Hong Hai Dang Nguyen - 103503191

# Assignment: Extension on performance

## Introduction

Nowadays, when it comes to mobile app development, apart from the design aspects and the features the application has, developers also have to take into consideration the performance of the app and how well it is optimized on different devices. It is the users' expectation that the app they use must run smoothly with no noticeable lag or slowdowns, otherwise the app will be deemed as malfunctioning, thus proving the importance of optimization for any application.

In this report, we will analyze the performance of a given app (with a list-based user interface) in different scenarios. From the experiments, I will give out the key observations for them and my opinion on which scenario the app performed best and vice versa.

## Experimental scenarios

### *Scenario 1: A constant icon*

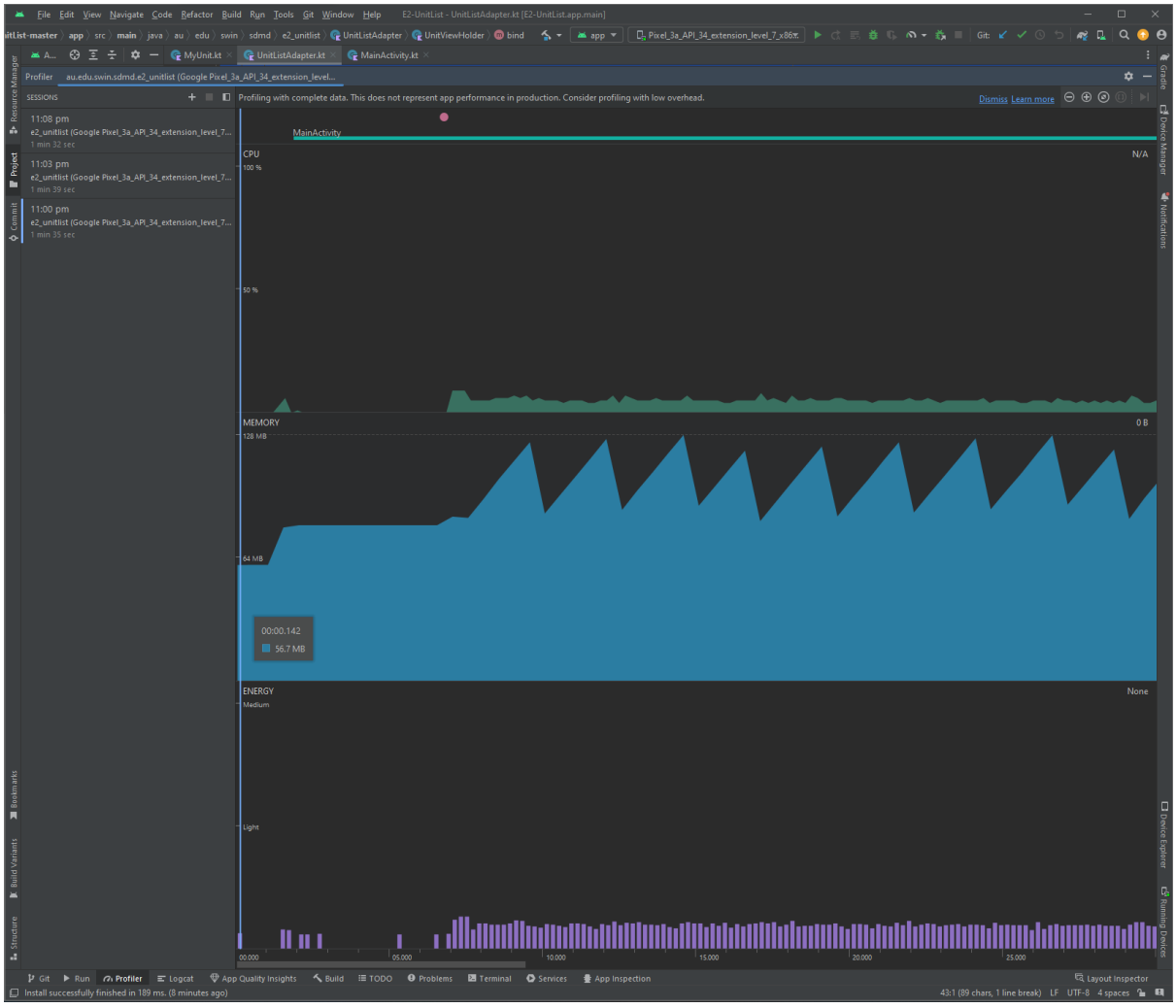


*Figure 1. The app initialization phase and when the FAB was first pressed.*

When the app was initialized for the first time, the CPU usage was at 6%. In terms of the memory, it had a jump from 57MB to 76MB, and then remained stable. When the FAB was pressed for the first time, the CPU usage was at 9%. In terms of the memory, it had a slight increase to around 80MB.

