Version: 2.6.0 – 2.12.0

Manual gestures

RNGH2 finally brings one of the most requested features: manual gestures and touch events. To demonstrate how to make a manual gesture we will make a simple one that tracks all pointers on the screen.

Step 1

First, we need a way to store information about the pointer: whether it should be visible and its position.

```
interface Pointer {
  visible: boolean;
  x: number;
  y: number;
}
```

Step 2

We also need a component to mark where a pointer is. In order to accomplish that we will make a component that accepts two shared values: one holding information about the pointer using the interface we just created, the other holding a bool indicating whether the gesture has activated. In this example when the gesture is not active, the ball representing it will be blue and when it is active the ball will be red and slightly bigger.

```
function PointerElement(props: {
  pointer: Animated.SharedValue<Pointer>,
  active: Animated.SharedValue<boolean>,
}) {
  const animatedStyle = useAnimatedStyle(() =>
({
    transform: [
      { translateX: props.pointer.value.x },
      { translateY: props.pointer.value.y },
      {
        scale:
          (props.pointer.value.visible ? 1 : 0)
*
          (props.active.value ? 1.3 : 1),
      },
    ],
    backgroundColor: props.active.value ? 'red'
: 'blue',
  }));

  return <Animated.View style={[styles.pointer,
animatedStyle]} />;
}
```

```
...

const styles = StyleSheet.create({
  pointer: {
    width: 60,
    height: 60,
    borderRadius: 30,
    backgroundColor: 'red',
    position: 'absolute',
    marginStart: -30,
    marginTop: -30,
  },
});
```

Step 3

Now we have to make a component that will handle the gesture and draw all the pointer indicators. We will store data about pointers in an array of size 12 as that is the maximum number of touches that RNGH will track, and render them inside an Animated.View.

```
export default function Example() {
  const trackedPointers:
Animated.SharedValue<Pointer>[] = [];
  const active = useSharedValue(false);

  for (let i = 0; i < 12; i++) {
    trackedPointers[i] =
      useSharedValue <
      Pointer >
      {
        visible: false,
        x: 0,
        y: 0,
      };
  }

  const gesture = Gesture.Manual();

  return (
    <GestureDetector gesture={gesture}>
      <Animated.View style={{ flex: 1 }}>
        {trackedPointers.map((pointer, index)
=> (
          <PointerElement pointer={pointer}
active={active} key={index} />
        ))}
      </Animated.View>
    </GestureDetector>
  );
}
```

Step 4

We have our components set up and we can finally get to making the gesture! We will start with onTouchesDown where we need to set position of the pointers and make them visible. We can get this information from the touches property of the event. In this case we will also check how many pointers are on the screen and activate the gesture if there are at least two.

```
const gesture =
Gesture.Manual().onTouchesDown((e, manager) =>
{
  for (const touch of e.changedTouches) {
    trackedPointers[touch.id].value = {
      visible: true,
      x: touch.x,
      y: touch.y,
    };
  }

  if (e.numberOfTouches >= 2) {
    manager.activate();
  }
});
```

Step 5

Next, we will handle pointer movement. In onTouchesMove we will simply update the position of moved pointers.

```
const gesture = Gesture.Manual()
    ...
    .onTouchesMove((e, _manager) => {
      for (const touch of e.changedTouches) {
        trackedPointers[touch.id].value = {
          visible: true,
          x: touch.x,
          y: touch.y,
        };
      }
    })
```

Step 6

We also need to handle lifting fingers from the screen, which corresponds to onTouchesUp. Here we will just hide the pointers that were lifted and end the gesture if there are no more pointers on the screen. Note that we are not handling

```
const gesture = Gesture.Manual()
    ...
    .onTouchesUp((e, manager) => {
      for (const touch of e.changedTouches) {
        trackedPointers[touch.id].value = {
          visible: false,
          x: touch.x,
          y: touch.y,
        };
      }

      if (e.numberOfTouches === 0) {
```

onTouchesCancelled as in this very basic case we don't expect it to happen, however you should clear data about cancelled pointers (most of the time all active ones) when it is called.

```
        manager.end();
      }
    })
```

Step 7

Now that our pointers are being tracked correctly and we have the state management, we can handle activation and ending of the gesture. In our case, we will simply set the active shared value either to true or false.

```
const gesture = Gesture.Manual()
  ...
  .onStart(() => {
    active.value = true;
  })
  .onEnd(() => {
    active.value = false;
  });
```

And that's all! As you can see using manual gestures is really easy but as you can imagine, manual gestures are a powerful tool that makes it possible to accomplish things that were previously impossible with RNGH.

Modifying existing gestures

While manual gestures open great possibilities we are aware that reimplementing pinch or rotation from scratch just because you need to activate in specific circumstances or require position of the fingers, would be a waste of time as those gestures are already there. Because of that you can use touch events with every gesture so that you can extract more informations about gesture than is sent to you in events. We also added a `manualActivation` modifier on all continous gestures, which prevents the gesture it is applied to from activating by itself thus giving you full control of its behavior.

This functionality makes another highly requested feature possible: drag after long press. Simply set `manualActivation` to `true` on a `PanGesture` and use `StateManager` to fail the gesture if the user attempts to drag the component sooner than the duration of the long press.