Examples of high-performance CSS animations

- Wizard loading animation
  - Inspect the animation with Chrome DevTools
  - How it works
- Pulsating Circle
  - Inspect the animation with Firefox DevTools
- Pure CSS 3D Sphere
- Conclusion

In this post find out how some popular animations found on CodePen have been created. These animations all use the performant techniques discussed in other articles in this section.

See Why are some animations slow? to learn the theory behind these recommendations, and the Animations Guide for practical tips.

Wizard loading animation

https://codepen.io/Craaftx/embed/ExyYRMK?height=458&theme-id=light&default-tab=result

View Wizard loading animation on CodePen

This loading animation is built entirely with CSS. The image plus all of the animation has been created in CSS and HTML, no images or JavaScript. To understand how it was created and how well it performs you can use Chrome DevTools.

Inspect the animation with Chrome DevTools

With the animation running, open the Performance tab in Chrome DevTools and record a few seconds of the animation. You should see in the Summary that the browser is not doing any Layout or Paint operations when running this animation.

Summary in DevTools

The summary after profiling the wizard animation.

To find out how this animation was achieved without causing layout and paint, inspect any of the moving elements in Chrome DevTools. You can use the **Animations Panel** to locate the various

animated elements, clicking on any element will highlight it in the DOM.

The Animations Panel showing the various parts of our animation.

Viewing and selecting items in the Chrome DevTools Animation Panel.

For example select the triangle, and watch how the box of the element transforms during its journey into the air, as it spins, and then returns to the start position.

Video showing how we can track the path of the triangle in Chrome DevTools.

With the element still selected look at the Styles Panel. There you can see the CSS which draws the shape of the triangle, and the animation being used.

How it works

The triangle is created by using the `::after` pseudo-element to add generated content, using borders to create the shape.

```
.triangle {
    position: absolute;
    bottom: -62px;
    left: -10px;
    width: 110px;
    height: 110px;
    border-radius: 50%;
}

.triangle::after {
    content: "";
    position: absolute;
    top: 0;
    right: -10px;
    width: 0;
    height: 0;
    border-style: solid;
    border-width: 0 28px 48px 28px;
    border-color: transparent transparent #89beb3 transparent;
}
```

You can find out more about making shapes with borders and generated content in The Shapes of CSS.

The animation is added with the following line of CSS,

```
animation: path_triangle 10s ease-in-out infinite;
```

Staying in Chrome DevTools you can find the keyframes by scrolling down the Style Panel. There you will find that the animation is created by using `transform` to change the position of the element and rotate it. The `transform` property is one of the properties described in the Animations Guide, which does not cause the browser to do layout or paint operations (which are the main causes of slow animations).

```
@keyframes path_triangle {
  0% {
    transform: translateY(0);
  }
  10% {
    transform: translateY(-172px) translatex(10px) rotate(-10deg);
  }
  55% {
    transform: translateY(-172px) translatex(10px) rotate(-365deg);
  }
  58% {
    transform: translateY(-172px) translatex(10px) rotate(-365deg);
  }
  63% {
    transform: rotate(-360deg);
  }
}
```

Each of the different moving parts of this animation uses similar techniques. The result is a complex animation which runs smoothly.

Pulsating Circle

https://codepen.io/peeke/embed/BjxXZa?height=458&theme-id=light&default-tab=result

View Pulsating Circle on CodePen

This type of animation is sometimes used to draw attention to something on a page. To understand the animation you can use Firefox DevTools.

Inspect the animation with Firefox DevTools

With the animation running, open the Performance tab in Firefox DevTools and record a few seconds of the animation. Stop the recording, in the waterfall you should see that there are no entries for **Recalculate Style**. You now know that this animation does not cause style recalculation, and therefore layout and paint operations.

details of the animation in the Firefox Waterfall

The Firefox DevTools Waterfall.

Staying in Firefox DevTools inspect the circle to see how this animation works. The `<div>` with a class of `pulsating-circle` marks the position of the circle, however it does not draw a circle itself.

```
.pulsating-circle {
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translateX(-50%) translateY(-50%);
    width: 30px;
    height: 30px;
}
```

The visible circle and animations are achieved using the `::before` and `::after` pseudo-elements.

