Performance

Measuring Performance

React Native Tools

Performance

opening developer menu in the simulator:

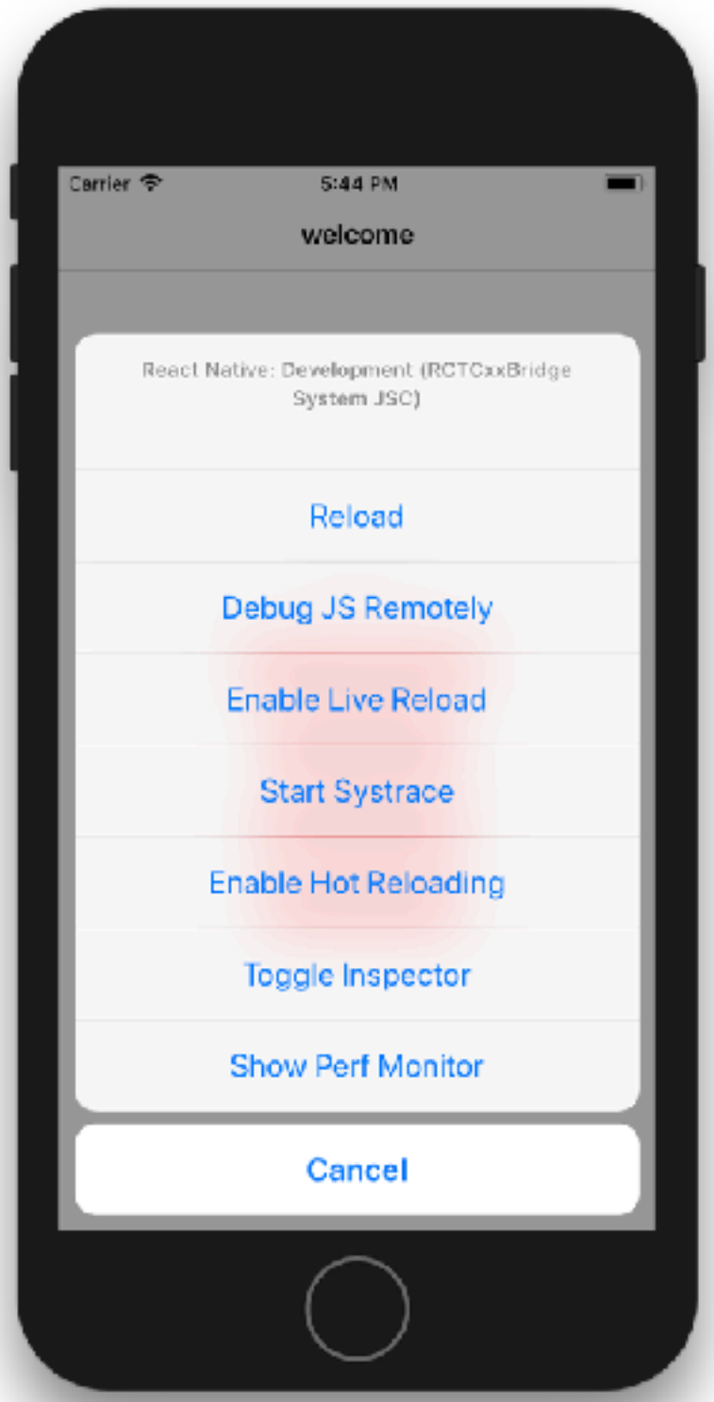
- iOS – cmd + d
- Android – cmd + m

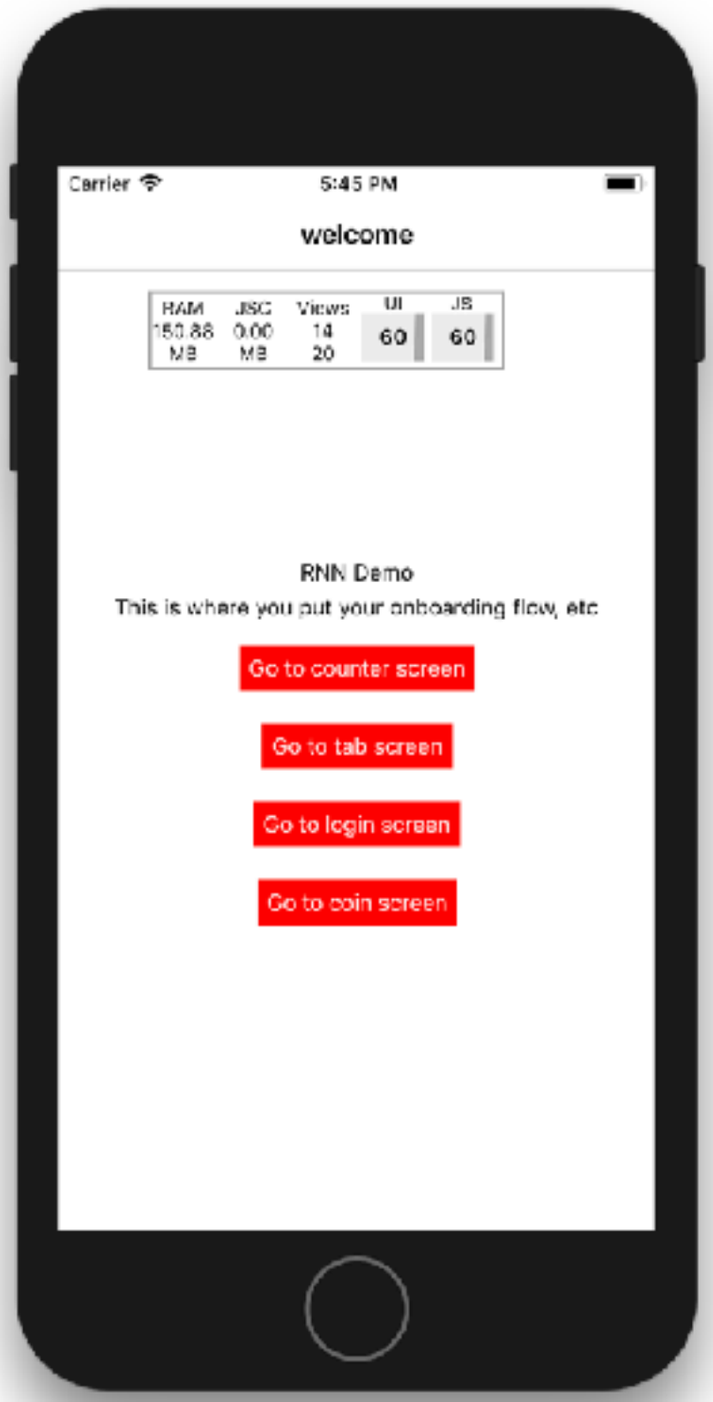
opening developer menu on the device:

- Debug builds: Shake.

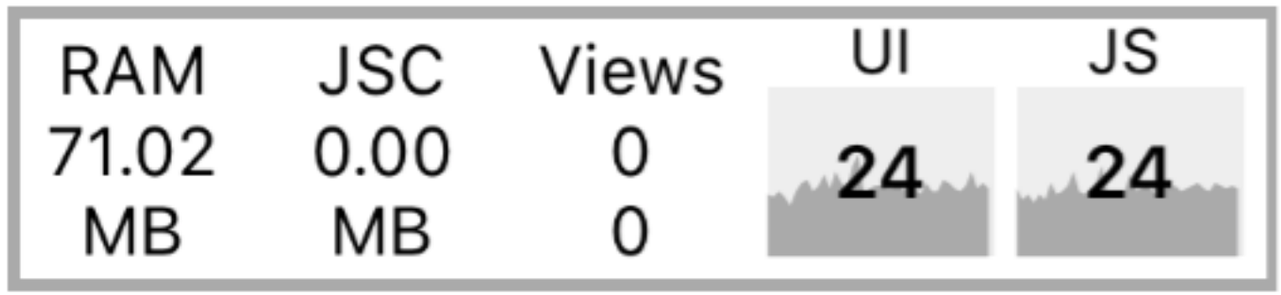
Performance

Choose Show Perf Monitor





Performance



Ram: Memory

JSC: Javascript heap / memory. Will only be updated as garbage collection occurs.

Views:

- Top number is the number of views on the screen.
- Bottom is the total number of views in the component. Bottom number typically larger but usually indicates you have something that could be improved / refactored.

JS frame rate (JavaScript thread):

- In most React Native applications, business logic will run on the JavaScript thread.
  - React application lives here.
  - API calls made here.
  - Touch events processed.
- Updates to native-backed views are batched and sent over to the native side at the end of each iteration of the event loop, before the frame deadline (if all goes well).

UI frame rate (main thread)

- Thread where all the native rendering occurs.
- Example: NavigatorIOS is better out of the box than Navigator. The reason for this is that the animations for the transitions are done entirely on the main thread, and so they are not interrupted by frame drops on the JavaScript thread.

Case Study

https://launchdrawer.com/i-made-react-native-fast-you-can-too-9e61c951ce0

Performance

I created an example app consisting of a scrollable list of images from Flickr. I'll talk through my thought process and provide links to pull requests and performance data as I perform various simple optimizations on the app. The source code for the app can be found here and you can see a gif of it in action below.

Example Source:

https://github.com/hgale/PerformanceExample

Performance

What is Overdraw?

“Overdraw refers to the system’s drawing a pixel on the screen multiple times in a single frame of rendering. For example, if we have a bunch of stacked UI cards, each card hides a portion of the one below it.

However, the system still needs to draw even the hidden portions of the cards in the stack. This is because stacked cards are rendered according to the painter’s algorithm: that is, in back-to-front order. This sequence of rendering allows the system to apply proper alpha blending to translucent objects such as shadows.”

All applications will have some amount of overdraw but too much of it can cause performance problems.

Performance

How much Overdraw should our app have?

According to the a case study by prominent Android developer Romain Guy:

> *“The amount of overdraw you can reasonably afford varies from device to device. A good rule of thumb is to aim for a maximum overdraw of 2x; this means you can draw the screen once, then draw twice again on top, painting each pixel 3 times total.”*

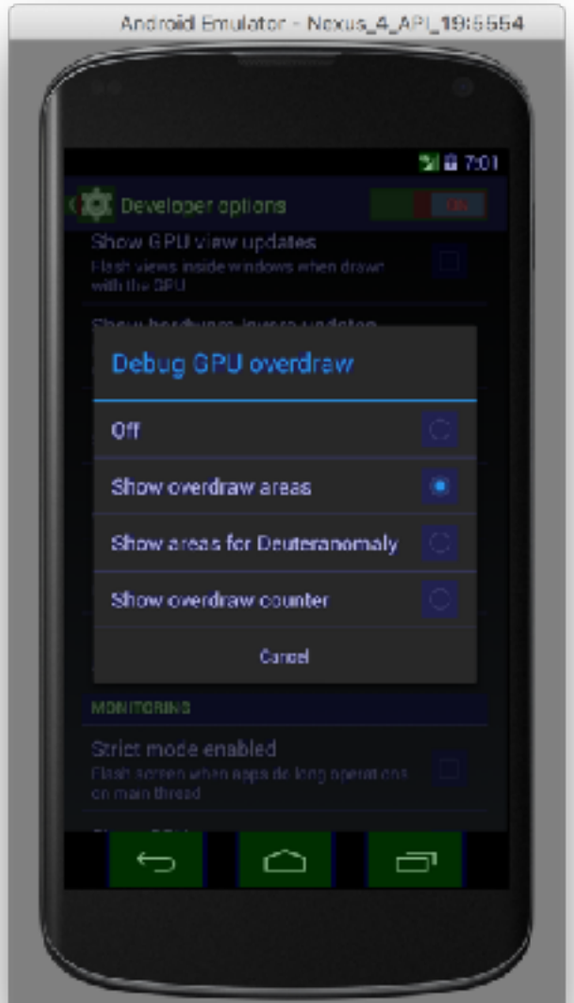
All applications will have some amount of overdraw but too much of it can cause performance problems.

Performance

How do you measure Overdraw?

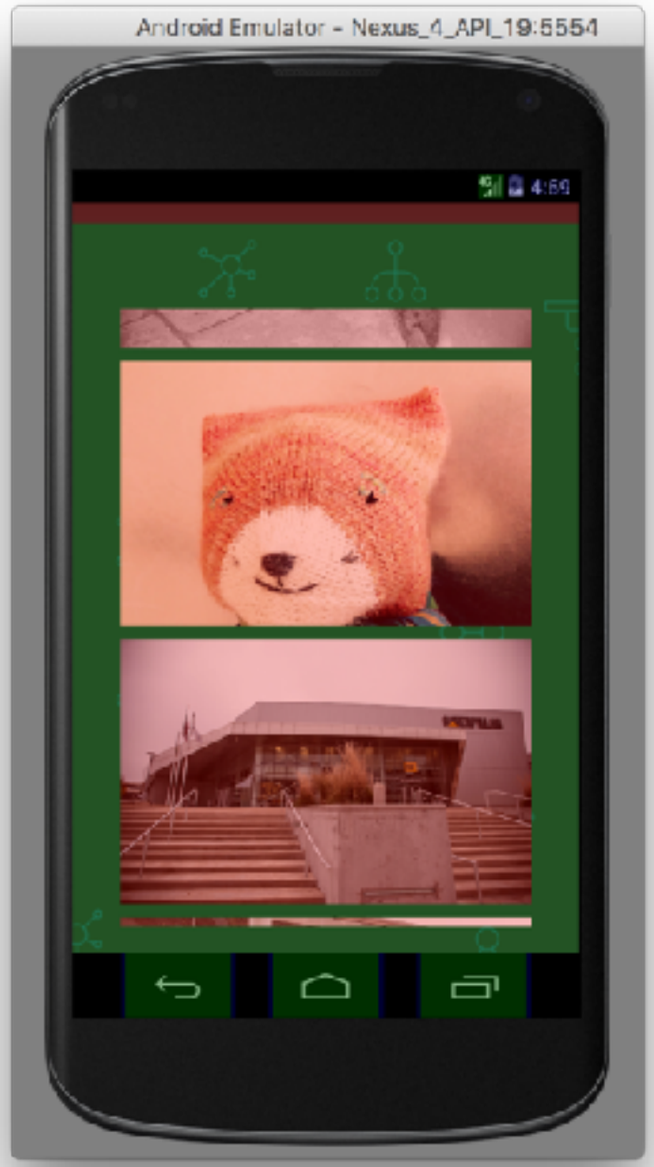
Google gives you two easy ways to tell how much overdraw your app has. This can be seen on a device by displaying either a numeric counter or a colored overlay highlighting parts of the UI that are overdrawn.

