Web Animation using JavaScript

Julian Shapiro

Foreword by David DeSandro, Founder of Metafizzy; Author/Developer of Masonry and Isotope
I dedicate this book to people who play Counter-Strike.
And to people who like the show Rick and Morty.
I would like to thank Yehonatan Daniv for providing support to Velocity’s users on GitHub, Anand Sharma for regularly inspiring me with his motion design work, and David DeSandro for writing this book’s foreword. I’d also like to thank Mat Vogels, Harrison Shoff, Adam Singer, David Caplan, and Murat Ayfer for reviewing drafts of this book.
This page intentionally left blank
# CONTENTS

- Foreword ............................................................ xii
- Introduction ....................................................... xiv

## CHAPTER 1  ADVANTAGES OF JAVASCRIPT ANIMATION ...................... 2
- JavaScript vs. CSS animation ..................................... 4
- Great performance .................................................. 6
- Features ..................................................................... 7
  - Page scrolling ..................................................... 7
  - Animation reversal ................................................ 7
  - Physics-based motion ............................................ 8
- Maintainable workflows ............................................ 9
- Wrapping up ........................................................ 10

## CHAPTER 2  ANIMATING WITH VELOCITY.JS .................................. 12
- Types of JavaScript animation libraries ......................... 14
- Installing jQuery and Velocity ..................................... 15
- Using Velocity: Basics ................................................ 16
  - Velocity and jQuery ............................................. 16
- Arguments .................................................................. 16
- Properties .................................................................. 18
- Values ....................................................................... 19
- Chaining .................................................................... 20
- Using Velocity: Options ............................................. 21
  - Duration .................................................................... 21
  - Easing ..................................................................... 21
  - Begin and Complete ............................................. 24
  - Loop ....................................................................... 25
  - Delay ........................................................................ 26
  - Display and Visibility ............................................. 27
- Using Velocity: Additional features ................................. 30
  - Reverse Command .................................................. 30
  - Scrolling ............................................................. 30
  - Colors ..................................................................... 31
Code technique: Organize sequenced animations ............ 65
Standard approach ........................................... 65
Optimized approach .......................................... 66
Code technique: Package your effects ....................... 69
Standard approach ........................................... 69
Optimized approach .......................................... 70
Design techniques ............................................ 73
Timing multipliers ............................................. 73
Use Velocity Motion Designer ............................... 74
Wrapping up .................................................... 77

CHAPTER 5 ANIMATING TEXT ................................. 78
The standard approach to text animation .................. 80
Preparing text elements for animation with Blast.js .... 82
How Blast.js works .......................................... 83
Installation ....................................................... 84
Option: Delimiter ............................................. 85
Option: customClass ......................................... 85
Option: generateValueClass ................................ 86
Option: Tag ..................................................... 87
Command: Reverse ............................................ 88
Transitioning text into or out of view ..................... 90
Replacing existing text ...................................... 90
Staggering ....................................................... 91
Transitioning text out of view ............................. 91
Transitioning individual text parts ......................... 94
Transitioning text fancifully ............................... 96
Textual flourishes ............................................ 97
Wrapping up .................................................... 100

CHAPTER 6 SCALABLE VECTOR GRAPHICS PRIMER .... 102
Creating images through code .............................. 104
SVG markup ................................................... 105
SVG styling ..................................................... 107
Support for SVG .............................................. 108
SVG animation ................................................ 109
It’s a special time when a developer first discovers jQuery’s `animate()`. I remember trying to animate any part of the page that wasn’t bolted to the main content. I created accordions, fly-out menus, hover effects, scroll transitions, magical reveals, and parallax sliders. Turning my websites from cold, static documents into moving, visual experiences felt like I was reaching another level as a web designer. But it was just bells and whistles. I realize now that for all the animation I added, I hadn’t actually improved the user experience of my websites.

All the same, it was thrilling. So what makes animation so exciting?

My apartment looks over downtown Brooklyn. I see people walk down the street. Plumes from smokestacks billow up. Pigeons flutter to perch on a ledge. A construction crane raises a section of a building. A single, heart-shaped balloon floats up into the Brooklyn sky (corny, I know, but I literally saw this happen twice). Cars drive over the Williamsburg Bridge. Clouds pass overhead.

The world is in motion.

This is how you expect the universe to work. Things move. Like the movements outside my window, each one is a one-sentence story. Together they tell the larger story of what is happening.

Yet this isn’t how digital interfaces work. Those little stories are missing. When things change, you have to fill in the story for yourself. When you press the Next button at an ATM, the screen suddenly changes. Did it move forward successfully? Was there an error? You have to read the screen again to interpret the results of your action. Utilizing motion removes this leap of understanding between interactions. Motion inherently communicates what has changed. It’s like writing tiny stories between states.

When a slide transition takes you to the next screen, animation helps you better understand what just happened. Wielding this power is what makes animation so thrilling. Like layout, color, and typography, animation helps you shape and direct the user experience. Animation is more than just making things move. It’s designing more effectively, and doing it thoughtfully.

Unfortunately, in the history of web animation, thoughtfulness hasn’t always been the highest priority. As developers, we’ve used Flash, animated GIFs, Java applets, marquee tags, and, more recently, CSS, JavaScript, and SVG to create animation that’s been, at best, a level of polish or, at worst, a gimmick. The idea of creating animation that’s both high-performance and user-friendly is relatively new.
So it’s a good thing you have this book in front of you. Julian Shapiro is one of the principal experts on animation on the web. In creating and supporting Velocity.js, he has developed an intimate knowledge of all the quirks and advantages of using motion on websites. *Web Animation using JavaScript* will give you not only the technical know-how required to implement animation in your websites, but, more importantly, the insights you’ll need to use animation effectively and craft compelling user experiences.

Animation libraries and technologies have made motion design more accessible than ever. But not every developer abides by best practices. The past couple of years have seen several trendy anti-patterns come and go. Scroll behavior has been hijacked. Mobile navigation has been pushed into menus accessible only via gestures. While adding animation is within the grasp of anyone who stumbles across `.animate()`, utilizing it to improve the user experience is one of the hallmarks of a dedicated developer. This book will help you become one of them.

David DeSandro
February 2015
Brooklyn, New York

*David DeSandro is the founder of Metafizzy
and author/developer of Masonry and Isotope.*
INTRODUCTION

In the early days of the web, animation was primarily used by novice developers as a last-ditch effort to call attention to important parts of a page. And even if they wanted animation to transcend its niche, it couldn’t: browsers (and computers) were simply too slow to deliver smooth web-based animation.

We’ve come a long way since the days of flashing banner ads, scrolling news tickers, and Flash intro videos. Today, the stunning motion design of iOS and Android dramatically improves the user experience—instead of deterring from it. Developers of the best sites and apps leverage animation to improve the feel and intuitiveness of their user interfaces. Animation’s rise to relevancy isn’t just a by-product of improved processing power; it reflects a better appreciation for best practices within the web development community. The tools you use to make a website are now considered less important than the quality of the resulting user experience. As obvious as this seems, it wasn’t always the case.

