. These calls can give you a **big performance hit** when you're not using it correctly. Even if we're talking about fractions of milliseconds, it's still important. If your UI cannot work butter-smoothly, locked at **60 FPS** (+1/-1) then something is not right. But it shouldn't be the case with your Vue/React/Angular app unless of course, you have done something truly bad or performed demanding tasks (BTC mining, WebGL, AI, and other number-crunching stuff 😁). That's because of how well optimized these tools are. So, let's do a case-study here and check some **DOM optimization techniques**, including that these libraries use, to know how it's done! **Enjoy!** 👍

Reflows

Starting with the most notorious one, here comes the **reflow** - your worst enemy and best friend at once. Reflow (also called **layout trashing** 🗑) is the name for all the processes that take place in your browser when you interact with DOM, CSS and all those kind of stuff. It means re-renders and re-calculations of your website's layout (element's positions and size). All that is nice - reflows handle all these complexities behind the scenes. Let's move on to the worse part then - reflow is a **user-blocking** operation! That means if there's too much work to do when performing reflow, your UI can **drop its**

frame rate, freeze or - in the worst scenario - even crush. These are all experiences that you probably don't want your users to have. With that said, it's important to deal with DOM and thus resulting in reflows with special care.

What exactly triggers the reflow then? There is a **great list** in form of GitHub gist if you would like to know more. But here let's take a quick look at the most important of 'em all:

- `getComputedStyle()` - extremely useful and **extremely costly**;
- **box metrics** and **scrolling** - stuff like `clientHeight`, `scrollTop`;
- **window properties** - `clientHeight`, `scrollY`;
- **events' position data** & **SVG**

So these are just the basic, more generic ones. Of course, some tasks like accessing a property have less performance overhead (reflow timing) than some more advanced methods like `getComputedStyle()`.

Batching

So, reflows aren't really good. What can we do to minimize them or at least optimize them to gain performance boost? 🚀 Well, quite a lot actually. First, the best and most popular technique is known as **batching**. What it basically means is that you should **group** your DOM **read and write operations** and commit them separately whenever possible. This process allows the browser to optimize your calls under-the-hood and results in overall improvement in performance.

```
const width = element.clientWidth + 10;
const width2 = element.clientWidth + 20;

element.style.width = width + 'px';
element.style.width = width2 + 'px';
```

```
const width = element.clientWidth + 10;
element.style.width = width + 'px';
const width2 = element.clientWidth + 10;
element.style.width = width2 + 'px';
1234567891011121314
```

Apart from that, you should also batch and **reduce** any other kind of DOM interactions. For example, let's take the standard way of adding a new element to your DOM tree. When you're adding just one or two it might not be worth the extra trouble. But when we're talking about **tens** or **hundreds** of elements, then it's really important to commit such call properly. What do I mean by it? Well, to just batch all of these calls into one, most likely with the help of `DocumentFragment`.

```
for(let i = 0; i < 100; i++){
   const element = document.createElement('div');
   document.body.appendChild(element);
}
```

```
const fragment = document.createDocumentFragment();
for(let i = 0; i < 100; i++){
   const element = document.createElement('div');
   fragment.appendChild(element);
}
document.body.appendChild(fragment);
12345678910111213
```

Such a simple change can lead to a big difference. I think, it goes without saying that you should apply the same practice/idea **whenever** and **wherever** you can. Besides that, what can also prove to be useful are your

browser's **dev tools**. You can use its **rendering timeline** to see all relevant data about how your DOM was rendered. Of course, it's only useful when you then put proper optimizations in place.

Miscellaneous

Now, let's talk about more general stuff. Most obvious advice will be just to keep things simple. But what does it mean in depth?

- **Reduce DOM depth** - Unnecessary complexity just makes things slower. Also, in many cases, when you update the parent node, the children may need to be updated to thus resulting in the **whole structure** formed under the specified node needed to be processed. The update might also invoke change all the way up the DOM tree. In short, it makes the reflow take more time.
- **Optimize CSS** - Naturally, the CSS rules that are not used aren't really needed at all. You should remove any of those. Next, complex **CSS selectors** can also cause a problem. But, if you have already followed the previous rule, these may prove to be useless, leaving no need for them in your code whatsoever. **Inlining** the styles that you change often is a good practice too. Obviously, in contrast, styles that are used by a number of elements should be made separately as a **CSS rule**.
- **Animations** - These can hit it pretty hard. You should limit your animations whenever possible only to transform and opacity properties. Also, it's always better to include them **out-of-the-flow**, meaning to set the `position` to either `absolute` or `fixed`. This ensures that your animations won't interfere with the rest of UI, causing even slower reflows. Besides that, let your browser know that specified properties are going to change by utilizing the `will-change` property. And lastly, you might want to animate using **CSS animations** or **Web Animations API**. This way all your animations are executed in special, separate **"compositor thread"** thus making them **non-blocking**.

