CSS Paint API

By Surma

Surma is a contributor to Web**Fundamentals**

New possibilities in Chrome 65

CSS Paint API (also known as “CSS Custom Paint” or “Houdini’s paint worklet”) is about to be enabled by default in Chrome Stable. What is it? What can you do with it? And how does it work? Well, read on, will ya’…

CSS Paint API allows you to programmatically generate an image whenever a CSS property expects an image. Properties like `background-image` or `border-image` are usually used with `url()` to load an image file or with CSS built-in functions like `linear-gradient()`. Instead of using those, you can now use `paint(myPainter)` to reference a *paint worklet*.

Writing a paint worklet

To define a paint worklet called `myPainter`, we need to load a CSS paint worklet file using `CSS.paintWorklet.addModule('my-paint-worklet.js')`. In that file we can use the `registerPaint` function to register a paint worklet class:

```
class MyPainter {
  paint(ctx, geometry, properties) {
    // ...
  }
}

registerPaint('myPainter', MyPainter);
```

Inside the `paint()` callback, we can use `ctx` the same way we would a `CanvasRenderingContext2D` as we know it from `<canvas>`. If you know how to draw in a `<canvas>`, you can draw in a paint worklet! `geometry` tells us the width and the height of the canvas that is at our disposal. `properties` I will explain later in this article.

Note: A paint worklet’s context is not 100% the same as a `<canvas>` context. As of now, text rendering methods are missing and for security reasons you cannot read back pixels from the canvas.

As an introductory example, let's write a checkerboard paint worklet and use it as a background image of a **<textarea>**. (I am using a textarea because it's resizable by default.):

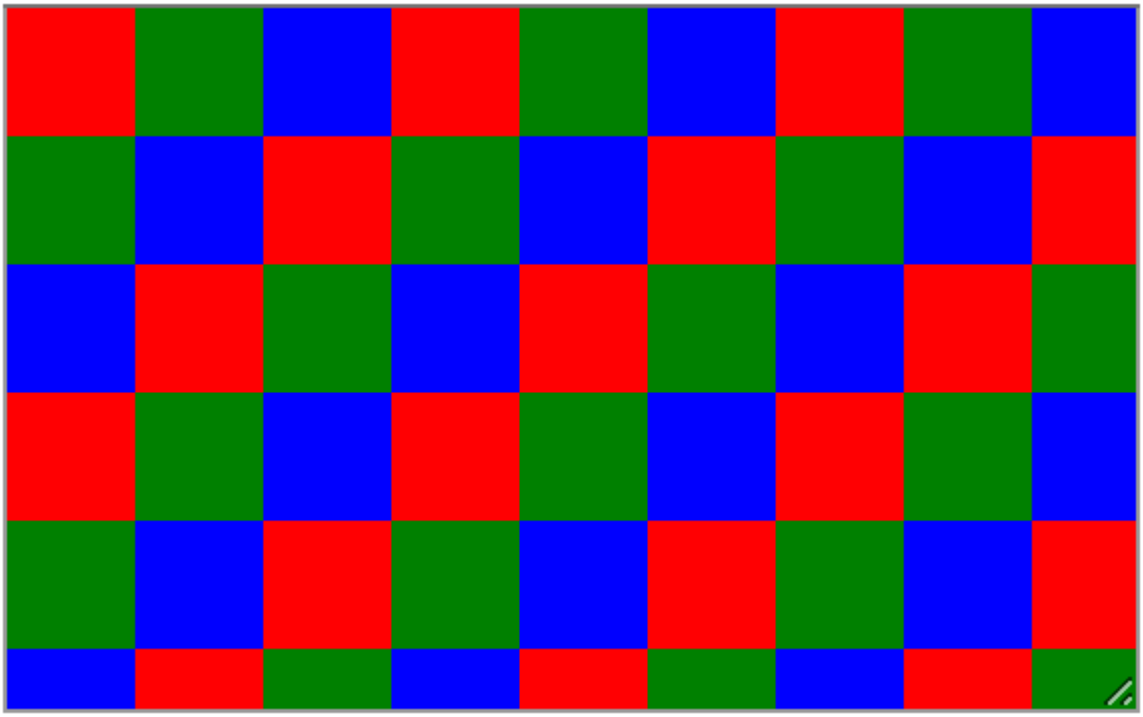
```
<!-- index.html -->
<!doctype html>
<style>
  textarea {
    background-image: paint(checkerboard);
  }
</style>
<textarea></textarea>
<script>
  CSS.paintWorklet.addModule('checkerboard.js');
</script>
```

```
// checkerboard.js
class CheckerboardPainter {
  paint(ctx, geom, properties) {
    // Use `ctx` as if it was a normal canvas
    const colors = ['red', 'green', 'blue'];
    const size = 32;
    for(let y = 0; y < geom.height/size; y++) {
      for(let x = 0; x < geom.width/size; x++) {
        const color = colors[(x + y) % colors.length];
        ctx.beginPath();
        ctx.fillStyle = color;
        ctx.rect(x * size, y * size, size, size);
        ctx.fill();
      }
    }
  }
}

// Register our class under a specific name
registerPaint('checkerboard', CheckerboardPainter);
```

If you've used **<canvas>** in the past, this code should look familiar. See the live demo here.

Note: As with almost all new APIs, CSS Paint API is only available over HTTPS (or **`localhost`**).



The difference from using a common background image here is that the pattern will be re-drawn on demand, whenever the user resizes the textarea. This means the background image is always exactly as big as it needs to be, including the compensation for high-density displays.