So, what makes animation in particular so useful? Whether it’s transitioning between chunks of content, designing intricate loading sequences, or alerting the user what to do next, animation complements text and layout to reinforce your site’s intended behavior, personality, and visual sophistication. Does your content bounce into view in a friendly way, or does it whip across the screen? This is the domain of motion design, and the decisions you make will establish the transcendent feeling of your app.

When users recommend your app to others, they’ll often try to describe it with words like “sleek” or “polished.” What they don’t realize is that they’re mostly referring to the motion design work that’s gone into the interface. This inability of the layman to make the distinction is precisely what great user interface (UI) designers strive for: animations that reinforce the interface’s objectives but don’t otherwise divert the user’s attention.

This book provides you with the foundation necessary to implement animation confidently and in a way that’s both technically maintainable and visually impactful. Throughout, it considers the balance between enriching a page with motion design and avoiding unnecessary flourishes.
Why is all of this so important? Why is it worth your time to perfect your transitions and easing combinations? For the same reason that designers spend hours perfecting their font and color combinations: refined products simply feel superior. They leave users whispering to themselves, “Wow, this is cool,” right before they turn to a friend and exclaim, “You gotta see this!”

**NOTE:** If you’re unfamiliar with basic CSS properties, you should pick up an introductory HTML and CSS book before reading this one.
The animation code found on most sites is nothing short of a mess. If there’s one thing experienced motion designers miss about the old, ugly days of Flash, it’s a structured approach to motion design.

The contemporary approach to structuring animation code is twofold: leverage the workflow features of an animation engine (in this case, Velocity.js) to make your code more terse and expressive, and use code organization best practices so that it’s easy to modify your work later.

Say goodbye to deep-nesting JavaScript callbacks and to dirtying your stylesheets with unwieldy CSS animations. It’s time to up your web animation game.
CSS ANIMATION WORKFLOW

In an attempt to better manage UI animation workflow, developers sometimes switch from JavaScript to CSS. Unfortunately, once animations reach a medium level of complexity, CSS animations typically result in a significantly worse workflow.

ISSUES WITH CSS

While CSS transitions are convenient when used sparingly in a stylesheet, they’re unmanageable in complex animations sequences (for example, when all elements sequentially load into view upon page load).

CSS tries to address this issue with a keyframes feature, which lets you separate animation logic into discrete time ranges:

```css
@keyframes myAnimation {
  0% { opacity: 0; transform: scale(0, 0); }
  25% { opacity: 1; transform: scale(1, 1); }
  50% { transform: translate(100px, 0); }
  100% { transform: translate(100px, 100px); }
}
#box { animation: myAnimation 2.75s; }
```

This specifies separate points within an animation’s timeline at which particular property values should be reached. It then assigns the animation to an element with an ID of #box, and specifies the duration of the keyframe sequence to complete within. Don’t worry if you don’t fully grasp the syntax above—you won’t be using it in this book. But before moving on, consider this: what happens when a client asks you to make the opacity animation one second longer, but keep the rest of the properties animating at their current durations? Fulfilling this request requires redoing the math so the percentage values properly align with a 1-second increase. Doing this isn’t trivial, and it certainly isn’t manageable at scale.
WHEN CSS MAKES SENSE

It’s important to point out a situation in which you should be using CSS rather than JavaScript for UI animation: when you’re animating simple style changes triggered by a user hovering over an element. CSS transitions lend themselves beautifully to these types of micro-interactions, allowing you to accomplish the task in just a few lines of very maintainable code.

Working in CSS, you first define a transition on the target element so that changes in the specified CSS properties animate over a predetermined duration:

```css
/* When this div's color property is modified, animate its change over a duration of 200ms */
div {
    transition: color 200ms;
}
```

You then specify the value that each particular CSS property should change toward, per the transition rule. In the case of the hover example, the div’s text color will change to blue when the user hovers over it:

```css
div:hover {
    color: blue;
}
```

That’s it. In only a few lines of code, CSS handles interaction state for you: when the user hovers away from the div, CSS will animate the change from blue back to the preexisting text color over a duration of 200ms.

WHAT DOES GOOD CODE LOOK LIKE?

Good code is expressive, meaning that its purpose is easy to grasp. This is crucial not only for coworkers attempting to integrate your foreign code, but also for yourself in the future, once you’ve forgotten your original approach. Good code is also terse, meaning that it accomplishes what it needs to in as few lines as possible; every line serves an important purpose, and it can’t be rewritten away. Lastly, good code is also maintainable, meaning that its individual parts can be updated without fear of compromising the integrity of the whole.
In contrast, coding this same effect in jQuery would entail the following:

```javascript
$div

// Register a mouseover event on this div, which calls an animation
.on("mouseover", function(){
    $(this).animate({ color: "blue" }, 200);
});

// When the user hovers off the element, animate the text color back
to black
.on("mouseout", function(){
    // Note: We have to remember what the original property value
    // was (black)
    $(this).animate({ color: "black" }, 200);
});
```

This might not look so bad, but the code isn’t taking advantage of the fact that JavaScript provides an infinite amount of logical control. It goes out of its way to do something that CSS is designed for: triggering logicless animations that occur on the same element that the user is interacting with. Above, you’re doing in JavaScript what you could have done in fewer, more expressive, and more maintainable lines of CSS. Even worse, you’re not getting any additional feature benefits by implementing this functionality in JavaScript.

In short, if you can easily use CSS transitions to animate an element that’s never being animated by JavaScript (meaning there’s no potential for conflict), then you should code that animation in CSS. For all other UI animation tasks—multi-element and multistep sequences, interactive drag animations, and much more—JavaScript animation is the superior solution.

Let’s explore the fantastic workflow techniques JavaScript provides.
The first technique has profound workflow benefits, especially for teams.

**STANDARD APPROACH**

In jQuery animation, it’s common to animate CSS classes onto elements using the UI add-on plugin (jQueryUI.com). When the module is loaded, jQuery’s `addClass()` and `removeClass()` functions are upgraded with animation support. For example, let’s say you have a CSS class defined in a stylesheet as follows:

```css
.fadeInAndMove {
  opacity: 1;
  top: 50px;
}
```

You can then animate the CSS properties of that class (opacity and top in this case) onto the target element along with a specified duration:

```javascript
// Animate the properties of the .fadeInAndMove class over a 1000ms duration
$element.addClass("fadeInAndMove", 1000);
```

The more common implementation of jQuery animation consists of inlining the desired animation properties within an `$animate()` call, which uses the syntax demonstrated in Chapter 1, “Advantages of JavaScript Animation”:

```javascript
$element.animate({ opacity: 1, top: 50 }, 1000);
```

Both implementations produce the same result. The difference is their *separation of logic*: The first implementation delegates the styling rules to a CSS stylesheet, where the rest of the page’s styling rules reside. The second mixes styling rules with the JavaScript logic responsible for triggering them.