To see the colored overlay, in the emulator (or your device), enable developer options then go into developer settings and select Debug GPU Overdraw, then select show Overdraw areas:



Performance

The example apps screen is now overlaid with various colors that correspond to overdraw.



Performance

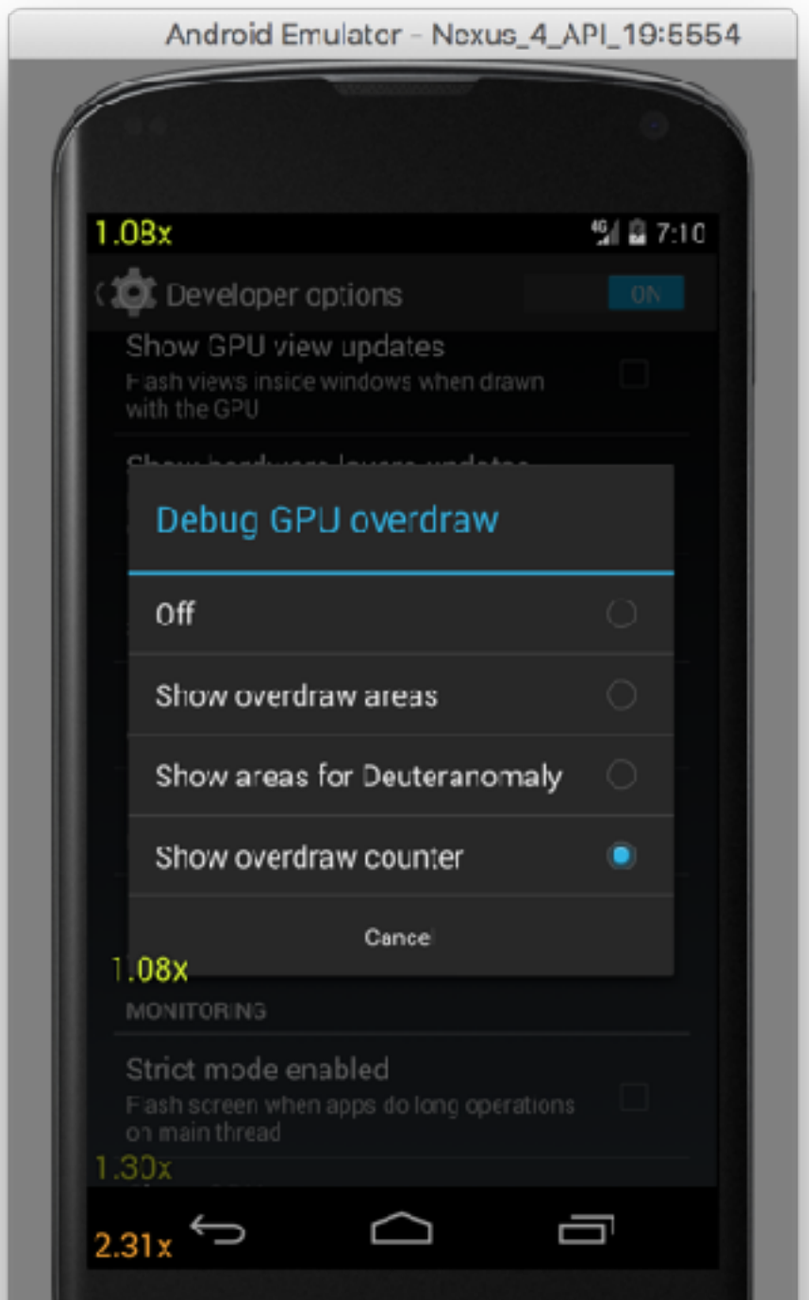
What do the colors mean?

According to the overdraw developer docs:

“The colors are hinting at the amount of overdraw on your screen for each pixel, as follows:
True color: No overdraw
Blue: Overdrawn once
Green: Overdrawn twice
Pink: Overdrawn three times
Red: Overdrawn four or more time””

Performance

To enable the overdraw counter go back into developer settings and select show overdraw counter.



Performance

The counter now shows that the app has an overdraw of 3.56. Lets see if we can figure out how to reduce this number.

Performance

What's causing the Overdraw?

If you go back to the colored overlay of the app you will notice that part of the background is red. Lets try commenting out the background component and see what happens. All I'm doing at this point is swapping the BgView component I created for a standard View:

```
<BgView>
   <ScrollView contentContainerStyle={{justifyContent: 'center'}} style={[{height:height}]}>
     {cells}
     </Text>
</BgView>



 <View>
    <ScrollView contentContainerStyle={{justifyContent: 'center'}} style={[{height:height}]}>
      {cells}
      </Text>
 </View>
```

Performance

Doing this causes the overdraw number to go from 3.56 to 1.81 and seems to suggest that this issue might be being caused by our background component.

Performance

The background component used in the example app looks like this:

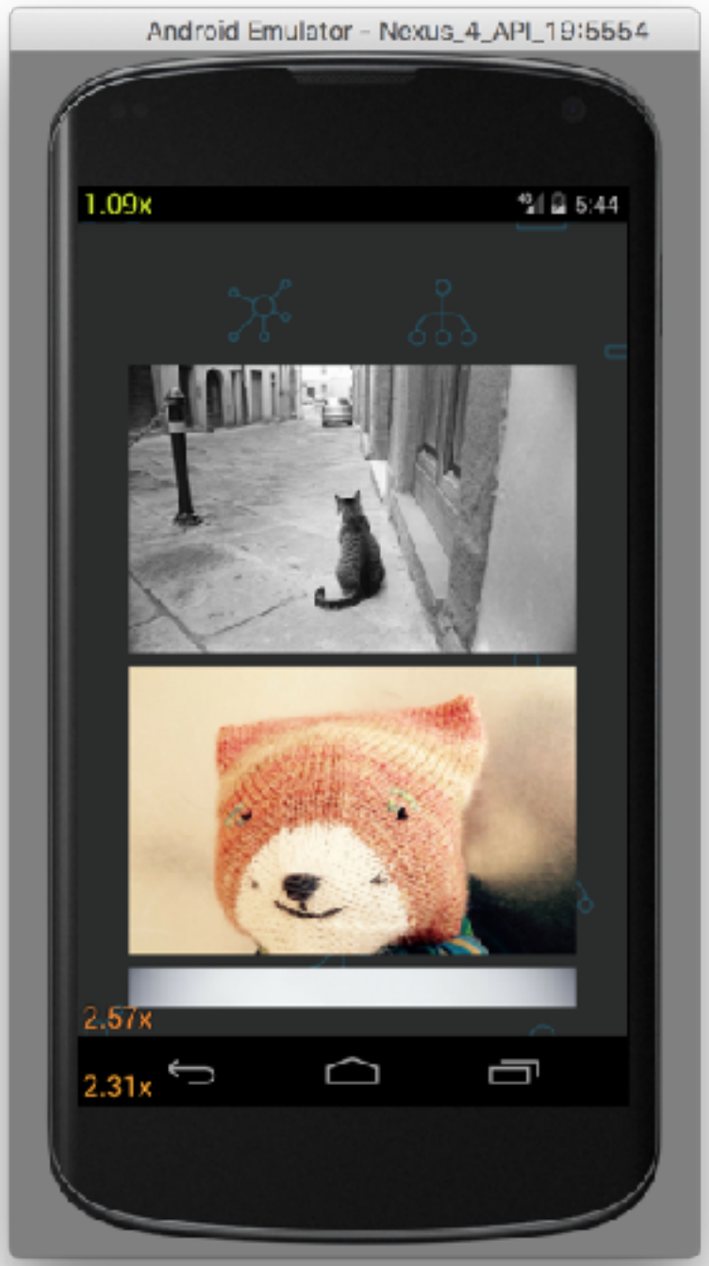
```
import bgImage from './assets/bg_transparent.png';

export const BgView = (props) => {
  const propsStyle () = {
    backgroundColor: 'transparent'
  }
   return(
    <Image
    source={bgImage}
    style={style.pageBackground}>
      <Text>{this.state.name}</Text>
    </Image>
  )
}
```

One thing that immediately stands out to me is the use of a transparent background color and the image file name bg_transparent.png. The bg_transparent.png is a transparent background image with some shapes on top.

The needless use of transparent colors and backgrounds in an app's design can often cause performance issues. Lets see what happens when we replace this transparent background with one that keeps the shapes but uses a solid color. This will allow us to get rid of all the transparent background colors being used.

Performance

As you can see below the overdraw number goes from 3.56 to 2.57. A significant improvement for just changing your background.



Diff:
https://github.com/hgale/PerformanceExample/pull/2

Performance

How are we rendering this list of images?

A common use case for mobile apps is to display long lists of items, as such native platforms have dedicated components for doing this efficiently. An example of this can be seen with UICollectionView for iOS and RecyclerView for Android.

As you can see below the performance example app renders all of its images at once inside of a ScrollView.

```
class HomeScreen extends React.Component {

  renderCells (data) {
    return data.map((cell, index) => {
      const {uri} = cell
      return (
        <View key={index} style={style.cellContainer}>
          <Image style={style.imageContainer} source={{uri:uri}}>
          </Image>
        </View>
      )
    })
  }

  renderImageCells (cellData) {
    return (
      <View style={style.imageCellsContainer}>
        {this.renderCells(cellData)}
      </View>
    )
  }

  render() {
    const { height } = Dimensions.get('window')
    let cells = this.renderImageCells(FlickrImages)
    return (
      <BgView>
        <ScrollView contentContainerStyle={{justifyContent: 'center'}} style={[{height:height}]}>
          {cells}
        </ScrollView>
      </BgView>
    );
  }
}
}
```

Performance

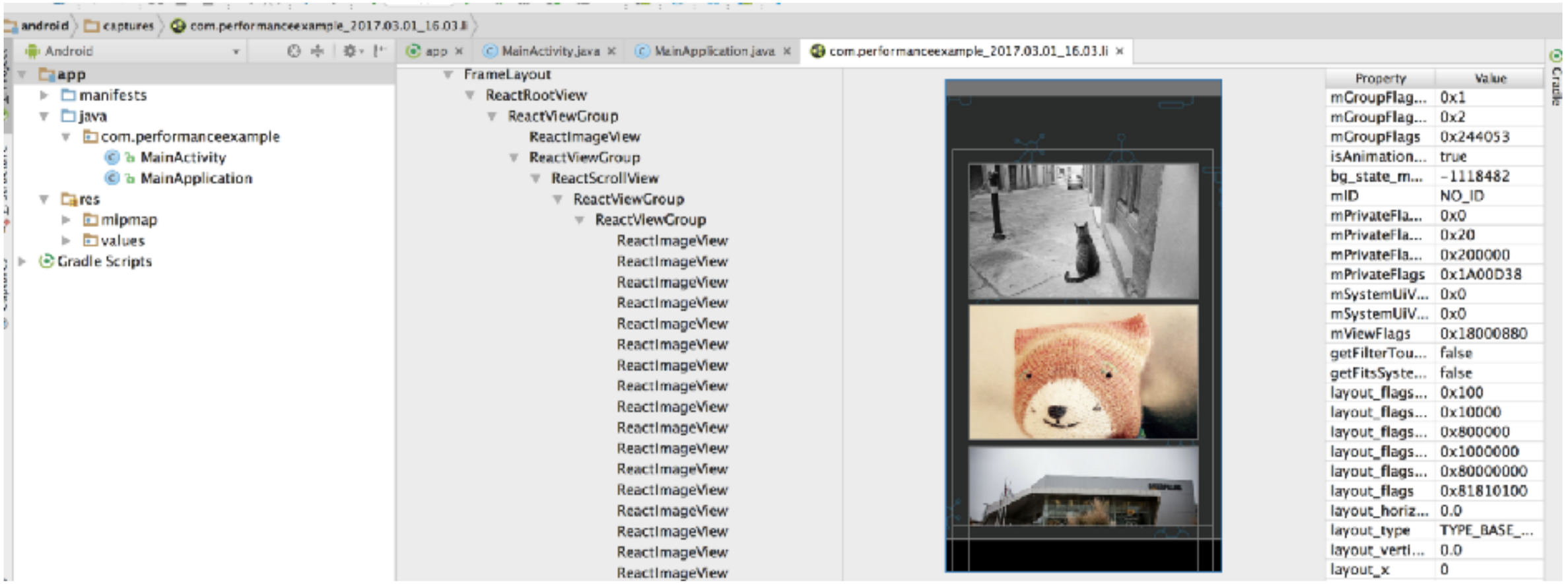
Why is this bad?

This is bad because it means that we are rendering all of the images including the ones that are not visible to the user on screen. On a mobile device with limited memory this will result in poor performance. Also the app will simply not work if the list of images exceeds the devices available memory.

