CSS Houdini: Present and Future of CSS

> If we grow the language in these few ways [that allow extensibility], then we will not need to grow it in a hundred other ways; the users can take on the rest of the task.
>
> Guy Steele, Growing a Language, 1998 ACM OOPSLA

CSS Houdini is a set of APIs under development by the CSS Working Group and the Technical Architecture Group to provide users with way to extend CSS to match their needs. The extensibility language is usually JavaScript but some of these are written in pure CSS.

To translate the tech speak: Houdini will provide the hooks for developers to create their own content types and in a performant way without having to wait for the working group to create specs for and browsers to implement.

Tab Atkins from the Chrome team presented about Houdini at CSS Day 2017. He does a much better job at explaining the APIs (expected since he's one of the editors).

Surma, from the Chrome team, created is Houdini ready yet? as a way to track what Houdini is working on and the status of these different projects. I've grouped

the table into three categories for the list below:

- Supported in (some) browser(s)
    - Paint API
    - CSS Typed Object Model
- Some Support
    - Layout API
    - Properties & Values API
    - Animation Worklet
- Future Looking (no specs currently available)
    - Parser API
    - Font Metrics API

Houdini APIs rely on Worklets, a separate specification. This spec provides the underpinnings for the other Houdini specifications.

Before we get started, it's important to know that **this is not CSS in JS!** It's a way to enhance CSS with Javascript to produce results that are impossible to produce with CSS alone, as it exists today.

Worklets

The worklets are lightweight web workers that are better suited than the heavier workers for the kinds of work Houdini is intended for. You will see several different kinds of worklets that are specific for each API.

CSS Typed Object Model

If you work with CSS in JS you will soon discover that all values CSS gives the JS parser are strings that you must parse to get the data you need to continue processing the styles. This is when you use the `element.style.attribute` syntax.

```
// Element styles.
el.style.opacity = 0.3;
typeof el.style.opacity === 'string' 'string'// Ugh. A string!?

// Stylesheet rules.
```

```
document.styleSheets[0].cssRules[0].style.opacity = 0.3;
```

The CSS Typed Object Model (Typed OM) adds types, methods, and an object model to CSS values. Instead of strings, values are exposed as JavaScript objects to facilitate performant (and sane) manipulation of CSS.

Instead of using `element.style`, you'll be accessing styles through a new `.attributeStyleMap` property for elements and a `.styleMap` property for stylesheet rules. Both return a `StylePropertyMap` object.

The functionality of the Stylesheet version of the

```
// Element styles.
// Set element style
el.attributeStyleMap.set('opacity', 0.3);
'opacity'// Get the element style's valuebuteStyleMap.get('opacity').value
// list a'opacity'// list all style attributes for element
console.log(el.attributeStyleMap)
// What kind of element is the value for the element?teStyleMap.get('opac

// Stylesheet rule'opacity'// Stylesheet rules.
const stylesheet = document.styleSheets[0];
stylesheet.cssRules[0].styleMap.set('background', 'blue');
```

So, why should we care?

- Numerical values are always returned as numbers, not strings, so you do less work and get consistent results
- Value clamping & rounding. Typed OM rounds and/or clamps values so they're within the acceptable ranges for a property
- Better performance. The browser has to do less work serializing and deserializing string values
- We can use Javascript error handling for our CSS
- CSS property names in Typed OM are always strings, matching what you actually write in CSS

Basic Usage: Feature Detection

I'm always one to code defensively. To make sure that the browser supports CSS Typed OM we can use something like the code below to make sure that the browser will work with the code we want to use. If it doesn't then we can fall back to using the current object model (`el.style.attribute`).

```
if (window.CSS && CSS.number) {
  // Work with CSS Typed OM.
  el.attributeStyleMap.set('opacity', 0.3);
} else {
  el.style.opacity = '0.3';
}
'opacity'
```

Setting Single Value Attributes

The basic thing to do is to set and retrieve values. Using the `attributeStyleMap` to set up the values for attributes.

Typed OM provides convenient methods to make it easier to create typed values without having to use strings. Note that some of these values are part of specifications currently under development and may have little or no support among current browsers.

