Controlling CSS Animations and Transitions with JavaScript

css-tricks.com/controlling-css-animations-transitions-javascript/

Published September 17, 2013 by Chris Coyier

The following is a guest post by Zach Saucier. Zach wrote to me telling me that, as a frequenter on coding forums like Stack Overflow, he sees the questions come up all the time about controlling CSS animations with JavaScript, and proved it with a bunch of links. I've had this on my list to write about for way too long, so I was happy to let Zach dig into it and write up this comprehensive tutorial.

Web designers sometimes believe that animating in CSS is more difficult than animating in JavaScript. While CSS animation does have some limitations, most of the time it's more capable than we give it credit for! Not to mention, typically more performant.

Coupled with a touch of JavaScript, CSS animations and transitions are able to accomplish hardware-accelerated animations and interactions more efficiently than most JavaScript libraries.

Let's jump straight in!

Quick Note: Animations and Transitions are Different

Transitions in CSS are applied to an element and specify that when a property changes it should do so over gradually over over a period of time. Animations are different. When applied, they just run and do their thing. They offer more fine-grained control as you can control different stops of the animations.

In this article we will cover each of them separately.

Manipulating CSS Transitions

There are countless questions on coding forums related to triggering and pausing an element's transition. The solution is actually quite simple using JavaScript.

To trigger an element's transition, toggle a class name on that element that triggers it.

To pause an element's transition, use `getComputedStyle` and `getPropertyValue` at the point in the transition you want to pause it. Then set those CSS properties of that element equal to those values you just got.

The following is an example of that approach.

This same technique can be used in more advanced ways. The following example also triggers a transition by changing a class name, but this time a variable keeps track of the current zoom rate.

Note we're changing `background-size` this time. There are many different CSS properties that can be transitioned or animated, typically one that have numeric or color values. Rodney Rehm also wrote a particularly helpful and informational article on CSS transitions which can be found here.

Using CSS "Callback Functions"

Some of the most useful yet little-known JavaScript tricks for manipulating CSS transitions and animations are the DOM events they fire. Like: `animationEnd`, `animationStart`, and `animationIteration` for animations and `transitionEnd` for transitions. You might guess what they do. These animation events fire when the animation on an element ends, starts, or completes one iteration, respectively.

These events need to be vendor prefixed at this time, so in this demo, we use a function developed by Craig Buckler called `PrefixedEvent`, which has the parameters `element`, `type`, and `callback` to help make these events cross-browser. Here is his useful article on capturing CSS animations with JavaScript. And here is another one determining which animation (name) the event is firing for.

The idea in this demo is to enlarge the heart and stop the animation when it is hovered over.

The pure CSS version is jumpy. Unless you hover over it at the perfect time, it will jump to a particular state before

enlarging to the final hovered state. The JavaScript version is much smoother. It removes the jump by letting the animation complete before applying the new state.

Manipulating CSS Animations

Like we just learned, we can watch elements and react to animation-related events: animationStart, animationIteration, and animationEnd. But what happens if you want to change the CSS animation mid-animation? This requires a bit of trickery!

The animation-play-state Property

The animation-play-state property of CSS is incredibly helpful when you simply need to pause an animation and potentially continue it later. You can change that CSS through JavaScript like this (mind your prefixes):

```
element.style.webkitAnimationPlayState = "paused";
element.style.webkitAnimationPlayState = "running";
```

However, when a CSS animation is paused using animation-play-state, the element is prevented from transforming the same way it is when an animation is running. You can't pause it, transform it, resume it, and expect it to run fluidly from the new transformed state. In order to do that, we have to get a bit more involved.

Obtaining the Current Keyvalue Percentage

Unfortunately, at this time, there is no way to get the exact current "percentage completed" of a CSS keyframe animation. The best method to approximate it is using a setInterval function that iterates 100 times during the animation, which is essentially: the animation duration in ms / 100. For example, if the animation is 4 seconds long, then the setInterval needs to run every 40 milliseconds (4000/100).

```
var showPercent = window.setInterval(function() {
 if (currentPercent < 100) {
 currentPercent += 1;
 } else {
 currentPercent = 0;
 }
 // Updates a div that displays the current percent
 result.innerHTML = currentPercent;
}, 40);
```

This approach is far from ideal, because the function actually runs less frequently than every 40 milliseconds. I find that setting it to 39 milliseconds is more accurate, but relying on that is bad practice, as it likely varies by browser and is not a perfect fit on any browser.

