A BOOK APART

Brief books for people who make websites

№ 19

Chris Coyier

PRACTICAL SVG

FOREWORD BY Val Head

MORE FROM A BOOK APART

Design for Real Life
Eric Meyer & Sara Wachter-Boettcher

Git for Humans
David Demaree

Going Responsive
Karen McGrane

Responsive Design: Patterns & Principles
Ethan Marcotte

Designing for Touch
Josh Clark

Responsible Responsive Design
Scott Jehl

You're My Favorite Client
Mike Monteiro

On Web Typography
Jason Santa Maria

Sass for Web Designers
Dan Cederholm

Just Enough Research
Erika Hall

Content Strategy for Mobile
Karen McGrane

Visit abookapart.com for our full list of titles.

Copyright © 2016 Chris Coyier
All rights reserved

Publisher: Jeffrey Zeldman
Designer: Jason Santa Maria
Executive Director: Katel LeDû
Editor: Caren Litherland
Technical Editor: Chris Lilley
Copyeditor: Lisa Maria Martin
Proofreader: Katel LeDû
Compositor: Rob Weychert
Ebook Producer: Ron Bilodeau

ISBN: 978-1-937557-43-0

A Book Apart
New York, New York
http://abookapart.com

10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

FOREWORD

RARELY, IF EVER, has a web project of mine been completed without Chris Coyier's help. Not in person of course, but through the expansive resource that is CSS-tricks.com. Whether I was scouring Google in a panic mere hours before a deadline, or casually trying to remember how exactly those @#$%&! CSS columns work, Chris and CSS Tricks were always there for me.

Think back to the last time you hunted for any CSS-related answers. I bet CSS Tricks appeared at the top of your search results. For years, Chris has been our super-smart web-design friend who always has our backs. Who better, then, to guide us through the wide-ranging world that is SVG?

SVG is changing the way we build the web. It's an amazing, powerful tool that is so simple, and yet so complex at the same time. As I like to say, "The more you use SVG, the more you realize you don't *know* SVG!" Good thing Chris has written this book to help! (With much better jokes, too.)

Chris includes everything you need to confidently make decisions about using SVG in your work. He doesn't purport to tell you *everything* about SVG; that would be unwieldy. But he does cover the most important aspects of using SVG like a pro—all the things you need to know for a strong start to your SVG adventures. Even if it's not your first time using SVG, you're still guaranteed to find some helpful tips. Yes, Chris is that good.

Learning SVG can seem like a daunting task, but it's a breeze when Chris guides you through it. Explore the full range of SVG capabilities, from the foundations of embedding options to the really fun stuff like animation (my favorite) and filters. Get ready to jump into the wonderful world of SVG with the most affable guide I know!

—Val Head

INTRODUCTION

HEY, EVERYBODY! Let's be honest. Look how short this book is. We don't have much time together. I think it's best if we get started on our little adventure with SVG right away.

Here's how easy it can be to use SVG (**FIG 0.1**):

```
<img src="dog.svg" alt="dog">
```

No joke!

Confession: I was aware of SVG's existence for years before I realized that was possible, in part because early implementations of SVG required the `object` or `iframe` element. SVG support on the HTML `img` element came much later (and with caveats).

Where did this `dog.svg` file come from, though? Well, the Noun Project is a particularly great source of such images (http://bkaprt.com/psvg/00-01/). There are lots of other places online to get your hands on vector graphics, too. Pretty much all stock photography sites let you filter search results by "vector," which is exactly what you need for SVG.

When downloading from the Noun Project, you have the option of downloading the PNG or SVG version. If you download both, you'll wind up with two files:

```
icon_364.png
icon_364.svg
```

They're both the same image of a dog. If you link to either of them from an `img` tag, you'll get the same image of the same dog. So what's the difference?

One important difference is file size. The PNG version is 40 KB and the SVG version is 2 KB—literally twenty times smaller! We'll come back to this later.

But the main difference is the file format itself. PNG (like its GIF and JPG cousins) is a *raster* image format. (Raster images are also sometimes called *bitmap* images; although the terms are largely interchangeable, we'll use "raster" for the purposes of this book.) Think of a raster as a grid of pixels. The difference

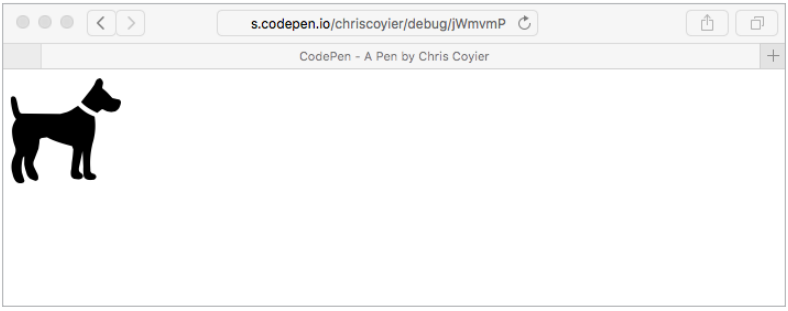

FIG 0.1: SVG being used in an `img` tag in HTML.

between raster formats is largely about how that grid of pixel information is compressed. This is all tremendously nerdy, but the common denominator is: *pixels*.

SVG is different. Think of SVG as a set of instructions for drawing shapes. In fact, this is not a metaphor or abstraction: SVG is literally a set of instructions for drawing shapes. I don't think you can write a book about SVG without saying this, so let's get it out of the way: SVG stands for *Scalable Vector Graphics*. "Vector" is the key word here. Think geometry: points, lines, curves, polygons. The instructions in SVG read something like this: "Go to such-and-such coordinate; draw a rectangle that is so tall and so wide. Or draw a line, or an ellipse, or follow some more complex instructions and draw the Coca-Cola logo."

Because SVG images are just sets of drawing instructions, they aren't tied to any particular pixel dimensions. Our `dog.svg` is happy to be displayed at 100 pixels wide or 2000 pixels wide. The exact same file will be equally *visually crisp* at any size. How excellent is that? I'll answer for you since this is a book and not a chat room, sheesh. *It's very excellent.*

This is in stark contrast to raster images, whose file size goes up dramatically the larger the dimensions get. Imagine

doubling the size of an image from 100 by 100 pixels to 200 by 200 pixels, as is recommended to ensure that a raster graphic remains visually crisp on a display with twice the pixel density (or what Apple calls a "Retina" display). Remember: that's not *twice* as much pixel data; it's *four times* as much. Four times as much data being sent across the network. Four times as much memory used to display it. Four times the bandwidth.

And Retina-style high resolution displays, which are around 2x normal pixel density, are only the beginning. The pixel density of screens keeps increasing. The screens look great, but they pose a huge challenge to building for the web. As designers, we simply can't quadruple the size of our sites—at least not without serious implications, like losing impatient customers to slow load times, or excluding users altogether.

Raster image bloat has become such a problem that we've invented new HTML elements and attributes—by which I mean `img srcset` and `picture`—to deal with it. With these newcomers, we're able to prepare and specify multiple images, and the most appropriate one will be used by the browser for the current screen. That's a book in itself; in fact, Scott Jehl's *Responsible Responsive Design* goes into much more detail about these two elements than I can here (http://bkaprt.com/psvg/00-02/). Often, a single SVG image will work as the most appropriate choice for any screen, which keeps things simple.

Let's review what we know about SVG so far:

- SVG can have a smaller file size than a similar raster image.
- SVG can be scaled to any size.
- Infinite scalability means that SVG looks crisp at any size.

Pretty compelling, right?

Let's add one more item to that list: SVG is supported in all modern browsers, both mobile and desktop, at least three versions back. So why the sudden resurgence of interest in SVG? Because even though SVG itself is fairly old in web years, this level of ubiquitous support is fairly recent.

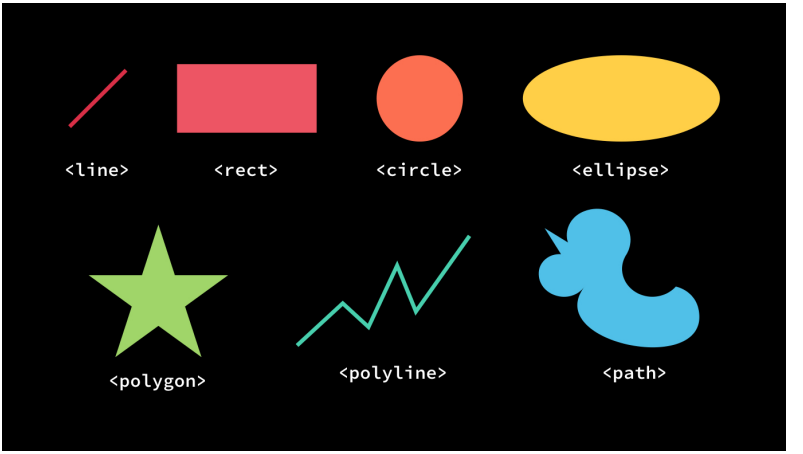

FIG 0.2: Example shapes you can draw with SVG elements.

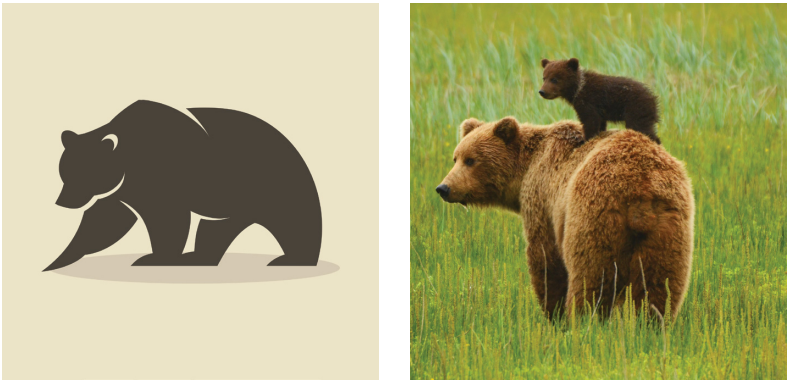

FIG 0.3: Fairly obvious example of vector vs. raster.

In 1998, when SVG was introduced, bandwidth was the big concern; an episode of *The Web Ahead* with Jen Simmons and Doug Schepers talks about this (http://bkaprt.com/psvg/00-03/). Most people used dial-up modems to access the internet; less than 3% had broadband (http://bkaprt.com/psvg/00-04/). Sending drawing instructions across the network was a

very appealing idea, since it would be much faster than sending actual graphics.

Today, we're still worried about bandwidth, mainly because of the often slow network connections of mobile devices. We'll get into the nitty-gritty of browser support (and what you can do for older browsers without support) later.

For those of you raising your WELL ACTUALLY index fingers, *en garde*! There are caveats to a lot of this. I'll attempt to cover all of them as we go along.

THE SHAPES OF SVG

This isn't going to be a book that dives deep into the SVG syntax, but it's worth being aware of the SVG elements that draw actual shapes. There are surprisingly few of them (**FIG 0.2**).

That's it! All of these except `path` are considered "basic shapes." `path` is unique because it can draw anything, including any shape the other elements can draw. In fact, under the hood, all the other shapes are just a convenient syntax for a path—syntactic sugar, as they say. Drawing only gets slightly more complicated when you factor in fills, strokes, masks, and filters. But this is pretty much all SVG has to work with for drawing things. That doesn't mean it's limited, though; these are fundamental elements of design in any two-dimensional medium.

In fact, whether you realize it or not, I bet you're already pretty good at knowing when you're looking at an image made from these shapes.

YOUR VECTOR INTUITION

While SVG might be somewhat new to you, I suspect that you can already intuit whether an image is *vector* or *raster*. Which of the two bear images in **FIG 0.3** is a vector graphic?

It's the one on the left. Easy, right? It looks like it was drawn from mathematical curves and shapes, and in fact it was. Here's another. Which one of the bears in **FIG 0.4** is vector?

FIG 0.4: Slightly less obvious example of vector vs. raster.

TRICK QUESTION. They are both vector. That crisp, cartoony look tells us this is a vector graphic. They are both a bit more complex than the vector bear from **FIG 0.3**, though. The file size probably won't be trivially small, especially for the shaded teddy bear on the right. Let's think about this for a second.

For the most part, if a graphic is vector, it should be in SVG format for use on the web. It will look better, be more efficient, and open up interesting design possibilities (which we'll cover in this book). If a graphic is raster, it should be in PNG or JPG format for use on the web. Or, if it's animated, in GIF format. Or WebP, a wonderful new format that is the most efficient of its kind but, like SVG, may require a fallback (http://bkaprt.com/psvg/00-05/). These formats are still the most appropriate choice. SVG isn't meant to replace raster graphics, unless they are being used to display a graphic that should have been vector to begin with.

The only time this rationale breaks down is if a vector image becomes *too complicated*, and consequently the file size of the SVG becomes too big to be practical. Does the image consist of three combined circles? That's about as simple as it gets. Is the image an oak tree with hundreds of detailed leaves? That means a lot of complexity and therefore a large file size. Sometimes

it's cut and dried (either use SVG or don't); sometimes it's more ambiguous. Your best bet is to test the file size both ways, gauge how important SVG's features are to the situation, and make the call. But remember: if you choose raster, hang on to that original vector source file in case you need to edit it and save a new raster version later!

SVG can also *contain* raster graphics. That may seem a bit weird at first, but it makes more sense when you think of SVG as a set of instructions: draw this thing over here, put this text over there, place this image back here. It makes SVG a good choice for graphics that mix vector and raster artwork.

Now that we know what some of SVG's benefits are, where to find SVG images, and when the SVG format is most appropriate, let's turn to how to actually use it on the web. We've really jumped right in here, haven't we? I didn't want to linger too long in the shallow end of the pool. If you're a complete beginner, I hope you'll stick with me and push yourself to learn more.

1 THE BASICS OF USING SVG

I WANT TO COVER all of the different ways we can use SVG on the web right away so that it doesn't feel so mysterious. There are three primary ways, each of which can be useful.

SVG AS HTML img

We already covered this one. You can use SVG images in HTML like this:

```
<img src="leaf-maple.svg" alt="Five-lobed sugar
  maple leaf">
```

Here we're using `img` with SVG just as we would with any other appropriate image format (like JPG, GIF, or PNG): to display an image within content.

In **FIG 1.1**, the leaf itself is part of the content of the article. A *content management system* (CMS) could conceivably produce leaf pages like this, the leaf image being part of the content unique to that page.

FIG 1.1: Displaying an SVG `image` within some content (http://bkaprt.com/psvg/01-01/).

Now imagine that that same CMS also automatically creates a weekly newsletter of newly-added leaves to send to members of the site. The newsletter uses a new type of template, but the content is the same. The image of the leaf should appear wherever that content goes. That's a *content image*, and a perfect use for SVG-as-`img`.

SVG AS CSS background-image

CSS plays well with SVG, too. It goes like this:

```
.main-header {
   background-image: url(header-bg.svg);
}
```

Here, you use SVG just as you would background images in any other appropriate file format: as part of the design of a page. In this case, the image itself isn't considered content.

In **FIG 1.2**, that spiky separator between the header and the content is a very small, repeating SVG image. Is it a nice visual element for the site? Yep. Is it a good use for SVG because it's small, scalable, and visually crisp? Yep. Is it *part of* the content? Nah.

FIG 1.2: Using SVG as a `background-image` in CSS (http://bkaprt.com/psvg/01-02/).

Imagine this is another blog post in our hypothetical CMS that sends out the weekly newsletter. This imaginary newsletter is just a way for us to think about what content *is* and the different ways in which it might be reused. But you can still *design* a newsletter. RSS is another story. When content is syndicated, you likely have no control over how that content is presented, aside from some barebones HTML. Would that spiky separator need to go out in the newsletter? Probably not. It isn't required for the content to make sense. So this isn't a content image; it's an image that is part of a template and part of the design applied to that template. This is a perfect time for SVG as CSS `background-image`.

INLINE SVG

Another way to use SVG is to drop it right into the HTML, hence the moniker "inline" SVG. Simply open up a **.svg** file in a text editor, copy all the code, and paste it into the HTML where you want that image to be. Like this:

```
<h1>Hey, there's an SVG image below me!</h1>

<svg viewBox="0 0 100 100">
   <rect x="10" y="10" width="100" height="100" />
   <!-- and all the shapes you need! —>
</svg>
```

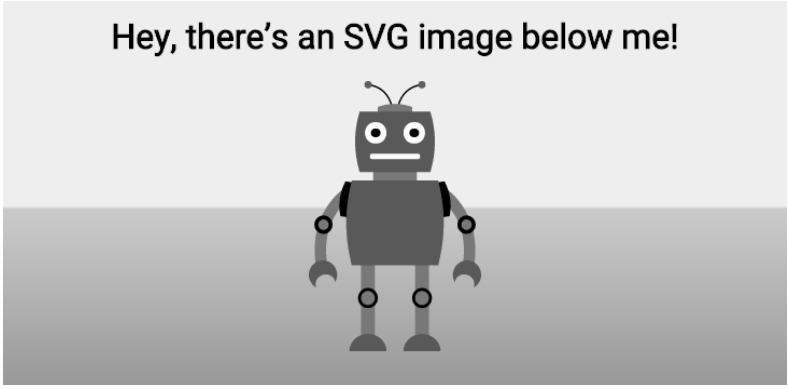

FIG 1.3: An example of inline SVG in HTML (http://bkaprt.com/psvg/01-03/).

SVG is right at home in HTML, because they are both markup languages (you know: angle brackets with tags, attributes, and familiar stuff like that). While SVG is a stand-alone image format, browsers that support SVG will parse this SVG and render it right within the HTML document. A document within a document!

This is a particularly compelling way to use SVG because:

- You can style the individual shapes with CSS, the same CSS you use for the rest of your site.
- You can affect the shapes with JavaScript, the same JavaScript you use for the rest of your site.
- The page doesn't need to make a network request for the image.
- You can make copies easily via the `use` element. (We'll cover that later.)
- You can handle accessibility as well as (or better than) you can with other methods through the use of proper accessibility-focused tags and ARIA attributes.

SVG and all of its descendant shapes are "in the DOM," as they say. This means that you have the same access and control

over them that you would over a `div` or `h3` or any other element. We could give our robot pretty red kneepads in CSS, have it talk when you click its mouth, dance when you hover over it, or just about anything else you can imagine. SVG used as `img` or `background-image` can't be controlled in this way, nor can it link to any other outside assets, like a stylesheet.

These things are unique to inline SVG, which makes it very powerful and compelling to use. We'll see inline SVG really sing a little later, when we look at things like building an icon system with it, animating it, and other design possibilities.

Those are the three *primary* ways you can use SVG on the web. There are a handful of additional ways, including linking to SVG through `object`, `embed`, or `iframe` elements. If you go that route, you retain some of the same interactive possibilities you get with inline SVG, but with the major caveat that everything needs to be embedded in the SVG source itself or linked to independently. Reach for these solutions in situations where inline SVG feels like too much "clutter" in the parent document, or if you need the interactivity or ability to link to other assets and don't mind the extra network requests. Personally, I find the use cases for these other methods few and far between, so I don't think we should squander what little time we have together on them.

BROWSER SUPPORT

Let's get more specific about browser support. After all, one thing that makes SVG so exciting is the excellent browser support it enjoys. A few notes first, though:

- Even if a given browser doesn't support SVG the way you want to use it, there's always a way to handle a fallback. That's all just part of the job, web buddies. We'll cover that in Chapter 9.
- SVG support isn't always *yep* or *nope*. Even if a browser largely supports SVG, particular features can have quirks. We'll cover those throughout the book as they crop up.

	SVG AS `<IMG>`	SVG AS `BACKGROUND-IMAGE`	INLINE `<SVG>`
Safari (any version)	✔	✔	5+
Chrome (any version)	✔	✔	✔
Firefox (any version)	✔	✔	✔
Opera (any version)	✔	✔	11.6+
Internet Explorer	9+	9+	9+

FIG 1.4: Support for SVG is robust across browsers—except for IE 8 and below.

	SVG AS `<IMG>`	SVG AS `BACKGROUND-IMAGE`	INLINE `<SVG>`
Any browser on iOS	✔	✔	5.1+
Android Browser (before Chrome became the default Android browser).	3+	3+	3+
Chrome on Android	✔	✔	✔
Firefox on Android	✔	✔	✔
Opera Mini	✔	✔	✘
Opera Mobile	✔	✔	12+
IE Mobile	✔	✔	✔

FIG: 1.5: Support for SVG across the biggest mobile browsers.

The tables in FIG 1.4 and 1.5 help explain the browser support situation across desktop and mobile.

Desktop browsers

SVG fares pretty well on the desktop (FIG 1.4). As you can see, support looks pretty good across browsers—the only one to watch out for is IE 8 (and down).

Mobile browsers

On the mobile front, things are a little more complicated, but mostly okay. There are far too many mobile browser/platform combos to detail here, so let's cover the biggest players (FIG 1.5).

The standout problems on mobile are Android 2.3 and down (no SVG support at all) and lack of inline SVG support in Opera Mini.

If you're interested in testing a specific combination of browser/version/platform yourself, I've created a very simple test case (http://bkaprt.com/psvg/01-04/). Also, the website Can I Use... tracks a ton of SVG support information that is worth consulting (http://bkaprt.com/psvg/01-05/).

Before we go too much further, let's make sure we know how to get our hands on SVG—either by creating it in vector software, or finding it online and working with it in that same software. That's next.

2 SOFTWARE

I ONCE HEARD ABOUT A GUY who could look at the source code for SVG and visualize what it drew. (The guy was Doug Schepers, and I heard this on *The Web Ahead*.) I can't do that and I bet you a nickel you can't, either. But I think it's really cool that the source code of SVG is intelligible. You *can* learn it and work with it, and there is some benefit to that.

But you don't actually have to *design* with the code; you can do that with design software. Fortunately, there is no shortage of software that speaks SVG.

DESKTOP SOFTWARE

Adobe Illustrator

Illustrator is the quintessential example of vector-based design software. It has been around forever. You can find tutorials and training materials on it everywhere. It has features galore. It's available for both Mac and PC for a monthly charge. You might choose Illustrator like you would choose a guitar over a tenor

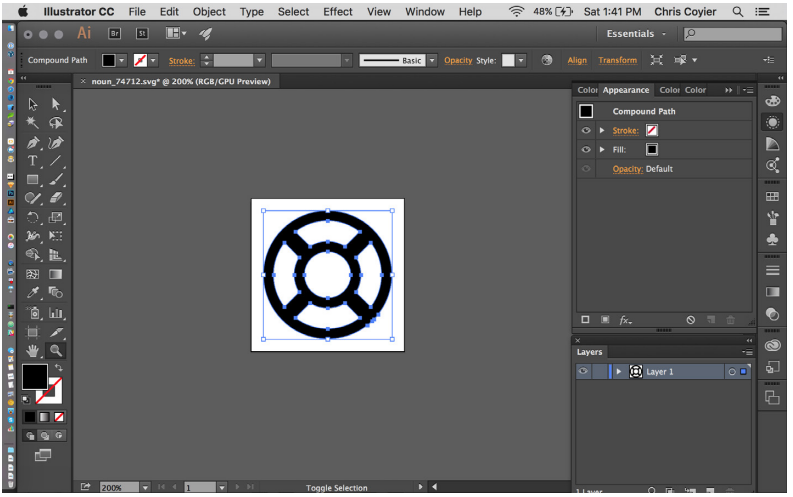

FIG 2.1: The Adobe Illustrator interface (http://bkaprt.com/psvg/02-01/).

banjo: it'll be a lot easier to find tutorials, documentation, and teachers for the guitar, because it's a more common and popular instrument than the tenor banjo.

Most relevant for us: Illustrator speaks SVG natively. You might think of the file format of Illustrator as .ai, but it can also be .svg. And not just in an "Export to SVG" kinda way—it's a "Save As" native format. (Unfortunately, going that route results in SVG that is not ideal for the web, but we'll cover how to address that in Chapter 5.)

When relevant, we'll be using Illustrator as the software of choice in this book, both because it's the program I know best and because it's the most widely used.

Sketch

Sketch is Mac-only software. It has been growing in popularity, seemingly fueled by folks yearning for a fresh take on screen-design software (and those who like the idea of software you just *buy* rather than pay for monthly).

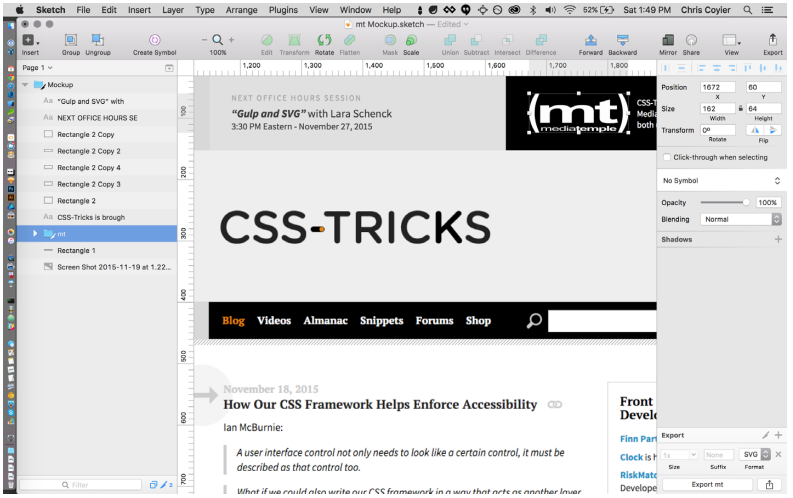

FIG 2.2: The Sketch interface (http://bkaprt.com/psvg/02-02/).

Most relevant for us: it has great SVG exporting features. You designate individual elements (or groups of elements) as "exportable" and then export them. You end up with cropped, immediately useable versions of what you exported. That meshes with a screen-design workflow pretty well, in my experience.

I use Sketch and like it, but if I'm producing a purely vector bit of artwork, I turn to Illustrator. For me, Sketch shines at layouts, and I love that it can output SVG assets from those layouts. But for vector editing, Illustrator wins hands down.

Inkscape

There are some compelling reasons to use Inkscape. The big one: it's free and open source. It works on Mac, PC, and Linux. SVG is its native file format. Bonus points: the Inkscape team contributes to SVG spec development.