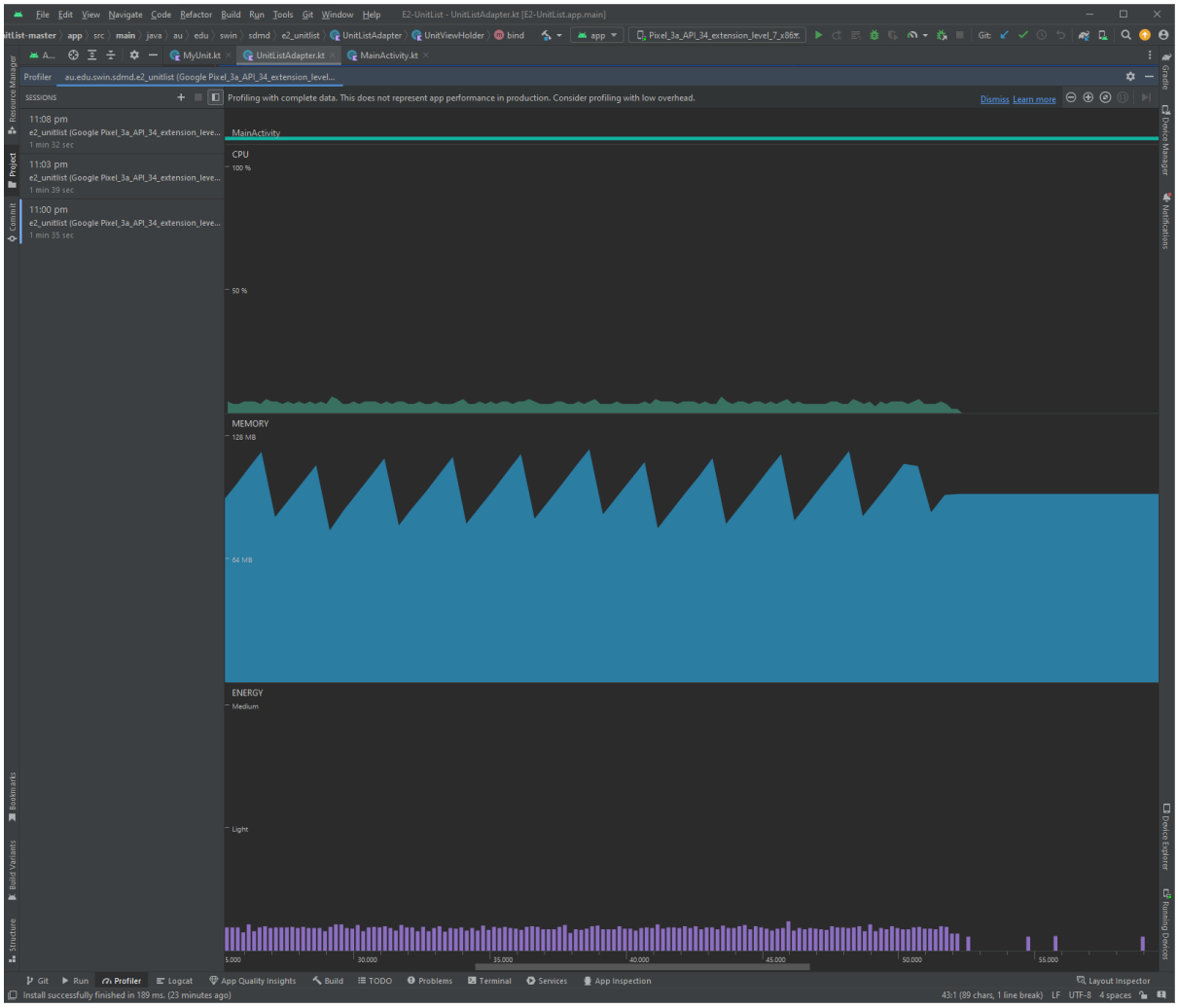


*Figure 2. The scrolling phase*

During the scrolling animation, the CPU usage fluctuated between 4% and 7%. In terms of the memory, the pattern was a jump from 80-86MB to a peak of 116-120MB and then swiftly returned to the aforementioned lower point. When the scrolling animation was finished and a new item was added at the bottom, the CPU usage was at 2%. In terms of the memory, it remained stable at 98MB.

