Huy Vu Tran - 104177995

Assignment: Extension on Performance

Introduction

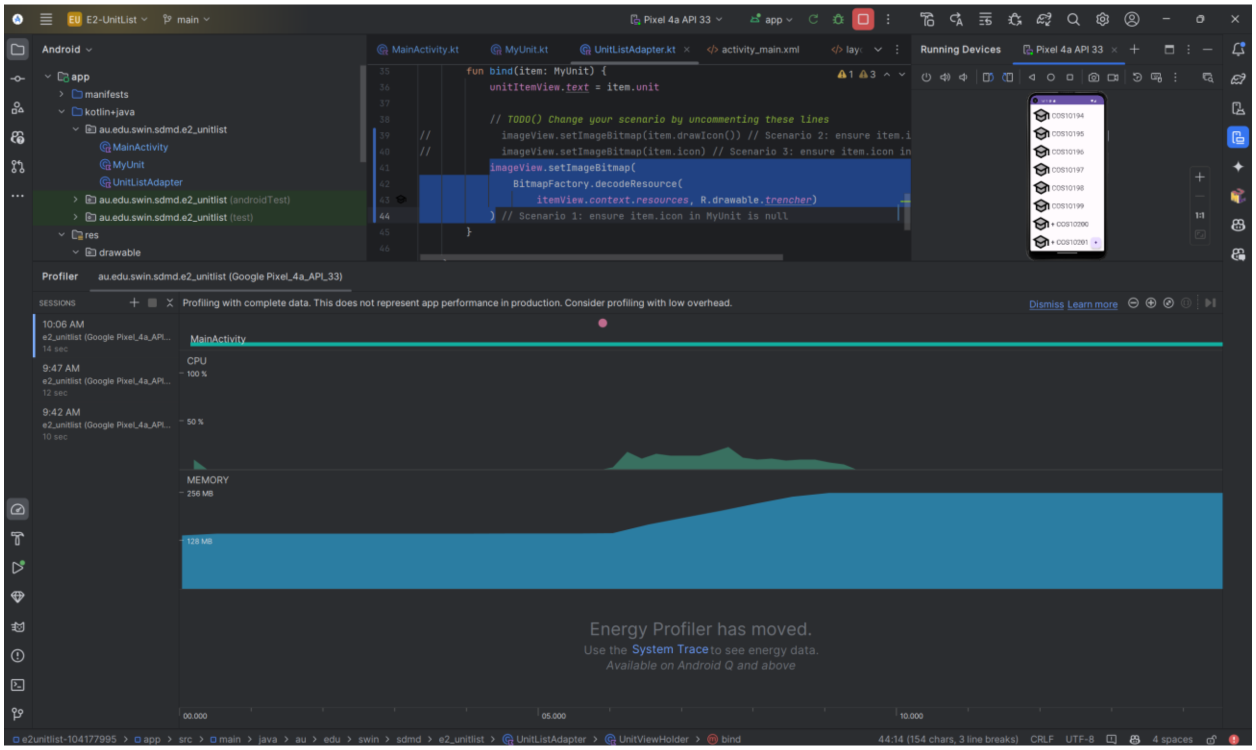
Optimizing the performance of mobile applications is crucial in today's fast-paced digital landscape. Users expect seamless and responsive experiences, and any performance bottlenecks can lead to frustration and dissatisfaction. In this task, we will explore the importance of performance optimization by examining an Android application that displays a list of strings with icons.

The app in question allows users to add new items to the list, which can have a significant impact on the overall performance of the application, especially when dealing with a large number of items. By utilizing the Profiler tool, we will investigate the performance implications of different approaches to handling the list and the associated icons.

Experiment

Scenario 1: A constant icon

For the first scenario, it could be illustrated that the CPU went up to 11% when we first opened the app for the first time. At this state, the program has to automatically load all the rows in the recycler view. This makes the program consume more CPU, and memory. After the 1st click to the add button, both the CPU and memory increased gradually. For the CPU, it reached its peak at 24%, before decreasing eventually, whereas the memory rose from 75MB to over 128MB after the click and stayed unchanged until ending the session.



```
fun bind(item: MyUnit) {
    unitItemView.text = item.unit

    // TODO() Change your scenario by uncommenting these lines
//      imageView.setImageBitmap(item.drawIcon()) // Scenario 2: ensure item.i
//      imageView.setImageBitmap(item.icon) // Scenario 3: ensure item.icon in
    imageView.setImageBitmap(
        BitmapFactory.decodeResource(
            itemView.context.resources, R.drawable.trencher)
    ) // Scenario 1: ensure item.icon in MyUnit is null
}
```

Profiling with complete data. This does not represent app performance in production. Consider profiling with low overhead.

