Rendering Time Analysis of In-Flight Entertainment Systems

Hugo Nilsson and Andreas Ruggieri

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF ENGINEERING LTH | LUND UNIVERSITY

MASTER THESIS 2026

ISSN 1650-2884
LU-CS-EX: 2026-44

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Rendering Time Analysis of In-Flight Entertainment Systems

Analys av Renderingstider för In-Flight Entertainment-system

June 18, 2026

LU-CS-EX: 2026-44

Hugo Nilsson and Andreas Ruggieri

LUND
UNIVERSITY

Rendering Time Analysis of In-Flight Entertainment Systems

Performance study of Android UI rendering times using the Android View system, Jetpack Compose, Glide, and Coil, in an In-Flight Entertainment context

Master's thesis work carried out at
the Department of Computer Science, Lund University

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Examiner: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Abstract

This thesis examines rendering performance in the context of Android-based in-flight entertainment (IFE) systems in collaboration with a company that specializes in software for such systems. Modern IFE systems are often required to run on resource-constrained hardware due to certification requirements and costly installation processes, resulting in long hardware lifecycles in the aviation industry. This adds to the importance of efficient rendering performance, as low loading times are a crucial aspect for the user experience.

To conduct the study, a prototype application was developed and benchmarked on real IFE hardware. Two Android UI frameworks, the traditional Android View system and Jetpack Compose, were compared in combination with two image loading libraries, Glide and Coil. The benchmarks consisted of rendering alternating views designed for this study while measuring rendering times. For each render, two metrics were collected using the Android Macrobenchmark library: Time to Initial Display (TTID) and Time to Full Display (TTFD).

The results showed that the image-loading library had the greatest impact on rendering performance. Configurations using Glide consistently outperformed those using Coil, especially during repeated rendering of image-heavy views, where caching had a larger impact on performance. However, additional experiments indicated that Coil performance could be improved by optimizing the prototype code. The differences in performance between the traditional Android View system and Jetpack Compose were significantly smaller, although the Android View system achieved slightly lower rendering times in the majority of benchmarks.

The study concludes that optimized image loading is crucial for reducing rendering times, while the choice of UI framework appears to have a smaller overall impact.

Keywords: Rendering time, Android, UI frameworks, Image loading libraries, In-flight entertainment system

Sammanfattning

Detta examensarbete undersöker renderingsprestanda i Android-baserade in-flight entertainment-system (IFE-system) i samarbete med ett företag som utvecklar mjukvara för sådana system. Moderna IFE-system behöver ofta köra på resursbegränsad hårdvara till följd av hårda certifieringskrav och höga installationskostnader som leder till långa livscykler för hårdvara inom flygindustrin. Detta ökar behovet av effektiv renderingsprestanda, då korta laddningstider är viktiga för en god användarupplevelse.

För att genomföra studien utvecklades en prototyp som prestandatestades på riktig IFE-hårdvara. Två UI-ramverk för Android, det traditionella Android View-systemet och Jetpack Compose, jämfördes i kombination med två bildladdningsbibliotek, Glide och Coil. Testerna utfördes genom att rendera ett antal vyer som utformats specifikt för denna studie samtidigt som renderingstiden mättes. För varje rendering registrerades två mått med hjälp av Androids Macrobenchmarkbibliotek: Time to Initial Display (TTID) och Time to Full Display (TTFD).

Resultaten visade att bildladdningsbiblioteket hade störst påverkan på renderingsprestanda. Konfigurationer som använde Glide var konsekvent snabbare än de som använde Coil, särskilt vid upprepad rendering av bildrika vyer där cachning hade större påverkan på prestanda. Ytterligare experiment indikerade dock att prestandan för Coil kunde förbättras genom optimeringar i prototypkoden. Skillnaderna i prestanda mellan det traditionella Android View-systemet och Jetpack Compose var betydligt mindre, även om Androids View-system uppvisade något kortare renderingstider i majoriteten av testerna.

Studien drar slutsatsen att optimerad bildladdning är avgörande för att minska renderingstider, medan valet av UI-ramverk verkar ha en mindre påverkan totalt sett.

Nyckelord: Renderingstid, Android, UI-ramverk, Bildladdningsbibliotek, In-flight entertainment system

Acknowledgements

We want to thank the case company for letting us write our thesis together with them, as well as granting us access to the office, testing equipment, and delicious breakfast. We are also very grateful to all the employees at the case company for being very welcoming, and for being there for us when we needed assistance.

A special thank you goes out to our supervisor at the case company, for helping us with the technical setup and getting us started with the project. His many valuable suggestions throughout the process have been of great help, and we highly appreciate our thoughts being challenged, which helped us think things through an extra time and come up with ideas we otherwise would have missed.

Lastly, we want to say an extra thank you to Jonas Skeppstedt, our supervisor at LTH, for his continuous guidance and insights during the entirety of the thesis process. His feedback and support has been invaluable, and we always left our meetings feeling confident and full of ideas.

Lund, Spring 2026
Hugo Nilsson and Andreas Ruggieri

Table of contents

1. Introduction 13
   1.1 Research questions 13
   1.2 Thesis disposition 14
   1.3 Division of work 14
2. Background 15
   2.1 IFE systems 15
      2.1.1 Aviation industry 15
      2.1.2 Hardware limitations 16
         2.1.2.1 CPU 16
   2.2 Programming environment 16
      2.2.1 Android Studio 17
      2.2.2 Kotlin 17
      2.2.3 Macrobenchmarks 17
      2.2.4 Android UI Frameworks 17
         2.2.4.1 Android View System 18
         2.2.4.2 Jetpack Compose 19
      2.2.5 Image-loading libraries 20
         2.2.5.1 Glide 21
         2.2.5.2 Coil 21
   2.3 Related work 21
   2.4 System and test environment of case company 22
3. Method 23
   3.1 Process overview 23
      3.1.1 Preparation 23
      3.1.2 Prototyping 24
   3.2 Benchmark design 25
      3.2.1 Scope 25
      3.2.2 Prototype design 26
         3.2.2.1 Benchmark configurations 27
      3.2.3 Metrics 28

3.2.4 Testing device 29
3.3 Benchmark iterations 29
4. Results 31
4.1 One view 31
4.1.1 Time to Initial Display 31
4.1.2 Time to Full Display 32
4.2 Alternating views 34
4.2.1 Time to Initial Display 34
4.2.2 Time to Full Display 34
4.3 Results per view 36
5. Discussion 39
5.1 UI frameworks 39
5.2 Image-loading libraries 40
5.3 View layout 40
5.4 Threats to validity 41
5.5 Future work 43
6. Conclusions 45
References 48
A. Rendered views 51
B. Benchmark data 54
B.1 One view 54
B.2 Alternating views 56
B.3 Results per view 58

1 Introduction

In-flight entertainment systems (IFE) are becoming increasingly common on commercial airliners today and play a major role in the passenger experience on longer flights. These systems offer passengers a range of opportunities, such as movies, games, flight maps, and other information about the flight. This project is conducted in collaboration with a company, later referred to as the case company, that specializes in developing software for such systems. During a previous project at this company, it was found that rendering time during navigation is a key factor for user experience in IFE systems. However, when testing the software, it was found that navigation between content-heavy views takes a significant amount of time on production hardware. The purpose of this report is to examine the current system architecture, analyze how view rendering is currently implemented, and compare the current approach to other alternatives.

As part of this investigation, we take into account the technical limitations of the screens used for IFE. The installation of new systems in an aircraft is a complicated and expensive process, which means that hardware is typically left installed for extended periods and thus becomes outdated compared to the advanced capabilities of modern tablets. This makes program optimization more complicated since modern software needs to run smoothly on outdated hardware, which must be taken into account when investigating software solutions. Therefore, it is not guaranteed that a program that runs smoothly on a modern computer or tablet has equivalent performance in an IFE context.

1.1 Research questions

In this thesis, the following research questions are examined and discussed:

- What methods could be used to reduce the rendering time when navigating within the IFE system, taking into account the system's limitations?
- How do the different methods perform compared to each other, and what are their advantages and disadvantages?

1.2 Thesis disposition

Chapter 2 presents the technical and contextual background necessary to understand the study, including programming environments, frameworks, and an overview of hardware limitations and the reasons behind them.

Chapter 3 describes the work process we used to carry out the research and produce the data that we used to answer the research questions.

Chapter 4 presents the results of the software benchmarks and analyzes the results, attempting to identify relevant trends and patterns. After this, the findings and suggested directions for further research are discussed in Chapter 5.

Finally, Chapter 6 concludes the thesis by building on the results and analysis to answer the research questions and summarize the main findings.

1.3 Division of work

The work was divided evenly throughout the thesis, although at times the authors have focused on different parts to make the work progress more efficiently.

Both students have participated in the literature search, but the majority has been done by Andreas. This was especially the case in the beginning, while Hugo focused more on the prototype and outlined the design of both the prototype and the views to be rendered. Hugo was responsible for developing the prototype using the UI framework Jetpack Compose, whereas Andreas implemented the prototype that is based on the traditional Android View system. Andreas was also the one integrating the image-loading libraries into each configuration.

Hugo handled the majority of the work related to the setup of the IFE screen and the problems encountered during benchmarking on the device. The benchmarking code was written by Hugo, whereas the actual benchmark process was conducted by both Hugo and Andreas. Plots and graphs were made by Hugo.

Concerning the report, work was divided such that Andreas focused primarily on the background section, whereas Hugo focused on writing the method section. The rest of the report was divided equally.

2 Background

This chapter presents the knowledge necessary to understand the study. It begins with a more in-depth explanation of the IFE context, outlining the capabilities and challenges of the systems. This is followed by an introduction to the programming languages, tools, and frameworks utilized during the project. Lastly, previous work relevant to this report is discussed, and a brief overview of the system used as a reference for this thesis is provided.

2.1 IFE systems

IFE systems are embedded systems deployed on commercial aircraft that help enhance the passenger flight experience. The history of the most basic IFE systems dates back almost 100 years, making it a market that has undergone substantial development. Over time, a considerable number of different IFE systems have existed, and various forms of such systems still exist today. This study focuses on seatback IFE devices, where a display unit is integrated into each passenger seat. These units act as embedded clients connected through an onboard network to a central content server. The server stores and distributes media content, while the seatback units are responsible for decoding, rendering, and presenting the user interface to the passenger.

2.1.1 Aviation industry

The aviation industry is a safety-critical industry with many regulatory requirements that must be strictly complied with. As a result, the process of installing and upgrading existing hardware and software is complex and cumbersome. There are many requirements and certifications that must be granted before electronics can be installed in an aircraft. For example, there exists a certification that determines the flammability requirements for all cabin materials, and there are standards that must be met before any electronic equipment is considered reliable for use in an aircraft. The testing process to meet the standards consists of multiple stages and, like most verification activities, is iterative in nature [17]. Consequently, achieving full compliance with the standard takes time, meaning that by the time electronic equipment is installed on a commercial aircraft, more advanced devices with improved performance may already be available on the market.

Another reason why the hardware of an IFE-system is not upgraded very often is that any upgrade typically requires retrofitting the aircraft, which is associated with high costs for airlines. Upgrading seatback systems averages 500,000 to 1.5 million USD per aircraft, and the operation can result in up to two weeks of downtime, which means loss of potential income [23].

2.1.2 Hardware limitations

Many IFE systems are subject to hardware limitations that significantly impact system design and overall performance. Although resource-constrained devices are typically defined as systems intentionally designed with limited processing power and memory, IFE devices differ in that their limitations arise primarily from the long lifecycle requirements and certification constraints mentioned above. Consequently, IFE systems are frequently constrained by hardware that remains in service for many years, leading to restricted CPU performance, memory capacity, and overall system resources compared to modern consumer devices.