The `::before` element creates the opaque ring that extends outside of the white circle, using an animation called `pulse-ring`, which animates `transform: scale`, and `opacity`.

```
.pulsating-circle::before {
    content: '';
    position: relative;
    display: block;
    width: 300%;
    height: 300%;
    box-sizing: border-box;
    margin-left: -100%;
    margin-top: -100%;
    border-radius: 45px;
    background-color: #01a4e9;
    animation: pulse-ring 1.25s cubic-bezier(0.215, 0.61, 0.355, 1) infinite;
}

@keyframes pulse-ring {
  0% {
    transform: scale(0.33);
  }
  80%, 100% {
    opacity: 0;
  }
}
```

Another way to see which properties are being animated is to select the **Animations** panel in Firefox DevTools. You will then see a visualization of the animations being used, and the properties that are being animated.

With the ::before pseudo-element selected we can see which properties are animating.

The white circle itself is created and animated using the `::after` pseudo-element. The animation `pulse-dot` uses `transform: scale` to grow and shrink the dot during the animation.

```
.pulsating-circle::after {
  content: '';
  position: absolute;
  left: 0;
  top: 0;
  display: block;
  width: 100%;
  height: 100%;
  background-color: white;
  border-radius: 15px;
  box-shadow: 0 0 8px rgba(0, 0, 0, 0.3);
  animation: pulse-dot 1.25s cubic-bezier(0.455, 0.03, 0.515, 0.955) -0.4s infinite;
}

@keyframes pulse-dot {
  0% {
    transform: scale(0.8);
  }
  50% {
    transform: scale(1);
  }
  100% {
    transform: scale(0.8);
  }
}
```

An animation like this could be used in various places in your application, it's important that these small touches don't impact the overall performance of your app.

Pure CSS 3D Sphere

https://codepen.io/iamlark/embed/jYzYJg?height=458&theme-id=light&default-tab=result

View Pure CSS 3D Sphere on CodePen

This animation seems incredibly complicated, however it uses techniques that we have already seen in the previous examples. The complexity comes from animating a large number of elements.

Open Chrome DevTools and select one of the elements with a class of `plane`. The sphere is made up of a set of rotating planes and spokes.

> The plane appears to be rotating.

The DOM Search Tool in Chrome DevTools can make it easier to find an element that you want to inspect.

These planes and spokes are inside a wrapper `<div>`, and it is this element which is rotating using `transform: rotate3d`.

```
.sphere-wrapper {
  transform-style: preserve-3d;
  width: 300px;
  height: 300px;
```

```
  position: relative;
  animation: rotate3d 10s linear infinite;
}

@keyframes rotate3d {
  0% {
    transform: rotate3d(1, 1, 1, 0deg);
  }
  25% {
    transform: rotate3d(1, 1, 1, 90deg);
  }
  50% {
    transform: rotate3d(1, 1, 1, 180deg);
  }
  75% {
    transform: rotate3d(1, 1, 1, 270deg);
  }
  100% {
    transform: rotate3d(1, 1, 1, 360deg);
  }
}
```

The dots can be found nested inside the `plane` and `spoke` elements, they use an animation which uses transform to scale and translate them. This creates the pulsing effect.

The dot rotates with the sphere and pulses.

```
.spoke-15 .dot,
.spoke-21 .dot {
  animation: pulsate 0.5s infinite 0.83333333s alternate both;
  background-color: #55ffee;
}

@-webkit-keyframes pulsate {
  0% {
    transform: rotateX(90deg) scale(0.3) translateZ(20px);
  }
  100% {
    transform: rotateX(90deg) scale(1) translateZ(0px);
  }
}
```

The work involved in creating this animation has been to get the timing right, to create the turning and pulsing effect. The animations themselves are quite straightforward, and use methods which perform very well.

You can see how this animation performs by opening Chrome DevTools and recording Performance while it is running. After the initial load, the animation is not triggering Layout or Paint, and runs smoothly.

Conclusion

From these examples you can see how animating a few properties using performant methods can create some very cool animations. By defaulting to the performant methods described in the Animations guide you can spend your time creating the effect you want, with fewer concerns about making the page slow.

□□□ #3
□ □□□ □□□ 2026-10-31 21:34:32 UTC
□ cocobook-sync □□□ 2026-11-04 17:36:24 UTC