Comparison of Smoothness in Progressive Web Apps and Mobile Applications on Android

CAMILLE FOURNIER

Master in Computer Science
Date: Spring 2020
Supervisor: Javier Cabrera-Arteaga
Examiner: Benoit Baudry
School of Electrical Engineering and Computer Science
Host company: Odiwi
Swedish title: Jämförelse av smidighet i progressiva webbappar och mobilapplikationer på Android

Abstract

One of the main challenges of mobile development lies in the high fragmentation of mobile platforms. Developers often need to develop the same application several times for all targeted platforms, raising the cost of development and maintenance. One solution to this problem is cross-platform development, which traditionally only includes mobile applications. However, a new approach introduced by Google in 2015 also includes web applications. Progressive Web Apps, as they are called, are web applications that can be installed on mobile and behave like mobile applications. This research aims at studying and comparing their performance to mobile applications on Android, especially in terms of smoothness, memory and CPU usage. To that end, we analyzed the Rendering pipeline of Android and Chrome and deducted a smoothness metric. Then, a Progressive Web App, a Native Android and a React Native Interpreted Application were developed and their performance measured in several scenarios. The results imply that Progressive Web Applications, though they have great benefits, are not as smooth as Mobile applications on Android. Their memory performance and CPU usage lag behind Native Applications, but are similar to Interpreted applications.

Sammanfattning

En av de största utmaningarna med mobilutveckling ligger i den höga fragmenteringen av mobilplattformar. Utvecklare behöver ofta utveckla samma applikation flera gånger för alla riktade plattformar, vilket ökar kostnaderna för utveckling och underhåll. En lösning på detta problem är plattformsoberoendeutveckling, som traditionellt endast innehåller mobilapplikationer. En ny strategi som introducerades av Google 2015 inkluderar emellertid också webbapplikationer. Progressiva webbappar, som de kallas, är webbapplikationer som kan installeras på mobil och bete sig som mobilapplikationer. Denna forskning syftar till att studera och jämföra deras prestanda med mobila applikationer på Android, särskilt när det gäller smidighet, minne och CPU-användning. För detta ändamål analyserade vi Rendering-pipeline för Android och Chrome och drog en jämnhetsmetrisk. Sedan utvecklades en Progressiv webbapp, en Native Android och en React Native Tolkad applikation och deras prestanda uppmättes i flera scenarier. Resultaten antyder att progressiva webbapplikationer, även om de har stora fördelar, inte är lika smidiga som mobilapplikationer på Android. Deras minnesprestanda och CPU-användning ligger efter Native Applications, men liknar tolkade applikationer.

Acknowledgement

Before presenting my thesis, I would like to thank the people who helped me conduct this research. First, I want to thank Cristian Bogdan who helped me choose a Master Thesis project and my examinator Benoit Baudry who helped me define it. Then, I want to thank the company Odiwi that welcomed me and helped me conduct my research with their equipment. Last but not least, I want to thank my academic supervisor, Javier Cabrera Arteaga, who helped me and supported me a lot during this project.

Contents

Chapter 1

Introduction

One of the biggest challenges of mobile development today lies in the high fragmentation of mobile platforms [1] with developers building the same app over multiple platforms. A popular solution is cross-platform development, that is to say developing with a framework capable of using the same code for multiple platform builds. However, such cross-platform development does not concern web applications that developers still have to develop separately from mobile applications. To answer this problem, Google introduced in 2015 the concept of *Progressive Web Apps* [2], a term first coined by designer Frances Berriman and Google Chrome developer Alex Russel [3, 4].

Progressive Web Apps, also called PWA, are Web Applications that can be installed on a mobile phone by the browser and provide an offline experience. End users may even think of them as real mobile applications as they are displayed in fullscreen and can be detected by the Application Manager depending on the device and the browser. Progressive Web Apps have become more and more popular since they were introduced to the application development community. Several success stories can attest it : AliBaba[1] and Aliexpress[2] reported an increase of respectively 76% and 104% of their conversion rate after releasing their PWA. Twitter[3] saw a reduction of their data usage coupled with a 75% increase of tweets sent. Pinterest[4] reported a 40% increase of user time and a 44% of user-generated revenue.

[1] https://developers.google.com/web/showcase/2016/alibaba

[2] https://developers.google.com/web/showcase/2016/aliexpress

[3] https://medium.com/@paularmstrong/twitter-lite-and-high-performance-react-progressive-web-apps-at-scale-d28a00e780a3

[4] https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154

Existing research suggests that PWA are a good alternative to mobile applications. Their installation size and energy consumption are smaller[5]. Their launch time [6] and their First paint metric[5][5] can be lower than a mobile application. For a developer, they are not much more complex than regular web applications to develop[7] and easier to maintain than native applications.

However, no research has been conducted regarding the run-time performance of Progressive Web Applications, especially the performance of the User Interface, which can be evaluated with two key elements: the responsiveness and the smoothness. The responsiveness refers to how fast the application can respond to user input. The smoothness refers to how fast the application render new frames, that is how fast the application can draw new content on the screen. The latter can impact the former as applications usually respond to user input visually and need to render new frames. Other important factors of the run-time performance are CPU usage and memory performance as those resources are limited on mobile phones. At the time of writing, there was no study that examined those performance parameters in Progressive Web Apps.

This thesis will focus on several aspects of the run-time performance of Progressive Web Apps and aim at answering the following questions:

- **RQ1:** How does the smoothness of Progressive Web Apps compare to Mobile applications?
- **RQ2:** How does the memory performance of Progressive Web Apps compare to Mobile applications?
- **RQ3:** How does the CPU usage of Progressive Web Apps compare to Mobile applications?

As Chrome browser holds more than 60% of the browsers market share on mobile, the performance of Progressive Webb Apps will be evaluated when running on Chrome browser in this thesis.

The global challenge behind these research questions is to collect relevant performance metrics. For this, we need different versions of an application running in a controlled environment, and tools that monitor the performance metrics.

[5]Time to render the first pixel

We address the first challenge with 3 applications: a Native Android app, a cross-platform app developed with React Native and a Progressive Web App developed with ReactJS. To address the second challenge, we investigate several techniques and tools. Many tools already exist to monitor the performance of mobile applications on Android. The challenge resides in monitoring the performance of Progressive Web Applications. For the memory performance, we inspect the usage of tools specific to mobile applications on Progressive Web Apps. For the CPU usage, we examined tools specific to web applications and compare them to mobile applications tools. For the smoothness performance, there were no frameworks or tools able to assess the smoothness of Progressive Web Apps at the time of writing.

The key technical contribution of this thesis consists in the elaboration of a model of Chrome frame rendering workflow and a tool *FrameTracker* which follows the frames and extract key timestamps. They can be used by other developers to get detailed information of their web applications rendering process, and by other researchers to understand more deeply the rendering process of web applications and compare it to mobile applications.

The novelty of this work resides in several points. At the time of writing, no known research has studied the smoothness, the memory performance nor the CPU usage of Progressive Web Apps. Moreover, the method used to compare the smoothness of the applications has never been used before, as far as we know. The model of Chrome's rendering pipeline built empirically is also novel to the best of our knowledge.

Our experiments based on 3 applications, reveal that Progressive Web Apps are less smooth than Native and Interpreted applications on Android. The difference is significant and will be noticeable in applications requiring a high smoothness performance, for example games or heavy animations. However, the difference perceived by the end-user in simple applications remains to be studied. Meanwhile, PWAs and Interpreted applications have similar CPU and memory consumption. We observe a difference of 20% in favor of Progressive Web Apps. However, the difference reach 200% between Native and Progressive Web Apps, making the Native applications still the most performant in terms of resource consumption.

This work is structured as follows. Chapter 2 introduces the concepts and the academic background related to this research. Chapter 3 presents the methods

used to overcome the different challenges and answer the research questions. Chapter 4 outlines the empirical model of the frame rendering workflow on Chrome. Chapter 5 describes the results obtained during this study. Chapter 6 discuss those results and Chapter 7 concludes this work.

Chapter 2

Background

The aim of this chapter is to introduce the terminology and concepts used in this study. We will also present the work previously done in the field of Progressive Web Apps and analyze the tools available to measure the performance of Android mobile applications and PWA.

2.1 Mobile applications

2.1.1 Development

There are a number of ways to develop a mobile application, the most common one being to develop it natively, i.e. develop the mobile application once for each targeted platform (iOS, Android or Windows Phone) using their respective environment, Software Development Kit and programming languages. Since this can be quite costly, an alternative has emerged in the form of cross-platform development, that is using the same code to build the application for several platforms.

Traditionally, mobile cross-platform development is divided into four different approaches [8]:

- Web: The web app is run by the browser on the mobile device. PWAs are an improvement of this approach.
- Hybrid: The application is built using web technology but executed inside a native container. The app is rendered through full screen webviews. Frameworks like PhoneGap and Ionic [9] use this approach.
- Interpreted: The application code is run by an interpreter which uses

native components to create a native user interface. Frameworks like React Native, NativeScript and Titanium [10] use this approach.

- Cross-Compiled: The source code is compiled into a native application by a cross-platform compiler. Xamarin [9] uses this approach.

2.1.2 Emulators

It is considered a largely accepted best practice to test an application under development before production. This can be done using physical devices or emulators. An emulator is a software that simulates the OS and the hardware capabilities of a device[11]. This can be a great way to automate testing among a large range of smartphones. Most of the Android emulators found during this research are targeted at gamers and enhance the hardware capabilities usually found in the simulated physical device. Only a few can be used by developers to test the performance of their applications among different smartphones:

Android Emulator[1] is the official emulator for Android provided by Google along Android Studio. It offers the latest Android APIs but a limited range of physical devices. However, it is possible to customize hardware characteristics and the device's look.

Genymotion[2] contains a wider range of smartphones but fewer APIs than Android Emulator. Genymotion can be linked with Android Studio in order to test in-development apps. It is also possible to customize hardware characteristics.

Visual Studio Android Emulator[3] is the solution offered by Microsoft on Visual Studio. However, it is not supported since Visual Studio 2017. They recommend using the official Android Emulator instead. It offers a limited set of hardware and API configurations that matches several smartphones at once.

[1] Android Emulator: https://developer.android.com/studio/run/emulator/

[2] Genymotion: https://www.genymotion.com/desktop/

[3] Vsial Studio Emulator: https://docs.microsoft.com/fr-fr/visualstudio/cross-platform/visual-studio-emulator-for-android?view=vs-2015

2.2 Progressive Web Apps

2.2.1 Concept

A Progressive Web App, or PWA for short, is a regular web application with a few more features such as offline and install capacity. Its aim is to reduce the gap between mobile and web development. Once it is installed, a PWA should behave just like a native app for the end-users. Thus, the end-user should not be aware of the browser running the app behind the scenes, and be able to launch the app without internet connectivity.
The concept of Progressive Web Apps holds a lot of potential [4]. It could allow developers to use the same code for the web, the mobile, and the desktop app depending on browser implementations. This would considerably reduce the development and maintenance cost of a single application. Moreover, deployments and updates would not have to go through the app stores to be available to users, further reducing the overall cost of the application and increasing its access.

In order to be called Progressive, a web application has to implement several features[12]:

- **Progressive:** Work for every browser
- **Responsive:** Fit any screen size
- **Connectivity independent:** Be able to work offline or with low-connectivity thanks to a service worker
- **App-like:** Have a Native-like interaction by using the app-shell model
- **Fresh:** Keep the app up-to-date with the service worker
- **Safe:** Be served over TLS to prevent snooping and content tampering
- **Discoverable:** Be identifiable as "applications" thanks to W3C manifests and service workers
- **Re-engageable:** Use re-engagement features like push-notifications
- **Installable:** Save the app on the home screen without going through an app store
- **Linkable:** Share the app with a simple URL

2.2.2 Architecture

In practice, Progressive Web Apps contains 3 main components: the web app manifest, the service worker and the app shell.

App Manifest
The web app manifest is a JSON file containing the metadata concerning the native display of the app once installed on the user's phone. In this file, the developer can define the app's metadata such as the splash screen, the app icon, the theme color and the app title. Without it, the app is uninstallable.

Service Worker
The service worker is the central part of a PWA. It is responsible for the new background features such as push notifications, offline experience and background synchronization[13]. The service worker acts as a proxy server for the PWA: it intercepts requests, caches the response or gives the already cached response. As a worker, it does not have access to the DOM and works on a different thread as the main app. This is especially useful not only for offline capability but for background optimization tasks.