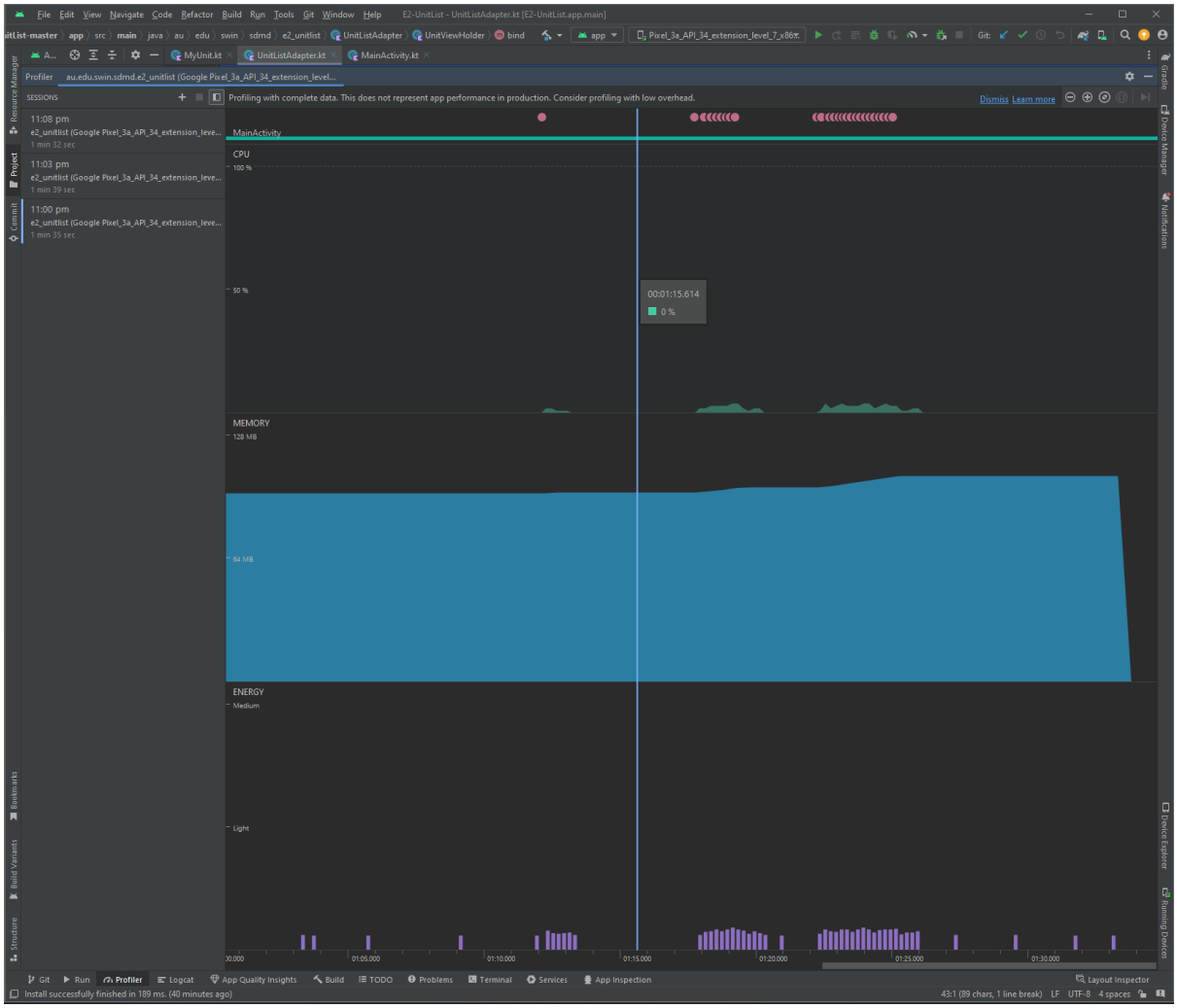


*Figure 3. One press of the FAB and multiple presses of the FAB.*

When the FAB was pressed only once, the CPU usage was at 2%. In terms of the memory, it had a slight increase from 98MB to 98.2MB. When the FAB was pressed multiple times, the CPU usage fluctuated between 3% and 5%. In terms of the memory, it had a steady rise from the first press at 98.2MB to the final press at 99MB.

