O'REILLY®

"This is a new generation of CSS books, for a new generation of CSS. Nobody is better at making sense of this new CSS than Lea Verou—among the handful of truly amazing coders I've known."
—**Jeffrey Zeldman, author, *Designing With Web Standards***

CSS SECRETS

BETTER SOLUTIONS TO EVERYDAY WEB DESIGN PROBLEMS

LEA VEROU

Free Sampler

FOREWORD BY ERIC A. MEYER

CSS SECRETS

BETTER SOLUTIONS TO EVERYDAY WEB DESIGN PROBLEMS

In this practical guide, CSS expert Lea Verou provides 47 undocumented techniques and tips to help intermediate-to-advanced CSS developers devise elegant solutions to a wide range of everyday web design problems.

Rather than focus on design, *CSS Secrets* shows you how to solve problems with code. You'll learn how to apply Lea's analytical approach to practically every CSS problem you face to attain DRY, maintainable, flexible, lightweight, and standards-compliant results.

Inspired by her popular talks at over 60 international web development conferences, Lea Verou provides a wealth of information for topics including:

- **Background & Borders**
- **Shapes**
- **Visual Effects**
- **Typography**
- **User Experience**
- **Structure & Layout**
- **Transitions & Animations**

"Lea Verou's encyclopaedic mind is one of a kind, but thanks to this generous book, you too can get an insight into what it's like to wield CSS to do just about anything you can think of. Even if you think you know CSS inside-out, I guarantee that there are still secrets in this book waiting to be revealed."

—Jeremy Keith
Shepherd of Unknown Futures, Clearleft

"If you want the inside scoop on fascinating CSS techniques, smart best practices, and some flat-out brilliance, don't hesitate—read this book. I loved it!"

—Eric A. Meyer

"*CSS Secrets* is an instant classic—so many wonderful tips and tricks you can use right away to enhance your UX designs!"

—Christopher Schmitt
Author of *CSS Cookbook*

"Lea is an exceedingly clever coder. This book is absolutely packed with clever and useful ideas, even for people who know CSS well. Even better, you'll feel more clever in your work as this book encourages pushing beyond the obvious."

—Chris Coyier
CodePen

CSS/Web Development

US $39.99 CAN $45.99

ISBN: 978-1-449-37263-7

53999

9 781449 372637

O'REILLY®

oreilly.com

About This Excerpt

CSS has come a long way. What may have seemed like a niche topic a few years ago has evolved into an incredibly useful and powerful tool for web designers and developers.

This practical excerpt is a collection from Lea Verou's *CSS Secrets*—a hands-on, practical guide that provides 47 undocumented techniques and tips to help intermediate-to-advanced CSS developers devise elegant solutions to a wide range of everyday web design problems.

Rather than focus on design, *CSS Secrets* shows you how to **solve problems with code**. You'll learn how to apply Lea's analytical approach to practically every CSS problem you face to attain DRY, maintainable, flexible, lightweight, and standards-compliant results.

Inspired by her popular talks at over 60 international web development conferences, Lea Verou provides a wealth of information for topics including:

- Background & Borders
- Shapes
- Visual effects
- Typography
- User Experience
- Structure & Layout
- Transitions & Animations

Throughout these pages, you'll get a taste of the depth and complexity of tips, and whet your appetite for the full book. Purchase your copy **here** *(oreil.ly/1LBef9s)*.

Praise for *CSS Secrets*

“*This is a new generation of CSS books, for a new generation of CSS. No longer a simple language tied to complicated browser hacks and workarounds, CSS is now a richly powerful and deeply complex ecosystem of over 80 W3C specifications. Nobody is better at making sense of this new CSS, and of providing design principles that help you solve problems with it, than Lea Verou—among the handful of truly amazing coders I've known.*”

— Jeffrey Zeldman
author, Designing with Web Standards

“*Lea Verou's encyclopaedic mind is one of a kind, but thanks to this generous book, you too can get an insight into what it's like to wield CSS to do just about anything you can think of. Even if you think you know CSS inside-out, I guarantee that there are still secrets in this book waiting to be revealed.*”

— Jeremy Keith
Shepherd of Unknown Futures, Clearleft

“*If you want the inside scoop on fascinating CSS techniques, smart best practices, and some flat-out brilliance, don't hesitate—read this book. I loved it!*”

— Eric A. Meyer

“ *Lea is an exceedingly clever coder. This book is absolutely packed with clever and useful ideas, even for people who know CSS well. Even better, you’ll feel more clever in your work as this book encourages pushing beyond the obvious.”*

— Chris Coyier
CodePen

“ *CSS Secrets is an instant classic—so many wonderful tips and tricks you can use right away to enhance your UX designs!”*

— Christopher Schmitt
author of CSS Cookbook

“ *There aren’t many books that provide as many practical techniques as Lea Verou’s CSS Secrets. Filled with dozens of solutions to common design problems, the book is a truly valuable collection of smart tips and tricks for getting things done well, and fast. Worth reading, even if you think that you know the ins and outs of CSS!”*

— Vitaly Friedman
cofounder and editor-in-chief of Smashing Magazine

“ *Without fail, whenever I read something written by Lea Verou, I manage to learn something new. CSS Secrets is no different. The book is broken down into easy-to-digest chunks filled with lots of juicy bits of knowledge. While some of the book is very forward looking, there is plenty that I’ve been able to take away and apply to my own projects right away.”*

— Jonathan Snook
web designer and developer

“ *Lea’s book is fantastic. She bends and contorts CSS to do things I’m pretty sure even the spec authors never imagined! You will learn multiple ways of accomplishing each graphic effect by trying out the techniques she walks through in each chapter. Later, in your work, you’ll find yourself saying, “hmm, that thing Lea did will work perfectly here!” Before you know it, your site is almost image free because your graphics are all in easy to maintain CSS components. What’s more, her techniques are fun, walking the line between practical and improbable!”*

— Nicole Sullivan
Principal Software Engineer, creator of OOCSS

“Lea Verou’s CSS Secrets is useful not so much as a collection of CSS tips, but as a textbook on how to solve problems with CSS. Her in-depth explanation of the thought process behind each secret will teach you how to create your own solutions to CSS problems. And don’t miss the Introduction, which contains some must-read CSS best practices.”

— Elika J. Etemad (aka fantasai)
W3C CSS Working Group Invited Expert

“Lea’s presentations have long been must-see events at web development conferences around the world. A distillation of her years of experience, CSS Secrets provides elegant solutions for thorny web design issues, while also—and more importantly—showing how to solve problems in CSS. It’s an absolute must-read for every frontend designer and developer.”

— Dudley Storey
designer, developer, writer, web education specialist

“I thought I had a pretty advanced understanding of CSS, then I read Lea Verou’s book. If you want to take your CSS knowledge to the next level, this is a must-own.”

— Ryan Seddon
Team Lead, Zendesk

“CSS Secrets is by far the most technical book that I have ever read on the topic. Lea has managed to push the boundaries of a language as simple as CSS so far that you will not be able to distinguish this from magic. Definitely not a beginner’s read; it’s heavily recommended to anyone thinking they know CSS all too well.”

— Hugo Giraudel
frontend developer, Edenspiekermann

“I often think that CSS can seem a bit like magic: a few rules can transform your web pages from blah to beautiful. In CSS Secrets, Lea takes the magic to a whole new level. She is a master magician of CSS, and we get to explore that magical world along with her. I can’t count how many times I said out loud while reading this book, “That’s so cool!” The only trouble with CSS Secrets is that after reading it, I want to stop everything else I’m doing and play with CSS all day.”

— Elisabeth Robson
cofounder of WickedlySmart.com and coauthor of Head First JavaScript Programming

“*CSS Secrets is a book that all web developers should have in their library. Using the information it contains you’ll learn numerous hints and tips to make CSS perform tasks you never thought possible. I was astonished at how often the author came up with simple and elegant lateral thinking solutions to problems that had bugged me for years.”*

— Robin Nixon
web developer, online instructor, and author of several books on CSS

“*As a master designer and programmer, Lea Verou’s book is as beautiful and as well thought out as her code. Whether you’re fairly new to CSS, or well versed in the intricacies of CSS3, this book has something for everyone.”*

— Estelle Weyl
Open Web Evangelist and coauthor of CSS: The Definitive Guide

CSS SECRETS

BETTER SOLUTIONS TO EVERYDAY WEB DESIGN PROBLEMS

LEA VEROU

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

CSS Secrets

by Lea Verou

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Mary Treseler and Meg Foley
Production Editor: Kara Ebrahim
Copyeditor: Jasmine Kwityn
Indexer: WordCo Indexing Services
Proofreader: Charles Roumeliotis
Interior Designer: Lea Verou
Cover Designer: Monica Kamsvaag
Illustrator: Lea Verou

See ***http://www.oreilly.com/catalog/errata.csp?isbn=0636920031123*** for release details.

Print History: First Edition, June 2015

Revision History for the First Edition:

2015-06-03	First Release
2015-07-17	Second Release

ISBN: 978-1-4493-7263-7
[TI]

In loving memory of
my mother & best friend, Maria Verou (1952–2013),
who left this world way too early.

Table of Contents

Foreword	xv
Preface	xvii
Words of thanks	xix
Making of	xxii
About this book	xxiv
CHAPTER 1	
Introduction	1
Web standards: friend or foe?	2
CSS coding tips	9
CHAPTER 2	
Backgrounds & Borders	23
1 Translucent borders	24
2 Multiple borders	28
3 Flexible background positioning	32
4 Inner rounding	36
5 Striped backgrounds	40
6 Complex background patterns	50
7 (Pseudo)random backgrounds	62
8 Continuous image borders	68
CHAPTER 3	
Shapes	75
9 Flexible ellipses	76
10 Parallelograms	84
11 Diamond images	90
12 Cutout corners	96
13 Trapezoid tabs	108
14 Simple pie charts	114
CHAPTER 4	
Visual Effects	129
15 One-sided shadows	130
16 Irregular drop shadows	134
17 Color tinting	138

18 Frosted glass effect 146

19 Folded corner effect 156

CHAPTER 5
Typography 167

20 Hyphenation 168

21 Inserting line breaks 172

22 Zebra-striped text lines 178

23 Adjusting tab width 182

24 Ligatures 184

25 Fancy ampersands 188

26 Custom underlines 194

27 Realistic text effects 200

28 Circular text 210

CHAPTER 6
User Experience 217

29 Picking the right cursor 218

30 Extending the clickable area 224

31 Custom checkboxes 228

32 De-emphasize by dimming 234

33 De-emphasize by blurring 240

34 Scrolling hints 244

35 Interactive image comparison 250

CHAPTER 7
Structure & Layout 261

36 Intrinsic sizing 262

37 Taming table column widths 266

38 Styling by sibling count 270

39 Fluid background, fixed content 276

40 Vertical centering 280

41 Sticky footers 288

CHAPTER 8
Transitions & Animations 293

42 Elastic transitions 294

43 Frame-by-frame animations 308

44 Blinking 314

45 Typing animation 320

46 Smooth state animations 328

47 Animation along a circular path 334

Index 347

Secrets by Specification

CSS Animations

w3.org/TR/css-animations

42	Elastic transitions	294
43	Frame-by-frame animations	308
44	Blinking	314
45	Typing animation	320
46	Smooth state animations	328
47	Animation along a circular path	334

