O'REILLY®

Early Release

RAW & UNEDITED

SVG Animations

FROM COMMON UX IMPLEMENTATIONS TO COMPLEX RESPONSIVE ANIMATION

SVG Animations

Sarah Drasner

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SVG Animations
by Sarah Drasner

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition
2016-12-09: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491939635* for release details.

978-1-491-93963-5

[FILL IN]

Table of Contents

1. Chapter 1: The Anatomy of an SVG. 5
2. Chapter 2: Animating with CSS. 17
3. Chapter 3: CSS Animation and Hand-Drawn SVG Sprites. 25
4. Chapter 4: Creating a Responsive SVG Sprite. 41
5. Chapter 5: UI/UX Animations with CSS. 51
6. Chapter 6: Animating Data Visualizations. 69

CHAPTER 1

Chapter 1: The Anatomy of an SVG

Scalable Vector Graphics are becoming increasingly popular as a means of serving images on the web. The format's advantages can be drawn from its name:

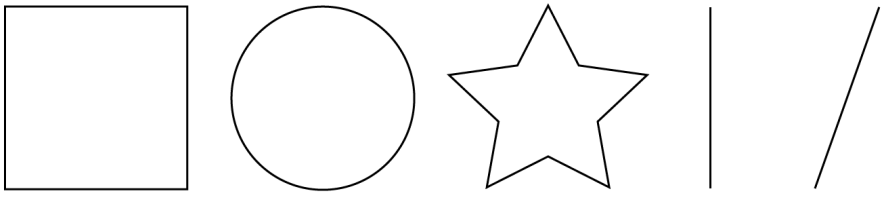
- SVG images are *Scalable*, which in an age of increasingly varied viewport sizes, is a huge boon to development. Even with srcset and the picture element, we have to cut different sizes. SVG avoids any need for image replacement, along with it's subsequent potential HTTP requests.
- *Vector* (rather than raster) meaning that because they are drawn with math, SVG files tend to have greater performance and smaller file size.

SVG is an XML file format, which means that one is able to describe shapes, lines, and text, while still offering a navigable DOM, which also means it can be performant *and* accessible.

In this first chapter, we'll lay the foundation of understanding of what this DOM is comprised of, because we'll be reaching within it in order to create complex animations. We'll be going over some of the syntax within the SVG DOM so that you know exactly what you're manipulating and can debug as needed. We won't be doing a deep dive into everything that the SVG DOM has to offer because it's out of the scope of this book. If you'd like more backstory, SVG Essentials and SVG Colors, Patterns, and Gradients are great resources.

SVG DOM Syntax

Consider this SVG graphic, with its resulting code:

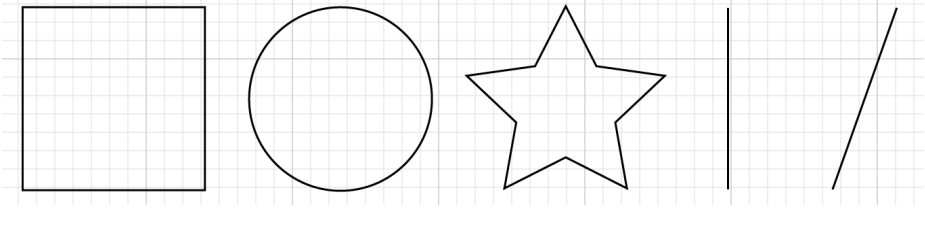


```
<svg x="0px" y="0px" width="450px" height="100px" viewBox="0 0 450 100">
 <rect x="10" y="5" fill="white" stroke="black" width="90" height="90"/>
 <circle fill="white" stroke="black" cx="170" cy="50" r="45"/>
 <polygon fill="white" stroke="black" points="279,5 294,35 328,40 303,62 309,94 279,79 248,94 254,
 <line fill="none" stroke="black" x1="410" y1="95" x2="440" y2="6"/>
 <line fill="none" stroke="black" x1="360" y1="6" x2="360" y2="95"/>
</svg>
```

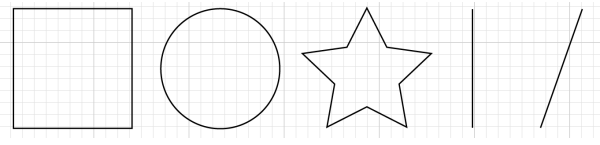
Looking at the SVG structure, most of the markup may appear familiar to you. The syntax is easy to read because of the commonalities shared with HTML. In the root <svg> element, we see a declaration of x and y values- both set to zero here, for the points in the coordinate matrix that we're starting at. The width and height are both designated, and you'll see that they correspond to the last two values in the `viewBox`.

ViewBox and PreserveAspectRatio

The SVG `viewBox` is a very powerful attribute, as it allows the SVG canvas to be truly be infinite, while controlling and refiining the viewable space. The four values here are as follows: `x`, `y`, `width`, and `height`. This space does not refer to pixels, but rather a more malleable space that can be adjusted to many different scales. Think of this as mapping out shapes and drawings on a piece of graph paper:

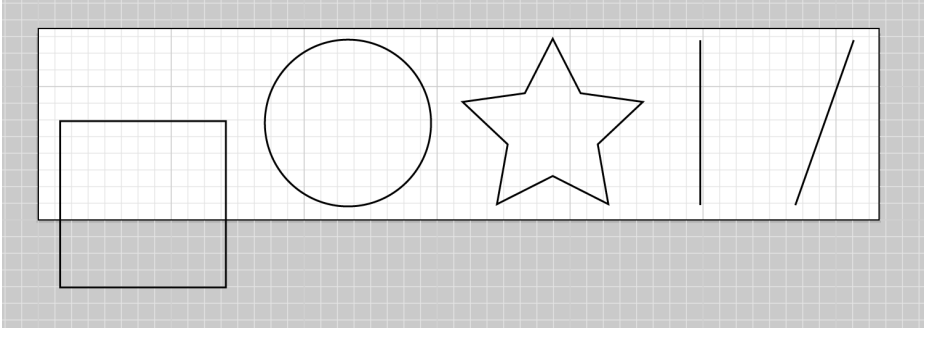


We can define coordinates based on this system, and the system itself can be self contained. We can then alter the size of this sheet of paper, and everything within it. If we designate half the width and height for the SVG, but retain the same viewbox, the result would be this:



This is part of the reason why SVG is such a powerful tool for responsive development- it can adjust to multiple viewports very easily.

SVG also stores information outside the `viewBox` area. If we move a shape outside this space, we'll see this:



The white area is what the viewer sees, while the white and grey area together hold the information that the SVG actually *contains*. This feature allows the SVG to be both scalable and easy to crop on the fly. This comes in very handy in responsive applications, particularly sprites.

There is one more tool you should be aware of with the `viewBox` and it is invisible in this example. Most SVGs you will see on the web won't even specify it because the default is what most people will want more than nine times out of ten. It is `preserveAspectRatio="xMidYMidmeet"`. This forces the drawing area to adjust itself with uniform scaling.

There are several other options as well. The first parameter, `xMidYMid`, determines whether or not to uniformly scale the element, and which part of the viewport to scale from, in camelCased naming. The default is to scale from the center, or Mid, but there are also Min or Max parameters as well, such as `xMinYMax`, and so on. You may also designate `none`, in which case the aspect ratio at its default percentages will be ignored, and it will be squashed and stretched to fill the available space.

The second value is defined by `meet` or `slice`. `meet` will attempt to scale the graphic as much as possible to fit inside containing `viewBox`, while keeping the aspect ratio

consistent. This functionality is similar to `background-size: contain;` in that the image will stay contained in the boundaries of the containing unit.

`slice` will allow the graphic within the `viewBox` to expand beyond what the user sees in the direction specified, while filling up the available area. You can think about it like `background-size: cover;` in that the image will push beyond the boundaries of the containing unit to fill up the available user space.

Further Resources

Sara Soueidan has an extremely intuitive and helpful interactive demo for you to play with in order to see this system in action: http://sarasoueidan.com/demos/interactive-svg-coordinate-system/index.html.

Amelia Bellamy-Royds has a great resource on CSS-Tricks with tons of cool demos: https://css-tricks.com/scale-svg/

Joni Trythall has a really nice resource about the viewBox and viewport as well: http://jonibologna.com/svg-viewbox-and-viewport/

Drawing Shapes

Within the SVG, we've defined five shapes. `rect` refers to rectangle or square. the x and y values, just as with the SVG itself, are where the shape begins : in this case, it's upper left corner. The shape's width and height use the same coordinate system.

```
<rect x="10" y="5" fill="white" stroke="black" width="90" height="90"/>
```

The fill and the stroke are designated here as black and white, and if nothing was specified here, the fill would default to black and the stroke would be none (i.e. invisible.

```
<circle fill="white" stroke="black" cx="170" cy="50" r="45"/>
```

Circle refers to, you guessed it, a circle- `cx` is the point where the center of the circle lies on the `x` axis, `cy` is the point where the center of the circle lies on the `y` axis, and `r` is the radius. You can also use an ellipse for oval shapes, the only difference being there are two radius values- `rx` and `ry`.

Polygon passes an array of values in a space-separated list, defined by points:

```
<polygon fill="white" stroke="black" points="279,5 294,35 328,40 303,62
309,94 279,79 248,94 254,62 230,39 263,35"/>
```

As you might assume, the first value refers to the x coordinate position, comma-separated from its matching y value to plot the points of this shape. The sides do not need to be equal in length.

Lines are fairly straightforward:

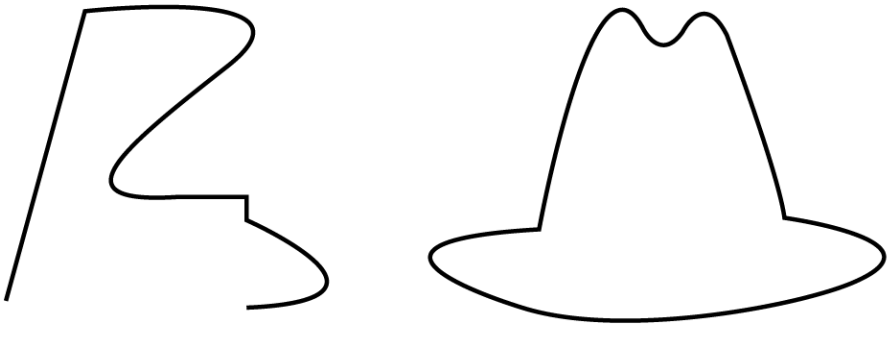
```
<line fill="none" stroke="black" x1="410" y1="95" x2="440" y2="6"/>

<line fill="none" stroke="black" x1="360" y1="6" x2="360" y2="95"/>
```

The first point is plotted at the x1 and y1 values, and end at the x2 and y2 values. I've shown two here so that you can see the syntax stay consistent whether or not the line is straight or on a diagonal. One difference is that if we

Responsive SVG, Grouping and Drawing Paths

Now, let's consider the following graphic and it's resulting code:



```
<svg viewBox="0 0 218.8 87.1">
 <g fill="none" stroke="#000">
   <path d="M7.3 75L25.9 6.8s58.4-6.4 33.5 13-41.1 32.8-11.2 30.8h15.9v5.5s42.6 18.8 0 20.6" />
   <path d="M133.1 58.2s12.7-69.2 24.4-47.5c0 0 4.1 8.6 9.5.9 0 0 5-10 10.4.9 0 0 12.2 32.6 13.6 4
 </g>
</svg>
```

The first thing we can notice about this SVG is that we've removed the width and height definitions. You can declare it elsewhere (usually in the CSS, though other methods work as well), which makes it very maleable, especially for responsive.

Width and Height Overrides

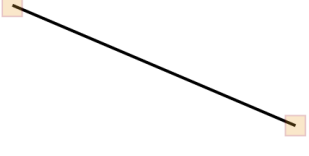
It's nice and easy to have CSS control all of the sizing and keep it in one place, but I also sometimes leave the width and height in for times I'm worried about the CSS not loading properly. If there's no fallback to the width and height inline, the SVG will scale to the available space, which can look pretty ostentatious. For that reason, you may consider writing it inline as well. The CSS will override the presentational attributes (but not inline styles).

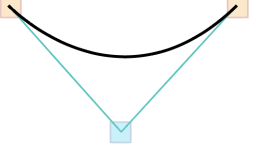
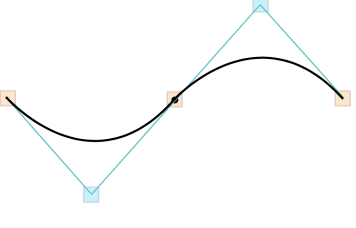
It can now scale in percentage, viewport units, and even be affected by media queries. The one catch is that you must declare a `viewBox` in this instance, it is no longer optional. The default behavior of an svg with width and height removed and `viewBox` declared is to scale to the maximum parameters of the containing element, which may be the <body>, a , or most anything else.

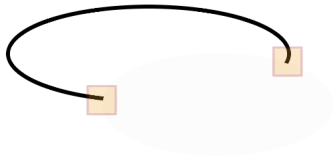
The second thing I'd like to point out is the `g` element. `g` stands for Group, and it's a way to nest and assemble multiple elements together in the SVG DOM. You may also notice that rather than defining the fill and stroke on elements themselves, we've done so on the group, and you can see it apply across the descendants.

The last and very pertinent thing to note is the path and path syntax. The path begins with d for data and is always designated with the `M` or `m` for `moveTo` command as the first value. This establishes a new point. Unlike the polygon/polyline command, the coordinates specified here are not always points on the final line.

The table below shows what each letter in a path means. Letters may be capital or lowercase. Capital letters specify an absolute coordinate, while lowercase establishes a relative coordinate.

Path Letter	Path Meaning	Image, where applicable
M, m	moveTo	start of the path, start of a new path
L, l	lineTo	
H, h	horizontal line drawn from current position	

Path Letter	Path Meaning	Image, where applicable
V, v	vertical line drawn from current position	
Curve Commands		
C,c	cubic-bezier	
S, s	reflecting cubic-bezier	
Q, q	quadratic bezier- where both sides share the same control point	
T, t	command control point that's been reflected	

Path Letter	Path Meaning	Image, where applicable
A,a	elliptical arc	
Ending Path		
Z, z	joins the end of a path to the most recent moveTo command	end of the path

Revisiting the graphic and code above, you can see the difference between the paths by noting which one has a z at the end of its path data.

Delving further into path data is out of the scope of this book, but for a great interactive demo on how path and path syntax works, check out: http://codepen.io/netsi1964/pen/pJzWoz

SVG on Export, Recommendations, and Optimization

You can absolutely create an SVG by hand, or create the SVG drawing with JavaScript with tools like d3.js. However, there are times when you may want to design and build the SVG in a graphics editor such as Adobe Illustrator, Sketch, or Inkscape. Layers inside the graphical interface will be exported as groups, complete with ids assigned as you have named your layers. You may find, though, that upon export, your SVG has a lot of information that the code in the examples above does not:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 18.1.1, SVG Export Plug-In . SVG Version: 6.00 Build 0)  -->
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/
     width="218.8px" height="87.1px" viewBox="0 0 218.8 87.1" enable-background="new 0 0 218.8 87.
  <g>
    <path fill="#FFFFFF" stroke="#000000" stroke-miterlimit="10" d="M133.1,58.2c0,0,12.7-69.2,24.4
        c0,0,5-10,10.4,0.9c0,0,12.2,32.6,13.6,43c0,0,39.8,5.4,15.8,15.4c-13.2,5.5-53.8,13.1-77.4,5
        />
    <path fill="#FFFFFF" stroke="#000000" stroke-miterlimit="10" d="M6.7,61.4c0,0-3.3-55.2,20.8-54
        s8.6-31.2,32.1-15.8S86.7,41,77.2,61.8C70.4,76.8,76.8,79,37.9,79c-0.4,0-0.9,0.1-1.3,0.1C9,8
  </g>
</svg>
```

Here's the earlier code again for comparison:

```
<svg viewBox="0 0 218.8 87.1">
 <g fill="none" stroke="#000">
   <path d="M7.3 75L25.9 6.8s58.4-6.4 33.5 13-41.1 32.8-11.2 30.8h15.9v5.5s42.6 18.8 0 20.6" />
   <path d="M133.1 58.2s12.7-69.2 24.4-47.5c0 0 4.1 8.6 9.5.9 0 0 5-10 10.4.9 0 0 12.2 32.6 13.6 4
```

```
  </g>
</svg>
```

You can see it's much smaller: without proper optimization, you can easily bloat SVG code.

Illustrator Tip

Be sure to use Illustrator's "Export as" setting to save off an SVG rather than using "Save As". This is only available in Illustrator CC 2015.2 or later.

Save as has older presets and is not fully optimized the way that the export setting has been adjusted to work. You'll end up with a much smaller and more precise output this way. I personally always retain a copy or several copies of the .ai source, because sometimes heavily modified SVGs don't backport well into Illustrator.

Some of this information is useful, and some we can do away with. The comment about Illustrator generating the code can certainly be removed. We also do not need the version, or layer information, as the web will not use it and we're trying to transmit as few bytes as possible.

If the `x` and `y` are defined as zero (usually the case), we can strip that out, too. The only case where we'd want to leave them in is if we're working with a child SVG nested inside of another SVG.

We can also strip away the xml definitions if we are using an SVG inline. I will recommend using inline SVG for animations throughout this book because the support for animation is stronger and it has less gotchas. However, there are times when using an SVG as a background image works well for animation as well, particular in chapter 3, when we talk about sprites. If you decide to use the SVG in an object or image, you should keep this xml markup, and sometimes leaving it out can cause issues in Microsoft browsers:

```
xmlns="http://www.w3.org/2000/svg"
```

If you're not sure whether to use it or not, it's better to err on the side of leaving it in.

You can also optimize paths. There are times that a graphics editor like Illustrator will export path data in decimal places that are extraneous and can be removed. It may also export with additional and unnecessary groups that clutter your code. These are only a few examples of possibilities for compression.

Reduce Path Points

If you're going to create a hand drawing, you can trace, it, but past that point you should use **Object > Path > Simplify**.

You will need to check the box that allows for preview because this can potentially ruin the image. It's also worth it to say that the image degrades quickly, so usually the most I can get away with is 91% or so. This still gives me a good return, with a high number of path point reduction.

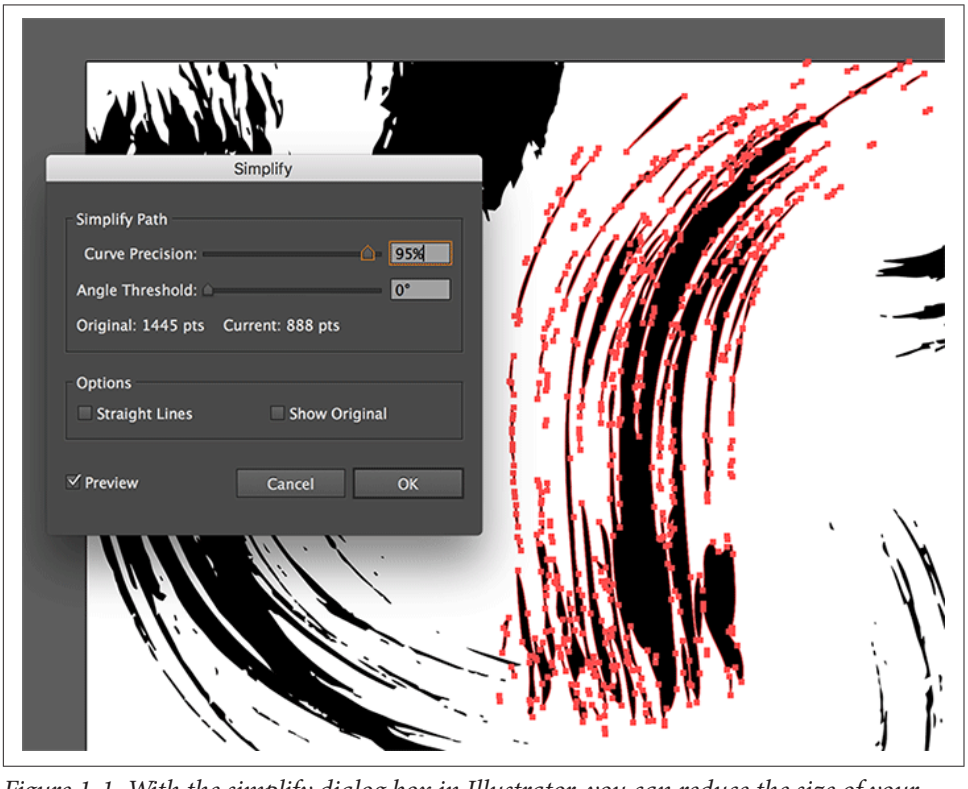


Figure 1-1. With the simplify dialog box in Illustrator, you can reduce the size of your files dramatically.

This is also probably the quickest way to accomplish this type of reduction. A more labor intensive way, that I will use for smaller pieces that are unnecessarily complex, is to redraw it with the pen tool.

Sometimes this is very little effort for a large payoff, but it really depends on the shape. You can also put a few shapes together, merge them with the path tool, and then modify the points with the white arrow, to simulate existing shapes.

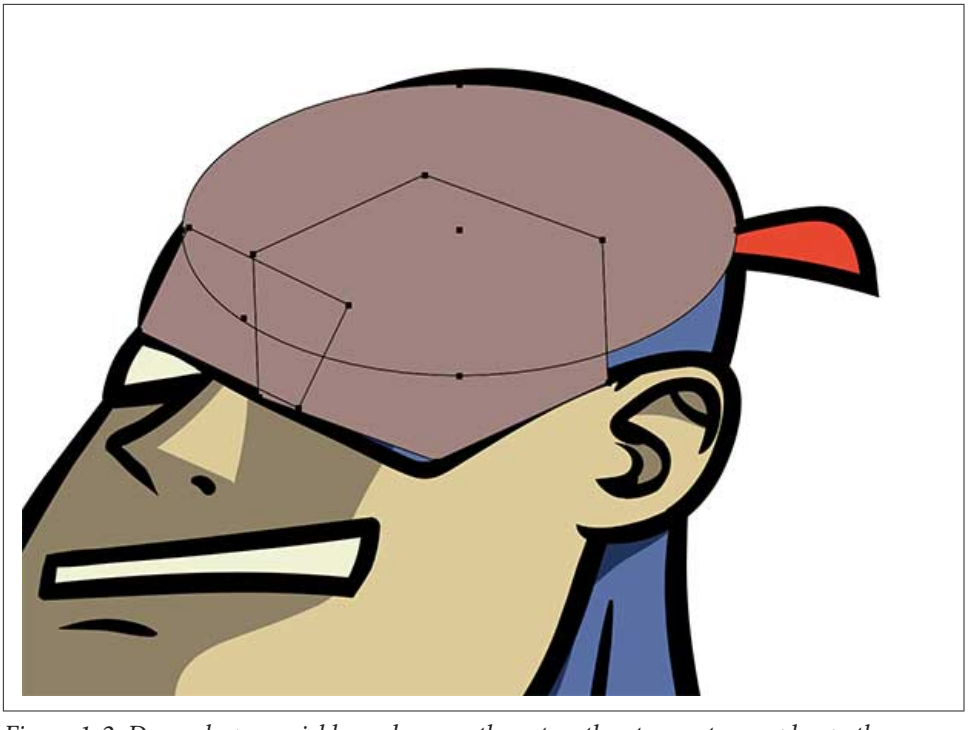


Figure 1-2. Draw shapes quickly and merge them together to create complex paths without a lot of path points

It may seem intimidating at first, but you can use the pen tool to really quickly make more complex areas. Then take all of these shapes and use the pathfinder tool to merge them all together. If it doesn't look quite right, don't fear! You can still reduce the opacity on what you made by a little (helps so that you can see what you're trying to emulate in the shape underneath). Then you can grab the direct selection tool, (A in quickkeys, the white arrow on the toolbar), and drag those small points around until you get a more refined shape. Never hurts to zoom in a bit to see the details there.

Optimization Tools

You don't need to strip this information out by hand, though. There are many great tools for optimizing SVG available, and they offer far more to help trim your code as well, such as rounding and rewriting numbers, merging path data, removing unneeded groups, and more. The list below are some of the open-source tools available, please consider that the options with a visual GUI of the output tend to be the most helpful as you can see what the optimization materializes in:

- SVGOMG- https://jakearchibald.github.io/svgomg/- Jake Archibald created a really nice web-based GUI for the terminal-based SVGO discussed later. This is the most robust and easy to work with method, with many different methods for optimizations featured with toggles. Shows the relative visual output and the byte saving comparison with the optimization and gzip.
- Peter Collingridge's SVG Editor- http://petercollingridge.appspot.com/svg-editor- very similar to SVGOMG, with slightly less options. A nice feature is that you can edit the SVG right in another panel in case you need to adjust the output just slightly. Also web-based with a nice visual interface.
- SVGO- https://github.com/svg/svgo- terminal-based, with no visual GUI- however you can add one with the SVGO-GUI- https://github.com/svg/svgo-gui. This requires a bit more setup but is a workflow boon if you're more comfortable working in your terminal than popping in and out of the browser. The functionality is what SVGOMG is powered by as well.

Please be aware that you will need to be changing and adjusting settings depending on what you're trying to achieve in your animation. Get comfortable with adjusting these options rather than settling for the defaults, as doing so will save you considerable time later. You may find that a very busy animation requires repeated optimizations while you're developing; for this reason, I recommend leaving the graphics editor and optimization tool open while working with your code editor, making your workflow as seamless as possible.

Default export settings to be aware of

Be mindful of some of the defaults when you're exporting. The ones that I find myself checking and unchecking the most are:

- **Clean IDs**- this will remove any carefully named layers you may have.
- **Collapse useless groups**- you might have grouped them to animate them all together, or just to keep things organized.
- **Merge paths**- nine times out of ten this one is ok, but sometimes merging a lot of paths keeps you from being about to move elements in the DOM around independently.
- **Prettify**- This is only necessary when you need to working within the SVG, for animation or other manipulation purposes.

CHAPTER 2

Chapter 2: Animating with CSS

You may find working with SVG code feels very familiar, mostly because an SVG has a DOM, just like standard HTML markup. This is hugely valuable when working with CSS animations, as manipulating markup with CSS is already a very comfortable process for most Front-End Developers.

For a very brief review, let's first establish that a CSS animation is created by defining two parameters. The keyframes themselves:

```
@keyframes animation-name-you-pick {
  0% {
   background: blue;
   transform: translateX(0);
 }
  50% {
   background: purple;
   transform: translateX(50px);
 }
  100% {
   background: red;
   transform: translateX(100px);
 }
}
```

Keyframe Syntax Hint:

You can also define `from` and `to` instead of percentages. If you declare nothing in either the initial keyframe or the ending keyframe, the animation will use the default or declared properties on the element. It may be worth double checking your work in all browsers if you do remove them though, due to strange and inconsistent bugs.

After you define the keyframe values, you have two options for animation syntax declaration. Long form, with each declaration defined separately:

```
.ball {
  animation-name: animation-name-you-pick;
  animation-duration: 2s;
  animation-delay: 2s;
  animation-iteration-count: 3;
  animation-direction: alternate;
  animation-timing-function: ease-in-out;
  animation-fill-mode: forwards;
}
```

Or shorthand (my preferred method as it uses less code):

```
.ball {
  animation: animation-name-you-pick 2s 2s 3 alternate ease-in-out forwards;
}
```

All of the above declarations are interchangeable in order in a space-separated list, except for the number values, which must be defined in the above order: duration, delay, and iteration count.

If we apply it to this very simple .ball div

```
.ball {
  border-radius: 50%;
  width: 50px;
  height: 50px;
  margin: 20px; //so that it's not hitting the edge of the page
  background: black;
}
```

the result is this, with interstitial states shown less opaque:

All of the resulting code in action is available in this demo: http://codepen.io/sdras/pen/c00ac22c90b9232a7c5631a8158b48c1

For more information and further detail into each animation property, such as what animation-fill-mode is, what eases are available in CSS, and which properties are animatable, there is extensive information in Estelle Weyl's book here: http://shop.oreilly.com/product/0636920041658.do

or you can also consult Dudley Storey's CSS Animations book: http://www.amazon.ca/gp/product/1430247223

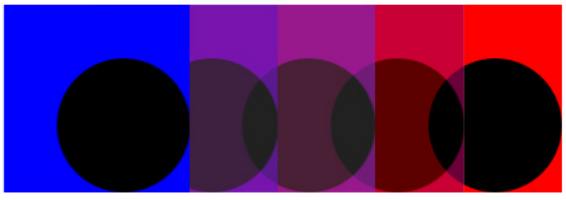
Animating with SVG

Let's say instead of drawing the ball with CSS, we had drawn it with SVG. We know how to do that from the last chapter. To get the same black circle as above, we would write:

```
<svg width="70px" height="70px" viewBox="0 0 70 70">
  <circle fill="black" cx="45" cy="45" r="25"/>
</svg>
```

We define the radius as half of 50, so 25px. Then we move the center of the circle on both the *x* and *y* axis (`cx` and `cy`) to half the radius plus that 20px margin we added in the CSS. We could also use margin on the SVG element to move it, but here I'm illustrating that you can draw coordinates alternatively directly in the SVG itself. If we move the circle over, the `viewBox` has to be larger to accommodate these coordinates, though, so it is the width plus the margin of space to the edge.

Now, if we place a class on the whole svg called ball, using the same animation declaration, we get this:



What happened here? It still moved across, as we were expecting. But the background is filling in the full background of the SVG, thus the entire viewBox. That's not really what we want. So, what happens if we move that class and target the circle instead?

You may have guessed why we have this output. There are two reasons:

1. The `circle` is moving inside the `viewBox`. Remember, if we move an internal SVG attribute, the viewBox will quite literally be a window with which you view these elements. So if we move the circle without the viewBox being large enough to accommodate those coordinates, it will be cut off when it moves out of that area.
2. SVG elements look like the HTML DOM, but are slightly different. We don't use background on SVG attributes, we use fill and stroke. An external stylesheet will also have a hard time overriding what is defined inline within the SVG. So let's take out the fill definition, and move that into our stylesheet.

The resulting code should be this:

HTML:

```
<svg width="200px" height="70px" viewBox="0 0 200 70">
 <circle class="ball3" cx="45" cy="45" r="25"/>
</svg>
```

CSS:

```
.ball3 {
 animation: second-animation 2s 2s 3 alternate ease-in-out forwards;
}

@keyframes second-animation {
  0% {
```

```
    fill: blue;
    transform: translateX(0);
  }
   50% {
    fill: purple;
    transform: translateX(50px);
  }
   100% {
    fill: red;
    transform: translateX(100px);
  }
 }
```

And the result is this, but with an SVG instead of an HTML div:

Benefits of Drawing with SVG

So, why SVG? Why learn SVG when you could also build something in CSS-styled HTML and animate that way?

First of all, even that small, simple circle was 5 lines less code altogether than the CSS version. SVG was built for drawing, unlike CSS, which was built for presentational formatting. Let's look at the code for the star from the first chapter of this book:

```
<polygon fill="white" stroke="black" points="279,5 294,35 328,40 303,62
309,94 279,79 248,94 254,62 230,39 263,35 "/>
```

It would be incredibly difficult to draw a star in CSS with such a small amount of code, and impossible to be that concise once compiled, if using a preprocessor.

Here's something I drew in Illustrator:

We could also probably draw this in CSS, but to what end? If you're working with a designer on a project, having them draw something for you in CSS is not typically an option, and drawings that you want to move can get much more complex than this. In SVG you can also make the whole image scale easily, and therefore, your whole animation can be responsive.

All of the information for the illustration is just **2KB gzipped**, as well, and can fill up a whole screen. That's pretty amazing if you consider raster image alternatives.

Applying what we just learned about the circle, we can look at some of these shapes and think about what we can do with them. We can group all of the cow together and make it jump over the moon. We can make the "surprise" expression disappear and appear. We can even make the helmet go up and down so it appears as though he's looking up. (That is actually what I did in the final animation.)

Silky-Smooth Animation

It's tempting to use all of the same properties that you use to affect layout with CSS: `margin`, `top`, `left`, etc. But browsers do not update values for all properties equally. To animate cheaply, your best bet is transforms and opacity. That might seem limiting, but transforms offer translation (positioning), scale, rotation. The combination

of these with opacity can be extremely powerful. It's surprising how much can be achieved with these properties in standard animations. Throughout this book, I will use these properties wherever possible while demonstrating various techniques. It is important to note that SVG DOM elements can't be hardware accelerated, but you should still be moving the SVG DOM with transforms and not margins or other CSS positioning.

For more information on how to properly keep your layout repaint costs low, check out:

http://jankfree.org/
http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/

and for information on the costs of individual properties, see:
http://csstriggers.com/

CHAPTER 3

Chapter 3: CSS Animation and Hand-Drawn SVG Sprites

SVG performs extremely well as an icon format, but we'll move a step further and use SVG sprites in performant complex animations using three different techniques. The first two techniques are closely related to cel animation, while the third, detailed in chapter four, is a technique I recommend for more complex responsive, and even interactive development.

Keyframe Animation With Steps() and SVG Sprites, Two Ways.

If you've ever seen a Looney Tunes or old Disney animation, you might have been impressed with the fluid movement, considering that every frame was hand-drawn. Such effects are possible on the web with SVG sprites, and we can stand on the shoulders of previous animators while employing new development techniques.

Of all web-based animation techniques, step animation most closely resembles these old hand-drawn cel animations. Cel is short for "celluloid" which is a type of transparent sheet. This material was used by animators to draw on top of their previous drawings, thereby defining a sequence and creating the illusion of movement. It functioned a bit like a flip-book. Each drawing was captured on film frame by frame. There were usually several layers to these drawings to save time- you wouldn't want to redraw the background again and again just to show the same scene.

In order to save steps in drawing, the background would be painted, and then the character, sometimes even pieces of the characters face, like the mouth or eyes, would be adjusted other layers to keep from having to repaint the character's body.

Cel Animations as Scoping

You can think of this technique like writing web page templates: you start from the base template and create smaller pieces so you can manage the individual thing that's happening in one piece separate from everything else.

Figure 3-1. Hand-painted cel with transparency. Image courtesy of John Gunn.

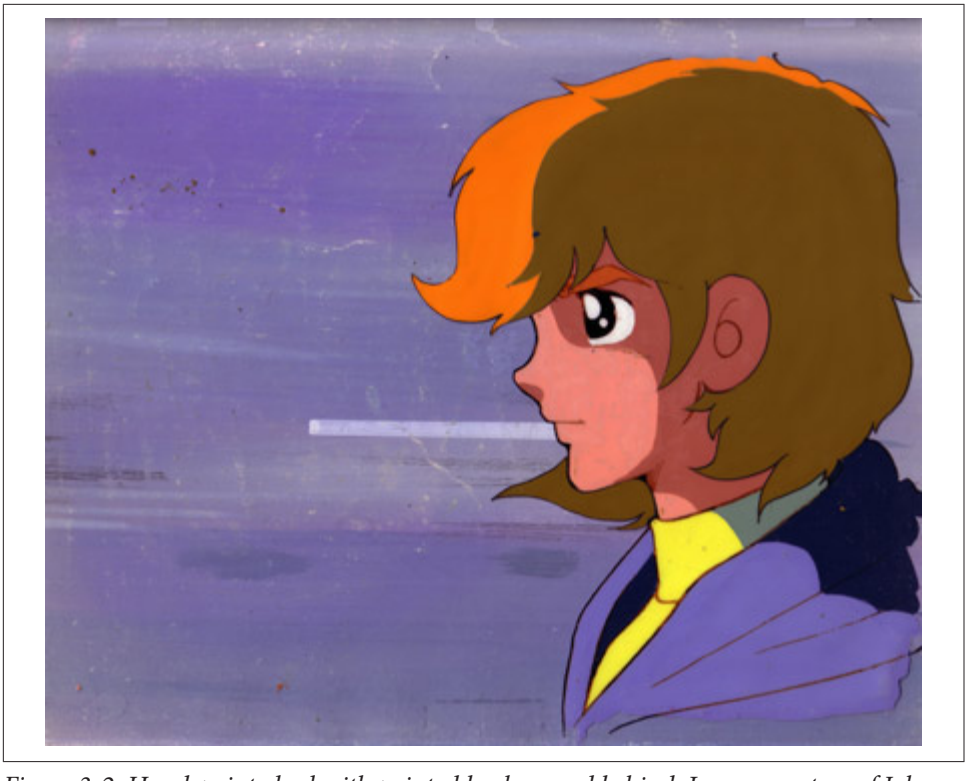


Figure 3-2. Hand-painted cel with painted background behind. Image courtesy of John Gunn.

Here, we mimic this analog process by using a single motionless background, then quickly showing a series of images on top. This gives the illusion of movement without any real interpolation. Instead of a series of separate image files though, we will simultaneously reduce HTTP requests and simplify our keyframes by using a single SVG sprite sheet. This technique is great for more complex shapes and expressive movement than simple transforms can offer.

Because this technique relies heavily on design, we'll go through the design workflow first and then go through the code. Here what the final animation looks like:

http://codepen.io/sdras/pen/LEzdea/

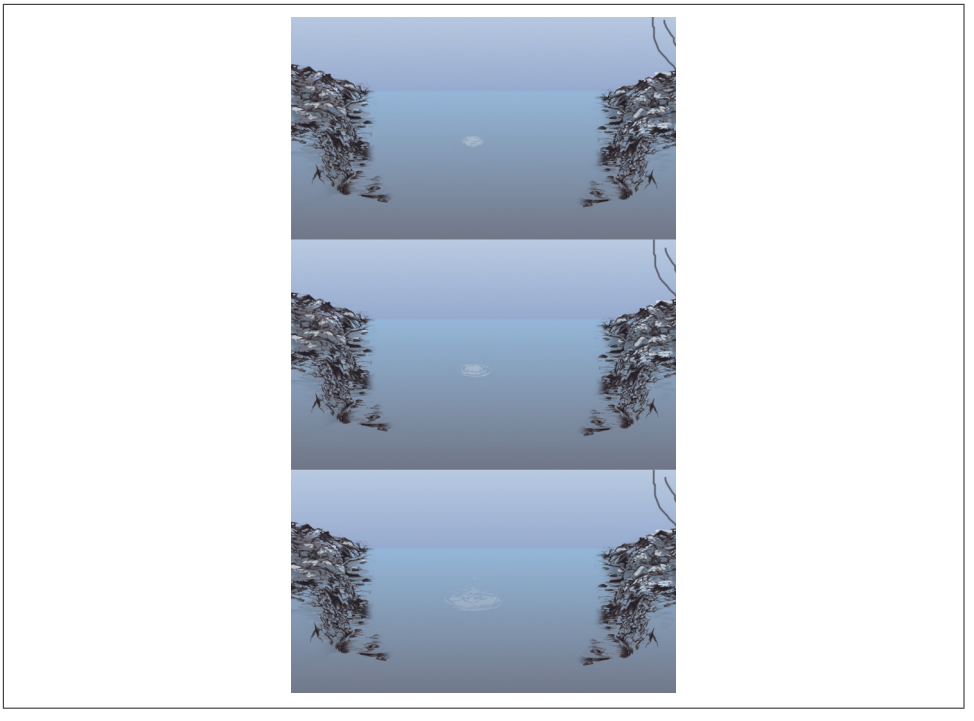


Figure 3-3. Stills for our splash animation

Typically when showing interpolated (i.e. rapidly changing) images on the web we should push the maximum frames per second possible to achieve the silkiest animation. This technique is one exception that rule. Since we have to draw every single frame, we're going to try to get as much bang for our hand-drawn buck as possible. Years ago animators spent a lot of time trying to find a good balance between realistic movement and the fewest number of drawings. Old film was shot at 24fps, and animators largely regarded "shooting on twos" (meaning one drawing over two frames, or 12fps) the standard for an illusion of movement. Dropping to anything lower than this, and your eye will discern a slight choppiness (which some animators even used as a creative decision!) We use their work in finding these bounds of illusion to our benefit, stick to the 12fps rule, and create a 21 part drawing for a 1.8s animation. The 21 here comes from the the number of frames that we chose, but can be any number you like.

There are two ways of creating the series of drawings for this type of animation; both work equally well, but use different automation processes for the images. The challenge we face in each workflow is keeping the drawing steadily placed in the center of

the frame throughout a large sprite: even the best drawing will look flawed if the drawing jumps as we run through each frame.

The "Drawing in Illustrator with a Template" way

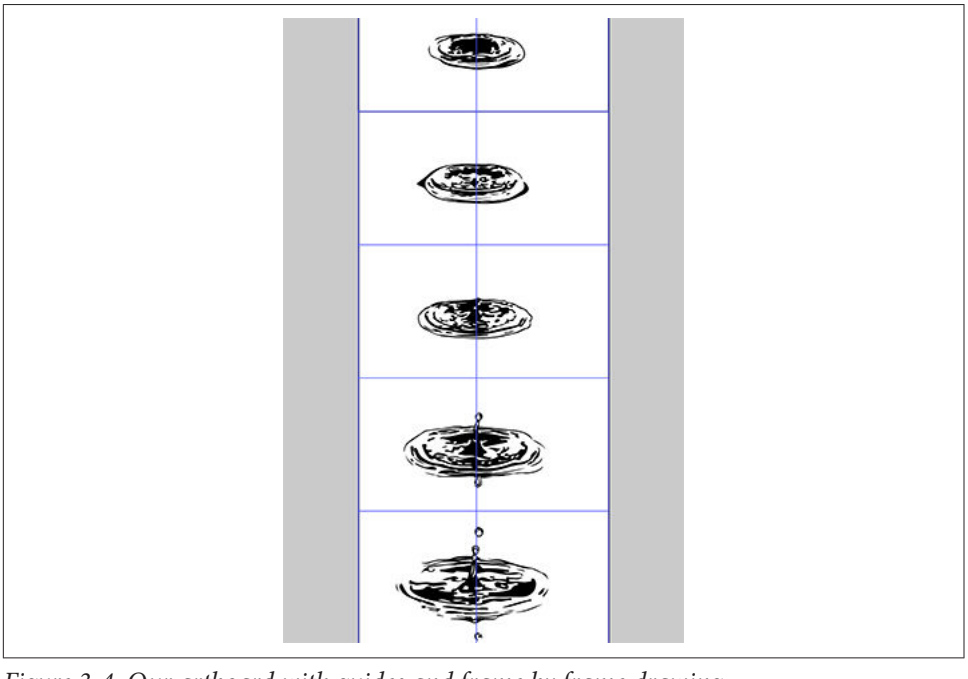


Figure 3-4. Our artboard with guides and frame by frame drawing

I use Illustrator for this technique but you could theoretically use Sketch or any other graphics editor. First, we decide how big the animation is and multiply that number by 21 in one direction (the number of frames in our animation), determining the length of our artboard. We drag a box around that area and choose **Object > Path > Split Into Grid**. Then we enter the rows we want (or columns if you wish to make a horizontal sprite sheet) and click OK. Then choose **View > Guides > Make Guides** and our template is all set.

If you're drawing directly in the graphics editor, I recommend placing your first drawing within the first box, and creating a box around it that frames it within the guides. We can then copy everything into the next box (including the box frame) using the alignment line or shift + drag, which will keep it steady. Use the box frame again to fit it into the next guides space.

Using the direct selection tool (the white arrow) you can then drag and reshape the pieces of your image for each frame. Fair warning: don't be tempted to front-load

your work here by copy-pasting it all at the start- this process works best if you build each frame from the next.

You can also do a screencast of something so that you can walk through the stills and place each image in the Illustrator doc and trace it, either through Illustrator's native trace functionality, or with the pen tool for a hand-drawn feel, and more concise paths. At the end of this process we will have a long sprite sheet. We can export that directly as an SVG, as well as a PNG, which we'll use as a fallback with a body class hook in modernizr. At this point, though, fallbacks might not be so necessary for you, so it's recommended you compare the caniuse.com tables for SVG support and compare to your analytics.

```
.splash {
  background: url('splash-sprite2.svg');
 ...
  animation: splashit 1.8s steps(21) infinite;
}

/* fallback */
.no-svg .splash {
  background: url('splash-sprite2.png');
}
```

The "Drawing in an SVG Editor or on Paper Frame-By-Frame and Using Grunt to Sprite" Way

The first process will still look like a an illustrator drawing, and you might find you like more of a hand-drawn feel. If this is the case, it's very easy to still draw by hand and scan it in. Old animation studios used lightboxes and celluloid sheets so that they could trace their previous drawing incrementally. You don't necessarily need these materials to try this technique, though. By placing a lamp underneath a glass table, you can easily make a poor-man's lightbox. This set-up shines enough light so that you can see through even regular opaque copy paper. To create each new frame, place a piece of paper or vellum over your last drawing and change the drawing slightly until you have a series. You can then scan this set of drawings and vectorize them, placing them correctly with reduced opacity and guides.

If you'd rather draw each piece frame by frame in the editor but don't know how many frames you will be creating, you can draw each one separately, shifting the image slightly each time, and saving every new version to a folder. Illustrator's new export settings are good enough that that you can do so without all the old cruft and comments. Be sure to export with **Export as > SVG** rather than save as>SVG, as this will yield better results. You must sure that what you're initially saving is indeed an SVG and not an AI (or any other) filetype. You can then use Grunticon to compress and sprite them automatically. Here's a great article explaining how to do so: http://css-tricks.com/inline-svg-grunticon-fallback/. Notably, Grunticon also generates a fallback PNG automatically.

Personally, I think if you draw each frame by hand, it makes the most sense to just make sure the placement on each artboard is consistent and use Grunticon, but the Illustrator template technique has the benefit of allowing you to see all of your work at once, which gives you more of a holistic understanding of what you're making.

Simple Code for Complex Movement

This type of sprite makes use of the smallest amount of code for the most amount of believable movement. We intentionally keep the code DRY (don't repeat yourself), simple, and clean. The greatest thing about this type of movement is that we rely enough on the sprite to not need a lot of code to achieve an illusion of movement through space.

We absolutely position a smaller area of movement because we want to show a consistent experience across desktop and mobile. Our aim is to cycle through the entire image, but stop momentarily at each individual picture in the image, and thankfully, `steps()` in CSS allows us to do just that. We've already done a lot of the heavy lifting in our design, so the code to create the effect is very small.

There's no need for complex percentages and keyframes. All we need to do is take the image height, and specify the background-position with that number as a negative integer on the 100% keyframe value based on the pixel value of the final frame:

```
@keyframes splashit {
        100% { background-position: 0 -3046px; }
}
```

Here, we don't have to make the SVG fluid to its container, because it's easy to have it take up the whole screen on mobile. On the splash div, we animate using `steps()` for the number of frames we had in the SVG:

```
.splash {
  background: url('splash-sprite2.svg');
 ...
  animation: splashit 1.8s steps(21) infinite;
}
```

Using an SVG rather than a PNG gives us the advantage of a crisp image on all displays, but providing a fallback is easily done. We use modernizr to create a class hook on body and can still animate it with the PNG we created. We don't simply use the PNG because on different resolutions, it will look fuzzy while the SVG will remain crisp.

```
/* fallback */
.no-svg .splash {
```

```
  background: url('splash-sprite2.png');
}
```

Using Modernizr

Modernizr is a feature detection library. It allows you to work with advanced features on the web while providing fallbacks, or progressively enhance features by checking to see if they are available. It's a highly customizable library that provides classes on the body that you can hook into for different experiences, like the `.no-svg` tag in the example above. I highly suggest working with a custom build for your unique purposes- the entire library is a lot of overhead you'll likely only use a small portion of it.

Simple Walk Cycle

If you take the steps() value out of the last animation, you'll see something interesting. Instead of creating a seamless moving drawing, the background rolls through. We can use that to our advantage for a nice layered background with spatial placement and movement.

Let's consider this walk cycle, which shows a ghostly figure walking through a looping, multi-dimensional, outlined landscape:

http://codepen.io/sdras/pen/azEBEZ

We use the previous technique with the cels/steps, with drawings that show a walk cycle. We can use a poor man's animation technique to change the color by shifting the color in each frame. Alternately, we could have used a filter with a shift for hue-rotate:

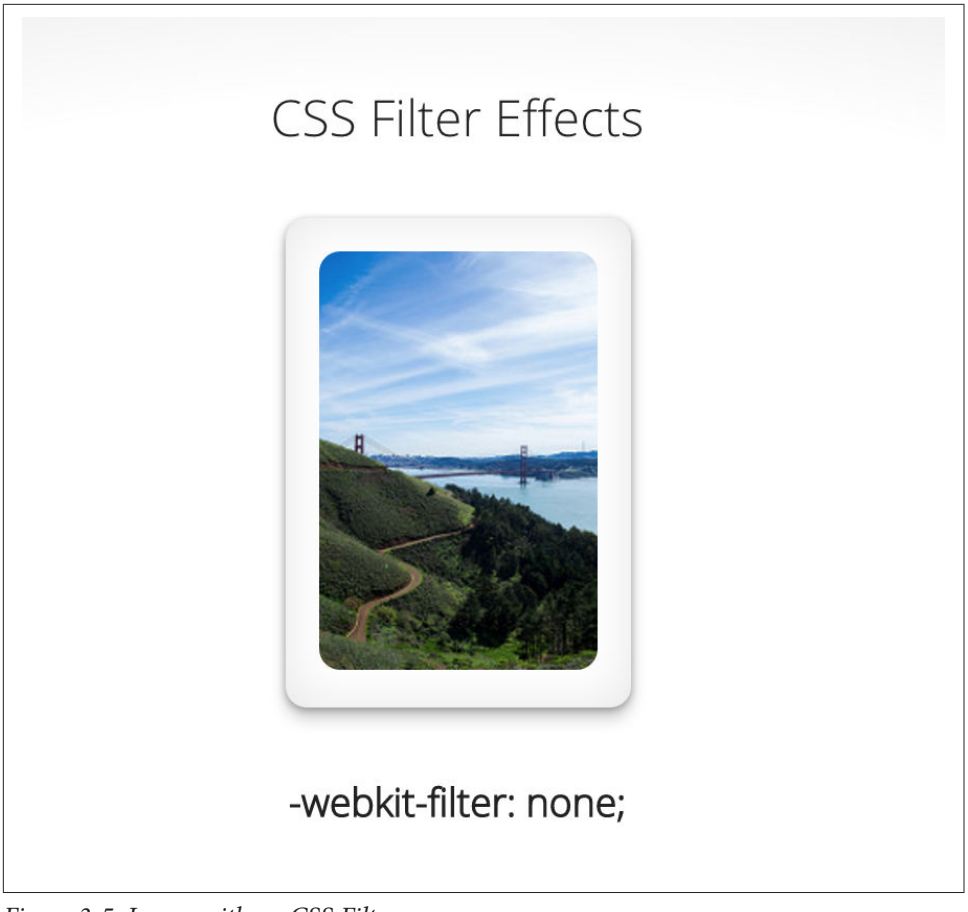


Figure 3-5. Image with no CSS Filter

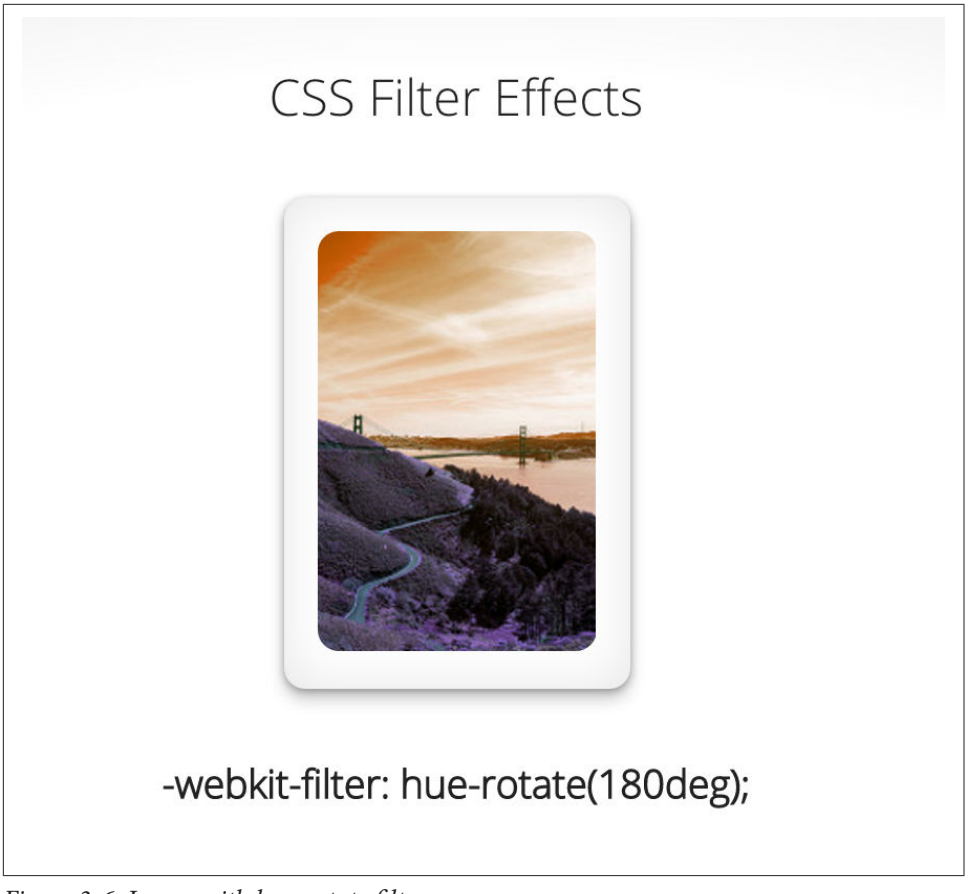


Figure 3-6. Image with hue-rotate filter

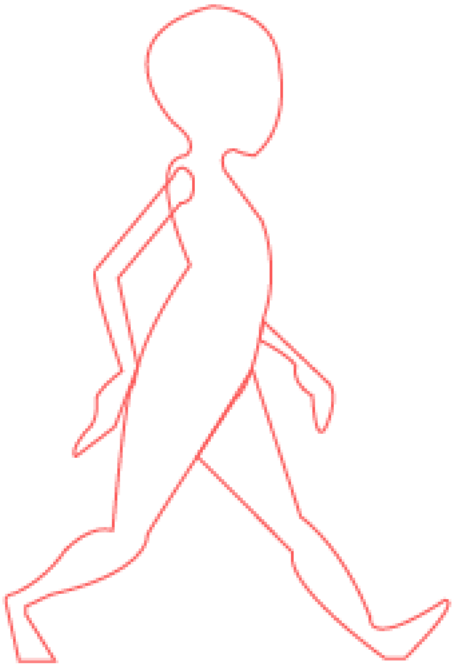
CSS Filter Demos

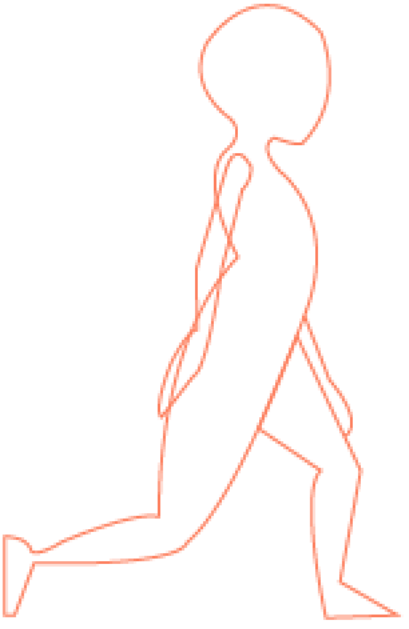
There are a number of great sites that demo great capabilities with CSS Filter effects. Here are just a few:

- The images above were taken from HTML5-Demos: http://html5-demos.appspot.com/static/css/filters/index.html
- CSS Filter Demos by Bennett Feely: http://bennettfeely.com/filters/
- CSS Reflex: http://www.cssreflex.com/css-generators/filter/
- My personal favorite, CSS Gram, by Una Kravets, which mixes filters to make some great Instagram-like effects: http://una.im/CSSgram/

But as long as we are creating all of these frames by hand, the amount of work to change the color here is minimal, and the cost of the filters on performance, while not huge, is a cost we can do without.

It's still important that the steps() and animation-duration ratio still fall around the 12fps range. We can scroll through each version of images presented by animating the background position of the SVG sprite sheet. In order to keep everything consistent, we've made all of the background images the same size, which lends itself well to the use of an `@extend` if you're working with Sass.