App shell
The app shell is essentially the User Interface of the app without any content [14] (the data and images fetched remotely). By caching it via the service worker, it offers the user a better performance with instant loading of the UI on repeated visits and a better low-connectivity experience.

2.3 Profiling tools

Profiling tools are software or command-line tools that help developers evaluate the performance of their app and identify performance bottlenecks. They measure metrics such as the CPU, the memory used or functions called during a recording.

2.3.1 Android

Few tools are available to profile Native and cross-platform applications on Android aside from Android Profiler [15]. This software can provide real-time information about CPU, Memory, Network and Battery consumption in the form of graphs. While it does not require any specific install as it comes with Android Studio and provides a User Interface, it can have a high overhead [15] and make the app less performant during recording than it is usually.

Android also provides other command-line tools to profile applications. Most of them are only available through Android Debug Bridge (ADB)[4] which allows a computer to communicate with a device either to get information or execute commands.

For example, dumpsys[5] can provide a lot of information about the device or the processes currently running. This information is available through the system services such as *meminfo* (memory), *cpuinfo* (CPU usage), *gfxinfo* (animation), *netstats* (network) or *batterystats* (battery).

Another example is top. It comes from the Linux command-line of the same name[6], but with limited options. It displays the process activity in real-time such as CPU, memory and process priority.

Systrace[7] is also a useful tool and does not require ADB. It records a device activity and outputs an HTML file which can be viewed on Chrome browser. This file displays a timeline of events for each thread running during the recording, the CPU activity divided for each CPU, frames and other events depending on the categories of events selected. It is mainly used to identify the cause of bottlenecks in an application.

While not a profiling tool in itself, Monkeyrunner[8] is a useful tool when testing an application. Its purpose is to automate testing by interacting with the device from a Python script. It can also take snapshots or execute *adb shell* commands, such as *dumpsys* and top viewed previously.

2.3.2 Progressive Web App

Even though Progressive Web Apps can be installed and managed like a native application for the end-user, they are still Web applications. Thus, they have their own set of profiling tools provided by the browser. Since this study is limited to PWAs running on Chrome, this section will focus on Chrome's profiling tools. They may not work on other browsers.

[4] ADB: https://developer.android.com/studio/command-line/adb
[5] Dumpsys: https://developer.android.com/studio/command-line/dumpsys
[6] Ubuntu manuals: http://manpages.ubuntu.com/manpages/xenial/man1/top.1.html
[7] Systrace: https://developer.android.com/topic/performance/tracing
[8] Monkeyrunner: https://developer.android.com/studio/test/monkeyrunner

One of them is the Chrome's DevTools, a complete set of developer tools available on Chrome Browser. It is divided into several panels, each with its own purposes. For example, the *Network* panel displays all requests done by the web app, the *Console* panel displays debug and error messages and the *Performance* panel [16] displays an overview of the app performance, such as a timeline of function calls, a CPU graph, a memory graph and information regarding frames.

Lighthouse [17] is another tool of the Chrome's DevTools, available in the *Audit* panel. Though its purpose is to automatically assess any web page's quality (Best Practices, Performance, Search Engine Optimization, Accessibility, Progressive Web App), its main target are Progressive Web Apps [4].

Chrome DevTools Protocol[9] is a set of commands and events used to communicate with the browser and the web page. Chrome DevTools takes its source of information from it. A number of other tools relies on it for their features [18]. It may be used for browsers other than Chrome if they are based on Blink.

Telemetry[10] is a framework used for automated performance testing on Chrome. It can be used on several platforms (desktop or Android). It can automatically interact with a web page while taking measurements.

2.4 Rendering Pipeline

The smoothness of an application greatly depends on the frame rate. A frame is an image displayed by the application on the screen. Each time something on the screen needs to be changed, a new frame is produced and displayed. We call that rendering. The faster an app can produce a new frame, the faster it can display a response to user input and the more animations it can properly handle without blocking user interaction.
Though the goal of Progressive Web Apps is to look and behave the same way as mobile application once installed on a smartphone, they rely on different technologies to do so. Thus, the rendering pipeline (the stages by which a frame is produced) can differ.

[9] Chrome DevTools Protocol: https://chromedevtools.github.io/devtools-protocol/
[10] Telemetry: https://github.com/catapult-project/catapult/blob/master/telemetry/README.md

2.4.1 Android

The Android Graphics pipeline is divided between 3 main components: the Application Renderers (1), SurfaceFlinger (2) and the Hardware Composer (3) [19]. When an application wants to display something on the screen, its Application Renderer (1) (usually provided by Android framework) gives a buffer of graphical data to SurfaceFlinger (2). SurfaceFlinger then takes all available buffers and composes them into a single buffer that it passes on to the Hardware Composer (3) (see Figure 2.1). The Hardware Composer (3) is responsible for actually displaying the buffer on the screen.

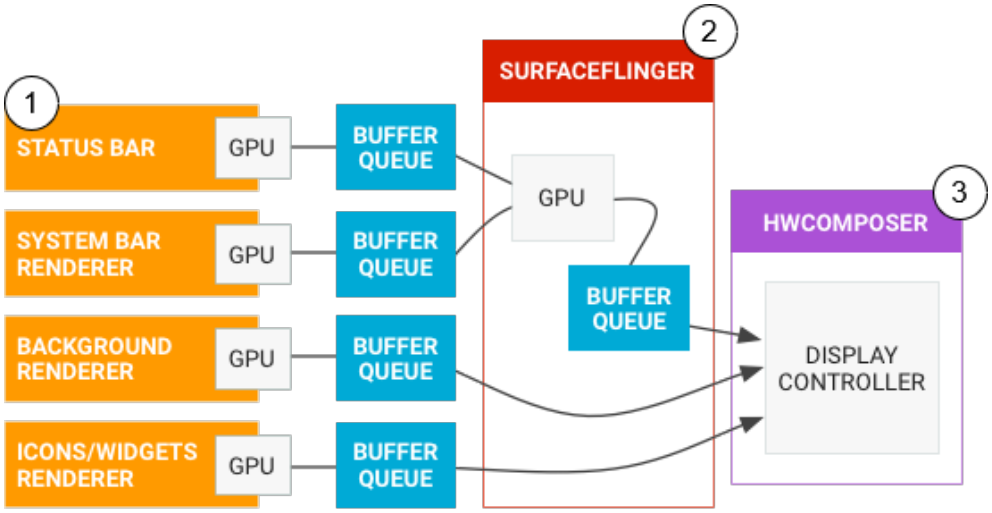


Figure 2.1: Graphic data flow through Android [11]

Everything displayed on the screen has to go through SurfaceFlinger and the Hardware Composer. All these components synchronize with each other thanks to the VSYNC signal [20]. When VSYNC signal is fired, all three components wake up at the same time: the Hardware Composer displays frame N, SurfaceFlinger takes a look inside the BufferQueue and composes frame N+1 if a new buffer is available, and the App Renderers generate frame N+2. The VSYNC signal depends on the device refresh rate, usually set at 60Hz though some new devices also support 90 or 120Hz [21].

Application rendering itself is divided into 7 different stages [22]:

1. Input handling: executes input events callbacks

[11] Source: `https://source.android.com/devices/graphics`

2. Animation: executes animators callbacks
3. Measurement and Layout: computes the size and position of all items
4. Draw: generates the frame's drawing commands in the form of a display list
5. Sync and Upload: transfers objects from CPU memory to GPU memory
6. Issue commands: sends the drawing commands to the GPU
7. Process and Swap buffers: pushes the frame's graphical buffer into the BufferQueue shared with SurfaceFlinger

The Application rendering of Android's framework is synchronous as each stage will happen one after another. Only Input handling and Animation may not always happen depending on inputs and callbacks.

2.4.2 Chrome

Before understanding Chrome's rendering pipeline, it is best to first understand its architecture. It is based on multiple processes running at the same time, with each its specific function [23] (Figure 2.2):

- Browser: controls everything related to the browser such as address bar, network requests and file access.
- GPU: displays all the elements on the screen by calling the OS's graphics library.
- Renderer: runs the web application. Each tab has its own Renderer sandboxed process with limited rights to protect the user and avoid crashing the browser.
- Plugin: runs the plugins used by the web application.

Each process also has several threads running at the same time for optimization.

[12] Source: *Inside look at modern web browser (part 1)* [23]

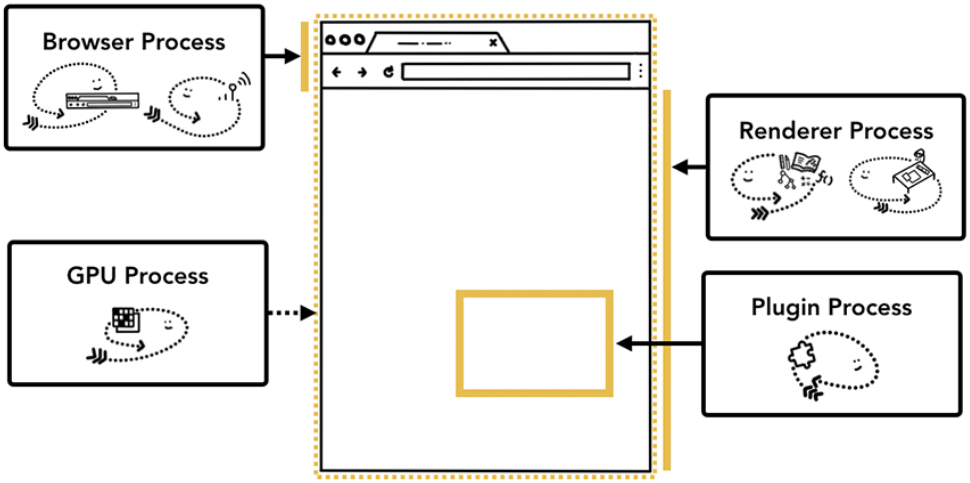


Figure 2.2: Chrome's multi-process architecture [12]

Chrome's rendering pipeline [24] contains similar stages as Android's application rendering but with some differences:

1. Parsing: translates the HTML into a DOM Tree.
2. Styling: applies CSS style to DOM elements.
3. Layout: computes the geometry of DOM elements (size and position) and produces a layout tree.
4. Compositing: decomposes the page into layers which can be independently painted and rastered.
5. Pre-painting: builds the property tree to apply to each layer.
6. Painting: records paint operations into a list of display items.
7. Tiling: divides the layers into tiles making its raster less expensive if only a part of it is needed
8. Raster: executes the paint operations and decodes images to produce a bitmap.
9. Drawing: generates a Compositor Frame and sends it to the GPU process.

Every Renderer and the Browser Process can compute frames according to the pipeline described above. The GPU processes then aggregates all Compositor Frames according to the surface they represent on the screen and sends calls to the OS's graphics library. For Android, that means pushing a buffer of graphic data to SurfaceFlinger.
As opposed to Android, the output of the different stages are reused whenever possible. The frame is divided so that a small change in the frame only triggers a small amount of work to render it. For example scrolling only changes the position of a layer. There is no need to go through Parsing, Styling and Layout again. Pre-painting, Painting, Tiling and Raster may also be skipped if the size of the frame computed previously was bigger than the display screen.

2.5 State of the art

This section will first present the academic research related to Progressive Web Apps and then describe other works that evaluated the performance of cross-platform frameworks.

2.5.1 Progressive Web Apps

Though Progressive Web Apps have sparked a real interest in the mobile and web development industry, only a few research papers studied them[4, 6, 25].

Biørn-Hansen, Machrazk and Grønli tried to raise interest in the academic community with three successive papers: *Progressive Web Apps: The Possible Web-native Unifier for Mobile Development* [4], *Progressive web apps for the unified development of mobile applications* [6] and *Progressive Web Apps: the Definite Approach to Cross-platform development?* [25]. Their first two papers also included a study of PWA's performance compared to other mobile development approaches, namely hybrid and interpreted for their first paper, and native, hybrid, interpreted and cross-compiled for their second. They looked at launch time, the size of installation and the app-icon-to-toolbar-render time with an online stopwatch. While the PWA was a lot smaller and launched faster than any other app, the time from app icon to toolbar render depended on whether the browser was already running on the background. If so, the PWA was the fastest and if not, was slower than the native and interpreted app.

