React and Pose - Make a complex slider in React using SVG - Creating complex interactions with SVGs

Written by Seth Corker on Benevolent Bytes

React can be used for more than you think, let's take advantage of React and Pose to create an interactive slider from an SVG that responds to the user dragging it.

<AutoplayVideo src={[{ src: "s3-bucket://build/make-a-complex-slider-in-react-using-svg/responsive-toggle.2019-07-05-7_06_10-pm-2.webm", type: "video/webm", },]} />

The SVG slider in action

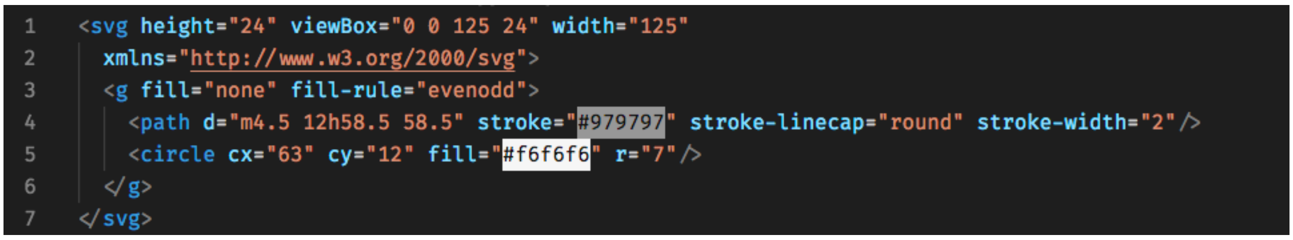
React is great for manipulating HTML in a declarative way, making it easy to understand and predictable. React can do everything you'd expect but there are some cool applications you might not have tried. SVGs are perfect for modern web development because they are a vector format which can fit the screen of any device and look great but they have a superpower. SVG is valid HTML and can be created inline alongside the rest of your markup. This means we can use the power of React to manipulate SVGs dynamically, we can also leverage the animation library Pose to make it easy and fun.

What are we making?

I've made a slider to demonstrate some key techniques you can utilise to get the effect you're looking for.

The simple SVG was created in Sketch and then each parameter was identified and replaced with props.

The slider source:



```
<svg height="24" viewBox="0 0 125 24" width="125"
  xmlns="http://www.w3.org/2000/svg">
  <g fill="none" fill-rule="evenodd">
    <path d="m4.5 12h58.5 58.5" stroke="#979797" stroke-linecap="round" stroke-width="2"/>
    <circle cx="63" cy="12" fill="#f6f6f6" r="7"/>
  </g>
</svg>
```

Figure 1: editing an SVG in a code editor, showing the paths and shapes used to construct an SVG

TIP:* You might get a lot more tags depending on the tool you use to generate the SVG, I stripped this out by running ImageOptim over it.*

Using React to control the SVG

We can easily identify the stroke and fill which could be changed. The path is a little more cryptic. The `d` attribute contains the information for drawing the path. Each letter is a command e.g. ***m** is **moveto** and accepts (x y) coordinates. *You can learn more about what each command does at the W3 spec.

It took some trial and error to identify which parts of the path to parameterize but I made it in the end. In the final version I broke up the path into two separate paths and the commands were changed to create a bezier instead of a simple line, this made the calculations more understandable and meant the bezier curves were easier to get right.

The final version with each parameter identified looks like this:

```
<svg
  height={HEIGHT}
  viewBox={`0 0 ${WIDTH} ${HEIGHT}`}
  width={WIDTH}
  xmlns="http://www.w3.org/2000/svg"
>
  <g fill="none" fillRule="evenodd">
    <path
      d={`
          M ${START_X},${CENTER_Y}
          S ${(x - START_X) * 0.5},${y}
          ${x},${y}
        `}
      stroke={leftColour}
      strokeLinecap="round"
      strokeWidth="4"
    />
    <path
      d={`
          M ${x},${y}
          S ${x + (END_X - x) * 0.5},${y}
            ${END_X},${CENTER_Y}
        `}
      stroke={rightColour}
      strokeLinecap="round"
      strokeWidth="2"
    />
    <SliderKnob
      cx={CENTER_X}
      cy={CENTER_Y}
      r="7"
      fill={knobColour}
      onValueChange={{ x: onXChange, y: onYChange }}
    />
  </g>
</svg>
```

The uppercase variables are constants e.g. `HEIGHT`, `WIDTH`, `START_X`, `START_Y`, etc

The important variables for achieving the movement are `x` and `y`. We need to keep track of the `SliderKnob` so we can move the endpoints of the two paths. The command `S` is used to create a curve and accepts the parameters (*x2 y2 x y*). The bezier control points or coefficients are what gives us a curve from the beginning of the line to the knob which we're moving. This does most of the hard work, now we need to make it interactive.

Making the slider respond to events

The Pose code was much easier than figuring out how to get the SVG to draw when the coordinates changed.

```
const SliderKnob = posed.circle({
  draggable: true,
  dragBounds: {
    left: MIN_X,
    top: -CENTER_Y + MARGIN * 2,
    bottom: CENTER_Y - MARGIN * 2,
    right: MAX_X,
  },
  dragEnd: {
    y: 0,
    transition: { type: "spring", damping: 80, stiffness: 300 },
  },
})
```

We add `draggable: true` to the config object to enable dragging and set the bounds to ensure the user can't drag it off the page with `dragBounds`. It was easier to set up constants and base everything off these, it also reduces the number of re-renders if these values were being passed in as props.

TIP: *To make it more flexible, a factory could be created that returns a component given a set of custom constants.*

The `dragEnd` property is used to reset the knob to `y=0` so and animate it here using a spring animation. This is what creates a loose snap back into place when we release the mouse.

Keeping track of X and Y

In order to ensure the x position is not reset to 0 when the mouse is released and to use x and y as parameters to control the SVG, we need to introduce some state. This is done using the `useState` hook for each coordinate.

```
// Keep track of X and Y for svg path positioning
const [x, setX] = React.useState(0)
const [y, setY] = React.useState(0)

const onXChange = v => {
  setX(v + CENTER_X)
  // Send a percentage to onChange/1
  onChange(Math.floor(((v - MIN_X) * 100) / (MAX_X - MIN_X)))
}

const onYChange = v => {
  setY(v + CENTER_Y)
}
```

The `onXChange` event will calculate the percentage and call an `onChange` callback so the parent knows the X position of the slider because what's the use if we can't hook it up to anything!

The final component can be used like so:

```
<Slider
  rightColour="#E1EDEB"
  leftColour="#5285CC"
  onChange={setValue}
  knobColour="#7DD2DB"
/>
```

I made the `rightColour`, `leftColour` (which correspond to the lines that form the slider track) and `knobColour` props so we can reuse the slider with different themes. An interactive slider, complete with code, for your delight There you have it, an interactive slider that can be dragged using React, Pose and SVG. I hope you learned something or at least had fun along the way.

Takeaway

SVGs are flexible and supported by all major browsers, they're are very powerful because you can manipulate them like you would any other elements on the web. If you want to adjust an SVG with CSS or do something a bit more complicated with React, there's nothing stopping you. Pose is a great animation library that makes it easy to animate HTML elements including SVGs so you should give it a shot. There are also some useful events such as drag, which we've used in the slider to make it interactive.

If you'd like to take a look at another, more real-world example of using Pose to animate regular old HTML. Have a look at, **Animate your React App with Pose**, it covers some easy to deliver tweaks to make a great looking experience.

- Check out the complete source code, svg-slider-pose repo.
- Learn more about getting started with pose.