Houdini: Demystifying CSS

By Surma

Surma is a contributor to Web**Fundamentals**

Note: I have updated each section of this article with the current state of the respective spec.

Have you ever thought about the amount of work CSS does? You change a single attribute and suddenly your entire website appears in a different layout. It's kind of *magic* in that regard. (Can you tell where I am going with this?!) So far, we – the community of web developers – have only been able to witness and observe the magic. What if we want to come up with our own magic? What if we want to *become the magician*? Enter Houdini!

The Houdini task force consists of engineers from Mozilla, Apple, Opera, Microsoft, HP, Intel and Google working together to expose certain parts of the CSS engine to web developers. The task force is working on a *collection of drafts* with the goal to get them accepted by the W3C to become actual web standards. They set themselves a few high-level goals, turned them into specification drafts which in turn gave birth to a set of supporting, lower-level specification drafts. The collection of these drafts is what is usually meant when someone talks about "Houdini". At the time of writing, the list of drafts is incomplete and some of the drafts are mere placeholders. That's how early in development of Houdini we are.

Caution: I want to give a quick overview of the Houdini drafts so you have an idea of what kind of problems Houdini tries to tackle. As far as the current state of the specs allow, I'll try to give code examples, as well. Please be aware that all of these specs are *drafts* and very volatile. There's no guarantee that these code samples will be even remotely correct in the future or that any of these drafts become reality.

The specifications

Worklets

(spec)

Worklets by themselves are not really useful. They are a concept introduced to make many of the later drafts possible. If you thought of Web Workers when you read "worklet", you are not wrong. They have a lot of conceptual overlap. So why a new thing when we already have workers? Houdini's goal is to expose new APIs to allow web developers to hook up their own code into the CSS engine and the surrounding systems. It's probably not unrealistic to assume that some of these code fragments will have to be run *every. single. frame*. Some of them have to by definition. Quoting the Web Worker spec:

> Workers [...] are relatively heavy-weight, and are not intended to be used in large numbers. For example, it would be inappropriate to launch one worker for each pixel of a four megapixel image.

That means web workers are not viable for the things Houdini plans to do. Therefore, worklets were invented. Worklets make use of ES2015 classes to define a collection of methods, the signatures of which are predefined by the type of the worklet. They are light-weight and short-lived.

CSS Paint API

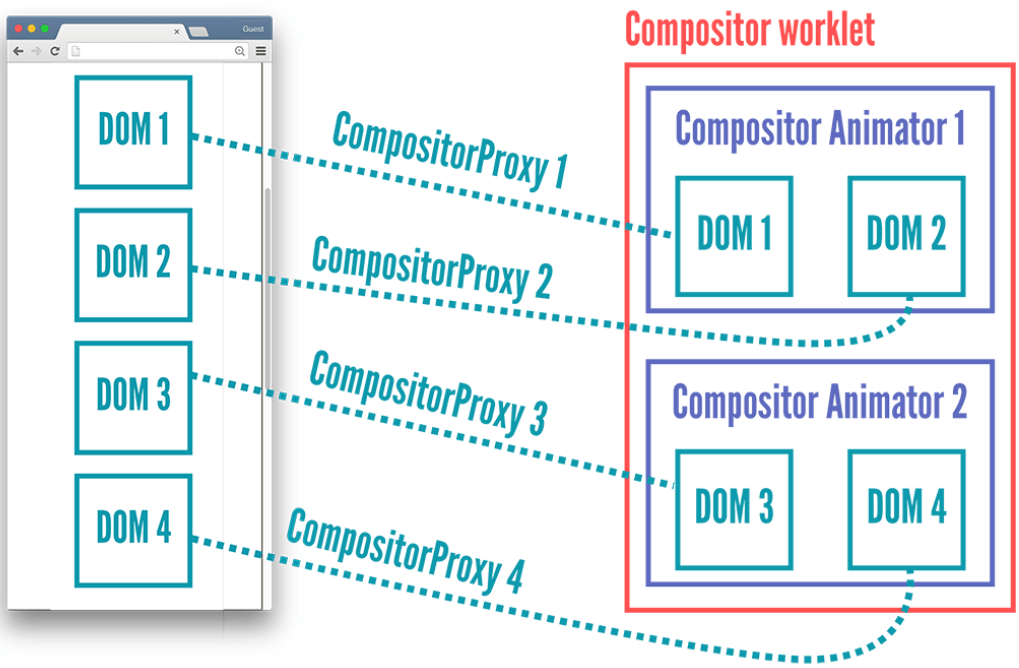
(spec)