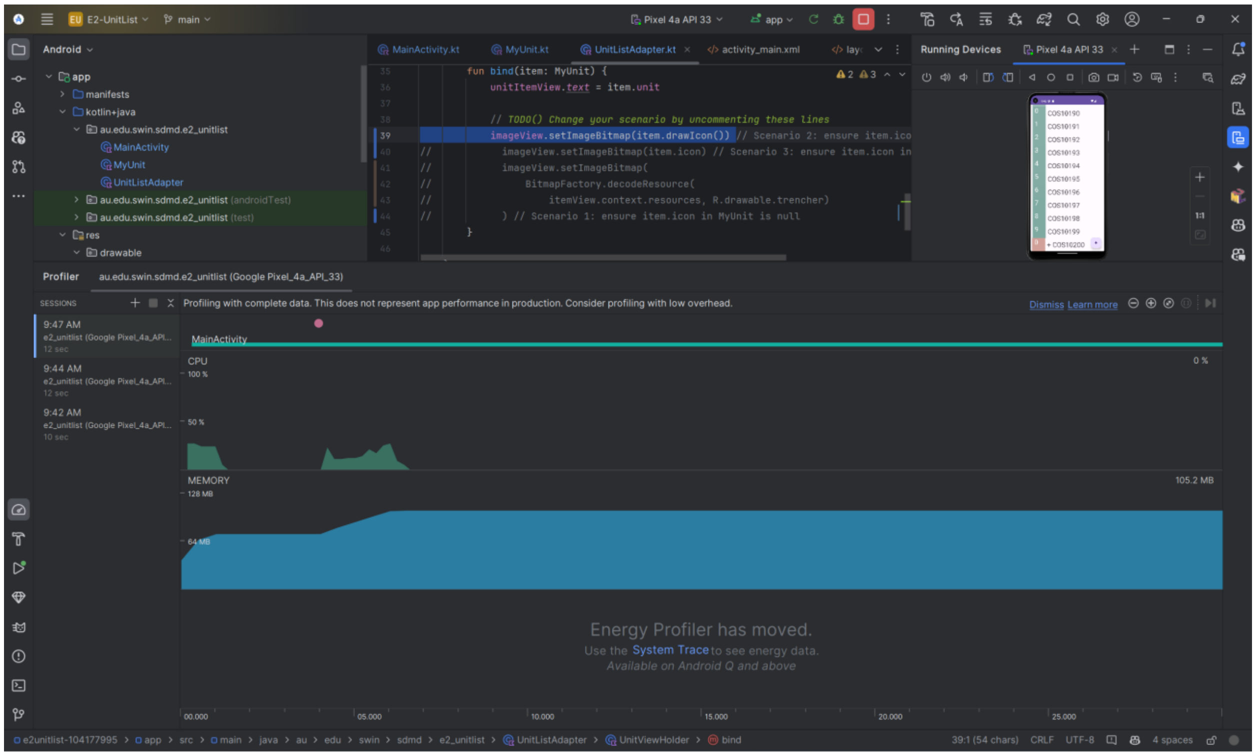
Energy Profiler has moved.
Use the System Trace to see energy data.
Available on Android Q and above

E2-UnitList
main
Pixel 4a API 33
app
Android
app
manifests
kotlin+java
au.edu.swin.sdmd.e2_unitlist
MainActivity
MyUnit
UnitListAdapter
au.edu.swin.sdmd.e2_unitlist (androidTest)
au.edu.swin.sdmd.e2_unitlist (test)
res
drawable
MainActivity.kt
MyUnit.kt
UnitListAdapter.kt
activity_main.xml
Running Devices
Pixel 4a API 33
Profiler
au.edu.swin.sdmd.e2_unitlist (Google Pixel_4a_API_33)
SESSIONS
10:06 AM
e2_unitlist (Google Pixel_4a_API...
14 sec
9:47 AM
e2_unitlist (Google Pixel_4a_API...
12 sec
9:42 AM
e2_unitlist (Google Pixel_4a_API...
10 sec
Dismiss
Learn more
MainActivity
CPU
100 %
50 %
MEMORY
256 MB
128 MB
00.000
05.000
10.000
e2unitlist-104177995 > app > src > main > java > au > edu > swin > sdmd > e2_unitlist > UnitListAdapter > UnitViewHolder > bind
44:14 (154 chars, 3 line breaks)
CRLF
UTF-8
4 spaces

