Prototypical Implementation of an iPad App for Drone Missions & Concurrency in Software Applications

Thursday 13th January, 2022 - 16:56

Flavio De Jesus Matias
University of Luxembourg
Email: flavio.dejesus.001@student.uni.lu

This report has been produced under the supervision of:
Benoît Ries
University of Luxembourg
Email: benoit.ries@uni.lu

***Abstract*—This document presents the bachelor semester project of Flavio De Jesus Matias under the tutoring of Benoît Ries. The first part of the project consists of presenting concurrency in software applications, which includes, among other things, the concepts, technologies, and methods used to create concurrent applications. The second part of the project consists of the prototypical implementation of an iPad app for drone missions based on a graphical user interface prototype from an earlier BSP project.**

1. Introduction

In a world where computer systems are evolving at a record-breaking pace, concurrency plays a major role when it comes to the efficient use of hardware resources. Concurrency reduces the latency and response time of computers through the efficient use of available resources and thus drastically improves the performance of a system. Without concurrency, computers would have to process each request individually, which is no longer feasible for the digitally connected world in which we live. As an example, let's imagine that the 1.9 billion daily active users of Facebook [1] were processed one at a time. The tremendous wait users would experience is simply not feasible. Hence, concurrency is required to handle most of these requests at the same time.

For this reason, in this Bachelor Semester Project, we will focus on how concurrency is achieved in software applications. The first part of the project will start with a brief introduction to concurrency, followed by the related concepts, methods, and technologies used. Additionally, the last section of the first part will focus on the performance improvements in a concurrent environment and highlight why one should write parallelizable code. The second part of the project is the prototypical implementation of an iPad app for drone missions. The app is part of a bigger project focusing on the monitoring of the resiliency of Luxembourg ecosystems such as natural ecosystems and agricultural fields.

2. Project description

2.1. Domains

2.1.1. Scientific. The scientific domain of this project is concurrency in software applications and in general. Concurrency is the fact of multiple events happening simultaneously during an overlapping time period instead of being executed sequentially. This report looks at what Concurrent Computing is, the concepts involved, and the necessary technologies and methods such as processes, threads, multithreading and multicore systems.

2.1.2. Technical. The technical domain of this project is the prototype implementation of an iPad app for drone missions. Mobile development for iPadOS is an advanced version of iOS that requires the latest version of Xcode, Apple's integrated development environment (IDE). Xcode provides the graphical user interface (GUI) for developers to develop apps, but the programming language used is Swift. In this project, we are using the SwiftUI framework to create the app's GUI ourselves. To connect and interact with the DJI Matrice 210 drone, the DJI Mobile SDK for iOS is required. In addition, Google's Firebase platform provides the underlying database for the app.

Swift
Swift is a general-purpose programming language developed by Apple in 2014. The purpose of this programming language was to replace the outdated Objective-C programming language and to create very performant applications for macOS, iOS and other operating systems that belong to the Apple ecosystem. [2]

SwiftUI
SwiftUI is a framework and user interface toolkit developed by Apple [3] to extend the Swift programming languages by providing innovative and simple ways to build user

interfaces for its iOS, iPad, and macOS apps. It extends Swift with a set of tools that allows developers to gain time by creating declarative UIs.

DJI Mobile SDK
The DJI Mobile SDK is the software development kit created by DJI [4] to give developers the possibility of controlling their drone capabilities. There are 2 different versions available, one for iOS and one for Android. Using the iOS version, we can connect to the DJI Matrice 210 drone with our app in order to plan and execute missions.

Firebase
Firebase is a platform provided by Google for developing web and mobile applications [5]. It offers a real-time database, which means that the data is synchronized in real-time between all connected devices. In addition, Firebase provides an authentication service that includes many types of authentication methods like classic email and password, Facebook, Twitter, etc.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The targeted scientific deliverable of this project is to conduct research on concurrency in software applications and in general. The targeted scientific question in this deliverable is "How is concurrency achieved in software applications?". The deliverable introduces concurrency and explain the concepts, technologies, and methods used to achieve concurrency in software applications. Furthermore, it will describe how concurrency is achieved in Swift, which is the programming language used to develop the technical deliverable.

2.2.2. Technical deliverables. The targeted technical deliverable of this project is to build the iPad application. This app is built following a complete graphical user interface from a previous BSP project [6]. The app will interact with a DJI Matrice 210 drone using the DJI Mobile SDK, whose goal is to monitor ecosystems such as agricultural fields. This deliverable represents the first version of the app and will contain some functionalities such as planning a mission and connecting to the drone, amongst others.