I feel like a snob writing this, but I find the Inkscape interface disconcerting. On the Mac, it uses X11, which provides an

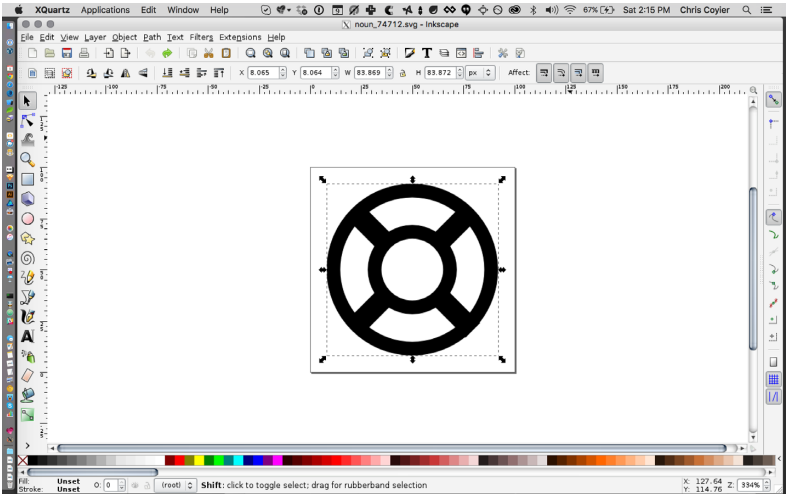

FIG 2.3: The Inkscape interface as run through X11 on a Mac (http://bkaprt.com/psvg/02-03/).

emulated non-native interface. This means that the entire UI (windows, menus, etc.) is not the normal operating system UI that Mac users (ahem, me) are used to </snob>. I hear on Linux it feels perfectly natural, and pretty close to natural on Windows.

MOBILE SOFTWARE

Autodesk Graphic

Graphic is a complete vector-editing app for iOS. As I was trying it, I was highly impressed that it featured all the tools and capabilities of desktop OS vector-editing software. It was a bit hard for me to get used to not having the fine-grained pointing abilities of a mouse, but that's likely just lack of practice. I imagine a skilled, pen-wielding artist might find having an interactive surface like the iPad pretty compelling.

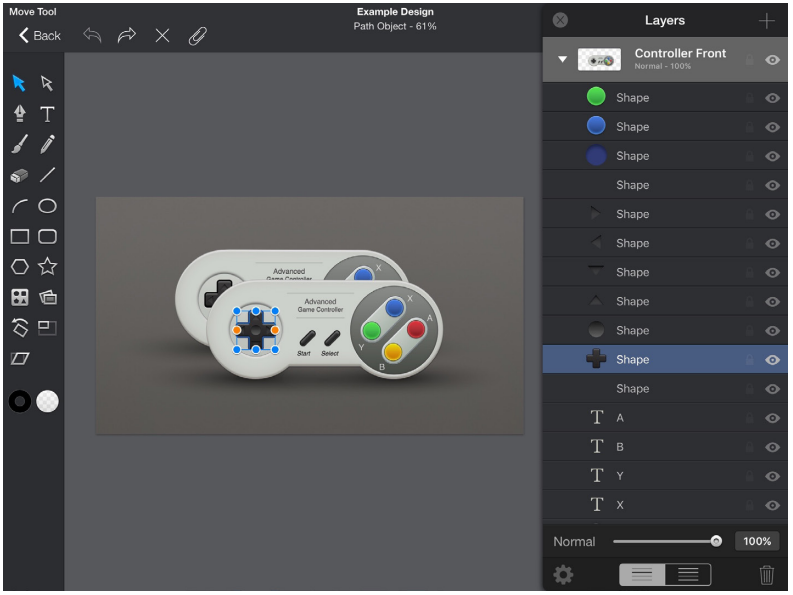

FIG 2.4: The Graphic interface on an iPad (graphic.com).

WEB SOFTWARE

SVG-Edit

Surely you could dig around and find little features that this app doesn't have, but it *does* have a surprising amount—like a layers system that allows you to put elements on top of (or below) other elements. Layer stacking can be tricky in SVG, since it doesn't have a native stacking system like `z-index` in CSS. Instead, SVG relies on source order: whichever elements come later in the source code are on top. SVG-Edit's layer system abstracts away this source-order shifting (http://bkaprt.com/psvg/02-04/).

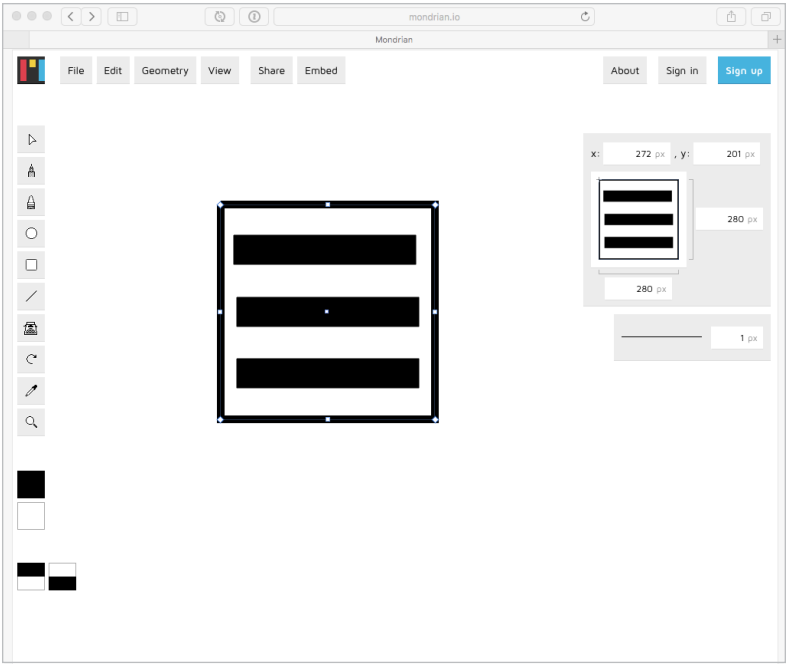

FIG 2.5: The Mondrian interface.

Method Draw

Method Draw is based on SVG-Edit (http://bkaprt.com/psvg/02-05/). It has a simple, clean, and well-considered design with some pretty great UI controls. Anything number-related is click-and-draggable, which makes for easy alterations to things like rotation, blur, and opacity (http://bkaprt.com/psvg/02-06/).

Mondrian

Mondrian is an open-source project that offers a similar set of features as the others (http://bkaprt.com/psvg/02-07/). One nice perk: it stores what you are working on automatically. When I

came back to Mondrian after a hiatus of several weeks, the icon I was working on in FIG 2.5 was waiting for me (mondrian.io).

It's great to see powerful, interactive, fast SVG software right in the browser. All three of these are free and open source. It's impressive stuff.

We now know a bit about the SVG format and how to use it on the web; we know about software that makes working with SVG easier and more practical. Now let's get into *actually* using it. SVG can help us with some of the crucial everyday work we do as front-end developers. It can solve real pain points and real problems—like how we create and implement icons on our sites.

3 BUILDING AN ICON SYSTEM

WE'VE BEEN USING ICONS on websites since the dawn of... websites. Styles come and go, but the functionality of icons is here to stay. They aid in quick visual differentiation and assist in conveying meaning—even across languages and cultures.

Take a look at a fairly simple page on GitHub.com (FIG 3.1).

All told, nearly two dozen icons there.

We *could* make each of those icons an image, like icon_pencil.svg, and use them in our HTML like this:

```
<li>
  <a href="">
    <img src="icons/icon_pencil.svg">
```

That works. There is nothing inherently wrong with using `img`. But it does mean that each unique image is a separate network request. That is, when a browser sees an `img` tag, it goes across the network to get that file and display it. The same is true for background images in CSS: every unique image referenced is a separate network request.

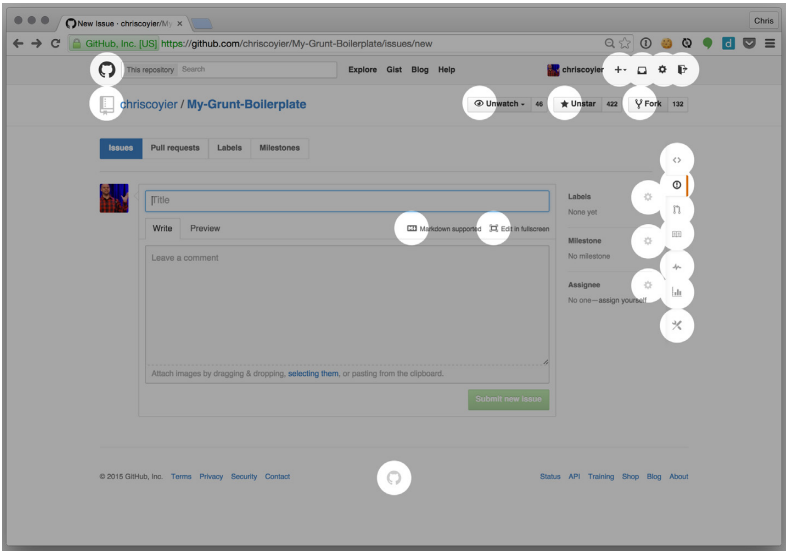

FIG 3.1: Screenshot of the New Issue page on GitHub.com with the icons highlighted.

When thinking about web performance, one of the first things to look at is reducing the number of network requests (sometimes called "HTTP requests," the protocol of the web—we'll call them network requests here). See Scott Jehl's *Responsible Responsive Design* or Lara Hogan's *Designing for Performance* (http://bkaprt.com/psvg/03-01/) for more on this.

Ideally, we could make *all* of our icons a single network request. That's one ingredient in building an icon system. And really, there are only two ingredients:

- Serve icons as a single resource to make a website faster.
- Make the system easy and convenient to use.

In fact, as front-end web development has been maturing, we have been solving this problem over and over again. Reducing the number of requests boosts performance to such an extent that we're almost always willing to complicate development

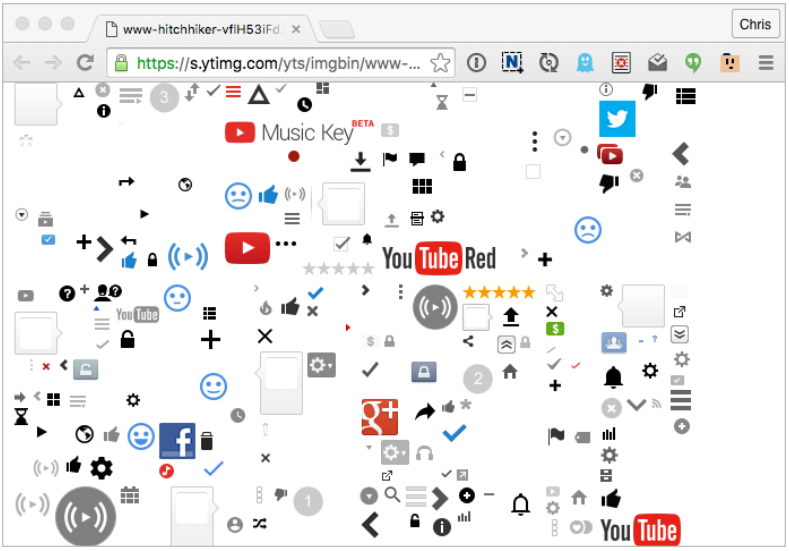

FIG 3.2: Example of a CSS sprite used by YouTube.com.

work in exchange. This is especially true on mobile, where latency is so much higher (http://bkaprt.com/psvg/03-02/).

One classic way to tackle an icon system—cleverly cribbed from video game development—is with CSS sprites. A CSS sprite is one big raster image with lots of smaller images placed onto it; YouTube puts this to effective use (FIG 3.2).

One big image means just one network request. To display one of the smaller images (like a single icon), we would make an element the exact size of the icon, use the large sprite image as the background-image in CSS, and then adjust the `background-position` to reveal only the smaller image.

The method is clever and effective. We can do the same thing with SVG: lay out a bunch of vector graphics on a single artboard, export it as one big SVG file, and do the same background shifting trickery.

But as long as we're going the SVG route, there's an even better way. Let's go through building an inline SVG system step by step so you can see how it works.

We're working inside one big SVG, so we'll start with this:

```
<svg>

</svg>
```

Between the opening and closing tags, we'll put the paths that do the actual drawing:

```
<svg>

  <!-- this draws the Twitter logo, one shape -->
  <path d="">

  <!-- this draws the CodePen logo, two separate
  shapes -->
  <path d="">
  <path d="">

</svg>
```

Then we'll wrap those paths in `g` tags. The `g` tag in SVG is like a `div` in HTML: it doesn't do much by itself other than group the things inside it. It's mostly useful for referencing and applying styles that can affect everything together.

```
<svg>

  <g id="icon-twitter>
    <path d="">
  </g>

  <g id="icon-codepen">
    <path d="">
    <path d="">
  </g>

</svg>
```

Then we'll wrap all of that in a `defs` tag. A `defs` tag essentially says, "SVG, don't try to actually draw any of the stuff inside this tag; I'm just defining it to use later." The SVG element itself will still try to render, though, so let's make sure it doesn't do that by shrinking it to nothing. Using `width="0"` and `height="0"` or `display="none"` is better than using CSS to hide the SVG; CSS takes longer to process, and `style="display: none;"` can have weird consequences, like failing to work at all on some iOS devices (http://bkaprt.com/psvg/03-03/).

```
<svg width="0" height="0" class="hide">
  <defs>
    <g id="icon-twitter">
      <path d="">
    </g>

    <g id="icon-codepen">
      <path d="">
      <path d="">
    </g>
  </defs>
</svg>
```

The chunk of SVG we've just built is our SVG icon system. We're going to be talking about this chunk of SVG a lot in this book, so let's give it a name: an *SVG sprite*.

As long as this SVG sprite is present in the HTML of our page, we can do this anywhere we want to draw one of those icons:

```
<svg class="icon icon-twitter" viewBox="0 0 100
  100">
  <title>@CoolCompany on Twitter</title>
  <use xlink:href="#icon-twitter" />
</svg>
```

This code snippet will draw that icon onto the page, just like an `img` would. One new thing here is the `viewBox` attribute, which we'll get to soon. The other new element is `use xlink:href`, which essentially says, "Go find the chunk of

SVG that has this ID and replace me with a clone of it." That's a beautiful thing right there. We can use `xlink:href` to draw any bit of SVG anywhere we like and repeat it as many times as we want without repeating actual drawing code.

You can see this in action in the footer on a previous design of CSS-Tricks, which had a `use`-based icon system (**FIG 3.3**).

These icons are properly accessible. They are nicely semantic. And—perhaps best of all—we can size, position, and style them easily.

```
.icon {
  /* make it match the font-size */
  width: 1em;
  height: 1em;

  /* make it match the text color */
  fill: currentColor;
}
```

These values are a pretty cool little trick! Instead of explicitly styling the SVG, we make it size and color itself to match the font properties. If we drop that SVG into a `button`, it will absorb the styles already happening in that button and style itself accordingly (**FIG 3.4**).

USING symbol

We can improve this setup a bit, though. There is another element in SVG that is perfect for icons: `symbol`. Think of `symbol` as "SVG within SVG."

Did you notice how we declared a `viewBox` on the `svg` where we drew the Twitter icon? That `viewBox` is specific to that Twitter icon. It defines the area in which that shape is drawn. It is essentially the coordinate system for the points and the aspect ratio. A `viewBox` whose value is `"0 0 100 100"` is really saying, "The grid starts at 0, 0 and goes to 100 along the x-axis and 100 along the y-axis." And that has to be correct for each icon, or it won't draw correctly.

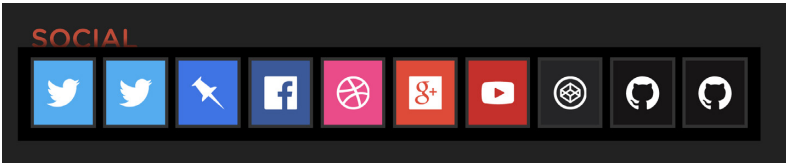

FIG 3.3: SVG icons in use in the CSS-Tricks.com footer (css-tricks.com). Note the repetitive shapes.

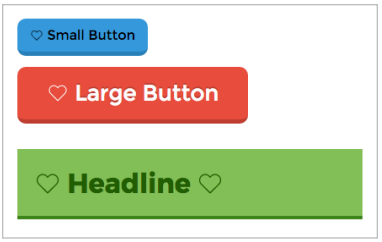

FIG 3.4: SVG icons automatically size themselves in proportion to the existing `font-size`.

We also used a `title` attribute to make sure we had basic screen-reader accessibility covered. We can shift the responsibility for both `title` and `viewbox` to the `symbol` element. Plus, SVG already knows not to draw `symbol` elements; it knows you are just defining them to `use` (actually draw) later.

Our chunk of SVG where we build the icon system now looks like this:

```
<svg width="0" height="0" class="visually-hidden">
  <symbol id="icon-twitter" viewBox="0 0 100 100">
    <title>@CoolCompany on Twitter</title>
    <path d="...">
  </symbol>

  <symbol id="icon-codepen" viewBox="0 0 200 175">
    <title>See Our CodePen Profile</title>
    <path d="...">
    <path d="...">
  </symbol>
</svg>
```

And then, when we go to draw the icon somewhere on our page, we can *do less* and *avoid a lot of repetition* if the icon is used in multiple places:

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

The result is exactly the same as with the initial snippet; it's just easier and less error-prone when implementing. In the second, the `use` element will draw the icon with the correct `viewBox` dimensions taken from the `symbol` element. We don't have to manually deal with `viewBox` anymore. Plus, the final `svg` in the document will contain the correct `title` and `desc`. See? Very awesome.

This still might look like a lot of code just to draw an icon, but it's comparable to any other icon-drawing technique, especially when you consider what you get from this method. Semantically, our markup says, "This is an image icon." Screen readers can announce whatever we want them to ("@CoolCompany on Twitter," for example), or nothing at all if that's more appropriate. We also get resolution independence. We get the ability to style the icon through CSS, and every other advantage of inline SVG, because that's exactly what we're using. Imagine trying to get all that functionality from, say, icon fonts—it would be difficult for some features and impossible for others!

Let's look at another example of real-world usage. The curvy tabs on a previous design of CSS-Tricks would have been difficult to do in CSS without substantial trickery (FIG 3.5). And speaking of trickery, I've seen some pretty over-the-top demos of drawings using just HTML and CSS. There is, of course, nothing wrong with a good bit of fun and experimentation, but I usually can't help but think, "You know, this is what SVG is for."

With inline SVG, we just define that shape as a `path` once, make it a `symbol`, and `use` it over and over as needed, even altering the style. And even though we were talking about an icon system here, note that this idea works for *any* SVG you want to use as part of a system.

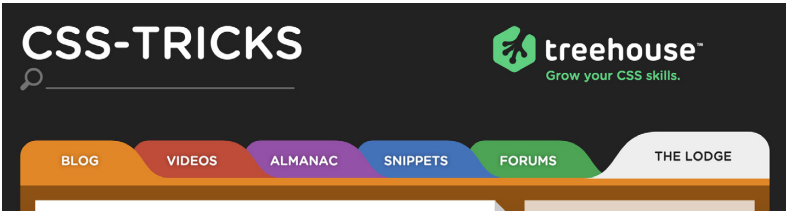

FIG 3.5: Each of these curved tabs is the same `path` used over and over again with a different fil`l.

A NOTE ABOUT use AND CSS STYLING

Let's say you have a `symbol` with two `path`s in it. Each `path` has a class, like `.path-1` and `.path-2`. In your CSS, you can target and style those with no problem.

```
.path-1 {
  fill: red;
}
.path-2 {
  fill: yellow;
}
```

You can use the `symbol` wherever you like:

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

Now say you'd like to create a variation on the normal styling. So you use the `.path-1` class on the `svg` as part of the selector.

```
.icon .path-1 {
  fill: green;
}
```

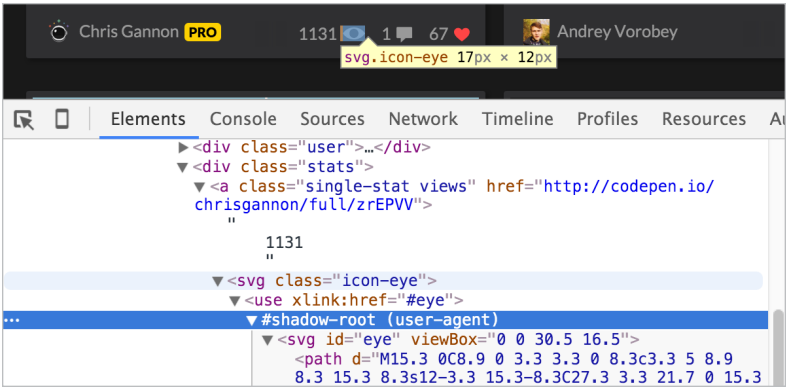

FIG 3.6: The `#shadow-root` highlighted here in Chrome's developer tools is the start of the Shadow DOM for this `use` element. At this writing, in order to see the Shadow DOM in DevTools, you need to go to Settings > General and make sure "Show user agent shadow DOM" is checked.

Sadly, that doesn't work. `use` is kind of magical in that it clones the elements inside the `symbol` and moves them into what is called the Shadow DOM. You can look at the Shadow DOM by using the developer tools in the browser (**FIG 3.6**).

I don't want to take us down a rabbit hole, but the Shadow DOM forms a boundary that CSS selectors don't go through. This may be remedied in the future. I've seen no less than three ideas presented on how CSS selectors might be able to penetrate the Shadow DOM. But for now, there's no good way to do variations on the same `symbol`—it's best just to create another copy and style it differently. GZIP will be your friend there, as it does a great job of compressing repetitive text.

There's a trick for getting two configurable colors per `use`, though. Imagine this symbol:

```
<symbol id="icon">
  <path d=" … " />
  <path fill="currentColor" d=" … " />
<symbol>
```

We'll use it more than once:

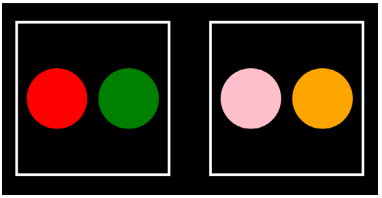

FIG 3.7: Two configurable colors per `use`. Remember that there is no limit to the number of colors an icon can use; this is just for variations on the exact same icon used more than once.

```
<svg class="icon icon-1">
  <use xlink:href="#icon" />
</svg>

<svg class="icon icon-2">
  <use xlink:href="#icon" />
</svg>
```

The first `path` has no fill of its own. It will default to black unless its parent `svg` has a fill set, in which case that color will cascade down to it. That's color number one. With the second path, we've used the keyword `currentColor` for the fill (you could do this from CSS as well). Whatever `color` you set on the parent `svg` will be used for the fill on it. That's color number two.

It goes like this:

```
.icon-1 {
  fill: red;       /* Color #1 */
  color: green;    /* Color #2 */
}

.icon-2 {
  fill: pink;      /* Color #1 */
  color: orange;   /* Color #2 */
}
```

I intentionally took a roundabout way to get here. Sorry about that! I wanted to sneak in some practical SVG learnin'. `symbol` is definitely the way to go when building an SVG system.

WHERE DO WE PUT ALL THOSE symbols?

Our SVG sprite will work great if we put it in the HTML. It won't show up visually, and we'll easily be able to use all the icons it contains.

Often, we work with HTML in the form of templates—perhaps our CMS gives us template files to work with, or we're using some other framework that compiles templates into the final HTML output. Dropping a large chunk of SVG code in our templates may feel unwieldy or messy. There is likely some way to "include" the SVG in that template, making things more manageable. So where do you put that "include"?

Well, putting the SVG sprite at the top of the document (that is, right after the opening `body` tag) is the least buggy way to proceed, because some browsers (like Safari on iOS 8.1) won't render the `use` at all if the sprite is defined after it. That's a shame, because putting the sprite at the bottom of the document would be better for performance, in the spirit of "deferred loading" of things deemed less important than the primary content.

Also, using `file_get_contents()` is safer than `include()` in PHP, as XML like `<?xml` will have trouble going through the PHP parser. Say you're using PHP:

```
</head>

<body>

  <!-- include your SVG sprite here -->
  <?php echo file_get_contents("svg/sprite.svg"); ?>

  ...

  <!-- use your sprite, here -->
  <a href="/" class="logo">
    <svg class="logo">
      <use xlink:href="#logo" />
    </svg>
  </a>
```

The advantage of doing it this way is that there are *zero* network requests for the icons. That's pretty nice!

But there are a couple of downsides to this approach. It isn't perfectly compatible with server-side HTML caching. If the site does that, every single page will have this identical chunk of SVG in it, which isn't very efficient ("bloated" cache, you could call it). Plus, browsers will need to read and parse that whole chunk of SVG with every single page load, before it gets to the (arguably more important) content below.

Including the sprite server-side also doesn't take advantage of browser caching very well. Browser caching happens when a browser holds onto a file so a network request doesn't need to be made for it. If we move our chunk of SVG into a separate file, we can tell the browser to cache it, and we've solved both problems.

We can make sure the browser caches the sprite by putting the file path in the `xlink:href` attribute of the `use` element. That's right, it doesn't have to be just an `#identifier`; it can be a file path (or *external source*):

```
<svg class="icon icon-twitter">
  <use xlink:href="/svg/sprite.svg#icon-twitter" />
</svg>
```

To ensure the file is cached by the browser, you could put this code in the site's `.htaccess` file (assuming an Apache server):

```
# Ensure SVG is served with the correct file type
AddType image/svg+xml .svg .svgz

# Add browser caching to .svg files
<IfModule mod_expires.c>
  ExpiresActive on
  ExpiresByType image/svg+xml "access plus 1 month"
</IfModule>
```

Remember that if you alter the icons in any way, you'll need to make sure fresh copies are used, not the stale ones in the cache. One way to do that is to use a URL parameter on the file

path, making it look like a different file to the browser. I use a simple system like this sometimes:

```
<?php $version = "1.2"; ?>

<svg class="icon icon-twitter">
  <use xlink:href="/svg/symbols.svg?version=<?php
  echo $version; ?>#icon-twitter" />
</svg>
```

You can get as clever with that as you want to, but the basic concept holds true. Don't write this one off as too much work! Remember that network requests are slow, so browser caching is one of the most effective ways to speed up a website.