The first approach is preferable due to the organizational cleanliness and flexibility gained by knowing where to look to make the appropriate style or logic changes to your code. CSS stylesheets exist for a reason; seasoned developers do not inline CSS into their HTML. That would conflate the purposes of HTML (structure) and CSS (styling), and make a site considerably more difficult to maintain.
The value of logic separation is further pronounced when working in a team environment, in which it’s common for developers and designers to bump heads while trying to edit the same file at the same time.

**OPTIMIZED APPROACH**

With the review of standard methods out of the way, let’s look at the optimized approach. It’s just as beneficial—and often the best methodology for JavaScript-centric animation workflows—to shift animation styling logic into a dedicated JavaScript file (for example, a `style.js`) rather than a dedicated CSS stylesheet. Sounds weird, right? Perhaps, but it works **brilliantly**. This technique leverages plain old JavaScript objects to help you organize your animation code.

For example, your `style.js` file might look like this:

```javascript
// This object is a parallel to the CSS class defined in the previous code example
var fadeIn = {
    opacity: 1,
    top: "50px"
};
```

In your `script.js`, which is the primary JavaScript file that controls animation logic, you would then have:

```javascript
// Pass our named properties object into Velocity
$element.velocity(fadeIn, 1000);
```

To recap, in your `style.js`, you’ve defined a JavaScript object that’s populated with the CSS properties you want to animate. This is the same object that’s then passed into Velocity as a first argument. You’re not doing anything fancy here—just saving objects to named variables, then passing those variables into Velocity instead of the raw objects themselves.

**NOTE:** This technique works equally well with jQuery’s `animate()` function.
a pain-free workflow is vital.
The benefit of switching from CSS to JavaScript to segregate logic is that your style.js file is uniquely capable of defining animation options—not just animation properties. There are many ways to specify an option: one is to assign two member properties to a parent animation object to which you assign an expressive name. The first property on the object defines the animation’s properties; the second defines its options.

In this case, your style.js file would look like this:

```javascript
var fadeIn = {
    // p is for "properties"
    p: {
        opacity: 1,
        top: "50px"
    },
    // o is for "options"
    o: {
        duration: 1000,
        easing: "linear"
    }
};
```

In the script.js file, you’d have:

```javascript
// Pass in our clean and re-usable animation objects
$element.velocity(fadeIn.p, fadeIn.o);
```

Pretty and clean, right? Someone skimming it would understand its purpose, and would know where to look to modify its properties—the style.js file. Further, the purpose of this animation is immediately evident: because you’ve named the animation object appropriately, you know that the code serves to fade an object into view. You no longer have to mentally parse animation properties to assess the purpose of the animation.

This approach discourages you from arbitrarily setting options for each individual animation on a page since there’s now a bank of premade animation objects you can easily pull from. This results in leaner code and more consistent motion design. Consistency, as you learned in the previous chapter, is a key component of great UX.
But the best part is that this approach lends itself perfectly to organizing your animation variations together. For example, if you typically fade button elements into view with a duration of 1000ms, but you fade modal windows into view with a duration of 3000ms, you can simply split your options object into two appropriately named variations:

```javascript
var fadeIn = {
    p: {
        opacity: 1,
        top: "50px"
    },
    // Options object variation #1 uses a fast duration
    oFast: {
        duration: 1000,
        easing: "linear"
    },
    // Variation #2 uses a slower duration
    oSlow: {
        duration: 3000,
        easing: "linear"
    }
};
// Animate using the fast duration.
$button.velocity(fadeIn.p, fadeIn.oFast);
/* Animate using the slow duration. */
$modal.velocity(fadeIn.p, fadeIn.oSlow);
```
Alternatively, you could nest “fast” and “slow” objects as children of a singular options object. The choice of which implementation to use is based on your personal preference:

```javascript
var fadeIn = {
  p: {
    opacity: 1,
    top: "50px"
  },
  o: {
    fast: {
      duration: 1000,
      easing: "linear"
    },
    slow: {
      duration: 3000,
      easing: "linear"
    }
  }
};
// Animate using the fast duration.
$button.velocity(fadeIn.p, fadeIn.o.fast);
/* Animate using the slow duration. */
$modal.velocity(fadeIn.p, fadeIn.o.slow);
```

If this seems like too much overhead, and if you have few enough lines of JavaScript to justify simply inlining all your animation logic, then don’t feel like a bad developer for skipping this approach altogether. You should always use whichever degree of abstraction best suits the scope of your project. The takeaway here is simply that animation workflow best practices do exist if you find yourself needing them.
CODE TECHNIQUE: ORGANIZE SEQUENCED ANIMATIONS

Velocity has a small add-on plugin called the *UI pack* (get it at VelocityJS.org/#uiPack). It enhances Velocity with features that greatly improve the UI animation workflow. Many of the techniques in this chapter, including the one discussed below, make use of it.

To install the UI pack, simply include a `<script>` tag for it after Velocity and before the ending `</body>` tag of your page:

```html
<script src="velocity.js"></script>
<script src="velocity.ui.js"></script>
```

The specific UI pack feature discussed in this section is called *sequence running*. It will forever change your animation workflow. It is the solution to messily nested animation code.

STANDARD APPROACH

Without the UI pack, the standard approach to consecutively animating separate elements is as follows:

```javascript
// Animate element1 followed by element2 followed by element3
$element1.velocity({ translateX: 100, opacity: 1 }, 1000, function() {
    $element2.velocity({ translateX: 200, opacity: 1 }, 1000, function() {
        $element3.velocity({ translateX: 300, opacity: 1 }, 1000);
    });
});
```

Don’t let this simple example fool you: in real-world production code, animation sequences include many more properties, many more options, and many more levels of nesting than are demonstrated here. Code like this most commonly appears in loading sequences (when a page or a subsection first loads in) that consist of multiple elements animating into place.
Note that the code shown above is different from chaining multiple animations onto the same element, which is hassle-free and doesn’t require nesting:

// Chain multiple animations onto the same element
$element1
    .velocity({ translateX: 100 })
    .velocity({ translateY: 100 })
    .velocity({ translateZ: 100 });

So what’s wrong with first code sample (the one with different elements)? Here are the main issues:

- The code bloats horizontally very quickly with each level of nesting, making it increasingly difficult to modify the code within your IDE.
- You can’t easily rearrange the order of calls in the overall sequence (doing so requires very delicate copying and pasting).
- You can’t easily indicate that certain calls should run parallel to one another. Let’s say that halfway through the overall sequence you want two images to slide into view from different origin points. When coding this in, it wouldn’t be obvious how to nest animations that occur after this parallel mini-sequence such that the overall sequence doesn’t become even more difficult to maintain than it already is.