CSS Backgrounds & Borders

w3.org/TR/css-backgrounds

1	Translucent borders	24
2	Multiple borders	28
3	Flexible background positioning	32
4	Inner rounding	36
5	Striped backgrounds	40
6	Complex background patterns	50
7	(Pseudo)random backgrounds	62
8	Continuous image borders	68
9	Flexible ellipses	76
12	Cutout corners	96
14	Simple pie charts	114
15	One-sided shadows	130
19	Folded corner effect	156
22	Zebra-striped text lines	178
26	Custom underlines	194
30	Extending the clickable area	224
32	De-emphasize by dimming	234
34	Scrolling hints	244
35	Interactive image comparison	250

CSS Backgrounds & Borders Level 4

dev.w3.org/csswg/css-backgrounds-4

12	Cutout corners	96

CSS Basic User Interface
w3.org/TR/css3-ui

2 Multiple borders 28

4 Inner rounding 36

35 Interactive image comparison 250

CSS Box Alignment
w3.org/TR/css-align

40 Vertical centering 280

CSS Flexible Box Layout
w3.org/TR/css-flexbox

40 Vertical centering 280

41 Sticky footers 288

CSS Fonts
w3.org/TR/css-fonts

24 Ligatures 184

25 Fancy ampersands 188

CSS Image Values
w3.org/TR/css-images

5 Striped backgrounds 40

6 Complex background patterns 50

7 (Pseudo)random backgrounds 62

8 Continuous image borders 68

12 Cutout corners 96

14 Simple pie charts 114

19 Folded corner effect 156

22 Zebra-striped text lines 178

26 Custom underlines 194

34 Scrolling hints 244

35 Interactive image comparison 250

CSS Image Values Level 4
w3.org/TR/css4-images

5 Striped backgrounds 40

6 Complex background patterns 50

14 Simple pie charts 114

CSS Intrinsic & Extrinsic Sizing
w3.org/TR/css3-sizing

36 Intrinsic sizing 262

CSS Masking
w3.org/TR/css-masking

11 Diamond images 90

12 Cutout corners 96

CSS Text
w3.org/TR/css-text

20 Hyphenation 168

23 Adjusting tab width 182

CSS Text Level 4
dev.w3.org/csswg/css-text-4

20 Hyphenation 168

CSS Text Decoration
w3.org/TR/css-text-decor

26 Custom underlines 194

27 Realistic text effects 200

CSS Transforms
w3.org/TR/css-transforms

10	Parallelograms	84
11	Diamond images	90
12	Cutout corners	96
13	Trapezoid tabs	108
14	Simple pie charts	114
19	Folded corner effect	156
35	Interactive image comparison	250
40	Vertical centering	280
47	Animation along a circular path	334

CSS Transitions
w3.org/TR/css-transitions

11	Diamond images	90
12	Cutout corners	96
17	Color tinting	138
33	De-emphasize by blurring	240
42	Elastic transitions	294

CSS Values & Units
w3.org/TR/css-values

3	Flexible background positioning	32
32	De-emphasize by dimming	234
40	Vertical centering	280
41	Sticky footers	288
45	Typing animation	320

Compositing and Blending
w3.org/TR/compositing

17	Color tinting	138
35	Interactive image comparison	250

Filter Effects
w3.org/TR/filter-effects

16	Irregular drop shadows	134
17	Color tinting	138
18	Frosted glass effect	146
33	De-emphasize by blurring	240
35	Interactive image comparison	250

Fullscreen API
fullscreen.spec.whatwg.org

32	De-emphasize by dimming	234

Scalable Vector Graphics
w3.org/TR/SVG

6	Complex background patterns	50
14	Simple pie charts	114
28	Circular text	210

Selectors
w3.org/TR/selectors

31	Custom checkboxes	228
38	Styling by sibling count	270

Backgrounds & Borders 2

8 Continuous image borders

Prerequisites

CSS gradients, basic **`border-image`**, the "Striped backgrounds" secret on page 40, basic CSS animations

The problem

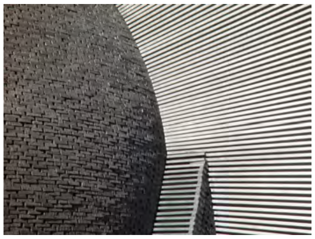


FIGURE 2.56

Our stone art image, used throughout this secret

Sometimes we want to apply a pattern or image **not as a background, but as a border**. For example, check out **Figure 2.57** for an element with a decorative border that is basically an image clipped to the border area. In addition, we want the image to resize to cover the entire border area regardless of the dimensions of our element. How would we attempt to do something like this with CSS?

At this point, there might be a very loud voice in your head screaming, *"**`border-image`**, **`border-image`**, we can use **`border-image`**, that's not a problem anymore!!!11."* **Not so fast, young padawan.** Recall how **`border-image`** actually works: it's basically **9-slice scaling**. You slice the image into nine boxes and apply them to the corners and sides accordingly. **Figure 2.58** offers a visual reminder of how this works.

How could we possibly slice our image via **`border-image`** to create the example in **Figure 2.57**? Even if we meticulously get it right for specific dimensions and border width, it wouldn't adjust properly for different ones. The issue is that there is no specific part of the image that we want to be at the corners; the part of the image shown in the corner squares changes with the dimensions of the element and border width. If you try it for a bit, you will likely also conclude that this is not possible with **`border-image`**. But then what can we do?

The easiest way is to use two HTML elements: one using a background with our stone art image, and one with a white background covering it for our content area:

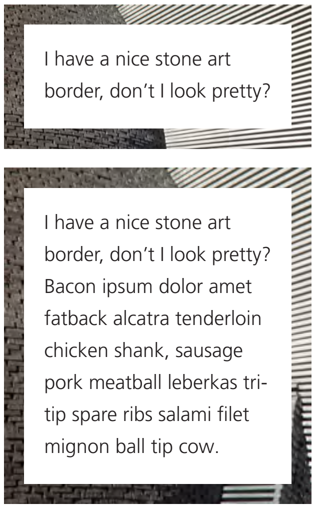

I have a nice stone art border, don't I look pretty?
I have a nice stone art border, don't I look pretty? Bacon ipsum dolor amet fatback alcatra tenderloin chicken shank, sausage pork meatball leberkas tri-tip spare ribs salami filet mignon ball tip cow.

FIGURE 2.57
Our image used as a border with varying heights

HTML

```html
<div class="something-meaningful"><div>
    I have a nice stone art border,
    don't I look pretty?
</div></div>
```

```css
.something-meaningful {
    background: url(stone-art.jpg);
    background-size: cover;
    padding: 1em;
}

.something-meaningful > div {
    background: white;
    padding: 1em;
}
```

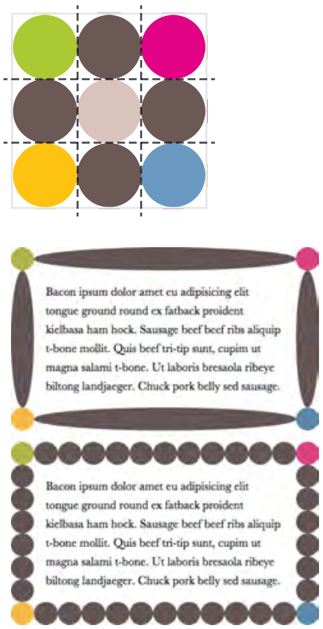
This works fine to create the "border" shown in **Figure 2.57**, but it requires an extra HTML element. This is suboptimal: not only does it mix presentation and styling, but modifying the HTML is simply not an option in certain cases. Can we do this with only one element?

The solution

Thanks to CSS gradients and the background extensions introduced in **Backgrounds & Borders Level 3** *(w3.org/TR/css3-background)*, we can achieve the exact same effect with only one element. The main idea is to use **a second background of pure white, covering the stone art image**. However, to make the second image show through the border area, we should apply different values of `background-clip` to them. One last thing is that we can only have a background color on the last layer, so we need to fake the white via a CSS gradient from white to white.

This is how our first attempt to apply this idea might look:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white),
            url(stone-art.jpg);
background-size: cover;
background-clip: padding-box, border-box;
```


Bacon ipsum dolor amet cu adipisicing elit tongue ground round ex fatback proident kielbasa ham hock. Sausage beef beef ribs aliquip t-bone mollit. Quis beef tri-tip sunt, cupim ut magna salami t-bone. Ut laboris bresaola ribeye biltong landjaeger. Chuck pork belly sed sausage.

Bacon ipsum dolor amet cu adipisicing elit tongue ground round ex fatback proident kielbasa ham hock. Sausage beef beef ribs aliquip t-bone mollit. Quis beef tri-tip sunt, cupim ut magna salami t-bone. Ut laboris bresaola ribeye biltong landjaeger. Chuck pork belly sed sausage.

FIGURE 2.58

A quick primer on `border-image`
Top: Our sliced image; the dashed lines indicate its slicing
Middle: `border-image: 33.34% url(…) stretch;`
Bottom: `border-image: 33.34% url(…) round;`
Play with the code at **`play.csssecrets.io/border-image`**

As we can see in **Figure 2.59**, the result is very close to what we wanted, but there is some weird repetition. The reason is that the default `background-origin` is `padding-box`, and thus, the image is sized based on the padding box and placed on the 0,0 point on the padding box. The rest is just repetitions of that first background tile. To correct this, we just need to set `background-origin` to `border-box` as well:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white),
            url(stone-art.jpg);
background-size: cover;
background-clip: padding-box, border-box;
```

```
background-origin: border-box;
```

These new properties are also available on the **background** shorthand, which can help us reduce our code significantly here:

```
padding: 1em;
border: 1em solid transparent;
background:
    linear-gradient(white, white) padding-box,
    url(stone-art.jpg) border-box 0 / cover;
```

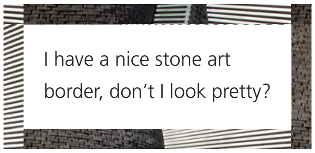


FIGURE 2.59
Our first attempt is very close to what we wanted

▶ **PLAY!** play.csssecrets.io/**continuous-image-borders**

Of course, we can use the same technique with **gradient-based patterns**. For example, take a look at the following code, which generates a **vintage envelope themed border**:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white) padding-box,
            repeating-linear-gradient(-45deg,
              red 0, red 12.5%,
              transparent 0, transparent 25%,
              #58a 0, #58a 37.5%,
              transparent 0, transparent 50%)
             0 / 5em 5em;
```

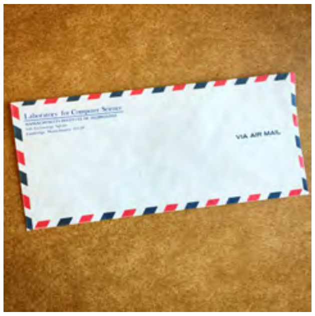


FIGURE 2.60
An actual vintage envelope

You can see the result in **Figure 2.61**. You can easily change the width of the stripes via the **background-size** and the thickness of the border via the **border** declaration. Unlike our stone art border example, this effect **is doable with border-image** too:

TIP! To see these issues in action, visit **play.csssecrets.io/vintage-envelope-border-image** and experiment with changing values.

