Version: 3.x

Handling gestures

In this section, we'll learn how to handle gestures with Reanimated. To achieve this, Reanimated integrates tightly with **React Native Gesture Handler**, another library created by Software Mansion.

Gesture Handler comes with plentiful gestures like `Pinch` or `Fling`. Right now we'll start simple and get to know `Tap` and `Pan` gestures as well as how to use `withDecay` animation function.

Just make sure to go through the Gesture Handler **installation steps** first and come back here to learn how to use it with Reanimated.

Handling tap gestures

Let's start with the simplest gesture - tapping. Tap gesture detects fingers touching the screen for a short period of time. You can use them to implement custom buttons or pressable elements from scratch.

In this example, we'll create a circle which will grow and change color on touch.

First, let's wrap our app with `GestureHandlerRootView`. Make sure to keep the `GestureHandlerRootView` as close to the actual root view as possible. That'll ensure that our gestures will work as expected with each other.

```
import { GestureHandlerRootView } from 'react-native-gesture-handler';

function App() {
  return (
    <GestureHandlerRootView style={{ flex: 1 }}>
      {/* rest of the app */}
    </GestureHandlerRootView>
  );
}
```

New tap gestures are defined with `Gesture.Tap()` in your component's body. You can define the behavior of the gesture by chaining methods like `onBegin`, `onStart`, `onEnd`, or `onFinalize` on the gesture. We'll use them to update a shared value just after the gesture begins and return to the initial value when the gesture finishes.

Expand the full code

```
export default function App() {
  const pressed = useSharedValue(false);

  const tap = Gesture.Tap()
    .onBegin(() => {
      pressed.value = true;
    })
    .onFinalize(() => {
      pressed.value = false;
    });
```

You can safely access the shared values because callbacks passed to gestures are automatically workletized for you.

We'd like our circle to change color from violet to yellow and smoothly scale by 20% on tap. Let's define that animation logic using `withTiming` in the `useAnimatedStyle`:

Expand the full code

```
  const animatedStyles = useAnimatedStyle(() => ({
    backgroundColor: pressed.value ? '#FFE04B' : '#B58DF1',
    transform: [{ scale: withTiming(pressed.value ? 1.2 : 1) }],
  }));
```

You need to pass your defined gesture to the `gesture` prop of the `GestureDetector` component. That component should wrap the view you'd like to handle gestures on. Also, remember to pass the defined `animatedStyles` to the view you want to animate like so:

Expand the full code

```
  return (
    <GestureHandlerRootView style={styles.container}>
      <View style={styles.container}>
        <GestureDetector gesture={tap}>
          <Animated.View style={[styles.circle, animatedStyles]} />
        </GestureDetector>
      </View>
    </GestureHandlerRootView>
  );
}
```

And that's it! Pretty straightforward, isn't it? Let's see it in the full glory in an interactive example:

Preview Code

Tap the circle

You can make the use of composing gestures to more complex behaviors. But what if we'd like to create something a bit more interesting?

Handling pan gestures

Let's spice things a bit by making the circle draggable and have it bounce back to it's staring position when released. Let's also keep the color highlight and scale effect we've added in the previous example. Implementing this behavior it's not possible with just a simple tap gesture. We need to reach for a pan gesture instead.

Luckily, all the gestures share a similar API so implementing this is nearly as easy as renaming the `Tap` gesture to `Pan` and chaining an additional `onChange` method.

Expand the full code

```
const offset = useSharedValue(0);

const pan = Gesture.Pan()
```

```
    .onBegin(() => {
      pressed.value = true;
    })
    .onChange((event) => {
      offset.value = event.translationX;
    })
    .onFinalize(() => {
      offset.value = withSpring(0);
      pressed.value = false;
    });
```

The callback passed to `onChange` comes with some **event data** that has a bunch of handy properties. One of them is `translationX` which indicates how much the object has moved on the X axis. We stored that in a shared value to move the circle accordingly. To make the circle come back to initial place all you have to do is to reset the `offset.value` in the `onFinalize` method. We can use `withSpring` or `withTiming` functions to make it come back with an animation.

All that's left to do is to adjust the logic in `useAnimatedStyle` to handle the offset.

Expand the full code

```
const animatedStyles = useAnimatedStyle(() => ({
  transform: [
    { translateX: offset.value },
    { scale: withTiming(pressed.value ? 1.2 : 1) },
  ],
  backgroundColor: pressed.value ? '#FFE04B' : '#b58df1',
}));
```

You can play around with the example below and see how the circle changes and reacts to the gesture:

Preview Code

Grab and drag the circle

Using `withDecay`

Remember when some time ago we said that we'll come back to `withDecay`? Now this is the time!

`withDecay` lets you retain the velocity of the gesture and animate with some deceleration. That means when you release a grabbed object with some velocity you can slowly bring it to stop. Sounds complicated but it really isn't!

Simply pass the final velocity in `onFinalize` method to the `velocity` property of `withDecay` function and let Reanimated handle it for you. To retain the new position of an object update the change on the X axis in `onChange` method like so:

Expand the full code

```
const pan = Gesture.Pan()
  .onChange((event) => {
    offset.value += event.changeX;
  })
  .onFinalize((event) => {
    offset.value = withDecay({
      velocity: event.velocityX,
      rubberBandEffect: true,
      clamp: [-(width.value / 2) + SIZE / 2, width.value / 2 - SIZE / 2],
    });
  });
```

The rest of the code is just to make sure the square stays inside the screen.

Play around and see how the square decelerates when let go with some speed!

Preview Code

Grab and release the square

Make sure to check check the full `withDecay` **API reference** to get to know rest of the configuration options.

Summary

In this section, we went through basics of handling gestures with Reanimated and Gesture Handler. We learned about `Tap` and `Pan` gestures and `withDecay` function. To sum up:

- Reanimated integrates with a different package called React Native Gesture Handler to provide seamless interactions.
- We create new gestures, such as `Gesture.Pan()` or `Gesture.Tap()`, and pass them to `GestureDetector`, which have to wrap the element we want to handle interactions on.
- You can access and modify shared values inside gesture callbacks without any additional boilerplate.
- `withDecay` lets you create decelerating animations based on velocity coming from a gesture.

What's next?

In this article, we've barely scratched the surface of what's possible with gestures in Reanimated. Besides **Tap** and **Pan** gestures Gesture Handler comes with many more e.g. **Pinch** or **Fling**. We welcome you to dive into the **Quick start** section of the React Native Gesture Handler documentation and explore all the possibilities that this library comes with.

In **the next section**, we'll learn more about animating colors.

Edit this page