**OPTIMIZED APPROACH**

Before you learn about the beautiful solution to this ugly problem, it’s important to understand two simple features of Velocity. First, know that Velocity accepts multiple argument syntaxes: the most common, when Velocity is invoked on a jQuery element object (like all the code examples shown so far), consists of a properties object followed by an options object:

// The argument syntax used thus far
$element.velocity({ opacity: 1, top: "50px" }, { duration: 1000, easing: "linear" });

An alternative syntax pairs with Velocity’s utility function, which is the fancy name given to animating elements using the base Velocity object instead of chaining off of a jQuery element object. Here’s what animating off the base Velocity object looks like:

// Velocity registers itself on jQuery's $ object, which you leverage here
$.Velocity({ e: $element, p: { opacity: 1, scale: 1 }, o: { duration: 1000, easing: "linear" } });
As shown above, this alternative syntax consists of passing Velocity a single object that contains member objects that map to each of the standard Velocity arguments (elements, properties, and options). For the sake of brevity, the member object names are truncated to the first letter of their associated objects (e for elements, p for properties, and o for options).

Further, note that you’re now passing the target element in as an argument to Velocity since you’re no longer invoking Velocity directly on the element. The net effect is exactly the same as the syntax you used earlier.

As you can see, the new syntax isn’t much bulkier, but it’s equally—if not more—expressive. Armed with this new syntax, you’re ready to learn how the UI pack’s sequence-running feature works: you simply create an array of Velocity calls, with each call defined using the single-object syntax just demonstrated. You then pass the entire array into a special Velocity function that fires the sequence’s calls successively. When one Velocity call is completed, the next runs—even if the individual calls are targeting different elements:

```javascript
// Create the array of Velocity calls
var loadingSequence = [
  { e: $element1, p: { translateX: 100, opacity: 1 },
    o: { duration: 1000 } },
  { e: $element2, p: { translateX: 200, opacity: 1 },
    o: { duration: 1000 } },
  { e: $element3, p: { translateX: 300, opacity: 1 },
    o: { duration: 1000 } }
];

// Pass the array into $.Velocity.RunSequence to kick off the sequence
$.Velocity.RunSequence(loadingSequence);
```

The benefits here are clear:

- You can easily reorder animations in the overall sequence without fear of breaking nested code.
- You can quickly eyeball the difference between properties and options objects across the calls.
- Your code is highly legible and expressive to others.
If you combine this technique with the previous technique (turning CSS classes into JavaScript objects), your animation code starts to look remarkably elegant:

```javascript
$.Velocity.RunSequence(
    [  
        { e: $element1, p: { translateX: 100, opacity: 1 }, o: slideIn.o },  
        { e: $element2, p: { translateX: 200, opacity: 1 }, o: slideIn.o },  
        { e: $element3, p: { translateX: 300, opacity: 1 }, o: slideIn.o } 
    ]);
```

Expressiveness and maintainability aren’t the only benefits to sequence running: you also gain the ability to run individual calls in parallel using a special `sequenceQueue` option which, when set to `false`, forces the associated call to run parallel to the call that came before it. This lets you have multiple elements animate into view simultaneously, giving a single Velocity sequence the power to intricately control timing that would normally have to be orchestrated through messy callback nesting. Refer to the inlined comments below for details:

```javascript
$.Velocity.RunSequence([  
    { elements: $element1, properties: { translateX: 100 },  
    → options: { duration: 1000 } },  
    // The following call will start at the same time as the first  
    → call since it uses the `sequenceQueue: false` option  
    { elements: $element2, properties: { translateX: 200 },  
    → options: { duration: 1000, sequenceQueue: false },  
    // As normal, the call below will run once the second call has completed  
    { elements: $element3, properties: { translateX: 300 },  
    → options: { duration: 1000 }
    ]);
```
**CODE TECHNIQUE: PACKAGE YOUR EFFECTS**

One of the most common uses of motion design is fading content in and out of view. This type of animation often consists of a series of individual animation calls that are chained together to deliver a nuanced, multistage effect.

**STANDARD APPROACH**

Instead of simply animating the opacity of an element toward 1, you might simultaneously animate its *scale* property so that the element appears to both fade in and grow into place. Once the element is fully in view, you might choose to animate its border thickness to `1rem` as a finishing touch. If this animation were to happen multiple times across a page, and on many different elements, it would make sense to avoid code repetition by turning it into a standalone function. Otherwise, you’d have to repeat this non-expressive code throughout your `script.js`:

```
$element
  .velocity({ opacity: 1, scale: 1 }, { duration: 500,
    easing: "ease-in-out" })
  .velocity({ borderWidth: "1rem" }, { delay: 200,
    easing: "spring", duration: 400 });
```

Unlike the sequencing technique discussed in the previous section, the code above consists of multiple animations that all occur on the *same* element. Chained animations on a singular element constitute an *effect*. If you were to improve this effect by implementing the first technique in this chapter (turning CSS classes into JavaScript objects), you’d have to go out of your way to uniquely name each argument object for each stage in the overall animation. Not only is it possible that these objects wouldn’t be used by other portions of the animation code due to the uniqueness of this particular sequence, but you’d have to deal with appending integers to each animation call’s respective objects to delineate them from one another. This could get messy, and could neutralize the organizational benefit and brevity of turning CSS classes into JavaScript objects.

Another problem with effects such as the one above is that the code isn’t very self-descriptive—its purpose isn’t immediately clear. Why are there two animation calls instead of one? What is the reasoning behind the choice of properties and options for each of these individual calls? The answers to these questions are irrelevant to the code that triggers the animation, and should consequently be tucked away.
OPTIMIZED APPROACH

Velocity’s UI pack lets you register effects that you can subsequently reuse across a site. Once an effect is registered, you can call it by passing its name into Velocity as its first parameter:

```javascript
// Assume we registered our effect under the name "growIn"
$element.velocity("growIn");
```

That’s a lot more expressive, isn’t it? You quickly understand the code’s purpose: An element will grow into view. The code remains terse and maintainable.

What’s more, a registered effect behaves identically to a standard Velocity call; you can pass in an options object as normal and chain other Velocity calls onto it:

```javascript
$element
  // Scroll the element into view
  .velocity("scroll")
  // Then trigger the "growIn" effect on it, with the following settings
  .velocity("growIn", { duration: 1000, delay: 200 })
```

If the UI pack is loaded onto your page, an effect such as this is registered using the following syntax:

```javascript
$.Velocity.RegisterEffect(name, {
  // Default duration value if one isn't passed into the call
defaultDuration: duration,
  // The following Velocity calls occur one after another,
  // with each taking up a predefined percentage of the effect’s total duration
  calls: [
    [ propertiesObject, durationPercentage, optionsObject ],
    [ propertiesObject, durationPercentage, optionsObject ]
  ],
  reset: resetPropertiesObject
});
```
Let’s break down this template step by step:

1. The first argument is the name of the effect. If the effect is responsible for bringing an element into view (as in, it fades an element’s opacity from 0 to 1), it’s important to suffix the effect with “In”.

