Trac Duc Anh Luong - 103488117

Extension (add-on) on Performance

Introduction

When it comes to mobile software development, many aspects need to be taken into consideration to develop and maintain an application. In addition to the beautiful and responsive UI design, we also need to think about optimisation and how to improve the performance of the software as mobile devices have constraints on resources that we need to work with. A beautifully designed application that takes forever to load and slows down the host machine will not attract any user to use the app. Therefore, in this report, we will take a deeper look into the given app on its performance under various scenarios. After that, we will analyze the pros and cons of each instance and give our conclusion.

Inspection and Analysis

Scenario 1: Dynamically draw the icon each time

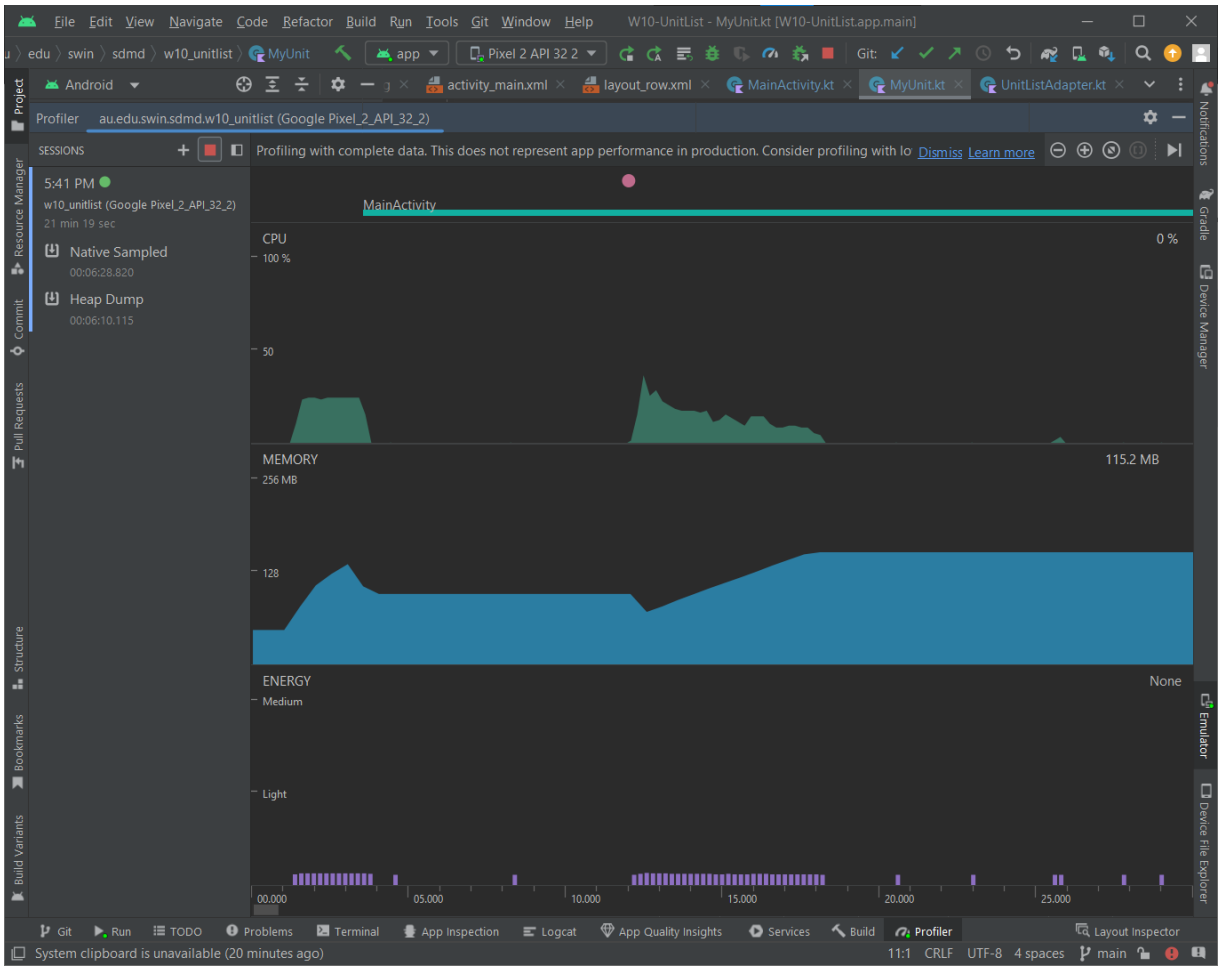

Figure 1: Open the app and click "+" the first time

For the first scenario, we can see that the memory and CPU spiked up when we first opened the app and pressed the “add” floating action button for the first time. At this state, the program has to automatically load all the rows in the recycler view and scroll down to the last row (COS100500) and display our newly added row (COS1000501). This makes the program consume more CPU, memory, and device energy. The CPU peaked at first and then decreased, while the memory increased to 200 and stayed the same as a straight line.