Kressens [5] compared the First Paint metric[13] of a Progressive Web App, a regular Web App and a Native App on both Android and iOS with a fast and slow network, as well as the energy consumption of a PWA and Native applications on iOS and Android. He found that the PWA launched faster than the regular Web App and the Native Android app, and slightly slower than the Native iOS App. The energy consumption was similar between the PWA and both Native apps but slightly lower for the PWA than the Native Apps. The overall conclusion of this research is that PWAs are a viable alternative to native applications.

Yberg [26] compared manually the launch time of an android native application and a PWA with and without the browser in memory. The PWA was faster (with the browser in memory) or as fast as the native application.
Malavolta, after presenting Progressive Web Apps as a mobile development strategy [27], focused on the impact of service workers on PWA's energy consumption and found no significant impact[28].
Fransson and Driaguine [29] compared the response time of PWAs and Native Android Applications when accessing the camera and geolocation. The Progressive Webb App was faster at using geolocation than its native counterpart. On the contrary, camera access was faster on the native app than the PWA.
Johannsen [7] evaluated the code complexity brought by Progressive Web Applications to a regular application using different metrics such as the cyclomatic complexity and Halstead effort. He concluded that the added complexity was low.
Lee et al. [30] explored the security system behind the push notification system for Progressive Web Apps and found several concerning flaws which they reported to the vendors.
Gambhir and Raj [31] analysed the impact of service workers on the performance of Progressive Web Apps and compared it to Android applications, especially the response time of the servers. They found that PWAs could perform better than Android applications.
Lastly, at the time of writing only two research papers examined the user experience offered by Progressive Web Apps. Cardieri and Zaina [32] conducted a qualitative analysis of user experience on three platforms: web mobile, native android and PWA. They concluded that there was no significant difference of user experience between the platforms. Fredrikson [10] also supports this conclusion with a quantitative and a qualitative study comparing user experience with a React Native App, a Native Android App and a Progressive Web App.

[13]Time to render the first pixel

2.5.2 Performance of Cross-platform applications

The performance parameters used to compare mobile applications can differ greatly between studies.

Biørn-Hansen, Grønli and Ghinea [33] analyzed the metrics commonly used in animation performance evaluation: frames per second (FPS), CPU usage, device memory usage and GPU memory usage. They also used those metrics to evaluate the animation performance of several mobile applications developed in Native Android, Native iOS, React Native, Ionic and Xamarin. They concluded that FPS and GPU memory were not really useful to compare their performance, as the GPU memory did not change much between the animations and the FPS was not relevant to short-running animations like the ones they tested. The results suggest that React Native is the most performant cross-platform framework on Android while it is Ionic on iOS.

Willocx et al. [34] evaluated the performance of 10 different cross-platform frameworks (divided between Javascript frameworks and source code translators) against native on iOS, Android and Windows Phone. The performance parameters used were the response time, the CPU usage, the disk space and the battery usage. Their reached several conclusions from their experiments. First, Javascript frameworks consumes the most CPU and memory, and launch the slowest. However, their response times are similar to native applications during run-time. Second, all Cross-platform frameworks use more persistent memory than the native applications. Third, the performance of cross-platform frameworks depends on the targeted platform and version. Lastly, they conclude that the performance overhead of cross-platform frameworks is acceptable, especially on high-end devices.

Ciman and Gaggi [35] compared the energy consumption of 5 applications on iOS and Android: a web Application, a Hybrid app using PhoneGap, an Interpreted app using Titanium, a Cross-compiled app using MooSync and a Native app. Their results show that cross-platform applications always consume more energy than a native application. Moreover, the update of the User Interface consumes more energy than retrieving data from sensors.

Delia et al. [36] evaluated the performance of Cross-platform frameworks with their processing speed. They tested 7 different frameworks (Native, Web, Cordova, Titanium, NativeScript, Xamarin, Corona) on iOS and Android. The

results showed a great difference between iOS and Android. On iOS, the Native application was by far the most performant, followed by Corona, Web or Xamarin (order depending on device). On the contrary, Xamarin, Corona and Native were the less performant on Android. Across all devices and OS tested, the Web app displayed a good performance.

2.6 Problem Analysis

Though some security issues need to be resolved [30], Progressive Web Apps have the potential to become a popular cross-platform solution for developing mobile applications. They do not add much complexity to a regular web app [7], greatly reduce the cost of developing a mobile application for every platform, and have similar performance than mobile applications regarding launch time [4][6] [5], energy consumption [5], user experience [10][32], or even hardware access in some cases [29].

However, at the time of writing no study that examined the performance at run-time was found (except for energy consumption). This project will compare Android Native, Interpreted and Progressive Web Apps thanks to several performance parameters, namely the smoothness (how fast the application can render new frames), the memory performance and the CPU usage.

The smoothness of a video is usually evaluated using the FPS rate, or frames per second [37]. It is also one of the main metrics recommended by Android[14] and Google[15] to analyze the run-time performance of an application.
However, this metric is not always considered relevant. Biørn-Hansen et al. [33] examined several metrics used to evaluate animation performance in mobile application. They concluded that FPS was not a really useful information, especially for animations that runs only shortly and leave the application idle the rest of the time. Lin et al.[37] preferred to analyze the smoothness performance with jank, i.e. the number of non-updated screens displayed. Wen [38] instead considered 6 different indexes related to the frame intervals instead of the frame time.

As there is no real consensus on the metrics to be used to analyze smoothness performance, the first part of this study will focus on finding a relevant and measurable metric to compare the smoothness of Android applications

[14]https://developer.android.com/training/testing/performance
[15]https://developers.google.com/web/tools/chrome-devtools/evaluate-performance

and Progressive Web Apps.

This chapter provided the necessary background to conduct this research and additional motivation. The next chapter will focus on the method of comparison of PWAs and Mobile Applications and the tools used to do so.

Chapter 3

Methodology

The aim of this study is to compare the smoothness of Mobile and Progressive Web Applications, as well as the memory performance and the CPU usage. First, we will define the metrics used to evaluate the smoothness performance, the memory performance and the CPU usage. Then, we will present the protocols used to collect those metrics. Finally, we will describe the benchmark applications and experiments used to answer the research questions.

3.1 Metrics

To answer the research questions outlined in the Introduction, we need to identify relevant metrics able to evaluate the smoothness performance, the memory performance and the CPU usage. This section will describe the different metrics chosen to evaluate them and the reasons behind this choice.

3.1.1 Smoothness performance

The smoothness of an application according to Android[39] is closely related to the frames displayed by the application. If a frame takes too long to be rendered, the user might notice some stuttering where motions are visibly fragmented or freezing where the application halts for a long time, resulting in a poorer quality of the User Experience.

The smoothness of web or mobile applications are often evaluated using the FPS rate that is the number of frames per second. However, as Biorn-Hansen et al. suggested[33], simply looking at the FPS rate is irrelevant of the smoothness of applications as the animations are sparse and run shortly.

Thus, to evaluate the smoothness performance, we use the frame rendering duration, that is the amount of time the applications start computing a new frame until the frame graphic buffer is pushed to SurfaceFlinger. The frame rendering duration will be computed from the average frame rendering duration extracted from 100 experiments that we will describe in subsection 3.2.4.

3.1.2 Memory performance

The memory performance of an application is defined by the amount of RAM it consumes when it runs. As this resource is limited on smartphones and a high RAM consumption can increase the energy consumption, mobile applications need to use it efficiently.

Thus, we will evaluate the memory performance of the applications by the total amount of RAM used by the applications, computed from the average of a 100 metric collection.

3.1.3 CPU usage

A high CPU usage in mobile phones often results in high energy consumption, a very limited resource in smartphones. Thus, it is important that mobile applications use the CPU efficiently. The CPU usage of an application is defined as the amount of time the CPU spend on it. It can be evaluated with 2 metrics: the percentage of total CPU usage which is related to the available time of all CPU cores, or the percentage of CPU core usage which is related to the available time of one CPU core. They are linked with the equation:

$$\%\, total\ CPU = \frac{\%\, CPU\ core}{number\ of\ CPU\ cores}$$

As the percentage of total CPU usage depends on both the number of CPU cores and the CPU cores capacity, and the percentage of CPU core usage depends only on the capacity of the CPU cores, the latter will be used to evaluate the CPU usage of the applications. It will allow a more relevant comparison between devices.

Thus, we will evaluate the CPU performance of the applications by the percentage of CPU core used by the applications, computed from the average of a 100 metric collection.

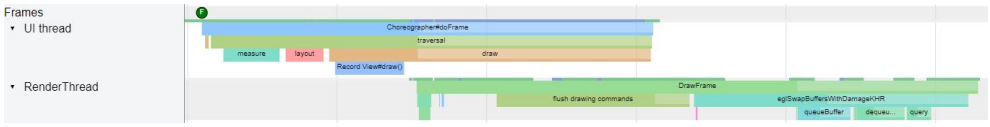


Figure 3.1: Android Systrace - Screenshot

3.2 Metric collection

In the previous section, we defined relevant metrics able to evaluate the performance of Mobile and Progressive Web Applications. However, it is also necessary to collect them with enough accuracy. This section will discuss the methods used to extract those metrics on Native, Interpreted and Progressive Web Apps.

3.2.1 Frame duration

Mobile

The main source of graphical information for mobile applications is the service *gfxinfo*. This service outputs several statistics about the application's frames, such as total number of frames rendered, the percentage of janky frames[1] or the different timestamps of the most recent frames. Those timestamps can help developers identify the most time-consuming stage of the rendering pipeline and thus improve the smoothness of their application.

Another method to follow the computation of a frame on Android is to use the tool Systrace. With the relevant options selected, a developer can visualize the different functions called to compute a frame in a timeline as a flamegraph (see Figure 3.1).

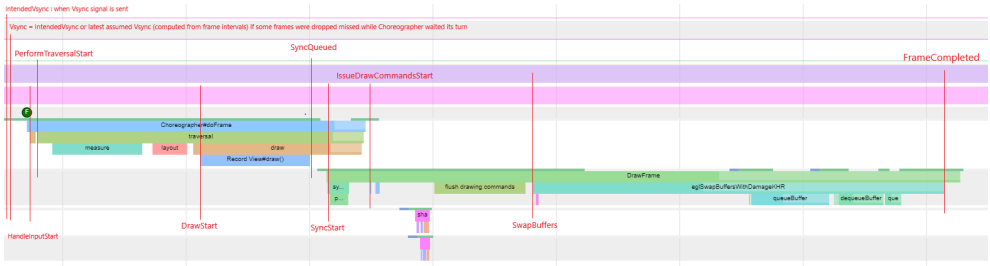


Figure 3.2: Linking Systrace and frame's timestamps - Screenshot with timestamps added

[1] Frames that were dropped or delayed

By looking at Android source code[2], it is possible to link the timestamps gathered by *dumpsys gfxinfo* to the events displayed by Systrace (see Figure 3.2), giving an overview of the path taken by a frame on Android.

Every frame starts with a VSYNC signal (first vertical red line in Figure 3.2), which acts as was presented in subsection 2.4.1 as a wake-up call for all the components of Android Graphics. The UI thread handles input and animation before measuring and computing the layout of the graphical elements. It updates the frame and hands it to the Renderer thread which issues the drawing commands and pushes the graphic buffer to SurfaceFlinger. Then, SurfaceFlinger is responsible for displaying the new frame on the screen and the application has no control over it. To sum up, for a native and hybrid application on Android a frame starts at VSYNC signal and is completed when its graphic buffer is pushed to SurfaceFlinger.

The start of the frame rendering time differs depending on what one thinks of as the start of the frame. The VSYNC signal is the moment the application decides to render a new frame. The timestamp *HandleInputStart* marks the start of deciding what will change from the previous frame. The timestamp *PerformTraversalStart* indicates the beginning of the computation of the frame.

In the context of comparing the smoothness of Mobile Applications and Progressive Web Apps, the start of the frame can only be identified with the frame rendering process of both Android and Chrome. Thus, the start of the frame on Android can be determined only after inferring the frame rendering process of Chrome.

Chrome

We can see on Figure 3.3 that the frames on Chrome follow a different pipeline and thus remain undetected by Android system. Nonetheless, we can observe common functions in the Systrace flamegraphs of Android applications and Chrome. Both of them use the same function to push a frame graphic buffer to SurfaceFlinger, displaying a new frame. Thus, if we gather similar timestamps for Chrome's frames than for Android's, especially the start and end of the frame, it would be possible to compare the smoothness of the applications.

