BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ELECTRONIC FLIGHT BAG

ELECTRONIC FLIGHT BAG

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. LUKÁŠ KÚŠIK**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. PETER CHUDÝ, Ph.D. MBA**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Master's Thesis Specification

Student:	**Kúšik Lukáš, Bc.**
Programme:	Information Technology
Field of study:	Application Development
Title:	**Electronic Flight Bag**
Category:	Computer Graphics

Assignment:

1. Perform a study on the state-of-the-art Electronic Flight Bags.
2. Research key features of an Electronic Flight Bag.
3. Design and implement an Electronic Flight Bag for Android OS.
4. Perform testing and evaluation of the developed Electronic Flight Bag.
5. Suggest future research directions.

Recommended literature:

- Specified by the supervisor

Requirements for the semestral defence:

- Items 1, 2 and partially item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Chudý Peter, doc. Ing., Ph.D. MBA**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	July 30, 2021
Approval date:	October 30, 2020

Abstract

The aim of this thesis is to create an Electronic Flight Bag (EFB) application for Android mobile devices. To accomplish this task, a research is done on the current state of regulations regarding EFBs and the state-of-the-art EFB applications found in the mobile marketplace. Based on this information, an EFB application focused on General Aviation pilots is designed and implemented. The final product contains features for Flight planning, custom aviation Maps, Logbook, Documents, Airport Catalog with global coverage and more. The built-in offline support ensures reliability in real-world conditions. Finally, the product attempts to innovate on existing EFB applications, by including features, such as Automated checklists and Augment Reality Preview.

Abstrakt

Cieľom tejto diplomovej práce je vytvoriť Electronic Flight Bag (EFB) aplikáciu pre mobilné telefóny s operačným systémom Android. Pre splnenie tejto úlohy bola preskúmaná aktuálna legislatíva ohľadom EFB aplikácií spolu s najmodernejšími EFB aplikáciami dostupnými na aplikačnom trhu. Na základe týchto informácií je navrhnutá a implementovaná EFB aplikácia určená pre pilotov všeobecného letectva. Výsledný produkt obsahuje funkcie pre plánovanie letu, vlastnú leteckú mapu, pilotný denník, katalóg letísk s dátami z celého sveta a ďalšie. Podpora offline zaručuje funkčnosť v reálnych podmienkach letu. Konečný produkt sa taktiež snaží inovovať nad existujúcimi EFB aplikáciami zahrnutím funkcionalít, akými sú napríklad automatické kontrolné zoznamy a náhľad v rozšírenej realite.

Keywords

Electronic Flight Bag, EFB, Android, mobilný vývoj, Jetpack Compose, všeobecné letectvo, rozšírená realita, keyword spotting, navigácia, kontrolné zoznamy, váha a vyváženie, pilot, user experience

Kľúčové slová

Electronic Flight Bag, EFB, Android, mobile development, Jetpack Compose, general aviation, augmented reality, keyword spotting, navigation, checklist, weight & balance, pilot

Reference

KÚŠIK, Lukáš. *Electronic Flight Bag.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Peter Chudý, Ph.D. MBA

Rozšírený abstrakt

Počas celej histórie letectva boli súčasťou výbavy každého pilota ťažké tašky s dokumentami a náčrtmi dôležitými pre vykonávanie práce pilota. S postupom vývoja technológií sa stali dostupnými elektronické zariadenia nazývané aj ako Electronic Flight Bag (EFB), ktoré túto situáciu zmenili.

Už prvé zariadenia EFB dokázali ukladať a zobrazovať dokumenty v digitálnej podobe, čím sa značne zjednodušila práca pre pilotov s týmito dokumentami, aj vďaka funkciám, akým je napríklad vyhľadávanie. Dôležitým prínosom z hľadiska aeroliniek je však práve zníženie celkovej hmotnosti lietadla odstránením klasických pilotných tašiek na palube, ktoré môžu vážiť až 20 kg. Zníženie hmotnosti sa priamo premieta do zníženej spotreby paliva, ktorá naprieč celou leteckou flotilou lietadiel môže ušetriť aerolinkám ročne aj niekoľko miliónov dolárov.

Zariadenia EFB, tak ako aj takmer všetky ostatné aspekty letectva, sú regulované sadou pravidiel a úprav. Historicky tieto pravidlá publikovali dva hlavné letecké úrady — Federálny letecký úrad v USA a Agentúra Európskej únie pre bezpečnosť letectva, pre krajiny Európy. Dnes sú tieto pravidlá celosvetovo zjednotené pod správou Medzinárodnej organizácie pre civilné letectvo (ICAO).

Tieto pravidlá rozdeľujú zariadenia EFB na rôzne kategórie z hľadiska prenositeľnosti a kritickosti funkcií ponúkaných týmito zariadeniami pre let. Na základe týchto kategórií sú následne pravidlami definované na zariadenia rôzne požiadavky, a to najmä z pohľadu spoľahlivosti daných zariadení.

Dnešné Electronic Flight Bag zariadenia ponúkajú oveľa viac funkcií ako len zobrazovanie elektronických dokumentov. Súčasťou tejto práce je aj prieskum trhu, ktorého účelom je analyzovať funkcionality, ktoré ponúkajú mobilné aplikácie EFB dostupné dnes na trhoch pre mobilné telefóny.

Cieľom tejto práce je navrhnúť a implementovať podobnú aplikáciu pre mobilné telefóny s operačným systémom Android. Na základe poznatkov získaných z prieskumu trhu bol vytvorený návrh funkcionalít, ktoré by mala výsledná aplikácia obsahovať. Spoločne s ním bol pre aplikáciu navrhnutý aj vlastný grafický dizajn.

Podľa návrhu bola následne vykonaná implementácia aplikácie. Počas implementácie boli využité moderné princípy a nástroje súčasného vývoja pre zariadenia Android, akými sú napríklad nástroj Jetpack Compose alebo nástroj Hilt. Pre projekt bola zvolená modulárna architektúra, ktorá podporuje prehľadnú prácu s jednotlivými časťami projektu.

Výsledná aplikácia s názvom NaviPilot ponúka množstvo užitočných funkcionalít, ktoré sú určené najmä pre pilotov všeobecného letectva. Okrem zobrazovania dokumentov sú medzi nimi funkcie ako plánovač trasy, pilotný denník a sada nástrojov pre uľahčenie výpočtov. Vstavaný plánovač trasy obsahuje aj nástroj pre výpočet vyváženia lietadla pred vzletom, so zabudovanou podporou pre tri populárne typy lietadiel, ktorá sa však dá rozšíriť o vlastné typy lietadiel s pomocou obsiahnutého editora.

Ďalej majú piloti možnosť zistiť informácie o pristávacích dráhach a rádiových frekvenciách letísk z celého sveta, vďaka stiahnuteľným balíčkom určeným pre jednotlivé krajiny. K letiskám si používatelia majú možnosť zobraziť si aj najnovšie správy NOTAM a zistiť aktuálny stav poveternostných podmienok cez kódy METAR, spolu s predpoveďou počasia prostredníctvom radarovej vrstvy. Pre potreby aplikácie bola vytvorená vlastná podkladová mapa, prispôsobená na využívanie pre letectvo. Vďaka vopred spomínaným stiahnuteľným balíčkom je aplikácia funkčná aj v podmienkach bez internetového pripojenia, čo zvyšuje spoľahlivosť aplikácie aj v podmienkach reálneho letu.

Počas letu aplikácia zobrazuje pilotovi polohu lietadla na mapovom podklade, spolu s aktuálnymi letovými informáciami. Tie aplikácia získava zo vstavaného GNSS prijímača v zariadení, avšak aplikáciu je možné taktiež prepojiť aj so simulátorom. Navigačné prostredie aplikácie navádza pilota pri lete na nasledujúce body trasy. Pilot je počas letu podporovaný vytvorenou sadou indikátorov, ktoré predpovedajú dráhu letu lietadla a zobrazujú aktuálny stav vetra pri vzlete a pristátí. K dispozícií je taktiež detekcia fázy letu, ktorá spúšťa vykonávanie vhodného kontrolného zoznamu pre daný typ lietadla, a systém varovaní, ktoré varujú pilota pri vstupe do vzdušných priestorov.

NaviPilot obsahuje taktiež viaceré experimentálne funkcie, akými sú napríklad automatické kontrolné zoznamy a náhľad v rozšírenej realite. Automatické kontrolné zoznamy umožňujú pilotovi prejsť vykonaním kontrolného zoznamu v štýle podobnom tomu pri posádkach kapitána s kopilotom, kedy aplikácia pilotovi jednotlivé položky zoznamu predčítava a pilot ich následne potvrdzuje svojím hlasom. Pre tieto účely bol implementovaný model strojového učenia, ktorý deteguje kľúčové slová špecifikované v kontrolnom zozname prostredníctvom mikrofónu.

Náhľad v rozšírenej realite umožňuje pilotovi vizualizovať trasu letu v priestore miestnosti, v ktorej sa nachádza, prostredníctvom kamery jeho mobilného zariadenia. Po otvorení náhľadu v rozšírenej realite sa na obrazovke telefónu trasa premietne nad skutočným reliéfom krajiny v jej okolí. Táto funkcionalita umožňuje pilotovi naštudovať si terén v okolí trasy letu a zvyšuje jeho povedomie o okolí.

V užívateľskom testovaní sa výsledná aplikácia stretla s pozitívnou odozvou, kde všetci opýtaní užívatelia uviedli, že by aplikáciu osobne využívali pri lietaní, či už v realite alebo na simulátore.

Na záver práce sú prebrané ďalšie možnosti vývoja aplikácie do budúcnosti, v ktorej by sa aplikácia mohla rozšíriť o podporu externých zariadení určených do kokpitu, ktoré dodávajú presnejšie dáta o polohe a orientácii lietadla, akými sú tie zo senzorov mobilných zariadení. Zároveň by sa aplikácia mohla rozšíriť o kvalitnejšie a rozsiahlejšie dáta, ktoré sú však dostupné iba z platených zdrojov.

Electronic Flight Bag

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Associate Professor Peter Chudý, Ph.D., MBA. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Kúšik
August 1, 2021

Acknowledgements

Belongs to Associate Professor Peter Chudý, Ph.D., MBA for sharing his experience, devoting his time to consultations and giving helpful pieces of advice regarding this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

ADS-B Automatic Dependent Surveillance–Broadcast.

AFM Aircraft Flight Manual.

AFMS Airplane Flight Manual Supplement.

AGL Height above Ground Level.

AHRS Attitude and Heading Reference System.

AI Artificial Intelligence.

AIM Aeronautical Information Manual.

AIP Aeronautical Information Publication.

AIRMET Airman's Meteorological Information.

AMM Aircraft Maintenance Manual.

API Application Programming Interface.

AR Augmented Reality.

ATC Air Traffic Control.

ATIS Automatic Terminal Information Service.

CDI Course Deviation Indicator.

CG Center of Gravity.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CS Chart Supplement.

DAO Data Access Object.

DP Instrument Departure Procedure.

EASA European Union Aviation Safety Agency.

ECL Electronic Checklist.

EFB Electronic Flight Bag.

EGNOS European Geostationary Navigation Overlay Service.

ETA Estimated Time of Arrival.

FAA Federal Aviation Administration.

FBO Fixed-base Operator.

FOM Flight Operations Manual.

GA General Aviation.

GNSS Global Navigation Satellite System.

GPS Global Positioning System.

GPX GPS Exchange Format.

IAP Instrument Approach Procedure.

ICAO International Civil Aviation Organization.

IDE Integrated Development Environment.

IFR Instrument Flight Rules.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

KML Keyhole Markup Language.

KWS Keyword Spotting.

LFBE Log-mel Filter Bank Energies.

MEL Minimum Equipment List.

METAR Meteorological Aerodrome Report.

MFCC Mel-frequency Cepstral Coefficients.

ML Machine Learning.

MLW Maximum Landing Weight.

MSL Mean sea level.

MTOW Maximum Takeoff Weight.

NLP Natural Language Processing.

NN Neural Network.

NOTAM Notice to Airmen.

OCR Optical Character Recognition.

OS Operating System.

OSM OpenStreetMap.

PDF Portable Document Format.

PED Portable Electronic Device.

PFD Primary Flight Display.

PIREP Pilot Report.

POH Pilot's Operating Handbook.

POI Point of Interest.

PPM Policy and Procedures Manual.

QRH Quick Reference Handbook.

REST Representational State Transfer.

RNN Recurrent Neural Network.

SAF Storage Access Framework.

SDK Software Development Kit.

SID Standard Instrument Departure.

SOP Standard Operating Procedure.

STAR Standard Terminal Arrival Route.

STC Supplemental Type Certificate.

SV Synthetic Vision.

TAC Terminal Area Charts.

TAF Terminal Aerodrome Forecast.

TC Type Certificate.

TCP/IP Internet Protocol Suite.

TDD Test Driven Development.

TFLite TensorFlow Lite.

TFR Temporary Flight Restriction.

TTS Text-to-Speech.

UI User Interface.

URL Uniform Resource Locator.

UX User Experience.

VFR Visual Flight Rules.

VR Virtual Reality.

VSI Vertical Speed Indicator.

WAAS Wide Area Augmentation System.

XML Extensible Markup Language.

XSD XML Schema Definition.

Chapter 1

Introduction

Throughout the history of aviation, heavy flight bags packed with essential charts and documents were playing a key role in pilots' equipment. With the advancement of technology and the broad availability of portable electronic devices with displays in the last decade, it only made sense to convert these documents to a digital form. The devices used for reading these documents became known as Electronic Flight Bags, or EFBs in short.

In the next chapter (Chapter 2), motivation driving the use of EFBs is explained, as well as the current regulations that apply to the use of EFBs in aviation. As time progressed, EFBs have evolved beyond just document readers. Today, pilots can view dynamic maps with real-time weather information, complete interactive checklists, improve their situational awareness using Synthetic Vision and much more. These features are further examined in Chapter 3, along with a presentation of three real-world EFB applications available on the market.

The goal of this thesis is to design and implement such an application for the Android Operating System (OS). The solution will focus on features useful for the General Aviation (GA) pilots, simplifying the whole process of flying from takeoff to landing. In Chapter 4, the features and the design aspect of the resulting application are proposed, together with a brief introduction to development for the Android mobile platform.

Chapter 5 explains the process of turning the proposal into a fully functional mobile application. The first section of this chapter describes the architecture chosen for the project and the reasoning behind this decision. Later sections provide details on the implementation of the individual application features, logically organized by the name of the screens, as they are found in the application itself.

With the EFB application ready to use, Chapter 6 describes the methods used to test the final product, including the results of user testing conducted with potential users of the application. With the feedback from users in mind, the penultimate Chapter 7 discusses future directions of research and development regarding the application. Finally, the last Chapter 8 concludes this thesis with a short recapitulation of the work that has been done.

Chapter 2

State-of-the-Art Electronic Flight Bag

As its name indicates, Electronic Flight Bag is an evolution of a regular flight bag, which pilots typically carry with them while flying. Inside, one usually finds navigational charts, operational manuals for the aircraft, checklists, and other documents relevant to the task of flying an airplane.

EFBs have been gradually put into use by numerous commercial airlines, in hopes of increasing profits and improving the safety of people on board. By equipping its 11,000 pilots with EFBs, *Delta Air Lines* approximated saving $13 million per year in fuel and associated costs [9]. Fuel savings are the direct result of removing traditional flight bags, which can weigh close to 20 kg.

Even small weight cuts account for considerable fuel consumption reduction, especially when accumulated over thousands of flights per day throughout the whole fleet. *United Airlines* began using a lighter paper for printing their in-flight magazine, reducing its weight by 28 g per magazine. This seemingly minuscule change, together with paper weight reduction of their seat-back service guides, caused a total reduction of 5kg per flight. *United Airlines* states, that this change led to saving 643 000 l of fuel, or $290 000 of fuel costs per year [47].

Besides reducing carbon emissions, EFBs help the environment by eliminating the need to use paper. By introducing EFBs to pilots, *airBaltic* estimated saving 2 million pages of paper every year by replacing their heavy flight bags, containing paper flight details, manuals, navigation, and reference material [1].

In the name of safety, aviation regulations govern nearly everything that can be governed, and EFBs are no exception. In the first half of the decade, two major aviation agencies each published their own regulations for the use of EFBs, the Federal Aviation Administration (FAA) in the United States, and European Union Aviation Safety Agency (EASA) in Europe. Since then, the International Civil Aviation Organization (ICAO) has published its own document standardizing the use of EFBs, to which the two aforementioned agencies adjusted their regulations in their newer editions.

The current documents, used as sources for this thesis, are *AC120-76D* [15] by FAA, *AMC-20* [13] by EASA and *Manual of Electronic Flight Bags* [48] by ICAO.

2.1 Categorization

All three agencies categorize EFBs into a hardware class depending on its mobility. Each agency specifies an EFB as either Portable or Installed. Their definition and rules describing their use are similar across all agencies' documents and the text below uses definitions taken from all three of the documents.

2.1.1 Portable EFB

According to *AMC-20* by EASA, portable devices are defined as „a portable EFB host platform, that is used on the flight deck, and that is not part of the certified aircraft configuration [13].“

As portable EFB devices are not part of the aircraft configuration, they are considered as Portable Electronic Devices (PEDs) and must adhere to their respective rules. Any equipment that can consume electrical energy is considered a PED. Typically, they are consumer devices brought on board by the crew members, passengers, or as part of the cargo, not included in the configuration of the certified aircraft [13].

For a PED to be considered an EFB, it must actively display Types A and/or B software application(s), as described in 2.2. Although a PED might contain software other than EFB, appropriate steps must be taken to prevent interference of non-EFB software to the function of EFB software [15].

The portable EFB can consume electrical energy either from an internal source such as a battery, or externally, by connecting to an aircraft power source [13]. A lithium battery-powered EFB must be properly certified by the manufacturer as to mitigate hazards due to continuous charging of the device, including battery overheat and the risk of leakage [48].

If the portable EFB draws power from an external power source, the power source must be rated for use by the EFB and it should be available to the extent required for its operation. Quick access to turn off the power source by unplugging or a power switch should be available to the operating pilot from a seated position [48].

2.1.2 Installed EFB

Installed EFB, means an EFB host platform that is installed in the aircraft and is considered as an aircraft part. Because of that, it requires full airworthiness approval, like all other aircraft parts [13]. Compared to portable EFBs, installed EFBs are subject to safety assessment addressing failure conditions of the hardware and design control. The approval of these EFBs is included in the aircraft's Type Certificate (TC) or in a Supplemental Type Certificate (STC) [48].

2.2 Applications types

Depending on the criticality of the features offered by the EFB, the requirement for the stability and redundancy of the device varies. The FAA discerns two application types (Type A and Type B). A short description and several examples of applications of each type can be found below, as listed by the *AC 120-76C* document [13].

2.2.1 Type A

Type A applications have a failure condition classification considered to have no effect on safety. They do not substitute for or replace any paper, system, or equipment required by airworthiness or operational regulations and do not require specific authorization for use [13].

The following applications are examples of Type A applications:

- Aircraft parts manuals.
- Minimum Equipment Lists (MELs).
- Federal, state, and airport-specific rules and regulations.
- Chart Supplements (CSs) data.
- Aeronautical Information Publication (AIP).
- Aeronautical Information Manual (AIM).
- Pilot flight and duty-time logs.
- Captain's report (i.e., captain's incident reporting form).
- Aircraft captain's logs.
- Current fuel prices at various airports.
- Computer-based training modules, check pilot, and flight instructor records.
- Airline Policy and Procedures Manuals (PPMs).

2.2.2 Type B

Applications of this type have a failure condition classification considered to be minor. These applications require specific authorization for operational authorization for use. They may substitute paper products of information required for dispatch or to be carried in the aircraft, but they may not substitute for or replace any installed equipment required by airworthiness or operating regulations [13].

The following applications are examples of Type B applications:

- Aircraft Flight Manuals (AFMs) and Airplane Flight Manual Supplement (AFMS).
- Flight Operations Manuals (FOMs).
- Maintenance manuals.
- Company Standard Operating Procedures (SOPs).
- Aircraft operating and information manuals.
- Aircraft performance data manuals (fixed non-interactive material).

- Airport performance restrictions manual (e.g., a reference for takeoff and landing performance calculations).
- Weight and balance calculations.
- Takeoff, en route, approach and landing, missed approach, go-around, performance calculations.
- Cost index modeling/flight optimization planning software.
- Interactive plotting for oceanic and remote navigation.
- Electronic aeronautical charts (e.g., arrival, departure, en route, area, approach, and airport charts).
- Electronic Checklists (ECLs), including normal, abnormal, and emergency.
- Weather and aeronautical information.
- Aircraft cabin and exterior video surveillance displays.
- Aircraft flight log and servicing records.
- Autopilot approach and autoland records.
- Aircraft Maintenance Manuals (AMMs).
- Notices to Airmen (NOTAMs).

Chapter 3

Market Research

3.1 FltPlan Go

Established in 1999, *Fltplan* has grown into the largest flight planning company in North America. More than 165,000 active users use their free, web-based flight planning & filing offering, together with their companion *FltPlan Go* EFB mobile application. Available for both iOS and Android mobile devices, *FltPlan Go* incorporates essential and beneficial features and tools for in-flight and offline use [20].

Figure 3.1: *FltPlan Go*'s split-screen interface on a tablet device [21].

3.1.1 Maps

Aeronautical charts are maps designed to aid pilots to navigate an airplane. They depict important information, such as airspace boundaries, waypoints, radio frequencies, safety hazards, and more. Apart from general charts providing an overview of an area, there are also specific charts describing procedures during different phases of flight, for instance, Standard Instrument Departure (SID) and Standard Terminal Arrival Routes (STARs), described in the next Section 3.1.2.