```
padding: 1em;
border: 16px solid transparent;
border-image: 16 repeating-linear-gradient(-45deg,
                   red 0, red 1em,
                   transparent 0, transparent 2em,
                   #58a 0, #58a 3em,
                   transparent 0, transparent 4em);
```

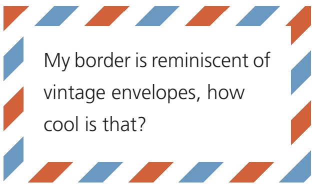

My border is reminiscent of vintage envelopes, how cool is that?

FIGURE 2.61
Our "vintage envelope" border

However, the **`border-image`** approach has several issues:

- We need to update **`border-image-slice`** every time we change the **`border-width`** and make them match.
- Because we cannot use **ems** in **`border-image-slice`**, we are **restricted to only pixels** for the border thickness.
- The stripe thickness needs to be encoded in the color stop positions, so we need to make four edits to change it.

▶ PLAY! play.csssecrets.io/**vintage-envelope**

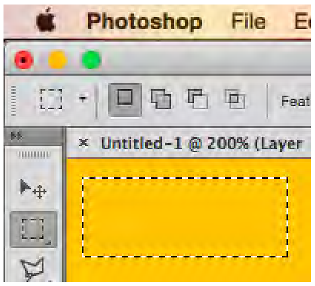


FIGURE 2.62
Marching ants are also used in Adobe Photoshop to indicate area selection

Another fun application of this technique is using it to make **marching ants borders**! Marching ants borders are dashed borders that seem to scroll like marching ants (if you imagine that the dashes are ants). These are incredibly common in GUIs; image editors use them almost always to indicate area selection (**Figure 2.62**).

To create marching ants, we are going to use a variation of the "vintage envelope" effect. We will convert the stripes to just black and white, reduce the width of the border to **`1px`** (notice how the stripes now turn to a dashed border?), and change the **`background-size`** to something appropriate. Then, we animate the **`background-position`** to **`100%`** to make it scroll:

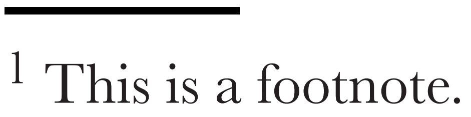
```
@keyframes ants { to { background-position: 100% } }

.marching-ants {
    padding: 1em;
    border: 1px solid transparent;
    background:
        linear-gradient(white, white) padding-box,
        repeating-linear-gradient(-45deg,
          black 0, black 25%, white 0, white 50%
        ) 0 / .6em .6em;
    animation: ants 12s linear infinite;
}
```

You can see a still of the result in **Figure 2.63**. Obviously, this is not only useful for marching ants, but also for **creating all sorts of custom dashed borders, with different color dashes and custom dash-gap width**.

Currently, the only way to achieve a similar effect via `border-image` is to use an animated GIF for `border-image-source`, as shown in **`chrisdanford.com/blog/2014/04/28/marching-ants-animated-selection-rectangle-in-css`**. When browsers start supporting gradient interpolation, we will also be able to do it with gradients, though in a messy, WET way.

FIGURE 2.63
It's not really possible to show marching ants in a book (a still just looks like dashed borders); visit the live example—it's fun!

▶ **PLAY!** `play.csssecrets.io/marching-ants`



1 This is a footnote.

FIGURE 2.64
Top border clipping, to mimic traditional footnotes

However, `border-image` can also be quite powerful, and even more when used with gradients. For example, assume we want a clipped top border, like the one commonly used in footnotes. All it takes is `border-image` and a vertical gradient, with the clipping length hardcoded. The border width is controlled by …`border-width`. The code would look like this:

```
border-top: .2em solid transparent;
border-image: 100% 0 0 linear-gradient(90deg,
                          currentColor 4em,
                          transparent 0);
padding-top: 1em;
```

The result is identical to **Figure 2.64**. In addition, because we specified everything in `em`s, the effect will adjust with `font-size` changes, and because we used `currentColor`, it will also adapt to `color` changes (assuming we want the border to be the same color as the text).

▶ **PLAY!** play.csssecrets.io/**footnote**

RELATED SPECS

- **CSS Backgrounds & Borders**
  w3.org/TR/css-backgrounds
- **CSS Image Values**
  w3.org/TR/css-images

Shapes 3

12 Cutout corners

Prerequisites

CSS gradients, **`background-size`**, the "Striped backgrounds" secret on page 40

The problem

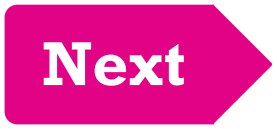


FIGURE 3.23
A button with cutout corners, creating an arrow shape that emphasizes its meaning

Cutting corners is not just a way to save money, but also a rather popular style in both print and web design. It usually involves cutting out one or more of an element's corners in a 45° angle (also known as *beveled corners*). Especially lately, with flat design winning over skeuomorphism, there has been an increase in the popularity of this effect. When the cutout corners are only on one side and occupy 50% of the element's height each, it creates an arrow shape that is very popular for buttons and breadcrumb navigation—see **Figure 3.23**.

However, CSS is still not well equipped for creating this effect in an easy, straightforward one-liner. This leads most authors toward using background images to achieve it, either by obscuring the cutout corners with triangles (when the backdrop is a solid color), or by using one or more images for the entire background, with the corner(s) already cut.

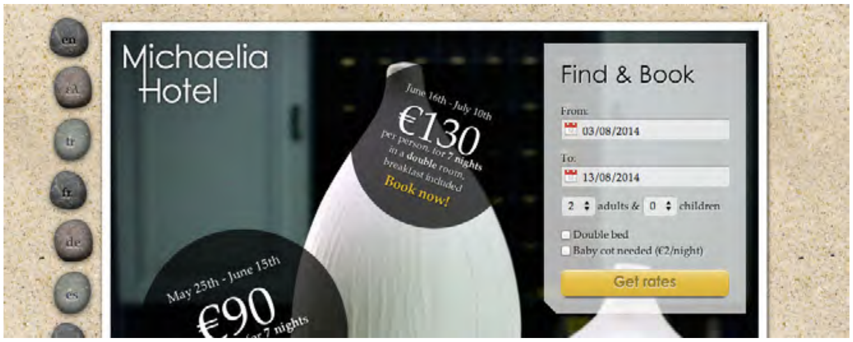


FIGURE 3.24
An example of a website where a cutout corner (bottom-left of the semi-transparent "Find & Book" box) really adds to the design

Such methods are clearly inflexible, difficult to maintain, and add latency, both by increasing HTTP requests and the total filesize of the website. Is there a better way?

The solution

One solution comes in the form of the omnipotent CSS gradients. Let's assume we only want **one cutout corner**, say the bottom-right one. The main trick is to take advantage of the fact that **gradients can accept an angle direction** (e.g., `45deg`) and color stop positions in absolute lengths, both of which are **not affected by changes in the dimensions of the element** the background is on.

Putting it all together, all we need is **one linear gradient**. It would need a transparent color stop for the cutout corner and another color stop in the same position, with the color we want for our background. The CSS looks like this (for a `15px` size corner):

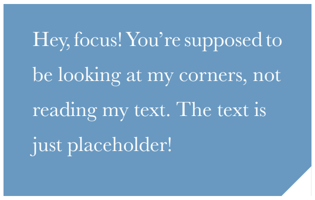

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.25
An element with the bottom right corner cut off, through a simple CSS gradient

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px, #58a 0);
```

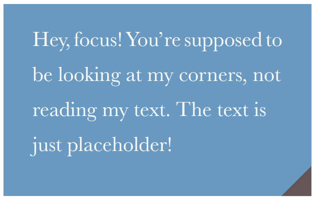
TIP! We are using **separate colors (🟦 #58a and 🟫 #655) for easier debugging**. In practice, both gradients would be the same color.

Simple, wasn't it? You can see the result in **Figure 3.25**. Technically, we don't even need the first declaration. We only included it as a **fall-back**: if CSS gradients are not supported, the second declaration will be dropped, so we still want to get **at least** a solid color background.

Now, let's assume we want **two cutout corners**, say the two bottom ones. We can't achieve this with only one gradient, so we will need two. Our first thought might be something like this:

```css
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px, #58a 0),
    linear-gradient(45deg, transparent 15px, #655 0);
```

However, as you can see in **Figure 3.26**, this doesn't work. By default, both gradients occupy the entire element, so **they obscure each other**. We need to make them smaller, by using `background-size` to make each gradient occupy only **half the element**:

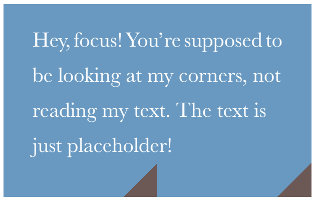


Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.26
Failed attempt to apply the cutout effect to both bottom corners

```css
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px, #58a 0)
        right,
    linear-gradient(45deg, transparent 15px, #655 0)
        left;
background-size: 50% 100%;
```

You can see what happens in **Figure 3.27**. As you can see, although `background-size` was applied, the gradients are **still covering each other**. The reason for this is that we forgot to turn `background-repeat` off, so **each of our backgrounds is repeated twice**. Therefore, our backgrounds are still obscuring each other—by repetition this time. The new code would look like this:



Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.27
`background-size` is not enough

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px, #58a 0)
        right,
    linear-gradient(45deg, transparent 15px, #655 0)
        left;
background-size: 50% 100%;
background-repeat: no-repeat;
```

You can see the result in **Figure 3.28** and verify that—finally—it works! At this point, you are probably able to figure out how to **apply this effect to all four corners**. You will need four gradients, and the code looks like this:

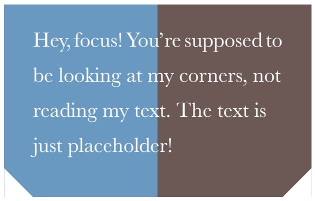

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.28
Our bottom-left and bottom-right cutout corners work now

```
background: #58a;
background:
    linear-gradient(135deg,  transparent 15px, #58a 0)
        top left,
    linear-gradient(-135deg, transparent 15px, #655 0)
        top right,
    linear-gradient(-45deg, transparent 15px, #58a 0)
        bottom right,
    linear-gradient(45deg, transparent 15px, #655 0)
        bottom left;
background-size: 50% 50%;
background-repeat: no-repeat;
```

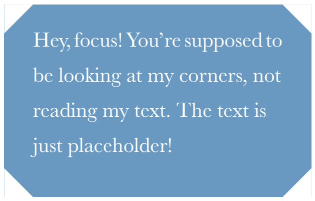

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.29
The effect applied to all four corners, with four gradients

You can see the result in **Figure 3.29**. One issue with the preceding code is that it's not particularly maintainable. It requires **five edits to change the background color** and **four to change the corner size**. A preprocessor mixin could help reduce the repetition. Here's how the code could look with SCSS:

SCSS