There's something you need to keep in mind when it comes to `use` with an external source: the cloned elements no longer share the same DOM, the way they do with internal references. You'll be able to inspect the Shadow DOM and it will appear the same, but you can't, for instance, style a path directly from CSS on the parent document, or `querySelectorAll` for an element from JavaScript on the parent document. You would have to do those things from inside the external source itself. You can, however, apply a fill color to the parent `svg` and have that cascade through the other elements and work, so `use` with an external source is still pretty great for icon systems.

THE FUTURE: FRAGMENT IDENTIFIERS AND HTTP/2

You know how we've been referencing icons in the `use` element with the hash symbol, like `#icon-twitter`? That's a *fragment identifier*. Fragment identifiers are so useful that they are essentially the basis for this entire icon system trip we're on. But SVG fragment identifiers are coming to HTML and CSS as well!

For instance, in an image tag in HTML:

```
<img src="sprite.svg#logo(viewBox(0,0,32,32))"
  alt="Logo">
```

Or in a `background-image` in CSS:

```
.logo {
  background: url("sprite.
  svg#logo(viewBox(0,0,32,32))");
}
```

Support for fragment identifiers used this way in HTML and CSS isn't quite here. If it were, you could base an entire icon system on it, I reckon, as it nicely solves the one-request issue. I delved into the details of fragment identifiers in an article for CSS-Tricks (http://bkaprt.com/psvg/03-07/).

Speaking of one-request, even that is destined to be a relic of the past. HTTP/2, the next iteration of HTTP1.x (which we're all using now), has many features that will actually encourage leaving individual icons to individual network requests. It's pretty technical stuff, but, for example, HTTP/2 connections will remain open and multiple; compressed requests can happen in parallel. That means multiple requests aren't much costlier than single requests, if at all. It also means that icons can be cached individually, so changing a single one doesn't invalidate the cache on the entire set. Concatenating requests will become a bad idea instead of a good one!

MAKING `use` WORK WITH AN EXTERNAL SOURCE

No version of Internet Explorer (and some older versions of WebKit browsers) currently supports file paths in `use xlink:href=""`. These browsers support `use` just fine, but only with inline SVG and fragment identifiers (e.g., `#hash`), not URLs. Luckily, there's a way around that!

To polyfill this external reference feature (that is, make it work in browsers where it doesn't), try Jonathan Neal's SVG

for Everybody script (http://bkaprt.com/psvg/03-04/). You could include the script on the page like this:

```
    <script src="/js/svg4everybody.min.js"></script>

</body>
```

More likely, though, you'd concatenate it into the rest of the scripts you're loading on your site.

The script is tiny (less than 1 KB before compression), simple, and clever. Here's how it works: it runs a test to see if the current browser doesn't support externally sourced SVG. If that's the case, it performs a network request for the SVG and then drops in what is needed, as if we were using straight-up inline SVG. If the browser supports externally sourced SVG, the script does nothing.

Here's a common example. You put this in your HTML:

```
<svg class="logo">
  <use xlink:href="/svg/sprite.svg#logo" />
</svg>
```

If SVG for Everybody determines that the current browser supports this, it leaves it alone. If SVG for Everybody determines that the current browser does *not* support this, it converts the earlier code into this:

```
<svg class="logo">
  <path id="logo" d="..." />
</svg>
```

The second snippet is more widely supported by browsers.

To perform this sleight of hand, SVG for Everybody tests the browser's User-Agent string—typically a frowned-upon practice in our field because of how easy it is to screw it up or get the opposite result of what you need. Here's the test SVG for Everybody runs:

```
/Trident\/[567]\b/.test(navigator.userAgent) ||
  (navigator.userAgent.match(/AppleWebKit\/(\d+)/)
  || [])[1] < 537
```

The result will be either true or false, generally matching older versions of Internet Explorer and WebKit-based browsers. If it comes back true, the script will do the rest of its magic. I don't think the test is so offensive in this particular case because:

- A false positive would mean it still works (`use` gets replaced).
- A false negative doesn't necessarily mean the browser doesn't support the external linking.

But by no means is it perfect. The false negative scenario can pose a problem sometimes. For instance, Android 3 through 4.2 could benefit from this script, but it returns a false negative, so the script doesn't do anything. Bummer.

AJAX FOR THE SPRITE

Another solution is to do an Ajax request for the SVG sprite and drop it onto the page. This is a nice fix, as the SVG sprite can still be browser cached, and it completely eliminates the external asset issue. It's just a little trickier than you might think.

Elements in the DOM have their own *namespace* that essentially tells the browser how to handle them. SVG and HTML have different namespaces. When HTML is read and parsed by the browser and it finds an `svg` tag, it automatically applies the correct namespace. But if you just append an `svg` tag into the DOM yourself, that SVG won't have the correct namespace by default, and thus it won't behave like an SVG element.

To create a new SVG element in JavaScript with the proper namespace, you need to do something like this:

```
var svgElement = document.createElementNS(
  "www.w3.org/2000/svg", "svg");
```

That goes for the SVG element and any child SVG element you append.

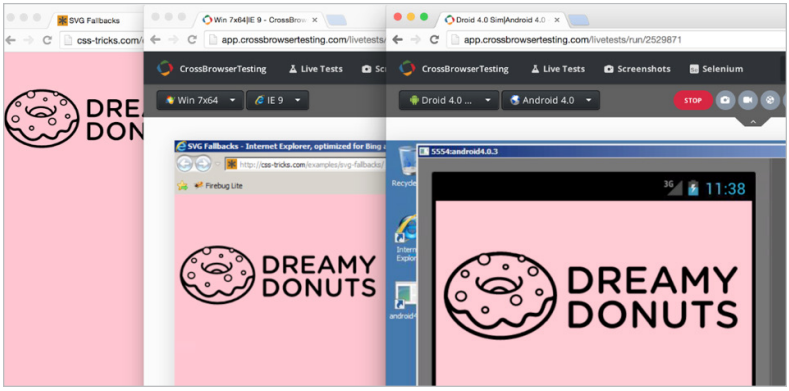

FIG 3.8: It's working! Cross-browser examples of SVG inserted via Ajax (http://bkaprt.com/psvg/03-05/). Shown in OS X Chrome, Windows 7 / IE 9, and Android 4.0 Native Browser.

If we create a normal HTML element, like a `div`, and then append the SVG sprite onto that `div`, the parser will run on it and ensure that everything is namespaced correctly.

So here's how we can use Ajax for our SVG sprite and add it to the page properly:

```
var ajax = new XMLHttpRequest();
ajax.open("GET", "svg/sprite.svg", true);
ajax.onload = function(e) {
  var div = document.createElement("div");
  div.innerHTML = ajax.responseText;
  document.body.insertBefore(div, document.body.
  childNodes[0]);
}
ajax.send();
```

Or you can ensure that the response type is a document, which also gets everything in good shape to append, and drop that in without needing the `div`.

```
var ajax = new XMLHttpRequest();
ajax.open("GET", "svg/sprite.svg", true);
```

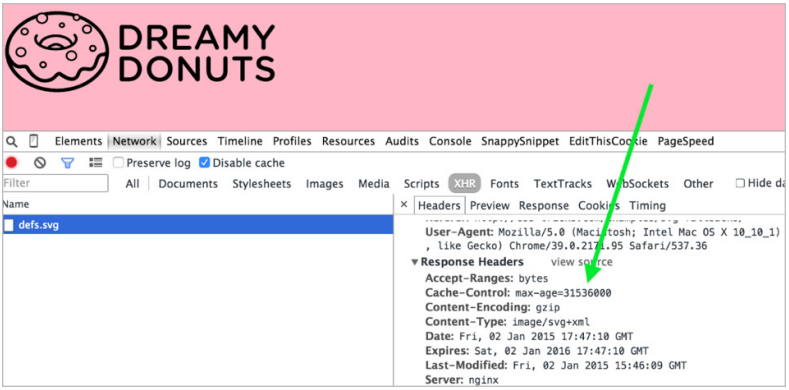

FIG 3.9: Proof of proper browser cache headers being served. Note the `Cache-Control: max-age=31536000`.

```
ajax.responseType = "document";
ajax.onload = function(e) {
  document.body.insertBefore(ajax.responseXML.
  documentElement, document.body.childNodes[0]);
}
ajax.send();
```

If you're using jQuery to help with Ajax, the response will automatically be a document. You need to force that into a string before inserting it into the `div`, like this:

```
$.get("svg/sprite.svg", function(data) {
  var div = document.createElement("div");
  div.innerHTML = new XMLSerializer().
  serializeToString(data.documentElement);
  document.body.insertBefore(div, document.body.
  childNodes[0]);
});
```

We can see that the caching is in effect by looking at the Cache-Control header of the XHR request for the SVG file (**FIG 3.9**). Lookin' good.

	ICON FONT	INLINE SVG
Are the icons vector?	They are, but browsers consider the icons text, so they are anti-aliased as such. Anti-aliasing can be great for text legibility, but can lead to icons not being as crisp as you might expect.	Yep. Straight-up crisp vector shapes.
Do you have CSS control over the icons?	You have the same control over them as you would with any text. You can control the size with `font-size`, the color, apply shadows with `text-shadow`, and the like.	You can do anything you could do with an icon font, only you have ever more control. You can select individual parts of the SVG and style them differently (i.e. multi-color icons) and you also get SVG-specific CSS properties, like `stroke` and `filter`.
Positioning	It can be frustrating to position a font icon. Because it's text, the space the icon occupies depends on `font-size`, `line-height`, `vertical align`, `letter-spacing`, and `word-spacing`. And also the box that glyph was designed in and the kerning information in the font.	SVG is just a box that is positioned and sized as told. Although you do need to be aware of how SVG scales with things like viewBox.
Failure possibilities	An icon font might fail because 1) it's being loaded cross-domain without the proper cross-origin headers, 2) a network or server failure serving the font file, 3) there is some weird bug that dumps the `@font-face` and shows a fallback font instead, or 4) a surprising browser doesn't support `@font-face`.	Inline SVG is right in the document. If the browser supports it, it displays it.

FIG 3.10: When you compare an icon font to an inline SVG icon system point for point, the SVG system emerges the clear winner.

SVG VS. THAT OTHER ICON SYSTEM: ICON FONTS

The reason I'm taking the time to go through all of this is because inline SVG really does make for an awesome icon system. It's even better, dare I say, than another extremely popular way to approach an icon system: icon fonts. An icon font is a custom font file that you load in CSS through an `@font-face` declaration; the font's glyphs are actually icons.

Icon fonts have one distinct advantage over an inline SVG icon system: they are supported in even really old versions of Internet Explorer, which only supports SVG in versions 9 and up. Every other comparable feature between the two comes down squarely on the side of SVG (http://bkaprt.com/psvg/03-06/).

I hate to make this a battle, but hey, let's make this a battle (FIG. 3.10).

What makes an inline SVG icon system even more persuasive: you don't have to build that SVG sprite yourself! You can make your computer do it for you. That's coming up in the next chapter.

We just covered a lot of ground. Think about the inline SVG icon system we reviewed and everything that went into it: All those `symbol` elements that wrap the shapes that draw the icons. The `viewBox` that defines the drawing area for them. Making sure they have unique IDs. Confirming they have accessibility tags that have been done correctly. Hiding the SVG element itself.

If we wanted to, we could do all that work by hand. There's value in a little elbow grease and hard work, right? No. There isn't. Not when these tasks can easily be handled by a computer. So next we'll look at build tools—automated processes that will make quick work of these things and simplify our lives as developers.

4 BUILD TOOLS

A BUILD TOOL is a name for any bit of software that facilitates tasks that help us build websites. The most common tasks for a build tool are things like compiling code with preprocessors (see Dan Cederholm's *Sass for Web Designers*) or compressing and concatenating other assets (see Scott Jehl's *Responsible Responsive Design*). But a task can be anything. *Computer! Move these files over here! Rename them like this! Add this comment at the top of them!* You know, things that computers are faster and better at than humans.

One thing we can have a build tool do for us is create an SVG sprite—that chunk of SVG `symbol`s I introduced in the last chapter—automatically from a folder of separate SVG images. It makes this workflow possible:

1. Create and edit an SVG file in Illustrator as needed.
2. Watch the build tool do its magic by automatically adding the SVG file we just created or edited to the SVG sprite.
3. Use the icons easily in your HTML.
4. Revel in the majesty of beautiful icons on your site.

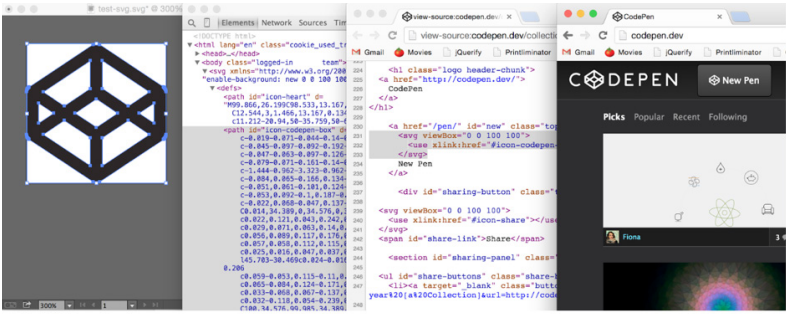

FIG 4.1: An automated SVG design workflow can start in GUI software like Adobe Illustrator and end on the actual website without too much manual work in between.

You no longer have to do any of the tedious manual construction of SVG in the special format needed (as we did in the last chapter); that all happens almost instantly as you work. And, unlike fat-fingered humans, the build process won't make any mistakes while doing it.

ICOMOON

One of my favorite build tools for SVG is IcoMoon (http://bkaprt.com/psvg/04-01/). Its website has a very simple interface: a big grid of icons. First, select the icon you want, and then hit the Generate SVG & More button (**FIG 4.2**).

Then click the SVG Download button (**FIG 4.3**). You'll get a ZIP file that includes the SVG sprite. That's a build tool!

The sprite, called `svgdefs.svg`, is in the root of that folder. It's a production-ready sprite file, and you can use it in any of the ways covered in the last chapter.

You aren't limited to the icons on the IcoMoon site; you can import your own. You can also create an account so that you can save your projects, making it easy to come back and add/remove/adjust icons and reexport them.

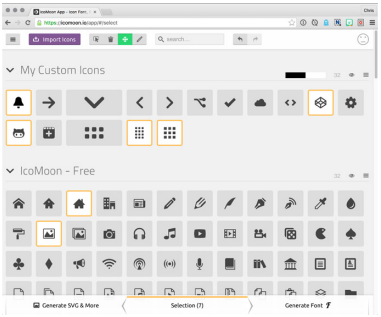

FIG 4.2: The IcoMoon interface.

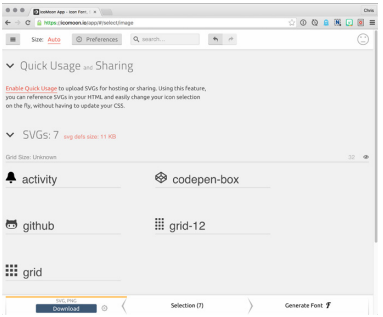

FIG 4.3: SVG exporting from IcoMoon.

GRUNT

IcoMoon has almost no learning curve and does a great job, but you can still level up. Let's take a look at using Grunt. It's a little more complex, but I promise you can handle it. I wrote an article that helped a lot of people get started with it called "Grunt For People Who Think Things Like Grunt Are Weird And Hard" (http://bkaprt.com/psvg/04-02/), which I'd recommend as a primer.

In a nutshell: you simply configure tasks for Grunt to do, run it, and it does them. *Grunt, preprocess my CSS! Grunt, minify my JavaScript! And in our case: Grunt, turn my individual SVGs into an SVG sprite!* (Grunt likes it when you are very clear and commanding like that.)

Note that Grunt can't run any of these tasks by itself. That's what plugins are for. The one we're going to use is grunt-svgstore (http://bkaprt.com/psvg/04-03/). Its sole purpose is spriting SVG. Once we've installed the plugin, we can configure Grunt to do what we want. We do this with Gruntfile.js, which lives in your project's root folder:

```
module.exports = function(grunt) {
  grunt.initConfig({
    svgstore: {
      default: {
        files: {
          "includes/defs.svg": ["svg/*.svg"]
        }
      }
    }
  });
  grunt.loadNpmTasks("grunt-svgstore");
  grunt.registerTask("default", ["svgstore"]);
};
```

And now in plain English:

Hey, Grunt. I'm going to keep all of my individual SVG images in a folder called "svg". When I type "grunt" into the command line, find all of those images and process them into an SVG sprite. Name that sprite defs.svg *and put it in a folder called "includes".*

Leveling up a little more, we can make that happen *automatically* whenever an SVG file is added, removed, or changed. The grunt-contrib-watch plugin is just for this. It watches the files and/or directories you tell it to watch in your projects, and runs grunt tasks when the files change in any way.

We'll tell it to watch our icons folder for SVG files and run svgstore when they change.

```
module.exports = function(grunt) {

  grunt.initConfig({

    svgstore: {
      default: {
        files: {
          "includes/defs.svg": ["svg/*.svg"]
        }
      }
    },
```

```
    watch: {
      svg: {
        files: ["icons/*"],
        tasks: ["svgstore"],
        options: {
          livereload: true
        }
      }
    }

  });

  grunt.loadNpmTasks("grunt-svgstore");
  grunt.loadNpmTasks("grunt-contrib-watch");

  grunt.registerTask("default", ["watch"]);

};
```

You'll kick things off by typing `grunt watch` into the command line at the project's root folder.

See the `livereload: true` option as part of the configuration for that watch task? That's the icing on the cake here. If you have the LiveReload browser extension installed and turned on, your browser will refresh after the task is finished (http://bkaprt.com/psvg/04-04/). That means you can pop open an SVG in Illustrator, make edits, save it, and the browser will automatically refresh, immediately showing you the changes right on your site. That's a dang delicious design workflow if I've ever seen one.

ANOTHER APPROACH: GRUNTICON

I would be remiss not to mention Grunticon as a build tool for an icon system (grunticon.com). Grunticon is a Grunt plugin just like grunt-svgstore is, but it takes an entirely different approach to SVG icons. It still takes a folder full of .svg files and combines them for you, but it combines them into a stylesheet containing

a bunch of class declarations that set a `background-image` for the icon. Like this:

```
.icon-cloud-sync {
  background-image: url("data:image/
  svg+xml;charset=US-ASCII,%3Csvg%20 …");
  background-repeat: no-repeat;
}
```

The SVG images are converted into a *data URL* and put directly into the stylesheet. We'll cover data URLs momentarily, but in a nutshell: a data URL is literally the SVG itself, specially encoded and turned into a long string right inside the URL. All the drawing information is right there; no network request is required to go get anything else. In that sense, the stylesheet is your sprite, because all the icons are combined into one request and can be used on demand.

The Grunticon approach has a couple of advantages:

- It handles fallbacks for you! (That deserved an exclamation point.) Grunticon gives you everything you need, including a fancy detection script, to serve icons that work everywhere. It loads an entirely different stylesheet with PNG versions, if needed.
- It's inherently automated, forcing you to have a system in place for icons.

It also has some drawbacks:

- You only get some of the advantages of SVG, like scalability.
- The elements aren't in the DOM.
- You can't style elements with other CSS, meaning you'll need duplicates for even differently colored versions. The duplicates don't hurt file size as much as you would think, since GZIP is great at repetitive text, but it's still harder to maintain.

Grunticon 2 mostly takes care of that disadvantage (http://bkaprt.com/psvg/04-05/). You can use an attribute to tell it to inject inline SVG:

```
<div class="icon-cart" data-grunticon-embed></div>
```

Grunticon will work some magic and inject the inline SVG of that icon for you, as long as the browser supports it. It requires a little DOM injection that you wouldn't need if you started with inline SVG, but it allows for variations and all the fancy powers inherent to inline SVG.

SVG icons at Lonely Planet

Here's a real-world example for you (FIG 4.4). When he was working for Lonely Planet, Ian Feather blogged about the virtues of switching from an icon font to SVG icons (http://bkaprt.com/psvg/04-06/). His article covers many of the problems inherent to icon fonts and the specific strengths of SVG icons in an evenhanded way, including counterarguments where appropriate. Lonely Planet has a Grunticon-powered system in place as I write.

A word on data URLs

Converting images into data URLs has long been a little performance trick that can be used on websites. The idea is that all of the information for the image is right there, so there's no need for a network request. You can do that with SVG, too. Here's an example of an `img`:

```
<img src="data:image/svg+xml;charset=UTF-8,
  <svg ... > ... </svg>">
```

You could put the whole SVG syntax right in there. That would be weird, though, since in that case you'd probably just use inline SVG. It makes more sense in CSS:

```
.icon {
  url("data:image/svg+xml;charset=UTF-8,
    <svg ... > ... </svg>");
}
```

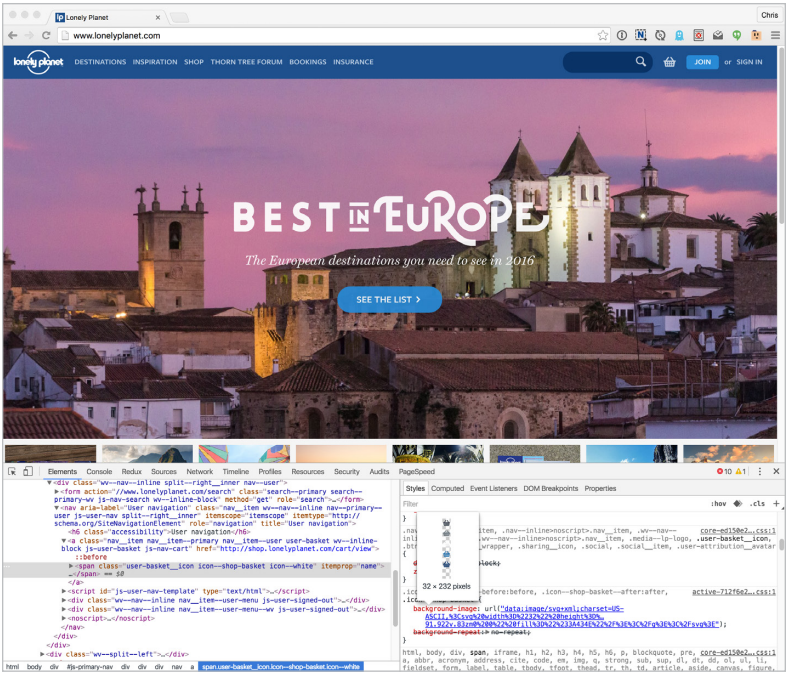

FIG 4.4: Icons at LonelyPlanet.com powered by Grunticon.

That will actually work in some browsers, but it's invalid and rightfully fails in compliant browsers. It's not the data URL that poses a problem, though—it's the angle brackets. The trick is to URL-encode those angle brackets and spaces (as Grunticon does) and it will work just fine:

```
.icon {
  background: url("data:image/svg+xml;
    charset=UTF-8,%3Csvg%20 ...");
}
```

The most common way you tend to encounter data URLs, though, is in the Base64 encoding:

```
.icon {
  background: url("...");
}
```

Base64 is typically used as an encoding format because it's safe. It uses only sixty-four characters, none of which are angle brackets or any other character that could be interpreted weirdly anywhere the data string could be used. The result, though, is an encoded string that is larger than the original. URL encoding does that, too, but because URL encoding ends up changing fewer characters, it retains SVG's fairly repetitive syntax, and thus lends itself better to compression.

GULP

Gulp is another very popular task runner (http://bkaprt.com/psvg/04-07/). Luckily for us, the svgstore plugin has a Gulp version as well (http://bkaprt.com/psvg/04-08/). After you've gone through the easy setup (follow the steps on the Gulp website), you'll tell Gulp what to do via gulpfile.js, in your project's root folder.

Here's an example that does exactly what our Grunt example did:

```
var gulp = require("gulp");
var svgstore = require("gulp-svgstore");

gulp.task("svgstore", function () {
  return gulp
    .src("icons/*.svg")
    .pipe(svgstore({
      inlineSvg: true,
      fileName: "sprite.svg",
      prefix: "icon-"
     }))
    .pipe(gulp.dest("includes/"));
});
```

Just type `gulp svgstore` into the command line, and sprite.svg will be created for you.

We can level up here as well. With Grunt, we added the watch task to help with our workflow. That's possible with Gulp, too, but rather than repeat ourselves, let's take the opportunity to cover some other things a build tool can be useful for.

Let's say we're working on a project and it comes up that we need a new icon—a downward-pointing arrow in a circle, say. We head to the Noun Project and find the perfect image. We name it arrow-down.svg. We drop it into our icons folder, and let Gulp integrate it into our sprite. Beautiful.

The icons in this project are often set next to text within buttons, like this:

```
<button>
  Download
  <svg class="icon">
    <use xlink:href="#icon-arrow-down" />
  </svg>
</button>
```

In our stylesheet, we set the text color for the button, and try to make sure that the SVG picks up the proper `fill` color:

```
button {
  color: orange;
}
button svg {
  fill: currentColor;
}
```

That should work great. But when we look at the site, we see that our icon is black (**FIG. 4.5**)! What the heck?!

We investigate our arrow-down.svg file and find this:

```
<svg ... >
  <path fill="#000" d="..." />
</svg>
```

FIG 4.5: The icon is black, even though we're trying to set the fill color in CSS (http://bkaprt.com/psvg/04-09/).

See the default `fill` attribute (`#000`) on the `path`? CSS can override that—quite easily, in fact. A *presentational attribute* like this isn't like an inline style on HTML elements, which can only be overwritten by powerful `!important` values. Presentational attributes on SVG elements are overwritten by any CSS that targets the element directly. They have a CSS specificity value of zero, as it were.

If we did this in our CSS instead, it would have worked:

```
path {
  fill: currentColor;
}
```

But as we know, `path` is just one of many SVG elements. We don't want to get into the mess of naming them all in CSS. It's much easier to set the SVG element itself and let the `fill` cascade through the other elements. And the `fill` will cascade through the other elements, unless there is a `fill` attribute on them, like we just encountered.

