C

The Web In Motion

Practical Considerations For Designing With Animation

SMASHING MAGAZINE

C

The Web In Motion

Practical Considerations For Designing With Animation

SMASHING MAGAZINE

Imprint

© 2015 Smashing Magazine GmbH, Freiburg, Germany
ISBN: 978-3-945749-10-4 (Kindle)
Cover Design: Veerle Pieters
eBook Strategy and Editing: Vitaly Friedman
Technical Editing: Cosima Mielke
Planning and Quality Control: Vitaly Friedman, Iris Lješnjanin
Tools: Elja Friedman
Syntax Highlighting: Prism by Lea Verou
Idea & Concept: Smashing Magazine GmbH

About This Book

After the golden times of Flash were over, animations led a rather shadowy existence on the web for quite some time. They were considered as unnecessary gimmicks and superfluous add-ons, but things were about to change. With apps already benefiting from their responsive interfaces, the importance of both animation and motion design, as well as their ability to make the user experience more delightful, was growing evermore.

Animation is not about mere decoration, but (when used sparingly) can turn out to be a catalyst for making the interaction with a website more intuitive and memorable. So, what is the current state of animation on the web? Where is it heading? And how can you tackle the possibilities and challenges it brings along? These are just a few of the many questions that are tackled and discussed in this eBook, which are bound to help you grasp what meaningful motion design is all about and how you can implement it into your own projects.

Furthermore, popular animation libraries are important to consider along the way, as well as taking a closer look at designing performant UI animations and diving deep enough into the CSS behind motion. To reapproach the JavaScript vs. CSS camps, a new API is introduced, among a good number of practical coding examples that will help you get to grips with the principles behind smooth and natural animations. So, what are we waiting for? Let’s put the web in motion.

TABLE OF CONTENTS

by Stephen Greig

Styling And Animating SVGs With CSS

by Sara Soueidan

About The Authors

The State Of Animation 2014

BY RACHEL NABORS ❧

The post-Flash era is hardly free of animation. CSS animation is quickly becoming a cornerstone of user-friendly interfaces on mobile and desktop, and JavaScript libraries already exist to handle complex interactive animations. In the wake of so much "CSS versus JavaScript animation" infighting, a new API specifically for web animation is coming out that might just unite both camps.

It's an exciting time for web animation, and also a time of grave miscommunication and misinformation. In 2014, I had the chance to travel the world to talk about using animation in user interfaces and design[1]. I met and interviewed dozens of people who use and champion both CSS and JavaScript. After interviewing so many developers, designers and browser representatives, I discovered a technical and human story to be told.

What you're about to read is purely observational and as unbiased an account as you will be able to find on the subject of web animation.

Flash May Be Gone, But The Era Of Web Animation Has Just Begun

Since the era of Flash, it’s become fashionable to think of animation as little more than decoration, a “flashy” afterthought, often in poor taste, like an unwelcome `blink` tag. But unless we want to display nothing other than copy on a screen, animation is still very much our friend.

For user interface designers, animation reinforces hierarchy, relationships, structure, and cause and effect. Research going back to the early ’90s[2] demonstrates that animation helps humans understand what’s happening on screen. Animation stitches together app states and offloads that work to the brain’s GPU — the visual cortex.

For interaction developers, complex visuals — from infographics on dashboards to video games on tablets — are impossible to create without animation to glue all the pieces together. If we want those things on the Internet, we still need animation.

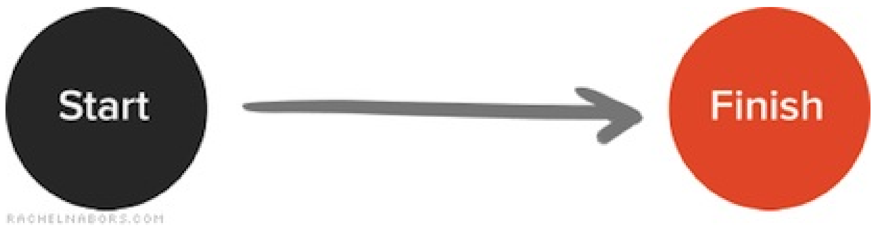
Sadly, we have fallen behind in the motion design race. Products that use animation to benefit their users will succeed where their static or animation-abusing competitors will fail. As it stands, native apps are beating the pants off the web. App developers have been incorporating animation into their designs and fleshing out workflow and prototyping tools like Flinto[3] and Mitya[4] from day one.

But things might be turning around. iOS’ Safari team pushed out the CSS animation and transition specifications so that websites can look and feel as good as iOS apps do. Even Google has picked up on this, putting animation front and center[5] in its Material Design recommendations, with careful do’s and don’ts to apply animations and transitions meaningfully, with purpose.

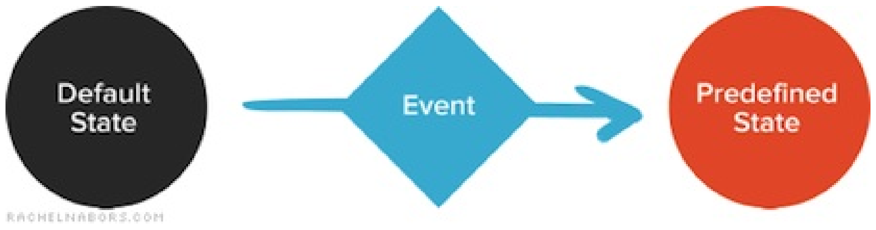
Animation is the natural next step in the evolution of our application and device ecosystem. It makes the digital world more intuitive and interesting for users of all ages. And so long as our devices sport GPUs, it’s not going away.

Animating All The Things

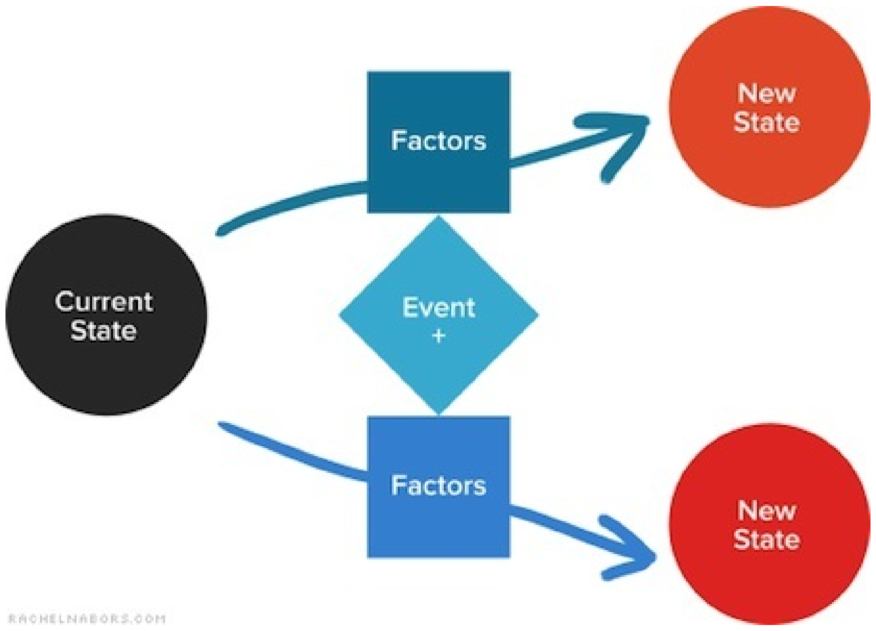
At its core, animation is just a visual representation of a rate of change over time and space. All animation can be distilled into three types: **Static animation** runs from a start point to an end point, with no branching or logic. This can be accomplished with CSS alone[6], as the abundance of CSS loading animations[7] testifies.



- **Stateful animation** in its simplest form takes boolean input — a click to open a menu and a click to close it[8], for instance — and animates between the two states. This is currently achieved in JavaScript frameworks by applying and removing classes with scoped CSS animation.

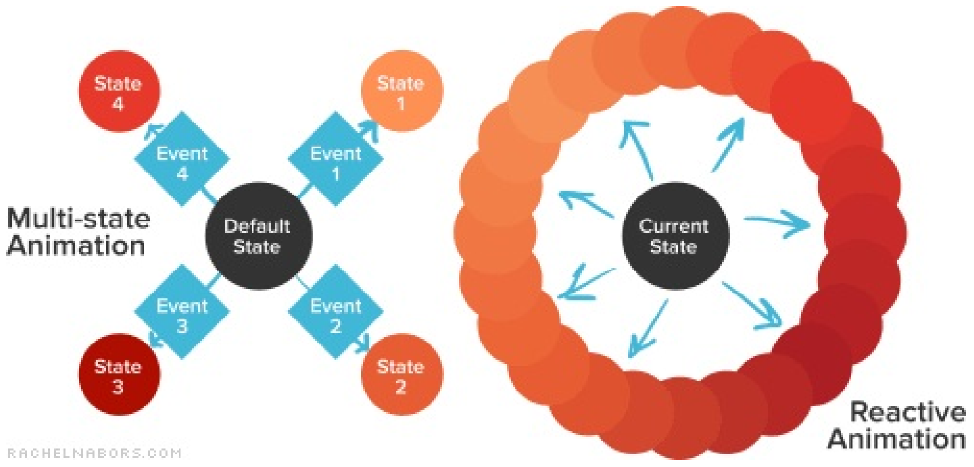


- **Dynamic animation** can have many outcomes depending on user input and other variables. It uses its own logic to determine how things should animate and what their endpoints are. It could entail “dragging” a page along behind your finger according to the speed of your swipe, or dynamically changing a graph as new data comes in. This is the trickiest kind of animation to accomplish with the tools at our disposal today. CSS alone cannot be used for this kind of animation.



MORE STATES != DYNAMIC ANIMATION

The astute CSS developer might point out that, with enough states, CSS animation could closely resemble dynamic animation. This is true to a point. But truly dynamic animation has more end states than you can possibly anticipate.



Just imagine the humble loading bar. Having a different class for every percentage point of “fullness” would easily run to a hundred lines of CSS, not to mention the number of times your JavaScript would have to touch the DOM to update the classes and the browser repaints. We definitely could write a more performant dynamic loader with JavaScript alone.

CSS animation is declarative: Aside from a handful of pseudo-classes, such as `:hover` and `:target`, it accepts context from neither the user nor the user’s surroundings. It does only what its author tells it to do and cannot respond to new inputs or a changing environment. There’s no way to create a CSS animation that says “If you’re in the middle of the page, do this; otherwise, do that.” CSS doesn’t contain that sort of logic.

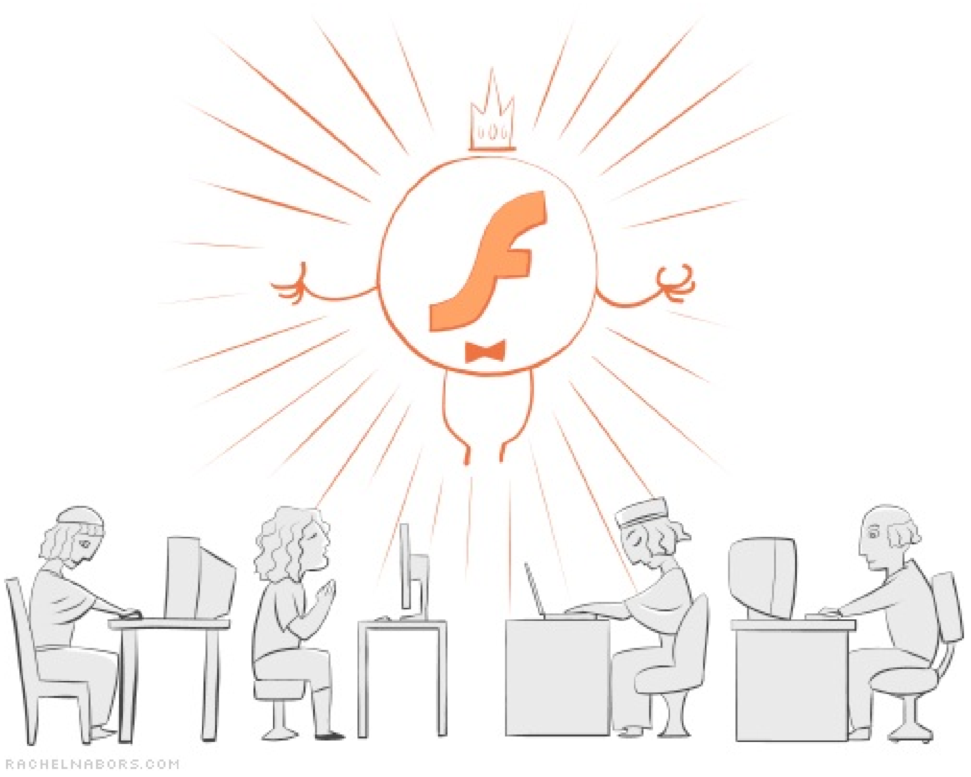
When CSS-first developers need logic, they often start by scoping CSS to state classes, with JavaScript handling the logic of when to apply which class. Frameworks such as AngularJS[9] support states, and many UI interactions adapt easily to a handful of states like “loading,” “open” and “selected.” These animations also degrade gracefully in old browsers, providing a much needed UX boost where CSS animation and transitions are supported.

Interaction developers have had a different time of it. CSS animation is often too declarative to handle the things these developers want to build. Paying clients demand reliable animation across a wide spread of browsers; so, many interaction developers and their studios have done what all clever developers do:

make labor-saving libraries customized to their own processes. Some of these libraries, like GSAP[10] and Velocity[11], are available to and developed for the public. But many others will never be released in the wild, because the people and agencies who created them don't have the time or money (or will) to support an open-source project.

And then we can release it to the public?
HAHA HAHA HAHA HAHA
Sure, right after we catch up on client work!
INHOUSE 4 LIFE
INHOUSE 4 LIFE
RACHELNABORS.COM

This is a deeply worrying story that I've heard over and over again, and it suggests that many developers are reinventing the wheel without moving the web forward. There isn't enough demand for something considered "nice to have" to support many competitors. It's easy to see how libraries like GSAP must be commercial in order to survive, or how sponsorships buoy libraries like Velocity.



And Flash was a benevolent dictator, for its people did have a visual timeline UI.

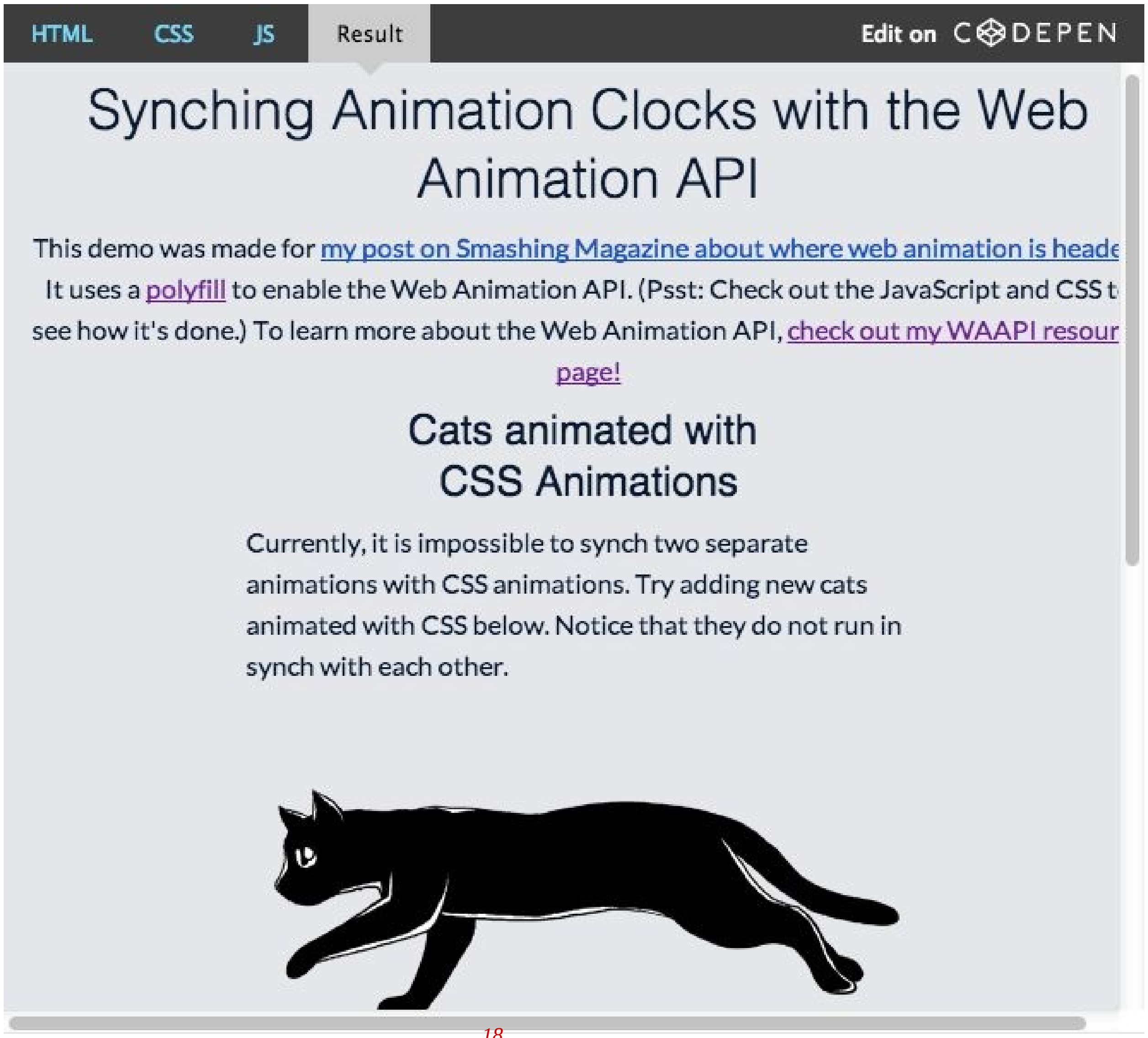
Flash did a great thing by giving interaction developers and UI designers a universal workflow that accommodates all kinds of animations and a platform on which to edit them. But since Steve Jobs announced back in 2010 that the iPhone would never support Flash[12], many former Flash developers have quietly gone into exile, taking their niche knowledge with them. Now, an entire generation of web designers has come online with relatively no knowledge of the possibilities and challenges we face when ramping up animation complexity.

But things are about to get quite… animated.

The Web Animation API: ~~The Greatest~~ *An* API You’ve Never Heard Of

The Web Animation API is a W3C specification that provides a unified language for CSS and SVG animations and that opens up the browser’s animation engine for developer manipulation. It does the following: provide an API for the animation engine, enabling us to develop more in-browser animation tools and letting animation libraries tap into performance boosts; give (qualifying) animation their own thread, getting rid of jank; support motion paths[13]; provide post-animation callbacks; reintroduce nested and sequenced animations[14] that we haven’t seen since Flash; enable us to pause, play, seek, reverse, slow down or speed up playback with timing dictionaries[15] and animation player objects[16].

Here’s just one example of what the Web Animation API can do that CSS alone cannot[17].



Synching Animation Clocks with the Web Animation API

This demo was made for my post on Smashing Magazine about where web animation is heade
It uses a polyfill to enable the Web Animation API. (Psst: Check out the JavaScript and CSS t
see how it's done.) To learn more about the Web Animation API, check out my WAAPI resour
page!

Cats animated with CSS Animations

Currently, it is impossible to synch two separate animations with CSS animations. Try adding new cats animated with CSS below. Notice that they do not run in synch with each other.

See the Pen Running on Web Animations API[18] *on CodePen.*

SUPPORT

The Web Animation API has been over two years in the making, and its features have been rolling out in Chrome and Firefox nightlies for the past six months. Mozilla is the major force behind the specification. Meanwhile, the Chrome team has been prioritizing the shipment of features.

Microsoft has the API “under consideration”[19] for Internet Explorer. Apple, surprisingly, has also adopted a wait-and-see approach for Safari. And we can only fantasize about when the API will hit those web app engines powered by ancient forks of open-source browsers[20].

Early adopters who want to explore the API can try out a polyfill for the Web Animations API[21], which is being replaced by Web Animations Next[22] literally any day now (more about the polyfill and the API can be found on the website for the Polymer project[23]). However, for browsers that don’t support the API, the polyfill is still less performant than GSAP, the reigning champion of JavaScript-based animation libraries. Thus, the polyfill isn’t something interactive that developers will want to put into production for high-performance projects.

It will be some time before this API is supported across the board. With half of browser makers waiting to see how developers will use it and most developers refusing to use a tool that isn’t widely supported, the API faces a chicken-and-egg scenario. However, in an on-stage conversation with Google’s Paul Kinlan at Fronteers, I suggested that, were the API to be fully supported in a closed and monetizable system for web apps, such as Google Play, developers would be able to safely use it in a walled garden until it reaches maturity and fuller support.

PERFORMANCE

The API’s authors and implementers hope that animation library developers will start feature-sniffing for the API’s support to tap into its performance benefits. Because the Web Animation API uses the CSS rendering engine, we can expect CSS levels of performance. Animations will run on their own thread as long as they don’t depend on anything happening in the main thread, such as JavaScript or layout.

Speaking of layout, reflowing remains one of the biggest processing hurdles for browsers. No CSS or JavaScript animation can get around it unless you’re pumping everything through WebGL straight to the GPU (which some clever in-house library developers have been doing). Aside from `opacity` and `transform`, animating the bulk of CSS properties will cause a reflow, a change in layout and/or a repaint of the pixels on the screen. The `will-change` CSS property helps some[24] by enabling us to point at something and tell the browser, “That, that thing is going to change. You do what you have to do change it efficiently.” The hope is that as browsers get smarter about graphics, they’ll move those elements into their own layer or otherwise optimize the way they handle those graphics. It’s the first step in eliminating hacks like `translateZ(0)`.

Such “performance hacks” often get slapped on as a magic fix whenever an animation is janking, but they often cause performance issues when used unwittingly. Performance decisions are truly best left to the browser in the long run. Fortunately, browser makers are scrambling to get fewer properties to trigger reflows, thus keeping them off the main thread. For animation library developers, this means that the Web Animation API could be a winning partner for performance in the near future.

TOOLS

Anyone working with web animation yearns for proper animation development tools: a timeline, property inspection, better editors, and the ability to pause, rewind and otherwise inspect an animation while debugging. The Web Animation API will open the guts of the CSS rendering engine to developers and the browser vendors themselves to create these tools. Both Chrome[25] and Firefox are preparing animation features for their development tools. This API opens up those possibilities.

The Web Animation Community Today

Not many people other than those working on it are aware of the Web Animation API. The standards community is eager for real-world feedback from interaction and animation library developers. However, many developers feel that the standardistas live in an ivory tower, far removed from the rigors of the trenches, the demands of clients and the lessons learned from Flash.

The old king’s champion sent into exile by the very people he once served.

To be fair, the standardistas haven’t exactly come out to meet their audience in the field. To join the “official” Web Animation API conversation, you must jump through some hoops, and getting on the email chain threatens to overflow the inbox of any busy developer. Alternatively, you could get on IRC and join the conversation there — if only designers used IRC. The conversation that needs to happen is unlikely to happen if the people who have the most to say simply aren’t there. Vendors and specification authors will need to find more ways to reach out to their key audience or else risk building an API without an audience.

But the standardistas aren’t the only ones with communication problems here. As a community, we are very judgmental and quick to deride things that we deem unworthy, be it Flash or an API we don’t like the look of. Many of us invest our egos in our tools and processes. But those things don’t define us.

What we create together defines us.

- **Animation library developers**, read the specification[26]. It is long, but if GreenSock's Jack Doyle can do it, so can you.

- **Interaction developers** who don't have all day to read a huge specification, just read the readme on the polyfill's page[27]. It's a perfect TLDR. Now that you know what's coming, you will know when it might be useful to you, whether for optimizing your library's performance or building an in-browser timeline UI.

- **Designers**, prioritize JavaScript at work. Play with the polyfill, and play with GSAP and Velocity. Find out what these things can do for your work that CSS alone cannot accomplish.

With web animation, we have a rare chance to put our egos aside and come together as a community to build a tool with which future generations of designers and developers can build great things. For their sake, I hope we can.

The art challenges the technology, and the technology inspires the art.
– John Lasseter, CCO Pixar

Resources

Rachel Nabors has an updated list of resources on the Web Animation API[28]. To join the unofficial conversation, look for the `#waapi` hash tag wherever you prefer to communicate.

- Web Animations[29] (API specification), W3C
- Web Animations polyfill[30] and Web Animation Next[31] (the next incarnation of the polyfill) GreenSock[32] animation library Velocity[33], a performant `.animate()` replacement for jQuery

JOIN THE CONVERSATION

- Official mailing list: email public-fx@w3.org[34], starting the subject line with `[webanimations] …`
- IRC: irc.w3.org#webanimations[35]

- Everywhere else: use the hash tag `#waapi` and engage with the community

MAKE A DIFFERENCE

People who have some familiarity with C++ coding can help implement the API in Firefox[36]. Brian Birtles[37] might even mentor you! And Mozilla is always looking for people to help write documentation on MDN[38].

Minor corrections to the specification (grammar, spelling, inconsistencies, etc.) can be submitted as pull requests on GitHub[39].

PEOPLE TO FOLLOW ON TWITTER

- Brian Birtles[40], a principal author of the specification and with Mozilla Japan Alex Danilo[41], Google platform team member and coauthor Tab Atkins — Googler[42], coauthor and contributor to the CSS specification Jack Doyle[43], member of GreenSock and GSAP

- Rachel Nabors[44], head of animation think tank Tin Magpie[45] ❧

1. http://www.slideshare.net/CrowChick/animation-and-the-future-of-ux-33573726
2. https://smartech.gatech.edu/bitstream/handle/1853/3627/93-17.pdf
3. https://www.flinto.com/
4. http://www.mitya-app.com/
5. http://www.google.com/design/spec/animation/authentic-motion.html
6. http://codepen.io/rachelnabors/full/rCost
7. http://codepen.io/collection/HtAne/
8. http://tympanus.net/Development/SimpleDropDownEffects/
9. https://docs.angularjs.org/api/ngAnimate
10. http://greensock.com/

11. http://julian.com/research/velocity/

12. https://www.apple.com/hotnews/thoughts-on-flash/

13. http://dirkschulze.github.io/specs/motion-1/

14. https://github.com/webanimations/webanimations-js#sequencing-and-synchronizing-animations

15. https://github.com/webanimations/webanimations-js#controlling-the-animation-timing

16. https://github.com/webanimations/webanimations-js#playing-animations

17. http://codepen.io/rachelnabors/pen/zxYexJ/

18. http://codepen.io/rachelnabors/pen/zxYexJ/

19. https://status.modern.ie/webanimationsjavascriptapi?term=animations

20. https://developer.amazon.com/appsandservices/solutions/platforms/webapps

21. https://github.com/webanimations/webanimations-js

22. https://github.com/webanimations/webanimations-next

23. https://www.polymer-project.org/platform/webanimations.html

24. https://dev.opera.com/articles/css-will-change-property/

25. http://src.chromium.org/viewvc/blink?view=revision&revision=183847

26. http://w3c.github.io/webanimations/

27. https://github.com/webanimations/webanimations-js#readme

28. http://rachelnabors.com/waapi/

29. http://w3c.github.io/webanimations/

30. https://github.com/webanimations/webanimations-js

31. https://github.com/webanimations/webanimations-next

32. http://greensock.com/

33. http://julian.com/research/velocity/

34. mailto:public-fx@w3.org

35. http://irc.w3.org#webanimations

36. https://developer.mozilla.org/en-US/docs/Introduction

37. https://twitter.com/brianskold

38. https://developer.mozilla.org/en-US/

39. https://github.com/w3c/webanimations

40. http://twitter.com/brianskold

41. http://twitter.com/alexanderdanilo

42. http://twitter.com/tabatkins

43. http://twitter.com/greensock

44. http://twitter.com/rachelnabors

45. http://tinmagpie.com/

A Quick Look Into The Math Of Animations With JavaScript

BY CHRISTIAN HEILMANN ❧

In school, I hated math. It was a dire, dry and boring thing with stuffy old books and very theoretical problems. Even worse, a lot of the tasks were repetitive, with a simple logical change in every iteration (dividing numbers by hand, differentials, etc.). It was exactly the reason why we invented computers. Suffice it to say, a lot of my math homework was actually done by my trusty Commodore 64 and some lines of Basic, with me just copying the results later on.

These tools and the few geometry lessons I had gave me the time and inspiration to make math interesting for myself. I did this first and foremost by creating visual effects that followed mathematical rules in demos, intros and other seemingly pointless things.

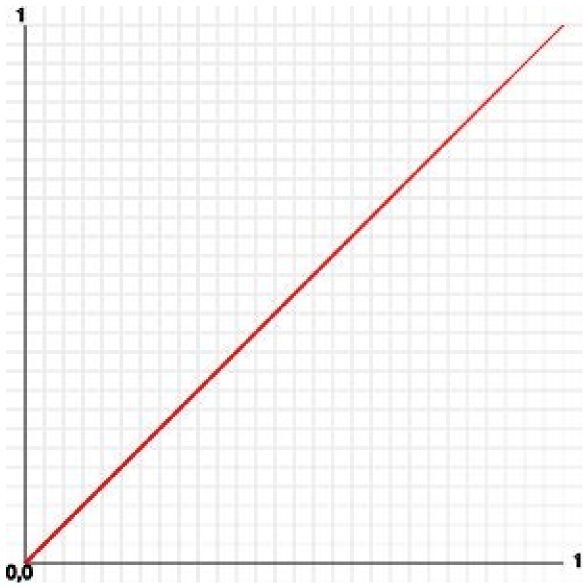
There is a lot of math in the visual things we do, even if we don't realize it. If you want to make something look natural and move naturally, you need to add a bit of physics and rounding to it. Nature doesn't work in right angles or linear acceleration. This is why zombies in movies are so creepy. This was covered on Smashing Magazine before in relation to CSS animation[1], but today let's go a bit deeper and look at the simple math behind the smooth looks.