```
/*--extend--*/
.area {
  width: 600px;
  height: 348px;
}

.fore, .mid, .bk, .container { @extend .area; }
```

To create the impression of fluid linear infinite movement, the 3 background images must be able to repeat seamlessly on the X axis so that when they scroll through. This can be achieved by making each end identical, or in this case, using an image that is sparse enough that it can completely flow through. If you're working with the latter, it's important to marry the beginning state and end state in a graphics editor like Illustrator or Sketch to ensure it looks ok while you're building the graphic.

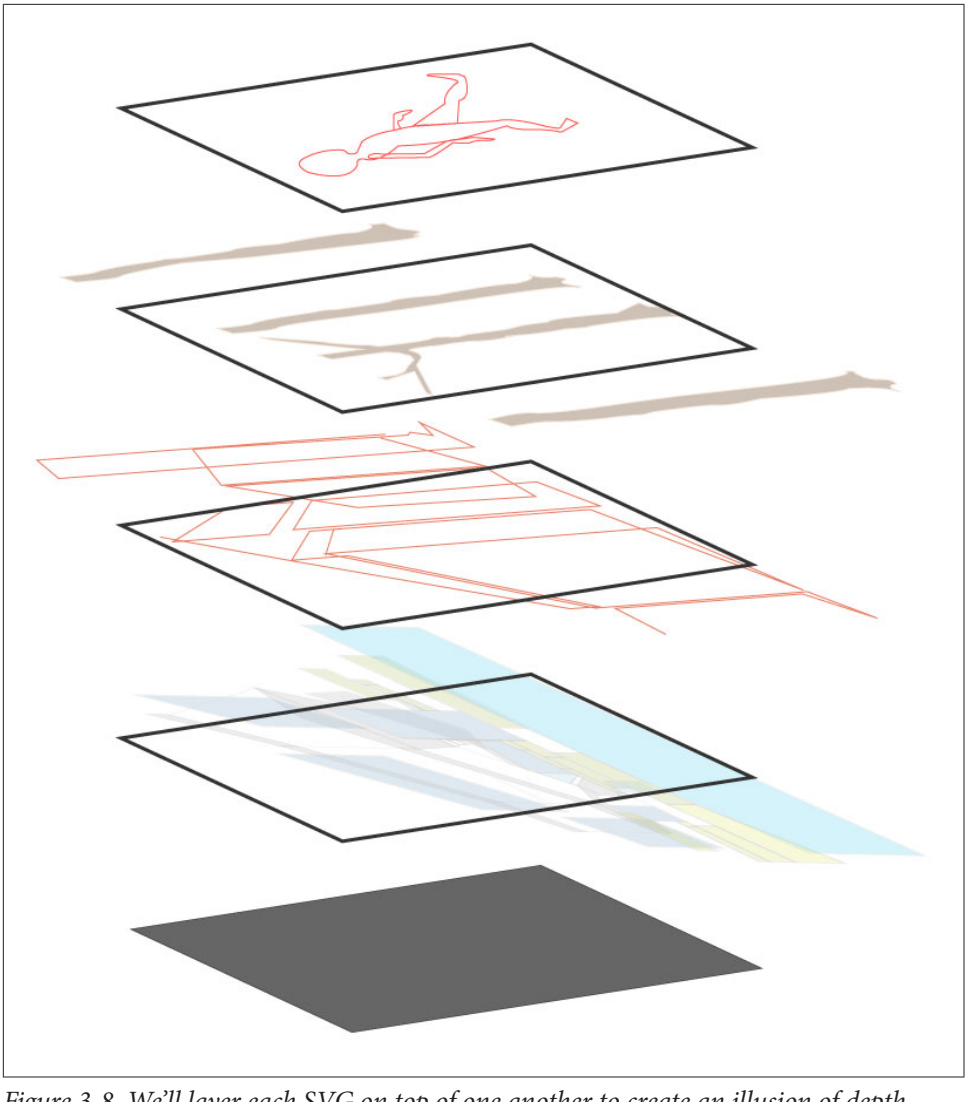


Figure 3-8. We'll layer each SVG on top of one another to create an illusion of depth.

Each element uses the same keyframe values, but we set apart their animations with an incremental decrease in seconds the further back their z-index. If you look around you, things that are closer to you are in sharper contrast and appear to move faster than things that are further away. Our animation will mimick this effect by increasing the second integer (and thus having longer animations) for the SVGs in the background.ATLAS-CURSOR-HERE This yields a nice parallax

effect. There are three parallaxed background images in this pen that don't include the figure.

```
.fore {
  background: url('fore.svg');
  animation: bk 7s -5s linear infinite;
}

.mid {
  background: url('mid.svg');
  animation: bk 15s -5s linear infinite;
}

.bk {
  background: url('bkwalk2.svg');
  animation: bk 20s -5s linear infinite;
}

@keyframes bk {
  100% { background-position: 200% 0; }
}
```

We don't need multiple intervals for this kind of animation because keyframes will interpolate values for us. In the event that the amount of pixels in the scrolling sprite sheets change in the future, we don't have to designate the amounts by setting a percentage. The use of staggered negative delays ensures that the animation is running from the start. All of the SVGs are optimized and have a PNG fallback.

CHAPTER 4

Chapter 4: Creating a Responsive SVG Sprite

The "scalable" part of SVG is perhaps the most powerful aspect of the graphics format. Using the viewBox attribute and our knowledge of shapes and paths, we can crop the SVG to any size on the fly, knowing that our intentions within the coordinate space will be preserved.

If we remove the width and height attributes from a common SVG, we'll see something interesting. The SVG expands itself to the full width of the viewport, maintaining the aspect ratio of everything within the DOM.

If we use CSS keyframes or JavaScript to move SVG attributes such as `circle` or `path` while scaling this SVG up or down, the increments that they will move will scale as well, along with the graphic. **This means that if you scale a complex SVG using percentage, flexbox, or other techniques, your animation will also scale accordingly.** This means that you don't have to adjust anything for mobile or other sizes, you can focus on writing it correctly one time.

The completed animation is completely scalable. In the CodePen below, you can randomly resize the animation while it's running and watch it instantly adjust.

http://codepen.io/sdras/full/jPLgQM/

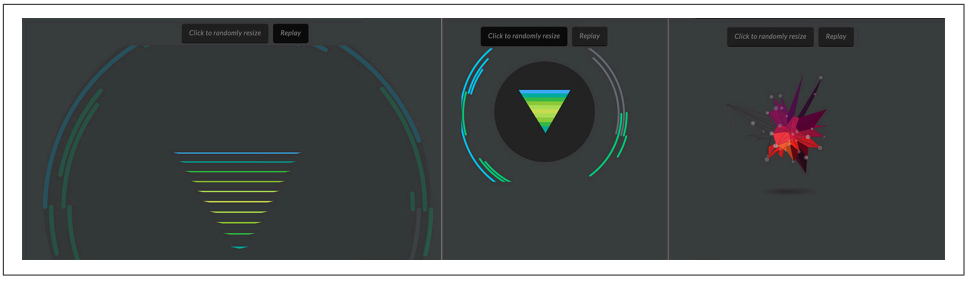


Figure 4-1. Different states of the same animation at different sizes

This becomes very useful for responsive development. The above animation uses a completely fluid approach. We design the whole thing first, and then slowly reveal things. Our initial SVG before we add any animation in looks like so:

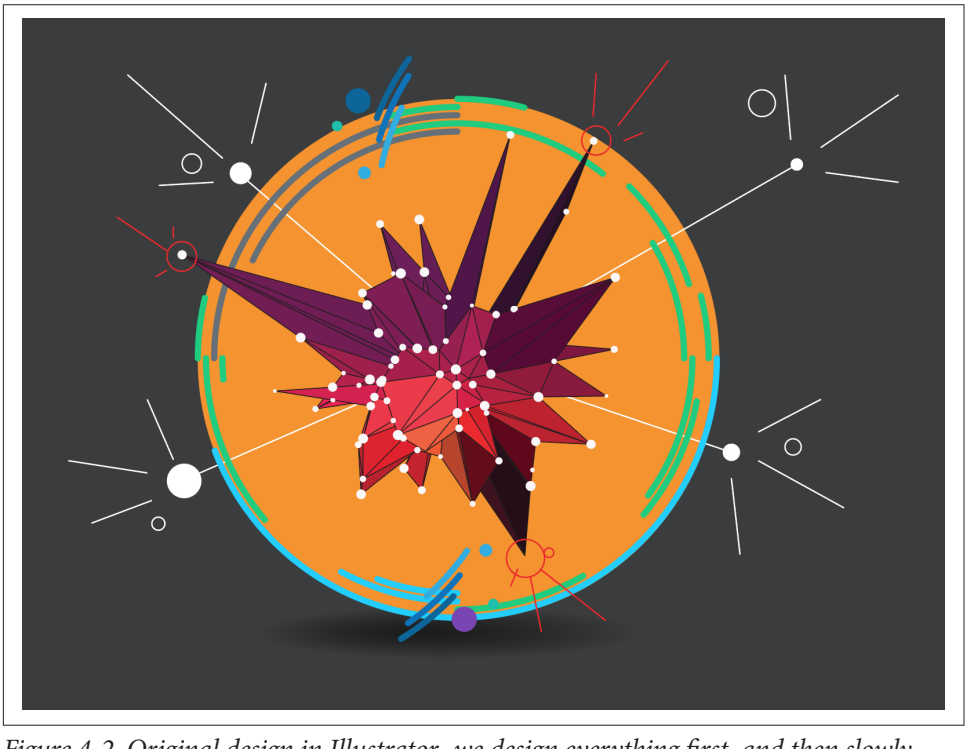


Figure 4-2. Original design in Illustrator- we design everything first, and then slowly reveal things.

We could also design for responsive SVG in two other ways. In this chapter, we'll go into a deep dive of a technique that uses SVG sprites, similar to the ones we created in Chapter 3. This is easy to work with in CSS. In chapter 10, we'll cover a more advanced JavaScript approach as we hide, show, collapse and rearrange our content.

SVG Sprite and CSS for Responsive Development

Joe Harrison has shown a really nice way of collapsing SVG sprites for less information on mobile, shown below. We're going to use that to our advantage and create a similarly, incrementally more complex sprite as we shift viewports from mobile to desktop.

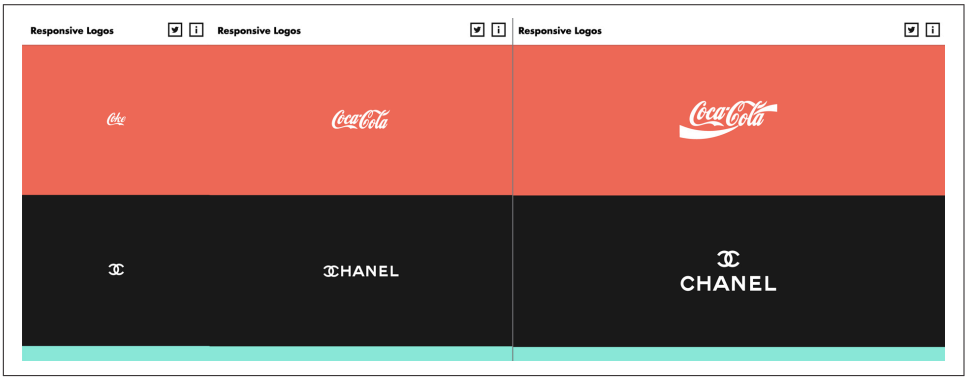


Figure 4-3. Joe Harrison's very impressive SVG sprites for logo

As our screen sizes condense and grow, the graphic also follows suit and collapses or reveals visual complexity. It's helpful to the user to be not be served overly visually complex graphics on small displays where too much information can become noise. Animation can modify with the same considerations as the typography and layout, adjusting to the viewport and clarifying the design.

Here's what we'll be making:
http://codepen.io/sdras/full/xbyopy/

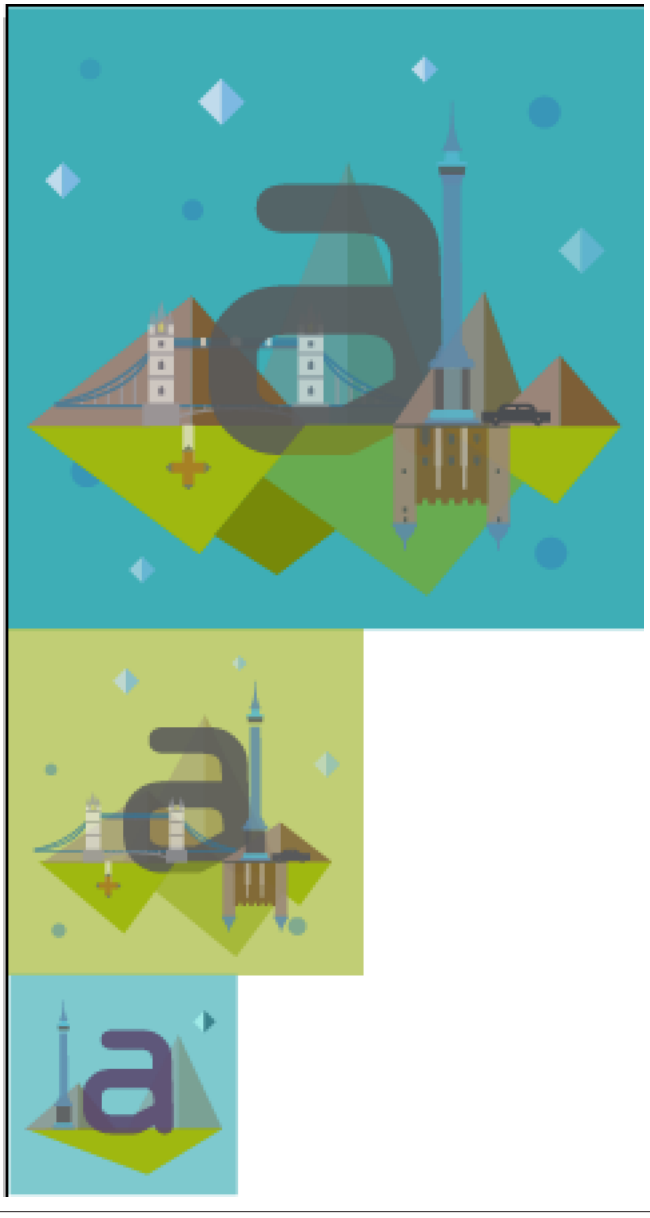


Figure 4-4. Designing our "map"

We're going to be working with this responsive illuminated drop-capital letter to show how a standalone illustration can adjust. The design was inspired by the incredibly decorated medieval manuscript, the Book of Kells, to show how a standalone drawing can adjust to different viewport sizes. We start from this design, which we'll

use as our "map". Other people plan differently, working in-browser or making sketches; choose whatever method makes you most productive.

Grouping and Drying It Out

Now that we know how the final product appears, we can refactor the design to group like sections together, based on what's most important for the associated viewport width. We can also simplify the design by identifying shapes used in both the first and second versions, keeping just one copy of each shared shape.

All of the elements are assigned semantic ID names such as "mountains", or "bridge". The most detailed shapes also get a shared ID that we can progressively show for larger viewports. If the first illustration is kells_1, the group particular to the second illustration is kells_2, and the last is kells_3.

In order to make the SVG scalable to shared container values, the last illustration becomes the same size as the first; SVG's built-in responsiveness will take care of the rest. The top graphic is more complex because it holds information for both the tablet and desktop viewport sizes.

What we end up creating only two areas of the sprite sheet, with both sharing the same width so that we can scale the whole image at once. The top graphic is more complex because it holds information for both the tablet and desktop instances.

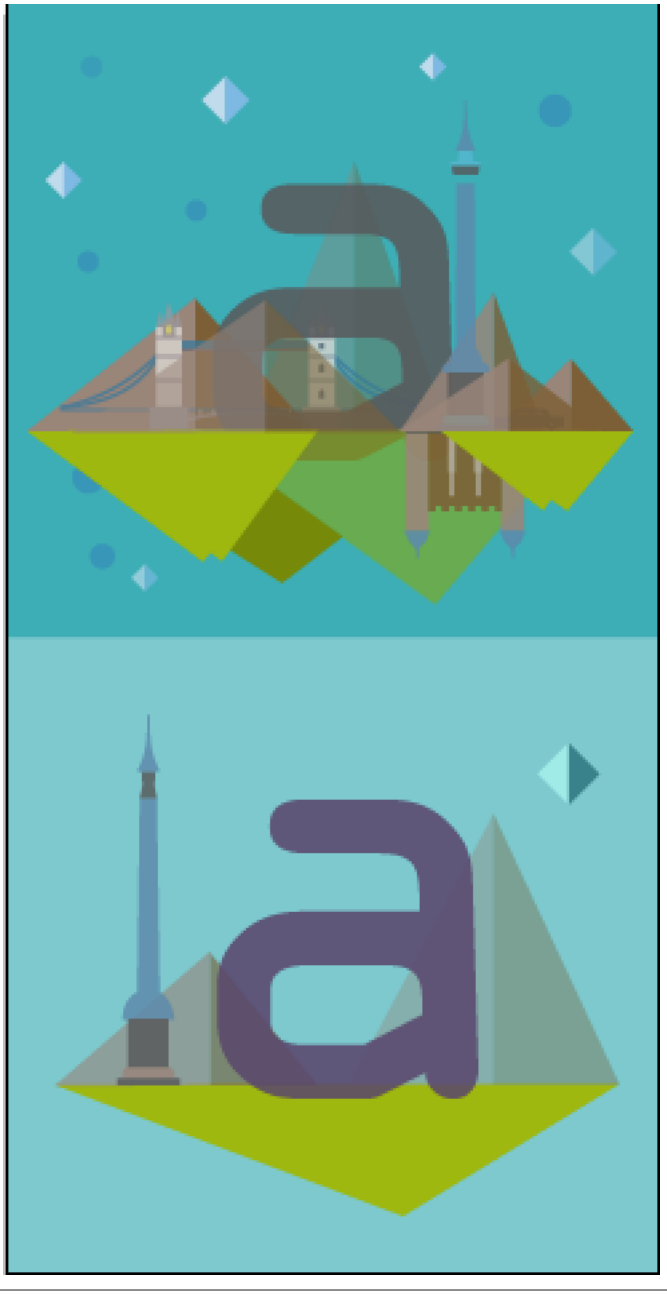


Figure 4-5. The sprite once we reduce repetition and get it ready for export.

Once we have this view, we run it through SVGOMG, using the web based-GUI to check that there's no distortion, checking the box to keep the names on the IDs on the groups and paths. We can later change the IDs to classes if we wish and clean up some of the cruft from the export. I do this by hand or with find and replace, but there are myriad ways to accomplish this, depending on the programmer.

The optimized SVG is placed inline in the HTML rather than a source URL background image like in the previous techniques. Now we can set areas to hide and show with a mobile-first implementation:

```
@media screen and ( min-width: 701px ) {
  .kells3, .kells2 {
    display: none;
  }
}
```

We also adjust the animation parameters slightly depending on the viewport in order to refine the movement for each version:

```
[class^="mountain"], [class^="grass"] {
  ...
  transform: skew(1.5deg);
}

@media screen and ( min-width: 500px ) {
  [class^="mountain"], [class^="grass"] {
    transform: skew(2deg);
  }
}
```

At this point the width and height are removed from the SVG and we can add in preserveAspectRatio="xMidYMid meet" (though that is the default, so it's not strictly necessary) to make the SVG fluid. With these alterations, it will adjust to the container size instead, which we set based on percentages. Flexbox or any other responsive container would work here too.

```
.initial {
  width: 50%;
  float: left;
  margin: 0 7% 0 0;
}
```

The Viewbox Trick

There is one catch- even if we assign the bottom layer a class and hide it, there will be an empty gap where the viewbox still accounts for that space. In order to account for that area, we can change the viewbox in the SVG to show only the top portion:

```
viewBox="0 0 490 474"
```

and that will work, but only for the two larger versions. The smallest version is now obscured, as the viewbox is providing somewhat of a window into another portion of the SVG spritesheet, so we will need to adjust it. This is akin to changing the background position in CSS to show different areas of a spritesheet.

But due to the fact that we're adjusting an SVG attribute, we will need JavaScript, as CSS doesn't yet have that capability. You can help vote or thumbs up on moving this into the CSS spec here:

https://github.com/w3c/fxtf-drafts/issues/7

```
var shape = document.getElementById("svg");

// media query event handler
if (matchMedia) {
        var mq = window.matchMedia("(min-width: 500px)");
        mq.addListener(WidthChange);
        WidthChange(mq);
}
// media query change
function WidthChange(mq) {
        if (mq.matches) {
    shape.setAttribute("viewBox", "0 0 490 474");
    shape.setAttribute("enable-background", "0 0 490 474");
        }
        else {
    shape.setAttribute("viewBox", "0 490 500 500");
    shape.setAttribute("enable-background", "0 490 500 500");
        }
};
```

At the time of publishing, Jake Archibald had just proposed moving these type of adjustments into a CSS spec: https://lists.w3.org/Archives/Public/www-style/2016Feb/0328.html. Hopefully it does gain adoption. If so, you will be able to update all of the viewbox changes in media queries and keep presentation implementation in one language.

Now when we scroll the browser window horizontally the viewport will shift to display only the part of the SVG we want to expose. Our code is now primed and ready to animate.

Responsive Animation

Due to the fact that when we export from a graphics editor we have a unique id for every different element, if we only want to change the ids to classes, we can still target them using a CSS attributeStartsWith selector:

```
[class^="mountain"], [class^="grass"] {
  animation: slant 9s ease-in-out infinite both;
```

```
    transform: skew(2deg);
  }
```

You'll see here that we begin with a transform set on that element; this keeps the keyframe animation nice and concise. The animation will assume that the 0% keyframe corresponds to the initial state of the element; to create a very efficient loop, we can define only the changes halfway through the animation sequence.

```
@keyframes slant {
  50% { transform: skew(-2deg); }
}
```

Some elements, such as the dots and stars, share a common animation, so we can reuse the declaration, adjusting the timing and delay as needed. The delay is negative offset because we want it to appear as though it's running from the start even though they are staggered. Animation keyframes will use the default positioning set on the element as 0% and 100% keyframes unless they are specified otherwise. We use this to our benefit to write the smallest amount of code possible:

```
@keyframes blink {
  50% { opacity: 0; }
}

[class^="star"] {
  animation: blink 2s ease-in-out infinite both;
}

[class^="dot"] {
  animation: blink 5s -3s ease-in-out infinite both;
}
```

We also need to add a viewport meta tag, which gives control over the page's width and scaling on different devices. The most common one will do: `<meta name="viewport" content="width=device-width, initial-scale=1">`

CHAPTER 5

Chapter 5: UI/UX Animations with CSS

In the previous chapters we've mostly covered standalone SVG animations. In this chapter, we'll go over more common use-cases of UI and UX elements that can be implemented with SVG and animated with CSS. In particular, we'll work through a common UX pattern of a transforming icon, which will allow you to see how something is built from start to finish, integrating the workflow into your own development process.

Context-Shifting in UX Patterns

Before we get into how to build typical UI/UX interactions into SVG Animations, let's go over the *why*. It's important to get the technique down, but it's just as important to use animation correctly.

Have you ever had a day at work where people kept interrupting you and putting you to different types of tasks? Work feels more frustrating when you can't get into a flow-based working style, and it makes you feel more disorganized and unproductive. It follows that using a website works the same way.

When users visit your site, they use a phenomenon called "saccade". Saccade is a rapid eye movement, used by your brain to create spatial relationships. Your never really "look" at a photo: your eye moves constantly to understand where things are placed in the picture, creating a mental map of the image.

When we create a website, we're creating a mental map for our users. Changes we make to site interactions can break that mental map. Modals are a good example: they often pop up out of nowhere, shattering the user's experience, and are an example of what I call "brute-force UX".

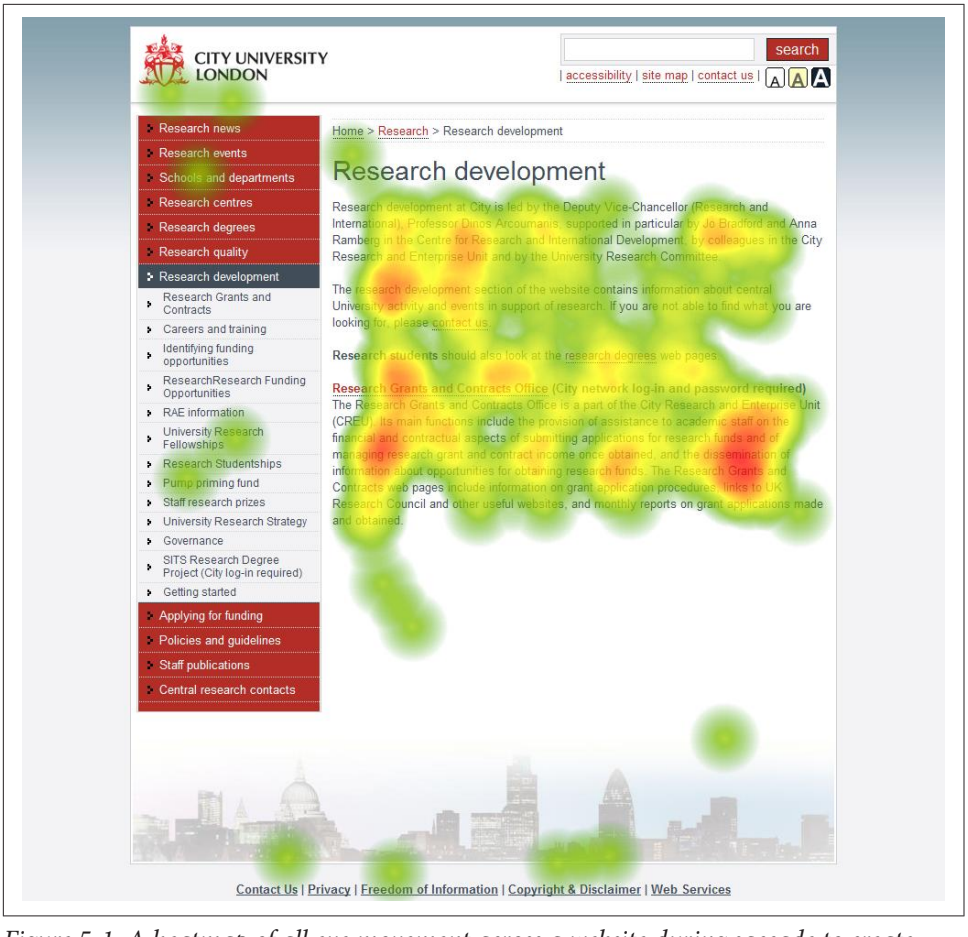


Figure 5-1. A heatmap of all eye movement across a website during saccade to create spacial awareness.

An animation that reduces friction in context-shifting succeeds by honoring the user's mental map: the user will retrieve and access things from consistent areas, the UX flows with their needs, and the whole experience feels more fluid. Creating animations that help guide your users take a bit of thinking, so let's break down some ways we can execute them.

- Morphing
- Revealing
- Isolation
- Style
- Anticipatory Cues
- Interaction

- Space Conservation

Before we dive into solutions, it's important to note that any one of these can be *overdone*. Our brains have evolved to take particular note of something in motion. This evolutionary trait is in place to keep us safe and alert; the part of your brain that kicks up adrenaline is also triggered when something unexpected moves on the screen. The web is a static, dull, box without animation: but when it comes to UX animation, subtlety is key.

Here's an example I built to show how an animation can retain context for a user: http://codepen.io/sdras/full/qOdwdB/, we'll refer back to this one a lot as we cover these premises.

Morphing

Morphing is probably the simplest way to help retain context for a user. Morphing means that the same element can become multiple pieces of information in different contexts, guiding the user's flow without changing anything very abruptly. Consider the animation above. There are many forms of morphing in the one interactive element used in the CodePen example. In the example, one frame morphs into the next: the pin expands to create the dialog, the contact button becomes the title, the text boxes shrink to become the loader dots, and so on, to give the whole experience a smooth user experience.

Figure 5-2. All of these states come from and return to one element.

Both SVG and CSS are good options for these kind of UI animations. I've found, from working with both, that each have their strength and weaknesses. CSS easily interpolates round to square and back again with`border-radius`. It can also handle large quantities of scale transforms gracefully, while SVG, beyond a few multitudes,

will appear pixelated before recovering. However, SVG is built for drawing. It is well suited for complex shapes.

You can tween path or even shape data with JavaScript and GreenSock's MorphSVG plugin. This is an unbeatable tool for this kind of technique, as unlike Snap.svg or even the poorly supported SMIL, MorphSVG allows you to easily transform between ***uneven amounts*** of path data, which allows for tons of wonderful effects. If you're interested in learning more about what you can do with SVG Morphing, please turn to Chapter 12 of this book, where we discuss it at length.

Revealing

Revealing is a very simple method of retaining context for the user, but revealing can be done in a way that breaks the user's context as well. Take your typical modal, for instance. This is an example of UX that comes when called, but it does the opposite of retaining context for the user: it suddenly shatters their focus and the spatial map they've created. As a user, I sometimes close modals with information I need because I find them so jarring.

Modals themselves are not the culprit here, though: it's the way we typically implement them. Here's an example of a modal that retains the context instead: it opens from it's origin, and replaces itself where it was. There's a transition between these two states, and as a user I know where that information "lives" and where to retrieve it again.

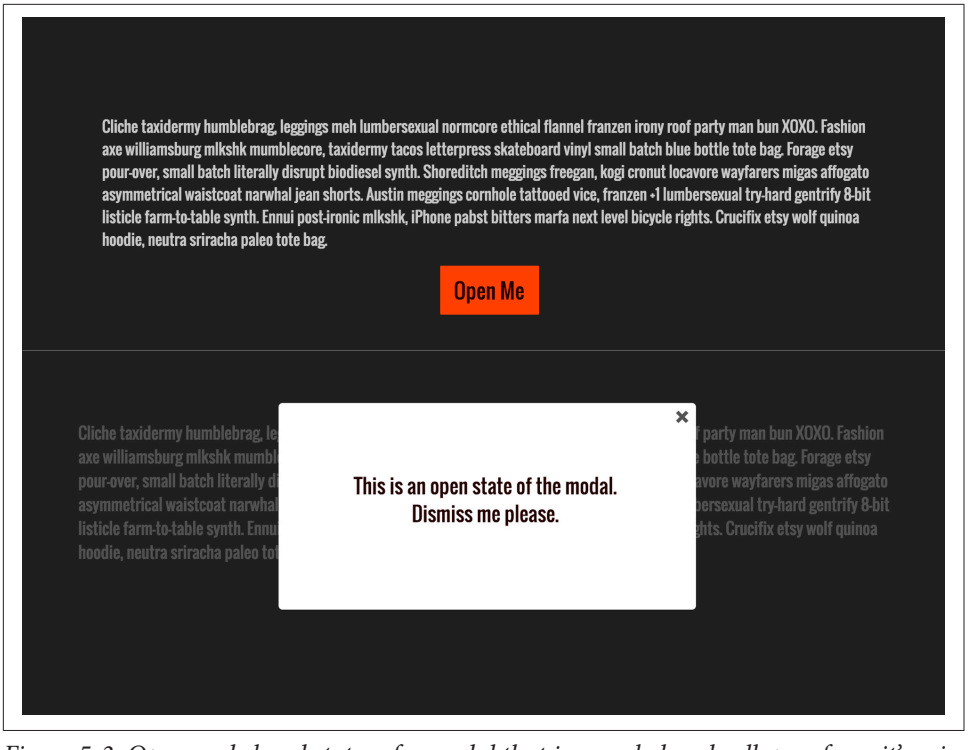


Figure 5-3. Open and closed states of a modal that is revealed and collapses from it's origin

http://codepen.io/sdras/full/yOjWdO/

You can see this in our original example as well. We reveal information from our location on the map, and see where it was put away. We don't need everything on the page at once, but we can see where we would get it should we need it.

Isolation

We've established that we're constantly scanning to create a spacial map with saccade, and thus isolation of pieces of our helps us wade through information faster. UIs can become cluttered: narrowing choices decreases the number of decisions, which helps users feel more empowered. Consider the following demo:

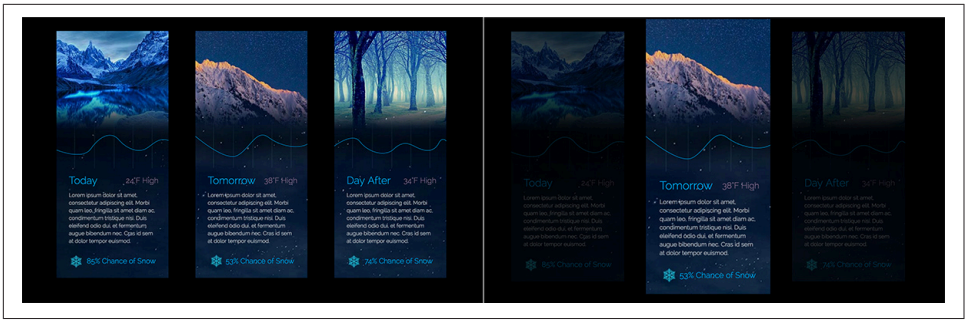


Figure 5-4. By isolating information and obscuring the rest on hover, the user is better able to scan and read the information provided.

http://codepen.io/sdras/full/qOdWEP

At first there's so much information on the page that it's hard to focus on one thing. But if we adjust the UI slightly (in this case adding a `:hover` state) we can clarify our user's attention.

Style

Style, design, branding and eases are all very closely tied. If you keep your animation style unified across your brand (and you should), this becomes your **Motion Design Language**. Motion Design Languages are important because it means that everyone is on the same page about what types of animations you're going to be creating. For this very reason, you can keep your code DRY (don't repeat yourself) by reusing eases in variables, interactions in functions, and keep consistent behavior across your site and even multiple platforms. I don't code Java for Android or Swift for iOS (yet), but I can retain consistency across these platforms and the web by nailing down a styleguide for animations that will apply to all of them.

How do eases come into play in this? Eases are a really strong piece of the branding of an animation. If you work for a stoic company like a bank or financial institution, your eases are more likely to be Sine or Circ; if you work for a more playful company like MailChimp or Wufoo, a Bounce or Elastic ease would be more suitable. (see accents in eases section below for a visual illustration of sine vs bounce eases)

Here are some sites that allow you to pick out the eases you could be using for your project:

- CSS: http://cubic-bezier.com/
- CSS: http://easings.net/
- GSAP: http://greensock.com/ease-visualizer

- React-Motion: http://chenglou.github.io/react-motion/demos/demo5-spring-parameters-chooser/

Accents in Eases

Eases can completely change the appearance and tone of an animation. Linear and Sine eases are expressed mathematically as more of a line, and will have an even transition between states, while something like a bounce or elastic ease will go back and forth between those states to create a sensation that jumps around and can potentially feel more playful.

You can use eases to draw attention to a particular action or event just as a designer uses accents in a palette. If you visit any major website, you'll note that there tends to be a primary color that's used everywhere, and an accent color that stands out from this color. The accent is used for things like Calls To Action (CTAs) so that users know where to click a button. Most of those CTAs are the real money-makers for the site, so their ability to stand out is of utmost importance.

We can apply the same technique to eases. In the example mentioned before, all of the eases were Sine eases, which are closer to a smooth, Linear ease. The only time we used a Bounce ease was confirmation that the form went through completely and was successful.

For more information on voice and tone in animation, check out Val Head's book on Designing Interface Animations:

http://rosenfeldmedia.com/books/designing-interface-animation/

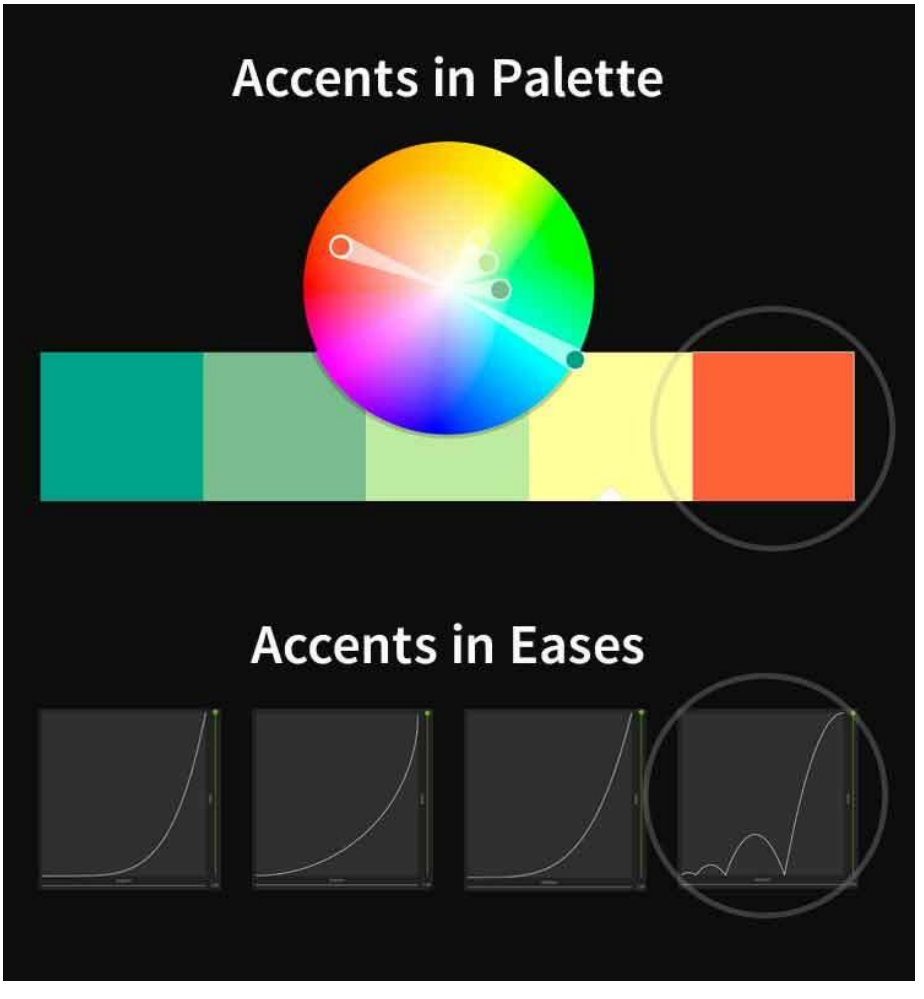


Figure 5-5. Just as we have accents in palette to draw attention, we can also have accents in eases.

Anticipatory Cues

Users providing information to a site experience a period of unrest: they're not sure what happened, who they gave their information to, or whether it worked. It often takes more than a second for their information to be processed, which makes anticipatory actions extremely important.

Other small examples of anticipation states are:

- A dropdown selection changing other contexts on the page
- A loading state
- A button being pushed
- A login being rejected
- A form being processed after submission
- Some data being saved

When these type of changes occur, it might not make sense to have a grand change, but still signify that the state of the page has changed or is in the process of transitioning- creating a context in and of itself. Considering the techniques we spoke about earlier, we might ask ourselves:

- Are we captivating the viewer during the transitional state, or is it simply a small means to arrive at the end state?
- Will this transitional state be reused for other instances? Does it need to be designed to be flexible enough for multiple placements and failure conditions?
- Does the movement need to express the activity? An example of this would be the user saving something that's not complete yet, in which case an anthropomorphization of "wait" would help communicate this.

Giving the user a loading state not only informs them that something is going on in the background, but if it's a custom loader, it can make the wait time *feel* less long and obtrusive, giving your site/app the **appearance of higher performance**.

Consider the following demo:

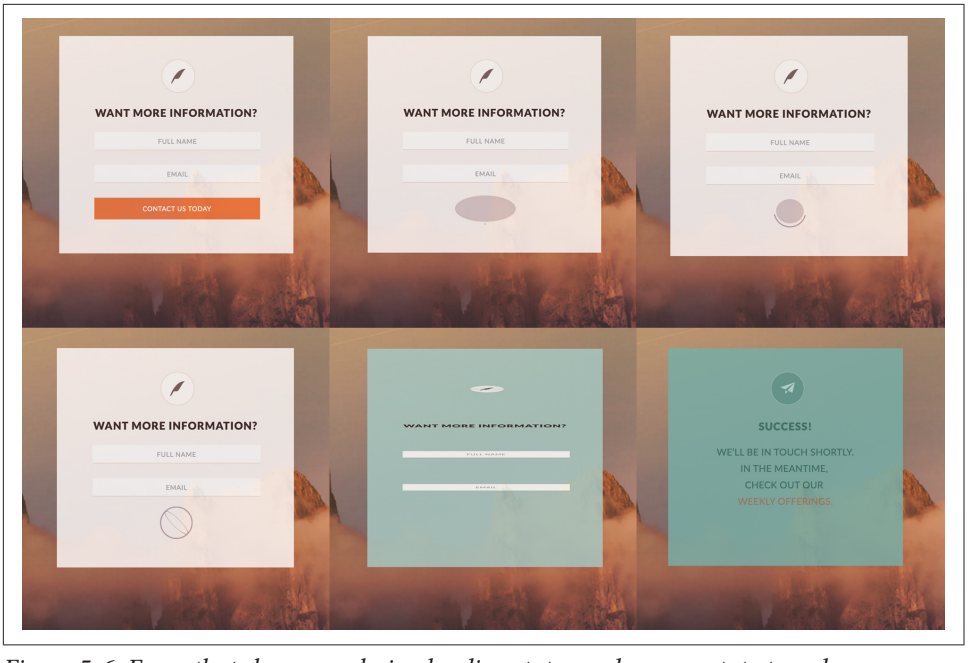


Figure 5-6. Form that shows morhping loading states and success state to reduce perceived wait time.

http://codepen.io/sdras/full/LEorev/

The wait transforms directly from the button, giving a smooth transition state. We get a bright green confirmation screen, but not before the loader animates. We were actually waiting for a second or two before the final confirmation, but it was almost unnoticeable.

Interaction

You learn more by *doing*. It's an old adage, but an accurate one. When a user engages with your UI, they are forming more meaningful structural awareness than they could by viewing it alone.

Rather than simply selecting an item and having it transition before the viewer's eyes, interconnectivity between UI states can be strongly reinforced when the user carries the action forward. Consider these very well done Drag and Drop Interactions by Mary Lou (Manoela Ilic) on Codrops.

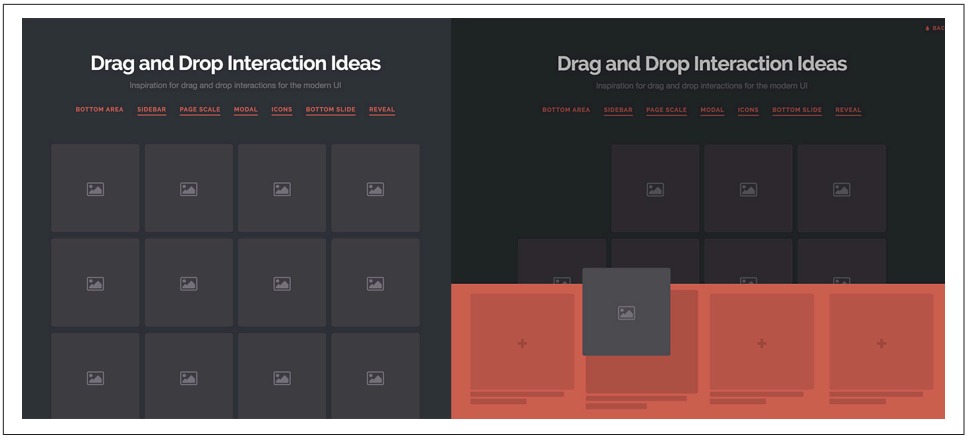


Figure 5-7. We move the element consistently to the same drawer which can be fetched from below.

As a user, you know where you put the item, and you know where to retrieve it later. It's at the bottom, right? There is no bottom- there is no drawer- it's just a div. But because we built the animation and interaction in a way that makes it seem like it occupies a space, and mimics a real-world interaction that they already know (a cabinet drawer) we've built a space that they feel they control easily.

Space Conservation

When we use animation to hide and display information that is not persistent on the page, we're able to offer the user more: more to access, more tools, and more controls, in a more limited amount of space. This becomes increasingly important as we build out responsive environments that need to collapse a lot of material without feeling cluttered. Consider the animation below:

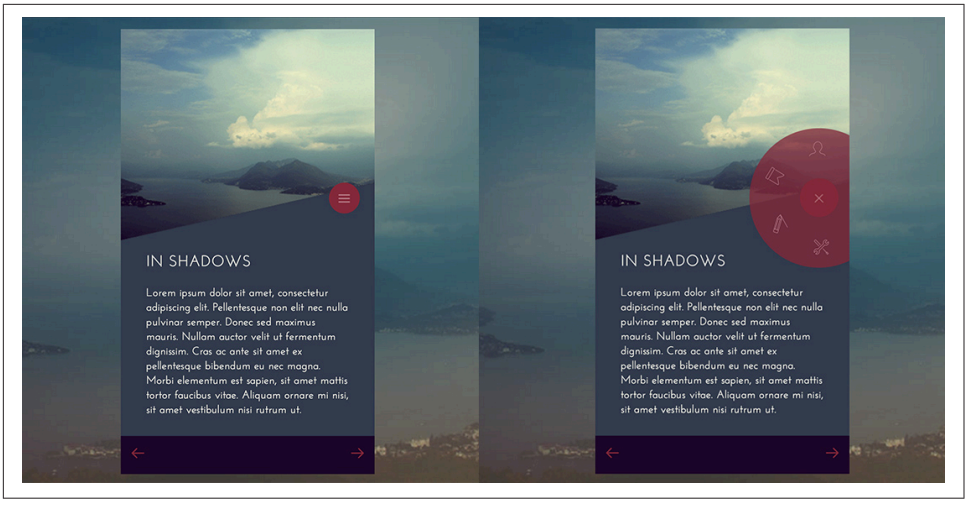


Figure 5-8. Using Sara Soueidan's tool, we can conserve space by hiding pieces that wait to animate in until they're called.

http://codepen.io/sdras/full/Kwjyzo/

We can honor the larger touch points that are needed for a mobile build while collapsing the navigation in a smaller space when it's not needed. This navigation was built with Sara Soueidan's Circulus tool, which build out a really nice circular SVG navigation that is easily animatable. https://sarasoueidan.com/tools/circulus/

Putting it All Together

These animation theories and concepts work best when combined. There's no right answer: the ability to be creative with SVG animation is part of it's strength. Understanding the core concepts means that we have all the base understanding that we need; the code follows naturally.

Icons that Transform

Now that we've discussed the "why" for animation in UI/UX patterns, let's go over the "how". For this example, we'll build out a pretty common use-case so you can see step-by-step how to create an interaction. This doesn't mean we'll always use the same approach, but if you follow along, you can see how we'd break down a simple interaction like this one to build it into our site using an SVG.

Icons are a nice way to add simple, useful, and informative animation to a site. Subtlety in this type of animation is key. If it's too verbose or flashy, sometimes it can distract rather than serving the user.

This type of interaction should never feel like it takes too long. A common practice is keeping the transition between 0 - 300ms. Anything longer than that and the user feels like the transition is less than instantanious.

It's also important to remember that any common UI or UX pattern that a user might see again and again should be subtle enough that it doesn't feel ardous on repeated viewing.

In our example, we're going to make a magnifying glass that morphs into a search field. We consider the beginning and intermediary states:

Figure 5-9. The beginning state of the magnifying glass

Figure 5-10. The end state, once the magnifying glass icon is clicked and the stem has become the input field, and the circle of the magnifying glass has become the dismiss area.

We're going to morph the stem into the line, and make the circle turn into the container for the x. Let's start with the magnifying glass.

In this example, we're going to reveal the input when the event fires. In the case of simple UI animations, we're moving a couple of small shapes from here to there, so simple storyboards are very helpful for planning them.

Focusing first on the stem being lengthened, let's consider the things that need to happen between states. The stem itself must get longer, it has to rotate slightly, and it has to transform into place.

Let's accommodate for the change in the size of the stem by lengthening the viewbox, so considering where we're starting with the svg:

```
<svg class="magnifier" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 32 34">
   <title>magnifier</title>
   <circle class="cls-1" cx="12.1" cy="12.1" r="11.6"/>
   <line class="cls-1" x1="20.5" y1="20" x2="33.1" y2="32.6"/>
 </svg>
```

We adjust the viewbox to `<svg class="magnifier" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 34">`

We'll also make sure the SVG is primed in CSS (SCSS) for the future transformation, and back.