2.1.2.1 CPU

A central processing unit (CPU) is the "brain" of a computer, responsible for executing instructions from software, performing calculations, and managing data flow within a computer. The CPU executes instructions continuously, and its operating speed, measured in hertz, determines how many instructions it can process per second. This means that a lower clock frequency results in a processor that can perform fewer instructions per second. However, CPU performance does not depend solely on the clock frequency. Factors such as processor architecture, the number of cores, and hardware optimization also affect performance. Older devices often have less powerful CPUs with lower processing capabilities and fewer architectural optimizations compared to the newer processors integrated into modern devices. Consequently, computationally intensive tasks, such as rendering complex user interfaces, may require more time to execute, which can lead to increased rendering times and reduced UI responsiveness.[21].

2.2 Programming environment

This section outlines the programming environment used in this study. It provides an overview of Android development and describes the programming language and tools employed when developing applications for the Android platform.

2.2.1 Android Studio

Android Studio is the official IDE for developing Android applications. The IDE provides a range of integrated tools that facilitate the development process, including a feature-rich emulator that replicates real devices and enables testing without physical hardware. Android Studio also includes built-in performance profiling tools that allow developers to monitor running applications and collect data such as CPU usage, memory consumption, and memory leaks.

2.2.2 Kotlin

Kotlin is a programming language that is widely used when building Android applications. Today, more than 60% of Android developers use it, and since 2019, Google has considered it the preferred language for Android development. Previously, Java was the standard programming language for building Android applications, but in later years, Kotlin has grown in popularity. Two advantages of Kotlin compared to Java are its built-in null safety and its reduced need for boilerplate code. Kotlin also offers strong interoperability with Java, making it possible for developers to integrate Kotlin into existing Java codebases and migrate projects incrementally. Within Android development, two different UI frameworks are commonly used with Kotlin, both of which will be presented later in this chapter.

2.2.3 Macrobenchmarks

The Macrobenchmark library is used to measure the performance of end-to-end user flows in Android applications, such as app startup, navigation between screens, scrolling, and other UI interactions. Android Studio provides integration with the library, allowing developers to generate a Macrobenchmark test module using a built-in template that requires only minimal setup and automatically configures the required test environment.

After creating a Macrobenchmark module, user behavior can be simulated through scripted interactions, allowing the benchmark to drive the application the same way a real user would. As a result, performance metrics can be gathered in a controlled and repeatable environment, ensuring consistent and comparable measurements [5].

2.2.4 Android UI Frameworks

UI frameworks assist developers in defining, rendering, and managing graphical interfaces by providing reusable components and automating how data changes update the screen. Historically, Android applications have relied primarily on the traditional View system with XML-based layouts. More recently, Google introduced Jetpack

Compose, which is a UI toolkit designed to simplify development and reduce boilerplate code. The following sections describe these two approaches in more detail.

2.2.4.1 Android View System

The traditional Android View system uses XML layouts, and it has been the standard method for building Android user interfaces since 2008. It still remains the foundation for many Android applications. The layout files are responsible solely for defining the structure of the user interface and do not contain any application logic. Instead, the application's logic is implemented in separate Kotlin or Java files, depending on the programming language used in the project.

In XML-based layouts, the user interface is represented as a hierarchical tree, where each component specifies its relationship to the other components within the structure. The hierarchical structure is made up of Views, commonly known as widgets, and ViewGroup objects, where ViewGroup objects act as containers that define the layout and positioning of child views. Common examples of views include elements of the user interface, such as buttons and text fields [9].

It is during runtime that the view objects are created in memory, through a process known in Android as layout inflation. This process involves parsing the XML code and mapping the hierarchical structure defined in XML into a tree of View and ViewGroup objects that later can be manipulated and rendered on the screen [15]. Ghielens et al. identify this runtime inflation step as a contributor to performance issues in XML-based Android layouts, noting that deeper or more complex hierarchies increase the size of the resulting view tree and therefore require more nodes to be traversed, which consumes additional CPU time[19].

Listing 2.1 Example of a simple user interface defined in XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Listing 2.2 Kotlin Activity demonstrating UI handling with XML Layout

```kotlin
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Inflates the XML layout and sets it as the activity's UI
        setContentView(R.layout.activity_main)

        // References to UI components defined in the XML layout
        val textView = findViewById<TextView>(R.id.text)
        val button = findViewById<Button>(R.id.button)

        // Updates the TextView
        textView.text = "TextView updated from Kotlin"

        // Sets a click listener on the button
        button.setOnClickListener {
            textView.text = "Button pressed"
        }
    }
}
```

Listing 2.1 defines a simple user interface consisting of a text field and a button. Each widget is assigned a unique identifier through the `android:id` attribute. During layout inflation, these identifiers are compiled into an auto-generated `R` class, where each ID becomes a constant under `R.id`. Listing 2.2 illustrates the corresponding Kotlin code, which inflates the XML layout, accesses the widgets using the generated identifiers before updating the text field and adding a click listener to the button.

2.2.4.2 Jetpack Compose

Jetpack Compose, commonly referred to as Compose in the Android community, is a modern user interface toolkit released by Google in 2021. In Compose, the UI is defined using composables, which are functions that describe different parts of the interface. When the state of the application changes, Compose uses a process called recomposition, where only the composables affected by the state change are updated and redrawn. By avoiding the need to traverse large view hierarchies, Compose can reduce some of the overhead associated with the traditional Android View system [11].

Jetpack Compose can be integrated into existing view-based projects through dedicated interoperability APIs, allowing developers to adopt Compose incrementally. According to the official Android developer documentation, Jetpack Compose is recommended for new projects because it reduces boilerplate code and can accelerate the development process [10] [6].

Listing 2.3 Example of a simple user interface built with Jetpack Compose

```kotlin
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Sets the activity content using a Composable instead of XML
        setContent {
            MainScreen()
        }
    }
}

@Composable
fun MainScreen() {
    // State that holds the current text
    var text by remember { mutableStateOf("Hello, I am a TextView") }

    Column {
        // Text element (equivalent to TextView)
        Text(text = text)

        // Button with click handler
        Button(onClick = {
            text = "Button pressed"
        }) {
            Text("Hello, I am a Button")
        }
    }
}
```

Listing 2.3 presents the same user interface as in Listing 2.1 and performs similar programmatic operations as shown in 2.2. Unlike the XML-based approach, the UI and its associated logic are expressed entirely in Kotlin, allowing the entire implementation to be handled within one coherent language environment.

2.2.5 Image-loading libraries

Image-loading libraries are third-party tools designed to efficiently handle the process of fetching, decoding, and displaying images in applications. Although they are commonly used to asynchronously load images from the network, they also manage images stored locally on the device, which is what is examined in this thesis.

Image loading is a resource-intensive task, and inefficient handling of images can negatively affect application performance, for example by causing slow response times or even triggering an Application Not Responding (ANR) [7]. ANRs are system-level errors raised by Android when the UI thread is blocked for too long, preventing the application from responding to user input [1]. In a paper by Li et al. [22], the authors found that a significant number of open-source Android applications suffer from image-related performance issues caused by non-functional

defects in the image-handling code. They report that 26.5% of the issues were due to repeated and redundant image decoding, indicating improper cache management, while more than 10% were caused by performing image decoding on the UI thread, thus blocking the rest of the application from executing.

To address these challenges, image-loading libraries implement various optimizations. They typically handle images asynchronously, performing fetching and decoding on background threads rather than on the main thread. To avoid repeated and redundant decoding, they also use multi-level caching, ensuring that an image only needs to be processed once and can be quickly reused if displayed again.

2.2.5.1 Glide

Glide, introduced in 2014, is an Android image loading library designed to efficiently manage image fetching, decoding, caching, and rendering. To reduce the overhead associated with these operations, Glide employs a multi-layered caching strategy before initiating a new image request [13].

As Glide was introduced prior to Jetpack Compose, it was originally developed to target the traditional Android View system. However, Glide has since added an integration for Compose, enabling its use within the newer UI framework. The Glide developers explicitly state that the integration is still in beta and that its APIs are being reconsidered and may change in future releases [14].

2.2.5.2 Coil

Coil, introduced in 2019, is a modern image loading library for Android that supports both the traditional Android View system and Jetpack Compose. According to its official documentation, the library emphasizes simplicity and performance. Like Glide, Coil performs several optimizations, such as memory and disk caching, and the library integrates with modern Kotlin language features, including coroutines [16].

2.3 Related work

In-flight entertainment is a specialized domain, and not much work has been done on rendering time analysis related to the software running on IFE devices. However, studies on UI performance in general Android applications do exist. One such study is by Fjodorovs & Kodors [18], who compared the rendering performance of two applications, one implemented using the traditional Android View system and the other with Jetpack Compose. The two applications shared identical logic, with the primary difference being the implementation of the UI. The application consisted of an image and placeholder text. Performance was measured by starting a timer before the application rendered the UI, then stopping the timer once the graphical

content had been rendered. The study concluded that, for this simple application, the rendering time was 46% longer when using Jetpack Compose compared to the Android View system. The experiments in the study were performed on an Android emulator running on a Windows 10 host machine. The virtual device was a Pixel 2 XL configured with Android 10, and was allocated 2 CPU cores and 2 GB RAM.

In the paper [24], Song et al. focus on identifying defects related to image loading in Android applications. They introduce IMGDroid, a static analysis tool designed to detect five categories of image-loading anti-pattern. In the paper, an anti-pattern is defined as a recurring coding practice that is known to cause image-loading defects. Based on their empirical study on 1,000 commercial Android applications, they demonstrated that image-loading defects are highly prevalent: 86.5% of the apps analyzed contained at least one defect, with an average of 6.32 defects per app. This shows that image-loading issues occur frequently in practice.

2.4 System and test environment of case company

The IFE solution of the case company is a fully developed production system that exceeds the scope of the prototype used in this study. The prototype used in this study is built on one of the configurations present in their existing implementation, which we compare against three additional configurations. More details on this comparison are provided in Section 3.

For testing purposes, access was granted to several generations of the company's IFE screens. This allowed for the evaluation of the prototype in a setting that more closely resembled a real deployment environment, providing more representative performance measurements than would be possible in a purely simulated setup.

3 Method

This chapter describes how research was conducted during the work on this thesis. It starts with a process overview that outlines the entire process and the included phases, which consist of preparation, prototyping, and analysis. The following section contains details of the benchmark design, including the scope of the project, how the prototype was implemented and altered between benchmarks, and which metrics were captured. The chapter ends with a description of the prototype iterations performed.

3.1 Process overview

Figure 3.1 shows an overview of the process used during this thesis.

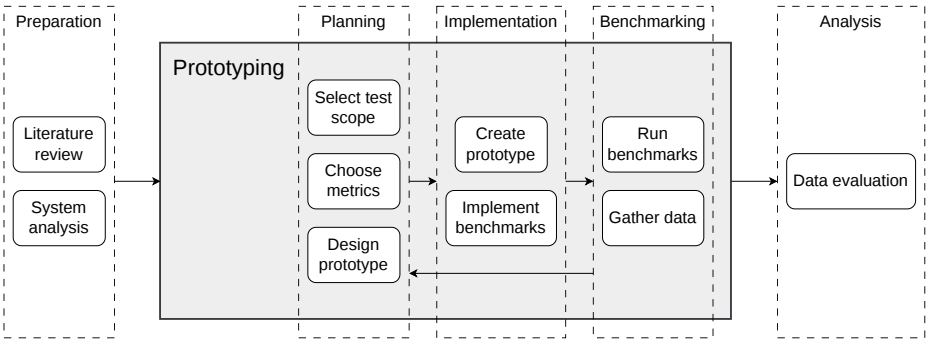

Figure 3.1 Overview of the work process

3.1.1 Preparation

Before conducting any benchmarks, it was necessary to gain an understanding of the field in which research would be conducted. The purpose of this was to learn what tools and methods are being used in the current system and then to use this knowledge to make decisions regarding what should be benchmarked and how these benchmarks should be designed. Two main methods were used for the preparatory research: a literature review and an analysis of the current system.