Going From 0 To 1 Without Being Boring

If you've just started programming and are asked to go from 0 to 1 with a few steps in between, you would probably go for a `for` loop:

```
for ( i = 0; i <= 1; i += 0.1 ) {
  x = i;
  y = i;
…
}
```

This would result in a line on a graph that is 45 degrees. Nothing in nature moves with this precision:

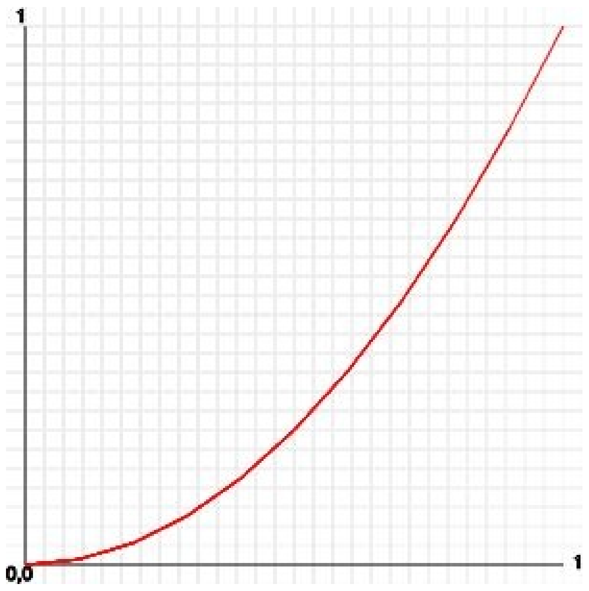


A simple way to make this movement a bit more natural would be to simply multiply the value by itself:

multiply the value by itself:

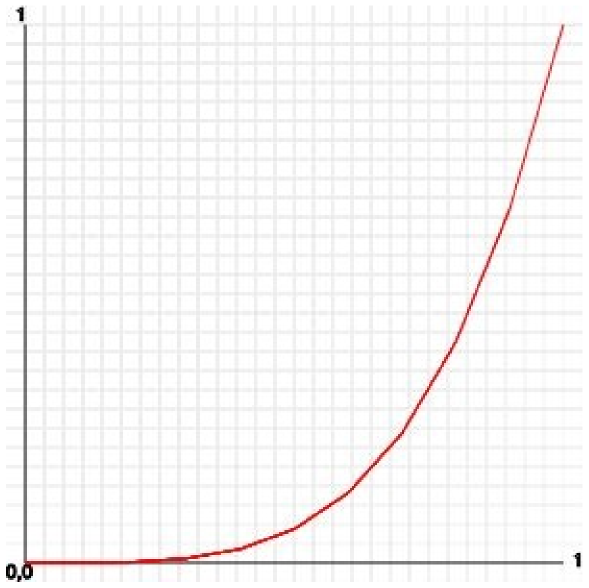
```
for ( i = 0; i <= 1; i += 0.1 ) {
  x = i;
  y = i * i;
}
```

This means that `0.1` is `0.01`, `0.2` is `0.04`, `0.3` is `0.09`, `0.4` is `0.16`, `0.5` is `0.25` and so on. The result is a curve that starts flat and then gets steeper towards the end:



You can make this even more pronounced by continuing to multiply or by using the “to the power of” `Math.pow()` function:

```
for ( i = 0; i <= 1; i += 0.1 ) {
  x = i;
  y = Math.pow( i, 4 );
}
```



This is one of the tricks of the easing functions used in libraries such as jQuery and YUI, as well as in CSS transitions and animations in modern browsers.

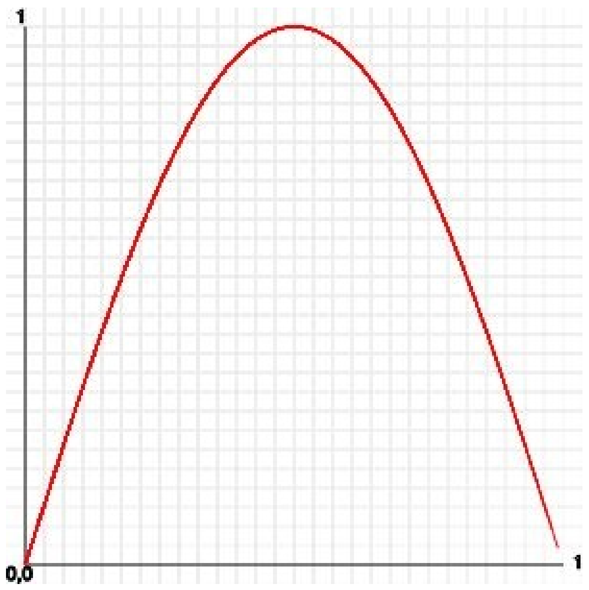
You can use this the same way, but there is an even simpler option for getting a value between 0 and 1 that follows a natural motion.

Not A Sin, Just A Natural Motion

Sine waves[2] are probably the best thing ever for smooth animation. They happen in nature: witness a spring with a weight on it, ocean waves, sound and light. In our case, we want to move from 0 to 1 smoothly.

To create a movement that goes from 0 to 1 and back to 0 smoothly, we can use a sine wave that goes from 0 to π in a few steps. The full sine wave going from 0 to π × 2 (i.e. a whole circle) would result in values from -1 to 1, and we don't want that (yet).

```
var counter = 0;

// 100 iterations
var increase = Math.PI / 100;

for ( i = 0; i <= 1; i += 0.01 ) {
  x = i;
  y = Math.sin(counter);
  counter += increase;
}
```

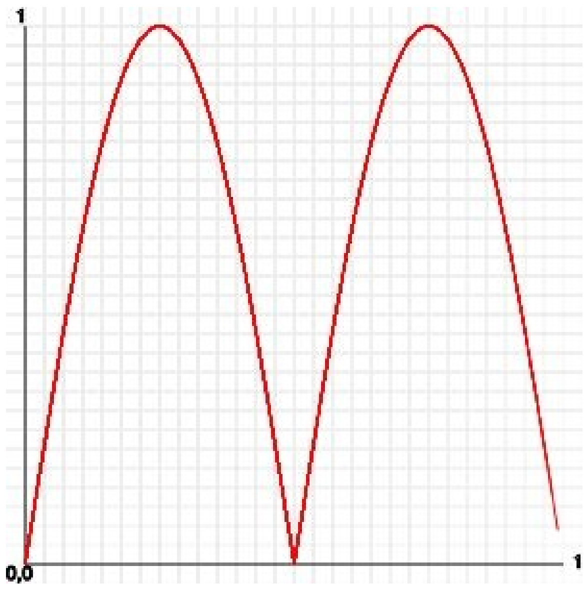


A quick aside on numbers for sine and cosine: Both `Math.sin()` and `Math.cos()` take as the parameter an angle that should be in radians[3]. As humans, however, degrees ranging from 0 to 360 are much easier to read. That's why you can and should convert between them with this simple formula:

```
var toRadian = Math.PI / 180;
var toDegree = 180 / Math.PI;

var angle = 30;

var angleInRadians = angle * toRadian;
var angleInDegrees = angleInRadians * toDegree;
```

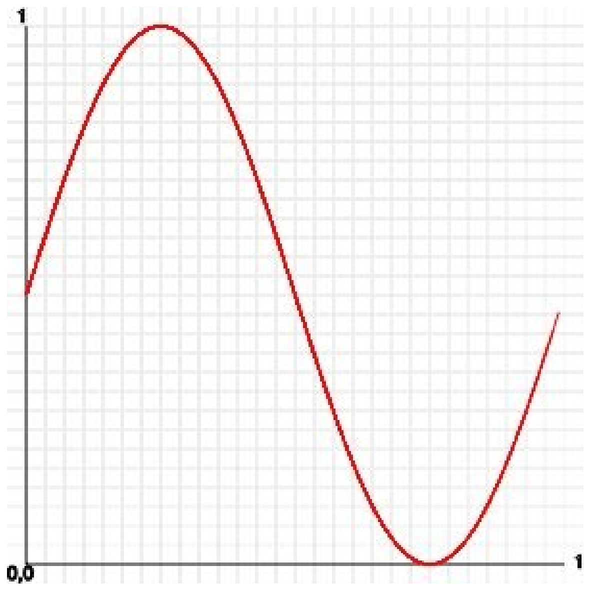
Back to our sine waves. You can play with this a lot. For example, you could use the absolute value of a full 2 × π loop:

```
var counter = 0;
// 100 iterations
var increase = Math.PI * 2 / 100;

for ( i = 0; i <= 1; i += 0.01 ) {
  x = i;
  y = Math.abs( Math.sin( counter ) );
  counter += increase;
}
```



But again, this looks dirty. If you want the full up and down, without a break in the middle, then you need to shift the values. You have to half the sine and then add 0.5 to it:

```
var counter = 0;
// 100 iterations
var increase = Math.PI * 2 / 100;

for ( i = 0; i <= 1; i += 0.01 ) {
  x = i;
  y = Math.sin( counter ) / 2 + 0.5;
  counter += increase;
}
```



So, how can you use this? Having a function that returns -1 to 1 to whatever you feed it can be very cool. All you need to do is multiply it by the values that you want and add an offset to avoid negative numbers.

For example, check out this sine movement demo[4].

Looks neat, doesn’t it? A lot of the trickery is already in the CSS:

```
.stage {
  width:200px;
  height:200px;
  margin:2em;
  position:relative;
  background:#6cf;
  overflow:hidden;
}
```

```
.stage div {
  line-height:40px;
  width:100%;
  text-align:center;
  background:#369;
  color:#fff;
  font-weight:bold;
  position:absolute;
}
```

The `stage` element has a fixed dimension and is positioned relative. This means that everything that is positioned absolutely inside it will be relative to the element itself.

The div inside the stage is 40 pixels high and positioned absolutely. Now, all we need to do is move the div with JavaScript in a sine wave:

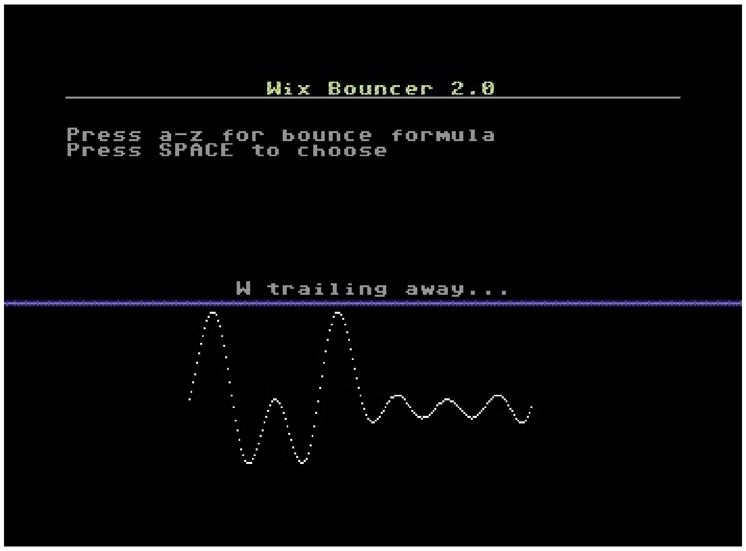
```
var banner = document.querySelector( '.stage div' ),
    start = 0;
function sine(){
  banner.style.top = 50 * Math.sin( start ) + 80 + 'px';
  start += 0.05;
}
window.setInterval( sine, 1000/30 );
```

The start value changes constantly, and with `Math.sin()` we get a nice wave movement. We multiply this by 50 to get a wider wave, and we add 80 pixels to center it in the stage element. Yes, the element is 200 pixels high and 100 is half of that, but because the banner is 40 pixels high, we need to subtract half of that to center it.

Right now, this is a simple up-and-down movement. Nothing stops you, though, from making it more interesting. The multiplying factor of 50, for example, could be a sine wave itself with a different value:

```
var banner = document.querySelector( '.stage div' ),
    start = 0,
    multiplier = 0;
function sine(){
  multiplier = 50 * Math.sin( start * 2 );
  banner.style.top = multiplier * Math.sin( start ) + 80 + 'px';
  start += 0.05;
}
window.setInterval( sine, 1000/30 );
```

The result of this[5] is a banner that seems to tentatively move up and down. Back in the day and on the very slow Commodore 64, calculating the sine wave live was too slow. Instead, we had tools to generate sine tables (arrays, if you will), and we plotted those directly. One of the tools for creating great sine waves so that you could have bouncing scroll texts was the Wix Bouncer:



Wix Bouncer 2.0

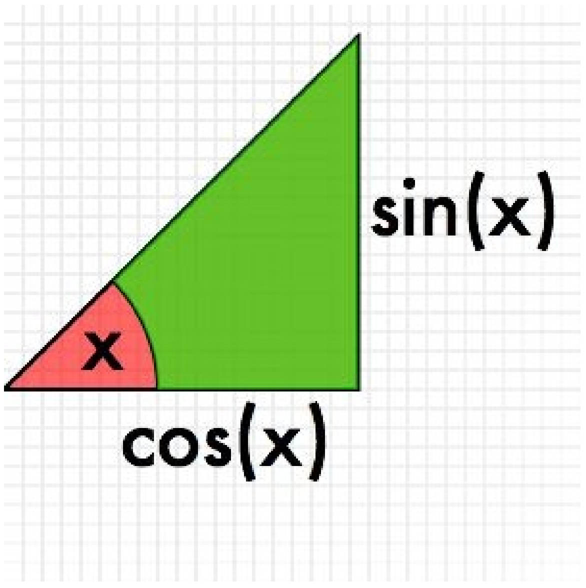
Press a-z for bounce formula
Press SPACE to choose

W trailing away...

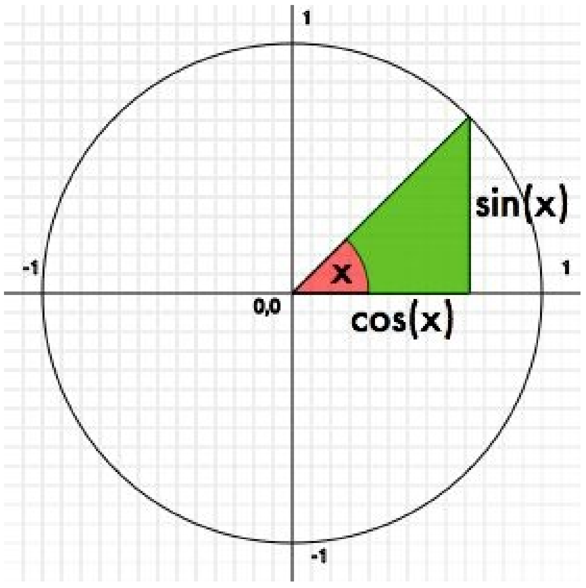
Circles In The Sand, Round And Round…

Circular motion is a thing of beauty. It pleases the eye, reminds us of spinning wheels and the earth we stand on, and in general has a “this is not computer stuff” feel to it. The math of plotting something on a circle is not hard.

It goes back to Pythagoras[6], who, as rumor has it, drew a lot of circles in the sand until he found his famous theorem[7]. If you want to use all the good stuff that comes from this theorem, then try to find a triangle with a right angle. If this triangle’s hypothenuse is 1, then you can easily calculate the horizontal leg as the cosine of the angle and the vertical leg as the sine of the angle:



How is this relevant to a circle? Well, it is pretty simple to find a right-angle triangle in a circle to every point of it:



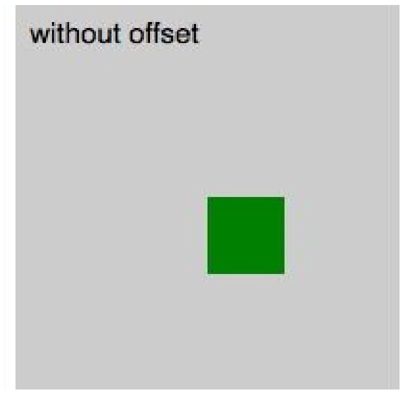
This means that if you want to plot something on a circle (or draw one), you can do it with a loop and sine and cosine. A full circle is 360°, or 2 × π in radians. We could have a go at it—but first, some plotting code needs to be done.

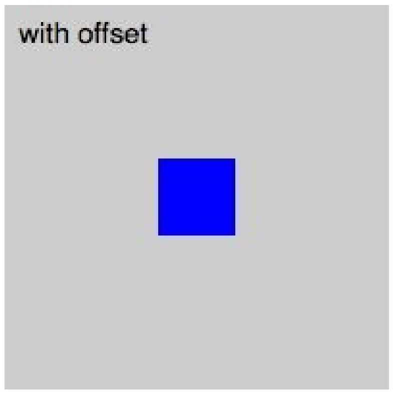
A Quick DOM Plotting Routine

Normally, my weapon of choice here would be canvas, but in order to play nice with older browsers, let's do it in plain DOM. The following helper function adds div elements to a stage element and allows us to position them, change their dimensions, set their color, change their content and rotate them without having to go through the annoying style settings on DOM elements.

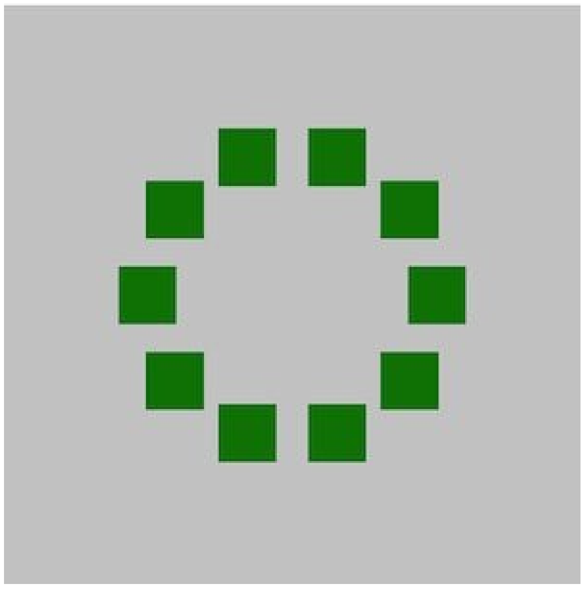
```
Plot = function ( stage ) {

  this.setDimensions = function( x, y ) {
    this.elm.style.width = x + 'px';
    this.elm.style.height = y + 'px';
    this.width = x;
    this.height = y;
  }
  this.position = function( x, y ) {
    var xoffset = arguments[2] ? 0 : this.width / 2;
    var yoffset = arguments[2] ? 0 : this.height / 2;
    this.elm.style.left = (x - xoffset) + 'px';
    this.elm.style.top = (y - yoffset) + 'px';
    this.x = x;
    this.y = y;
  };
  this.setbackground = function( col ) {
    this.elm.style.background = col;
  }
  this.kill = function() {
    stage.removeChild( this.elm );
  }
  this.rotate = function( str ) {
    this.elm.style.webkitTransform = this.elm.style.MozTransform =
    this.elm.style.OTransform = this.elm.style.transform =
    'rotate('+str+')';
  }
  this.content = function( content ) {
    this.elm.innerHTML = content;
  }
  this.round = function( round ) {
    this.elm.style.borderRadius = round ? '50%/50%' : '';
  }
  this.elm = document.createElement( 'div' );
  this.elm.style.position = 'absolute';
  stage.appendChild( this.elm );

};
```

The only things that might be new here are the transformation with different browser prefixes and the positioning. People often make the mistake of creating a div with the dimensions `w` and `h` and then set it to `x` and `y` on the screen. This means you will always have to deal with the offset of the height and width. By subtracting half the width and height before positioning the div, you really set it where you want it—regardless of the dimensions. Here’s a proof:





Now, let’s use that to plot 10 rectangles in a circle, shall we?

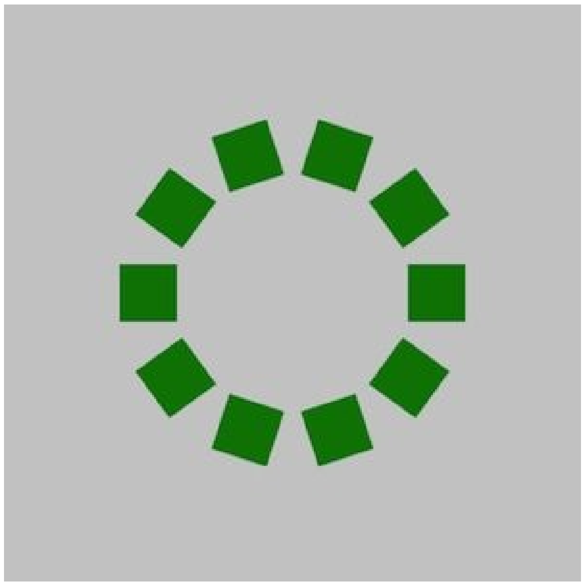


```
var stage = document.querySelector('.stage'),
    plots = 10;
    increase = Math.PI * 2 / plots,
    angle = 0,
    x = 0,
    y = 0;
```

```
for( var i = 0; i < plots; i++ ) {
  var p = new Plot( stage );
  p.setBackground( 'green' );
  p.setDimensions( 40, 40 );
  x = 100 * Math.cos( angle ) + 200;
  y = 100 * Math.sin( angle ) + 200;
  p.position( x, y );
  angle += increase;
}
```

We want 10 things in a circle, so we need to find the angle that we want to put them at. A full circle is two times `Math.PI`, so all we need to do is divide this. The x and y position of our rectangles can be calculated by the angle we want them at. The x is the cosine, and the y is the sine, as explained earlier in the bit on Pythagoras. All we need to do, then, is center the circle that we're painting in the stage (`200,200` is the center of the stage), and we are done. We've painted a circle with a radius of 100 pixels on the canvas in 10 steps.

The problem is that this looks terrible. If we really want to plot things on a circle, then their angles should also point to the center, right? For this, we need to calculate the tangent of the right-angle square, as explained in this charming "Math is fun" page[8]. In JavaScript, we can use `Math.atan2()`[9] as a shortcut. The result looks much better:



```
var stage = document.querySelector('.stage'),
    plots = 10;
    increase = Math.PI * 2 / plots,
    angle = 0,
    x = 0,
    y = 0;

for( var i = 0; i < plots; i++ ) {
  var p = new Plot( stage );
  p.setBackground( 'green' );
  p.setDimensions( 40, 40 );
  x = 100 * Math.cos( angle ) + 200;
  y = 100 * Math.sin( angle ) + 200;
  p.rotate( Math.atan2( y - 200, x - 200 ) + 'rad' );
  p.position( x, y );
  angle += increase;
}
```

Notice that the rotate transformation in CSS helps us heaps in this case. Otherwise, the math to rotate our rectangles would be much less fun. CSS transformations also take radians and degrees as their value. In this case, we use `rad`; if you want to rotate with degrees, simply use `deg` as the value.

How about animating the circle now? Well, the first thing to do is change the script a bit, because we don't want to have to keep creating new plots. Other than that, all we need to do to rotate the circle is to keep increasing the start angle[10]:

```
var stage = document.querySelector('.stage'),
    plots = 10;
    increase = Math.PI * 2 / plots,
    angle = 0,
    x = 0,
    y = 0,
    plotcache = [];

for( var i = 0; i < plots; i++ ) {
  var p = new Plot( stage );
  p.setBackground( 'green' );
  p.setDimensions( 40, 40 );
  plotcache.push( p );
}

function rotate(){
  for( var i = 0; i < plots; i++ ) {
    x = 100 * Math.cos( angle ) + 200;
    y = 100 * Math.sin( angle ) + 200;
    plotcache[ i ].rotate( Math.atan2( y - 200, x - 200 ) + 'rad'
);
    plotcache[ i ].position( x, y );
    angle += increase;
  }
  angle += 0.06;
}

setInterval( rotate, 1000/30 );
```

Want more? How about a rotating text message[11] based on this? The tricky thing about this is that we also need to turn the characters 90° on each iteration:

SMASHING MAGAZINE

```
var stage = document.querySelector('.stage'),
    message = 'Smashing Magazine '.toUpperCase(),
    plots = message.length;
    increase = Math.PI * 2 / plots,
    angle = -Math.PI,
    turnangle = 0,
    x = 0,
    y = 0,
    plotcache = [];

for( var i = 0; i < plots; i++ ) {
  var p = new Plot( stage );
  p.content( message.substr(i,1) );
  p.setDimensions( 40, 40 );
  plotcache.push( p );
}
function rotate(){
  for( var i = 0; i < plots; i++ ) {
    x = 100 * Math.cos( angle ) + 200;
    y = 100 * Math.sin( angle ) + 200;
    // rotation and rotating the text 90 degrees
    turnangle = Math.atan2( y - 200, x - 200 ) * 180 / Math.PI + 90
+ 'deg';
    plotcache[ i ].rotate( turnangle );
    plotcache[ i ].position( x, y );
    angle += increase;
  }
  angle += 0.06;
}

setInterval( rotate, 1000/40 );
```

Again, nothing here is fixed. You can make the radius of the circle change constantly[12], as we did with the bouncing banner message earlier (below is only an excerpt):

```
multiplier = 80 * Math.sin( angle );
for( var i = 0; i < plots; i++ ) {
  x = multiplier * Math.cos( angle ) + 200;
  y = multiplier * Math.sin( angle ) + 200;
  turnangle = Math.atan2( y - 200, x - 200 ) * 180 / Math.PI + 90 +
'deg';
  plotcache[ i ].rotate( turnangle );
  plotcache[ i ].position( x, y );
  angle += increase;
}
```

```
angle += 0.06;
```

And, of course, you can move the center of the circle[13], too:

```
rx = 50 * Math.cos( angle ) + 200;
ry = 50 * Math.sin( angle ) + 200;
for( var i = 0; i < plots; i++ ) {
  x = 100 * Math.cos( angle ) + rx;
  y = 100 * Math.sin( angle ) + ry;
  turnangle = Math.atan2( y - ry, x - rx ) * 180 / Math.PI + 90 +
'deg';
  plotcache[ i ].rotate( turnangle );
  plotcache[ i ].position( x, y );
  angle += increase;
}
angle += 0.06;
```

For a final tip, how about allowing only a certain range of coordinates[14]?

```
function rotate() {
  rx = 70 * Math.cos( angle ) + 200;
  ry = 70 * Math.sin( angle ) + 200;
  for( var i = 0; i < plots; i++ ) {
    x = 100 * Math.cos( angle ) + rx;
    y = 100 * Math.sin( angle ) + ry;
    x = contain( 70, 320, x );
    y = contain( 70, 320, y );
    turnangle = Math.atan2( y - ry, x - rx ) * 180 / Math.PI + 90 +
'deg';
    plotcache[ i ].rotate( turnangle );
    plotcache[ i ].position( x, y );
    angle += increase;
  }
  angle += 0.06;
}
function contain( min, max, value ) {
  return Math.min( max, Math.max( min, value ) );
}
```

SUMMARY

This was just a quick introduction to using exponentials and sine waves and to plotting things on a circle. Have a go with the code, and play with the numbers. It is amazing how many cool effects you can create with a few changes to the radius or by multiplying the angles. To make it easier for you to do this, below are the examples on JSFiddle to play with:

- Sine bouncing message[15]
- Double sine bouncing message[16]
- Offset issue with plotting[17]
- Distributing elements on a circle[18]

- Distributing elements on a circle with correct angles[19]
- Rotating a circle of boxes[20]
- Oscillating rotating message[21]
- Rotating message in a circle movement[22]
- Boxed rotated message scroller[23] ❧

—

1. http://coding.smashingmagazine.com/2011/09/14/the-guide-to-css-animation-principles-and-examples/
2. http://en.wikipedia.org/wiki/Sine_wave
3. http://en.wikipedia.org/wiki/Radian
4. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/sinejump.html

5. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/growingsinejump.html

6. http://en.wikipedia.org/wiki/Pythagoras

7. http://en.wikipedia.org/wiki/Pythagorean_theorem

8. http://www.mathsisfun.com/algebra/trig-finding-angle-right-triangle.html

9. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Math/atan2

10. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/rotatingcircle.html

11. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/rotatingmessage.html

12. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/growrotatingmessage.html

13. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/rotaterotatingmessage.html

14. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2011/09/boxedrotatingmessage.html

15. http://jsfiddle.net/codepo8/tgEx6/11/

16. http://jsfiddle.net/codepo8/tgEx6/2/

17. http://jsfiddle.net/codepo8/tgEx6/4/

18. http://jsfiddle.net/codepo8/tgEx6/8/

19. http://jsfiddle.net/codepo8/tgEx6/9/

20. http://jsfiddle.net/codepo8/tgEx6/7/

21. http://jsfiddle.net/codepo8/tgEx6/10/

22. http://jsfiddle.net/codepo8/tgEx6/5/

23. http://jsfiddle.net/codepo8/tgEx6/

Animating Without jQuery

BY JULIAN SHAPIRO ❧

There's a false belief in the web development community that CSS animation is the only performant way to animate on the web. This myth has coerced many developers to abandon JavaScript-based animation altogether, thereby (1) forcing themselves to manage complex UI interaction within style sheets, (2) locking themselves out of supporting Internet Explorer 8 and 9, and (3) forgoing the beautiful motion design physics that are possible only with JavaScript.

Reality check: JavaScript-based animation is often as fast as CSS-based animation — sometimes even faster. CSS animation only appears to have a leg up because it's typically compared to jQuery's `$.animate()`, which is, in fact, very slow. However, JavaScript animation libraries that bypass jQuery deliver incredible performance by avoiding DOM manipulation as much as possible. These libraries can be up to 20 times faster than jQuery.

So, let's smash some myths, dive into some real-world animation examples and improve our design skills in the process. If you love designing practical UI animations for your projects, this article is for you.

Why JavaScript?

CSS animations are convenient when you need to sprinkle property transitions into your style sheets. Plus, they deliver fantastic performance out of the box — without your having to add libraries to the page. However, when you use CSS transitions to power rich motion design (the kind you see in the latest versions of iOS and Android), they become too difficult to manage or their features simply fall short.

Ultimately, CSS animations limit you to what the specification provides. In JavaScript, by the very nature of any programming language, you have an infinite amount of logical control. JavaScript animation engines leverage this fact to provide novel features that let you pull off some very useful tricks:

- cross-browser SVG support[1],
- physics-based loader animations[2],
- timeline control[3],
- Bezier translations[4].

Note: If you're interested in learning more about performance, you can read Julian Shapiro's "CSS vs. JS Animation: Which Is Faster?[5]" and Jack Doyle's

"Myth Busting: CSS Animations vs. JavaScript[6]." For performance demos, refer to the performance pane[7] in Velocity's documentation and GSAP's "Library Speed Comparison[8]" demo.

Velocity and GSAP

The two most popular JavaScript animation libraries are Velocity.js[9] and GSAP[10]. They both work with and without[11] jQuery. When these libraries are used alongside jQuery, there is no performance degradation because they completely bypass jQuery's animation stack.

If jQuery is present on your page, you can use Velocity and GSAP just like you would jQuery's `$.animate()`. For example, `$element.animate({ opacity: 0.5 });` simply becomes `$element.velocity({ opacity: 0.5 })`.

