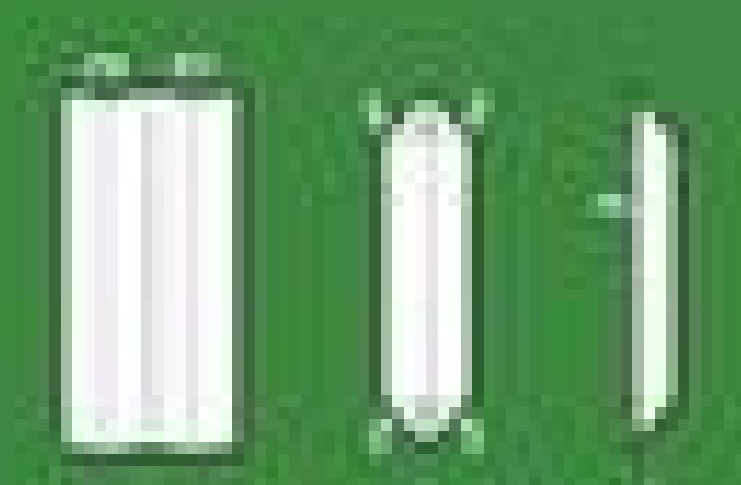


MODERN CSS



CUTTING-EDGE TECHNIQUES

Modern CSS

Copyright © 2018 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-18-9

Project editor: Craig Buckler

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: www.sitepoint.com

Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

CSS has grown from a language for formatting documents into a robust language for designing web applications. Its syntax is easy to learn, making CSS a great entry point for those new to programming. Indeed, it's often the second language that developers learn, right behind HTML.

As CSS's feature set and abilities have grown, so has its depth. In this book, we'll be exploring some of the amazing things that developers can do with CSS today; things that in the past might only have been achievable with some pretty complex JavaScript previously, if at all.

Conventions Used

CODE SAMPLES

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back
at school.</p>
```

Where existing code is required for context, rather than repeat

all of it, `:` will be displayed:

```
function animate() {  
  :  
  new_variable = "Hello";  
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➞ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-  
➞design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

TIPS, NOTES, AND WARNINGS

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

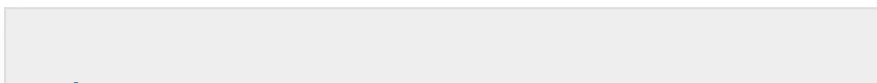
Chapter 1: Using CSS Transforms in the Real World

BY ILYA BODROV-KRUKOWSKI

In this article, we'll learn how CSS transforms can be used in the real world to solve various tasks and achieve interesting results. Specifically, you'll learn how to adjust elements vertically, create nice-looking arrows, build loading animations and create flip animations.

Transformations of HTML elements became a CSS3 standard in 2012 and were available in some browsers before then. Transformations allow you to transform elements on a web page — as explained in our [101 article on CSS transforms](#). You can rotate, scale, or skew your elements easily with just one line of code, which was difficult before. Both 2D and 3D transformations are available.

As for browser support, 2D transforms [are supported by all major browsers](#) — including Internet Explorer, which has had support since version 9. As for 3D transformations, IE [has only partial support](#) since version 10.



Anem, Excuse Me ...

This article won't focus on the basics of transformations. If you don't feel very confident with transformations, I recommend reading about [2D](#) and [3D transforms](#) first.

Vertically Aligning Children

Any web designer knows how tedious it can be to vertically align elements. This task may sound very simple to a person who's not familiar with CSS, but in reality there's a jumble of techniques that are carefully preserved between generations of developers. Some suggest using `display: inline` with `vertical-align: middle`, some vote for `display: table` and accompanying styles, whereas true old school coders are still designing their sites with tables (just joking, don't do that!). Also, it's possible to solve this task with Flexbox or Grids, but for smaller components, transforms may be a simpler option.

Vertical alignment can be more complex when element heights are variable. However, CSS transformations provide one way to solve the problem. Let's see a very simple example with two nested `div`s:

```
<div class="parent">
  <div class="child">
    Lorem ipsum dolor sit amet, consectetur
    adipiscing elit, sed do eiusmod tempor
    ↪incidunt ut labore et dolore magna aliqua.
    Ut enim ad minim veniam, quis
    ↪nostrud exercitation ullamco laboris nisi ut
    aliquip ex ea commodo consequat.
    ↪Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore
```



```
</div>
</div>

<div class="parent">
  <div class="child">
    Lorem ipsum dolor sit amet, consectetur
    adipiscing elit, sed do eiusmod tempor
    incididunt ut labore et dolore magna aliqua.
    Ut enim ad minim veniam
  </div>
</div>
```

Nothing complex: just two nested blocks with a Lorem Ipsum text of different length.

Let's set width, height, and border for the parent, as well as some spacing to make it look nicer:

```
.parent {
  height: 300px;
  width: 600px;
  padding: 0 1em;
  margin: 1em;
  border: 1px solid red;
}
```

Also, enlarge the children font size a bit:

```
.child {
  font-size: 1.2rem;
}
```

As a result, you'll see something like this:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam

And now your customer says: “Please align the text vertically so that it appears in the middle of these red boxes”. Nooo! But fear not: we’re armed with transformations! The first step is to position our children relatively and move them 50% to the bottom:

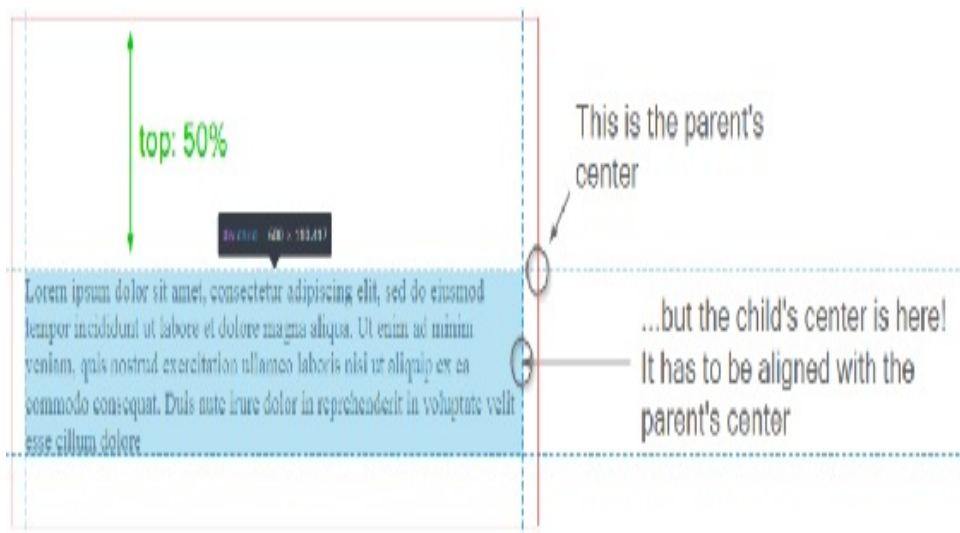
```
.child {  
  font-size: 1.2rem;
```

```
position: relative;
top: 50%;
}
```

After that, apply a secret ingredient — the `translateY` function — which will reposition the elements vertically:

```
.child {
  font-size: 1.2rem;
  position: relative;
  top: 50%;
  transform: translateY(-50%);
}
```

So, what's happening here? `top: 50%` moves the top of the child element to the center of the parent:



But this is not enough, because we want the child's center to be aligned with the parent's center. Therefore, by applying the `translateY` we move the child up 50% of its height:



Some developers have reported that this approach can cause the children to become blurry due to the element being placed on a “half pixel”. A solution for this is to set the perspective of the element:

```
.child {  
  // ...  
  transform: perspective(1px) translateY(-50%);  
}
```

This is it! Your children are now aligned properly even with variable text, which is really nice. Of course, this solution is a little hacky, but it works in older browsers in contrast to CSS Grid. The final result can be seen on CodePen:

Live Code

The final result can be seen on CodePen: see the Pen [CSS Transforms: Vertical-Align](#).

Creating Arrows

Another very interesting use case for transformations is

creating speech bubble arrows that scale properly. I mean, you can definitely create arrows with a graphical editor, but that's a bit tedious, and also they may not scale well if you use a bitmap image.

Instead, let's try to stick with a pure CSS solution. Suppose we have a box of text:

```
<div class="box">
  <div class="box-content">
    Lorem ipsum dolor sit amet, consectetur
    adipiscing elit, sed do eiusmod tempor
    incididunt ut labore et dolore magna aliqua.
    Ut enim ad minim veniam
  </div>
</div>
```

It has some generic styling to make it look like a speech bubble:

```
html {
  font-size: 16px;
}

.box {
  width: 10rem;
  background-color: #e0e0e0;
  padding: 1rem;
  margin: 1rem;
  border-radius: 1rem;
}
```

Lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed do
eiusmod tempor
incidunt ut labore et
dolore magna aliqua. Ut
enim ad minim veniam

I'm using `rem` units here so that if the root's font size is changed, the box is scaled accordingly.

Next, I'd like to make this bubble become more "bubble-ish" by displaying an arrow to the right. We'll achieve that by using a `::before` pseudo-selector:

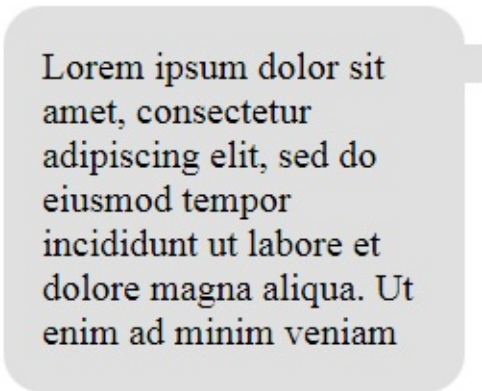
```
.box::before {  
  content: '';  
  width: 1rem;  
  height: 1rem;  
  background-color: #e0e0e0;  
  overflow: hidden;  
}
```

The larger the width and height you assign, the larger the arrow will be.

Now move the arrow to the right:

```
.box {  
  // ...  
  position: relative;  
}  
  
.box::before {  
  // ...  
  position: absolute;  
  right: -0.5rem;  
}
```

Currently, however, this element doesn't look like an arrow at all:



Lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed do
eiusmod tempor
incididunt ut labore et
dolore magna aliqua. Ut
enim ad minim veniam

Let's align this small box vertically. As long as its dimensions are static, this is a simple task:

```
.box::before {  
  // ...  
  top: 50%;  
}
```

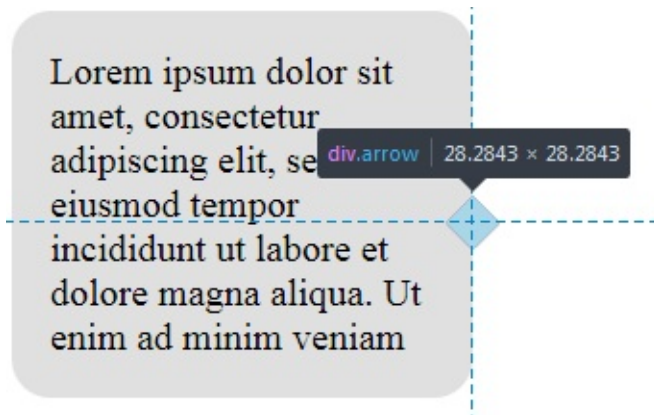
It is not precisely aligned, but we'll fix that later.

Now it's time for transformation to step in. All we need to do is rotate this small box to turn it to a triangle, which will look just like an arrow:

```
.box::before {  
  // ...  
  transform: rotate(45deg);  
}
```

Here's a screenshot from the developer console (the box is highlighted with blue so you can see how it's actually

positioned):



Then, just move the arrow a bit to the top using negative margin so that it's positioned precisely at the middle of the box:

```
.box::before {  
  // ...  
  margin-top: -0.5rem;  
}
```

That's it! Now, even if you need to adjust font size on the page, your box and the arrow are going to be scaled properly! Here's the final result:

Live Code

Here's the final result: See the Pen [CSS Transformations: Arrows](#)

Creating a “Jumping Ball” Loader

Unfortunately, things on the web don't happen instantly. Users often have to wait for an action to complete, be it sending an email, posting their comment, or uploading photos. Therefore, it's a good idea to display a "loading" indicator so that users understand they'll have to wait for a few seconds.

Previously, when there were no CSS animations and transformations, we would probably use a graphical editor to create an animated GIF loader image. This is no longer necessary because CSS3 offers powerful options. So, let's see how transformations can be utilized in conjunction with animations to create a nice-looking jumping ball.