Figure 1: Open the app and click the add button in the first scenario

Scenario 2: A generated icon created on bind

For the second scenario, we can see that the CPU rocketed to 28% when we first opened the app for the first time. However, unlike the first scenario when the memory quite stablized when initializing the app, in the second scenario, the memory index climbed significantly from 40MB to 74MB. This makes the program consume more CPU, and memory. After the 1st click to the add button, the CPU result fluctuated, and reached the highest point at 28%. In the same way, there was a moderate escalation in the memory index to 105MB and keeping it states before the program ends.



```
    fun bind(item: MyUnit) {
        unitItemView.text = item.unit

        // TODO() Change your scenario by uncommenting these lines
        imageView.setImageBitmap(item.drawIcon()) // Scenario 2: ensure item.ico
//        imageView.setImageBitmap(item.icon) // Scenario 3: ensure item.icon in
//        imageView.setImageBitmap(
//            BitmapFactory.decodeResource(
//                itemView.context.resources, R.drawable.trencher)
//        ) // Scenario 1: ensure item.icon in MyUnit is null
    }
```

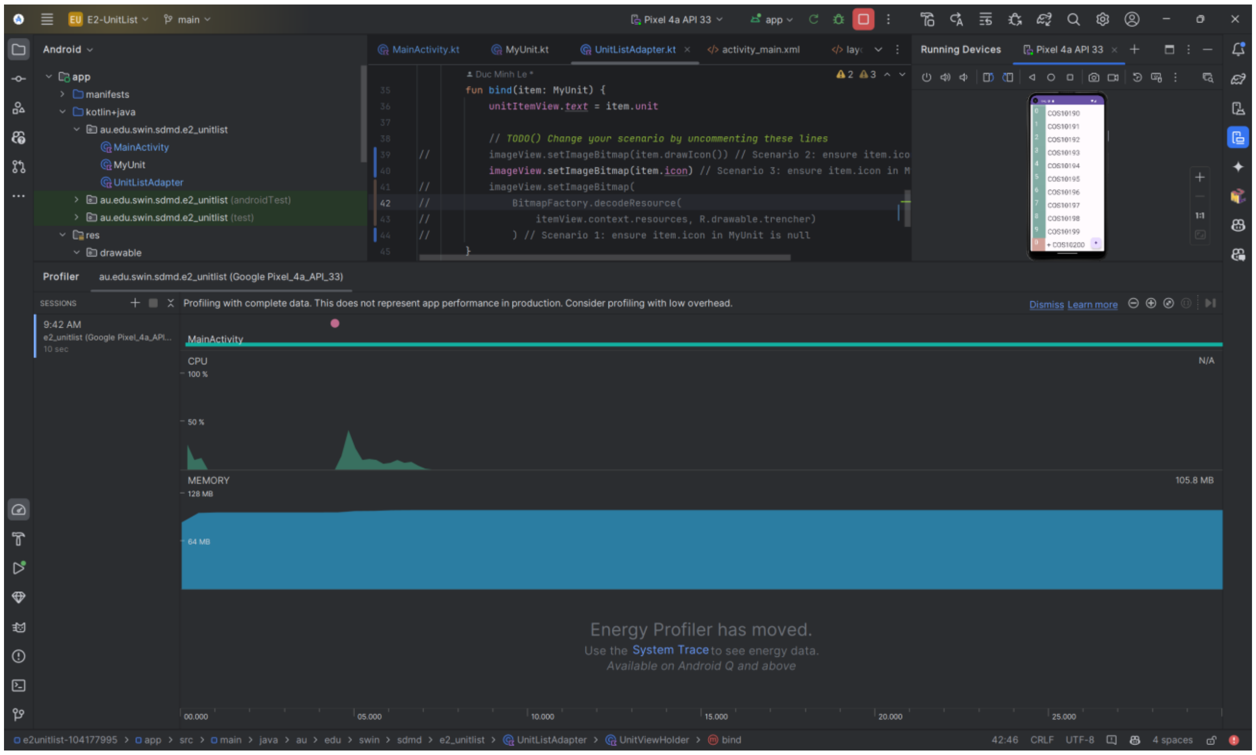
Profiling with complete data. This does not represent app performance in production. Consider profiling with low overhead.