2. The second argument is an object that defines the effect’s behavior. The first property in this object is \texttt{defaultDuration}, which lets you specify the duration the full effect should take if one is not passed into the Velocity call that triggers the effect.

3. The next property in the object is the \texttt{calls} array, which consists of the Velocity calls that constitute the effect (in the order that they should occur). Each of these array items is an array itself, which consists of the call’s properties object, followed by the optional percentage of the total duration which that call should consume (a decimal value that defaults to 1.00), followed by an optional options object for that specific call. Note that Velocity calls specified within the \texttt{calls} array accept only the \texttt{easing} and \texttt{delay} options.

4. Finally, you have the option of passing in a \texttt{reset} object. The \texttt{reset} object is specified using the same syntax as a standard Velocity properties map object, but it is used to enact an immediate value change upon completion of the full effect. This is useful when you’re animating the opacity and scale properties of an element down to 0 (out of view), but want to return the element’s scale property to 1 after the element is hidden so that subsequent effects needn’t worry about the properties beyond \texttt{opacity} they must reset on the element for their calls to properly take effect. In other words, you can leverage the \texttt{reset} properties map to make effects self-contained, such that they leave no clean up duties on the target elements.

In addition to the \texttt{reset} object, another powerful workflow bonus of the UI pack’s effect registration is automatic \texttt{display} property toggling. When an element begins animating into view, you want to ensure its \texttt{display} value is set to a value other than “none” so the element is visible throughout the course of its animation. (Remember, \texttt{display: none} removes an element from the page’s flow.) Conversely, when fading an element out, you often want to ensure its \texttt{display} value is switched to "none" once its opacity hits 0. This way, you remove all traces of the element when you’re done using it.

Using jQuery, \texttt{display} toggling is accomplished by chaining the \texttt{show()} and \texttt{hide()} helper functions onto animations (oftentimes messily buried within nested callbacks). With Velocity’s UI pack, however, this logic is taken care of automatically when you suffix your effect names with “In” and “Out” as appropriate.
Let's register two UI pack effects—one for the “In” direction and one for the “Out” direction—and call the element “shadowIn” since it consists of fading and scaling an element into view, then expanding its `boxShadow` property outward:

```javascript
$.Velocity.
  .RegisterEffect("shadowIn", {
    defaultDuration: 1000,
    calls: [
      [ { opacity: 1, scale: 1 }, 0.4 ],
      [ { boxShadowBlur: 50 }, 0.6 ]
    ]
  })
  .RegisterEffect("shadowOut", {
    defaultDuration: 800,
    calls: [
      // We reverse the order to mirror the "In" direction
      [ { boxShadowBlur: 50 }, 0.2 ],
      [ { opacity: 0, scale: 0 }, 0.8 ]
    ]
  });
```

If your effect’s name ends with “Out”, Velocity will automatically set the element’s `display` property to “none” once the animation is complete. Conversely, if your effect’s name ends with “In”, Velocity will automatically set the element’s `display` property to the default value associated with the element’s tag type (for example, "inline" for anchors, "block" for `div` and `p`). If your effect’s name does not contain one of these special suffixes, the UI pack will not perform automatic `display` setting.

Registering effects not only improves your code, but also makes it highly portable between projects and among fellow developers. When you’ve designed an effect you love, now it’s painless to share the effect’s registration code with others so they can use it too. Pretty neat!
DESIGN TECHNIQUES

The techniques discussed so far in this chapter will improve your workflow during the coding phase of motion design. The techniques covered in this section focus on the design phase, where you’re still experimenting to find the perfect animation that fits your UI. This phase requires a lot of creativity and a lot of repetition, and is accordingly ripe for workflow improvements.

TIMING MULTIPLIERS

The first design technique is to use a global timing multiplier. This consists of sprinkling in a multiplier constant against all of your animations’ delay and duration values.

Start by defining your global timing multiplier (arbitrarily designated as M for multiplier):

```javascript
var M = 1;
```

Then, bake the multiplier into the duration and delay option values within each animation call:

```javascript
$element1.animate({ opacity: 1 }, { duration: 1000 * M });
$element2.velocity({ opacity: 1 }, { delay: 250 * M });
```

**NOTE:** if you use SASS or LESS, which provide support for variable usage within stylesheets, this technique applies equally to CSS animations!

Embedding a multiplier constant will help you quickly modify the M constant in one location (presumably at the top of your style.js) in order to quickly speed up or slow down all of the animations across your page. Benefits of such timing control include:

- Slowing down animations to perfect the timing of individual animation calls within a complex animation sequence. When you’re constantly refreshing your page in order to tweak a multi-element animation sequence to perfection, seeing the sequence in slow motion makes it significantly easier to assess how individual elements interact with one another.
Speeding up animations when you’re performing repetitive UI testing. When you’re testing a site for purposes other than animation, evaluating the end state of UI animations (how elements wind up) is more important than testing the animations’ motion. In these situations, it saves time and reduces headaches to speed up all the animations across your page so you’re not repeatedly waiting for your animations to play out on each page refresh.

Velocity has a handy implementation of this functionality called mock, which functions as a behind-the-scenes global multiplier so you don’t have to sprinkle in the M constant by hand. Like the example shown above, mock multiplies both the duration and the delay values. To turn mock on, temporarily set $.Velocity.mock to the multiplier value you want to use:

```javascript
// Multiply all animation timing by 5
$.Velocity.mock = 5;
// All animations are now time-adjusted
// The duration below effectively becomes 5000ms
$element.velocity({ opacity: 1 }, { duration: 1000 });
```

Velocity’s mock feature also accepts a Boolean value: setting mock to true sets all durations and delays to 0ms, which forces all animations to complete within a single browser timing tick, which occurs every few milliseconds. This is a powerful shortcut for quickly turning off all animations when they’re getting in the way of your UI development and testing.

**USE VELOCITY MOTION DESIGNER**

Velocity Motion Designer (VMD) was crafted with the sole purpose of helping developers streamline the creation phase of motion design. VMD is a bookmarklet that you load onto a page in order to design animations in real time. It allows you to double-click elements to open a modal that lets you specify animation properties and options for that element. You then hit Enter on your keyboard to watch the animation play out immediately—without a page refresh.

**NOTE:** Get Velocity Motion Designer at http://velocityjs.org/#vmd.
make motion design fun.
Once you’ve designed all your element animations exactly the way you want them, you can export your work into one-for-one Velocity code, which you can place immediately into an IDE for use in production. (The resulting code is also fully compatible with jQuery.)