Before EFBs began being used, pilots used solely paper aeronautical charts for navigation. Naturally, due to the paper's limited size, the map has to be split into multiple sectors. Moreover, each situation requires a different level of detail and there are also specific charts for VFR and IFR types of flight. Aeronautical charts in the U.S. are published by the FAA. These include, but are not limited to [18]:

- Visual Flight Rules (VFR) Navigation Charts — Include cities and towns, roads, railroads, and other distinct visual landmarks.
  - Sectional Aeronautical Charts — Charts designed for visual navigation of slow to medium speed aircraft. Scale 1:500,000.
  - VFR Terminal Area Charts (TAC) — Depict the airspace designated as Class B airspace. While similar to sectional charts, TACs have more detail as their scale is 1:250,000.
  - Caribbean VFR Aeronautical Charts — Designed to assist familiarization of foreign aeronautical and topographic information. Scale 1:1,000,000.
  - U.S. Gulf Coast VFR Aeronautical Chart, Grand Canyon VFR Aeronautical Chart, and Helicopter Route Charts.
- Instrument Flight Rules (IFR) Navigation Charts — Provide aeronautical information for navigation under IFR conditions.
  - IFR En Route Low Altitude Charts — Includes airways; VHF NAVAIDs; limits of controlled airspace; minimum en route and obstruction clearance altitudes; airway distances, etc., below 18,000 feet MSL.
  - IFR En Route High Altitude Charts — Includes the jet route structure; VHF NAVAIDs; selected airports; reporting points, etc., above 18,000 feet MSL.
  - Instrument Departure Procedure — Designed to expedite clearance delivery and to facilitate the transition between takeoff and en route operations.
  - Instrument Approach Procedure Charts — Portray the aeronautical data that is required to execute instrument approaches to airports.
  - Standard Terminal Arrival Route Charts — Described in section 3.1.2.
  - Airport Diagrams — Designed to assist in the movement of ground traffic at locations with complex runway/taxiway configurations.
- Planning Charts — Designed for preflight and en route flight planning for IFR/VFR flights.
- Chart Supplements and Publications.

Taking into advantage the infinite canvas of a digital screen and the large storage size of current portable devices, this quantity of different types of charts can be displayed on an EFB to the pilot in a couple of touches. The pilot does not need to physically list through papers when finding a specific chart or switching to a different section or detail level.

FltPlan Go features a movable map with sectional, en route, TAC, IFR terminal, and other charts, organized into togglable layers. In addition to aeronautical charts, there are layers commonplace in standard map applications, such as street, satellite, and topographic layers. Also available are informational layers of airports, state outlines, and fuel prices.

As per regulations, EFBs are allowed to overlay own-ship position, if available, in relation to the displayed charts. This allows the pilot to better retain spatial awareness when following the flight route or performing maneuvers. As the source of the position, *FltPlan Go* is able to use either the built-in GNSS receiver of the device or connect to a compatible device installed in the cockpit to increase reliability, as discussed in later Subsection 3.3.4.

During the flight, the plane's position is continuously logged and visualized by a feature named „breadcrumbs". Breadcrumbs track the flight path, so that the pilot can replay and review it later, at different playback speeds. The path can be exported which allows the user to examine the path in programs such as Google Earth.

As the pilot nears the approach phase of the route, *FltPlan Go* provides an ability to overlay approach charts directly on the map, as can be seen in Figure 3.1. Combined with the display of the own-ship position, this feature makes it easier for the pilot to see the relevant charts and stay in context during the approach.

The ability to display dynamic information on the map enables the option to display the weather conditions, in real-time. *FltPlan Go* offers the possibility to view weather radar, winds, and METARs as dynamic layers, through the use of color gradients and wind arrows. The pilot can further preview the evolution of weather conditions using animation, which can be helpful in deciding whether to make changes to the flight path because of unpleasant weather conditions.

Lastly, one of the classic purposes of a map is distance measuring. Obtaining a distance measurement using a traditional paper map requires ruler measurements and numerical calculations, which are prone to errors. Using the measurement tool feature of the map in *FltPlan Go* ensures accurate and fast distance measurements, so that the pilot can stay focused on other tasks.

3.1.2 Airport information

The Airport tab compiles useful information about airports saved in *FltPlan Go's* database. The pilot can search for an airport by its name, ID, and city name, or select an airport directly from the map interface. To make it easier to find airports later, airports can be marked as favorites, and searches are saved in history, where the user can quickly go back to their previous queries.

The airport detail screen contains information about the airport's runway length and the ground, tower, and Automatic Terminal Information Service (ATIS) frequencies. Through convenient action buttons, the user can view the airport diagrams described in Section 3.1.1, with the runway, taxiway, and gate details, useful for the pilot before takeoff or after landing. For more information, the user can open the relevant pages of the Chart Supplement (CS) document.

The pilot can examine the current weather situation at the selected airport through METARs, Terminal Aerodrome Forecasts (TAFs), Notices to Airmen (NOTAMs) and Pilot

Reports (PIREPs). They are updated in real-time either through a Wi-Fi connection or through a compatible Automatic Dependent Surveillance–Broadcast (ADS-B) receiver. *FltPlan Go* also includes a convenient feature named „Runway Wind Calculator“, using which the pilot can calculate the tailwind and crosswind from the runway they are using [22].

3.2 Garmin Pilot

Garmin is an American company founded in 1989. Today, with more than 15,000 associates in 80 offices around the world, they bring GPS navigation and wearable technology to the automotive, aviation, marine, outdoor, and fitness markets [26]. Their aviation products include flight decks & displays, autopilots, navigation & radios, flight instruments, sensors, and more. This section will describe some of the features of their EFB application *Garmin Pilot* shown in Figure 3.2, available for iOS and Android devices.

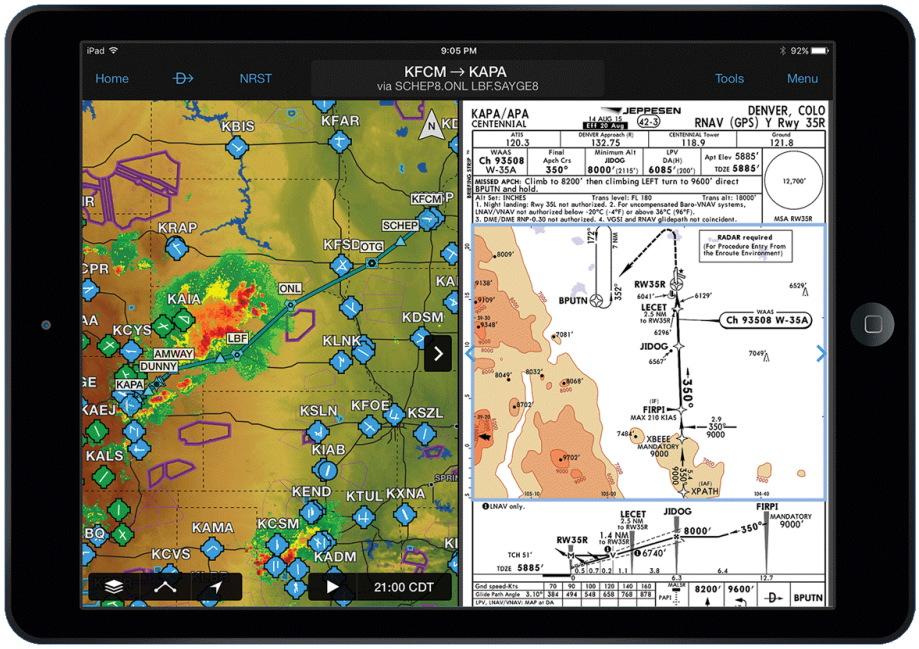


Figure 3.2: *Garmin Pilot's* User Interface [49].

3.2.1 Documents

Historically, the first EFBs simply provided the ability to store paper documents, such as manuals or charts, in a digital form. Today, it still stands as one of the most important features of EFBs, with airlines such as *Qatar Airways* still innovating the approach with the use of centralized documentation management. This system provides a single repository to revise and track all operational and non-operational company manuals, that are then available live on-board on an EFB [3].

Similarly, *Garmin Pilot* offers an option to link to third-party cloud storage services like Dropbox and access any documents stored there. This way, the pilot can upload any set of documents they might need through their computer in advance, or even share documents with other pilots.

While saving space and weight, digital documents are also much easier to update than physical documents, as they do not need to be reprinted and brought on board, simplifying the logistics involved. With the centralized documentation management approach, any change to the documentation is automatically synchronized, ensuring that every procedure or chart is kept up to date.

An immense advantage of electronic documents is the ability to quickly search through them. Especially in situations unfamiliar to the pilot, the possibility to find a key piece of information by searching by keyword instead of flipping through the pages of a paper manual allows the pilot to spend more time resolving the problem, rather than finding a solution to it.

In addition to search, *Garmin Pilot* has the ability to sort and organize documents by marking them with colors. The user can add and remove bookmarks or annotate the files directly by touch or using a stylus. The document can then be shared directly from *Garmin Pilot* to other applications installed on the device [28].

3.2.2 Checklists

A checklist refers to a document used by pilots and flight crew as a tool to verify that all required actions are done in totality and in the correct order. In their paper forms, checklists are usually available to pilots in a Quick Reference Handbook (QRH). A checklist is categorized as normal, abnormal, or emergency, determined by the prevalence and severity of the situation it applies to.

Although a checklist can be stored electronically in the same way as any other document by the EFB (as described in Section 3.2.1), an EFB application can instead present the checklist in a more user-friendly manner while also enabling interaction with the pilot or even with the aircraft.

Garmin Pilot allows the pilot to view checklists arranged by aircraft binders. In each binder belonging to an aircraft, the checklists are tabbed under normal, abnormal, and emergency categories. After choosing a category, a list of checklists is shown by their titles, sorted by the phase of flight they apply to. Finally, tapping on a checklist displays its content on the screen in the form of a list. Each item can be marked either as complete or incomplete, as indicated by a checkmark field next to it.

There are checklists for several aircraft models already included in the application, however, checklists for other aircraft types can be added by entering the information manually. Each checklist is fully customizable, allowing the user to reorder, edit or add new items. A checklist item can be set to initiate an action in the *Garmin Pilot* application, enabling dynamic checklist content tailored to the current situation. For example, the pilot can find the actual frequency of the ground control radio in an After Landing checklist, as determined by the current flight plan.

3.2.3 Synthetic Vision

Synthetic Vision (SV) depicts a forward-looking attitude display of the topography immediately in front of the aircraft. The depicted imagery is derived from the aircraft's attitude, position, and databases of terrain, obstacles, and other relevant features. The Synthetic Vision terrain display shows land contours, large water features, towers, and other obstacles over 200 feet AGL that are included in the obstacle database [28].

Figure 3.3: Synthetic Vision feature of *Garmin Pilot* [27].

As seen in Figure 3.3, the SV feature contains a two-dimensional overlay with flight instruments similar to a Primary Flight Display (PFD), such as an altimeter, airspeed indicator, Vertical Speed Indicator and horizontal situation indicator. When paired with a compatible device capable of receiving ADS-B traffic information described in Section 3.3.4, traffic symbols are displayed at their approximate locations as dots.

The SV is programmed to display visual alerts to indicate the presence of terrain and obstacle threats relevant to the projected flight path. Terrain alerts are displayed in red and yellow shading. Traffic close to the position of the aircraft will result in a traffic alert. As per regulations, Synthetic Vision is intended for situational awareness only and should not form the basis of maneuver decisions for the pilot.

3.3 Foreflight Mobile

Foreflight is a company formed in the year 2007. Serving personal, business, military, commercial, and education sectors, they have established themselves as a leader in the aviation industry, with their EFB offering in the form of mobile flight planning applications [23].

Their flagship product is the *ForeFlight Mobile* application seen in Figure 3.4, aimed at both individual pilots and professional flight crews. It offers features for flight planning, charts, weather, airport information, document management, flight logging, Synthetic Vision, and more. It is available for *iPhone* and *iPad* devices running on the iOS platform.

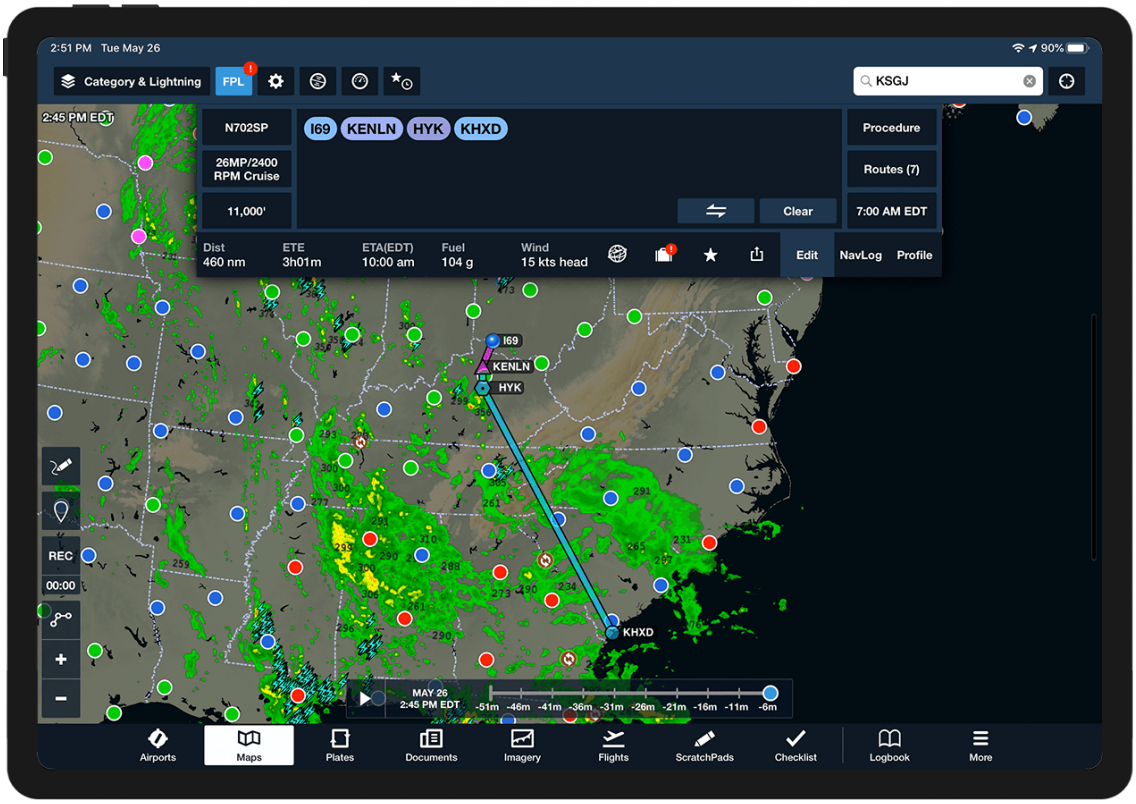


Figure 3.4: Map interface of *Foreflight Mobile* [51].

3.3.1 Flight planning

ForeFlight Mobile offers multiple ways to create a flight route. The user can input the departure and destination either manually, or use the provided search function. A route can also be created directly from the map interface. Once the departure and destination points are chosen, the application will calculate a valid flight route between them, together with the Estimated Time of Arrival (ETA).

The flight time estimate takes into account not only the type of the aircraft, and the selected performance profile, but also real-time predictions of the weather and wind conditions along the route at the time of flight. If the route is planned more than 6-7 days in the future, *ForeFlight* will use historical winds to calculate performance, based on the average wind speed and direction along the planned route over the past 40 years [25].

To choose a different route, the pilot can take advantage of the route advisor. The route advisor window displays multiple potential routes between the selected departure and destination airports, as well as their estimated flight time and fuel consumption. The pilot can then choose the recommended best route overall offered by the route advisor, taking into account not only the fuel and time savings, but also previous ATC clearances in the past.

The altitude advisor feature models winds conditions along the route at various altitudes, showing the net average tailwind, estimated flight duration, and fuel burn per each altitude level. For altitudes unreachable by the aircraft according to its climb performance, the data is not displayed. The pilot can decide to pick the best altitude for their flight based on the recommendation, saving on fuel cost and or time.

The procedure advisor allows the pilot to add Standard Terminal Arrival Route (STAR) procedures, Standard Instrument Departure (SID) procedures, approaches, and VFR traffic patterns to the flight plan. The procedures are selected based on the airports that are part of the current flight plan. Both the Departure and the Arrival button display an inset map of the selected procedures over the main map, helping the pilot quickly review the flight route in the context of the procedure charts.

At the arrival airport, *ForeFlight* optionally displays a list of local Fixed-base Operators (FBOs) and their offerings, for instance, refueling or maintenance services. Next to each FBO, the user can find important information such as the price of jet fuel, phone number, and radio frequency.

3.3.2 3D Preview

This feature allows the user to explore their planned route in an accurately portrayed realistic 3D environment, with photo-real aerial imagery overlaid over high-resolution terrain data. Besides the planning phase, this mode can also be used during the flight, as well as to replay a recorded flight.

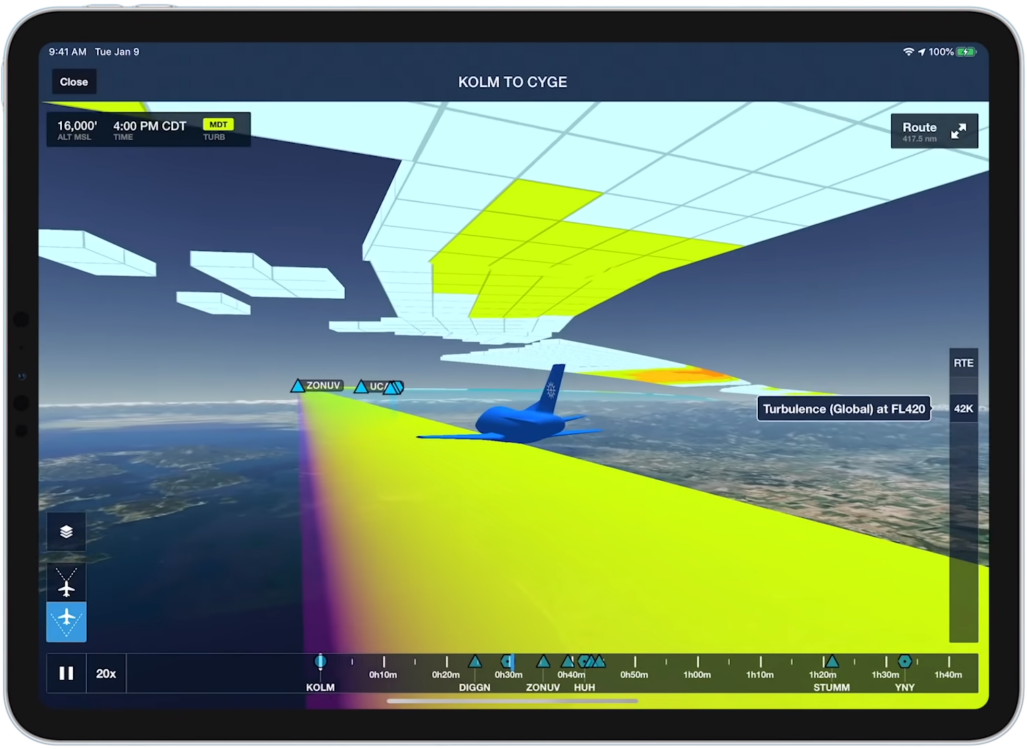


Figure 3.5: *ForeFlight's* 3D Preview mode [24].

By touching and scrubbing the timeline at the bottom of the screen, the user can interactively move in time alongside the planned route. The route is displayed as a line drawn through the 3D space between waypoints as specified by the flight plan. Thanks to the satellite imagery, the pilot can familiarize themselves with ground features and landmarks along the route.

ForeFlight has the ability to visualize the weather conditions throughout the flight, in 3D, as shown in Figure 3.5. This feature allows the pilot to see the icing and turbulence

forecasts at each point of the route ahead of time, exactly as they will be when the aircraft flies through. This helps the pilot quickly prepare for the flight conditions beforehand, or even make changes to the flight plan accordingly to fly around severe weather [25].

3.3.3 Alerts

ForeFlight provides a number of in-app audio and visual alerts that help to keep pilots aware of potential hazards and improve situational awareness in flight and on the ground. Alerts are displayed in the upper third of the screen in a red or beige-colored rectangle, depending on the severity of the alert. An alert persists for several seconds, but it can also be dismissed manually by the pilot by tapping on it. The following are some examples of the alert types in *ForeFlight*, as described in the manual [25]:

Runway Proximity Advisor
Triggers when taxiing near or onto a runway. The system automatically runs in the background and displays the alert regardless of the currently opened windows in the application. When approaching a runway, the system provides an alert that includes the name of the runway. Upon entering the runway, the system also displays the length of the remaining runway.

Cabin Altitude Advisor
If the device is equipped with a separate barometric pressure sensor or connected to an external device from Section 3.3.4, the system monitors the cabin pressure and provides alerts when passing 12 000 ft MSL and 25 000 ft MSL.

Terrain/Obstacle Alerts
Using the current position and terrain data, the system displays an alert dialog when it detects an obstacle close to the aircraft. The dialog features an overview map with the plane position and hazard areas highlighted. When nearby airports or approach paths, the alert sensitivity is reduced in order to reduce nuisance alerts.

Temporary Flight Restriction Alerts
When nearing a known Temporary Flight Restriction (TFR) zone, the system provides an alert warning the pilot of its position with a warning „TFR Ahead“, „TFR Below“, „TFR Above“ and „Inside TFR“. The system also warns of soon-to-be active TFR zones, if they are set to become active within the next five minutes.

Overheat Alerts
To prevent unwanted shutdowns, *ForeFlight* monitors the device temperature and displays an alert when it is overheating. The pilot can then take measures to cool down the device, such as moving the device out of direct sunlight or lowering the screen brightness.

3.3.4 External devices connectivity

The device running an EFB application usually possesses a limited set of receivers and sensors useful for building a model of the current flight situation. These generally include a GNSS receiver for positional information, magnetometer, accelerometer, and sometimes also a separate barometer. Nevertheless, the device sensors and receivers may not be

suitable for use in the conditions inside the aircraft. As an example, the presence of metal in the cockpit can introduce errors in the compass heading readings. As an additional example, the cockpit canopy might partially block the GNSS signal from reaching the device, causing it to lose position fix.

In order to improve and or extend the data provided to the EFB applications, there are plenty of portable devices on the market that serve this purpose. One of these devices, supported by *ForeFlight*, is *Sentry*, shown in Figure 3.6.

Figure 3.6: *Sentry*, a portable ADS-B & GNSS Receiver [71].

Sentry includes a highly accurate, reliable GNSS receiver with support for Wide Area Augmentation System (WAAS), an extremely accurate navigation system developed for civil aviation. It uses a system of precisely surveyed ground stations to provide corrections to the GNSS navigation signal, caused by GNSS satellite orbit drift and signal delays caused by the ionosphere. These correction messages are then broadcast through communication satellites to receivers onboard aircraft using the same frequency as GNSS [17]. This system is available in North America, although similar systems are in use in other regions, such as the European Geostationary Navigation Overlay Service (EGNOS) in Europe.

Next after position, *ForeFlight* can utilize the built-in Attitude and Heading Reference System (AHRS) for additional data regarding the aircraft's attitude. This data can be used as a source by Synthetic Vision systems described in Subsection 3.2.3. To ensure that the data is accurate, the pilot needs to first calibrate the device in the application by specifying the *Sentry's* mount location and leaving the device stationary [25].