These two libraries also work when jQuery is not present on the page. This means that instead of chaining an animation call onto a jQuery element object — as just shown — you would pass the target element(s) to the animation call:

```
/* Working without jQuery */
Velocity(element, { opacity: 0.5 }, 1000); // Velocity

TweenMax.to(element, 1, { opacity: 0.5 }); // GSAP
```

As shown, Velocity retains the same syntax as jQuery's `$.animate()`, even when it's used without jQuery; just shift all arguments rightward by one position to make room for passing in the targeted elements in the first position.

GSAP, in contrast, uses an object-oriented API design, as well as convenient static methods. So, you can get full control over animations.

In both cases, you're no longer animating a jQuery element object, but rather a raw DOM node. As a reminder, you access raw DOM nodes by using `document.getElementByID`, `document.getElementsByTagName`, `document.getElementsByClassName` or `document.querySelectorAll` (which works similarly to jQuery's selector engine). We'll briefly work with these functions in the next section.

Working Without jQuery

(*Note*: If you need a basic primer on working with jQuery's `$.animate()`, refer to the first few panes in Velocity's documentation.[12])

Let's explore `querySelectorAll` further because it will likely be your weapon of choice when selecting elements without jQuery:

```
document.querySelectorAll("body"); // Get the body element
document.querySelectorAll(".squares"); // Get all elements with the
"square" class
document.querySelectorAll("div"); // Get all divs
document.querySelectorAll("#main"); // Get the element with an id
of "main"
document.querySelectorAll("#main div"); // Get the divs contained
by "main"
```

As shown, you simply pass `querySelectorAll` a CSS selector (the same selectors you would use in your style sheets), and it will return all matched elements in an array. Hence, you can do this:

```
/* Get all div elements. */
var divs = document.querySelectorAll("div");

/* Animate all divs at once. */
Velocity(divs, { opacity: 0.5 }, 1000); // Velocity
TweenMax.to(divs, 1, { opacity: 0.5 }); // GSAP
```

Because we’re no longer attaching animations to jQuery element objects, you may be wondering how we can chain animations back to back, like this:

```
$element // jQuery element object
    .velocity({ opacity: 0.5 }, 1000)
    .velocity({ opacity: 1 }, 1000);
```

In Velocity, you simply call animations one after another:

```
/* These animations automatically chain onto one another. */
Velocity(element, { opacity: 0.5 }, 1000);
Velocity(element, { opacity: 1 }, 1000);
```

Animating this way has no performance drawback (as long as you cache the element being animated to a variable, instead of repeatedly doing `querySelectorAll` lookups for the same element).

(*Tip:* With Velocity's UI pack, you can create your own multi-call animations and give them custom names that you can later reference as Velocity's first argument. See Velocity's UI Pack documentation[13] for more information.)

This one-Velocity-call-at-a-time process has a huge benefit: If you're using promises[14] with your Velocity animations, then each Velocity call will return an actionable promise object. You can learn more about working with promises in Jake Archibald's article[15]. They're incredibly powerful.

In the case of GSAP, its expressive object-oriented API allows you to place your animations in a timeline, giving you control over scheduling and synchronization. You're not limited to one-after-the-other chained animations; you can nest timelines, make animations overlap, etc:

```
var tl = new TimelineMax();
/* GSAP tweens chain by default, but you can specify exact
insertion points in the timeline, including relative offsets. */
tl
```

```
  .to(element, 1, { opacity: 0.5 })
  .to(element, 1, { opacity: 1 });
```

JavaScript Awesomeness: Workflow

Animation is inherently an experimental process in which you need to play with timing and easings to get exactly the feel that your app needs. Of course, even once you think a design is perfect, a client will often request non-trivial changes. In these situations, a manageable workflow becomes critical.

While CSS transitions are impressively easy to sprinkle into a project for effects such as hovers, they become unmanageable when you attempt to sequence even moderately complex animations. That's why CSS provides keyframe animations, which allow you to group animation logic into sections.

However, a core deficiency of the keyframes API is that you must define sections in percentages, which is unintuitive. For example:

```
@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;
}
```

What happens if the client asks you to make the `translateX` animation 1 second longer? Yikes. That requires redoing the math and changing all (or most) of the percentages.

Velocity has its UI pack[16] to deal with multi-animation complexity, and GSAP offers nestable timelines[17]. These features allow for entirely new workflow possibilities.

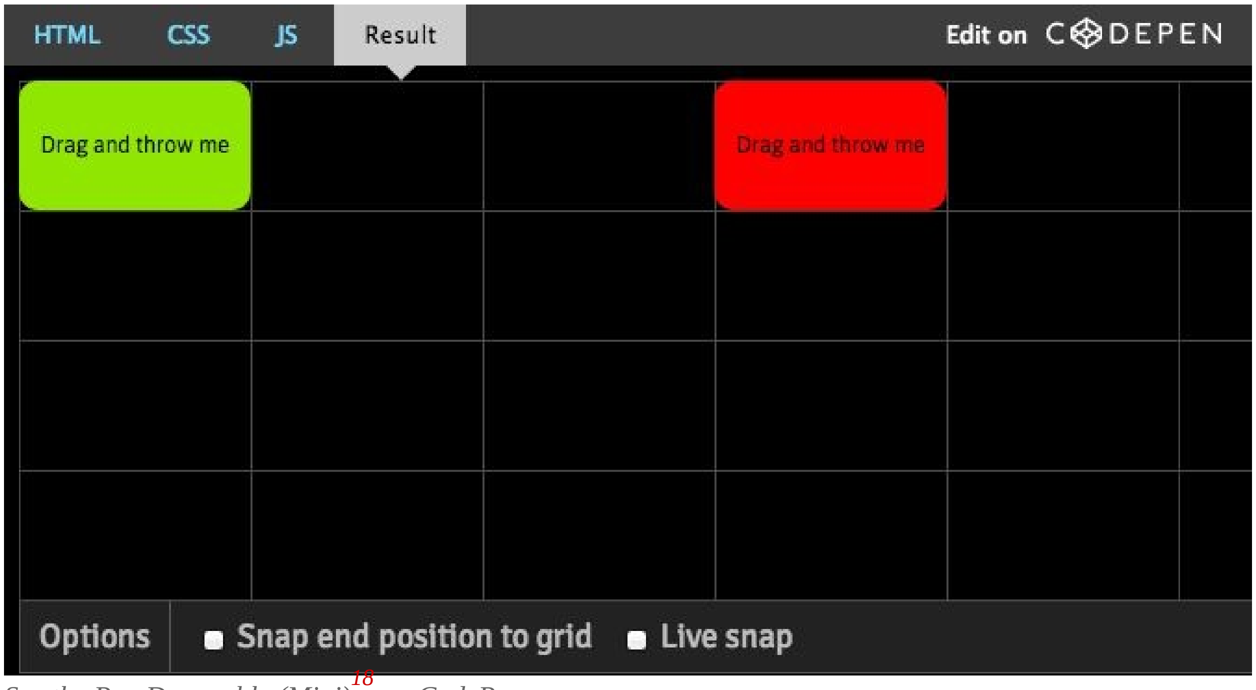
But let's stop preaching about workflow and actually dive into fun animation examples.

JavaScript Awesomeness: Physics

Many powerful effects are achievable exclusively via JavaScript. Let's examine a few, starting with physics-based animation.

The utility of physics in motion design hits upon the core principle of what makes for a great UX: interfaces that flow naturally from the user's input — in other words, interfaces that adhere to how motion works in the real world.

GSAP offers physics plugins that adapt to the constraints of your UI. For example, the ThrowPropsPlugin tracks the dynamic velocity of a user's finger or mouse, and when the user releases, ThrowPropsPlugin matches that corresponding velocity to naturally glide the element to a stop. The resulting animation is a standard tween that can be time-manipulated (paused, reversed, etc.):

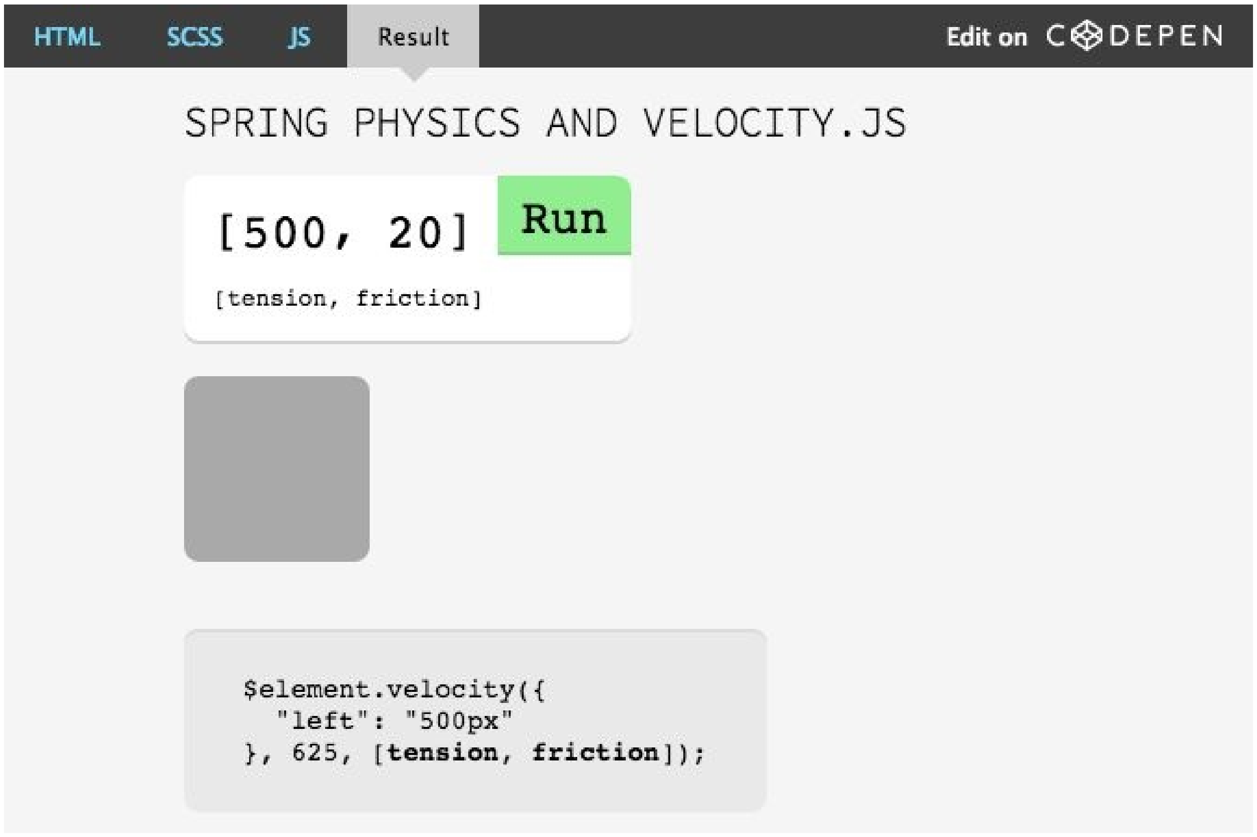


See the Pen Draggable (Mini)[18] *on CodePen.*

Velocity offers an easing type based on spring physics. Typically with easing options, you pass in a named easing type; for example, `ease`, `ease-in-out` or `easeInOutSine`. With spring physics, you pass a two-item array consisting of tension and friction values (in brackets below):

```
Velocity(element, { left: 500 }, [ 500, 20 ]); // 500 tension, 20
friction
```

A higher tension (a default of 500) increases the total speed and bounciness. A lower friction (a default of 20) increases ending vibration speed. By tweaking these values, you can separately fine-tune your animations to have different personalities. Try it out:



```
$element.velocity({
  "left": "500px"
}, 625, [tension, friction]);
```

See the Pen Velocity.js – Easing: Spring Physics (Tester)[19] *on CodePen.*

JavaScript Awesomeness: Scrolling

In Velocity, you can enable the user to scroll the browser to the edge of any element by passing in `scroll` as Velocity's first argument (instead of a properties map). The `scroll` command behaves identically to a standard Velocity call; it can take options and can be queued.

```
Velocity(element, "scroll", { duration: 1000 };
```

Refresh this page to re-run the demo.

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

Second element. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna

See the Pen Velocity.js – Command: Scroll w/ Container Option[20] *on CodePen.*

You can also scroll elements within containers, and you can scroll horizontally.

See Velocity’s scroll documentation[21] for further information.

GSAP has ScrollToPlugin[22], which offers similar functionality and can automatically relinquish control when the user interacts with the scroll bar.

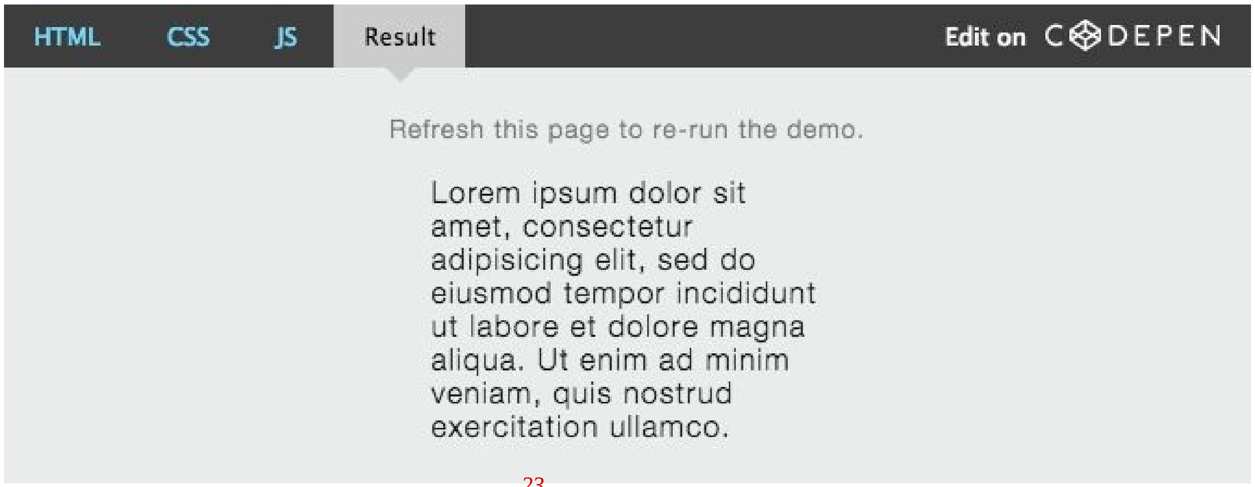
JavaScript Awesomeness: Reverse

Both Velocity and GSAP have reverse commands that enable you to animate an element back to the values prior to its last animation.

In Velocity, pass in `reverse` as Velocity’s first argument:

```
// Reverse defaults to the last call’s options, which you can
extend
Velocity(element, "reverse", { duration: 500 });
```

Click on the “JS” tab to see the code that powers this demo:



Refresh this page to re-run the demo.

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

See the Pen Velocity.js – Command: Reverse[23] *on CodePen.*

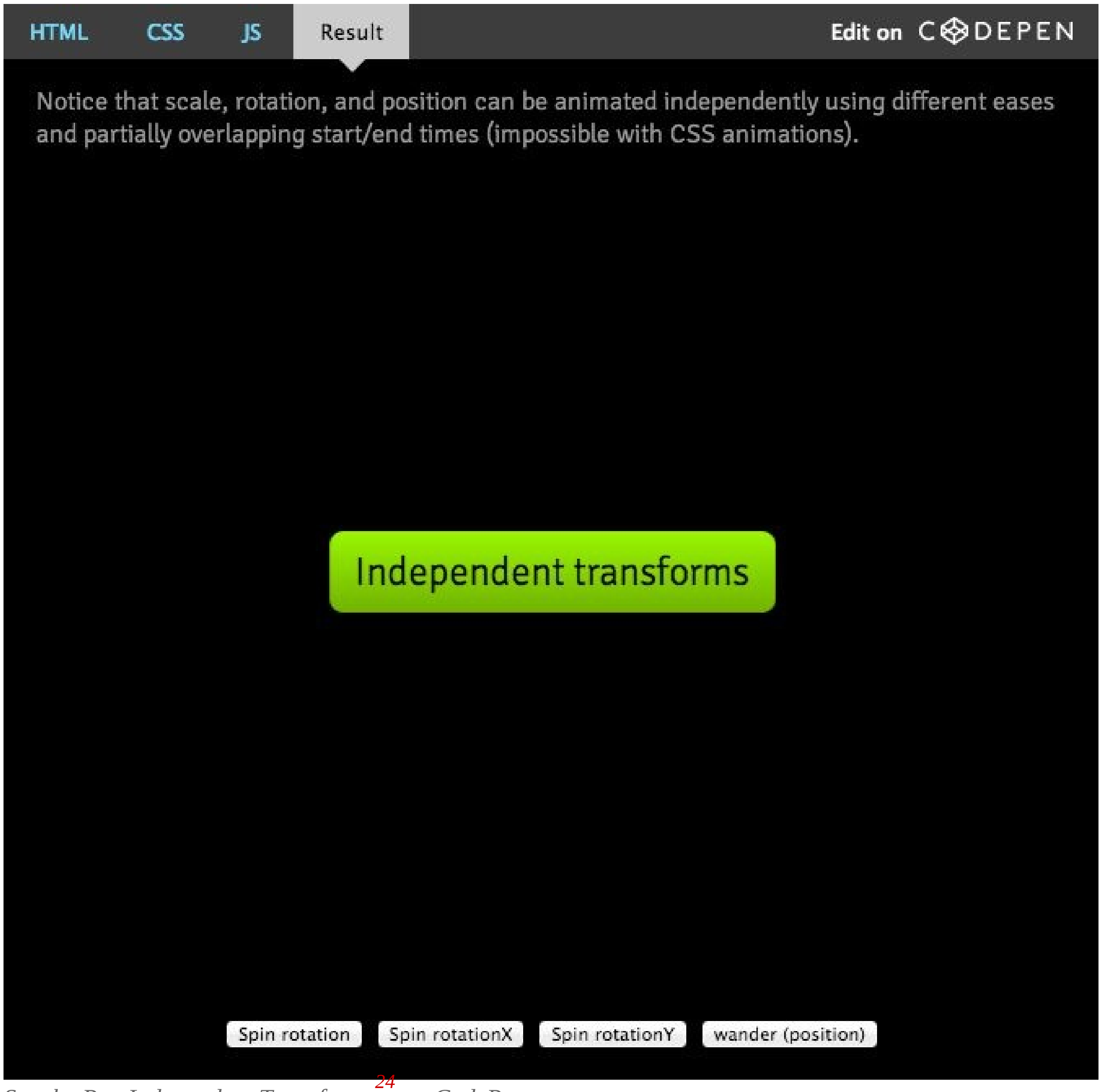
In GSAP, you can retain a reference to the animation object, then invoke its `reverse()` method at any time:

```
var tween = TweenMax.to(element, 1, {opacity:0.5});
tween.reverse();
```

JavaScript Awesomeness: Transform Control

With CSS animation, all transform components — scale, translation, rotation and skew — are contained in a single CSS property and, consequently, cannot be animated independently using different durations, easings and start times.

For rich motion design, however, independent control is imperative. Let’s look at the dynamic transform control that’s achievable only in JavaScript. Click the buttons at any point during the animation:


HTML
CSS
JS
Result
Edit on CODEPEN
Notice that scale, rotation, and position can be animated independently using different eases and partially overlapping start/end times (impossible with CSS animations).
Independent transforms
Spin rotation
Spin rotationX
Spin rotationY
wander (position)

See the Pen Independent Transforms[24] *on CodePen.*

Both Velocity and GSAP allow you to individually animate transform components:

```
// Velocity
```

```
/* First animation */
Velocity(element, { translateX: 500 }, 1000);
/* Trigger a second (concurrent) animation after 500 ms */
Velocity(element, { rotateZ: 45 }, { delay: 500, duration: 2000,
queue: false });

// GSAP
/* First animation */
TweenMax.to(element, 1, { x: 500 });
/* Trigger a second (concurrent) animation after 500 ms */
TweenMax.to(element, 2, { rotation: 45, delay: 0.5 });
```

Wrapping Up

- Compared to CSS animation, JavaScript animation has better browser support and typically more features, and it provides a more manageable workflow for animation sequences.

- Animating in JavaScript doesn’t entail sacrificing speed (or hardware acceleration). Both Velocity and GSAP deliver blistering speed and hardware acceleration under the hood. No more messing around with `null-transform` hacks.

- You don’t need to use jQuery to take advantage of dedicated JavaScript animation libraries. However, if you do, you will not lose out on performance.

FINAL NOTE

Refer to Velocity[25] and GSAP's documentation[26] to master JavaScript animation. ❧

—

1. http://codepen.io/sol0mka/full/jpecs/
2. http://codepen.io/timothyrourke/full/wojke/
3. http://codepen.io/GreenSock/full/yhEmn/
4. http://codepen.io/GreenSock/full/LuIJj/
5. http://davidwalsh.name/css-js-animation
6. http://css-tricks.com/myth-busting-css-animations-vs-javascript/
7. http://velocityjs.org
8. http://codepen.io/GreenSock/full/srfxA/

9. http://velocityjs.org
10. http://greensock.com/gsap/
11. http://velocityjs.org/#dependencies
12. http://velocityjs.org/#arguments
13. http://velocityjs.org/#uiPack
14. http://velocityjs.org/#promises
15. http://www.html5rocks.com/en/tutorials/es6/promises/
16. http://velocityjs.org/#uiPack
17. http://greensock.com/sequence-video
18. http://codepen.io/GreenSock/pen/f5005e85b22ae5b7d5b1075c488cedde/
19. http://codepen.io/julianshapiro/pen/hyeDg/

20. http://codepen.io/julianshapiro/pen/kBuEi/

21. http://velocityjs.org/#scroll

22. http://greensock.com/docs/#/HTML5/GSAP/Plugins/ScrollToPlugin/

23. http://codepen.io/julianshapiro/pen/hBFbc/

24. http://codepen.io/GreenSock/pen/kingu/

25. http://velocityjs.org

26. http://greensock.com/docs/#/HTML5/GSAP/

Faster UI Animations With Velocity.js

BY JULIAN SHAPIRO ❧

From a motion design perspective, Facebook.com is phenomenally static. It’s purposefully dumbed down for the broadest levels of compatibility and user comfort. Facebook’s iOS apps, on the other hand, are fluid. They prioritize the design of motion; they feel like living, breathing apps.

This article serves to demonstrate that this dichotomy does not need to exist; websites can benefit from the same level of interactive and performant motion design found on mobile apps.

Before diving into examples, let’s first address why motion design is so beneficial:

- **Improved feedback loops**
  As a UI and UX designer, you should use patterns as much as possible since users will be subconsciously looking for them. Responsive motion patterns, in particular, are the key to pleasurable interactions: When a button has been clicked, do you feel that it has reacted to the pressure of your mouse? When a file has been saved, do you get the strong sense that your data has truly been transferred and stored?

- **Seamless content transitions**
  Motion design allows you to avoid contextual breaks; modals fading in and out (as opposed to switching pages entirely) are a popular example of this.

- **Filled dead spots**
  When users are performing an unengaging task on your page, you can raise their level of arousal through sound, colors, and movement. Diverting a user's attention is a great way to make a pot boil faster.

- **Aesthetic flourishes**
  For the same aesthetic reasons that UI designers spend hours perfecting their pages' color and font combinations, motion designers perfect their animations' transition and easing combinations: Refined products simply feel superior.

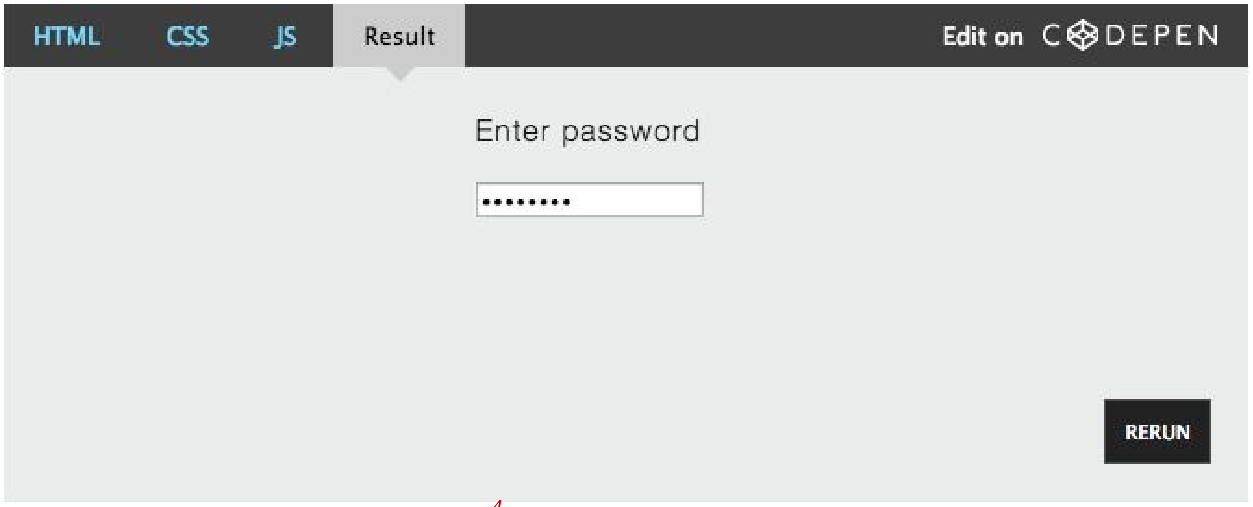
In the examples below, we'll be using Velocity.js[1] — a popular animation engine that drastically improves the speed of UI animation. (Velocity.js behaves identically to jQuery's `$.animate()` function, while outperforming both jQuery animation and CSS animation libraries.) In particular, this article focuses on Velocity.js' UI pack[2], which allows you to quickly inject motion design into your pages. You may optionally watch this article's accompanying codecast[3] (5 minutes) for a preview of what we'll cover.

UI Pack Overview

After including the UI pack (only 1.8 KB ZIP'ed) on your page, you'll gain access to UI effects that are organized into two categories:

CALLOUTS

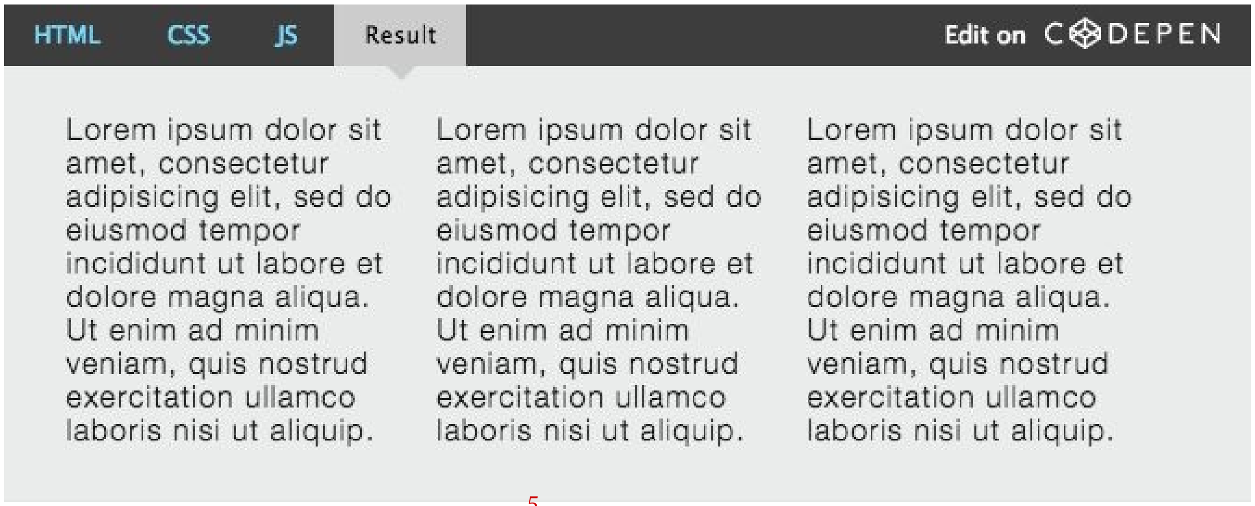
Callouts are effects that call attention to an element in order to heighten user experience, such as shaking an element to indicate an input error, flashing an element to indicate that something has changed on the page, or bouncing an element to indicate that a message awaits the user.



See the Pen Velocity.js – UI Pack: Callout[4] *on CodePen.*

TRANSITIONS

Transitions are effects that cause an element to appear in or out of view. Every transition is associated with either an “in” or “out” direction. The value of transitions is in revealing and hiding content in a way that’s visually richer than merely animating an element’s opacity. Here’s `slideUpIn`, a transition that incorporates a subtle slide effect:



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

See the Pen Velocity.js – UI Pack: Transition[5] *on CodePen.*

If you’ve paid attention to the evolution of iOS’ UI motion design, you’ll have noticed that over a dozen transition effects help make iOS’ interface pleasurable to interact with. This diversity of transitions is what Velocity.js’ UI pack brings to everyday websites.

Note that, thanks to Velocity.js’ performance, as well as the optimizations afforded by the UI pack, all of the pack’s effects are 100% ready for large-scale production use.

Let’s dive into some simple code examples.

Let's dive into some simple code examples.