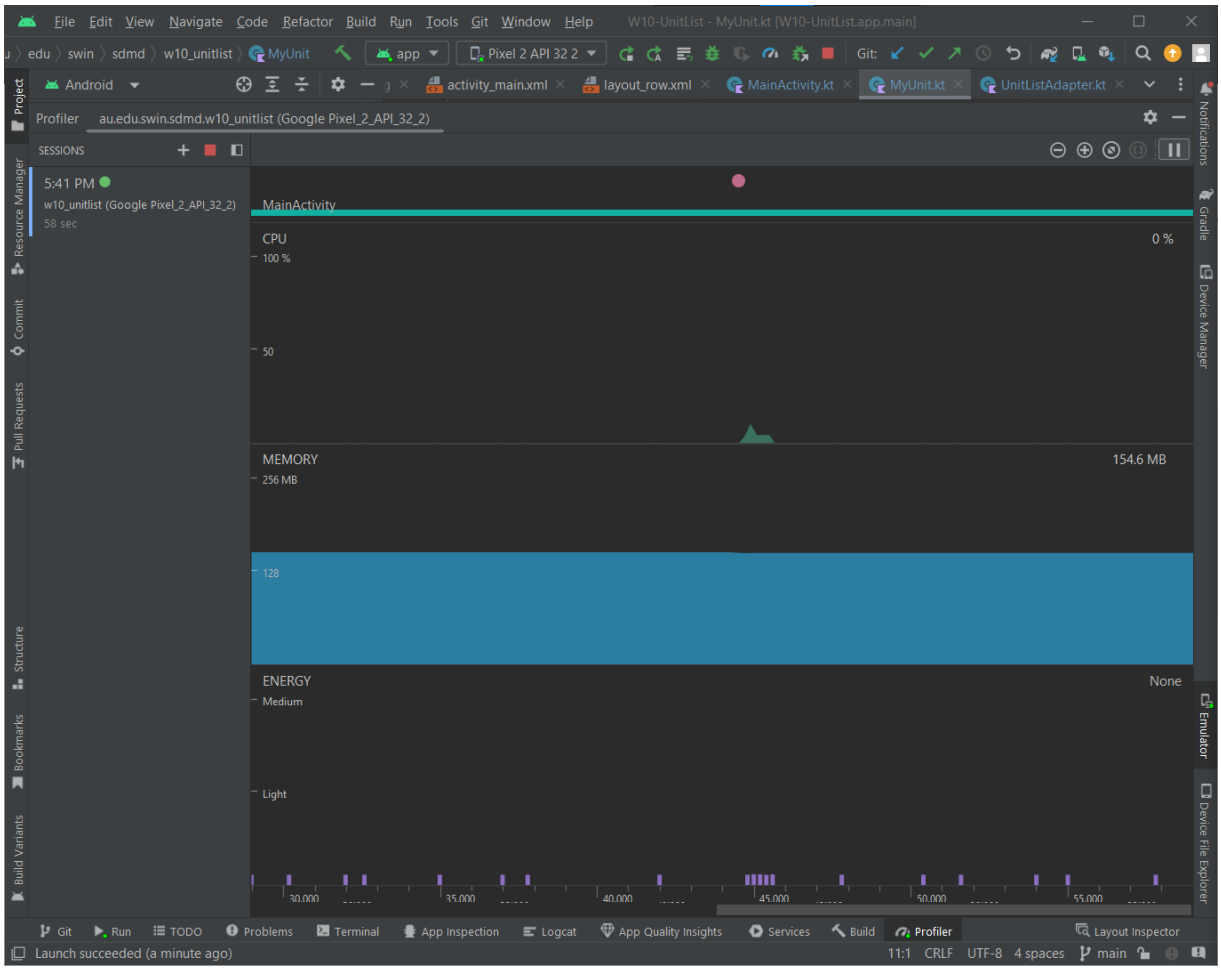

Figure 2: Click “+” the second time

After the 2nd click, the program was then stabled and the CPU only had a small spike at around 4-6% while the memory increased by 0.2MB. This indicates that the phone has loaded all of the pre-requisite resources.