Moreover, *Sentry* contains an Automatic Dependent Surveillance–Broadcast (ADS-B) receiver. Using this technology, aircraft can broadcast their position to other aircraft, which in turn can be displayed in EFB applications. Apart from traffic information, numerous weather-related data can be received through the ADS-B receiver, which is convenient in situations without an internet connection. *ForeFlight's* support includes the following ADS-B weather products: [25]

- Radar, lightning, turbulence, cloud tops.
- Meteorological Aerodrome Reports (METARs).
- Terminal Aerodrome Forecasts (TAFs).
- Temporary Flight Restrictions (TFRs).
- Pilot Reports (PIREPs).

- Airman's Meteorological Information (AIRMET).

Finally, *Sentry* includes a built-in carbon monoxide sensor. When the CO level reaches hazardous levels, it raises an audio and in-app alarm. *Sentry* allows to connect up to five Wi-Fi devices. Passengers or the co-pilot can connect to the device and receive route, weather, and traffic information, helping the pilot stay focused on the task at hand by being on the lookout for any possible hazards during the flight [71].

Chapter 4

Application Design and Specification

This chapter serves as a preparation for the realization of the final application. After an introduction to Android development and its latest trends, a set of features that should be implemented in the resulting product are defined. Finally, based on the proposed features, a graphic design of the User Interface is put forward.

4.1 Android development

Android is an open-source operating system designed for smartphones and tablets. The first public version was initially released in 2008, together with the first commercially available Android device *HTC Dream*. Developed by the *Open Handset Alliance* led by *Google*, Android has since then been updated regularly with new features for both users and developers. As of today, Android has grown to be the best-selling mobile operating system globally with over 70 % market share [67]. This can be attributed largely to the abundance of device models that run on Android in all price ranges, making it accessible to the vast population.

An integral part of the Android operating system is its application ecosystem. Users can download nearly 3 billion applications from the *Google Play* store, spanning many categories, from social networks to niche utility tools [2]. As Android devices come in many different form factors, display resolution and equipment, development for Android poses a unique challenge in ensuring that the product is compatible with as many devices as possible.

Fortunately, developers do not have to acquire various physical devices in order to test their code on a plethora of devices. Instead, they can use the provided *Android Emulator* which emulates a physical mobile device, with configurable parameters such as screen size or display resolution. Moreover, the developer can control various device sensors and receivers, such as an GNSS receiver or an accelerometer, in order to test applications that interact with the real-world environment on their local development machine.

Applications for Android are developed using the *Android Software Development Kit (SDK)* in *Java*, *Kotlin* and *C++* programming languages. The official Integrated Development Environment (IDE) for Android is the *Android Studio*. Built upon *JetBrains IntelliJ IDEA*, it has been created specifically for Android development. It includes a layout editor,

real-time profilers, built-in code linter, translation editor, and other tools, simplifying the development experience.

Kotlin is a modern programming language developed by *JetBrains* and has been the preferred development language for Android since 2019. Compared to *Java*, it provides several language features which result in shorter and cleaner code. Migrating project codebase from *Java* to *Kotlin* is very straightforward, as *Kotlin* is fully interoperable with *Java* and compiles to JVM bytecode.

One of the many practical features, that *Kotlin* supports, are coroutines. Coroutines are essentially lightweight threads with minimal overhead compared to normal threads. Long-running operations in a coroutine can be suspending instead of blocking, allowing the calling thread to continue executing code. This non-blocking way of performing asynchronous tasks is an effective way to solve the common problem of loading data to be displayed in User Interfaces, without blocking the User Interface thread itself.

Particular problem developers face when developing applications for Android, is the fragmentation of Android versions on consumer devices. When a new Android version is released, it takes a while before new devices with the latest version hit the market and get in hands of the users. Existing devices can take months to receive an update to the new version, provided that the manufacturer decides it is worth supporting the device in the first place.

Fragmentation leads to a significant lag in the availability of new features, that become available in the latest Android versions. Since the actual share of devices running the latest version is small, developers are discouraged to implement these new features in their applications, if they can work on features that apply to all of their users, instead of just a fraction of them. *Google* is attempting to eliminate the issue of fragmentation by efforts, such as the *Project Treble*, where they were able to raise the adoption of the newest Android version just before the release of the next version from 8.9 % to 22.6 % [55].

With the aim of bringing new features to older platform versions, *Google* created the *AndroidX* library. This library provides backward compatibility for many new features introduced to the Android platform, providing a feature parity between old and new versions of Android. Developers use this library heavily, as it allows them to target a higher number of users, while also benefiting from the latest features and fixes.

In an effort to improve developer experience, Android provides a suite of libraries under the name *Android Jetpack* [31]. This set of libraries helps developers to follow best practices, reduce boilerplate code and write code that works consistently across Android versions. The most notable libraries included in *Jetpack* are:

- Room — Data persistence backed by an *SQLite* database.
- Work — Background task scheduling and execution.
- Navigation — Improves structuring of UI screens and the navigation between them.
- Databinding — Enables binding UI components to data sources in a declarative format.
- Camera — Simplifies working with device camera through providing a compatibility interface.
- Material Design Components — A set of UI components implemented according to the Material Design [36].

4.2 Application features

The resulting application of this project should be built upon the knowledge gathered from the conducted market research in Chapter 3. The existing applications already approach the subject in a matter that is familiar to pilots, who are the main target group for this application. Because of that, the resulting application should not deviate too much regarding the core concepts, but instead, should extend their functionality furthermore.

As the EFB application is developed for mobile devices, it is considered to be a Portable EFB and should adhere to their respective rules, as described in Section 2.1.1. As a guide, the Manual of Electronic Flight Bags (EFBs) published by ICAO should be used, since it includes many recommendations with respect to the development of EFB software in Appendix A [48].

The principal concept of the EFB application should revolve around simplicity in terms of the workload placed on the pilot when operating the application. This should be achieved by leaving the user to perform only the actions necessary, and automating the rest as applicable. The application should try to display useful information and offer appropriate actions to the pilot based on the current context. Below is a description of features that should be offered by the application, in a way that fulfills this goal concept.

Documents

Starting from the core EFB features, the Documents feature should allow the pilot to read digital documents such as Aircraft Flight Manuals (AFMs), Quick Reference Handbooks (QRHs) or any other documents as needed by the pilot. Similar to other implementations detailed in Section 3.2.1, the interface should allow the user to organize the documents into categories and customize their order. The application may already provide several widely available documents, but the user should be able to upload documents of their own, or at least connect the application to his cloud storage. Moreover, the documents should be searchable by the user. If a document does not include a textual representation of its content, an Optical Character Recognition (OCR) should be performed to enable the search function.

Checklists

Next, the Checklists feature should allow the user to add and edit checklists, as described in Section 3.2.2. Users should be able to split checklists into categories and edit their content. The option to go through a checklist should automatically be offered to the pilot when applicable. For example, if the system recognizes that the pilot is on a final approach, it should offer the pilot an option to go through the before-landing checklist.

In-flight crews composed of the captain and a co-pilot, the document *120-71B* by the FAA specifies that flight-related checklists should be accomplished by one crew member reading the checklist and the second crewmember confirming and responding to each item, as appropriate [16]. The application should attempt to imitate this practice and include a Text-to-Speech (TTS) feature that will read the checklist item aloud. Subsequently, it should listen for the correct response from the pilot, as described by the checklist, using speech recognition. The pilot should therefore be able to complete a checklist using his voice only, without being distracted from controlling the aircraft.

Airport catalog

As presented in the airport information Section 3.1.2, there are many pieces of information linked to an airport that are important for the pilot. A user should be able to find all this information in one place, in a comprehensive screen including the general data about the airport, runways, and frequencies. In addition, the screen should display the current weather information and METAR, as well as relevant documents regarding the airport's procedures and current NOTAMs. The airport information window should be accessible either directly from the map, or from a searchable database of airports.

Flight planning

Before a flight, the pilot must be able to enter their destination into the system. The application may then automatically compute a suitable flight route through waypoints as defined by Aeronautical Information Publication (AIP), taking into account the aircraft performance data and load weight, as provided by the pilot. An option to manually input the route waypoints should also be available. The airport and waypoint input fields should provide auto-complete functionality to make the selection process easier. All flights should be automatically saved as drafts, so that the pilot can plan a flight at home and return to it later before the flight. The complete flight plan is then passed on to the Dashboard interface described next.

Dashboard / Map

The main feature of the final EFB application will be the Dashboard. During the flight, this mode should display basic flight information, such as the current position, altitude, heading and speed. This information should be sourced from the device's built-in GNSS receiver and barometer, although support for connecting to an external device discussed in Section 3.3.4 may also be implemented.

The aircraft's position should be depicted on a movable map, according to the display of own-ship regulations in Section 2.2. The map should contain important waypoints, points of interest, and areas of interest relevant to the pilot. Since the internet connection might not be available in-flight, the map data should be available offline.

The current route, as defined by the flight plan, should be drawn on the map with its waypoints highlighted. Information about the next waypoint on the route, such as the name, distance, and ETA should be visible to the pilot. As the aircraft moves, it should leave a visual breadcrumb trail behind the aircraft on the map, with distinctive color-coded segments tracking the altitude. In order to help the pilot approximate the turn radius, a turning rate indicator should be drawn at the position of the aircraft.

Having access to real-time weather data, visual areas of precipitation and clouds should be highlighted on the map, to help the pilot navigate around areas with undesirable weather. Wind direction and speed should be visualized appropriately. The weather layer may be toggleable by the user. An alert should be raised when nearing an area with harsh weather conditions.

If available, real-time traffic may be displayed on the map. Other aircraft should be properly distinguished from own aircraft. Each aircraft should be marked with its call sign, current altitude, and its flight heading. Traffic information may be sourced from the internet, or from an ADS-B receiver. If another aircraft comes to close proximity to the pilot's own aircraft, a traffic alert should be raised.

Various alerts should be displayed and read aloud to the pilot when raised. These alerts may include the aforementioned weather and traffic alerts, terrain proximity alerts, airspace border alerts, and others. The alerts should be dismissible when unneeded and have a certain mute period, in order to prevent the alert from becoming an annoyance.

Calculations

Conversions between various units should be built into the application, in an accessible manner. Conversions should be available between quantities used commonly by pilots, such as temperature, velocity, mass, and volume. Other practical tools, for instance, a runway wind or density altitude calculator, may be available too.

Logbook and replays

Flights marked as finished will be a part of the Logbook. The logbook should provide basic statistics about the total number and duration of all flights. The user should have the ability to replay any past flight. Choosing to replay a flight should open the Dashboard view, with the past flight data used as the source of information. In replay mode, the position of the aircraft is replayed as recorded. The replay speed should be selectable and the user should have the ability to manually seek through the timeline. Lastly, the pilot should have the option to export the flight path to GPS Exchange Format (GPX) or Keyhole Markup Language (KML) files.

Connection with simulators

Today's flight simulators often model the real world to great accuracy, including, but not limited to, airports, terrain, aircraft, and navigational waypoints. Hence, the EFB application can also double as a companion for simulator pilots. After connecting the application to a simulator, the system will receive data about the position of the aircraft in the simulation and display it on the map, as if the position came from the built-in GNSS receiver. Apart from simulator pilots, this feature will also be essential for testing and debugging purposes.

Offline support

A working internet connection is not guaranteed in the typical environment where the application is expected to be used. Because of that, all features that use data downloaded from the internet have to be carefully considered to support offline conditions gracefully. At the least, the User Interface should explicitly inform the user that a feature is not available when offline. However, the best solution would be to make the feature work in full capacity even without an internet connection, through techniques to the likes of pre-loading and disk caching.

4.3 User Interface

The design of the User Interface should take into consideration the normal conditions during the application's use. The color palette should be chosen appropriately to ensure sufficient contrast in the brightly lit environment of the plane's cockpit. Even though an effort is put into automating common actions through the use of voice interaction, any UI elements

still needed should be adequately large for the pilot to interact with, when the device is mounted in the cockpit.

Upon launching the application, the user is greeted with the Home screen. The first of the screens in Figure 4.1, the Home screen immediately draws an eye to the button for creating a new flight, pulling the user deeper into the application through the main flow. For quick access, the most recent saved and finished flights are displayed, with the option to jump to a list of all flights (Logbook). Furthermore, navigational buttons for documents, airport catalog, tools, and settings are placed at the bottom of the screen.

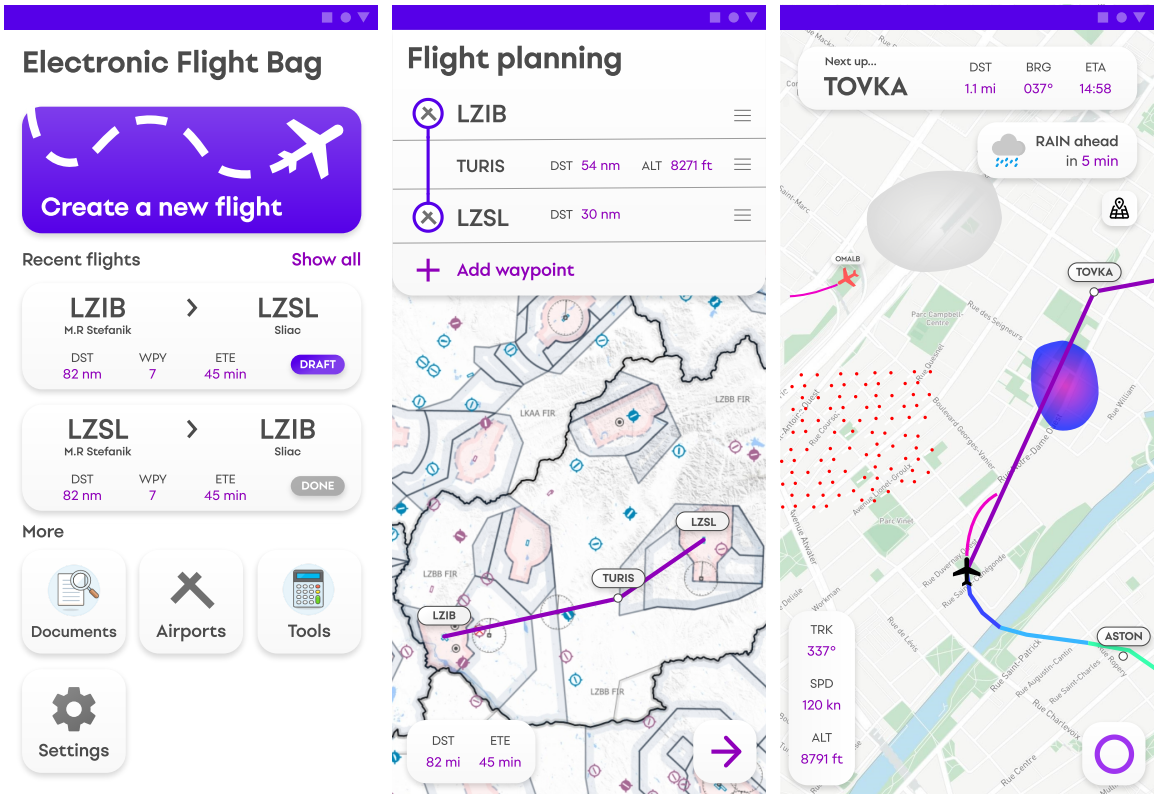


Figure 4.1: Proposed graphic design of the main application screens.

When creating a flight or editing an existing one, the user encounters a familiar interface inspired by popular car navigation applications. The flight planning screen in the middle of Figure 4.1, consists of a movable map with the current flight route highlighted, together with a waypoint list. The user can set their destination, add new waypoints or reorder/delete existing ones. The altitude of each waypoint can be configured here as well. At the bottom, the total flight distance and the estimated time duration of the flight are displayed.

The Dashboard screen, shown on the right in Figure 4.1, is the primary screen used during a flight. It is, again, inspired by established design principles found in car navigation systems, in order to feel intuitive to the pilot. Its main element is an interactive map, automatically centered on the plane's position, rotated in the direction of the plane's heading. The current flight route is highlighted by a distinctive line, with flight route waypoints marked by their names. The map is capable of displaying weather elements in the form of colored areas signifying the presence of rain and clouds. The wind is designed to be depicted using small flowing particles, from which the wind speed and direction can be read.

At the very top, the next waypoint on the route is displayed, along with information about the distance, bearing and ETA located next to it. In the lower-left corner, the current aircraft speed, altitude, and track are displayed. The lower right corner is occupied by an action button that opens a dialog box containing buttons for easy access to Documents, the Airport catalog, and Tools screens. When an alert is raised during the flight, a bubble, color-coded based on the alert's severity, is shown in the upper part of the screen. When there are multiple alerts present, they will be automatically reordered based on their priority.

Chapter 5

Application Implementation

This chapter introduces the final EFB application, named NaviPilot. The first section discusses the architectural approach, as well as key paradigms and tools that have been chosen for the project. The following sections present the individual proposed features as they are implemented. Also included are the concepts of the Android Operating System and the libraries, that were used during the development of the application.

5.1 Project architecture

The development project is split into modules constituting a hierarchy of the main application module, which depends on several supporting sub-modules. Each sub-module is an independent entity, which contains classes and resources required for a specific use-case. Modularization results in a cleaner project structure, encourages code reuse, as well as reduces the build times by avoiding recompilation of unchanged modules during development iterations. Figure 5.1 portrays a diagram of the structure of the project's modules.

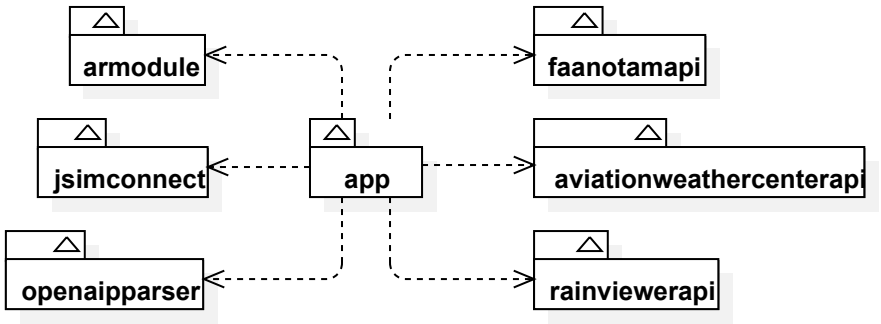


Figure 5.1: Diagram of the NaviPilot project modules.

Gradle is an open-source build automation tool [44]. It is the default build system for Android development and it is used as the build system of this project as well. It allows configuring the project and its dependencies, through domain-specific configuration files written in *Groovy* or *Kotlin* languages. Each module can specify external libraries from online maven repositories and other project modules as its dependencies. Upon project compilation, *Gradle* automatically computes the dependency graph of the modules, handles transitive dependencies, and downloads missing artifacts as necessary.

The main module of the project is the *app* module. It contains the core application code for UI, application business logic, and interaction with the database. Individual classes are packaged logically by the application screen to which they apply. Resources, such as fonts, graphics, and strings are saved in the default resources folder, as is the standard practice in Android projects. The Android resource system is flexible in the capability of providing different resources based on the context of the device size, resolution, language and more.

Navigation

Since Android's very first release, Android applications had used Activities as the main unit of the User Interface. The Android documentation describes an Activity as a single, focused thing that the user can do [30]. Each application screen would typically be represented by its own Activity class, which would then be created and disposed during runtime as needed when the user wants to navigate between screens. Activities have a lifecycle that is ultimately managed by the OS. This means, that an Activity can potentially be disposed anytime by the system when resources need to be freed.

Developers need to be prepared to handle the occurrence of these lifecycle events. An activity moves through several states as it is created, moved into the foreground, paused, stopped, and destroyed. The system provides callbacks for these transitions and it is the responsibility of the developer to graciously take care of these events, in order to prevent any crashes or loss of data.

The Honeycomb version of Android introduced Fragments. They are designed to be smaller and more modular UI elements compared to Activities. An Activity can contain one or more Fragments, which allows creating composite User Interfaces that are particularly suitable for the then-upcoming tablet devices with large screens. With Fragments, multiple variants of a User Interface can be created, showing either a single full-screen view or a master-detail layout when the screen is sufficiently large.

Nowadays, the current trend in modern Android development takes advantage of the Navigation library from the Jetpack suit [39]. Using Navigation, developers can create entire navigation graphs connecting individual content areas, called destinations. Destinations may be whole screens or individual pages of a tab component. Each destination can define its arguments, required or optional, which in combination with the *SafeArgs* plugin enables writing navigation code in a type-safe manner.

The Navigation library shifts the Android UI paradigm from multiple Activities made for each screen, to one where the whole application consists of a single Activity, which contains only a Navigation host container. The container is used to display Fragments, that are swapped in and out as necessary by a Navigation controller. The Navigation library provides several advantages compared to traditional approaches, such as automatic handling of Fragment transactions, support for back navigation, built-in transition animations, and integrated support for deep links.

NaviPilot uses the Navigation library to conduct navigation between application screens. The project's navigation graph is saved in the *nav_graph.xml* file stored in the resources folder. Opening the file in the Android Studio IDE displays a window containing a visual representation of the graph. The visual editor allows the developer to easily create new destinations, define their arguments, and wire transitions between destinations using the mouse pointer.

Jetpack Compose

Traditionally, Android had used a view system consisting of widgets called Views. These were the building blocks of any User Interface on Android, providing common UI elements such as *TextView*, *Button* and *Checkbox*. To arrange these widgets, several types of layouts were available, like the *LinearLayout*, *RelativeLayout* and *FrameLayout*. Although these layouts were simple on their own, developers could nevertheless serve more complex use-cases by nesting the layouts into themselves. To provide support for complex layouts out-of-the-box, the Android team released the *ConstraintLayout* library in 2017. It featured a powerful constraint-based system that could create UI layouts that were hard to implement using the simple layout types.

As the Android ecosystem developed, many design flaws of the view system were progressively more apparent. Due to the view framework being tightly coupled to the core Android environment, any drastic reworks of the view system had to be avoided in order to ensure compatibility with old versions. This meant that the technical debt kept on growing, while only minor issues could be solved in hopes of keeping the system maintained. In this situation, it only made sense to develop a new solution for UI development, now called Jetpack Compose [34].

Part of the *AndroidX* suite of libraries, Jetpack Compose aims to redefine the way User Interfaces are created in Android development. Compose introduces a declarative way of building UIs, where the developer describes how to transform a certain state to pixels on the screen. Compared to an imperative approach, this results in less code and is more intuitive to the developer, since any changes to the state are automatically handled by Compose which results in the UI being updated.