The values in the current Typed OM specification are:

- Length
    - CSS.em();
    - CSS.ex();
    - CSS.ch();
    - CSS.ic();
    - CSS.rem();
    - CSS.lh();
    - CSS.rlh();
    - CSS.vw();
    - CSS.vh();
    - CSS.vi();
    - CSS.vb();

        - CSS.vmin();
        - CSS.vmax();
        - CSS.cm();
        - CSS.mm();
        - CSS.Q();
        - CSS.in();
        - CSS.pt();
        - CSS.pc();
        - CSS.px();
    - angle
        - CSS.deg();
        - CSS.grad();
        - CSS.rad();
        - CSS.turn();
    - time
        - CSS.s();
        - CSS.ms();
    - frequency
        - CSS.Hz();
        - CSS.kHz();
    - resolution
        - CSS.dpi();
        - CSS.dpcm();
        - CSS.dppx();
    - flex
        - CSS.fr();

For more information about the units listed above check the CSS Values and Units Level 4 specification.

You can also use strings. The parser will convert it to numbers under the hood. The two declarations below are equivalent.

```
el.attributeStyleMap.set('margin-top', CSS.px(10));
el.attributeStyleMap.set('margin-top', '10px');
```

Complex Numeric Values

There are times when you need to represent more than a single value, for example when doing something like `calc(1em + 10px)`.

We can get more complex values by using mathematical functions that are part of the Typed OM. The Spec makes the following aritmethic functions available that will return either a typed value or a calc() function depending on the parameters passed.

- add();
- sub();
- mul();
- div();
- min();
- max();

```
// Multiply 2 values, keep the unit
CSS.deg(90).mul(2)
// {value: 180, unit: "deg"}

// Pick the bigger of the two values
CSS.percent(50).max(CSS.vw(50)).toString()
// "max(50%, 50vw)"

// Add to numbers of the same unit
CSS.px(1).add(CSS.px(2))
// {value: 3, unit: "px"}

// multiple values:
CSS.s(1).sub(CSS.ms(200), CSS.ms(300)).toString()
// "calc(1s + -200ms + -300ms)"

// or pass a `CSSMathSum`:
const sum = new CSSMathSum(CSS.percent(100), CSS.px(20));
CSS.vw(100).add(sum).toString() // "calc(100vw + (100% + 20px))"
```

CSS Math Values

The values created by `cssMathValue` objects contain one or more mathematical expressions. Notice how we use the `toString()` method to explicitly convert the values to strings that we can feed back to CSS for it to work consistently.

Also note that unless we usually get a `calc()` function that we can drop in to

the stylesheets to continue manipulating or leave in the stylesheet as the final value.

```
new CSSMathSum(CSS.vw(100), CSS.px(-10)).toString();
// "calc(100vw + -10px)"

new CSSMathNegate(CSS.px(42)).toString()
// "calc(-42px)"

new CSSMathInvert(CSS.s(10)).toString()
// "calc(1 / 10s)"

new CSSMathProduct(CSS.deg(90), CSS.number(Math.PI/180)).toString();
// "calc(90deg * 0.0174533)"

new CSSMathMin(CSS.percent(80), CSS.px(12)).toString();
// "min(80%, 12px)"

new CSSMathMax(CSS.percent(80), CSS.px(12)).toString();
// "max(80%, 12px)"
```

Keyword Values

There are values that are not represented by a number + unit combination, like the values for display or `inherited`. Use `CSSKeyWordValue` in those cases.

```
el.attributeStyleMap.set('display', new CSSKeywordValue('initial'));
el.attributeStyleMap.'initial'lay').value // 'initial'').value // 'StyleMa
'display'// undefined// undefined
```

Retrieving values: attributeStyleMap versus computedStyleMap

We can then retrieve the value (without a unit) or the unit itself.

```
el.attributeStyleMap.get('margin-top').value  // 10
```

```
el.attributeStyleMap.get('margin-top').unit 'margin-top'// 'px'
```

There are situations where the element may have more than a single value like `background-image`. In this situation, `get` will only give us the first value for the attribute. Fortunately we have a `getAll` method that would return all the values in a map.

```
el.attributeStyleMap.getAll('background-image')
```

Computed Styles versus Attribute Styles

`attributeStyleMap` gives you the value you entered but this presents one interesting case for me. What happens when we give it a percentage or other relative values like em or rem?