Ultimately, VMD saves countless hours of development time by preventing constant IDE and browser tab switching and repeated UI state retriggering. Further, it streamlines the designer-to-developer workflow by allowing the two teams to work alongside one another in real time: with VMD, designers can implement motion design without having to familiarize themselves with a site’s JavaScript or CSS. They can simply hand off the exported Velocity code to the developers to integrate into the codebase at their discretion.

VMD is a highly visual tool—visit VelocityJS.org/#vmd to see the walkthrough video.
As you implement animation workflow techniques, you’ll notice the intimidating black box of motion design beginning to unfold. The beautifully intricate loading sequences found on cutting-edge sites like Stripe.com and Webflow.com will start to make sense to you. You’ll gain confidence in your ability to code animation sequences, and this newfound skill will reduce friction in your development routine, making it not only easier but also significantly more fun to accomplish your motion design goals.
Symbols and Numbers

$.animate() 13
3D
   CSS primer on 156
   transforms 96

A
Adobe After Effect, animating text and 80
Adobe Photoshop, SVG and 104
Alerts, leveraging user response 42–43
Android
   purchasing older devices from eBay 144
   realities of web performance 118
Animation demo
   behaviors 148–149
   code section for animation setup 153–154
   code section for circle animation 160–164
   code section for circle creation 154–155
   code section for container animation 156–159
   code structure 150–152
   overview of 147
   review 165
Animation libraries
   bypassing jQuery 6
   page scrolling functions 7
   SVG support 108
   types of 14
Animation reversal, performance features of JavaScript 7–8
Animations. See also Motion design
   breaking into steps 48–49
   effects on neighboring elements 130
   limiting in motion design 45
   mirroring 44
   older browsers problem 139
   older browsers solutions 139–140
   optimized coding approach to organizing sequenced animations 66–68
   performance. See Performance
   reducing concurrency 43
   reducing variety 44
   staggering 49
   standard coding approach to organizing sequenced animations 65–66
   of text. See Text animation workflows. See Workflows
Animations, with SVG
   animated logo example 112–113
   overview of 109
   passing properties 109
   positional attributes vs. transforms 110–111
   presentational attributes 110
Arguments, Velocity 16–18
Attributes, SVG markup 105–106
B

backgroundColor property, Velocity support for CSS color properties 31–32
backwards option, benefits in text animation 92–93
Baselines, load testing and 120
Batching DOM additions
  code section for circle creation 155
  problem 126–127
  solutions 127–128
begin option, Velocity 24
Bézier curves, easing values in Velocity 22
Blast.js
  customClass option 85–86
  delimiter option 85
  generateValueClass option 86–87
  how it works 83–84
  installing on pages 84–85
  preparing text elements using 82–83
  reverse option 88–89
  tag option 87–88
Blue, Velocity support for CSS color properties 31–32
body tag, installing Blast and 84
Bold text, tag option in Blast and 88
Boolean values, generateValueClass option in Blast 86–87
borderColor property, Velocity support for CSS color properties 31–32
border-radius set property, in behavior of animation demo 148
Bottlenecks
  problem 133
  solutions 133–134
Bottom line, performance affecting 117
box-shadow property, CSS
  in behavior of animation demo 148
  overview of 138

Browsers
  animations on older browsers
    problem 139
  animations on older browsers
    solution 139–140
  bottlenecks and 133
  finding performance threshold early on 141–143
  positional attributes vs. transforms and 110
  realities of web performance 118
  support for older versions 4
BrowserStack.com, testing browsers on 142
Buttons, uses of SVG 109

C

Callback functions, begin and complete options in Velocity 24
Chaining
  effects and 69
  using Velocity with jQuery and 16
  in Velocity 20
character delimiter, Blast.js 82, 85
Chrome, realities of web performance 118
circle element
  in behavior of animation demo 148
code section for circle animation 160–164
code section for circle creation 154–155
code structure for animation demo 153–154
SVG presentational attributes 106
SVG styling 106
Classes
  customClass option in Blast 85–86
  generateValueClass option in Blast 86–87
<table>
<thead>
<tr>
<th>Code/coding techniques</th>
<th>Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>code section for animation</td>
<td>code section for container animation</td>
</tr>
<tr>
<td>setup 153–154</td>
<td>156–159</td>
</tr>
<tr>
<td>code section for circle animation</td>
<td>code structure for animation demo</td>
</tr>
<tr>
<td>160–164</td>
<td>153–154</td>
</tr>
<tr>
<td>code section for circle creation</td>
<td>SVG (⟨svg⟩)</td>
</tr>
<tr>
<td>154–155</td>
<td>105</td>
</tr>
<tr>
<td>code section for container animation</td>
<td>text elements</td>
</tr>
<tr>
<td>156–159</td>
<td>80</td>
</tr>
<tr>
<td>code structure for animation demo</td>
<td>150–152</td>
</tr>
<tr>
<td>creating images through code in SVG</td>
<td>104</td>
</tr>
<tr>
<td>optimized approach to organizing sequenced animations</td>
<td>optimized approach to packaging effects</td>
</tr>
<tr>
<td>66–68</td>
<td>70–72</td>
</tr>
<tr>
<td>optimized approach to packaging effects</td>
<td>optimized approach to separating styling from logic</td>
</tr>
<tr>
<td>69</td>
<td>60–65</td>
</tr>
<tr>
<td>standard approach to organizing sequenced animations</td>
<td>standard approach to separating styling from logic</td>
</tr>
<tr>
<td>65–66</td>
<td>59–60</td>
</tr>
<tr>
<td>standard approach to packaging effects</td>
<td>what good code looks like</td>
</tr>
<tr>
<td>69</td>
<td>57</td>
</tr>
<tr>
<td>standard approach to separating styling from logic</td>
<td>color property, Velocity support for CSS color properties</td>
</tr>
<tr>
<td>59–60</td>
<td>31–32</td>
</tr>
<tr>
<td>what good code looks like</td>
<td>Colors</td>
</tr>
<tr>
<td>57</td>
<td>performance benefits of using opacity instead of</td>
</tr>
<tr>
<td>132</td>
<td>Velocity options</td>
</tr>
<tr>
<td>31–32</td>
<td>complete option, Velocity</td>
</tr>
<tr>
<td>24</td>
<td>Compression, SVG and</td>
</tr>
<tr>
<td>104</td>
<td>3D primer</td>
</tr>
<tr>
<td>Concurrency</td>
<td>CSS</td>
</tr>
<tr>
<td>problem 133</td>
<td>animation effects on neighboring elements</td>
</tr>
<tr>
<td>reducing in motion design 43</td>
<td>130–131</td>
</tr>
<tr>
<td>solutions 133–134</td>
<td>appropriate uses of CSS workflow</td>
</tr>
<tr>
<td>Consistency, pattern recognition and understanding and</td>
<td>57–58</td>
</tr>
<tr>
<td></td>
<td>benefit of switching to JavaScript for segregation of logic</td>
</tr>
<tr>
<td></td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>comparing SVG positional attributes with CSS transforms</td>
</tr>
<tr>
<td></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>comparing Velocity display and visibility options with</td>
</tr>
<tr>
<td></td>
<td>27–29</td>
</tr>
<tr>
<td></td>
<td>comparing Velocity properties with CSS properties</td>
</tr>
<tr>
<td></td>
<td>18–19</td>
</tr>
<tr>
<td></td>
<td>comparing Velocity values with CSS values</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>easing values in Velocity</td>
</tr>
<tr>
<td></td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>fine-grained control of Blast elements</td>
</tr>
<tr>
<td></td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>issues with CSS workflow</td>
</tr>
<tr>
<td></td>
<td>56–57</td>
</tr>
<tr>
<td></td>
<td>JavaScript compared with</td>
</tr>
<tr>
<td></td>
<td>4–9</td>
</tr>
<tr>
<td></td>
<td>perspective properties</td>
</tr>
<tr>
<td></td>
<td>156–157</td>
</tr>
<tr>
<td></td>
<td>separating styling from logic</td>
</tr>
<tr>
<td></td>
<td>59–60</td>
</tr>
<tr>
<td></td>
<td>sneaky images and</td>
</tr>
<tr>
<td></td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>SVG styling compared with</td>
</tr>
<tr>
<td></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>Velocity arguments corresponding to</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Velocity support for CSS transform property</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>customClass option, Blast.js</td>
</tr>
<tr>
<td></td>
<td>85–86</td>
</tr>
</tbody>
</table>