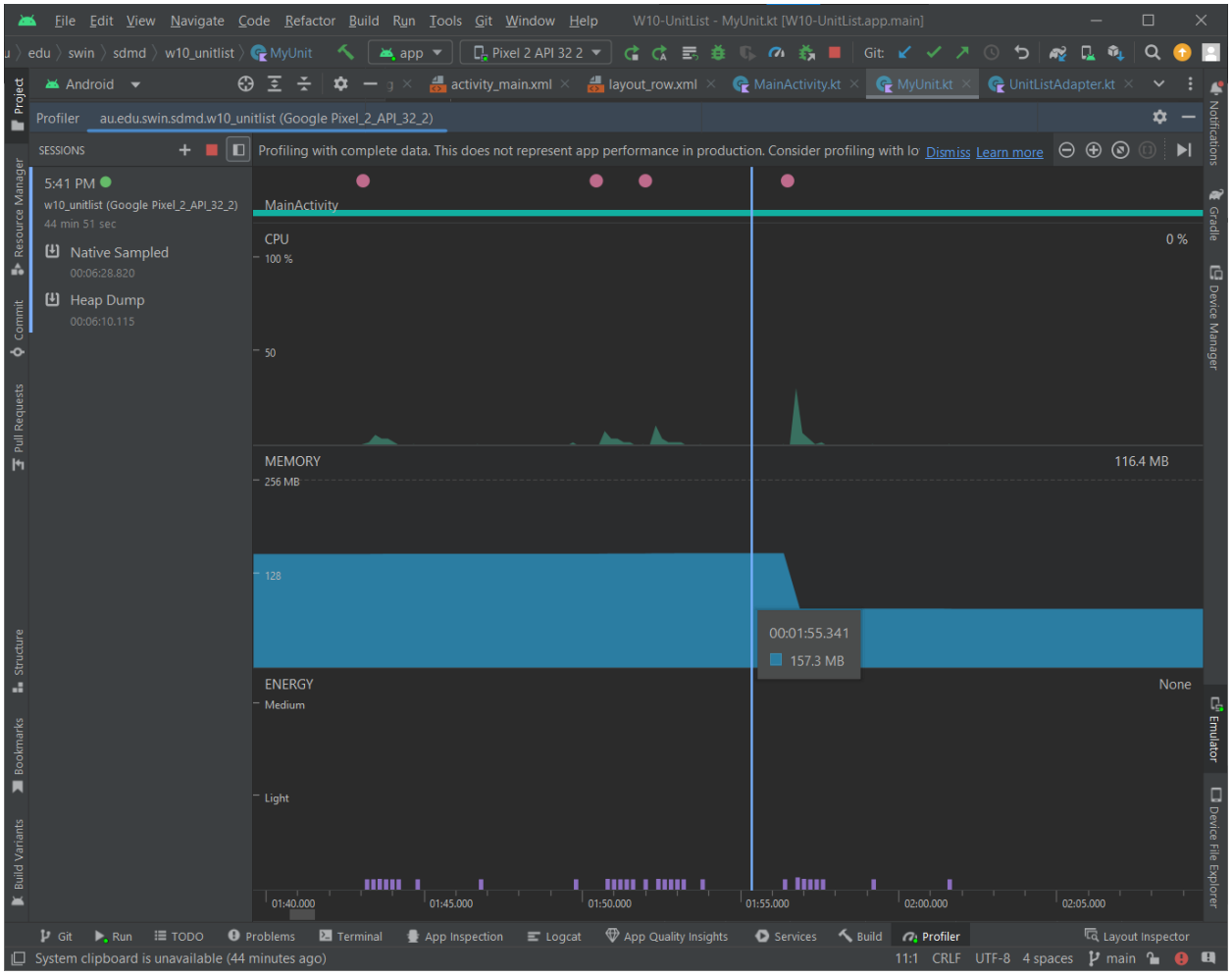

Figure 3: Click the "+" button for the 10th time

After the 10th click, the memory is then released and reduced from 157.3 MB to 81.9 MB. The number around 160 MB is the threshold that signals the memory to be released and returns to the initial memory allocation. The CPU also spiked to 21% instead of the usual 6-10% increase. This indicates the heap dump and how the memory refreshes itself.

