Building performant expand & collapse animations

By Paul Lewis
Paul is a Design and Perf Advocate

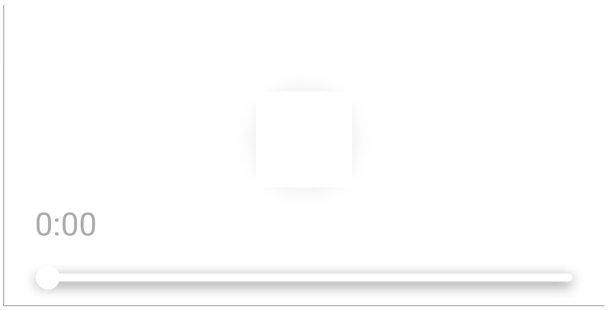
By Stephen McGruer
Stephen is a contributor to Web**Fundamentals**

TL;DR

Use scale transforms when animating clips. You can prevent the children from being stretched and skewed during the animation by counter-scaling them.

Previously we've posted updates on how to create performant parallax effects and infinite scrollers. In this post we're going to look over what's involved if you want performant clip animations. If you want to see a demo, check out the Sample UI Elements GitHub repo.

Take, for example, an expanding menu:



Some options for building this are more performant than others.

Bad: Animating width and height on a container element

You could imagine using a bit of CSS to animate the width and height on the container element.

```css
.menu {
  overflow: hidden;
  width: 350px;
  height: 600px;
  transition: width 600ms ease-out, height 600ms ease-out;
}

.menu--collapsed {
  width: 200px;
  height: 60px;
}
```

The immediate problem with this approach is that it requires animating **`width`** and **`height`**. These properties require calculating layout and paint the results on every frame of the animation, which can be very expensive, and will typically cause you to miss out on 60fps. If that's news to you then read our Rendering Performance guides, where you can get more information on how the rendering process works.

Bad: Use the CSS clip or clip-path properties.

An alternative to animating **`width`** and **`height`** might be to use the (now-deprecated) **`clip`** property to animate the expand and collapse effect. Or, if you prefer, you could use **`clip-path`** instead. Using **`clip-path`**, however, is less well supported than **`clip`**. But **`clip`** is deprecated. Right. But don't despair, this isn't the solution you wanted anyway!

```css
.menu {
  position: absolute;
  clip: rect(0px 112px 175px 0px);
  transition: clip 600ms ease-out;
}

.menu--collapsed {
  clip: rect(0px 70px 34px 0px);
}
```

While better than animating the **`width`** and **`height`** of the menu element, the downside of this approach is that it still triggers paint. Also the **`clip`** property, should you go that route, requires that the element it's operating on is either absolutely or fixed positioned, which can require a little extra wrangling.

Good: animating scales

Since this effect involves something getting bigger and smaller, you can use a scale transform. This is great news because changing transforms is something that doesn't require layout or paint, and which the browser can hand off to the GPU, meaning that the effect is accelerated and significantly more likely to hit 60fps.

The downside to this approach, like most things in rendering performance, is that it requires a bit of setting up. It's totally worth it, though!

Step 1: Calculate the start and end states

With an approach that uses scale animations, the first step is to read elements that tell you the size the menu needs to be both when it's collapsed, and when it's expanded. It may be that for some situations you can't get both of these bits of information in one go, and that you need to — say — toggle some classes around to be able to read the various states of the component. If you need to do that, however, be cautious: `getBoundingClientRect()` (or `offsetWidth` and `offsetHeight`) forces the browser to run styles and layout passes if styles have changed since they were last run.

```
function calculateCollapsedScale () {
  // The menu title can act as the marker for the collapsed state.
  const collapsed = menuTitle.getBoundingClientRect();

  // Whereas the menu as a whole (title plus items) can act as
  // a proxy for the expanded state.
  const expanded = menu.getBoundingClientRect();
  return {
    x: collapsed.width / expanded.width,
    y: collapsed.height / expanded.height
  }
}
```

In the case of something like a menu, we can make the reasonable assumption that it will start out being its natural scale (1, 1). This natural scale represents its expanded state meaning you will need to animate from a scaled down version (which was calculated above) back up to that natural scale.

But wait! Surely this would scale the contents of the menu as well, wouldn't it? Well, as you can see below, yes.

So what can you do about this? Well you can apply a *counter*-transform to the contents, so for example if the container is scaled down to 1/5th of its normal size, you can scale the contents *up* by 5x to prevent the contents being squashed. There are two things to notice about that:

1. **The counter-transform is also a scale operation**. This is *good* because it can also be accelerated, just like the animation on the container. You may need to ensure that the elements being animated get their own compositor layer (enabling the GPU to help out), and for that you can add `will-change: transform` to the element or, if you need to support older browsers, `backface-visiblity: hidden`.
2. **The counter-transform must be calculated per frame.** This is where things can get a little trickier, because assuming that the animation is in CSS and uses an easing function, the easing itself need to be countered when animating the counter-transform. However, calculating the inverse curve for — say — `cubic-bezier(0, 0, 0.3, 1)` isn't all that obvious.

It may be tempting, then, to consider animating the effect using JavaScript. After all, you could then use an easing equation to calculate the scale and counter-scale values per frame. The downside of any JavaScript-based animation is what happens when the main thread (where your JavaScript runs) is busy with some other task. The short answer is that your animation can stutter or halt altogether, which isn't great for UX.