The solution? Get rid of that dang `fill` attribute! To be fair, there normally isn't anything wrong with `fill` attributes; in fact, they allow for multicolor icons, which is one of the strengths of using SVG for icon systems. But in this case, we want it gone. Black is also the default `fill` color, so it's especially useless to us.

One way to strip that `fill` attribute is just to open the SVG file and remove it. Even Illustrator is smart enough to leave black-filled shapes without a `fill` attribute alone, so it won't put it back in case you do a bit of manual editing.

But we're talking build tools here. Let's make our `fill`-attribute stripping part of our Gulp setup and let it happen auto-

matically. That makes it impossible to screw up, which is one of the things that's so appealing about build tools.

If you've ever worked with jQuery, you know that removing attributes from elements is trivially easy. We could select all elements that have a `fill` attribute and then remove it.

```
$("[fill]").removeAttr("fill");
```

If only we could do that with Gulp. Er, wait. We can! Cheerio is an implementation of jQuery for the server and it has a Gulp plugin (http://bkaprt.com/psvg/04-10/). We can run that exact line of code from within Gulp, which I think is just *so cool*.

While we're at it, let's add in SVG optimization, too, so you can see how easy it is for Gulp to "pipe" from one task to the next. Here's everything all together.

```
var cheerio = require("gulp-cheerio");
var svgmin = require("gulp-svgmin");
var svgstore = require("gulp-svgstore");

gulp.task("svgstore", function () {
  return gulp
    .src("icons/*.svg")
    .pipe(svgmin())
    .pipe(svgstore({
      fileName: "sprite.svg",
      prefix: "icon-" }))
    .pipe(cheerio({
      run: function ($) {
        $("[fill]").removeAttr("fill");
      },
      parserOptions: { xmlMode: true }
    }))
    .pipe(gulp.dest("includes/"));
});
```

Group hug.

FIG 4.6: With the fill attribute removed from the path, the color cascades in like we want it to (http://bkaprt.com/psvg/04-11/).

OTHER BUILD TOOLS

Grunt and Gulp aren't the only players in the build tools market. Other projects let you do the same kind of thing but have slightly different approaches. Check out Broccoli (http://bkaprt.com/psvg/04-12/), Brunch (brunch.io), or whatever else happens to be the build tool du jour. You may prefer the way one works over another. As long as it saves you time and effort: awesome.

WHAT ELSE CAN BUILD TOOLS DO FOR US?

Another process ripe for automation with a build tool is optimization—as in reducing SVG images' file size without compromising their quality. Imagine every vector point in an SVG file having five levels of decimal precision (e.g., `rect x="12.83734" y="28.48573" width="100.23056" height="50.42157"`). That's likely more precision—and a larger file size—than we need. Should we go in there and manually edit those values? Heck no! A build tool can do that for us.

In fact, precision adjustments are among many things a build tool for optimization can do for us. Let's turn to that next.

5 OPTIMIZING SVG

I've yet to come across any editing software capable of exporting SVG that is perfectly optimized for use on the web. That's a little strange, since the primary destination of SVG is the web. But SVG software has other things to worry about, too—like having compelling features for designers, and ensuring that you can open the files you created in the exact same condition in which you left them.

Imagine a guide in Adobe Illustrator: you know, one of those pale blue lines that help you position things on the canvas. There is no concept of a guide in the SVG syntax, but you'd hate to lose those guides every time you saved an SVG document in Illustrator. So, if you export an SVG from Illustrator and leave the "Preserve Illustrator Editing Capabilities" box checked, there will be some extra code in there defining where your guides go. In fact, there will be a *lot* of extra code in there, and it's so proprietary that it may as well not be SVG at all. However, if you want to make sure that the document opens in exactly the same condition you left it in, leaving that box checked is a good idea.

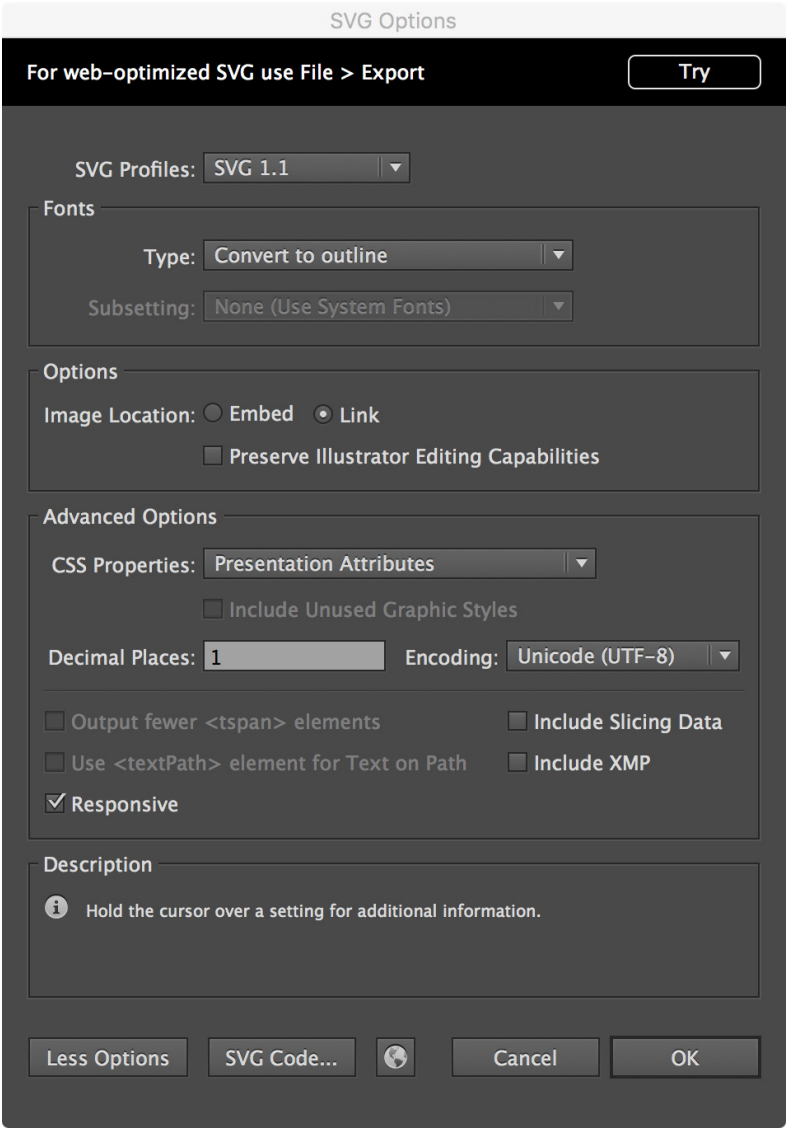

FIG 5.1: Detailed SVG Options screen in Adobe Illustrator.

A typical SVG workflow involves doing the design work in what you might think of as a master-editable version of an image, and then exporting and processing an optimized version to use on the web. Having a file-naming convention (icon-menu-master.svg and icon-menu.svg, say) or keeping the versions in separate folders is a good way to stay organized as you work.

In short:

- Always save a development version.
- When saving a file to use on the web, export a *copy*.

Let's continue using Adobe Illustrator as an example. If we were working on a new SVG master image and saving it for the first time, we would select "Save As…", choose "SVG" as the file format, and then confront a slew of options.

The options in **FIG 5.1** are what I would recommend for saving a master document. Remember: this file isn't intended for web use; it's intended to be a perfect copy, with all your designer conveniences left intact.

When it's time to export a copy you intend to use on the web, use File > Export and save a copy under a different name. You'll get a far more streamlined set of options (**FIG 5.2**).

I would recommend these options for most cases. This new copy will have a far smaller file size and is nearly ready for web use. It actually *is* ready for web use; it's just still not as well optimized as it could be. We'll look at some options for further optimization shortly, but first let's understand what SVG optimization actually does.

Below is some example SVG output from Illustrator. I should mention that during the creation of this book, both Illustrator and Sketch made huge strides in the quality of their SVG exports. The code shown here is an older example of code exported from Illustrator. It still serves as a good example for us, though, and there are lots of reasons you may continue to run into code like this. For instance: you may have an older version of Illustrator; this sort of code could appear in SVG you download from the internet; or you may be using different software that exports to SVG in a similar way.

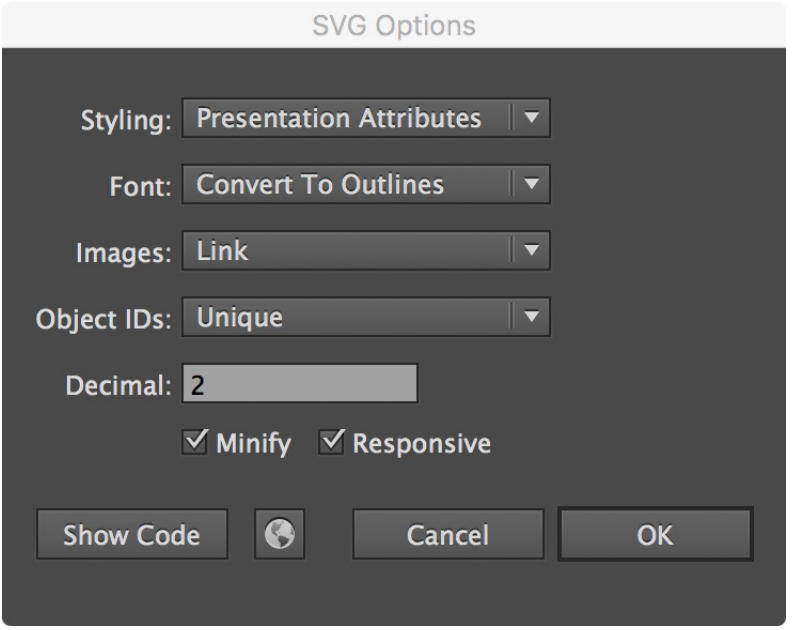

FIG 5.2: Export SVG Options from Adobe Illustrator.

Strikethrough text indicates code that is useless on the web and can safely be deleted:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 18.1.1, SVG Export
  Plug-In . SVG Version: 6.00 Build 0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/
  svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://
  www.w3.org/2000/svg" xmlns:xlink="http://www.
  w3.org/1999/xlink" x="0px" y="0px"
  viewBox="0 0 300 200" enable-background="new 0 0
  300 200" xml:space="preserve">
```

```
<rect x="45.7" y="49.3" fill="#FFFFFF"
  stroke="#000000" stroke-miterlimit="10"
  width="36.44237" height="34.23123"/>
</svg>
```

Even more of that code could probably go. I'm sure Illustrator does things the way it does for a reason, but that doesn't mean we shouldn't try to trim the fat. Remember: the smaller the file we serve, the faster it will be—and fast is good!

We could clean all of this up by hand, but this is stuff that lends itself to automation very well. And fortunately for us, there are some great optimization tools already out there.

SVGO

The leading project in automated SVG optimization is SVGO (http://bkaprt.com/psvg/05-01/). It's a Node.js-based command line tool. Meaning:

1. You have to have Node.js installed to use it—it only takes a second to install (nodejs.org).
2. You use it by typing commands into the terminal. For instance, `svgo dog.svg dog.min.svg` will optimize dog.svg into a new file, dog.min.svg (thus preserving the original). The terminal command `svgo -f .` will optimize the entire folder of SVG images you're currently in.

If we ran SVGO on the example SVG file we just showed, we would get this:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0
  300 200" enable-background="new 0 0 300 200"><path
  fill="#fff" stroke="#000" stroke-miterlimit="10"
  d="M45.7 49.3h36.4v34.2h-36.4z"/></svg>
```

65.5 percent smaller! (SVGO calls it "65.5 percent profit," which I love.)

INTEGRATING SVGO INTO GRUNT

Developer Sindre Sorhus has created a Grunt plugin for SVGO called grunt-svgmin (http://bkaprt.com/psvg/05-02/). Remember how, when we were using Grunt to build our icon system in the last chapter, any file we changed in our svg folder would instantly trigger the svgstore task and build our SVG sprite? Now, let's alter that to first *optimize* the SVG, and *then* build it into a sprite.

Since we already have Grunt set up, we'll run the following from the command line at our project's root folder to install the plugin:

```
npm install --save-dev grunt-svgmin
```

Next, we'll configure svgmin to take our entire icons folder and optimize all of the images it finds there into an icons-optimized/ folder. Then we'll reconfigure svgstore to build the sprite from the icons-optimized/ folder.

Finally, we'll alter our watch task to run svgmin first, then svgstore. Here's everything together:

```
module.exports = function(grunt) {

  grunt.initConfig({

    svgmin: {
      dist: {
        files: [{
          expand: true,
          cwd: "icons/",
          src: ["*.svg"],
          dest: "icons-optimized/"
        }]
      }
    },
```

```
    svgstore: {
      default: {
        files: {
          "includes/defs.svg": ["icons-optimized/
            *.svg"]
        }
      }
    },

    watch: {
      svg: {
        files: ["icons/*"],
        tasks: ["svgmin", "svgstore"],
        options: {
          livereload: true
        }
      }
    }

  });

  grunt.loadNpmTasks("grunt-svgstore");
  grunt.loadNpmTasks("grunt-svgmin");
  grunt.loadNpmTasks("grunt-contrib-watch");

  grunt.registerTask("default", ["watch"]);

};
```

As we did before, we'll kick things off by typing `grunt watch` into the command line at the project's root. Whenever any SVG is added, deleted, or modified, an optimized sprite will be generated.

An alternative would be to optimize the finished sprite. That may be more efficient, but having a copy of each optimized icon might be useful, too, so that you can inspect issues in an isolated way.

Another advantage to using SVGO through Grunt is that it provides easy configuration for turning SVGO plugins on and off. We can pass an `options` object in the configuration, like this:

```
svgmin: {
  options: {
    plugins: [
      { removeViewBox: false }
    ]
  },
  dist: {
```

The names of the plugins in the configuration, like removeViewBox, correspond to the plugin's file name: removeViewBox.js (http://bkaprt.com/psvg/05-03/).

SVGO AS A DESKTOP APP

Here's the deal with command-line tools like SVGO: I'm not afraid of them. None of us should be afraid of them. Designers often get accused of being afraid by people who feel more comfortable with the command line.

Listen. If an alternative interface does the same thing but is more comfortable for you and more in line with the rest of the work you're doing, why not use it? It's not a cop-out; it's a responsible choice. If it works for you, do it.

Take SVGO GUI, for example (http://bkaprt.com/psvg/05-04/). It's a program that gives you a drag-and-drop window you can use to optimize SVG. It just takes the files you give it and runs them through SVGO in the background for you.

SVGO-GUI has some limitations. For example, you can't tell it to rename the SVG as it optimizes it. You can't see a before-and-after visual. You can't specify which options (plugins) you want it to use. It just performs the default behavior of SVGO, and that's it.

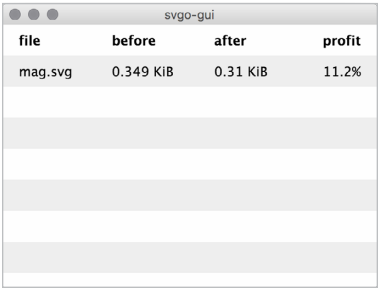

FIG 5.3: The simple interface from SVGO-GUI. This is the entire app.

BE CAREFUL

For the most part, I've had good luck with SVGO. But I've seen cases where what comes out is visually different than what went in. This seems to occur mostly with rather complicated SVG files, perhaps of dubious provenance. You know, like images downloaded from FreeVectorsAndSkeezyAdvertisements.com.

The relatively flat shapes we mostly work with on the web typically pose no problem. There are two main things to watch out for:

- **Reducing decimal precision too much.** Say you have a number like 3.221432 in an SVG file. Rounding that down to 3.2 will save bytes, but will also reduce precision. Maybe you won't notice the difference; maybe you will. You'll certainly notice it more the smaller the `viewBox` and the larger the viewport. (See Chapter 3 for more on `viewBox`.) Imagine a shape drawn in a coordinate system that goes from `0,0` to `100,100` (that's what the `viewBox` does). There are points like 31.875,42.366. The SVG is then drawn in a 100-by-100-pixel space (that's the viewport). Chopping off a few decimals probably won't hurt (rounding up to 31.9,42.4, for instance). Now imagine drawing the same graphic in a larger viewport, say 2000 by 2000 pixels. That extra precision might be more noticeable. Or say the same graphic is drawn in a `viewbox` of `0,0 10,10`. Again, extra precision may be needed for it to look as intended.

- **The removal of attributes.** SVGO can be configured to do things like remove all ID attributes. That might be useful; on the other hand, if you do it by accident, it might remove a vital reference for your CSS or JavaScript.

OTHER OPTIMIZATION TOOLS

SVGO is the most popular SVG optimization tool, I suspect, because it's open source on GitHub and built on the popular Node.js—and thus easy to incorporate into other tools and build processes. But it's not the only tool on the block.

Scour is a Python-based SVG optimizer; its creator, Jeff Schiller, calls it a "scrubber" (http://bkaprt.com/psvg/05-05/) built specifically to clean up the SVG output from early versions of Illustrator. Rather than plugins, Scour has configuration options that allow you to customize exactly what you want it to do to your SVG. One detail of note from the original project documentation:

> *My current stats show a Median Reduction Factor of 48.19% and a Mean Reduction Factor of 48.53% over 25 sample files for version 0.19 of scour, before GZIP compression.*

I suspect that those numbers are similar for any SVG compression tool. You're looking at *halving* the size of SVG files—and that's *before* GZIP compression, which is even more effective.

Another option is Peter Collingridge's SVG Optimiser (http://bkaprt.com/psvg/05-06/). It's not an open-source tool; you have to use it on the web. But he also offers a visual version of the tool, SVG Editor, where you can select different optimization options and see the results in a preview area (http://bkaprt.com/psvg/05-07/). That's nice, because there are no surprises that way. What you see is what you get (**FIG 5.4**).

In the same vein, Jake Archibald created an in-browser app for SVGO called SVGOMG (http://bkaprt.com/psvg/05-08/), which allows you to toggle and alter the settings and see the output immediately (**FIG 5.5**).

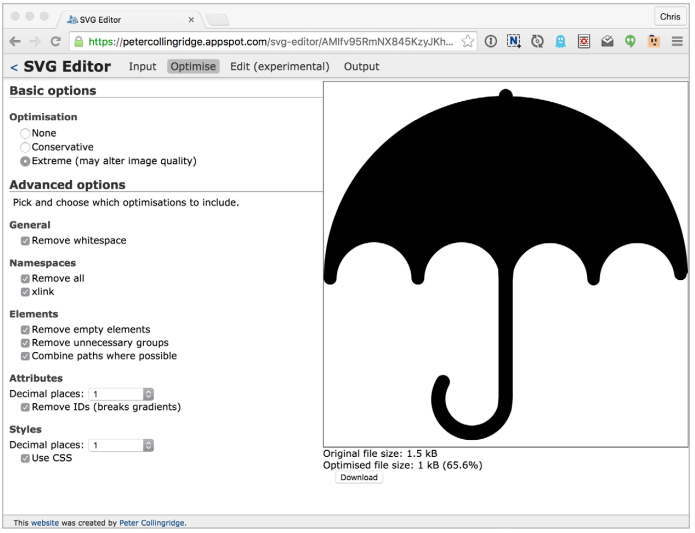

FIG 5.4: The SVG Editor options and preview screen.

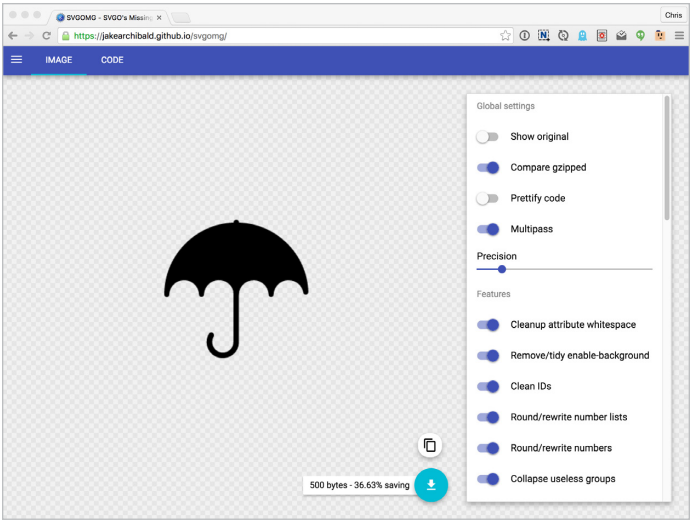

FIG 5.5: The SVGOMG options and preview screen.

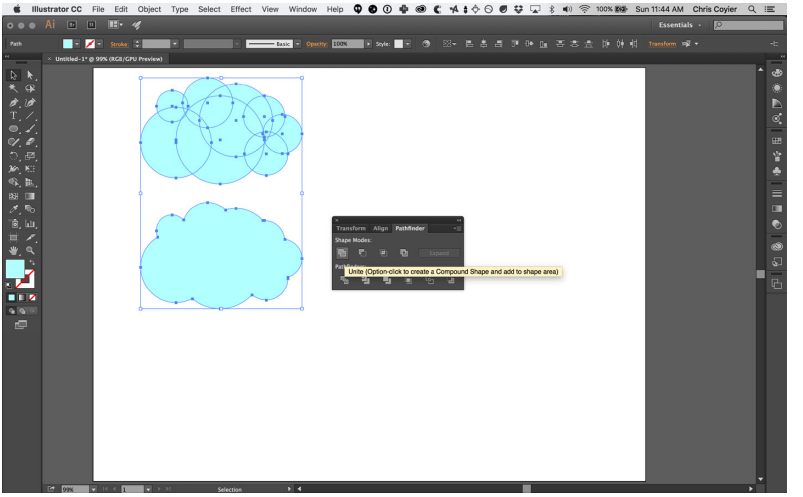

FIG 5.6: An example of manual simplification of SVG: combining shapes with Illustrator's Pathfinder tool.

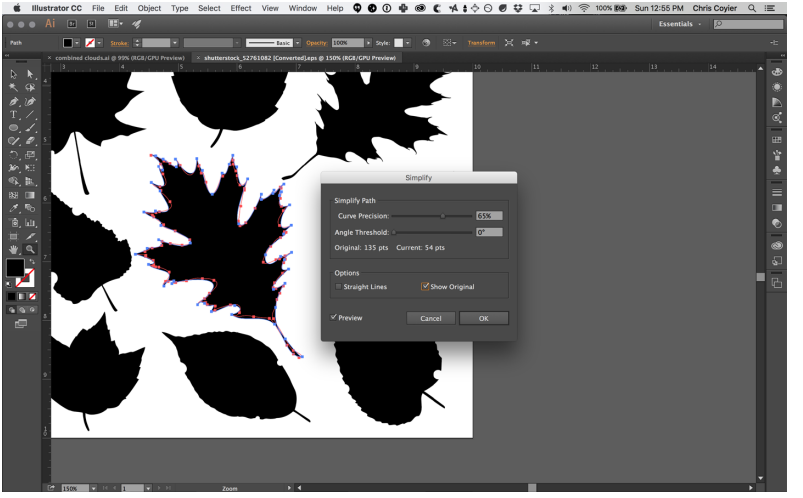

FIG 5.7: Manually simplifying paths in Adobe Illustrator with the Simplify options. Note that "Show Original" is checked here, permitting the original and simplified versions to be compared.

HAND-OPTIMIZING SVG

While there are big gains to be made with optimization tools, you can be your own optimization tool! (I just called you a tool. Sorry about that.) You shouldn't waste valuable time deleting metadata or trimming white space, but there's nothing wrong with spending time doing things that only *you* can do.

For instance, say you have an SVG image of a cloud, but it's actually made up of a bunch of overlapping circles (**FIG 5.6**). Using the Pathfinder palette in Illustrator to combine those shapes into a single path would likely result in a smaller file size.

Or, perhaps you have a shape that looks kinda grungy or roughed-up. There will probably be a lot of points making up those non-straight edges. Perhaps you can find places to remove some of those points without affecting the design too much, saving file size. Try playing with Object > Path > Simplify to see if you can reduce some of those points while keeping the image visually acceptable (**FIG 5.7**).

Now that we have optimization under our belts, let's move on to a subject near and dear to front-end developers' hearts: sizing and scaling. SVG is rather unique in how it does these things, so you'll want to get a handle on them.

6 SIZING AND SCALING SVG

YOU'LL PROBABLY WANT to exert some sizing control over any graphic you put on a website. *Hey! You! Logo! You should be this size:*

```
<img src="logo.png" class="logo" />

.logo {
  width: 220px;
  height: 80px;
}
```

And so shall it be.

But if the element you are resizing happens to be `svg`, the result might not be exactly what you expect. Sizing `svg` is a little more complicated than sizing an `img`. I'm not saying this to scare you. It's almost complicated in a *good* way, because it gives you more control and opens up some interesting possibilities.

Keep these two concepts in mind when you're working with the size of SVG images:

FIG 6.1: Viewport and `viewBox` in perfect harmony. This happens when you apply no width or height to the `svg` (either via attribute or CSS), or if you do, they match the aspect ratio of the `viewBox`.

- The viewport is simply the height and width of the element: the visible area of the SVG image. It's often set as `width` and `height` attributes right on the SVG itself, or through CSS.
- The `viewBox` is an attribute of `svg` that determines the coordinate system and aspect ratio. The four values are `x`, `y`, `width`, and `height`.

Say we're working with some SVG like this:

```
<svg width="100" height="100" viewBox="0 0 100 100"
  ... >

<!-- alternatively: viewBox="0, 0 100, 100" -->
```

In this case, the viewport and `viewBox` are in perfect harmony (**FIG 6.1**). The SVG will be drawn in the exact area it visually occupies.

Now say we double the width and height, like this:

```
<svg width="200" height="200" viewBox="0 0 100 100"
  ... >
```

FIG 6.2: With the viewport enlarged and `viewBox` kept the same, the graphic scales up to fit the viewport.

Will the `svg` just draw in a 100 by 100 space in the upper left side of the 200 by 200 element? Nope. Everything inside the `svg` will scale up perfectly to be drawn in the new, larger space (**FIG 6.2**).