Scenario 2: Statically draw the icon

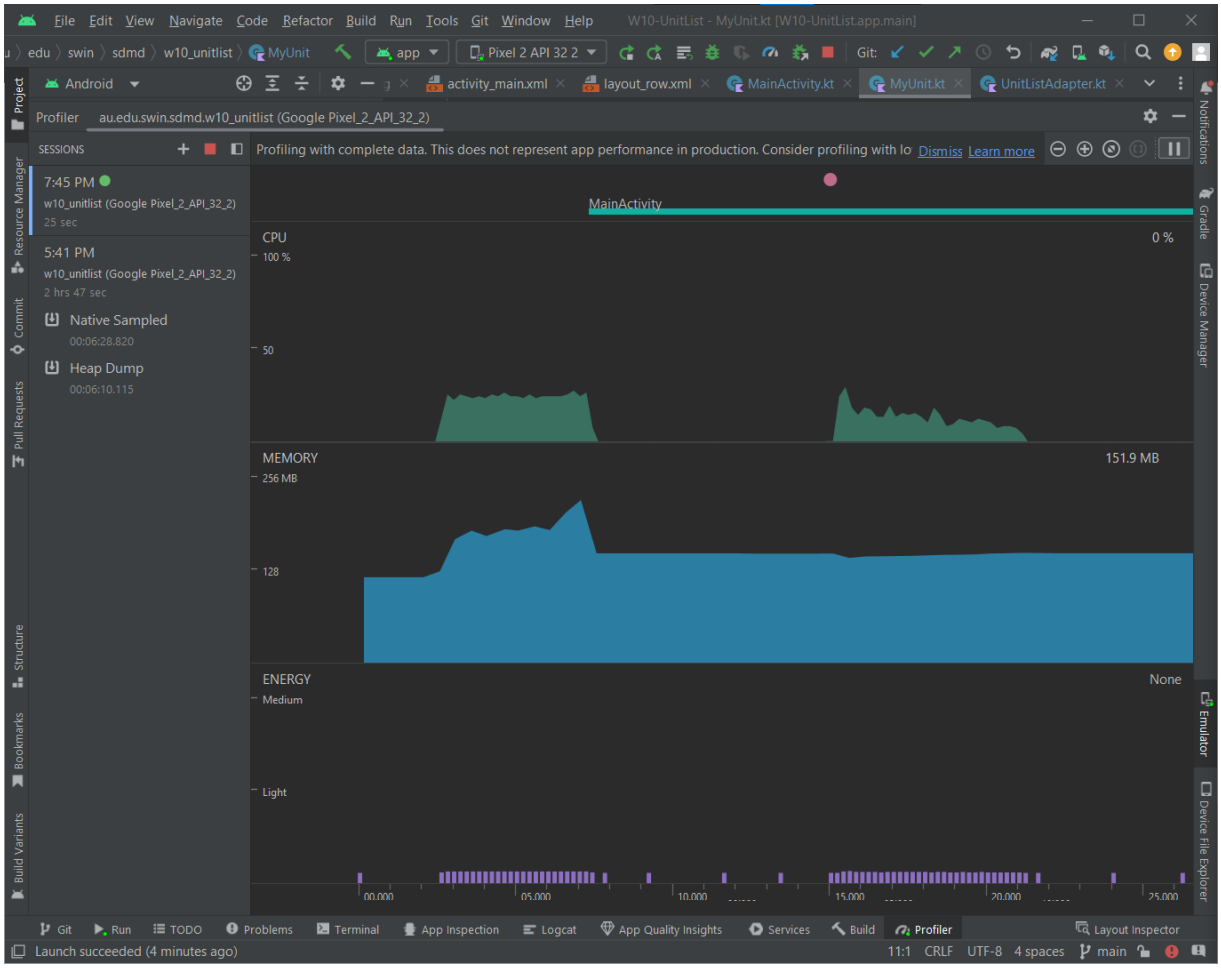

Figure 4: Click the + button after initialization

When you click the add button after the program first started, the CPU spiked at 30% with 152 MB of RAM allocated to the program. We can see that even though the CPU usage increased, the RAM did not increase and even dropped by around 5 MB. This again was due to the program having to create 500 items in the recycler view and scroll to the bottom. However, the draw method is static, therefore the program does not need to create new objects and allocate memory to those MyUnit objects.

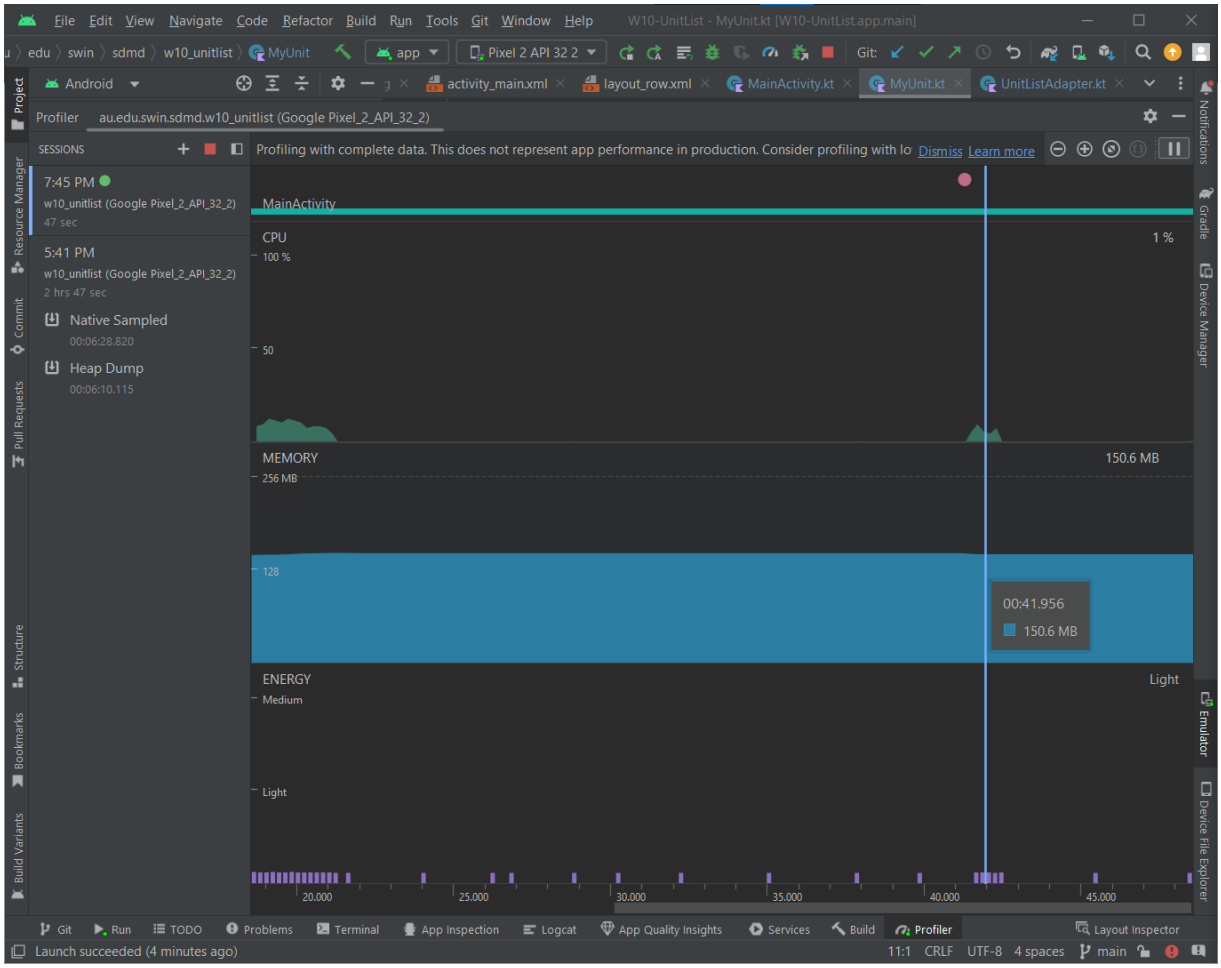

Figure 5: 2nd + click in the program

Similar to the dynamic way to draw the icon, the RAM increases by 0.2 MB every time a new object is drawn. The CPU usage was also similar at around 4-6% at peak.

