Tanner West Follow
25 de agosto · 5 minutos de leitura · Ouço

Save

# Construindo um aplicativo de desenho à mão com React Native Skia e Gesture Handler

Foto de Erik Mclean no Unsplash

Agora que a biblioteca de gráficos Skia 2D está disponível no React Native, temos muitas oportunidades para criar aplicativos com uso intensivo de gráficos que não eram possíveis antes. Um objetivo pessoal meu por um tempo era explorar o React Native Gesture Handler, então Skia cruzou meu radar e eu queria ter alguma experiência com ele também. A ideia me ocorreu para construir um aplicativo de desenho à mão de prova de conceito que captura a entrada do usuário via Gesture

Handler e a pinta em um componente de tela usando RN Skia. Este artigo aborda os conceitos básicos de como colocar esse aplicativo em funcionamento.


## Introdução ao React Native Gesture Handler

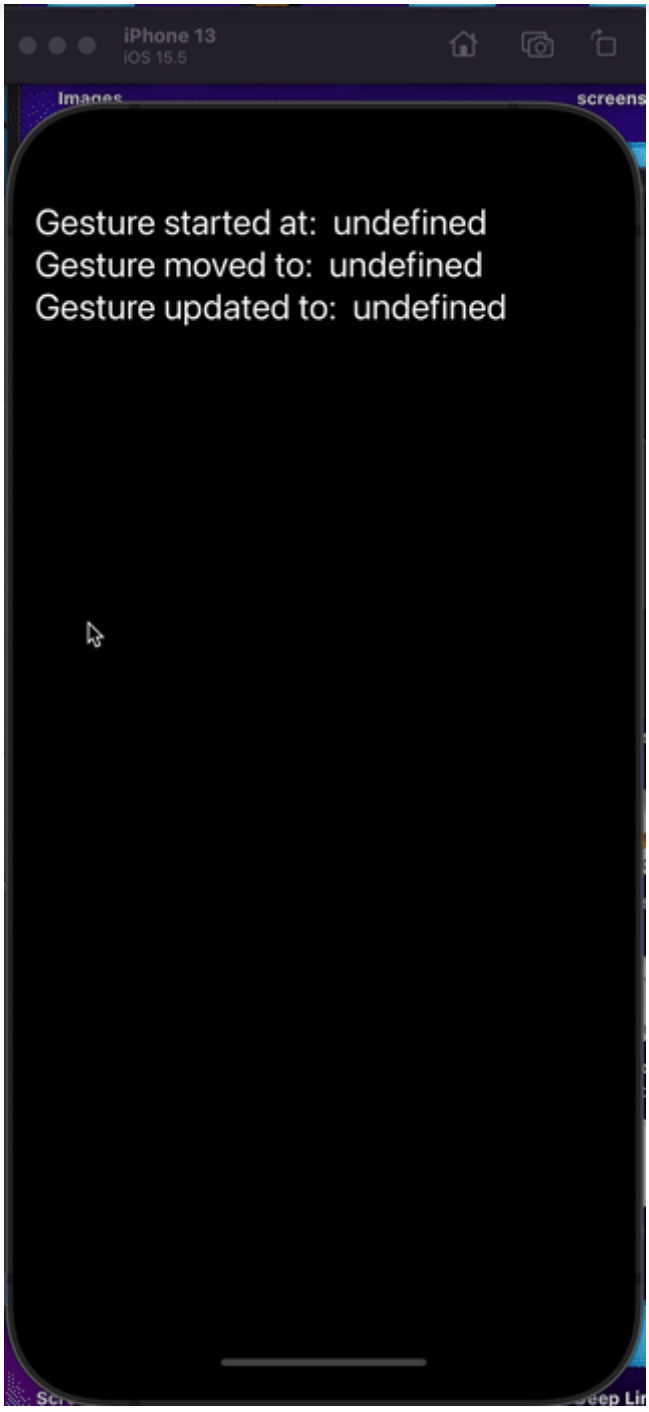
O primeiro passo neste projeto é entender a API do RN Gesture Handler. Você pode ler mais sobre isso na documentação do Software Mansion , mas é bem direto. Veja como é um componente básico:

```
1   import React, { useState } from "react";
```

```
5     GestureDetector,
6     GestureHandlerRootView,
7   } from "react-native-gesture-handler";
8
9   export default function GestureDemo() {
10    const [tGestureStart, setTGestureStart] = useState<undefined | string>();
11    const [tGestureMove, setTGestureMove] = useState<undefined | string>();
12    const [tGestureUpdate, setTGestureUpdate] = useState<undefined | string>();
13    const [tGestureEnd, setTGestureEnd] = useState<undefined | string>();
14
15    const pan = Gesture.Pan()
16      .onStart((g) => {
17        // Start gesture looks like this
18        // {
19        //   "absoluteX": 178,
20        //   "absoluteY": 484,
21        //   "handlerTag": 2,
22        //   "numberOfPointers": 1,
23        //   "oldState": 2,
24        //   "state": 4,
25        //   "target": 115,
26        //   "translationX": -4.3333282470703125,
27        //   "translationY": 0,
28        //   "velocityX": -172.0102558333448,
29        //   "velocityY": 0,
30        //   "x": 178,
31        //   "y": 484,
32        // }
33        setTGestureStart(`${Math.round(g.x)}, ${Math.round(g.y)}`);
34      })
35      .onTouchesMove((g) => {
36      // Move gesture looks like this
37      // {
38      //   "allTouches": Array [
39      //     {
40      //       "absoluteX": 123.33332824707031,
41      //       "absoluteY": 449,
42      //       "id": 0,
43      //       "x": 123.33332824707031,
44      //       "y": 449,
45      //     },
46      //   ],
47      //   "changedTouches": Array [
48      //     {
49      //       "absoluteX": 123.33332824707031,
```


```
52      //       "x": 123.33332824707031,
53      //       "y": 449,
54      //     },
55      //   ],