INDEX 169
Data transfer indicators, preview options in motion design 41
Debouncing, event handlers 135–136
delay option, Velocity 26
Delay values
  staggering durations and 91
timing multipliers and 73
Delimiters, Blast.js 82, 85
Design techniques. See also Motion design
  page scrolling in Web design 7
timing multipliers 73–74
  VMD (Velocity Motion Designer) 74–76
Device Lab 142
display option, Velocity 27–28
div
  in behavior of animation demo 148
  Blast.js 82
  HTML elements 83
tag option in Blast 88
DOM (Document Object Model)
  batching DOM additions for improved performance 126–128, 155
  layout thrashing problem 121–122
  layout thrashing solution 122–123
  retrieving raw DOM elements 33–34
  SVG elements as DOM elements 104
duration option, Velocity 21
Durations
  limiting in motion design 45
  staggering 91
timing multipliers and 73

Easing options, Velocity 21–23
eBay, purchasing older devices from 144
Effects
  fade effect in UI pack 91
  fanciful effects in text 96
  flourishes in text 97–98
  optimized coding approach to packaging 70–72
  standard coding approach to packaging 69
  transition.fadeOut effect in UI pack 92
Elegance aspects, of motion design
  breaking animation into steps 48–49
  flowing from triggering elements 49
  graphics use 50
  not being frivolous 47
  opacity use 48
  overview of 39–40
  staggering animations 49
  using appropriate personality features 47–48
Element nodes, HTML 83
Elements
  animation effects on neighboring elements 130–132
circle element. See circle element
  fine-grained control of Blast elements 94
  flowing from triggering elements 49
  HTML element manipulation 148
  image rendering problems 137
  image rendering solutions 137–138
  JEOs (jQuery element objects) 123–124, 126–128
  preparing text elements for animation using Blast.js 82–83
  retrieving raw DOM elements 33–34
  span elements 87–88
  SVG elements compared with HTML elements 104
text elements 80
eq() function, jQuery  94
Event handlers, debouncing  135–136
Experimentation, benefits of repeatedly experimenting  51–52

F
Fade effect, in UI pack  91
Familiarity, use of conventions in making design choices  41
fill, SVG
  presentational attributes  105
  styling  107
Flags, leveraging user response  42–43
Flourishes, in text  97–98
Flow, creating from triggering elements  49
Force-feeding feature (Velocity),
  for avoiding layout thrashing problem  124–125
Privolous design, uses and abuses of  47

G
generateValueClass option,
  Blast.js  86–87
gets
  JEOs as culprit in layout thrashing  123–124
  layout thrashing and  121–122
Global timing multipliers  73–74
Gradients, CSS  138
Graphics
  in elegant motion design  50
  SVG and  104, 109
Green, Velocity support for CSS color properties  31–32
GSAP animation library  14
H
Height, SVG presentational attributes  105
Hidden setting, display and visibility options  28
Hover state animations, uses of CSS  6, 57–58
HTML
  coding web pages  80
  element manipulation  148
  element nodes  83
  SVG elements compared with HTML elements  104
I
Images
  creating through code in SVG  104
  rendering problems  137
  rendering solutions  137–138
  sneaky image problems  139
  sneaky image solutions  139–140
  img element  138
Incentives, visceral nature of interactions and  43
Infinite looping, in Velocity  25–26
  See also Loops
Inkscape  104
Inline status indication, engaging users in tasks  42
In-progress indicators, preview options in motion design  41–42
Internet Explorer
  animations on older browsers problem  139
  finding performance threshold early on  141–143
  positional attributes vs. transforms and  110
  realities of web performance  118
iOS, purchasing older devices from eBay 144
Irreversible actions, indicators for 43

J
Janks (stutters), layout thrashing and 121
JavaScript vs. CSS
  animation reversal feature in JavaScript 7–8
overview of 4
page scrolling feature in JavaScript 7
performance benefits 6
physics-based motion in JavaScript 8
review 10
workflow maintenance 9
JEOs (jQuery element objects)
  batching DOM additions for improved performance 126–128
as culprit in layout thrashing 123–124
jQuery
  easing options 22–23
fine-grained control of Blast elements 94
installing 15
JavaScript animation libraries that bypass 6
required by Blast 84–85
slowness of animation features in 4
standard coding approach to separating styling from logic 59
using Velocity with 16
using Velocity without 33–34
Velocity compared with 13
jQuery element objects. See JEOs (jQuery element objects)

L
Latency, search engine performance and 117
Layout thrashing
  force-feeding feature in Velocity for avoiding 124–125
JEOs (jQuery element objects) as culprit in 123–124
problem 121–122
solutions 122–123
Load testing, realities of web performance and 120
Logic
  optimized coding approach to separating from styling 59–60
standard coding approach to separating from styling 59
Logos
  animated logo example in SVG 112–113
uses of SVG 109
loop option, Velocity 25–26
Loops
  code section for container animation 159
  layout thrashing and 121–122