The basic unit of a Compose view is a *Composable*. A *Composable* can be any widget or a layout containing other *Composables*. The Compose library provides several default widgets for many daily use-cases, however, the developer can easily create their own. In contrast to the old view framework which primarily used XML files to define Views, *Composables* are defined in code by writing a method with the *@Composable* annotation.

Compose leverages many features of the Kotlin language, namely named arguments and default parameters. Migrating to Compose is effortless since Compose provides interoperability with the old view system. Classic Views can be used in Compose and vice versa, enabling an incremental migration of the codebase. During the development of this project, the Compose library was in the beta stage of development. Nevertheless, NaviPilot uses Compose for the vast majority of its User Interface.

ViewModel

In Android, UI-based classes, such as Activities and Fragments, are recreated during configuration changes. If the view layer is responsible for loading information from the data layer, the user could lose any state stored in the screen by changing the phone's orientation from portrait to landscape. This would result in a very poor user experience and must therefore be avoided.

As recommended by the best practices guide for Android development published by Google, a ViewModel layer is introduced to act as a bridge between the UI and the data layer [37]. To implement the ViewModel layer, the ViewModel class of the Jetpack ViewModel library is used [43]. The ViewModel class is designed to survive configuration changes, allowing to hold the information even in the event of screen rotation. The re-

sulting separation of concern also aids in testability, as each layer can be mocked as needed, making it possible to test each component in separation.

Reactive programming

Reactive programming is a paradigm describing the way information is passed between components. Data often form an asynchronous stream of events, for instance, a stream of locations originating from a GNSS receiver. If the goal is to display the current location in the UI, reading the location from the stream only once and displaying it is not sufficient. Periodically polling for the latest location in a set interval is not perfect either, since the location data may become available sporadically. A low polling interval could result in some updates being skipped while setting a polling interval too high would be inefficient.

What is needed is an event-based observer pattern, where the observer sends a message to its subscribers when new data is available. In Java/Kotlin, this behavior can easily be achieved by using e.g. callbacks or the Java *Observer* interface. In Android, however, views and other components have their own specific lifecycles. In cases when the view is disposed, any callbacks that were not unregistered may get leaked. This may result in an application crash when the disposed view would attempt to be updated when new data is received.

Kotlin's coroutines are lifecycle-aware by default. All coroutines must run in a lifecycle scope. When the lifecycle scope is disposed, all coroutines and their children are automatically canceled. The *AndroidX* Lifecycle library [38] provides lifecycle implementations for Android platform components such as Fragments and Activities. These lifecycle entities are tied to the component's own lifecycles, so that when the component is disposed, the coroutine's lifecycle scope is disposed as well, allowing the coroutine to clean up graciously.

This behavior is useful not only to automatically cancel lifecycle-sensitive subscriptions, but also to stop any long-running background operations in general, running in coroutines. Asynchronous operations, which result in a single deferred value, are constructed with the *async* function block. In order to return multiple values in an asynchronous stream, the Kotlin coroutines library offers an entity called *Flow*.

A *Flow* is an observable which carries the benefits of the coroutine system. Subscribers observing a *Flow* are automatically canceled when their lifecycle scope is disposed, which greatly reduces application errors and results in a leaner code. Subscribers and *Flows* use the suspend mechanism to synchronize during the event collection, avoiding thread blocking. Therefore, *Flows* can be used on the main thread which also renders the UI, without causing it to freeze or jank.

Data streams represented by *Flows* can be manipulated by a plethora of built-in operators. Many of them, such as *map*, *filter* and *take* are similar to the ones used to manipulate *Lists* and *Sequences*. Multiple *Flows* can be combined using the *combine* and *zip* operators. Using the *zip* operator on two *Flows* produces a new *Flow* of pairs, constructed from the values of the two *Flows* in a zip-like fashion.

NaviPilot uses *Flows* to provide data to UIs and to pass data between components whenever it is possible. Several libraries used in this project provide out-of-the-box integration with coroutines and *Flows*. In many cases where *Flows* are not available, extension functions are created that convert the callback-styled code to a *Flow*, using the *callbackFlow* builder.

5.2 Documents

One of the main features of the EFB application is the ability to store and read documents. From the user's perspective, it is important to be able to bring their own set of documents to the EFB application, as a generic set of pre-loaded documents would not be sufficient for all users. It is reasonable to assume, that each user has their own unique set of documents, organized in a way according to their preferences.

The initial implementation plan devised was to create an internal document index of files uploaded by the user, which would be stored in a database. The documents could then be labeled by their categories with tags and organized in a folder structure as required, while the documents themselves could be saved at their original locations. However, it is not possible to access files stored anywhere on the device's file system, due to Android's security limitations and the Storage Access Framework (SAF). The documents would have to be copied to the internal application directory, which would result in unnecessary file duplication, wasting storage space.

Storage Access Framework

Working with files on the Android platform has traditionally been done using the standard Java File Application Programming Interface (API). With the Android 4.4 release, the Storage Access Framework has been introduced, aiming to simplify browsing and opening documents for users [29]. When an application wants to access a file using the SAF, a system dialog containing a file picker interface is invoked. The user then has to manually select the file they want to let the application access. The advantage of SAF is, that it supports files saved outside of the device filesystem, such as cloud environments.

Applications targeting Android 11, have full access only to its internal application directories and public media directories. To access other locations, the user must grant access through the SAF first. This action needs to be repeated for every file, although access to a directory tree can be requested too. The restriction of file access has naturally caused issues for many applications, namely file managers, that depended on the file access as a core functionality, although the issue has since then been addressed by Google with possible exemptions [77].

To ensure compatibility with the newest versions of Android, NaviPilot uses the SAF to access documents. When opening the Documents screen for the first time, the user is prompted to choose a root directory containing all of his documents. The premise of this approach is, that the user already stores their documents somewhere in a familiar structure of properly named folders. By the virtue of SAF, even cloud storage locations are supported, their offline support is however limited. After a root document directory is picked, NaviPilot displays a file browsing experience shown in Figure 5.2, where the user can navigate between sub-folders and open documents.

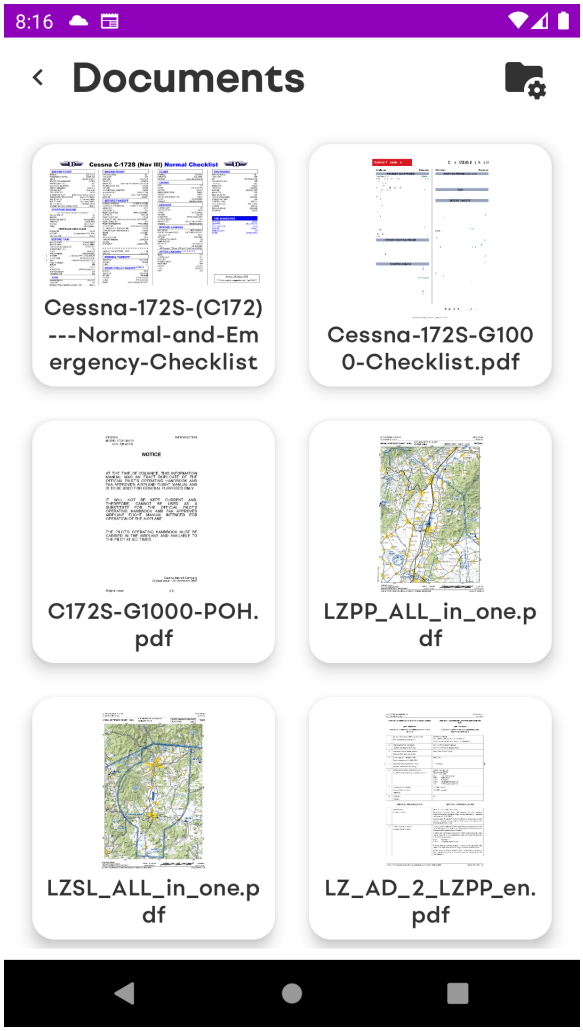


Figure 5.2: Documents interface.

Rendering thumbnails

NaviPilot uses the Coil library [7] to display file thumbnail images in the Documents User Interface. Using an image-loading library like Coil provides many advantages over direct rendering with Compose. The built-in disk cache provides faster loading times and the support for many image formats provides wider compatibility for media content. However, the Coil library does not yet support rendering previews of Portable Document Format (PDF) formatted files. Since PDF files are expected to be the main file format of the user's documents, this capability is essential to be added.

The Coil library provides the ability to extend the range of the supported file formats through custom *Decoders*. Classes implementing the *Decoder* interface read a *BufferedSource* as input and return a *Drawable* object that is trivial to draw on screen. *PdfCoilDecoder* is a *Decoder* that uses the PdfBox-Android library to render the first page of a PDF file to a bitmap, which is then converted to a *Drawable*. Coil provides the requested dimensions of the image, which are used to define the size of the resulting bitmap. After registering the *PdfCoilDecoder* as a *Decoder* for PDF files, Coil automatically uses the *PdfCoilDecoder* decoder to display thumbnails of PDF files when requested.

5.3 Airport catalog

In order to display information about airports, it is first needed to find a reliable source of information. Ideally, the source would also be provided free of charge. After carrying out research, the following data sources have been identified as viable:

- EAD Basic — The European AIS Database [12].
- Laminar Data Hub [72].
- OpenAIP — Worldwide aviation database [68].

The first choice, a service provided by Eurocontrol, provides free access to the AIP libraries of European countries. The unpaid tier is, however, limited to only human-readable PDF documents, which would be difficult to process into machine-readable data. The second option is a cloud platform created by Snowflake company, which provides a Representational State Transfer (REST)-based API for worldwide aviation data. This service would be sufficient, had it not included only a limited free access trial period.

OpenAIP is a web-based crowd-sourced aeronautical information platform that allows registered users to add, edit and download aeronautical data in many common formats used in General Aviation. Its goal is to deliver free, current, and precise navigational data to anyone [68]. It includes information about the world's airports, airspaces, navigational aids, and thermal hotspots. The platform's complete database is available to be downloaded in machine-readable XML files, making it suitable for further data processing tasks.

The OpenAIP XML files have a fairly complex structure of information, as the included entities possess various attributes, such as radio frequencies of airports or the geometry of airspaces. In order to work with the data in Java/Kotlin code, deserialization into model classes needs to be performed first. Since the model classes for OpenAIP entities are not readily provided, they need to be created first.

The concrete structure of an XML file can be approximated by observing the data itself. Fortunately, in this case, it is not necessary to do so, as OpenAIP publishes an XML Schema Definition (XSD) file containing the description of the exact schema of the OpenAIP XML files. Using the XSD file, it is possible to write the correct model classes for the OpenAIP data.

Writing the model classes by hand would be a time-consuming and error-prone task. Instead, an automated approach has been chosen, making use of the available XSD file. The model classes are automatically generated from schema before compilation using the *schema-gen* plugin for *Gradle* [11]. The model classes, together with the code responsible for downloading and parsing the OpenAIP catalog, are packaged into the *openaipparser* module, made available to be used by the rest of the project.

General information

Opening the Airport Catalog screen takes the user to an airport picker interface. The user can search through the airport database by the name or the ICAO code using a search bar located at the top of the screen. Tapping on a search result takes the user to a screen composed of several tabs with information related to the airport, as presented in Figure 5.3.

The first tab of the screen contains general information about the airport. The tab displays a map centered on the airport's location, showing the immediate vicinity of the

airport. Below the map are the airport's runways, their type, and length. Radio frequencies belonging to the airport are displayed at the bottom of the screen.

NOTAMs

The user has the ability to browse NOTAMs of the selected airport in the NOTAM tab of the Airport Catalog. The NOTAMs are displayed in a list of cards, each containing the NOTAM content and its issue and expiration dates. The NOTAMs are retrieved through an online API provided by the Federal Aviation Administration [19]. The raw data comes in the JSON format, which is deserialized afterward using the Kotlin serialization library into data classes, reverse-engineered from the JSON data. The NOTAM retrieval functionality is wrapped in the *faanotamapi* module of the project.

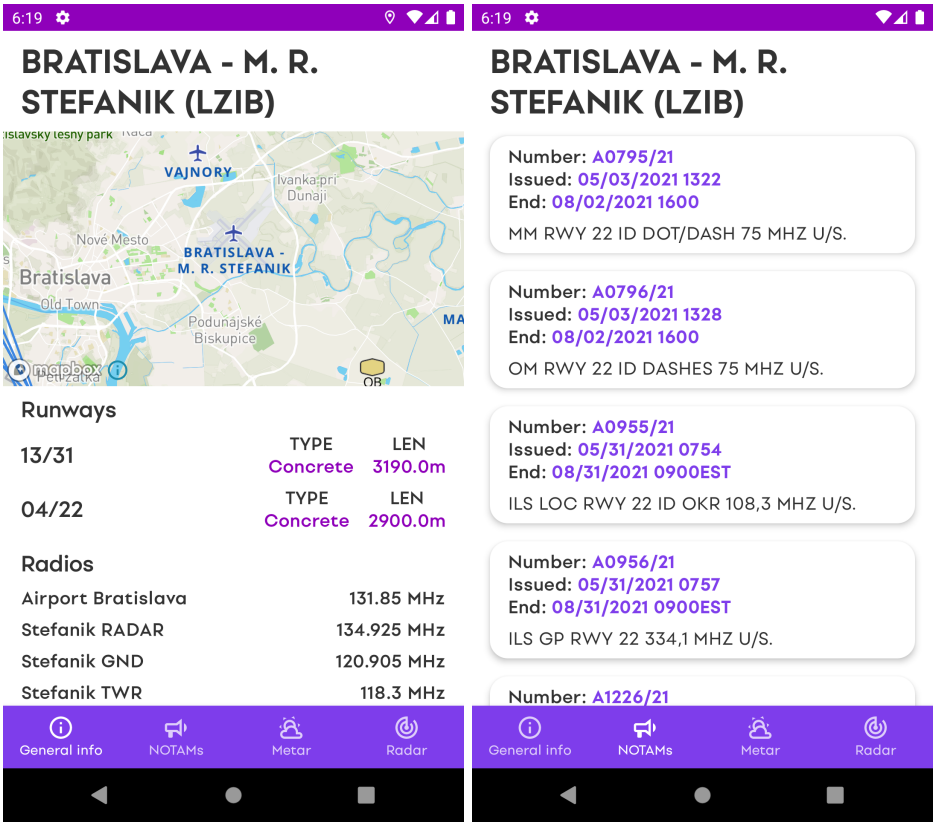


Figure 5.3: General information and NOTAMs tabs of the Airport Catalog.

METAR

In a similar fashion to NOTAMs, the user can also read the latest METAR information published for an airport. Downloading of METAR information is handled by the *aviationweathercenterapi* project module. The data is provided by NOAA's Aviation Weather Center [64]. METAR information downloaded from their service is parsed into data classes automatically generated from provided XSD schema files using *schema-gen*, in a process similar to the one used with OpenAIP data described earlier.

In order to increase conciseness, raw METAR information is encoded in a highly abbreviated string of characters. To make the weather information easier to digest for the pilot,

NaviPilot presents the METAR information in a more human-readable format. Pieces of information, such as the current temperature, wind, and visibility, are extracted from the METAR text using the open-source MetarParser library [52]. The individual values are then displayed in the interface below the original METAR code, as seen in Figure 5.4.

Weather radar

The last tab of the Airport Catalog allows the user to preview the weather conditions of the selected airport on a map. The tab displays the weather radar layer, described later in the Maps Section 5.8, overlaid on top of the base map. The layer displays real-time precipitation in the area around the airport. The intensity of the precipitation is visualized through a color gradient ranging from green to red. The user can also view a forecast of the radar by changing the value of the slider at the top of the screen.

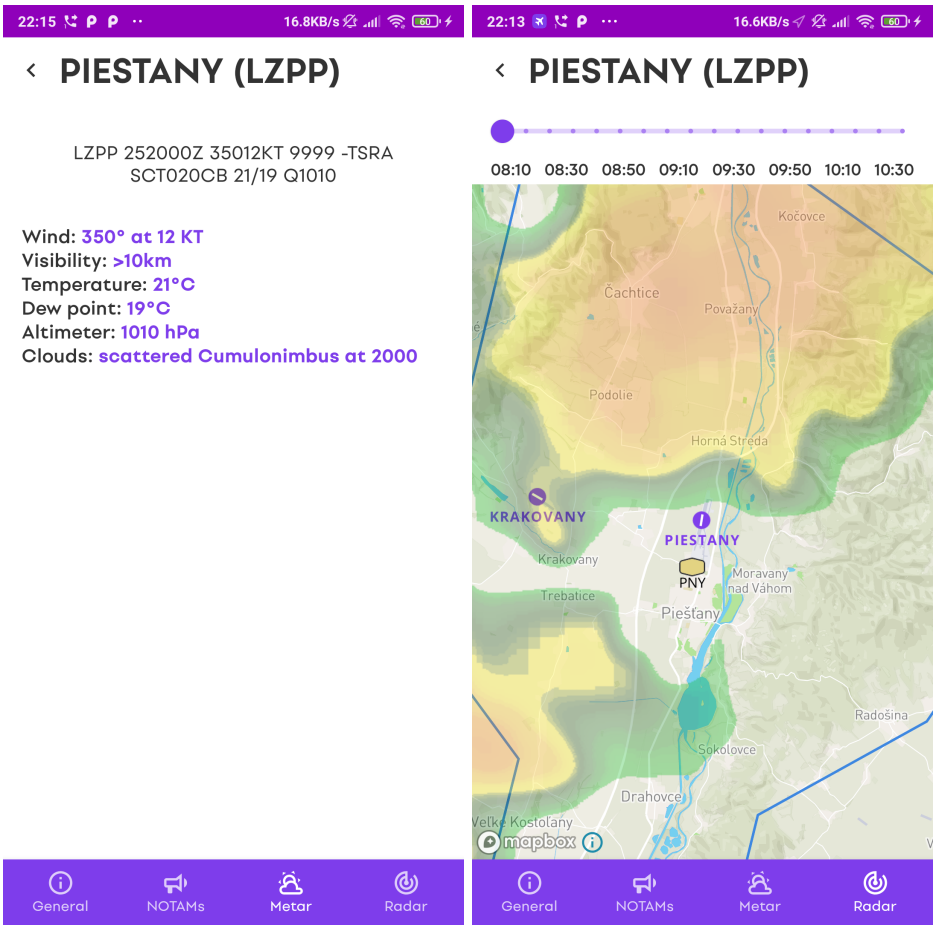


Figure 5.4: Airport Catalog tabs METAR and Radar.

5.4 Aircraft profiles

The Aircraft profiles feature allows the pilot to define the properties of multiple aircraft models frequently flown by the pilot. Before each flight, the pilot can choose which profile to use during the flight, giving the application the ability to utilize the information about the aircraft being flown. NaviPilot includes an Aircraft profile editor displayed in Figure 5.5,

where the user can specify an aircraft's Maximum Takeoff Weight (MTOW), Maximum Landing Weight (MLW), empty weight, and the characteristics of other weight items, such as fuel tanks or baggage compartments.

In addition, the user has the ability to input the aircraft's Center of Gravity (CG) envelope. The envelope is defined as a list of points on weight vs moment chart, that represent an area wherein the total weight and moment of the aircraft are within the aircraft's specifications. The CG envelope of an aircraft is usually defined in the aircraft's Pilot's Operating Handbook (POH) or Aircraft Flight Manual (AFM) [14]. The CG envelope and other parameters of the Aircraft profile are used during the Weight and Balance calculations in the flight planning phase described later in Section 5.9.

NaviPilot comes with three pre-installed Aircraft profiles for the popular Cessna 172S, Cessna 152, and Zlín Z-142 planes. These profiles include accurate information about the aircraft, sourced from their respective Pilot's Operating Handbooks. The default profiles make it easier for users flying these airplanes to get started using the application.

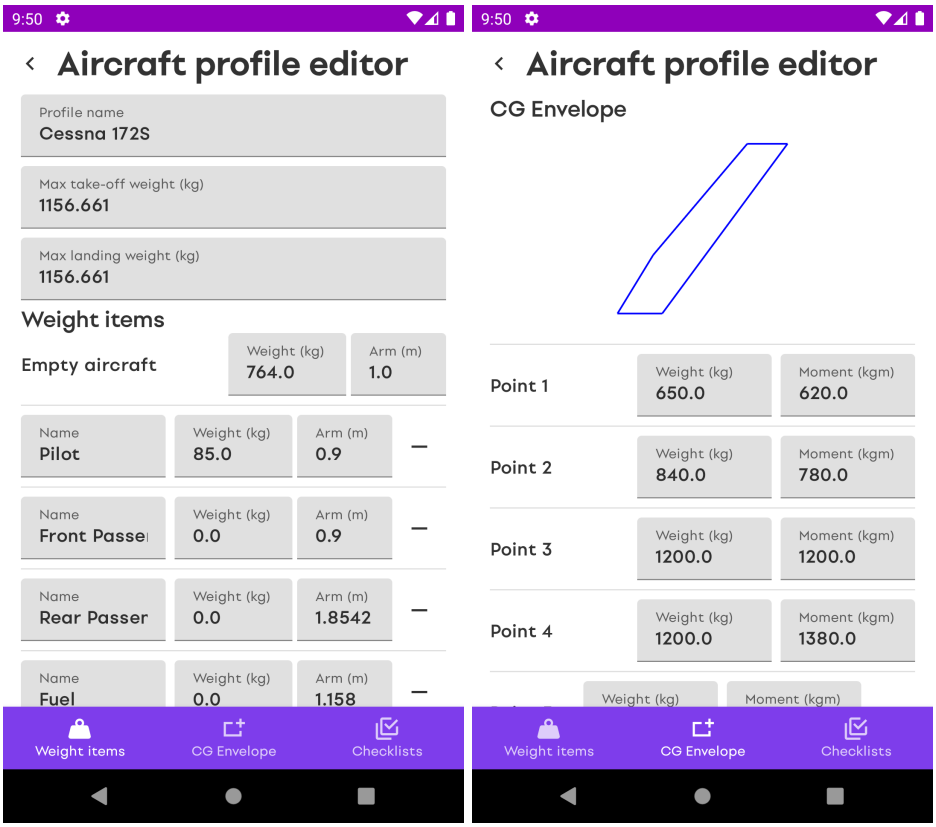


Figure 5.5: The Aircraft profile editor screen.

DataStore

Aircraft profiles and other application settings need to be persisted on the device between application launches. DataStore is a data storage solution that allows the developer to store key-value pairs or typed objects with protocol buffers. Jetpack DataStore uses Kotlin coroutines and *Flows* to store data asynchronously, consistently, and transactionally [35]. It has been designed as a successor to the Android's standard *SharedPreferences* framework, which has its limitations in terms of consistency in certain conditions [63]. Due to the

library being designed with the intent of storing small amounts of data, it is the perfect candidate for the use-cases described above.