56      //   "eventType": 2,
57      //   "handlerTag": 2,
58      //   "numberOfTouches": 1,
59      //   "state": 4,
60      //   "target": 115,
61      // }
62        setTGestureMove(
63          `${Math.round(g.changedTouches[0].x)}, ${Math.round(
64            g.changedTouches[0].y
65          )}`
66        );
67      })
68      .onUpdate((g) => {
69        // Update gesture looks like
70        // {
71        //   "absoluteX": 229,
72        //   "absoluteY": 400.3333282470703,
73        //   "handlerTag": 2,
74        //   "numberOfPointers": 1,
75        //   "state": 4,
76        //   "target": 115,
77        //   "translationX": 0,
78        //   "translationY": 1.6666717529296875,
79        //   "velocityX": 0,
80        //   "velocityY": 24.111687246989227,
81        //   "x": 229,
82        //   "y": 400.3333282470703,
83        // }
84        setTGestureUpdate(`${Math.round(g.x)}, ${Math.round(g.y)}`);
85      })
86      .onEnd((g) => {
87        // End gesture looks like this
88        // {
89        //   "absoluteX": 213.3333282470703,
90        //   "absoluteY": 542.6666564941406,
91        //   "handlerTag": 2,
92        //   "numberOfPointers": 0,
93        //   "oldState": 4,
94        //   "state": 5,
95        //   "target": 115,
96        //   "translationX": -66,
```


```
100         //   "x": 213.3333282470703,
101         //   "y": 542.6666564941406,
102         // }
103         setTGestureEnd(`${Math.round(g.x)}, ${Math.round(g.y)}`);
104       });
105     return (
106       <GestureHandlerRootView style={{ flex: 1 }}>
107         <GestureDetector gesture={pan}>
108           <SafeAreaView style={{ flex: 1, backgroundColor: "black" }}>
109             <Text
110               style={{ color: "white", fontSize: 24 }}
111             >{`Gesture started at:  ${tGestureStart}`}</Text>
112             <Text
113               style={{ color: "white", fontSize: 24 }}
114             >{`Gesture moved to:  ${tGestureMove}`}</Text>
115             <Text
116               style={{ color: "white", fontSize: 24 }}
117             >{`Gesture updated to:  ${tGestureUpdate}`}</Text>
118             <Text
119               style={{ color: "white", fontSize: 24 }}
120             >{`Gesture ended at:  ${tGestureEnd}`}</Text>
121           </SafeAreaView>
122         </GestureDetector>
123       </GestureHandlerRootView>
```


Envolvemos o componente do qual queremos capturar gestos em um `<GestureDetector>` e passamos um `gesture` prop para o tipo de gesto ao qual queremos reagir. Este prop assume a forma de um `Gesture` objeto que configuramos com o código que queremos executar quando ocorrerem determinados eventos (quando o gesto inicia, atualiza e termina). Se você quiser reagir a vários tipos de gestos, basta envolver seu componente em um adicional `<GestureDetector>` para cada tipo de gesto.

