Animated Sparkles in React

Filed under **React** *on* May 19th, 2020

On the web, we have two semantic tags we can use when we want to indicate that part of a sentence is particularly significant: `<strong>` and `<em>`.

- The `<strong>` element is meant for things "of great seriousness or urgency", like warnings, **according to MDN**.
- The `<em>` element is meant to indicate verbal stress, so that your internal narrator can accurately reproduce the message and infer the correct meaning. For example: "Why would *Jerry* bring anything?", and "Why would Jerry *bring* anything?"*

These two elements are often used together to emphasize dire warnings or critically serious situations: *Do not* **open the window on the space ship,** *you will be sucked into space.*

But what about when we want to emphasize something positive? In our non-digital lives, we can use tone, timing, and body language to express all kinds of emotions. It kinda sucks that our only tools on the web are associated with stressful and serious situations.

For my blog, I want more expressiveness than these tags alone can offer. So I'm working on some new ones*. I started with this ***spicy cursive variant***, and I recently added a second: **Sparkly text**.

The sparkles indicate that something is new and shiny, or something that has captured my affection. It's meant to increase prominence, but in a positive way!

We can use it on more than just text, as well. Here are some examples:

Rainbow text	On an image	Twinkle button
Rainbow!		**Go!**

Today we're going to explore how this was built, so that you can have sparkly text in your project as well!

Intended audience

This blog post is written for folks who are comfortable with React (including custom hooks). It also helps to have a bit of animation experience, though it shouldn't be necessary.

Planning it out

From an interface perspective, I imagine it working like this:

```
{/* It should be able to wrap images */}
<Sparkles>
  <img src="gold-cat.png" alt="A golden cat statue" />
</Sparkles>

{/* It should also be able to wrap inline text */}
<p>
  Next race: <Sparkles>Rainbow Road</Sparkles>.
</p>
```

Each sparkle will be its own HTML element, a `span`. We'll have some sort of loop that adds a couple elements a second. Each element will use keyframe animations to twinkle: a combination of scaling and rotating.

If we're not careful, we'll wind up polluting the DOM with a bunch of stale sparkle elements, so we'll also need to clean up after ourselves; we'll do periodic garbage collection, and remove nodes that have finished twinkling.

Finally, we'll position each sparkle randomly within the box formed by the child element.

With this game plan in mind, let's start building! First, we need a sparkle asset.

Creating an asset

In this tutorial, we'll use the following asset:

It's an SVG, not a JPG or PNG. We want it to be an SVG so that we can dynamically alter it: by using inline SVG elements, we can change the `fill` color using JS!

There are many ways to get an appropriate SVG:

- Search **the Noun Project** for one.
- **Download** the one I created (right-click and "Save As..."). It's released under Creative Commons Zero, which means you can do whatever you want with it, without attribution.
- Make your own! I created a **one minute tutorial** for how to create this sparkle in Figma.

I use Figma for all of my illustration needs. It's an incredible, free piece of software. It's cross-platform, and comes in desktop and web-app varieties (it's even built with React!).

Learning Figma as a Developer

I am not a designer, and I'd say that my Figma skills are somewhere between "beginner" and "intermediate". I am absolutely not proficient with it in the way that a professional designer would be.

Even with rudimentary skills, though, it sometimes feels like a super power. Being able to quickly come up with my own assets has helped me so many times on my side-projects. Not to mention how quick it is to prototype! I'd highly recommend taking some time to familiarize yourself with it, or another design tool.

Generating Sparkles

We need a function that will create a new “sparkle” instance.

Each sparkle should have an ID, a random size, and a random position. Here's a first pass:

```
// Default color is a bright yellow
const DEFAULT_COLOR = 'hsl(50deg, 100%, 50%)';

const generateSparkle = (color = DEFAULT_COLOR) => {
  return {
    id: String(random(10000, 99999)),
    createdAt: Date.now(),
    // Bright yellow color:
    color,
    size: random(10, 20),
    style: {
      // Pick a random spot in the available space
      top: random(0, 100) + '%',
      left: random(0, 100) + '%',
      // Float sparkles above sibling content
      zIndex: 2,
    },
  }
}
```

The `random` function is a utility that generates a random number within a range. View full snippet.

random position. We use percentages for layout since we don't actually know the width and height of our container.

We'll create a new `SparkleInstance` component, which will consume some of this data to render a sparkle.