5.5 Checklists

Each Aircraft profile has the ability to store checklists applicable to the particular aircraft. The default Aircraft profiles already provide checklists gathered from the aircraft's POHs. These checklists provide a great starting point for the user to edit and create additional checklists, using the built-in Checklist editor shown in Figure 5.6.

A Checklist is made up of several checklist items, where each item consists of a description and a designated action keyword. The user can choose from a predefined set of action keywords, which are supported by the Automated checklist feature described in the section below, or specify a custom keyword instead. Each checklist is given a name and optionally a category label, such as *Pre take-off* or *Landing*, designating the flight phase to which the respective checklist applies. The category label is then used to automatically bring up relevant checklists in the Dashboard during the flight, using flight phase-detection described later in Section 5.11.

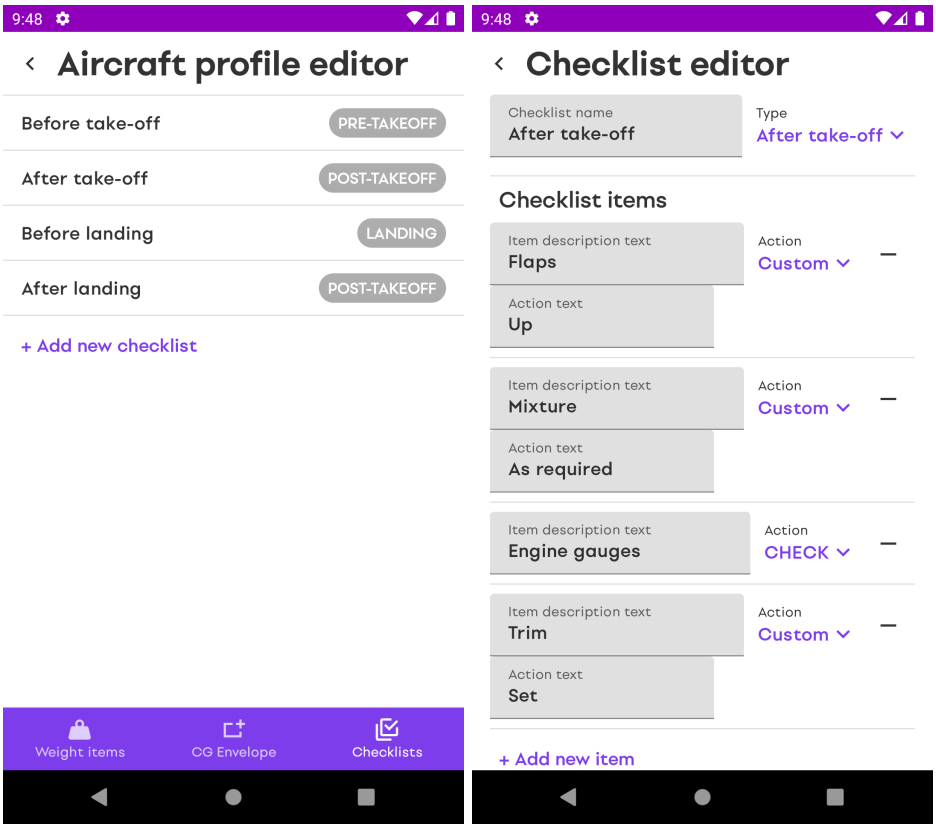


Figure 5.6: Checklists and the Checklist editor.

Automated checklists

The Checklists segment of the application features specification in Section 4.2 proposed a feature that automatically recognizes the pilot's responses during a checklist procedure with speech recognition. More formally, this problem can be classified as a Keyword Spot-

ting (KWS) task. Historically, KWS problems were solved using various approaches, however, most of the latest models are based on Machine Learning (ML) and Neural Networks.

Machine Learning techniques, such as convolutional (CNN) and recurrent (RNN) neural networks, have shown strong performance for various speech processing tasks, including Keyword Spotting. As is currently the case in other Natural Language Processing (NLP) problems, some of the latest models utilize the widely popular transformer architecture and are able to achieve state-of-the-art performance in the Google Speech Commands dataset benchmark [4].

The raw speech input is a stream of audio data from an audio source, such as a file or a microphone. A raw audio stream consists of the amplitude values of an underlying analog signal, sampled at a specified frequency and sample resolution. Audio input may contain multiple tracks, typically from the left and right audio channels, although, for the purposes of KWS, a single track is sufficient.

For deep learning based speech-recognition, audio in its raw format is not optimal as an input feature. Human-engineered speech features from traditional speech processing techniques, such as Log-mel Filter Bank Energies (LFBE) and Mel-frequency Cepstral Coefficients (MFCC), translate time-domain speech signal into a set of frequency-domain spectral coefficients. This step reduces the dimension of the input data while preserving the defining features of the speech signal, greatly improving the performance of the model [79].

Neural networks usually require an input of a predefined size. To reduce an infinitely-sized stream to a fixed size, a sliding window of a specified length is applied. Each window is run through a classifier which assigns a label to the input based on its contents. However, since it is not known in advance where the keyword starts or ends, it is very likely that a keyword occurrence would be split into two windows, avoiding detection. One of the ways to resolve the problem is to shorten the stride of the windows, in order to make them overlap.

Overlapping windows are inherently inefficient, since data in the overlapping regions must be processed by the Neural Network more than once. Efficient use of resources is a key factor when running in the context of a mobile device. A more efficient solution can be achieved by modifying the model to accept a small input frame, together with an internal state from the previous prediction. This allows the network to process the audio input in a true streaming fashion, which drastically reduces the computational cost [8].

In order to implement the Automated checklist feature, a model has been trained using a library created by Google researchers, which enables automatic conversion of non-streaming models to streaming ones with minimum effort [69]. The TC-ResNet architecture is chosen as the base model for this task, due to its high-ranking benchmark results [6].

The training data for the model is composed of voice recording samples collected in the Google Speech dataset [76], filtered to the following keywords: *On*, *Off*, *Left*, *Right*, *Up*, *Down*, *Forward*, *Backward*, *Go*, *Stop*, *Zero* and *One*. In addition, the dataset has been extended by self-recorded samples of the *Check* keyword.

The model has been trained for 120,000 learning steps, with a gradually decreasing learning rate. The training process finished with a reported test accuracy set at 96.79 %.

The KWS model in question is created using the TensorFlow framework. TensorFlow is a popular end-to-end open-source platform for Machine Learning [73]. TensorFlow models can be ported to mobile devices by converting them to the TensorFlow Lite (TFLite) format. TFLite models can subsequently run inference on Android phones using the ML Kit SDK. In addition, the Android Studio IDE supports TensorFlow Lite with the ability to generate model bindings, making it easy to work with TFLite models in code.

In the NaviPilot project, access to the final TFLite model is provided by the *SpeechRecognitionEngine* class. The class slices the live audio stream from the microphone into windows of a set size. Each window is fed into the model, which in turn provides an estimated score for each keyword on its output.

The raw output scores of the model are converted to probabilities using the softmax function. A keyword detection event is raised, when the average value of the last 10 probabilities of a keyword exceeds a threshold of 80 %.

In practice, the trained model is able to recognize the keywords from the Google Speech dataset fairly accurately in perfect conditions. However, noisy environments sometimes cause false triggering of the keywords, which may be reduced by further raising the activation threshold.

In contrast, the self-recorded *Check* keyword shows a poor detection performance. This may be caused by the small number of samples and their low variance, which could be improved by gathering more recordings of the keyword from different speakers.

When comparing the streaming version of the model to the non-streaming version, the latter seems to perform better in terms of accuracy and latency. When uttering a keyword, the streaming version of the model seems to raise the output score for the keyword only after a noticeable delay. Moreover, the high score for the keyword is susceptible to linger in the same value long after the keyword utterance ended.

Due to the reasons stated above, NaviPilot uses the non-streaming version of the model in the end. The model can be tested in a debug screen located in the Settings. The debug screen provides real-time output of the keyword scores from the microphone audio stream, together with a live feed of keyword detection events.

5.6 Logbook

In the Logbook screen, the user can view their flights organized in a list. All flights are automatically saved in the Logbook, whether they are drafts, flights in progress, or finished flights. The status of the flight is indicated by a badge, located next to the general flight information about the flight, such as the origin, destination, length, and the number of waypoints.

Tapping on a flight takes the user to the Flight planner screen if it is a draft, the Dashboard if it is a flight in progress, or to the Flight detail screen, if it is a finished flight. The Flight detail screen contains a map window where the user can review the flight path of the flight. The flight path segments are colored based on the altitude of the segment. The flight's flight plan can be quickly reused as a starting point for a new flight with the press of a button, located at the top of the screen.

The user has the ability to export the flight path to a KML file using the *osmbonuspack* library [62]. Alternatively, export to GPX format is also available, thanks to the *JPX* Java library [78]. The Logbook and Flight detail screen can be seen in Figure 5.7.

Room

The flight data is persisted in an SQLite database located on the device. Working with the SQLite database is carried out using the Jetpack Room library. The Room persistence library provides an abstraction layer over SQLite to allow fluent database access through compile-time verification of SQL queries and annotated functions [41]. In Room, objects

saved into the database are modeled as entities, that can additionally define relational dependencies between themselves.

Interacting with the database in Room is achieved by defining Data Access Objects (DAOs) interfaces. A DAO interface contains methods annotated by directives, such as *@Query*, *@Insert* or *@Delete* annotations. At compile time, Room automatically generates implementations of the DAO based on the methods defined. Room fully supports Kotlin coroutines, enabling asynchronous access to the database. Additionally, by using a compatible observable return type in a query method, such as Kotlin's *Flow*, the callers can subscribe and automatically receive changes to the data in real-time.

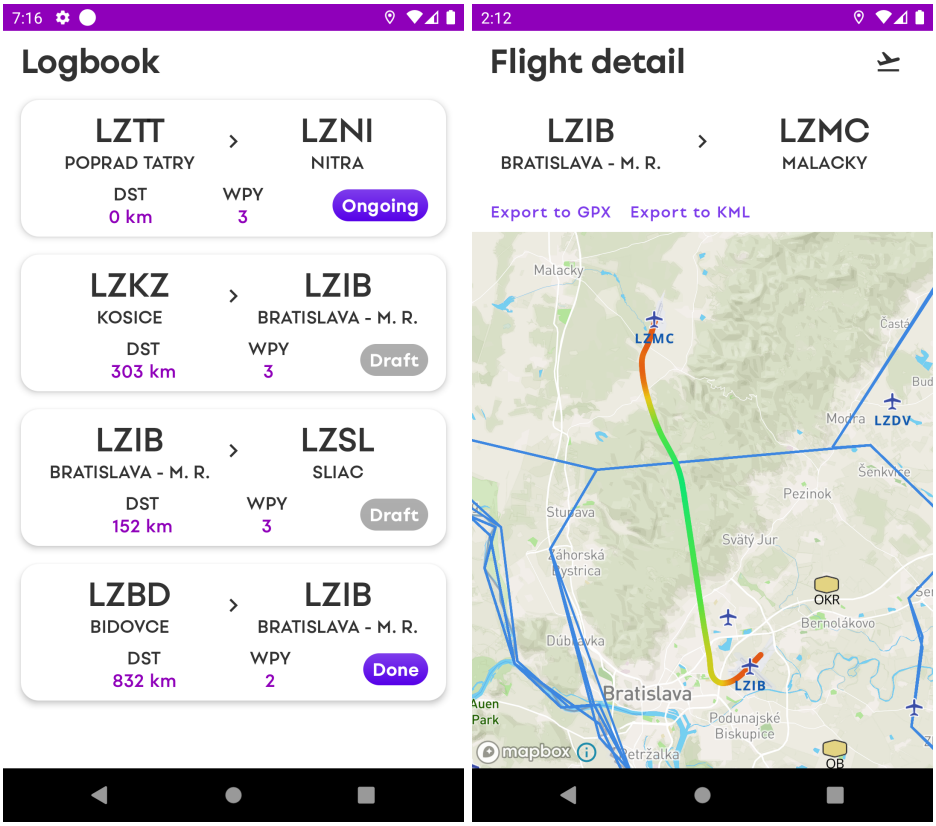


Figure 5.7: Logbook and Flight detail screens.

Paging

Loading all flights stored in the database in situations when only a few of them are displayed on the screen at once is not effective. The Paging library from the Jetpack suite helps the developer load and display small chunks of data at a time [40]. It integrates cleanly with the Room library, although abstractions over other data sources can be written by the developer as well. With *LazyColumn* of Compose and the Paging library, the Logbook screen is able to show a list of flights using Room efficiently, loading them only when needed as the user scrolls down.

5.7 Data connectivity

Before proceeding to work on the in-flight features of the application, it is first needed to establish a way of simulating a device in a testing flight environment. The application's default source of position during the flight is the GNSS receiver. For location testing purposes, the Android OS provides a way of mocking the GPS position on real devices through the built-in developer's settings.

In addition, the Device Emulator features a special location control panel, where the developer can set the emulator's location through a provided map interface. This option, however, does not allow to control the altitude of the device, although another option allows the developer to set the device to follow a predefined path imported from a GPX or KML file.

A more flexible proposition comes from utilizing flight simulators for testing. Many flight simulators today include means to access the data from the simulation in high detail. This widespread support is caused mostly by the popularity of after-market modifications that are highly prevalent amongst this genre. Coupled with the fact, that many flight simulators model the world in full scale, it is easy to test most use-cases of the application, simply by launching a custom flight scenario in the simulator.

Microsoft Flight Simulator, released in 2020, is the latest release of the popular *Microsoft Flight Simulator* series. *Microsoft Flight Simulator* allows players to fly planes in a realistic real-world environment, recreated using a modern game engine. The engine uses real satellite imagery and photogrammetry data processed by Microsoft Azure's Artificial Intelligence (AI) [45]. The virtual world features accurately modeled landmarks and over 37,000 airports.

Developers and mod-creators can interface with the *Microsoft Flight Simulator* using the *SimConnect* SDK add-on. *SimConnect* allows to record and monitor flight scenarios, retrieve and subscribe to simulation variables, change the weather conditions, and much more [54]. After enabling the add-on, *SimConnect* acts as a server listening on a pre-defined port, where clients can connect using the *SimConnect* protocol over TCP/IP.

The connection between NaviPilot and the simulator is managed by the *SimConnectService* object. It leverages the open-source *jSimConnect* Java library to initiate a connection and listen to events from the simulator through *SimConnect* [46]. The *SimConnectService* object periodically attempts to connect to the simulator with the IP address and port number provided by the user in the Simulator connection setup screen.

When a connection is successfully established, the service subscribes to information about the plane's current coordinates and attitude. This information is then available to other project components through an observable *Flow*. When an error or a disconnect occurs, all subscribers are properly canceled using the standard Kotlin coroutine mechanism.

All application components which use location do so through the *LocationEngine* interface of the *Mapbox* library described later in Section 5.8. The default *LocationEngine* implementations provided by the *Mapbox* library use the built-in device location sensors. *SimConnectLocationEngine* is a custom implementation of the *LocationEngine* interface, which uses the *SimConnectService* to receive data from the simulation. Changing the source of location in the application between the built-in GNSS receiver and the simulation is thus made as easy as switching the implementations of the *LocationEngine* at runtime. A class diagram depicting the *LocationEngine* interface can be seen in Figure 5.8.

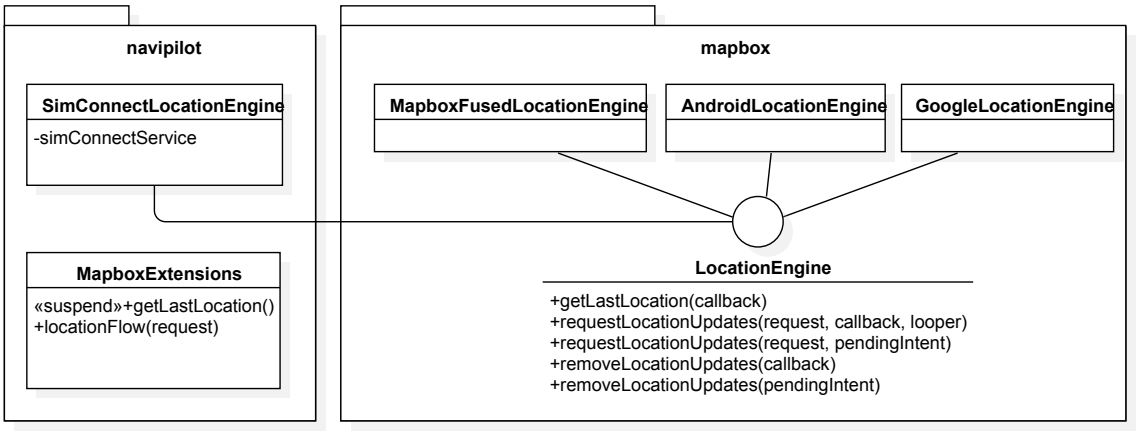


Figure 5.8: Class diagram of the *LocationEngine* interface and its implementations.

More data sources could be added in the future by creating other implementations of the *LocationEngine*. An external ADS-B & GNSS Receiver like *Sentry*, illustrated in subsection 3.3.4, would make up for a great candidate for a new data source. Integration with this device would provide a more accurate location and attitude data in a real-world scenario, compared to the data coming from the built-in sensors of the mobile device.

5.8 Maps

Maps belong to the core elements of the NaviPilot application. They are used to show the current position of the aircraft in the Dashboard and help visualize the flight path in the Flight Planner. In order to display maps in the application, the Maps SDK from Mapbox is used [58]. To use the Maps SDK in this project's codebase, a custom Map Composable has been made, as the library itself does not yet support Compose.

The Maps SDK comes with several map styles installed by default, such as Street, Outdoors and Satellite, each being suitable for a different purpose. These styles contain data derived from the OpenStreetMap (OSM). OSM is widely considered as one of the best map sources of the world. It is built by a community of mappers that contribute and maintain map data, which is then available to the general public under an open license, free of charge.

Creating a custom map style

Since the default Mapbox map styles are not intended for aviation usage, a custom style is created. This style includes Points of Interest (POIs) and areas important for the pilot, such as airports, navigation aids, and airspace bounds. The custom map style also emphasizes relevant natural features, such as rivers and mountains, that can be helpful for the pilot when navigating the environment. Simultaneously, map information not related to the piloting task is suppressed or removed, reducing unnecessary clutter. A showcase of the custom style from the application can be seen in Figure 5.9.

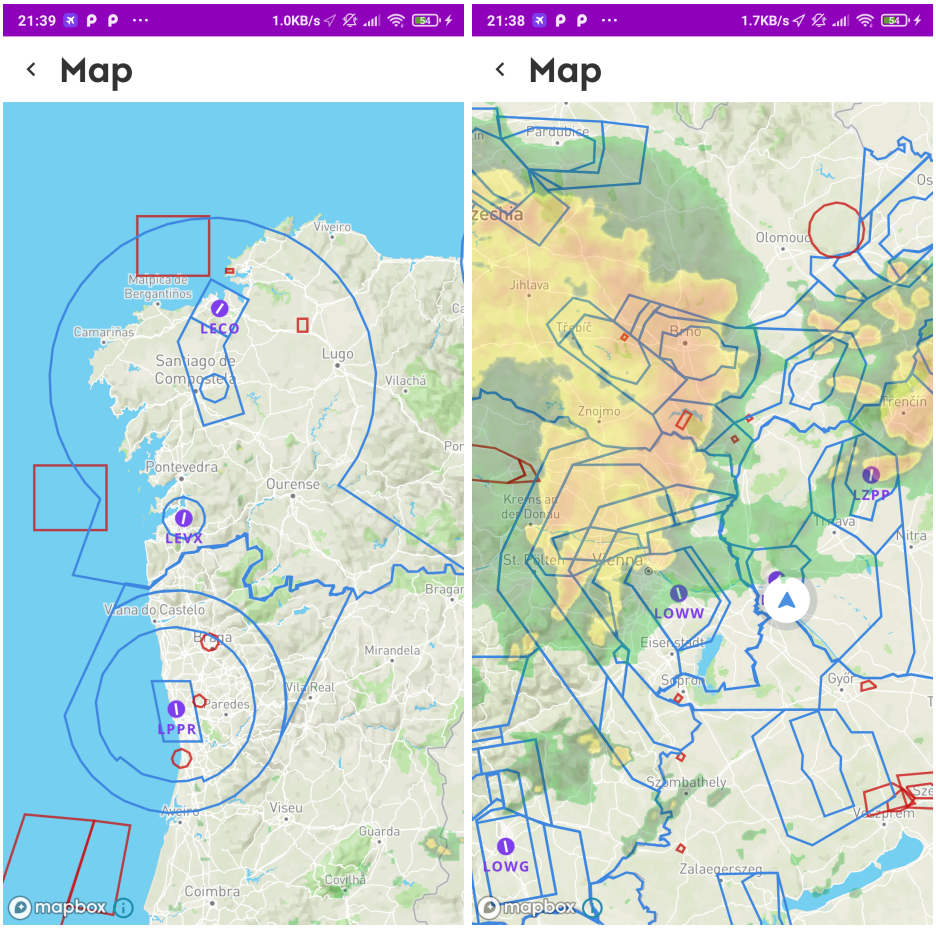


Figure 5.9: Custom NaviPilot map style and the Weather radar layer.

Mapbox Studio is a powerful online tool used for creating custom map styles. Styles created by this tool are fully compatible with the Maps SDK client library, making it easy to use the newly created styles in the application. It is important to note, that a style does not contain any spatial data by itself. A style receives data from one or more tilesets. A tileset is a collection of raster or vector data broken up into a uniform grid of square tiles at up to 22 preset zoom levels [60].

Mapbox, again, provides several default tilesets to choose from. These tilesets contain data from the OSM, raster satellite images, terrain data, and more. None of these default tilesets contain the aviation data necessary, which is why creating a new tileset is needed. The OpenAIP dataset described in Section 5.3 is used as the source to create the tileset since it already contains the necessary airport, navigation aid, and airspace data.

The processing pipeline consists of three steps done in sequence. First, all available OpenAIP files are downloaded and saved. Since the data format of the OpenAIP files is not recognized by Mapbox Studio, it is necessary to convert the files to a format that is supported. GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON), that is compatible with Mapbox Studio [5]. The OpenAIP data is converted to the GeoJSON format in the second step, using the OpenAIP2GeoJSON open-source library [50] that has been modified with small changes intended for this project.