Status update: Paint API is enabled by default in Chrome 65. Read the detailed introduction.

Compositor worklet

Note: The API described here is obsolete. Compositor worklet has been redesigned and is now proposed as “Animation Worklet”. More details on the current iteration of the API can be found here.

Even though the compositor worklet spec has been moved to the WICG and will be iterated on, it’s the one the specs that excites me the most. As you might know, some operations are outsourced to the graphics card of your computer by the CSS engine, although that is dependent on both your grapics card and your device in general. A browser usually takes the DOM tree and, based on specific criteria, decides to give some branches and subtrees their own layer. These subtrees paint themselves onto it (maybe using a paint worklet in the future). As a final step, all these individual, now painted, layers are stacked and positioned on top of each other, respecting z-indices, 3D transforms and such, to yield the final image that is visible on your screen. This process is called “compositing” and is executed by the “compositor”. The advantage of this process is that you don’t have to make *all* the elements repaint themselves when the page scrolls a tiny bit. Instead, you can reuse the layers from the previous frame and just re-run the compositor with the changed scroll position. This makes things fast. This helps us reach 60fps. This makes Paul Lewis happy.



As the name suggests, the compositor worklet lets you hook into the compositor and influence the way an element’s layer, which has already been painted, is positioned and layered on top of the other layers. To get a little more specific, you can tell the browser that you want to hook into the compositing process for a certain DOM node and can request access to certain attributes like scroll position, `transform` or `opacity`. This will force this

element on to its own layer and *on each frame* your code gets called. You can move your layer by manipulating the layers transform and change its attributes (like `opacity`) allowing you to do fancy-schmancy things at a whopping 60 fps. Here's a *full* implementation for parallax scrolling using the compositor worklet.

```
// main.js
window.compositorWorklet.import('worklet.js')
  .then(function() {
    var animator = new CompositorAnimator('parallax');
    animator.postMessage([
      new CompositorProxy($('.scroller'), ['scrollTop']),
      new CompositorProxy($('.parallax'), ['transform']),
    ]);
  });

// worklet.js
registerCompositorAnimator('parallax', class {
  tick(timestamp) {
    var t = self.parallax.transform;
    t.m42 = -0.1 * self.scroller.scrollTop;
    self.parallax.transform = t;
  }

  onmessage(e) {
    self.scroller = e.data[0];
    self.parallax = e.data[1];
  };
});
```

My colleague Robert Flack has written a polyfill for the compositor worklet so you can give it a try – obviously with a much higher performance impact.

Layout worklet

(spec)

Note: First real spec draft has been been proposed. Implementation is a good while away.

Again, the specification for this is practically empty, but the concept is intriguing: write your own layout! The layout worklet is supposed to enable you to do `display: layout('myLayout')` and run your JavaScript to arrange a node's children in the node's box. Of course, running a full JavaScript implementation of CSS's `flex-box` layout will be slower than running an equivalent native implementation, but it's easy to imagine a scenario where