The square aspect ratio still matches perfectly. That's why it's not particularly useful to think of the numbers anywhere in SVG as pixels, because they aren't pixels; they're just numbers on an arbitrary coordinate system.

What if the aspect ratios don't match, though?

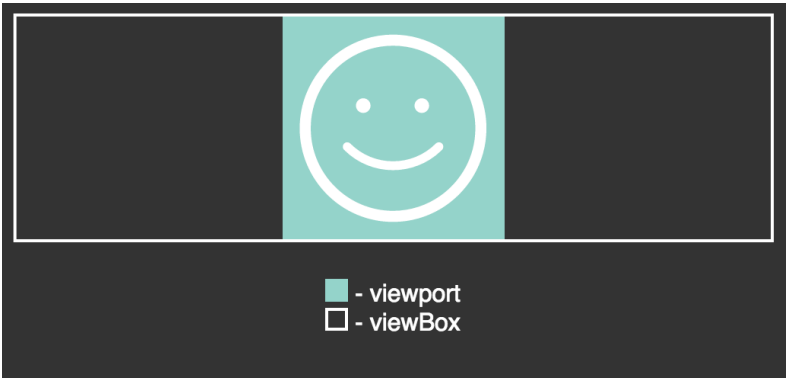

FIG 6.3: The viewport is enlarged, but no longer matches the aspect ratio of the `viewBox`. So by default, the image is drawn as large as possible without being cut off, and centered on the long dimension.

```
<svg width="300" height="75" viewBox="0 0 100 100"
  ... >
```

What happens now, by default, is that the SVG will draw itself as large as it can, centered along the longest dimension (**FIG 6.3**).

If you want to regain some control over this behavior, there's an attribute for the `svg` element that can help!

preserveAspectRatio

It looks like this:

```
<svg preserveAspectRatio="xMaxYMax" ... >
```

The `x` and `Y` parts of that value are followed by `Min`, `Mid`, or `Max`. The reason SVG normally centers in the viewport is because it has a default value of `xMidYMid`. If you change that to `xMaxYMax`, it tells the SVG: *Make sure you go horizontally as far to the right as you can, and vertically as far to the bottom as you can. Then be as big as you can be without cutting off.*

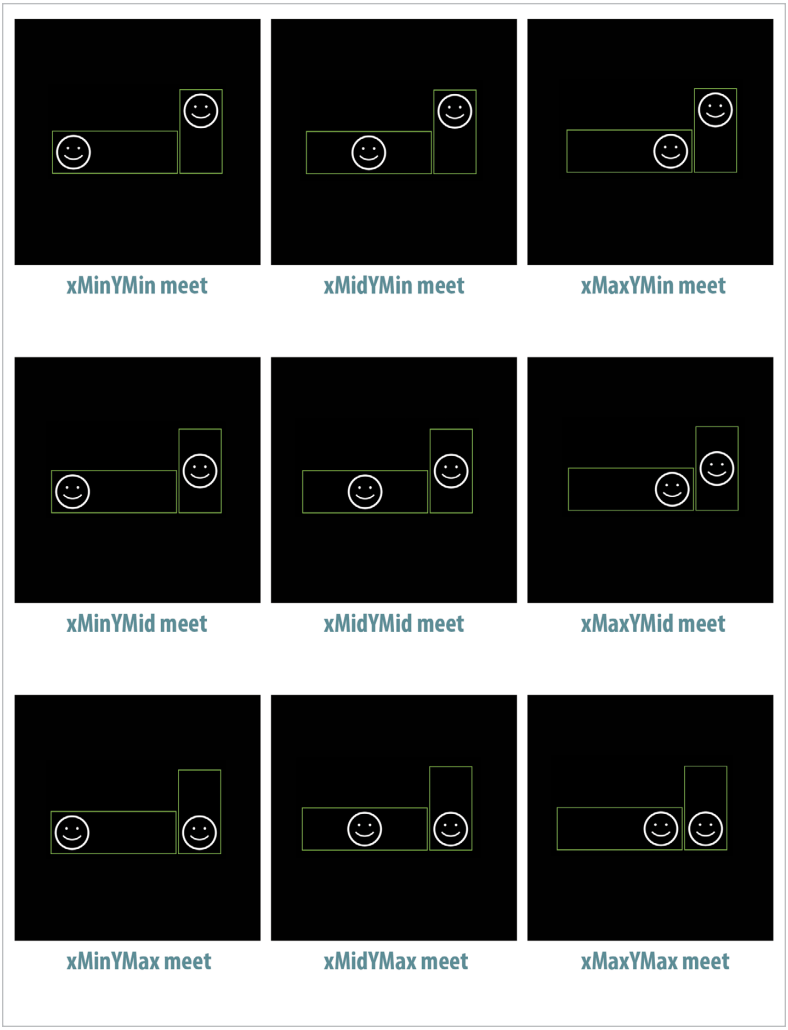

FIG 6.4: Examples of `preserveAspectRatio` values with `meet`.

FIG 6.5: Examples of `preserveAspectRatio` values with `slice`.

The "without cutting off" part is another aspect of `preserveAspectRatio`. The default value is `xMidYMid meet`—note the "meet." You can replace `meet` with `slice` to say instead: *Fill the area entirely; cutting off is okay*.

There are nine possible alignment values combined with `meet` (**FIG 6.4**).

There are also nine possible alignment values combined with `slice` (**FIG 6.5**).

I made a testing tool for playing with this idea (http://bkaprt.com/psvg/06-01/). Sara Soueidan also wrote an in-depth article on this subject, where she makes an excellent observation relating this idea to CSS (http://bkaprt.com/psvg/06-02/). The `background-size` property has two keywords it can take: `contain` and `cover`. The `contain` value means "make sure this entire image is viewable, even if you have to shrink it," which makes it just like `meet`. The `cover` value means "make sure this covers the entire area, even if you have to cut parts off," which makes it just like `slice`.

Even the alignment part of the value has a matching CSS counterpart: `background-position`. The default `background-position` is `0 0`, meaning "top left." That's just like `xMinYMin`. If you were to change that to, say, `50% 100%`, that would be like `xMidYMax`!

FIG 6.6 has some examples to make that connection a little clearer.

Remember: these aren't interchangeable bits of code; they are just conceptually related.

What if you want to throw aspect ratio out the window and have SVG scale to the viewport, like a raster image would? Turn `preserveAspectRatio` off (**FIG 6.7**)!

```
<svg preserveAspectRatio="none" viewBox="0 0 100 100">
```

Amelia Bellamy-Royds wrote a comprehensive article on scaling SVG, in which she covers things like the fact that `svg` can essentially contain other `svg` with different aspect ratios and behavior, so you can make some parts of an image scale and others not, which is pretty cool and unique to SVG (http://bkaprt.com/psvg/06-03/).

`preserveAspectRatio="xMinYMax meet"`	`background-position: 0 100%; background-size: contain;`
`preserveAspectRatio="xMidYMid meet"`	`background-position: 50% 50%; background-size: contain;`
`preserveAspectRatio="xMinYMax slice"`	`background-position: 100% 0; background-size: cover;`
`preserveAspectRatio="xMidYMid slice"`	`background-position: 50% 100%; background-size: cover;`

FIG 6.6: `preserveAspectRatio` values and the CSS properties they are similar to.

FIG 6.7: Example of `preserveAspectRatio="none"`. Poor little buggers.

Approaches to artboard sizing

When you draw SVG in editing software, that software likely gives you some kind of artboard to draw on. That's not a technical SVG term; it's essentially a visual metaphor for `viewBox`.

Let's say you're working with a whole set of icons for a site. One approach is to make all artboards hug each edge of the icon (**FIG 6.8**).

Here's a quick trick to get that artboard cropping in Illustrator: select the Artboard tool and then "Fit to Artwork Bounds" from the Presets menu (**FIG 6.9**).

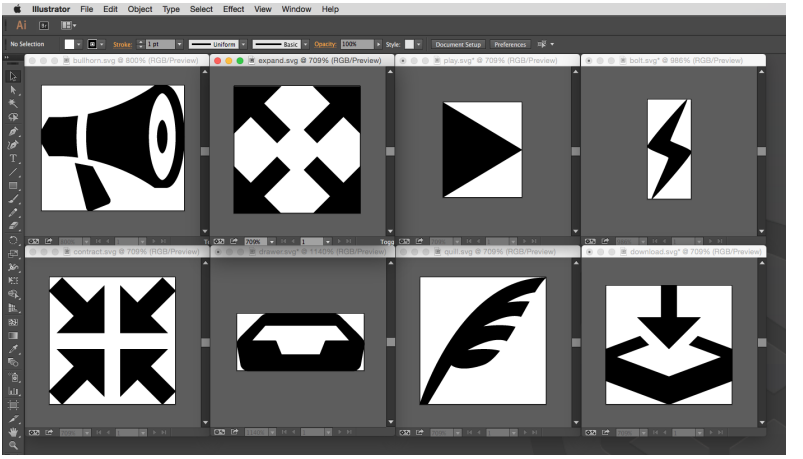

FIG 6.8: Example of graphics in Adobe Illustrator cropped to their edges.

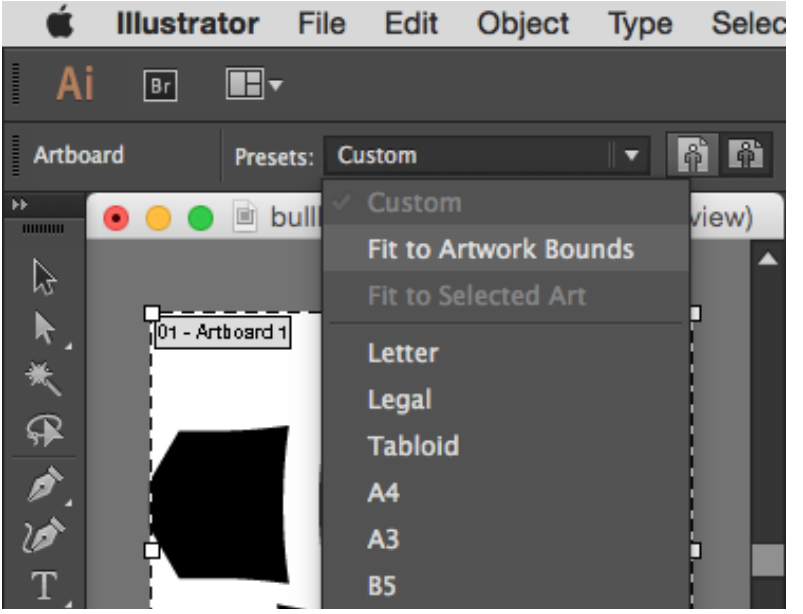

FIG 6.9: The menu option in Adobe Illustrator for resizing an artboard to the edges of a graphic.

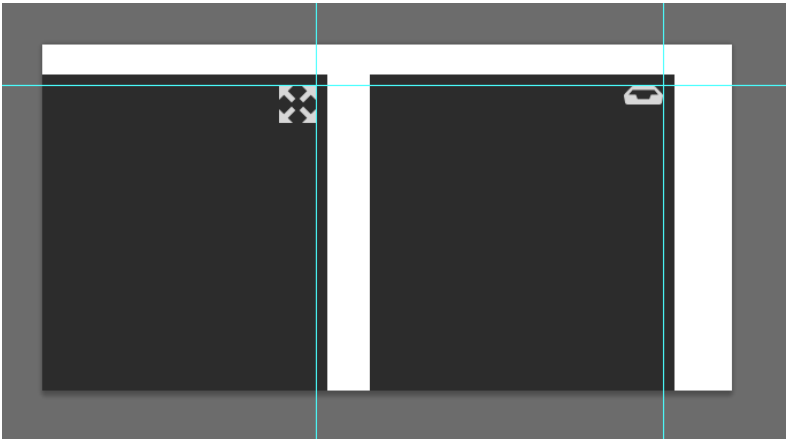

FIG 6.10: Icons aligning to edges without little bits of extra space you have to account for.

The big advantage to this technique is alignment (**FIG 6.10**). If you want to align any edge of any of these icons to anything else, that's easy to do. There is no mysterious space you need to contend with, or tweaky positional CSS.

```
.icon.nudge {
  position: relative;
  right: -2px; /* UGHCKKADKDKJ */
}
```

The big disadvantage to the cropping technique is relative sizing. Imagine you take the practical step of sizing your icon's width and height, like this:

```
.icon {
  width: 1em;
  height: 1em;
}
```

A tall, skinny icon will shrink to fit in that space and potentially appear awkwardly small. Or perhaps you're trying to have

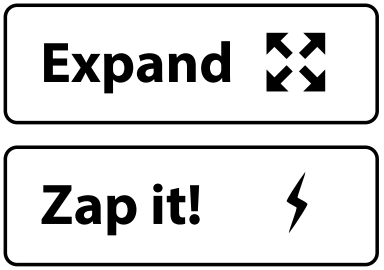

FIG 6.11: Two icons sized in the same square space within a button. The top one fits nicely, but the bottom one floats awkwardly in space.

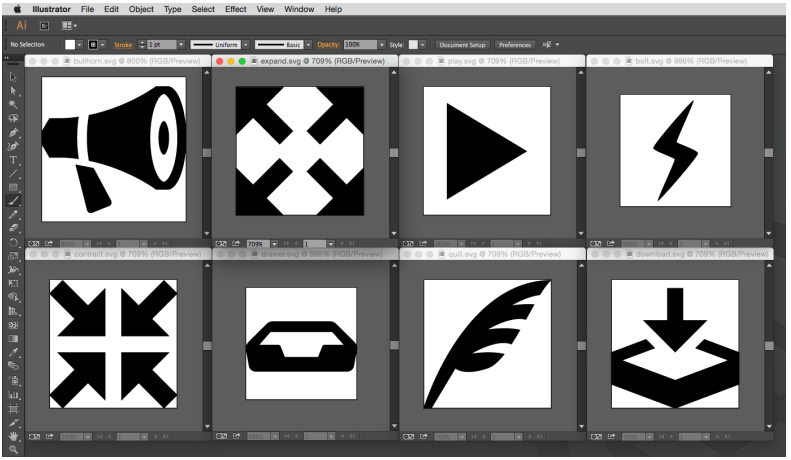

FIG 6.12: Example of Illustrator graphics whose artboards are equal in size.

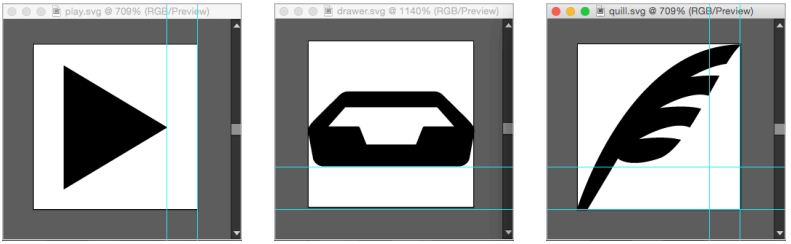

FIG 6.13: You can adjust icons' relative sizing, but that can make alignment more difficult.

an intentionally small star shape as an icon, except the star has a squarish aspect ratio and thus grows to fill the space, appearing bigger than you want it to.

Here's an example where two icons are sized identically as a square (**FIG 6.11**). The "expand" icon looks right at home, since it has a square aspect ratio to match. But the "zap it" icon has a tall and narrow aspect ratio, so it looks wimpy, like it's floating in the same square area.

The other approach here is to make consistently sized artboards (**FIG 6.12**).

The advantages and disadvantages are exactly inverse here. You might have alignment issues, because not all edges of the icons touch the edge of the `viewBox`, which can be frustrating and might require tweaking sometimes (**FIG 6.13**).

You won't have relative sizing issues, though, because the `viewBox` is the same for all of them. If any particular icon looks too big or small, you can adjust the artwork to bring it more in line with the set.

Since we're learning about sizing, now is the perfect time to bring up how SVG fits into the flexible world of responsive design.

RESPONSIVE SVG

One of the hallmarks of responsive design is fluid layout. Content—images included—is designed to fit its containers and the screen. If responsive design is new to you, Ethan Marcotte's seminal 2010 article on the subject is a fine place to start learning about it (http://bkaprt.com/psvg/06-04/). SVG jibes extremely well with responsive design:

- Responsive designs are flexible. So is SVG! It renders well at any size.
- Responsive web design is a philosophy of caring about how a website looks and behaves in any browser. Comparatively smaller SVG files and performance-responsible tactics like an SVG icon system can be a part of that.

But perhaps SVG's most obvious connection to responsive design is the possibility to react to CSS `@media` queries. Media queries move, hide, or show elements with CSS based on things like the width or height of the browser window. Those elements can be anything: sidebars, navigation, ads, what have you. They can be SVG elements as well.

Imagine a logo that displays different levels of detail depending on how much space is available. That's exactly what Joe Harrison was thinking when he created a really neat demo using well-known logos (http://bkaprt.com/psvg/06-05/, **FIG 6.14**).

On the web, we've always had the ability to swap out images with other ones. What's appealing here is that we aren't *swapping out* images; these are all the *same* image. Or at least they could be. That signature "D" all by itself could be the same exact "D" used in the most complex version of the logo. Easy-cheesy in CSS.

Say we organize the SVG like so:

```
<svg class="disney-logo">
  <g class="magic-castle">
    <!-- paths, etc -->
  </g>
  <g class="walt">
    <!-- paths, etc -->
  </g>
  <g class="disney">
    <path class="d" />
    <!-- paths, etc -->
  </g>
</svg>
```

This, by the way, is pretty easy to do in Illustrator (**FIG 6.15**). The groups and names you create there turn into IDs in the SVG output, and you can use those IDs to do the styling. Personally, though, I prefer using classes because they aren't unique (so you don't accidentally end up with multiple identical IDs on the page) and because classes have a lower and more manageable level of CSS specificity. It's easy enough to change IDs to classes with a bit of find-and-replace maneuvering in a code editor.

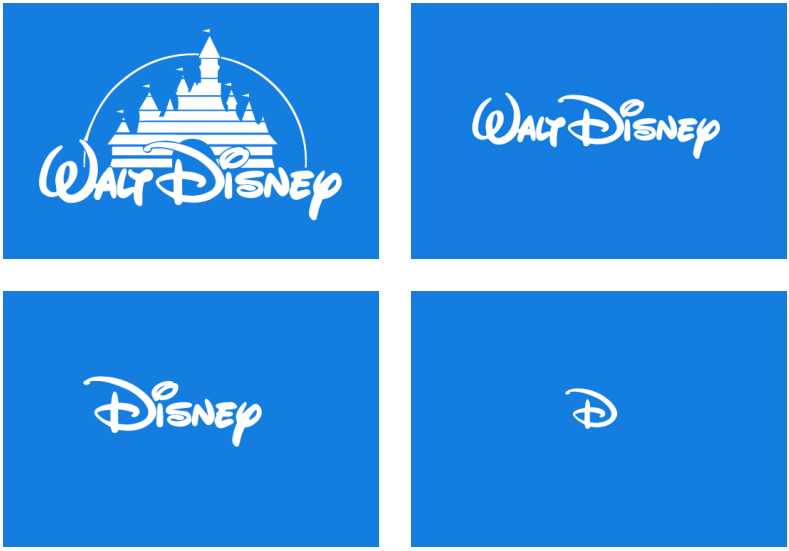

FIG 6.14: Joe Harrison's demo of the Disney logo at different sizes.

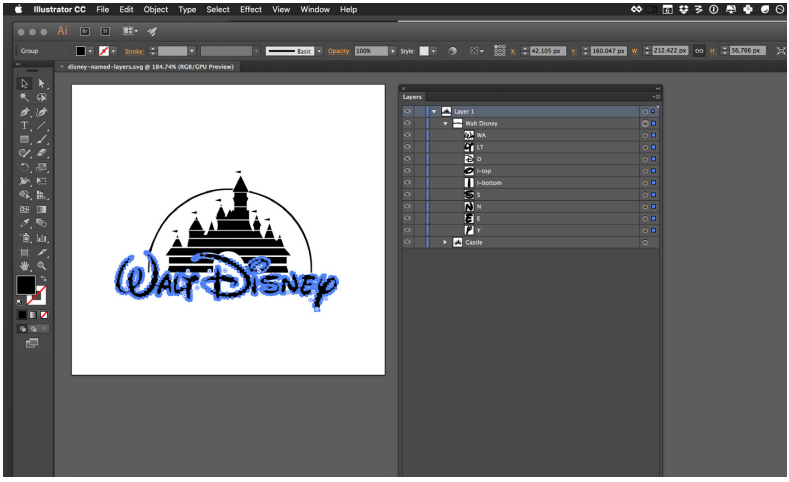

FIG 6.15: Named layers and named shapes in Adobe Illustrator.

The corresponding CSS could be something like this:

```
@media (max-width: 1000px) {
  .magic-castle {
    display: none;
  }
}
@media (max-width: 800px) {
  .walt {
    display: none;
  }
}
@media (max-width: 600px) {
  .disney > *:not(.d) {
    display: none;
  }
}
```

Mind you, this is a contrived example of hiding parts of the images at different breakpoints, but that's exactly how you would do it, along with some likely sizing adjustments. Anything you can do with CSS is on the table here. Perhaps some animation is appropriate at some breakpoints but not at others. Perhaps you change stroke sizes to beef up or trim down icons at different sizes. Perhaps you change some fill colors to simplify adjacent shapes.

And things can get even fancier! Depending on how the SVG is used, those media queries might actually be different. SVG used as `img`, `iframe`, or `object` has its own viewport. That means CSS *embedded inside of it* reacts to media queries based on that, rather than the whole browser window viewport. That means you would write, say, width-based media queries based on the width of the image, not of the entire page.

That's a very appealing idea: an element that arranges itself based on attributes of itself, rather than the page. *Am I this wide? Do this. Am I this tall? Do this.* That way, the SVG reacts to the situation it's in rather than the arbitrary document it happens to be part of.

As I write, this is referred to as "element queries" in CSS, but it doesn't actually exist yet in regular HTML/CSS. Once again, SVG is ahead of the curve.

GRADUATION INTO ANIMATION

Speaking of things SVG is good at, let's move into animation next. Everything we have been building on so far has prepared us for this. Hang on tight!

7 ANIMATING SVG

THERE ARE THREE distinctly different ways to animate SVG: with CSS, with SMIL, and with JavaScript. All of them are compelling and appropriate in different situations. SVG can accommodate anything from minor little UI-embellishing animations to full-on immersive, interactive, animated worlds.

ANIMATING SVG WITH CSS

Readers who have made it this far, I reckon, are well acquainted with CSS and find transitions and `@keyframe` animations fairly straightforward and comfortable.

Lucky for you, the same CSS techniques you use for animating and transitioning HTML elements also work on inline SVG elements.

Let me give you an example. Say we've designed a display ad in SVG (FIG 7.1). We want to slowly animate a series of clouds

FIG 7.1: A sample advertisement designed as an SVG.

FIG 7.2: Adding an extra set of clouds. (Level up: use use for the second set!)

horizontally across the ad to add a little visual interest. To do this, we'll duplicate some clouds we already have in our original artwork (**FIG 7.2**).

```
<svg>
  <g class="clouds">
    <path d=" ... " />
    <path d=" ... " />
    <path d=" ... " />
    <path d=" ... " />
  </g>
</svg>
```

The idea is that we can animate the whole row of clouds so that the duplicates are in the exact same position as the originals, but then instantly jump back to their initial position. That way, the clouds can infinitely animate.

```
.clouds {
  animation: move-clouds 15s linear infinite;
}
@keyframes move-clouds {
  to {
    transform: translateX(-50%);
  }
}
```

That's all there is to it! I've put up a demo on CodePen (http://bkaprt.com/psvg/07-01/).

So under what circumstances would you choose CSS to animate SVG?

1. You're doing most of your design work in CSS and want to keep it there. You like the syntax of CSS.
2. You're using inline SVG so you can keep the CSS together with your site's main stylesheets.
3. The animation is fairly simple and CSS is able to animate the things you need to animate (like positions, fills, strokes).
4. You're hoping to exploit the performance and browser optimizations of CSS animations.
5. You want to put a block of `style` inside the SVG, hoping it works in SVG as `img` or `background-image`. Mileage may vary. It works today in Chrome, but not in Firefox. SMIL animation works in both—that's coming up next (http://bkaprt.com/psvg/07-02/)!

Why might you *avoid* CSS animations on SVG?

1. CSS can't animate everything you might want to animate, like the position of an individual point. CSS can animate properties but not attributes. (Presentational attributes are properties.)

2. Your animation is fairly complex, and you need better tools than `@keyframes` or transitions. For instance, you might want to start one part of an animation when another ends, without having to match up timing manually.
3. You're experiencing buggy or broken behavior. Needless to say, there is quite a bit of this. Internet Explorer doesn't support CSS animations on SVG elements at all. Firefox doesn't support percentage-based `transform-origin`, which is sometimes vital to an animation. It's...a complicated love story (http://bkaprt.com/psvg/07-03/).

ANIMATING SVG WITH SMIL

SVG actually has its own syntax for animation built right into it. It's a part of SMIL (pronounced "smile"), which stands for *Synchronized Multimedia Integration Language* (http://bkaprt.com/psvg/07-04/). The `animate` tag is our primary weapon here. While it can get complicated, the syntax is pretty straightforward and declarative:

```
<circle r="30" cx="50" cy="50" fill="orange">

  <animate
     attributeName="cx"
     from="50"
     to="450"
     dur="1s"
     begin="click"
     fill="freeze" />

</circle>
```

In this example, an orange circle moves to the right by `400` when it's clicked, and then stays there.

FIG 7.3 shows the basic attributes of `animate` (http://bkaprt.com/psvg/07-05/).

`attributeName`	**Required:** Must be a valid attribute on the element the animation applies to.
`from`	**Optional:** If left out, the animation will start at the current values of that attribute.
`to`	**Required:** The value the attribute should be animated to.
`dur`	**Required:** The duration of the animation. Supports time values like `2s` or `1250ms`.
`xlink:href`	**Required:** If the animation is not within an element to be animated. Specifies the `#id` of the element.
`begin / end`	**Optional:** Specifies when the animation should begin or end, like on click.
`fill`	**Optional:** Specifies what happens when the animation completes. `"freeze"` means "remain as the animation ends". `"remove"` means to remove any effect the animation had. It's like `animation-fill-mode`.