Earlier, we created an illustration in Figma:

We can export this as an SVG, and wind up with something that looks like this:

```
<svg width="160" height="160" viewBox="0 0 160 160" fill="none"
xmlns="http://www.w3.org/2000/svg">
  <path d="M80 0C80 0 84.2846 41.2925 101.496 58.504C118.707
75.7154 160 80 160 80C160 80 118.707 84.2846 101.496
101.496C84.2846 118.707 80 160 80 160C80 160 75.7154 118.707
58.504 101.496C41.2925 84.2846 0 80 0 80C0 80 41.2925 75.7154
58.504 58.504C75.7154 41.2925 80 0 80 0Z" fill="#FFC700"></path>
</svg>
```

The nice thing about SVGs is that they're already almost JSX! We can use a nifty tool like **svg2jsx** to tweak the handful of small details that need to change. We'll use this SVG as the basis for a new React component, `SparkleInstance`:

```
function SparkleInstance({ color, size, style }) {
  return (
    <Svg
      width={size}
      height={size}
      viewBox="0 0 160 160"
      fill="none"
      style={style}
    >
      <path
        d="all that stuff from before"
        fill={color}
      />
    </Svg>
  );
}

const Svg = styled.svg`
  position: absolute;
  pointer-events: none;
  z-index: 2;
`;
```

Every sparkle instance will have its own color, size, and position, so these become props for our new component. Previously-fixed values in our SVG become dynamic, powered by props.

I've wrapped the `svg` in a styled-component, `Svg`. This lets us add some baseline styles for our sparkle.

These examples use **styled-components**, but this tutorial isn't specific to any styling solution. You can use whichever CSS tool you're already using.

sit it next to whatever `children` we've passed it:

```
function Sparkles({ children }) {
  const sparkle = generateSparkle();

  return (
    <Wrapper>
      <SparkleInstance
        color={sparkle.color}
        size={sparkle.size}
        style={sparkle.style}
      />
      <ChildWrapper>
        {children}
      </ChildWrapper>
    </Wrapper>
  );
}

const Wrapper = styled.span`
  position: relative;
  display: inline-block;
`;

const ChildWrapper = styled.strong`
  position: relative;
  z-index: 1;
  font-weight: bold;
`;
```

To review:

- We've created a single `SparkleInstance` with a random size and position.
- We've given it a z-index of 2.
- Our `children` are wrapped in a `ChildWrapper`, which is a `strong` tag with a z-index of 1.
- Both of these elements are wrapped within a `Wrapper`.

With this done, we have a sparkle being generated haphazardly above our wrapped element! Click/tap the button to generate a random new sparkle:

Regenerate

Notice how you can still trigger the button even if you click/tap right where the sparkle is. This is because we added `pointer-events: none` — Our clicks/taps pass right through it!

1. Add the animation, so that each sparkle appears to twinkle.
2. Periodically generate and clean up sparkles.

Twinkling animation

We want our sparkles to change in two ways:

- It should rotate, relatively slowly
- It should grow and shrink

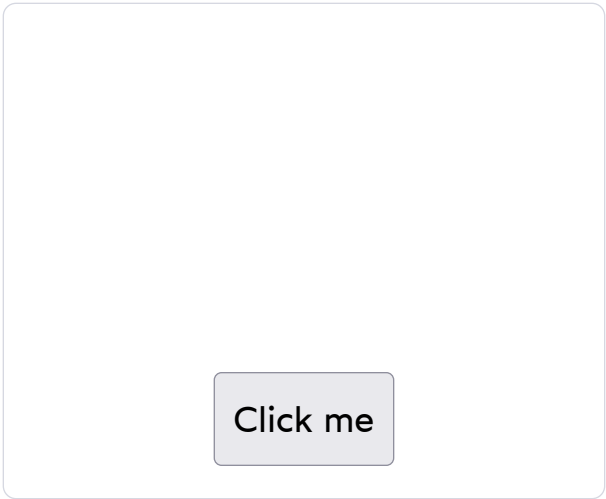
The `transform` property can help us with both of these goals. As a first stab, we can do something like this:

```
// This "keyframes" function is from styled-components,
// and it generates a CSS `@keyframe` at-rule under the hood.
const sparkleAnimation = keyframes`
  0% {
    transform: scale(0) rotate(0deg);
  }
  50% {
    transform: scale(1) rotate(90deg);
  }
  100% {
    transform: scale(0) rotate(180deg);
  }
`;

const Svg = styled.svg`
  position: absolute;
  animation: ${sparkleAnimation} 600ms forwards;
`;
```

I'm using a lot of pretty advanced keyframe animation stuff here. If you're not sure what's going on, I have a blog post that digs into all this stuff: **"An Interactive Guide to Keyframe Animations"**.

Our animation starts at `scale(0)`, which means it's shrunk down to the point that it's invisible (0x its normal size). At the 50% mark, we've grown to its full size (1x), and rotated it 90 degrees. By the time the animation completes, we've rotated another 90 degrees, and shrunk back down to 0x size.