Using The UI Pack

Callouts and transitions are referenced via Velocity's first parameter: Pass in an effect's name instead of passing in a standard property map. For comparison, here's the syntax of a *normal* Velocity.js call, which behaves identically to jQuery's `$.animate()`:

```
$elements.velocity({ opacity: 0.5 });
```

In contrast, below are Velocity.js calls using effects from the UI pack:

```
/* Shake an element. */
$elements.velocity("callout.shake");

/* Transition an element into view using slideUp. */
$elements.velocity("transition.slideUpIn");
```

Just as with normal Velocity.js calls, UI effects may be chained onto each other and may take options:

```
/* Call the shake effect with a 2000ms duration, then slide the
elements out of view. */
$elements
```

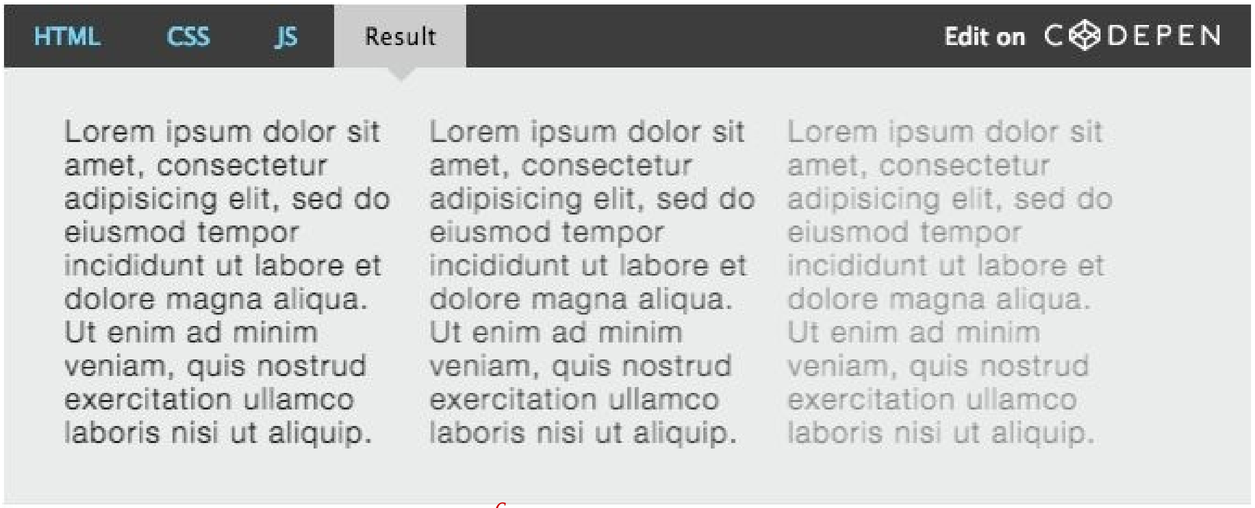
```
    .velocity("callout.shake", 2000)
    .velocity("transition.slideDownOut");
```

Effects from the UI pack optionally take three unique options: `stagger`, `drag` and `backwards`.

STAGGER

Specify `stagger` in milliseconds to successively delay the animation of each element in a set by the specified amount. (Setting a `stagger` value prevents elements from animating in parallel, which tends to lack elegance.)

```
/* Animate elements into view with intermittent delays of 250ms. */
$divs.velocity("transition.slideLeftIn", { stagger: 250 });
```



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

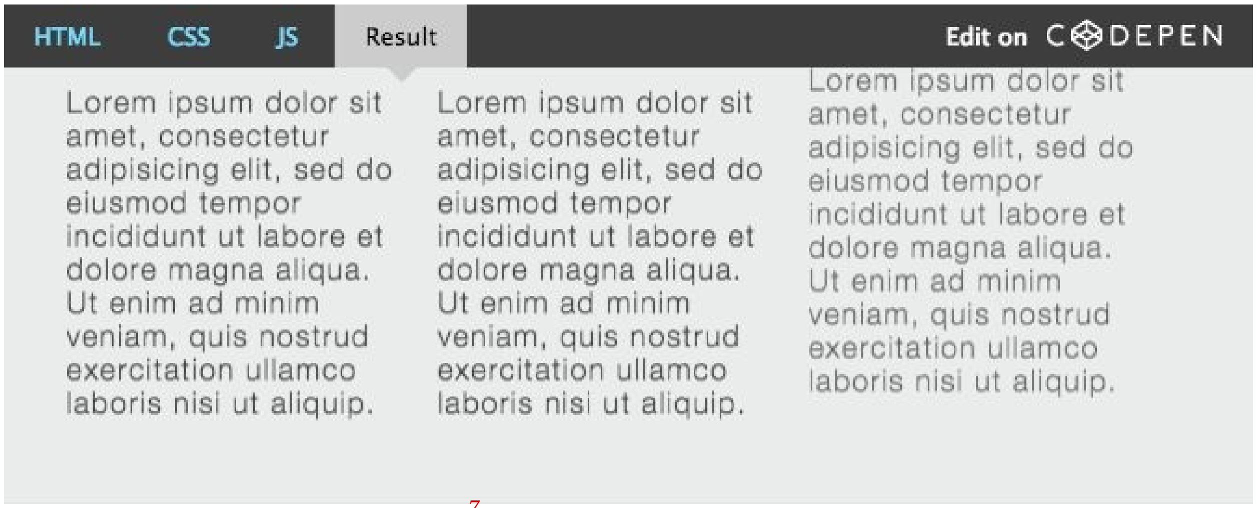
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

See the Pen Velocity.js – UI Pack: Stagger[6] *on CodePen.*

DRAG

Set `drag` to `true` to successively increase the animation duration of each element in a set. The last element will animate with a duration equal to the animation's original value, whereas the elements prior to the last will have their duration values gradually approach the original value.

The result is a cross-element easing effect. (If you've ever been wowed by motion typography demos made with After Effects, drag is a key yet subtle component behind their visual richness.)



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

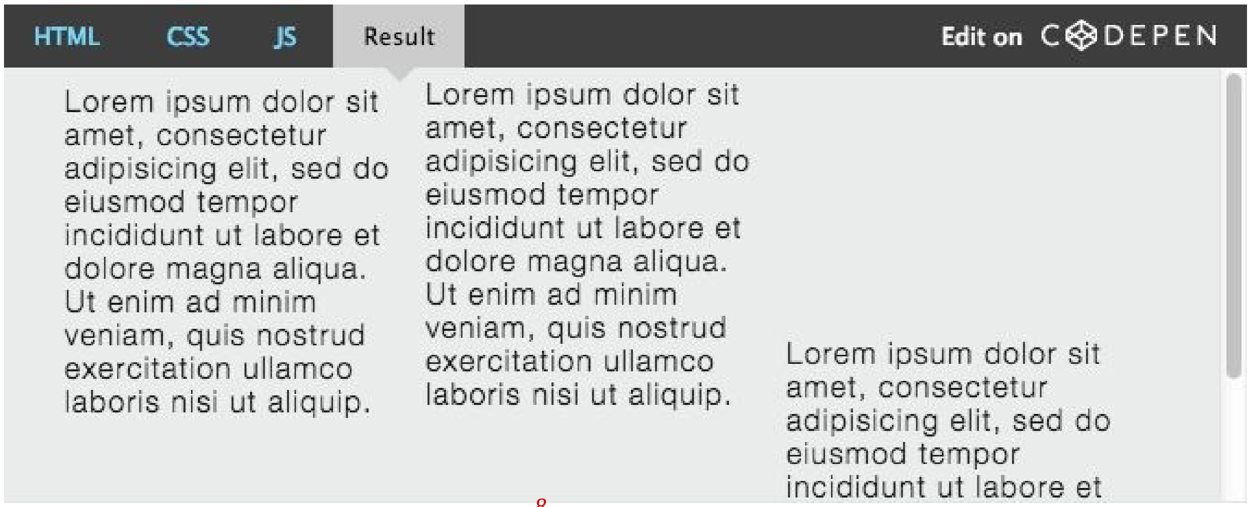
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

See the Pen Velocity.js – UI Pack: Drag[7] *on CodePen.*

BACKWARDS

Set `backwards` to `true` to animate starting with the last element in a set. This option is ideal for an effect that transitions elements out of view, because the `backwards` option mirrors the behavior of elements transitioning into view (which, by default, animate in a forward direction, from the first element to the last).



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et

See the Pen Velocity.js – UI Pack: Backwards[8] *on CodePen.*

Together, these three options bring the power of motion design suites to the Web. When used sparingly, the results are beautiful — so long as you design with user experience in mind:

Designing For UX

Spicing up a page with motion design can escalate quickly[9]. Here are a few considerations to keep in mind:

- **Make them finish quickly**
  When applying transitions, developers often make the mistake of letting them run too long, causing users to wait needlessly. Never let UI flourishes slow down the apparent speed of your page. If you have a lot of content fading in, keep the animation's total duration short.

- **Use an appropriate effect**
  For example, don't use a playful bounce effect on a page that features formal content.

- **Use them sparingly**
  Having transitions in every corner of your page is overkill.

- **Avoid extreme repetition**
  Avoid transitions of medium-to-long duration in places where they'll be repeatedly triggered.

- **Experiment**
  Find the right duration, stagger, drag and backwards combinations that will produce the right fit for each of your individual animations.

Benefits Of JavaScript-Based Animation

Let’s step back and contextualize why powering these types of UI transitions through JavaScript is a good idea in the first place. After all, up until now, these effects have been most commonly applied using pre-made CSS classes from libraries such as Animate.css[10]:

- Because the UI pack’s effects behave identically to standard animation calls in Velocity.js, they can be chained and take options. (Doing this with raw CSS can be cumbersome.)

- The effects have all been optimized for performance (minimal DOM interaction).

- Elements animated via the UI pack automatically switch to `display: none` after transitioning out, and back to `display: block` or `inline` before transitioning in. (Doing this via CSS requires multiple calls and messy code.)

- Velocity.js doesn’t leave your text blurry. If you’ve applied transition effects via CSS before, then you’ll know that text can look fuzzy when you’re done animating and haven’t removed the associated class. This doesn’t happen in Velocity.js because its underlying engine completely clears unneeded transformation effects upon completion of an animation.

- The effects work everywhere but Internet Explorer 8, where they gracefully fall back to simply fading in and out.

With all of these benefits, including the effects' great performance across all browsers and devices (including older mobile devices), you have no excuse to not start experimenting with motion design on your sites. Enjoy!

Note: Look through Velocity.js' documentation[11] to play around with all of the UI pack's effects.

LINKS

- Download Velocity on GitHub[12]
- Velocity demo gallery[13] ❧

—

1. http://velocityjs.org

2. http://velocityjs.org/#uiPack
3. https://www.youtube.com/watch?v=CdwvR6a39Tg&hd=1
4. http://codepen.io/julianshapiro/pen/Fybjq/
5. http://codepen.io/julianshapiro/pen/aLhFC/
6. http://codepen.io/julianshapiro/pen/mqsnk/
7. http://codepen.io/julianshapiro/pen/lxfie/
8. http://codepen.io/julianshapiro/pen/fEKsw/
9. https://www.youtube.com/watch?v=FONN-0uoTHI
10. https://github.com/daneden/animate.css
11. http://velocityjs.org/#uiPack
12. https://github.com/julianshapiro/velocity

13. http://codepen.io/collection/tIjGb/

Using Motion For User Experience On Apps And Websites

BY DREW THOMAS ❧

Digital experiences are emulating real life more and more every day. This may seem counterintuitive, considering the hate that rains down on skeuomorphic visual design, but there's a lot more to emulating real life than aesthetics. Interface designers can emulate real-life physics and movement on a digital screen. This type of motion is becoming more common, which is why it's becoming easier for people to understand computers. We're not getting better, the interfaces are!

A quick and common example is how iOS opens and closes apps. The transitions are very subtle, but they're realistic. Tapping an app icon doesn't just snap a new app on to the screen. Instead, users see the app physically grow out of the icon. In reverse, pressing the home key shrinks the app back into the icon.

Those interactions are based on properties of the real world. The app appears to come from somewhere physical and disappear back to that place. The high quality and realistic transitions here go a long way toward helping the user understand what's happening and why.

Opening an iOS app without a transition[1] *vs. with the transition.*[2]

In this article, I'll cover a little bit of the history of motion on the web, why that's important, and what the future of motion on the web will look like. (Hint: motion is really important for usability, and it's changing everything.) Then I'll explain the CSS behind motion and how to use motion well.

The History Of Motion On The Web

It was only 2011 when all major browsers officially recognized CSS animation, and even now it requires browser prefixes to work everywhere. In large part, the push for CSS-driven animation was sparked by the death of Flash, where "movement was common" is an understatement.

In the days of Flash, some websites were basically movies. There was a lot of movement and animation, but most of it was unnecessary to navigate and absorb the content. It was for wow effect at best.

Flash was eventually forced out of the picture, but designers and developers were left without any really good tools for movement and animation on the web.

JavaScript and jQuery became really popular, and they were huge leaps forward, but there are all kinds of reasons not to rely on JavaScript for your site to function. Plus, JavaScript animation was, and in some ways still is, taxing for browsers. Some motion was possible, but it needed to be used sparingly.

It wasn’t long before the CSS3 animation and transitions specs were accepted and implemented by modern browsers.

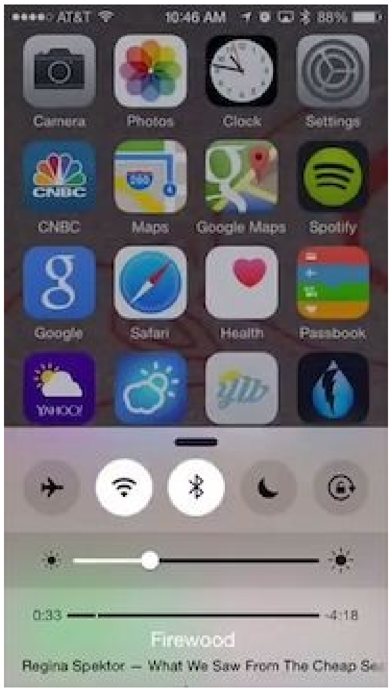
Designers now have the ability to take advantage of hardware acceleration and can control movement with their style sheets, further separating content and visual markup. In addition, today’s average computers are more than capable of rendering complex animations, and even phones are powerful enough to process an impressive amount of movement.

The Future Of Motion On The Web

The combination of capable machines and evolving CSS specs means things are going to change in interface design. Websites and apps are going to start taking advantage of motion and what it can do for usability. It’s already happening in a lot of ways, but here are some examples to look out for.

LAYERS

Layers are everywhere in modern web and app interfaces. Apple really pushed the concept of layers with iOS7. An example is the Control Center, which slides up from the bottom as a new layer that partially covers whatever’s on the screen.



The iOS Control Center slides in over the current screen as a new layer. (View a video of the animation[3]*)*

Although layers aren’t movement in themselves, they go hand in hand because they work best when they animate in and animate out.

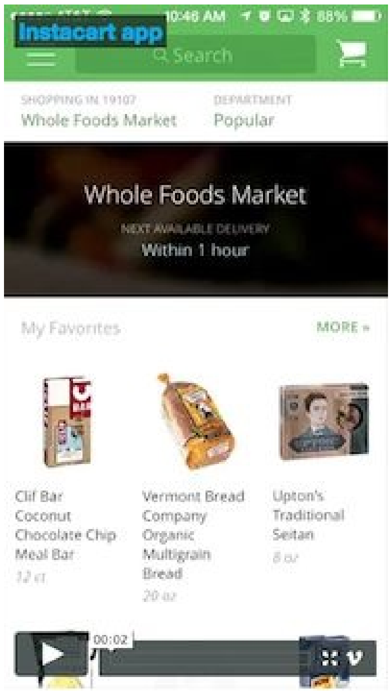
Layers are important because designers can keep information hidden on another layer until it’s called on, instead of refreshing the entire page to display large amounts of new information. This allows users to think less and understand more. It gives them context, which is the next thing you’ll start to see a lot of with motion.

CONTEXT

Context is a broad term. For this discussion, I use it to refer to elements and pages that don’t just snap from one state to another without showing where they came from and why. Context helps us remove the digital mystery and therefore it helps users’ brains focus less on interpreting the interface and more on the content and their goals.

To illustrate how transitions can convey context, take a look at the Instacart iOS app. Tapping on an item to see more detail about it doesn’t just snap open a new view with the item details.

While that would likely be understood by most users, take a look at the image below to see what happens instead. We see the item’s picture move from its current position to a new position above the details view. We completely understand what happened and how it relates to the previous view. In fact, this doesn't even feel like we're switching from one view to another. This seems much more natural than that.



The transition into the detail view in the Instacart app helps to give the user context. (View a video of the transition[4])

The effect is subtle, but it has huge usability implications. Another example is the newly popular drawer menu, where clicking a hamburger icon reveals a full menu.

If the user taps the icon and their entire screen is instantly replaced by the menu, they have no context as to where that menu came from and why. It won't completely derail anyone, but it's not a good user experience.

All it needs is to slide in from the left and suddenly the user has context for what’s happening: “Oh, the menu was just sitting offscreen, waiting to be called.”

You can see a drawer menu example in almost every popular app these days and on most mobile versions of websites. The GMail and Facebook apps are both excellent examples of this concept.

THE SINGLE PAGE APPLICATION

The next trend we’ll see are single page applications (SPAs). As we add motion and transitions to parts of our user interfaces, we’ll start to want more control of the interface as a whole (not the interface within each page). Websites can now handle all kinds of transitions from state to state within a page, but what about the transition from page to page? Any small gap when the screen goes white or shifts content around hurts usability.

That explains the rising popularity of the single page application. Right now, there are a lot of popular frameworks to build SPAs, and they’re more like native mobile applications than webpages (at least in some ways).

The sign-in and sign-up process for Dance It Yourself (an SPA I’m currently building) illustrates this well. If you go to http://app.danceityourself.com[5], you’ll see the page initially loads like a normal website, but if you tap the Sign Up button, instead of refreshing the page, the content either slides up from the bottom (on smaller screens) or in from the left (larger screens). The technique uses JavaScript to add a class to the Sign Up page, which triggers a CSS transition.

The result is a smooth, fast and logical transition from one screen to another. Once you sign in to the app, the entire experience is treated the same way. All the movement and transitions are driven by logic and context, and they make this web application feel more like a native application than a website.

How To Do CSS Motion

Single page applications present a good opportunity to take advantage of CSS motion, but there are plenty of other places to use it, including potentially every element on every website you make from now on. But how do we actually do it? What does the CSS look like?

To understand the basics of CSS motion, it’s important to start simple. What follows are explanations with examples, but they’re definitely *minimum viable examples*. Follow some of the links to learn much more about the in-depth aspects of each type of CSS motion.

CSS TRANSITIONS

There are many times when a little transition can go a long way. Instead of changing properties of an element in a split second, a transition gives the user some real context and a visual clue as to what's happening and why.

This helps usability because it removes the mystery behind digital state change. In real life, based on physics, there is always a transition from any one thing to another. The human brain understands this, so it's important to translate that visual information into our interfaces.

To start explaining CSS transitions, let's first look at a state change without any transition.

```css
button {
   margin-left:0;
}

button:hover {
   margin-left:10px;
}
```

When the user hovers over the button, it jumps 10 pixels to the right. Check out the demo to see it in action[6].

Now let's add the most basic form of a transition. I've left out browser prefixes, but they're in the demos, because we still need to use them in production code.

but they’re in the demos, because we still need to use them in production code.

```
button {
   margin-left:0;
   transition: margin-left 1s;
}

button:hover {
   margin-left:10px;
}
```

That code will animate the `margin-left` CSS property when a user hovers over the button. It will animate from 0 to 10px in 1 second.

Here’s a demo for that[7]. Notice how unnatural it looks, though.

Next, we’ll make the motion look a little more realistic with just a small adjustment.

```
button {
   margin-left:0;
   transition: margin-left .25s ease-out;
}

button:hover {
   margin-left:10px;
}
```

Here’s that demo[8]. This example looks nice and natural. There’s probably little reason to animate the `margin-left` property of a button. You can imagine how this can apply to many different circumstances.

The last important thing to know about CSS transitions (and CSS animation for that matter), is that we can’t animate every CSS property. As time goes on, we’ll be able to animate more and more, but for now, we need to stick to a select few. Here’s a list of all the properties that will animate using the CSS `transition` property[9].

When we talk about the hover state, it’s easy to see how CSS transitions apply, but also consider triggering transitions by adding an additional class to an element. This trick will come in handy. How the class gets added has to do with your implementation, but any time a class is added or removed, it will trigger the CSS transition.

```css
button {
   margin-left:0;
   transition: margin-left .25s ease-out;
}

button.moveRight {
   margin-left:10px;
}
```

CSS ANIMATIONS

The basic CSS for an animation is a little more complicated, but it’s similar to

CSS transitions in a lot of ways.

The reasons to use CSS animations are also similar to transitions, but there are some different applications. We want to emulate real life as much as possible so that human brains can do less work to understand what’s going on. Unlike transitions, however, animations can be looped and can move independently of user input. Therefore, we can use animation to draw attention to elements on a page. Or we can add subtle movement to illustrations or background elements to give our interfaces some life.

Animation benefits may seem less tangible, but they’re equally as important. It pays to add some fun to our interfaces. Users should love to use our products, and animation can have a big impact on the overall user experience.

Here’s a shorthand example of a CSS animation. We use a block of CSS keyframes and give it a name, and we assign that keyframe animation to an element. Again, since browser prefixes add a lot of code, I didn’t include them. I did, however, include them in the demo, because, unfortunately, we still need to include all browser prefixes in the real world.

```
div.circle {
   background:#000;
   border-radius:50%;
   animation:circleGrow 800ms ease-in-out infinite alternate both;
}

@keyframes circleGrow {
    0% {
        height:2px;
        width:2px;
    }
    50% {
        height:40px;
        width:40px;
    }
    100% {
        height:34px;
        width:34px;
    }
}
```

Here’s the animation demo[10].

To break it down, there are really only two things going on here.

First, there’s the `animation` property itself. It’s very much like the `transition` property, but it has a lot more that we can control. I used the shorthand version in my example, but just like the `transition` property, each part can be controlled as a separate CSS property (you probably do this with `background` all the time).

The shorthand `animation` property breaks down like this:

animation: [animation name (from keyframe block)] [duration] [timing function] [delay] [number of times the animation repeats] [animation direction] [fill mode]

→ Here's a more thorough explanation of all the different CSS animation properties[11].

The second thing going on is the keyframes block. At a very basic level, this is self explanatory. Set any number of percentages from 0–100 to represent how far through the animation we are from start (0%) to finish (100%). Then add any styles for that stage of the animation. When the animation runs, all styles will animate between the values you specify at each percentage number.

Again, not all properties animate, but as time goes on, we'll be able to do more and more.

How To Do CSS Motion Well

Now that you know how to write the CSS for motion, it's time to think about using motion well. All of the concepts here will fail if executed improperly. Transition and animation need to feel real. If they don't, they'll be surprisingly

Transition and animation need to feel real. If they don't, they'll be surprisingly distracting, and the distraction will actually hurt usability.

The trick to making motion look natural is two-fold: easing and object weight.

EASING

You may have noticed the easing part in the code examples. In real life, objects start moving gradually and slow to a stop. Things don't just start moving at 100% speed. That's where the third property for the transition style comes in from the examples: `ease-out` or `ease-in`. Sometimes, your best bet is actually `ease-in-out` (here's a list of all the possible easing (timing) functions[12]).

WEIGHT

Weight, on the other hand, is not a specific property of the transition or animation style. Weight mainly affects motion speed, and the basic concept is that smaller objects would have less physical weight in real life, so they'd move faster than larger objects. That's why we increased the transition speed on the button from the second to the third example above. A small button seems really slow when it takes 1 second to move 10 pixels. A quarter of a second seems much more natural. (You can also use milliseconds, as in the example below.)

```
transition: margin-left 250ms ease-out;
```

A Tip If You’re Just Getting Started

This all may seem like a lot. If you’re new to CSS transitions and animation, I’d recommend one important thing. Build in steps. If you write an entire, complex keyframes block in one shot and then add timing, easing and looping into the animation property, you’ll find out very quickly that you’re confused. It will be hard to tweak and edit that animation. Start simple, and build the animation up by testing and iterating.

Coming Full Circle

When you’re up and running and using CSS motion, you’ll start to notice all kinds of different uses for these techniques. In most cases, it’s much more than a bell and whistle or a superfluous add-on. Movement is a tool, and it conveys context, meaning, importance and more. It can be just as important as any other usability technique that we use today.

As interface designers take advantage of motion, and as interfaces start to behave more like objects and environments in the real world, usability and user experience will improve as well. Humans will have to think less about computer interfaces and therefore the interfaces will be easier to learn and easier to use. Users may feel like they’re getting smarter or more tech savvy, but really, the interfaces are just conforming more to the ideas and concepts they’re already familiar with in real life.

So take advantage of CSS motion as a usability tool. Help your users by giving them realism and context. The world on the small screen doesn't have to be so different from the real world around us, and the more similar it is, the easier it is for users to understand it. ❧

—

1. http://player.vimeo.com/video/116757193?byline=0&portrait=0&loop=1
2. http://player.vimeo.com/video/116757194?byline=0&portrait=0&loop=1
3. http://player.vimeo.com/video/116756637?byline=0&portrait=0&loop=1
4. http://player.vimeo.com/video/116757192?byline=0&portrait=0&loop=1
5. http://app.danceityourself.com
6. http://codepen.io/drewbrolik/pen/opskq
7. http://codepen.io/drewbrolik/pen/ivzfK

8. http://codepen.io/drewbrolik/pen/LFijf

9. http://css3.bradshawenterprises.com/transitions/#animatable

10. http://codepen.io/drewbrolik/pen/mJgqb

11. http://www.css3files.com/animation/

12. http://css3.bradshawenterprises.com/transitions/#differentTiming

Understanding CSS Timing Functions

BY STEPHEN GREIG ❧

People of the world, strap yourself in and hold on tight, for you are about to experience truly hair-raising excitement as you get to grips with the intricacies of the hugely interesting CSS timing function!

OK, so the subject matter of this article probably hasn't sent your blood racing, but all jokes aside, the timing function is a bit of a hidden gem when it comes to CSS animation, and you could well be surprised by just how much you can do with it.

First of all, let's set the scene and ensure we're all familiar with the scenarios in which the timing function is relevant. As alluded to, the functionality becomes available when you're working in the realm of CSS animation, which includes CSS transitions as well as keyframe-based animation. So, what exactly is it and what does it do?

The CSS Timing Function Explained

It's one of the less obvious animation-based CSS properties, whereas most of its counterparts are rather self-explanatory. Nevertheless, the gist of it is that it enables you to control and vary the acceleration of an animation — that is, it enables you to control and vary the acceleration of an animation — that is, it defines where the animation speeds up and slows down over the specified duration.

While it does not affect the actual duration of an animation, it could have profound effects on how fast or slow the user *perceives* the animation to be. If you're not sold having learned of its actual purpose, then stick with me here because the timing-function property gets a lot more interesting than the definition suggests.

Note: There is no actual property named "timing-function." When I refer to this "property," I am actually referring to both the `transition-timing-function` and the `animation-timing-function` properties.

Before moving on, let's just familiarize ourselves with the syntax and where it fits into the whole process of defining an animation in CSS. To keep things simple, let's use a CSS transition for this example. We'll begin with the full array of transition properties:

```css
div {
   transition-property: background;
   transition-duration: 1s;
   transition-delay: .5s;
   transition-timing-function: linear;
}

/* This could, of course, be shortened to: */
div {
    transition: background 1s .5s linear;
}
```

The shorthand for defining a transition is fairly lenient, the only requirement for the order being that the delay parameter must be stated after the duration value (but not necessarily immediately after). Furthermore, the `transition-duration` value is the only one that is actually required for the code to function; and because the default values of the other parameters are adequate most of the time, your transitions seldom need to be anything more than the following snippet:

```
div {
    transition: 1s;
}

/* This is the same as saying: */
div {
    transition: all 1s 0s ease;
}
```

But that’s a bit boring. While the defaults are often sufficient for standard hover events and such, if you’re working on something a little more substantial, then the timing function is a serious trick for fine-tuning an animation!

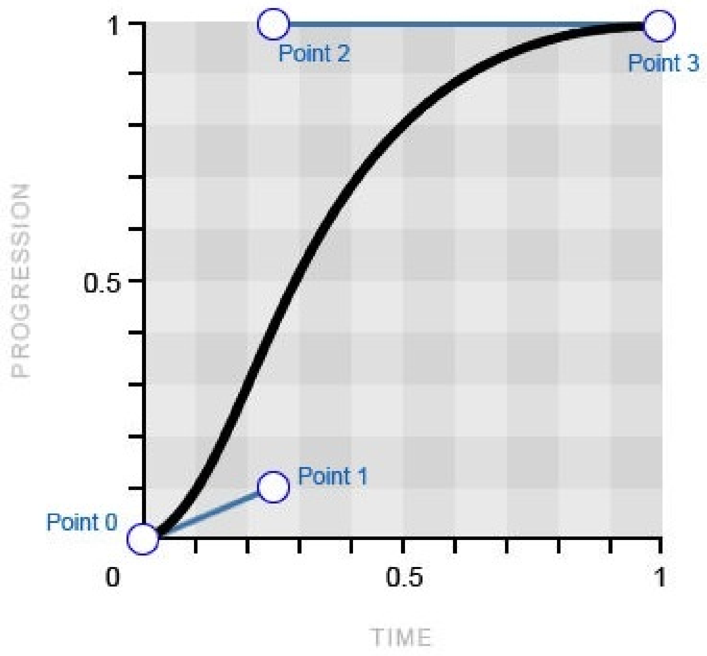
Anyway, you now have an idea of what the timing function does. Let’s look at how it does it.

Looking Under the Hood

Many of you probably don't look past the available keywords for the timing-function property, of which there are five: `ease` (the default), `ease-in`, `ease-out`, `ease-in-out` and `linear`. However, these keywords are simply shorthands for defining the Bézier curve.

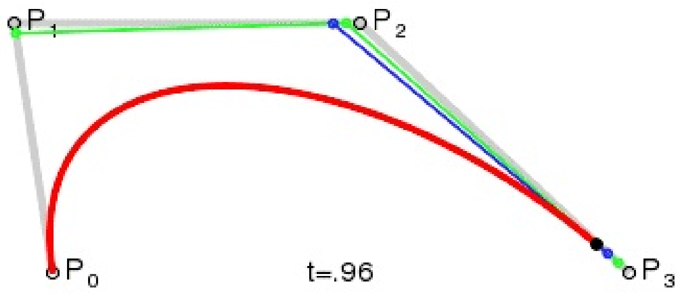
The what?!

You might not be familiar with the term, but I'd wager that you've actually seen a Bézier curve — heck, if you've used graphical editing software, then you've probably even created one! That's right, when you use the Pen or Path tool to create a nice smooth curve, then you're drawing a Bézier curve! Anyway, to get to the point of this section, the Bézier curve is the magic behind the timing function; it basically describes the acceleration pattern on a graph.



This Bézier curve translates to the `ease` *keyword.*

If you’re anything like me the first time I saw a Bézier curve like this, then you might be wondering how on earth that curve could be formed from four points plotted on a graph! I couldn’t possibly tell you in words, but, fortunately, I have a particularly fantastic GIF to do the job for me, courtesy of Wikipedia.



A cubic Bézier curve being drawn (View the animated GIF on Wikipedia[1])

Because this curve is formed from four points, we refer to it as a “cubic” Bézier curve, as opposed to a “quadratic” curve (three points) or a “quartic” curve (five points).