***Scenario 2: A generated icon created on bind***

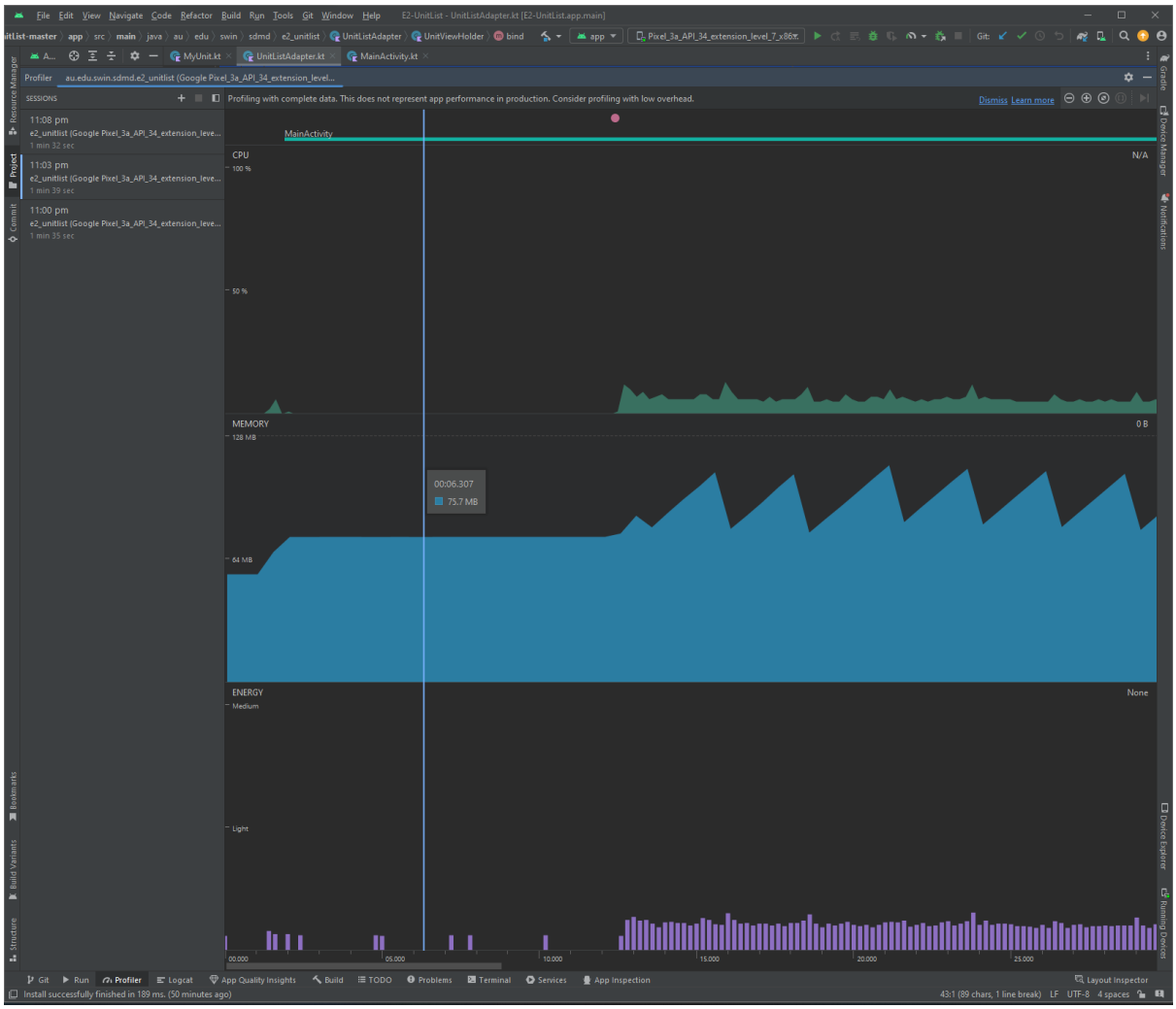


*Figure 4. The app initialization phase and when the FAB was first pressed.*

When the app was initialized for the first time, the CPU usage was at 7%. In terms of the memory, it had a jump from 58MB to 75.7MB, and then remained stable. When the FAB was pressed for the first time, the CPU usage was at 9%. In terms of the memory, it remained at 76MB.