Step 2: Build CSS Animations on the fly

The solution, which may appear odd at first, is to create a keyframed animation with our own easing function dynamically and inject it into the page for use by the menu. (Big thanks to Chrome engineer Robert Flack for pointing this out!) The primary benefit of this is that a keyframed animation that mutates transforms can be run on the compositor, meaning that it isn't affected by tasks on the main thread.

To make the keyframe animation we step from 0 to 100 and calculate what scale values would be needed for the element and its contents. These can then be boiled down to a string, which can be injected into the page as a style element. Injecting the styles will cause a Recalculate Styles pass on the page, which is additional work that the browser has to do, but it will do it only once when the component is booting up.

```
function createKeyframeAnimation () {
  // Figure out the size of the element when collapsed.
  let {x, y} = calculateCollapsedScale();
  let animation = '';
  let inverseAnimation = '';

  for (let step = 0; step <= 100; step++) {
    // Remap the step value to an eased one.
    let easedStep = ease(step / 100);

    // Calculate the scale of the element.
    const xScale = x + (1 - x) * easedStep;
    const yScale = y + (1 - y) * easedStep;

    animation += `${step}% {
      transform: scale(${xScale}, ${yScale});
    }`;

    // And now the inverse for the contents.
    const invXScale = 1 / xScale;
    const invYScale = 1 / yScale;
    inverseAnimation += `${step}% {
      transform: scale(${invXScale}, ${invYScale});
    }`;

  }

  return `
  @keyframes menuAnimation {
    ${animation}
  }

  @keyframes menuContentsAnimation {
    ${inverseAnimation}
  }`;
}
```

The endlessly curious may be wondering about the `ease()` function inside the for-loop. You can use something like this to map values from 0 to 1 to an eased equivalent.

```
function ease (v, pow=4) {
  return 1 - Math.pow(1 - v, pow);
}
```

You can use Google search to plot what that looks like%20from%200%20to%201) as well. Handy! If you're in need of other easing equations do check out Tween.js by Soledad Penadés, which contains a whole heap of them.

Step 3: Enable the CSS Animations

With these animations created and baked out to the page in JavaScript, the final step is to toggle classes enabling the animations.

```
.menu--expanded {
  animation-name: menuAnimation;
  animation-duration: 0.2s;
  animation-timing-function: linear;
}

.menu__contents--expanded {
  animation-name: menuContentsAnimation;
  animation-duration: 0.2s;
  animation-timing-function: linear;
}
```

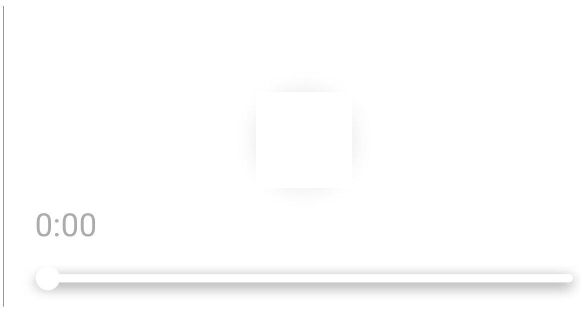
This causes the animations to run that were created in the previous step. Because the baked animations are already eased, the timing function needs to be set to `linear` otherwise you'll ease between each keyframe which will look very weird!

When it comes to collapsing the element back down there are two options: update the CSS animation to run in reverse rather than forwards. This will work just fine, but the "feel" of the animation will be reversed, so if you used an ease-out curve the reverse will feel eased *in*, which will make it feel sluggish. A more appropriate solution is to create a *second pair* of animations for collapsing the element. These can be created in exactly the same way as the expand keyframe animations, but with swapped start and end values.

```
const xScale = 1 + (x - 1) * easedStep;
const yScale = 1 + (y - 1) * easedStep;
```

A more advanced version: circular reveals

It's also possible to use this technique to make circular expand and collapse animations.



The principles are largely the same as the previous version, where you scale an element, and counter-scale its immediate children. In this case the element that's scaling up has a `border-radius` of 50%, making it circular, and is wrapped by *another* element that has `overflow: hidden`, meaning that you don't see the circle expand outside of the element bounds.

A word of warning on this particular variant: Chrome has blurry text on low DPI screens during the animation because of rounding errors due to the scale and counter-scale of the text. If you're interested in the details for that there's a bug filed that you can star and follow.

The code for the circular expand effect can be found in the GitHub repo.

Conclusions

So there you have it, a way to do performant clip animations using scale transforms. In a perfect world it would be great to see clip animations be accelerated (there's a Chromium bug for that made by Jake Archibald), but until we get there you should be cautious when animating `clip` or `clip-path`, and definitely avoid animating `width` or `height`.

It would also be handy to use Web Animations for effects like this, because they have a JavaScript API but can run on the compositor thread if you only animate `transform` and `opacity`. Unfortunately support for Web Animations isn't great, though you could use progressive enhancement to use them if they're available.

```
if ('animate' in HTMLElement.prototype) {
  // Animate with Web Animations.
} else {
  // Fall back to generated CSS Animations or JS.
}
```

Until that changes, while you *can* use JavaScript-based libraries to do the animation, you might find that you get more reliable performance by baking a CSS animation and using that instead. Equally, if your app already relies on JavaScript for its animations you may be better served by being at least consistent with your existing codebase.

If you want to have a look through the code for this effect take a look at the UI Element Samples Github repo and, as always, let us know how you get on in the comments below.

Last updated July 2, 2018.