Version: 2.6.0 – 2.12.0

About Gesture Handlers

Gesture handlers are the core building blocks of this library. We use this term to describe elements of the native touch system that the library allows us to instantiate and control from Javascript using React's Component interface.

Each handler type is capable of recognizing one type of gesture (pan, pinch, etc.) and provides gesture-specific information via events (translation, scale, etc.).

Handlers analyze touch stream synchronously in the UI thread. This allows for uninterrupted interactions even when the Javascript thread is blocked.

Each handler works as an isolated state machine. It takes touch stream as an input and based on it, it can flip between states. When a gesture starts, based on the position where the finger was placed, a set of handlers that may be interested in recognizing the gesture is selected. All the touch events (touch down, move, up, or when other fingers are placed or lifted) are delivered to all of the handlers selected initially. When one gesture becomes active, it cancels all the other gestures (read more about how to influence this process in "Cross handler interactions" section).

Gesture handler components do not instantiate a native view in the view hierarchy. Instead, they are kept in library's own registry and are only connected to native views. When using any of the gesture handler components, it is important for it to have a native view rendered as a child. Since handler components don't have corresponding views in the hierarchy, the events registered with them are actually hooked into the underlying view.

Available gesture handlers

Currently, the library provides the following list of gestures. Their parameters and attributes they provide to gesture events are documented under each gesture page:

- `PanGestureHandler`
- `TapGestureHandler`
- `LongPressGestureHandler`
- `RotationGestureHandler`

- `FlingGestureHandler`
- `PinchGestureHandler`
- `ForceTouchGestureHandler`

Discrete vs continuous

We distinguish between two types of gestures: discrete and continuous.

Continuous gesture handlers can be active for a long period of time and will generate a stream of gesture events until the gesture is over. An example of a continuous handler is `PanGestureHandler` that once activated, will start providing updates about translation and other properties.

On the other hand, discrete gesture handlers once activated will not stay in the active state but will end immediately. `LongPressGestureHandler` is a discrete handler, as it only detects if the finger is placed for a sufficiently long period of time, it does not track finger movements (as that's the responsibility of `PanGestureHandler`).

Keep in mind that `onGestureEvent` is only generated in continuous gesture handlers and shouldn't be used in the `TapGestureHandler` and other discrete handlers.

Nesting handlers

Handler components can be nested. In any case it is recommended that the innermost handler renders a native view component. There are some limitations that apply when using `useNativeDriver` flag. An example of nested handlers:

```
class Multitap extends Component {
  render() {
    return (
      <LongPressGestureHandler
        onHandlerStateChange={this._onLongpress}
        minDurationMs={800}>
        <TapGestureHandler
          onHandlerStateChange={this._onSingleTap}
          waitFor={this.doubleTapRef}>
          <TapGestureHandler
            ref={this.doubleTapRef}
            onHandlerStateChange={this._onDoubleTap}
            numberOfTaps={2}>
            <View style={styles.box} />
          </TapGestureHandler>
        </TapGestureHandler>
      </LongPressGestureHandler>
    );
  }
}
```

Using native components

Gesture handler library exposes a set of components normally available in React Native that are wrapped in `NativeViewGestureHandler`. Here is a list of exposed components:

- `ScrollView`
- `FlatList`
- `Switch`
- `TextInput`
- `DrawerLayoutAndroid` (**Android only**)

If you want to use other handlers or buttons nested in a `ScrollView`, use the `waitFor` property to define interaction between a handler and `ScrollView`

Events with `useNativeDriver`

Because handlers do not instantiate native views but instead hook up to their child views, directly nesting two gesture handlers using `Animated.event` is not currently supported. To workaround this limitation we recommend placing an `<Animated.View>` component in between the handlers.

Instead of doing:

```
const PanAndRotate = () => (
  <PanGestureHandler onGestureEvent={Animated.event({ ... }, { useNativeDriver: true
})}>
    <RotationGestureHandler onGestureEvent={Animated.event({ ... }, {
useNativeDriver: true })}>
      <Animated.View style={animatedStyles}/>
    </RotationGestureHandler>
  </PanGestureHandler>
);
```

Place an `<Animated.View>` in between the handlers:

```
const PanAndRotate = () => (
  <PanGestureHandler onGestureEvent={Animated.event({ ... }, { useNativeDriver: true
})}>
    <Animated.View>
      <RotationGestureHandler onGestureEvent={Animated.event({ ... }, {
useNativeDriver: true })}>
        <Animated.View style={animatedStyles}/>
      </RotationGestureHandler>
    </Animated.View>
  </PanGestureHandler>
);
```

Another consequence of handlers depending on their native child components is that when using a `useNativeDriver` flag with an `Animated.event`, the child component must be wrapped by an `Animated.API` e.g. `<Animated.View>` instead of just a `<View>`:

```
class Draggable extends Component {
  render() {
    return (
      <PanGestureHandler onGestureEvent={Animated.event({ ... }, { useNativeDriver:
true })}>
        <Animated.View style={animatedStyles} /> {/* <-- NEEDS TO BE Animated.View
*/}
      </PanGestureHandler>
    );
  }
};
```