To start, create an empty `div`:

```
<div class="loader"></div>
```

Balls are usually round (Captain Obvious to the rescue!), so let's give it width, height, and border radius:

```
.loader {  
  border-radius: 50%;  
  width: 50px;  
  height: 50px;  
}
```

Let's also use a [CSS background gradient editor](#) to generate some gradient:

```
.loader {  
  border-radius: 50%;  
  width: 50px;  
  height: 50px;  
  background: linear-gradient(to bottom, #cb60b3
```

```
0%, #c146a1 50%, #a80077 51%, #db36a4  
  - 100%);  
}
```

Here's our shiny purple ball that resembles pokeball:



How do we make it jump? Well, with a help of CSS animations! Specifically, I'm going to define an infinite animation called `jump` that takes 1.5 seconds to complete and performs the listed actions back and forth:

```
.loader {  
  // ...  
  animation: jump 1.5s ease-out infinite  
  alternate;  
}
```

Next, let's ask ourselves: what does it mean when we say "to jump"? In the simplest case, it means moving up and down on the Y axis (vertically). So, let's use the `translateY` function again and define keyframes:

```
@keyframes jump {  
  from {  
    transform: translateY(0px)  
  }  
  to {  
    transform: translateY(-50px)  
  }  
}
```

So, initially the ball is at coordinates $(0, 0)$, but then we move it up to $(0, -50)$. However, currently we might not

have enough space for the ball to actually jump, so let's give it some margin:

```
.loader {  
  margin-top: 50px;  
}
```

Of course, we can do more. For instance, let's rotate this ball while it jumps:

```
@keyframes jump {  
  from {  
    transform: translateY(0px) rotate(0deg)  
  }  
  to {  
    transform: translateY(-50px) rotate(360deg)  
  }  
}
```

Also, why don't we make it smaller? For that, let's utilize a **scale** function that changes the element's width and height using the given multipliers:

```
@keyframes jump {  
  from {  
    transform: translateY(0px) rotate(0deg)  
    scale(1,1);  
  }  
  to {  
    transform: translateY(-50px) rotate(360deg)  
    scale(0.8,0.8);  
  }  
}
```

Note, by the way, that all functions should be listed for the **transform** property in both **from** and **to** sections, because otherwise the animation won't work properly!

Lastly, let's add some opacity for the ball:

```
@keyframes jump {  
  from {  
    transform: translateY(0px) rotate(0deg)  
    scale(1,1);  
    opacity: 1;  
  }  
  to {  
    transform: translateY(-50px) rotate(360deg)  
    scale(0.8,0.8);  
    opacity: 0.8;  
  }  
}
```

Live Code

That's it! Our loader element is ready, and here's the final result: See the Pen [CSS Transformations: Loader Ball](#).

Creating a “Spinner” Loader with SVG

We've already seen how to create a simple “jumping ball” loader with just a few lines of code. For a more complex effect, however, you can utilize SVGs which are defined with a set of special tags.

More on SVG

This article isn't aimed at explaining how SVG images work and how to create them, but SitePoint [has a couple of articles](#) on the topic.

As an example, let's create a spinner loader. Here's the

corresponding SVG:

```
<svg class="spinner" viewBox="0 0 66 66"
xmlns="http://www.w3.org/2000/svg">
  <!-- 1 -->

  <circle class="path spinner-border" cx="33"
cy="33" r="31" stroke="url
  (#gradient)"><!-- 2 -->

  <linearGradient id="gradient"> <!-- 3 -->
    <stop offset="50%" stop-color="#000" stop-
opacity="1"></stop>
    <stop offset="65%" stop-color="#000" stop-
opacity=".5"></stop>
    <stop offset="100%" stop-color="#000" stop-
opacity="0"></stop>
  </linearGradient>

  <circle class="path spinner-dot" cx="37" cy="3"
r="2"></circle> <!-- 4 -->

</svg>
```

Main things to note here:

1. Our canvas has a viewport of 66x66.
2. This defines the actual circle that's going to spin. Its center is located at (33, 33) and the radius is 31px. We'll also have a stroke of 2px, which means $31^2 + 2^2 = 66$. `stroke="url(#gradient)"` means that the color of the stroke is defined by an element with an ID of `#gradient`.
3. That's our gradient for the spinner's stroke. It has three breakpoints that set different opacity values, which is going to result in a pretty cool effect.
4. That's a dot that's going to be displayed on the spinner's stroke. It will look like a small "head".

Now let's define some styling for the canvas and scale it up to 180x180:

```
.spinner {
  margin: 10px;
```

```
width: 180px;  
height: 180px;  
}
```

Now the `.spinner-border`, `.spinner-dot`, and some common styles for them:

```
.spinner-dot {  
  stroke: #000;  
  stroke-width: 1;  
  fill: #000;  
}  
  
.spinner-border {  
  fill: transparent;  
  stroke-width: 2;  
  width: 100%;  
  height: 100%;  
}  
  
.path {  
  stroke-dasharray: 170;  
  stroke-dashoffset: 20;  
}
```

Here's how our spinner looks at this stage. For now it doesn't spin, of course:



Now let's make it spin, which effectively means rotating it by 360 degrees:

```

.spinner {
  // ...
  animation: rotate 2s linear infinite;
}

@keyframes rotate {
  to {
    transform: rotate(360deg);
  }
}

```

This is an infinite animation that takes 2 seconds to complete.

Also, we can achieve an interesting effect of a “snake trying to bite its tail” with a skew function. Remember that I’ve called that small dot a “head”? Why don’t we pretend that it is a snake’s head then? In order to make it look more realistic, we’ll skew it on the X axis:

```

.spinner-dot {
  // ...
  animation: skew 2s linear infinite alternate;
}

@keyframes skew {
  from {
    transform: skewX(10deg)
  }
  to {
    transform: skewX(40deg)
  }
}

```

Now it seems like the snake is really trying to drag to its tail:



Live Code

Here's the final result of our spinner: See the [Pen CSS Transformations: Loaders with SVGs by SitePoint](#).

Creating a Flip Animation

The last example is a photo with a flip animation. When you hover over a photo, it flips and its description is shown. It can be useful for Instagram-like websites.

So, first of all, let's create a layout:

```
<section class="container">

  <figure class="photo">
    

    <figcaption class="back side">
      Lorem ipsum dolor sit amet, consectetur
      adipiscing elit, sed do eiusmod tempor
      ➔ incididunt ut labore et dolore magna
      aliqua.
    </figcaption>
  </figure>
```



```

<figure class="photo">
  

  <figcaption class="back side">
    Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
    ↳ incididunt ut labore et dolore magna
aliqua.
  </figcaption>
</figure>

<figure class="photo">
  

  <figcaption class="back side">
    Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
    ↳ incididunt ut labore et dolore magna
aliqua.
  </figcaption>
</figure>

</section>

```

Here we have a container with three photos. These photos are going to actually have two sides: front and back, just like a coin has heads and tails. The front contains the actual image, whereas the back (which is not visible initially) contains the description.

Style the container by giving it some margin:

```

.container {
  margin: 10px auto;
}

```

Each photo adjusts according to the viewport's width and floats to the left:

```
.photo {  
  width: 22vw;  
  height: 20vw;  
  min-width: 130px;  
  min-height: 100px;  
  float: left;  
  margin: 0 20px;  
}
```

The images themselves should maintain the aspect ratio and try to fill the parent:

```
.photo img {  
  object-fit: cover;  
}
```

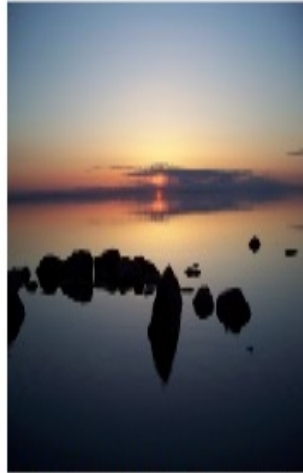
Each side should occupy 100% of the parent's width and height:

```
.side {  
  width: 100%;  
  height: 100%;  
}
```

Now we have images with descriptions below which looks pretty generic:



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Next, what I'd like to do is place the description right above the image (on the Z axis). For that, let's adjust some **position** rules:

```
.photo {  
  // ...  
  position: relative;  
}  
  
.side {  
  // ...  
  position: absolute;  
}
```

The text is now displayed right over the image:



Now it's time for some magic. I'd like children of the `.photo` block to be positioned in 3D with the help of `transform-style` property. This will allow us to achieve a feeling of perspective:

```
.photo {  
  // ...  
  transform-style: preserve-3d;  
}
```

The backface should not be visible initially:

```
.side {  
  // ...  
  backface-visibility: hidden;  
}
```

As for the `.back` itself, rotate it 180 degrees on the Y axis:

```
.back {
```

```
transform: rotateY(180deg);  
text-align: center;  
background-color: #e0e0e0;  
}
```

This will result in the description being hidden.

Next, define the hover animation:

```
.photo:hover {  
  transform: rotateY(180deg);  
}
```

Now when you hover over the container, the `.photo` will rotate and you'll be able to see its back with the description. You'll probably want this to happen more smoothly, so add a simple CSS transition:

```
.photo {  
  // ...  
  transition: transform 1s ease-in-out;  
}
```

Live Code

Here's the final result: See the Pen [CSS Transformations: 3D and flip](#).

A Word of Caution

Without any doubts, CSS transformations and animations are very powerful tools that can be used to create interesting and beautiful effects. However, it's important to be reasonable about their usage and not abuse them. Remember that you're

creating websites for users and not for yourself (in most cases, anyway). Therefore, CSS should be utilized to introduce better user experience, rather than to show all the cool tricks you've learned so far.

For one thing, too many effects on the page distracts the users. Some visitors may have motion sickness or vestibular disorders, so they'll find it very hard to use a website with fast animations. On top of that, you should make sure that the page works with older browsers, because it may happen that some important element is hidden or inaccessible when the animations don't work.

Conclusion

In this article, we have seen how CSS transformations in conjunction with other techniques can be used to solve various design tasks. We've seen how to vertically align elements on the page, create scalable arrows, bouncing and spinning loaders, and how to implement flip animation. Of course, you've learned only some techniques, but the only limit is your imagination.

Chapter 2: Variable Fonts: What They Are, and How to Use Them

BY CLAUDIO RIBEIRO

In this article, we'll take a look at the exciting new possibilities surrounding variable fonts — now bundled with the OpenType scalable font format — which allows a single font to behave like multiple fonts.

How We Got Here

When HTML was created, fonts and styles were controlled exclusively by the settings of each web browser. In the mid '90s, the first typefaces for screen-based media were created: Georgia and Verdana. These, together with the system fonts — Arial, Times New Roman, and Helvetica — were the only fonts available for web browsers (not exactly the only ones, but the ones we could find in every operating system).

With the evolution of web browsers, innovations like the `` tag on Netscape Navigator and the first CSS specification allowed web pages to control what font was displayed. However, these fonts needed to be installed on the

user's computer.

In 1998, the CSS working group proposed the support of the @font - face rule to allow any typeface to be rendered on web pages. IE4 implemented the technology but the distribution of fonts to every user's browser raised licensing and piracy issues.

The early 2000s saw the rise of image replacement techniques which substituted HTML content with styled-text images. Each piece of text had to be sliced in programs like Photoshop. This technique had the major advantage of allowing designers to use any typeface available without having to deal with font licensing.

In 2008, @font - face finally made a comeback when Apple Safari and Mozilla Firefox implemented it. This came out of the necessity of providing a simple way for designers and developers to use custom fonts rather than inaccessible images.

It wasn't until the arrival of the CSS3 Fonts Module in 2012 that font downloading became viable. Once implemented, a font downloaded by a web page could only be used on that page and not copied to the operating system. Font downloading allowed remote fonts to be downloaded and used by the browser, meaning that web designers could now use fonts that were not installed on the user's computer. When web designers found the font they wished to use, they just needed to include the font file on the web server, and it would be automatically downloaded to the user when needed. These

fonts were referenced using the `@font - face` rule.

To use the `@font - face` rule we have to define a font name and point to the font file:

```
@font-face {  
  font-family: Avenir Next Variable;  
  src: url(AvenirNext_Variable.woff);  
}
```

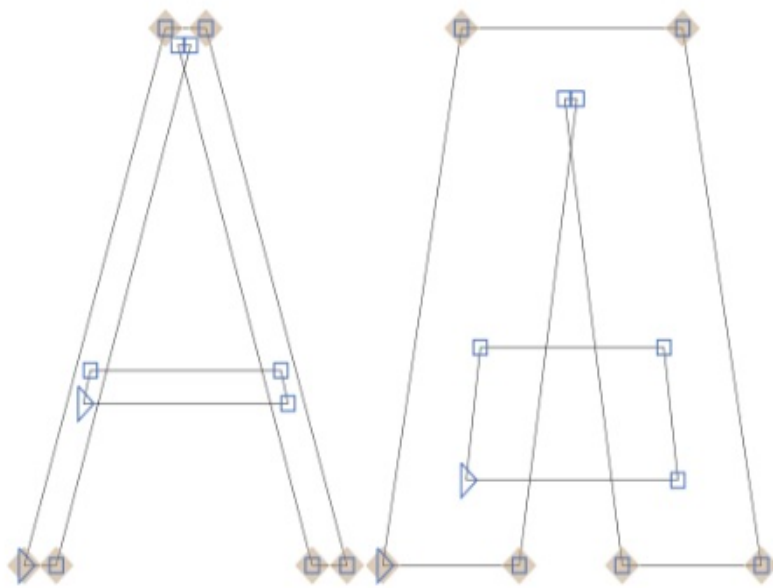
The font file can be one of five different formats: TTF, WOFF, WOFF2, SVG or EOT. Each has its own advantages and disadvantages. Putting it simply, EOT was created by Microsoft and only is supported by Internet Explorer. TTF is the basic type font created by Microsoft and Apple, and it works almost perfectly everywhere. SVG is based on image replacement techniques and is only suitable for the Web. Finally, WOFF and WOFF2 were also created exclusively for the Web and are basically TTF files with extra compression.

Variable Fonts

Version 1.8 of OpenType (the computer scalable font format) was released in 2016. A brand new technology shipped with it: OpenType font variations, also known as **variable fonts**.