System analysis

The first step of the preparation phase was to analyze the current system in order to build an understanding of how page rendering during navigation is currently performed and under what circumstances the loading times become notable for a user.

Gaining an understanding of the system was mainly achieved by investigating the source code and interviewing developers, during which it was found that various optimization methods were used when creating pages and caching them whenever appropriate. However, the actual rendering of the pages on the screen is done using a popular UI framework for Android, together with an image-loading library. Because these parts are responsible for rendering, the main focus of this study, we decided that the benchmarks should be designed to test different variations of UI frameworks and image-loading libraries.

As the next step in system analysis, we manually navigated the current system on one of the test screens with the intention of finding situations where page rendering was particularly slow. At this stage, no tools were used to measure rendering times. We noted that as the number of components on a view increased, the render time also seemed to increase. Pages with a higher quantity of media and text took notably longer to render than pages with fewer components, and we occasionally observed that large images were rendered later than other content on the same page. Based on the findings of the usage test, we concluded that the content to be rendered seemingly has an impact on rendering time. Because of this, we deemed it worthwhile to render different pages with various levels of content during benchmarking, primarily to obtain results that are not only relevant for one type of page, but also to see if any particular type of page stands out in terms of rendering time and performance.

3.1.2 Prototyping

To compare different UI rendering methods, we built a prototype to use for benchmarking. There were multiple reasons why we chose this as our method, but two factors were especially important when the decision was made. Firstly, the time that would otherwise be spent integrating the various methods into the current system could instead be used to create a larger number of benchmarking scenarios, thus generating more data to be analyzed. Secondly, a prototype allowed for more flexibility and adaptability when designing our benchmarks. The current system has multiple optimization methods in place that could affect the measurement results, but this might also make the results very specific to this system.

The prototyping phase of the project was an iterative process, where the learnings from each iteration were taken into account when refining and improving the prototype and benchmarking processes. Each iteration consisted of three stages: Planning, Implementation, and Benchmarking.

The first stage of the prototyping process was planning the upcoming iteration, mainly regarding what should be benchmarked and how the prototype should be designed. We made many of the largest decisions during the first iteration — such as the general structure and functionality of the prototype and what metrics to capture — and then kept them for the duration of the prototyping process. In subsequent iterations, the planning stage was instead used to analyze previous iterations and, based on this, decide on tweaks to the existing process. Using the Android Macrobenchmark library is an example of a decision we made in a later iteration.

The second stage of the prototyping consisted of implementing what was decided during the planning stage. This meant building or modifying components of the prototype and writing the benchmarks that would produce the data to be analyzed.

The third and last stage of a prototyping iteration was to run the benchmarks. This was done by installing the prototype on the IFE screen used for testing at the case company and running the benchmarks on it. To make the conditions as equal as possible, the configurations were benchmarked right after each other on the same screen. Before benchmarking a configuration, all other configurations were uninstalled.

3.2 Benchmark design

This section describes the specifics of the benchmarks. It presents what was benchmarked, the program that ran the benchmarks, the metrics that were measured, and the device that was used during the benchmarks.

3.2.1 Scope

For this study, two UI frameworks and two image-loading libraries were benchmarked. The UI frameworks were Android's View system and Jetpack Compose (BOM version 2024.09.00), and the image-loading libraries were Glide (version 4.16.0) and Coil (version 2.7.0). The prototypes were developed and benchmarked using Android API level 36, with a minimum supported API level of 24. Both Glide and Coil are used with the default settings provided by the libraries. The Android View system was chosen because it is widely used and has historically been the dominant framework for Android user interface development. Compose was selected because it is the alternative suggested by Google for new projects [12]. Glide and Coil are among the most popular image-loading libraries and were chosen because of their support for both Android's View system and Compose.

3.2.2 Prototype design

The prototype consists of a start page that contains five buttons, each leading to one of the views used for benchmarking. The different views can be found in Appendix A. Pressing a button causes the program to draw the corresponding view on the screen, and the user needs to press "back" to return to the start screen. The views consist of a number of images and, in some cases, a color gradient overlay together with some text. All images used in the views were taken with the same camera, have the same original resolution, and have only been cropped to the correct proportions before being copied to the testing device. This means that the images are stored and handled by the program in a larger resolution than they are displayed on the screen. The images used in view 1 were stored in a resolution of 1784x4010 pixels, and the images used in views 2-5 were stored in a resolution of 6014x3383 pixels. All images were stored in JPG format, and had sizes ranging from 451 to 4073 kilobytes.

To allow for automated benchmarking and enable measurements through startup performance metrics, the app had the functionality to launch into a specified view. The start screen was mainly used when designing views to manually load them and ensure that they were rendered as intended, but for the actual benchmarks, a view was instead selected using intents when launching the app. Intents are objects included with the call to start the program, which can be used to specify what should be shown when the app starts [8]. In this prototype, the intent includes an integer that specifies which view to render.

After a view has been rendered and all images are drawn on the screen, we send a signal to the system using the built-in method `reportFullyDrawn()` on the activity. The total rendering time, from the startup signal being called to the view being fully rendered, is then sent to the system logs of the device running the benchmarks, from where it can be read [2]. Callbacks from the image-loading libraries, which are invoked when an image has been successfully loaded and displayed, are used to determine when the images are fully displayed on the screen. Each drawn image adds to a counter, and when this counter reaches a target value, specifically the number of images to be rendered within the current view, `reportFullyDrawn()` is called.

The benchmarks themselves were conducted using the Android Macrobenchmark library with the code shown in Listing 3.1. The benchmark uses `StartupTimingMetric`, which measures the time in milliseconds from the program start to both when the first frame is rendered and when the fully-drawn signal is sent [3]. Startup mode `WARM` is used, which is equivalent to a user backing out of the program before opening it again without fully closing it in between. This means that the results will not include the time required to start the app process, as it is running continuously throughout the duration of the benchmark [4]. To avoid results being affected by a fixed rendering order, a randomizer is used to determine the order in which the

views are rendered. Before each iteration of the benchmark loop, the next view to be rendered is selected by using the randomizer and updating the intent. Because startup mode `WARM` is used, the benchmark loop will be executed once before the actual benchmarks to start the program. To avoid any caching during startup, the first iteration uses view number 0, which is just a blank screen. The actual benchmark loop contains only two lines of code. The first line starts the app on the selected view, and the second line adds a short delay to give the fully-drawn signal enough time to be sent before the next iteration. No explicit compilation mode was selected in the Macrobenchmark tests. Therefore, the default compilation behavior for the framework was used.

Listing 3.1 The macrobenchmark used in the study

```
const val n = [Number of iterations]
val viewList = listOf(1, 2, 3, 4, 5)
val randomizer = Random([Randomizer seed])

@Test
fun renderTest() {
    var viewNumber = 0
    val intent = Intent()
    intent.setClassName("[package name]", "[package name].MainActivity")

    var isFirstIteration = true

    benchmarkRule.measureRepeated(
        packageName = "[package name]",
        metrics = listOf(StartupTimingMetric()),
        iterations = n,
        startupMode = StartupMode.WARM,
        setupBlock = {
            // First iteration is not included in the results
            // Therefore, rendering blank View 0 for the first iteration
            if (isFirstIteration) {
                isFirstIteration = false
            } else {
                viewNumber = viewList[randomizer.nextInt(viewList.size)]
            }

            intent.putExtra("TARGET_VIEW", viewNumber)
        }
    ) {
        startActivityAndWait(intent)
        device.wait(
            Until.hasObject(By.text("[Text that never appears]")), 100)
    }
}
```

3.2.2.1 Benchmark configurations

Instead of implementing both UI frameworks in the same project, two separate prototypes were created to simplify dependency management and reduce implementation complexity. The functionality of both prototypes is the same, which allows the

Macrobenchmark code used for benchmarking to be identical for both versions. This helped to ensure that the measurements received from the program are comparable for all configurations of the program. The only difference between the prototypes is in which UI framework handles rendering and navigation.

The other benchmark configurations are selected by editing various variables within the code. We created two classes responsible for image loading: one using Glide and one using Coil. Both classes implement an interface we created that is called whenever an image is to be rendered. This allows for the selection of the image-loading library to be done by switching which class is used for the particular benchmark run. Both image-loading libraries were used with their default cache size settings, as provided by their respective libraries. No modifications were made to memory or disk cache configurations, ensuring that all performance differences reflect the libraries' standard behavior.

The number of times the benchmarks will run is chosen through one variable, and there is a list of integers from which views are selected. The list can be modified to include only views to be used in the specific benchmark iteration. Lastly, the views are selected using a randomizer from the Kotlin standard library. The randomizer is always created using the same seed, making the view order identical for each benchmark that includes the same views [20].

3.2.3 Metrics

For each benchmark, two metrics were measured: **Time to initial display *(TTID)*** and **Time to full display *(TTFD)***. Both metrics are concepts taken from the Macrobenchmark library and were measured in milliseconds. **TTID** is the time it takes from the initial call to the UI framework until the first frame containing visible content from the view is displayed on the screen. This value is obtained from the macrobenchmark results. **TTFD** is the time it takes from the initial call to the UI framework until the fully-drawn signal is sent to the system. This value is read by capturing the log messages every time the fully-drawn signal is sent.

Ideally, both TTID and TTFD would be obtained from the resulting file of the macrobenchmark tests. When performing initial benchmarks on an emulator, both TTID and TTFD were included in this file, but on the physical device used for the actual benchmarks, only TTID was available. This is due to a permission restriction on the physical device that hinders the macrobenchmark module from receiving the fully-drawn signal. Because of this, it was not possible to obtain both metrics from the same location, and the TTFD values were instead collected through the logs of the testing device.

3.2.4 Testing device

All experiments were conducted on an IFE display unit running Android. The device was connected to a computer using Android Studio and the Android Debug Bridge (ADB), which allows for deployment and performance measurements.

The hardware specifications of the device are as follows:

- **SoC:** Qualcomm APQ8096pro
- **CPU:** Quad-core ARMv8 (AArch64)
- **Memory:** 8 GB RAM

3.3 Benchmark iterations

The results presented in Chapter 4 were collected in two separate iterations. For both iterations, each benchmark was run once for each of the four possible combinations of UI framework and image-loading library. All benchmarks were run on the same, physical, device and each benchmark within an iteration was performed on the same occasion, uninstalling the program used for the previous benchmark before starting a new one.

Iteration 1

For the first benchmark iteration, the same view was rendered 100 times in a row without any other views being rendered in between. The view used for this is shown in Figure 3.2

Iteration 2

The second benchmark iteration was similar to the first, but instead of rendering the same view every time, this iteration alternated between five different views that can all be found in Appendix A. The benchmarks ran a total of 500 iterations, alternating between views using the randomizer with the same seed for each benchmark. Views 1 and 2 were rendered 100 times each, views 3 and 4 were rendered 99 times, and view 5 was rendered 102 times.