These tips can improve your performance drastically! So, just use them anytime you can.

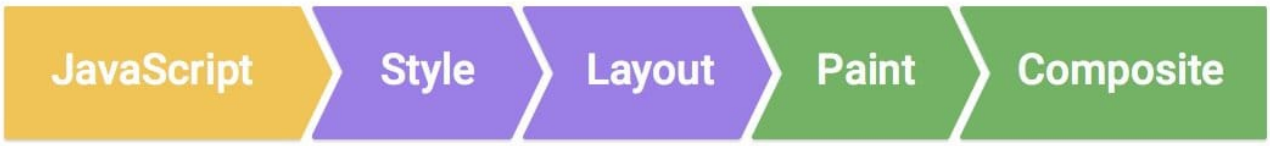
Photo by Scott Webb / Unsplash

From different perspective

Now that we know that reflows that handle the view updates for us are the root of all evil 😈, let's sum it up and take a look at all previous info from a bit different perspective.

Everything that happens on your screen should retain that hyped **60 FPS** that everyone craves. It means that the screen should refresh **60 times per second** (or more for devices with higher refresh rates). And what it means even more specifically is that everything that happens on this one, single frame (JS, reflows and etc.) should happen under **10 ms** (in fact you have around 16 ms but browser uses this 6 ms for internal housekeeping stuff). With that said, when the task is too big and it takes too long (more than 10 ms) the frame rate **drops** and lags happen.

Let's take a look at this diagram to see **what exactly happens** on this single frame:



What happens on a single frame - diagram taken from developers.google.com

I think **JavaScript** part needs no further explaining other than that it is what usually **triggers the visual changes** (it can also be CSS animations, Web Animation API and etc.).

Style marks the time when **style calculations** take place. Here all your CSS rules are processed and **applied** (CSS selectors stuff).

Layout and paint steps are the most important for us here because these can be easily optimized. **Layout** step is the reflows origin place. Here, after your styles have been already applied in the previous step, the properties that may require **geometry recalculation** are being handled. This includes `width`, `height`, `left`, `top` and etc. The change of these properties may require to **update other elements**, including the ones down and top the DOM tree.

What you can do to optimize this step is to either manage changes to these properties wisely or have a good DOM hierarchy that doesn't require too many changes on one element update. Of course, you can also change the `position` property. An element that is outside of normal flow **won't trigger a change** in other elements. When no layout property is changed, the browser **omits this step**.

After that comes the **paint** step. Here properties that don't interfere with layout are handled. These include `background`, `color`, `shadow` and alike. Generally pure visuals. Repaints aren't as costly as layout changes and (just like before) are **omitted when not needed**.

Composite is a final, always required step. Here all previously created layers are glued together to for the final result. This will be later painted pixel by pixel to your screen.

I think these insides into how all this happens can really inspire you to further dig into how you can optimize your code. In addition, if you think that your application is fast enough without any optimization, just think what you could do with this **extra computing power** - more visuals, better animations - options are practically endless! 🌟

A word on virtual DOM

After all these tricks and tips, I think you can now easily understand what's so magical behind this whole **virtual DOM** thing that has been lately so popular mainly to big influence that **React** and **Vue** have. It allows you to keep your visual nodes' data in a form of JS native structures, thus not requiring to access DOM (with reflows and stuff as a result)!

So, how does it work in a nutshell? Well, you first interact with the VDOM and apply your changes to it. Then (I might have skipped some more detailed things, but it's that much important 😁) comes the **reconciliation** step. Here the new VDOM tree is **compared** with the old one to differentiate the changes. These are later applied to real DOM.

Now, the reconciliation step is where the discussion like React vs Vue (performance-wise) really has its origins. This comparison is practically the most important and crucial idea behind what's known to many as virtual DOM. This is the place where React 16 (**React Fibre**) has done awesome work on optimizations. But Vue is equally impressive, with its virtual DOM implementation being able to **selectively choose** which nodes are needed to be updated (instead of how React does it - by updating the whole tree down). Anyway, these two has done a really good job on **improving the performance** and **development experience** of great many JS programmers, so big **thumbs up** for that! 👍