3. Prerequisites

3.1. Scientific pre-requisites

This Bachelor semester project does not have any scientific pre-requisites, as all important concepts and terms are presented and explained to the reader. For some people, however, the information in this report may not be sufficient and it is therefore recommended that the reader have some knowledge of computer science such as the BiCS Computing Infrastructure course in the 2nd semester.

3.2. Technical pre-requisites

The technical pre-requisite of this project is the knowledge of the programming language Swift and the framework SwiftUI. In the technical deliverable, the app will be created using the SwiftUI framework, which provides UI development tools for Apple devices such as iPhone, iPad, and Macs. Some elements of the framework will be presented and explained, however, it is expected that the reader has some knowledge of SwiftUI in order to fully understand the simple parts of the app. The expected knowledge consists of the basic SwiftUI elements such as VStack, HStack, ZStack.

4. Concurrency in Software Applications

4.1. Requirements

The main requirement for this deliverable is to present concurrency in general and in software applications. The deliverable should answer the scientific question "How is concurrency achieved in software applications?" by starting with an introduction to concurrency and then the related terms and technologies. In addition, the deliverable should list and explain concepts related to concurrency as well as concurrency control and why it is important. Ultimately, the deliverable should explain why one should consider writing parallelizable code. This yields the following functional requirements:

- **FR1:** The deliverable shall introduce concurrency in software applications and in general.
- **FR2:** The deliverable shall introduce concurrency-related terms and technologies.
- **FR3:** The deliverable shall list and explain concepts related to concurrency.
- **FR4:** The deliverable shall briefly present concurrency control and process synchronization mechanisms.
- **FR5:** The deliverable shall explain why one should consider writing parallelizable code.

4.2. Design

This deliverable discusses concurrency in software applications. Several books and scientific publications are studied in order to provide the information presented in this deliverable:

- The Structured Phase of Concurrency [7]
- The symbiosis of concurrency and verification: teaching and case studies [8]
- Parallel Programming for Multicore and Cluster Systems [9]
- Multicore Processors - A Necessity [10]
- Multicore Processors - An Overview [11]
- Multicore Programming: Increasing Performance through Software Multi-threading [12]
- Introduction To Concurrency In Programming Languages [13]
- Mastering Concurrency Programming with Java 9 [14]
- Concurrency Control: Methods, Performance and Analysis [15]
- Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores and Condition Variables [16]
- An Introduction to Parallel Programming with OpenMP [17]

The entire section 4.3 is divided into **five** different parts of the production of the scientific deliverable.

1) The first part introduces concurrency in software applications by talking about what concurrency is, why it was invented, and what general problems it solves.
2) The second part, focusses on the concurrency-related terms and technologies. This section explains important terms in the context of concurrency and includes processes, threads, CPUs, and CPU cores, among others.
3) The third part discusses the concepts of concurrency. The focus lies on the difference between concurrency and parallelism in software applications in addition to possible problems that can arise in such applications.
4) The fourth part introduces concurrency control and why it is important. In addition, process synchronization techniques are briefly explained.
5) Finally, the final part focusses on why one should consider writing parallelizable code, as well as its scalability and performance limitations.

4.3. Production

4.3.1. Introduction to concurrency in software applications. Concurrency is the tendency for several events to happen at the same time. Different steps of a single process (instance of a program, more about processes in section 4.3.2) can be broken down and executed by several computers simultaneously. In the end, the results can be concatenated and give the same result as without concurrency. Although concurrency was first invented in the 19th and early 20th century to solve problems such as handling multiple trains on the same railroad, the computer science of concurrency only began with the introduction of the mutual exclusion problem by Edsger Dijkstra's in 1965 [18] [19]. The goal of concurrency is to divide the work among different computers with the ultimate goal of increasing computer performance and throughput. Typically, concurrent processes are more efficient than non-concurrent processes [7] if we consider the computer throughput in a fixed amount of time. Thus, it is important to address and consider the benefits of concurrency.

4.3.2. Concurrency-related terms and technologies. There are several terms and technologies that have to be known in the context of concurrency in order to create and understand concurrent computer programs. The most important ones are introduced and explained below.