FIG 7.3: Attributes of the `animate` tag.

FIG 7.4: Example of a star morphing into a check mark.

Using just these attributes, we can build the sort of animation that is outside the realm of possibility in CSS: shape-shifting. Imagine a form that slowly changes its shape over time:

We see color-changing and position-changing effects in animations on the web regularly, but shape-shifting is more unusual. Its relative scarcity stems from the fact that CSS just doesn't offer access to this visual attribute. But SMIL can do it!

Shapes are drawn from data in attributes on the shape elements themselves, for instance the `d` in `path d=""`. In the star-to-check-mark example, the shape element is a `polygon` and the

attribute is `points`, as in `polygon points=""`. So to animate the the shape, we animate the `points` attribute.

```
<polygon points=" ... shape 1 points ... ">

  <animate
    id="animate-to-check"
    attributeName="points"
    dur="500ms"
    to=" ... shape 2 points ... " />

</polygon>
```

We can trigger that animation with SMIL events, like we did in the orange circle example, but we can also trigger animations like this with JavaScript.

```
var ani = document.getElementById("animation-to-
  check");
ani.beginElement();
```

Why use SMIL at all? Here are the main reasons why you might reach for it:

- You can use it to animate things that CSS can't, like the shape of elements or the movement of elements along a path.
- You're working in the SVG directly and you like working there. Or you like the declarative syntax in general.
- The animation may work even when using SVG as `img` or `background-image`, where CSS or JavaScript will not.
- You want interactive features (hovers, clicks, etc.) without using JavaScript.
- You need timings that depend on other timings—*Start this animation when this other one ends, plus a second*—without having to keep everything in sync manually.

Are there reasons not to use SMIL? Sure:

- Blink has deprecated SMIL (http://bkaprt.com/psvg/07-06/), which means that, at some point, SMIL animations will likely stop working in Chrome and Opera. Microsoft browsers have never supported, and likely will never support, SMIL—and that's a big strike against it (http://bkaprt.com/psvg/07-07/).
- It's very repetitive. A single `animate` can only animate one element. Even if you want a second element to do the exact same animation, you'll need to duplicate it. GZIP is good at compressing repetitive code, so this poses no serious file-size concerns—but still, awkward.

Some great articles by Sarah Drasner can help us out here. In "Weighing SVG Animation Techniques (with Benchmarks)", she gathers some standard measures for comparing the performance of a number of SVG animation techniques (http://bkaprt.com/psvg/07-08/). The story still isn't perfectly clear—CSS and SMIL tend to perform the best generally, but you need to take care to ensure that you're getting hardware acceleration. Plus, there is evidence that JavaScript animation can outperform them all in some cases (http://bkaprt.com/psvg/07-09/). Sarah also wrote a guide detailing alternatives to SMIL features (http://bkaprt.com/psvg/07-10/).

Embedded animations

When you use SMIL, the animation code is embedded into the SVG syntax itself. The same is true if you use CSS to animate the SVG and put that CSS in a `style` block within the SVG code. When this is the case, depending on the browser, that animation might work even if you use the SVG as an `img` or `background-image`.

```
<img src="animated-graphic.svg" alt="Might just
  work!">
```

People often reach for GIFs when they need to show animation in an `img`, but GIFs can have large file sizes and (therefore)

create a serious performance burden. Animation embedded into SVG and used in an `img` is an attractive alternative, but unfortunately it's not supported nearly as widely as GIF's (for example, it doesn't work in Firefox).

This is not a comprehensive look at SMIL. There are many more attributes you can apply if you want to do more specific things. For instance, you can run an animation multiple times (e.g., `repeatCount="3"`), animate the numbers in specific blocks (e.g., `by="10"`), or control whether or not the animation is allowed to restart and when (e.g., `restart="whenNotActive"`).

There are even other animation elements, like `animateTransform`, that allow you to animate the `transform` attribute on SVG elements. You can't do that with `animate` alone (http://bkaprt.com/psvg/07-11/)—for instance, this doesn't work:

```
<animate
  attributeName="transform"
  from="rotate(0 60 70)"
  to="rotate(360 60 70)"
  dur="10s" />
```

Instead, you need to do this:

```
<animateTransform
  attributeName="transform"
  type="rotate"
  from="0 60 70"
  to="360 60 70"
  dur="10s" />
```

Another SMIL element opens up an interesting animation possibility: animating an element along a `path`. Imagine a little paper airplane floating across the screen, or a ball rolling down a hill. `animateMotion` is our friend here. It can animate any other SVG element along a `path` (but only a `path`; other basic shapes don't work). Here's a very simple example of animating an element in a circle (http://bkaprt.com/psvg/07-12/):

```
<svg viewBox="0,0 10,10" width="200px"
  height="200px">

  <path
    id="theMotionPath"
    fill="none"
    stroke="lightgrey"
    stroke-width="0.25"
    d="
      M 5 5
      m -4, 0
      a 4,4 0 1,0  8,0
      a 4,4 0 1,0 -8,0
      "
  />

  <circle r="1" fill="red">
    <animateMotion dur="5s" repeatCount="indefinite">
       <mpath xlink:href="#theMotionPath" />
    </animateMotion>
  </circle>

</svg>
```

That may not look tremendously simple at first blush, but remember that you probably won't be crafting that `path` by hand; you'll just be referencing it by ID.

This has long been nearly impossible in CSS. The most you could do was animate position values or get very tricky with `transform`s (see Lea Verou's post on this topic, for example [http://bkaprt.com/psvg/07-13/]). But new CSS properties can help: `motion-path: path()` and `motion-offset`. Blink already supports motion paths (http://bkaprt.com/psvg/07-14/), perhaps motivated by the SMIL deprecation. You can take the `path` data and use it directly, which makes moving a SMIL `path` animation to CSS quite easy! Here's how:

```
.move-me {
  motion-offset: 0
  motion-path: path("M 5 5 m -4, 0 a 4,4 0 1,0 8,
    0 a 4,4 0 1,0 -8,0");
}
.move-me:hover {
  motion-offset: 100%;
}
```

For a more comprehensive look at SMIL animations, check out Sara Soueidan's "Guide to SVG Animations" and the spec (http://bkaprt.com/psvg/07-15/, http://bkaprt.com/psvg/07-16/).

ANIMATING PATHS

There's a little trick we can do with the stroke on SVG shapes: we can make it look as if the shape is drawing itself. It's clever as heck. A blog post by Jake Archibald first made me aware of the trick (http://bkaprt.com/psvg/07-17/).

Here's how it works (FIG 7.5, http://bkaprt.com/psvg/07-18/):

1. Imagine you make an SVG shape with a stroke. You set strokes with attributes like these: `stroke="black"` and `stroke-width="2"`.
2. Strokes can be dashed, with an attribute like this: `stroke-dasharray="5, 5"`, meaning "a dash five long followed by a space five long."
3. The dashes can be longer, as in `stroke-dasharray="30, 5"`. In fact, they can be any length.
4. You can also offset the stroke, which moves the starting position of those dashes, with an attribute like this: `stroke-dashoffset="30"`.
5. Imagine a dash so long that it covers the entire shape, and a space after it that is equally long. You could offset the stroke so that it looks like it's entirely covering the shape, or offset it so that it looks like there is no stroke at all.
6. Now imagine an animation that animates from fully offset back to `0`. The shape will "draw itself."

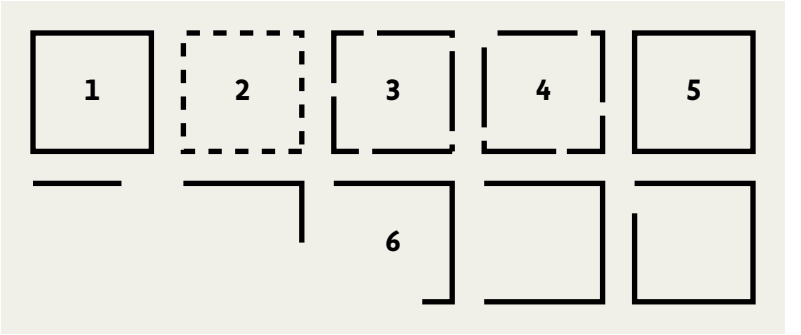

FIG 7.5: Animating paths makes it look as if a shape is drawing itself.

You can do all that in CSS, but with a dash of JavaScript, it becomes a little more foolproof. `path` elements have a property you can access via JavaScript that tells you exactly how long the element is:

```
var path = document.querySelector(".path");
var length = path.getTotalLength();
```

The resulting `length` number is exactly what the dash length and stroke offset need to be to do this trick.

ANIMATING SVG WITH JAVASCRIPT

JavaScript can animate SVG because JavaScript is all-powerful. In other words, JavaScript can manipulate things in the DOM. For instance, you can select an element with JavaScript and change the class name on it. JavaScript 101. A class name is just an attribute on an element. A circle's `cx` is just an attribute on an element, too. It controls the position of the center of the circle on the x-axis. JavaScript can change that.

JavaScript also has the ability to run a loop. Let's say we increased the `cx` attribute by `10` every 10 milliseconds:

```
var circle = document.getElementById(
  "orange-circle"), positionX = 0;

var interval = setInterval(function() {

  positionX += 10;

  if (positionX > 500) {
    positionX = 0;
  }

  circle.setAttribute("cx", positionX);

}, 20);
```

That's animation (http://bkaprt.com/psvg/07-19/)! That orange circle will animate from left to right over and over and over again.

It's not particularly efficient, though. For starters, `setInterval` isn't ideal for animations because the browser can't really optimize it. It can't, for example, stop the animation when it's not visible, or make it as smooth as possible.

The best approach to animation looping in JavaScript is `requestAnimationFrame` (http://bkaprt.com/psvg/07-20/). At its most basic:

```
function doAnimation() {
  // Do animation things
  requestAnimationFrame(doAnimation);
}
requestAnimationFrame(doAnimation);
```

That loop will run as close to 60 *frames per second* (FPS) as it can. The idea is that 60 FPS is what is required to make an animation appear very smooth to our eyes. That's great; it just means that that loop runs very fast, and that it's up to you to figure out the timing and duration.

There's an even better way to proceed here: use a JavaScript library built for animation. I wanted to cover the attribute-alter-

ing and looping concepts first, because that's what any library is actually doing under the hood. But these concepts offer us ways to declare animations that are easier to write and read, give us powerful options, and do the hard work behind the scenes for us.

Before we look at some examples, let's consider why we would want to use JavaScript to animate SVG:

- You're already working primarily in JavaScript and like to keep your work there. Or you just like the syntax of JavaScript.
- You're working with a data source in JavaScript.
- You need JavaScript to do math, loops, logic, or other programmery things.
- You need JavaScript to normalize some cross-browser bugs for you, like the known bugs with CSS `transform`s on SVG (http://bkaprt.com/psvg/07-21/).

And why would you avoid JavaScript here?

- Libraries add additional, significant weight to the page.
- JavaScript only works when it is available in the browser and loads properly.
- It only works on inline SVG or in contexts where external references are allowed, like `object` and `iframe`.

Snap.svg

Snap.svg is heralded as the "jQuery of SVG" (http://bkaprt.com/psvg/07-21/). It can be used to create and manipulate SVG, as well as animate it. To use it, you'll need to add the script to your page before you write any Snap.svg-specific JavaScript.

```
<script src="snap.svg.js"></script>
```

It has no other dependencies, so after you add it, you're ready to use it (http://bkaprt.com/psvg/07-22/).

```
<script>
  // create a new <svg> on the page
  // or use Snap("#existing-svg")
  var s = Snap(800, 600);

  // Draw a <circle>
  // Those attributes are cx, cy, and r
  var bigCircle = s.circle(150, 150, 100);

  // Manipulate the fill attribute to be "green"
  bigCircle.attr({
    fill: "green"
  });

  // Animate the radius and fill over one second
  bigCircle.animate({
    r: 50,
    fill: "lightgreen"
  }, 1000);
</script>
```

And that's only a drop in the bucket of Snap.svg's capabilities. Just as jQuery can help with anything DOM-related, like cloning elements, manipulating attributes, or attaching event handlers, so, too, can Snap.svg. Except that Snap.svg will do everything correctly in SVG, while jQuery may not. For instance, you'd think you could use jQuery's `.addClass()` method on an SVG element. Unfortunately, that will fail—the workaround is `.attr("class", "foo")`—but Snap.svg's identical `.addClass()` *will* work.

Read the Snap.svg documentation for a full look at what it can do; also, check out Pens tagged "snapsvg" on CodePen for a bunch of examples (http://bkaprt.com/psvg/07-23/, http://bkaprt.com/psvg/07-24/).

Greensock

Greensock is a robust and performant library focused on animation (greensock.com). It wasn't developed specifically for SVG, but it works great with it. Google even recommends Greensock to those looking for a dedicated animation library (http://bkaprt.com/psvg/07-25/).

Say you have existing SVG like this:

```
<svg width="260" height="200" viewBox="0 0 260 200">
  <rect id="rect" x="20" y="60" width="220"
    height="80" fill="#91e600" />
</svg>
```

Let's target that `rect`, turn it in a circle, and halve the size of it over five seconds:

```
TweenMax.to("#rect", 5, {
  rotation: 360,
  transformOrigin: "50% 50%",
  scale: 0.5
});
```

This is a particularly pertinent example, because if you try to do this exact same animation in CSS, you'll run into a lot of trouble. Here's why:

- IE and Opera won't do CSS transforms on SVG elements at all, let alone animate them.
- Firefox won't honor the percentage-based `transformOrigin` we've set there (nor will it do keywords).
- Safari will break the animation if the page is zoomed in either direction, not scaling the elements in sync with each other.

Greensock, beyond providing a nice API for animations as well as executing them smoothly, normalizes all of these bugs across browsers so the animation will work as expected. Pretty nice!

Using a JavaScript library for animation doesn't mean you're neglecting performance. In fact, the opposite may be true. In some cases, JavaScript will yield better performance. In a video, Greensock's Jack Doyle tests pure CSS against Greensock in some fairly intense animations, confirming that performance is a tricky thing; lots of factors ultimately affect the viewing and interacting experience (http://bkaprt.com/psvg/07-26/). Things like memory usage, painting area, and time, as well as the the effect on the frames per second a page can display, all impact performance.

Greensock's own CodePen account has loads of examples and demos on how to use it (http://bkaprt.com/psvg/07-27/).

REFLECTING ON ANIMATION

Congratulations—you've just made it through a whole lot of SVG animation information. There is much to love about animating SVG:

- You can easily jump into it using what you already know about animating in CSS.
- You can control and animate more design features (like strokes) than you can with HTML elements.
- You can really get serious about SVG animation, creating whole experiences, interactive spaces, and complex timelines.
- Lastly, you can port that knowledge back to animating HTML when needed.

The fact that SVG goes so well with HTML, CSS, and JavaScript is a good reason for SVG to be in every front-end developer's toolbox. And SVG has design features that we haven't even touched on yet that cross the boundaries between these languages. Let's dig into some of those features next.

8 SOME DESIGN FEATURES

BEYOND DRAWING and animating shapes, SVG has several features that can alter how the image ends up looking. We'll review four of them.

FILTERS

You may already know that CSS has filters. For instance:

```
.grayscale-me {
  filter: grayscale(100%);
}
```

SVG probably looks at those and is like: "Cute, kid." SVG filters are the original gangsters of filters. A similar filter defined in SVG would look like this:

```
<filter id="grayscale">
  <feColorMatrix type="saturate" values="0" />
</filter>
```

You can then apply that via CSS like so:

```
.grayscale-me {
  filter: url("#grayscale"); /* space separate
    multiple filters */

  /* or in an external file */
  filter: url("filters.svg#grayscale");

  /* you could even use a data URI here! */
}
```

Or you can apply it to an SVG element, like this:

```
<g filter="url(#grayscale)">
  <!-- grayscale all the stuff in here -->
</g>
```

With a filter like that available, it would be easy to design an interaction where, for instance, there is a grid of Adorable Avatars in grayscale; the one being hovered over or tapped goes back to full color, as shown in **FIG 8.1** (http://bkaprt.com/psvg/08-01/).

```
img {
  filter: url("#grayscale");
}
img:hover, img:focus {
  filter: none;
}
```

While CSS filters may be a bit easier to use, SVG filters can do anything that CSS filters can, and with deeper browser support.

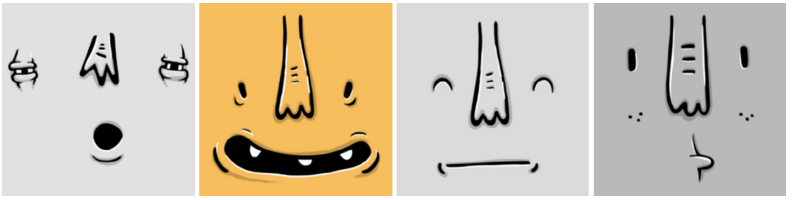

FIG 8.1: A row of avatars with a grayscale SVG filter applied. The second, on being hovered over or tapped, reverts to full color.

FIG 8.2: A row of avatars with a blur SVG filter applied. The filter gets stripped from the second image on hover or tap.

FIG 8.3: A row of avatars with a sepia SVG filter applied.

FIG 8.4: Okay, I think these little fellers have had enough.

The practical stuff, like blur, is all there in SVG filters (FIG 8.2):

```
<filter id="blur">
  <feGaussianBlur in="SourceGraphic"
  stdDeviation="3" y="-"/>
</filter>
```

But things can quickly get complex. Here's the "color matrix" required for converting colors to sepia tone (FIG 8.3):

```
<filter id="sepia">
  <feColorMatrix type="matrix" values=".343 .669
    .119 0 0 .249 .626 .130 0 0 .172 .334 .111 0 0
    .000 .000 .000 1 0" />
</filter>
```

Or something really weird and otherworldly, like adding "turbulence" (FIG 8.4):

```
<filter id="turbulence" >
  <feTurbulence type="fractalNoise"
    baseFrequency="0.015" numOctaves="2"
    result="turbulence_3" data-filterId="3" />
  <feDisplacementMap xChannelSelector="R"
    yChannelSelector="G" in="SourceGraphic"
    in2="turbulence_3" scale="65" />
</filter>
```

And we're just getting our toes wet. Note how the turbulence filter included two filter operations. Filters can include any number of operations, each affecting the last. As Chris Lilley told me in an email: "In some ways [filters] are more like a flow-based programming language than markup."

Lucas Bebber built some beautiful "Gooey Menus" with SVG filters that are fun to explore (http://bkaprt.com/psvg/08-02/). They combine blurring and deblurring and shadowing and compositing and all kinds of fancy.

There is practically no limit to what SVG filters can do to graphics. If this appeals to you, I'd encourage you to check out the spec (http://bkaprt.com/psvg/08-03/).

PATTERNS

Patterns are repeated designs. Imagine a polka-dot dress or those baggy chef pants with all the little different kinds of peppers on them that you used to wear in middle school and oh god the loneliness. Nobody laid out every single polka dot or pepper; they were created from patterned fabric.

Here are two reasons SVG patterns are cool:

- They make quick work of designs that would otherwise be too complex (too many points; too big a file).
- They are made from other chunks of SVG!

Imagine a repeating site background (**FIG 8.5**). *Very nice*, you think to yourself, *but can I use SVG for that?* It seems like an awful lot of vector points; the file size is probably too big to be practical. That would be an understandable thought, but this file is only about one kilobyte. That's because we made that complex-looking pattern from one tiny little shape (**FIG 8.6**).

The `pattern` element provides the magic here. It's an element designed to be used as a fill that will repeat over and over in a grid, like CSS `background-image`s can. A `pattern` is essentially a `rect`. It takes the same attributes: `x`, `y`, `width`, and `height`. The difference is that it doesn't render all by itself, just like the `symbol` element we used back in Chapter 3! You give it an ID so other elements can reference it.

Any SVG element that *does* render can use the `pattern` as a `fill`. Here, let's fill the entire SVG area with circles:

```
<svg width="100%" height="100%">

  <!-- this rectangular area won't render,
       but anything drawn inside of it can
       be used to fill other shapes -->
```

FIG 8.5: That pattern in the background sure looks complex.

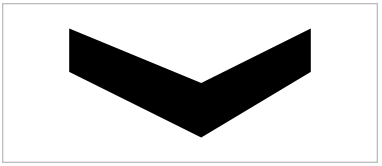

FIG 8.6: This small chevron shape can become the basis of a `pattern`.

```
    <pattern id="pattern-circles"
      x="0" y="0" width="20" height="20"
      patternUnits="userSpaceOnUse">
      <circle cx="10" cy="10" r="10" fill= "#f06d06" />
    </pattern>
      <rect x="0" y="0" width="100%" height="100%"
    fill="url(#pattern-circles)" />

  </svg>
```

Note the `patternUnits="userSpaceOnUse"` on the `pattern` element. That ensures that both the pattern and the element using the pattern exist in the same coordinate system (`x`, `y`, `width`, and `height`). In my experience, this prevents a boatload of confusion. If you ever find yourself in a situation where you want a `pattern` to have its own new coordinate system, look into the `objectBoundingBox` value for the `patternUnits` and `patternContentUnits` attributes.

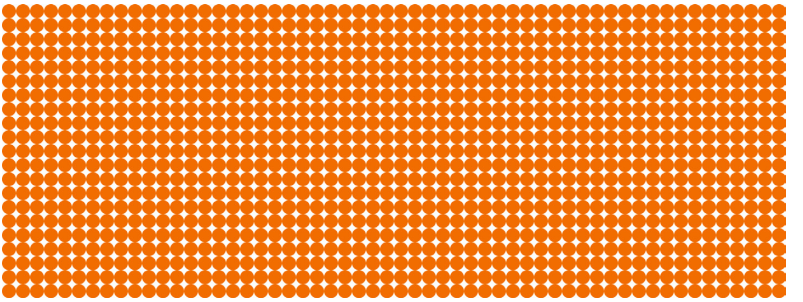

FIG 8.7: A demo in which a single `circle` makes up a simple `pattern` (http://bkaprt.com/psvg/08-04/).

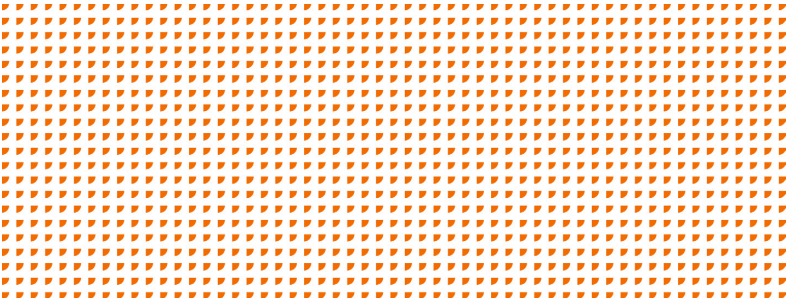

FIG 8.8: Moving the position of our `circle` results in a very different look for our `pattern`.

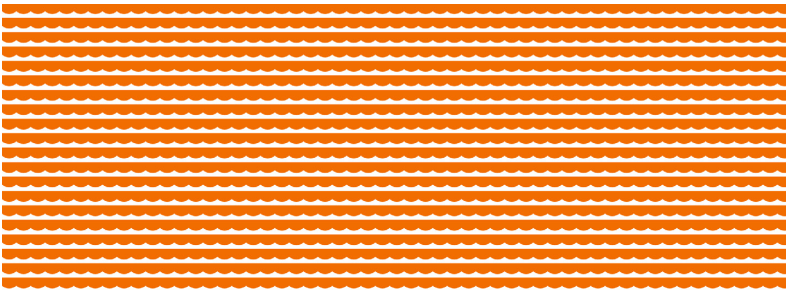

FIG 8.9: Changing the size of our `circle` results in yet another different look for our `pattern`.

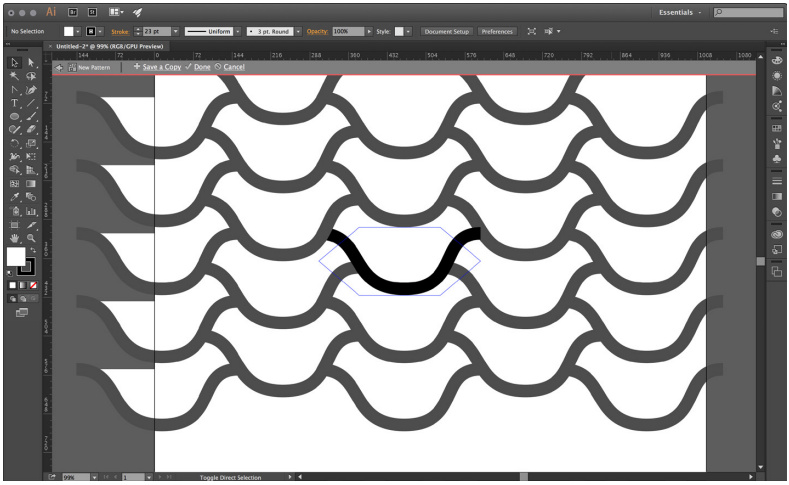

FIG 8.10: Adobe Illustrator's Pattern Options are quite good.

In **FIG 8.7**, the circle repeats perfectly, just touching the next circle in the pattern. That's because the width of the circle (radius × 2) and width of the pattern rectangle are exactly the same. And the circle is positioned in the middle of that pattern rectangle.

What if we move the center of the circle to `0,0`? Then we'll only see the bottom right corner of the circle, because that's the only part of it that's visible within the pattern rectangle. Anything outside of that area is cut off (**FIG 8.8**).

Or say we increase the radius of the circle beyond the edges, and position it centered at the top. Our pattern might then resemble curtains (or waves) (**FIG 8.9**).

I've set up a playground for playing around with these `pattern` attributes (http://bkaprt.com/psvg/08-05/).

Illustrator has a pretty good tool for working with patterns, and thankfully it also saves to SVG well! If you go to Object › Pattern › Make, the Pattern Options panel will open, and the interface will shift into pattern-editing mode (**FIG 8.10**).

First you make the pattern (click Done at the top of the screen when you have it how you want it); then Illustrator makes a new "swatch" for this pattern under the Swatches panel. Now you can create or select other elements and apply this pattern to them, just like you would in the SVG syntax with `fill="url(#pattern)"`.