```scss
@mixin beveled-corners($bg,
         $tl:0, $tr:$tl, $br:$tl, $bl:$tr) {
    background: $bg;
    background:
        linear-gradient(135deg, transparent $tl, $bg 0)
            top left,
        linear-gradient(225deg, transparent $tr, $bg 0)
            top right,
        linear-gradient(-45deg, transparent $br, $bg 0)
            bottom right,
        linear-gradient(45deg, transparent $bl, $bg 0)
            bottom left;
    background-size: 50% 50%;
    background-repeat: no-repeat;
}
```

Then, where needed, it would be used like this, with 2–5 arguments:

SCSS

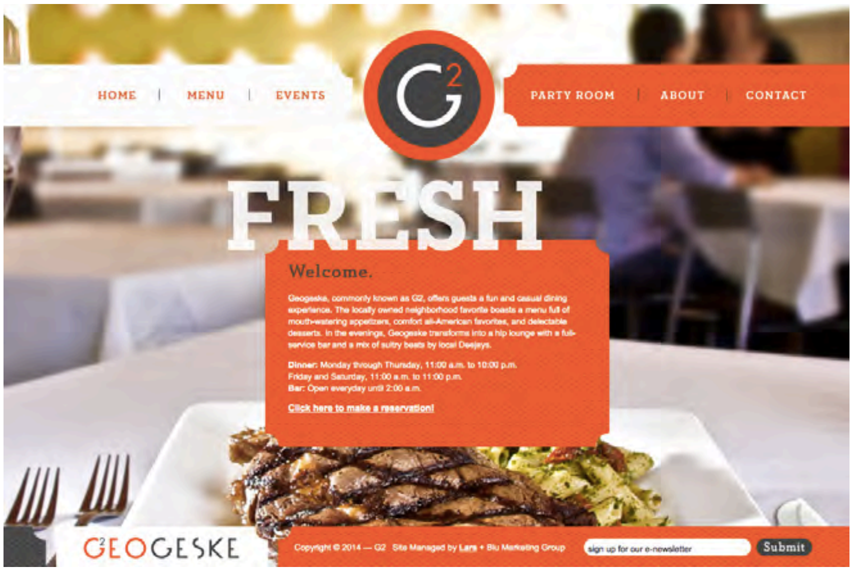
```scss
@include beveled-corners(#58a, 15px, 5px);
```

In this example, the element we will get a **`15px`** top-left and bottom-right cutout corner and a **`5px`** top-right and bottom-left one, similar to how **`border-radius`** works when we provide fewer than four lengths. This is due to the fact that we provided default values for the arguments in our SCSS mixin, and yes, these default values can refer to other arguments as well.

▶ **PLAY!** play.csssecrets.io/**bevel-corners-gradients**

Curved cutout corners

A variation of the gradient method works to create curved cutout corners, an effect many people refer to as "inner border radius," as it looks like an



Welcome.

Geogeske, commonly known as G2, offers guests a fun and casual dining experience. The locally owned neighborhood favorite boasts a menu full of mouth-watering appetizers, comfort all-American favorites, and delectable desserts. In the evenings, Geogeske transforms into a hip lounge with a full-service bar and a mix of sultry beats by local Deejays.

Dinner: Monday through Thursday, 11:00 a.m. to 10:00 p.m.
Friday and Saturday, 11:00 a.m. to 11:00 p.m.
Bar: Open everyday until 2:00 a.m.

Click here to make a reservation!

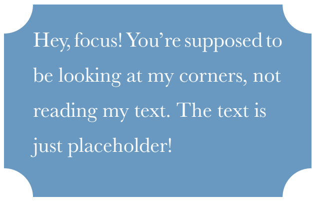
Copyright © 2014 — G2 Site Managed by Lars + Blu Marketing Group

FIGURE 3.30

An excellent use of curved cutout corners in **`g2geogeske.com`**; the designer has made them the central design element, as they are present in the navigation, the content, and even the footer

inverse version of rounded corners. The only difference is using radial gradients instead of linear ones:

```
background: #58a;
background:
    radial-gradient(circle at top left,
             transparent 15px, #58a 0) top left,
    radial-gradient(circle at top right,
             transparent 15px, #58a 0) top right,
    radial-gradient(circle at bottom right,
             transparent 15px, #58a 0) bottom right,
    radial-gradient(circle at bottom left,
             transparent 15px, #58a 0) bottom left;
background-size: 50% 50%;
background-repeat: no-repeat;
```



Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.31

Curved cutout corners, with radial gradients

You can see the result in **Figure 3.31**. Just like in the previous technique, the corner size can be controlled through the color stop positions and a mixin would make the code more maintainable here as well.

Inline SVG & border-image solution

While the gradient-based solution works, it has quite a few issues:

- The code is very **long and repetitive**. In the common case, where we want the same corner size on all four corners, we need to make four edits to modify it. Similarly, we need to make four edits to modify the background color, five counting the fallback.
- It is messy to downright impossible (depending on the browser) to animate between different corner sizes.

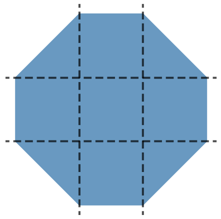


FIGURE 3.32
Our SVG-based border image, with its slicing

Thankfully, there are a couple different methods we could use, depending on our needs. One of them is to use `border-image` with an inline SVG that generates the corners. Given how `border-image` works (if you don't remember, take a look at the quick primer in **Figure 2.58**), can you imagine how our SVG would look?

Because dimensions don't matter (`border-image` takes care of scaling and SVGs scale perfectly regardless of dimensons—ah, the joy of vector graphics!), every measurement could be 1, for easier, shorter, numbers. The corners would be of length 1, and the straight edges would also be 1. The result (zoomed) would look like **Figure 3.32**. The code would look like this:

```
border: 15px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"
         width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
    </svg>');
```

Note that we used a slice size of `1`. This does not mean 1 pixel; it is referring to the coordinate system of the SVG file (hence the lack of units). If we had specified it in percentages, we would need to approximate $\frac{1}{3}$ of the image with something like `33.34%`. Approximating numbers is always risky, because not all browsers use the same level of precision. However, by using units of the coordinate system of the SVG file, we're saved from precision headaches.

The result is shown in **Figure 3.33**. As you can see, our cutout corners are there, but there is no background. We can solve that in two ways: either by specifying a background, or by adding the keyword `fill` to our `border-image` declaration, **so that it doesn't discard the middle slice**. In this case, we are going to go with specifying a background, because **it will also act as a fallback**.

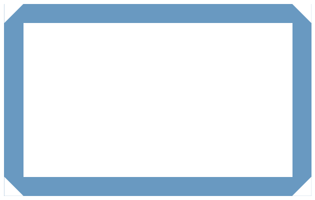


FIGURE 3.33

Applying our SVG on the `border-image` property

In addition, you may have noticed that our corners are **smaller than with the previous technique**, which can be baffling. But we specified a `15px` border width! The reason is that with the gradient, the `15px` was along the *gradient line*, which is perpendicular to the direction of the gradient. The border width, however, is not measured diagonally, but horizontally/vertically. Can you see where this is going? Yup, it's the ubiquitous Pythagorean theorem again, that we also saw in the **"Striped backgrounds" secret on page 40**. **Figure 3.34** should help make things clearer. Long story short, to achieve the same size, we need to use a border width that is $\sqrt{2}$ times larger than the size we would use with the gradient method. In this case, that would be $15 \times \sqrt{2} \approx 21.213203436$ pixels, which is sensible to approximate to `20px`, unless we really, absolutely **need** the diagonal size to be as close to `15px` as possible:

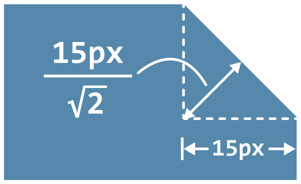


FIGURE 3.34

Specifying a `border-width` of `15px`, results in a (diagonally measured) corner size of $\frac{15}{\sqrt{2}} \approx 10.606601718$, which is why our corners looked smaller

```
border: 20px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"
         width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
    </svg>');
background: #58a;
```

However, as you can see in **Figure 3.35**, this doesn't exactly have the expected result. Where did our laboriously created cutout corners go? Fear not, young padawan, for our corners are still there. You can understand

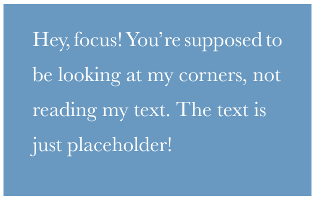

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.35
Where did our nice corners go?!

what's happening if you set the background to a different color, such as ■ **#655**.

As you can see in **Figure 3.36**, the reason our corners disappeared was that the background we specified was obscuring them. All we need to do to fix this is to use **background-clip** to prevent the background from extending to the border area:

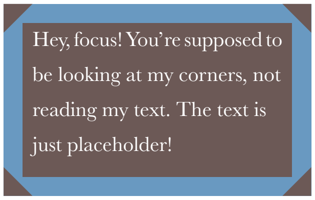

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.36
Changing our **background** to another color solves the … disappearing corners mystery

```
border: 20px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"\
         width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
    </svg>');
background: #58a;
background-clip: padding-box;
```

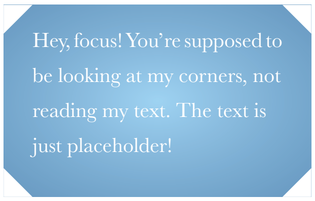

Hey, focus! You're supposed to be looking at my corners, not reading my text. The text is just placeholder!

FIGURE 3.37
Our cutout corners with a radial gradient background

The issue is now fixed and our box now looks exactly like **Figure 3.29**. However, we can easily **change the corner size in only one place**: we just modify the border width. **We can even animate it**, because **border-width** is animatable! We can also change the background with only **two edits instead of five**. In addition, because our background is now independent of the corner effect, we can even specify a gradient on it, or any other pattern, as long as it's still ■ **#58a** toward the edges. For example, check out **Figure 3.37** for an example using a radial gradient from **hsla(0,0%,100%,.2)** to **transparent**.

There is only one small issue remaining. If **border-image** is not supported, the fallback is not only the absence of corners. Due to background clipping, it also looks like there is **less spacing between the box edge and its content**. To fix that, we could just give our border a color that is identical to the background:

```
border: 20px solid #58a;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"\
            width="3" height="3" fill="%2358a">\
        <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
        </svg>');
    background: #58a;
    background-clip: padding-box;
```

This color is ignored when **`border-image`** applies, but will provide a **more graceful fallback** when it doesn't, which will look like **Figure 3.35**. As a drawback, this **increases the number of edits** we need to make to change the background color to three.

▸ **PLAY!** `play.csssecrets.io/`**`bevel-corners`**

*Hat tip to **Martijn Saly** (`twitter.com/martijnsaly`) for coming up with the initial idea of using **`border-image`** and inline SVG as a solution for beveled corners, in **a tweet of his from January 5, 2015** (`twitter.com/martijnsaly/status/552152520114855936`).*

HAT TIP

Clipping path solution

LIMITED SUPPORT

While the **`border-image`** solution is very compact and relatively DRY, it still has its limitations. For example, we still need to have either a solid color background, or a background that is a solid color toward the edges. What if we want a different kind of background, such as a texture, a pattern, or a linear gradient?

There is another way that doesn't have these limitations, though it of course has other limitations of its own. Remember the **`clip-path`** property from the **"Diamond images" secret on page 90**? The amazing thing about CSS clipping paths is that we can mix percentages (which refer to the element dimensions) with absolute lengths, offering us tremendous flexibility.

For example, the code for the clipping path to clip an element in a rectangle with beveled corners of **`20px`** size (measured horizontally) would look like this:

```
background: #58a;
clip-path: polygon(
    20px 0, calc(100% - 20px) 0, 100% 20px,
    100% calc(100% - 20px), calc(100% - 20px) 100%,
    20px 100%, 0 calc(100% - 20px), 0 20px
);
```

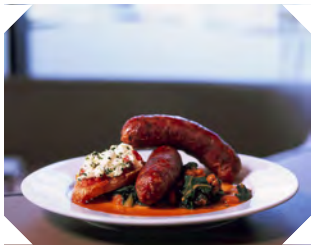


FIGURE 3.38

An image styled with beveled corners, via `clip-path`

Despite the code being short, this doesn't mean it's DRY, which is one of its biggest issues if you're not using a preprocessor. In fact, it's the most WET of the pure CSS solutions we presented, with eight (!) edits required to change the corner size. On the other hand, we can change the background in only one place, so there's that.

Among its benefits is that we can have **any background we want**, or even **clip replaced elements such as images**. Check out **Figure 3.38** for an image styled with beveled corners. None of the previous methods can do this. In addition, it is also animatable, not only to different corner sizes, but different shapes altogether. All we need to do is use a different clipping path.

Beyond its WETness and its limited browser support, it also has the drawback that **it will clip text, if there is no sufficient padding**, as it just clips the element without distinguishing between its parts. In contrast, the

FUTURE Cutout corners

In the future, we won't have to resort to CSS gradients, clipping, or SVG for this effect. A new property, **`corner-shape`**, is coming in **CSS Backgrounds & Borders Level 4** *(dev.w3.org/csswg/css-backgrounds-4/)* to save us from these pains. It will be used in conjunction with **`border-radius`** to produce cutout corners of different shapes, with their sizes defined in **`border-radius`**. For example, specifying **`15px`** cutout corners on all sides would be as simple as:

```
border-radius: 15px;
corner-shape: bevel;
```

gradient method will just let the text overflow beyond the corners (because they're just a background) and the **`border-image`** method will act just like a border and make the text wrap.

▶ **PLAY!** `play.csssecrets.io/`**`bevel-corners-clipped`**

RELATED SPECS

- **CSS Backgrounds & Borders**
  w3.org/TR/css-backgrounds
- **CSS Image Values**
  w3.org/TR/css-images
- **CSS Transforms**
  w3.org/TR/css-transforms
- **CSS Masking**
  w3.org/TR/css-masking
- **CSS Transitions**
  w3.org/TR/css-transitions
- **CSS Backgrounds & Borders Level 4**
  dev.w3.org/csswg/css-backgrounds-4

4 Visual Effects

17 Color tinting

Prerequisites

HSL color model, **`background-size`**

The problem

Adding a color tint to a grayscale image (or an image that has been converted to grayscale) is a popular and elegant way to give visual unity to a group of photos with very disparate styles. Often, the effect is applied statically and removed on **`:hover`** and/or some other interaction.

Traditionally, we use an image editing application to create two versions of the image, and write some simple CSS code to take care of swapping them. This approach works, but it adds bloat and extra HTTP requests, and is a maintenance nightmare. Imagine deciding to change the color of the effect: you would have to go through all the images and create new monochrome versions!

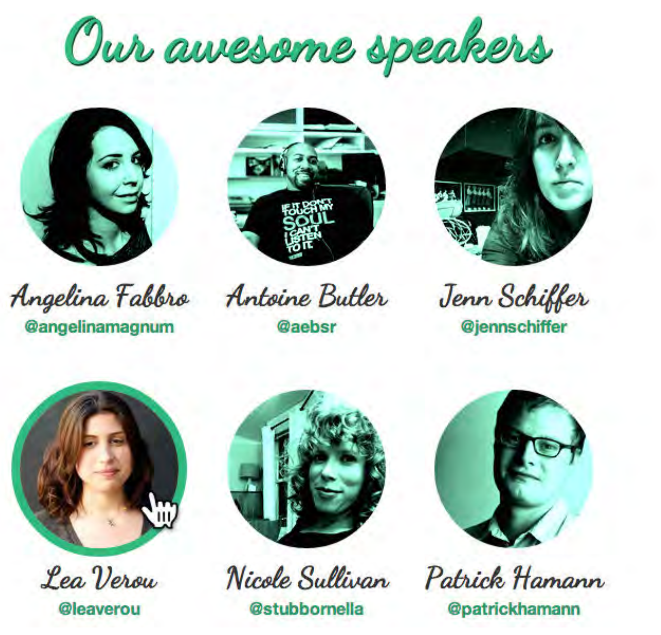


FIGURE 4.9
The CSSConf 2014 website used this effect for speaker photos, but showed the full color picture on hover and focus

Other approaches involve overlaying a semi-transparent color on top of the image or applying opacity to the image and overlaying it on a solid color. However, this is not a real tint: in addition to not converting all the colors in the image to tints of the target color, it also reduces contrast significantly.

There are also scripts that turn images into a `<canvas>` element and apply the tint through JavaScript. This does produce proper tinting, but is fairly slow and restrictive.

Wouldn't it be so much easier to be able to apply a color tint to images straight from our CSS?

Filter-based solution

LIMITED SUPPORT

Because there is no single filter function specifically designed for this effect, we need to get a bit crafty and combine **multiple filters**.

The first filter we will apply is `sepia()`, which gives the image a **desaturated orange-yellow tint**, with most pixels having a hue of around 35–40. If this is the color we wanted, then we're done. However, in most cases it won't be. If our color is more saturated, we can use the `saturate()` filter to increase the saturation of every pixel. Let's assume we want to give the image a tint of 🟥 `hsl(335, 100%, 50%)`. We need to increase saturation quite a bit, so we will use a parameter of 4. The exact value depends on your case, and we generally have to eyeball it. As **Figure 4.11** demonstrates, this combined filter gives our image a **warm golden tint**.

As nice as our image now looks, we didn't want to colorize it with this orangish yellow, but with a deep, bright pink. Therefore, we also need to apply a `hue-rotate()` filter, to **offset the hue of every pixel** by the degrees we specify. To make the hue 335 from around 40, we'd need to add around 295 (335 - 40) to it:

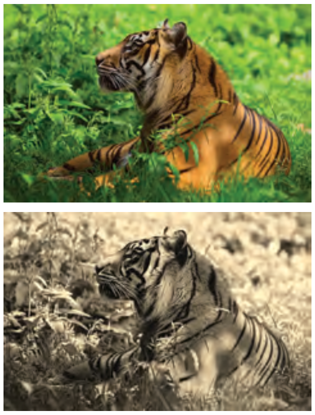


FIGURE 4.10
Top: Original image
Bottom: Image after `sepia()` filter

```
filter: sepia() saturate(4) hue-rotate(295deg);
```

At this point, we've colorized our image and you can check out how it looks in **Figure 4.12**. If it's an effect that gets toggled on `:hover` or other states, we could even apply CSS transitions to it:

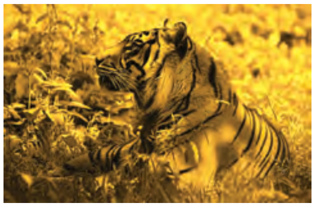


FIGURE 4.11
Our image after adding a `saturate()` filter

```
img {
    transition: .5s filter;
    filter: sepia() saturate(4) hue-rotate(295deg);
}

img:hover,
img:focus {
    filter: none;
}
```

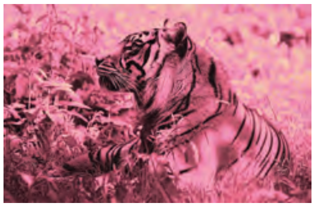


FIGURE 4.12
Our image after adding a `hue-rotate()` filter as well

▸ PLAY! play.csssecrets.io/**color-tint-filter**

Blending mode solution

The filter solution works, but you might have noticed that the result is not exactly the same as what can be obtained with an image editor. Even though we were trying to colorize with a very bright color, the result still looks rather **washed out**. If we try to increase the parameter in the `saturate()` filter, we start getting a **different, overly stylized effect**. Thankfully, there is a better way to approach this: **blending modes!**

If you've ever used an image editor such as Adobe Photoshop, you are probably already familiar with blending modes. When two elements overlap, **blending modes control how the colors of the topmost element blend with the colors of whatever is underneath it**. When it comes to colorizing images, the blending mode you need is `luminosity`. The `luminosity` blending mode **maintains the HSL lightness of the topmost element, while adopting the hue and saturation of its backdrop**. If the backdrop is our color and the element with the blending mode applied to it is our image, isn't this essentially what color tinting is supposed to do?

To apply a blending mode to an element, there are two properties available to us: `mix-blend-mode` for applying blending modes to **entire elements** and `background-blend-mode` for applying blending modes to **each background layer** separately. This means that to use this method on an image we have two options, neither of them ideal:

- Wrapping our image in a container with a background color of the color we want
- Using a `<div>` instead of an image, with its `background-image` set to the image we want to colorize and a second background layer underneath with our color

Depending on the specific use case, we can choose either of the two. For example, if we wanted to apply the effect to an `<img>` element, we would need to wrap it in another element. However, if we already have another element, such as an `<a>`, we can use that:

LIMITED SUPPORT

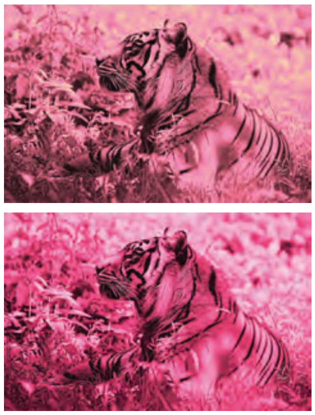


FIGURE 4.13
Comparison of the filter method (top) and the blending mode method (bottom)

HTML
```html
<a href="#something">
```

```
    <img src="tiger.jpg" alt="Rawrrr!" />
</a>
```

Then, you only need two declarations to apply the effect:

```
a {
    background: hsl(335, 100%, 50%);
}

img {
    mix-blend-mode: luminosity;
}
```

Just like CSS filters, blending modes degrade gracefully: if they are not supported, no effect is applied but the image is still perfectly visible.

An important consideration is that while **filters are animatable, blending modes are not**. We already saw how you can animate the picture slowly fading into monochrome with a simple CSS transition on the `filter` property, but you cannot do the same with blending modes. However, do not fret, as this does not mean animations are out of the question, it just means we need to think outside the box.

As already explained, `mix-blend-mode` blends the whole element with whatever is underneath it. Therefore, if we apply the `luminosity` blending mode through this property, the image is always going to be blended with **something**. However, using the `background-blend-mode` property blends each background image layer with the ones underneath it, unaware of anything outside the element. What happens then when we only have one background image and a `transparent` background color? You guessed it: **no blending takes place**!

We can take advantage of that observation and use the `background-blend-mode` property for our effect. The HTML will have to be a little different:

HTML