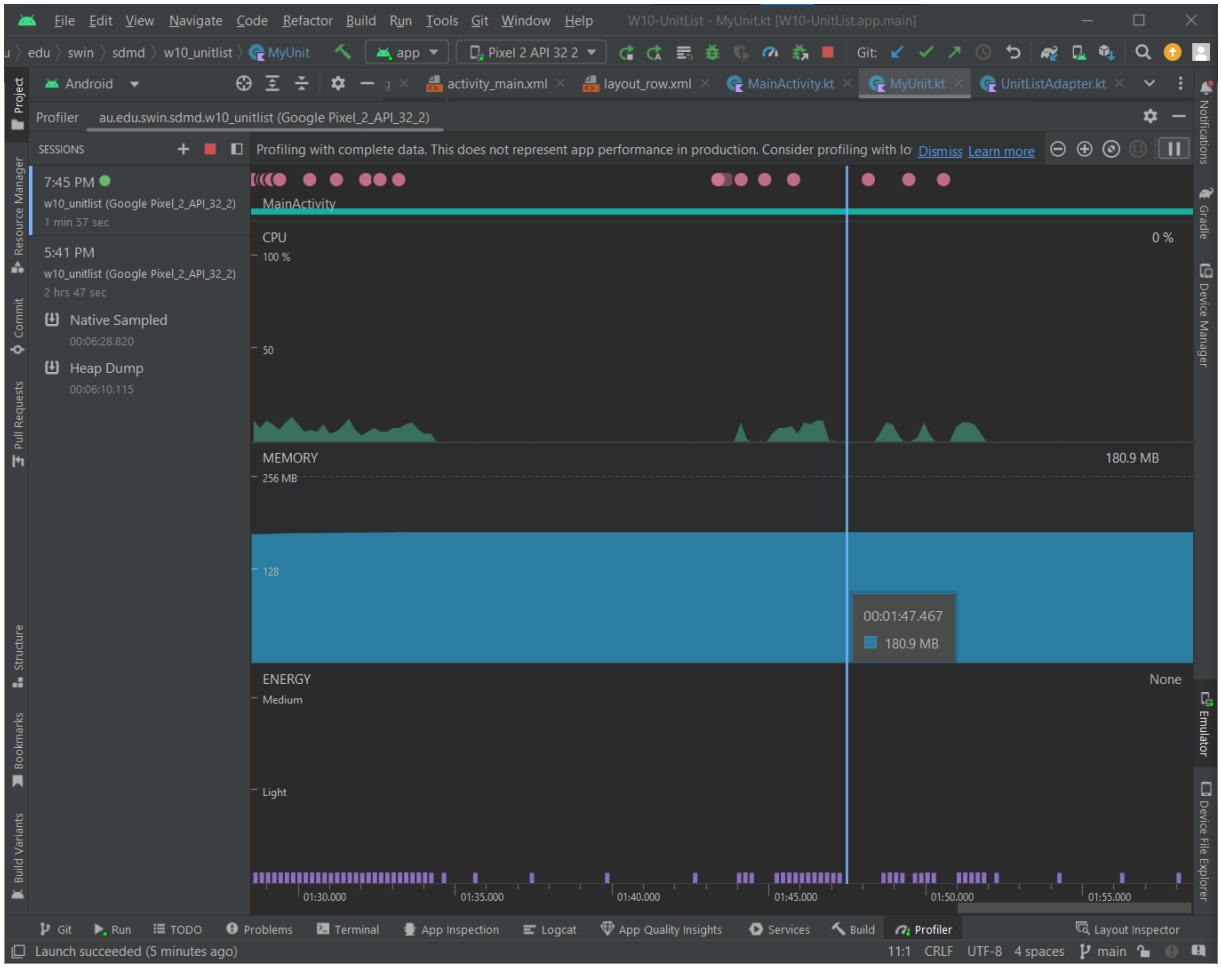

Figure 6: The program after multiple + clicks and scrolls

After multiple clicks, we can now see the difference between the different methods to draw the icon. The memory usage will increment gradually after each object is created and there will be no instances where the memory is released and reduced significantly as in the 1st scenario.

However, in contrast to scenario 1, when each action - such as scrolling or adding - was implemented, it immediately added memory usage to the initial memory allocation. It was only released when the program was destroyed or crashed. Second, when a new row was added from the top of the page, memory consumption briefly decreased by roughly 1MB, gradually growing over time before becoming stable at 147MB.