Introducing The cubic-bezier() Function

Now then, this is where things get really exciting, as I reveal that you can actually get access to this curve through the `cubic-bezier()` function, which can simply be used in place of the keywords of the timing-function property value. I appreciate that you may need a moment to contain your excitement…

With the `cubic-bezier()` function, you can manipulate the Bézier curve whichever way you desire, creating completely custom acceleration patterns for your animation! So, let’s look at how this function works and how it enables you to create your very own Bézier curve.

First, we know that the curve is formed by four points, which are referred to as point 0, point 1, point 2 and point 3. The other important thing to note is that the first and last points (0 and 3) are already defined on the graph, with point 0 always sitting at `0,0` (bottom left) and point 3 always sitting at `1,1` (top right).

That leaves just point 1 and point 2 available for you to plot on the graph, which you can do using the `cubic-bezier()` function! The function takes four parameters, the first two being the x and y coordinates of point 1 and the latter two being the x and y coordinates of point 2.

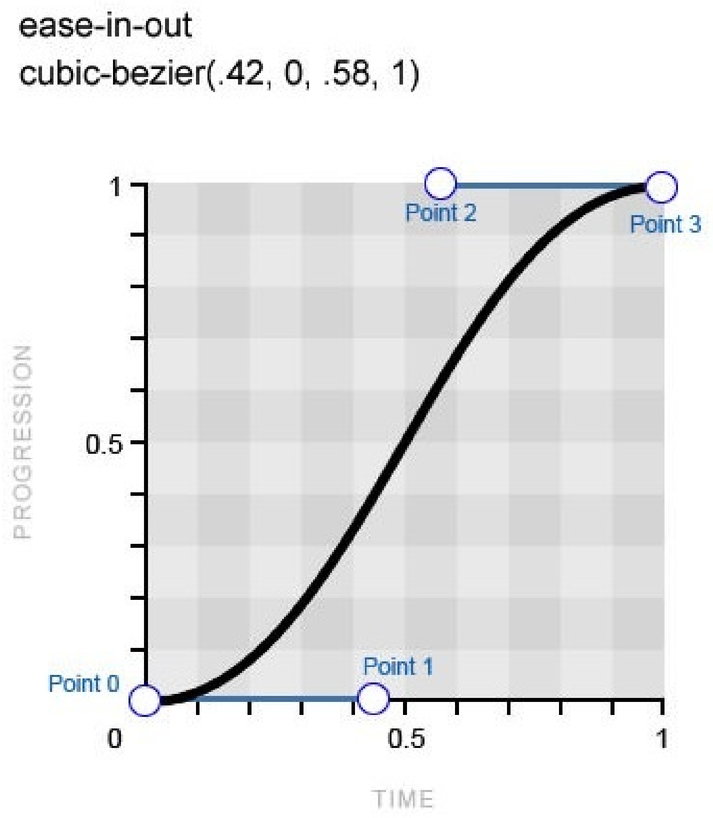
```css
transition-timing-function: cubic-bezier(x, y, x, y);
```

To get comfortable with the syntax, as well as with how it creates the curve and with its physical effect on the animation, I'll take you through the five timing-function keywords, their equivalent `cubic-bezier()` values and the resulting effect on the animation.

EASE-IN-OUT

Let's start with the `ease-in-out` keyword because the logic behind this curve and how it translates to the animation are probably the easiest to understand.

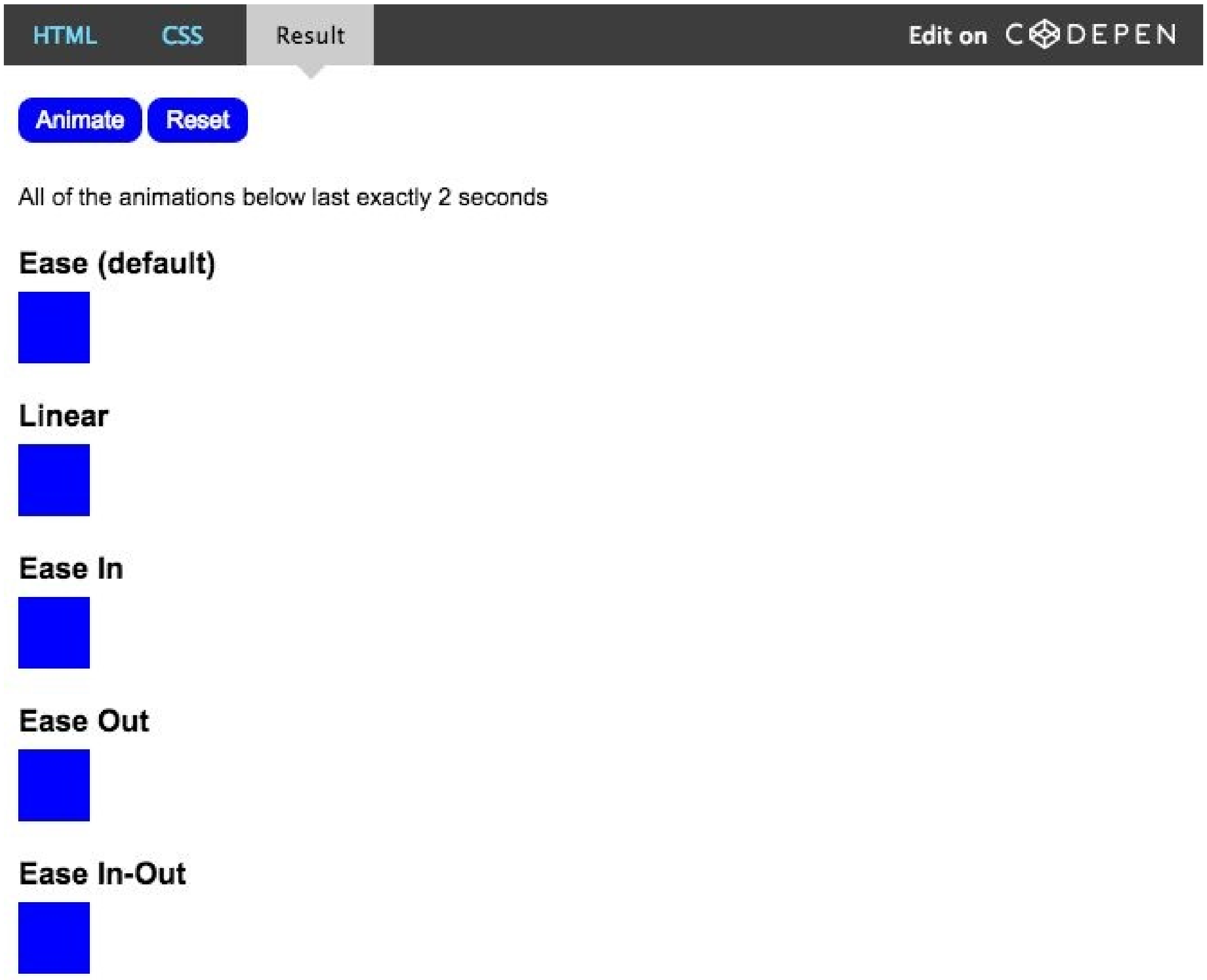
```css
/* The cubic-bezier() equivalent of the ease-in-out keyword */
transition-timing-function: cubic-bezier(.42, 0, .58, 1);
```



A perfectly symmetrical Bézier curve, which means that the animation will ease in to full speed and then ease out at exactly the same rate.

You can see that point 1 is positioned at 0.42 along the x-axis and at 0 on the y-axis, whereas point 2 is positioned at 0.58 on the x-axis and 1 on the y-axis. The result is a perfectly symmetrical Bézier curve, which means that the animation will ease in to full speed and then ease out at exactly the same rate; hence, the name of this keyword.

If you look at the demo below, you will see the physical effect of the `ease-in-out` value, as well as how it compares to the other keyword values.



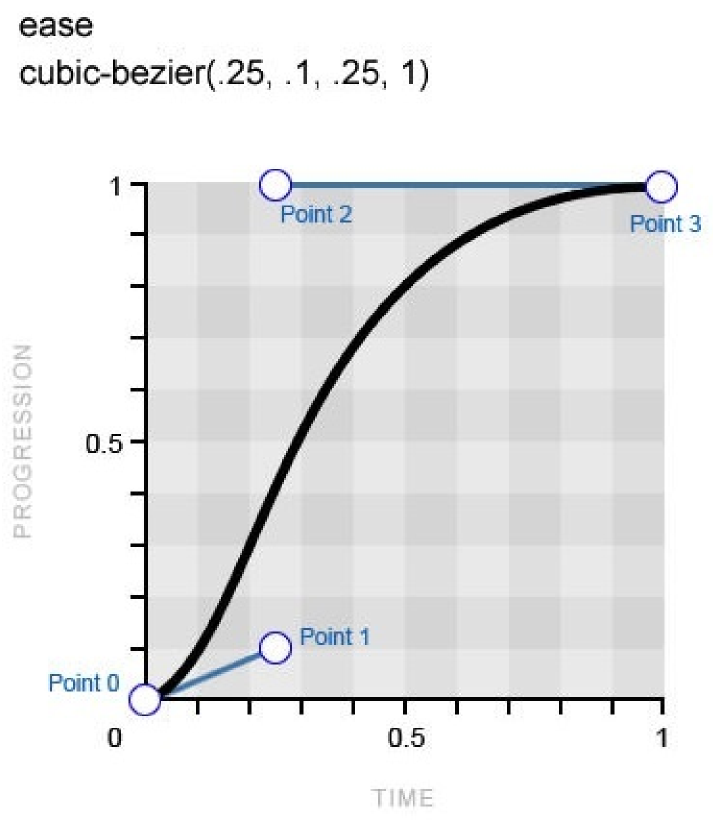
See the Pen on CodePen[2]

EASE

The `ease` keyword is the default value of the CSS timing-function property, and it is actually quite similar to the previous one, although it eases in at a faster rate before easing out much more gradually.

```
/* The ease keyword and its cubic-bezier() equivalent */
```

```
transition-timing-function: cubic-bezier(.25, .1, .25, 1);
```



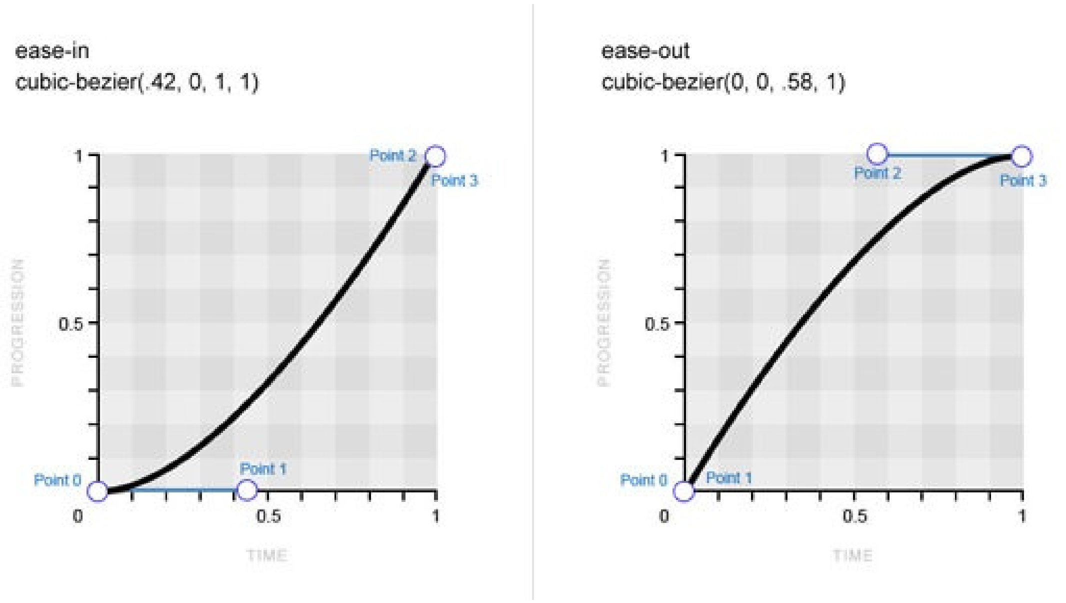
The curve for the keyword `ease` *eases in at a faster pace before easing out much more gradually.*

You can see a sharper incline in the curve initially, while the finish is more drawn out, which directly translates to the physical effect of this timing function on the animation. Don't forget to refer to the earlier demo after reviewing each of these examples to compare the effects.

EASE-IN AND EASE-OUT

The `ease-in` and `ease-out` keywords are, unsurprisingly, exact opposites. The former eases in before maintaining full speed right through to the finish, while the latter hits full speed right off the bat before easing to a finish. The `ease-in-out` keyword that we looked at previously is, as logic would suggest, a perfect combination of these two Bézier curves.

```
/* The ease-in keyword and its cubic-bezier() equivalent */
transition-timing-function: cubic-bezier(.42, 0, 1, 1);

/* The ease-out keyword and its cubic-bezier() equivalent */
transition-timing-function: cubic-bezier(0, 0, .58, 1);
```

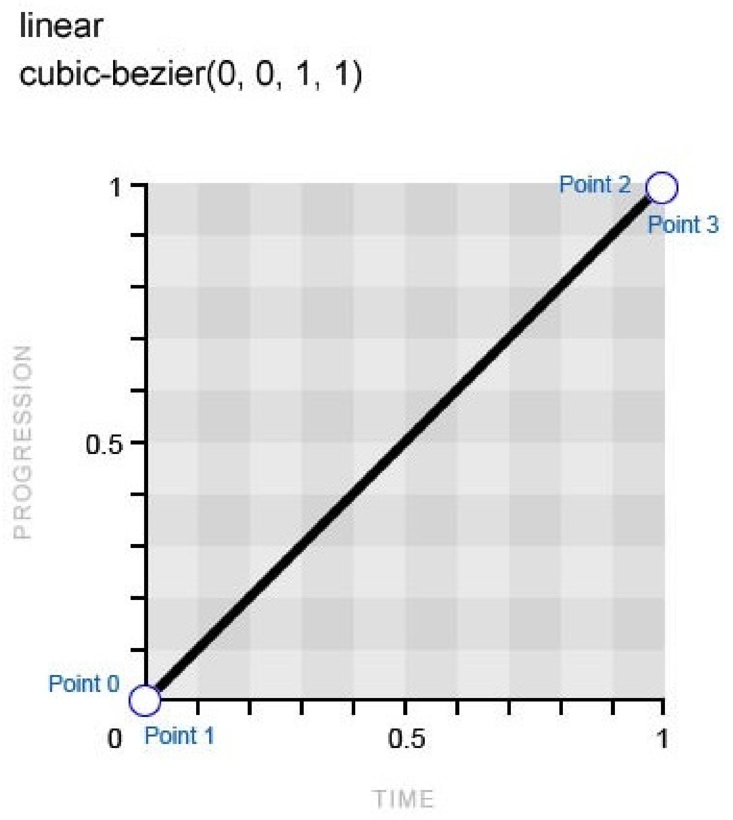


Bézier curve for the `ease-in` *keyword (left) and the* `ease-out` *keyword (right). (View large version*[3]*)*

LINEAR

The final keyword to address does not correspond to a curve at all. As the name implies, the `linear` timing-function value maintains the same speed throughout the animation, which means that the resulting Bézier curve (or lack of) would be just a straight line. There is simply no varying acceleration pattern to depict on the graph.

```
/* The linear keyword and its cubic-bezier() equivalent */
transition-timing-function: cubic-bezier(0, 0, 1, 1);
```



The `linear` timing-function value maintains the same speed throughout the animation.

If you refer back to the demo, you will probably notice that, despite all of the examples having the exact same duration values, some of the animations appear to be slower than the others. Why is that? Well, to take the `ease-in-out` keyword as an example, we know that it starts and finishes at a slower pace, which means that it has to cover the middle ground of the animation at a much faster pace. This effectively ensures that we perceive the actual animation to be both shorter and faster, while, say, the linear animation appears longer and more drawn out.

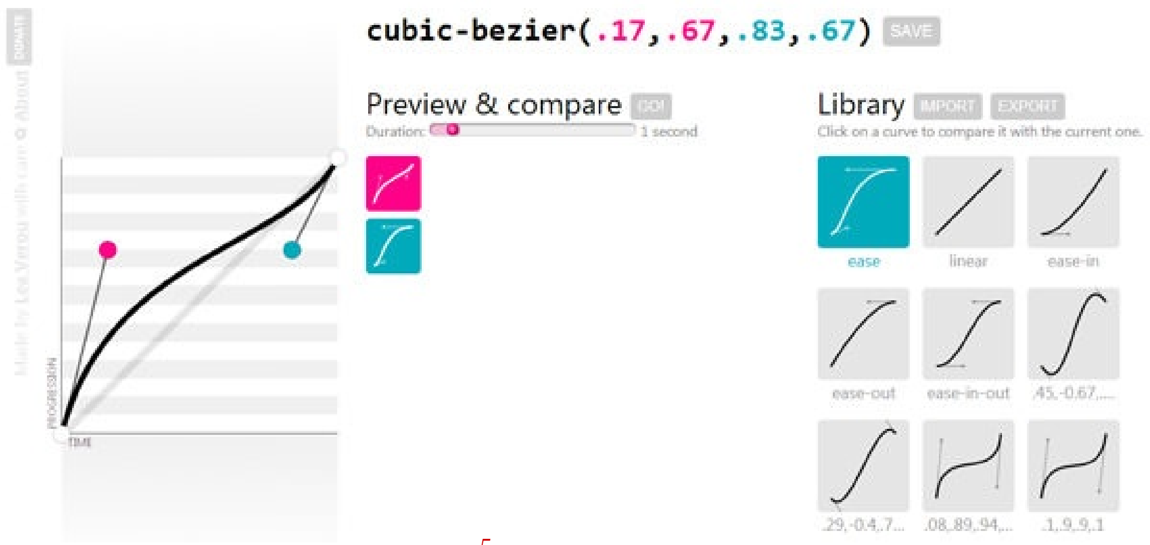
You might feel that this article is very slowly easing into its full pace (see what I did there?), so now it’s finally time to step things up a bit and look at how to use the `cubic-bezier()` function to create our own custom timing functions.

Creating Custom Acceleration Patterns With The cubic-bezier() Function

Now that we’ve seen how the keywords equate to the corresponding Bézier curves and seen their effect on the animation, let’s look at how to manipulate the curve to create custom acceleration patterns.

You should now be able to plot points 1 and 2 on the graph using the `cubic-bezier()` function and have a pretty solid idea of how this will affect the animation. However, considering that you’re plotting points on a graph that you typically can’t see, obviously this could get very tedious very quickly.

Fortunately, people like Lea Verou exist, who seemingly won’t rest until CSS development couldn’t be made any easier! Lea has developed the aptly named Cubic Bézier, which is basically a playground for creating completely custom Bézier curves and comparing them in action to the predefined keywords. What this means for you is that, instead of tediously editing the decimals in your `cubic-bezier()` functions, you can just visit Cubic Bezier[4] and play around with the curve until you’ve achieved the desired effect. Much more convenient.

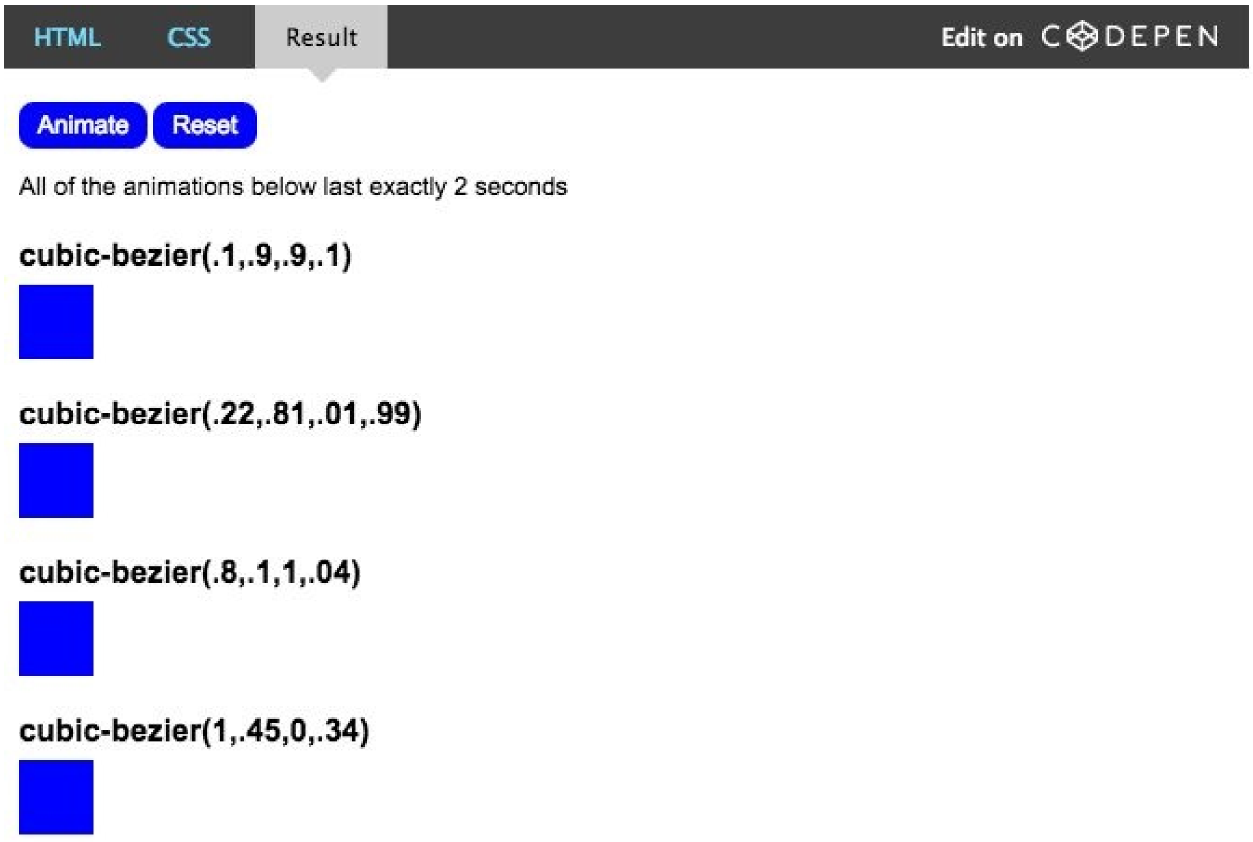


Lea Verou’s superbly useful Cubic Bézier[5]

The shorthand keywords give you great options for timing functions to start with, but the differences between them often appear minor. Only when you start to create custom Bézier curves will you realize the radical effect the timing function can have on an animation.

Just look at the following working examples to see the extreme differences between the animations, despite all of them having the exact same duration values.

values.



All of the animations below last exactly 2 seconds

cubic-bezier(.1,.9,.9,.1)

cubic-bezier(.22,.81,.01,.99)

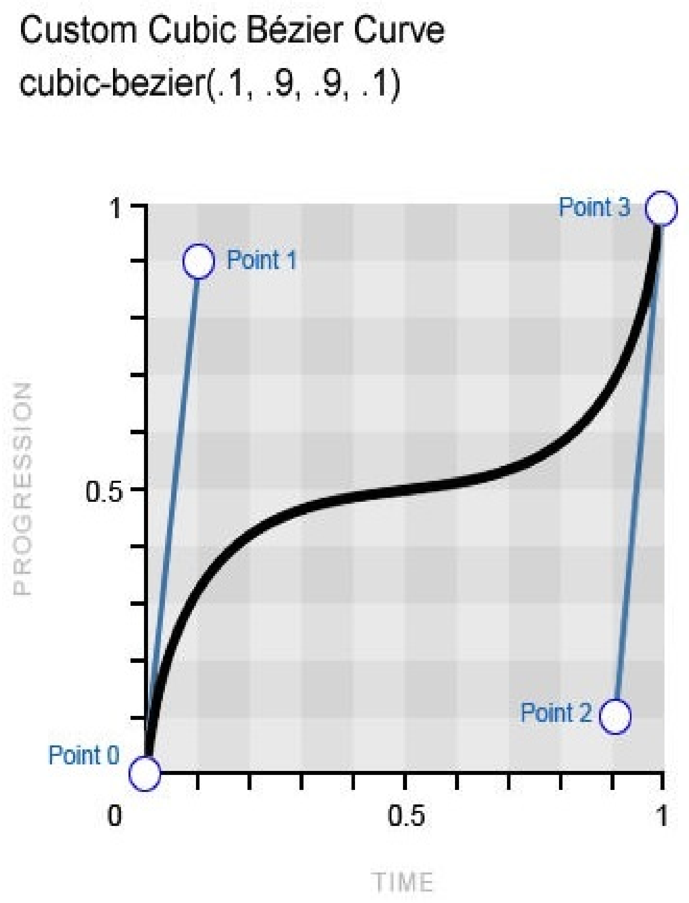
cubic-bezier(.8,.1,1,.04)

cubic-bezier(1,.45,0,.34)

See the Pen on CodePen[6]

Let's take a closer look at the first of these examples and try to understand why it produces such a radically different effect.

```
/* cubic-bezier() values for first example from preceding demo page
*/
transition-timing-function: cubic-bezier(.1, .9, .9, .1);
```



Example of a custom Bézier curve

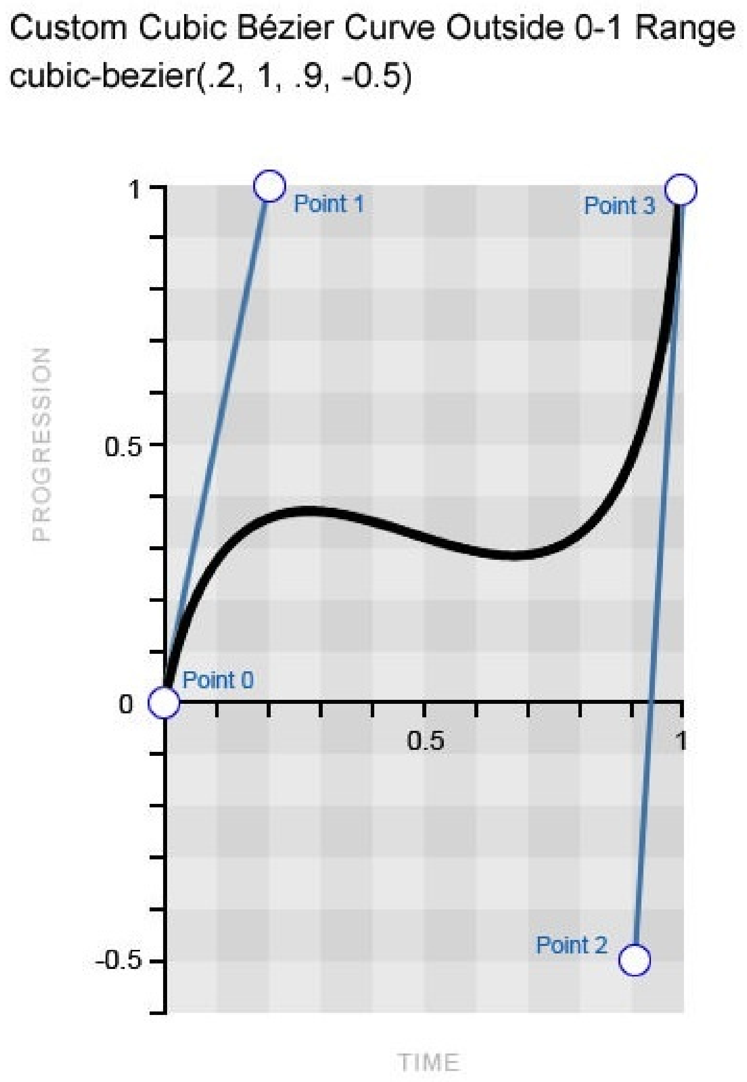
The main difference between this timing function and the default keywords is the sharp inclines of the Bézier curve against the “progression” scale (the y-axis). This means that the animation progresses in short bursts, with a long gradual pause in the middle (where the curve levels out). This pattern contrasts starkly with what we’ve become used to with the timing-function keywords, which take the opposite approach, with the easing periods coming at the beginning or the end of the animation, rather than in the middle.

With custom Bézier curves now in the bag, surely we have all but exhausted the capabilities of the `cubic-bezier()` function, right?! You’d think so, but this crafty beggar has got even more up its sleeve!

Getting Creative With Bézier Curves

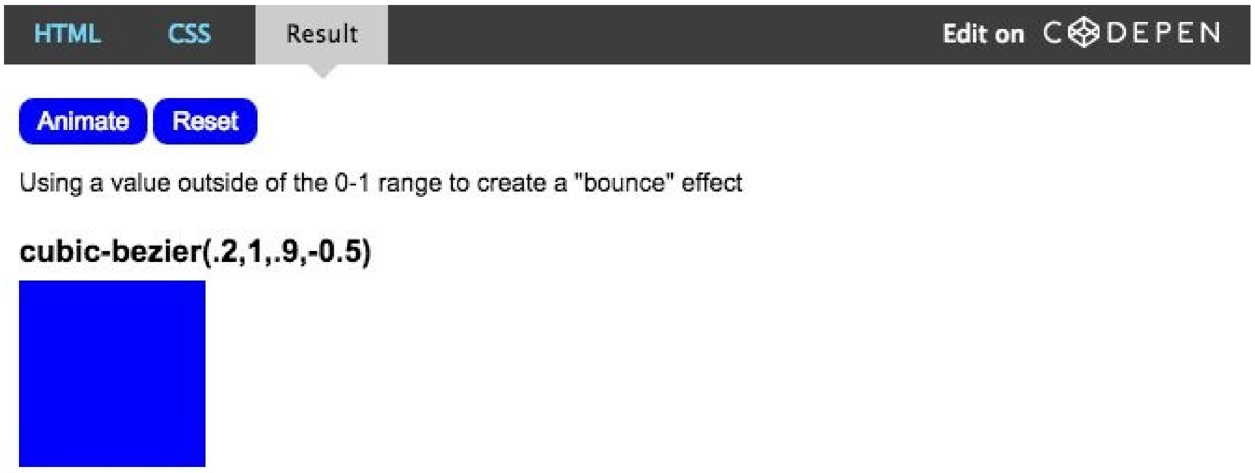
Yep, it’s true: Bézier curves get even more exciting! Who’d have thought? The scope widens with the revelation that only the time scale (x-axis) is limited to the 0–1 range on the graph, whereas the progression scale (y-axis) can extend below and beyond the typical 0–1 range.