Obtaining the Animation's Current CSS Property Values

In a perfect world, we would be able to select an element that's using a CSS animation, remove that animation, and give it a new one. It would then begin the new animation, starting from its current state. We don't live in that perfect world, so it's a bit more complex.

Below we have a demo to test a technique of obtaining and changing a CSS animation "mid stream", as it were. The animation moves an element in a circular path with the starting position being at the top center ("twelve o'clock", if you prefer) When the button is clicked, it should change the starting position of the animation to the element's current location. It travels the same path, only now "begins" at the location it was at when you pressed the button. This change of origin, and therefore change of animation, is indicated by changing the element's color to red in the first keyframe.

We need to get pretty deep to get this done! We're going to have to dig into the stylesheet itself to find the original animation.

You can access the stylesheets associated with a page by using document.styleSheets and iterate through it

using a for loop. The following is how you can use JavaScript to find a particular animation's values in a CSSKeyFrameRules object:

```
function findKeyframesRule(rule) {
 var ss = document.styleSheets;
 for (var i = 0; i < ss.length; ++i) {
 for (var j = 0; j < ss[i].cssRules.length; ++j) {
 if (ss[i].cssRules[j].type == window.CSSRule.WEBKIT_KEYFRAMES_RULE &&
 ss[i].cssRules[j].name == rule) {
 return ss[i].cssRules[j]; }
 }
 }
 return null;
}
```

Once we call the function above (e.g. `var keyframes = findKeyframesRule(anim)`), you can get the get the animation length of the object (the total number of how many keyframes there are in that animation) by using `keyframes.cssRules.length`. Then we need to strip the "%" from each of the keyframes so they are just numbers and JavaScript can use them as numbers. To do this, we use the following, which uses JavaScript's .map method.

```
// Makes an array of the current percent values
// in the animation
var keyframeString = [];
for(var i = 0; i < length; i ++)
{
 keyframeString.push(keyframes[i].keyText);
}

// Removes all the % values from the array so
// the getClosest function can perform calculations
var keys = keyframeString.map(function(str) {
 return str.replace('%', '');
});
```

At this point, `keys` will be an array of all of the animation's keyframes in numerical format.

Changing the Actual Animation (finally!)

In the case of our circular animation demo, we need two variables: one to track how many degrees the circle had traveled since its most recent start location, and another to track how many degrees it had traveled since the original start location. We can change the first variable using our `setInterval` function (using time elapsed and degrees in a circle). Then we can use the following code to update the second variable when the button is clicked.

```
totalCurrentPercent += currentPercent;
// Since it's in percent it shouldn't ever be over 100
if (totalCurrentPercent > 100) {
 totalCurrentPercent -= 100;
}
```

Then we can use the following function to find which keyframe of the animation is closest to the total current percent, based on the array of possible keyframe percents we obtained above.

```
function getClosest(keyframe) {
 // curr stands for current keyframe
 var curr = keyframe[0];
 var diff = Math.abs (totalCurrentPercent - curr);
 for (var val = 0, j = keyframe.length; val < j; val++) {
 var newdiff = Math.abs(totalCurrentPercent - keyframe[val]);
 // If the difference between the current percent and the iterated
 // keyframe is smaller, take the new difference and keyframe
 if (newdiff < diff) {
 diff = newdiff;
 curr = keyframe[val];
 }
 }
 return curr;
}
```

To obtain the new animation's first keyframe value to use in the calculations later on, we can use JavaScript's .indexOf method. We then delete the original keyframes so we can re-create new ones.

```
for (var i = 0, j = keyframeString.length; i < j; i ++) {
 keyframes.deleteRule(keyframeString[i]);
}
```

Next, we need to change the % into a degree of the circle. We can do this by simply multiplying the new first percentage by 3.6 (because 10 0* 3.6 = 360).