Scenario 3: Constantly draw the icon

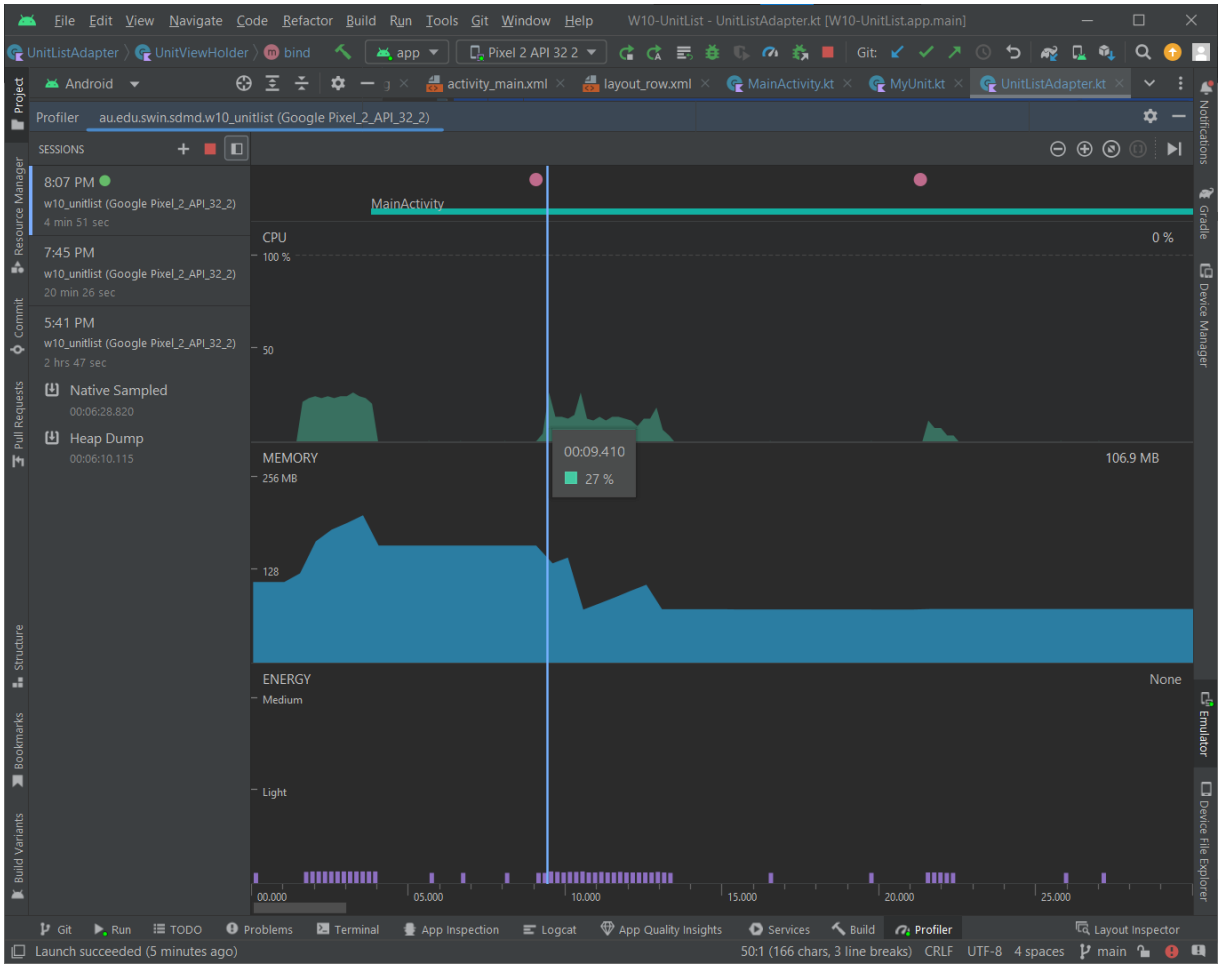

Figure 7: The 1st + click after initialization

In this approach, we can see the huge difference in memory performance in comparison to the 2 aforementioned methods of statically and dynamically drawing the icons. In the first click, the CPU spiked up at 27%, relatively lower than the first 2 approaches. The main difference is that the memory dropped significantly from 162 MB to 75 MB. This is an improvement in terms of performance as the program will be responsive and relatively faster at the start than the 2 previous approaches.