The progression scale is exactly what you’d think it is, with the bottom end (0) marking the beginning of the animation and the top end (1) marking the end of the animation. Typically, the cubic Bézier curve always travels north along this progression scale at varying intensities until it reaches the end point of the animation. However, the ability to plot points 1 and 2 outside of this 0–1 range allows the curve to meander back *down* the progression scale, which actually causes reverse motion in the animation! As ever, the best way to understand this is through visuals:



A custom Bézier curve using a value outside of the typical 0–1 range

You can see that point 2 is plotted outside of the typical 0–1 range at -0.5, which in turn pulls the curve downward. If you look at the following demo, you'll see that this creates a bouncing effect in the middle of the animation.

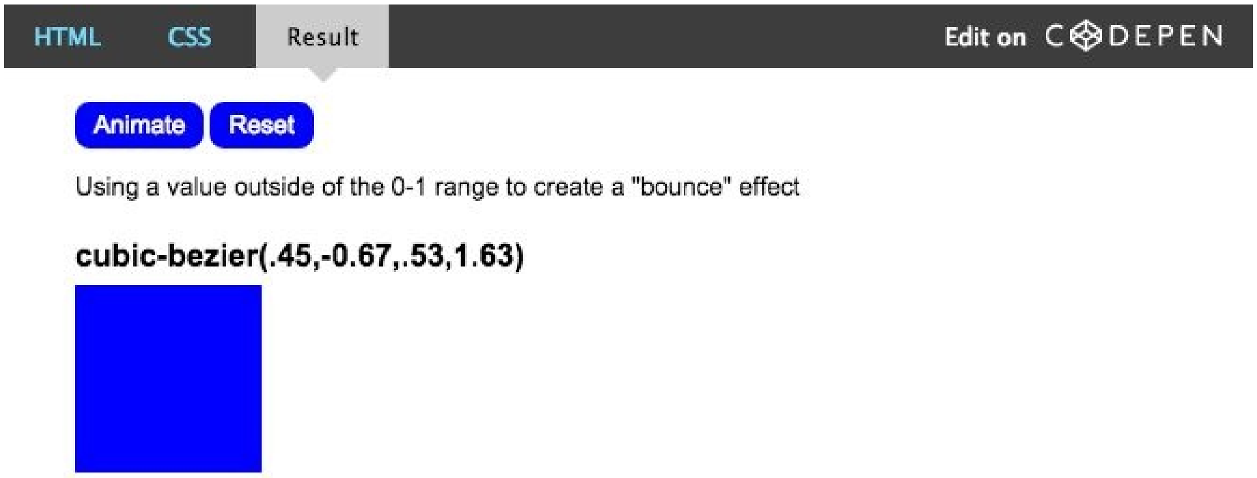


Using a value outside of the 0-1 range to create a "bounce" effect

cubic-bezier(.2,1,.9,-0.5)

See the Pen on CodePen[7]

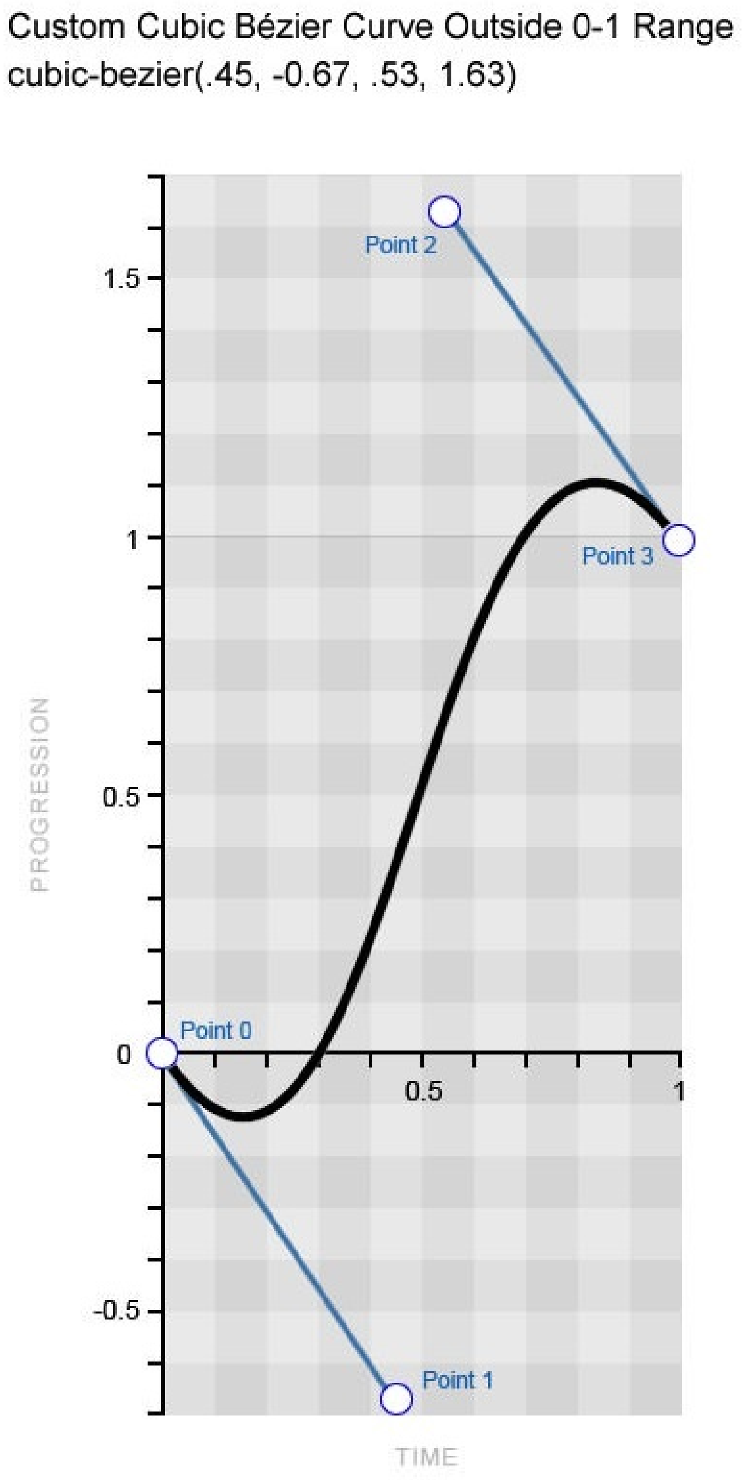
Conversely, you could place this backward motion at the beginning of the animation and also make it temporarily run past its intended finishing point. Think of it like taking a couple of steps back to get a running start; then, at the end, your momentum carries you past your destination, causing you to take a couple of steps back to ensure that you arrive at the intended destination. Look at the working example to really understand what we’re talking about here. In addition, the Bézier curve that produces this effect can be seen below.



Using a value outside of the 0-1 range to create a "bounce" effect

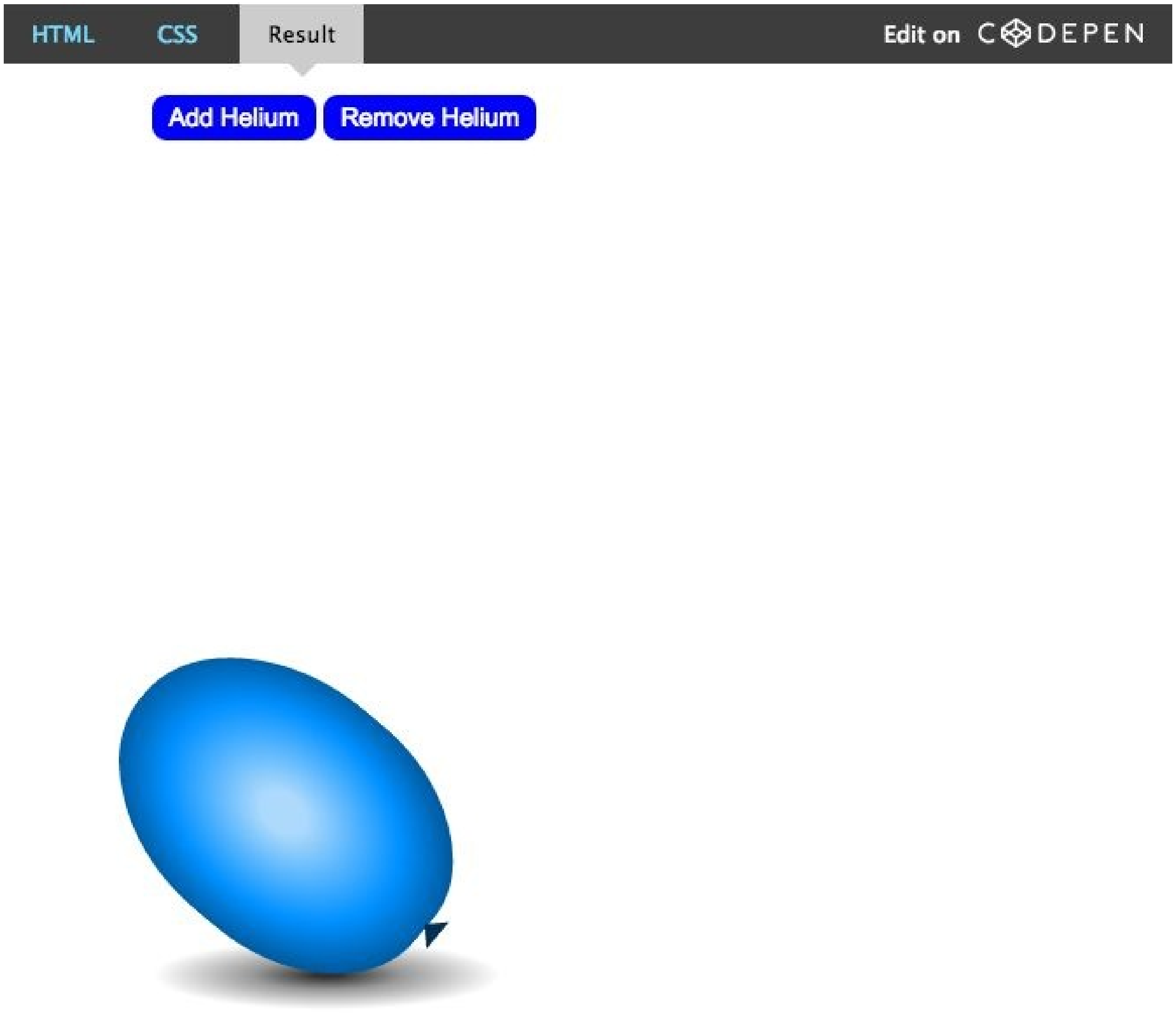
cubic-bezier(.45,-0.67,.53,1.63)

See the Pen on CodePen[8]



A custom Bézier curve using a value outside of the typical 0–1 range

You should now have a pretty good idea of how `cubic-bezier()` values outside of the typical 0–1 range can physically affect how an animation plays out. We can look at moving boxes all day long, but let’s finish this section with an example that clearly demonstrates this type of creative timing function.

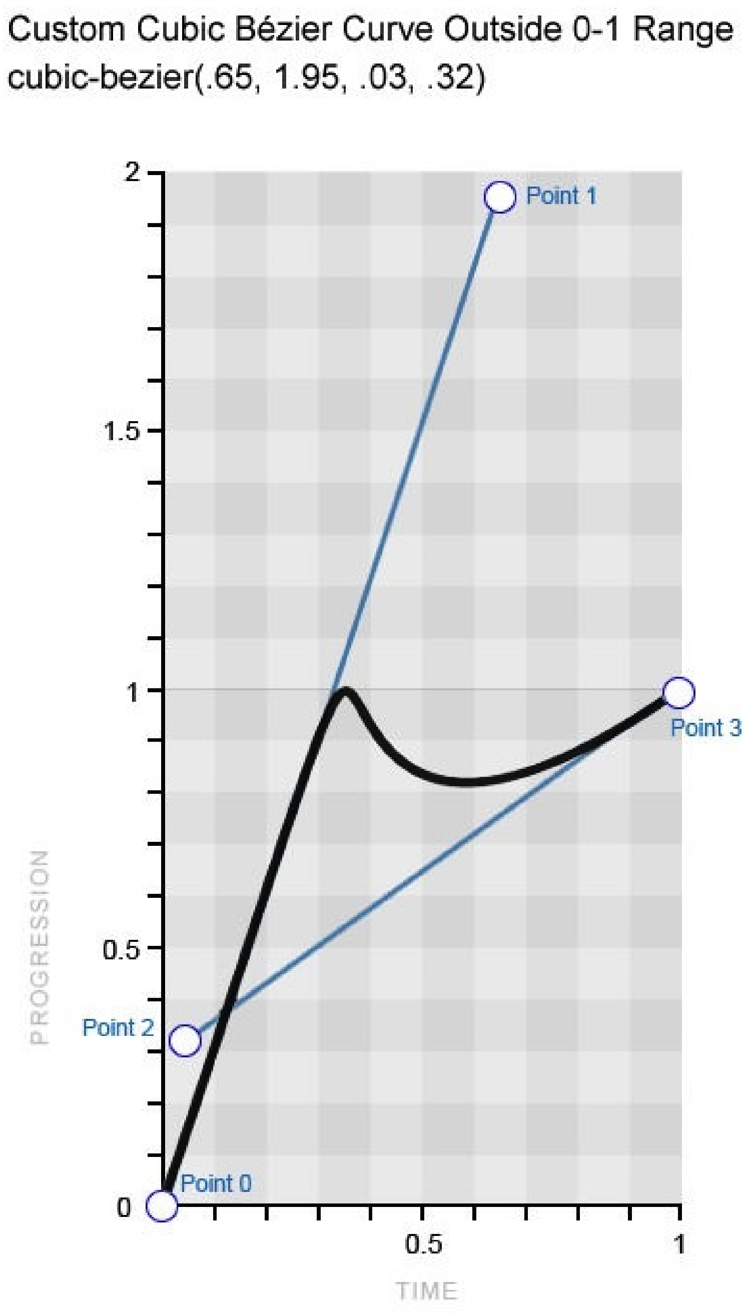


See the Pen on CodePen[9]

That’s right: We’re animating a floating balloon! …What? Haven’t you always wanted to do that with CSS?

The brief for this animation is to “add helium” to the balloon on click so that it floats to the “ceiling,” where it will bounce slightly before sticking to the top, as it naturally would. Using a `cubic-bezier()` value outside of the 0–1 range allows us to create the bounce and ultimately helps to produce a realistic effect. The following snippet shows the coordinates used in the `cubic-bezier()` function, and the resulting Bézier curve appears below that.

```
/* The cubic-bezier() values for the bouncing balloon */
transition-timing-function: cubic-bezier(.65, 1.95, .03, .32);
```



A custom Bézier curve to emulate a bouncing balloon

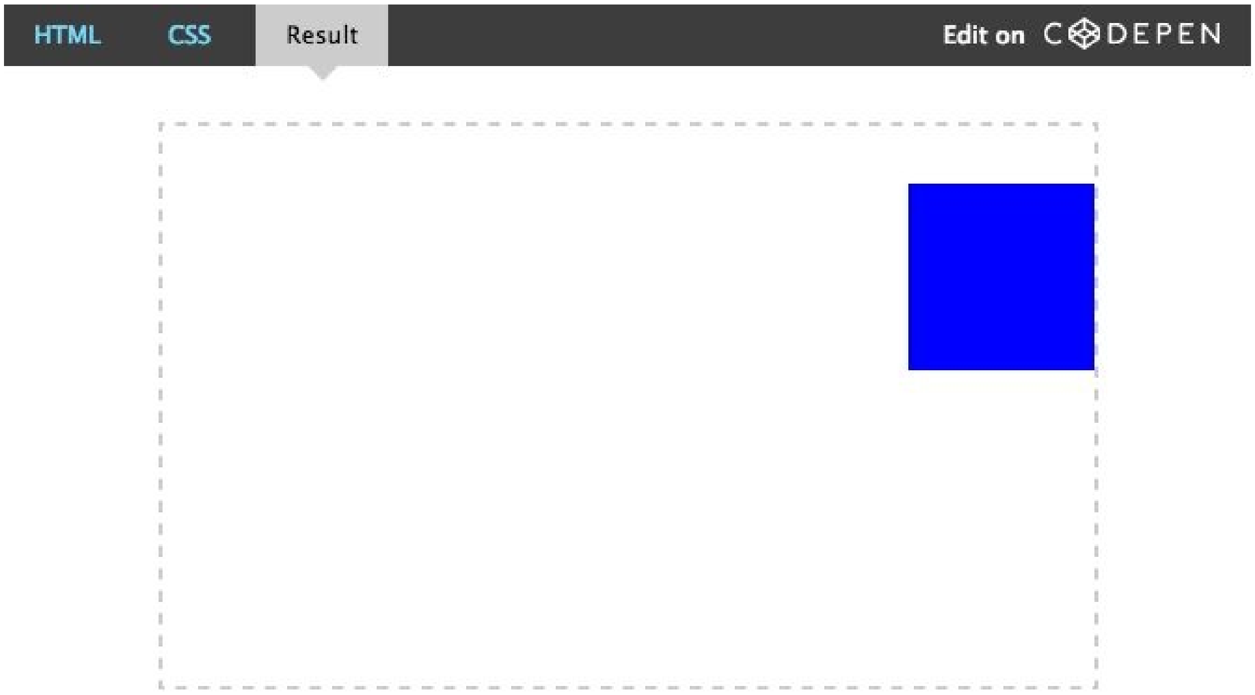
This example explains particularly well how the curve translates to the resulting animation because it reflects it almost perfectly. First, you can see that the curve travels from the beginning of the progression scale to the end in a straight line, indicating that the balloon travels from the start of the animation to the finish at a constant speed. Then, very similarly to the balloon, the curve bounces off the tip of the scale and goes into reverse before returning steadily and gradually to the top. All quite straightforward really!

Once you’ve mastered the curve and the art of manipulating it to do what you want, you’ve nailed it.

Timing Functions And Keyframe-Based CSS Animation

One final thing to note before moving on (yes, there is more to cover!) is how timing functions behave when applied to CSS keyframe animation. The concepts are all exactly the same as those in the transition examples we’ve been working with so far, but with one exception that is important to be aware of: When you apply a timing function to a set of keyframes, it is executed per keyframe, rather than for the animation as a whole.

To clarify, if you have four keyframes that move a box around four corners of a square, and you apply the “bouncing” timing function that we used in the earlier balloon example, then *each* of the four movements would experience the bounce, rather than the entire animation. Let’s see this in action and view the code.



See the Pen on CodePen[10]

```
@keyframes square {
   25% {
      top:200px;
      left:0;
   }

   50% {
      top:200px;
      left:400px;
   }

   75% {
      top:0;
      left:400px;
   }
}

div {
   animation: square 8s infinite cubic-bezier(.65, 1.95, .03, .32);
   top: 0;
   left: 0;
   /* Other styles */
}
```

Note that if the `100%` keyframe isn't defined, then the element will simply return to its original style, which is the desired result in this case, so defining it is not necessary. It is clearly evident from the demo that the timing function is applied to each of the four keyframes because they each appear to bounce off the walls of the container.

If you need certain keyframes to take on a timing function that is different from the others, then go ahead and apply a separate timing-function value directly to the keyframe, as indicated in the following snippet.

```
@keyframes square {
  50% {
    top: 200px;
    left: 400px;
    animation-timing-function: ease-in-out;
  }
}
```

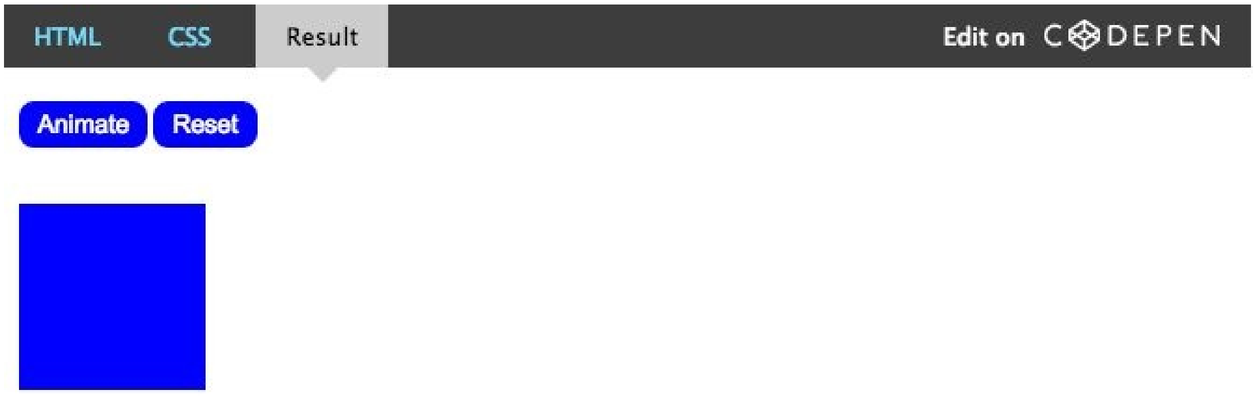
Introducing The steps() Timing Function

Did you think we were done with timing functions? Ha, think again, pal! I told you that there's a lot more to CSS timing functions than a few predefined easing functions!

functions!

For this section, we can swap our curves for straight lines as we explore the concept of “stepping functions,” which are achieved through the `steps()` timing function.

The `steps()` function is more of a niche tool, but it’s useful to have in the toolkit nonetheless. It enables you to break up an animation into steps, rather than the usual tweened motion that we’re used to. For example, if we wanted to animate a square moving 400 pixels to the right in four steps over 4 seconds, then the square would jump 100 pixels to the right every second, rather than animating in a continuous motion. Let’s examine the syntax that we’d need for this particular example, which should be an absolute breeze now that we’ve tackled the intricacies of the `cubic-bezier()` function!



See the Pen on CodePen[11]

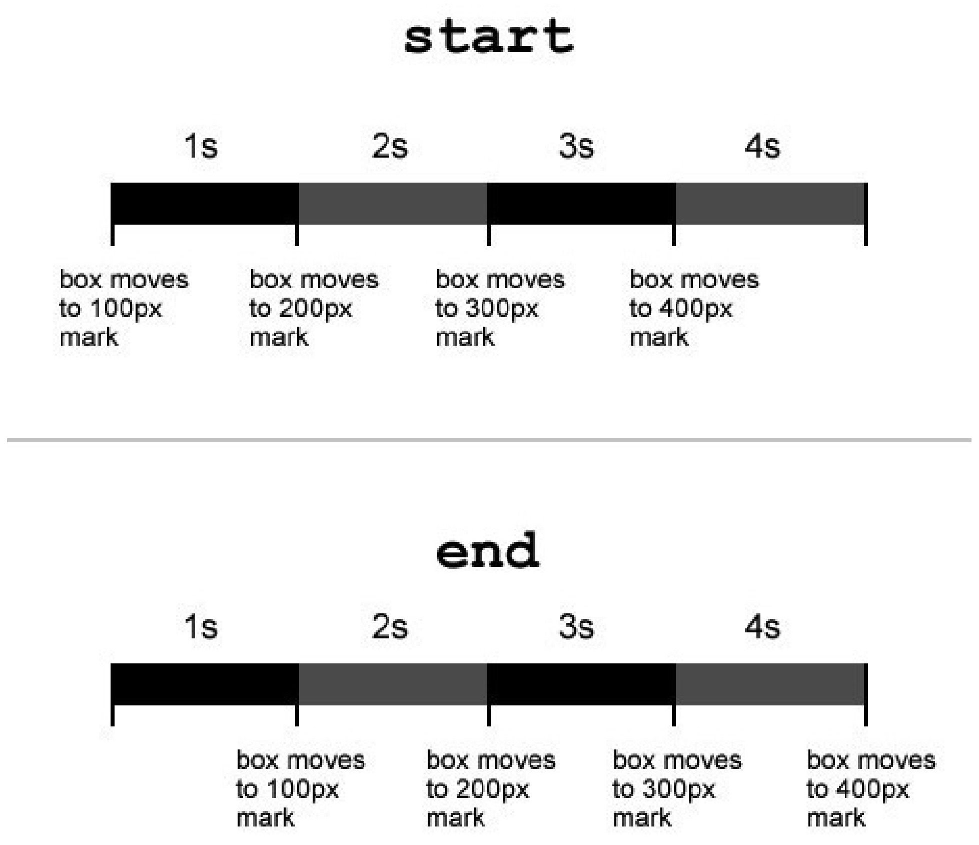
```
div {
   transition: 4s steps(4);
}
```

```
div:target {
   left: 400px;
}
```

As you can see, it’s a simple matter of stating the number of steps to divide the animation into — but bear in mind that this number must be a positive integer, so no negatives or decimals. However, a second, optional parameter affords us slightly more control, the possible values for which are `start` and `end`, the latter being the default value.

```
transition-timing-function: steps(4, start);
transition-timing-function: steps(4, end);
```

A value of `start` would run the animation at the beginning of each step, whereas a value of `end` would run the animation at the end of each step. Using the previous “moving box” example, the following image does a far better job of explaining the difference between these values.



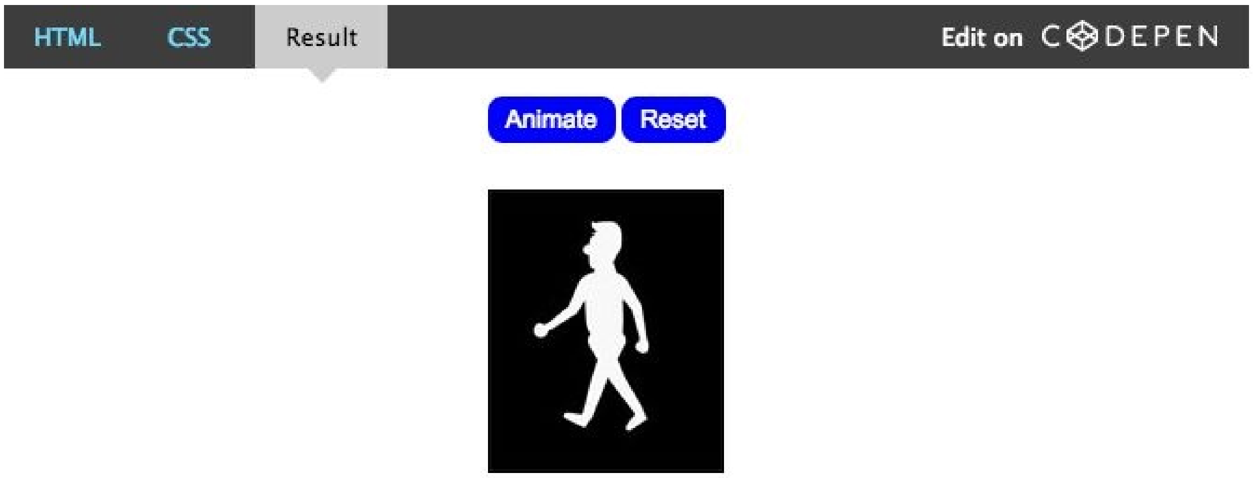
The difference between the start and end values in the `steps()` function.

You can see that with a value of `start`, the animation begins as soon as it is triggered, whereas with a value of `end`, it begins at the end of the first step (in this case, one second after being triggered).

And just to ensure that this overview is truly comprehensive, there are also two predefined keywords for the `steps()` function: `step-start` and `step-end`. The former is equivalent to `steps(1, start)`, and the latter is the same as `steps(1, end)`.

Creative Use Cases For Stepping Functions

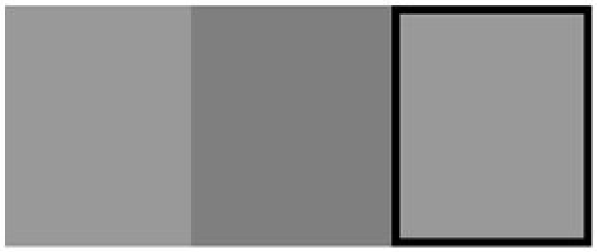
OK, so you probably don’t have much of a need to animate a moving box in steps very often, but there are plenty of other cool uses for the `steps()` function. For example, if you have all of the sprites for a basic cartoon, then you could use this technique to play it through frame by frame, using just a couple of CSS properties! Let’s look at a demo and the code that makes it function.



See the Pen on CodePen[12]

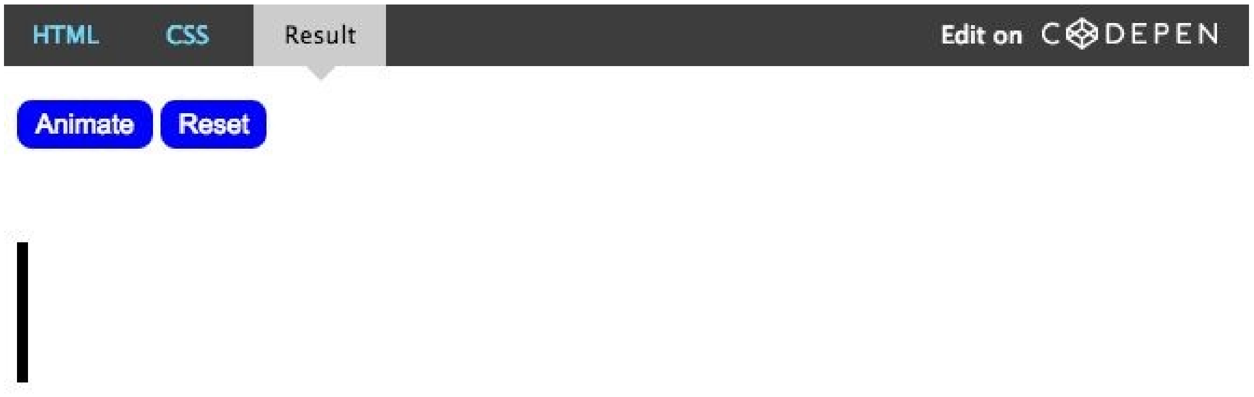
```
div {
   width: 125px;
   height: 150px;
   background: url(images/sprite.jpg) left;
   transition: 2s steps(16);
   /* The number of steps = the number of frames in the cartoon */
}

div:target {
   background-position: -2000px 0;
}
```

First, we have a small rectangular box (125 pixels wide), which has a background image (2000 pixels wide) containing 16 frames side by side. This background image is initially flush with the left edge of the box; so, all we need to do now is move the background image all the way to the left so that all 16 frames pass through the small rectangular window. With a normal animation, the frames would just slide in and out of view as the background image moves leftwards; however, with the `steps()` function, the background image can move to the left in 16 steps, ensuring that each of the 16 frames snaps in and out of view, as desired. And just like that, you are playing a basic cartoon using just a CSS transition!