CPU

The central processing unit (CPU), or processor, is the main chip in most electronic devices that can run programs. A CPU is a general-purpose device that can interpret binary signals (0 or 1). CPUs are directly connected to the motherboard and require adequate cooling to prevent damage. Because it's a very powerful component that executes billions of instructions per second, CPUs get extremely hot. CPUs are made up of several small components, but we'll only focus on one of them: the CPU cores. Nowadays there are single and multi-core processors. More on this will be presented in the section **Single-Core vs. Multi-Core Processors**.

Processes
A process is basically an instance of a program being executed in binary format by a processor. In other words, a process is a program being loaded into memory and executed [9]. Each process gets its own dedicated address space in memory and a unique process identification ID called **PID** in the operating system. The address space allocated in memory is made up of several parts, which can be seen in Figure 1. Every part is briefly described below:

- **Stack:** The memory stack includes the temporary data associated with the process. This includes, for instance, parameters of functions, local variables and return addresses.
- **Heap:** The heap is responsible for the dynamic allocation of memory. This type of memory is necessary for variables for which the compiler cannot determine the size before executing the program. Thus, these variables have to be stored using dynamic memory in the heap at run-time while executing the process.
- **Data:** The data section contains all the global and static variables that were initialized before executing the program.
- **Code:** The code section of memory is home to all the executable instructions of the program as well as constant variables. Since constant values and the program instructions don't change over time, this section is read-only and cannot be changed.

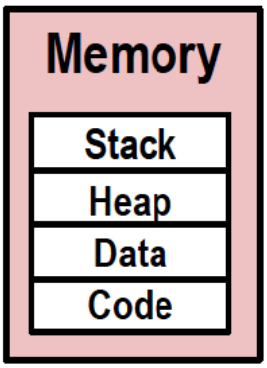

Figure 1: Process address space in memory (from [20])

Throughout the life cycle of a process, it goes through different states. There are **5** different states in an operating system. Each process can only have one state at a time following the diagram in Figure 2. These are the 5 states:

- **Start:** Initial state
- **Ready:** Process is waiting to be assigned a processor by the operating system in order to be executed.
- **Running:** Process is assigned a processor and can execute its instructions.
- **Waiting:** Process is waiting for a resource, for instance, user input or document which is being used by another process.
- **Terminated:** Process finished executing and is removed from the main memory.

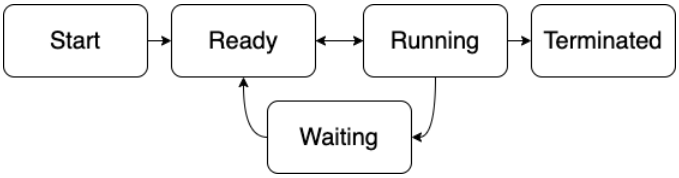

Figure 2: Process states diagram

Threads
Threads are small sets of instructions that live inside of a parent process [9]. A single process can create and terminate multiple threads in order to fulfill a specific job. Threads inside a process all share the same address space, which means that thread creation is much faster compared to processes because no new address space has to be created. Although threads do have their own stack for local variables, the data is still accessible by all other threads because of the shared address space inside of a process. Similar to processes, threads have their own thread identification ID called **TID** and have the same purpose as processes in regards to parallel execution. This means that since processes can be executed in parallel by the OS, threads can also be used for the same purpose, with the advantage that they are a lot faster to generate. The benefits of using threads will be presented in the sections about **Multithreading** and **Hyper-threading**. Finally, there are 2 different types of threads: **user-level threads** and **kernel threads**, but we won't go into detail about the difference.

Single-core vs. Multi-core processors
As already briefly introduced in the section 4.3.3, there are single and multi-core processors. In its most basic meaning, a CPU core receives and executes instructions. Single-core processors can only perform one task at a time because it only has *one* core. To compensate for the fact that only one task is executed at the same time, we can execute the tasks *faster*. To do this, we need to increase the CPU clock speed, which increases power consumption and heat distribution, resulting in an inefficient CPU. Introducing multi-core processors keeps the clock speed efficient while executing at similar speeds and consuming less energy [10]. In fact, multi-core processors are just single-core processors with more cores on the same CPU chip [11].