```html
<div class="tinted-image"
     style="background-image:url(tiger.jpg)">
</div>
```

Then we only need to apply CSS to that one `<div>`, as this technique does not require any extra elements:

```css
.tinted-image {
    width: 640px; height: 440px;
    background-size: cover;
    background-color: hsl(335, 100%, 50%);
    background-blend-mode: luminosity;
    transition: .5s background-color;
}

.tinted-image:hover {
    background-color: transparent;
}
```

However, as mentioned previously, **neither of the two techniques are ideal**. The main issues at play here are:

- The **dimensions of the image need to be hardcoded** in the CSS code.
- **Semantically**, this is not an image and will not be read as such by screen readers.

Like most things in life, there is no perfect way to do this, but in this section we've seen three different ways to apply this effect, each with its own pros and cons. The one you choose depends on the specific needs of your project.

▶ PLAY! play.csssecrets.io/**color-tint**

HAT TIP

*Hat tip to **Dudley Storey** (`demosthenes.info`) for coming up with **the animating trick for blending modes** (`demosthenes.info/blog/888/Create-Monochromatic-Color-Tinted-Images-With-CSS-blend`).*

RELATED SPECS

- **Filter Effects**
  `w3.org/TR/filter-effects`
- **Compositing and Blending**
  `w3.org/TR/compositing`
- **CSS Transitions**
  `w3.org/TR/css-transitions`

Typography 5

25 Fancy ampersands

Prerequisites

Basic font embedding through **`@font-face`** rules

The problem

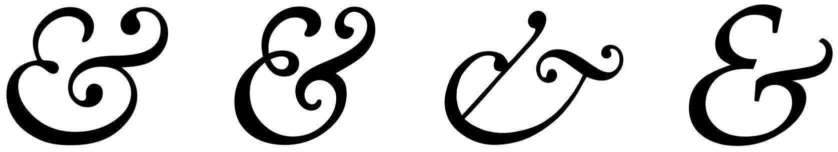


FIGURE 5.18
A few nice ampersands in fonts that are readily available in most computers; from left to right: Baskerville, Goudy Old Style, Garamond, Palatino (all italic)

You will find many hymns to the humble ampersand in typographic literature. No other character can instantly add the elegance a nicely designed ampersand has the power to add. Entire websites have been devoted to finding the font with the best looking ampersands. However, the font with the nicest ampersand is not necessarily the one you want for the rest of your text. After all, a really beautiful and elegant effect for headlines is **the**

contrast between a nice sans serif font and beautiful, intricate serif ampersands.

Web designers realized this a while ago, but the techniques employed to achieve it are rather crude and tedious. They usually involve wrapping every ampersand with a `<span>`, through a script or manually, like so:

HTML

```html
HTML <span class="amp">&amp;</span> CSS
```

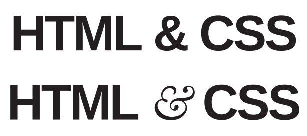


FIGURE 5.19 Our "HTML & CSS" headline, before and after the ampersand treatment

Then, we apply the font styling we want to just the `.amp` class:

```css
.amp {
    font-family: Baskerville, "Goudy Old Style",
                 Garamond, Palatino, serif;
    font-style: italic;
}
```

This works fine and you can see the before and after in **Figure 5.19**. However, the technique to achieve it is rather messy and sometimes even downright impossible, when we cannot easily modify the HTML markup (e.g., when using a CMS). Can't we just tell CSS to style certain characters differently?

The solution

It turns out that we can, indeed, style certain characters (or even ranges of characters) with a different font, but the way to do it is not as straightforward as you might have hoped.

We usually specify multiple fonts (*font stacks*) in `font-family` declarations so that in case our top preference is not available, the browser can fall back to other fonts that would also fit our design. However, many authors forget that **this works on a per-character basis as well**. If a font is available, but only contains a few characters, it will be used for those characters and the browser will fall back to the other fonts for the rest. This applies to **both local and embedded fonts** included through `@font-face` rules.

It follows that if we have a font with only **one character** (guess which one!), it will only be used for that one character, and all others will get the second, third, etc. font from our font stack. So, we have an easy way to only style ampersands: create a web font with just the ampersand we want, include it through `@font-face`, then use it first in your font stack:

```
@font-face {
    font-family: Ampersand;
    src: url("fonts/ampersand.woff");
}

h1 {
    font-family: Ampersand, Helvetica, sans-serif;
}
```

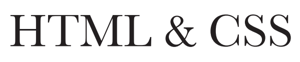


FIGURE 5.20
Including local fonts through `@font-face` results in them being applied to the whole text by default

While this is very **flexible**, it's suboptimal if all we wanted was to style ampersands with one of the **built-in fonts**. Not only is it a hassle to create a font file, it also adds an **extra HTTP request**, not to mention the potential legal issues, if the font you were going for forbids subsetting. **Is there a way to use local fonts for this?**

You might know that the `src` descriptor in `@font-face` rules also accepts a `local()` function, for specifying **local font names**. Therefore, instead of a separate web font, you could instead specify a font stack of local fonts:

```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville'),
         local('Goudy Old Style'),
         local('Garamond'),
         local('Palatino');
}
```

However, if you try to apply the Ampersand font now, you will notice that our serif font was applied to the **entire text** (**Figure 5.20**), as these fonts include all characters. This doesn't mean we're going the wrong way; it just means **we are missing a descriptor** to declare that we are only interested in the ampersand glyph from these local fonts. Such a descriptor exists, and its name is `unicode-range`.

The `unicode-range` descriptor only works inside `@font-face` rules (hence the term *descriptor*; it is **not** a CSS property) and limits the characters used to a subset. It works with both local and remote fonts. Some browsers are even smart enough to not download remote fonts if those characters are not used in the page!

Unfortunately, `unicode-range` is as **cryptic** in its syntax as it is useful in its application. It works with *Unicode codepoints*, not literal characters. Therefore, before using it, you need to find the hexadecimal codepoint of the character(s) you want to specify. There are numerous online sources for that, or you can just use the following snippet of JS in the console:

! `String#charCodeAt()` returns **incorrect results** for Unicode characters beyond the BMP (Basic Multilingual Plane). However, 99.9% of the characters you will need to look up will be in it. If the result you get is in the D800-DFFF range, it means you have an "astral" character and you're better off using a proper online tool to figure out what its Unicode codepoint is. The ES6 method `String#codePointAt()` will solve this issue.

JS
```
"&".charCodeAt(0).toString(16); // returns 26
```

Now that you have the hex codepoint(s), you can prepend them with `U+` and you've already specified a single character! Here's how the declaration would look for our ampersand use case:

```
unicode-range: U+26;
```

If you wanted to specify a **range** of characters, you would still need one `U+`, like so: `U+400-4FF`. In fact, for that kind of range, you could have used **wildcards** and specified it as `U+4??` instead. **Multiple characters or ranges are also allowed**, separated by commas, such as `U+26, U+4??, U+2665-2670`. In this case, however, a single character is all we need. Our code now looks like this:

```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville'),
         local('Goudy Old Style'),
         local('Palatino'),
         local('Book Antiqua');
    unicode-range: U+26;
}

h1 {
    font-family: Ampersand, Helvetica, sans-serif;
}
```

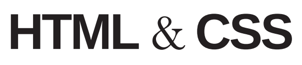


FIGURE 5.21
Applying a different font to our ampersands, with the help of font stacks and the `unicode-range` descriptor

If you try it out (**Figure 5.21**), you will see that we did, in fact, apply a different font to our ampersands! However, the result is still not exactly what we want. The ampersand in **Figure 5.19** was from the italic variant of the Baskerville font, as in general, **italic serif fonts tend to have much nicer ampersands**. We're not styling the ampersands directly, so how can we italicize them?

Our first thought might be to use the `font-style` descriptor in the `@font-face` rule. However, this does not have the effect we want at all. It merely tells the browser to use these fonts in italic text. Therefore, it will make our Ampersand font be completely ignored, unless the whole headline is italic (in which case, we will indeed get the nice italic ampersand).

To find a font's *PostScript Name* in Mac OS X, select it in the **FontBook** application and press ⌘I.

Unfortunately, the only solution here is a bit of a hacky one: instead of using the font family name, we need to use the *PostScript Name* of the **individual font style/weight** we want. So, to get the italic versions of the fonts we used, the final code would look like this:

```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville-Italic'),
         local('GoudyOldStyleT-Italic'),
         local('Palatino-Italic'),
            local('BookAntiqua-Italic');
        unicode-range: U+26;
    }

    h1 {
        font-family: Ampersand, Helvetica, sans-serif;
    }
```

And this finally works great to give us the ampersands we wanted, just like in **Figure 5.19**. Unfortunately, if we need to customize their styling even more (e.g., to increase their font size, reduce their opacity, or anything else), we would need to go the HTML element route. However, if we only want a different font and font style/weight, this trick works wonders. You can use the same general idea to also style **numbers** with a different **font, symbols, punctuation—the possibilities are endless!**

▶ PLAY! play.csssecrets.io/**ampersands**

Hat tip to ***Drew McLellan*** *(allinthehead.com) for coming up with* ***the first version of this effect*** *(24ways.org/2011/creating-custom-font-stacks-with-unicode-range).*

HAT TIP

■ **CSS Fonts**
w3.org/TR/css-fonts

RELATED SPECS

6 User Experience

33 De-emphasize by blurring

Prerequisites

Transitions, the "Frosted glass effect" secret on page 146, the "De-emphasize by dimming" secret on page 234

The problem

In the **"De-emphasize by dimming" secret on page 234**, we saw a way to de-emphasize parts of a web app by dimming them, through a semi-transparent black overlay. However, when there is a lot going on the page, we need to dim it quite a lot to provide sufficient contrast for text to appear on it, or to draw attention to a lightbox or other element. A more elegant way, shown in **Figure 6.14**, is to blur everything else in addition to (or instead of) dimming it. This is also more realistic, as it creates depth by mimicking **how our vision treats objects that are physically closer to us when we are focusing on them**.

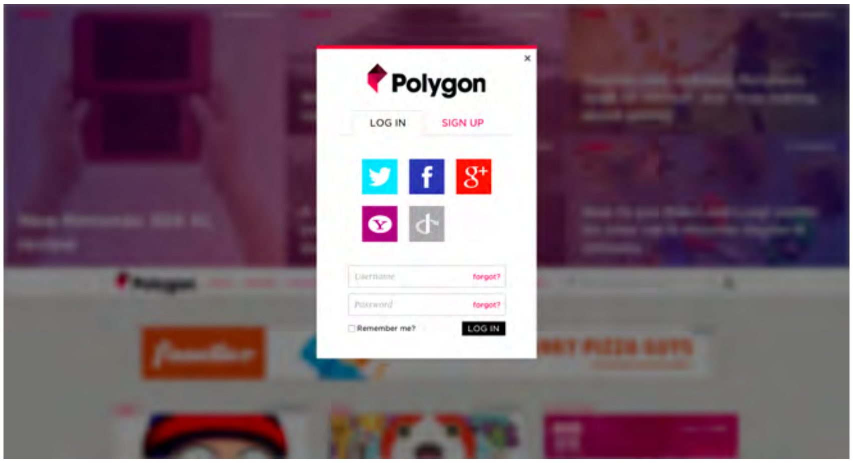


FIGURE 6.14

The gaming website **`polygon.com`** features an excellent example of drawing user attention to a dialog box by blurring everything else behind it

However, this is a far more difficult effect to achieve. Until **Filter Effects** (*`w3.org/TR/filter-effects`*), it was impossible, but even with the **`blur()`** filter, it is quite difficult. What do we apply the blur filter to, if we want to apply it to everything *except* a certain element? If we apply it to the **`<body>`** element, everything in the page will be blurred, including the element we want to draw attention to. It's very similar to the problem we addressed in the **"Frosted glass effect" secret on page 146**, but we cannot apply the same solution here, as anything could be behind our dialog box, not just a background image. What do we do?

The solution

LIMITED SUPPORT

Unfortunately, we will need an extra HTML element for this effect: we will need to wrap everything in our page except the elements that shouldn't be blurred in a wrapper element, so that we can apply the blurring to it. The **`<main>`** element is perfect for this, because it serves a double purpose: it both marks up the main content of the page (dialogs aren't usually main content) and gives us the styling hook we need. The markup could look like this:

HTML

```html
<main>Bacon Ipsum dolor sit amet…</main>
<dialog>
```

We assume that all `<dialog>` elements will be initially hidden and at most one of them will be visible at any time.

