Pushing Houdini Forward

I've written about Houdini before and how awesome it is. The articles I've written are:

- CSS Houdini: Present and Future of CSS
- CSS Houdini: Properties & Values
- CSS Houdini: CSS Typed Object Model

But, because it's not widely deployed and not all APIs have equal level of support in the browsers where they work, it's hard to get something that works well without having to rely on writing two versions of the code.

It wasn't until I saw Design System Magic with CSS Houdini that I realized that you can combine the different APIs and make fully working designs with them. It also prompted me to start looking at coombination of the different APIs and how to provide API fallbacks for browsers that have not implemented them.

In CSS Houdini & The Future of Styling, Una Kravets makes an interesting case for Houdini Custom Properties and Houdini APIs to style the web now and how much powerful these APIs can make your styles and design systems.

Examples

Most of the Houdini APIs will take CSS elements as input. We can leverage Houdini Custom Properties. An example, taken from the specification.

The body of the page contains the following content. In thehead of the document we add the styles:

```
<style>
  #example {
    --circle-color: deepskyblue;

    background-image: paint(circle);
    font-family: sans-serif;
    font-size: 36px;
    transition: --circle-color 1s;
  }

  #example:focus {
    --circle-color: purple;
  }
</style>
```

In the body of the document we add the element we add the textarea element we'll be working in and a script that will add the custom property, using `CSS.registerProperty` and load our paint worklet.

We feature test that the methods are available before we run them. If they are not available we log the fact to the console; in a production application we may want to add the custom property via CSS and load a polyfill for the Paint API.

```
<textarea id="example">CSS is awesome.</textarea>

<script></textarea>
if ('registerProperty' in CSS) {
  CSS.registerProperty({
    name: '--circle-color',
    syntax: '<color>',
    initialValue: 'deepskyblue',
    inherits: false,
  });
  console.log('property successfuly registered');
} else {
  console.log('Houdini custom properties not supported');
}

if ('paintWorklet' in CSS) {
  CSS.paintWorklet.addModule('circle.js');
  console.log('paint worklet added successfully');
} else {
  console.log('Paint API not supported or not working properly');
}
</script>
```

The paint worklet for this example registers input properties that we'll take from the page's existing properties and custom properties. The browser doesn't care how we created the custom property, only that it exists.

The syntax of the Paint Worklet is a subset of the Canvas API. Text rendering methods are missing and for security reasons you cannot read back pixels from the canvas.

```
registerPaint('circle', class {
  static get inputProperties() {
    return ['--circle-color'];
  }'--circle-color'ize, properties) {
    // Get fill color from property
    const color = properties.get('--circle-color');

    '--circle-color'// Determine the center point and radius.le = size.wi
    const yCircle = size.height / 2;
    const radiusCircle = Math.min(xCircle, yCircle) - 2.5;

    // Draw the circle \o/
    ctx.beginPa/ Draw the circle \o/
    ctx.beginPath();
    ctx.arc(xCircle, yCircle, radiusCircle, 0, 2 * Math.PI);
    ctx.fillStyle = color;
    ctx.fill();
  }
});
```

Other worklets you may find in the whild will have inputArguments instead. I'm researching how to use input arguments... the examples I've found don't work in Chrome (stable or cannary).

Polifilling

Houdini is awesome when it works, but what do we do when it doesn't?

Different areas of the Houdini universe have different ways to polyfill the APIs and not all APIs have been implemented to where having a polyfill.

There is a CSS Paint Polyfill from Jason Miller.

PostCSS Register Custom Property works by converting CSS-based custom element syntax (basically writing Houdini properties in CSS) using the syntax below:

```css
@property --theme {
  syntax: '<color>+';
  initial-value: #fff;
  inherits: true;
}
```

and converting it to Javascript

```javascript
if ("registerProperty" in CSS) {
  CSS.registerProperty({
    name: "--theme",
    syntax: "<color>+",
    initialValue: "#fff",
    inherits: true
  });
}
```

Packaging Ideas together

Another way to support Houdini APIs is to package them for consumption like Una Kravets did with Extra.css.

Rather than provide a do-it-yourself framework where you're responsible for all the details, it provides ready to use examples that you just link to your page.

The following example, taken from: https://extra-css.netlify.com/ illustrates the process.

In the HTML document we load the paint Worklet as a Javascript file.

```
<h1>Hello<br/<br/>ld</h1>
<p>content g</h1>ere</p>

</p><!-- This is where we include the worklet -->ript
  src='https://unpkg.com/extra.css/crossOut.js'></script>
</script>
```

The CSS portion is where the magic happens. The CSS Paint API allows you to define custom paints, defined in the paint worklet, that we can use everywhere you ocan use an image.

We wrap our CSS in a @supports statement to make sure that the browser supports the feature we're working with before we actually use it. We can also leverage the cascade to make sure we have something that works, either CSS variables, Houdini variables and APIs or soomething else.

```
@supports (background: paint(something)) {
  h1 {
    --extra-crossColor: #fc0;
    --extra-crossWidth: 3;

    background: paint(extra-crossOut);
    line-height: 1.5;
  }

  span {
    --extra-crossColor: #d4f;
    background: paint(extra-crossOut);
  }
}
```

More

- Drafts of Houdini Specifications
- Is Houdini ready yet? tracks status of Houdini APIs support across browsers
- State of Houdini (Chrome Dev Summit 2018) Video
- CSS Houdini Experiments

    - SVG Path layout
- CSS Houdini
- The CSS Paint API
- Say Hello to Houdini and the CSS Paint API