That's pretty cool, but it's also quite static. Would we want to write a new worklet every time we wanted the same pattern but with differently sized squares? The answer is no!

Parameterizing your worklet

Luckily, the paint worklet can access other CSS properties, which is where the additional parameter **`properties`** comes into play. By giving the class a static **`inputProperties`** attribute, you can subscribe to changes to any CSS property, including custom properties. The values will be given to you through the **`properties`** parameter.

```
<!-- index.html -->
<!doctype html>
<style>
  textarea {
    /* The paint worklet subscribes to changes of these custom properties. */
    --checkerboard-spacing: 10;
    --checkerboard-size: 32;
    background-image: paint(checkerboard);
  }
</style>
```

```
<textarea></textarea>
<script>
  CSS.paintWorklet.addModule('checkerboard.js');
</script>
```

```
// checkerboard.js
class CheckerboardPainter {
  // inputProperties returns a list of CSS properties that this paint function ge
  static get inputProperties() { return ['--checkerboard-spacing', '--checkerboar

  paint(ctx, geom, properties) {
    // Paint worklet uses CSS Typed OM to model the input values.
    // As of now, they are mostly wrappers around strings,
    // but will be augmented to hold more accessible data over time.
    const size = parseInt(properties.get('--checkerboard-size').toString());
    const spacing = parseInt(properties.get('--checkerboard-spacing').toString())
    const colors = ['red', 'green', 'blue'];
    for(let y = 0; y < geom.height/size; y++) {
      for(let x = 0; x < geom.width/size; x++) {
        ctx.fillStyle = colors[(x + y) % colors.length];
        ctx.beginPath();
        ctx.rect(x*(size + spacing), y*(size + spacing), size, size);
        ctx.fill();
      }
    }
  }
}

registerPaint('checkerboard', CheckerboardPainter);
```

Now we can use the same code for all different kind of checkerboards. But even better, we can now go into DevTools and fiddle with the values until we find the right look.

Note: It would be great to parameterize the colors, too, wouldn't it? The spec allows for the `paint()` function to take a list of arguments. This feature is not implemented in Chrome yet, as it heavily relies on Houdini's Properties and Values API, which still needs some work before it can ship.

Browsers that don't support paint worklet

At the time of writing, only Chrome has paint worklet implemented. While there are positive signals from all other browser vendors, there isn't much progress. To keep up to date, check Is Houdini Ready Yet? regularly. In the meantime, be sure to use progressive enhancement to keep your code running even if there's no support for paint worklet. To make sure things work as expected, you have to adjust your code in two places: The CSS and the JS.

Detecting support for paint worklet in JS can be done by checking the `CSS` object:

```
if ('paintWorklet' in CSS) {
  CSS.paintWorklet.addModule('mystuff.js');
}
```

For the CSS side, you have two options. You can use `@supports`:

```
@supports (background: paint(id)) {
  /* ... */
}
```

A more compact trick is to use the fact that CSS invalidates and subsequently ignores an entire property declaration if there is an unknown function in it. If you specify a property twice — first without paint worklet, and then with the paint worklet — you get progressive enhancement:

```
textarea {
  background-image: linear-gradient(0, red, blue);
  background-image: paint(myGradient, red, blue);
}
```

In browsers *with* support for paint worklet, the second declaration of `background-image` will overwrite the first one. In browsers *without* support for paint worklet, the second declaration is invalid and will be discarded, leaving the first declaration in effect.

CSS Paint Polyfill

For many uses, it's also possible to use the CSS Paint Polyfill, which adds CSS Custom Paint and Paint Worklets support to modern browsers.

Use cases

There are many use cases for paint worklets, some of them more obvious than others. One of the more obvious ones is using paint worklet to reduce the size of your DOM. Oftentimes, elements are added purely to create embellishments using CSS. For example, in Material Design Lite the button with the ripple effect contains 2 additional `<span>` elements to implement the ripple itself. If you have a lot of buttons, this can add up to quite a number of DOM elements and can lead to degraded performance on mobile. If you implement the ripple effect using paint worklet instead, you end up with 0 additional elements and just one paint worklet. Additionally, you have with something that is much easier to customize and parameterize.

Another upside of using paint worklet is that — in most scenarios — a solution using paint worklet is small in terms of bytes. Of course, there is a trade-off: your paint code will run whenever the canvas's size or any of the parameters change. So if your code is complex and takes long it might introduce jank. Chrome is working on moving paint worklets off the main thread so that even long-running paint worklets don't affect the responsiveness of the main thread.

To me, the most exciting prospect is that paint worklet allows to efficient polyfilling of CSS features that a browser doesn't have yet. One example would be polyfill conic gradients until they land in Chrome natively. Another example: in a CSS meeting it was decided that you can now have multiple border colors. While this meeting was still going on, my colleague Ian Kilpatrick wrote a polyfill for this new CSS behavior using paint worklet.

Thinking outside the "box"

Most people start to think about background images and border images when they learn about paint worklet. One less intuitive use case for paint worklet is `mask-image` to make DOM elements have arbitrary shapes. For example a diamond:

`mask-image` takes an image that is the size of the element. Areas where the mask image is transparent, the element is transparent. Areas where the mask image is opaque, the element opaque.

Now in Chrome

Paint worklet has been in Chrome Canary for a while. With Chrome 65, it is enabled by default. Go ahead and try out the new possibilities that paint worklet opens up and show us what you built! For more inspiration, take a look at Vincent De Oliveira's collection.

Note: Breakpoints are currently not supported in CSS Paint API, but will be enabled in a later release of Chrome.

Last updated July 2, 2018.