Ta Quang Tung – 104222196

COS30017 – Software Development for Mobile Devices – Extension: Performance

This report presents the results of using Android Studio's Profiler to measure the CPU and memory usage of an application which displays a long list of strings with an icon. Two variables are present in the experiment: the number of list items and how the icon is created in the application. The results show noticeable CPU and memory differences between pre-generating icons versus generating them on the fly and between using decoded icons versus generated icons.

Setup

Hardware: Personal Android Phone running on Android 12 with 4GB of RAM and an octo-core 2.40GHz CPU.

Application: The application being experimented on features a RecyclerView which is used to render a long list of unit codes with an icon. The app has a floating action button which inserts one item at the end of the list and scrolls to that new item when pressed. Each row in the list consists of one ImageView for the icon and one TextView for the unit code. The icon is generated with one of three strategies: pre-generated with each row item, generated on the fly from scratch as the user scrolls the list, and decoded on the fly from a drawable resource as the user scrolls the list. The experiment will switch between these strategies and also change the total number of items in the list.

Test cases and methodology

This experiment runs 6 test cases, each of which generates 500 or 2000 list items with one of the three icon generation strategies. For each test case, the Profiler will be run in Low Overhead mode. After the app is started, the floating action button will be pressed, creating a new list item and scrolling to the bottom of the list. This effectively goes over all list items and causes the ImageViews to update based on one of the three strategies. Inside the Profiler, statistics of interest are:

- The CPU usage time (in seconds) from when the button is pressed and when the list is scrolled to the bottom and the app settles.
- The average CPU usage (in percentage) during this time.
- The memory usage (in MB) before the button is pressed.
- The memory usage (in MB) from when the button is pressed and when the list is scrolled to the bottom and the app settles.
- The memory usage (in MB) after the app settles.

Results

The following table shows the results recorded by the Profiler after running 6 tests:

Test case	CPU usage time	Average CPU usage	Memory before press	Memory after press, before settle	Memory after settle
500 items, icon pre-generated	~6 seconds	~12%	~158.8MB	Steady climb to ~232MB	~232MB
500 items, icons	~6 seconds	~12%	~83.2MB	Spikes up to ~142.7MB	~132MB

generated on-the-fly					
500 items, decoded on-the-fly	~8 seconds	~13%	~80.2MB	Spikes up to ~183MB	~156MB
2000 items, icons pre-generated	~22 seconds	~12%	~389MB	Steady climb to ~462MB	~462MB
2000 items, icons generated on-the-fly	~22 seconds	~12%	~82MB	Spikes up to ~151MB	~124MB
2000 items, icons decoded on-the-fly	~30 seconds	~13%	~80MB	Spikes up to ~182MB	~170.3MB

Findings

All 6 test cases share roughly the same average CPU consumption of around 12%. The most noticeable difference is the CPU use time of the *2000 items – icons decoded on-the-fly* test case, which was 8 seconds higher than the other test cases with the same item count. Interestingly, generating the icons from scratch was faster than decoding them from the drawable. The discrepancy could be because the drawable had to be opened and copied into a Bitmap repeatedly, which was more computationally expensive that generating the Bitmap from scratch.

In terms of memory consumption, pre-generating the icons measured the worst performance out of all. With 2000 items, this strategy used up to 462MB of RAM. Meanwhile, the other strategies used only 124MB and 170.3MB respectively. This difference is because when the icons were pre-generated, all 2000 of them had to be kept in memory, whereas when the icons were created on the fly, only the visible ones were stored. Generating Bitmaps from scratch proved to be the most memory-efficient strategy out of the three. Strangely enough, generating 2000 Bitmaps used slightly less memory than generating 500 of them, although this could be due to the state of the device at the time of measurement. Decoding Bitmaps from the drawable consumed slightly more memory than generating them from scratch, but this difference pales in comparison with pre-generation.

Conclusion

The results of this experiment show that it is inefficient to pre-generate content for long lists, especially resource-intensive content such as images. Creating them on the fly is more efficient as only the content visible to the user is kept in memory. This is important to bear in mind as for long lists, it is very unlikely that the user will scroll through every item. Storing unviewed items is a waste of resources and can slow down the device.

The results also show a performance difference between generating Bitmap images from scratch versus decoding them from a file. Decoding from files uses more memory and CPU time because the resource has to be converted into an input stream and written to a Bitmap. That said, this method may be more suitable for complex images which cannot be drawn from scratch.

Reflection on Concurrency

When developing for Android, it is important not to run any time-consuming tasks on the main thread as this can block the UI from updating quickly, causing an Application Not Responding error. Concurrency can greatly increase the performance of an application by deferring time-consuming tasks such as data fetching or processing to a different thread, allowing the main thread to respond to user actions and update the views accordingly.