The last step involves uploading the created GeoJSON file to Mapbox Studio as a tileset. While Mapbox provides several ways of achieving this goal, in this case, the *tilesets* [61]

command-line utility is used. After the upload is completed and processed, the tileset is made available in the Mapbox Studio environment, ready to be used in a map style. The processing pipeline is written in the Python language, in the form of a Jupyter notebook, located in the *openaipmaptiles* module.

With the OpenAIP tileset ready, the final map style can be created. As a base layer, the Mapbox Streets style is used. It is edited according to the design specifications described above, in order to be more suitable for aviation navigation. Airspaces colored by their types are drawn as polygons on top of the base layer. Next, the navigational aids, as well as the airports, are added to the map, in the form of an icon with a name label below. The airport icon symbolizes a runway, that is rotated to match the orientation of one of the real runways of the airport.

In order to make the map more readable and performant, several map styling rules are created. At small zoom levels, the airspace polygons are hidden, and only the international airports are displayed. As the zoom level increases, the airspace borders become visible, and the airport labels transition from icon-only, through displaying the ICAO code, to showing the full name of the airports.

Weather radar layer

As per the design specification of the map feature located in Section 4.2, the map should provide a visual indication of the weather conditions in the surrounding area. RainViewer is a weather forecast application for Android [70]. Their public Weather Maps API provides weather radar data of the past 2 hours and forecasts of up to 30 minutes. The weather data is available in the form of a map tileset, which can be displayed as a layer on a Mapbox map.

A wrapper for the Weather Maps API has been created in the *rainviewerapi* module. The raw data in JSON format is downloaded from the API periodically every minute. Using the Kotlin serialization library, the data is deserialized into prepared data classes. The data contains a list of map tileset URLs together with their timestamps. The latest tileset URL is parsed and published to other application components through an observable *Flow*.

Any map component in the application can display the weather radar layer, if enabled. To ensure that the radar data is always the latest that is available, the map subscribes to the *Flow* of the radar URLs of the *rainviewerapi* module. Whenever a new tileset URL with the latest data is received, it is automatically loaded in place of the old tileset, and subsequently refreshed on the display. The weather radar overlay is made transparent in order to avoid covering the underlying map. A preview of the weather layer displayed on a map can be seen in Figure 5.9.

5.9 Flight planner

Creating a new flight or editing an existing draft takes the user to the Flight planner screen. This is the place where the user can choose the origin, the waypoints, and the destination of their flight. During the editing process, the current flight path is visualized on a map. The waypoints and the destination airport can be selected either through the map or through a dialog. A screenshot of the Flight planner interface is shown in Figure 5.10.

A custom place picker dialog has been created, which allows the user to search for POIs by typing a query into a search bar. The search results are then displayed in a list sorted by the distance to the user. In addition to POIs, the user can choose a coordinate waypoint

through a point picker interface. The interface shows a map where the user can select the point by tapping on the desired location.

Another way to select a waypoint is by searching for a place, such as a city or a landmark. The place query is handled by the Mapbox Search SDK [59]. The SDK returns a list of place results for a given query, which is presented to the user in the picker interface. The place waypoint is then confirmed by tapping on one of the displayed results.

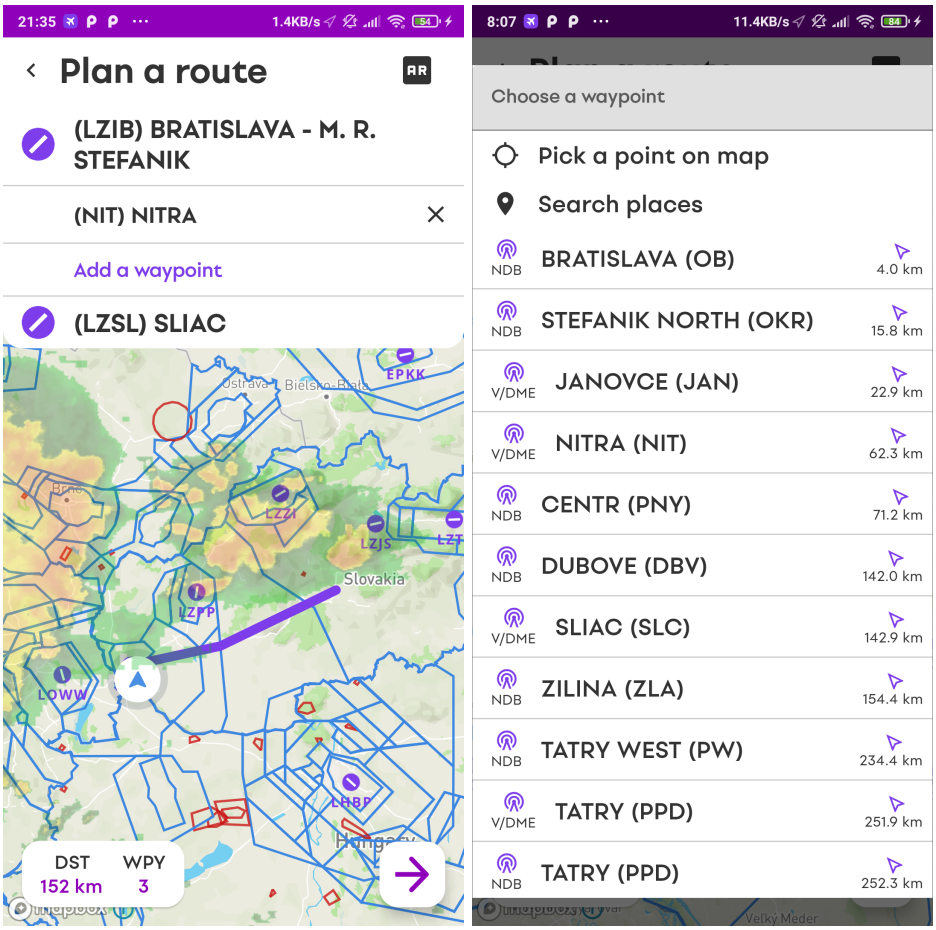


Figure 5.10: Flight planner interface and the waypoint picker.

Weight and Balance

After creating the flight plan, the user proceeds to the Weight and Balance screen. The purpose of this screen is to provide an overview of the aircraft's weight and balance for the pilot. After selecting the Aircraft profile that matches the aircraft that the user will fly, the screen presents the maximum take-off and landing weights of the aircraft, together with its CG envelope. The individual weight items, as specified in the Aircraft profile, are displayed next to input fields, where the user inputs in the actual weight values of each weight item.

As the user fills out the weights, the total weight value and the current CG point in the CG envelope automatically update to reflect the actual weight and balance of the aircraft. This information helps the pilot decide whether the aircraft is suitable for flight with the selected weight configuration. The Weight and Balance screen can potentially prevent errors

and save a considerable amount of the pilot's time, as the weight and balance calculations are usually carried out by the pilot manually using pen and paper.

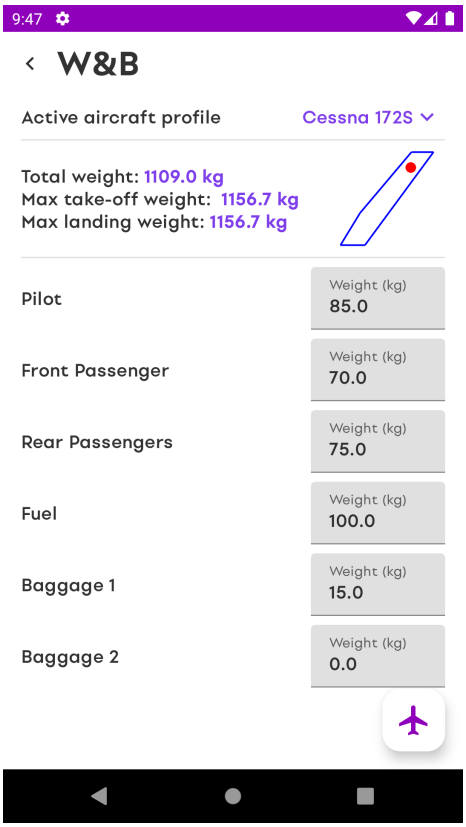


Figure 5.11: Weight and Balance screen.

5.10 AR Preview

Augmented Reality (AR) is a technology that can provide unique experiences to its users. It works on the principle of combining virtual information with the real-world environment. With the broad availability of hand-held devices capable of delivering this type of experience, this technology is now becoming a core part of some of the most popular mobile applications, such as *Pokémon GO* or *Ingress* [66].

The continually increased availability of developer tools also plays a great role in lowering the entry barriers for AR development. ARCore is Google's platform for building augmented reality experiences for Android and iOS devices. It provides various APIs, such as motion tracking and light estimation, that enable the developer to interact with the real-world environment [33]. With its constant updates, it is arguably the best option to create AR-enabled applications on Android.

To test out this technology, NaviPilot incorporates a feature that allows the user to visualize the planned flight path using Augmented Reality. By pointing the phone's camera on an empty surface, the pilot can view a miniature relief of the land around the path, with the flight path drawn as it would cross through the real world. Sample screenshots of this feature can be viewed in Figure 5.12. The AR Preview can give the pilot a better spatial

awareness of the surrounding terrain later in the flight, especially with regards to the hills and mountains located in the vicinity of the flight route.

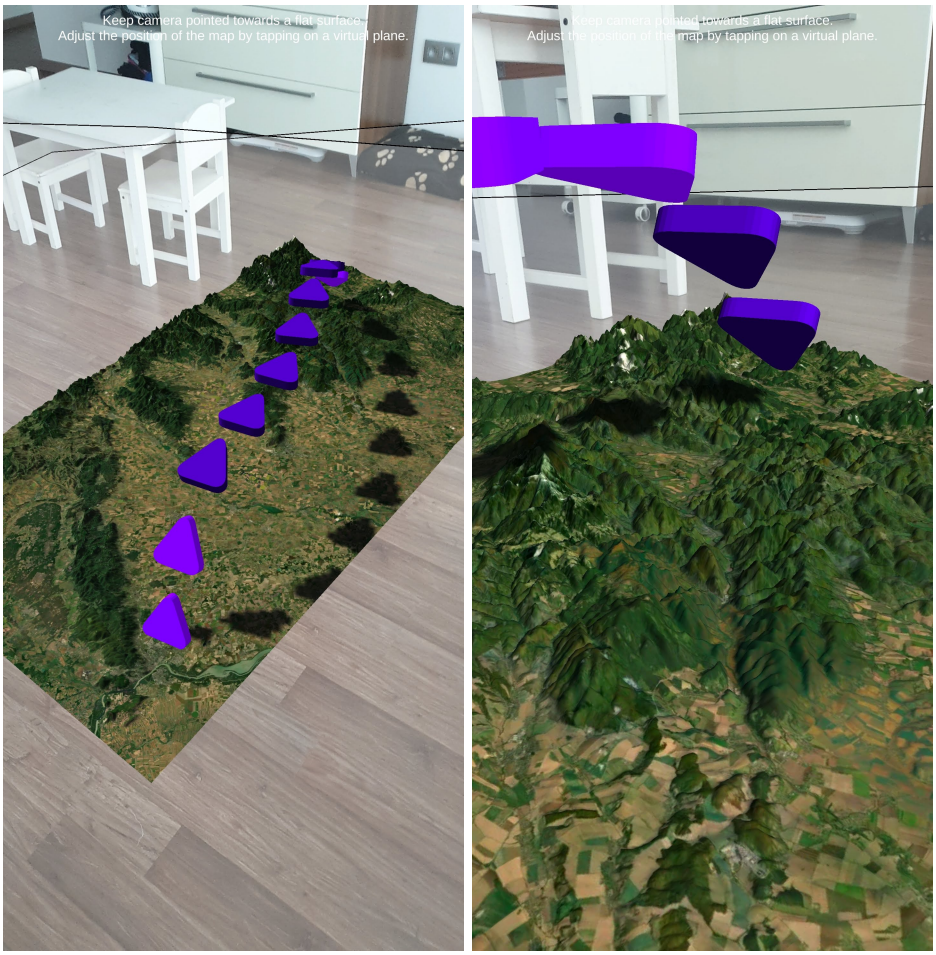


Figure 5.12: Augmented Reality Preview of a planned flight.

To create this AR experience, the free game engine Unity is used. Unity is the leading platform for creating interactive, real-time content. Unity provides tools that help build 2D, 3D, and Virtual Reality (VR) games and apps for desktop, web, mobile, and console platforms [75]. It allows the developer to work with several Augmented Reality platforms, including ARCore, in a multi-platform way, through the AR Foundation package.

Working with maps in Unity is made possible using the Mapbox SDK for Unity package. The package makes it easy to add drag-and-drop maps with POIs, 3D buildings & terrain, place-based AR, and more [57]. The SDK provides the *AbstractMap* object, which can be used to display a mesh map of an area. The map can be configured to use various map styles and includes the possibility to automatically apply terrain extrusion based on data from an elevation tileset.

The Unity project is exported to a *Gradle* module named *armodule*, which is finally integrated as a dependency of the main *app* module. The export process, usually done within the Unity editor, has been integrated into the project in the form of a *Gradle* task, enabling to run the export as a part of the build process of the project. The main application can launch the AR experience through the *ARFlightVisualizerActivity*. This Activity manages

the Unity Player running the experience, while also handling the communication that is required to pass the flight path data from the application to Unity.

5.11 Flight engine

The heart of the NaviPilot in-flight system lies in the Flight engine component. A Flight engine instance is bound to a specific flight, distinguished by its ID. It handles loading its data from the database or creating it if it does not already exist. The Flight engine uses an instance of a Location engine (described in Section 5.7) to receive data about the aircraft. The data is processed and stored in the Flight engine's *Flight* object. Other application components can subsequently receive the data through subscribing to changes in the *Flight* object, via Kotlin's *MutableStateFlow*. Functionalities of the *Flight engine* are decomposed into multiple sub-component engines, as shown in the diagram in Figure 5.13.

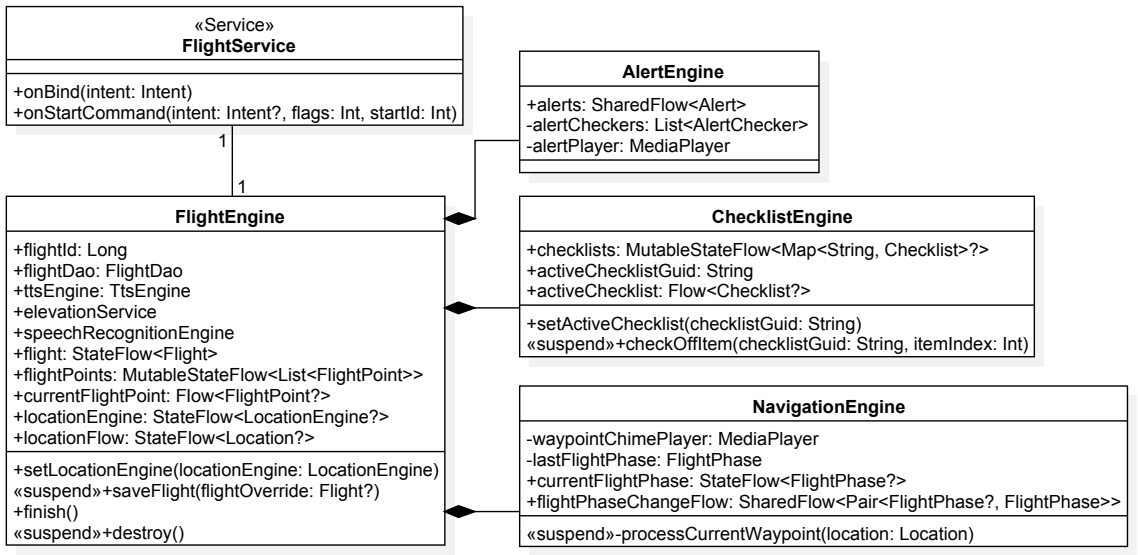


Figure 5.13: Class diagram of the Flight engine component and its subcomponents.

The Flight engine lives inside a *FlightService*. In the context of an Android application, a Service is an application component that can perform long-running operations in the background [42]. The *FlightService* is started whenever the user starts a flight in the application. *FlightService* ensures that NaviPilot keeps tracking the flight even when the application is closed or moved into the background. A notification created by the *FlightService* keeps the user informed about the fact that a flight is in progress. When the user opts to finish a flight, *FlightService* coordinates with the Flight engine to save the flight data in the database.

Alert engine

The Alert engine provides a common mechanism for the detection of alerts. Alerts are meant to notify the pilot of various events they might be interested in. The alert engine can detect several types of alerts, handled by so-called alert checkers. An alert checker observes the flight data and potentially other sources of information. Whenever an alert should be raised, it notifies the Alert engine. An alert checker is created by implementing

the *AlertChecker* interface, which contains a single method returning an observable *Flow* of *Alert* objects.

The airspace alert checker monitors the aircraft's location and the nearby airspaces. It keeps track of the airspaces in which the aircraft is located. When the aircraft enters a new airspace, an alert is raised. The alert includes the name of the airspace, its type, and its associated frequency, if it has one. The list of the airspaces and their geometry bounds are sourced from the OpenAIP dataset, as discussed in Section 5.3.

Navigation engine

The *Flight* object keeps a list of all waypoints and the index of the next waypoint in its properties. The navigation engine processes incoming locations and checks whether the aircraft has closed in to the next waypoint beyond a certain threshold. When a waypoint is passed, the Navigation engine increments the next waypoint index and triggers a sound chime with a TTS message announcing the heading towards the next waypoint.

Another responsibility of the Navigation engine is the detection of flight phases. This feature is useful mainly for other application components, such as the Automated checklists feature. In order to determine whether the plane is landed, flying, or approaching, the engine uses a rule-based system that uses the current position and elevation of the aircraft. The current flight phase and flight phase change events are exposed to other components through Kotlin *Flows*.

Deciding whether the aircraft is on the ground requires knowing the altitude above ground, otherwise known as elevation. However, the output from a GNSS receiver returns altitude values above mean sea level, ignoring the terrain. If the terrain height below a certain point is known, the elevation of that point can then be calculated by subtracting the terrain height from the altitude.

There are several publicly available datasets with global terrain elevation data. The datasets come in various resolutions, usually ranging from 1 km to 30 m. However, even the low-resolution datasets, such as GLOBE [65], have a size of over 300 MB when compressed. The large size makes them unsuitable for use in a mobile application, notwithstanding the lack of resolution required for accurate elevation calculations.

The Mapbox company provides a pair of map tilesets that contain elevation data compiled from a variety of sources [56]. The first tileset is a vector tileset, which contains lines and polygons describing the terrain's landcover and contours. Elevation mapped to 10 m height increments can be queried from an attribute of the contour layer.

The second tileset contains global elevation data stored in raster tiles. The height is encoded in the color values of the raster's pixels in 0.1 m increments. The resolution of the data is based on the current zoom level, with the data being provided up to zoom level 15. In order to read the elevation of a certain point, a pixel closest to the point is sampled. The terrain height at that point in meters is then calculated according to the formula below, where the R, G, and B values are the red, green, and blue components of the sampled color.

$$height = -10000 + ((R * 256 * 256 + G * 256 + B) * 0.1)$$

The Mapbox SDK for Android does not provide a direct method to access raster layer data from a map instance displayed in the UI. The *MapSnapshotter* object is used for generating static maps of a bounded region. These maps are as lightweight as images and are suitable for cases when there is a need to display multiple maps in the UI, for example

in lists. Since the output of a *MapSnapshotter* is a bitmap, it is trivial to extract pixel values from it.

The *ElevationService* object handles all elevation queries from other application components through an asynchronous queue implemented by a coroutine *Actor*. The service reuses a single *MapSnapshotter* instance to take snapshots of the raster terrain tileset. When taking a snapshot, the camera of the *MapSnapshotter* is centered on the queried location and fully zoomed in. The requested snapshot has dimensions of 1x1 (single pixel), in order to avoid using unnecessary memory space.

The advantage of the *MapSnapshotter* approach is, that it integrates nicely with the offline maps mechanism of the Mapbox library. Since NaviPilot pre-loads low zoom level terrain tiles of the whole world during the initial setup, the *ElevationService* can respond to all queries even when offline. The nature of the map tiling mechanism ensures, that at low zoom levels, the query returns an averaged height value that is still fairly accurate. Higher detail terrain tiles are pre-loaded for countries installed in Offline packages, providing a higher terrain accuracy in regions where the pilot expects to fly. More information about the offline support and Offline packages can be found in Section 5.14.

Checklist engine

The Checklist engine provides a back end for the In-flight checklists feature described later in Section 5.12. The engine is responsible for loading the checklists attached to the Aircraft profile used during the flight and keeping track of their progress. Whilst in flight, the checklist engine listens to flight phase changes published by the Navigation engine and selects the proper checklist to be made active based on the checklists' category labels. Since the Checklist engine runs inside a service, the TTS and speech recognition capabilities are available even when the application is running in the background.

5.12 Dashboard

The Dashboard is the main interface presented to the pilot when flying. It is tightly coupled to the Flight engine and its sub-components, described in the previous section. According to the design specification proposed in Chapter 4, the Dashboard displays basic flight information obtained from the Flight engine in an indicator panel in the bottom-left corner. A map, which takes most of the screen's real estate, tracks the current position of the plane. Behind the plane, a breadcrumb line visualizes the aircraft's flight path. Another line depicts the planned flight route. The Dashboard interface is presented in Figure 5.14.

Information about the next waypoint obtained from the Navigation Engine is displayed in a card on the top of the screen. In order to uplift the user experience, all information cards presented in the Dashboard interface are animated using the built-in animation capabilities of Jetpack Compose. The action button, located in the bottom right corner, opens a dialog that allows the pilot to quickly access other application features, such as Documents or Maps. Upon pressing the back button, the pilot is given the option to either finish the flight or minimize it, in which case the flight is continued in the background. Afterward, the Dashboard can be opened again from the Home screen, by clicking on a banner displayed at the top of the screen, which signifies that a flight is in progress.