cutting corners can yield a performance gain. Imagine a website consisting of nothing but tiles á la Windows 10 or a Masonry-style layout. Absolute/fixed positioning is not used, neither is **z-index** nor do elements ever overlap or have any kind of border or overflow. Being able to skip all these checks on re-layout could yield a performance gain.

```
registerLayout('random-layout', class {
    static get inputProperties() {
      return [];
    }
    static get childrenInputProperties() {
      return [];
    }
    layout(children, constraintSpace, styleMap) {
        Const width = constraintSpace.width;
        Const height =constraintSpace.height;
        for (let child of children) {
            const x = Math.random()*width;
            const y = Math.random()*height;
            const constraintSubSpace = new ConstraintSpace();
            constraintSubSpace.width = width-x;
            constraintSubSpace.height = height-y;
            const childFragment = child.doLayout(constraintSubSpace);
            childFragment.x = x;
            childFragment.y = y;
        }

        return {
            minContent: 0,
            maxContent: 0,
            width: width,
            height: height,
            fragments: [],
            unPositionedChildren: [],
            breakToken: null
        };
    }
});
```

Typed CSSOM

(spec)

Note: An “almost complete” implementation has landed in Chrome Canary behind the “Experimental Web Platform features” flag.

Typed CSSOM (CSS Object Model or Cascading Style Sheets Object Model) addresses a problem we probably all have encountered and just learned to just put up with. Let me illustrate with a line of JavaScript:

```
$('#someDiv').style.height = getRandomInt() + 'px';
```

We are doing math, converting a number to a string to append a unit just to have the browser parse that string and convert it back to a number for the CSS engine. This gets even uglier when you manipulate transforms with JavaScript. No more! CSS is about to get some typing!

This draft is one of the more mature ones and a polyfill is already being worked on. (Disclaimer: using the polyfill will obviously add *even more* computational overhead. The point is to show how convenient the API is.)

Instead of strings you will be working on an element's **`StylePropertyMap`**, where each CSS attribute has it's own key and corresponding value type. Attributes like **`width`** have **`LengthValue`** as their value type. A **`LengthValue`** is a dictionary of all CSS units like **`em`**, **`rem`**, **`px`**, **`percent`**, etc. Setting **`height: calc(5px + 5%)`** would yield a **`LengthValue{px: 5, percent: 5}`**. Some properties like **`box-sizing`** just accept certain keywords and therefore have a **`KeywordValue`** value type. The validity of those attributes could then be checked at runtime.

```
<div style="width: 200px;" id="div1"></div>
<div style="width: 300px;" id="div2"></div>
<div id="div3"></div>
<div style="margin-left: calc(5em + 50%);" id="div4"></div>
var w1 = $('#div1').styleMap.get('width');
var w2 = $('#div2').styleMap.get('width');
$('#div3').styleMap.set('background-size',
  [new SimpleLength(200, 'px'), w1.add(w2)])
$('#div4')).styleMap.get('margin-left')
  // => {em: 5, percent: 50}
```

Properties and values

(spec)

Note: Spec is pretty stable. No accesible implementation as of yet.

Do you know CSS Custom Properties (or their unofficial alias "CSS Variables")? This is them but with types! So far, variables could only have string values and used a simple search-and-

replace approach. This draft would allow you to not only specify a type for your variables, but also define a default value and influence the inheritance behavior using a JavaScript API. Technically, this would also allow custom properties to get animated with standard CSS transitions and animations, which is also being considered.

```
["--scale-x", "--scale-y"].forEach(function(name) {
document.registerProperty({
    name: name,
    syntax: "<number>",
    inherits: false,
    initialValue: "1"
  });
});
```

Font metrics

Font metrics is exactly what it sounds like. What is the bounding box (or the bounding boxes when we are wrapping) when I render string X with font Y at size Z? What if I go all crazy unicode on you like using ruby annotations? This has been requested a lot and Houdini should finally make these wishes come true.

But wait, there's more!

There's even more specs in Houdini's list of drafts, but the future of those is rather uncertain and they are not much more than placeholders for ideas. Examples include custom overflow behaviors, CSS syntax extension API, extension of native scroll behavior and similarly ambitious things all of which enable things on the web platform that weren't possible before.

Demos

I have open-sourced the code for the demo (live demo using polyfill) videos I made so you can get a feeling on what working with worklets feels like. I will update the repository with new demos as new APIs are landing in Canary.

If you want to get involved, there's always the Houdini mailing list.

---

registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.