Optimizing Flatlist Configuration

Terms

- **VirtualizedList:** The component behind `FlatList` (React Native's implementation of the `Virtual List` concept.)
- **Memory consumption:** How much information about your list is being stored in memory, which could lead to an app crash.
- **Responsiveness:** Application ability to respond to interactions. Low responsiveness, for instance, is when you touch on a component and it waits a bit to respond, instead of responding immediately as expected.
- **Blank areas:** When `VirtualizedList` can't render your items fast enough, you may enter a part of your list with non-rendered components that appear as blank space.
- **Viewport:** The visible area of content that is rendered to pixels.
- **Window:** The area in which items should be mounted, which is generally much larger than the viewport.

Props

Here are a list of props that can help to improve `FlatList` performance:

removeClippedSubviews

TYPE	DEFAULT
Boolean	False

If `true`, views that are outside of the viewport are detached from the native view hierarchy.

Pros: This reduces time spent on the main thread, and thus reduces the risk of dropped frames, by excluding views outside of the viewport from the native rendering and drawing traversals.

Cons: Be aware that this implementation can have bugs, such as missing content (mainly observed on iOS), especially if you are doing complex things with transforms and/or absolute positioning. Also note this does not save significant memory because the views are not deallocated, only detached.

maxToRenderPerBatch

TYPE	DEFAULT
Number	10

It is a `VirtualizedList` prop that can be passed through `FlatList`. This controls the amount of items rendered per batch, which is the next chunk of items rendered on every scroll.

Pros: Setting a bigger number means less visual blank areas when scrolling (increases the fill rate).

Cons: More items per batch means longer periods of JavaScript execution potentially blocking other event processing, like presses, hurting responsiveness.

updateCellsBatchingPeriod

TYPE	DEFAULT
Number	50

While `maxToRenderPerBatch` tells the amount of items rendered per batch, setting `updateCellsBatchingPeriod` tells your `VirtualizedList` the delay in milliseconds between batch renders (how frequently your component will be rendering the windowed items).

Pros: Combining this prop with `maxToRenderPerBatch` gives you the power to, for example, render more items in a less frequent batch, or less items in a more frequent batch.

Cons: Less frequent batches may cause blank areas, More frequent batches may cause responsiveness issues.

initialNumToRender

TYPE	DEFAULT
Number	10

The initial amount of items to render.

Pros: Define precise number of items that would cover the screen for every device. This can be a big performance boost for the initial render.

Cons: Setting a low `initialNumToRender` may cause blank areas, especially if it's too small to cover the viewport on initial render.

windowSize

TYPE	DEFAULT
Number	21

The number passed here is a measurement unit where 1 is equivalent to your viewport height. The default value is 21 (10 viewports above, 10 below, and one in between).

Pros: Bigger `windowSize` will result in less chance of seeing blank space while scrolling. On the other hand, smaller `windowSize` will result in fewer items mounted simultaneously, saving memory.

Cons: For a bigger `windowSize`, you will have more memory consumption. For a lower `windowSize`, you will have a bigger chance of seeing blank areas.

List items

Below are some tips about list item components. They are the core of your list, so they need to be fast.

Use basic components

The more complex your components are, the slower they will render. Try to avoid a lot of logic and nesting in your list items. If you are reusing this list item component a lot in your app, create a component only for your big lists and make them with as little logic and nesting as possible.

Use light components

The heavier your components are, the slower they render. Avoid heavy images (use a cropped version or thumbnail for list items, as small as possible). Talk to your design team, use as little effects and interactions and information as possible in your list. Show them in your item's detail.

Use shouldComponentUpdate

Implement update verification to your components. React's `PureComponent` implement a `shouldComponentUpdate` with shallow comparison. This is expensive here because it needs to check all your props. If you want a good bit-level performance, create the strictest rules for your list item components, checking only props that could potentially change. If your list is basic enough, you could even use

```
shouldComponentUpdate() {
  return false
}
```

Use cached optimized images

You can use the community packages (such as react-native-fast-image from @DylanVann) for more performant images. Every image in your list is a `new Image()` instance. The faster it reaches the `loaded` hook, the faster your JavaScript thread will be free again.

Use getItemLayout

If all your list item components have the same height (or width, for a horizontal list), providing the getItemLayout prop removes the need for your `FlatList` to manage async layout calculations. This is a very desirable optimization technique.

If your components have dynamic size and you really need performance, consider asking your design team if they may think of a redesign in order to perform better.

Use keyExtractor or key

You can set the `keyExtractor` to your `FlatList` component. This prop is used for caching and as the React `key` to track item re-ordering.

You can also use a `key` prop in your item component.

Avoid anonymous function on renderItem

For functional components, move the `renderItem` function outside of the returned JSX. Also, ensure that it is wrapped in a `useCallback` hook to prevent it from being recreated each render.

For class componenents, move the `renderItem` function outside of the render function, so it won't recreate itself each time the render function is called.

```
const renderItem = useCallback(({item}) => (
   <View key={item.key}>
      <Text>{item.title}</Text>
   </View>
 ), []);

return (
  // ...

  <FlatList data={items} renderItem={renderItem} />;
  // ...
);
```

Is this page useful?

Edit this page

Last updated on ***Jun 21, 2023***