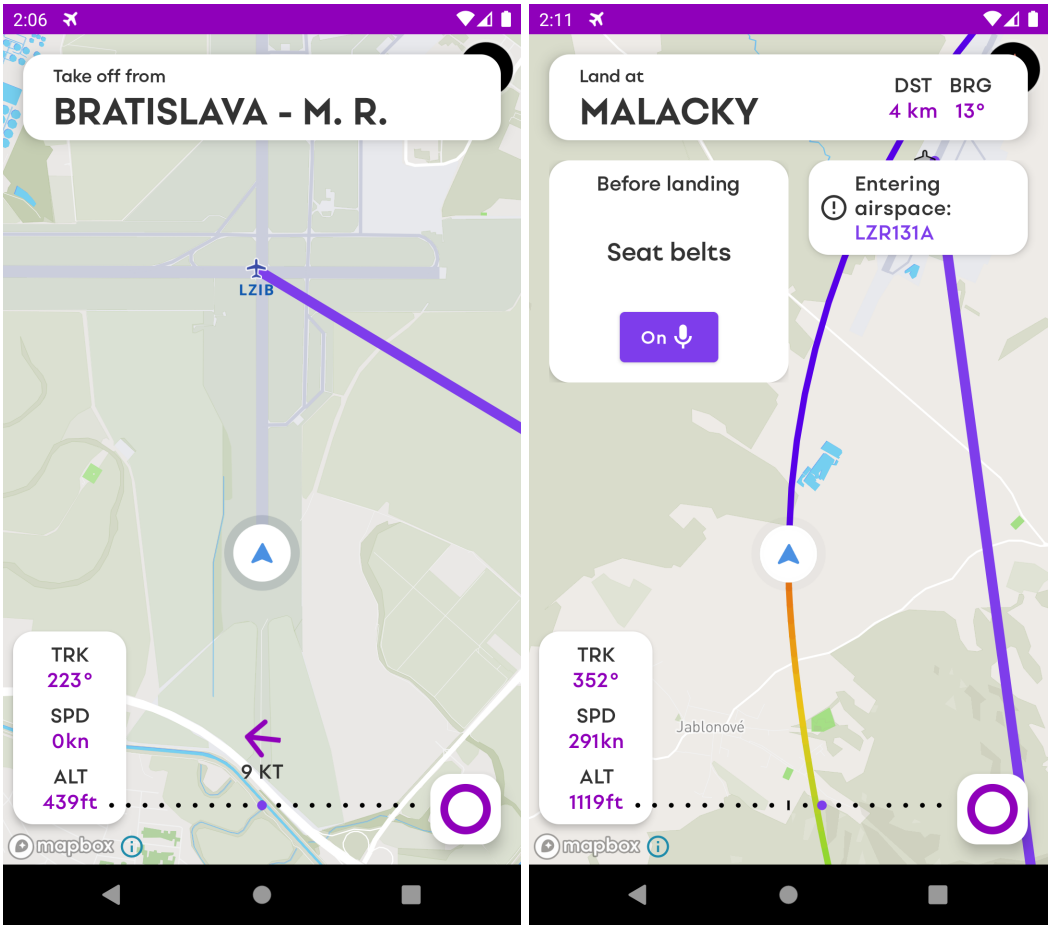


Figure 5.14: Dashboard interface.

Turning rate indicator

The Dashboard displays a line on the map in front of the aircraft, which approximates the aircraft's future flight path. The prediction is calculated by processing the stream of the aircraft's positions. The angles and distances between each position are continuously calculated. The aircraft's true turn angle and speed are estimated from the angle and distance deltas using a weighted average of the last 8 values. The number has been chosen so as to achieve a balance between smooth prediction and high prediction latency. The weighted average uses linearly decaying weights, in order to increase the accuracy of the estimation by giving more importance to recent values.

To approximate the next position of the aircraft, the estimated angle and speed are used to walk in a set distance and direction from the current position. By repeating this procedure by a certain number of steps, a set of points forming the prediction path can be obtained. Increasing the number of iterations results in a longer prediction path, at the cost of lower accuracy.

Course Deviation Indicator

The Course Deviation Indicator (CDI) instrument indicates the aircraft's position relative to the flight path. The pilot can use this instrument to determine, whether the aircraft is

laterally on the flight path, or whether the aircraft has deviated from the path to either left or right. In the Dashboard, the CDI is composed of a black-dotted scale and a purple indicator. When the purple indicator is located in the middle of the scale, it means that the aircraft is on course. If the purple indicator is off-center, the pilot must steer the aircraft until the indicator returns to the center, at which point, the pilot has intercepted the flight path.

The CDI is a custom-made Composable drawn using the graphics capabilities provided by Compose natively. The data passed to the indicator are computed by the Navigation engine, described in Section 5.11. To calculate the deviation value, it is first needed to calculate the distance between the aircraft's position and the closest point on the flight path relative to the aircraft. However, the distance on its own is not enough to decide whether the aircraft is located to the left or to the right of the flight path. In order to determine at which side the aircraft is located, the two points used in the first step are converted to two-dimensional vectors using their latitude and longitude. Afterward, the side is established by the sign of the cross product between these two vectors.

Wind indicator

During take-offs and landings, information about the current wind conditions is essential to the pilot. Usually, the pilot receives this information from the radio communication, or from the current METAR published for the airport in question. When the aircraft is taking off and landing, the Dashboard in NaviPilot automatically displays a wind indicator at the bottom of the screen. The indicator retrieves the current wind direction and speed from the airport's current METAR, if it is available. The indicator itself consists of a wind speed label and an arrow, which rotates in the direction of the wind relative to the aircraft's heading.

Alerts

During the flight, alerts generated from the Alert engine described in Section 5.11 are displayed in the Dashboard. When an alert arrives, it appears in the top-right corner of the screen, accompanied by an acoustic chime. The alert itself is displayed in a card, which contains the alert's description and an icon, denoting the alert's priority and its type.

The alert's content is automatically read aloud using a Text-to-Speech (TTS) engine provided by the Android system. Multiple alerts can be active, in which case they form a queue. If there are more than three alerts active, the oldest alert is dismissed, in order to avoid obscuring the map. Alerts are dismissed automatically by default after a set period of time, although they can also be dismissed manually with an intuitive swipe gesture.

In-flight checklists

The Dashboard has the ability to present a checklist card containing checklists defined in the current Aircraft profile. The checklist card is invoked automatically when a flight phase change is detected, provided that the Aircraft profile contains a checklist that is marked with the appropriate category label.

The card displays the checklist items one at a time, as they are checked off. The content of the checklist item is displayed in a textual form in the card, but also read aloud using the TTS engine. To check off a checklist item, the user can either press a button at the bottom of the checklist card, or say the keyword using his voice, if the keyword is supported by

the Automated checklists model described in Section 3.2.2. In order to make it simple to determine whether the keyword is supported, the button displays a small microphone icon, signifying that the keyword is ready to be detected by voice.

5.13 Tools

The tools screen includes a converter between physical units that could be useful to the pilot during flight preparations. The supported quantities include weight, speed, length, and more. Choosing a quantity takes the user to a converter screen with a list of units of that selected quantity, as shown in Figure 5.15. Each unit has a text field next to it, displaying the current quantity value converted to that specific unit and the symbol of the unit. All text fields are automatically synchronized, so that when a user inputs a quantity into a text field of one of the units, all other text fields are recomputed to show the converted values.

Under the hood, the unit converter screen uses the Unit of Measurement API library to carry out conversions. The API provides a set of Java language programming interfaces for handling physical units and quantities [74]. The Compose interface is backed by a single state variable storing the current quantity value in a unit-less format, using the Quantity class of the Unit of Measurement API. Since view widgets in Compose do not have internal states on their own, the text fields can observe and mutate the main state variable directly. This eliminates the need to synchronize multiple states, potentially avoiding errors that could arise due to the added complexity.

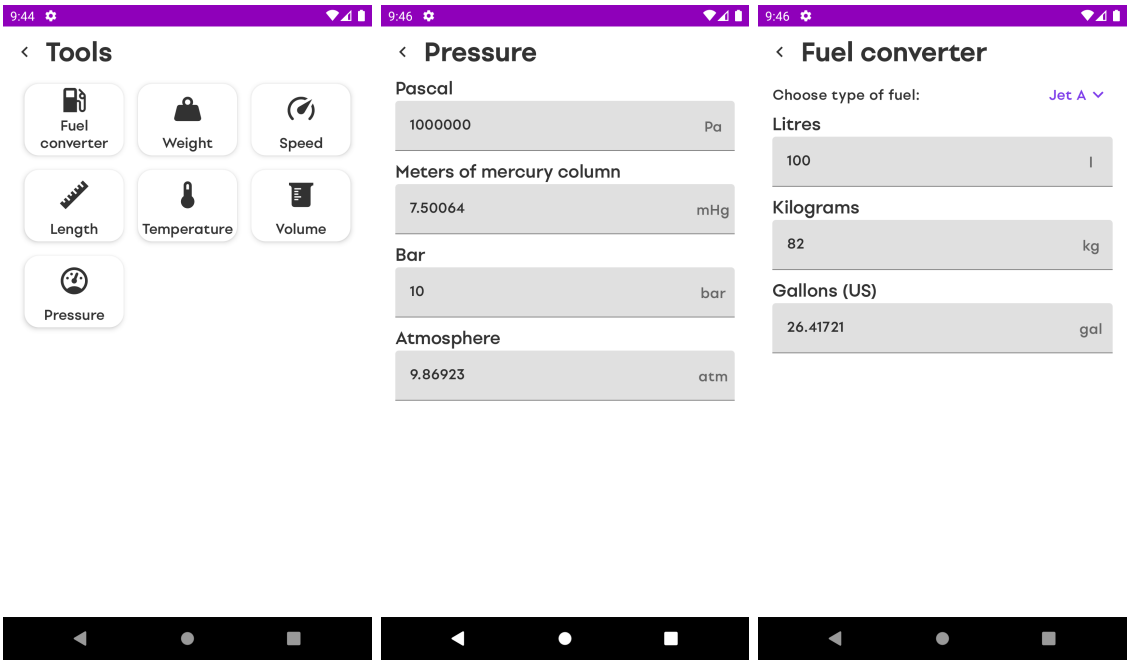


Figure 5.15: Tools screen, Unit Converter screen, and Fuel Converter screen.

Built similarly to the Unit Converter, the second computation tool found in NaviPilot is the Fuel Converter. This screen enables the pilot to quickly calculate the weight of the fuel if the volume is known, and vice versa. Fuel Converter supports multiple popular fuel

types, used by General Aviation pilots. This feature is useful during weight and balance calculations before the flight, as described in the previous Section 5.9.

5.14 Offline support

In NaviPilot, there are two data sources that require an internet connection — Maps and Aeronautical Information Publication (AIP) data, downloaded from OpenAIP. Both of these features have been implemented in a way that makes them unaffected by the lack of internet connectivity, except for the first initialization. Below are more details on how this has been achieved.

Offline maps

Maps in NaviPilot are implemented using the Mapbox library, as described in Section 5.8. The Mapbox SDK for Android includes offline support out-of-the-box. Besides automatically caching tiles during normal usage, the developer can create offline regions defined by coordinate bounds and a zoom level. Once an offline region is defined, it is automatically pre-cached to disk, ready to be used during offline conditions.

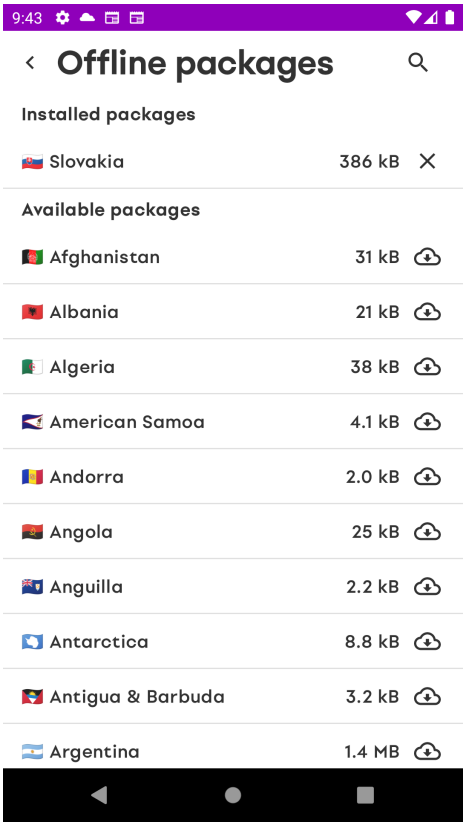


Figure 5.16: Offline packages screen.

Offline packages

Several features of the application, such as the Airport Catalog and Flight planner, rely on data retrieved from OpenAIP. In order to speed up the loading process and save bandwidth,

the OpenAIP catalog is cached on disk. To facilitate fast search, parsed POIs are in addition stored in an indexed SQLite database, using the Room library (described in Section 5.6),

However, downloading the entire OpenAIP catalog is often not necessary, since the user is likely to need data only of a few countries where they plan to fly. The introduction of Offline packages aims to save bandwidth and declutter the interface by letting users download data only of the countries, in which they are interested.

The user can download and delete these packages in the Offline packages screen. The screen displays a list of currently installed packages and the ones available for download, retrieved from the current OpenAIP catalog. After initiating a download of a package, the OpenAIP data for the selected country, together with offline maps, are downloaded in the background. A screenshot of the Offline packages interface can be seen in Figure 5.16.

First run initialization

It would result in a bad user experience if the user opened a map that had some tiles missing due to the lack of internet connection. That is why, besides downloading offline maps for countries installed in Offline packages, a low-resolution offline map of the entire world is pre-loaded as well.

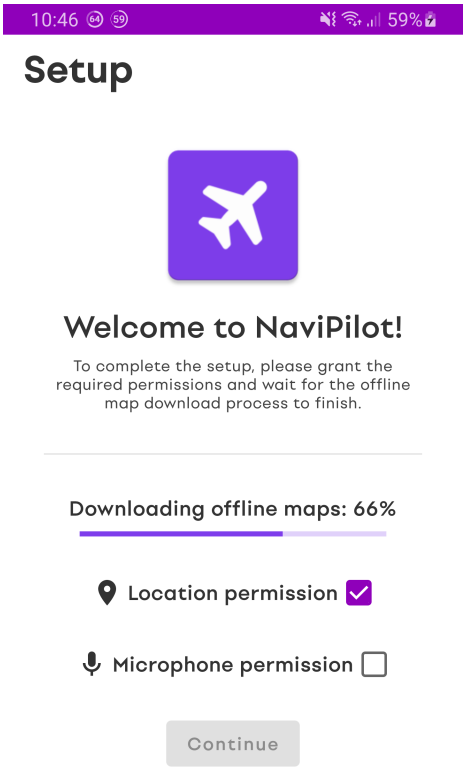


Figure 5.17: Setup screen displayed during the first launch.

This process should ideally be carried out in the background, while the user is allowed to use the application normally. However, possible inconsistencies could arise in cases where the download process would be interrupted. Due to this, a Setup screen is introduced. The Setup screen shown in Figure 5.17 serves as a gateway to the application, allowing the user to use the application only when the required offline maps of the world have finished downloading.

The Setup screen also takes advantage of the opportunity to request permissions required by the system to ensure the application can work properly. NaviPilot requires the location permission and the microphone permission, the latter of which is utilized by the Automated checklists feature described in Section 3.2.2. The permission request process is simplified using the open-source library Peko [10].

Chapter 6

Testing and Evaluation

This chapter outlines the methods used to test the developed application. These include automated unit tests, analysis of resource usage, and user testing. Both testing specifications and the evaluation of their results are described below.

6.1 Unit tests

The project contains a handful of unit tests scattered throughout the modules of the project. These tests are written for features that are critical, or where it made sense to adopt an approach based on the principles of Test Driven Development (TDD). This approach involves writing the unit tests describing the behavior of the code before the actual implementation. Features in this project where this approach is suitable are mainly various converter functions and functions transforming collections.

6.2 Resource usage

The Android Studio IDE includes several tools aimed to help the programmer debug and analyze the behavior of the application in development. One of these tools is the Android Profiler [32]. The Profiler provides real-time data about the application's CPU, memory, network and battery usage. The data is presented through a series of continuously updated graphs, located in a designated tool window inside the IDE.

To measure the resource usage of the application, several readings of the resource usage values from the Android Profiler are made in different application scenarios. The measurements are carried out on a Galaxy S9 phone with the Android 10 OS, running the debug version of the application. The results of the measurements are presented in Table 6.1.

	Setup	**Home (Idle)**	**Flight planning**	**Dashboard**
CPU	20 %–50 %	0 %	20 %	30 %–45 %
Memory	250 MB–300 MB	250 MB	500 MB	600 MB
Network	Intensive	None	Light	Light
Energy	Light	None - Light	Light	Light

Table 6.1: Resource usage by application screen.

It is important to note, that although the table above states, that the network is used in the Flight planning and Dashboard screens, the application does not require it. As described in Section 5.14, NaviPilot supports offline conditions without any limitations. The network usage is caused by the Mapbox library, which uses the connection to refresh any stale map tiles, and to download tiles that are more detailed than the ones saved in the offline cache.

6.3 User testing

In order to measure the user experience and usability of the application, several testing sessions have been conducted with potential users. In total, 10 users have been asked to perform a set of tasks and answer questions about their experience. The tasks involve typical use cases of the application, that would be regularly performed by the users during standard use.

Each user has been first given a quick introduction to the application, in order to familiarise themselves with the goal of the application and the extent of its features. Afterward, the users performed the tasks given. During this time, the actions and the behavior of the users have been observed. If a user got stuck during one of the steps, this fact has been noted and they were given a hint in order to allow them to continue. The time taken to complete the tasks has been recorded and compared to the time achieved by the author. The list of tasks given to the users can be found below:

1. Download the Offline package for the country of Slovakia.
2. Find the frequency of the M. R. Stefanik airport in Bratislava.
3. Find the current temperature according to METAR for the Poprad Tatry airport.
4. Create a new Aircraft profile for a test plane model with the following parameters:
   - Maximum takeoff weight is 1200 kg.
   - Maximum landing weight is also 1200 kg.
   - Empty aircraft weight is 800 kg, with 0.1 m arm.
   - Add two weight items, for baggage and fuel.
   - Add two new CG envelope points.
5. Create a new Checklist for the Cessna 172S aircraft with items:
   - Seat belts — ON.
   - Baro — SET.
6. Plan a new flight with the following waypoints:
   - Takeoff from Bratislava airport.
   - VOR-DME Jánovce.
   - The city of Hlohovec.
   - Coordinate point to the south of Piešťany.
   - Land at Piešťany airport.

7. After confirming the planned flight, determine whether it is feasible to fly with Cessna 172S if:
   - The pilot weighs 75 kg.
   - The front passenger weighs 65 kg.
   - There is 110 kg of fuel in the fuel tank.
8. Convert pressure of 1.5 bar to mHg.
9. Determine the weight of 120 l of the Jet A fuel.
10. Setup a Documents directory and open a document.

Overall, all users were able to complete the tasks without any major issues. They were able to find the correct screen, where they could complete the task most of the time. In a small number of cases, when they opened the wrong screen, they quickly realized that fact and opened the next alternative screen, which was the correct one. The median recorded times of the author and the users for each task, together with their minimums and maximums are shown in Table 6.2. A chart showing recording times by each user per task is shown in Figure 6.1.

Task #	Author time	Minimum time	Median time	Maximum time
1	8 s	8 s	14 s	49 s
2	5 s	5 s	8.5 s	24 s
3	8 s	10 s	15.5 s	50 s
4	49 s	56 s	89.5 s	280 s
5	32 s	27 s	54.5 s	81 s
6	28 s	22 s	59 s	120 s
7	12 s	15 s	27.5 s	35 s
8	10 s	9 s	22 s	25 s
9	7 s	6 s	10.5 s	21 s
10	10 s	12 s	27.5 s	47 s

Table 6.2: Table of user testing task times.

Most of the tasks were completed in tens of seconds, while some users were able to complete some short tasks even below the 10-second mark. The longer tasks, such as the tasks number 4, 5, and 6, have a noticeably high variation in the recorded times between the users. These discrepancies have been caused mainly by the input-intensive nature of the tasks, as some users had trouble typing information into the text fields, since the testing has been carried out online over a remote desktop connection sharing an *Android Emulator* screen, which introduced some latency. Moreover, many users were not familiar with the Android OS environment at all, which coupled with the fact, that they had to use a keyboard and mouse peripherals to control the device, resulting in additional time lost.

Compared to the time of the author, the times of the users were higher in general, which is to be expected from first-time users. Repeated experiments have shown, that once the user completed a task for the first time, they were able to reduce their time when repeating the task to up to a third of the original time.

The user times recorded for the task number 1, as shown in Figure 6.1, show an interesting pattern, where one group of users were able to complete the task quicker than the other group. This can be attributed to the way the users chose to complete the task of downloading the Offline package for the country of Slovakia. Here, the users who chose to use the included search function were able to complete the task faster than those, who decided to scroll through the list of packages.

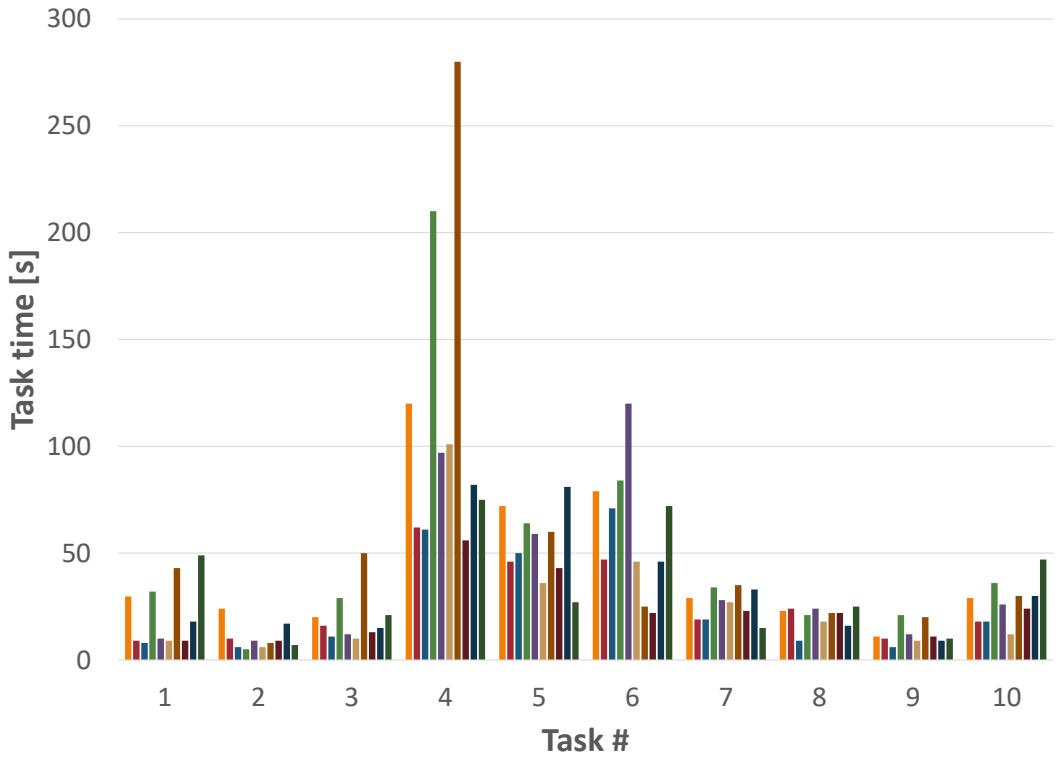
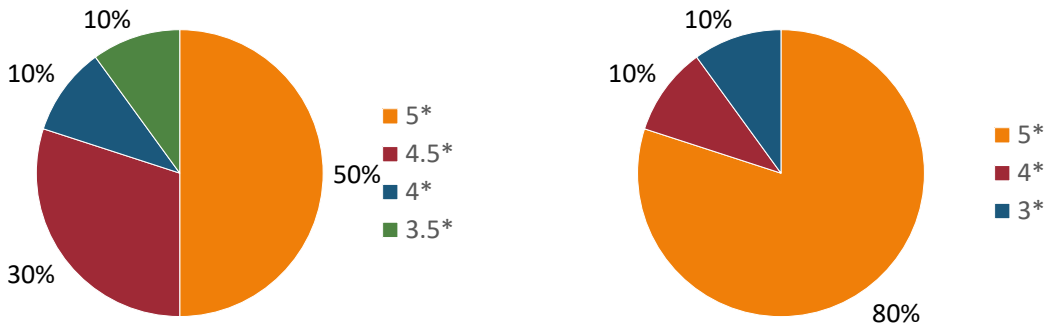


Figure 6.1: Chart of times achieved by testing users per task.