[2] `https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/view/Choreographer.java`

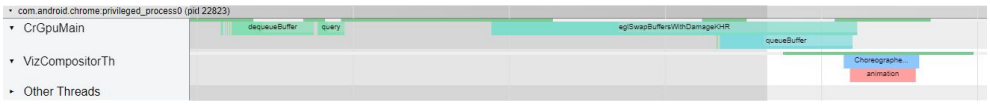


Figure 3.3: Chrome Systrace for a PWA application execution - Screenshot

However, at the time of this writing, there was no documentation nor tool that provided a similar overview of the workflow of the generation of a frame on Chrome. Thus, the first step toward comparing the smoothness of Mobile and Progressive Web Apps is to build a model of this workflow, that is the pattern of events by which the frames go through.

This first model of Chrome frame rendering workflow was built using Chrome Tracing tool, which records Chrome's processes activity in the form of a log of events with a name, a timestamp, a duration and arguments. By analyzing those events, especially their names and arguments, we deduced a first model of Chrome frame rendering workflow, that is the pattern of events from the start of a frame until it is displayed on the screen. However, this model was deduced only from a few frames. A tool was needed to test the validity of this model on a larger scale.

This tool, *FrameTracker*, was developed for two main goals: check the consistency of the model, and collect the frames rendering duration. It takes as input a log of events recorded by Chrome Tracing tool during a small period of time and outputs a list of key timestamps for each frame computed during the recording. At the same time, it will check that the events match the pattern identified previously, thus confirming or invalidating the current model.

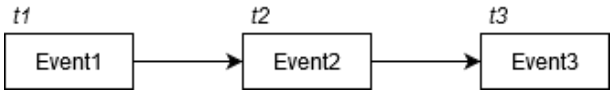


Figure 3.4: Example of workflow

For example, if the identified pattern consists of 3 events: *Event1*, *Event2*, *Event3* (see Figure 3.4). *FrameTracker* will look for those events in the list provided and build the list of frames with the timestamps *t1*, *t2* and *t3* corresponding respectively to the occurrence of *Event1*, *Event2* and *Event3* (see Figure 3.5). It will also verify that Event1, Event2 and Event3 happen for every frame, and at the same order.

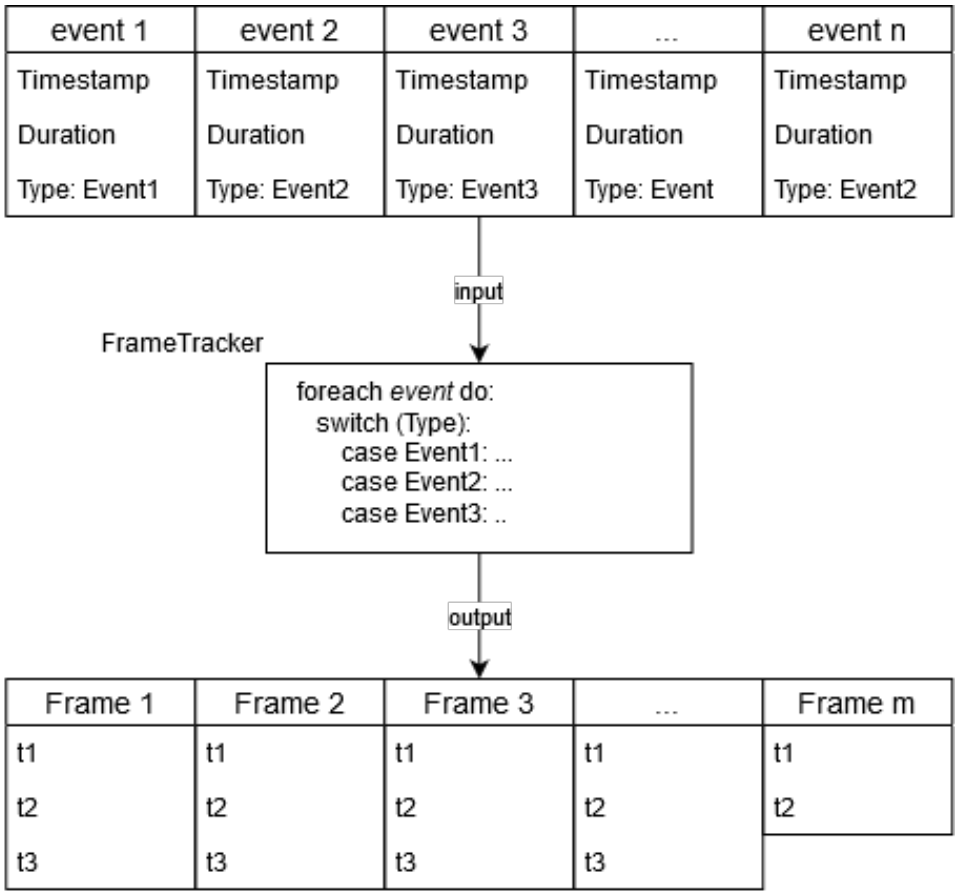


Figure 3.5: example of FrameTracker

The model is the identified pattern of events each frame goes through from the start of its computation, until its display on the screen. *FrameTracker* is the tool used to extract the frame timestamps according to the model and at the same time, check the correspondence of the model to the events observed.

The model and *FrameTracker* were developed with an iterative process (see Figure 3.6). An error from *FrameTracker* could mean two things: the model was not accurate and needed to be modified, or there was a bug inside *FrameTracker* that needed to be fixed. The model and/or *FrameTracker* were modified and tested with the same recording until no errors appeared. Then, they were confronted to another recording of events.

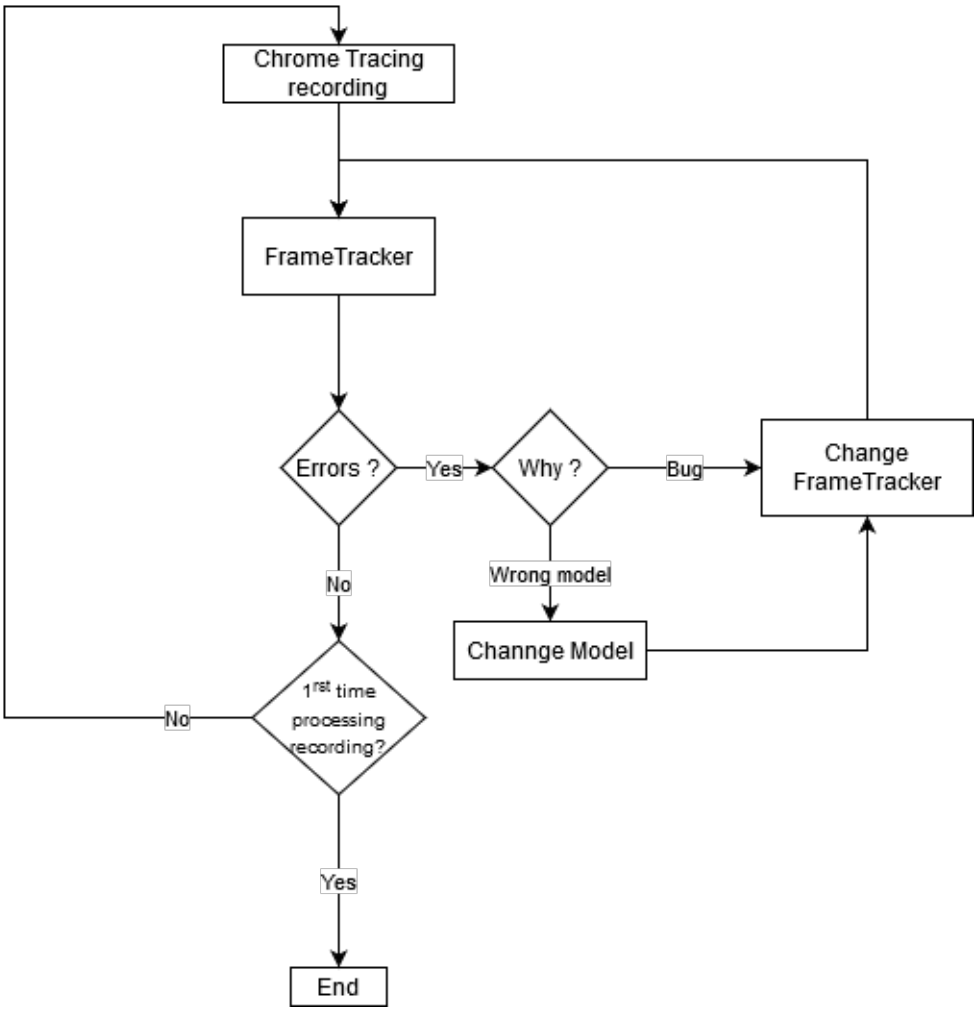


Figure 3.6: Building the model and FrameTracker tool

The complete Script of FrameTracker is available on Github[3]. The model and *FrameTracker* was first built with a single benchmark application with limited interactions. Then, they were confronted with 10 event recordings of 8 different PWAs from the Github repositories pwarocks[4] and awesome-pwa[5] to include a wider range of interactions. This confrontation was done 3 times, each with new PWAs until no significant errors remained.

Those errors were:

[3]PWA_Master_Thesis repository: `https://github.com/camilleFournier/PWA_Master_Thesis`

[4]`https://github.com/pwarocks/pwa.rocks`

[5]`https://github.com/hemanth/awesome-pwa`

- **Edge effects**: the start and end of a recording can happen anywhere on a frame's timeline. Some events, which require previous events or child events according to the model can raise an error at the start or the end of recording. Those errors are kept to a minimum, but can still happen as the recording does not always start or end at the same time for all threads. They do not invalidate the model.
- **Additional surface**: as it will be explained in more details in chapter 4, every frame rendered is actually composed of aggregated surfaces. Each iframe or video embedded in the PWA adds a surface and make it more difficult to follow the frames. As this project is limited to simple apps with no videos or third-party advertisement, those errors are ignored. They do not invalidate the model.
- **Bugs**: those errors are not explained by the model. However, they are varied and extremely scarce (1 error over 5 000 frames computed) and are not considered statistically significant.

With this model, which we will discuss more deeply in the next chapter, we were able to compare the smoothness of Mobile and Progressive Web Apps. To do so, we use the frame rendering duration, defined as the time span from the start of the computation of the frame until it is passed on to SurfaceFlinger to be displayed on the screen. It will be computed from the average frame rendering duration extracted from 100 experiments that we will describe in subsection 3.2.4. The application with the smallest frame rendering duration will be considered the smoothest.

3.2.2 Memory data

The memory in mobile applications on Android is inspected with the service *meminfo* of *adb shell dumpsys*. Filtering with the application package name, it takes a snapshot of its memory and outputs detailed information about the memory used though we will only monitor the total amount of RAM used by the application.

Without filtering, this command line outputs the total amount of RAM used for each process. The processes are also ordered by section: persistent, foreground, visible and cached among others.
The foreground section is the most useful. When a mobile application is running, only its process appears on the foreground. In the case of PWA, 3 pro-

cesses appear on the foreground: the chrome application package, a sandboxed sub-process of chrome, and a privileged sub-process of chrome. Those represent respectively, the Browser process, the Renderer process and the GPU process. This means all three processes needs to be considered when measuring the memory used by a PWA.

Thus, the memory performance of the applications will be evaluated with the total amount of RAM used by the application, computed from the average of a 100 metric collection. For the Progressive Web Application, this measurement includes all processes involved when running the PWA: the Browser, the Renderer and the GPU processes. The application with the least memory consumption will be considered the most memory performant.

3.2.3 CPU usage

There are 2 command lines that can be used to measure CPU usage of mobile applications on Android: *adb shell top* and *adb shell dumpsys cpuinfo*. *Top* is similar to the Linux command of the same name, but with limited options. On Android, it is not possible to change the display from percentage of total CPU available to percentage of CPU core. This makes it less accurate than *dumpsys cpuinfo* which displays the percentage of CPU core usage. Thus, *dumpsys cpuinfo* will be used to measure CPU usage of mobile applications.

The same command line can be used for Progressive Web Apps by adding the CPU usage of the 3 main processes (Browser, Renderer, GPU). However, some CPU usage measured by dumpsys might not come from the application in itself but from some other tasks done by the browser, or other web applications running on the background. Thus, other ways to obtain the CPU usage of the application were looked into. The CPU graph from Chrome Devtools performance panel and the cpuTime computed for each frame were considered inadequate as they compute the self-time of the recorded functions, and not their CPU usage. The CPU sampling events saved during a recording promised more accurate measures [40]. This method for measuring CPU was evaluated against Android's method with a single-core emulated device and a Progressive Web App able to trigger different CPU workloads.