Finally we create the new rules based on the variables obtained above. The 45-degree difference between each rule is because we have 8 different keyframes that go around the circle. 360 (degrees in a circle) divided by 8 is 45.

```
// Prefix here as needed
keyframes.insertRule("0% {
 -webkit-transform: translate(100px,100px) rotate(" + (multiplier + 0) + "deg)
 translate(-100px,-100px) rotate(" + (multiplier + 0) + "deg);
 background-color:red;
}");
keyframes.insertRule("13% {
 -webkit-transform: translate(100px,100px) rotate(" + (multiplier + 45) + "deg)
 translate(-100px,-100px) rotate(" + (multiplier + 45) + "deg);
}");
...continued...
```

Then we reset the current percent setInterval so it can be run again. Note the above is WebKit prefixed. To make it more cross-browser compatible you could possibly do some UA sniffing to guess which prefixes would be needed:

```
// Gets the browser prefix
var browserPrefix;
navigator.sayswho= (function(){
 var N = navigator.appName, ua = navigator.userAgent, tem;
 var M = ua.match(/(opera|chrome|safari|firefox|msie)\/?\s*(\.?\d+(\.\d+)*)/i);
 if(M && (tem = ua.match(/version\/([\.\d]+)/i))!= null) M[2] = tem[1];
 M = M? [M[1], M[2]]: [N, navigator.appVersion,'-?'];
 M = M[0];
 if(M == "Chrome") { browserPrefix = "webkit"; }
 if(M == "Firefox") { browserPrefix = "moz"; }
 if(M == "Safari") { browserPrefix = "webkit"; }
 if(M == "MSIE") { browserPrefix = "ms"; }
})();
```

If you'd like to investigate further, Russell Uresti's answer in this StackOverflow post and the corresponding example are helpful.

Turning Animations into Transitions

As we've seen, manipulating CSS transitions can be simplified using JavaScript. If you don't end up getting the results you want with CSS animations, you can try making it into a transition instead and working with it that way. They are of about the same difficulty to code, but they may be more easily set and edited.

The biggest problem in turning CSS animations into transitions is when we turn animation-iteration into the equivalent transition command. Transition has no direct equivalent, which is why they are different things in the first place.

Relating this to our rotation demo, a little trick is to multiply both the transition-duration and the rotation by x. Then you need to have/apply a class to trigger the animation, because if you applies the changed properties directly to the element, well, there won't be much of a transition to be had. To kick off the transition (fake animation), you apply the class to the element.

In our example, we do it on page load:

Manipulating CSS Matrixes

Manipulating CSS animations can also be done through using a CSSMatrix. For example:

```
var translated3D =
 new WebKitCSSMatrix(window.getComputedStyle(elem, null).webkitTransform);
```

But the process can get confusing, especially to those just starting out using CSS animations.

Resetting CSS Animations

The trick to do this the correct way can be found here on CSS Tricks. The trick is essentially (if possible) remove the class that started the animation, trigger reflow on it somehow, then apply the class again. If all else fails, rip the element off the page and put it bak again.

Use Your Head

Before starting to code, thinking about and planning how a transition or animation should run is the best way to minimize your problems and get the effect you desire. Even better than Googling for solutions later! The techniques and tricks overviewed in this article may not always be the best way to create the animation your project calls for.

Here's a little example of where getting clever with HTML and CSS alone can solve a problem where you might have thought to go to JavaScript.

Say we want a graphic to rotate continuously and then switch rotational direction when hovered. Learning what was covered in this article, you might want to jump in and use an animationIteration event to change the animation. However, a more efficient and better-performing solution can be found using CSS and an added container element.

The trick would be to have the spiral rotate at x speed in one direction and, when hovered, make the parent element rotate at 2x speed in the opposite direction (starting at the same position). The two rotations working against each other creates a net effect of the spiral rotating the opposite direction.

The same concept was used in this example for a StackOverflow question.

Links!

Related stuff you may find interesting.

- Animo.js - "A powerful little tool for managing CSS animations"
- Thank God We Have A Specification! - Smashing Magazine article on transition quirks

In Summary

- getComputedStyle is helpful for manipulating CSS transitions.
- transitionEnd and its related events are quite helpful when manipulating CSS transitions and animations using JavaScript.
- Changing a CSS animation from its current values can be done by obtaining the stylesheets in JavaScript, but can be quite involved.
- In JavaScript, CSS transitions are generally easier to work with than CSS animations.
- CSS Matrices are generally a pain to deal with, especially for beginners.
- Thinking about what should be done and planning how to do it are essential in coding animations.