Na gravação acima, observe como `onTouchesMove` é chamado antes `onStart` e

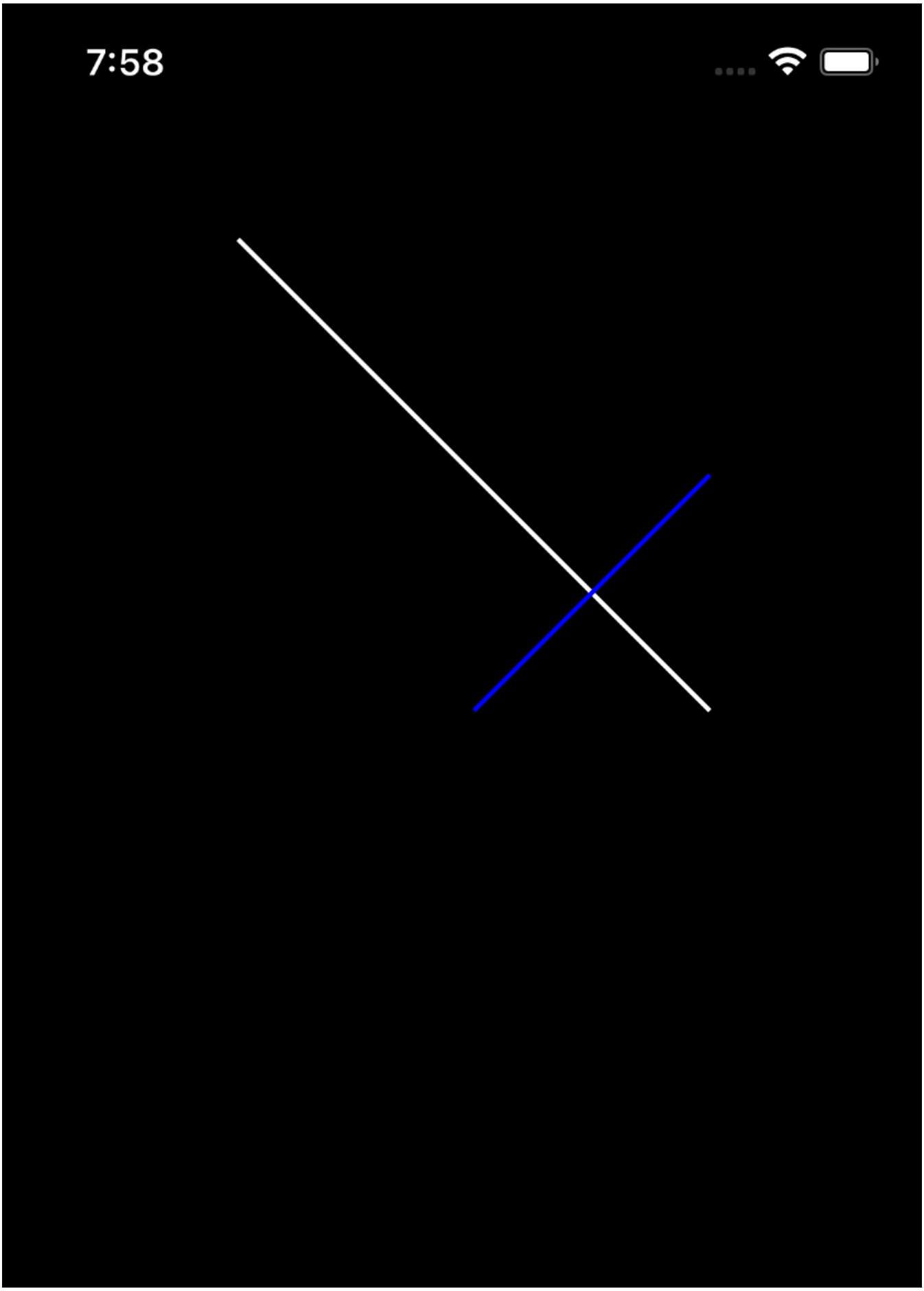
vez que o movimento foi reconhecido como um gesto. Essa distinção será relevante para nosso aplicativo de desenho!