This technology allows a single font to behave like multiple fonts. It's done by defining variations within the font. These variations come from the fact that each character only has one outline. The points that construct this outline have instructions on how they should behave. It's not necessary to define

multiple font weights because they can be interpolated between very narrow and very wide definitions. This also makes it possible to generate styles in between — for example, semi-bold and bold. These variations may act along one or more axes of the font. On the image below, we have an example of this outline interpolation on the letter A.



WHY ARE VARIABLE FONTS RELEVANT?

Variable fonts can bring both simplicity to our font structure and performance improvements. Take for example a situation where our website needs five font styles. It would be significantly smaller and faster to provide a single variable font capable of rendering those five styles than to have to load five different font files.

Using Variable Fonts

Using Variable Fonts

There are currently two different ways to use variable fonts. First, we'll look at the modern way of implementing those. The CSS specification strongly prefers using `fontoptical-sizing`, `fontstyle`, `fontweight` and `fontstretch` for controlling any of the standard axes.

FONTOPTICAL-SIZING

This property allows developers to control whether or not browsers render text with slightly different visual representations to optimize viewing at different sizes. It can take the value `none`, for when the browser cannot modify the shape of glyphs, or `auto` for when it can. On a browser that supports `fontoptical-sizing`, a font where the value is set to `auto` can vary like the font in the image below:

Kepler Display
Kepler Subhead
Kepler Regular
Kepler Caption

With the value set to `none` there would be no variation to the font.

FONTSTYLE

This property specifies whether a font should be styled with a normal, italic, or oblique face from its font family. It can take the `normal`, `italic` or `oblique` values.

FONTWEIGHT

This property specifies the weight (or boldness) of the font. One thing to note is that, with normal fonts, named instances can be defined. For example, `bold` is the same as `fontweight: 700` or `extra-light` is the same as `fontweight: 200`. The `fontweight` property can also be any number between 1 and 1000, but when using variable fonts, due to the interpolability, we can have a much finer granularity. For example, a value like `fontweight: 200.01` is now possible.

FONTSTRETCH

This property selects a normal, condensed, or expanded face from a font. Just like the `fontweight` property, it can be a named instance like `extra-condensed` or `normal` or a percentage between 0% and 100%. Also, named instances will map to known percentages. For example, `extra-condensed` will map to 62.5%.

For this example, I created a very simple page with a single `<h1>` heading and `<p>` paragraph.

Live Code

See the [Pen Variable Fonts HTML](#).

Currently, our unstyled webpage looks like this:

This is an h1 header

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

To use a variable font, we must find a suitable file. The [v-fonts project](#) provides a font repository where we can search and experiment with all type of variable fonts. I decided to use the [AvenirNext](#) font and link it from its [official GitHub page](#).

Then we need to create a CSS file to load this new font:

```
@font-face {
  font-family: 'Avenir Next Variable';
  src: url('AvenirNext_Variable.ttf')
  format('truetype');
}

body {
  font-family: 'Avenir Next Variable', sans-serif;
}
```

Live Code

See the [Pen Loaded variable font](#).

Due to browser support issues, this example will only apply font variations in Safari and Chrome.

This is an h1 header

This is an h2 header.

Lorem ipsum dolor sit amet, consetetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

With our font loaded, we can now use the `fontweight` property to manipulate the weight axis of our variable font.

```
h1 {  
  font-family: 'Avenir Next Variable';  
  fontweight: 430;  
}
```

Live Code

See the Pen [SourceSans variable font](#).

Most of the time we'll also need two different font files: one for italic and one for regular fonts (roman). This happens because the construction of these fonts can be radically different.

Using fontvariation-settings

The second way of using variable fonts is by using the `fontvariation-settings` CSS property. This property was introduced to provide control over OpenType or TrueType font variations, by specifying the four-letter axis names of the features you want to vary along with their

variation values. Currently, we have access to the following aspects of the font:

- **wght** — weight, which is identical to the `fontweight` CSS property. The value can be anything from 1 to 999.
- **wdth** — width, which is identical to by the `fontstretch` CSS property. It can take a keyword or a percentage value. This axis is normally defined by the font designer to expand or condense elegantly.
- **opsz** — optical sizing, which can be turned on and off using the `fontoptical-sizing` property.
- **ital** — italicization, which is controlled by the `fontstyle` CSS property when is set to `italic`.
- **slnt** — slant, which is identical to the `fontstyle` CSS property when it's set to `oblique`. It will default to a 20 degree slant, but it can also accept a specified degree between `-90deg` and `90deg`.

According to the specification, this property is a low-level feature designed to handle special cases where no other way to enable or access an OpenType font feature exists. Because of that, we should try to use `@font-face` instead.

Using the same webpage and font as above, let's try to set the weight and width on our font using the low-level controller:

```
p {  
  fontvariation-settings: 'wght' 630, 'wdth' 88;  
}
```

Live Code

See the [Pen Variable fonts 1](#).

This is an h1 header

This is an h2 header.

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

The declaration is equivalent to the following CSS declaration:

```
p {  
  fontweight: 630;  
  fontstretch: 88;  
}
```

Performance

Performance is a key advantage of variable fonts. The `AvenirNext_Variable.ttf` font file is only 89kB but creates a range of weights. A comparable standard font may have a smaller file but would only support one weight and style. Further options require additional files and raise the page weight accordingly.

But we can go even further. When we talked about font formats, we said that WOFF2 files are essentially TTF files with extra compression. WOFF2 files are smaller because they use custom preprocessing and compression algorithms to deliver ~30% file-size reduction over other formats.

Google offers a command line tool that will compress our file converting it to a `woff2` format.

On the tool's [official GitHub page](#), we can find all the information about it. To install it on a Unix environment, we can use the following commands:

```
git clone --recursive
https://github.com/google/woff2.git
cd woff2
make clean all
```

After installing it, we can use it to compress our variable font file by using:

```
woff2_compress AvenirNext_Variable.ttf
```

And we end up with a 42kb file, which halved the file size. To use this file, we just need to modify the sourced file and its format to accommodate this new file:

```
src: url('AvenirNext_Variable.woff2')
format('woff2');
```

We now have a single 42Kb file which could be used for all the font weights on our page. The only downside to the woff2 format is that it's not supported by Internet Explorer.

SERVING DIFFERENT FILES FOR DIFFERENT BROWSERS

While some modern browsers already support variable fonts (Firefox developer editions have some level of support, Chrome 62, Chrome Android, Safari iOS, and Safari), there might be the case where we find one that doesn't.

To get around this, we'll need to either serve non-variable for those browsers or use an operating system font fallback.

Browsers that support variable fonts will download the file marked as `format('woff2-variations')`, while browsers that don't will download the single style font marked as `format('ttf')`. This is possible because we can repeat references to variables in each rule. If the first fails, the second will be loaded. Just like the following:

```
@font-face {
  font-family: 'Avenir Next Variable';
  src: url('AvenirNext_Variable.woff2')
  format('woff2-variations');
  src: url('AvenirNextLTPro-Bold.otf')
  format('truetype');
}

html {
  font-family: 'Avenir Next Variable', sans-serif;
}

h1 {
  font-family: 'Avenir Next Variable';
  fontweight: 430;
}

h2 {
  font-family: 'Avenir Next Variable';
  fontweight: 630;
}
```

The next example uses a different file format from the one we're working with, but uses the same principle:

Live Code

See the [Pen Multiple fonts](#).

If we check the result on a browser that supports variable fonts, like Chrome, we can see the following:

This is an h1 header

This is an h2 header.

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

On a browser that doesn't support variable fonts, like Firefox, the second font will be loaded and the result will look like this:

This is an h1 header

This is an h2 header.

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

But if we still have to serve non-variable fonts to browsers that don't support variable ones, don't we lose all the performance we just gained with the variable font? If a browser can only load the standard font, we lose the performance and rendering benefits of variable fonts. In those situations, it may be preferable to fallback to a comparable operating system font rather than substitute it with many fixed fonts.

CONCLUSION

Despite being available for several years now, variable fonts are still in their infancy. Browser support is scarce and there

are few fonts to choose from. However, the potential is enormous, and variable fonts will permit better performance while simplifying development. For example, SitePoint's own front page currently loads eight font files with a total of 273kB. Variable fonts could reduce this dependency and cut page weight further.

Chapter 3: Scroll Snap in CSS: Controlling Scroll Action

BY TIFFANY B. BROWN

The following is a short extract from Tiffany's new book, CSS Master, 2nd Edition.

As the web platform grows, it has also gained features that mimic native applications. One such feature is the CSS Scroll Snap Module. Scroll snap lets developers define the distance an interface should travel during a scroll action. You might use it to build slide shows or paged interfaces—features that currently require JavaScript and expensive DOM operations.

Scroll snap as a feature has undergone a good deal of change. An earlier, 2013 version of the specification — called Scroll Snap *Points* at the time — defined a coordinates-and-pixels-based approach to specifying scroll distance. This version of the specification was implemented in Microsoft Edge, Internet Explorer 11, and Firefox.

Chrome 69+ and Safari 11+ implement the latest version of the specification, which uses a box alignment model. That's

what we'll focus on in this section.

Watch Out!

Many of the scroll snap tutorials currently floating around the web are based on the earlier CSS Scroll Snap **Points** specification. The presence of the word “points” in the title is one sign that the tutorial may rely on the old specification. A more reliable indicator, however, is the presence of the `scroll-snap-points-x` or `scroll-snap-points-y` properties.

Since scroll snap is really well-suited to slide show layouts, that's what we'll build. Here's our markup.

```
<div class="slideshow">
  
  
  
  
  
</div>
```

That's all we need. We don't need to have an outer wrapping element with and an inner sliding container. We also don't need any JavaScript.

Now for our CSS:

```
{
  box-sizing: border-box;
}
```

```

html, body {
  padding: 0;
  margin: 0;
}

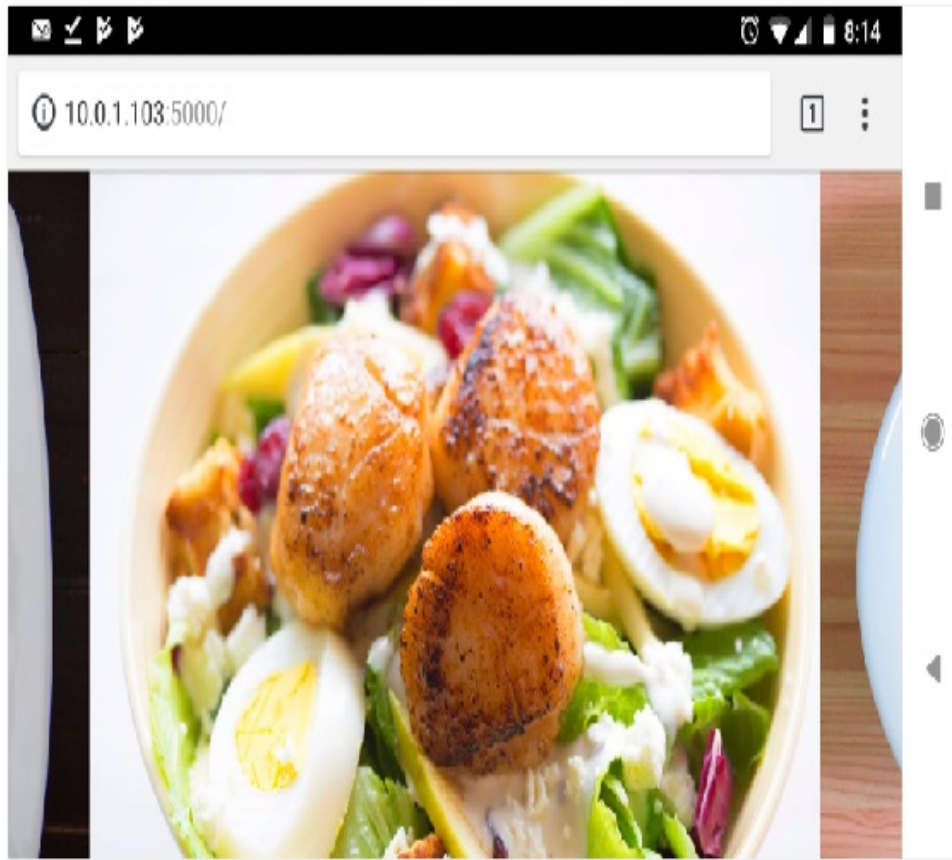
.slideshow {
  scroll-snap-type: x mandatory; / Indicates
  scroll axis and behavior
  overflow-x: auto;                Should be either
  `scroll` or `auto` */
  display: flex;
  height: 100vh;
}

.slideshow img {
  width: 100vw;
  height: 100vh;
  scroll-snap-align: center;
}

```

Adding `scroll-snap-type` to `.slideshow` creates a scroll container. The value for this property, `x mandatory` describes the direction in which we'd like to scroll, and the *scroll snap strictness*. In this case, the `mandatory` value tells the browser that it *must* snap to a snap position when there is no active scroll operation. Using `display: flex` just ensures that all of our images stack horizontally.

Now the other property we need is `scroll-snap-align`. This property indicates how to align each image's scroll snap area within the scroll container's snap port. It accepts three values: `start`, `end`, and `center`. In this case, we've used the `center` which means that each image will be centered within the viewport as shown below.



For a more comprehensive look at Scroll Snap, read [Well-Controlled Scrolling with CSS Scroll Snap](#) from Google Web Fundamentals guide.