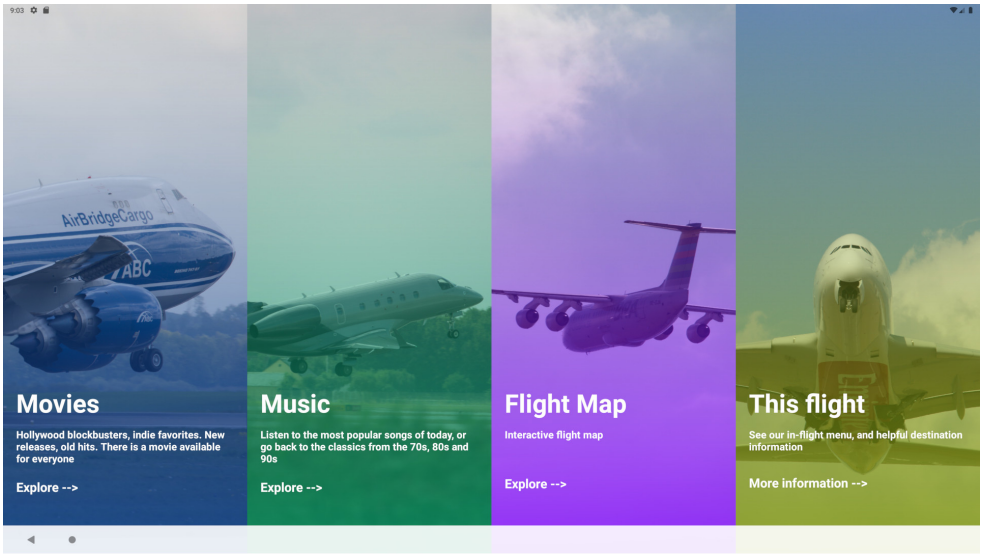

Figure 3.2 View 1, designed to resemble the layout of a realistic IFE system

4 Results

This chapter presents and analyzes the results of the benchmarks described in chapter 3. In the first section, the focus is on the results when the same view is rendered multiple times. The following section focuses on the results when alternating views are used. The last section briefly mentions the results for the individual views during the benchmark with alternating views.

The naming convention used throughout this chapter is as follows: ComposeGlide and ComposeCoil refer to Jetpack Compose used together with Glide and Coil, respectively, while XMLGlide and XMLCoil refer to the traditional Android View system using XML-based layouts with Glide or Coil.

All results presented in this chapter are acquired from the real hardware on which the benchmarks were performed.

4.1 One view

This section covers the results of the first benchmark iteration. During this iteration, the same view was rendered 100 times. The view being rendered was view 1, containing four images, all with color gradients and text overlays.

4.1.1 Time to Initial Display

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	132.01	263.85	130.68
ComposeGlide	125.86	231.77	124.79
XMLCoil	137.47	251.24	136.33
XMLGlide	133.37	227.36	132.42

Table 4.1 Average Time to Initial Display (ms), one view

Table 4.1 shows the TTID results from the first benchmark. For all four configurations, the first iteration is considerably slower compared to the average of the remaining iterations, differing by 95 to 135 milliseconds. Excluding the first iteration, the average TTID is similar for all four configurations, with a difference of less than

12 ms between the fastest, Compose with Glide, and the slowest, XML layouts with Coil. Because of how similar the results are, and that we only access the arithmetic mean of the renders, no strong conclusions can be drawn from this data.

4.1.2 Time to Full Display

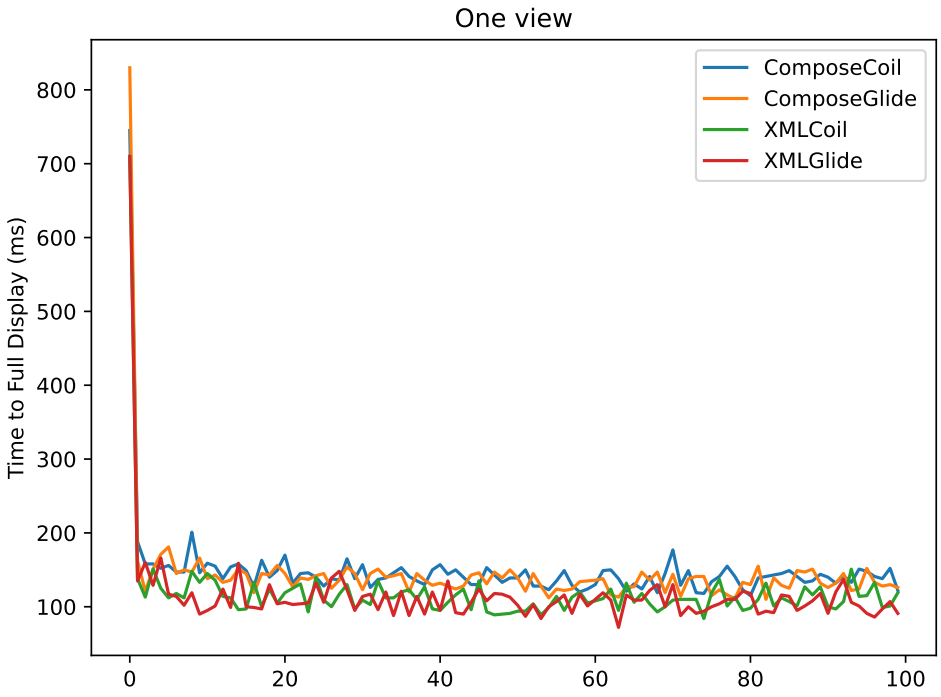

Figure 4.1 Results when rendering one view

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	148.04	745	142.01
ComposeGlide	143.23	830	136.29
XMLCoil	118.38	709	112.41
XMLGlide	115.02	710	109.01

Table 4.2 Average Time to Full Display (ms), one view

As with the TTID, there is a considerable difference between the results for the first iteration and the remaining iterations, which can be seen in Table 4.2. However, for TTFD, this difference is much larger. For all configurations, the difference is larger than 595 milliseconds, ranging all the way up to a difference of 695 milliseconds.

Looking at the graph in Figure 4.1, it is clearly visible that the first iteration is the only major outlier before the results stabilize for all configurations. The TTFD for the first iteration stands out for Compose with Glide, being almost 100 milliseconds slower than the second slowest, which is also using Compose. The differences in TTFD for the first iterations of Compose and the Android View system are notable, but it needs to be taken into account that there is only one measurement available for each configuration.

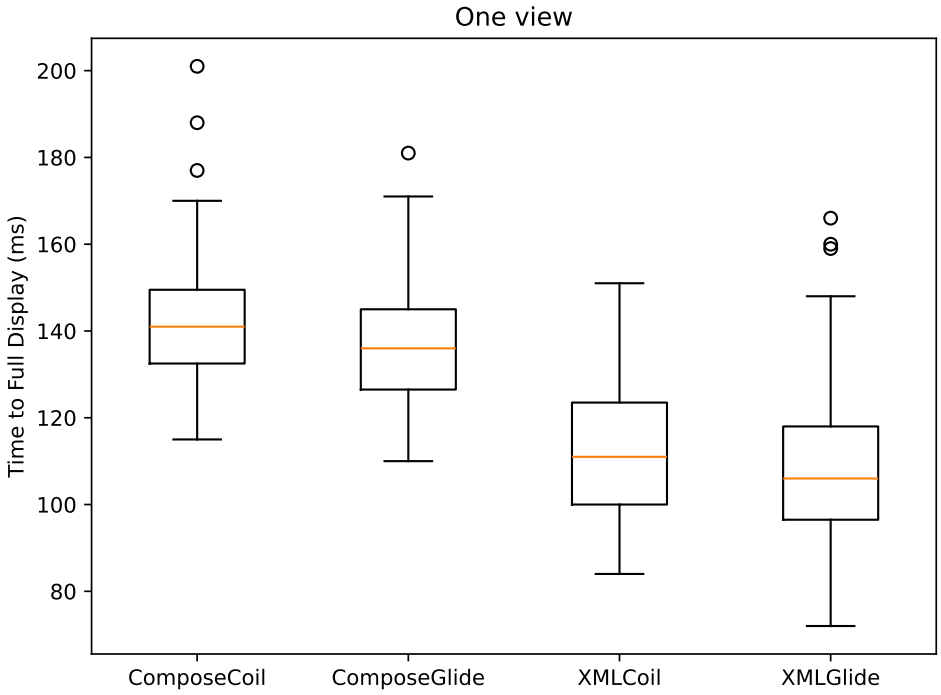

Figure 4.2 Results when rendering one view, excluding first render

When excluding the first iteration, the average TTFD results for all configurations are similar. The difference between the fastest and slowest configurations is only 33 milliseconds, and the graphs for each configuration follow similar patterns. When comparing UI frameworks, a different result than for TTID can be observed. Both configurations using Compose were faster when measuring TTID, but when looking at the results for TTFD, it is the configurations using XML layouts that are faster. This is also shown in Figure 4.2, where the boxes related to configurations using the traditional Android view system show lower values than those using Compose. The results also show slightly lower rendering times for configurations using Glide compared to those using Coil, but this difference is relatively small.

4.2 Alternating views

This section covers the results of the second benchmark iteration. During this iteration, the five views shown in Appendix A were alternately rendered a total of 500 times. The view to be rendered for each iteration was chosen using a randomizer.

4.2.1 Time to Initial Display

Configuration	All iterations	First iterations	Remaining iterations
ComposeCoil	155.19	245.2	154.28
ComposeGlide	110.29	178.67	109.6
XMLCoil	126.94	226.17	125.94
XMLGlide	108.78	150.22	108.36

Table 4.3 Average Time to Initial Display (ms), alternating views

Table 4.3 shows the TTID for the second benchmark. The first iteration is consistently slower than the average of the subsequent ones, differing by about 70–100 milliseconds. When the first iteration is excluded, the average TTID is nearly identical for the two configurations using Glide, slightly higher for XML layouts with Coil, and even higher for Compose with Coil. The difference between the fastest configuration, XML layouts with Glide, and the slowest, Compose with Coil, is nearly 46 milliseconds, making the latter roughly 40% slower. When comparing the different UI frameworks, the XML-based configurations show slightly lower overall times.

4.2.2 Time to Full Display

Configuration	All iterations	First iterations	Remaining iterations
ComposeCoil	722.1	1380.8	715.44
ComposeGlide	268.51	1306.6	258.03
XMLCoil	631.79	1294.2	625.1
XMLGlide	202.86	1229	192.49

Table 4.4 Average Time to Full Display (ms), alternating views

Table 4.4, together with the plots in Figure 4.3 and Figure 4.4, presents the TTFD results for the benchmark involving alternating views. The table shows that, as with the other results, the average time for the first iterations across the different views is significantly slower than the average of the subsequent iterations. For all configurations, the average TTFD of the remaining iterations is more than half a second faster than that for the initial display. For XML layouts with Glide, the average time is reduced by more than one second.