This GIF demonstrates the concept of the background image passing through the "window" in steps, so that each frame snaps in and out of view. (View it in action[13])

Another creative use of the `steps()` function that I've found comes courtesy of Lea Verou (who else?), who has come up with a particularly clever typing animation, which I'll break down for you now.



See the Pen on CodePen[14]

First, you need some text, and — unfortunately — you need to know exactly how many characters you're working with because you'll need to use this number in the CSS. Another requirement is that the font must be monospaced, so that all characters are exactly the same width.

```
<p>smashingmag</p>

.text {
   width: 6.6em;
   width: 11ch; /* Number of characters */
   border-right: .1em solid;
   font: 5em monospace;
}
```

The text we're working with has 11 characters. With the help of the `ch` CSS unit, we can actually use this figure to define the width of the paragraph, although we should specify a fallback width for browsers that don't support this unit. The paragraph is then given a solid black border on the right side, which will become the cursor. Now everything is in place; we just need to animate it, which is extremely simple.

Two separate animations are required: one for the cursor and one for the typing. To achieve the former, all we need to do is make the black border blink, which couldn't be simpler.

```
@keyframes cursor {
   50% {
     border-color: transparent;
   }
}

.text {
   /* existing styles */
   animation: cursor 1s step-end infinite;
}
```

As intended, the black border simply switches between black and transparent and then loops continuously. This is where the `steps()` function is vital because, without it, the cursor would just fade in and out, rather than blink.

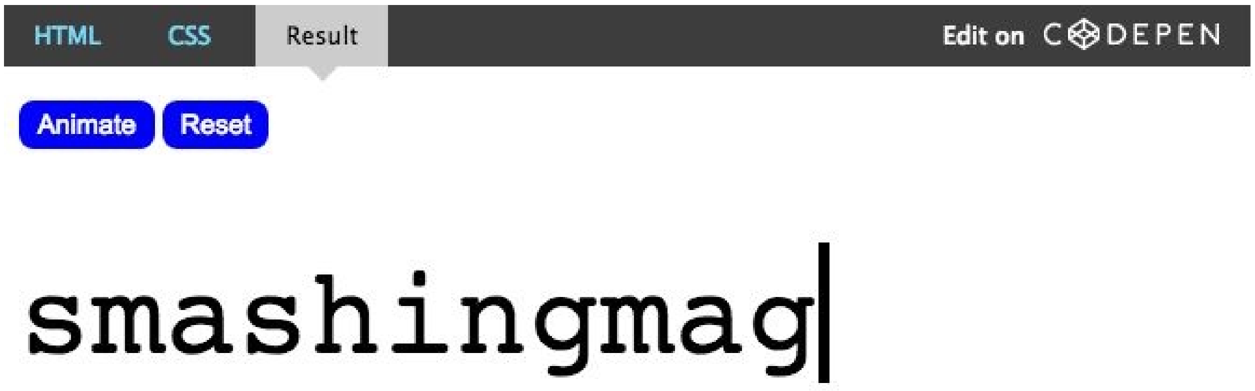
Finally, the typing animation is just as simple. All we need to do is reduce the width of the paragraph to zero, before animating it back to full width in 11 steps (the number of characters).

```
@keyframes typing {
   from {
      width: 0;
   }
}
```

```css
.text {
   /* existing styles */
   animation: typing 8s steps(11),
              cursor 1s step-end infinite;
}
```

With this single keyframe in place, the text will reveal itself one letter at a time over 8 seconds, while the black `border-right` (the cursor) will blink away continuously. The technique is very simple yet extremely effective.

Just to add to this excellent use of the `steps()` function by Lea Verou, reversing the effect and making the text appear to be deleted is also a cinch. To achieve this, just change the keyframe keyword so that it reads `to` rather than `from`, and then add an `animation-fill-mode` parameter of `forwards` to the set of `animation` rules. This will ensure that once the text "deletes" (i.e. when the animation has finished), the text will remain deleted. Look at the demo below to see this in action.



See the Pen on CodePen[15]

The downside to both of the examples featured in this section is that you must know the number of frames or characters beforehand in order to specify the right number of steps, and if this number changes at all, then you will need to alter the code as well. Still, the `steps()` function has shown its worth here and is another fantastic piece of functionality of the CSS timing function.

The Browser Support Situation

We’ve established that you can’t use a CSS timing function unless the browser supports CSS-based animation — namely, the CSS Transitions and CSS Animation (keyframe-based) modules. Fortunately, support is in pretty great shape these days.

SUPPORT FOR CSS TRANSITIONS

Browser	Prefixed support	Unprefixed support
Internet Explorer	N/A	10+
Firefox	4+ (`-moz-`)	16+
Chrome	4+ (`-webkit-`)	26+
Safari	3.1+ (`-webkit-`)	6.1+
Opera	10.5+ (`-o-` prefix)	12.1+

Although all current browser versions have dropped the prefix for transitions (which is awesome), you'd be wise to still include the `-webkit-` prefix to cater to old mobile browsers. I think the need to include the `-moz-` and `-o-` prefixes, however, has passed, as long as you progressively enhance, which you should be doing anyway!

SUPPORT FOR CSS ANIMATION

Browser	Prefixed support	Unprefixed support
Internet Explorer	N/A	10+
Firefox	5+ (`-moz-`)	16+
Chrome	4+ (`-webkit-`)	Not supported
Safari	4+ (`-webkit-`)	Not Supported
Opera	12 (`-o-` prefix), 15+ (`-webkit-` prefix)	12.1 only (not supported since switch to WebKit)

Again, for keyframe animations, include only the `-webkit-` prefix alongside your unprefixed code.

Evidently, browser support for CSS-based animation is in excellent shape, but support is slightly more fragmented when it comes to the nuances of timing functions. The following table explains in more detail.

functions. The following table explains in more detail.

SUPPORT FOR CSS TIMING FUNCTIONS

Browser	**Basic support**	**cubic-bezier() outside of 0-1 range**	**steps()**
Internet Explorer	10+	10+	10+
Firefox	4+	4+	4+
Chrome	4+	16+	8+
Safari	3.1+	6+	5.1+
Opera	10.5+	12.1+	12.1+

Again, although certain browsers have taken a little longer to support the full range of timing-function capabilities, you can see that support is now universal across current browser versions.

Summary

So, what have we learned about CSS timing functions? Time to recap.

- They define where an animation accelerates and decelerates.
- There is a great deal more to them than just the predefined keywords.
- You can create bounce effects with `cubic-bezier()` values outside of the 0–1 range.
- You can break an animation into any number of steps, rather than tweened motion.
- Browser support is in fantastic shape and ever improving.

Finally, although these techniques are now supported across the board, this wouldn't be an article about CSS3 techniques if I didn't mention progressive enhancement. Always work from the bottom up; that is to say, ensure that your work is acceptable and accessible on devices and browsers that can't deal with this functionality before enhancing for browsers that can cope with them.

Other than that, go wild! Happy curving and stepping!

OTHER RESOURCES

- “Cubic Bézier[16],” Lea Verou
  A playground for creating and comparing Bézier curves

- “Timing Functions[17],” Mozilla Developer Network
  A more technical overview of Bézier curves.

- “Bézier Curves[18],” Wikipedia
  More information ❧

—

1. http://en.wikipedia.org/wiki/File:Bezier_3_big.gif
2. http://codepen.io/stephengreig/pen/bHzqm/
3. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/04/05-bezier-curve-ease-in-ease-out.jpg
4. http://cubic-bezier.com/
5. http://cubic-bezier.com/

6. http://codepen.io/stephengreig/pen/baFhH/

7. http://codepen.io/stephengreig/pen/kILDb/

8. http://codepen.io/stephengreig/pen/xcCqj/

9. http://codepen.io/stephengreig/pen/vbqBh/

10. http://codepen.io/stephengreig/pen/rscGb/

11. http://codepen.io/stephengreig/pen/Gwbry/

12. http://codepen.io/stephengreig/pen/tuvfp/

13. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/04/13-frames-concept.gif

14. http://codepen.io/stephengreig/pen/Blbcs/

15. http://codepen.io/stephengreig/pen/LmohC/

16. http://cubic-bezier.com/

17. https://developer.mozilla.org/en/docs/Web/CSS/timing-function

18. http://en.wikipedia.org/wiki/B%C3%A9zier_curve

Styling And Animating SVGs With CSS

BY SARA SOUEIDAN ❧

CSS can be used to style and animate scalable vector graphics, much like it is used to style and animate HTML elements. In this article, which is a modified transcript of a talk I recently gave[1] at CSSconf EU[2] and From the Front[3], I'll go over the prerequisites and techniques for working with CSS in SVG.

I'll also go over how to export and optimize SVGs, techniques for embedding them and how each one affects the styles and animations applied, and then we'll actually style and animate with CSS.

Introduction

Scalable vector graphics (SVG) is an XML-based vector image format for two-dimensional graphics, with support for interactivity and animation. In other words, SVGs are XML tags that render shapes and graphics, and these shapes and graphics can be interacted with and animated much like HTML elements can be.

Animations and interactivity can be added via CSS or JavaScript. In this article, we'll focus on CSS.

There are many reasons why SVGs are great and why you should be using them today:

- SVG graphics are scalable and resolution-independent. They look great everywhere, from high-resolution “Retina” screens to printed media.
- SVGs have very good browser support[4]. Fallbacks for non-supporting browsers are easy to implement, too, as we’ll see later in the article.
- Because SVGs are basically text, they can be gzipped, making the files smaller that their bitmap counterparts (JPEG and PNG).
- SVGs are interactive and styleable with CSS and JavaScript.
- SVG comes with built-in graphics effects such as clipping and masking operations, background blend modes, and filters. This is basically the equivalent of having Photoshop photo-editing capabilities right in the browser.
- SVGs are accessible. In one sense, they have a very accessible DOM API, which makes them a perfect tool for infographics and data visualizations

and which gives them an advantage over HTML5 Canvas because the content of the latter is not accessible. In another sense, you can inspect each and every element in an SVG using your favorite browser’s developer tools, just like you can inspect HTML elements. And SVGs are accessible to screen readers if you make them so. We’ll go over accessibility a little more in the last section of this article.

- Several tools are available for creating, editing and optimizing SVGs. And other tools make it easier to work with SVGs and save a lot of time in our workflows. We’ll go over some of these tools next.

Exporting SVGs From Graphics Editors And Optimizing Them

The three most popular vector graphics editors are:

- Adobe Illustrator[5],
- Inkscape[6],
- Sketch[7].

Adobe Illustrator is a paid application from Adobe. It is a highly popular editor, with a nice UI and many capabilities that make it the favorite of most designers.

Inkscape is a popular free alternative. Even though its UI is not as nice as Illustrator's, it has everything you need to work with vector graphics.

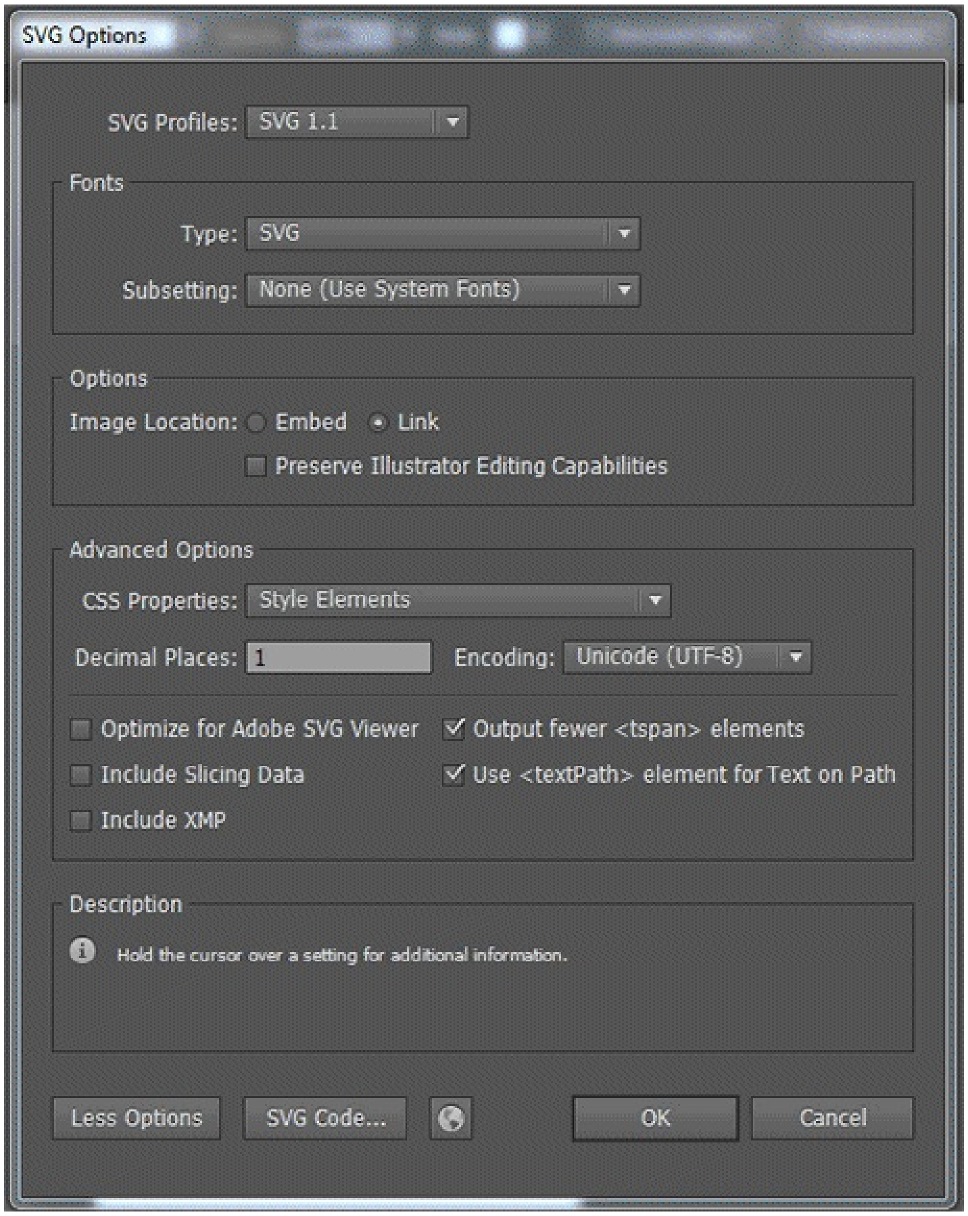
Sketch is a Mac OS X-only graphics app. It is not free either, but it has been making the rounds[8] among designers lately and gaining popularity[9], with a lot of resources and tools[10] being created recently to improve the workflow.

Choose any editor to create your SVGs. After choosing your favorite editor and creating an SVG but before embedding it on a web page, you need to export it from the editor and clean it up to make it ready to work with.

I'll refer to exporting and optimizing an SVG created in Illustrator. But the workflow applies to pretty much any editor, except for the Illustrator-specific options we'll go over next.

To export an SVG from Illustrator, start by going to "File" → "Save as," and then choose ".svg" from the file extensions dropdown menu. Once you've chosen the .svg extension, a panel will appear containing a set of options for exporting the SVG, such as which version of SVG to use, whether to embed images in the graphic or save them externally and link to them in the SVG, and how to add the styles to the SVG (by using presentation attributes or by using CSS properties in a `<style>` element).

The following image shows the best settings to choose when exporting an SVG for the web:



SVG Options

SVG Profiles: SVG 1.1

Fonts

Type: SVG

Subsetting: None (Use System Fonts)

Options

Image Location: Embed [✓ Link]

☐ Preserve Illustrator Editing Capabilities

Advanced Options

CSS Properties: Style Elements

Decimal Places: 1 Encoding: Unicode (UTF-8)

☐ Optimize for Adobe SVG Viewer

☑ Output fewer <tspan> elements

☐ Include Slicing Data

☑ Use <textPath> element for Text on Path

☐ Include XMP

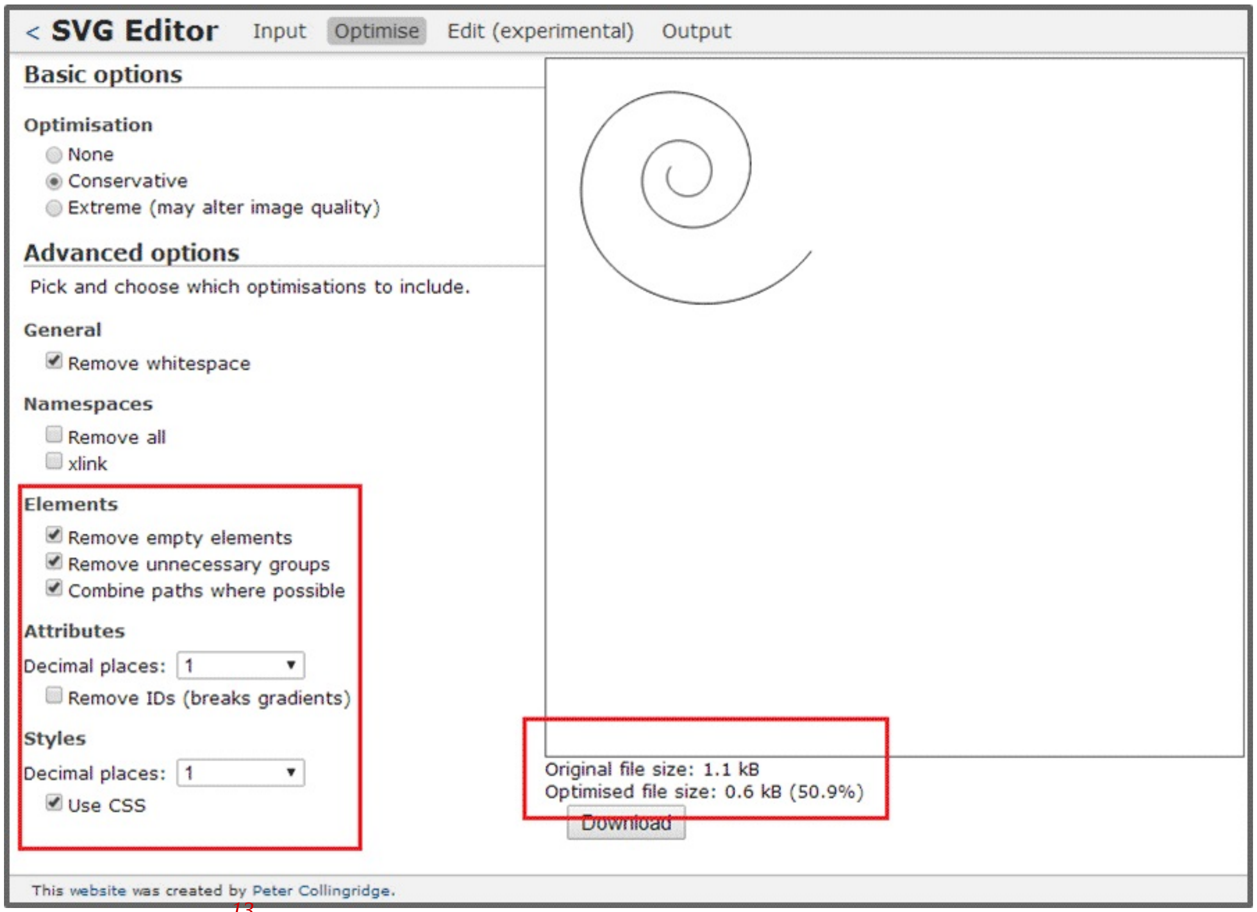
Description

Hold the cursor over a setting for additional information.

The reasons why the options above are best are explained in Michaël Chaize's excellent article "Export SVG for the Web With Illustrator CC[11]."

Whichever graphics editor you choose, it will not output perfectly clean and optimized code. SVG files, especially ones exported from editors, usually contain a lot of redundant information, such as meta data from the editor, comments, empty groups, default values, non-optimal values and other stuff that can be safely removed or converted without affecting the rendering of the SVG. And if you're using an SVG that you didn't create yourself, then the code is almost certainly not optimal, so using a standalone optimization tool is advisable.

Several tools for optimizing SVG code are out there. Peter Collingridge's SVG Editor[12] is an online tool that you input SVG code into either directly or by uploading an SVG file and that then provides you with several optimization options, like removing redundant code, comments, empty groups, white space and more. One option allows you to specify the number of decimal places of point coordinates.


< SVG Editor Input Optimise Edit (experimental) Output

Basic options

Optimisation

- None
- [✓ Conservative]
- Extreme (may alter image quality)

Advanced options

Pick and choose which optimisations to include.

General

- [✓] Remove whitespace

Namespaces

- [ ] Remove all
- [ ] xlink

Elements

- [✓] Remove empty elements
- [✓] Remove unnecessary groups
- [✓] Combine paths where possible

Attributes

Decimal places: 1

- [ ] Remove IDs (breaks gradients)

Styles

Decimal places: 1

- [✓] Use CSS

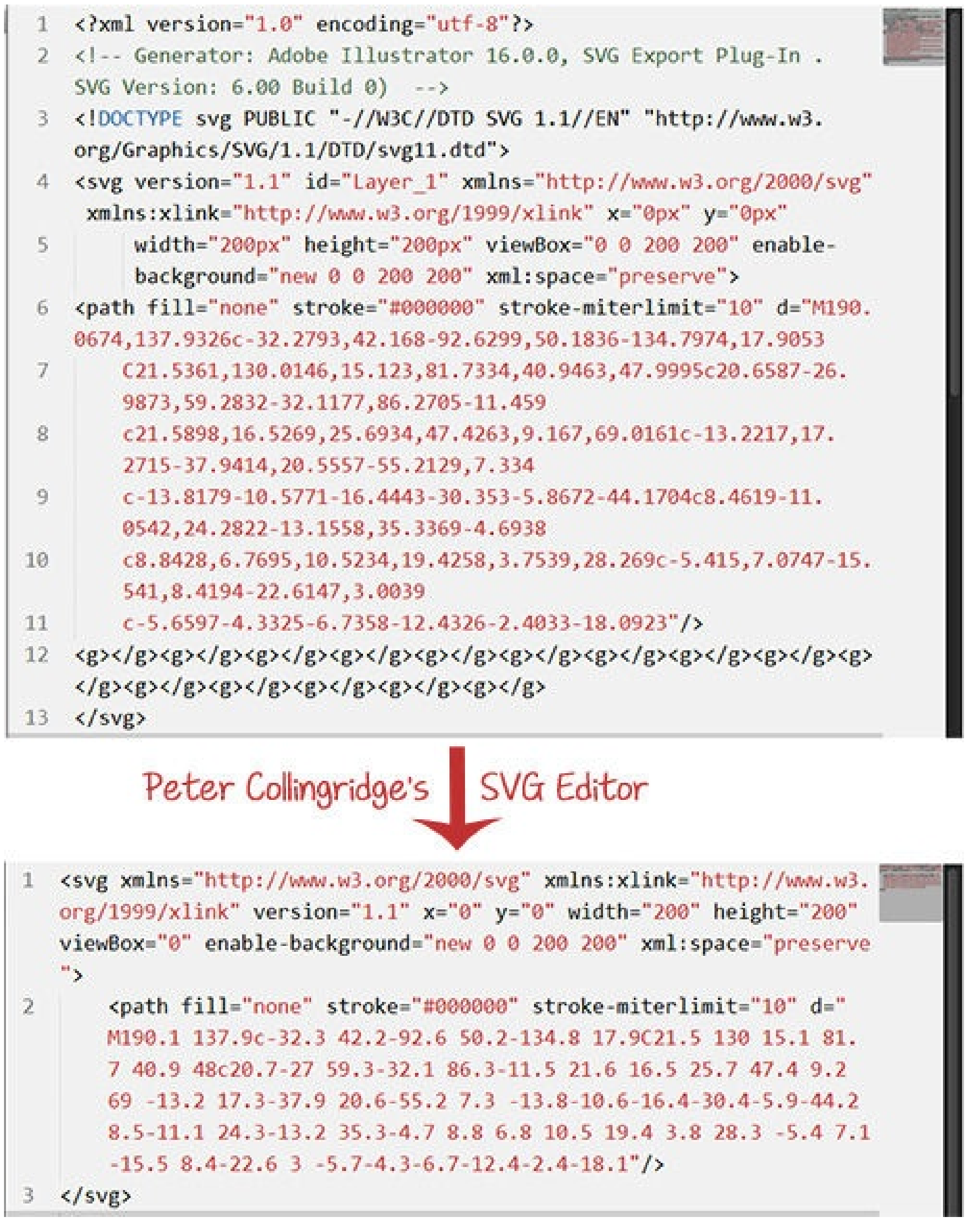
Original file size: 1.1 kB
Optimised file size: 0.6 kB (50.9%)

(View large version[13]*)*

Peter's optimizer can also automatically move inline SVG properties to a style block at the top of the document. The nice thing about it is that, when you check an option, you can see the result of the optimization live, which enables you to better decide which optimizations to make. Certain optimizations could end up breaking your SVG. For example, one decimal place *should* normally be enough. If you're working with a path-heavy SVG file, reducing the number of decimal places from four to one could slash your file's size by as much as half. However, it could also entirely break the SVG. So, being able to preview an optimization is a big plus.

Peter’s tool is an online one. If you’d prefer an offline tool, try SVGO[14] (the “O” is for “optimizer”), a Node.js-based tool that comes with a nice and simple drag-and-drop GUI[15]. If you don’t want to use an online tool, this one is a nice alternative.

The following screenshot (showing the path from the image above) is a simple before-and-after illustration of how much Peter’s tool optimizes SVG.



```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 16.0.0, SVG Export Plug-In .
SVG Version: 6.00 Build 0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.
org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
     width="200px" height="200px" viewBox="0 0 200 200" enable-
     background="new 0 0 200 200" xml:space="preserve">
<path fill="none" stroke="#000000" stroke-miterlimit="10" d="M190.
0674,137.9326c-32.2793,42.168-92.6299,50.1836-134.7974,17.9053
    C21.5361,130.0146,15.123,81.7334,40.9463,47.9995c20.6587-26.
    9873,59.2832-32.1177,86.2705-11.459
    c21.5898,16.5269,25.6934,47.4263,9.167,69.0161c-13.2217,17.
    2715-37.9414,20.5557-55.2129,7.334
    c-13.8179-10.5771-16.4443-30.353-5.8672-44.1704c8.4619-11.
    0542,24.2822-13.1558,35.3369-4.6938
    c8.8428,6.7695,10.5234,19.4258,3.7539,28.269c-5.415,7.0747-15.
    541,8.4194-22.6147,3.0039
    c-5.6597-4.3325-6.7358-12.4326-2.4033-18.0923"/>
<g></g><g></g><g></g><g></g><g></g><g></g><g></g><g></g><g></g><g>
</g><g></g><g></g><g></g><g></g><g></g>
</svg>
```

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.
org/1999/xlink" version="1.1" x="0" y="0" width="200" height="200"
viewBox="0" enable-background="new 0 0 200 200" xml:space="preserve
">
    <path fill="none" stroke="#000000" stroke-miterlimit="10" d="
    M190.1 137.9c-32.3 42.2-92.6 50.2-134.8 17.9C21.5 130 15.1 81.
    7 40.9 48c20.7-27 59.3-32.1 86.3-11.5 21.6 16.5 25.7 47.4 9.2
    69 -13.2 17.3-37.9 20.6-55.2 7.3 -13.8-10.6-16.4-30.4-5.9-44.2
    8.5-11.1 24.3-13.2 35.3-4.7 8.8 6.8 10.5 19.4 3.8 28.3 -5.4 7.1
    -15.5 8.4-22.6 3 -5.7-4.3-6.7-12.4-2.4-18.1"/>
</svg>
```

Notice the size of the original SVG compared to the optimized version. Not to mention, the optimized version is much more readable.

After optimizing the SVG, it’s ready to be embedded on a web page and further customized or animated with CSS.

Styling SVGs With CSS

The line between HTML and CSS is clear: HTML is about content and structure, and CSS is about the look. SVG blurs this line, to say the least. SVG 1.1 did not require CSS to style SVG nodes — styles were applied to SVG elements using attributes known as “presentation attributes.”

Presentation attributes are a shorthand for setting a CSS property on an element. Think of them as special style properties. They even contribute to the style cascade, but we’ll get to that shortly.

The following example shows an SVG snippet that uses presentation attributes to style the “border” (`stroke`) and “background color” (`fill`) of a star-shaped polygon:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="300px"
height="300px" viewBox="0 0 300 300">
  <polygon
fill = "#FF931E"
stroke = "#ED1C24"
stroke-width = "5"
points = "279.1,160.8 195.2,193.3 174.4,280.8   117.6,211.1
27.9,218.3 76.7,142.7 42.1,59.6 129.1,82.7 197.4,24.1 202.3,114 "/>
</svg>
```

The `fill`, `stroke` and `stroke-width` attributes are presentation attributes.

In SVG, a subset of all CSS properties may be set by SVG attributes, and vice versa. The SVG specification lists the SVG attributes that may be set as CSS properties[16]. Some of these attributes are shared with CSS, such as `opacity` and `transform`, among others, while some are not, such as `fill`, `stroke` and `stroke-width`, among others.

In SVG 2, this list will include `x`, `y`, `width`, `height`, `cx`, `cy` and a few other presentation attributes that were not possible to set via CSS in SVG 1.1. The new list of attributes can be found in the SVG 2 specification[17].