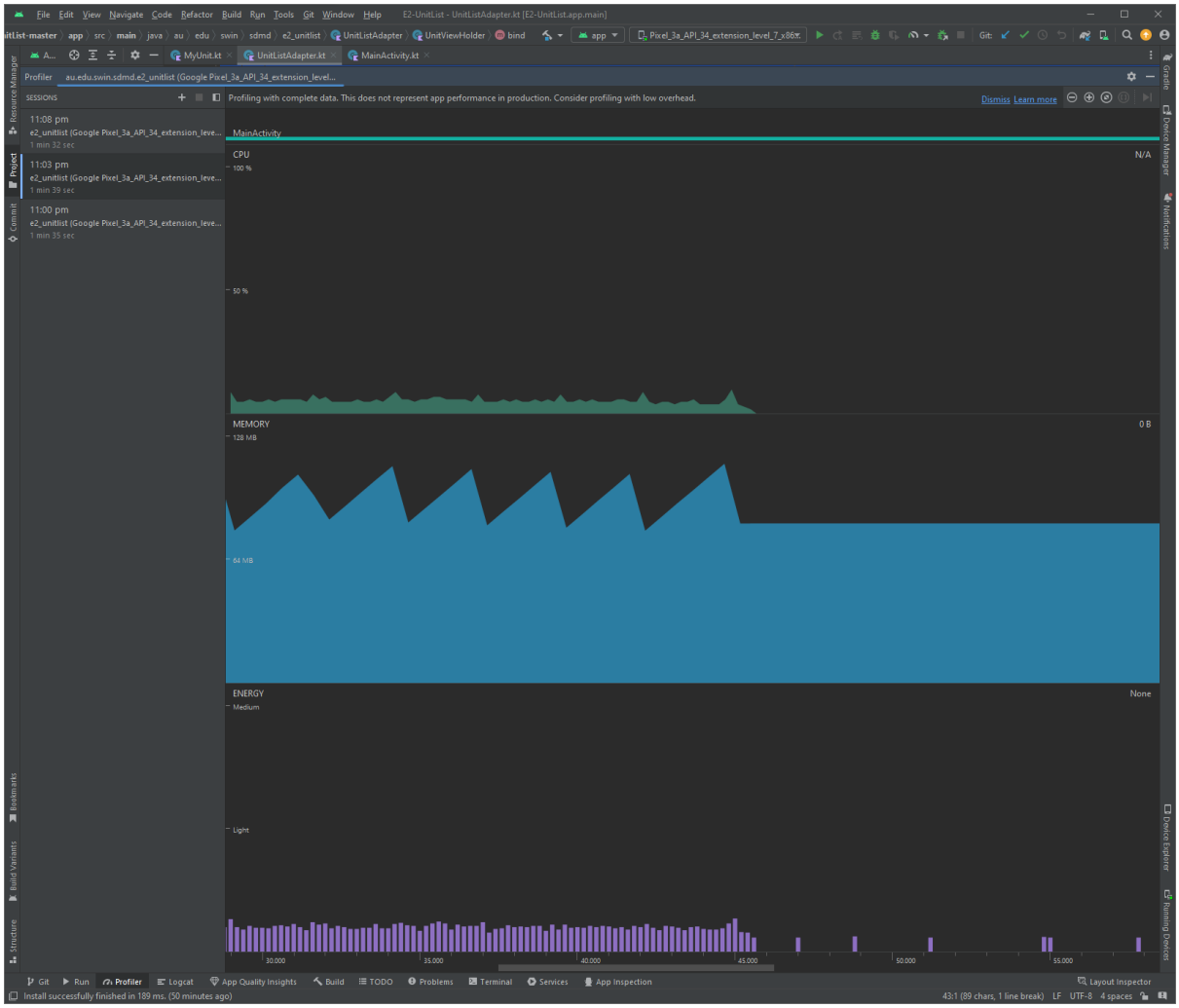


*Figure 5. The scrolling phase*

During the scrolling animation, the CPU usage fluctuated between 4% and 7%. In terms of the memory, the pattern was a jump from 81-86MB to a peak of 106-111MB and then swiftly returned to the aforementioned lower point. When the scrolling animation was finished and a new item was added at the bottom, the CPU usage was at 3%. In terms of the memory, it remained stable at 80MB.