Chapter 4: Real World Use of CSS with SVG

BY CRAIG BUCKLER

SVG is a lightweight vector image format that's used to display a variety of graphics on the Web and other environments with support for interactivity and animation. In this article, we'll explore the various ways to use CSS with SVG, and ways to include SVGs in a web page and manipulate them.

The Scalable Vector Graphic (SVG) format has been an open standard since 1999, but browser usage became practical in 2011 following the release of Internet Explorer 9. Today, SVG is well supported across all browsers, although more advanced features can vary.

SVG Benefits

Bitmap images such as PNGs, JPGs and GIFs define the color of individual pixels. An 100x100 PNG image requires 10,000 pixels. Each pixel requires four bytes for red, green, blue and transparency so the resulting file is 40,000 bytes (plus a little more for meta data). Compression is applied to reduce the file size; PNG and GIF use ZIP-like lossless compression while

JPG is lossy and removes less noticeable details.

Bitmaps are ideal for photographs and more complex images, but definition is lost as the images are enlarged.

SVGs are vector images defined in XML. Points, lines, curves, paths, ellipses, rectangles, text, *etc.* are *drawn* on an SVG canvas. For example:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0
0 800 600">
  <circle cx="400" cy="300" r="250" stroke-
width="20" stroke="#f00" fill="#ff0" >
</svg>
```

The `viewBox` defines a co-ordinate space. In this example, an 800x600 area starting at position 0,0 has a yellow circle with a red border drawn in the center:



The benefits of vectors over bitmaps:

- the SVG above uses less than 150 bytes, which is considerably smaller than an equivalent PNG or JPG
- SVG backgrounds are transparent by default
- the image can scale to any size without losing quality
- SVG code/elements can be easily generated and manipulated on the server or browser
- in terms of accessibility and SEO, text and drawing elements are machine and human-readable.

SVG is ideal for logos, charts, icons, and simpler diagrams.

Only photographs are generally impractical, although SVGs have been used for lazy-loading placeholders.

SVG Tools

It's useful to understand the basics of SVG drawing, but you'll soon want to create more complex shapes with an editor that can generate the code. Options include:

- Adobe Illustrator (commercial)
- Affinity Designer (commercial)
- Sketch (commercial)
- Inkscape (open source)
- Gravit Designer (free, online)
- Vectr (free, online)
- SVGEEdit (open source, online)
- Boxy SVG (free, app, online but Blink browsers only)
- Vecteezy - (free, online but Blink browsers only)
- SVG charting libraries — which generally create SVG charts using data passed to JavaScript functions.

Each has different strengths, and you'll get differing results for seemingly identical images. In general, more complex images require more complex software.

The resulting code can usually be simplified and minimized further using SVGO (plugins are available for most build tools) or Jake Archibold's SVGOMG interactive tool.

SVG as Static Images

SVGs as Static Images

When used within an HTML `` tag or CSS `background-url`, SVGs act identically to bitmaps:

```
<!-- HTML image -->

```

```
/* CSS background */
.myelement {
  background-image: url('mybackground.svg');
}
```

The browser will disable any scripts, links, and other interactive features embedded into the SVG file. You can manipulate an SVG using CSS in an identical way to other images using `transform`, `filters`, *etc.* The results of using CSS with SVG are often superior to bitmaps, because SVGs can be infinitely scaled.

CSS INLINED SVG BACKGROUNDS

An SVG can be inlined directly in CSS code as a background image. This can be ideal for smaller, reusable icons and avoids additional HTTP requests. For example:

```
.mysvgbackground {
  background: url('data:image/svg+xml;utf8,<svg
xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 800 600"><circle cx="400" cy="300"
r="50" stroke-width="5" stroke="#f00"
fill="#ff0" ><svg>') center center no-repeat;
}
```

Standard UTF-8 encoding (rather than base64) can be used, so

it's easy to edit the SVG image if necessary.

CSS WITH SVG: RESPONSIVE SVG IMAGES

When creating a responsive website, it's normally practical to omit `` `width` and `height` attributes and allow an image to size to its maximum width via CSS:

```
img {  
  display: block;  
  max-width: 100%;  
}
```

However, an SVG used as an image has no implicit dimensions. You might discover the `max-width` is calculated as zero and the image disappears entirely. To fix the problem, ensure a default `width` and `height` is defined in the `<svg>` tag:

```
<svg xmlns="http://www.w3.org/2000/svg"  
width="400" height="300">
```

Don't Add Dimensions

The dimensions should not be added to inlined SVGs, as we'll discover in the next section ...

HTMLInlined SVG Images

SVGs can be placed directly into the HTML. When this is

done, they become part of the DOM and can be manipulated with CSS and JavaScript:

```
<p>SVG is inlined directly into the HTML:</p>

<svg id="invader"
xmlns="http://www.w3.org/2000/svg"
viewBox="35.4 35.4 195.8 141.8">
  <path d="M70.9 35.4v17.8h17.7V35.4H70.9zm17.7
17.8v17.7H70.9v17.7H53.2V53.2H35.
  ↪4V124h17.8v17.7h17.7v17.7h17.7v-
17.7h88.6v17.7h17.7v-17.7h17.7V124h17.7V53.2h-17.
  ↪7v35.4h-17.7V70.9h-17.7V53.2h-
17.8v17.7H106.3V53.2H88.6zm88.6 0h17.7V35.4h-17.
  ↪7v17.8zm17.7 106.2v17.8h17.7v-17.8h-17.7zm-124
0H53.2v17.8h17.7v-17.8zm17.7-70.
  ↪8h17.7v17.7H88.6V88.6zm70.8 0h17.8v17.7h-
17.8V88.6z"/>
  <path d="M319 35.4v17.8h17.6V35.4H319zm17.6
17.8v17.7H319v17.7h-17.7v17.7h-17.
  ↪7V159.4h17.7V124h17.7v35.4h17.7v-
17.7H425.2v17.7h17.7V124h17.7v35.4h17.7V106.
  ↪3h-17.7V88.6H443V70.9h-17.7V53.2h-17.7v17.7h-
53.2V53.2h-17.7zm88.6 0h17.7V35.
  ↪4h-17.7v17.8zm0 106.2h-35.5v17.8h35.5v-17.8zm-
88.6 0v17.8h35.5v-17.8h-35.5zm0-70.
  ↪8h17.7v17.7h-17.7V88.6zm70.9 0h17.7v17.7h-
17.7V88.6z">
</svg>

<p>The SVG is now part of the DOM.</p>
```

No `width` or `height` attributes are defined for the SVG, so it can size to the containing element or be directly sized using CSS:

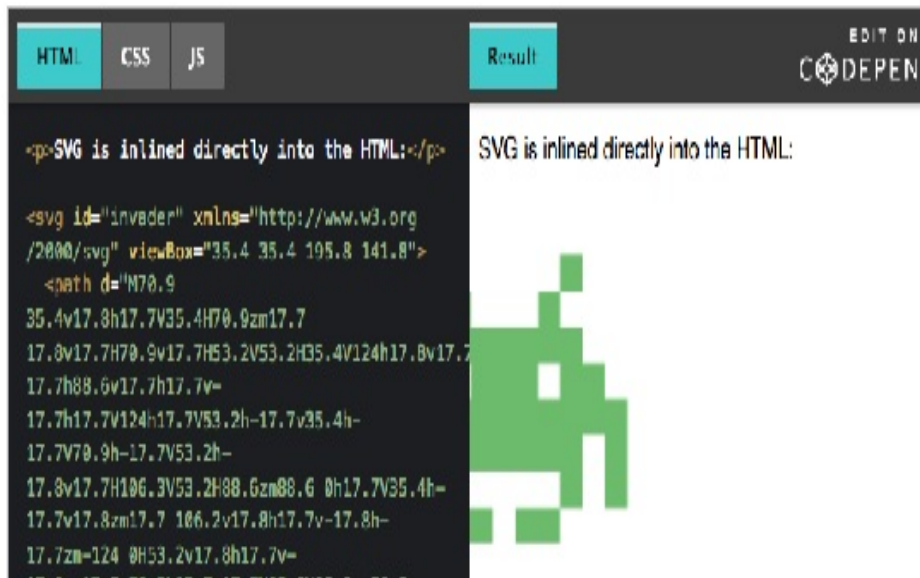
```
#invader {
  display: block;
  width: 200px;
}

#invader path {
  stroke-width: 0;
```

```
fill: #080;  
}
```

Live Code

See the Pen [HTMLInlined SVG](#).



SVG elements such as paths, circles *etc.* can be targeted by CSS selectors and have the styling modified using standard SVG attributes as CSS properties. For example:

```
/* CSS styling for all SVG circles */  
circle {  
  stroke-width: 20;  
  stroke: #f00;  
  fill: #ff0;  
}
```

This overrides any attributes defined within the SVG because the CSS has a higher specificity. SVG CSS styling offers several benefits:

- attribute-based styling can be removed from the SVG entirely to reduce the page weight
- CSS styling can be reused across any number of SVGs on any number of pages
- the whole SVG or individual elements of the image can have CSS effects applied using `:hover`, `transition`, `animation` *etc.*

SVG Sprites

A single SVG file can contain any number of separate images. For example, this `folders.svg` file contains folder icons generated by [IcoMoon](#). Each is contained within a separate `<symbol>` container with an ID which can be targeted:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <defs>
    <symbol id="icon-folder" viewBox="0 0 32 32">
      <title>folder</title>
      <path d="M14 4l4 4h14v22h-32v-26z"></path>
    </symbol>
    <symbol id="icon-folder-open" viewBox="0 0 32 32">
      <title>open</title>
      <path d="M26 30l6-16h-26l-6 16zM4 12l-4 18v-26h9l4 4h13v4z">
      </path>
    </symbol>
    <symbol id="icon-folder-plus" viewBox="0 0 32 32">
      <title>plus</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM22 22h-4v4h-4v-4h-4v-4h4v-4h4v4h4v4z">
      </path>
    </symbol>
    <symbol id="icon-folder-minus" viewBox="0 0 32 32">
      <title>minus</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM22 22h-12v-4h12v4z"></path>
    </symbol>
    <symbol id="icon-folder-download" viewBox="0 0
```

```

32 32">
    <title>download</title>
    <path d="M18 8l-4-4h-14v26h32v-22h-14zM16
271-7-7h5v-8h4v8h5l-7 7z"></path>
    </symbol>
    <symbol id="icon-folder-upload" viewBox="0 0
32 32">
    <title>upload</title>
    <path d="M18 8l-4-4h-14v26h32v-22h-14zM16
1517 7h-5v8h-4v-8h-5l7-7z"></path>
    </symbol>
  </defs>
</svg>

```

The SVG file can be referenced as an external, cached resource in an HTML page. For example, to show the folder icon at `#icon-folder`:

```

<svg class="folder" viewBox="0 0 100 100">
  <use xlink:href="folders.svg#icon-folder"></use>
</svg>

```

and style it with CSS:

```

svg.folder { fill: #f7d674; }

```

The method has a couple of drawbacks:

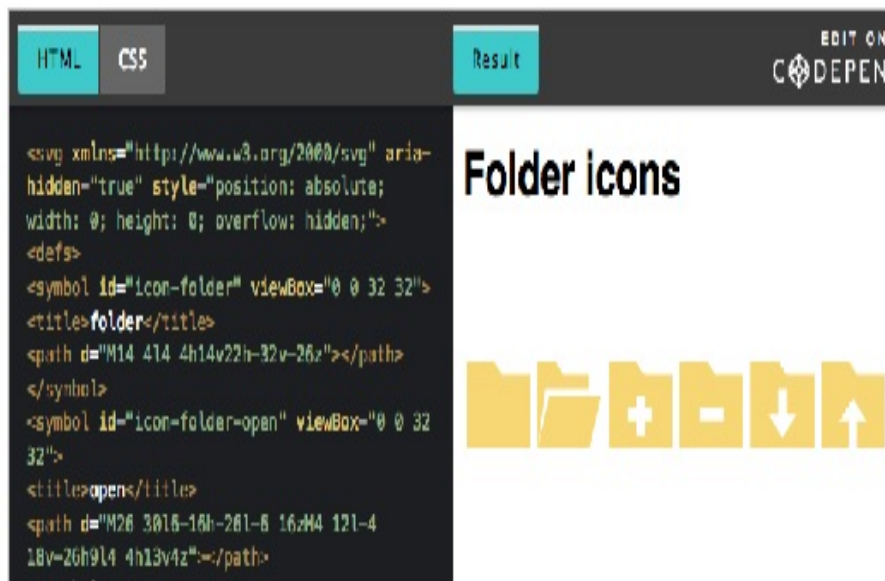
1. It fails in IE9+.
2. CSS styling only applies to the `<svg>` element containing the `<use>`. The `fill` here makes every element of the icon the same color.

To solve these issues, the SVG sprite can be embedded within page HTML, then hidden using `display: none` or similar techniques. An individual icon can be placed by referencing the ID:

```
<svg><use xlink:href="#icon-folder"></use></svg>
```

Live Code

See the Pen [SVG sprites](#).



This works in all modern browsers from IE9+ and it becomes possible to style individual elements within each icon using CSS.

Unfortunately, the SVG set is no longer cached and must be reproduced on every page where an icon is required. The solution (*to this solution!*) is to load the SVG using Ajax — which is then cached — and inject it into the page. The [IcoMoon](#) download provides a JavaScript library, or you could use [SVG for Everybody](#).