Let's see what this animation looks like. Click the trigger to generate a sparkle (I've blown it up so that we can see the effect clearly).



🤔 This isn't super twinkly, is it? I see two problems:

1. Each step is eased, so you wind up with a jerky 2-step animation; it sorta pauses in the middle, since it's easing to the 50% keyframe.
2. There are two properties being tweened—rotation and scale—and they're happening in total lockstep. I want these properties to be handled separately, so that their timing and easing can be independently controlled.

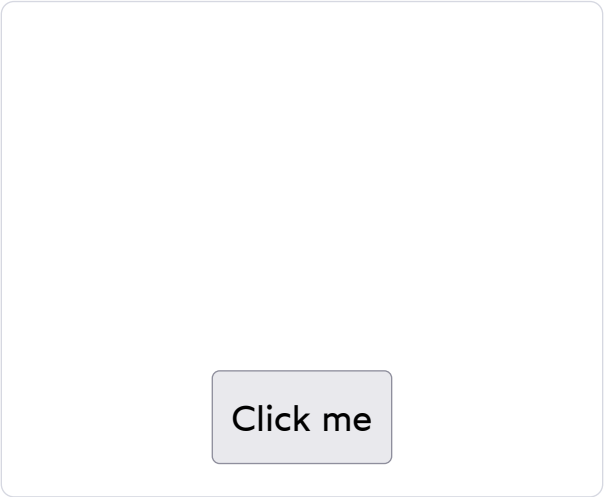
The first thing we need to do is separate out the animations: I want to be able to control the scale and rotation separately. In order for this to work, I need a wrapping div: you ***can*** put multiple keyframe animations on an element, but not if they both modify the same property. In our case, both keyframes tweak the `transform` property.

Instead of a single keyframe on the SVG, let's add a second keyframe to a parent element. We'll tweak the easings so that our rotation is linear, while our scaling parent has a symmetrical ease:

```
function SparkleInstance({ color, size, style }) {
  return (
    <Wrapper>
      <Svg>
        {/* Same stuff here */}
      </Svg>
    </Wrapper>
  );
}

const growAndShrink = keyframes`
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1);
  }
  100% {
    transform: scale(0);
  }
`;

const spin = keyframes`
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(180deg);
  }
`;

const Wrapper = styled.div`
  position: absolute;
  pointer-events: none;
  animation: ${growAndShrink} 600ms ease-in-out forwards;
`
```

```
const Svg = styled.svg`
  animation: ${spin} 600ms linear forwards;
`;
```

With this split, things are looking a lot smoother:



Generation and Cleanup

One final task lays in our path: Dynamically generating a bunch of sparkles, and cleaning them up after they've finished twinkling.

My first instinct was to reach for `setInterval`. This method lets you schedule updates in an asynchronous loop. We can add 1 new sparkle every 500ms, for example.

The problem with this approach is that it feels super robotic/synthetic. I wanted something that felt more organic and haphazard. I didn't want such a staccato rhythm of new sparkles!

I created a new hook, `useRandomInterval`. It works like `setInterval`, except you pass it two numbers, a min and a max. For each iteration, it picks a random number in that range. This leads to a much more natural effect. Here's a side-by-side comparison, each generating an average of 2 sparkles a second:

Constant Interval	Random Interval
Sparkle Text	Sparkle Text

The beauty of custom hooks is that they can totally abstract a lot of complex stuff. I've published this `useRandomInterval` hook **as a snippet**. If you're curious, you can read about how it works, but don't feel obligated; feel free to copy/paste it, and use it as you'd use `setInterval`.

Inside our interval, we'll do two things:

1. Generate a new sparkle.
2. Clean up any old sparkles.

Here's what that looks like:

```
function Sparkles({ children }) {
  const [sparkles, setSparkles] = React.useState([]);

  useRandomInterval(() => {
    const now = Date.now();

    // Create a new sparkle
    const sparkle = generateSparkle();

    // Clean up any "expired" sparkles
    const nextSparkles = sparkles.filter(sparkle => {
      const delta = now - sparkle.createdAt;
      return delta < 1000;
    });

    // Include our new sparkle
    nextSparkles.push(sparkle);

    // Make it so!
    setSparkles(nextSparkles);
  }, 50, 500);

  return (
    <Wrapper>
      {children}
    </Wrapper>
  )
}

const Wrapper = styled.span`
  position: relative;
  display: inline-block;
`;
```

Accessibility

Whimsical features like sparkly text are great, but it's important that they don't come at the expense of accessibility.

In **Accessible Animations in React**, we looked at how the "prefers reduced motion" media query allows people to indicate that they don't want to see any animations. The **usePrefersReducedMotion hook** lets us access that value from within JS.