## Entendendo o básico do Canvas no RN Skia

Se você já trabalhou com gráficos 2D como SVGs no passado, esses conceitos devem ser familiares. De qualquer forma, eles são fáceis de pegar. Uma tela implementa um sistema de coordenadas XY que usamos para desenhar nossos gráficos. Por padrão, as coordenadas 0, 0 estão no canto superior esquerdo da tela. O eixo X se move da esquerda para a direita e o eixo Y é para cima e para baixo.

React Native Skia nos dá muitas maneiras convenientes de desenhar em nossa tela. Por exemplo, se você quiser usar o `Path` componente para desenhar uma linha, você pode usar a notação clássica de caminho SVG e passar o valor para seu componente como uma string. Alternativamente, você pode usar o objeto de Skia `Path` para construir um caminho ponto a ponto.

```
const Lines = () => {

  const path = Skia.Path.Make();
  path.moveTo(300, 200);
  path.lineTo(200, 300);

  return (
    <>
      <View style={{ flex: 1 }}>
        <Canvas style={{ flex: 1, backgroundColor: "black" }}>

          {/* Build a path using the classic SVG path notation */}
          <Path
            path={"M 100 100 L 300 300"}
            strokeWidth={2}
            color="white"
            style="stroke"
          />

          {/* Build a path using the Path object API (see above) */}
          <Path path={path} strokeWidth={2} color="blue" style="stroke" />

        </Canvas>
      </View>
    </>
```


7:58

Entender como funciona a notação SVG no exemplo anterior é fundamental para entender o aplicativo que vamos construir. Você definitivamente deve ler o artigo do MDN sobre o assunto , mas certifique-se de entender pelo menos estas duas coisas:

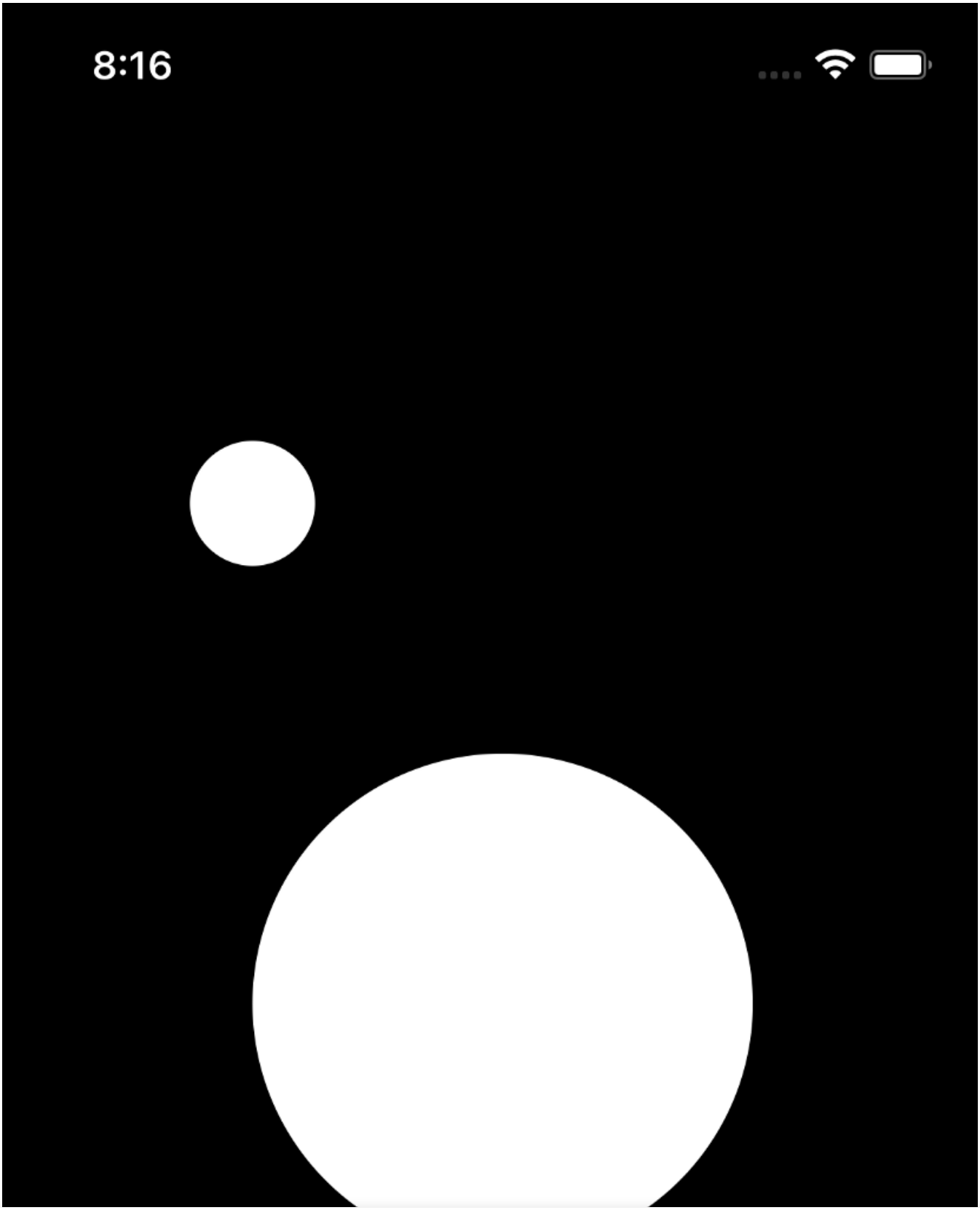
- Instruções com a sintaxe `M {xCoordinate} {yCoordinate}` dizem ao canvas para "mover-se" para as coordenadas fornecidas; esta instrução por si só não desenha nada, mas fornece as coordenadas iniciais para a próxima operação.
- Instruções com a sintaxe `L {xCoordinate} {yCoordinate}` dizem à tela para desenhar uma linha de sua posição atual para as coordenadas fornecidas.