We can use different units defined outside the Typed OM like `window.getComputedStyle()` to get the resulting value of a property after all parent object values are calculated.

The difference between `window.getComputedStyle()` (outside the Typed OM) and `element.computedStyleMap()` (part of the Typed OM) is that the `window.getComputedStyle()` method gets the value after all calculations have been done. If we need to get the value as we're working with it then we can use the `element.computedStyleMap()` method.

CSS Transforms

CSS Transforms are created with aCSSTransformValue objects passing an array of transform values (e.g. CSSRotate, CSScale, CSSSkew, CSSSkewX, CSSSkewY). As an example, say you want to re-create this CSS:

```
transform: rotateZ(45deg) scale(0.5) translate3d(10px,10px,10px);
```

Translated into Typed OM:

```
const transform =  new CSSTransformValue([
```

```
  new CSSRotate(CSS.deg(45)),
  new CSSScale(CSS.number(0.5), CSS.number(0.5)),
  new CSSTranslate(CSS.px(10), CSS.px(10), CSS.px(10))
]);
```

In addition to its verbosity (lolz!), CSSTransformValue has some cool features. It has a boolean property to differentiate 2D and 3D transforms and a .toMatrix() method to return the DOMMatrix representation of a transform:

```
new CSSTranslate(CSS.px(10), CSS.px(10)).is2D // true
new CSSTranslate(CSS.px(10), CSS.px(10), CSS.px(10)).is2D // false
new CSSTranslate(CSS.px(10), CSS.px(10)).toMatrix() // DOMMatrix
```

Custom Properties

CSS `var()` become a CSSVariableReferenceValue object in the Typed OM. Their values get parsed into CSSUnparsedValue because they can take any type (px, %, em, rgba(), etc).

```
const foo = new CSSVariableReferenceValue('--foo');
// foo.variable === '--foo'

// Fallback values:
const padding = new CSSVariableReferenceValue(
    '--default-padding', new CSSUnparsedValue(['8px']));
'--default-padding'
```

If you want to get the value of a custom property, there's a bit of work to do:

```
<style>
  body {
    --foo: 10px;
  }
</style>
<script>
```

```
  const styles = doc</style>erySelect
  const styles = document.querySelector('style');
  const foo = styles.sheet.cssRules[0].styleMap.get('--foo').trim();
  console.log(CSSNumericValue.p'--foo'o).value); // 10
```

CSS Positions

CSSPositionValue objects represent a space-separated list of values, like those used by object-position.

```
const position = new CSSPositionValue(CSS.px(5), CSS.px(10));
el.attributeStyleMap.set('object-position', position);

console.log(position.x.value, position.y.value);
// → 5, 10
```

CSS Images

CSSImageValue objects represent one or more values for images, like those used in background-image, list-style-image, and border-image-source. As we discussed earlier these elements may return one or more images as the value so we need to retrieve them with `element.attributeStyleMap.getAll()` to make sure we capture all the images used in the element.

Parsing values

The Typed OM introduces parsing methods to the web platform! This means you can finally parse CSS values programmatically, before trying to use it! This new capability is a potential life saver for catching early bugs and malformed CSS.

Parse a full style:

```
const css = CSSStyleValue.parse(
    'transform', 'translate3d(10px,10px,0) scale(0.5)');
'translate3d(10px,10px,0) scale(0.5)'// → css instanceof CSSTransformValue
// → css.toString() === 'translate3d(10px, 10px, 0) scale(0.5)'
```

Parse values into CSSUnitValue:

```
CSSNumericValue.parse('42.0px') // {value: 42, unit: 'px'}

// But it's easier to use the factory functions:
CSS.px(42.0) // '42px'
```

Error handling

Because we're working with Javascript to

```
try {
  const css = CSSStyleValue.parse('transform', 'translate4d(bogus value)'
  'translate4d(bogus value)'// use css
} catch (err) {
  console.err(err);
}
```

Links and Resources

- CSS Paint API
- Working with the new CSS Typed Object Model
- CSS Houdini
- CSS Houdini Experiments