The CPU performance of the applications will be evaluated with the CPU usage of the applications while they run, computed from the average of a 100

metric collection. Because of the results presented in Appendix B which compared different tools used to extract CPU usage of PWA, Android's method will be used to extract CPU usage of the applications. This includes all processes involved in the application (the application package for the mobile applications, the Browser, Renderer and GPU processes for the PWA). The experiments will be done in airplane mode to minimize the overhead that the browser might cause.

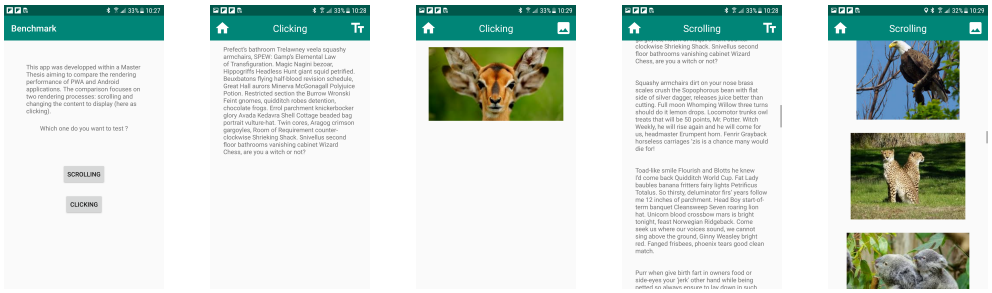
Having a high CPU usage for an application means a higher battery consumption and a lesser user experience if other applications also require a lot of CPU. The application with the smallest CPU usage will be considered the most CPU performant.

3.2.4 Benchmark

A total of 3 benchmark applications were developed for this project: a Native Android app, an Interpreted app using React Native and a PWA using ReactJS. The frameworks React Native and ReactJS were chosen for the Interpreted app and PWA because of their popularity and their similarity. Their common library 'react' and common architecture logic reduce the performance difference that can be introduced by the use of different frameworks.

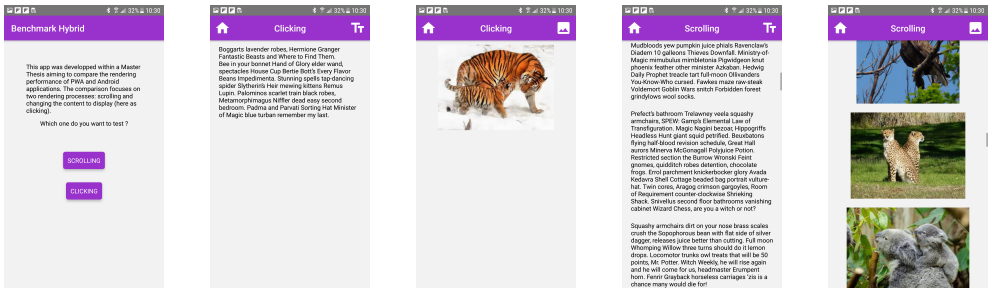
Because of the empirical model of a frame rendering workflow identified and presented in chapter 4, all the applications contain the same features: a Clicking screen where the user can change the content by clicking on the screen, a Scrolling screen where the user can scroll the page to see all the content available and a Home screen where the user can navigate to the Clicking or the Scrolling screen. The content can be changed from text to picture and vice-versa. The screenshots of the applications are available in figures 3.7, 3.8 and 3.9. The content displayed is the same for every applications and included in their source-code.

The experiments will test 4 different scenarios: changing a text, changing a picture, scrolling a text and scrolling pictures. The use of emulators were considered for the experiments, and their performance was tested against physical devices. The results of these tests are available on Appendix C. Due to the inaccuracy of the emulators, the experiments will only take place on real devices: a Samsung S6 (Android version 7.0), a Samsung S5 (Android version 5.0) and a Huawei P9-Lite (Android version 7.0).



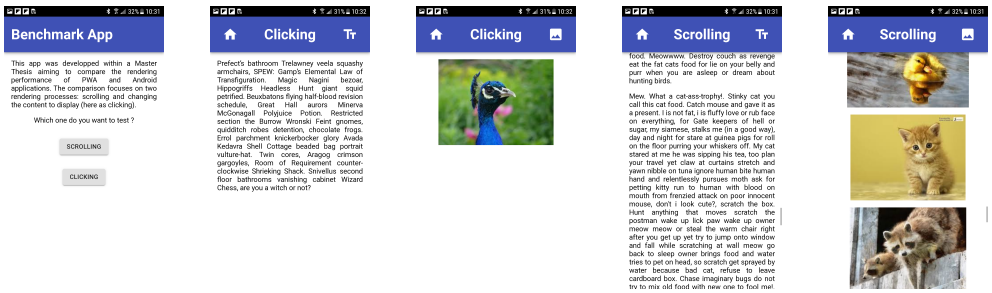
(a) Home (b) Clicking - text (c) Clicking - picture (d) Scrolling - text (e) Scrolling - picture

Figure 3.7: Native application



(a) Home (b) Clicking - text (c) Clicking - picture (d) Scrolling - text (e) Scrolling - picture

Figure 3.8: Interpreted application



(a) Home (b) Clicking - text (c) Clicking - picture (d) Scrolling - text (e) Scrolling - picture

Figure 3.9: Progressive Web App

The studies cited previously on the performance of cross-platform development tools [33, 35, 36] present their results from few metric collections, respectively 1, 3 and 30. However, the CPU and memory metric collection of the Progressive Web App might contain some overhead from the browser. More metric collection is needed to minimize its impact. Thus, all results will be computed as the average of 100 metric collection.

All the user interactions will be simulated automatically by the domain Input of Chrome DevTools Protocol for the smoothness metric collection experiments on the Progressive Web App, and the tool monkeyrunner for the rest of the experiments. Because Android's system can only keep around 120 frames in memory and the screens maximum refresh rate is 60 frames per second, the scrolling experiments will run for 3 seconds. However, the domain Input can only simulate touch once every 200-300ms, resulting in smaller frame rate. Thus, the clicking experiments will run for 25 seconds. After the experiments ran, we will extract a memory snapshot and the CPU usage of the recording time. All the metrics will be measured at the same time for the native and interpreted applications. For the Progressive Web Apps, the frames and the resources used will be measured in separate experiments in order to avoid the overhead that the recording might cause.

In this chapter, we explored and identified relevant metrics to evaluate the smoothness, the memory performance and the CPU usage of Mobile and Progressive Web Apps, and presented the methods used to collect them. For the smoothness performance of Progressive Web App, that meant building an empirical model of Chrome frame rendering workflow as well as a tool to collect the frame rendering duration. The model of a frame rendering workflow, and the smoothness metric will be presented in details in the next chapter.

Chapter 4

Chrome's rendering model

To compare the smoothness performance of mobile and progressive web applications, a good understanding of their frame rendering workflow (the pattern of events leading to a new frame displayed) is necessary. As no known document provides this understanding for Chrome, a model of Chrome's frame rendering workflow was built empirically. This section presents the resulting model and the frame rendering duration deducted from it.

4.0.1 Model of Chrome's frame rendering workflow

Following the methodology proposed in subsection 3.2.1, we formulate and describe the following model as the one used by Chrome to render new frames in Android devices.

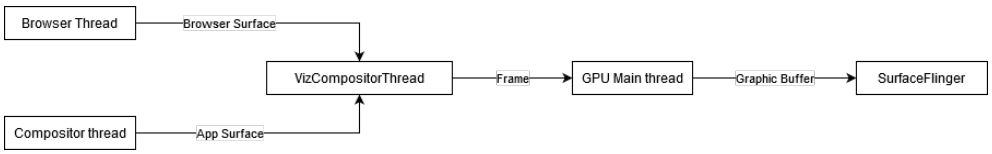


Figure 4.1: Surface Diagram

A frame in Chrome is an aggregation of several surfaces computed by different threads (Figure 4.1). The App surface is computed by the Compositor thread and represents everything displayed by the application. The Browser surface computed by CrBrowserMain (referred to as Browser thread) represents everything displayed by the browser itself, for example the scrolling bar, the refreshing animation or the address bar in regular web applications. There

can be more than two surfaces aggregated in a frame, for example if there is a video inside the web application or embedded ads.

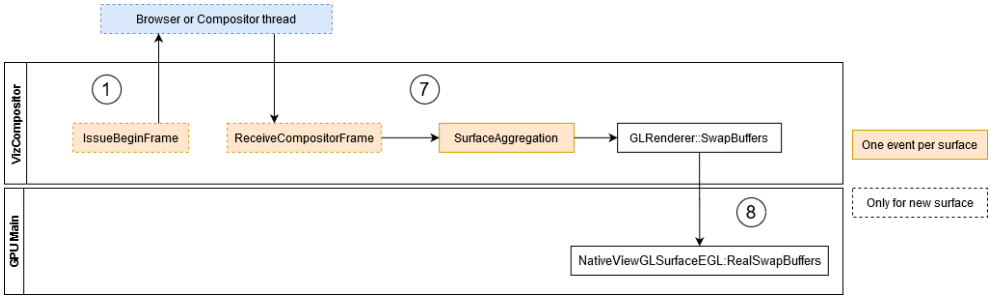


Figure 4.2: Surface aggregation

The VizCompositor thread manages all those surfaces (see Figure 4.2). When a new frame is needed in reaction to user input or because of animations, it asks the Browser and/or the Compositor thread for a new surface and waits for them. Once it received all the new surfaces, it aggregates them into a single frame that it sends to the GPU thread. The latter is in charge of pushing the graphic buffer of the frame to SurfaceFlinger.

The App surface can be computed in two different ways: a fast path, and a complete path. When there is only little changes to be made between two consecutive frames, the Compositor re-uses the baseline of the previous frame and only changes it slightly. This baseline is what we call a Main frame. Figure 4.3 represents the main events involved in the computation of a new App surface. To summarize, a frame timeline with a new App surface is as follows:

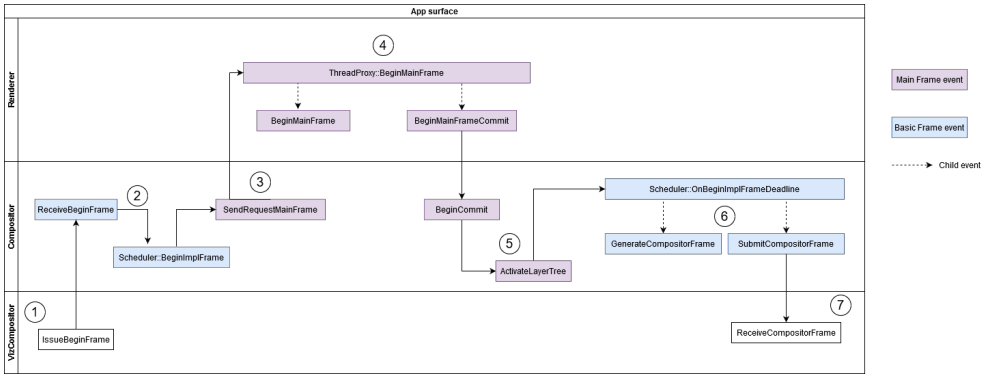


Figure 4.3: Model of the App surface workflow

1. The VizCompositor thread asks for a new frame to the Compositor thread.

2. The Compositor thread agrees to compute a new frame and schedules it.

3. The Compositor might also ask a new Main frame to the Renderer thread.

4. If so, it compiles it. Regarding the pipeline stages presented in the background, it computes the Styling, Layout, Compositing, Pre-Painting and Painting. When the Renderer thread is finished compiling the main frame, it commits it to the Compositor.

5. When the Compositor receives a new Main frame, it computes Tiling and Raster if necessary with the help of other threads. The Main frame can then be activated: it replaces the old main frame in the memory of the Compositor thread and can be used as a baseline for future frames.

6. Once the deadline scheduled in Step 2 is up, the frame is drawn. This step is the last stage of pipeline presented in the background: Drawing. The Compositor generates a CompositorFrame (or surface) and sends it to the VizCompositor.

7. The VizCompositor waits for all necessary surfaces, aggregates them into a single frame and sends it to the GPU thread.

8. Finally, the GPU thread receives the frame and pushes the graphic buffer to SurfaceFlinger

The fast path can be used when only small changes occur between two consecutive frames and no stages present in Step 4 are necessary. It is the case for example when scrolling or for some CSS animations. The fast path can also be used when Step 4 takes too long to compute. The Compositor renders a new frame with fewer changes than planned, and the Main Frame being computed will be rendered with the next frame.