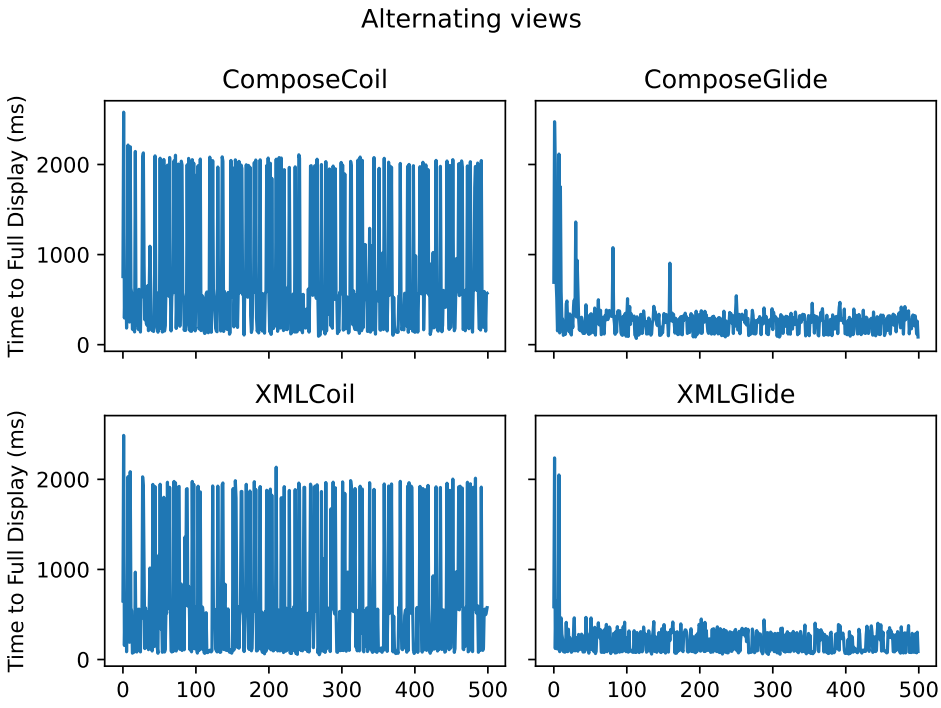

Figure 4.3 Results when rendering alternating views

Figure 4.3 also reveals that the graphs for the different configurations have different profiles. All configurations show spikes initially for the first renders of the different views before stabilizing at lower levels, but the configurations using Glide stabilize at a level that is significantly lower than those using Coil. In the configuration using Compose and Glide, there are also some spikes visible in the data. When looking at the scatter plots per view found in Appendix B.3, the outliers can be found in the results for View 3 and View 5, which contain 25 images each.

Figure 4.4 show a much larger spread among the results for configurations using Coil compared to those using Glide. Both configurations using Coil show a large number of outliers, that originate from the renders of views with more images. Some outliers can also be seen for the configuration using Compose and Glide, these are the spikes that could be seen in the graph.

The average times during the initial iterations range between 1229 and 1380 milliseconds, with XML layouts with Glide being the fastest and Compose with Coil being the slowest. The absolute difference between them is nearly 150 milliseconds. However, in relative terms, this corresponds to a 12% slowdown. When excluding the first iteration, the average TTFDs for the 2 configurations using Glide are signif-

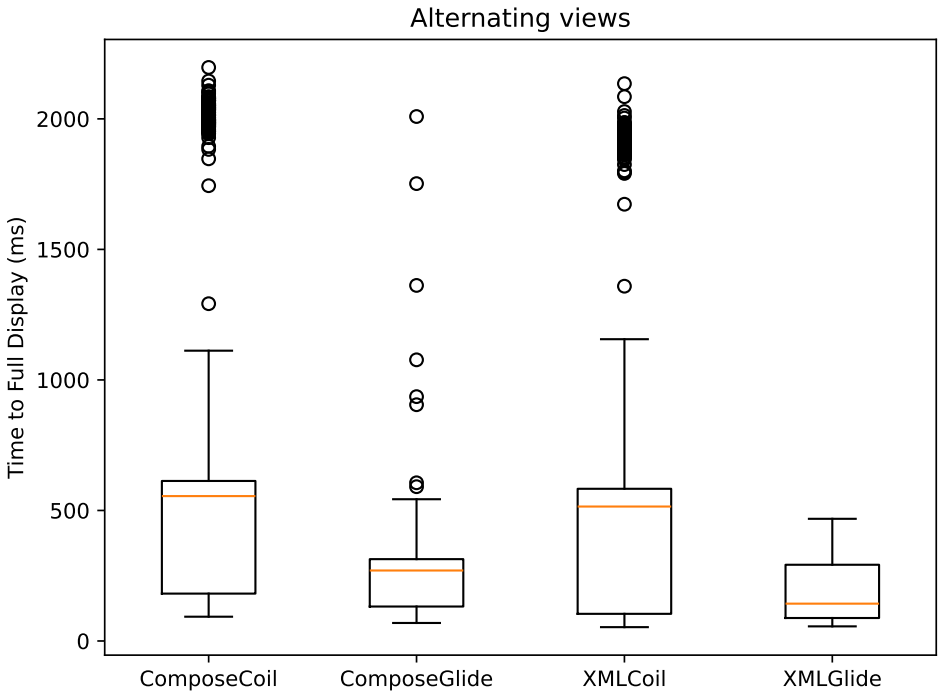

Figure 4.4 Results when rendering alternating views, excluding first renders

icantly lower than those for the configurations with Coil. XML layouts with Glide has an average of 192 milliseconds, whereas Compose with Coil has an average of over 715 milliseconds. A framework-level difference is also observable, as the XML layout configurations are consistently faster than their Compose counterparts when paired with the same image-loading library.

4.3 Results per view

After the second benchmark, using alternating views, the TTFD results were separated based on the view that was rendered. The separated results can be found in Appendix B.3. When ignoring the first iteration, it can be seen that views 3 and 5, containing 25 images each, were considerably slower than views 2 and 4, which contained only one image each. The difference is much larger for configurations using Coil than for those using Glide. The difference between views with many images and views with few images is also large for the first iterations, but in those cases, there is no notable difference between Coil and Glide. View 1 had a different layout compared to the other views, containing four images with individual overlays and

different proportions than those used in the other views, but the TTFD results for this view were relatively similar to those of the views with only one image.

Adding a color gradient with a text overlay on the images, which was done for views 4 and 5, in most cases makes the TTFD slightly bigger than for similar views without the overlay, especially for the first iterations. However, there are many exceptions to this pattern in the results for different types of configurations. Hence, it is not possible to make any certain assumptions based on this.

5 Discussion

This chapter discusses the results presented in Chapter 4. The discussion focuses on three different comparisons: between the UI frameworks, between the image loading libraries, and between the different views. Threats to the validity of the study are discussed, along with aspects that would have been done differently had the study been conducted again. Lastly, future work in the area is suggested.

5.1 UI frameworks

For the TTIDs - which, as described in the method section, measure the time until the first frame of the new view is drawn - Android's View system and Jetpack Compose perform similarly during the benchmark. In the benchmark with one view, Compose is marginally faster, whereas in the benchmark with alternating views, the XML-based configurations are slightly faster. The differences are small and do not clearly lean in any direction, so no clear conclusion can be drawn about which framework performs better based on this metric. The TTID metric measures the time until the first feedback is given to the user of the system that something is happening. Since this metric provides the user with immediate confirmation that the system has reacted to their input, a faster time can contribute to a more responsive experience.

For the TTFDs measurements, Android's View system showed a slightly faster time, although the difference was small. One possible reason why Android's View system is faster at rendering the full view could be that the prototype has a very shallow view hierarchy with few nested views, making XML layout inflation relatively inexpensive. The views of the real IFE application that we looked at in the beginning of this study also exhibited relatively shallow layouts. However, other views within an IFE system may be more complex, with deeper hierarchies, which could increase the cost of measuring and laying out views and thereby degrade the performance of the Android View system. Jetpack Compose could potentially scale better with more complex layouts since it avoids the traditional view hierarchy used in the Android View system. The TTFD metric is crucial for user experience, as a faster time reduces the delay before the user sees the complete content of the new view, thus improving the overall user experience.

5.2 Image-loading libraries

As illustrated by the graphs and tables in Chapter 4, Glide outperforms Coil in almost all of the conducted benchmarks. This is the case for the TTID results, where Glide consistently achieves lower times than Coil. Glide is also faster than Coil when comparing TTFDs, and the difference is particularly evident in the average TTFDs for alternating views. A likely explanation is that Glide is more effective at caching and reusing images once they have been decoded. Since image decoding is typically the most time-consuming part of the rendering process, bypassing it when a cached bitmap is available allows the image to be drawn on the screen much more quickly. This behavior can be visualized in the graphs titled *ComposeGlide* and *XMLGlide*, as the highest rendering times are concentrated at the beginning of the graphs.

For the Coil-based configurations, the rendering times for TTFD do not reduce nearly as much as for Glide, indicating that caching was likely not functioning as expected. This behavior stood out when the results were analyzed, and a minor adjustment to the Coil implementation later showed that it was possible to achieve a similar behavior for configurations using Coil as for those using Glide. The code adjustment, among other things, consisted of specifying the size each image should occupy more exactly, thus not requiring the library to calculate it itself. Without this adjustment, it is likely that the library decoded the image at its original size, cached the full-sized bitmap, and only then scaled it to the required dimensions. With the code modifications, Coil would instead first resize the image, decoding it at the target size, and cache the version that would actually be reused. As the code used during the benchmarks uses only the simplest possible function call for each library, it is possible that Glide does this automatically, whereas Coil does not and needs more adjustments for caching to work optimally.

Some minor benchmarks with the code adjustment were performed on a virtual device in Android Studio, and the initial analysis of those results implied that Coil behaved more similarly to Glide. However, due to time limitations, it was not possible to re-do the full benchmarks on the IFE device after the modifications, but it is possible that the rendering times for Coil would have been faster there as well.

5.3 View layout

The results for the individual views show a clear correlation between the number of images in a view and the TTFD. Views 3 and 5, both containing 25 images, were significantly slower for all configurations when compared to the other three views. What is more surprising is how the TTFDs for view 1, containing four images, are consistently similar to those of views 2 and 4, which contain only one image each. There are a few factors that could contribute to this. One of these is related to the

lack of image loading optimization mentioned above. Because the layout is different for view 1 compared to the other views, it is possible that it is written in a way that is better optimized for how the image loading libraries handle image rendering. Another possibility is that the results are affected by the size of the original images. Because of how the images were cropped before being stored on the device, the images used for view 1 were only roughly one fourth the size of the images used in the other views.

When using the configuration with Compose and Glide, we could see some spikes present in the data for views 3 and 5. These were not present in the data for the other views, and the cause could not be determined within the scope of this study. Due to time limitations, we could not attempt to reproduce these outliers. However, the theory we have is that the spikes are originating from the UI framework. In the results for configurations using Coil, the results are consistently higher. This might mean that there are occasions in the configuration using Compose and Coil where the UI framework has these spikes, but this is hidden in the results by the image loading library being slower.

From the view results, it can be concluded that image loading is one of the most important factors for render time. As the images are the largest resources in the prototype used for this study, it is logical that they would have a significant impact. The fact that adding overlays with color gradients and text has a very minor impact on the TTFD further strengthens the thesis that image rendering is what takes up most of the time when rendering the views. Because the images are rendered on the screen in a different resolution than they are stored, resizing also needs to be done in addition to reading the image files, which adds to the time needed to render the images, especially in the first iteration when they are not yet cached.

5.4 Threats to validity

The biggest threat to the validity of this study is the fact that it was conducted on a prototype. Despite the efforts to make the prototype realistic, there are multiple factors that differ from the real system that could potentially impact the results. Furthermore, because the prototype was developed specifically for this study, it is possible that characteristics affecting performance that can be found in the production system might not be fully reflected in our application.

One of the biggest differences is that the prototype is a significantly smaller application than the actual application used in real IFE systems. The prototype contains only five different views and a landing page, which allows the image-loading libraries to keep all images stored in the cache memory simultaneously. In a real IFE application that contains hundreds of views, this would likely not be possible. Another impor-

tant factor related to the size of the program is the hierarchical tree structure in XML layouts. As the XML-based configurations performed better than Compose in the benchmarks on the prototype, this indicates that searching through the tree for components was faster than creating the objects with Compose, but this is not necessarily the case for a system with a larger component tree. Because of the time limitations, the prototype approach was still deemed the best way to benchmark as many factors as possible, but it needs to be taken into account that it is possible that the results would not be linearly scalable.

Another aspect to take into account is that the prototype code was written to be simple, and written by programmers who were new to the programming language. While best practices were followed, it is possible that there are more efficient solutions that could have reduced the render time further. An example of this could be observed for the configurations using Coil, where a modification made to the code after the benchmarks managed to significantly reduce the TTFD, and it is possible that similar improvements exist in other parts of the code. Furthermore, we did not look into the technical low-level details of the image processing pipelines of Glide and Coil, such as caching behavior, that may have affected the results. If the study was to be done again, more time would have been spent on researching code optimization and technical details regarding the different UI frameworks and image-loading libraries, but the results in this thesis still provide insights into how well the different configurations perform when used as simple as possible.