Multithreading vs. Multiprocessing
Multithreading refers to the concurrent execution of several threads on single and multicore systems as well as multiprocessor systems. The goal is to split the workload of an application into multiple threads and potentially run them on different CPU cores, thereby decreasing the overall execution time of the application. As explained in the section 4.3.3, there are various synchronization problems when running multiple threads at the same time. Since multi-core systems can run multiple threads in parallel, both threads can access the same critical section of code at the same time. This could

be prevented if the execution order of threads in a process could be specified by the programmer, but its the scheduling algorithm of the OS which is responsible for determining the execution order. One thing that programmers can influence is the thread priority by which they are executed. Programmers might assume that higher priority threads execute first, and therefore no synchronization mechanism is required between a high priority thread and a lower priority thread. While this may be the case for single-core systems (since the threads are executed sequentially and threads with high priority are selected first by the scheduling algorithm), this is not the case with multi-core systems [12]. This is because both the high and low priority threads can run in parallel with two available cores. Hence the necessity of the synchronization mechanisms introduced in the section 4.3.4.

Instead of multithreading, it is also possible to execute multiple processes on multiple CPUs. Running multiple processes concurrently does not require synchronization mechanisms because each process has its own address space. However, process address space is much larger compared to threads, as threads all share the same common address space of the parent process. Replacing the multithreading mechanism by a multiprocessing mechanism would fill the memory of most computers in no time due to the large address spaces. We can say that (in general) using synchronization mechanisms in multithreading is more practical than using multiprocessing systems.

Hypter-threading

Hyper-Threading (HT) is a hardware technology invented by Intel that is found in various Intel CPU chips. The purpose is to recognize each physical core in the CPU as two *logical cores* [21]. As we can see from this report, more cores = more threads running in parallel = more work in less time. This means that, for example, in a quad-core Intel CPU with HT technology, the operating system recognizes 8 cores. This technology enables each CPU core to do two things at the same time, increasing the overall performance of the computer and the number of possible background tasks [12].

4.3.3. Concepts related to concurrency. Let's start with the difference between concurrency and parallelism since both concepts are very similar and therefore not difficult to confuse. Let's assume we have three processes but only one core available:

- **Concurrency:** Programs are executed sequentially on single-core processors. This means that any running process is given limited CPU time to execute its commands. Assuming three processes for a single available core, each process would have to run one after the other. Rather than fully executing each process by giving them the CPU time they need, each process is given a very short CPU time to run. The CPU then rapidly switches back and forth between each process, creating the illusion of executing several processes at the same time [13]. This behavior is known as the concurrent execution of processes and is illustrated in Figure 3.

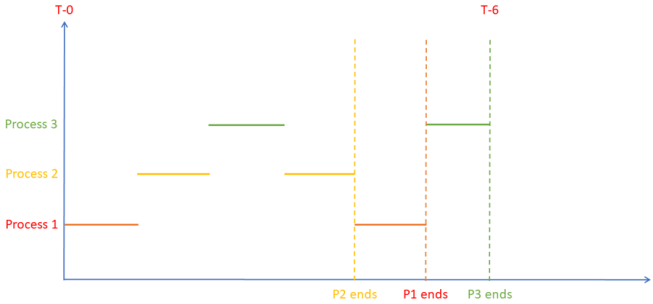

Figure 3: Example of concurrent execution of 3 processes taking 6 time units to finish (from [22])

- **Parallelism:** Parallelism cannot be achieved in single-core processors (not assuming Hyper-threading or similar technologies). Instead, let's assume we have a multi-core processor with three available cores. Each of our processes can *really* be executed in parallel without creating an illusion as each process can run on an available core at the same time. In fact, real word computers combine both parallelism and concurrency. An example can be seen in Figure 4.

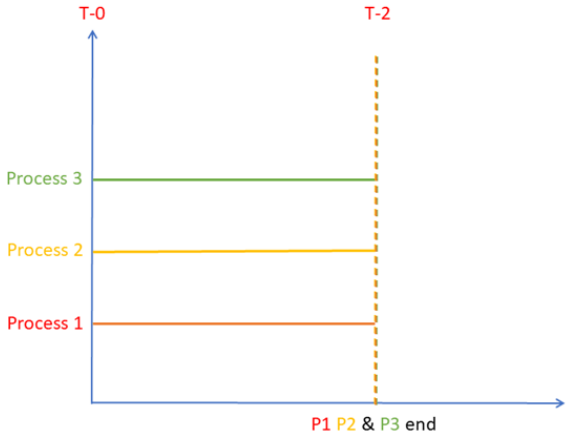

Figure 4: Example of parallel execution of 3 processes taking only 2 time units to finish (from [22])

Other concepts related to this field are possible problems that can appear in concurrent or parallelized applications. Let's briefly introduce some possible problems in concurrent applications [14]:

- **Race condition:** A race condition occurs when several tasks (e.g. threads) access shared memory at the same time without using a control mechanism. In this scenario, the output is unpredictable and depends on the order in which each instruction is executed.
- **Deadlock:** A deadlock occurs when at least 2 (or more) tasks are waiting for a shared resource to be freed. The problem here is that this exact resource is being blocked by a third task which is also waiting for a

shared resource to be freed by either one of the 2 other tasks. This creates some kind of vicious circle and the program cannot continue without someone/something intervening.