What is especially useful about working with patterns in Illustrator is that we aren't limited to the repeating rectangles. You can define a pattern with offset rectangles (like a brick wall) or a grid of hexagons. This opens up some pretty cool pattern opportunities (read: almost any design set in repeating hexagons looks cool). SVG still only supports repeating rectangles through `pattern`, but that's precisely what is wonderful about Illustrator: it does the hard work for you of converting that pattern to one that can be represented as a rectangular tile, such that it can be drawn with `pattern`.

You probably know you can create repeating patterns in CSS as well. In fact, the default is `background-repeat: repeat`. You can simply create a rectangular bit of SVG and repeat it that way, which is a pretty great option if you're already using the pattern as a background. Otherwise, in order to set other content on top of the pattern set in inline SVG, you'd have to position it into place on top using `position: absolute;`—a rather blunt tool for the job.

If you'd like to play around more with `pattern`, SVGeneration is a pretty neat site for that, providing patterns that make use of features unique to SVG, as well as a UI to customize them and show you the code (svgeneration.com).

CLIPPING AND MASKING

Clipping and masking are related concepts because they are both capable of hiding parts of an image. But the distinction between them can be confusing, so let's clear that up right now:

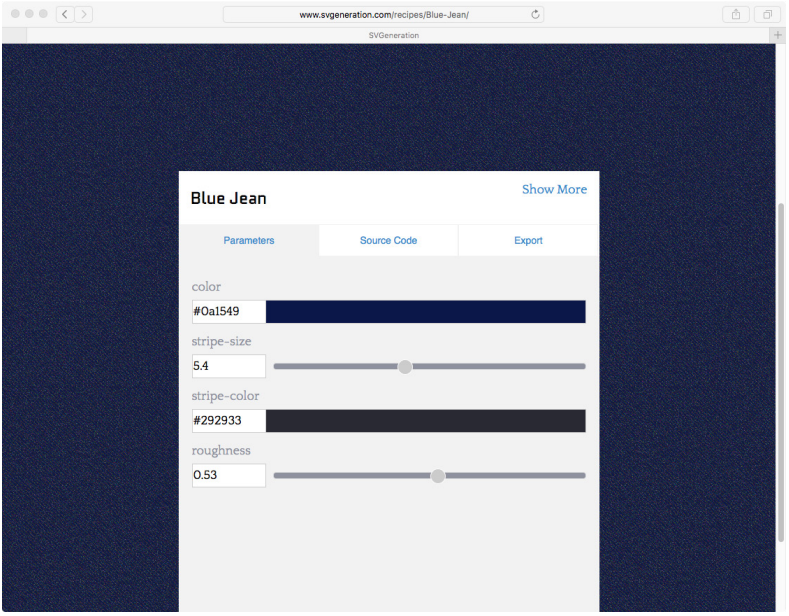

FIG 8.11: SVGeneration is a playground for interesting SVG patterns.

- Clipping is created from vector paths. Anything outside the path is hidden; anything inside is shown.
- Masking is created from images. Black parts of the image mask hide; white parts show through. Shades of gray force partial transparency—imagine a black-to-white gradient over an image that "fades out" the image.

Clipping is done with the `clipPath` element. Any SVG elements you put inside of the element don't render all by themselves (again, like `symbol`), but can be applied to other elements to clip them.

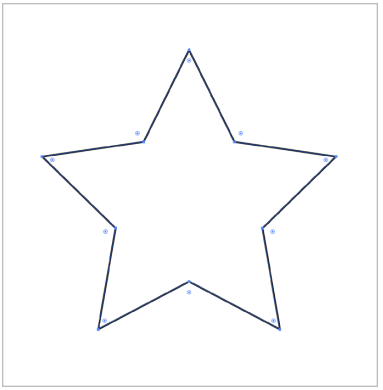

FIG 8.12: A star `polygon` drawn in Adobe Illustrator.

FIG 8.13: A bunch of gradient-filled `circle`s drawn in Adobe Illustrator.

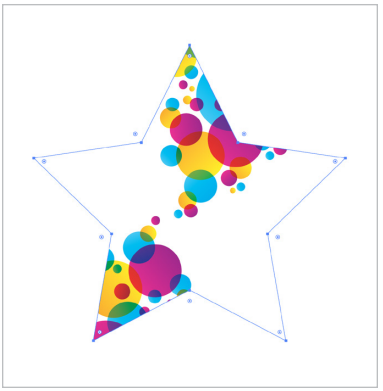

FIG 8.14: The star shape being used as a `clipPath` for the `circle`s.

Let's say we have a `polygon` of a star shape (**FIG 8.12**). And we also have a bunch of other fun SVG shapes laid out (**FIG 8.13**). We can use the star shape as a clipping path for the circle shapes (**FIG 8.14**):

```
<svg viewBox="0 0 1200 1000">
  <defs>
     <clipPath id="clip-star">
       <polygon points="..." />
     </clipPath>
  <defs>

  <g clip-path="url(#clip-star)">
    <circle ... />
    <!-- all those cool circles -->
  </g>
</svg>
```

In Illustrator, you can apply clipping paths like this by selecting multiple elements, making sure the topmost element is the clipping path you want to apply, and going to Object › Clipping Mask › Make. Note that Illustrator calls it a "mask" here, but it's actually a clipping path.

A clipping path is black and white in the sense that the part of the image being clipped is either hidden entirely or shown entirely. A mask is a bit different. A mask covers the entire area with an image of its own. Where that masking image is black, it *hides* the image below (and prevents user interaction as well, a sort of `pointer-events: none;`). Where that masking image is white, it *reveals* the image below. Any grays in that masking image partially reveal the image, depending on their value.

Perhaps the easiest way to make this distinction is to picture a white-to-black gradient (**FIG 8.15**).This gradient can be created in SVG and applied to a `rect`, and then put inside a `mask` ele-

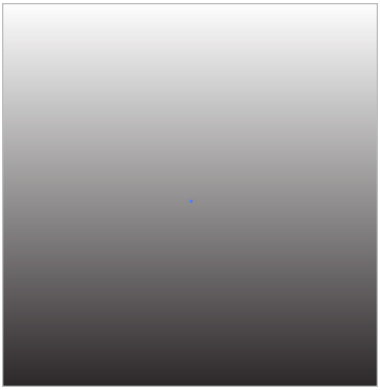

FIG 8.15: A white-to-black gradient applied to a `rect`.

FIG 8.16: Our gradient `rect` made into a `mask` and applied to our `circle`s.

FIG 8.17: A photograph applied as a luminance mask to SVG shapes.

ment. If we apply that `mask` to the same fun circles we were working with before, we get some nice results (**FIG 8.16**).

The code looks like this:

```
<svg viewBox="0 0 1200 1000">
  <mask maskUnits="userSpaceOnUse" id="fade">
    <linearGradient id="gradient" x1="0" y1="0"
      x2="0" y2="100%">
      <stop offset="0" style="stop-color: #FFFFFF" />
      <stop offset="1" style="stop-color: #000000" />
    </linearGradient>
    <rect fill="url(#gradient)" width="100%"
  height="100%" />
  </mask>
  <g mask="url(#fade)">
    <circle ... />
    <!-- all those cool circles -->
  </g>
</svg>
```

FIG 8.17 shows a photographic image applied as a mask, yielding a pretty wild outcome. A mask lets you do the same thing a clipping path does: you can fill the shapes you make with black or white as needed, and you can reverse the mask by simply reversing the colors. But masks are a bit more flexible—unlike clipping paths, masks can do partial masking.

Masks have another distinctive feature: they have two different types. We already looked at the default, `mask-type="luminance"`, which is based on color. There's another one: `mask-type="alpha"`. Alpha masks don't take color into account, only the alpha channel itself. For instance, if you use an SVG element as part of a mask with no fill at all, that's considered fully alpha transparent and will show the image beneath (http://bkaprt.com/psvg/08-06/).

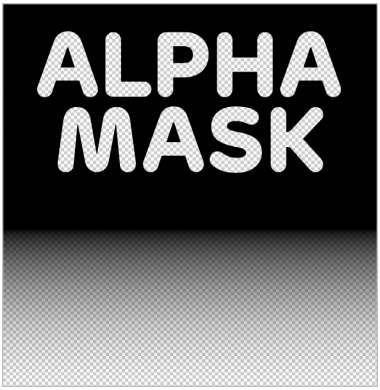

FIG 8.18: Example of a raster image with some fully opaque pixels, some fully transparent pixels, and some in-between pixels.

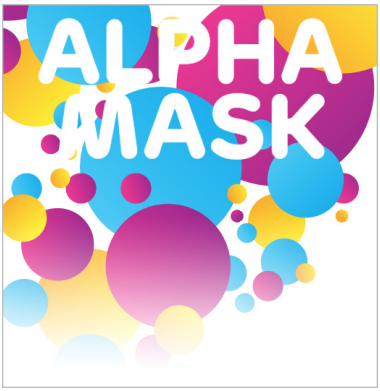

FIG 8.19: The raster image applied as an alpha mask. Note how the mask hides the areas where the raster image was alpha transparent. This is perhaps the opposite of what you might expect, but the end result is that the element is as transparent as the mask.

You can use raster images as masks if you want (**FIG 8.18**):

```
<mask maskUnits="userSpaceOnUse" id="fade" mask-
  type="alpha">
  <image y="200" width="100%" height="500"
    xlink:href="alpha-mask.png" />
</mask>
```

Peter Hrynkow has a very clever technique utilizing SVG masks (http://bkaprt.com/psvg/08-07/). Say you want a photographic image, and JPG is the best choice in terms of file size

FIG 8.20: Example of a JPG image being used within SVG, so it can be masked to have alpha transparency.

and quality. But you also want alpha transparency around the edges of the subject in the photograph. You could save from Photoshop as a "PNG-24," but then the file size will be a lot bigger than it would be with JPG.

Hrynkow's solution is to use two images. The photograph, and a black-and-white image as the mask. Then apply that mask in inline SVG!

```
<svg viewBox="0 0 560 1388">
  <defs>
    <mask id="canTopMask">
      <image width="560" height="1388"
        xlink:href="can-top-alpha.png"></image>
    </mask>
  </defs>
  <image mask="url(#canTopMask)" id="canTop"
    width="560" height="1388" xlink:href=
    "can-top.jpg"></image>
</svg>
```

BEYOND BASIC FILLS AND STROKES

Fills can be, and often are, solid colors: `fill="#F06D06"`, `fill="rgba(255,0,0,0.6);"`, and the like. But a fill can also be a gradient, much like how in CSS a background can have a solid color or a gradient as part of `background-image`. Using the same syntax you would use if you were applying a pattern fill, you reference the ID of where you have defined the gradient:

```
<path fill="url(#id)" ... >
```

Here's the definition for a lovely rainbow gradient from a Pen by yoksel (http://bkaprt.com/psvg/08-08/):

```
<linearGradient id="MyGradient" x1="0" y1="0"
  x2="100%" y2="0%">
  <stop offset="0%" stop-color="crimson" />
  <stop offset="10%" stop-color="purple" />
  <stop offset="10%" stop-color="red" />
  <stop offset="20%" stop-color="crimson" />
  <stop offset="20%" stop-color="orangered" />
  <stop offset="30%" stop-color="red" />
  <stop offset="30%" stop-color="orange" />
  <stop offset="40%" stop-color="orangered" />
  <stop offset="40%" stop-color="gold" />
  <stop offset="50%" stop-color="orange" />
  <stop offset="50%" stop-color="yellowgreen" />
  <stop offset="60%" stop-color="gold" />
  <stop offset="60%" stop-color="green" />
  <stop offset="70%" stop-color="yellowgreen" />
  <stop offset="70%" stop-color="steelblue" />
  <stop offset="80%" stop-color="skyblue" />
  <stop offset="80%" stop-color="mediumpurple" />
  <stop offset="90%" stop-color="steelblue" />
  <stop offset="90%" stop-color="purple" />
  <stop offset="100%" stop-color="mediumpurple" />
</linearGradient>
```

FIG 8.21: SVG `text` with a `linearGradient` as a fill. Note that some colors fade gradually from one to the next, and some change brusquely. The smooth changes happen between `stop`s with different offsets (e.g., between red at 10% and crimson at 20%). The brusque changes result from `stop`s with the same offset, such as red to orange (30%) or gold to green (60%).

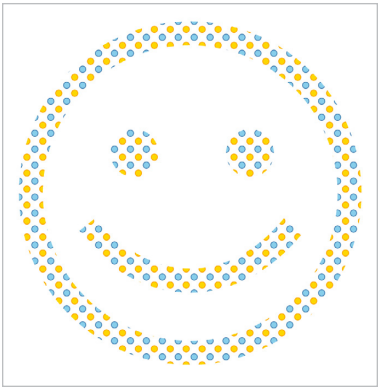

FIG 8.22: A `pattern` used as a fill for a stroke.

We can apply that to a `path` or any other basic shape, but we can also apply it to `text` (**FIG 8.21**). It produces that cool knock-out effect that is difficult to pull off cross-browser with CSS.

Another cool thing you can do in SVG that you can't do as intuitively in CSS is apply a fill to a stroke. Another Pen by yoksel demonstrates this beautifully (http://bkaprt.com/psvg/08-09/, **FIG 8.22**).

Those are nice thick `stroke-width="30"` strokes there!

COMBINING FEATURES

These features of SVG aren't mutually exclusive and, in fact, combining them is great fun. For example, you can animate a clipping path:

```
<svg id="drawing" viewBox="0 65.4 792 661.1">
  <defs>
    <clipPath id="clip-star">
      <polygon id="star" points="...">
         <animate ... />
      </polygon>
    </clipPath>
  </defs>
  <g clip-path="url(#clip-star)">
    <!-- stuff! -->
  </g>
</svg>
```

Or you can combine technologies here, and do that animation with Snap.svg instead of SMIL (http://bkaprt.com/psvg/08-09/):

```
var drawing = Snap("#drawing"),
    star = drawing.select("#star");

star.animate({
  transform: "s0.5 r45 t25 25"
  // that special string means this:
  // scale(1.5) rotate(35deg) translate(25px, 25px)
}, 1000);
```

Here's another idea! Say you want an image to fade from black and white to color—not on hover, but you want the black and white of the left half to fade into color on the right half (**FIG 8.23**). We can do that by placing two images on top of each other. Could be any SVG, but if we want to use photographic images, those can be SVG too, with `<image xlink:href="">`. Since we're going to use the image twice, let's make a `symbol`

FIG 8.23: A full-color image fades to grayscale by placing a grayscale-filtered copy on top and masking half of it with a gradient.

so that we don't repeat ourselves. Then we'll apply the grayscale filter and the gradient mask to the second image. That second image will be on top, because SVG uses document order as paint order: whatever comes next in the source order is on top of what came before.

```
<svg width="500" height="366">

  <!-- We're going to use the image twice, so let's
    make it a repeatable <symbol> -->
  <symbol id="image">
    <image x="0" y="0" width="500" height="366"
      xlink:href="https://s3-us-west-2.amazonaws
      .com/s.cdpn.io/3/rainbow-face.jpg"></image>
  </symbol>
```

```
  <!-- A filter we can use to grayscale whatever -->
  <filter id="grayscale">
    <feColorMatrix type="saturate" values="0" />
  </filter>

  <!-- A black-to-white gradient mask that we can
    apply to whatever -->
  <mask maskUnits="userSpaceOnUse" id="fade">
    <linearGradient id="gradient" x1="0" y1="0"
      x2="100%" y2="0">
      <stop offset="0" style="stop-color: #FFFFFF" />
      <stop offset="0.4" style="stop-color: #FFFFFF" />
      <stop offset="0.6" style="stop-color: #000000" />
      <stop offset="1" style="stop-color: #000000" />
    </linearGradient>
    <rect fill="url(#gradient)" width="100%"
  height="100%" />
  </mask>

  <!-- Place the image once, in full color -->
  <use xlink:href="#image"></use>

  <!-- Place the image again, on top, both
    grayscaling it and masking it -->
  <use xlink:href="#image" filter="url(#grayscale)"
    mask="url(#fade)"></use>

</svg>
```

Things really get fun when you combine these design features—you can even apply filters to patterns (**FIG 8.24**)!

And I could go on forever. With SVG, design effects can be combined and recombined infinitely. But I'll force myself to stop here—now it's your turn to build on the foundation I've sketched out.

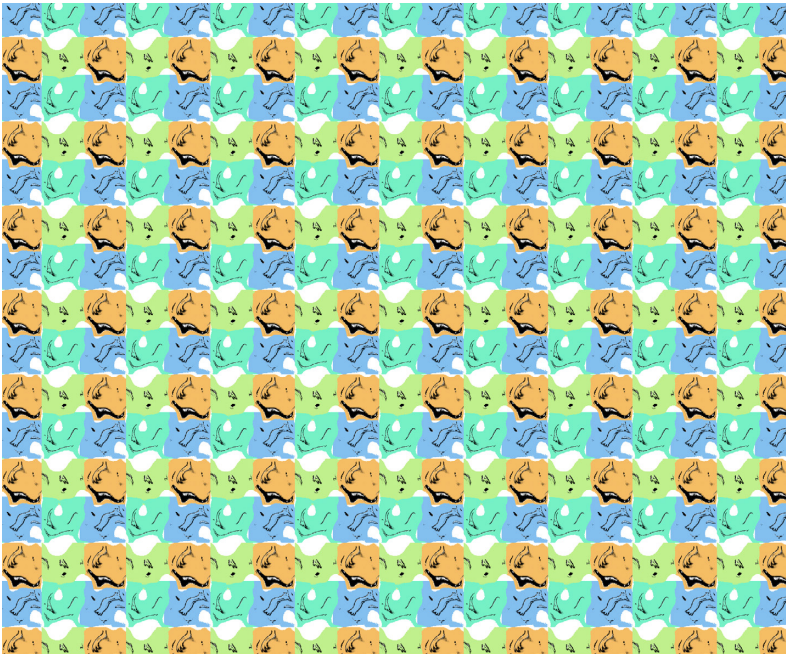

Fig 8.24: A turbulence `filter` applied to `image`s within a `pattern` used to fill a `rect`.

What I've tried to do is introduce you to a bunch of possibilities, so that the next time you're all, "I wonder if I could make a graphic of Susan Kare skateboarding through the screen of an old Macintosh icon shooting a laser beam at a hamburger icon and it blows up into a big rainbow," your brain will be like: "I got some ideas."

9 FALLBACKS

We've talked about how browser support for SVG is pretty good, but not ubiquitous. IE 8 and Android 2.3 don't support it at all, and I'd say those are pretty reasonable browser-support targets for a lot of websites. Does that mean we give up on using SVG for sites that need to accommodate them? Absolutely not. We don't need to punish newer browsers for older browsers' lack of support, just as we don't need to punish older browsers with broken design and functionality.

The fallback approach that will work for you depends on how you are using SVG. But first things first.

DO YOU NEED A FALLBACK AT ALL?

Sometimes the way we use images is purely complementary. The website will function just fine without them. For instance, imagine our SVG icon system, and a shopping cart icon next to the phrase "View Cart" in a button. Without that icon, it's

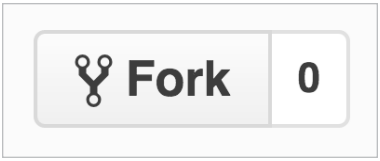

FIG 9.1: The Fork button on GitHub.com is an example of an image that doesn't need a fallback.

still a button that says "View Cart." Not as fancy, but perfectly functional. In this case, we could feasibly skip worrying about a fallback at all.

SVG used in CSS as a background image also commonly falls into this category. Although we will cover a fallback technique for that, images applied in CSS are generally decorative. Making sure that the content on top of the background is still usable may be the only fallback you need.

FALLBACK FOR SVG-AS-img

The simplest approach here is to use SVG as you normally would:

```
<img src="dog.svg" alt="dog">
```

Then replace that `src` with something that will be supported, like dog.png. That's exactly what the SVGeezy script by Ben Howdle does (http://bkaprt.com/psvg/09-01/). If the script determines that a browser doesn't support SVG in this way, it will replace the `src`, and end up looking like this in the DOM:

```
<img src="images/dog.svg" alt="dog">
<!--
    this will be turned into this

       <img src="images/dog.png" alt="dog">

    if the browser doesn't support SVG this way -->
```

```
    <script src="/js/libs/svgeezy.js"></script>
    <script>
      svgeezy.init(false, "png");
    </script>

  </body>
```

The file type is configurable, but it's on you to ensure that the fallback image is located in the same location as the SVG.

This works, but it comes at a cost beyond just loading additional JavaScript. Non-supporting browsers will actually download both versions of the image before ultimately using the fallback. That means we've already betrayed our ideal of not punishing older browsers.

We can beat this double-download problem, and we'll get to that in a moment. But first, we can learn something else very useful. SVGeezy tests for SVG-as-`img` support with this function:

```
supportsSvg: function() {
  return document.implementation.hasFeature("http://
  www.w3.org/TR/SVG11/feature#Image", "1.1");
}
```

Um, `document.implementation.hasFeature`? What is this wizardry? Is there some native JavaScript API for testing feature support? For that sort of thing, we normally turn to Modernizr, a library built from an amalgam of clever tests developers have concocted to coerce browsers into telling us whether they support a certain feature or not (http://modernizr.com).

Turns out this unusual `hasFeature` thing is no dream come true. It's actually quite old and largely deprecated, but it's still interesting. `document.implementation.hasFeature` returns true for everything, except when you use it exactly as shown above, in which case it returns true or false perfectly in all known browsers based on whether that browser supports

SVG-as-`img` or not. So in case you need this information, now you know that it's quite easy to get.

And why might you need to know if a browser supports SVG-as-`img` in JavaScript? Perhaps you'll use that information to decide if you're going to load additional scripts to help with a fallback. Or you'll use it as part of a test to make a general determination about what kind of experience your site is going to deliver to that browser, a technique now known as *mustard-cutting* (http://bkaprt.com/psvg/09-02/).

You could easily build your own SVG-as-`img` fallback system:

```
// do the test
if (!document.implementation.hasFeature("http://
  www.w3.org/TR/SVG11/feature#Image", "1.1")) {
  // if the browser doesn't support SVG-as-<img>
  // find all images
  var images = document.getElementsByTagName("img"),
    i, src, newsrc;
  // loop through them
  for (i = 0; i < images.length; i++) {
    src = images[i].src;
    ext = src.split(".").pop();
    // if the image is .svg (note this doesn't
      account for ?query strings
    if (ext === "svg") {
      // replace that with .png
      newsrc = src.replace(".svg", ".png");
   images[i].setAttribute("src", newsrc);
    }
  }
}
```

These techniques also require a little more elbow grease: you have to create the fallback images yourself and place them alongside their `.svg` comrades. So when the `src` is replaced, the `.png` version is there to pop into position. Wouldn't it be nice if that were automated?

That's what SVGMagic helps you do (http://bkaprt.com/psvg/09-03/). Simply use SVG-as-`img` like you normally would. Then load and initialize the plugin. If it decides the browser needs fallback PNGs, it will automatically create them for you. It does so by making a network request to SVGMagic's servers, which return the new images. Be forewarned, though, that that's a dependency over which you have no control.

We still haven't found a way around the double-download issue, so let's do that next.

One way of circumventing it is to use the `picture` element. The `picture` element is often thought of as a solution for responsive images (serving differently *sized* images as needed), but it can also serve different image *types* depending on support.

Here's how that works in HTML:

```
<picture>
  <source type="image/svg+xml" srcset="graph.svg">
  <img src="graph.png" alt="A lovely graph">
</picture>
```

If the browser supports SVG this way, `source` will be used; otherwise, the fallback PNG in the `img` tag will be served, without the double-download. It's pretty great to have a fallback solution like this right in HTML. The rub is that the browser *also* needs to support `picture`, which is so new that any browser that supports it also supports SVG.

That doesn't eliminate this solution from the running, though, thanks to Picturefill, a script that makes `picture` work in any browser (http://bkaprt.com/psvg/09-04/). This sort of script is called a *polyfill*, by the way. When you load up Picturefill, the `picture` syntax will work great, delivering SVG to supporting browsers and the fallback otherwise.

As a nice side bonus, you can use the `picture` syntax to deliver differently-sized fallback images, if necessary. For instance, you might want to serve a PNG that is 800 pixels wide for a desktop IE 8, but that would be a waste of bandwidth for

a non-Retina iPhone, so you'd like to serve a 320-pixel version there. That's all possible with `picture`.

There is a catch though, beyond loading the 7 KB script. The reason double-downloads happen in the first place is because of that `img src` in the HTML. Browsers do what is called *prefetching*—they zoom through a page looking for resources they can start downloading right away. Prefetching is a good thing, because it helps make the web fast. But we have no control over it. In the `picture` syntax we looked at, notice this part: `img src="graph.png"`. Prefetching will catch that and download graph.png, whether it ends up being used or not.

In a browser that natively supports `picture`, the prefetcher will be smart enough not to do that. But we can't count on that (hence the polyfill). To solve this, we can just skip the `src` and make the markup more like this:

```
<picture>
  <source srcset="graph.svg" type="image/svg+xml">
  <img srcset="graph-small.png, graph-large.png
    1000" alt="A lovely graph">
</picture>
```

Solved! Even though, as I write this, that's technically invalid code, it works with Picturefill and in browsers that already support `picture`. So you could even safely pull out Picturefill one day and not worry about breaking anything. I wouldn't be terribly surprised if this eventually becomes valid markup, since it's so useful.

FALLBACK FOR SVG AS background-image

This method is pretty great, because it relies on some simple sleight of hand right in the CSS, rather than on some other technology or much additional code.

```
.my-element {
  background-image: url(fallback.png);
  background-image:
    linear-gradient(transparent, transparent),
    url(image.svg);
}
```

It does the trick because of some serendipitous overlap in features that browsers support. Two forces are at work here: multiple backgrounds and "old" syntax gradients. A browser that supports both of these things also necessarily supports SVG as `background-image`. Thus, the SVG background image we supplied here will be shown (the gradient is completely transparent and will have no effect). If either of those things fails, the whole declaration fails, and the fallback `background-image` declaration takes effect.