SVG Effects on HTML Content

SVG has long supported:

- **masks**: altering the visibility of parts of an element
- **clipping**: removing segments of an element so a standard regular box becomes any other shape
- **filters**: graphical effects such as blurring, brightness, shadows, *etc.*

These effects have been ported to the CSS mask, clip-path, and filter properties. However, it's still possible to target an SVG selector:

```
/* CSS */
.myelement {
  clip-path: url(#clip);
}
```

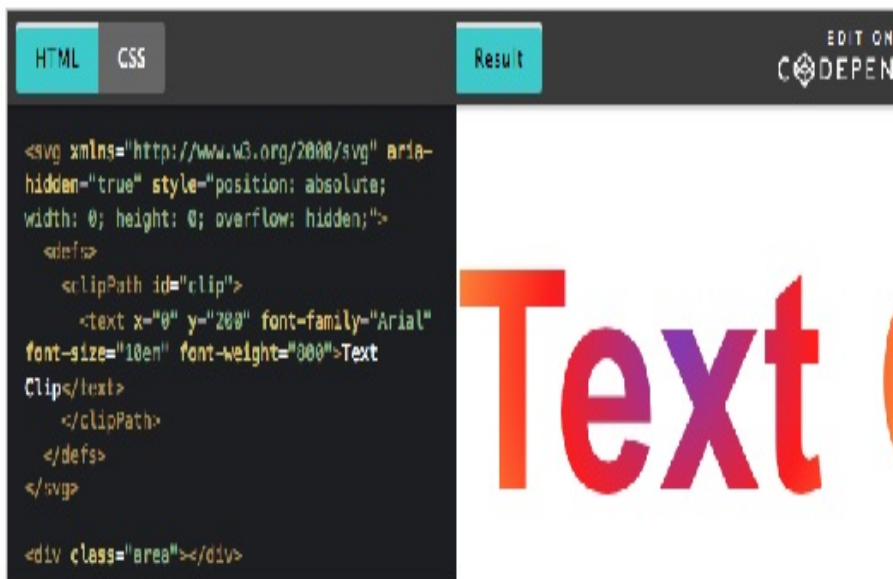
This references an effect within an HTML-embedded SVG:

```
<svg xmlns="http://www.w3.org/2000/svg" aria-
hidden="true"
  style="position: absolute; width: 0; height:
0; overflow: hidden;">
  <defs>
    <clipPath id="clip">
      <text x="0" y="200" font-family="Arial"
font-size="10em" font-weight="800">
        Text Clip</text>
      </clipPath>
    </defs>
  </svg>
```

to produce effects such as clipped text with an image or gradient background:

Live Code

See the Pen SVG clipping.



Portable SVGs

Finally, standalone SVGs can contain CSS, JavaScript and base64-encoded fonts or bitmap images! Anything outside the realms of XML should be contained within `<![CDATA[...]]>` sections.

Consider the following `invader.svg` file. It defines CSS styling with hover effects and a JavaScript animation which modifies the `viewBox` state:

```
<svg id="invader"
  xmlns="http://www.w3.org/2000/svg"
  viewBox="35.4 35.4 195.8 141.8">
  <!-- invader images:
  https://github.com/rohieb/space-invaders !-->
  <style>/* <![CDATA[
    path {
      stroke-width: 0;
      fill: #080;
    }
  ]]>
```

```

    path: hover {
      fill: #c00;
    }
  ]]> </style>
  <path d="M70.9 35.4v17.8h17.7V35.4H70.9zm17.7
17.8v17.7H70.9v17.7H53.2V53.2H35.
  ↳4V124h17.8v17.7h17.7v17.7h17.7v-
17.7h88.6v17.7h17.7v-17.7h17.7V124h17.7V53.2h-17.
  ↳7v35.4h-17.7V70.9h-17.7V53.2h-
17.8v17.7H106.3V53.2H88.6zm88.6 0h17.7V35.4h-17.
  ↳7v17.8zm17.7 106.2v17.8h17.7v-17.8h-17.7zm-124
0H53.2v17.8h17.7v-17.8zm17.7-70.
  ↳8h17.7v17.7H88.6V88.6zm70.8 0h17.8v17.7h-
17.8V88.6z">
  <path d="M319 35.4v17.8h17.6V35.4H319zm17.6
17.8v17.7H319v17.7h-17.7v17.7h-17.
  ↳7V159.4h17.7V124h17.7v35.4h17.7v-
17.7H425.2v17.7h17.7V124h17.7v35.4h17.7V106.
  ↳3h-17.7V88.6H443V70.9h-17.7V53.2h-17.7v17.7h-
53.2V53.2h-17.7zm88.6 0h17.7V35.
  ↳4h-17.7v17.8zm0 106.2h-35.5v17.8h35.5v-17.8zm-
88.6 0v17.8h35.5v-17.8h-35.5zm0-70.
  ↳8h17.7v17.7h-17.7V88.6zm70.9 0h17.7v17.7h-
17.7V88.6z">
  <script>/ <![CDATA[
    const
      viewX = [35.4, 283.6],
      animationDelay = 500,
      invader =
document.getElementById('invader');

    let frame = 0;

    setInterval(() => {
      frame = ++frame % 2;
      invader.viewBox.baseVal.x = viewX[frame];
    }, animationDelay);

  ]]> */</script>
</svg>

```

When referenced in an HTML `` or CSS background, the SVG becomes a static image of the initial state (in essence, the first animation *frame*).

However, open the image in its own browser tab and all the effects will return.

This could be useful for distributing images, demonstrations or small documents which require a level of embedded interactivity.

Sophisticated SVGs

SVGs offer a wide range of technical possibilities both within and outside web pages. When combining CSS with SVG, it becomes possible to style and animate the whole image or individual elements in interesting ways.

This article describes ways to manipulate SVG images, but they are regularly used for smaller visual enhancements such as:

- form focus highlights and validation
- turning a hamburger menu into a an X close icon
- creating lava-lamp-like morphing.

Despite the age of SVG technology, web developers are still discovering ways to transform boring block-based pages with subtle effects through using CSS with SVG.

Chapter 5: CSS and PWAs: Some Tips for Building Progressive Web Apps

BY DAVID ATTARD

In recent years we've seen a number of major shifts in the online experience, mostly coming from the proliferation of mobile devices. The evolution of the Web has taken us from single versions of a website, to desktop versus mobile versions, to responsive sites that adapt according to screen size, then to native mobile apps, which either recreate the desktop experience as a native app, or act as a gateway to the responsive version.

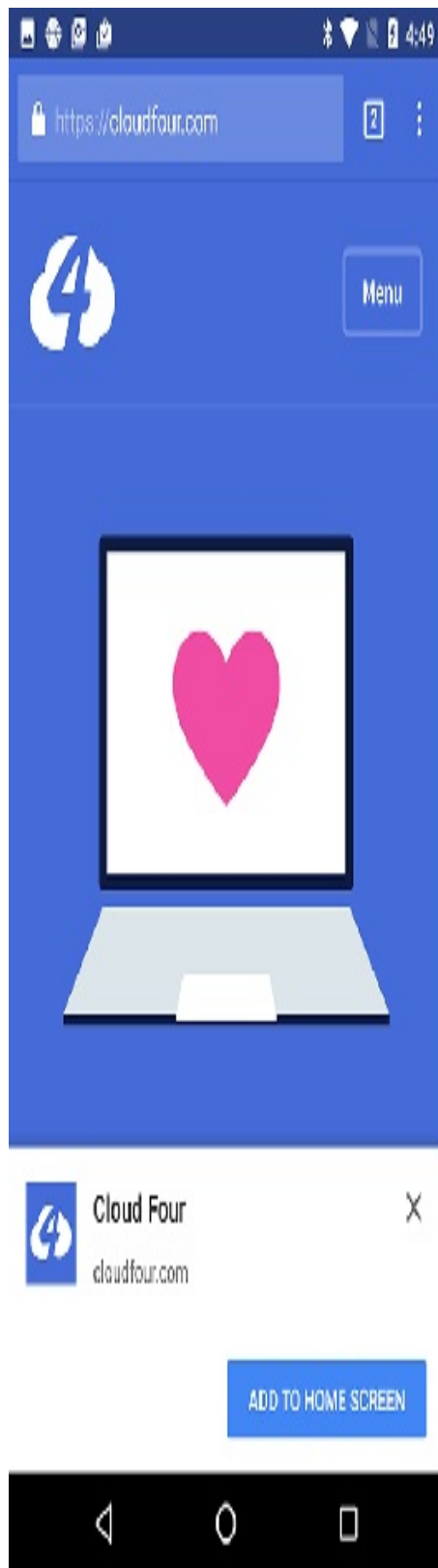
The latest iteration of all of this is the progressive web app (PWA). A PWA is a software platform that aims to combine the best of both the Web and the native experience for website/app users.

In this article on CSS and PWAs, we're going to discuss a number of techniques that can be used when creating the CSS required for the development of PWAs.

What are PWAs?

There are three main features of a PWA. As you'll see, what makes a web app progressive is the “fixing” of problems typically associated with web apps, by adopting some of the techniques used by native apps to resolve these issues.

1. **Reliable.** A PWA should reliably load like a native app (no matter the state of the network). This is contrary to a web page, which typically does not load if the device is disconnected from the network.
2. **Fast.** The performance of a PWA should be independent of such things as geography, network speed, load or other factors that are beyond the control of the end user.
3. **Engaging.** PWAs should mimic the native app's immersive, full-screen experience without requiring the need of an app store, even supporting such features as push notifications.



There are other features PWA features, but for now, we'll

keep to the most important ones described above.

Google has been at the forefront of pushing these kind of apps, but the adoption of PWAs has been picking up with vendors and plenty of other companies on the Web helping the adoption and embracing the concept of PWAs.

The following are comments from Itai Sadan, CEO of Duda, who was present at Cloudfest 2018:

Progressive web apps represent the next great leap in the evolution of web design and online presence management ... they take the best aspects of native apps, such as a feature-rich experience and the ability to be accessed offline, and incorporate them into responsive websites. This creates an incredible web experience for users without the need to download anything onto their device.

Anyone providing web design services to clients is going to want to offer PWAs because over time, just like with mobile and responsive web design, it will become the industry standard.

What is Required for Developing PWAs?

Developing a PWA is not different from developing a standard web application, and it may be possible to upgrade your existing codebase. Note that for deployment, HTTPS is a requirement, although you can do testing on the localhost. The

requirements for an app to become a PWA are discussed below.

1. CREATE A MANIFEST FILE

PWAs must be available to install directly via a website which has been visited, meaning there's no need for a third-party app store to get the app installed.

To enable this, the app needs to provide a manifest.json file — a JSON file that allows the developer to control how the PWA appears, what needs to be launched and other parameters.

A typical manifest file appears below. As we can see, the properties are setting a number of look-and-feel settings that will be used on the home screen of the device where the app will be installed.

```
{
  "short_name": "PWA",
  "name": "My PWA",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "192x192",
      "type": "image.png",
    }
  ],
  "start_url" : "./index.html",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff",
}
```

The styling of the PWA starts from the manifest file, but

there's no real CSS involved in this part. It's just straight up properties, which define the application's name, icons, primary colors, *etc.*

2. USING A SERVICE WORKER

A service worker is essentially a specific type of web worker, implemented as a JavaScript file that runs independently of the browser — such that it's able to intercept network requests, caching or retrieving resources from the cache, and delivering push messages as necessary.

The service worker is what makes the PWA truly offline capable.

3. INSTALL THE SITE ASSETS, INCLUDING CSS

The first time the Service worker is registered, an install event is triggered. This is where all of the site assets are installed, including any CSS, JS and other media and resource files required by the application:

```
self.addEventListener('install', function(e) {
  e.waitUntil(
    caches.open('airhorner').then(function(cache)
    {
      return cache.addAll([
        '',
        'index.html',
        'index.html?homescreen=1',
        '?homescreen=1',
        'stylesmain.css',
        'scriptsmain.min.js',
        'soundsairhorn.mp3'
      ])
    })
  )
})
```

```
    });  
  })  
);  
});
```

Developing PWAs is not very different from developing web apps, as long as the fundamental requirements have been met.

This is where the CSS starts to get involved, with the files defined that will be used to style the progressive web app.

CSS and PWAs: Some Considerations

When considering CSS and PWAs, there are things we need to keep in mind. All of these are decisions that need to be taken before the development of a progressive web app starts.

SHOULD THE APP FOLLOW PLATFORMSPECIFIC UIS?

If we opt for one platform in favor of another (let's say Android in favor of iOS) we risk alienating or putting at a disadvantage that part of the audience we didn't consider.

We're also tying our fortunes to that platform — whether good fortunes or bad ones. It's also quite likely that platform designs change as they evolve between different versions.

My opinion is that vendor tie-in should be avoided as much as possible.

PLATFORMAGNOSTIC DESIGN

Based on our previous consideration, the ideal is to opt for a mostly platform-neutral design.

If this path is chosen, we should ensure that the result doesn't stray too much in form and function from the UI that the native platform exposes.

One needs to use standard behaviors and perform-extensive user testing to ensure no UX problems have been introduced on specific platforms. As an example, it's highly recommended to avoid custom-written components and opt for standard HTML5 controls, which the browser can optimize for the UI and best experience.

DEVICE CAPABILITIES

The way forward for PWAs — even if at this point they're mostly focused on devices — is to become a holistic solution for all platforms, including desktops. As of May 2018, Chrome supports PWAs on desktops, and other vendors will soon be supporting this too.