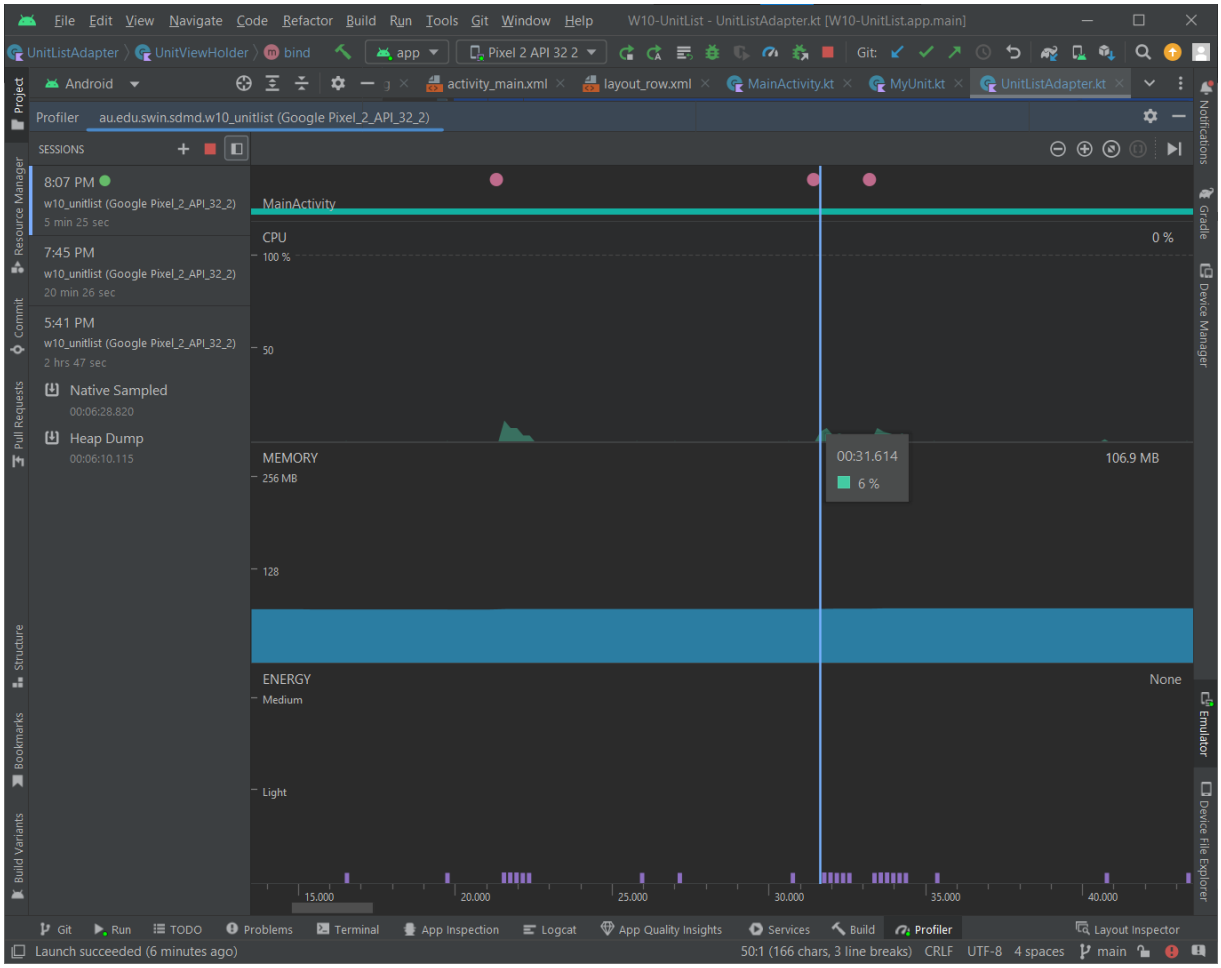

Figure 8: Clicks after the initialization

In the clicks after initialization, the RAM peaked at around 4-6% only. However, the memory allocation is gained by more than 5 MB after each object has been added. This can be a setback as the memory usage will be expanded at a much higher rate than adding the icon statically and dynamically.