Because this is so straightforward, it's kind of tempting to exploit it for images used in HTML as well. After all, it would be pretty easy to just use a `div` with a background instead of an image. The danger here is that background images aren't content. If the content here were ever syndicated through email or RSS, you'd lose the background images entirely. Not to mention that that usage is semantically incorrect, and you'd have to take extra steps to ensure accessibility. There is no `alt` text for background images.

Another possibility here would be to run the JavaScript test for SVG-as-`img` (the support is identical for background images), add a class to to the `html` element (something like `<html class="no-svg">`) and then:

```
.my-element {
  background-image: url(image.svg);
}
html.no-svg .my-element {
  background-image: url(fallback.png);

}
```

This works, but since it requires JavaScript, I prefer the multiple-backgrounds trick.

FALLBACK FOR INLINE SVG

Building a fallback for inline SVG is a little trickier than the other two, but certainly doable. Let's break it into two categories.

Do you want to start with SVG and gracefully degrade?

This method is forward-thinking in the sense that down the road, you can decide to pull out the fallback if you think browser support is good enough.

Here's a possible approach, one I quite like:

1. Use inline SVG normally:

```
<svg class="icon icon-cart" xmlns=
  "http://www.w3.org/2000/svg">
  <use xlink:href="#icon-cart"></use>
</svg>
```

2. Make an inline SVG test in JavaScript:

```
var supportsSvg = function() {
  var div = document.createElement("div");
  div.innerHTML = "<svg/>";
  return (div.firstChild && div.firstChild.
  namespaceURI) == "http://www.w3.org/2000/svg";
};
```

3. If the browser doesn't support inline SVG, add a class name to the `html` element:

```
if (supportsSvg()) {
  document.documentElement.className += "no-svg";
};
```

4. Use that class name to set a background image on the `svg`:

```
html.no-svg .icon-key {
  display: inline-block;
  width: 33px;
  height: 33px;
  margin-right: 0.25em;
  vertical-align: middle;
  /* Even better, a sprite */
  background: url(icon-fallbacks/key.png) no-repeat;
}
```

You can see the technique in action on CodePen (http://bkaprt.com/psvg/09-05/).

There are only a couple of reasons to you might choose to avoid this method:

1. If JavaScript doesn't load or run in a browser that also doesn't support inline SVG, no icon will be present.
2. You need to go back further than IE 8.

IE 8 will need the fallback background image, but it can be set directly on the `svg` element like we did in the fourth step, provided that it has the correct `xmlns` attribute. IE 7 and older won't allow that. You can still use this technique, though; just wrap the `svg`s in a `div` or `span` and apply classes to them instead. Doing things this way will ensure that a proper fallback will be present even if JavaScript doesn't load or isn't available.

Or would you rather start with a `span` and progressively enhance?

Imagine a Close button that you hope will appear simply as a cross shape: ×. Start with a `span` containing the text "Close"—worst-case scenario, your button will still say "Close."

```
<button aria-label="Close">
  <span class="inline-svg" data-xlink="#icon-
  close">Close</span>
</button>
```

Now, if you run the JavaScript test we just went over and see that the browser *does* support inline SVG, inject it and replace the `span`.

```
if (supportsSvg()) {
  var inlineSvgs = document.querySelectorAll(
    "span.inline-svg");

  for (i = 0; i < inlineSvgs.length; i++) {
    var span = inlineSvgs[i];
    var svgNS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";
    var svg = document.createElementNS(svgNS,
      "svg");
    var use = document.createElementNS(svgNS,
      "use");

    // Prepare the <use> element
    use.setAttributeNS(xlinkNS, "xlink:href",
      span.getAttribute("data-xlink"));

    // Append it
    svg.appendChild(use);

    // Prepare the SVG
    svg.setAttribute("class", "inline-svg");

    // Inject the SVG
    span.parentNode.insertBefore(svg, span);

    // Get rid of the <span>
    span.remove();
  }
}
```

Dave Rupert thought this up (http://bkaprt.com/psvg/09-06/).

BEWARE THE DOUBLE-DOWNLOAD

These aren't the only SVG fallbacks you'll see out there. I'm intentionally only showing you the ones I think work best and are the most responsible. A lot of the techniques I see around are guilty of either a double-download or extreme complexity.

Remember these rules of thumb:

- If you see `img src=""` anywhere in the HTML, that *will* trigger a download. If you try to replace that source with JavaScript, that will trigger *another* download.
- If you see a CSS `background-image` on a selector that matches anything in the HTML, that *will* trigger a download.
- Double-downloads are awful for performance. Avoid them if possible.
- Ideally, circumvent situations where every single fallback is a separate network request.

MAKING SVG ACCESSIBLE

Although there is much we can do to create accessible experiences on the web, accessibility is a huge topic that I can only touch on briefly here. For a fantastic general resource, see the Accessibility Project (a11yproject.com).

In matters of accessibility, best practices are relevant to SVG in the same way they're relevant to HTML. For instance, when we're using SVG as the source of an `img`, we need to provide proper `alt` text that explains what's going on in the image.

But then there is inline SVG, which is a whole area of focus unto itself. Here I'll defer to Léonie Watson, who wrote a wonderful article describing all of the things we can do to create accessible SVG (http://bkaprt.com/psvg/09-07/). I'll attempt to summarize the piece here.

Use SVG

I enjoy this one—it serves as a good reminder of the breadth of accessibility concerns. Remember that SVG is visually crisp; for the visually impaired, SVG's sharpness can be hugely beneficial.

Use inline SVG

I've attempted to extol the virtues of inline SVG throughout this book—and I'm not done yet! Inline SVG allows *assistive technologies* (AT) like screen readers more access to information than is possible using SVG any other way.

Use `title` and `desc`

Use `title` like you would an `alt` attribute on an `img`. It allows an AT user to identify what it is. Use `desc` to provide more detailed information.

Use an ARIA role

An element's role defines its purpose. To guarantee that all browsers apply the correct role to SVG, define it explicitly.

Combining Watson's recommendations so far, properly accessible SVG looks like this:

```
<svg aria-labelledby="title desc" role="img">
  <title id="title">Green rectangle</title>
  <desc id="desc">A light green rectangle with
    rounded corners and a dark green border.</desc>
  <rect width="75" height="50" rx="10" ry="10"
    fill="#90ee90" stroke="#228b22" stroke-fill="1" />
</svg>
```

FIG 9.2: Text used in an SVG with the `text` element.

Use `text`

SVG can render regular ol' web text just like HTML can. In HTML, you mark up text with elements like `h1`, `p`, and the like. In SVG, you do that with the `text` element, which serves a similar function and makes text just as accessible as those HTML elements do.

The text is also copy-and-pasteable and SEO-friendly, which is nice. Plus, the text scales in proportion with the rest of the SVG—a lot harder to pull off in HTML.

SVG also has access to all the same fonts that the rest of your document does. So if you are loading up a custom `@font-face` font for your site (as we are in these examples), you can use that font in your SVG just fine.

```
<text font-family="Custom Font, sans-serif"
  font-size="14" letter-spacing="10">Bluegrass
  Festival</text>
```

One limitation of `text` in SVG is that it can't autowrap or reflow, a problem that may be solved in SVG 2.

Make interactive elements focusable

If your SVG is interactive, you can use `a` links around the interactive elements to make them focusable from the keyboard—for instance, if part of the image has a hover state, or is clickable as a link:

FIG 9.3: SVG `text` is selectable like any other real text on the web, and it scales proportionally with the rest of the SVG.

```
<a xlink:href="http://example.com">
  <rect width="75" height="50" rx="20" ry="20"
    fill="#90ee90" stroke="#228b22" stroke-fill="1" />
  <text x="35" y="30" font-size="1em"
    text-anchor="middle" fill="#000000">Website</text>
</a>
```

Create an alternative

If the the title and description aren't enough to explain exactly and clearly what is happening in the SVG to someone who can't see it, create an alternative experience. For instance, perhaps SVG is being used to draw a chart. Note that SVG is particularly great at drawing charts on the web. While it doesn't have any

charting-specific features, all of its features lend themselves well to drawing bar charts, line graphs, pie charts, and the like. There are robust charting libraries that output entirely in SVG, like amCharts and Highcharts (http://bkaprt.com/psvg/09-08/, http://bkaprt.com/psvg/09-09/).

Now, a string of text typically isn't sufficient to explain a chart full of information. So perhaps you could use JavaScript to build an SVG chart from data in the form of a `table` that, though visually hidden, is still available for AT. That way, the user will experience it either as a perfectly useful table, or as an SVG chart from that same data. You can even use that same table of data to build different types of visualization as needed.

Here's a block of code that stitches all of these pieces together:

```
<svg aria-labelledby="title desc" viewBox="0 0 400
  327">

  <title id="title">Graves Mountain Bluegrass
    Festival</title>
  <desc id="desc">Advertisement for Graves Mountain
    Bluegrass Festival. Blue skies and three
    differently sized blue-tinted mountains.</desc>

  <a xlink:href="http://gravesmountain.com"
    tabindex="0" role="link">
    <polygon opacity="0.2" fill="#7DACDC"
    points="0,327 200,173.797 400,327 "/>
    <!-- and other shapes -->

    <text x="35" y="30">Graves</text>
    <!-- and other text -->
  </a>

</svg>
```

An accessibility checklist like this can be valuable: it gets you thinking about accessibility, and sends you off and running in the right direction. I would warn against thinking of a checklist as a "just do these things and then wipe your hands of it" reflex, however. Instead, I'd encourage this sort of mindset:

1. Summon your powers of empathy and try to use the things you build with different impairments in mind.
2. Use the things you build with different assistive technologies, such as a screen reader like JAWS (http://bkaprt.com/psvg/09-10/).
3. Consult with accessibility professionals.

We just covered a considerable amount of technical stuff. Admittedly, for most of us, dealing with fallbacks and managing accessibility may not be the most enjoyable part of working with graphics. But I think it's pretty dang rewarding knowing that our graphics are as helpful as they can be to anybody using our sites.

Most rewarding of all: now we're fully armed with the information we need when people open a conversation with, "Well, I'd like to use SVG, but [insert excuse about support here]." Now it's our turn to hold up our WELL ACTUALLY fingers.

CONCLUSION

WE'VE LEARNED A LOT in a short amount of time, my friends. We've learned about the virtues of SVG, when it is the appropriate choice, and how we can get it onto our sites. We've looked at tools that help you work with it, from elaborate software like Illustrator down to command-line tools like Grunt. We've learned about icon systems, fallbacks, and sizing concerns. We've considered performance, accessibility, and responsiveness. We've learned about some of the design possibilities of SVG, like animation, filters, and masks.

Yet there is so much we didn't cover. Attributes left out, values unspoken, entire tags unsung. In part, that's because of this book's brevity: not only would a 500-page slog through every little detail of SVG make for a dry read, but it would also be a poor substitute for "the internet." Quite a bit has been written about SVG that you can search for and find, including plenty of articles by yours truly. I even maintain an up-to-date compendium of SVG information that points to the best resources I know of (http://bkaprt.com/psvg/10-01/).

Another reason to have covered only what we did here is that these are the features I use on a daily basis as a front-end developer. A different developer would likely have covered slightly different things.

For instance, SVG is *so perfect* for charting. I can imagine a whole chapter (or book!) on building charts with SVG. If I needed to build a charting system, I would reach for SVG in a heartbeat. But I haven't done that myself yet, so I merely touched on it here. And that's only one thing I could have explored in much greater depth.

I hope that reading this book has shown you a way into SVG and what it looks like to use it. One of my favorite sayings is "Pave the cow paths." Like, if you're building a sidewalk, build it where people are already walking. In the web community, this idea has a lot of currency; it means we should standardize things around what developers are already doing.

It doesn't quite fit for SVG, though: the sidewalks are already there. The foundation is sound, but years of neglect have left a few cracks and a bit of grass growing up over the edges.

By using SVG now, we're doing ourselves and our users a favor. But we're also doing the web a favor by forging a new path that browser makers and standards organizations can see and react to, which in turn will help make SVG even stronger.

So pave the cow paths—but also patch the cracks, pull the weeds, and clear some ground for new, hitherto unimagined trails.

RESOURCES

I've sprinkled links to resources through this book, and I hope you've followed them as they've cropped up. This section lists documentation, texts, sites, and tools that I find particularly valuable, but that I didn't explicitly cover in the book or explore in detail. Many of these resources have interactive elements—the web is perfect for that; books, not so much. Be sure to check out these gems:

- **"SVG,"** Mozilla Developer Network. This is a directory page that links to hundreds of individual pages covering every SVG element, attribute, DOM interface, effect, and much more. It is likely the most comprehensive resource on SVG (http://bkaprt.com/psvg/11-01/).
- **"A Complete Guide to SVG Fallbacks,"** Amelia Bellamy-Royds. This covers every possible scenario of SVG usage and browsers that don't support it (http://bkaprt.com/psvg/11-02/).
- **"SVG on the Web - A Practical Guide,"** Jake Giltsoff. A long-form, open-source reference guide that lists quite a few resources as well (svgontheweb.com).
- **"Inline SVG vs Icon Fonts [CAGEMATCH],"** by me. My complete, point-by-point analysis compares the two most popular techniques for icon systems (http://bkaprt.com/psvg/03-06/).
- **"Ten Reasons We Switched from an Icon Font to SVG,"** Ian Feather. Ian's rationale for moving to an SVG icon system on Lonely Planet influenced my own thinking (http://bkaprt.com/psvg/04-06/).
- **"Use SVG for Icons,"** Pete LePage. If you aren't convinced by lone developers' opinions, perhaps you'll like the official advice from Google (http://bkaprt.com/psvg/11-03/).
- **IcoMoon:** An interactive tool for building SVG sprites for use as an icon system. One of the longest-running and best tools for the job (http://bkaprt.com/psvg/04-01/).
- **Grunticon:** Perhaps the most popular SVG-based icon system tool, Grunticon generates a conditionally-loaded CSS file where each icon is converted into a class that

applies a data URL `background-image` of that icon (http://bkaprt.com/psvg/11-04/).

- **grunt-svgstore:** Fabrice Weinburg's grunt plugin combines multiple SVG files into a single sprite. I dreamed up the idea for a tool like this; Weinburg built what I believe is the first of its kind (http://bkaprt.com/psvg/04-03/).
- **gulp-svg-sprite:** The Gulp equivalent of Weinburg's plugin, by Joschi Kuphal (http://bkaprt.com/psvg/11-05/).
- **"Understanding SVG Coordinate Systems and Transformations,"** Sara Soueidan. Sara has done lots of SVG evangelism. This article is among many of hers that plumb one aspect of SVG deeply; it includes an interactive exploration tool (http://bkaprt.com/psvg/11-06/).
- **Scalable Vector Graphics (SVG) 1.1:** This is the official specification for SVG by the W3C (http://bkaprt.com/psvg/11-07/).
- Scalable Vector Graphics (SVG) 2. Like any good new specification, this draft specification for the upcoming version of SVG "adds new features commonly requested by authors." I know I'm excited for things like wrapping text (http://bkaprt.com/psvg/11-08/).
- **"Bespoke SVG Reference,"** Chris Nager. The discovery that `path` is the ultimate SVG shape element—all shapes are ultimately drawn with it—was a real eye-opener for me. The `path` syntax may look a bit strange, but Nager breaks down how simple it actually is (http://bkaprt.com/psvg/11-09/).
- **SVGeneration:** This interactive tool does a wonderful job of showing off interesting SVG design possibilities generated from very little code (svgeneration.com).
- **SVG Fancy Town:** This is my personal collection of SVG demos that I find particularly impressive (http://bkaprt.com/psvg/11-10/).
- **"The Image That Called Me,"** Mario Heiderich. This is probably the most widely cited text regarding SVG and security. A lot of the danger boils down to the fact that SVG is XML-based and can contain JavaScript, which makes it an XSS concern (http://bkaprt.com/psvg/11-11/, PDF).
- **"Tips for Creating Accessible SVG,"** Léonie Watson. A classic article on how to structure SVG to be accessible (http://bkaprt.com/psvg/09-07/).

- **Can I Use...:** This is the most popular site for tracking browser support of browser features, including SVG. Admirably, the site doesn't track SVG as a blanket "yes or no" in terms of support, but looks at individual SVG features and partial support levels of those features (http://bkaprt.com/psvg/11-12/).
- **"You Don't Know SVG,"** Dmitry Baranovskiy. A video of a talk bursting with SVG possibilities (http://bkaprt.com/psvg/11-13/).
- **SVG Weirdness:** This GitHub repo organized by Emil Björklund tracks unexpected cross-browser bugs in SVG (http://bkaprt.com/psvg/11-14/).
- **Chromium Bug Tracker:** A collection of SVG bugs reported to the Chromium project (http://bkaprt.com/psvg/11-15/).
- **Bugzilla:** A collection of SVG bugs reported to the Mozilla project (http://bkaprt.com/psvg/11-16/).

ACKNOWLEDGMENTS

Special thanks to Lea Verou, who was writing about SVG before it got cool again and who introduced me to Chris Lilley—whom I can thank not only for tech editing this book but, more importantly, for ushering SVG into fruition in the first place.

Thanks to Sara Soueidan who has been tirelessly advocating SVG to front-end developers and helping them see the light.

Thanks to Katel LeDû for shepherding this book's creation and to Caren Litherland for keeping my foot out of my mouth and making sure my sentences done read more gooder (sorry). Thanks to the other authors and everyone involved at A Book Apart, the existence of which inspires me. I feel honored to be a part of it.

I'd like to shout out to Fabrice Weinberg, who created the original svgstore Grunt plugin based on an idea I had to automate an SVG icon system. I hopped on that right away and haven't looked back.

Props to the team at Noun Project who have been curating the world's greatest source of SVG iconography. They bet on SVG early on and I'm very glad they did.

REFERENCES

Shortened URLs are numbered sequentially; the related long URLs are listed below for reference.

Introduction

00-01 http://thenounproject.com/term/dog/364/

00-02 https://abookapart.com/products/responsible-responsive-design

00-03 http://5by5.tv/webahead/67

00-04 https://en.wikipedia.org/wiki/Broadband#cite_note-20

00-05 https://developers.google.com/speed/webp/faq

Chapter 1

01-01 http://codepen.io/chriscoyier/pen/qEdzqB

01-02 http://codepen.io/chriscoyier/pen/XJmJjX/

01-03 http://codepen.io/chriscoyier/pen/pvEGVm

01-04 http://codepen.io/chriscoyier/pen/PwWPNa

01-05 http://caniuse.com/#cats=SVG

Chapter 2

02-01 http://www.adobe.com/products/illustrator.html

02-02 http://bohemiancoding.com/sketch/

02-03 https://inkscape.org/en/

02-04 http://svg-edit.googlecode.com/svn/branches/stable/editor/svg-editor.html

02-05 https://github.com/duopixel/Method-Draw

02-06 http://editor.method.ac/

02-07 https://github.com/artursapek/mondrian

Chapter 3

03-01 http://designingforperformance.com

03-02 https://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/

03-03 https://css-tricks.com/forums/topic/swapping-out-svg-icons-problems-with-mobile-safari/

03-04 https://github.com/jonathantneal/svg4everybody
03-05 https://css-tricks.com/examples/svg-fallbacks/
03-06 https://css-tricks.com/icon-fonts-vs-svg/
03-07 https://css-tricks.com/svg-fragment-identifiers-work/

Chapter 4

04-01 https://icomoon.io/app/#/select
04-02 http://24ways.org/2013/grunt-is-not-weird-and-hard/
04-03 https://github.com/FWeinb/grunt-svgstore
04-04 http://feedback.livereload.com/knowledgebase/articles/86242-how-do-i-install-and-use-the-browser-extensions
04-05 http://filamentgroup.com/lab/grunticon-2.html
04-06 http://ianfeather.co.uk/ten-reasons-we-switched-from-an-icon-font-to-svg/
04-07 http://gulpjs.com
04-08 https://github.com/w0rm/gulp-svgstore
04-09 http://codepen.io/chriscoyier/pen/rerEYW
04-10 https://github.com/KenPowers/gulp-cheerio
04-11 http://codepen.io/chriscoyier/pen/yOqdve
04-12 https://github.com/broccolijs/broccoli

Chapter 5

05-01 https://github.com/svg/svgo
05-02 https://github.com/sindresorhus/grunt-svgmin
05-03 https://github.com/svg/svgo/tree/master/plugins
05-04 https://github.com/svg/svgo-gui
05-05 http://codedread.com/scour/
05-06 http://petercollingridge.appspot.com/svg-optimiser
05-07 http://petercollingridge.appspot.com/svg-editor/
05-08 https://jakearchibald.github.io/svgomg/

Chapter 6

06-01 http://codepen.io/chriscoyier/pen/MYJBgR/
06-02 http://sarasoueidan.com/blog/svg-coordinate-systems/
06-03 https://css-tricks.com/scale-svg/

06-04 http://alistapart.com/article/responsive-web-design

06-05 http://responsivelogos.co.uk

Chapter 7

07-01 http://codepen.io/team/wufoo/pen/ZYQQNQ

07-02 http://codepen.io/chriscoyier/pen/emvJjG

07-03 https://css-tricks.com/svg-animation-on-css-transforms/

07-04 http://en.wikipedia.org/wiki/Synchronized_Multimedia_Integration_Language

07-05 http://codepen.io/chriscoyier/pen/9b1b826a238a601b4122f79b7017c443

07-06 https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/5o0yiO440LM

07-07 https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/6509024-svg-animation-elements

07-08 http://css-tricks.com/weighing-svg-animation-techniques-benchmarks/

07-09 https://www.youtube.com/watch?v=1ZWugkJV5Ks

07-10 https://css-tricks.com/smil-is-dead-long-live-smil-a-guide-to-alternatives-to-smil-features/

07-11 http://codepen.io/chriscoyier/blog/examples-of-animatetransform

07-12 http://codepen.io/chriscoyier/pen/MYxoVa?editors=110

07-13 http://lea.verou.me/2012/02/moving-an-element-along-a-circle/

07-14 https://www.chromestatus.com/feature/6190642178818048

07-15 http://css-tricks.com/guide-svg-animations-smil/

07-16 http://www.w3.org/TR/SVG/animate.html#AnimateElement

07-17 http://jakearchibald.com/2013/animated-line-drawing-svg/

07-18 https://css-tricks.com/svg-line-animation-works/

07-19 http://codepen.io/chriscoyier/pen/baea7cfc6f133ca14bd414c7b82b287e

07-20 http://css-tricks.com/using-requestanimationframe/

07-21 http://css-tricks.com/svg-animation-on-css-transforms/

07-22 http://codepen.io/chriscoyier/pen/wBJMJw

07-23 http://snapsvg.io/docs/

07-24 http://codepen.io/tag/snapsvg/

07-25 https://developers.google.com/web/fundamentals/look-and-feel/animations/css-vs-javascript

07-26 https://www.youtube.com/watch?v=1ZWugkJV5Ks&feature=youtu.be

07-27 http://codepen.io/GreenSock/

Chapter 8

08-01 http://avatars.adorable.io/

08-02 http://codepen.io/lbebber/pen/LELBEo

08-03 http://www.w3.org/TR/SVG/filters.html

08-04 http://codepen.io/chriscoyier/pen/xbqdav

08-05 http://codepen.io/chriscoyier/pen/MYmbwv/

08-06 http://mcc.id.au/blog/2012/12/mask-type

08-07 http://peterhrynkow.com/how-to-compress-a-png-like-a-jpeg/

08-08 http://codepen.io/yoksel/pen/xmshn

08-09 http://codepen.io/yoksel/pen/smdFh

Chapter 9

09-01 http://benhowdle.im/svgeezy/

09-02 http://responsivenews.co.uk/post/18948466399/cutting-the-mustard

09-03 https://dirkgroenen.github.io/SVGMagic/

09-04 https://github.com/scottjehl/picturefill

09-05 http://codepen.io/chriscoyier/pen/zxwMQv

09-06 http://codepen.io/davatron5000/pen/GgqWGm?editors=101

09-07 http://www.sitepoint.com/tips-accessible-svg/

09-08 http://www.amcharts.com

09-09 http://www.highcharts.com/

09-10 http://www.freedomscientific.com/JAWSHQ/JAWSHeadquarters01

Conclusion

10-01 http://css-tricks.com/mega-list-svg-information/

Resources

11-01 https://developer.mozilla.org/en-US/docs/Web/SVG

11-02 https://css-tricks.com/a-complete-guide-to-svg-fallbacks/

11-03 https://developers.google.com/web/fundamentals/design-and-ui/media/images/use-icons?hl=en

11-04 https://github.com/filamentgroup/grunticon

11-05 https://github.com/jkphl/gulp-svg-sprite

11-06 https://sarasoueidan.com/blog/svg-coordinate-systems/

11-07 http://www.w3.org/TR/SVG11/

11-08 https://svgwg.org/svg2-draft

11-09 https://medium.com/@chrisnager/bespoke-svg-reference-e22eb733272

11-10 http://codepen.io/collection/svfAa

11-11 https://www.owasp.org/images/0/03/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf

11-12 http://caniuse.com/#search=svg

11-13 https://www.youtube.com/watch?v=SeLOt_BRAqc

11-14 https://github.com/emilbjorklund/svg-weirdness/issues

11-15 https://bugs.chromium.org/p/chromium/issues/list?can=2&q=svg&colspec=ID+Pri+M+Week+ReleaseBlock+Cr+Status+Owner+Summary+OS+Modified&x=m&y=releaseblock&cells=tiles

11-16 https://bugzilla.mozilla.org/buglist.cgi?quicksearch=svg

INDEX

W

X

Z

ABOUT THE AUTHOR

Chris Coyier is a web designer and developer. He writes about all things web at CSS-Tricks, talks about all things web at conferences around the world and on his podcast ShopTalk, and cofounded the web-coding playground CodePen.

Photo by James Willamor

ABOUT A BOOK APART

We cover the emerging and essential topics in web design and development with style, clarity, and above all, brevity—because working designer-developers can't afford to waste time.

COLOPHON

The text is set in FF Yoga and its companion, FF Yoga Sans, both by Xavier Dupré. Headlines and cover are set in Titling Gothic by David Berlow.