Another way to set the styles of an SVG element is to use CSS properties. Just like in HTML, styles may be set on an element using inline style attributes:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" style="width:
300px; height: 300px;" viewBox="0 0 300 300">
<polygon
  style = "fill: #FF931E; stroke: #ED1C24; stroke-width: 5;"
  points = "279.1,160.8 195.2,193.3 174.4,280.8   117.6,211.1
27.9,218.3 76.7,142.7 42.1,59.6 129.1,82.7 197.4,24.1 202.3,114 "/>
</svg>
```

Styles may also be set in rule sets in a `<style>` tag. The `<style>` tag can be placed in the `<svg>` tag:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="300px"
height="300px" viewBox="0 0 300 300">
  <style type="text/css">
  <![CDATA[
  selector {/* styles */}
  ]]>
  </style>
  <g id=".."> … </g>
</svg>
```

And it can be placed outside of it, if you're embedding the SVG inline in the document:

```
<!DOCTYPE html><!-- HTML5 document -->
<html>
<head> … </head>
<body>
<style type="text/css">
  /* style rules */
</style>
<!-- xmlns is optional in an HTML5 document →
<svg viewBox="0 0 300 300">
<!-- SVG content -->
</svg>
</body>
</html>
```

And if you want to completely separate style from markup, then you could always link to an external style sheet from the SVG file, using the `<?xml-stylesheet>` tag, as shown below:

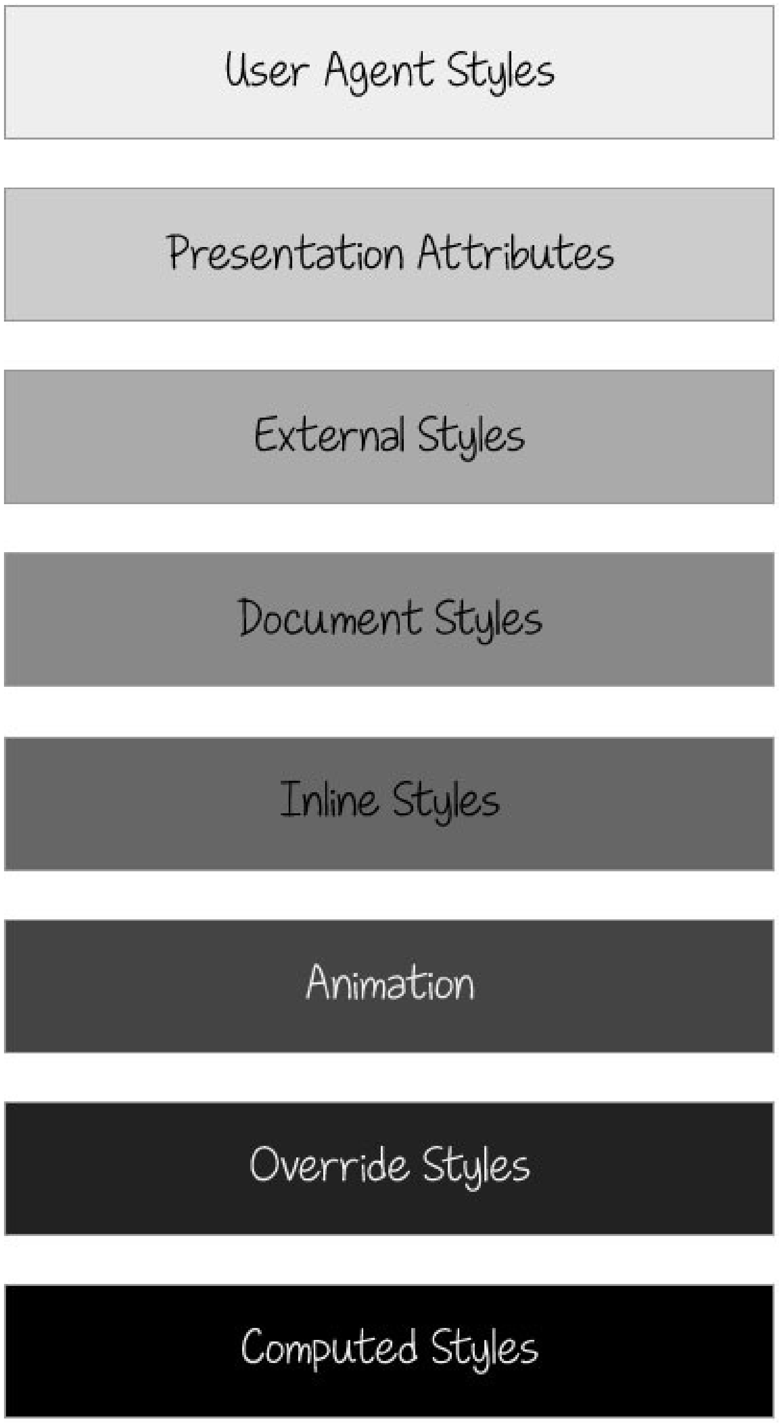
```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet type="text/css" href="style.css"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width=".."
height=".." viewBox="..">
  <!-- SVG content -->
</svg>
```

STYLE CASCADES

We mentioned earlier that presentation attributes are sort of special style properties and that they are just shorthand for setting a CSS property on an SVG node. For this reason, it only makes sense that SVG presentation attributes would contribute to the style cascade.

Indeed, presentation attributes count as low-level “author style sheets” and are overridden by any other style definitions: external style sheets, document style sheets and inline styles.

The following diagram shows the order of styles in the cascade. Styles lower in the diagram override those above them. As you can see, presentation attribute styles are overridden by all other styles except for those specific to the user agent.



For example, in the following code snippet, an SVG circle element has been drawn. The fill color of the circle will be deep pink, which overrides the blue fill specified in the presentation attribute.

```
<circle cx="100" cy="100" r="75" fill="blue" style="fill:deepPink;"
/>
```

SELECTORS

Most CSS selectors can be used to select SVG elements. In addition to the general type, class and ID selectors, SVGs can be styled using CSS2's dynamic pseudo-classes[18] (`:hover`, `:active` and `:focus`) and pseudo-classes[19] (`:first-child`, `:visited`, `:link` and `:lang`. The remaining CSS2 pseudo-classes, including those having to do with generated content[20] (such as `::before` and `::after`), are not part of the SVG language definition and, hence, have no effect on the style of SVGs.

The following is a simple animation of the fill color of a circle from deep pink to green when it is hovered over using the tag selector and the `:hover` pseudo-class:

```
<style>
circle {
  fill: deepPink;
  transition: fill .3s ease-out;
}

circle:hover {
  fill: #009966;
}
```

```
</style>
```

Much more impressive effects can be created. A simple yet very nice effect comes from the Iconic[21] icons set, in which a light bulb is lit up when hovered over. A demo of the effect[22] is available.

Notes

Because presentation attributes are expressed as XML attributes, they are case-sensitive. For example, when specifying the fill color of an element, the attribute must be written as `fill="…"` and not `Fill="…"`.

Furthermore, keyword values for these attributes, such as the `italic` in `font-style="italic"`, are also case-sensitive and must be specified using the exact case defined in the specification that defines that value.

All other styles specified as CSS properties — whether in a style attribute or a `<style>` tag or in an external style sheet — are subject to the grammar rules specified in the CSS specifications, which are generally less case-sensitive. That being said, the SVG "Styling"[23] specification recommends using the exact property names (usually, lowercase letters and hyphens) as defined in the CSS specifications and expressing all keywords in the same case, as required by presentation attributes, and not taking advantage of CSS's ability to ignore case.

Animating SVGs With CSS

SVGs can be animated the same way that HTML elements can, using CSS keyframes and animation properties or using CSS transitions.

In most cases, complex animations will usually contain some kind of transformation — a translation, a rotation, scaling and/or skewing.

In most respects, SVG elements respond to `transform` and `transform-origin` in the same way that HTML elements do. However, a few inevitable differences result from the fact that, unlike HTML elements, SVG elements aren't governed by a box model and, hence, have no margin, border, padding or content boxes.

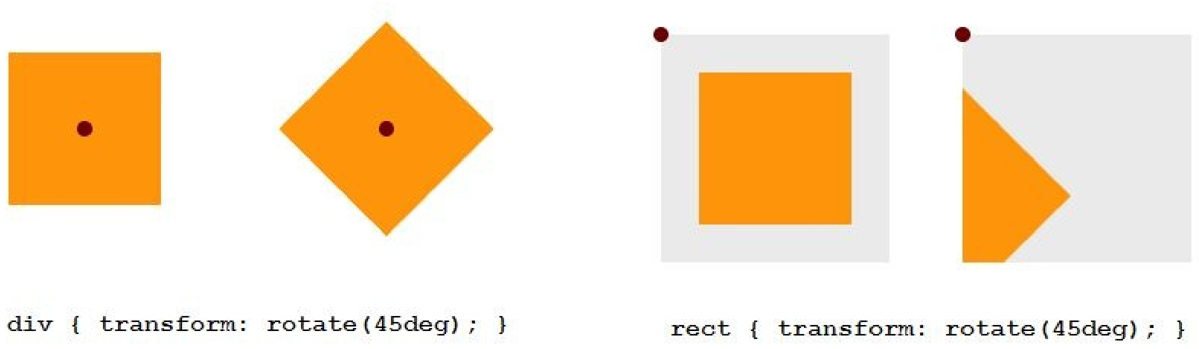
By default, the transform origin of an HTML element is at `(50%, 50%)`, which is the element's center. By contrast, an SVG element's transform origin is positioned at the origin of the user's current coordinate system, which is the `(0, 0)` point, in the top-left corner of the canvas.

Suppose we have an HTML `<div>` and an SVG `<rect>` element:

```
<!DOCTYPE html>
…
```

```
<div style="width: 100px; height: 100px; background-color: orange">
</div>
<svg style="width: 150px; height: 150px; background-color: #eee">
  <rect width="100" height="100" x="25" y="25" fill="orange" />
</svg>
```

If were were to rotate both of them by 45 degrees, without changing the default transform origin, we would get the following result (the red circle indicates the position of the transform origin):


div { transform: rotate(45deg); }
rect { transform: rotate(45deg); }

(View large version[24])

What if we wanted to rotate the SVG element around its own center, rather than the top-left corner of the SVG canvas? We would need to explicitly set the transform origin using the `transform-origin` property.

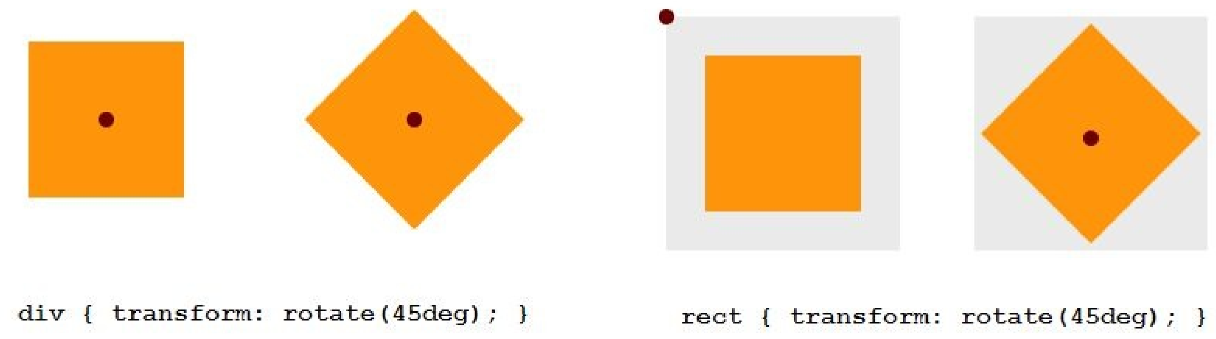
Setting the transform origin on an HTML element is straightforward: Any value you specify will be set relative to the element's border box.

In SVG, the transform origin can be set using either a percentage value or an absolute value (for example, pixels). If you specify a transform-origin value in percentages, then the value will be set relative to the element’s bounding box, which includes the stroke used to draw its border. If you specify the transform origin in absolute values, then it will be set relative to the SVG canvas’ current coordinate system of the user.

If we were to set the transform origin of the `<div>` and `<rect>` from the previous example to the center using percentage values, we would do this:

```
<!DOCTYPE html>
<style>
  div, rect {
  transform-origin: 50% 50%;
}
</style>
```

The resulting transformation would look like so:


div { transform: rotate(45deg); }
rect { transform: rotate(45deg); }

(View large version[25])

That being said, at the time of writing, setting the transform origin in percentage values currently does not work in Firefox. This is a known bug[26]. So, for the time being, your best bet is to use absolute values so that the transformations behave as expected. You can still use percentage values for WebKit browsers, though.

In the following example, we have a pinwheel on a stick that we'll rotate using CSS animation. To have the wheel rotate around its own center, we'll set its transform origin in pixels and percentages:

```
<svg>
<style>
.wheel {
  transform-origin: 193px 164px;
  -webkit-transform-origin: 50% 50%;
  -webkit-animation: rotate 4s cubic-bezier(.49,.05,.32,1.04)
infinite alternate;
  animation: rotate 4s cubic-bezier(.49,.05,.32,1.04) infinite
alternate;
}

@-webkit-keyframes rotate {
  50% {
    -webkit-transform: rotate(360deg);
  }
}

@keyframes rotate {
  50% {
    transform: rotate(360deg);
  }
}
```

```
</style>
<!-- SVG content -->
</svg>
```

You can check out the live result on Codepen[27]. Note that, at the time of writing, CSS 3D transformations are *not* hardware-accelerated when used on SVG elements; they have the same performance profile as SVG transform attributes. However, Firefox *does* accelerate transforms on SVGs to some extent.

Animating SVG Paths

There is no way to animate an SVG path from one shape to another in CSS. If you want to *morph* paths — that is, animate from one path to another — then you will need to use JavaScript for the time being. If you do that, I recommend using Snap.svg[28] by Dmitry Baranovskiy, the same person behind the SVG library Raphaël[29].

Snap.svg is described as being to SVG what jQuery is to HTML, and it makes dealing with SVGs and its quirks a lot easier.

That being said, you could create an animated line-drawing effect using CSS. The animation would require you to know the total length of the path you're animating and then to use the `stroke-dashoffset` and `stroke-dasharray` SVG properties to achieve the drawing effect. Once you know the length of the path, you can animate it with CSS using the following rules:

```
#path {
stroke-dasharray: pathLength;
stroke-dashoffset: pathLength;
/* transition stroke-dashoffset */
transition: stroke-dashoffset 2s linear;
}

svg:hover #path{
  stroke-dashoffset: 0;
}
```

In the example above, the path is drawn over the course of two seconds when the SVG is hovered over.

In the next demo, we'll use the same technique and then use a CSS transition — with a delay — to light up the bulb once the path's animation ends.

```
#cable {
  stroke: #FFF2B1;
  stroke-dasharray: 4000 4000;
  stroke-dashoffset: 4000;
  stroke-width: 4;
  transition: stroke-dashoffset 8s linear;
}

svg:hover #cable {
  stroke-dashoffset: 0;
}
```

```
/* turn lamp on */
.inner-lamp{
  fill:grey;
  transition: fill .5s ease-in 6s;
}

svg:hover .inner-lamp {
  fill: #FBFFF8;
}
/* … */
```

You can view the live demo on JS Bin[30]. Note that you can also write `stroke-dasharray: 4000;` instead of `stroke-dasharray: 4000 4000` — if the two line and gap values are equal, then you can specify only one value to be applied to both.

Sometimes, you might not know the exact length of the path to animate. In this case, you can use JavaScript to retrieve the length of the path using the `getTotalLength()` method:

```
var path = document.querySelector('.drawing-path');
path.getTotalLength();
//set CSS properties up
path.style.strokeDasharray = length;
path.style.strokeDashoffset = length;
//set transition up
path.style.transition = 'stroke-dashoffset 2s ease-in-out';
// animate
path.style.strokeDashoffset = '0';
```

The snippet above is a very simplified example showing that you can do the same thing we did with CSS but using JavaScript.

Jake Archibald has written an excellent article explaining the technique[31] in more detail. Jake includes a nice interactive demo that makes it easy to see exactly what's going on in the animation and how the two SVG properties work together to achieve the desired effect. I recommend reading his article if you're interested in learning more about this technique.

Embedding SVGs

An SVG can be embedded in a document in six ways, each of which has its own pros and cons.

The reason we're covering embedding techniques is because the way you embed an SVG will determine whether certain CSS styles, animations and interactions will work once the SVG is embedded.

An SVG can be embedded in any of the following ways:

1. as an image using the `<img>` tag:
   ```
   <img src="mySVG.svg" alt="" />
   ```

2. as a background image in CSS:
   ```
   .el {background-image: url(mySVG.svg);}
   ```

3. as an object using the <object> tag:
   ```
   <object type="image/svg+xml" data="mySVG.svg"><!-- fallback
   here --></object>
   ```

4. as an iframe using an <iframe> tag:
   ```
   <iframe src="mySVG.svg"><!-- fallback here --></iframe>
   ```

5. using the <embed> tag:
   ```
   <embed type="image/svg+xml" src="mySVG.svg" />
   ```

6. inline using the <svg> tag:
   ```
   <svg version="1.1" xmlns="http://www.w3.org/2000/svg" …>
   <!-- svg content -->
   </svg>
   ```

The `<object>` tag is the primary way to include an external SVG file. The main advantage of this tag is that there is a standard mechanism for providing an image (or text) fallback in case the SVG does not render. If the SVG cannot be displayed for any reason — such as because the provided URI is wrong — then the browser will display the content between the opening and closing `<object>` tags.

```
<object type="image/svg+xml" data="mySVG.svg">
  <img src="fallback-image.png" alt="…" />
</object>
```

If you intend using any advanced SVG features, such as CSS or scripting, then the HTML5 `<object>` tag is your best bet.

Because browsers can render SVG documents in their own right, embedding and displaying an SVG using an iframe is possible. This might be a good method if you want to completely separate the SVG code and script from your main page. However, manipulating an SVG image from your main page's JavaScript becomes a little more difficult and will be subject to the same-origin policy[32].

The `<iframe>` tag, just like the `<object>` tag, comes with a default way to provide a fallback for browsers that don't support SVG, or those that do support it but can't render it for whatever reason.

```
<iframe src="mySVG.svg">
  <img src="fallback-image.png" alt="…" />
</iframe>
```

The `<embed>` tag was never a part of any HTML specification, but it is still widely supported. It is intended for including content that needs an external plugin to work. The Adobe Flash plugin requires the `<embed>` tag, and supporting this tag is the only real reason for its use with SVG. The `<embed>` tag does not come with a default fallback mechanism.

An SVG can also be embedded in a document inline — as a “code island” — using the `<svg>` tag. This is one of the most popular ways to embed SVGs today. Working with inline SVG and CSS is a lot easier because the SVG can be styled and animated by targeting it with style rules placed anywhere in the document. That is, the styles don’t need to be included between the opening and closing `<svg>` tags to work; whereas this condition is necessary for the other techniques.

Embedding SVGs inline is a good choice, as long as you’re willing to add to the size of the page and give up backwards compatibility (since it does not come with a default fallback mechanism either). Also, note that an inline SVG cannot be cached.

An SVG embedded with an `<img>` tag and one embedded as a CSS background image are treated in a similar way when it comes to CSS styling and animation. Styles and animations applied to an SVG using an external CSS resource will not be preserved once the SVG is embedded.

The following table shows whether CSS animations and interactions (such as hover effects) are preserved when an SVG is embedded using one of the six embedding techniques, as compared to SVG SMIL animations[33]. The last column shows that, in all cases, SVG animations (SMIL) are preserved.

	CSS Interactions (e.g. `:hover`)	**CSS Animations**	**SVG Animations (SMIL)**
`<img>`	No	Yes only if inside `<svg>`	Yes
CSS background image	No	Yes only if inside `<svg>`	Yes
`<object>`	Yes only if inside `<svg>`	Yes only if inside `<svg>`	Yes
`<iframe>`	Yes only if inside `<svg>`	Yes only if inside `<svg>`	Yes
`<embed>`	Yes only if inside `<svg>`	Yes only if inside `<svg>`	Yes
`<svg>` (inline)	Yes	Yes	Yes

The behavior indicated in the table above is the standard behavior. However, implementations may differ between browsers, and bugs may exist.

Note that, even though SMIL animations will be preserved, SMIL interactions will not work for an SVG embedded as an image (i.e. `<img>` or via CSS).

Making SVGs Responsive

After embedding an SVG, you need to make sure it is responsive.

Depending on the embedding technique you choose, you might need to apply certain hacks and fixes to get your SVG to be cross-browser responsive. The reason for this is that the way browsers determine the dimensions of an SVG differs for some embedding techniques, and SVG implementations among browsers also differ. Therefore, the way SVG is handled is different and requires some style tweaking to make it behave consistently across all browsers.

I won't get into details of browser inconsistencies, for the sake of brevity. I will only cover the fix or hack needed for each embedding technique to make the SVG responsive in all browsers for that technique. For a detailed look at the inconsistencies and bugs, check out my article on Codrops[34].

Whichever technique you choose, the first thing you'll need to do is remove the `height` and `width` attributes from the root `<svg>` element.

You will need to preserve the `viewBox` attribute and set the `preserveAspectRatio` attribute to `xMidYMid meet` — if it isn't already set to that value. Note that you might not need to explicitly set `preserveAspectRatio` to `xMidYMid meet` at all because it will default to this value anyway if you don't change it.

When an SVG is embedded as a CSS background image, no extra fixes or hacks are needed. It will behave just like any other bitmap background image and will respond to CSS’ background-image properties as expected.

An SVG embedded using an `<img>` tag will automatically be stretched to the width of the container in all browsers (once the width has been removed from the `<svg>`, of course). It will then scale as expected and be fluid in all browsers except for Internet Explorer (IE). IE will set the height of the SVG to 150 pixels, preventing it from scaling correctly. To fix this, you will need to explicitly set the width to 100% on the `<img>`.

```
<img src="mySVG.svg" alt="SVG Description." />
img {
  width: 100%;
}
```

The same goes for an SVG embedded using an `<object>` tag. For the same reason, you will also need to set the width of the `<object>` to 100%:

```
object {
  width: 100%;
}
```

Even though `<iframe>` has a lot in common with `<object>`, browsers seem to handle it differently. For it, all browsers will default to the default size for replaced elements in CSS[35], which is 300 by 150 pixels.

The only way to make an iframe responsive while maintaining the aspect ratio of the SVG is by using the “padding hack” pioneered by Thierry Koblentz on A List Apart[36]. The idea behind the padding hack is to make use of the relationship of an element’s padding to its width in order to create an element with an intrinsic ratio of height to width.

When an element’s padding is set in percentages, the percentage is calculated relative to the width of the element, even when you set the top or bottom padding of the element.

To apply the padding hack and make the SVG responsive, the SVG needs to be wrapped in a container, and then you’ll need to apply some styles to the container and the SVG (i.e. the iframe), as follows:

```
<!-- wrap svg in a container -->
<div class="container">
  <iframe src="my_SVG_file.svg">
    <!-- fallback here -->
  </iframe>
</div>

.container {
  /* collapse the container's height */
  height: 0;

  /* specify any width you want (a percentage value, basically) */
  width: width-value;

  /* apply padding using the following formula */
  /* this formula makes sure the aspect ratio of the container
equals that of the SVG graphic */
  padding-top: (svg-height / svg-width) * width-value;
  position: relative;    /* create positioning context for SVG */
}
```

The `svg-height` and `svg-width` variables are the values of the height and width of the `<svg>`, respectively — the dimensions that we removed earlier. And the `width-value` is any width you want to give the SVG container on the page.

Finally, the SVG itself (the iframe) needs to be positioned absolutely inside the container:

```
iframe {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
}
```

We position the iframe absolutely because collapsing the container's height and then applying the padding to it would push the iframe beyond the boundaries of the container. So, to "pull it back up," we position it absolutely. You can read more about the details in my article on Codrops[37].

Finally, an SVG embedded inline in an `<svg>` tag becomes responsive when the height and width are removed, because browsers will assume a width of 100%

and will scale the SVG accordingly. However, IE has the same 150-pixel fixed-height issue for the `<img>` tag mentioned earlier; unfortunately, setting the width of the SVG to 100% is not sufficient to fix it this time.

To make the inline SVG fluid in IE, we also need to apply the padding hack to it. So, we wrap `<svg>` in a container, apply the padding-hack rules mentioned above to the container and, finally, position the `<svg>` absolutely inside it. The only difference here is that we do not need to explicitly set the height and width of `<svg>` after positioning it.

```
svg {
  position: absolute;
  top: 0;
  left: 0;
}
```

Using CSS Media Queries

SVG accepts and responds to CSS media queries as well. You can use media queries to change the styles of an SVG at different viewport sizes.

However, one important note here is that the viewport that the SVG responds to is the viewport of the SVG itself, not the page’s viewport, unless you are embedding the SVG inline in the document (using `<svg>`).

An SVG embedded with an `<img>`, `<object>` or `<iframe>` will respond to the viewport established by these elements. That is, the dimensions of these elements will form the viewport inside of which the SVG is to be drawn and, hence, will form the viewport to which the CSS media-query conditions will be applied. This is very similar in concept to element queries[38].

The following example includes a set of media queries inside an SVG that is then referenced using an `<img>` tag:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" viewBox="0 0
194 186">
  <style>
    @media all and (max-width: 50em) {
      /* select SVG elements and style them */
    }
    @media all and (max-width: 30em) {
      /* styles  */
    }
  </style>
  <!-- SVG elements here -->
</svg>
```

When the SVG is referenced, it will get the styles specified in the media queries above when the `<img>` has a `max-width` of `50em` or `30em`, respectively.

```
<img src="my-logo.svg" alt="Page Logo." />
```

You can learn more about media queries inside SVGs in Andreas Bovens's article for Dev.Opera[39].

Final Words

SVGs are images, and just as images can be accessible, so can SVGs. And making sure your SVGs are accessible is important, too.

I can’t emphasize this enough: Make your SVGs accessible. You can do several things to make that happen. For a complete and excellent guide, I recommend Leonie Watson’s excellent article on SitePoint[40]. Her tips include using the `<title>` and `<desc>` tags in the `<svg>`, using ARIA attributes and much more.

In addition to accessibility, don’t forget to optimize your SVGs and provide fallbacks for non-supporting browsers. I recommend Todd Parker’s presentation[41].

Last but not least, you can always check support for different SVG features on Can I Use[42]. I hope you’ve found this article to be useful. Thank you for reading. ❧

—

1. https://www.youtube.com/watch?v=lf7L8X6ZBu8
2. http://2014.cssconf.eu/
3. http://2014.fromthefront.it/
4. http://caniuse.com/#feat=svg
5. http://www.adobe.com/products/illustrator.html
6. https://inkscape.org/en/
7. http://bohemiancoding.com/sketch/
8. https://medium.com/@jm_denis/discovering-sketch-25545f6cb161
9. http://hackingui.com/design/sketch-design/why-i-moved-to-sketch/
10. http://www.sketchappsources.com/
11. http://creativedroplets.com/export-svg-for-the-web-with-illustrator-cc/

12. http://petercollingridge.appspot.com/svg-editor

13. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/10/02-svg-editor-large-preview-opt.png

14. https://github.com/svg/svgo

15. https://github.com/svg/svgo-gui

16. http://www.w3.org/TR/SVG/propidx.html

17. http://www.w3.org/TR/SVG2/styling.html#SVGStylingProperties

18. http://www.w3.org/TR/2008/REC-CSS2-20080411/selector.html#dynamic-pseudo-classes

19. http://www.w3.org/TR/2008/REC-CSS2-20080411/selector.html#q15

20. http://www.w3.org/TR/2008/REC-CSS2-20080411/generate.html

21. https://useiconic.com/

22. http://tutsplus.github.io/Styling-Iconic/styling/index.html

23. http://www.w3.org/TR/SVG11/styling.html#StylingWithCSS

24. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/10/05-transform-svg-html-large-preview-opt.png

25. http://media.mediatemple.netdna-cdn.com/wp-content/uploads/2014/10/06-transform-svg-html-large-preview-opt.png

26. https://bugzilla.mozilla.org/show_bug.cgi?id=891074

27. http://codepen.io/SaraSoueidan/pen/d0f94390e6c9af38fa562974399b6222?editors=100

28. http://snapsvg.io/

29. http://raphaeljs.com/

30. http://jsbin.com/haxaqa/1/edit?html,output

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

32. http://en.wikipedia.org/wiki/Same-origin_policy

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

34. http://tympanus.net/codrops/2014/08/19/making-svgs-responsive-with-css/

35. http://www.w3.org/TR/CSS2/visudet.html#inline-replaced-width

36. http://alistapart.com/article/creating-intrinsic-ratios-for-video/

37. http://tympanus.net/codrops/2014/08/19/making-svgs-responsive-with-css/

38. http://responsiveimagescg.github.io/eq-usecases/

39. https://dev.opera.com/blog/how-media-queries-allow-you-to-optimize-svg-icons-for-several-sizes/

40. http://www.sitepoint.com/tips-accessible-svg/

41. https://docs.google.com/presentation/d/1CNQLbqC0krocy_fZrM5fZ-YmQ2JgEADRh3qR6RbOOGk/pub?start=true&loop=false&delayms=5000#slide=id.p

42. http://caniuse.com/#search=svg

About The Authors

Christian Heilmann

An international Developer Evangelist working for Mozilla in the lovely town of London, England. Twitter: @codepo8[1].

Drew Thomas

Drew Thomas is the chief creative officer and a co-founder of Brolik[2], a Philadelphia digital agency. While Brolik is his focus, he also considers himself a “maker” and tinkers with all kinds of side projects, both digital and physical. Twitter: @drewbrolik[3].

Julian Shapiro

Julian Shapiro is a startup founder and a developer. His first startup, NameLayer, was acquired by Techstars. His current focus is advancing motion design on the web. Follow him for tweets on UI animation: @Shapiro[4].

Rachel Nabors

Rachel Nabors is an interaction developer and award-winning cartoonist. She travels the world, speaking about and training teams in the art of web animation. When not biking around her home city of Portland, she makes interactive comics at her company Tin Magpie[5]. You can catch her as @rachelnabors[6] on Twitter and at rachelnabors.com[7].

Sara Soueidan

Sara Soueidan is a freelance front-end web developer, consultant, author and speaker from Lebanon — focusing on HTML5, SVG, CSS, and JavaScript. She's a contributing author to the Smashing Book #5 and an author and team member at Codrops. She writes for various high-profile blogs and magazines including the Adobe Dream Weaver Blog, Opera Developers' Blog, Smashing Magazine, netmag, and CSS-Tricks, among others. You can find her writing on her blog[8], and follow her on Twitter @SaraSoueidan[9].

Stephen Greig

Stephen specialises in design and front-end development, which is convenient as he works as a Web Designer/Front-end guy out of Nottingham in the UK. Stephen has particular expertise in the more experimental and cutting edge CSS3 modules and is the author of extensive publication, CSS3 Pushing the Limits. Follow him on Twitter (@Stephen_Greig[10]) or head over to his personal blog (tangledindesign.com[11]) to learn more.

About Smashing Magazine

Smashing Magazine[12] is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy[13].

Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised. Since its emergence back in 2006 Smashing Magazine has proven to be a trustworthy online source.

—

1. http://www.twitter.com/codepo8

2. http://www.brolik.com

3. http://twitter.com/drewbrolik

4. https://twitter.com/shapiro

5. http://tinmagpie.com/

6. http://twitter.com/rachelnabors

7. http://rachelnabors.com/

8. http://sarasoueidan.com

9. http://twitter.com/SaraSoueidan

10. https://twitter.com/stephen_greig

12. http://www.tangledindesign.com

13. http://www.smashingmagazine.com

13. http://www.smashingmagazine.com/publishing-policy/

TABLE OF CONTENTS