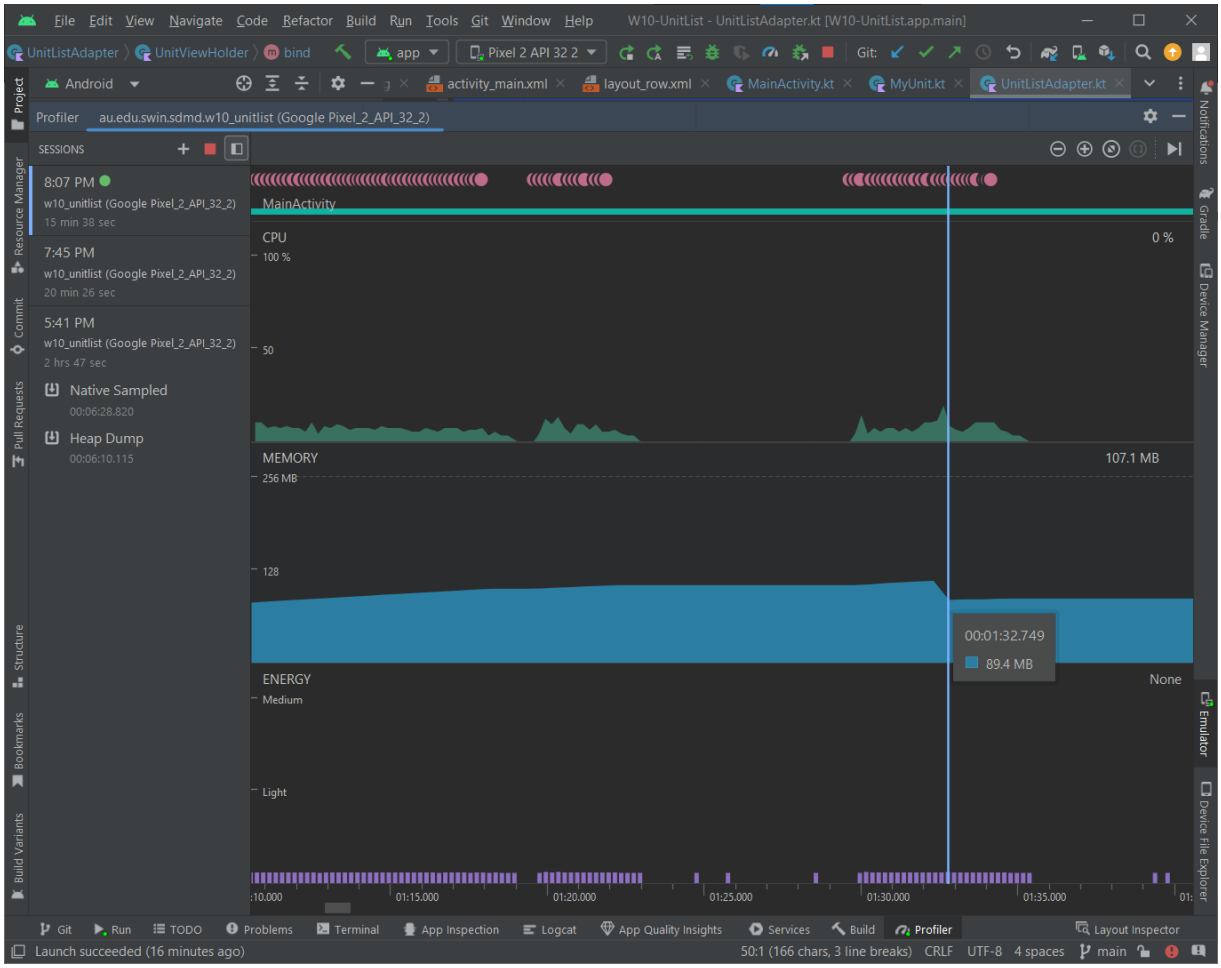

Figure 9: Memory release point after cumulating

After reaching a threshold of 107 MB, the memory was released and was lower to 87 MB. The CPU usage time extended similarly to the 2 initial approaches.

Analysis

After analyzing the CPU and RAM usage data from the three approaches, it is evident that the first approach of dynamically drawing the icon was the most optimal in terms of performance. With criteria in place that allowed for effective memory allocation and release, the RAM and CPU consumption was far lower than with the other two options. As a result, the software handled enormous data amounts and protracted processes without becoming sluggish or unstable. The third method, which had a rapid memory release at first, appeared promising, but when more items were added to the list, memory consumption rapidly increased due to inefficient memory allocation. Overall, the first strategy performed the best and ought to be used more frequently when creating Android apps.

In Approach 2, the static icon painting technique was used, which helped in the first phases to reduce CPU utilization and allocated memory. However, because the program doesn't release

memory, the application is more likely to experience memory-related problems as the number of items in the list increases, such as memory leaks or poor performance. This can be particularly problematic in circumstances when the app needs to handle big data sets or long-running processes since the memory usage can quickly spiral out of control and cause the app to crash or become unstable. To avoid these problems, it's crucial to carefully control memory allocation and release. You should also use alternative concurrency technologies, including multithreading and asynchronous programming, to improve the responsiveness and speed of your app.

About multithreading, the process describes breaking down to working on multiple tasks concurrently. Multiprocessing and asynchronous input/output are also among the crucial aspects that make concurrent approaches helpful when it comes to handling numerous items in a file. Overall, by using concurrent approaches to load a large number of items from a file, you can take advantage of the available hardware resources to speed up the process and reduce the overall time to load the data.

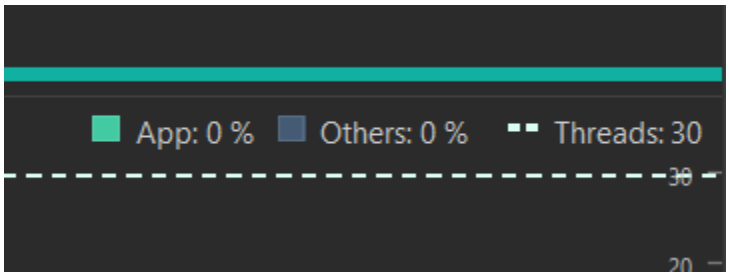

Figure 10: Threads

The takeaway from this task is that we need to use concurrent approaches to work with lists and images, as they take up a lot of memory usage. Therefore, the proper manner of presenting lists and images needs to be chosen to prevent memory leaks, overflow memory, and CPU process. With the limited amount of resources provided by mobile devices, this is a key consideration. Static methods proved to have major drawbacks when the CPU and RAM accumulate and this could exceed the machine's capacity. Due to the aforementioned instances, dynamic icons proved to be the most appropriate managed performance method while working with a long list.

When it comes to loading multiple items from a file, concurrent methods can be beneficial. Due to its multitasking nature, this approach's key benefit was that it took less time to process a large dataset. Three methods can be used to carry out a concurrent approach: parallel processing, asynchronous programming, and multithreading.

A task is divided into smaller, independent pieces that can be processed separately using the three fundamental procedures. The file can be divided into smaller chunks, with each one being handled by a separate thread. As a result, the time needed to read or load the contents into memory can be greatly reduced.

Conclusion

Many factors need to be taken into account when developing and maintaining an application for mobile software. Although mobile devices have limited resources, we also need to consider optimization and ways to enhance the performance of the software in addition to the attractive and responsive UI design. No user will be drawn to use an app that has a gorgeous design but loads slowly and burdens the host computer. As a result, we will examine the supplied app's performance in further detail throughout this report. We will next weigh the advantages and disadvantages of each situation before concluding. However, concurrent and dynamic approaches have proven their efficiency over static ones.