Performance

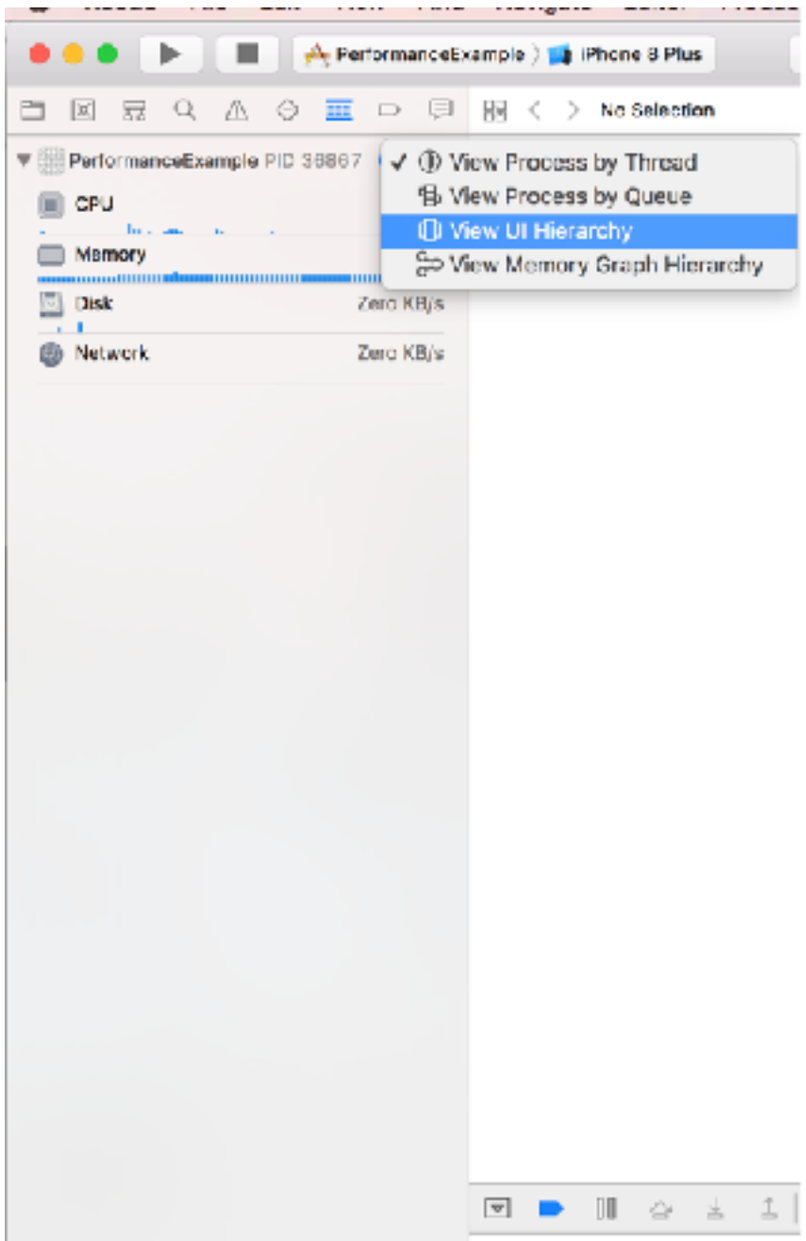
Why is this bad?

To illustrate this point, lets go into Android Studio and look at the layout inspector while the app is running. The layout inspector allows you to see the hierarchy of views being displayed at any given time. This is what it looks like for our ScrollView implementation:



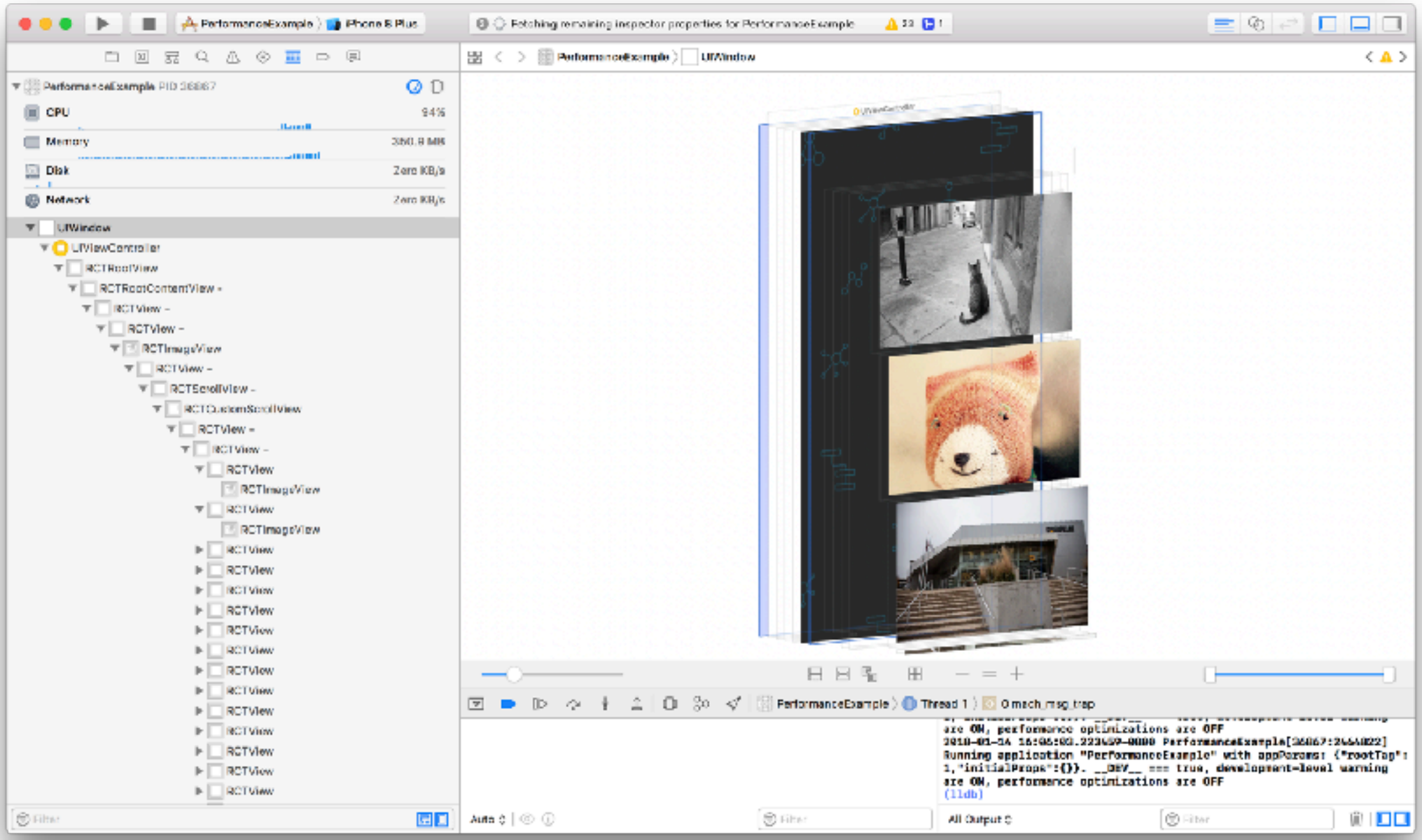
Performance

Xcode Allows you to do the same thing with the hierarchy viewer which you can access like so, by pausing the app and going to the following menu on the left hand side:



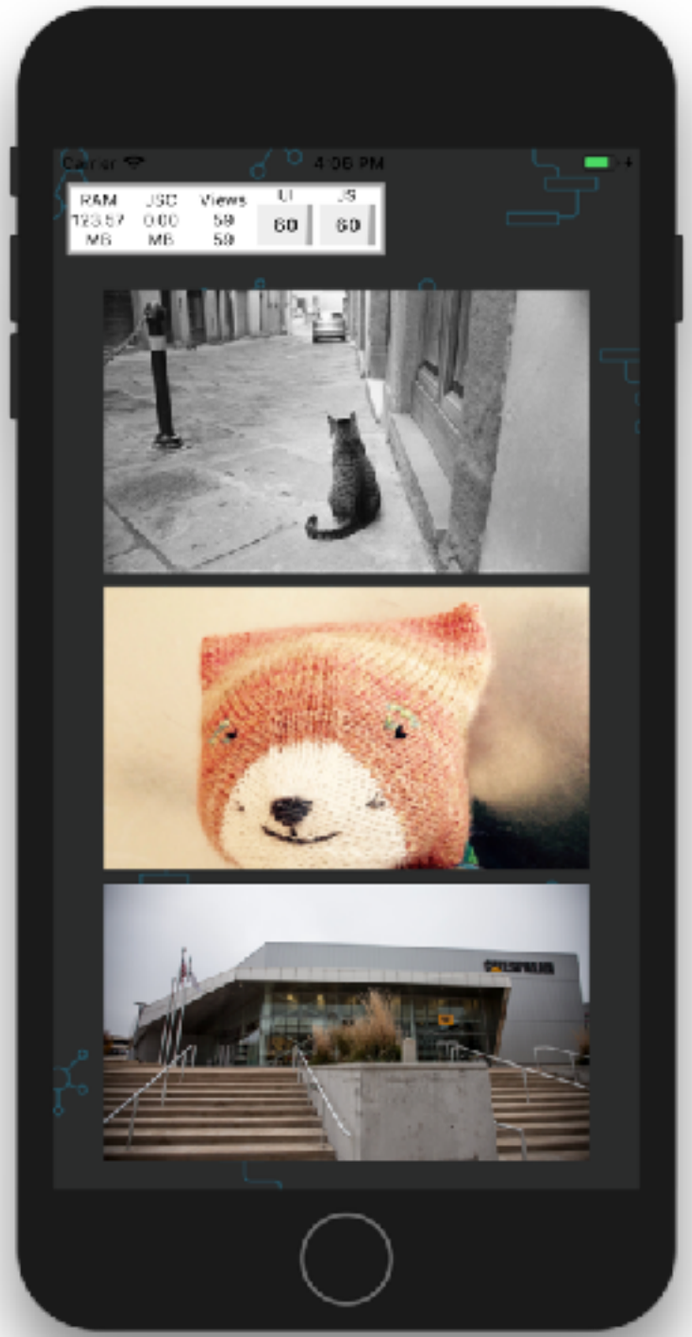
Performance

When you do this you see that all of the views are being displayed, even the ones not visible to the user are being rendered at once:



Performance

However, you could have figured this out by just looking at the react native perf tool, like so:



Performance

So what's a more efficient way to display a list of items with React Native? The answer is to use a ListView.

What does a ListView do?

"The ListView component displays a vertically scrolling list of changing, but similarly structured, data.
ListView works well for long lists of data, where the number of items might change over time. Unlike the more generic ScrollView, the ListView only renders elements that are currently showing on the screen, not all the elements at once."

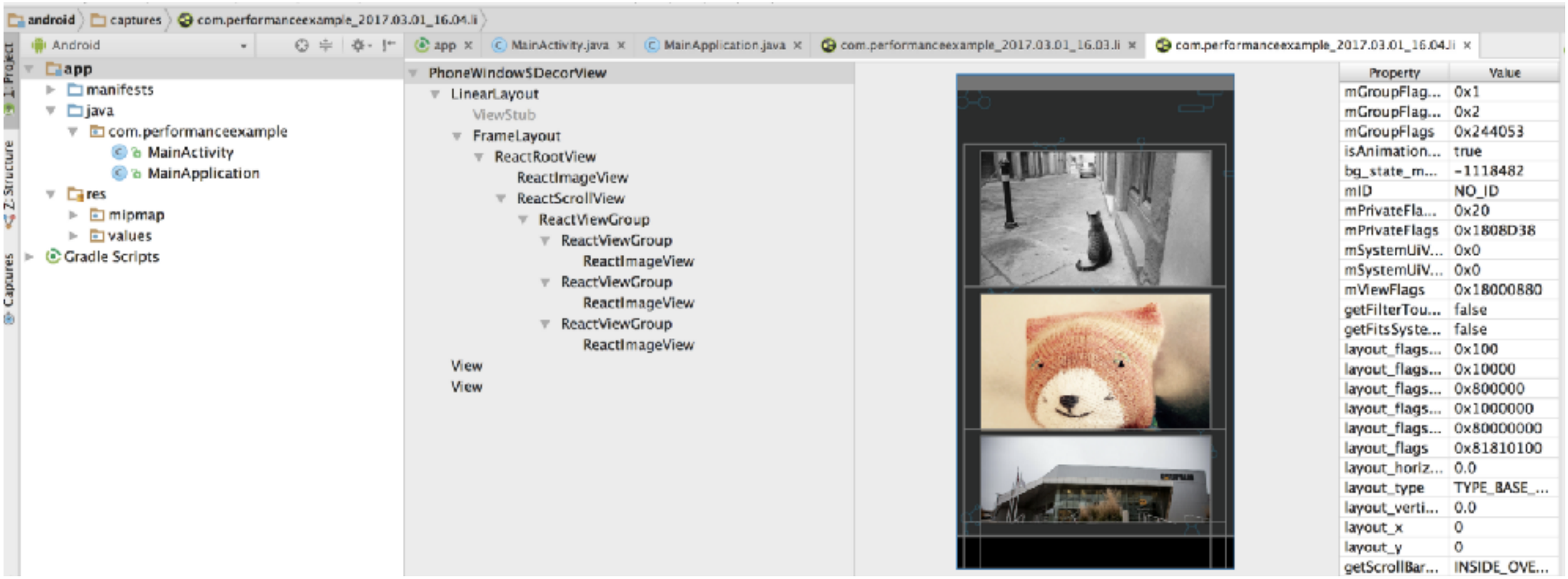
Diff:
https://github.com/hgale/PerformanceExample/pull/3/

ListView:

```
import ImageRow from ‘./ImageRow'

class HomeScreen extends React.Component {
  constructor (data) {
     super (props)
     const ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1.uri !== r2.uri})
     this.state = {
       dataSource: ds.cloneWithRows(FlickrImages)
     }
  }

  render() {
    const { height } = Dimensions.get('window')
    let cells = this.renderImageCells(FlickrImages)
    return (
      <BgView>
        <ListView
        contentContainerStyle={{justifyContent: 'center'}}
        style={[{height:height}]}
        dataSource={this.state.dataSource}
        renderRow={(rowData) => <ImageRow {...rowData} />}
        />
      </BgView>
    );
  }
}
}


import React from 'react'
import { View, Image } from 'react-native'

import style from './Style'

const ImageRow = (props) => (
  <View style={style.cellContainer}>
    <Image style={style.imageContainer} source={{uri:props.uri}}/>
  </View>
)
export default ImageRow
```