During development, we divided the responsibility of coding the two prototypes using different UI frameworks between us. Because of this, a potential factor in the results is the developer responsible for the specific prototype. To reduce the impact of the choice of developer, as much of the code as possible was identical in the two prototypes, and the code that did differ was reviewed and discussed together by both developers.

For this study, the images that were rendered on the screen were fetched from the local storage of the testing device. In a real airplane setting, media resources are stored on a central server in the aircraft, and from there, they are fetched by the seatback devices when needed. It is possible that the different configurations differ in how effectively they handle images that are fetched from a remote server, and this difference would not be reflected in the results from the prototype benchmarks.

In the study, we also assume that the success callbacks provided by Glide and Coil are invoked in comparable stages of the image rendering process, after an image has been fully loaded and delivered to the UI-thread. Minor differences in the exact timing of these callbacks could influence the measured rendering times. However, we consider it unlikely that these potential timing differences would be large enough to affect the overall conclusions of the report.

Lastly, it could also be argued that the results for the first render of a view are more relevant in an IFE context, as a user will usually navigate to different parts of the system rather than alternate between the same screens. Most measurements in this study are taken when the program returns to a view that has already been rendered once, while each view has only one data point for the first render. Examining render performance when rendering a view with cached data is still relevant, as a user will return to different menus in a real system. However, with more time, additional benchmarks that restart the program between renders would have provided an even deeper understanding of the performance of the different configurations in varying situations.

5.5 Future work

A topic that could be investigated further is how optimizations for the image-loading libraries affect the results. As stated previously, only the most basic function call of each image-loading library has been used. However, it would be of interest to analyze how the different libraries could be optimized and tailored to function optimally. This could introduce variations in the results, since the caching for each configuration may exhibit distinct behavior, which in turn could lead to different conclusions.

In this study, we were unable to properly investigate the spikes shown in the TTFD results for the configuration using Compose and Glide on views with 25 images. Another study that could be conducted is attempting to reproduce similar spikes and try to understand where the spikes are originating from and the potential reasons behind them.

Another topic to explore is how the image-loading libraries and UI frameworks perform in scenarios involving scrollable views, and whether the results differ from those observed during navigation. Since the results of this study suggest that images are the components that require the most time to process and render, the views that would be most interesting to investigate are image-rich views. In our evaluation of the current system, we noted that rendering delays occurred not only during navigation between views but also when scrolling through image-rich interfaces. These observations suggest that scrolling introduces additional performance challenges that were outside the scope of this thesis. Since this study focused exclusively on navigation performance, a complementary study examining scrolling behavior under similar experimental conditions could provide valuable insights.

It could also be of interest to integrate the different libraries into a real and existing IFE application and compare how the rendering times differ in a larger context. As these benchmarks were conducted on a small prototype consisting of only 5 views, the results might show larger differences when used in a real IFE application that

uses images of different sizes and quality and with more complex layouts. It would also be interesting to compare the different UI frameworks in a real IFE application, but as it would require a substantial amount of work to convert a system using one UI framework to the other, it was not deemed possible for the time frame of this thesis.

A more detailed analysis of how each library and framework behaves could also be interesting to examine. For instance, a study could examine the caching strategies of the libraries, their image-decoding processes, and their approaches to bitmap handling, as well as differences in memory allocation between the two UI frameworks.

6 Conclusions

In this thesis, different configurations for handling view rendering in Android-based IFE systems have been compared, and benchmarks have been performed to evaluate the different implementations. Based on the background, results, and discussion presented in the previous chapters, the research questions stated in the introduction will be answered in the best possible way. The two research questions are answered separately below:

1. *What methods could be used to reduce the rendering time when navigating within the IFE system, taking into account the system's limitations?*

   The findings from this thesis indicate that rendering times during navigation are affected by both the selected image-loading library and the UI framework. While the impact of the image-loading library was shown to have a substantial effect on rendering times, the influence of the UI framework remains less clear and should be investigated further, ideally in a larger application featuring more complex view structures.

2. *How do the different methods perform compared to each other, and what are their advantages and disadvantages?*

   The results show that the Glide library outperforms the Coil library in almost all of the performed benchmarks. Beside the most important aspect that is the rendering time, the advantage of Glide compared to Coil is its maturity and long-standing use in Android development. However, if Jetpack Compose is used as the UI framework, we believe that Glide should be used with awareness that its Compose integration is still in beta and is subject to potential API and performance changes. It was found that it is possible to significantly reduce the rendering time of configurations using Coil with code optimizations, but configurations using Glide perform well even without such optimizations.

   Regarding the UI frameworks, although the Android View system performed marginally better in the benchmarks, it is not possible to confidently determine which one is definitely faster than the other. When considering the broader advantages and disadvantages of the frameworks, much comes down to developer preference. The Android View system offers a strict separation between UI and logic, which some developers may find beneficial for maintaining structure, though it also requires more boilerplate and familiarity with both

XML and Kotlin. Jetpack Compose, on the other hand, more closely resembles conventional programming languages, meaning that developers experienced in Java and other conventional programming languages may find it more intuitive and faster to adopt.

References

[1] Android Developers. *ANRS*. URL: https://developer.android.com/topic/performance/vitals/anr.

[2] Android Developers. *App startup time*. URL: https://developer.android.com/topic/performance/vitals/launch-time (visited on 2026-04-22).

[3] Android Developers. *App startup time*. URL: https://developer.android.com/reference/kotlin/androidx/benchmark/macro/StartupTimingMetric (visited on 2026-04-22).

[4] Android Developers. *App startup time*. URL: https://developer.android.com/topic/performance/vitals/launch-time (visited on 2026-04-23).

[5] Android Developers. *Benchmark your app*. URL: https://developer.android.com/topic/performance/benchmarking/benchmarking-overview.

[6] Android Developers. *Build better apps faster with Jetpack Compose*. URL: https://developer.android.com/compose (visited on 2026-04-09).

[7] Android Developers. *Handling bitmaps*. URL: https://developer.android.com/develop/ui/views/graphics.

[8] Android Developers. *Intents and intent filters*. URL: https://developer.android.com/guide/components/intents-filters (visited on 2026-04-22).

[9] Android Developers. *Layouts in views*. URL: https://developer.android.com/develop/ui/views/layout/declaring-layout (visited on 2026-04-07).

[10] Android Developers. *Migrate XML Views to Jetpack Compose*. URL: https://developer.android.com/develop/ui/compose/migrate/migrate-xml-views-to-jetpack-compose (visited on 2026-04-09).

[11] Android Developers. *Thinking in Compose*. URL: https://developer.android.com/develop/ui/compose/mental-model.

[12] Android Developers. *Why adopt Compose*. URL: https://developer.android.com/develop/ui/compose/why-adopt (visited on 2026-04-24).

[13] Bumptech. *About Glide*. URL: https://bumptech.github.io/glide/.

[14] Bumptech. *Compose*. URL: https://bumptech.github.io/glide/int/compose.html.

[15] Codemia. *What does it mean to inflate a view from an xml file*. URL: https://codemia.io/knowledge-hub/path/what_does_it_mean_to_inflate_a_view_from_an_xml_file.

[16] Coil Contributors. *Overview*. URL: https://coil-kt.github.io/coil/.

[17] elfnochips. *Everything You Need to Know About DO-160 Testing: Ensuring Aerospace Equipment Reliability*. URL: https://medium.com/@einfochips/everything-you-need-to-know-about-do-160-testing-ensuring-aerospace-equipment-reliability-71f21a3306c4.

[18] I. Fjodorovs and S. Kodors. "Jetpack compose and xml layout rendering performance comparison". *Human, Environment, and Technology* (2021). URL: https://journals.rta.lv/index.php/HET/article/view/6779/5638.

[19] W. Ghielens, G. Laghari, and S. Demeyer. *Evaluating the Performance of Android Layout Frameworks: A Case Study*. URL: https://glaghari.github.io/docs/ALF_technical_report_2018.pdf.

[20] JetBrains. *Random*. URL: https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.random/-random.html (visited on 2026-04-22).

[21] Lenovo. *What is a CPU?* URL: https://www.lenovo.com/au/en/glossary/what-is-cpu/.

[22] W. Li, Y. Jiang, C. Xu, Y. Liu, X. Ma, and J. Lü. "Characterizing and detecting inefficient image displaying issues in android apps". In: *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[23] Mordor Intelligence. *NORTH AMERICA COMMERCIAL AIRCRAFT IN-FLIGHT ENTERTAINMENT AND CONNECTIVITY SYSTEM MARKET*. URL: https://www.mordorintelligence.com/industry-reports/north-america-commercial-aircraft-in-flight-entertainment-system-market.

[24] W. Song, M. Han, and J. Huang. "Imgdroid: a static analyzer for detecting image loading defects in android applications". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 164–165. DOI: 10.1109/ICSE-Companion52605.2021.00069.

A Rendered views

Presented below are the different views that were rendered during the project. The view in fig. A.1 was rendered when only one view was used, and all views were alternated between during benchmarks involving alternating views.

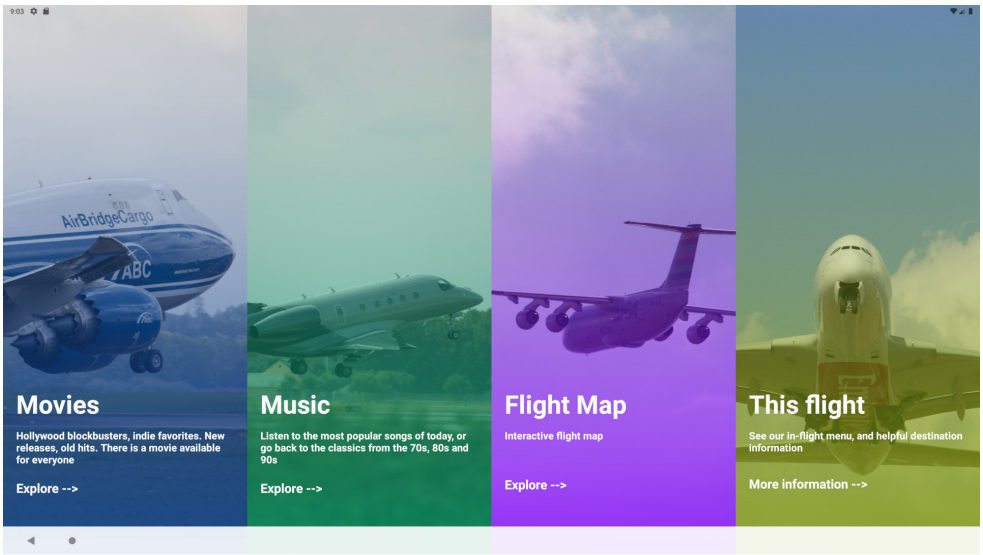

Figure A.1 View designed to resemble the layout of a realistic IFE system

Figure A.2 View with only one image

Figure A.3 View with multiple images

Figure A.4 View with one image, and a gradient overlay with text

Figure A.5 View with multiple images, and a gradient overlay with text

B Benchmark data

B.1 One view

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	132.01	263.85	130.68
ComposeGlide	125.86	231.77	124.79
XMLCoil	137.47	251.24	136.33
XMLGlide	133.37	227.36	132.42

Table B.1 Average Time to Initial Display (ms), one view

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	148.04	745	142.01
ComposeGlide	143.23	830	136.29
XMLCoil	118.38	709	112.41
XMLGlide	115.02	710	109.01