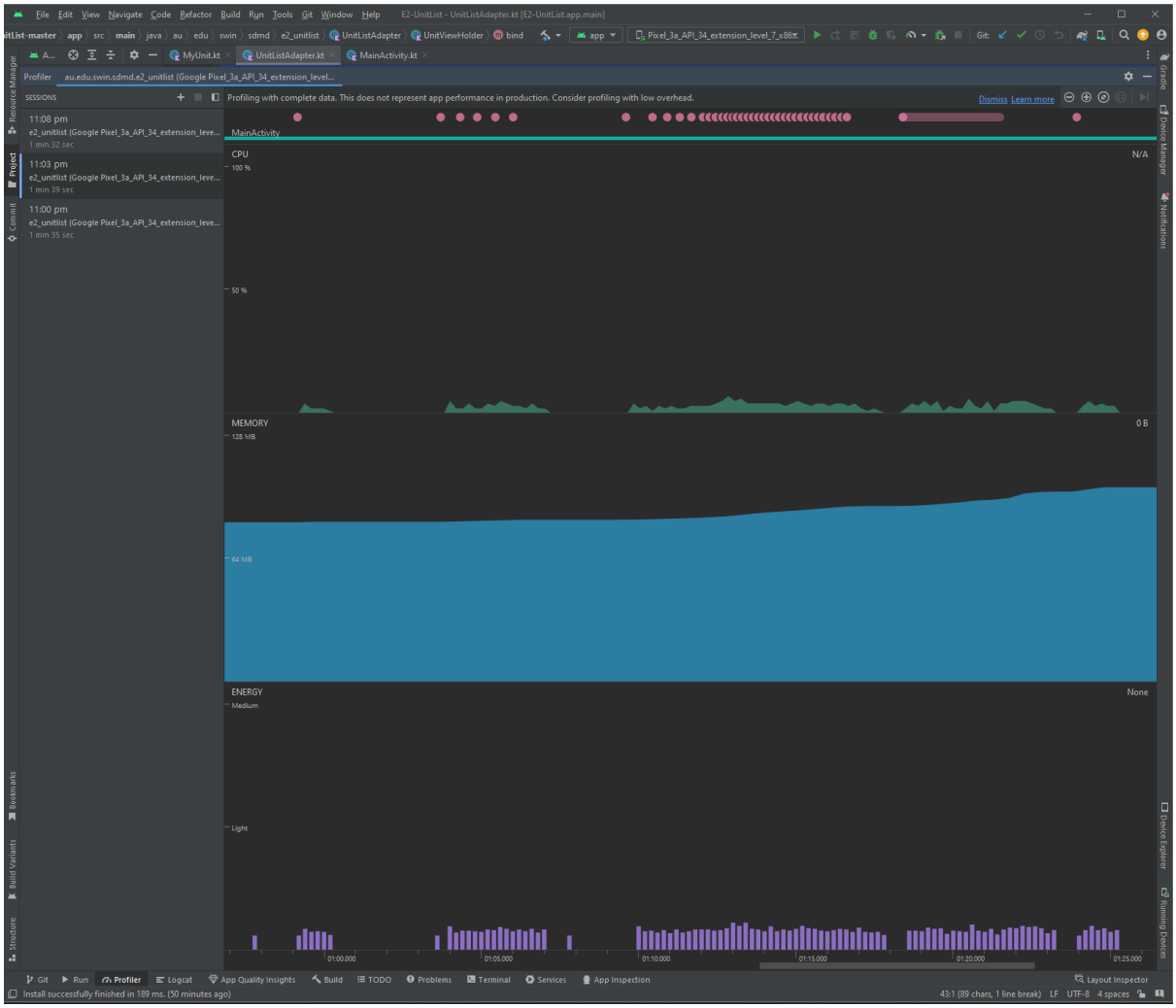


*Figure 6. One press of the FAB and multiple presses of the FAB.*

When the FAB was pressed only once, the CPU usage was at 5%. In terms of the memory, it had a slight increase from 80MB to 80.2MB. When the FAB was pressed multiple times, the CPU usage fluctuated between 3% and 6%. In terms of the memory, it had a steady rise from the first press at 80.2MB to the final press at 90MB.