M
Maintenance, of workflows 9
Markup, SVG 105–106
Max values, code section for animation setup 154
Min values, code section for animation setup 154
Mock feature, Velocity 74
Motion design
  alerts and flags for leveraging user response 42–43
appropriate personality features 47–48
breaking animation into steps 48–49
conventions in making design choices 41
engaging users in tasks 42
experimenting repeatedly 51–52
flowing from triggering elements 49
graphics use 50
indicators of severity of irreversible actions 43
limiting animations 45
limiting durations 45
mirroring animations 44
not being frivolous 47
opacity use 48
overview of 37
previewing outcomes 41
reducing concurrency 43
reducing variety 44
review 53
staggering animations 49
utility and elegance of 39–40
UX (user experience) improved by 38
visceral nature of interactions 43
Mozilla Developer Network, directory of SVG elements 114
Multi-animation sequences, solutions to concurrency issues 134
Multipliers, timing multipliers as design technique 73–74

P
Page scrolling, performance features of JavaScript 7
See also scroll command
Performance
animation effects on neighboring elements problem 130
animation effects on neighboring elements solution 131–132
animations on older browsers problem 139
animations on older browsers solution 139–140
batch DOM additions problem 126–127
batch DOM additions solutions 127–128
bottleneck concurrency problems 133
bottleneck concurrency solutions 133–134
features of JavaScript 6
finding performance threshold early on 141–143
force-feeding feature in Velocity for avoiding layout thrashing 124–125
image rendering problems 137
image rendering solutions 137–138
JEOS (jQuery element objects) and 123–124
layout thrashing problem 121–122
layout thrashing solution 122–123
overview of 117
realities of web performance 118–120

O
Opacity
animation of 161
flourishes in text 97–98
going beyond overuse of 48
performance benefits of using instead of color 132
opacity property 161
outlineColor property, Velocity support for CSS color properties 31–32
Performance (continued)
review 145
scroll and resize event problems 135
scroll and resize event solutions 135–136
sneaky image problems 139
sneaky image solutions 139–140
Personality, using appropriate personality features in motion design 47–48
Perspective properties, CSS 156–157
Physics-based motion, performance features of JavaScript 8
Pixels, image rendering problems 137
Positional attributes, SVG 110–111
Presentational attributes, SVG 105, 110
Previews, previewing outcomes in motion design 41
Properties
in behavior of animation demo 148
CSS perspective properties 156–157
CSS shadow properties 138
passing properties in SVG animations 109
Velocity 18–19
Velocity support for CSS color properties 31–32
px, as default unit in Velocity 19–20

S
Safari, realities of web performance 118
Scalable vector graphics. See SVG (scalable vector graphics)
Scale, CSS transform property 32
scroll command
overview of 30–31
Velocity page scrolling 7
scroll events
performance problems 135
performance solutions 135–136
Scrolling, page animation and 137
Search engines, latency and 117
sentence delimiter, Blast options 84–85
Sequence running, in UI pack 65
Sequenced animations
optimized coding approach to organizing 66–68
standard coding approach to organizing 65–66
sets, layout thrashing and 121–122
setup
code section for animation setup 153–154
code structure for animation demo 150
Shadow properties, CSS 138

R
Random numbers, code section for animation setup 153
Red, Velocity support for CSS color properties 31–32
resize events
performance problems 135
performance solutions 135–136
Text nodes 80

text-shadow property, CSS 138

Thresholds, finding performance threshold early 141–143

Timing control
delay option 26

JavaScript vs. CSS 4

Timing multipliers, as design technique 73–74

transform property, Velocity 31–32

Transforms
3D CSS primer 156
3D transforms 96

animation effects on neighboring elements and 131

comparing SVG positional attributes with CSS transforms 110–111

transition.fadeOut effect, in UI pack 92

transition.perspectiveDown effect, in UI pack 96

Transitions
individual text parts 94–95

limiting durations 45

replacing existing text 90

staggering durations 91

text out of view 91–93

text using fanciful effects 96

text visibility 80

Translations
3D CSS primer 156

animation effects on neighboring elements and 131

animation of 162–163

code section for circle animation 160

CSS transform property 32

mirroring and 44

Triggers, flowing from triggering elements 49

Trigonometric easings, easing values in Velocity 22

U

UI (user interface)

conventions in making design choices 41

motion design improving user experience 38

UI animation libraries 14

UI animation workflow 65

UI pack
fade effect in 91

getting and installing 65

optimized coding approach to packaging effects 70–72

stagger feature in 133–134

transition.fadeOut effect 92

transitioning text fancifully 96

Unit types, values in Velocity 19–20

User experience. See UX (user experience)

User interface. See UI (user interface)

Utility aspects, of motion design

alerts and flags for leveraging user response 42–43

conventions in making design choices 41

engaging users in tasks 42

indicators of severity of irreversible actions 43

limiting animations 45

limiting durations 45

mirroring animations 44

overview of 39–40
previewing outcomes 41
reducing concurrency 43
reducing variety 44
visceral nature of interactions 43
Utility function, Velocity 66
UX (user experience)
  motion design improving 38
  performance affecting 117
  physics-based motion in JavaScript enhancing 8

V
Values
  code section for animation setup 154
  value functions 161
  Velocity 19–20
Variety, reducing in motion design 44
Velocity
  animation demo. See Animation demo
  arguments 16–18
  begin and complete options 24
  chaining 20
  color options 31–32
  compared with jQuery 13
  containing animation logic within 29
  delay option 26
  display and visibility options 27–28
  downloading and installing 15
  duration option 21
  easing options 21–23
  force-feeding feature for avoiding layout thrashing 124–125
  loop option 25–26
  mock feature 74
optimized coding approach to organizing sequenced animations 66–68
page scrolling functions 7
passing properties in SVG animations 109
physics-based motion 8
properties 18–19
resource for SVG attributes and styling properties 114
reverse command 30
review 33–34
scroll command 30–31
transform property 31–32
types of animation libraries 14
UI pack 65
using with jQuery 16
using without jQuery 33–34
values 19–20
Velocity Motion Designer (VMD) 74–76
Video. See also Images
  image rendering problems 137
  image rendering solutions 137–138
Visibility
  replacing existing text 90
  transitioning text out of view 91–93
  transitioning text visibility 80
visibility option, Velocity 27–28
Visual processing, leveraging primal instincts in motion design 42–43
VMD (Velocity Motion Designer) 74–76

W
Web design, use of page scrolling in 7
Web performance, realities of 118–120
Width, SVG presentational attributes 105
word delimiter, Blast options  85

Workflows

CSS appropriate uses  57–58
CSS issues  56–57
maintainability of  9
optimized coding approach to organizing sequenced animations  66–68
optimized coding approach to packaging effects  70–72
optimized coding approach to separating styling from logic  60–65
overview of  55
review  77
standard coding approach to organizing sequenced animations  65–66

standard coding approach to packaging effects  69
standard coding approach to separating styling from logic  59–60
timing multipliers as design technique  73–74
VMD (Velocity Motion Designer)  74–76

X

x value, SVG presentational attributes  105

Y

y value, SVG presentational attributes  105