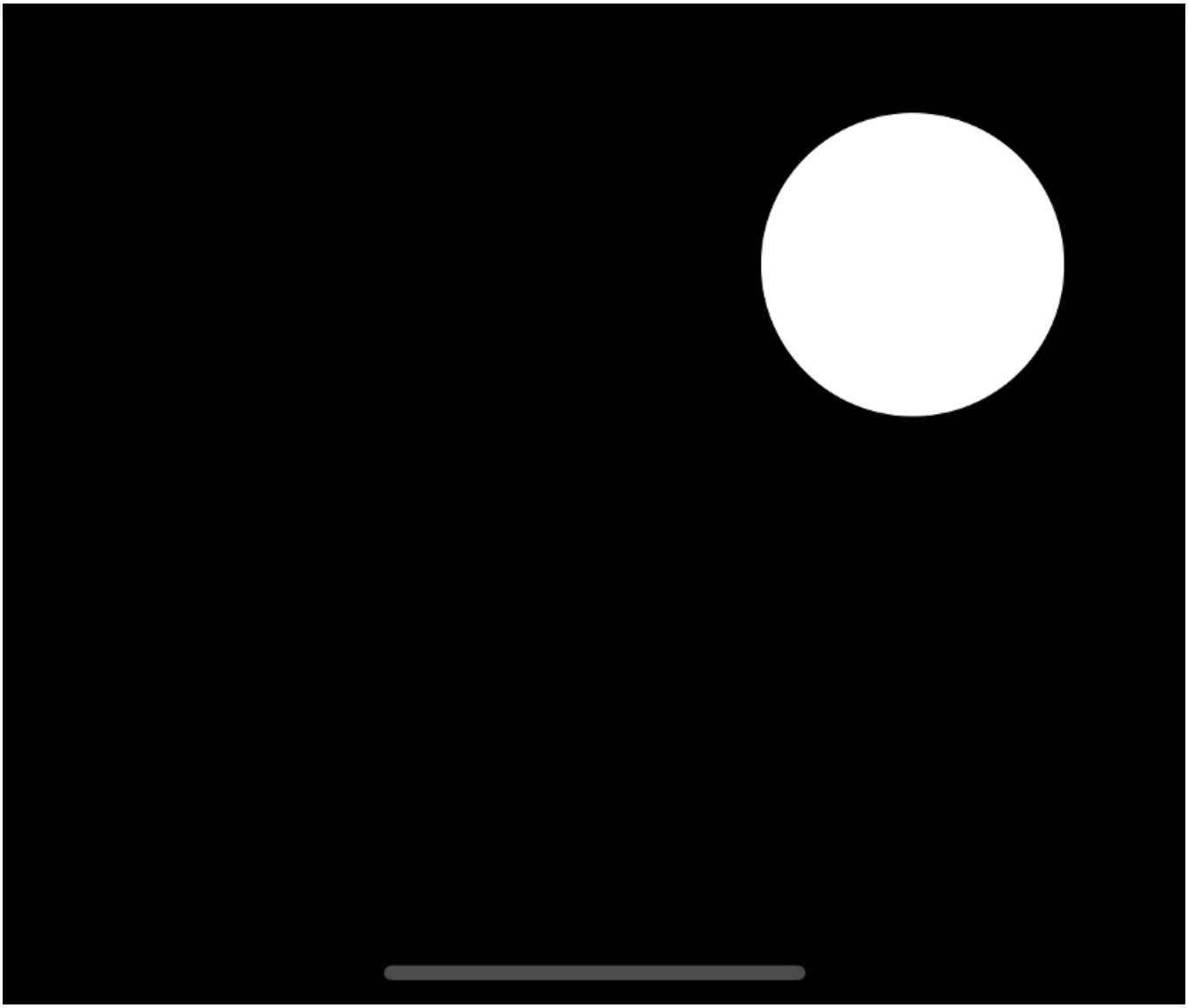
Também podemos facilmente desenhar formas em nossa tela. Veja como desenhamos círculos, por exemplo:

```
1  const Circles = () => {
2    return (
3      <>
4        <View style={{ flex: 1 }}>
5          <Canvas style={{ flex: 1, backgroundColor: "black" }}>
6            <Circle cx={200} cy={400} color="white" r={100} />
```

```
10          </View>
11        </>
12      );
13    };
```