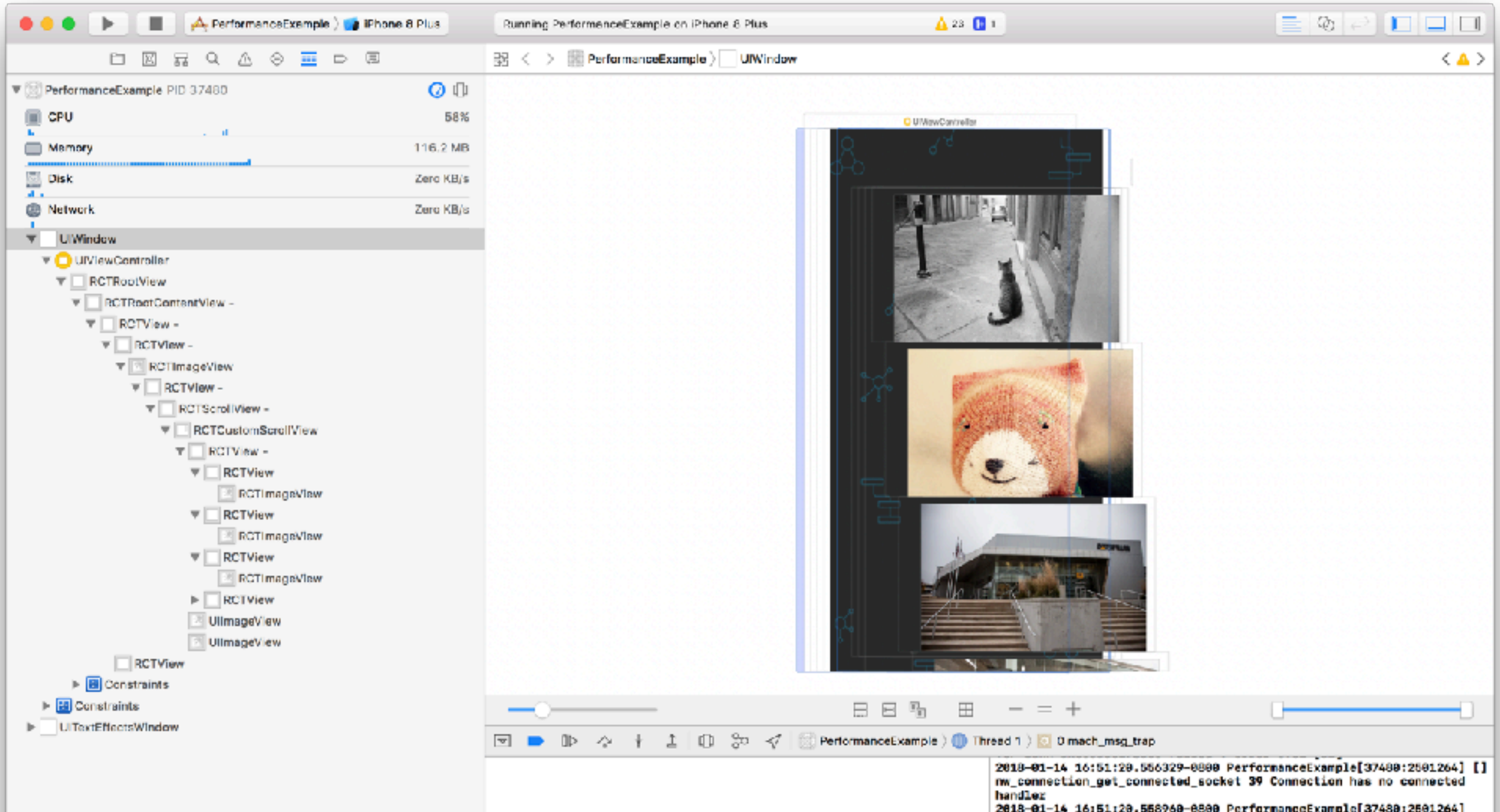
Performance

This is what the ListView implementation looks like in layout inspector. As you can see only three images are rendered as opposed to the entire list:



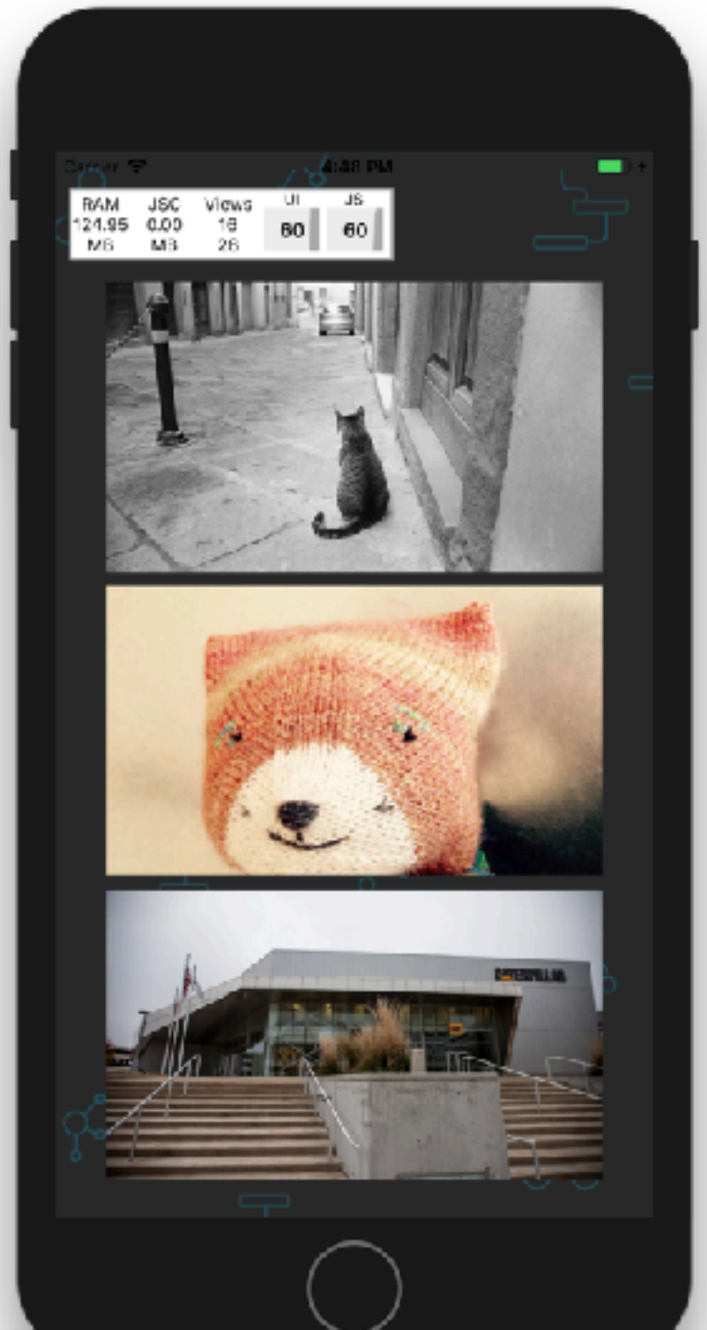
Performance

This is what the ListView implementation looks like in Xcode:



Performance

This is what the ListView implementation looks like in the perf monitor:



It’s 2018, should I still use ListView?

No, use FlatList or SectionList.

Working with lists

map vs FlatList vs SectionList

Working with lists

FlatList:

A FlatList is more performant compared to a ListView. ListView rendering can get slow once the number of items grows larger. FlatList significantly improves memory usage and efficiency (especially for large or complex lists) while also significantly simplifying the props — no more dataSource necessary!

- Scroll loading (onEndReached).
- Pull to refresh (onRefresh / refreshing).
- Configurable viewability (VPV) callbacks (onViewableItemsChanged / viewabilityConfig).
- Horizontal mode (horizontal).
- Intelligent item and section separators.
- Multi-column support (numColumns)
- scrollToEnd, scrollToIndex, and scrollToItem
- Better Flow typing.

Besides simplifying the API, the new list components also have significant performance enhancements, the main one being nearly constant memory usage for any number of rows. This is done by 'virtualizing' elements that are outside of the render window by completely unmounting them from the component hierarchy and reclaiming the JS memory from the react components, along with the native memory from the shadow tree and the UI views.

Working with lists

SectionList:

A performant interface for rendering sectioned lists, supporting the most handy features:

- List header support.
- List footer support.
- Item separator support.
- Section header support.
- Section separator support.
- Multi-column support (numColumns)
- scrollToEnd, scrollToIndex, and scrollToItem
- Better Flow typing.

If you don't need section support and want a simpler interface, use FlatList.

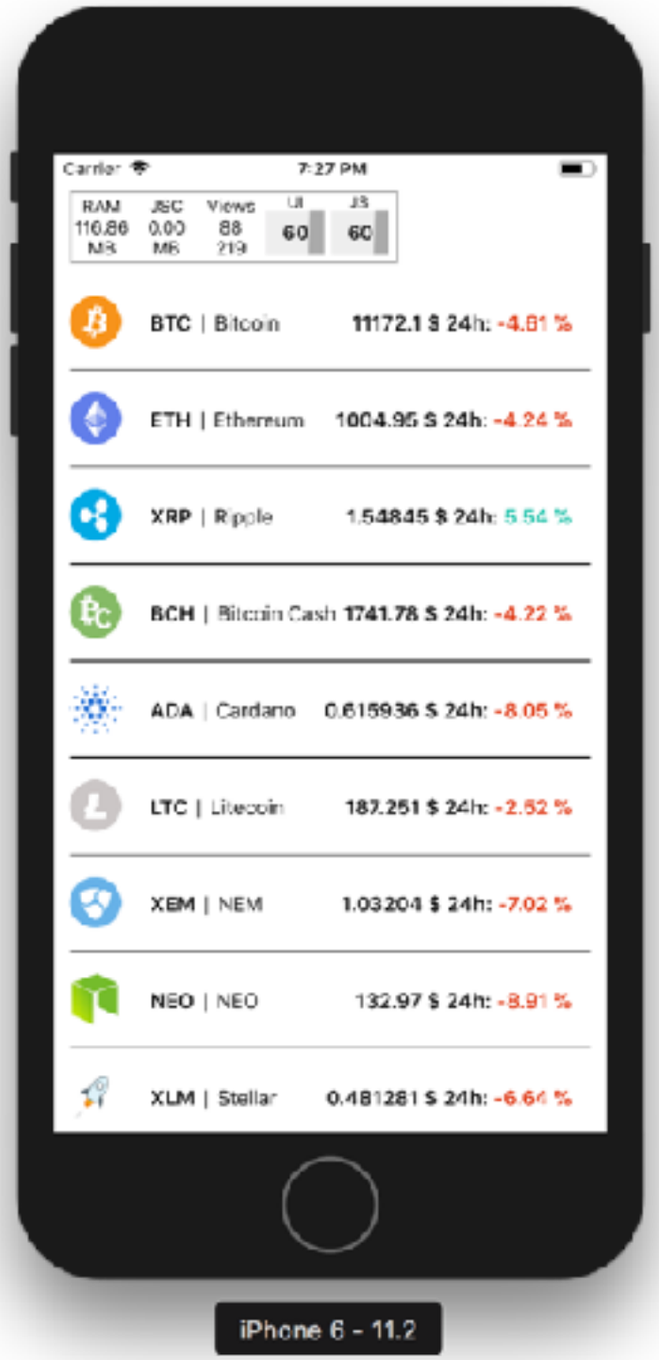
Convert crypto coin demo to use a Flatlist:

https://github.com/hgale/ComponentAndState/pull/1

https://facebook.github.io/react-native/docs/flatlist.html

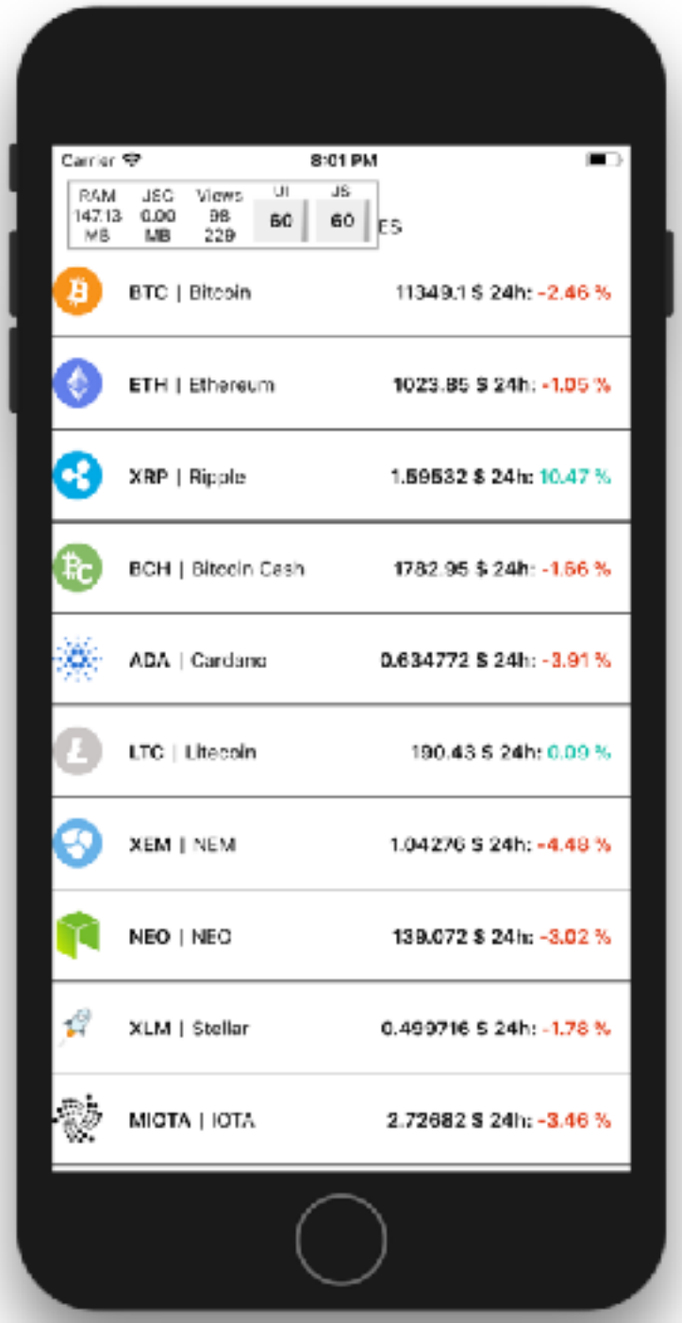
Diff:
https://github.com/hgale/ComponentAndState/pull/2

Before:



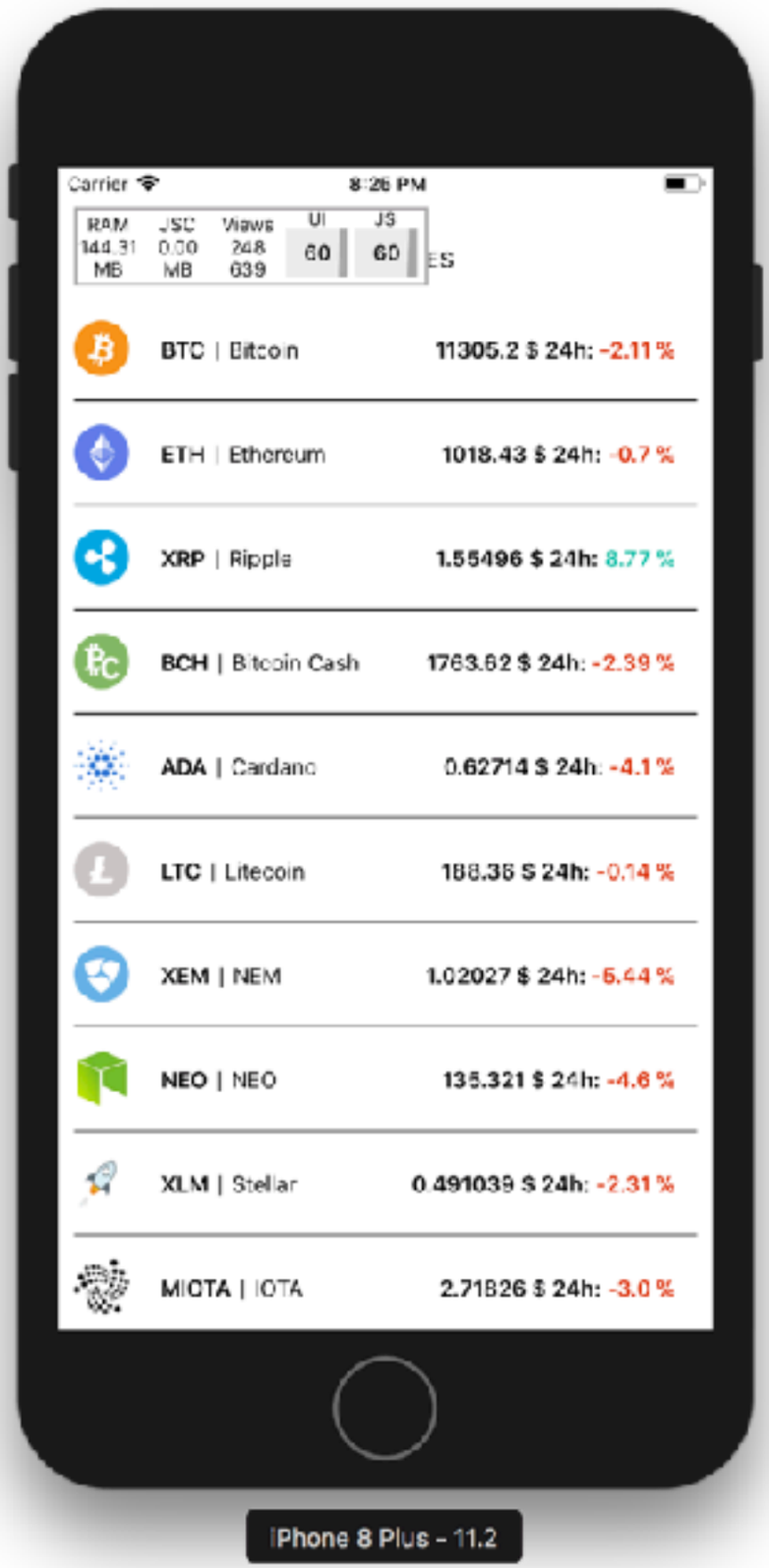
iPhone 6 - 11.2

After:

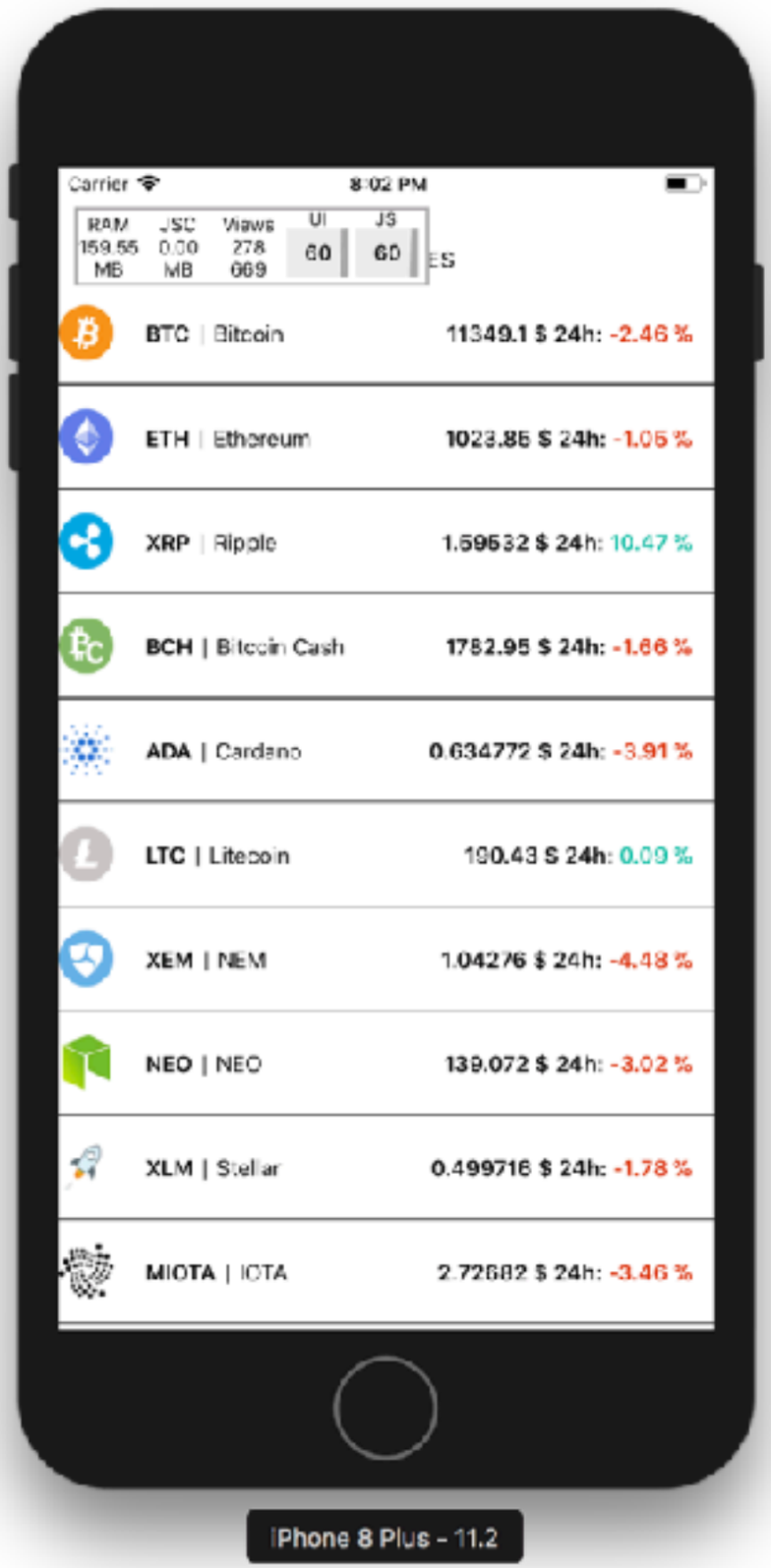


iPhone 6 Plus - 11.2

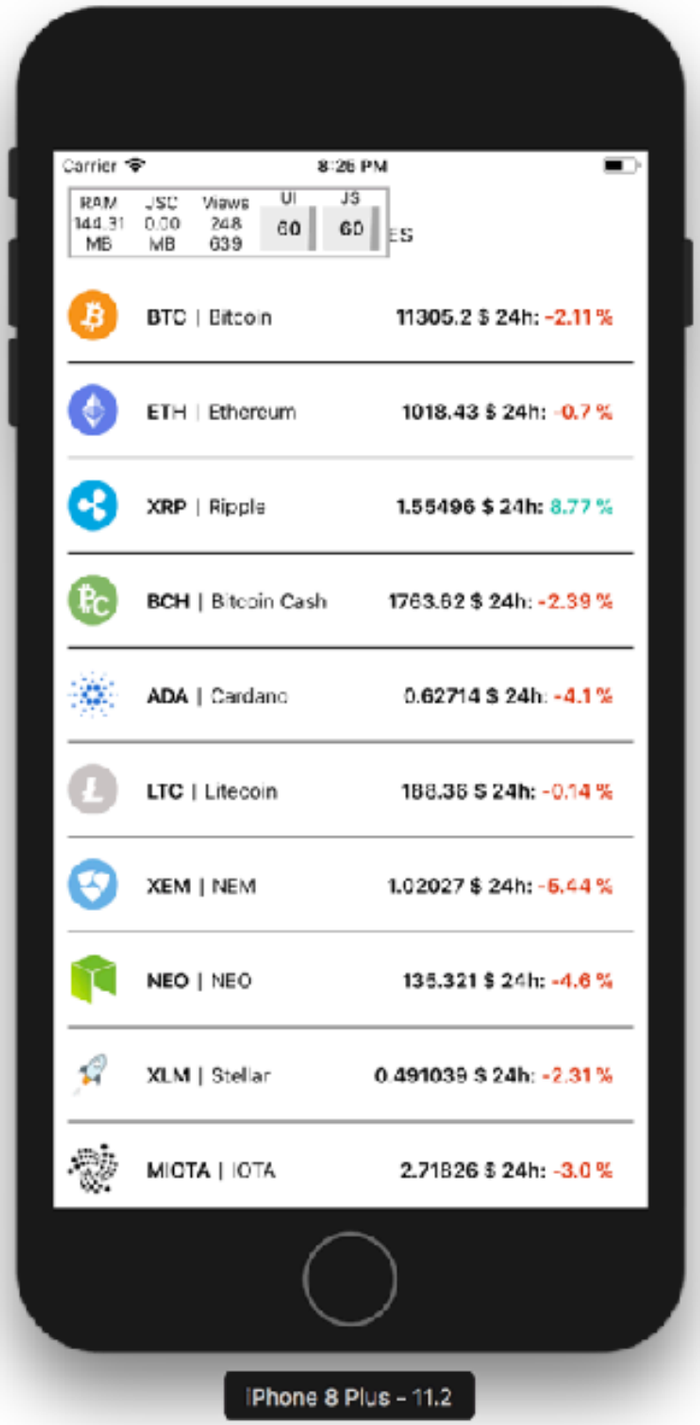
Before 30 items:



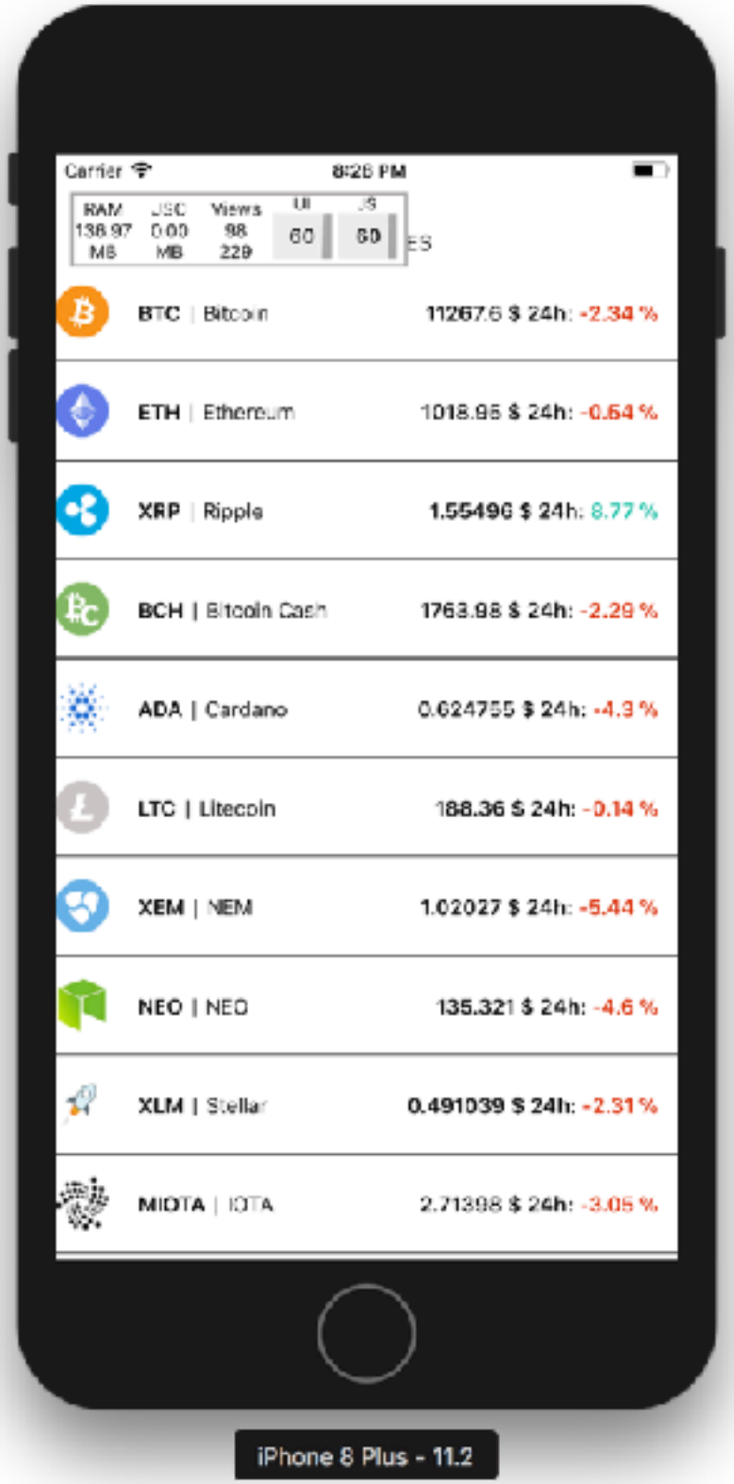
After 30 items:



Before 30:



After: initialNumToRender={10}
removeClippedSubviews

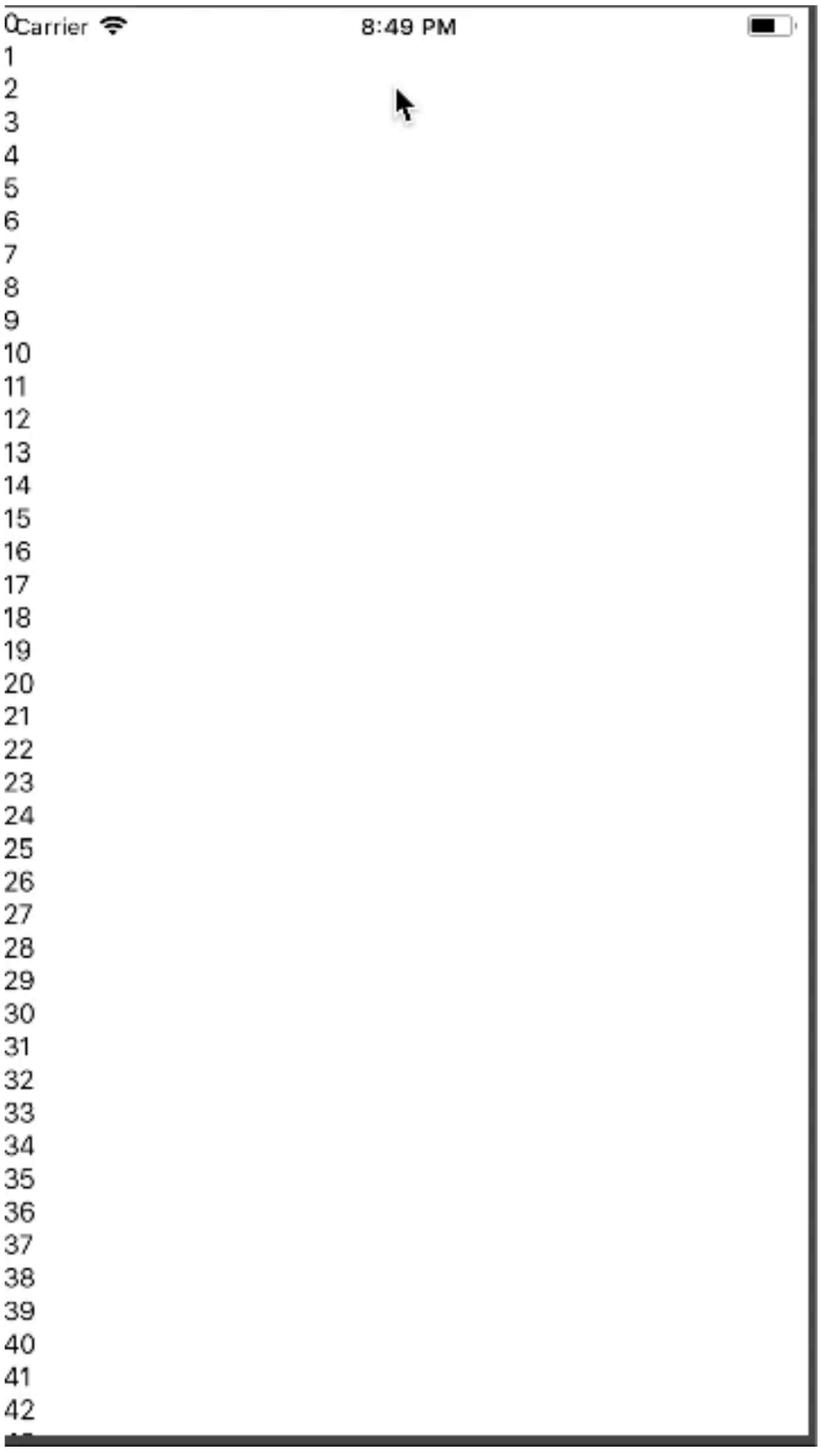


So where does Flatlist make a difference?:

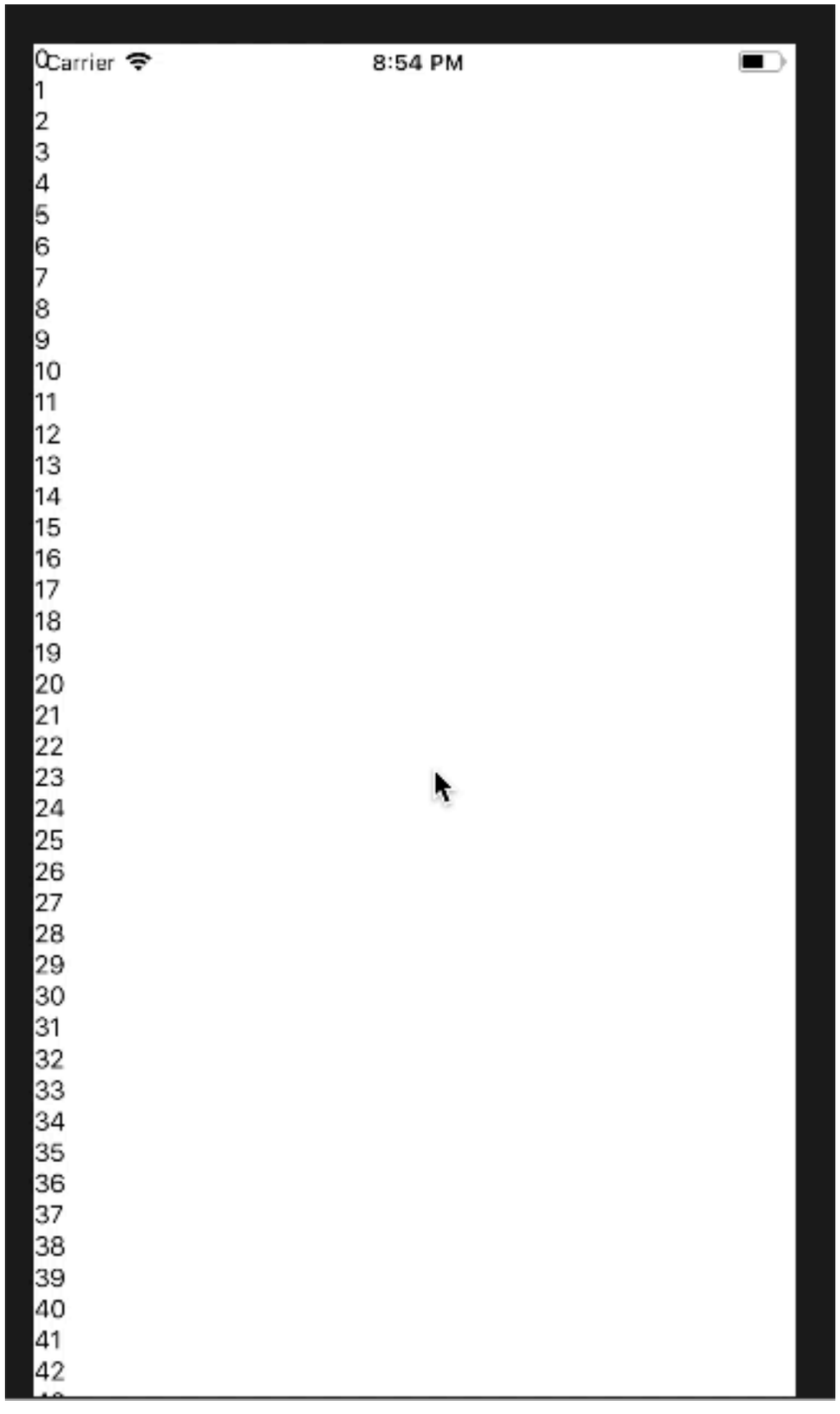
Numbers demo:

https://github.com/hgale/ComponentAndState/pull/3
http://matthewsessions.com/2017/05/15/optimizing-list-render-performance.html

Before:


Carrier
8:49 PM
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

After FlatList:



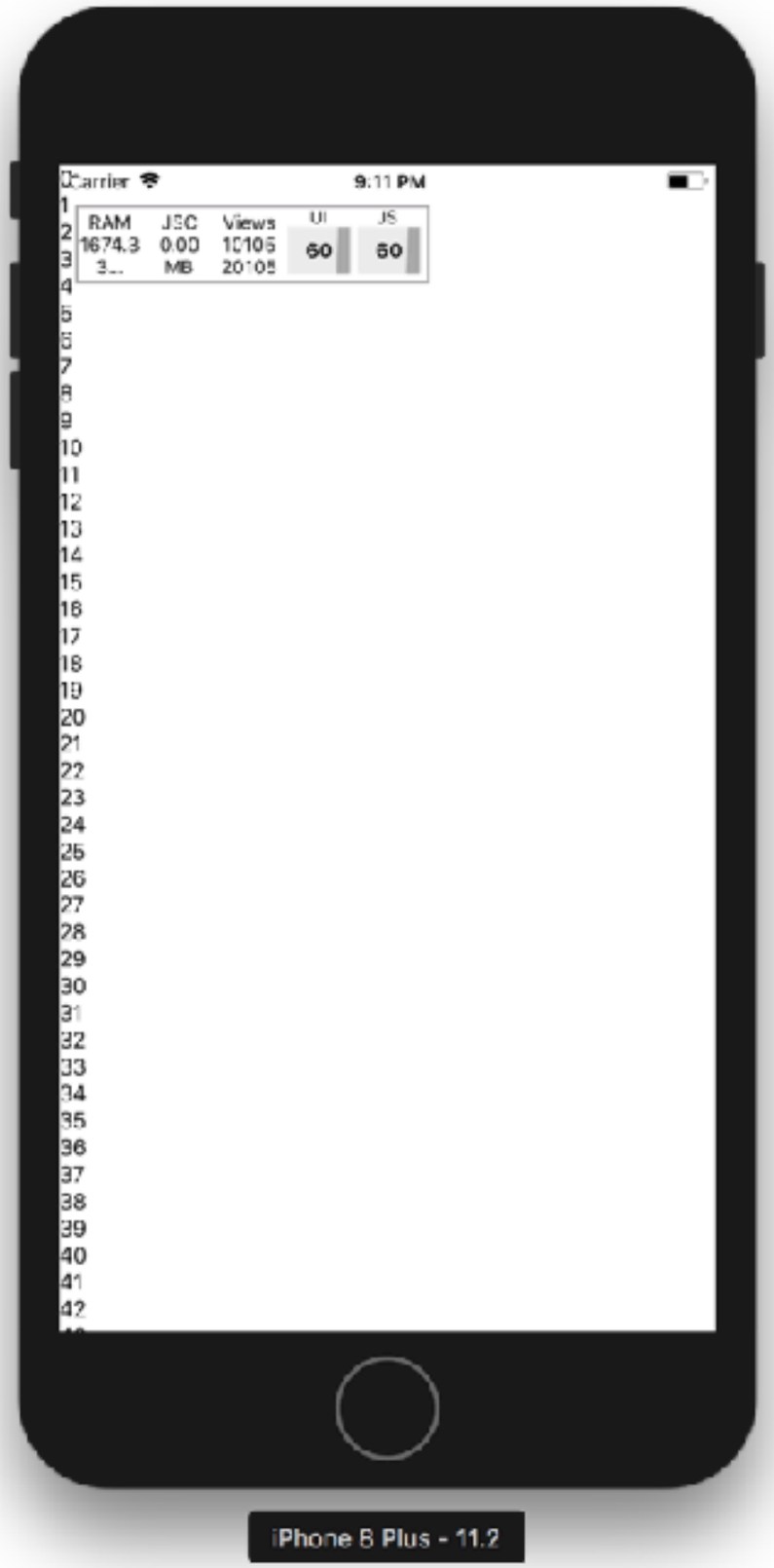
Working with lists

What are we optimizing? :

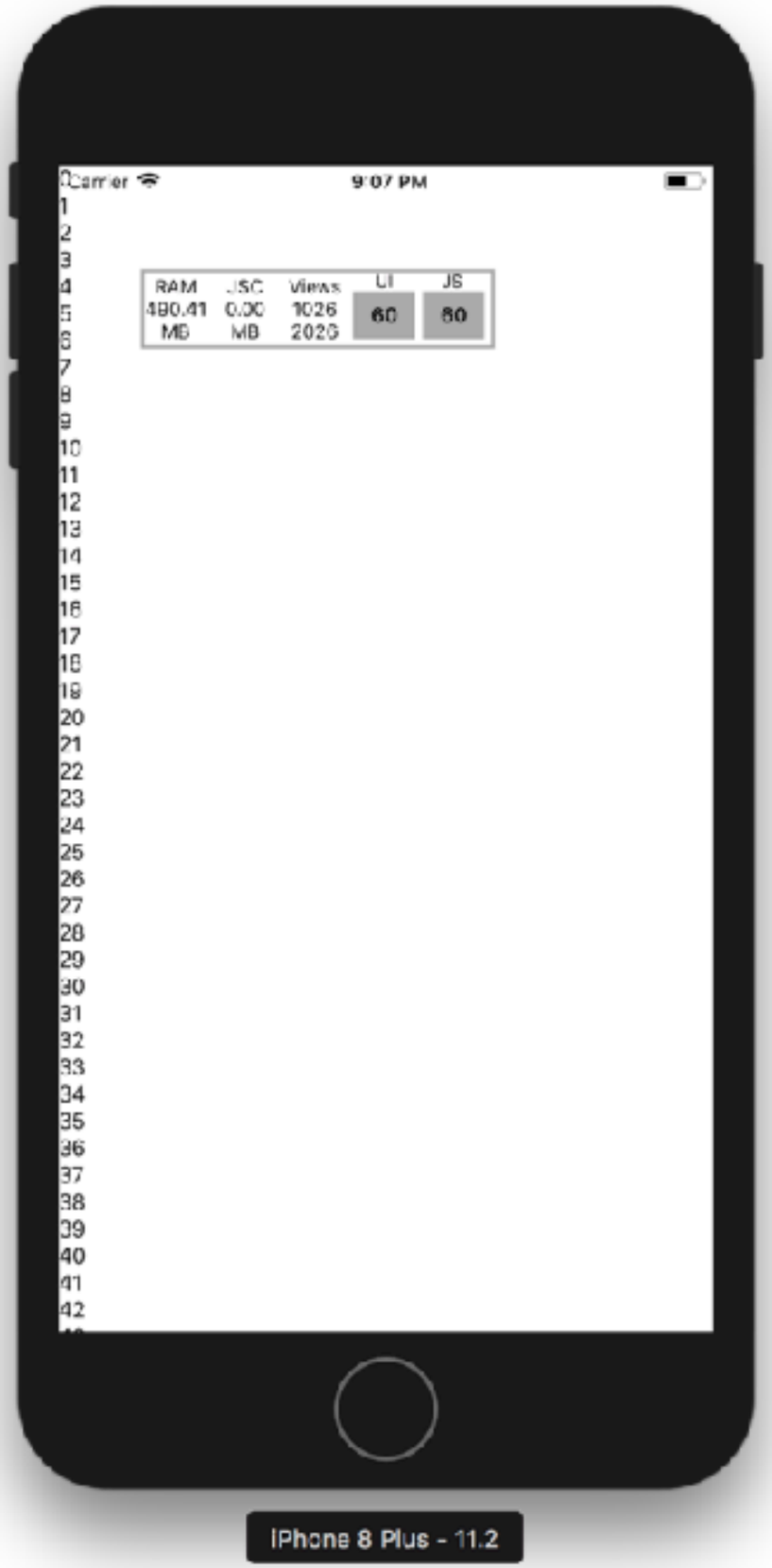
“React has a ton of optimizations for making updates to the render tree extremely performant, such as virtual dom diffing and the shouldComponentUpdate method on a component. But when you are doing an initial render, the only such optimization, is to just not render as much. That can be hard though when rendering a long list of complex views.”