***Scenario 3: A generated icon but created on initialization***

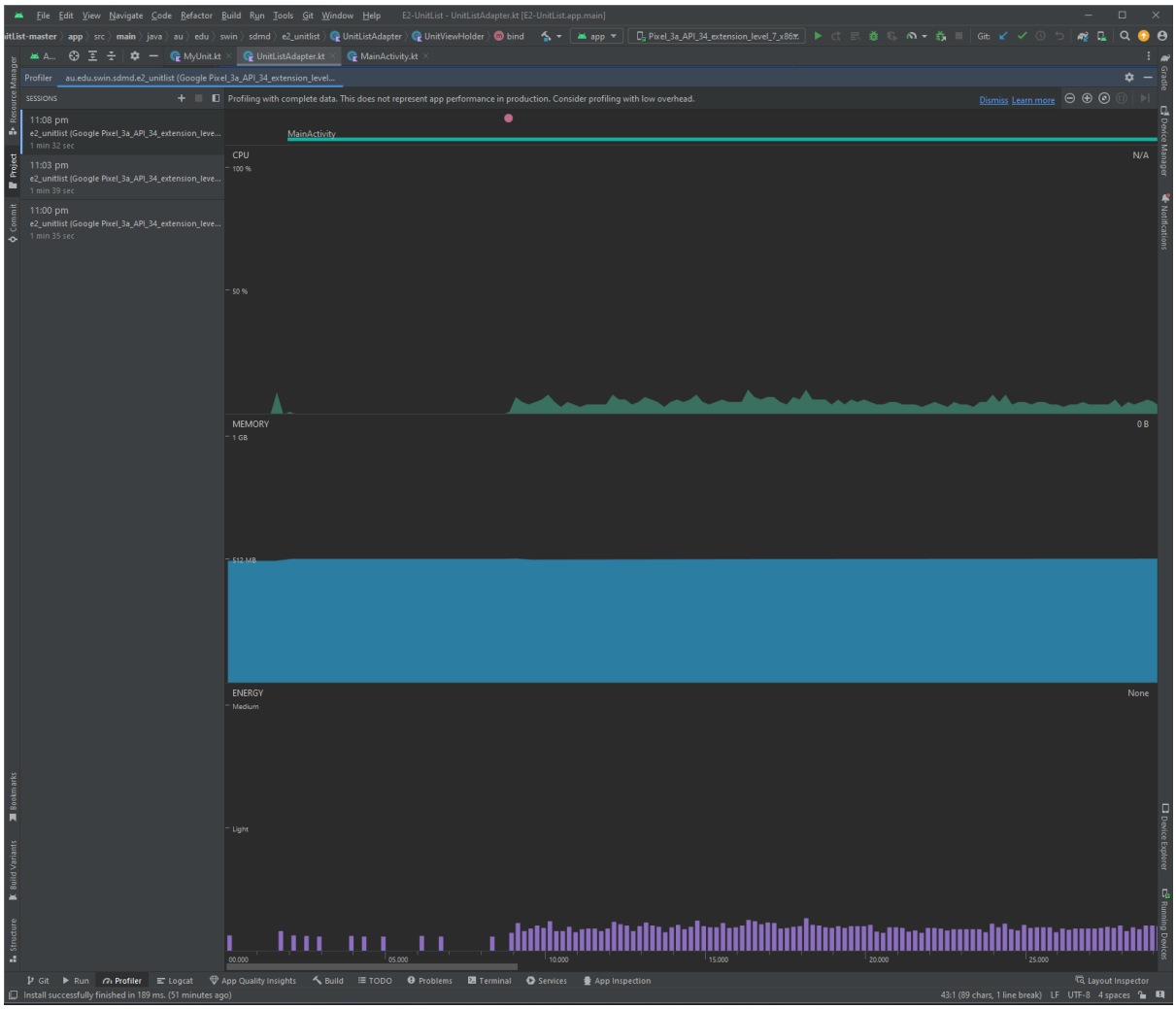


*Figure 7. The app initialization phase and when the FAB was first pressed.*

When the app was initialized for the first time, the CPU usage was at 7%. In terms of the memory, it had a jump from 512MB to 520MB, and then remained stable. When the FAB was pressed for the first time, the CPU usage was at 11%. In terms of the memory, it slightly increased to 521MB.