```
    O HAI, I’m a dialog. Click on me to dismiss.
</dialog>
<!-- any other dialogs go here too -->
```

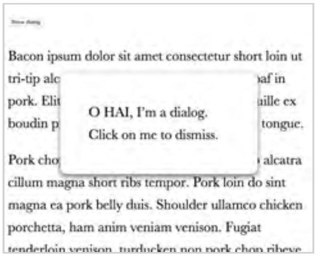

Bacon ipsum dolor sit amet consectetur short loin ut
tri-tip alc af in
pork. Elit ille ex
boudin p tongue.
O HAI, I'm a dialog.
Click on me to dismiss.
Pork cho alcatra
cillum magna short ribs tempor. Pork loin do sint
magna ea pork belly duis. Shoulder ullamco chicken
porchetta, ham anim veniam venison. Fugiat

FIGURE 6.15

A plain dialog with no overlay to de-emphasize the rest of the page

You can see how this looks with no overlay in **Figure 6.15**. Then, we need to apply a class to the `<main>` element every time we make a dialog appear and apply the blur filter then, like so:

```
main.de-emphasized {
    filter: blur(5px);
}
```

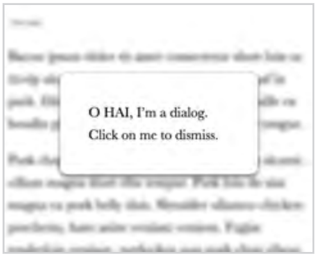

O HAI, I'm a dialog.
Click on me to dismiss.

FIGURE 6.16

Blurring the `<main>` element when the dialog is visible

As you can see in **Figure 6.16**, this already is a huge improvement. However, right now the blurring is applied immediately, which doesn’t look very natural and feels like rather awkward UX. Because **CSS filters are animatable**, we can instead smoothly transition to the blurred page:

```
main {
    transition: .6s filter;
}

main.de-emphasized {
    filter: blur(5px);
}
```

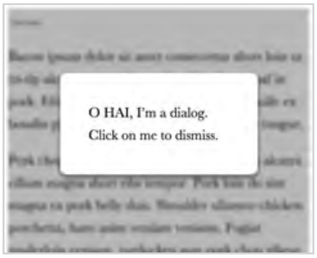

O HAI, I'm a dialog.
Click on me to dismiss.

FIGURE 6.17

Applying both blurring and dimming, both via CSS filters

It’s often a good idea to combine the two de-emphasizing effects (dimming and blurring). One way to do this is using the `brightness()` and/or `contrast()` filters:

```
main.de-emphasized {
    filter: blur(3px) contrast(.8) brightness(.8);
}
```

You can see the result in **Figure 6.17**. Dimming via CSS filters means that if they are not supported, there is **no fallback**. It might be a better idea to perform the dimming via some other method, which can also serve as a fallback (e.g., the **`box-shadow`** method we saw in the previous secret). This would also save us from the "halo effect" you can see on the edges of **Figure 6.17**. Notice how in **Figure 6.18** where we used a shadow for the dimming, this issue is gone.

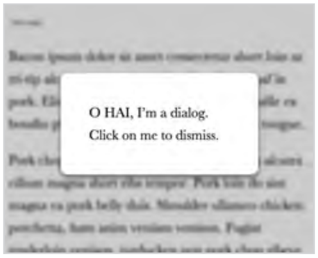

O HAI, I'm a dialog.
Click on me to dismiss.

FIGURE 6.18

Applying blurring via CSS filters and dimming via a **`box-shadow`**, which also serves as a fallback

▸ PLAY! `play.csssecrets.io/`**`deemphasizing-blur`**

HAT TIP

*Hat tip to **Hakim El Hattab** (`hakim.se`) for coming up with **a smiliar effect** (`lab.hakim.se/avgrund`). In addition, in Hakim's version of the effect, the content also becomes smaller via a **`scale()`** transform, to further enhance the illusion that the dialog is getting physically closer to us.*

RELATED SPECS

- **Filter Effects**
  `w3.org/TR/filter-effects`
- **CSS Transitions**
  `w3.org/TR/css-transitions`

Structure & Layout 7

41 Sticky footers

Prerequisites

Viewport-relative units (see the "Vertical centering" secret on page 280), **`calc()`**

The problem

This is one of the oldest and most common problems in web design, so common that most of us have experienced it at one point or another. It can be summarized as follows: a footer with any block-level styling, such as a background or shadow, works fine when the content is sufficiently long, but breaks on shorter pages (such as error messages). The breakage in this case being that the footer does not "stick" at the bottom of the viewport like we would want it to, but at the bottom of the content.

Specifically, the issue appears on pages whose content is shorter than the viewport height minus the footer height.

It is not only its ubiquity that made it popular, but also how **deceptively easy it looks at first**. It's a textbook case of the type of problem that requires significantly more time to solve than expected. In addition, **this is still not a solved problem in CSS 2.1**: almost all classic solutions require a fixed height for the footer, which is flimsy and rarely feasible. Furthermore, all of them are **overly complicated**, **hacky**, and have **specific markup requirements**. Back then, this was the best we could do, given the limitations of CSS 2.1. But can we do better with modern CSS, and if so, how?

Fixed height solution

We will work with an extremely bare-bones page with the following markup inside the `<body>` element:

```html
HTML
<header>
    <h1>Site name</h1>
</header>
<main>
    <p>Bacon Ipsum dolor sit amet…
    <!-- Filler text from baconipsum.com --></p>
</main>
<footer>
    <p>© 2015 No rights reserved.</p>
    <p>Made with ♥ by an anonymous pastafarian.</p>
</footer>
```

We have also applied some basic styling to it, including a background on the footer. You can see how it looks in **Figure 7.23**. Now, let's reduce the content a bit. You can see what happens then, in **Figure 7.24**. This is the sticky footer problem in all its glory! Great, we have recreated the problem, but how do we solve it?

If we assume that our footer text will never wrap, we can deduce a CSS length for its height:

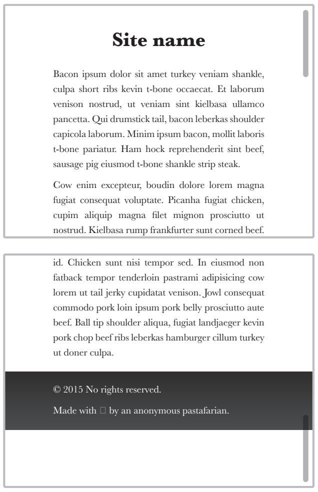
$$2 \text{ lines} \times \text{line height} + 3 \times \text{paragraph margin} + \text{vertical padding} =$$

$$2 \times \texttt{1.5em} + 3 \times \texttt{1em} + \texttt{1em} = \texttt{7em}$$

If you've never had the pleasure of pulling your hair out and diving in the existing literature for this problem, here are a few popular links with existing, widely used solutions that have served many a web developer before CSS Level 3 specs were conceived:

- `cssstickyfooter.com`
- `ryanfait.com/sticky-footer`
- `css-tricks.com/snippets/css/sticky-footer`
- `pixelsvsbytes.com/blog/2011/09/sticky-css-footers-the-flexible-way`
- `mystrd.at/modern-clean-css-sticky-footer`

The last two are the most minimal in the lot, but still have their own limitations.



Site name

Bacon ipsum dolor sit amet turkey veniam shankle, culpa short ribs kevin t-bone occaecat. Et laborum venison nostrud, ut veniam sint kielbasa ullamco pancetta. Qui drumstick tail, bacon leberkas shoulder capicola laborum. Minim ipsum bacon, mollit laboris t-bone pariatur. Ham hock reprehenderit sint beef, sausage pig eiusmod t-bone shankle strip steak.

Cow enim excepteur, boudin dolore lorem magna fugiat consequat voluptate. Picanha fugiat chicken, cupim aliquip magna filet mignon prosciutto ut nostrud. Kielbasa rump frankfurter sunt corned beef.

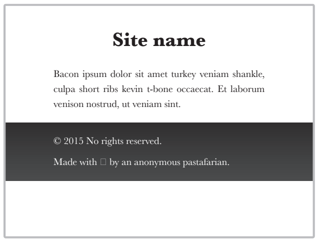
id. Chicken sunt nisi tempor sed. In eiusmod non fatback tempor tenderloin pastrami adipisicing cow lorem ut tail jerky cupidatat venison. Jowl consequat commodo pork loin ipsum pork belly prosciutto aute beef. Ball tip shoulder aliqua, fugiat landjaeger kevin pork chop beef ribs leberkas hamburger cillum turkey ut doner culpa.

© 2015 No rights reserved.

Made with □ by an anonymous pastafarian.

FIGURE 7.23

How our simple page looks when its content is sufficiently long


Site name

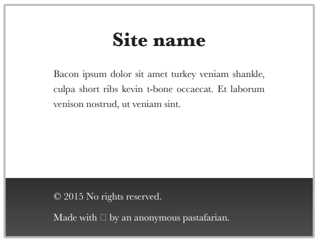
Bacon ipsum dolor sit amet turkey veniam shankle, culpa short ribs kevin t-bone occaecat. Et laborum venison nostrud, ut veniam sint.

© 2015 No rights reserved.

Made with □ by an anonymous pastafarian.

FIGURE 7.24
The sticky footer problem in all its glory

Be careful when using `calc()` with subtraction or addition: the `+` and `-` operators **require** spaces around them. This very odd decision was made for future compatibility. If at some point keywords are allowed in `calc()`, the CSS parser needs to be able to distinguish between a hyphen in a keyword and a minus operator.


Site name

Bacon ipsum dolor sit amet turkey veniam shankle, culpa short ribs kevin t-bone occaecat. Et laborum venison nostrud, ut veniam sint.

© 2015 No rights reserved.

Made with □ by an anonymous pastafarian.

FIGURE 7.25
The footer after we've applied CSS to make it stick

Similarly, the header height is `2.5em`. Therefore, by using viewport-relative units and `calc()`, we can "stick" our footer to the bottom with essentially one line of CSS:

```
main {
    min-height: calc(100vh - 2.5em - 7em);
    /* Avoid padding/borders screwing up our height: */
    box-sizing: border-box;
}
```

Alternatively, we could apply a wrapper around our `<header>` and `<main>` elements so that we only need to calculate the footer height:

```
#wrapper {
    min-height: calc(100vh - 7em);
}
```

This works (**Figure 7.25**) and it seems to be slightly better than the existing fixed height solutions, mainly due to its minimalism. However, except for very simple layouts, this is **not practical at all**. It requires us to assume that the footer text will **never wrap**, we need to edit the `min-height` **every time we change the footer metrics** (i.e., it is not DRY), and unless we're willing to add a wrapper HTML element around our header and content, we need to do the same calculations and modifications for the header as well. Surely, in this day and age we can do better, right?

▶ **PLAY!** play.csssecrets.io/**sticky-footer-fixed**

Flexible solution

Flexbox is perfect for these kinds of problems. We can achieve perfect flexibility with only a few lines of CSS and there is no need for weird calculations or extra HTML elements. First, we need to apply **`display: flex`** to the **`<body>`** element, as it's the parent of all three of our main blocks, to toggle Flexible Box Layout (Flexbox) for all three of them. We also need to set **`flex-flow`** to **`column`**, otherwise they will be all laid out horizontally on a single row (**Figure 7.26**):

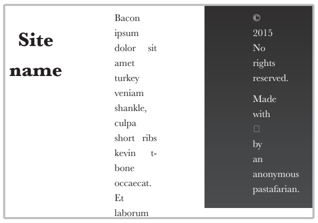

Site name
Bacon ipsum dolor sit amet turkey veniam shankle, culpa short ribs kevin t-bone occaecat. Et laborum
© 2015 No rights reserved.
Made with □ by an anonymous pastafarian.

FIGURE 7.26

Applying **`flex`** without applying anything else arranges the children of our element horizontally

```
body {
    display: flex;
    flex-flow: column;
}
```

At this point, our page looks about the same as it did before all the Flexbox stuff, as every element occupies the entire width of the viewport and its size is determined by its contents. Ergo, we haven't really taken advantage of Flexbox yet.

To make the magic happen, we need to specify a **`min-height`** of **`100vh`** on **`<body>`**, so that it occupies **at least the entire height of the viewport**. At this point, the layout still looks exactly like **Figure 7.24**, because even though we have specified a minimum height for the entire body element, the heights of each box are still determined by their contents (i.e., they are *intrinsically determined*, in CSS spec parlance).

What we need here is for the height of the header and footer to be **intrinsically** determined, but the height of the content should flexibly stretch to all the leftover space. We can do that by applying a **`flex`** value that is larger than **`0`** (**`1`** will work) to the **`<main>`** container:

```
body {
    display: flex;
    flex-flow: column;
    min-height: 100vh;
}

main { flex: 1; }
```

TIP! The **`flex`** property is actually a shorthand of **`flex-grow`**, **`flex-shrink`**, and **`flex-basis`**. Any element with a **`flex`** value greater than **`0`** becomes flexible and **`flex`** controls the ratio between the dimensions of different flexible elements. For example, in our case, if **`<main>`** had **`flex: 2`** and **`<footer>`** had **`flex: 1`**, the height of the footer would be **twice** the height of the content. Same if the values were **4** and **2** instead of **2** and **1**, because **it's their relationship that matters**.

That's it, no more code required! The perfect sticky footer (same visual result as in **Figure 7.25**), with only four simple lines of code. Isn't Flexbox beautiful?

▸ **PLAY!** play.csssecrets.io/**sticky-footer**

HAT TIP

*Hat tip to **Philip Walton** (philipwalton.com) for coming up with **this technique** (philipwalton.github.io/solved-by-flexbox/demos/sticky-footer).*

RELATED SPECS

- **CSS Flexible Box Layout**
  w3.org/TR/css-flexbox
- **CSS Values & Units**
  w3.org/TR/css-values

8 Transitions & Animations

44 Blinking

Prerequisites

Basic CSS animations, the "Frame-by-frame animations" secret on page 308

The problem

Remember the old `<blink>` tag? Of course you do. It has become a cultural symbol in our industry, reminding us of the humble, clumsy beginnings of our discipline, and always willing to serve as an inside joke between old-timers. It is universally despised, both because it violated separation of structure and style, but mainly because its overuse made it a pain for anyone browsing the Web in the late 90s. Even its own inventor, Lou Montulli, has said *"[I consider] the blink tag to be the worst thing I've ever done for the Internet."*

However, now that the nightmare of the `<blink>` tag is long behind us, we sometimes still find ourselves needing a blinking animation. It feels weird at first, a bit like discovering some sort of strange perversion inside us that we never knew we had. The identity crisis stops when we realize that there are a few use cases in which blinking can **enhance usability, rather than reduce it**.

A common UX pattern is blinking a few times (no more than three!) to indicate that a change has been applied somewhere in the UI or to highlight the current link target (the element whose id matches the URL `#hash`). Used in such a limited way, blinking can be very effective to draw the user's attention to an area, but due to the limited number of iterations, it doesn't have the adverse effects the `<blink>` tag did. Another way to keep the good of blinking (directing user attention) without the bad (distracting, annoying, seizure inducing) is to "smoothe" it out (i.e., instead of alternating between an abrupt "on" and "off" state, to have a smooth progression between the two).

However, how do we implement all this? The CSS-only replacement for the `<blink>` tag, `text-decoration: blink`, is too limited to allow us to do what we want, and even if it was powerful enough, its browser support is very poor. Can we use CSS animations for this, or is JS our only hope?

The solution

There are actually multiple ways to use CSS animations to achieve any kind of blinking: on the whole element (via `opacity`), on the text color (via `color`), on its border (via `border-color`), and so on. In the rest of this section, we will assume that we want to blink the text only, as that is the most common use case. However, the solution for other parts of an element is analogous.

Achieving a smooth blink is rather easy. Our first attempt would probably look like this:

```
@keyframes blink-smooth { to { color: transparent } }

.highlight { animation: 1s blink-smooth 3; }
```

This *almost* works. Our text smoothly fades from its text color to transparent, however it then **abruptly jumps back** to the original text color. Plotting the change of text color over time helps us figure out why this happens (**Figure 8.18**).

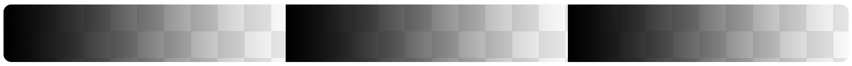


FIGURE 8.18
The progression of our text color over three seconds (three iterations)

This might actually be desirable. In that case, we are done! However, when we want the blinking to be smooth both when the text fades out and when it fades in, we have a bit more work to do. One way to achieve this would be by changing the keyframes to make the switch happen in the middle of each iteration:

```
@keyframes blink-smooth { 50% { color: transparent } }

.highlight {
    animation: 1s blink-smooth 3;
}
```

This looks like the result we wanted. However, although it doesn't show in this particular animation (because it's difficult to differentiate between timing functions with color/opacity transitions), it's important to keep in mind that the animation is accelerating both when it fades in and when it fades out, which could look unnatural for certain animations (e.g., pulsating animations). In that case, we can pull a different tool out of our toolbox: `animation-direction`.

The only purpose of `animation-direction` is to reverse either all iterations (`reverse`), every even one (`alternate`) or every odd one (`alternate-reverse`). What is great about it is that **it also reverses the timing function**, creating far more realistic animations. We could try it on our blinking element like so:

```
@keyframes blink-smooth { to { color: transparent } }
```

```
.highlight {
    animation: .5s blink-smooth 6 alternate;
}
```

Note that we had to double the number of iterations (instead of the duration, like the previous method), as now one fade-in/fade-out pair consists of two iterations. For the same reason, we also cut **`animation-duration`** in half.

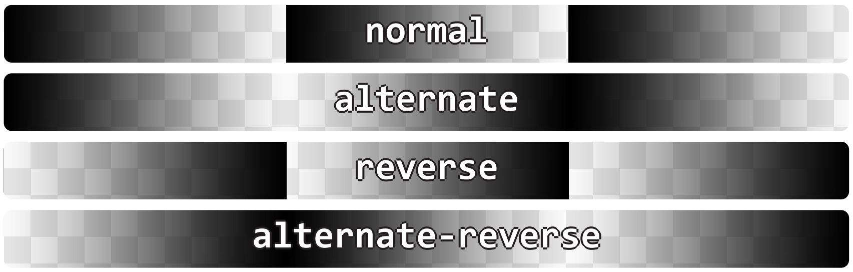


FIGURE 8.19 All four values of **`animation-direction`** and their effect on a color animation from **`black`** to **`transparent`** over three iterations

If we want a smooth blink animation, we're done at this point. However, what if we want a classic one? How do we go about it? Our first attempt might look like this:

```
@keyframes blink { to { color: transparent } }

.highlight {
    animation: 1s blink 3 steps(1);
}
```

However, this will fail spectacularly: absolutely nothing will happen. The reason is that **`steps(1)`** is essentially equivalent to **`steps(1, end)`**, which means that the transition between the current color and **`transparent`** will happen in one step, and **the value switch will occur at the end** (**Figure 8.20**). Therefore, **we will see the start value for the entire length of the animation, except one infinitesimally short point**

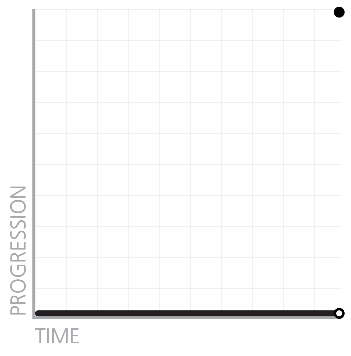


FIGURE 8.20

What **`steps(1)`** actually does to our animation

in time at the end. If we change it to **`steps(1, start)`** the opposite will happen: the switch will occur at the start, so we will only see transparent text, with no animation or blinking.

A logical next step would be to try **`steps(2)`** in both its flavors (start and end). Now we do see some blinking, but it's between semi-transparent text and transparent or semi-transparent and normal respectively, for the same reason. Unfortunately, because we cannot configure **`steps()`** to make the switch in the middle, but only at the start and end, the only solution here would be to adjust the animation keyframes to make the switch at **`50%`**, like we did earlier:

```css
@keyframes blink { 50% { color: transparent } }

.highlight {
    animation: 1s blink 3 steps(1); /* or step-end */
}
```

This finally works! Who would have guessed that a classic abrupt blink would have been harder to accomplish than a modern, smooth one? CSS never ceases to surprise….

▶ **PLAY!** play.csssecrets.io/**blink**

- **CSS Animations**
  w3.org/TR/css-animations

RELATED SPECS