Version: 2.6.0 – 2.12.0

Pinch gesture

A continuous gesture that recognizes pinch gesture. It allows for tracking the distance between two fingers and use that information to scale or zoom your content. The gesture activates when fingers are placed on the screen and change their position. Gesture callback can be used for continuous tracking of the pinch gesture. It provides information about velocity, anchor (focal) point of gesture and scale.

The distance between the fingers is reported as a scale factor. At the beginning of the gesture, the scale factor is 1.0. As the distance between the two fingers increases, the scale factor increases proportionally. Similarly, the scale factor decreases as the distance between the fingers decreases. Pinch gestures are used most commonly to change the size of objects or content onscreen. For example, map views use pinch gestures to change the zoom level of the map.

Config

Properties common to all gestures:

`enabled(value: boolean)`

Indicates whether the given handler should be analyzing stream of touch events or not. When set to `false` we can be sure that the handler's state will **never** become `ACTIVE`. If the value gets updated while the handler already started recognizing a gesture, then the handler's state it will immediately change to `FAILED` or `CANCELLED` (depending on its current state). Default value is `true`.

`shouldCancelWhenOutside(value: boolean)`

When `true` the handler will cancel or fail recognition (depending on its current state) whenever the finger leaves the area of the connected view. Default value of this property is different depending on the handler type. Most handlers' `shouldCancelWhenOutside` property defaults to `false` except for the `LongPressGesture` and `TapGesture` which default to `true`.

`hitSlop(settings)`

This parameter enables control over what part of the connected view area can be used to begin recognizing the gesture. When a negative number is provided the bounds of the view will reduce the area by the given number of points in each of the sides evenly.

Instead you can pass an object to specify how each boundary side should be reduced by providing different number of points for `left`, `right`, `top` or `bottom` sides. You can alternatively provide `horizontal` or `vertical` instead of specifying directly `left`, `right` or `top` and `bottom`. Finally, the object can also take `width` and `height` attributes. When `width` is set it is only allow to specify one of the sides `right` or `left`. Similarly when `height` is provided only `top` or `bottom` can be set. Specifying `width` or `height` is useful if we only want the gesture to activate on the edge of the view. In which case for example we can set `left: 0` and `width: 20` which would make it possible for the gesture to be recognize when started no more than 20 points from the left edge.

IMPORTANT: Note that this parameter is primarily designed to reduce the area where gesture can activate. Hence it is only supported for all the values (except `width` and `height`) to be non positive (0 or lower). Although on Android it is supported for the values to also be positive and therefore allow to expand beyond view bounds but not further than the parent view bounds. To achieve this effect on both platforms you can use React Native's View hitSlop property.

`withRef(ref)`

Sets a ref to the gesture object, allowing for interoperability with the old API.

`withTestId(testID)`

Sets a `testID` property for gesture object, allowing for querying for it in tests.

`cancelsTouchesInView(value)` (iOS only)

Accepts a boolean value. When `true`, the gesture will cancel touches for native UI components (`UIButton`, `UISwitch`, etc) it's attached to when it becomes `ACTIVE`. Default value is `true`.

`runOnJS(value: boolean)`

When `react-native-reanimated` is installed, the callbacks passed to the gestures are automatically workletized and run on the UI thread when called. This option allows for changing this behavior: when `true`, all the callbacks will be run on the JS thread instead of the UI thread, regardless of whether they are worklets or not. Defaults to `false`.

`simultaneousWithExternalGesture(otherGesture1, otherGesture2, ...)`

Adds a gesture that should be recognized simultaneously with this one.

IMPORTANT: Note that this method only marks the relation between gestures, without composing them. `GestureDetector` will not recognize the `otherGestures` and it needs to be added to another detector in order to be recognized.

`requireExternalGestureToFail(otherGesture1, otherGesture2, ...)`

Adds a relation requiring another gesture to fail, before this one can activate.

IMPORTANT: Note that this method only marks the relation between gestures, without composing them. `GestureDetector` will not recognize the `otherGestures` and it needs to be added to another detector in order to be recognized.

Properties common to all continous gestures:

`manualActivation(value: boolead)`

When `true` the handler will not activate by itself even if its activation criteria are met. Instead you can manipulate its state using state manager.

Callbacks

Callbacks common to all gestures:

`onBegin(callback)`

Set the callback that is being called when given gesture handler starts receiving touches. At the moment of this callback the handler is not yet in an active state and we don't know yet if it will recognize the gesture at all.

`onStart(callback)`

Set the callback that is being called when the gesture is recognized by the handler and it transitions to the active state.

`onEnd(callback)`

Set the callback that is being called when the gesture that was recognized by the handler finishes. It will be called only if the handler was previously in the active state.

`onFinalize(callback)`

Set the callback that is being called when the handler finalizes handling gesture - the gesture was recognized and has finished or it failed to recognize.

`onTouchesDown(callback)`

Set the `onTouchesDown` callback which is called every time a finger is placed on the screen.

`onTouchesMove(callback)`

Set the `onTouchesMove` callback which is called every time a finger is moved on the screen.

`onTouchesUp(callback)`

Set the `onTouchesUp` callback which is called every time a finger is lifted from the screen.

`onTouchesCancelled(callback)`

Set the `onTouchesCancelled` callback which is called every time a finger stops being tracked, for example when the gesture finishes.

Callbacks common to all continous gestures:

`onUpdate(callback)`

Set the callback that is being called every time the gesture receives an update while it's active.

`onChange(callback)`

Set the callback that is being called every time the gesture receives an update while it's active. This callback will receive information about change in value in relation to the last received event.

Event data

Event attributes specific to `PinchGesture`:

`scale`

The scale factor relative to the points of the two touches in screen coordinates.

`velocity`

Velocity of the pan gesture the current moment. The value is expressed in point units per second.

`focalX`

Position expressed in points along X axis of center anchor point of gesture

`focalY`

Position expressed in points along Y axis of center anchor point of gesture

Event attributes common to all gestures:

`state`

Current state of the handler. Expressed as one of the constants exported under `State` object by the library.

`numberOfPointers`

Represents the number of pointers (fingers) currently placed on the screen.

Example

```
const scale = useSharedValue(1);
const savedScale = useSharedValue(1);

const pinchGesture = Gesture.Pinch()
  .onUpdate((e) => {
```

```
      scale.value = savedScale.value * e.scale;
    })
    .onEnd(() => {
      savedScale.value = scale.value;
    });

  const animatedStyle = useAnimatedStyle(() => ({
    transform: [{ scale: scale.value }],
  }));

  return (
    <GestureDetector gesture={pinchGesture}>
      <Animated.View style={[styles.box, animatedStyle]} />
    </GestureDetector>
  );
```