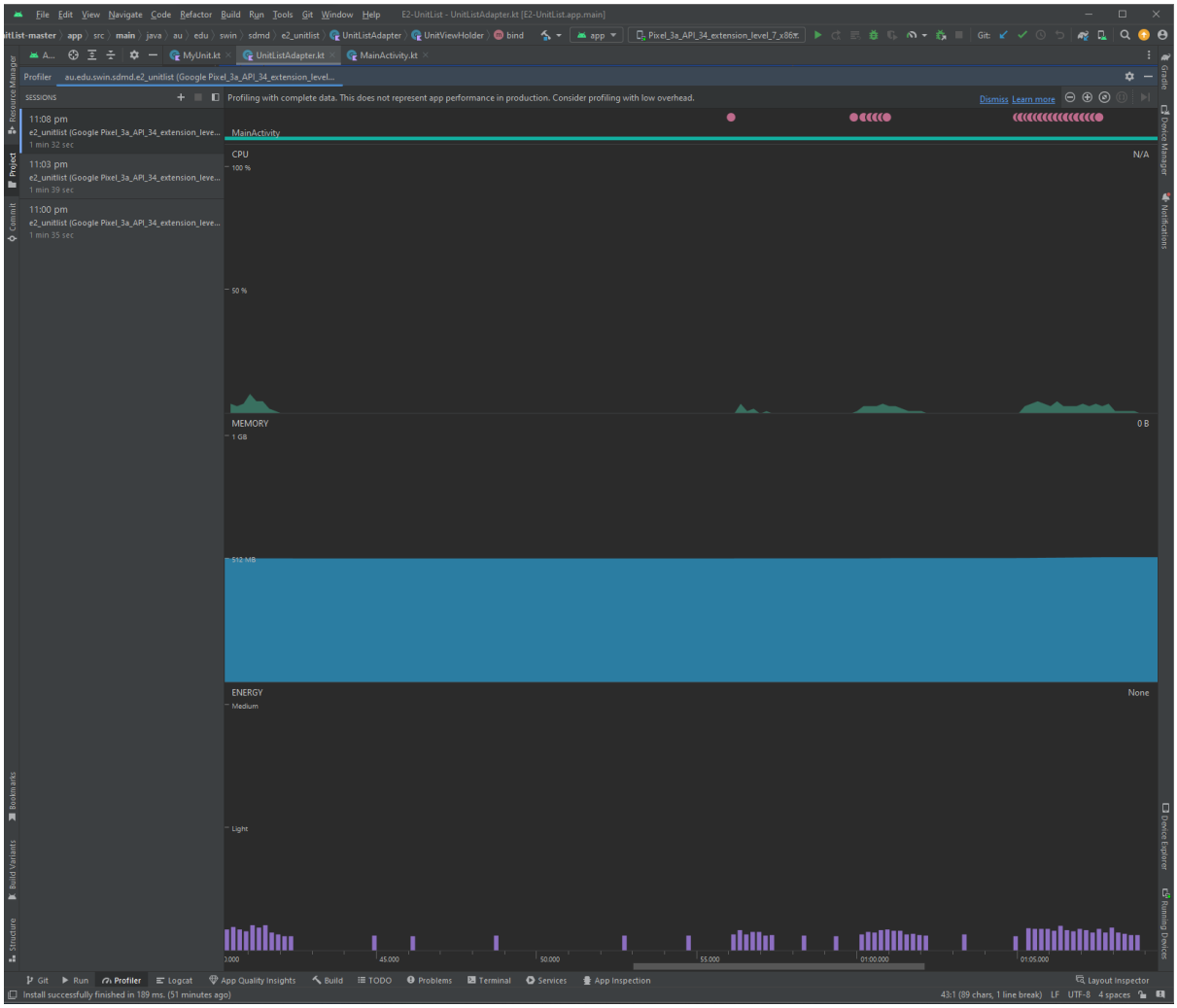


*Figure 8. The scrolling phase, one press of the FAB and multiple presses of the FAB.*

In general, in terms of the memory for all phases in the image above, it fluctuated slightly between 516MB and 520MB.

During the scrolling animation, the CPU usage fluctuated between 4% and 6%. When the scrolling animation was finished and a new item was added at the bottom, the CPU usage was at 3%. When the FAB was pressed only once, the CPU usage was at 6%. When the FAB was pressed multiple times, the CPU usage fluctuated between 2% and 6%.

## Analysis and comparison

From the experiments, it can be seen that Scenario 1 is relatively comparable to Scenario 2. The CPU usage of the 2 scenarios was almost the same, and the memory usage fluctuated in a spike-like graph shape during the scrolling animation, as well as it gradually increased after any press of the FAB. The only noticeable difference was the amount of memory usage. This can be seen most clearly during the scrolling animation:

- In Scenario 1, the memory usage experienced a jump from 80-86MB to a peak of 116-120MB, and then remained stable at 98MB after the animation ended.

- In Scenario 2, the memory usage experienced a jump 81-86MB to a peak of 106-111MB, and then remained stable at 80MB after the animation ended.

The memory usage in other phases in these 2 scenarios also had a difference. In general, it can be seen that Scenario 1 used more memory than Scenario 2 in all phases.

Scenario 3 has the most differences in comparison to the two mentioned scenarios. The most noticeable aspect is the memory usage, with a significantly higher usage at around 518MB for all phases. And unlike the two mentioned scenarios, during the scrolling animation, it didn't fluctuate a lot like the spike-like graph and remained fairly stable between 516MB and 520MB.

Overall, in my opinion, based on my observation and analysis, Scenario 2's performance is the most efficient management way for a fixed resource in comparison to the other two scenarios. When opposed to constantly constructing icons (as in Scenario 1) or during initialization (as in Scenario 3), dynamically drawing the icon during each "bind()" was the most optimal in terms of performance and is often more memory-efficient.

Scenario 3 was the worst-case situation, however. Because it creates icons during initialization while maintaining constant icons for all things, it seems to have performed less efficiently in comparison to the previous two scenarios. The CPU usage was relatively comparable to Scenario 1 and 2, yet its memory usage was significantly higher.

## Key takeaway

The key takeaway from this assignment is that since lists and pictures use a lot of memory, we must use concurrent approaches while working with them. In order to avoid memory leaks, overflowing memory, and CPU processing, the appropriate way to show lists and pictures must be chosen. With the limited resources afforded by mobile devices, this is an important factor. Static approaches have significant problems as the CPU and RAM accumulate, perhaps exceeding the machine's capability. From the experimental screnario, dynamic icons were shown to be the best performance solution when working with a lengthy list.

Concurrent methods might be useful for loading several things from a file. This method's main advantage was that it processed a huge dataset faster because of its multitasking nature. A concurrent approach can be implemented via parallel processing, asynchronous programming, and multithreading. These three basic techniques are used to break down a task into smaller, independent portions that may be handled individually. It is possible to split the file into smaller pieces, with each piece being processed by a different thread. The time required to read or load the items into memory might be significantly decreased as a consequence.

## Conclusion

In conclusion, when creating and maintaining an application for mobile software, several considerations must be made. Given the limited resources of mobile devices, optimization and methods to improve the performance of applications must also be taken into account in addition to the user interface's (UI) attractiveness and responsiveness. And to achieve the best efficiency, much thought must be given to how pictures are created and handled, especially in list-based interfaces. Working with lists and photos in Android apps is a significant challenge in terms of balancing memory utilization, CPU efficiency, and user experience.