med-9797f5f7b9b4_skia-drawing-circles.tsx hosted with ❤ by GitHub view raw

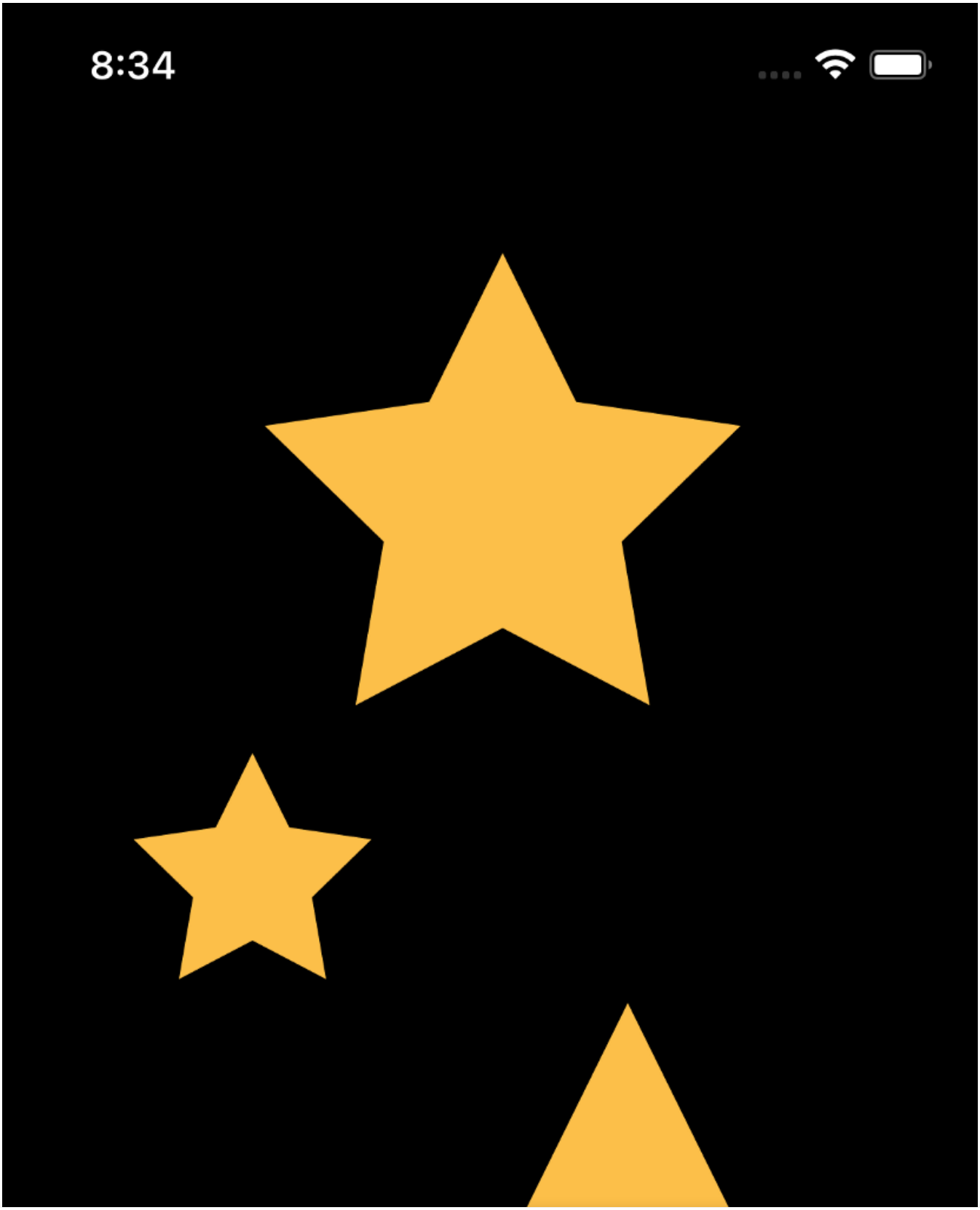




Finalmente, usando o `ImageSvg` componente do React Native Skia, podemos desenhar qualquer SVG arbitrário para nossa tela, incluindo aqueles que importamos de fontes externas.

```
1  const SvgImages = () => {
2    const svgStar =
3      '<svg class="star-svg" version="1.1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http:/
4    const svgImage = Skia.SVG.MakeFromString(svgStar);
5
6    return (
7      <View style={{ flex: 1 }}>
8        <Canvas style={{ flex: 1, backgroundColor: "black" }}>
9          {!!svgImage && (
10           <>
11             <ImageSVG width={200} height={200} x={100} y={100} svg={svgImage} />