Table B.2 Average Time to Full Display (ms), one view

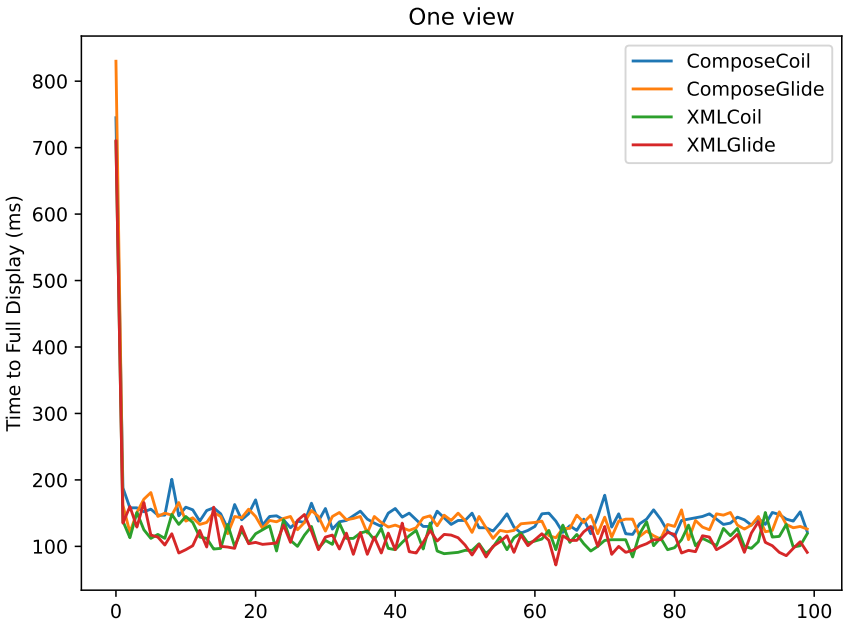

Figure B.1 TTFD results when rendering one view

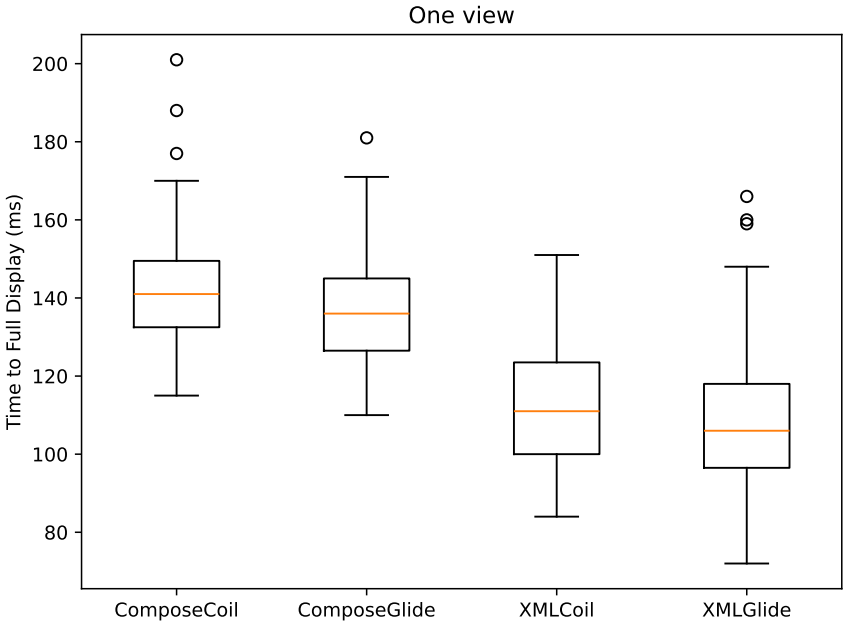

Figure B.2 TTFD results when rendering one view, excluding first render

B.2 Alternating views

Configuration	All iterations	First iterations	Remaining iterations
ComposeCoil	155.19	245.2	154.28
ComposeGlide	110.29	178.67	109.6
XMLCoil	126.94	226.17	125.94
XMLGlide	108.78	150.22	108.36

Table B.3 Average Time to Initial Display (ms), alternating views

Configuration	All iterations	First iterations	Remaining iterations
ComposeCoil	722.1	1380.8	715.44
ComposeGlide	268.51	1306.6	258.03
XMLCoil	631.79	1294.2	625.1
XMLGlide	202.86	1229	192.49

Table B.4 Average Time to Full Display (ms), alternating views

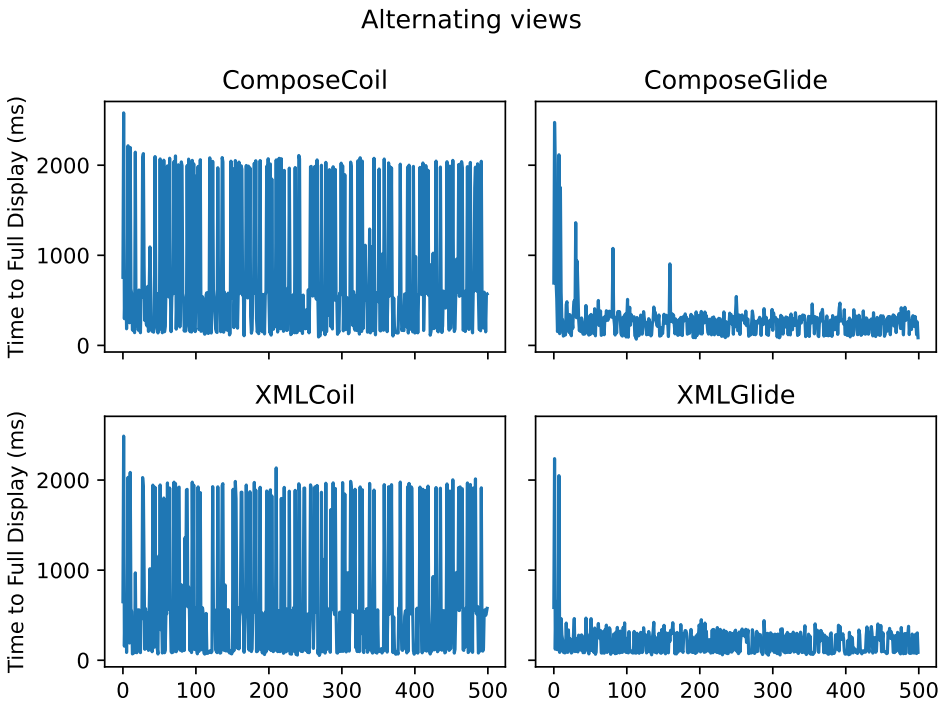

Figure B.3 TTFD results when rendering alternating views

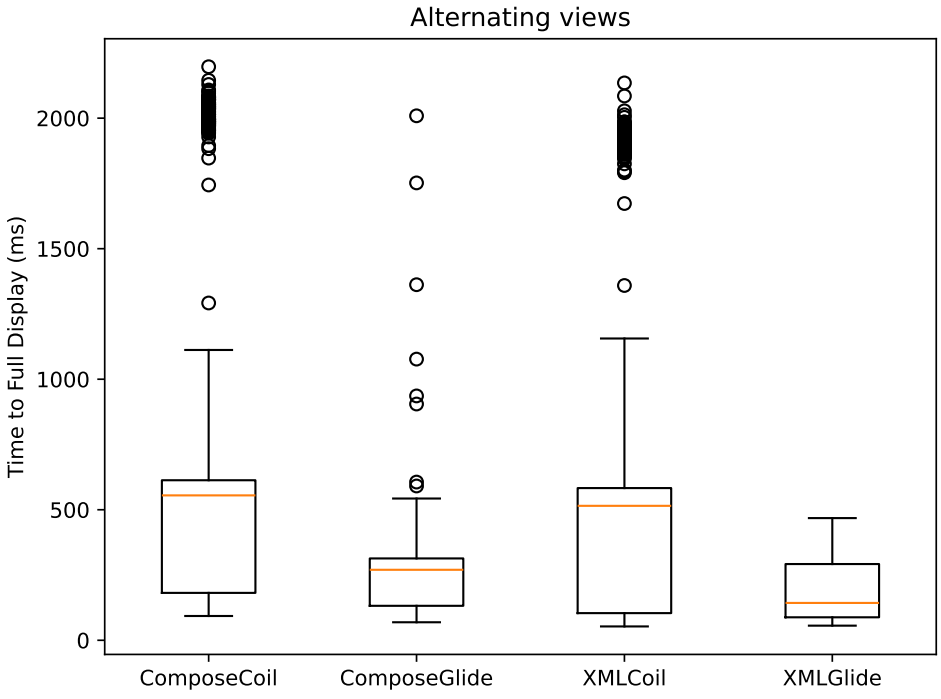

Figure B.4 TTFD results when rendering alternating views, excluding first renders

B.3 Results per view

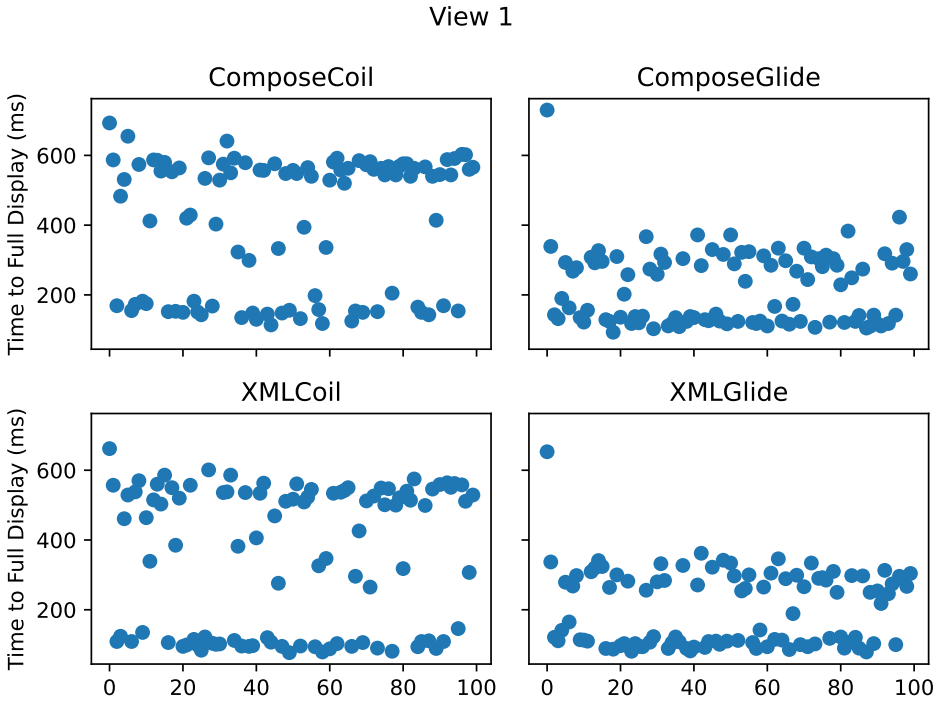

Figure B.5 TTFD results for view 1 when rendered with alternating views

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	412.47	693	409.64
ComposeGlide	219.05	730	213.89
XMLCoil	352.94	662	349.82
XMLGlide	203.66	653	199.12