“FlatList can immediately improve native side rendering. It’s a common pattern in iOS development to have lists recycle views. So if you have a very long list, the app is actually only rendering the views that are actually on screen. As you scroll through the list, and one of the rows leaves the screen, it will get recycled and added to the other side of the list with the data for the next row in the list. This improves memory usage for an app drastically as it is not required to allocate memory for every row in a list. But it has another advantage of only rendering views that are in screen. Which means that views that are not on screen won’t even get rendered. FlatList behaves much like these native UI list views. FlatList exposes this behavior to lists in React Native which can drastically decrease render time for complex lists."

Before:



After:



For more details see:

https://facebook.github.io/react-native/blog/2017/03/13/better-list-views.html

Working with lists

Basics

```
const people = [
  <Text>Chris</Text>,
  <Text>Amanda</Text>,
  <Text>Jason</Text>,
  <Text>Jennifer</Text>
]

return (
  <View>
    { people }
  </View>
);
```

Working with lists

Map

```
render() {
  const people = ['Chris', 'Amanda', 'Jason', ‘Jennifer'];

  return (
    <ScrollView>
      { people.map(person => <Text>{person}</Text>) }
    </ScrollView>
  );
}
```

Working with lists

Map - key

```
render() {
  const people = ['Chris', 'Amanda', 'Jason', ‘Jennifer'];

  return (
    <ScrollView>
      { people.map((person, index) => <Text key={index}>{person}</Text>) }
    </ScrollView>
  );
}
```

Working with lists

Map - key - uuid

```
npm install uuid
```

```
const uuidV4 = require('uuid/v4');
render() {
  const people = ['Chris', 'Amanda', 'Jason', ‘Jennifer'];

  return (
    <ScrollView>
      { people.map(person => <Text key={uuidV4()}>{person}</Text>) }
    </ScrollView>
  );
}
```

Working with lists

Map - class method

```
renderPerson = (person, index) => {
  return (
    <View>
      <Text>{person}</Text>
    </View>
  )
}

render() {
  const people = ['Chris', 'Amanda', 'Jason', ‘Jennifer'];

  return (
    <ScrollView>
      { people.map(this.renderPerson) }
    </ScrollView>
  ;
}
```

Working with lists

Map - class method

```
renderPerson = (people) => {
  return people.map((person, index) => {
    return (
      <Text key={index}>{person}</Text>
    );
  });
}

render() {
  const people = ['Chris', 'Amanda', 'Jason', 'Jennifer'];
  return (
    <ScrollView>
      { this.renderPerson(people) }
    </ScrollView>
  );
}
```

Working with lists

FlatList

At a minimum needs a `data` and `renderItem` prop

```
render() {
  const data = [{ key: 'Chris' }, { key: 'Amanda' }];
  return (
    <View>
      <FlatList
        data={data}
        renderItem={({ item }) => <Text>{item.key}</Text>}
      />
    </View>
  );
}
```

Working with lists

FlatList

If there is no key in the data array, a keyExtractor function must be passed in as a prop.

```
render() {
  const data = [{ name: 'Chris' }, { name: 'Amanda' }];

  return (
    <View>
      <FlatList
        data={data}
        renderItem={({ item }) => <Text>{item.name}</Text>}
        keyExtractor={item => item.name}
      />
    </View>
  );
}
```

Working with lists

FlatList

`renderItem` can also be, and usually is, moved into a class method

```
renderItem = ({ item }) => {
  return <Text>{item.name}</Text>
}

render() {
  const data = [{ name: 'Chris' }, { name: 'Amanda' }];

  return (
    <View>
      <FlatList
        data={data}
        renderItem={this.renderItem}
        keyExtractor={item => item.name}
      />
    </View>
  );
}
```

Working with lists

FlatList

Also, data is usually either stored as state or received as props.

```
state = {
  data: [{ name: 'Chris' }, { name: 'Amanda' }],
}

renderItem = ({ item }) => {
  return <Text>{item.name}</Text>
}

render() {
  return (
    <View>
      <FlatList
        data={this.state.data}
        renderItem={this.renderItem}
        keyExtractor={item => item.name}
      />
    </View>
  );
}
```

Working with lists

FlatList

Item Separator

```
state = {
  data: [{ name: 'Chris' }, { name: 'Amanda' }],
}

renderItem = ({ item }) => {
  return <Text>{item.name}</Text>
}

render() {
  return (
    <View>
      <FlatList
        data={this.state.data}
        renderItem={this.renderItem}
        keyExtractor={item => item.name}
        ItemSeparatorComponent={() => <View style={styles.divider}
      />
    </View>
  );
}
```

Working with lists

FlatList

refreshing prop

```
state = {
  data: [{ name: 'Chris' }, { name: 'Amanda' }],
}

renderItem = ({ item }) => {
  return <Text>{item.name}</Text>
}
```

```
render() {
  return (
    <View>
      <FlatList
          data={this.state.data}
          renderItem={this.renderItem}
          keyExtractor={item => item.name}
          refreshing={this.state.refreshing}
          onRefresh={this.onRefresh}
        />
    </View>
  );
}
```

Working with lists

FlatList
Horizontal

```
state = {
  data: [{ name: 'Chris' }, { name: 'Amanda' }],
}

renderItem = ({ item }) => {
  return <Text>{item.name}</Text>
}
```

```
render() {
  return (
    <View>
      <FlatList
          data={this.state.data}
          renderItem={this.renderItem}
          keyExtractor={item => item.name}
          horizontal
      />
    </View>
  );
}
```

Working with lists

SectionList

At a minimum needs a `sections`, `renderSectionHeader`, and `renderItem` prop

```
const data = [
  {data: [{ name: 'Chris' }], key: 'Basketball'},
  {data: [{ name: 'Amanda' }], key: 'Baseball'},
  {data: [{ name: 'Jennifer' }, { name: 'Mike' }], key: 'Football'},
];

return (
  <View style={styles.container}>
    <SectionList
      renderItem={({item}) => <Text>{item.name}</Text>}
      renderSectionHeader={(section) => {
        return <Text style={styles.header}>{section.section.key}</Text>
      }}
      sections={data}
    />
  </View>
);
```

Working with lists

SectionList

Sticky Section Headers.
Enabled by default.

```
const data = [
  {data: [{ name: 'Chris' }], key: 'Basketball'},
  {data: [{ name: 'Amanda' }], key: 'Baseball'},
  {data: [{ name: 'Jennifer' }, { name: 'Mike' }], key: 'Football'},
];

return (
  <View style={styles.container}>
    <SectionList
      renderItem={({item}) => <Text>{item.name}</Text>}
      renderSectionHeader={(section) => {
        return <Text style={styles.header}>{section.section.key}</Text>
      }}
      sections={data}
      stickySectionHeadersEnabled={false}
    />
  </View>
);
```

Performance

What about doing this in native code?

Okay so what if we we implemented our scrolling list of images in native code? What kind of performance gains would we see?

Android RecyclerView Diff:
https://github.com/hgale/PerformanceExample/pull/4

Performance

What's are the differences between React Native's ListView and its native counterparts?

- The comparison between the ListView and the RecyclerView is not as dramatic as between the ListView and the ScrollView.
- This initial memory footprint and view hierarchy of the app look basically the same for both the React Native ListView implementation and the RecyclerView.
- One big difference is that the React Native ListView is implemented entirely in Javascript and does not depend on any native code. I.e you would need to provide a UICollectionVIew implementation as well as the RecyclerView one.
- Another big difference is that both UICollectionView for iOS and RecyclerView for Android use view recycling. This is as opposed to the React Native ListView which renders items lazily just when they are about to appear.
- The React Native ListView does not recycle the underlying view container like UICollectionView or RecyclerView. This results in more freeing and allocating of rows as the user scrolls, which is very CPU-intensive.

Article containing results:

https://launchdrawer.com/i-made-react-native-fast-you-can-too-9e61c951ce0

Native Code & Performance

Performance

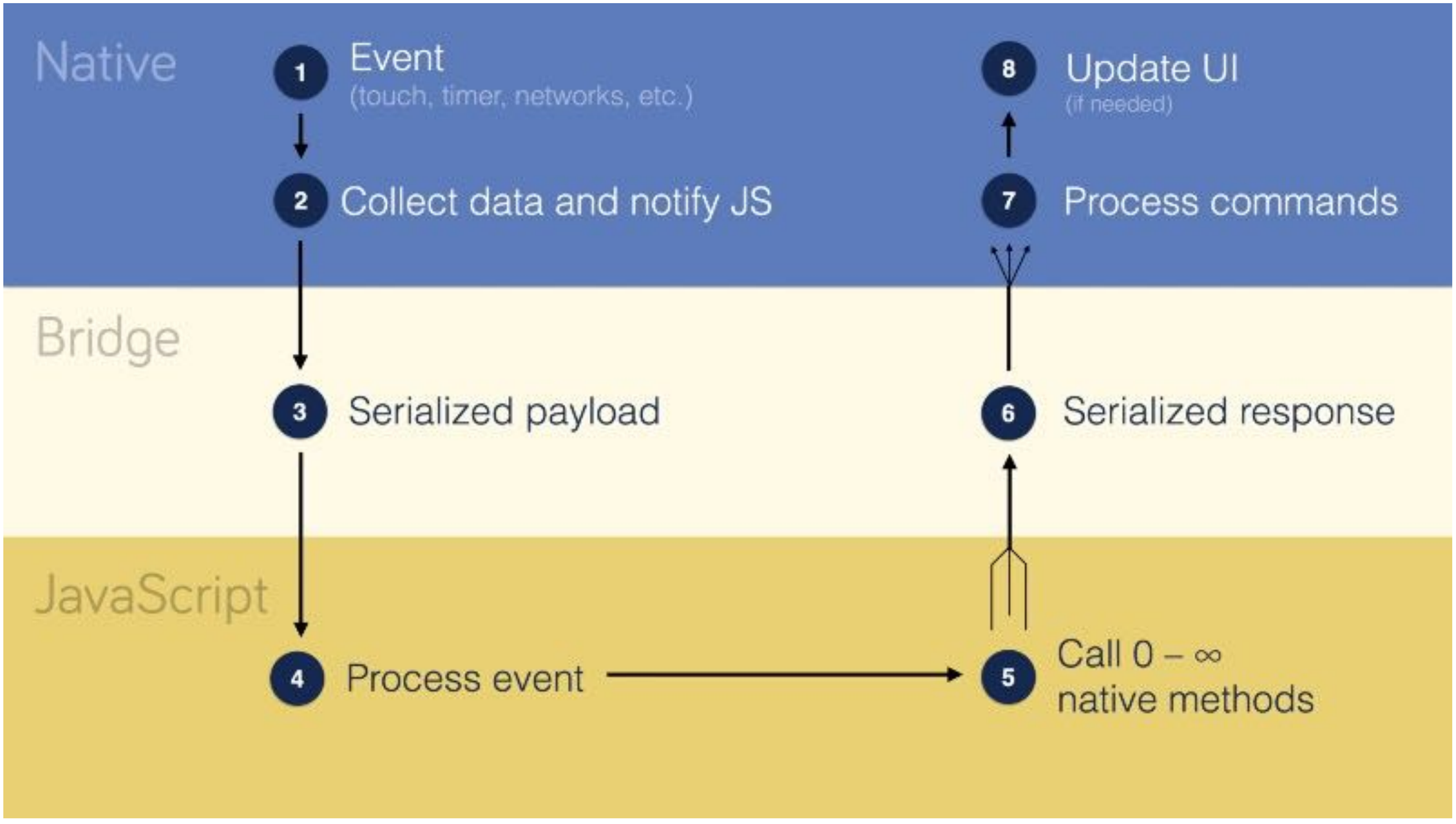
React Native under the hood

All React Native app's run in two different realms:

- The native realm — The kingdom of Objective-C/Swift in iOS and Java in Android. This is where we interface with the OS and where all Views are rendered. UI is manipulated exclusively on the main thread, but there can be others for background computation. React Native does most of the heavy lifting in this realm for us.
- The JS realm — Javascript is executed in its own separate thread by a Javascript-engine. Our business logic, including which Views to display and how to style them, is usually implemented here.

Variables defined in one realm cannot be directly accessed in the other. This means that all communication between the two realms must be done explicitly over a bridge.

Understanding the bridge


Native
1
Event
(touch, timer, networks, etc.)
2
Collect data and notify JS
8
Update UI
(if needed)
7
Process commands
Bridge
3
Serialized payload
6
Serialized response
JavaScript
4
Process event
5
Call 0 – ∞
native methods

Performance

Key Concept:

Performance bottleneck's in React Native often occur when we move from one realm to the other. In order to architect performant React Native apps, we must keep passes over the bridge from Javascript to Native to a minimum.

Performance Limitations of React Native and How to Overcome Them - Tal Kol:

Talk: https://www.youtube.com/watch?v=OmiXlJ4ZzAo
Code: https://github.com/wix/rn-perf-experiments2

Article: https://hackernoon.com/react-native-performance-an-updated-example-6516bfde9c5c

Performance

Example

Performance

Things to note:

- As the user is scrolling down the list of cards, the header image dissolves slowly into the gray background.
- The second — if the user scrolls up and is already at the top of the list, for the sake of continuity of movement the header image zooms slightly until it bounces back (this is called an overscroll effect).