Your CSS and styling considerations need to factor all of this and design for this from the get-go.

The beauty of working with a PWA, though, is that you can use a combination of CSS and the Service Worker implementation to enhance or limit the functionality based on the resources available.

GRACEFUL DEGRADATION AND PROGRESSIVE ENHANCEMENT

CSS in general is able to fall back gracefully; any unsupported properties are simply ignored.

Having said that, one also needs to make sure that critical elements have the right fallbacks, or are not missing any essential styling.

An alternative approach to graceful degradation is progressive enhancement. This is a concept that we should always keep in mind when working on our PWA. For example, we should test first for the support of a Service Worker API before we attempt to use it, using code similar to this:

```
if (!('serviceWorker' in navigator)) {  
  console.log('Service Worker not supported');  
  return;  
}
```

Variations of this logic can be used to handle different use cases, such as the styling for specific platforms, and others that we'll mention later on.

GENERAL SUGGESTIONS

Although PWAs have a number of advantages when it comes to the user experience, they shift a lot of responsibility back to the developer when it comes to dealing with the nuances of different technology.

Speaking as a developer/Product Manager, who understands the delicate balance between user needs and the limited availability of development resources, I would always recommend finding a middle ground that covers as many use cases as possible, without putting too much overhead on the development and testing teams.

The emergence of design standards such as Material Design, and frameworks such as Bootstrap, helps to establish platformagnostic designs.

The actual framework used is typically able to address devices of different capabilities, while the design school provides a homogeneous look and feel across platforms, allowing the developer to focus on the App's features and functions.

If, on the other hand, you'd rather go down the whole separate look and feel, you'll be able to use your service worker to be able to do this.

JavaScript provides a number of functions that can help to take decisions based on the platform and capabilities available. You can, therefore, use your code to test for the platform and then apply a stylesheet accordingly.

For example, the navigator.platform method returns the platform on which the app is running, while the navigator.userAgent returns the browser being used.

The browser agent is unreliable when it comes to detecting the browser, so the code below is more of a demonstration of a

concept rather than code that should be used in a live environment.

The `navigator.platform` is a more reliable method, but the sheer number of platforms available makes it cumbersome to use in our example.

```
/**
 * Determine the mobile operating system.
 * This function returns one of 'iOS', 'Android',
 * 'Windows Phone', or 'unknown'.
 *
 * @returns {String}
 */

function getMobileOperatingSystem()
{
    var userAgent = navigator.userAgent ||
    navigator.vendor || window.opera;
    // Windows Phone must come first because its UA
    also contains "Android"
    if (/windows phone/i.test(userAgent))
    {
        return "Windows Phone";
    }

    if (/android/i.test(userAgent))
    {
        return "Android";
    }

    if (/iPad|iPod/i.test(userAgent) &&
    !window.MSStream)
    {
        return "iOS";
    }

    return "unknown";
    // return "Android" - one should either handle
    the unknown or fallback to a specific platform,
    let's say Android
}
```

Using the return value of

`getMobileOperatingSystem()` above, you can then register a specific CSS file accordingly.

From the code above, we can see that such logic can get very convoluted and difficult to maintain, so I would only recommend using it in situations where a platformagnostic design is not suitable.

Another option is to use a simpler color scheme, only CSS applied to the primary styles that match the underlying OS, though this could still “break” in the case where users have skinned their device.

PWA Frameworks

When learning how to develop a PWA, it’s great to create everything manually: it’s an excellent way of learning all the fundamentals concepts of building progressive web apps.

Once you’ve become familiar with all the important aspects, you might then start using a few tools to help you out, increasing your development efficiency.

As with most development scenarios, frameworks are available to make development of PWAs faster and more efficient.

Each of these frameworks uses specific CSS techniques to ensure that the development process is maintainable, scalable and achieves the needs of both the developer and the end user.

By using such frameworks, you can ensure that your PWA works nicely on most devices and platforms, because the frameworks usually have cross-platform capabilities, although they may offer limited backward compatibility. This is another of those decisions you'll need to take when deciding what you'll be using to develop your progressive web app. By using frameworks, you cede some of the control you'd have if writing everything from scratch.

Below we'll suggest a number of frameworks/tools that can be used to aid development of PWAs.

A word of advice, though: frameworks add a lot of overhead when it comes to performance.

We recommend that you only use these resources when starting out, eventually opting out of using them and going for minimalistic, lean stylesheets, using a platformagnostic design.

1. CREATE REACT APP

React has all of the components in place to allow the development of a PWA, by using such libraries as the Create React App.

This is a great example of creating a React PWA with this library.

2. ANGULAR

Given that Angular is a product of Google and how we've

seen the company pushing for PWAs, it's no surprise that Angular has full support for PWAs.

If you're used to working with Angular, you could consider using this as your framework of choice.

Angular 2+ supports the implementation of PWA features (such as service workers and manifest files) natively through the framework using the following commands:

```
ng add @angular/pwa --project project_name
```

This is a great article which guides you through creating a PWA with Angular.

IONIC

Ionic is another framework for PWAs. The framework

- leverages Angular to enable the creation of native apps using web technologies
- uses Cordova to run the app on devices such as phones
- has a built-in service worker and manifest support.

This is a premium framework that includes a number of developer-oriented and team-oriented features such as rapid prototyping, to make development faster.

PWAs and Performance

One of the fundamentals of progressive web apps remains that

of a fast and engaging user experience.

For this reason, when considering the CSS, one needs to ensure to keep things as lean and minimalistic as possible.

This is one of the aspects where frameworks start to suffer. They add extra CSS resources that you're not using, which can reduce performance in PWAs.

A number of considerations you might want to keep in mind:

- use HTTP/2 on your server
- use such hints as `rel=preload` to allow early fetching of critical CSS and other resources
- use the [NetworkInformationAPI](#) and a caching implementation to access cached resources rather than downloading them
- inline critical CSS directly into the HTML document to optimize performance — which typically should be done for anything above the fold
- keep resources as lean and as small as possible
- minify all of your CSS resources and implement other optimizations such as compressing resources, optimizing images and use optimized image and video formats.

The [Google guidelines on performance](#) have other details you should keep in mind.

GOOGLE LIGHTHOUSE

Speaking of performance, the Google Lighthouse is a performance monitoring tool centered specifically around increasing performance, both of websites and progressive web apps.

Lighthouse, which used to be a plugin for Chrome, is today built-in with the Chrome Developer tools. It can be used to run tests against the progressive web app. The test generates a report which has plenty of detail to help you keep your development within the performance parameters of a PWA.

Wrapping Up

Using CSS and PWAs together has a few differences from using CSS to develop your web application or website (particularly in terms of performance and responsiveness). However, most techniques that can be used with web development can be suitably adopted for development of progressive web apps. Whether you use frameworks or build everything from scratch, weigh the benefits against the disadvantages, take an informed decision and then stick with it.

Chapter 6: 20 Tips for Optimizing CSS Performance

BY CRAIG BUCKLER

In this article, we look at 20 ways to optimize your CSS so that it's faster-loading, easier to work with and more efficient.

According to the latest [HTTP Archive reports](#), the web remains a bloated mess with the mythical median website requiring 1,700Kb of data split over 80 HTTP requests and taking 17 seconds to fully load on a mobile device.

[The Complete Guide to Reducing Page Weight](#) provides a range of suggestions. In this article, we'll concentrate on CSS. Admittedly, CSS is rarely the worst culprit and a typical site uses 40KB spread over five stylesheets. That said, there are still optimizations you can make, and ways to change how we use CSS that will boost site performance.

1. Learn to Use Analysis Tools

You can't address performance problems unless you know

where the faults lie. Browser DevTools are the best place to start: launch from the menu or hit F12, Ctrl + Shift + I or Cmd + Alt + I for Safari on macOS.

All browsers offer similar facilities, and the tools will open slowly on badly-performing pages! However, the most useful tabs include the following ...

The **Network** tab displays a waterfall graph of assets as they download. For best results, disable the cache and consider throttling to a lower network speed. Look for files that are slow to download or block others. The browser normally blocks browser rendering while CSS and JavaScript files download and parse.

The **Performance** tab analyses browser processes. Start recording, run an activity such as a page reload, then stop recording to view the results. Look for:

1. Excessive *layout/reflow* actions where the browser has been forced to recalculate the position and size of page elements.
2. Expensive *paint* actions where pixels are changed.
3. *Compositing* actions where the painted parts of the page are put together for displaying on-screen. This is normally the least processor-intensive action.

Chrome-based browsers provide an **Audits** tab which runs Google's Lighthouse tool. It's often used by Progressive Web App developers, but also makes CSS performance suggestions.

ONLINE OPTIONS

Alternatively, use online analysis tools that are not influenced

by the speed and capabilities of your device and network.
Most can test from alternative locations around the world:

- [Pingdom Website Speed Test](#)
- [GTmetrix](#)
- [Google PageSpeed Insights](#)
- [WebPageTest](#)

2. Make Big Wins First

CSS is unlikely to be the direct cause of performance issues.
However, it may load heavy-hitting assets which can be optimized within minutes. Examples:

- Activate HTTP/2 and GZIP compression on your server
- Use a content delivery network (CDN) to increase the number of simultaneous HTTP connections and replicate files to other locations around the world
- Remove unused files.

Images are normally the biggest cause of page bulk, yet many sites fail to optimize effectively:

1. Resize bitmap images. An entry-level smartphone will take multi-megapixel images that can't be displayed in full on the largest HD screen. Few sites will require images of more than 1,600 pixels in width.
2. Ensure you use an appropriate file format. Typically, JPG is best for photographs, SVG for vector images, and PNG for everything else. You can experiment to find the optimum type.
3. Use image tools to reduce file sizes by stripping metadata and increasing compression factors.

That said, be aware that xKb of image data is **not** equivalent to

xKb of CSS code. Binary images download in parallel and require little processing to place on a page. CSS blocks rendering and must be parsed into an object model before the browser can continue.

3. Replace Images with CSS Effects

It's rarely necessary to use background images for borders, shadows, rounded edges, gradients and some geometric shapes. Defining an "image" using CSS code uses considerably less bandwidth and is easier to modify or animate later.

4. Remove Unnecessary Fonts

Services such as [Google Fonts](#) make it easy to add custom fonts to any page. Unfortunately, a line or two of code can retrieve hundreds of kilobytes of font data. Recommendations:

1. Only use the fonts you need.
2. Only load the weights and styles you require — for example, roman, 400 weight, no italics.
3. Where possible, limit the character sets. Google fonts allows you to pick certain characters by adding a `&text=` value to the font URL — such as `fonts.googleapis.com/css?family=Open+Sans&text=SitePon` for displaying "SitePoint" in Open Sans.
4. Consider [variable fonts](#), which define multiple weights and styles by interpolation so files are smaller. Support is currently [limited to Chrome, Edge, and some editions of Safari](#) but should grow rapidly. See [How to Use Variable Fonts](#).

5. Consider OS fonts. Your 500Kb web font may be on-brand, but would anyone notice if you switched to the commonly available Helvetica or Arial? Many sites use custom web fonts, so standard OS fonts are considerably less common than they were!

5. Avoid `@import`

The `@import` at-rule allows any CSS file to be included within another. For example:

```
/* main.css */
@import url("base.css");
@import url("layout.css");
@import url("carousel.css");
```

This appears a reasonable way to load smaller components and fonts. *It's not.* `@import` rules can be nested so the browser must load and parse each file in series.

Multiple `<link>` tags within the HTML will load CSS files in parallel, which is considerably more efficient — especially when using HTTP/2:

```
<link rel="stylesheet" href="base.css">
<link rel="stylesheet" href="layout.css">
<link rel="stylesheet" href="carousel.css">
```

That said, there may be more preferable options ...

6. Concatenate and Minify

Most build tools allow you to combine all partials into one

large CSS file that has unnecessary whitespace, comments and characters removed.

Concatenation is less necessary with HTTP/2, which pipelines and multiplexes requests. In some cases, separate files may be beneficial if you have smaller, regularly-changing CSS assets. However, most sites are likely to benefit from sending a single file that is immediately cached by the browser.

Minification may not bring considerable benefits when you have GZIP enabled. That said, there's no real downside.

Finally, you could consider a build process that orders properties consistently within declarations. GZIP can maximize compression when commonly used strings are used throughout a file.

7. Use Modern Layout Techniques

For many years it was necessary to use CSS `float` to lay out pages. The technique is a hack. It requires lots of code and margin/padding tweaking to ensure layouts work. Even then, floats will break at smaller screen sizes unless media queries are added.

The modern alternatives:

- CSS Flexbox for one-dimensional layouts which (can) wrap to the next row according to the widths of each block. Flexbox is ideal for menus, image galleries, cards, *etc.*

- CSS Grid for two-dimensional layouts with explicit rows and columns. Grid is ideal for page layouts.

Both options are simpler to develop, use less code, can adapt to any screen size, and render faster than floats because the browser can natively determine the optimum layout.

8. Reduce CSS Code

The most reliable and fastest code is the code you need never write! The smaller your stylesheet, the quicker it will download and parse.

