Version: 3.x

# Animating styles and props

In the last section, we learned how to make simple animations, what shared values are and how to use them. Now, we'll learn a different way of passing animation styles to components. We'll also go over the difference between animating styles and props and how to handle them using `useAnimatedStyle` and `useAnimatedProps`.

## Animating styles

As we learned in the previous section we can animate styles by passing shared values inline to the elements' `style` property:

```
function App() {
  const width = useSharedValue(100);

  return <Animated.View style={{ width }} />;
}
```

In basic cases, this syntax works well but it has one big downside. It doesn't allow to access the value stored in a shared value. For example, it's not possible to build more complex animations by using inline styling to multiply this value (or do any other mathematical operation) before assigning it to the `style` prop.

```
<Animated.View style={{ width: width * 5 }} /> // this won't work
```

Let's suppose we have an example with a box which moves to the right on every button press:

```
function App() {
  const translateX = useSharedValue(0);

  const handlePress = () => {
    translateX.value = withSpring(translateX.value + 50);
  };
```

```
  return (
    <View style={styles.container}>
      <Animated.View style={[styles.box, { transform: [{ translateX }] }]} />
      <Button onPress={handlePress} title="Click me" />
    </View>
  );
}
```

If we would like to customize how our shared value changes based on some user input, (e.g. multiplying it by 2 or following some other mathematical equation) we couldn't use inline styling.

Luckily, the `useAnimatedStyle` hook comes to the rescue. It adds additional control and flexibility over your animation. This can be really useful when creating a bit more complicated animations which include conditional statements or loops.

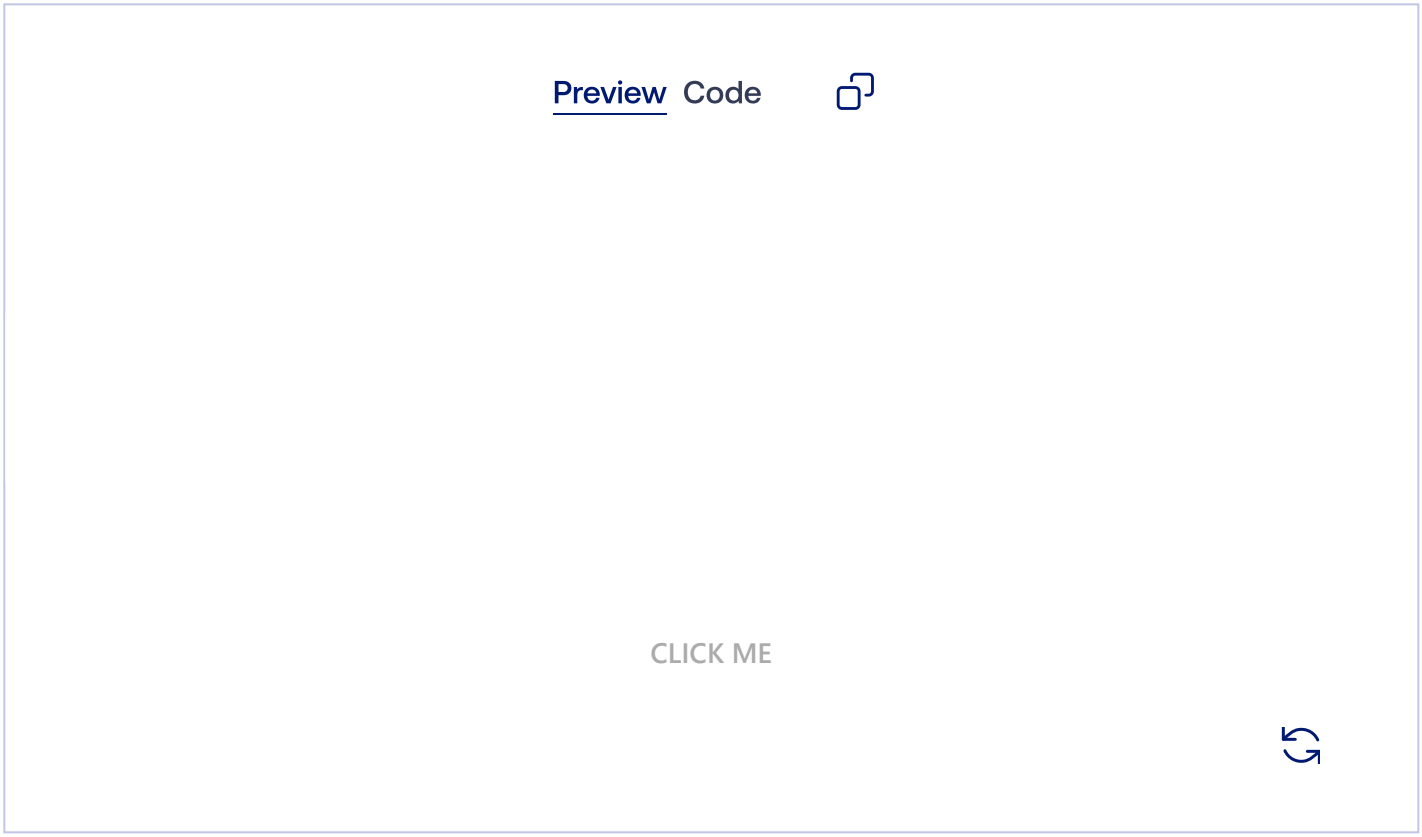
Let's see it in action:

Expand the full code

```
export default function App() {
  const translateX = useSharedValue(0);

  const handlePress = () => {
    translateX.value += 50;
  };

  const animatedStyles = useAnimatedStyle(() => ({
    transform: [{ translateX: withSpring(translateX.value * 2) }],
  }));

  return (
    <>
      <Animated.View style={[styles.box, animatedStyles]} />
      <View style={styles.container}>
        <Button onPress={handlePress} title="Click me" />
      </View>
    </>
  );
```

`useAnimatedStyle` lets you access the value stored in a shared value. Thanks to that we could multiply the value by `2` before assigning it to `style`. This hook has one more advantage over passing animations to inline styles. It allows you to keep all the animation-related logic in one place.

You can see it in action in the example below:




## Animating props

Most of the values that developers animate ( `width` , `color` , `transform` etc.) are modified by passing them as an object to the `style` property of an element. But that's not always the case.

Sometimes we'd like to animate not just styles but also the props which are passed to the component.

For example, let's say we would like to animate SVG elements. Instead of passing values to the `style` property, values are defined as props:

```
<Circle cx="50" cy="50" r="10" fill="blue" />
```

Reanimated comes with just a handful of built-in components like `Animated.View` or `Animated.ScrollView` . For components which aren't a part of Reanimated, to make their props animatable, we need to wrap them with `createAnimatedComponent` :

```
import Animated from 'react-native-reanimated';
import { Circle } from 'react-native-svg';

const AnimatedCircle = Animated.createAnimatedComponent(Circle);
```

To animate the radius of the SVG circle we can simply pass the shared value as a prop:

```
function App() {
  const r = useSharedValue(10);

  return (
    <Svg>
      <AnimatedCircle cx="50" cy="50" r={r} fill="blue" />
    </Svg>
  );
}
```

This approach works just fine but same as `useAnimatedStyle` for animating styles we can encapsulate the animation logic and gain access to the `.value` property of a shared value by using `useAnimatedProps`.

So if we'd like to smoothly increase the radius of a circle by `10px` on each button press we could use `useAnimatedProps`:

Expand the full code

```
const AnimatedCircle = Animated.createAnimatedComponent(Circle);

export default function App() {
  const r = useSharedValue(20);

  const handlePress = () => {
    r.value += 10;
  };

  const animatedProps = useAnimatedProps(() => ({
    r: withTiming(r.value),
  }));

  return (
    <View style={styles.container}>
```

```
      <Svg style={styles.svg}>
        <AnimatedCircle
          cx="50%"
          cy="50%"
          fill="#b58df1"
          animatedProps={animatedProps}
        />
      </Svg>
      <Button onPress={handlePress} title="Click me" />
    </View>
  );
}
```

In a function which `useAnimatedProps` takes as an argument, we return an object with all the props we'd like to animate. Then we can pass the value which `useAnimatedProps` returns to the `animatedProps` prop of an Animated component.

Check out the full example below:

Preview Code

CLICK ME

## Summary

In this section, we went through the differences between animating styles and props and how to use `useAnimatedStyle` and `useAnimatedProps`. To sum up:

- Passing shared values to inline styles is a simple way of creating animations but it has some limitations.
- Difference between animating `props` and `styles` is that props are not passed to the `style` object, but rather as separate props of the component.
- By using `useAnimatedStyle` and `useAnimatedProps`, you can access the value stored in a shared value. This can add additional control over the animation.
- You can make your own animatable components by wrapping them with `Animated.createAnimatedComponent`.

## What's next?

In **the next section**, we'll learn more about animation functions and how to customize their behavior.

Edit this page