12             <ImageSVG width={100} height={100} x={50} y={300} svg={svgImage} />
13             <ImageSVG width={300} height={300} x={100} y={400} svg={svgImage} />
```

```
16          </Canvas>
17        </View>
18      );
19    };
```



## Desenhar com Gesture Handler e Skia

Agora que entendemos o básico do Gesture Handler e do Skia, estamos prontos para juntar os dois conceitos e começar a desenhar!

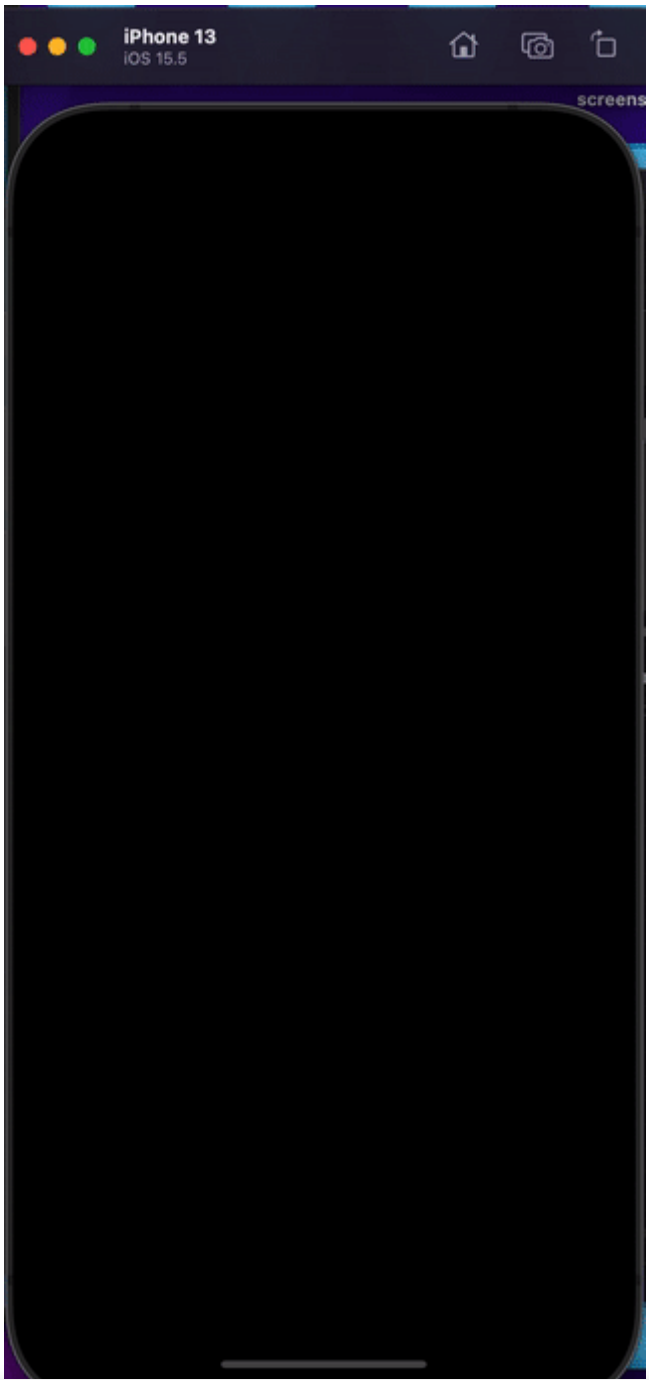
Criaremos um recurso de desenho à mão livre que captura dados de gestos de panorâmica do usuário e os usa para desenhar caminhos no Skia Canvas. Na verdade, isso é muito mais fácil do que pode parecer à primeira vista, se você tiver em mente estes conceitos:

- O manipulador de gestos `Gesture.Pan.onUpdate()` executará nossa função de retorno de chamada quando o usuário mover o dedo, fornecendo novas coordenadas do dedo do usuário a cada vez
- Skia Paths pode ser definido usando a notação SVG, que pode incluir uma série de `L {xCoord} {yCoord}` instruções para desenhar linhas

Com esses princípios em mente, confira este componente:

```
1  import React, { useState } from "react";
2  import { View } from "react-native";
3  import {
4    Gesture,
5    GestureDetector,
6    GestureHandlerRootView,
7  } from "react-native-gesture-handler";
8  import { Canvas, Path } from "@shopify/react-native-skia";
9
10 interface IPath {
11   segments: String[];
12   color?: string;
13 }
14
15 export default function Draw() {
16   const [paths, setPaths] = useState<IPath[]>([]);
17
18   const pan = Gesture.Pan()
19     .onStart((g) => {
20       const newPaths = [...paths];
21       newPaths[paths.length] = {
22         segments: [],
23         color: "#06d6a0",
24       };
25       newPaths[paths.length].segments.push(`M ${g.x} ${g.y}`);
26       setPaths(newPaths);
27     })
28     .onUpdate((g) => {
29       const index = paths.length - 1;
30       const newPaths = [...paths];
31       if (newPaths?.[index]?.segments) {
32         newPaths[index].segments.push(`L ${g.x} ${g.y}`);
33         setPaths(newPaths);
34       }
35     })
36     .minDistance(1);
37
38   return (
39     <GestureHandlerRootView style={{ flex: 1 }}>
40       <GestureDetector gesture={pan}>
41         <View style={{ flex: 1, backgroundColor: "black" }}>
42           <Canvas style={{ flex: 8 }}>
```


```
                path={p.segments.join(" ")}
                strokeWidth={5}
                style="stroke"
                color={p.color}
              />
            ))}
          </Canvas>
        </View>
      </GestureDetector>
    </GestureHandlerRootView>
  );
}
```



As partes mais importantes deste componente para entender são as linhas 25 e 32.

- Em 25, quando o Gesture Handler detecta o início de um gesto, colocamos um novo objeto de caminho em nosso `paths` array. Este objeto tem um array chamado `segments` que conterá uma lista de instruções de notação SVG para desenhar esse caminho. Inicialmente, criamos um comando “mover” para iniciar nosso caminho nas coordenadas iniciais de nosso gesto.
- Each time the gesture updates, we push a new path segment into our array (line 32). These segments contain a command in SVG notation to draw a line to the updated coordinates.

Then, we simply map over the values in `paths` inside our canvas to draw them. The `path` prop for each `Path` component is created by joining each segment string to form valid SVG notation.

There you have it! With fewer than 60 lines of code, we created a basic hand drawing component with React Native Gesture Handler and Skia.The power and potential of these two libraries combined really impress me. I hope you feel the same way!

Confira meu repositório RN Skia Draw no GitHub para um aplicativo Expo pronto para instalar que demonstra esses princípios e também mostra como "carimbar" as imagens de estrelas SVG mostradas acima `Canvas` usando um `Tap` gesto!

Como sempre, siga-me aqui no Medium e no Twitter @useRNRocket para mais conteúdo React Native! Envie-me um tweet com comentários, perguntas ou correções.