All developers start with good intentions, but CSS can bloat over time as the feature count increases. It's easier to retain old, unnecessary code rather than remove it and risk breaking something. A few recommendations to consider:

- Be wary of large CSS frameworks. You're unlikely to use a large percentage of the styles, so only add modules as you need them.
- Organize CSS into smaller files (partials) with clear responsibilities. It's easier to remove a carousel widget if the CSS is clearly defined in `widgets/_carousel.css`.
- Consider naming methodologies such as BEM to aid the development of discrete components.
- Avoid deeply nested Sass/preprocessor declarations. The expanded code can become unexpectedly large.
- Avoid using `!important` to override the cascade.
- Avoid inline styles in HTML.

Tools such as UnCSS can help remove redundant code by analyzing your HTML, but be wary about CSS states caused

by JavaScript interaction.

9. Cling to the Cascade!

The rise of CSS-in-JS has allowed developers to avoid the CSS global namespace. Typically, randomly generated class names are created at build time so it becomes impossible for components to conflict.

If your life has been improved by CSS-in-JS, then carry on using it. However, it's worth understanding the benefits of the CSS cascade rather than working against it on every project. For example, you can set default fonts, colors, sizes, tables and form fields that are universally applied to every element in a single place. There is rarely a need to declare every style in every component.

10. Simplify Selectors

Even the most complex CSS selectors take milliseconds to parse, but reducing complexity will reduce file sizes and aid browser parsing. Do you really need this sort of selector?!

```
body > main.main > section.first h2:nth-of-  
type(odd) + p::first-line > a[href$=  
=".pdf"]
```

Again, be wary of deep nesting in preprocessors such as Sass, where complex selectors can be inadvertently created.

11. Be Wary of Expensive Properties

Some properties are slower to render than others. For added jankiness, try placing box shadows on all your elements!

```
*, ::before, ::after {  
  box-shadow: 5px 5px 5px rgba(0,0,0,0.5);  
}
```

Browser performance will vary but, in general, anything which causes a recalculation before painting will be more costly in terms of performance:

- `border-radius`
- `box-shadow`
- `opacity`
- `transform`
- `filter`
- `position: fixed`

12. Adopt CSS Animations

Native CSS transitions and animations will always be faster than JavaScript-powered effects that modify the same properties. CSS animations will not work in older browsers such as IE9 and below, but those users will never know what they're missing.

That said, avoid animation for the sake of it. Subtle effects can

enhance the user experience without adversely affecting performance. Excessive animations could slow the browser and cause motion sickness for some users.

13. Avoid Animating Expensive Properties

Animating the dimensions or position of an element can cause the whole page to re-layout on every frame. Performance can be improved if the animation only affects the *compositing* stage. The most efficient animations use:

- `opacity` and/or
- `transform` to translate (move), scale or rotate an element (the original space the element used is not altered).

Browsers often use the hardware-accelerated GPU to render these effects. If neither are ideal, consider taking the element out of the page flow with `position: absolute` so it can be animated in its own layer.

14. Indicate Which Elements Will Animate

The `will-change` property allows CSS authors to indicate an element will be animated so the browser can make performance optimizations in advance. For example, to declare that an element will have a `transform` applied:

```
.myelement {  
  will-change: transform;  
}
```

Any number of comma-separated properties can be defined.

However:

- use `will-change` as a last resort to fix performance issues
- don't apply it to too many elements
- give it sufficient time to work: that is, don't begin animations immediately.

15. Adopt SVG Images

Scalable vector graphics (SVGs) are typically used for logos, charts, icons, and simpler diagrams. Rather than define the color of each pixel like JPG and PNG bitmaps, an SVG defines shapes such as lines, rectangles and circles in XML. For example:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0  
0 800 600">  
<circle cx="400" cy="300" r="50" stroke-width="5"  
stroke="#f00" fill="#ff0" />  
</svg>
```

Simpler SVGs are smaller than equivalent bitmaps and can infinitely scale without losing definition.

An SVG can be inlined directly in CSS code as a background image. This can be ideal for smaller, reusable icons and avoids additional HTTP requests. For example:

```
.mysvgbackground {  
background: url('data:image/svg+xml;utf8,<svg  
xmlns="http://www.w3.org/2000/svg"  
→viewBox="0 0 800 600"><circle cx="400" cy="300"  
r="50" stroke-width="5"  
→stroke="#f00" fill="#ff0" ><svg>') center center  
no-repeat;  
}
```

16. Style SVGs with CSS

More typically, SVGs are embedded directly within an HTML document:

```
<body>  
<svg class="mysvg"  
xmlns="http://www.w3.org/2000/svg" viewBox="0 0  
800 600">  
  <circle cx="400" cy="300" r="50" >  
</svg>  
</body>
```

This adds the SVG nodes directly into the DOM. Therefore, all SVG styling attributes can be applied using CSS:

```
circle {  
stroke-width: 1em;  
}  
  
.mysvg {  
stroke-width: 5px;  
stroke: #f00;  
fill: #ff0;  
}
```

The volume of embedded SVG code is reduced and the CSS styles can be reused or animated as necessary.

Note that using an SVG within an `` tag or as a CSS background image means they're separated from the DOM, and CSS styling will have no effect.

17. Avoid Base64 Bitmap Images

Standard bitmap JPGs, PNGs and GIFs can be encoded to a base64 string within a data URI. For example:

```
.myimg {  
  background-image:  
  url('data:image/png;base64,ABCDEFetc+etc+etc');  
}
```

Unfortunately:

- base64 encoding is typically 30% larger than its binary equivalent
- the browser must parse the string before it can be used
- altering an image invalidates the whole (cached) CSS file.

While fewer HTTP requests are made, it rarely provides a noticeable benefit — especially over HTTP/2 connections. In general, avoid inlining bitmaps unless the image is unlikely to change often and the resulting base64 string is unlikely to exceed a few hundred characters.

18. Consider Critical CSS

Those using Google page analysis tools will often see

suggestions to “*inline critical CSS*” or “*reduce render-blocking stylesheets*”. Loading a CSS file blocks rendering, so performance can be improved with the following steps:

1. Extract the styles used to render elements above the fold. Tools such as criticalCSS can help.
2. Add those to a `<style>` element in the HTML `<head>`.
3. Load the main CSS file asynchronously using JavaScript (perhaps after the page has loaded).

The technique undoubtedly improves performance and could benefit Progressive Web or single-page apps that have consistent interfaces. Gains may be less clear for other sites/apps:

- It’s impossible to identify the “fold”, and it changes on every device.
- Most sites have differing page layouts. Each one could require different critical CSS, so a build tool becomes essential.
- Dynamic, JavaScript-driven events could make above-the-fold changes that are not identified by critical CSS tools.
- The technique mostly benefits the user’s first page load. CSS is cached for subsequent pages so additional inlined styles will *increase* page weight.

That said, Google will love your site and push it to #1 for every search term. (*SEO “experts” can quote me on that. Everyone else will know it’s nonsense.*)

19. Consider Progressive Rendering

Rather than using a single site-wide CSS file, **progressive rendering** is a technique that defines individual stylesheets for

separate components. Each is loaded immediately before a component is referenced in the HTML:

```
<head>

<!-- core styles used across components -->
<link rel='stylesheet' href='base.css' >

<head>
<body>

<!-- header component -->
<link rel='stylesheet' href='header.css' >
<header>...</header>

<!-- primary content -->
<link rel='stylesheet' href='content.css' >
<main>

    <!-- form styling -->
    <link rel='stylesheet' href='form.css' >
    <form>...</form>

</main>

<!-- footer component -->
<link rel='stylesheet' href='footer.css' >
<footer>...</footer>

</body>
```

Each `<link>` still blocks rendering, but for a shorter time, because the file is smaller. The page is usable sooner, because each component renders in sequence; the top of the page can be viewed while remaining content loads.

The technique works in Firefox, Edge and IE. Chrome and Safari “optimize” the experience by loading all CSS files and showing a white screen while that occurs — but that’s no worse than loading each in the `<head>`.

Progressive rendering could benefit large sites where individual pages are constructed from a selection of different components.

20. Learn to Love CSS

The most important tip: *understand your stylesheets!*

Adding vast quantities of CSS from StackOverflow or Bootstrap may produce quick results, but it will also bloat your codebase with unused junk. Further customization becomes frustratingly difficult, and the application will never be efficient.

CSS is easy to learn but difficult to master. You can't avoid the technology if you want to create effective client-side code. A little knowledge of CSS basics can revolutionize your workflow, enhance your apps, and noticeably improve performance.

Chapter 7: Advanced CSS Theming with Custom Properties and JavaScript

BY AHMED BOUCHEFRA

Throughout this tutorial on CSS theming, we'll be using CSS custom properties (also known as CSS variables) to implement dynamic themes for a simple HTML page. We'll create dark and light example themes, then write JavaScript to switch between the two when the user clicks a button.

Just like in typical programming languages, variables are used to hold or store values. In CSS, they're typically used to store colors, font names, font sizes, length units, *etc.* They can then be referenced and reused in multiple places in the stylesheet. Most developers refer to “CSS variables”, but the official name is **custom properties**.

CSS custom properties make it possible to modify variables that can be referenced throughout the stylesheet. Previously, this was only possible with CSS preprocessors such as Sass.

Understanding `:root` and `var ()`

Before creating our dynamic theming example, let's understand the essential basics of custom properties.

A custom property is a property whose name starts with two hyphens (--) like `--foo`. They define variables that can be referenced using `var()`. Let's consider this example:

```
:root {  
  --bg-color: #000;  
  --text-color: #fff;  
}
```

Defining custom properties within the `:root` selector means they are available in the global document space to all elements. **:root** is a CSS pseudo class which matches the root element of the document — the `<html>` element. It's similar to the `html` selector, but with higher specificity.

You can access the value of a `:root` custom property anywhere in the document:

```
div {  
  color: var(--text-color);  
  background-color: var(--bg-color);  
}
```

You can also include a fallback value with your CSS variable. For example:

```
div {  
  color: var(--text-color, #000);  
  background-color: var(--bg-color, #fff);  
}
```

If a custom property isn't defined, their fallback value is used instead.

Defining custom properties inside a CSS selector other than the `:root` or `html` selector makes the variable available to matching elements and their children.

CSS Custom Properties vs Preprocessor Variables

CSS preprocessors such as Sass are often used to aid front-end web development. Among the other useful features of preprocessors are variables. But what's the difference between Sass variables and CSS custom properties?

- CSS custom properties are natively parsed in modern browsers. Preprocessor variables require compilation into a standard CSS file and all variables are converted to values.
- Custom properties can be accessed and modified by JavaScript. Preprocessor variables are compiled once and only their final value is available on the client.

Writing a Simple HTML Page

Let's start by creating a folder for our project:

```
$ mkdir css-variables-theming
```

Next, add an `index.html` inside the project's folder:

```
$ cd css-variables-theming
$ touch index.html
```

And add the following content:

```
<nav class="navbar">Title</nav>
<div class="container">
  <div>
    <input type="button" value="Light/Dark"
id="toggle-theme" >
    <div>
      <h2 class="title">What is Lorem Ipsum?</h2>
      <p class="content">Lorem Ipsum is simply dummy
text of the printing and
      ↳typesetting industry...</p>
    </div>
  </div>
  <footer>
    Copyright 2018
  </footer>
```

We are adding a navigation bar using a `<nav>` tag, a footer, and a container `<div>` that contains a button (that will be used to switch between light and dark themes) and some dummy *Lorem Ipsum* text.

Writing Basic CSS for Our HTML Page

Now let's style our page. In the same file using an inline `<style>` tag in the `<head>` add the following CSS styles:

```
<style>
* {
  margin: 0;
}
html{
  height: 100%;
```

```

}
body{
  height: 100%;
  font-family: -apple-system,
BlinkMacSystemFont"Segoe UI", "Roboto", "Oxygen",
  ↳"Ubuntu", "Cantarell","Fira Sans", "Droid
Sans", "Helvetica Neue",sans-serif;
  display: flex;
  flex-direction: column;
}
nav{
  background: hsl(350, 50%, 50%);
  padding: 1.3rem;
  color: hsl(350, 50%, 10%);
}
.container{
  flex: 1;
  background:hsl(350, 50%, 95%);
  padding: 1rem;
}
p.content{
  padding: 0.7rem;
  font-size: 0.7rem;
  color: hsl(350, 50%, 50%);
}
.container h2.title{
  padding: 1rem;
  color: hsl(350, 50%, 20%);
}
footer{
  background: hsl(350, 93%, 88%);
  padding: 1rem;
}
input[type=button] {
  color:hsl(350, 50%, 20%);
  padding: 0.3rem;
  font-size: 1rem;
}
</style>

```

CSS3 HSL (Hue, Saturation, Lightness) notation is used to define colors. The hue is the angle on a color circle and the example uses 350 for red. All page colors use differing variations by changing the saturation (percentage of color) and lightness (percentage).

Using HSL allows us to easily try different main colors for the theme by only changing the hue value. We could also use a CSS variable for the hue value and switch the color theme by changing a single value in the stylesheet or dynamically altering it with JavaScript.

This is a screen shot of the page:



Live Code

Check out the related pen: See the Pen [CSS Theming 1](#).

Let's use a CSS variable for holding the value of the hue of all

colors in the page. Add a global CSS variable in the `:root` selector at the top of the `<style>` tag:

```
:root{
  --main-hue : 350;
}
```

Next, we replace all hard-coded `350` values in `hsl()` colors with the `--main-hue` variable. For example, this is the `nav` selector:

```
nav{
  background: hsl(var(--main-hue) , 50%, 50%);
  padding: 1.3rem;
  color: hsl(var(--main-hue), 50%, 10%);
}
```

Now if you want to specify any color other than red, you can just assign the corresponding value to `--main-hue`. These are some examples:

```
:root{
  --red-hue: 360;
  --blue-hue: 240;
  --green-hue: 120;
  --main-hue : var(--red-hue);
}
```

We are defining three custom properties for red, blue and green, then assigning the `--red-hue` variable to `--main-hue`.

This a screen shot of pages with different values for `--main-hue`:

| Title | Title | Title |
|---|---|---|
| <div>Light/Dark</div> <h2>What is Lorem Ipsum?</h2> <p>Lorem ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.</p> | <div>Light/Dark</div> <h2>What is Lorem Ipsum?</h2> <p>Lorem ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.</p> | <div>Light/Dark</div> <h2>What is Lorem Ipsum?</h2> <p>Lorem ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.</p> |
| Copyright 2018 | Copyright 2018 | Copyright 2018 |

CSS custom properties offer a couple of benefits:

- A value can be defined in a single place.
- That value can be named appropriately to aid maintenance.
- The value can be dynamically altered using JavaScript. For example, the `--main-hue` can be set to any value between 0 and 360.

Using JavaScript to dynamically set the value of `--main-hue` from a set of predefined values or user submitted value for hue (it should be between 0 and 360) we can provide the user with many possibilities for colored themes.

The following line of code will set the value of `--main-hue` to 240 (blue):

```
document.documentElement.style.setProperty('--main-hue', 240);
```

Live Code

Check out the following pen, which shows a full example that allows you to dynamically switch between red, blue and green colored themes: See the Pen [CSS Theming 2](#).

This is a screen shot of the page from the pen:



Adding a CSS Dark Theme

Now let's provide a dark theme for this page. For more control over the colors of different entities, we need to add more variables.

Going through the page's styles, we can replace all HSL colors in different selectors with variables after defining custom properties for the corresponding colors in `:root`:

```
:root{
  /*...*/
  --nav-bg-color: hsl(var(--main-hue) , 50%, 50%);
  --nav-text-color: hsl(var(--main-hue), 50%,
10%);
  --container-bg-color: hsl(var(--main-hue) , 50%,
95%);
  --content-text-color: hsl(var(--main-hue) , 50%,
50%);
  --title-color: hsl(var(--main-hue) , 50%, 20%);
  --footer-bg-color: hsl(var(--main-hue) , 93%,
88%);
  --button-text-color: hsl(var(--main-hue), 50%,
20%);
}
```

Appropriate names for the custom properties have been used. For example, `--nav-bg-color` refers to *the color of the nav background*, while `--nav-text-color` refers to *the color of nav foreground/text*.

Now duplicate the `:root` selector with its content, but add a theme attribute with a *dark* value:

```
:root[theme='dark']{
  /*...*/
}
```

This theme will be activated if a *theme* attribute with a *dark* value is added to the `<html>` element.

We can now play with the values of these variables manually, by reducing the lightness value of the HSL colors to provide a dark theme, or we can use other techniques such as CSS filters like `invert()` and `brightness()`, which are commonly used to adjust the rendering of images but can also be used with any other element.

Add the following code to `:root[theme='dark']`:

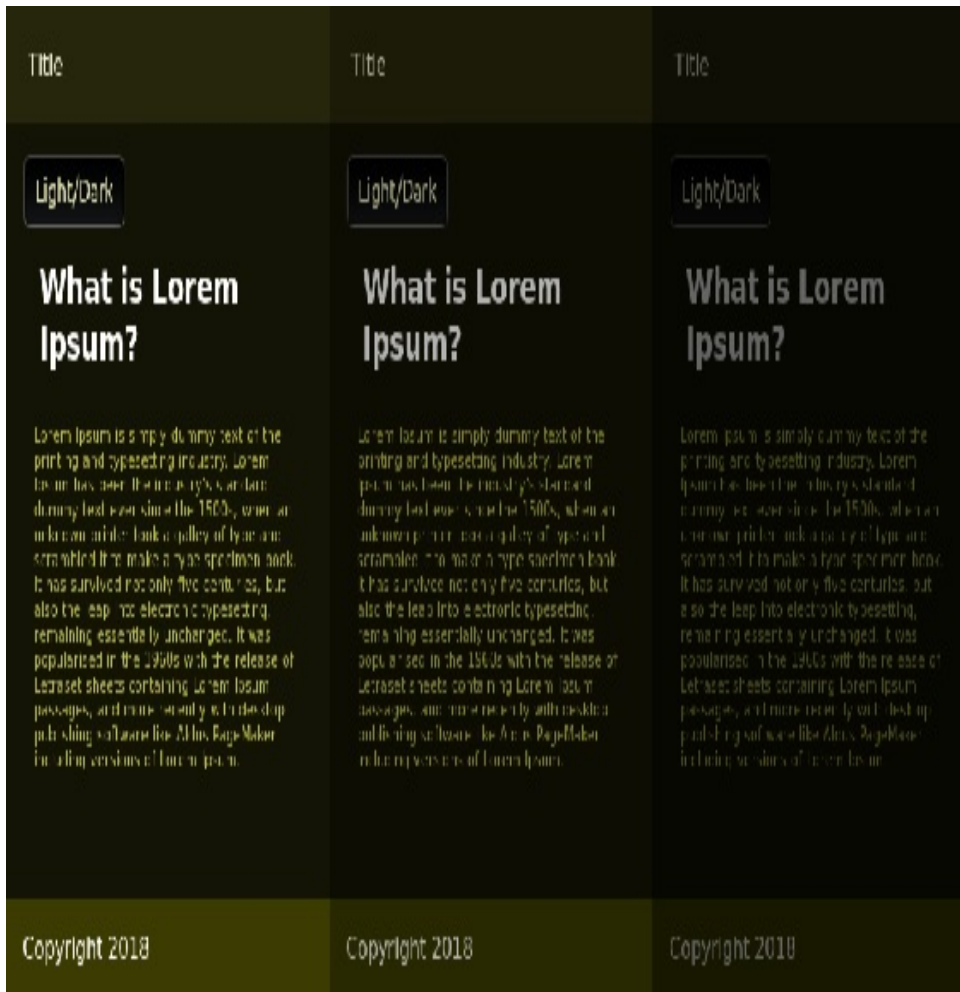
```
:root[theme='dark'] {
  --red-hue: 360;
  --blue-hue: 240;
  --green-hue: 120;
  --main-hue: var(--blue-hue);
  --nav-bg-color: hsl(var(--main-hue), 50%, 90%);
  --nav-text-color: hsl(var(--main-hue), 50%,
10%);
  --container-bg-color: hsl(var(--main-hue), 50%,
95%);
  --content-text-color: hsl(var(--main-hue), 50%,
50%);
  --title-color: hsl(--main-hue, 50%, 20%);
  --footer-bg-color: hsl(var(--main-hue), 93%,
88%);
  --button-text-color: hsl(var(--main-hue), 50%,
20%);
  filter: invert(1) brightness(0.6);
}
```

The `invert()` filter inverts all the colors in the selected elements (every element in this case). The value of inversion can be specified in percentage or number. A value of **100%** or **1** will completely invert the hue, saturation, and lightness values of the element.

The `brightness()` filter makes an element brighter or darker. A value of `0` results in a completely dark element.

The `invert()` filter makes some elements very bright. These are toned down by setting `brightness(0.6)`.

A dark theme with different degrees of darkness:



Switching Themes with JavaScript

Let's now use JavaScript to switch between the dark and light themes when a user clicks the *Dark/Light* button. In `index.html` add an inline `<script>` before the closing `</body>` with the following code:

```
const toggleBtn = document.querySelector("#toggle-theme");
toggleBtn.addEventListener('click', e => {
  console.log("Switching theme");

  if(document.documentElement.hasAttribute('theme'))
  {
    document.documentElement.removeAttribute('theme');
  }
  else{
    document.documentElement.setAttribute('theme',
'dark');
  }
});
```

Document.documentElement refers to the the root DOM Element of the document — that is, `<html>`. This code checks for the existence of a *theme* attribute using the `.hasAttribute()` method and adds the attribute with a *dark* value if it doesn't exist, causing the switch to the dark theme. Otherwise, it removes the attribute, which results in switching to the light theme.

Changing CSS Custom Properties with JavaScript

Using JavaScript, we can access custom properties and change their values dynamically. In our example, we hard-coded the brightness value, but it could be dynamically changed. First,

add a slider input in the HTML page next to the *dark/light* button:

```
<input type="range" id="darknessSlider"
name="darkness" value="1" min="0.3"
max="1" step="0.1" />
```

The slider starts at **1** and allows the user to reduce it to **0.3** in steps of **0.1**.

Next, add a custom property for the darkness amount with an initial value of **1** in `:root[theme='dark']`:

```
:root[theme='dark']{
  /*...*/
  --theme-darkness: 1;
}
```

Change the `brightness` filter to this custom property instead of the hard-coded value:

```
filter: invert(1) brightness(var(--theme-
darkness));
```

Finally, add the following code to synchronize the value of `--theme-darkness` with the slider value:

```
const darknessSlider =
document.querySelector("#darknessSlider");
darknessSlider.addEventListener('change', (e)=>{
  const val = darknessSlider.value
  document.documentElement.style.setProperty('--
theme-darkness', val);
});
```

We're listening for the change event of the slider and setting the value of `--theme-darkness` accordingly using the `setProperty()` method.

We can also apply the `brightness` filter to the light theme. Add the `--theme-darkness` custom property in the top of the `:root` selector:

```
:root{  
  /*...*/  
  --theme-darkness: 1;  
}
```

Then add a `brightness` filter in the bottom of the same selector:

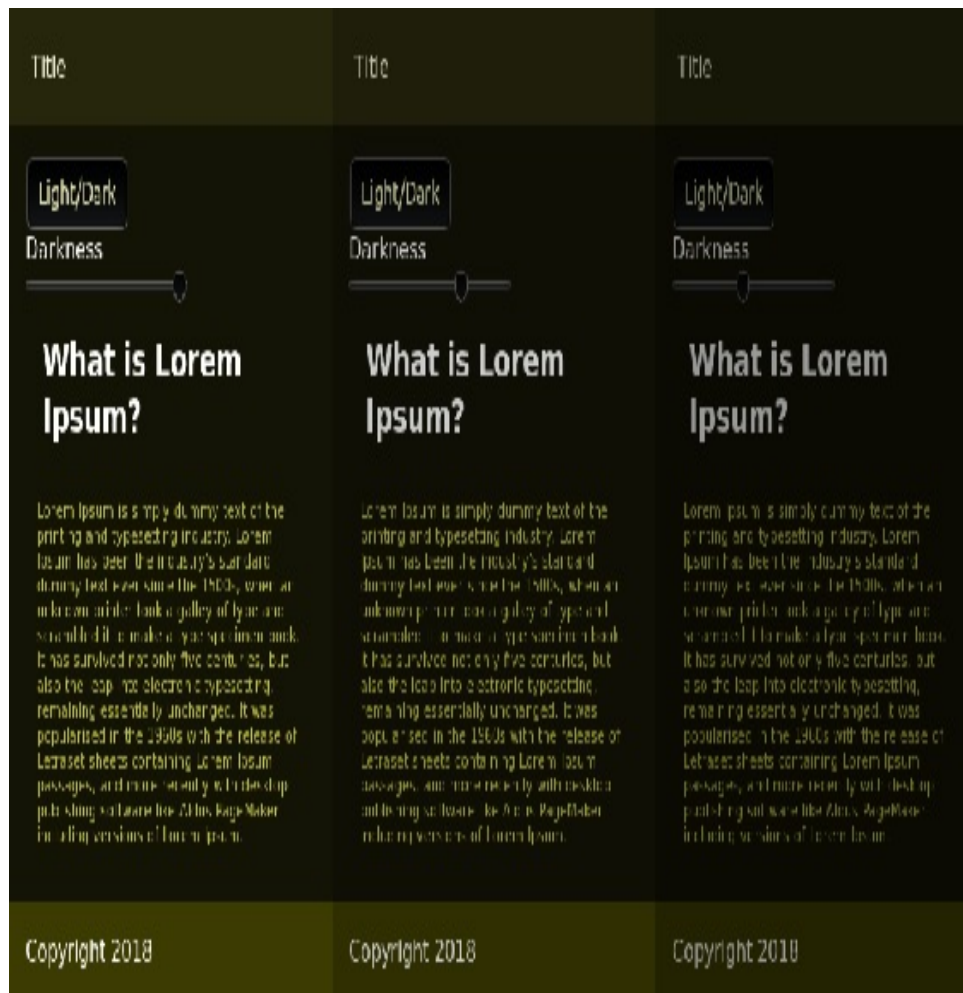
```
:root{  
  /*...*/  
  filter: brightness(var(--theme-darkness));  
}
```

Live Code

You can find the code of this example in the following pen:
[See the Pen CSS Theming.](#)

You can find the code of this example in the following pen:

Here's a screenshot of the dark theme of the final example:



Here's a screenshot of the light theme:



Conclusion

In this tutorial, we've seen how to use CSS custom properties to create themes and switch dynamically between them. We've used the HSL color scheme, which allows us to specify colors with hue, saturation and lightness values and CSS filters (invert and brightness) to create a dark version of a light theme.

Here are some links for further reading if you want to learn more about CSS theming:

- Using CSS custom properties (variables)
- HSL and HSV
- CSS Custom Properties and Theming
- Dark theme in a day