Table B.5 Average time to full display in ms for view 1

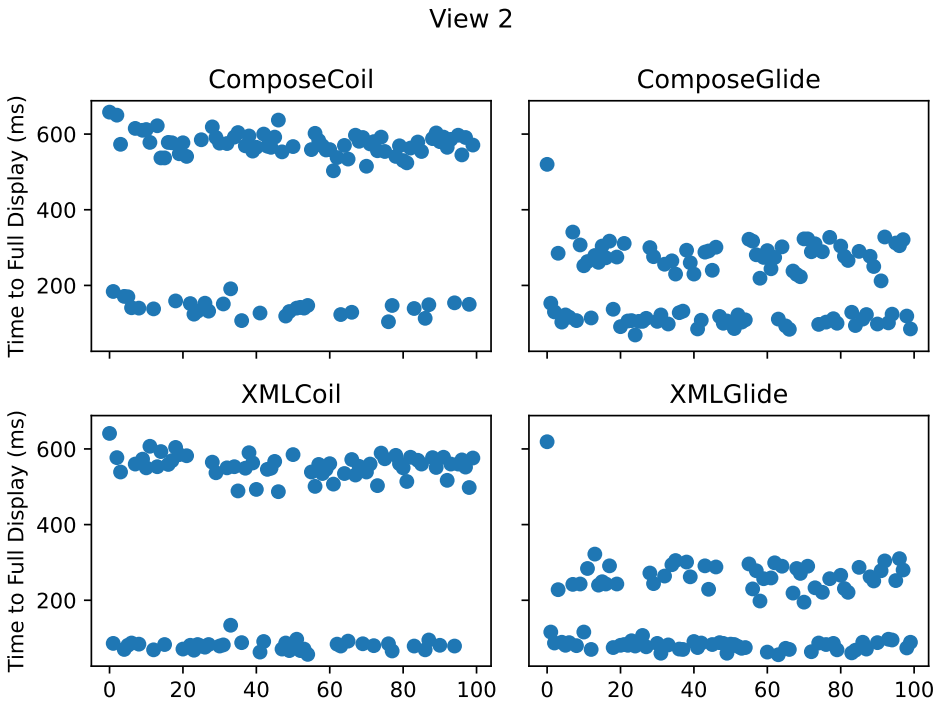

Figure B.6 TTFD results for view 2 when rendered with alternating views

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	441.13	658	438.94
ComposeGlide	204.2	520	201.01
XMLCoil	370.05	641	367.31
XMLGlide	171.8	619	167.28

Table B.6 Average time to full display in ms for view 2

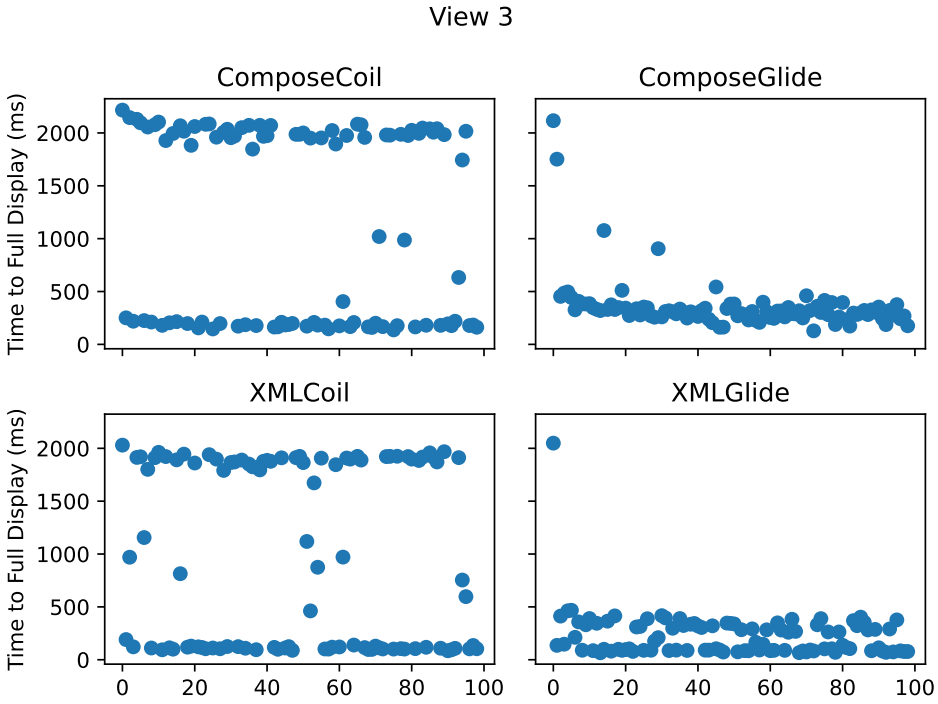

Figure B.7 TTFD results for view 3 when rendered with alternating views

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	1149.72	2216	1138.84
ComposeGlide	358.24	2116	340.31
XMLCoil	989.37	2030	978.76
XMLGlide	229.27	2049	210.7

Table B.7 Average time to full display in ms for view 3

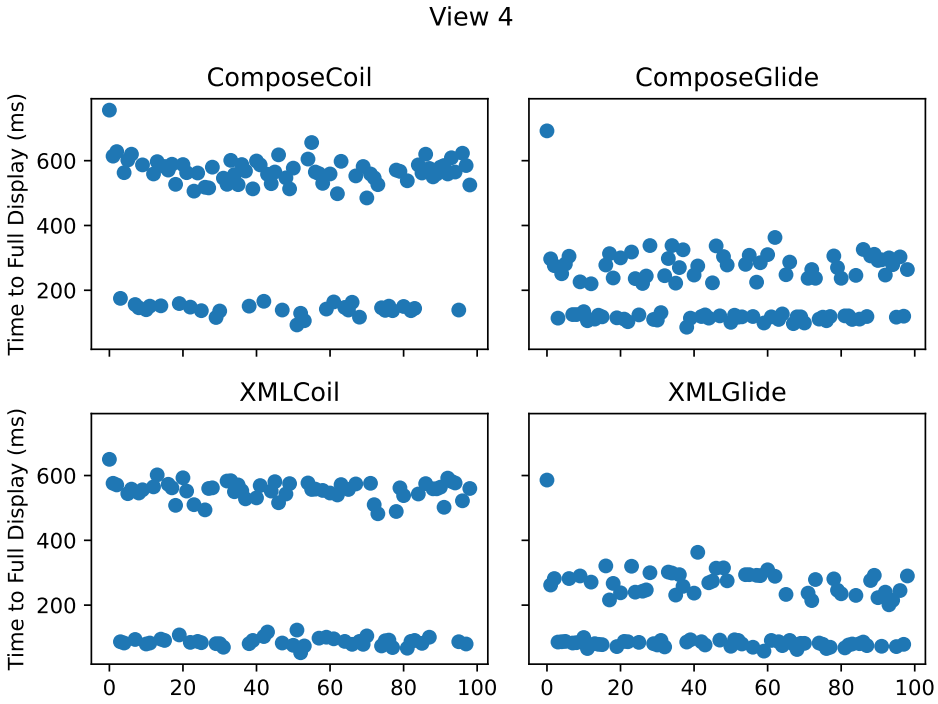

Figure B.8 TTFD results for view 4 when rendered with alternating views

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	435.81	756	432.54
ComposeGlide	208.01	692	203.07
XMLCoil	356.68	650	353.68
XMLGlide	177.03	586	172.86

Table B.8 Average time to full display in ms for view 4

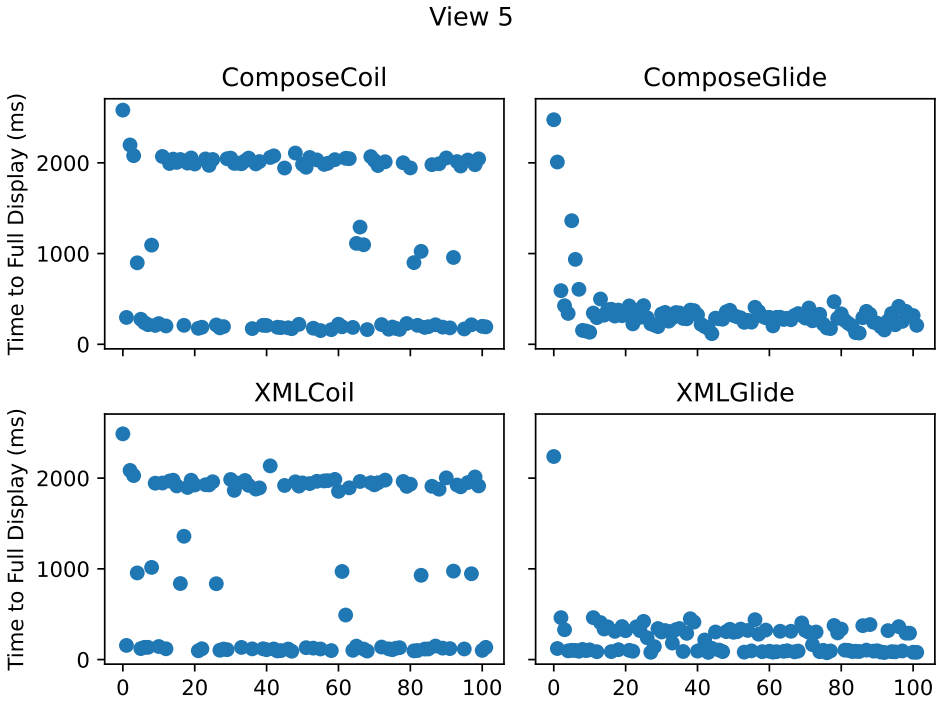

Figure B.9 TTFD results for view 5 when rendered with alternating views

Configuration	All iterations	First iteration	Remaining iterations
ComposeCoil	1163.93	2581	1149.9
ComposeGlide	351.7	2475	330.67
XMLCoil	1081.75	2488	1067.83
XMLGlide	231.96	2238	212.1

Table B.9 Average time to full display in ms for view 5

EXAMENSARBETE Rendering Time Analysis of In-Flight Entertainment Systems
STUDENTER Hugo Nilsson, Andreas Ruggieri
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Christoph Reichenbach (LTH)

Snabbare underhållning på 10 000 meters höjd

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hugo Nilsson, Andreas Ruggieri**

När en flygpassagerare lutar sig tillbaka för att välja film på skärmen framför sig är ett trögt system ett frustrerande bekymmer. Detta arbete har jämfört olika renderingsmetoder, med förhoppningen att bidra till snabbare och mer responsiva in-flight entertainment-system (IFE-system).

Det är ett välkänt problem som inte är unikt för IFE-system. Man trycker på skärmen, ingenting händer, och just som man ska försöka igen blinkar skärmen till och bilder dyker sakta upp en efter en. Det är en vanlig källa till frustration, och det sista man vill utstå när man är fast på ett flygplan i flera timmar och bara vill se en film i lugn och ro. Därför valde vi att undersöka sätt att korta ner laddningstiden, med fokus på just IFE-system.

Vi valde att fokusera på de delar i programmet som ansvarar för att rita det som syns på skärmen: UI-ramverk och bildladdningsbibliotek. UI-ramverket är programmets konstnär, som efter kodens instruktioner planerar hur skärmen ska se ut och sedan målar upp det användaren får se på skärmen. Bildladdningsbiblioteket är sedan ett verktyg som hämtar, anpassar och visar bilder korrekt på skärmen. Det kan dessutom spara undan bearbetade bilder för att snabbare kunna visa dem igen nästa gång de behövs.

Med hjälp av en egenbyggd applikation testade vi olika kombinationer av UI-ramverk och bildladdningsbibliotek på en riktig IFE-enhet, genom att rendera skärmar likt den nedan och mäta hur lång tid det tog. Resultaten från dessa tester gav ingen tydlig vinnare av varken UI-ramverken eller bildladdningsbiblioteken. De visade däremot att det är bildhanteringen som hade störst påverkan på renderingstiden. Första gången en skärm med 25 bilder visades tog det upp till fyra gånger så lång tid som för en skärm med bara en bild. Effektiv bildhantering framstår därför som en nyckel till snabbare och mer responsiva IFE-system.