After completing the tasks, the users were given the following five questions related to their experience with the application:

1. How would you rate (1 to 5) the usability of the application as an EFB?
2. How would you rate (1 to 5) the graphic design of the application?
3. Which functionality of the application did you find the most interesting?
4. Which useful features do you miss in the application?
5. Would you personally use the application when flying on a simulator or in the reality?

As is apparent from the answers to the first two questions, shown in Figure 6.2, the application has received an overall positive response. The graphic design suited nearly all of the participants, as they found the color scheme easy on their eyes. From the usability



(a) Usability rating

(b) Graphic design rating

Figure 6.2: Charts representing the feedback given by users.

perspective, the application was found suitable as an EFB device for pilots of General Aviation, flying according to VFR, by most of the users. When answering the question number 5, all participants have stated, that they would personally use the application when flying. Based on these facts, the goal of creating an EFB application can be considered to be successfully fulfilled.

Some users have decreased the usability rating due to some issues with the User Interface, which hindered the process of inputting text and values or finding the correct button to press when performing the testing tasks. Based on the feedback, several changes have been made to the UI, such as better distinction of actionable buttons using a different color, or the inclusion of an arrow for back navigation in the top bar. Another change that has been made is, that when inputting values to number fields, the previous value now gets cleared upon focus.

When the users were asked which feature seems the most interesting to them, their answers have varied greatly. Features considered as interesting by some were the real-time calculation of the Center of Gravity in the Weight and Balance screen and the ability to bring their own checklists and create new Aircraft profiles using the integrated editor. This option of customizability was highly welcomed, as some pilots expressed that other EFB applications, that they have used in the past, have been lacking in this department.

Other users chose the Flight planner and the Tools screen as the most interesting to them, due to the way they were approached from the User Experience (UX) standpoint, where making any changes would immediately reflect on the information presented on the screen. At last, the remaining users liked the way the application is able to present the information in a user-friendly manner, such as in the METAR tab of the Airport detail described in Section 5.3.

Missing features, according to the users, included support for saving frequently flown flights, search and filtering capabilities in the Logbook, and the availability of navigation fixes in the waypoint database. Two users have requested integrated support for charts and the ability to overlay them over the map, a feature found in many commercially available Electronic Flight Bags, as seen in Subsection 3.1.1 in the Market Research chapter.

Other suggestions made by the users were the inclusion of a legend to the maps displayed in the application and the availability of flight statistics, which have been omitted from the

design proposed in Section 4.2. Although the map interface already shows the borders of restricted airspaces, one of the users has suggested that the application provide an explicit warning when the user plans a flight route that passes through restricted airspace.

Conversations with pilots during user testing revealed, that according to them, the most important features during flights are those, that help pilots quickly find information, such as radio frequencies and weather conditions at an airport, and warn them before entering into airspaces and restricted flight areas. NaviPilot already provides access to airport information and airspace alerts, as described in Section 5.11. However, there is room for improvement left in this regard, where these types of information could be displayed to the user more prominently, by including additional popups and information cards during the flight.

Chapter 7

Future Research

The application provides a solid ground for further development and the inclusion of additional features. Many features requested by the users during user testing, such as the integrated support for charts with arrival/departure procedures and an extended waypoint database, are directly tied to the source of the data. As described in Section 5.3, NaviPilot uses the freely available OpenAIP as its data source. Turning the application into a commercial product could provide funds for higher quality paid data sources, which would open the doors to implementing these types of features.

The upcoming v10 release of the Mapbox Android SDK library, used to display maps in the application, as explained in Section 5.8, promises to provide increased rendering and offline performance [53]. Moreover, the newly added support for rendering 3D terrain could provide a whole new experience in the Dashboard. Displaying the position of the airplane in a 3D environment could provide better situational awareness to the pilot during flights.

The support for alerts, described in Section 5.11, could be extended by implementing additional types, apart from the airspace warnings. The extra alert types could include terrain alerts, weather alerts, traffic alerts, and more. The traffic alert feature would use data from another potential feature, which would use an internet connection, or an external ADS-B receiver to display live traffic on the map.

Such a device, similar to the one described in Subsection 3.3.4 in the Market Research chapter, would potentially also be able to provide accurate positional and attitude data about the aircraft. Since NaviPilot was designed and architected to include support for both the embedded GNSS receiver of the mobile device and the data from a simulator, it is ready to be extended to include support for these external devices in the future.

Finally, the Automated checklists feature, outlined in Section 3.2.2, has a lot of room for further development. Currently, it is in an experimental state, serving as a proof of concept with only three supported keywords. This feature could be extended to support additional keywords, while the accuracy could be improved using newer models and longer training. However, the Automated checklists feature, and Keyword Spotting tasks in general, are inherently complicated and could be treated as a full project on their own.

Chapter 8

Conclusion

Electronic Flight Bags (EFBs) are helpful pieces of software, as proved by the introduction part of this thesis, where the motivation behind the use of EFBs is explained. EFB applications must, however, adhere to certain regulations, summarized in Chapter 2. Afterward, a market research has been conducted on existing EFB applications, examining the common features offered by them.

Based on this newly acquired knowledge, a task has been set to design and implement such an application. The application's features have been described, in accordance with the regulations and user expectations as established by the leading EFB applications on the market. Following the feature description, a User Interface design has been proposed.

The feature specifications and UI design have been used to implement a full-fledged application for the Android OS, named NaviPilot. The codebase of the application showcases the latest practices and tools of modern Android development, while its modular architecture offers an easy path forward for further expansion.

The final product offers the basic functions of an EFB application, such as the support for Documents, Logbook, Flight planning, aviation Maps, Tools, and the Airport Catalog with information about airports available globally. NaviPilot can be customized by the users with the built-in Aircraft profile and Checklist editors, in order to tailor to each user's individual needs. The integrated offline support ensures, that the application stays in working order even during conditions without an internet connection.

While these features are vital for an Electronic Flight Bag, they are already commonly available in commercial EFB applications, as seen in the Market Research Chapter 3. In an effort to bring innovation to the existing EFB applications, NaviPilot includes two experimental features, the Automated checklists feature and the AR Preview, described in sections 3.2.2 and 5.10.

The final product has been tested with potential users in the User testing Section 6.3, where it has received positive feedback, completing the goal of this thesis to develop an EFB application. However, as discussed in the Future Research Chapter 7, the application has a lot of room to grow in the future, with the potential inclusion of promising features such as 3D Maps and the support for connection to external devices, which would further improve the usability of the application during real-world flights.

Bibliography

[1] airBaltic. *United Airlines saves 643,000 litres of fuel by using lighter paper on inflight magazine*. July 2014. [retrieved 2020-11-22]. Available at: https://www.internationalairportreview.com/news/17234/airbaltic-pilots-go-green-with-ipads/.

[2] AppBrain. *Number of Android apps on Google Play*. June 2021. [retrieved 2021-06-17]. Available at: https://www.appbrain.com/stats/number-of-android-apps.

[3] Arconics. *Qatar Airways Goes Live with AeroDocs Documentation Management by Arconics*. November 2016. [retrieved 2021-01-09]. Available at: https://www8.garmin.com/aboutGPS/waas.html.

[4] Berg, A., O'Connor, M. and Cruz, M. T. *Keyword Transformer: A Self-Attention Model for Keyword Spotting*. 2021.

[5] Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T. et al. *The GeoJSON Format* [RFC 7946]. RFC Editor, august 2016. DOI: 10.17487/RFC7946. Available at: https://rfc-editor.org/rfc/rfc7946.txt.

[6] Choi, S., Seo, S., Shin, B., Byun, H., Kersner, M. et al. *Temporal Convolution for Real-time Keyword Spotting on Mobile Devices*. 2019.

[7] Coil Contributors. *Coil*. July 2021. [retrieved 2021-07-24]. Available at: https://coil-kt.github.io/coil/.

[8] Coucke, A., Chlieh, M., Gisselbrecht, T., Leroy, D., Poumeyrol, M. et al. *Efficient keyword spotting using dilated convolutions and gating*. 2019.

[9] Delta Airlines. *Delta to equip 11,000 pilots with Microsoft Surface 2 tablet devices*. 2013. [retrieved 2020-11-20]. Available at: https://news.delta.com/delta-equip-11000-pilots-microsoft-surface-2-tablet-devices.

[10] Devcic, M. *PEKO*. June 2021. [retrieved 2021-06-09]. Available at: https://github.com/deva666/Peko.

[11] Easterling, R. *Schema-gen*. February 2021. [retrieved 2021-06-05]. Available at: https://github.com/reaster/schema-gen.

[12] EUROCONTROL. *European AIS Database*. June 2021. [retrieved 2021-06-17]. Available at: https://www.eurocontrol.int/service/european-ais-database.

[13] European Union Aviation Safety Agency. *Easy Access Rules for Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20)*. February 2020. Amendment 16. Available at: https://www.easa.europa.eu/sites/default/files/dfu/AMC-20%20%28Amendment%2016%29.pdf.

[14] Federal Aviation Administration. *Aircraft weight and balance handbook 1999: Faa-h-8083-1*. US Department of Transportation, FAA, 1999. ISBN 1619544814.

[15] Federal Aviation Administration. *AC 120-76D - Authorization for Use of Electronic Flight Bags*. October 2017. Available at: https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_120-76D.pdf.

[16] Federal Aviation Administration. *Standard Operating Procedures and Pilot Monitoring Duties for Flight Deck Crewmembers*. October 2017. Available at: https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_120-71B.pdf.

[17] Federal Aviation Administration. *GNSS Frequently Asked Questions - WAAS*. April 2019. [retrieved 2021-01-09]. Available at: https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/faq/waas/.

[18] Federal Aviation Administration. *AIP AERONAUTICAL INFORMATION PUBLICATION UNITED STATES OF AMERICA*. TWENTY-SIXth ed. 2020. Available at: https://www.faa.gov/air_traffic/publications/media/aip_basic_7_16_20.pdf.

[19] Federal Aviation Administration. *PilotWeb*. June 2021. [retrieved 2021-06-14]. Available at: https://pilotweb.nas.faa.gov/PilotWeb/.

[20] Fltplan. *Fltplan About*. [retrieved 2020-12-01]. Available at: https://www.fltplan.com/about_fltplan.html.

[21] Fltplan. *FltPlan Go*. [retrieved 2020-12-01]. Available at: https://play.google.com/store/apps/details?id=com.fltplan.go.

[22] FltPlan. *FltPlan Go User's Manual*. September 2020. Available at: https://flttrack.fltplan.com/TutorialPDFs/FtPlanGo-iPad-Users-Manual.pdf.

[23] Foreflight. *Foreflight About*. [retrieved 2020-11-30]. Available at: https://foreflight.com/about/foreflight/.

[24] Foreflight. *ForeFlight Feature Focus: Forecast Weather in 3D Preview*. [retrieved 2020-11-30]. Available at: https://www.youtube.com/watch?v=DeTbx17-EYU.

[25] Foreflight. *Pilot's Guide to FOREFLIGHT MOBILE*. 84th ed. Available at: http://cloudfront.foreflight.com/docs/ff/12.9/v12.9%20-%20foreflight%20mobile%20pilot%20guide%20optimized.pdf?_ga=2.117387526.143618085.1606758318-1791363413.1603964506.

[26] Garmin. *Garmin About*. [retrieved 2020-12-02]. Available at: https://www.garmin.com/en-US/company/about-garmin/.

[27] Garmin. *Garmin pilot adds synthetic vision capability*. [retrieved 2020-12-02]. Available at: https://www.garmin.com/en-US/blog/aviation/garmin-pilot-adds-synthetic-vision-capability/.

[28] Garmin. *Garmin Pilot™ for iOS*. November 2020. Available at: https://static.garmin.com/pumac/190-01501-00_ae.pdf.

[29] Google. *Open files using storage access framework*. [retrieved 2021-06-04]. Available at: https://developer.android.com/guide/topics/providers/document-provider.

[30] Google. *Activity*. June 2021. [retrieved 2021-06-17]. Available at: https://developer.android.com/reference/android/app/Activity.

[31] Google. *Android Jetpack*. July 2021. [retrieved 2021-07-20]. Available at: https://developer.android.com/jetpack.

[32] Google. *The Android Profiler*. May 2021. [retrieved 2021-07-25]. Available at: https://developer.android.com/studio/profile/android-profiler.

[33] Google. *ARCore overview*. June 2021. [retrieved 2021-06-07]. Available at: https://developers.google.com/ar/discover.

[34] Google. *Build better apps faster with Jetpack Compose*. June 2021. [retrieved 2021-06-15]. Available at: https://developer.android.com/jetpack/compose.

[35] Google. *DataStore*. June 2021. [retrieved 2021-06-10]. Available at: https://developer.android.com/topic/libraries/architecture/datastore.

[36] Google. *Design - Material Design*. July 2021. [retrieved 2021-07-20]. Available at: https://material.io/design.

[37] Google. *Guide to app architecture*. June 2021. [retrieved 2021-06-13]. Available at: https://developer.android.com/jetpack/guide#best-practices.

[38] Google. *Handling Lifecycles with Lifecycle-Aware Components*. June 2021. [retrieved 2021-06-16]. Available at: https://developer.android.com/topic/libraries/architecture/lifecycle.

[39] Google. *Navigation*. June 2021. [retrieved 2021-06-17]. Available at: https://developer.android.com/guide/navigation.

[40] Google. *Paging library overview*. June 2021. [retrieved 2021-06-08]. Available at: https://developer.android.com/topic/libraries/architecture/paging/v3-overview.

[41] Google. *Save data in a local database using Room*. June 2021. [retrieved 2021-06-08]. Available at: https://developer.android.com/training/data-storage/room.

[42] Google. *Services overview*. January 2021. [retrieved 2021-06-12]. Available at: https://developer.android.com/guide/components/services.

[43] Google. *ViewModel Overview*. April 2021. [retrieved 2021-06-17]. Available at: https://developer.android.com/topic/libraries/architecture/viewmodel.

[44] Gradle. *What is Gradle?* June 2021. [retrieved 2021-06-13]. Available at: https://docs.gradle.org/current/userguide/what_is_gradle.html.

[45] Hardawar, D. *How 'Microsoft Flight Simulator' became a 'living game' with Azure AI*. September 2020. [retrieved 2021-07-22]. Available at: https://www.engadget.com/microsoft-flight-simulator-azure-ai-machine-learning-193545436.html.

[46] Harjula, M. *JSimConnect - a simconnect java client library*. March 2021. [retrieved 2021-06-05]. Available at: https://github.com/mharj/jsimconnect.

[47] Hugo, M. *United Airlines saves 643,000 litres of fuel by using lighter paper on inflight magazine*. January 2018. [retrieved 2020-11-20]. Available at: https://www.traveller.com.au/united-airlines-saves-643-litres-of-fuel-by-using-lighter-paper-on-inflight-magazine-h0nfej#ixzz5c6aJITkq.

[48] International Civil Aviation Organization. *Manual of Electronic Flight Bags (EFBs)*. Secondth ed. 2018. Available at: http://www.icscc.org.cn/upload/file/20190102/Doc.10020-EN%20Manual%20of%20Electronic%20Flight%20Bags%20(EFBs).pdf.

[49] Jeppesen. *Jeppesen charts on Garmin Pilot*. [retrieved 2020-12-02]. Available at: https://ww2.jeppesen.com/jeppesen-charts-on-garmin-pilot/.

[50] Knudian. *OpenAIP2GeoJSON*. April 2020. [retrieved 2021-06-07]. Available at: https://github.com/Knudian/OpenAIP2GeoJSON.

[51] Koebbe, B. *ForeFlight releases biggest update for 2020 – how to use each new feat...* [retrieved 2020-11-30]. Available at: https://ipadpilotnews.com/2020/05/foreflights-biggest-update-for-2020-how-to-use-the-new-features/.

[52] Kpadey, J.-K. *MetarParser*. July 2021. [retrieved 2021-07-24]. Available at: https://github.com/mivek/MetarParser.

[53] Lee, T. *Maps SDK 10 Release Candidate*. June 2021. [retrieved 2021-07-28]. Available at: https://www.mapbox.com/blog/maps-sdk-v10-release-candidate.

[54] Lockheed Martin Corporation. *SimConnect*. 2016. [retrieved 2021-06-05]. Available at: http://www.prepar3d.com/SDKv3/LearningCenter/utilities/simconnect/simconnect.html.

[55] Malchev, I. *Number of Android apps on Google Play*. October 2019. [retrieved 2021-01-03]. Available at: https://android-developers.googleblog.com/2019/10/all-about-updates-more-treble.html.

[56] Mapbox. *Access elevation data*. June 2021. [retrieved 2021-06-15]. Available at: https://docs.mapbox.com/help/troubleshooting/access-elevation-data/.

[57] Mapbox. *Maps for Unity*. June 2021. [retrieved 2021-06-07]. Available at: https://www.mapbox.com/unity.

[58] Mapbox. *Maps SDK for Android*. June 2021. [retrieved 2021-06-07]. Available at: https://docs.mapbox.com/android/maps/guides/.

[59] Mapbox. *Search*. July 2021. [retrieved 2021-07-22]. Available at: https://www.mapbox.com/search-service.

[60] Mapbox. *Tilesets*. June 2021. [retrieved 2021-06-07]. Available at: https://docs.mapbox.com/studio-manual/reference/tilesets/.

[61] Mapbox. *Tilesets-cli*. April 2021. [retrieved 2021-06-07]. Available at: https://github.com/mapbox/tilesets-cli/.

[62] MKergall. *OSMBonusPack*. May 2021. [retrieved 2021-06-13]. Available at: https://github.com/MKergall/osmbonuspack.

[63] Muntenescu, F. and Sathyanarayana, R. *Prefer Storing Data with Jetpack DataStore*. September 2020. [retrieved 2021-06-10]. Available at: https://android-developers.googleblog.com/2020/09/prefer-storing-data-with-jetpack.html.

[64] National Oceanic and Atmospheric Administration. *AVIATION WEATHER CENTER - METARs*. June 2021. [retrieved 2021-06-14]. Available at: https://www.aviationweather.gov/metar.

[65] National Oceanic and Atmospheric Administration. *GLOBE: Get the Data*. June 2021. [retrieved 2021-06-15]. Available at: https://www.ngdc.noaa.gov/mgg/topo/globeget.html.

[66] Niantic. *The games redefining our reality*. June 2021. [retrieved 2021-06-07]. Available at: https://nianticlabs.com/en/products/.

[67] O'Dea, S. *Market share of mobile operating systems worldwide 2012-2020*. November 2020. [retrieved 2020-12-31]. Available at: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[68] openAIP team. *OpenAIP*. February 2020. [retrieved 2021-06-04]. Available at: http://www.openaip.net/.

[69] Rybakov, O., Kononenko, N., Subrahmanya, N., Visontai, M. and Laurenzo, S. Streaming Keyword Spotting on Mobile Devices. *Interspeech 2020*. ISCA. Oct 2020. DOI: 10.21437/interspeech.2020-1003. Available at: http://dx.doi.org/10.21437/Interspeech.2020-1003.

[70] Schastlyvyi, O. *Weather Maps API*. June 2021. [retrieved 2021-06-14]. Available at: https://www.rainviewer.com/api/weather-maps-api.html.

[71] Sentry. *Sentry™ - The Most Full-featured Portable ADS-B & GPS Receiver*. [retrieved 2020-12-03]. Available at: https://flywithsentry.com/sentry.

[72] Snowflake Software. *Laminar Data Hub Overview*. June 2021. [retrieved 2021-06-17]. Available at: https://developer.laminardata.aero/.

[73] Tensorflow. *An end-to-end open source machine learning platform*. June 2021. [retrieved 2021-06-11]. Available at: https://www.tensorflow.org/.

[74] Units of Measurement project. *Units of Measurement - About*. July 2021. [retrieved 2021-07-23]. Available at: http://unitsofmeasurement.github.io/pages/about.html.

[75] Unity. *Unity Platform*. June 2021. [retrieved 2021-06-07]. Available at: https://unity.com/products/unity-platform.

[76] Warden, P. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018.

[77] Whitwam, R. *Scoped Storage in Android 11 will have exemptions for older APIs and 'core' apps like file managers*. February 2020. [retrieved 2021-06-04]. Available at: https://www.androidpolice.com/2020/02/19/scoped-storage-in-android-11-will-have-exemptions-for-older-apis-and-core-apps-like-file-managers/.

[78] Wilhelmstötter, F. *JPX*. April 2021. [retrieved 2021-06-13]. Available at: https://github.com/jenetics/jpx.

[79] Zhang, Y., Suda, N., Lai, L. and Chandra, V. *Hello Edge: Keyword Spotting on Microcontrollers*. 2018.

Appendix A

Contents of the Included Storage Media

```
/
├── program_source/ ..... project source code
├── report_source/ ..... technical report source
├── screenshots/ ..... screenshots from the application
├── design_files/ ..... design files
├── xkusik00_report.pdf ..... thesis document
├── xkusik00_report_print.pdf ..... thesis document for print
├── project_requirements.txt ..... project requirements
├── NaviPilotDemo.mp4 ..... application usage demo video
└── NaviPilot.apk ..... application installation package
```