The general component layout of the screen:

```
render() {
  return (
    <View style={styles.container}>
      <Image
        source={require('../img/bg.jpg')}
        style={[styles.backgroundImage, {
          opacity: this.state.imageOpacity,
          transform: [{
            scale: this.state.imageScale
          }]
        }]}
        />
        <ListView
          dataSource={this.state.dataSource}
          renderRow={this.renderRow.bind(this)}
          renderHeader={this.renderHeader.bind(this)}
          renderScrollComponent={this.renderScroll.bind(this)}
       />
     </View>
  );
}
```

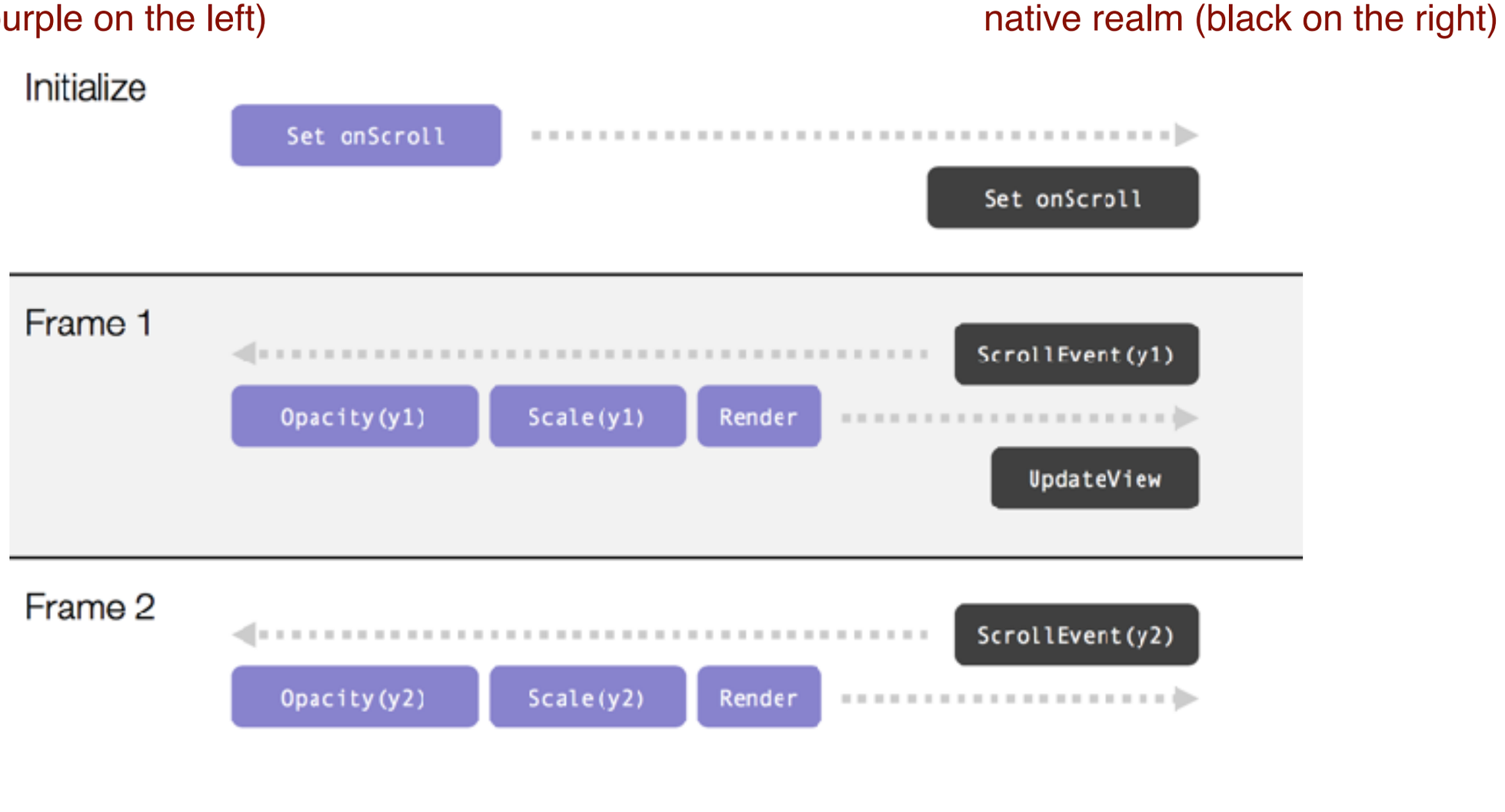
Scroll Event:

```
renderScroll(props) {
  return (
    <ScrollView
      {...props}
      scrollEventThrottle={16}
      onScroll={this.onScroll.bind(this)}
    />
  );
}

// the logic for onScroll applying our effects using setState
onScroll(event) {
  const scrollY = event.nativeEvent.contentOffset.y;
  if (scrollY >= 0) {
    let newOpacity = 1.0 - (scrollY / 250.0);
    if (newOpacity < 0) newOpacity = 0;
    this.setState({
      imageOpacity: newOpacity,
      imageScale: 1.0
    });
  } else {
    let newScale = 1.0 + 0.4*(-scrollY / 200.0);
    if (newScale > 1.4) newScale = 1.4;
    this.setState({
      imageOpacity: 1.0,
      imageScale: newScale
    });
  }
}
```

Performance

Why is this a problem?:

- Just like any other view event, scroll events originate in the native realm
- Our JavaScript logic runs in the JavaScript realm.
- Once we re-render, the new view properties must be applied to the actual native views back in the native realm.
- This means we're passing over the bridge twice for every frame.
- On busy apps, this will prevent us from running at 60 FPS. Let's improve.



How can we fix this?

Performance

A second implementation attempt — native scroll listener:

- Our performance bottleneck stems from implementing our scroll listener in JavaScript.
- Since scroll events originate in the native realm, executing our onScroll logic in JavaScript will always incur overhead.
- Let’s try moving the onScroll logic to native.

Our new layout:

```
render() {
  return (
    <View style={styles.container}>
      <NativeWrapper scrollViewHandle={this.state.scrollViewHandle}>
        <Image
          style={styles.backgroundImage}
          source={require('../img/bg.jpg')}
        />
      </NativeWrapper>
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderRow.bind(this)}
        renderHeader={this.renderHeader.bind(this)}
        renderScrollComponent={this.renderScroll.bind(this)}
      />
    </View>
  );
}


renderScroll(props) {
  return (
    <ScrollView
      {...props}
      ref={(element) => {
        const handle = ReactNative.findNodeHandle(element);
        this.setState({scrollViewHandle: handle});
      }}
    />
  );
}
```

Objective-C implementation:

```
- (void)setScrollViewHandle:(NSNumber *)scrollViewHandle
{
  _scrollViewHandle = scrollViewHandle;

  dispatch_async(RCTGetUIManagerQueue(), ^{
    [self.bridge.uiManager addUIBlock:^(__unused RCTUIManager *uiManager, NSDictionary<NSNumber *, UIView *>
*viewRegistry)
    {
      UIView *view = viewRegistry[scrollViewHandle];
      if ([view conformsToProtocol:@protocol(RCTScrollableProtocol)])
      {
        [view addScrollListener:self];
      }
   }];
  });
}

// the logic for onScroll applying our effects using native code
- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
  CGFloat scrollY = scrollView.contentOffset.y;
  if (scrollY >= 0)
  {
    CGFloat newOpacity = 1.0 - (scrollY / 250.0);
    if (newOpacity < 0) newOpacity = 0;
    self.alpha = newOpacity;
    self.transform = CGAffineTransformScale(CGAffineTransformIdentity, 1.0, 1.0);
  }
  else
  {
    CGFloat newScale = 1.0 + 0.4*(-scrollY / 200.0);
    if (newScale > 1.4) newScale = 1.4;
    self.alpha = 1.0;
    self.transform = CGAffineTransformScale(CGAffineTransformIdentity, newScale, newScale);
  }
}
```

Performance

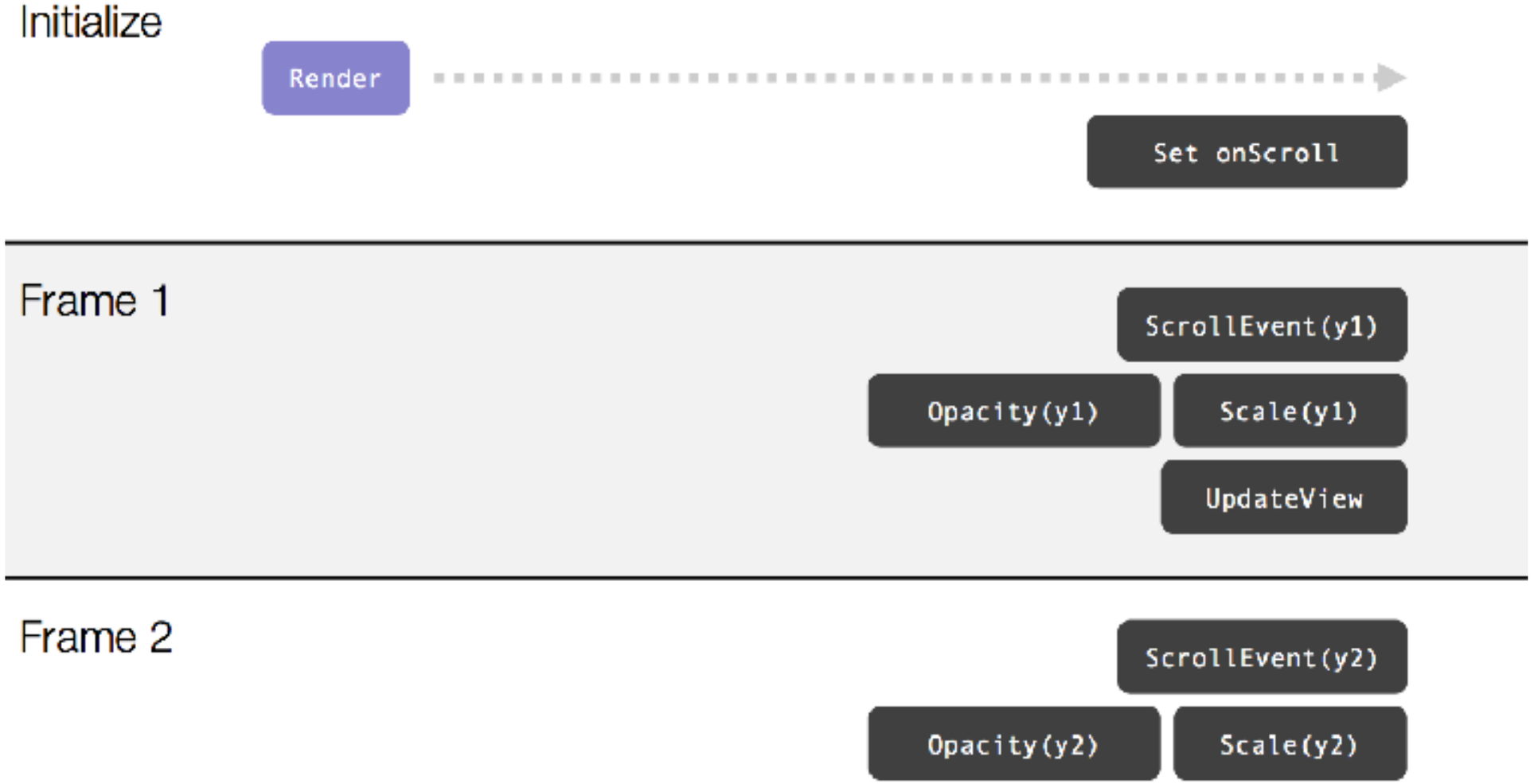
What changed?

- The code actually looks almost exactly the same.
- Instead of calling setState to update the view properties, we can simply change the view properties directly since we're running in the native realm.
- The only challenge here once again is hooking our native scroll listener. Once the numeric node handle of the ScrollView arrives via props, we need to translate it back to a view object reference in order to access the underlying ScrollView instance.

Performance

We expect performance to be much better since this implementation is tailored for reducing passes over the bridge.

Let’s count passes by analyzing what’s running in the JavaScript realm (purple on the left) and what’s running in the native realm (black on the right):



Performance

Whats wrong with this approach?

- This implementation may be fast, but it’s rather complex and requires native expertise.
- If this was a cross platform application we would now have to do the same thing for Android.

How can we fix this?

Use a declarative API

- A declarative API allows us to declare behaviors in advance in JavaScript and serialize the entire declaration and send it once over the bridge during initialization.
- general purpose native driver — one that you don't need to write yourself — will execute the behavior in the native realm according to the declared specification.
- The behavior we want consists of two parts:
  - Listening to scroll position changes.
  - Updating view properties (opacity and scale).
- In order to update view properties we can use Animated, the animation library that's part of React Native core.
- Animated can drive an Animated.Value based on scroll events as well.

Layout Implementation:

```
// initialize an Animated.Value that we'll work with
constructor(props) {
  super(props);
  this.state = {
    scrollY: new Animated.Value(0)
  };
}

// the general layout of the entire screen
// contains the interpolation of opacity and scale from the Animated.Value
render() {
  return (
    <View style={styles.container}>
      <Animated.Image
        style={[styles.backgroundImage, {
          opacity: this.state.scrollY.interpolate({
            inputRange: [0, 250],
            outputRange: [1, 0]
          }),
          transform: [{
            scale: this.state.scrollY.interpolate({
              inputRange: [-200, 0, 1],
              outputRange: [1.4, 1, 1]
            })
          }]
        }]}
        source={require('../img/bg.jpg')}
      />
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderRow.bind(this)}
        renderHeader={this.renderHeader.bind(this)}
        renderScrollComponent={this.renderScroll.bind(this)}
      />
    </View>
  );
}
```

Handling Scroll Event:

```
renderScroll(props) {
  return (
    <Animated.ScrollView
      {...props}
      scrollEventThrottle={16}
      onScroll={
        Animated.event([{
          nativeEvent: { contentOffset: { y: this.state.scrollY } }
        }], {
          useNativeDriver: true
        })
      }
    />
  );
}
```

Performance

Whats happening?

- Animated.Value is used to hold the scroll position at any given time.
- ListView is provided with a custom ScrollView component, Animated.ScrollView.
- Animated.ScrollView lets us declare that our Animated.Value is driven by the contentOffset property of the native onScroll event.
- From this point forward, we can use the standard Animated approach of interpolating view properties based on an Animated.Value.
- This requires changing our header image to Animated.Image and allows us to interpolate both the opacity and transform.scale from scroll position.
- Note that the entire implementation is declarative. We no longer have an imperative onScroll function that performs calculations for our effects.
- Also note that we've specified useNativeDriver. The implementation of the Animated library in recent versions of React Native finally contains a native driver that can execute the entire declaration from the native realm without using the bridge.

Performance

Example

Performance

Things to note:

- https://facebook.github.io/react-native/docs/animations.html#using-the-native-driver
- https://facebook.github.io/react-native/docs/animated.html

“By using the native driver, we send everything about the animation to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame. Once the animation has started, the JS thread can be blocked without affecting the animation.

You can use the native driver by specifying useNativeDriver: true in your animation configuration.”

Using the native driver for normal animations is quite simple. Just add `useNativeDriver: true` to the animation config when starting it.

```
Animated.timing(this.state.animatedValue, {
  toValue: 1,
  duration: 500,
  useNativeDriver: true, // <-- Add this
}).start();
```