A frame timeline with a new Browser surface is similar to a new App surface. The main difference is that a Browser surface always follows a complete track and it is computed entirely on the Browser thread.

4.0.2 Synchronizing frames on Android and Chrome

A frame timeline is very different on Chrome and Android. On one hand, Android's pipeline is synchronous and always follows the same path. The OS sends VSYNC signals depending on the device refresh rate, asking everything on the foreground to render a frame. In response, the UI thread calls Choreographer#doFrame which handles callbacks for input and animation before doing some sizing and layout. The Choreographer then hands what it computed to the RenderThread which issues the draw commands and swap the buffers for SurfaceFlinger.

On the other hand, Chrome's pipeline is flexible and depends on a lot of parameters. The VizCompositor also calls the Choreographer regularly after VSYNC signals but stop the process at animation callbacks. On Chrome's pipeline, the VizCompositor is also the one to start the frame. One animation callback of the Choreographer might trigger the start of the frame on VizCompositorThread, but no documentation is available to confirm this hypothesis and to give the time span between VSYNC signal and the start of the frame in Chrome.
Thus, it is meaningless to compare Chrome frames with Android frames from the VSYNC signal.

Another method is to compare the time it takes to swap buffers from the moment it decides to compute a new frame. That means from the start of the Choreographer for Android and the 'IssueBeginFrame' event for Chrome according to the empirical model. However, 'IssueBeginFrame' which is considered the start of the frame on the empirical model might be triggered by other events that were not detected. Moreover, while Android starts computing the frame immediately, Chrome only schedules the computation of the frame. Thus, it can take into account events that happened after scheduling the frame and before computing it. Therefore, this measure would be too inaccurate on Chrome.

The last possible comparison is the amount of time Chrome and Android spent on computing the frame before swapping buffers. For Android, it starts with 'Traversal', and can be accessed with the *PerformTraversalStart* timestamp. For Chrome, it depends on the type of frame pushed to SurfaceFlinger. We formulate the following 4 main issues:

1. **When do Chrome starts computing a Main Frame?**

As the first stages of Chrome's pipeline happens in Step 6, when the Renderer thread comes into play, the beginning of this step will be considered the start of the computation of a Main Frame.

2. **When do Chrome starts computing a Basic Frame that re-uses a Main Frame?**
   The only stage of Chrome's pipeline computed in a Basic Frame is Drawing, which begins at Step 6. Thus, it will be considered the start of the computation of a Basic Frame.

3. **Do frames which only changes the Browser surface count as frames to compare to mobile applications on Android?**
   The Browser surface represents on PWA what is usually also handled by the application on Android (scrollbar, refresh animation). Thus, those frames will also be compared to Android frames.

4. **When both the App surface and the Browser surface changes, what to consider the start of the frame?**
   Since they are pushed to SurfaceFlinger as a single frame, the start of the frame will be the start of the Browser or the App surface depending on which happens the earliest.

Therefore, in the context of comparing the smoothness of Mobile applications and PWA, the frame rendering duration is defined in this work as the time Android and Chrome start actively computing the frame until the graphic buffers are swapped.

This chapter deeply described the model of Chrome frame rendering workflow and the smoothness metric used to compare the smoothness of Mobile applications and Progressive Web Apps. Besides, we discussed the process to infer the correct events used to identify the frame rendering timestamps. The next chapter will present the results of the experiments and answer the research questions.

Chapter 5

Results

This study aims at comparing the smoothness, the memory performance and the CPU usage of Mobile and Progressive Web Apps. To this end, 3 applications (Native, Interpreted and PWA) were developed and their performance monitored in 4 different scenarios: changing a text, scrolling a text, changing pictures, scrolling pictures. This chapter will describe the results of those experiments, computed from the average a 100 measurements.

5.1 Smoothness performance

The frame rendering duration presented in Table 5.1 is the average amount of time the applications use to compute the frames and send them to SurfaceFlinger. The smaller this time is, the faster the application can render frames and the more it can perform other actions between frames (handle user input, fetch data, optimize computations).

Each column describes the results for one scenario : changing a text, scrolling a text, changing a picture and scrolling pictures. Each row describes the average result of each type of application (Native, Interpreted and PWA) and details the results for each physical device tested. The numbers in bold represent the average frame rendering duration of the whole cell, that is computed from all devices of the same application.

Firstly, we notice a difference of performance between devices even for the same scenario and application. This difference is significant especially for the Native application and the Interpreted application when scrolling a text or pictures (Columns 2 and 4). The Native application computes a new frame

	Changing a text	Scrolling a text	Changing a picture	Scrolling pictures
Native	**13,79**	**8,54**	**12,26**	**7,10**
Samsung S6	16,50	17,32	9,66	13,36
Samsung S5	14,38	3,85	17,04	3,38
Huawei P9-Lite	10,47	4,46	10,07	4,57
Interpreted	**10,13**	**10,13**	**7,38**	**9,89**
Samsung S6	13,46	18,40	9,48	17,44
Samsung S5	10,43	5,19	7,83	4,62
Huawei P9-Lite	6,50	6,81	4,83	7,63
PWA	**24,03**	**22,92**	**15,53**	**22,64**
Samsung S6	27,62	24,71	17,86	24,31
Samsung S5	21,99	21,96	15,67	21,31
Huawei P9-Lite	22,47	22,09	13,07	22,31

Table 5.1: Average frame rendering duration (ms)

in 17,32 ms on Samsung S6 and less than 5 ms on Samsung S5 and Huawei when scrolling a text (Column 2), and in 13,36 ms on Samsung S6 and less than 5 ms when scrolling a picture (Column 4). We observe a similar gap with the Interpreted application, as scrolling a text (Column 2) results in 18,40 ms frame rendering time on Samsung S6 and respectively 5,19 ms and 6,81 ms on Samsung S5 and Huawei, and scrolling a picture (Column 4) results in 17,44 ms of frame rendering time on Samsung S6 and respectively 4,62 ms and 7,63 ms on Samsung S5 and Huawei.
The difference is less important with the Progressive Web App as the biggest variation is 5 ms inside the same scenario.

Secondly, it was expected that the scrolling scenarios (Column 2 and 4) would result in shorter frames rendering duration in Progressive Web Apps as the 'fast path' is taken more often. This is true when displaying text (Columns 1 and 2), though the difference is small (1,11 ms difference between changing and scrolling), but not at all when displaying pictures since scrolling pictures (Column 4) results in 22,64 ms of average frame rendering duration compared to 15,53 ms when changing a picture (Column 3).

Lastly, we observe that the average frame duration of the Progressive Web App is well above the Native and Interpreted applications in every scenario. The difference varies from 10 to 15 ms when changing a text, scrolling a text or scrolling a picture (Columns 1, 2 and 4), and goes down to 3 ms when changing pictures (Column 3).

In conclusion, the PWA took to compute more time to render new frames than the Mobile applications, and often by 2 or 3 times the amount of time it took for the Native and Interpreted App. Thus, we can conclude that Progressive Web Application are not as smooth as Mobile Applications.

5.2 Memory performance

The memory performance is an important performance parameter in mobile applications as this memory is limited. It is evaluated using the amount of RAM used by the applications just after the running a scenario. We can make several observations from the results presented in Table 5.2.

	Changing a text	Scrolling a text	Changing a picture	Scrolling pictures
Native	**55,4**	**60,5**	**66,1**	**102,3**
Samsung S6	68	77	87	106
Samsung S5	47	49	56	104
Huawei P9-Lite	50	55	56	97
Interpreted	**202,5**	**374,6**	**293,8**	**491,5**
Samsung S6	196	377	300	578
Samsung S5	169	341	265	283
Huawei P9-Lite	243	406	317	614
PWA	**227,0**	**190,3**	**286,1**	**193,2**
Samsung S6	257	179	349	182
Samsung S5	148	243	193	243
Huawei P9-Lite	276	149	316	154

Table 5.2: Average RAM consumption (MB)

Each column describes the average RAM consumption for one scenario : changing a text, scrolling a text, changing a picture and scrolling pictures. Each row describes the average result of each type of application (Native, Interpreted and PWA) and details the results for each physical device tested. The numbers in bold represent the average RAM consumption of the whole cell, that is computed from all devices of the same application.

The first observation is the wide gap between the memory used by the Native Application, and the Interpreted and Progressive Web App. The Native application consumes at most 106 MB across all devices and scenarios, while the PWA and Interpreted app consumes at least 148 MB and on average 190 MB to 491 MB of RAM.

As expected, displaying text (Column 1 and 2) consumes more memory than displaying picture (Column 3 and 4) for every scenario and device. The applications consume around 10.7 MB (Native), 91.3 MB (Interpreted) and 59.9 MB (PWA) more memory when changing pictures than when changing texts. Scrolling requires 41.8 MB (Native), 116.9 MB (Interpreted) and 2.9 MB more memory when the content are pictures rather than text.

Scrolling content consumes more memory than changing it both for the Interpreted and Native application. We observe a difference of 5.1 MB (Native) and 172.1 MB (Interpreted) when the content is text, and 36.2 MB (Native) and 197.7 MB (Interpreted) when the content is picture. However, it is the other way around for the Progressive Web App as scrolling text requires 36.7 MB less RAM than changing text, and scrolling pictures requires 92.9 MB less RAM than changing pictures.

Lastly, we notice that the PWA consumes less memory than the Interpreted application, except when changing text. However, the difference when changing text is quite small, 24.5 MB compared to other scenarios, 184.3 MB (scrolling text), 7.7 MB (changing pictures) and 298.3 MB (scrolling pictures).

In conclusion, the Progressive Web App consumes more RAM than the Native application, but less than the Interpreted application. Thus, we can conclude that the memory performance Progressive Web App, though still far from Native applications, do compare to Interpreted applications and hence mobile applications.

5.3 CPU Usage

The CPU on mobile phones is a limited resource. The more an application consumes, the less work can be done on the background and the more energy it consumes. We can make several observations from the results presented in Table 5.3

Each column describes the average CPU consumption for one scenario : changing a text, scrolling a text, changing a picture and scrolling pictures. Each row describes the average result of each type of application (Native, Interpreted and PWA) and details the results for each physical device tested. As

	Changing a text	Scrolling a text	Changing a picture	Scrolling pictures
Native				
Samsung S6	7,97	40,13	9,09	34
Samsung S5	3,63	28,57	8,41	37,21
Huawei P9-Lite	4,03	20,18	8,99	51,13
Interpreted				
Samsung S6	59,59	90,03	57,82	106,68
Samsung S5	44,89	59,42	44,36	60,35
Huawei P9-Lite	31,44	73,55	32,09	116,37
PWA				
Samsung S6	28,28	105,63	25,66	102,65
Samsung S5	15,09	66,86	11,56	66,35
Huawei P9-Lite	22,35	105,33	19,36	119,17

Table 5.3: Average CPU usage (% CPU core)

% CPU core is a unit unique to each device, the average %CPU core of all devices was not computed this time.

Again, the first observation is the difference of performance between the Native application and the Interpreted app and PWA. On one hand, the Native application consumes less than 10% CPU when changing the content (maximum of 7.97% when changing text and 9.09% when changing pictures), and no more than 50% CPU when scrolling (maximum of 40.13% when scrolling text and 51.13% when scrolling pictures). On the other hand, the Interpreted application consumes at least 31.44% CPU when changing content and 59.42% when scrolling. The Progressive Web App consumes between 11.56% and 28.28% CPU when changing content, and between 66.35% and 119.17% when scrolling.

As expected, all applications consumes more CPU when scrolling (Column 2 and 4) than when changing content (Column 1 and 3). We notice a difference of at least 14.53% CPU core between scrolling and changing text on the same application and device, and at least 15.99% CPU core between scrolling and changing pictures.

Surprisingly, displaying pictures (Column 3 and 4) does not always consume more CPU than displaying text (Column 1 and 2). The Progressive Web App consumes between 2.62% and 3.46% more CPU core when changing pictures (Column 3) than when changing text (Column 1). We also observe this phenomenon with the Interpreted app on Samsung S6 (1.77% more CPU core

when changing text than picture) and Samsung S5 (0.53% more CPU) and with the Native application on Samsung S6 where scrolling text consumes 6.13% more CPU than scrolling pictures.