```
.magnifier {
  line {
    transform: rotate(0deg) translateY(0px);
  }
  circle {
    transform: scale(1);
  }
}
```

We can change the state here in a few ways- in jQuery we would use a simple class operation, and in React we would address the state directly by calling getInitialState initially and then setting state with event handlers. Due to the fact that most people at the time of publishing are more familiar with jQuery, I've used it to demonstrate this, though we'll address React in future chapters. We'll use jQuery 3.0 (now backported to 1.X and 2.X as well) because it supports class operations on SVG.

If we think back to how the line element draws on to the SVG DOM, all we need to do to update the length of the line is alter value on the x2 attribute. In this case, we'll lengthen it from 33.1 to 300.

```
$( document ).ready(function() {
  $(".main").on("click", function() {
    var magLine = $(this).find(".magnifier line");

    if ($(this).hasClass("open")) {
      $(this).removeClass("open");
      magLine.attr("x2", 33.1);
    } else {
```

```
      $(this).addClass("open");
      magLine.attr("x2", 300);
    }

  });
});
```

At this point, the line is lengthened but it's drawn outside the viewbox because we haven't rotated and transformed it yet. Let's do that in CSS:

```
.open .magnifier {
  line {
    transform: rotate(-2.5deg) translateY(14px);
  }
  circle {
    transform: scale(0.5);
  }
}
```

CSS Transforms on SVG DOM Elements

As you experiment with CSS and SVG with transforms, you might notice that cross-browser stability begins to become hairy with complex movement, particularly when you're adjusting something with a transform-origin. This is a major reason I work with GreenSock. GreenSock not only makes your SVG animations stable, but also fixes some transform-origin stacking behaviors that are defined counter-intuitively in the spec.

We don't really need a full CSS animation with keyframes to interpolate, because it's just from point A to point B, so we're going to use a transition. We'll also use a couple of custom eases in SCSS, which we'll reuse as variables. One nice trick, and a possible gotcha is that ease-out functions are nice for entrances, while ease-in functions are great for exits. With that in mind, we're going to use a quad easing function:

```
$quad: cubic-bezier(0.25, 0.46, 0.45, 0.94);
$quad-out: cubic-bezier(0.55, 0.085, 0.68, 0.53);

.open .magnifier {
  line {
    transition: 0.65s all $quad;
    transform: rotate(-2.5deg) translateY(14px);
  }
  circle {
    transition: 0.35s all $quad;
    transform: scale(0.5);
  }
}
```

You'll notice we're using the entrance animations on the open state. This part maybe seem a little backwards: the .open animations will be treated as our entrance animation state while our exit animations should be added to the initial property. It seems a little counter-intuitive at first, but makes sense the more you work with it. The exit animations make more visual sense when they collapse together and we'll make them a little faster because it feels a little better when they fade more quickly.

```
.magnifier {
  line {
    transition: 0.25s all $quad-out;
    transform: rotate(0deg) translateY(0px);
  }
  circle {
    transition: 0.25s all $quad-out;
    transform: scale(1);
  }
}
```

Next let's work on the circle and the x-out. In this case, we've added the x-out as it's own SVG and positioned it appropriately, but we could have just as easily included it in the initial SVG. I didn't do so because when I was initially creating the animation, I wasn't sure where I would position it. Keeping it separate when creating our assets allowed for me to retain a little more flexibility in iterations. If your storyboards and designs are more formalized, it might serve for better cross-browser stability to have all elements contained within the same SVG DOM.

The other reason to separate the elements was due to transform-origin values. When the line is encapsulated within its own SVG, I could easily declare 50% 50%, and refer to the center of the x; if I was using a larger SVG structure, the values would be more difficult to define. In future chapters, I'll cover some GreenSock features that help a great deal with transform-origin values and designation, but for CSS, it's worth it to tread lightly, due to cross-browser bugs and only one option for declaration.

```
.x-out {
  width: 6px;
  padding: 5px 6px;
  transition: 0.5s all $quad;
  cursor: pointer;
  line {
    stroke-width: 2px;
    opacity: 0;
    transform: scale(0);
    transform-origin: 50% 50%;
  }
}

//firefox hack for padding on x, as we mentioned previously in
//the warning about cross-browser stability issues
@-moz-document url-prefix() {
  .x-out {
```

```
    padding: 5px !important;
  }
}

.open .x-out line {
  opacity: 1;
  transform: scale(1);
  transition: 0.75s all $quad;
}
```

Finally, we can see that we'll need to add an input for this to truly work. We'll make sure the SVG and the input are in the same height placement with some absolute positioning:

```
.magnifier, input, .x-out {
  margin-left: 30vw;
  margin-top: 40vh;
  pointer: cursor;
  position: absolute;
}

.magnifier, input {
  width: 400px;
}
```

Then we'll make sure that the input has no default native browser styling but that the font-size matches that of the size of the SVG:

```
input {
  font-size: 35px;
  padding-left: 30px;
  background: none;
  cursor: pointer;
  box-shadow: none;
  border: none;
  outline: none;
}
```

Now that the input is close to invisible, we want to allow the user to know that the area is focused and ready for user execution. We also want to clear the selection in the event that they close the search open state. We'll do so with JavaScript:

```
$( document ).ready(function() {
  $(".main").on("click", function() {
    var magLine = $(this).find(".magnifier line"),
        mainInput = $(this).find("input");

    if ($(this).hasClass("open")) {
      $(this).removeClass("open");
      magLine.attr("x2", 33.1);
      mainInput.val("");
    } else {
      $(this).addClass("open");
```

```
        magLine.attr("x2", 300);
        mainInput.focus();
      }

    });
  });
```

Figure 5-11. This is an image caption

http://codepen.io/sdras/full/BKaYyG

If you're completely morphing an entire path in SVG, please check out chapter_, because JavaScript, and GreenSock's MorphSVG in particular, is the best option for that kind of movement. But typical simple movements can be achieved without any additional libraries.

This is, of course, just one way of working with one UX pattern. Most patterns, you'll find, will follow suit with this type of decision-making and problem-solving in order to achieve some nice effects and moments.

There are some nice open source libraries that already do this type of interaction out of the box, such as Sara Soueidan's https://sarasoueidan.com/demos/navicon-transformicons/, or Dennis Gaebel's fork http://www.transformicons.com/. These might be worth checking out if you don't desire something custom.

Chapter 6: Animating Data Visualizations

Data Visualizations are an extremely useful way to present myriad information. Luckily, due to the relative popularity of some libraries such as d3 and chartist, small pieces of animation are not only possible, but easy to achieve if you've been following so far. These are by far not the only libraries that can create data visualizations, but there are so many to choose from that I picked my favorite of the many that I've worked with.

Chartist, at the time of publishing, uses the now-deprecated SMIL to animate, so I don't recommend that you use their native animations functions. D3 also offers some native animations, but you may find that now that you've learned some CSS implementations, it may be simpler and certainly more performant to draw the data visualization on the screen, and then animate it.

Chartist vs. D3 for Configuration

It's very simple to create responsive charts and graphs in Chartist, making it very beginner-friendly. The library creates a wrapper for the SVG, so some JavaScript functionality becomes a little obfuscated and less straightforward as one might think. For this reason, I strongly suggest using Chartist to draw up a simple graphs with simple CSS animations.

D3.js, on the other hand, is not quite as beginner friendly but very easy to work with and make extendable. The sky is the limit on what you can create in D3, which has made it the library of choice for many beautiful data visualizations across the web.

Simply put, there's no one right way, and you should work with what works for your workflow, site implementation, and even what you're the most excited about.

Teaching how to work with either Chartist or D3 to build charts and graphs is out of the scope of this book, but Chartist has wonderful live, interactive documentation, and there's another incredible O'Reilly book that's a great resource for learning D3, Interactive Data Visualization. http://chimera.labs.oreilly.com/books/1230000000345 I used this book to learn this technology and I can't recommend it highly enough.

Why Use Animation in Data Visualization?

Animation in data visualization can be extremely powerful as a performant piece of the data's structure. Here are a few ways that animation can aid a data visualization:

- Filtering
- Reordering
- Storytelling
- Change over time
- Staggering pieces for clarity

In the last chapter we discussed the importance of retaining context for users. Filtering data allows us to retain consistent *elements*, while shifting their *meaning* by rearranging them. Consider this data visualization:

http://www.nytimes.com/interactive/2012/02/13/us/politics/2013-budget-proposal-graphic.html

The New York Times presents the same data in many different contexts, allowing the user to process the information in an extremely powerful, multi-dimensional way. The user has greater insight into the information by seeing it in a variety of different contexts, while the area of the representation remain unchanged.

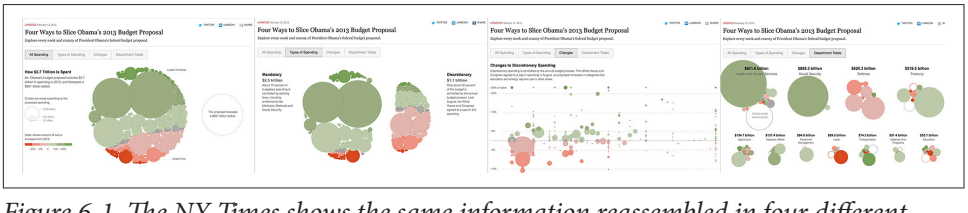


Figure 6-1. The NY Times shows the same information reassembled in four different ways to give the information new meaning, and adjusts each view with animation to retain context between states.

Even the most informative data lacks meaning if it is not engaging.[1] That's why storytelling is extremely important when it comes to data visualization.

1 See "Storytelling in data visualization" by Emma Whitehead and Tobias Sturt, from the Graphical Web, 2014 https://www.youtube.com/watch?v=cmmjTLCyqlY

I live in San Francisco where there is an ongoing housing crisis. Many families are being thrown out of their homes through a loophole in the law called the Ellis Act. The Ellis Act evictions are shown very powerfully in the data visualization below, by showing the change in evictions over time in a timeline. We'll learn how to make interactive timeline animations like this one in Chapter 11.

http://www.antievictionmappingproject.net/ellis.html

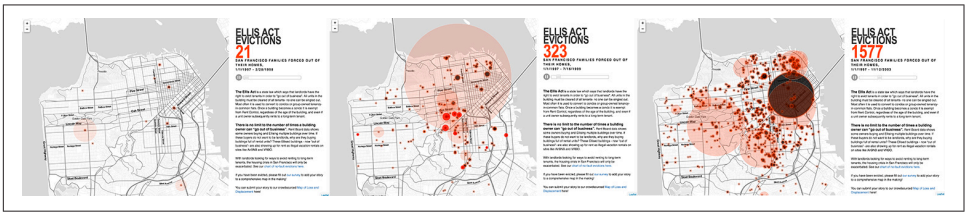


Figure 6-2. As the timeline moves forward, the heat spots come in faster and faster and fill up the city. The data visualization isn't just showing us data here, it's telling a story, and showing the impact.

If we look under the hood, we can see that that it's all SVG: the visualization is hiding and displaying the bursts of evictions depending on when they occur are in the timeline. This linear story is very effective, as more and more locations on the map "explode" in the animation.

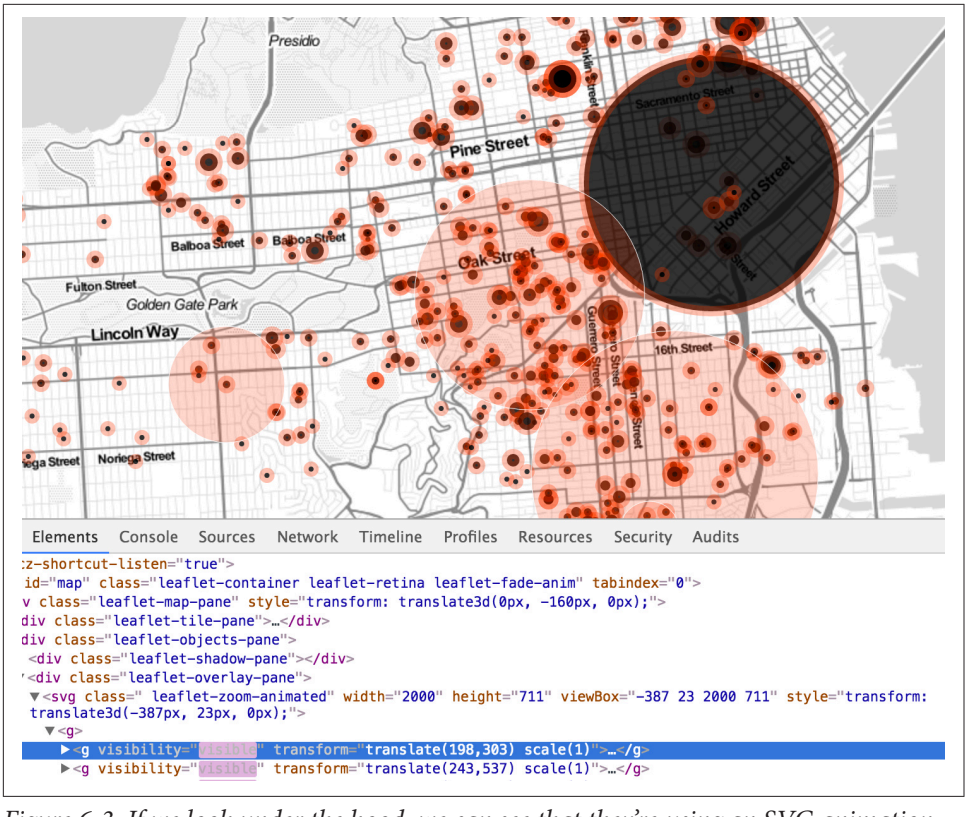


Figure 6-3. If we look under the hood, we can see that they're using an SVG animation, modifying and animating these groups for the sudden bursts on the screen.

Let's build out our own small version of something like this so you can see how it's done.

D3 with CSS Animation Example

As a very beginning starting point, D3 has a ton of nice blocks that you can work with and modify. Blocks are demos that often show the code and implementation details of a D3 example. Take care: blocks are not part of the library; they are examples individual contributors have posted, and licenses and versions may vary.

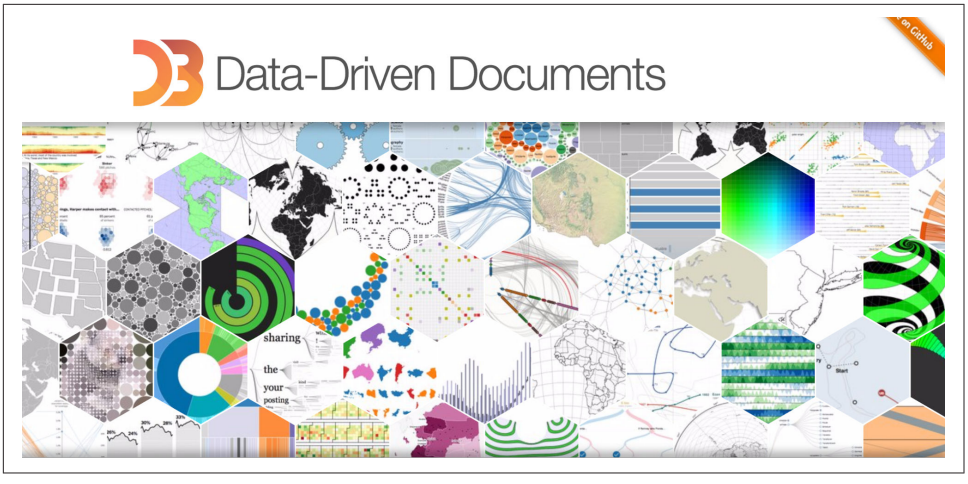


Figure 6-4. This is an image caption

You may find that beautiful as the readymade blocks may be, you still need to style them for your own site or animate them to bring them to life.

In our case, we chose a map of 3000 Walmart locations across the US:

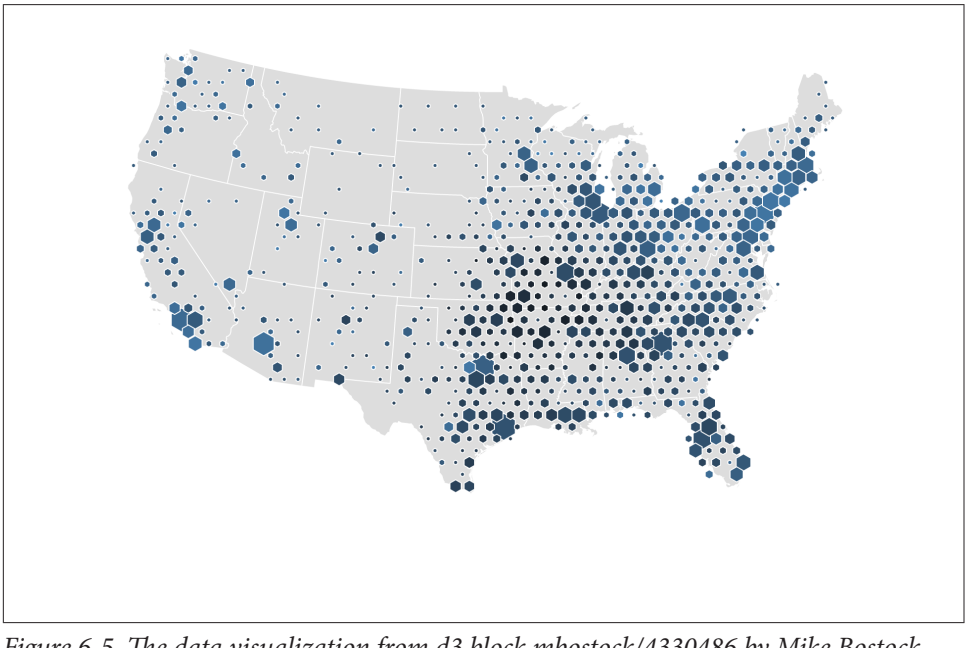


Figure 6-5. The data visualization from d3 block mbostock/4330486 by Mike Bostock that we will use to demo progressive animation.

With just a few styles and a few simple SCSS functions, we can convert this static document into something that presents itself progressively:

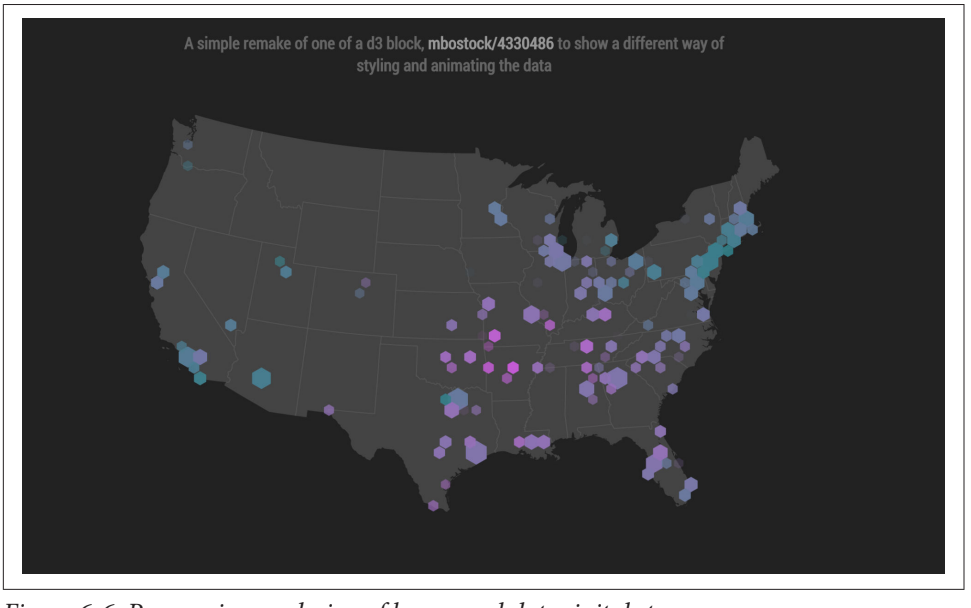


Figure 6-6. Progressive rendering of hexagonal data, inital stage

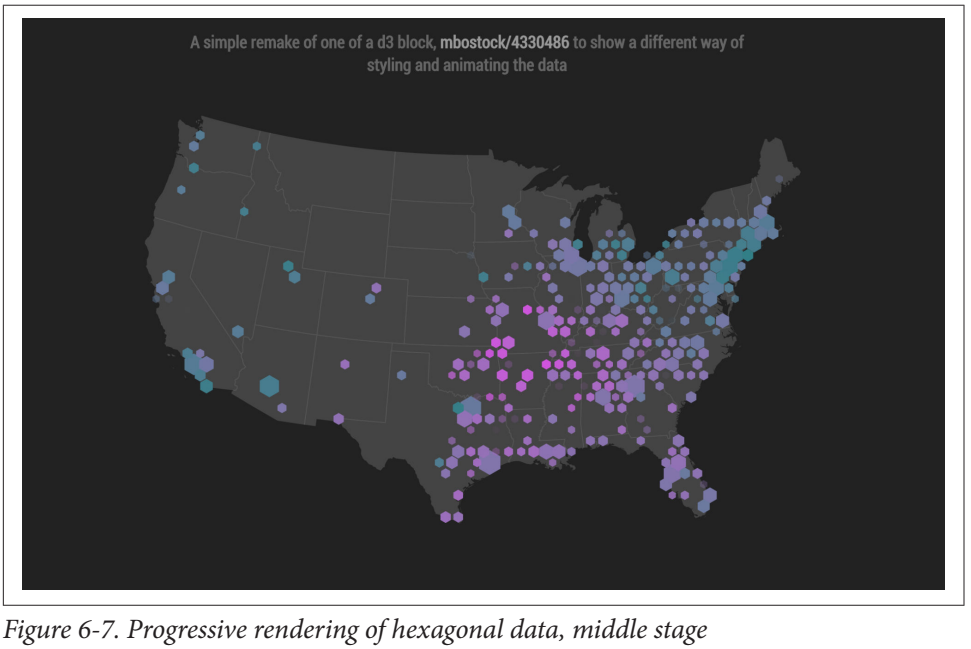


Figure 6-7. Progressive rendering of hexagonal data, middle stage

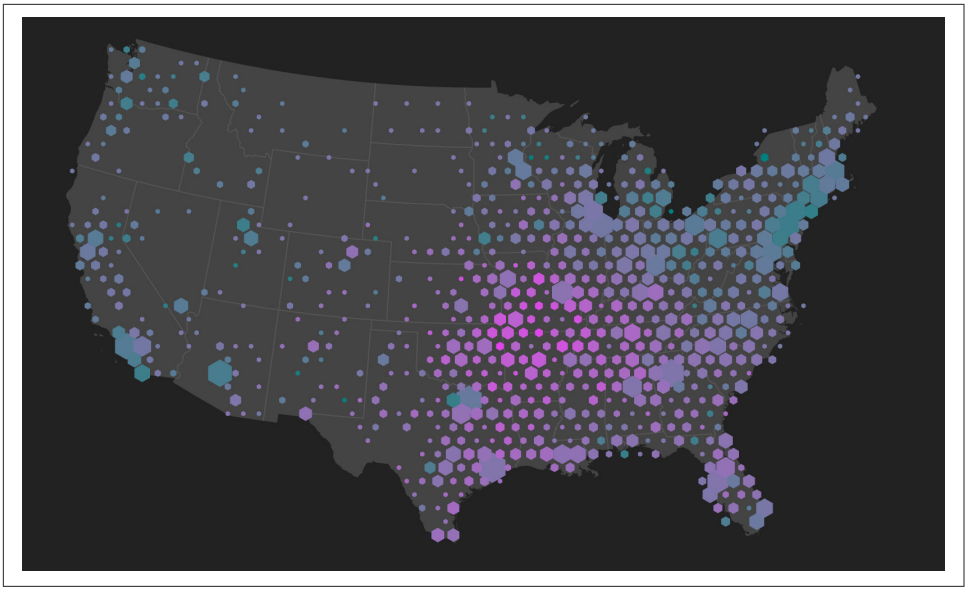


Figure 6-8. Progressive rendering of hexagonal data, rendering complete

http://codepen.io/sdras/full/qZBgaj/

In order to change the base styling, we will need classes to distinguish the different types of SVG paths. In this case, the D3 code already assigns the necessary classes using the .attr() function.

JavaScript:

```
svg.append("path")
      .datum(topojson.feature(us, us.objects.land))
      .attr("class", "land")
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);

  svg.append("g")
      .attr("class", "hexagons")
    .selectAll("path")
```

SCSS

```
svg {
  position: absolute;
  left: 50%;
  margin-left: -500px;
```

```
}

path {
  fill: none;
  stroke-linejoin: round;
}

.land {
  fill: #444;
}

.states {
  stroke: #555;
}
```

It doesn't make much sense to add an extra class on every hexagon path in order to animate them as we can use the `nth-child` selector. Sass also helps us create a stagger in our animations by allowing us to create a function. We set the hexagons to `opacity: 0;` initially in order to bring them in slowly:

```
.hexagons path {
  opacity: 0;
}

$elements: 2000;
@for $i from 0 to $elements {
  .hexagons path:nth-child(#{$i}) {
    $per: $i/50;
    animation: 2s #{$per}s ease hexagons both;
  }
}

@keyframes hexagons {
  100% {
    opacity: 1;
  }
}
```

The result is a pretty slim amount of code for a beautiful and exciting way to progressively show data. For a timeline showing progression, please refer to further chapter __, where we tie a GSAP timeline together with jQuery UI pieces to create interaction and progression.

Chartist With CSS Animation Example

Let's also make a simple Chartist example for comparison. Working from the point where we have a full line chart that's styled for our needs, we've decided it would be most interesting to have these lines animate in. This allows for the user to see the data unveil itself, and the staggering pieces are easier for our user to process.

In order create the illusion of an SVG drawing, we need to get the length of the SVG path, which we can do with .getTotalLength(). For a full list of SVG path interface operations, MDN has a great resource: https://developer.mozilla.org/en-US/docs/Web/API/SVGPathElement

```
setTimeout (
  function() {
    var path = document.querySelector('.ct-series-d path');
    var length = path.getTotalLength();
    console.log(length);
  },
3000);

//output
a: 1060.49267578125
b: 1665.3359375
c: 1644.7210693359375
d: 1540.881103515625
```

We're going to use that data to animate the path in. We can make it look like it's drawing itself with CSS.

First, let's set a `stroke-dasharray` on one of the paths

```
.ct-series-a {
  fill: none;
  stroke-dasharray: 20;
  stroke: $color1;
}
```

Figure 6-9. The path with a stroke-dasharray

That can be as long as we want it to be. We can also set a `stroke-dashoffset`, which can also be as long as we want it to be. As the name suggests, `stroke-dashoffset` offsets the stroke by any amount, and, thankfully, it's also an animatable property.

We can now use the console output and our little table to create an animation that takes the full length of the whole stroke and also offsets it by that much. This makes our data visualization look like it was drawn into the viewBox. We are also using the same information a few times, so we can use a mixin to dry it out. We also have different values for the dashoffset and dasharray, so to keep it dry, we animate to 0 instead of the other way around.

```
@mixin pathseries($length, $delay, $strokecolor) {
  stroke-dasharray: $length;
  stroke-dashoffset: $length;
  animation: draw 1s $delay ease both;
  fill: none;
  stroke: $strokecolor;
  opacity: 0.8;
}

.ct-series-a {
  @include pathseries(1093, 0s, $color1);
}

@keyframes draw {
  to {
    stroke-dashoffset: 0;
  }
}
```

http://codepen.io/sdras/full/oxNmRM

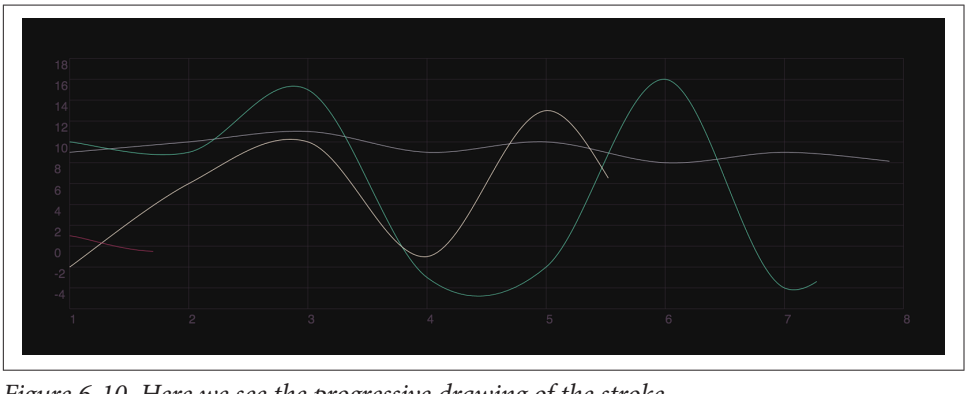


Figure 6-10. Here we see the progressive drawing of the stroke...

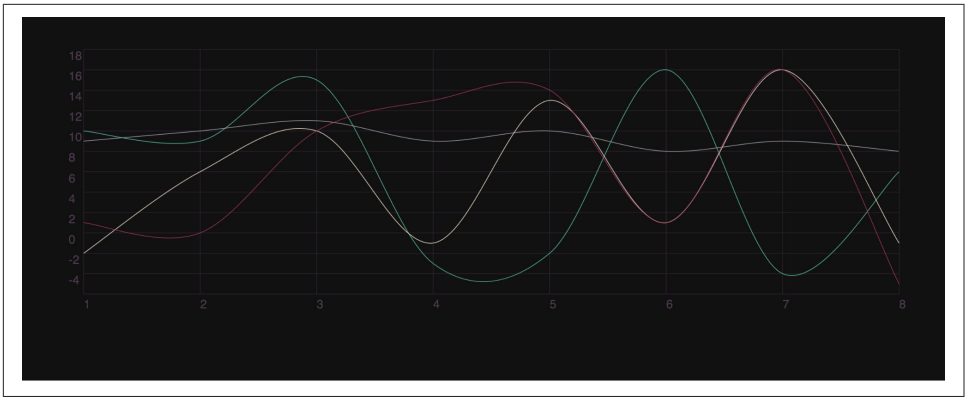


Figure 6-11. And here its conclusion.

This is just one way we can animate chartist visualizations, you can see many examples on their page here, https://gionkunz.github.io/chartist-js/examples.html- and the way you code entrances, exists, and persistent states in Chartist and CSS is entirely up to you- the sky's the limit.

We're going to get into even more exciting ways of working with data visualization in future chapters, but for that we'll need to learn how to work with JavaScript. Up next is a quick comparison of animation techniques, and then we'll switch languages.

Animating with D3

In this section we'll go over the simplest possible example of how to animate with d3 instead of CSS. We'll use version 4 for these examples (versions 3 and 4 have breaking changes between version, so using version 3 of the library will not work with this example.)

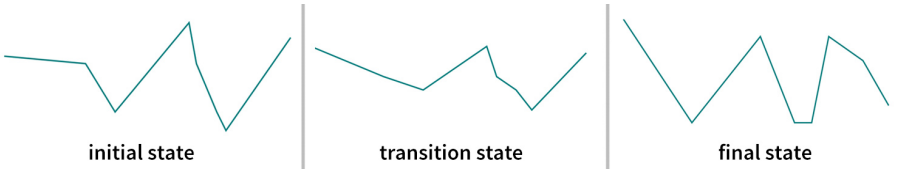
If you recall from the first chapter that a polyline is a series of points plotted on an x,y coordinate, you can also see why it might be so useful for a very simple data visualization. If you look at the code below, you can probably make sense of it, with your prior knowledge of SVG:

```
var line = d3.line();
var data1 = [[0, 0], [200, 300], [400, 50], [500, 300], [550, 300], [600, 50], [700, 120], [775, 2
var data2 = [[0, 100], [220, 120], [300, 250], [500, 10], [520, 120], [575, 250], [600, 300], [775

d3.select('#path1')
  .attr('d', line(data1))
  .transition()
  .attr('d', line(data2))
  .delay(1000)
  .duration(3000)
  .ease(d3.easeBounce);
```

We set the line to d3.line() method, we set the attribute of that line to the x and y coordinates of two fields of data. We then call transition between the two states of the line attributes, and optionally we can declare delays, durations, and eases.

The code above will transition this line from one state to another:



You can apply this same method of animating with other things as well- colors, coordinates, you name it. D3 is very flexible this way.

Animating Different Path Point Amounts

Even though d3 is flexible in that it allows for most things that SVGs are capable of, so there is almost no end to what you can make, SVGs are pretty finicky about animating between different path values, so D3 inherits that quirk. If our second dataset had a different length than the first, we'd find the transition effect to be pretty unweildy, ugly, or just fail entirely. That's why GreenSock's MorphSVG is extremely handy, and would work for this as well (see Chapter 12 for more details.)

There is also a library built outside of d3 that allows for graceful path animations, called d3-interpolate-path. The repo is available here:

https://github.com/pbeshai/d3-interpolate-path and there's a nice blog post about it as well:

https://bocoup.com/weblog/improving-d3-path-animation

Staggers are pretty easy in d3, and shares similarities with CSS- in that you calculate a new delay for each element. If you're familiar with for loops in JavaScript, this implementation will likely look familiar to you:

```
transition.delay(function(d, i) { return i * 10; });
```

If we were to use this in a color interpolation, it would look something like this: (we'll update the last code sample to a scatterplot so that the colors are easier to see)

```
var dataset = [ 5, 17, 15, 13, 25, 30, 15, 17, 35, 10, 25, 15],
    w = 300,
    h = 300;

//create svg
```

```
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

//create shapes in svg with data
svg.selectAll("circle")
   .data(dataset)
   .enter()
   .append("circle")
   .attr("class", "circles")
   .attr("cx", function(d, i) {
      return 10 + (i * 22)
    })
   .attr("cy", function(d) {
     return 200 - (d * 5)
   })
   .attr("r", function(d) {
     return (d / 2)
   })
  .transition()
    .style("fill", "teal")
    .duration(1500)
    .delay(function(d, i) { return i * 100; });
```

Codepen example available here:

http://bit.ly/2fpuPe3

Chaining and Repeating

For more complex effects, we could also add multiple transitions and even create loops. To chain animations, we would add a .transition() method between two states as we did before, but to make the whole thing repeat, we would have update our syntax a little and use some recursion. Here's an example of both:

```
.transition()
    .on("start", function repeat() {
        d3.active(this)
            .style("fill", "purple")
          .transition()
            .style("fill", "blue")
          .duration(2000)
          .transition()
          .duration(2000)
            .on("start", repeat);
      });
```

Codepen here:

http://bit.ly/2goB8mh

Please keep in mind that if you'd like to create very complex chaining or interaction, you might consider switching to GreenSock for animation. We'll cover GreenSock in some later chapters and you'll find that it plays nicely with D3's output while giving you a lot more finite control of timelines and sequencing.