Energy Profiler has moved.
Use the System Trace to see energy data.
Available on Android Q and above

Figure 2: Open the app and click the add button in the second scenario

Scenario 3: A generated icon created on initialisation

For the third scenario, it can be highlighted that the CPU spiked up to 27% when we first opened the app for the first time. Likewise, the memory index also had a slightly upward trend by over 10%. After the 1st click to the add button, the CPU soared up to 42% which was the higest CPU point among 3 scenarios, before it declined remarkably. In the same way, there was a moderate escalation in the memory index to 105MB and keeping it states before the program ends. Nevertheless, the index of memory in the third scenario stayed unchanged after the application running.



```
fun bind(item: MyUnit) {
    unitItemView.text = item.unit

    // TODO() Change your scenario by uncommenting these lines
//  imageView.setImageBitmap(item.drawIcon()) // Scenario 2: ensure item.ico
    imageView.setImageBitmap(item.icon) // Scenario 3: ensure item.icon in M
//  imageView.setImageBitmap(
//      BitmapFactory.decodeResource(
//          itemView.context.resources, R.drawable.trencher)
//      ) // Scenario 1: ensure item.icon in MyUnit is null
}
```

Profiling with complete data. This does not represent app performance in production. Consider profiling with low overhead.

Energy Profiler has moved.
Use the System Trace to see energy data.
Available on Android Q and above

Figure 3: Open the app and click the add button in the third scenario

Evaluation

Best Performing Scenario

Among the three scenarios tested, the first scenario, which involved using a constant icon, performed the best. This scenario showed the lowest CPU and memory usage overall. Initially, the CPU usage went up to 11% when the app was opened and peaked at 24% after the first click of the add button before gradually decreasing. The memory usage also rose from 75MB to just over 128MB and then stabilized. The lower resource consumption in this scenario can be attributed to the constant nature of the icon, which does not require additional processing or memory allocation beyond the initial load. This efficiency in handling resources makes the constant icon approach the most performant among the scenarios tested.

Worst Performing Scenario

The third scenario, where a generated icon is created on initialization, performed the worst. This scenario saw the highest CPU spike, reaching up to 27% upon opening the app and soaring to 42% after the first click of the add button. This was the highest CPU usage observed among the scenarios. The memory usage also showed a significant increase, starting at a higher baseline compared to the other scenarios and continuing to rise moderately. The high CPU and memory consumption in this scenario can be attributed to the overhead of generating icons during initialization, which places a substantial load on the system at startup and during list operations.

Takeaways

This task has highlighted the importance of optimizing list and image handling in mobile applications. The experiments demonstrate that using preloaded or constant icons can significantly reduce CPU and memory usage, enhancing the app's overall performance. Conversely, dynamically generating icons or performing heavy computations during initialization can lead to high resource consumption and potentially degrade user experience. Efficient memory management and minimizing unnecessary processing are crucial for maintaining optimal app performance, especially when dealing with large lists and images.

Performance Improvement through Concurrency

Performance could be significantly improved by using concurrency, particularly when loading a large number of items into a list from a file. By offloading the item loading process to a background thread, the main UI thread can remain responsive, preventing the app from becoming sluggish or unresponsive. Implementing asynchronous loading techniques, such as using AsyncTask or Kotlin Coroutines, can help manage resource-intensive operations without blocking the main thread. Additionally, leveraging pagination or lazy loading can further optimize performance by only loading visible items initially and fetching additional data as needed. This approach not only enhances responsiveness but also improves the user experience by ensuring smooth and efficient handling of large datasets.

Conclusion

This performance analysis of the Android application has provided valuable insights into the importance of optimization when dealing with lists and images. The three scenarios explored highlight the profound impact that design choices can have on the overall performance and user experience of the app.

Furthermore, these analysis underscore the need for careful consideration of resource management strategies when developing mobile applications. Techniques such as asynchronous loading, pagination, and lazy loading can significantly improve performance by distributing workloads and reducing the strain on the main UI thread.

Ultimately, this task has demonstrated the critical role of performance optimization in delivering seamless and responsive user experiences. By understanding the trade-offs and best practices revealed in this analysis, developers can make informed decisions to enhance the overall quality and competitiveness of their mobile applications.