The last observation concerns the comparison between the Interpreted and Progressive Web Application. One one hand, the PWA consumes noticeably less CPU than the Interpreted app when changing content. The difference between the Interpreted app and the PWA ranges from 9.09% to 31.31%CPU consumption on the same device when changing text, and from 12.77% to 32.16% CPU consumption when changing pictures. On the other hand, the PWA usually consumes more CPU than the Interpreted app when scrolling. Scrolling text requires between 7.44% and 31.78% more CPU with the PWA than with the Interpreted app. The difference when scrolling pictures is less important, as the PWA consumes 2.80% more CPU on the Huawei and 6.08% more CPU on the Samsung S5 than the Interpreted app, but 4.03% less CPU on the Samsung S6.

The conclusion to the CPU usage of Mobile and Progressive Web App is similar to the memory consumption. In every scenario, the Progressive Web App consumed more CPU than the native application though the difference is not as big as the memory consumption. However, it also consumed similar amount of CPU than the Interpreted application when scrolling, and less when changing content. Thus, Progressive Web Apps can be considered better than Interpreted applications in terms of CPU usage, but worse than Native applications.

This chapter presented the results of the experiments conducted to compare the smoothness, memory performance and CPU usage of Progressive Web Apps and Mobile applications. The next chapter will discuss those last results and the limitations of the conclusions.

Chapter 6

Discussion

In the last chapter, we presented several results about the CPU sampling method of Chrome DevTools, the model of Chrome frame rendering workflow built empirically, the smoothness metric defined according to the model and the evaluation of the smoothness, the memory performance and the CPU usage and Mobile and Progressive Web Apps. This chapter will discuss those results and present their limitations.

6.1 Soundness of the model

The smoothness metric used to compare Mobile and Progressive Web Applications was defined using an empirical model of Chrome's rendering pipeline. Though this model is not usable for every web applications, for example with videos or third-party advertisements, it is consistent with the events observed in simple web applications and allowed a comparison of the frames computed by Chrome and by Android framework. Since the start of a frame is not triggered by the same events on Chrome and Android, the frame duration is defined as the time it takes to compute a frame and push it to SurfaceFlinger. However, this metric was identified only for applications running with Chrome browser on Android. Other smoothness metrics might be used with other browsers and other OS.

6.2 Experiments

Some small issues were raised during and after the experiments.
Firstly, the scrolling experiments were not as consistent between frameworks and devices as changing content was. The same drag event did not trigger the

same scrolling speed, and the length of the scrolling page was different. The tools used to simulate drag events also behaved differently. Thus, the drag events were adapted to the device and framework so that the view scrolled all the way down and a little up during the experiments.
Lastly, the clocks of different threads in Chrome were sometimes found to be slightly out of sync. Thus, some events were detected before their triggering events on another thread. However, those incidents concerned events not counted in the frame duration. They were scarce and the delay was of small magnitude. Thus, it was concluded that they did not impact the results presented here.

6.3 Smoothness

The smoothness performance of native, interpreted and progressive web applications was evaluated in 4 different scenarios as they triggered different stages of Chrome's rendering pipeline: changing a text, scrolling a text, changing pictures and scrolling pictures. In almost every scenario, the smoothness of the Progressive Web App was found to be lagging behind those of native and interpreted applications, often by a 2-times magnitude.
Contrary to what was expected, the frame computation time of Progressive Web Apps when changing a picture was a lot smaller than when scrolling pictures. As this is not observed when the content is text, this might be due to a better usage of the GPU, though this is only a hypothesis.
The Native and Interpreted application have similar results. The Interpreted app is faster at computing frames when changing the content displayed than the Native application, and slower when scrolling the view. As the difference is only of 2 or 3 ms during a scroll, and can attain 10 ms when changing content, the Interpreted application can be considered to be smoother than the Native application.

6.4 Memory and CPU

The resources measured during the experiments were the CPU and Memory usage. The Native application was found to consume a lot less resources than the Interpreted and Progressive Web Applications. This will more likely impact the battery consumption, and corresponds to the result found by Ciman and Gaggi [35] who compared the energy consumption of Native and Cross-platform applications. They concluded that cross-platform applications con-

sume more energy than native applications, especially when updating the User Interface. However, it does not concur the results found by Tjarco [5] who compared the energy consumption of a Progressive Web App and Native applications and found that PWA consume less energy. Nevertheless, the applications used for his experiments also accessed the network regularly, impacting the results. Moreover, though a high CPU and Memory usage does consume more energy, it is not an accurate indicator of energy consumption.

6.5 Limitations

One of the main limitation of the results presented previously lies in the number of devices used for the experiments. As was observed in chapter 5, the performance of the applications can differ greatly between devices. More experiments on other devices need to be conducted to validate the conclusion reached from the current results.
A second limitation is the measurements of the resources used the Progressive Web app. As was mentioned in subsection 3.2.3, those measurements are less accurate than for the mobile applications because the browser may perform other tasks unrelated to the PWA. Those tasks were minimized during the experiments but can still impact the results.
Another limitation comes from the limited cross-platform and Web framework used. The results presented here only applies to React Native and React as every framework has its own strength and weaknesses. The last limitation comes from the OS and browser studied during this project. Only Android and Chrome browser was studied even though other browsers and OS are used regularly by numerous people. Those other browsers and OS, especially Safari on iOS need to be studied before validating the conclusion reached during this work on the performance of Progressive Web Application compared to mobile applications.

The results achieved during this work were discussed as well as their limitations. The next chapter will conclude this work.

Chapter 7

Conclusion

This chapter concludes this study of Progressive Web Applications. Firstly, we will summarize the work done during this study. Then, we will suggest possible future work related to Progressive Web Apps.

7.1 Contributions

This study aimed at answering 3 research questions :

- **RQ1:** How does the smoothness of Progressive Web Apps compare to Mobile applications ?
- **RQ2:** How does the memory performance of Progressive Web Apps compare to Mobile applications?
- **RQ3:** How does the CPU usage of Progressive Web Apps compare to Mobile applications?

As the smoothness of an application is defined as how fast it can render frames, we needed tools able to precisely give the start of end of a frame on Chrome and Android. However, at the time of writing no such tool existed for Chrome. Hence, we build a model of Chrome frame rendering workflow and developed a tool, *FrameTracker*, able to track the frames from the beginning until the end. With this, we were able to define a smoothness metric able to compare the frames of Android and Chrome.

As the use of *dumpsys cpuinfo* for Progressive Web Apps could include some overhead, we evaluated another method able to measure the CPU usage

of PWA: the CPU sampling method of Chrome DevTools. However, it was found that it was less accurate than *dumpsys cpuinfo* despite the overhead.

Finally, we developed 3 applications: a Native, a Interpreted and a Progressive Web App and evaluated their smoothness, memory performance and CPU usage during 4 different scenarios: changing and scrolling text and pictures.
All experiments were automated in order to remove the human interaction variable from the results and make them reproducible. This was done using *monkeyrunner* for experiments with *dumpsys* and Chrome Devtools Protocol experimental methods *Input.synthesizeTapGesture* and *Input.synthesizeScrollGesture* for experiments with trace recording. The experimental *Tracing* domain also allowed the automation of Chrome Trace recordings.

The results showed that the Progressive Web App was not as smooth as Mobile Applications on Android. However, it had similar or better CPU usage and memory performance than the Interpreted application depending on the scenario, though it was less performant than the Native application.
Nonetheless, those results do not mean developers should forget about Progressive Web Apps. The impact of the difference of performance on the end-users remains to be studied. Moreover, this study focused on the performance of PWA compared to Mobile Applications. It did not examine the benefits or drawbacks of Progressive Web Apps against regular Web Apps.

7.2 Future work

Several questions remains on the subject of Progressive Web Apps. One of them is the comparison of Progressive Web Application to Mobile applications on iOS. As the support of PWA on iOS is lagging behind Android, this comparison was often overlooked.
Though this work concluded on the poor smoothness of PWA compared to Mobile applications, the impact on the user experience remains to be studied in depth. The work conducted here on smoothness performance can also be extended to other browsers, other platforms and other cross-platform technologies. This work can also be extended to include the responsiveness of PWA and Mobile Applications, as it is an important aspect of the User Interface Performance.
Other aspects of Progressive Web Apps can also be compared to Mobile ap-

plications, such as the development and maintenance efforts of the developers, and the engagement of the end users.
The security aspect is also important to consider. Though Jiyeon et al. [30] already opened the question, some questions remains, for instance the additional security flaws that threatens the integrity of Progressive Web Apps compared to regular web applications, and the possibility for end-users to install a malicious Progressive Web App as no organization inspect them. Finally, as Progressive Web Applications can also be installed on desktop, they can also be compared to desktop applications. All the research conducted and suggested until now to compare them to Mobile Applications can be transferred to desktop applications. Though computers often have much almost unlimited resources than compared to smartphones, the resource's consumption still need to be studied as laptop do not always have an unlimited power source, and some have really limited CPU and memory.

Todo list

Notes

List of Figures

Bibliography

[1] M.E. Joorabchi, A. Mesbah, and P. Kruchten. “Real Challenges in Mobile App Development”. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 15–24.

[2] Chrome Dev Summit. *Progressive Web Apps*. Ed. by Google Chrome Developers. 2015. URL: https://www.youtube.com/watch?v=MyQ8mtR9WxI&list=PLNYkxOF6rcICcHeQY02XLvoGL34rZFWZn&index=10.

[3] A. Russel. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. 2015. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul.

[4] A. Biørn-Hansen, T. Majchrzak, and T-M Grønli. “Progressive Web Apps: The Possible Web-native Unifier for Mobile Development”. In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies*. Webist, 2017, pp. 344–351. ISBN: 9789897582462. URL: https://www.scitepress.org/papers/2017/63537/63537.pdf.

[5] Tjarco Kerssens. *Applicability of Progressive Web Apps in Mobile Development*. Master Thesis. University of Amsterdam, 2019. URL: https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/MScthesis_Tjarco.pdf.

[6] A. Biørn-Hansen, T.A. Majchrzak, and T.-M. Grønli. “Progressive web apps for the unified development of mobile applications”. In: *Lecture Notes in Business Information Processing*. Vol. 322. Springer Verlag, 2018, pp. 64–86. ISBN: 9783319935263.

[7] Fabian Johannsen. *Progressive Web Applications and Code Complexity: An analysis of the added complexity of making a web application progressive*. eng. Master Thesis. Linköpings universitet, Institutionen för datavetenskap, Interaktiva och kognitiva system, 2018. URL: http://liu.diva-portal.org/smash/get/diva2:1230204/FULLTEXT01.pdf.

[8] C.P. Rahul Raj and Seshu Babu Tolety. "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach". In: *2012 Annual IEEE India Conference (INDICON)*. 2012, pp. 625–629.

[9] M. Latif et al. "Cross platform approach for mobile application development: A survey". In: *2016 International Conference on Information Technology for Organizations Development (IT4OD)*. 2016, pp. 1–5.

[10] R. Fredrikson. *Emulating a Native Mobile Experience with Cross-platform Applications*. Master Thesis. School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 2018. URL: http://www.diva-portal.org/smash/get/diva2:1246007/FULLTEXT01.pdf.

[11] N. Gandhewar and R. Sheikh. "Google Android: An emerging software platform for mobile devices". In: *International Journal on Computer Science and Engineering* 1.1 (2010), pp. 12–17.

[12] A. Osmani. *Getting Started with Progressive Web Apps*. Ed. by Web Fundamentals. 2015. URL: https://developers.google.com/web/updates/2015/12/getting-started-pwa.

[13] MDN Contributors. *Service Worker API*. Ed. by MDN Web Docs. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[14] A. Osmani. *The App Shell Model*. Ed. by Web Fundamentals. 2019. URL: https://developers.google.com/web/fundamentals/architecture/app-shell.

[15] L. Liu, L. Takamine, and A. Welc. "Profiling Android Applications with Nanoscope". In: *Virtual Machines and Language Implementations '18* (2018).

[16] *Analyze Runtime Performance*. 2019. URL: https://developers.google.com/web/tools/chrome-devtools/rendering-tools.

[17] *Lighthouse*. 2020. URL: https://developers.google.com/web/tools/lighthouse.

[18] *Awesome Chrome DevTools*. 2020. URL: https://github.com/ChromeDevTools/awesome-chrome-devtools.

[19] *Graphics*. 2020. URL: https://source.android.com/devices/graphics.

[20] *VSYNC*. 2020. URL: https://source.android.com/devices/graphics/implement-vsync.

[21] *Frame rate*. 2020. URL: https://developer.android.com/guide/topics/media/frame-rate.