In this case, I want to do two things:

1. Disable the "twinkling" animation.
2. Disable the random interval that adds them and cleans them up.

If the person prefers reduced motion, we can generate 3-4 sparkles and present them statically:

*This is what people will see when they **prefer reduced motion**.*

We'll initialize our `sparkles` state with this set of sparkles, and disable our `useRandomInterval` loop if motion is disabled:

```
function Sparkles({ children }) {
  // Generate 4 sparkles initially
  const [sparkles, setSparkles] = React.useState(() => {
    return range(4).map(() => generateSparkle(color));
  });

  const prefersReducedMotion = usePrefersReducedMotion();

  useRandomInterval(
    () => {/* Unchanged stuff here */},
    prefersReducedMotion ? null : 50,
    prefersReducedMotion ? null : 500
  );

  // Render sparkles
}
```

`range` is a utility function I use to generate an array. I use it here to create an array full of 4 random sparkles. **See the full snippet**.

`null`

Happily, this hook is fully responsive, meaning that the user can toggle their "prefers reduced motion" status on and off, and our sparkles will freeze as-needed.

One last step—we need to disable both animations when the media query is matched, in CSS:

```
const Wrapper = styled.div`
  position: absolute;
  pointer-events: none;

  @media (prefers-reduced-motion: no-preference) {
    animation: ${growAndShrink} 600ms ease-in-out forwards;
  }
`;

const Svg = styled.svg`
  @media (prefers-reduced-motion: no-preference) {
    animation: ${spin} 600ms linear forwards;
  }
`;
```

Pulling it all together

Here's the final version of the code we've built:

```
const DEFAULT_COLOR = '#FFC700';

const generateSparkle = color => {
  const sparkle = {
    id: String(random(10000, 99999)),
    createdAt: Date.now(),
    color,
    size: random(10, 20),
    style: {
      top: random(0, 100) + '%',
      left: random(0, 100) + '%',
    },
  };

  return sparkle;
};

const Sparkles = ({ color = DEFAULT_COLOR, children, ...delegated
}) => {
  const [sparkles, setSparkles] = React.useState(() => {
    return range(3).map(() => generateSparkle(color));
  });

  const prefersReducedMotion = usePrefersReducedMotion();

  useRandomInterval(
    () => {
```

In order for this to work, you'll need a few dependencies:

- a random utility function
- a range utility function
- the usePrefersReducedMotion hook
- the useRandomInterval hook

```
      const delta = now - sp.createdAt;
      return delta < 750;
    });

    nextSparkles.push(sparkle);

    setSparkles(nextSparkles);
  },
  prefersReducedMotion ? null : 50,
  prefersReducedMotion ? null : 450
);
```

Just the beginning

This `<Sparkles>` component is sort of like an MVP; it does the job, but there's a lot of room for improvement.

```
return (
  <Wrapper {...delegated}>
    {sparkles.map(sparkle => (
      <Sparkle
        key={sparkle.id}
        color={sparkle.color}
        size={sparkle.size}
        style={sparkle.style}
      />
    ))}
    <ChildWrapper>{children}</ChildWrapper>
  </Wrapper>
);
};
```

On this blog, I've taken the liberty of making a few other changes:

- Sparkles can appear either in front of or behind the children
- Sparkle positioning isn't quite random, I try and pick nice arrangements
- You can click sparkly text to disable the effect
- Sparkles are only generated when the element's on-screen, using the IntersectionObserver API.

The code snippet above is meant to serve as a starting point for your own tweaks and customizations. A big part of what makes this effect delightful is that it's **unique**. Be creative, and add your own tweaks to it!

```
const Sparkle = ({ size, color, style }) => {
  const path =
    'M26.5 25.5C19.0043 33.3697 0 34 0 34C0 34 19.1013 35.3684
26.5 43.5C33.234 50.901 34 68 34 68C34 68 36.9884 50.7065 44.5
43.5C51.6431 36.647 68 34 68 34C68 34 51.6947 32.0939 44.5
25.5C36.5605 18.2235 34 0 34 0C34 0 33.6591 17.9837 26.5 25.5Z';
```

Instead of making an opaque NPM package, we've constructed this effect from scratch. Now it's your turn to build on top of it, and add your own unique touch.

I can't wait to see what you come up with!

```
      <SparkleSvg width={size} height={size} viewBox="0 0 68 68"
fill="none">
        <path d={path} fill={color} />
      </SparkleSvg>
    </SparkleWrapper>
  );
```

Last updated on *# of hits*

May 19th, 2020

Josh W Comeau

Move it, football head!

Want to know when I publish new content?
Enter your email to join my free newsletter:

david@live.com

INTERACTIVE COURSES

CSS for JavaScript Devs

The Joy of React

GENERAL

About Josh

About This Blog

Contact