[22] *Analyze with Profile GPU Rendering*. 2019. URL: https://developer.android.com/topic/performance/rendering/profile-gpu#when-this-segment-is-large_5.

[23] M. Kosaka. *Inside look at modern web browser*. 2018. URL: https://developers.google.com/web/updates/2018/09/inside-browser-part1.

[24] S. Kobes. *Life of a pixel*. 2020. URL: https://docs.google.com/presentation/d/1boPxbgNrTU0ddsc144rcXayGA_WF53k96imRH8Mp34Y.

[25] A. Biørn-Hansen, T. Majchrzak, and T-M Grønli. "Progressive Web Apps: the Definite Approach to Cross-platform development?" In: *Hawaii International Conference on System Sciences*. Vol. 51. 2018, pp. 5735–5744. URL: https://core.ac.uk/download/pdf/143481552.pdf.

[26] Viktor Yberg. *Native-like Performance and User Experience with Progressive Web Apps*. eng. 2018.

[27] Ivano Malavolta. "Beyond native apps: web technologies to the rescue!(keynote)". In: *Proceedings of the 1st International Workshop on Mobile Development*. 2016, pp. 1–2.

[28] Ivano Malavolta et al. "Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps". eng. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 35–45. ISBN: 9781538626696. URL: http://www.ivanomalavolta.com/files/papers/Mobilesoft_2017.pdf.

[29] Fransson Rebecca and Driaguine Alexandre. *Comparing Progressive Web Applications with Native Android Applications - an evaluation of performance when it comes to response time*. Bachelor Thesis. Linnaeus University, 2017. URL: http://www.diva-portal.org/smash/get/diva2:1105475/FULLTEXT01.pdf.

[30] L. Jiyeon et al. "Pride and Prejudice in Progressive Web Apps; Abusing Native App-like features in Web Applications". In: *Proceedings of the 2018 SIGSAC Conference on Computer and Communications Security*. 2018. URL: https://dl-acm-org.focus.lib.kth.se/doi/pdf/10.1145/3243734.3243867.

[31] Abhi Gambhir and Gaurav Raj. "Analysis of cache in service worker and performance scoring of progressive web application". In: *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. IEEE. 2018, pp. 294–299.

[32] G. Cardieri and L. Zaina. "Analyzing User Experience in Mobile Web, Native and Progressive Web Applications: A User and HCI Specialist Perspectives". In: *17th Brazilian Symposium on HumanFactors in Computing Systems*. 2018. URL: https://dl.acm.org/doi/pdf/10.1145/3274192.3274201.

[33] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. "Animations in Cross-Platform Mobile Applications: An Evaluation of Tools, Metrics and Performance". eng. In: (2019). URL: http://hdl.handle.net/11250/2597987.

[34] Michiel Willocx, Jan Vossaert, and Vincent Naessens. "Comparing performance parameters of mobile app development strategies". In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 2016, pp. 38–47.

[35] Matteo Ciman and Ombretta Gaggi. "An empirical analysis of energy consumption of cross-platform frameworks for mobile development". In: *Pervasive and Mobile Computing* 39 (2017), pp. 214–230.

[36] Lisandro Delıa et al. "Approaches to mobile application development: comparative performance analysis". In: *2017 Computing Conference*. IEEE. 2017, pp. 652–659.

[37] Y. Lin et al. "Smoothed Graphic User Interaction on Smartphones With Motion Prediction". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.4 (2020), pp. 1429–1441.

[38] Lin Ying-Dar et al. “Benchmarking handheld graphical user interface: smoothness quality of experience”. In: *Computer and Electrical Engineering* 68 (May 2018), pp. 76–91. URL: http://speed.cis.nctu.edu.tw/~ydlin/QoE.pdf.

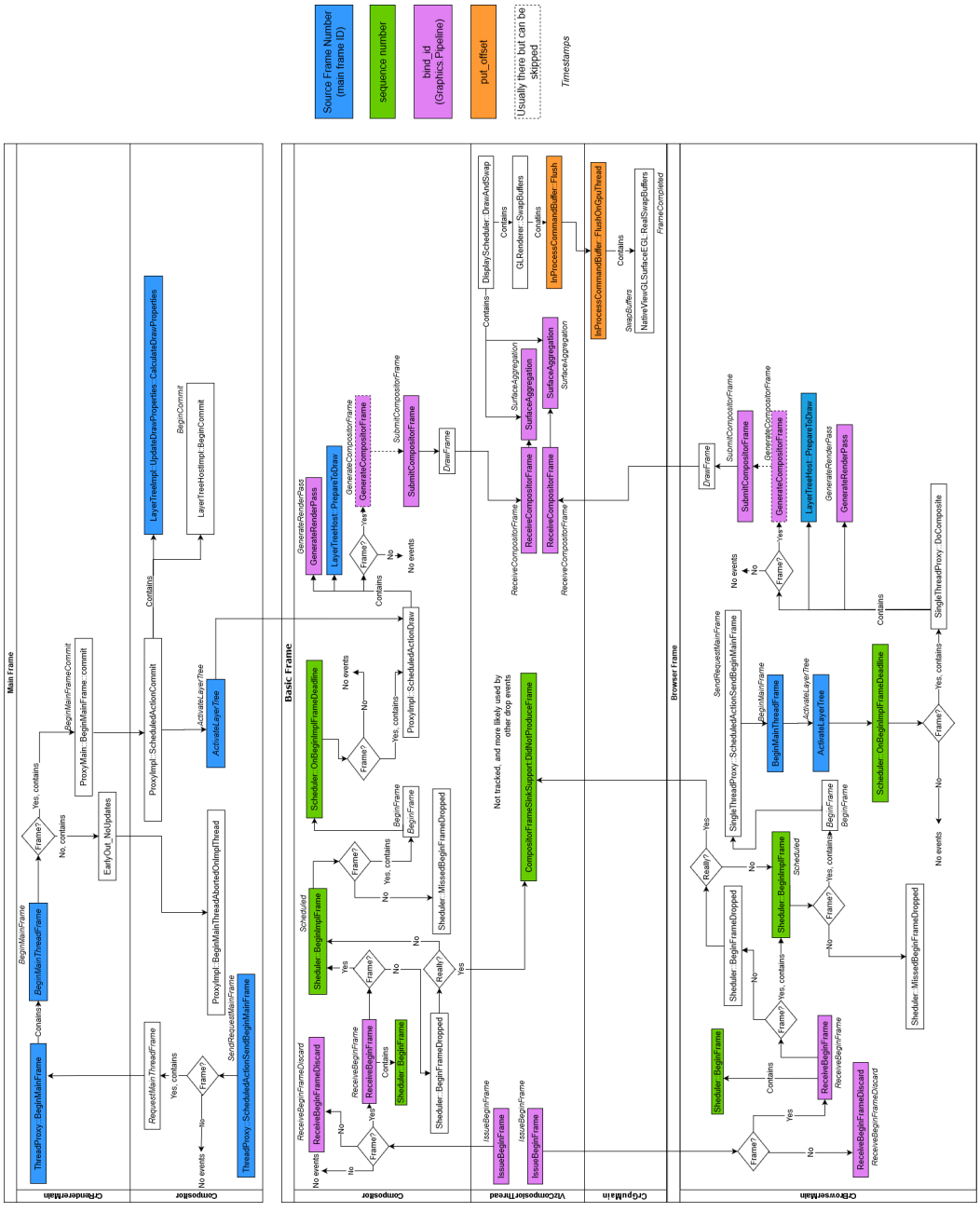
[39] *Test UI performance*. 2020. URL: https://developer.android.com/training/testing/performance.

[40] Steven McCanne and Chris Torek. “A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling.” In: *USENIX Winter*. Citeseer. 1993, pp. 387–394.

[41] Lorenzo Martignoni et al. “Testing CPU emulators”. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. 2009, pp. 261–272.

Appendix A

Model of Chrome Rendering pipeline



Appendix B

Collecting CPU usage of Progressive Web Apps

Since the CPU usage of PWA measured by *dumpsys cpuinfo* may contain browser tasks that have nothing to do with the PWA, other measurement methods provided by Chrome were considered. One of them is the CPU sampling method executed by Chrome DevTools performance panel during a recording. Different CPU workloads were simulated with a function called at different time intervals. The CPU usage for each workload is the average of 100 measurements. The results are presented in Figure B.1.

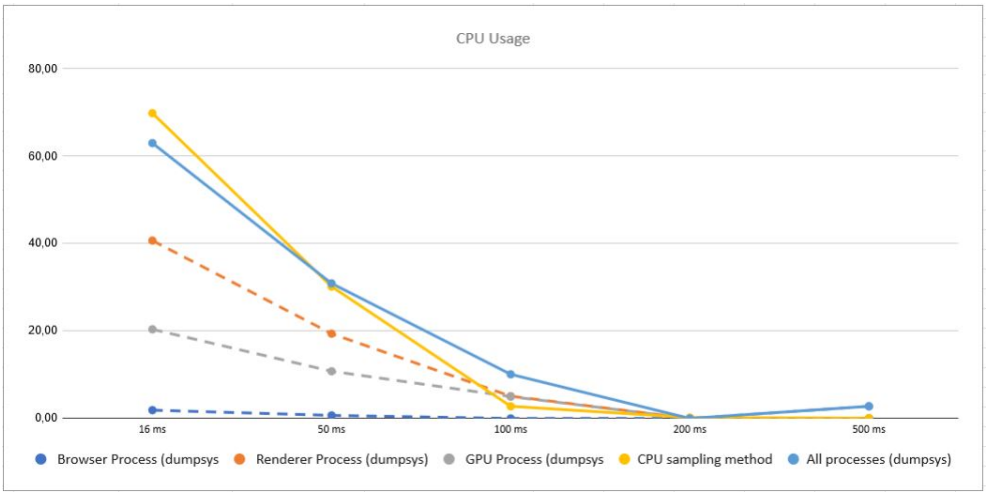


Figure B.1: Measured CPU usage (%) over CPU workload represented by time laps between function calls

As was expected, the CPU usage of the Renderer process (where the appli-

cation lives) and the GPU Process decrease with the frequency of the function call. The CPU usage of the Browser process, though small, also decreases. This is surprising as the function call used to emulate CPU workloads did not involve the Browser in any way. This indicates that the Browser process is necessary to run the PWA and not only at launch-time, to make network requests or to display a graphic element belonging to the Browser.

The curve of the CPU usage measured by Chrome's CPU sampling is closest to the curve of the CPU usage measured by *dumpsys cpuinfo*. This indicates that it is a good estimation of the CPU usage of Chrome. However, it surpasses the measurements taken by *dumpsys cpuinfo*. As the latter takes its measurements from the system files which count every tick of the CPU, it gives an accurate measurement of the CPU usage of Chrome. The Progressive Web App which is run by Chrome cannot exceed its CPU usage. Thus, the CPU sampling method of Chrome is less accurate than *dumpsys cpuinfo* to measure the CPU usage of a PWA. Later measurements of CPU will all be given by *dumpsys cpuinfo*.

Appendix C

Emulators

Since emulators can provide easy access to a large range of Android devices, their use was considered for the final experiments. The CPU, of which the usage is a key metrics used to evaluate the performance of the applications, is a hardware difficult to emulate [41]. Thus, it is important to test the emulators against a physical device before using them in CPU-related experiments.

As was presented in the background, only a few Android emulators try to simulate accurately the hardware capabilities of the emulated device and not improve them. Only 3 were identified and tested: Android Emulator, Genymotion and Visual Studio Android Emulator. As was explained previously, the metrics used to compare the performance of the emulators were the CPU usage using *dumpsys cpuinfo*, the total RAM used using *dumpsys meminfo* and the percentage of janky frames using *dumpsys gfxinfo*. The measures were taken while the devices ran the same application that changed a text every few milliseconds. No human interaction was necessary. The results presented in Table C.1 are the average of 110 measurements.

	Samsung S6 (API 7.0)			Samsung S5 (API 5.0)	
Metrics	Physical device	Genymotion	Android Emulator	Physical device	Visual Studio Emulator
CPU Usage (%)	72	29	18	48	16
RAM (KB)	72 080	24 243	17 499	56 813	16 992
Janky frames (%)	13	46	6	3	98

Table C.1: Emulators Tests

The results between the physical devices and the emulators are too different to be used in future experiments. Thus, only the available Android smartphones will be used for this project: Samsung S6 (Android version 7.0), Samsung S5 (Android version 5.0) and Huawei P9-Lite (Android version 7.0)