- **Livelock:** A livelock occurs when two tasks block the resources that the other requires. To avoid this state, the tasks release their blocked resource, but the cycle then starts again and ultimately never ends.
- **Resource starvation:** Resource starvation occurs when a task never gets the resource it is waiting for. When poor algorithms are used to distribute the resources and there is a case where more than one task is waiting for a particular resource, certain tasks may have to wait a very long time before they can continue to execute.

4.3.4. Concurrency control and why it is important. One way to prevent the problems introduced in section 4.3.3 from occurring is by using concurrency control mechanisms. Concurrency control was introduced to ensure the correctness of shared data [15]. When the same data is accessed by multiple transactions simultaneously, for instance in a database, data correctness is one of the most important things to preserve because otherwise, it can lead to data loss, inconsistency, and the storage of wrong data. In the context of process synchronization, we can distinguish between a Mutex (locking mechanism) and a Semaphore (signaling mechanism) in order to manage and protect access to shared resources.

- **Mutex:** A Mutex is a locking mechanism that ensures that only one task (either process or thread) can access a given resource at the same time. A mutex can only be released by its owner, so we can say that the mutex is *owned* by a particular task.
- **Semaphore:** A Semaphore is a signaling mechanism that ensures that a fixed number of tasks (either process or thread) can access a given resource at the same time. There are two types of semaphores: **binary** and **counting** semaphores. Binary semaphore's value can only be 0 or 1 and are therefore used in a similar way to a Mutex: for resources that can only be used by one task at a time. Counting semaphore's value can be any integer *n*, where *n* represents the total number of tasks that can access the critical resources at the same time. Semaphores have two different use cases, mutual exclusion (similar to Mutex) and condition synchronization (limit the amount of tasks that can access a given resource at the same time) [16].

There are several well-known computer science problems that illustrate these concepts. Two of them are the Sleeping Barber Problem [23] and the Cigarette Smokers Problem [24].

4.3.5. Why should one write parallelizable code?. Writing parallelizable code can significantly reduce the execution time of a process. Two parallelizable pieces of code are not dependant on one another and can therefore be run in parallel because they may be running different services of an application. The order of execution of both codes is not crucial for the app and neither one depends on the result of the other. An example of a non-parallelizable code is the registration and authentication of a user, as one cannot authenticate the user before it is registered in the app nor do both in parallel. In theory, the time taken for a program to run (T) is the number of instructions (I) multiplied by the average time needed to complete each instruction (t_{av}) [17].

$$T = I \times t_{av} \tag{1}$$

Since in multi-processor systems, one can execute a program in parallel, the time taken should be divided by the number of available processors (N).

$$T_p = \frac{T}{N} \tag{2}$$

In reality, however, this is wrong as most programs cannot run 100% in parallel and still require some sequential work. It is for this reason that the Amdahl's Law is required.

Amdahl's Law

Amdahl's Law is used to predict the theoretical speed-up (SU) when using multiple processors to execute the same program in the context of parallel computing. Every program has a serial and a parallel part. The greater the parallel part is, the faster a program can be executed. Since the serial (S) and parallel (P) part are indicated as a percentage, one can define the following:

$$S + P = 1 \Longrightarrow S = 1 - P \tag{3}$$

When using multiple processors (N), the Amdahl's Law can be defined as the following equation:

$$\text{SU} = \frac{1}{S + (\frac{P}{N})} \tag{4}$$

Table 1 indicates how many times a program can execute faster according to a fixed number of available processors and the parallelizable part of a program (P). As one can see, a program that is only 25% parallelizable can only run 33% faster, no matter how many processors are at its disposal. This clearly indicates that increasing the amount of available processors does **not** increase the speed at which a program is executed. From the same table, it can be seen that the higher the parallelizable percentage is (P), the greater the increase in execution speed is according to Amdahl's Law.

N	P = 25%	P = 50%	P = 90%	P = 99%
10	1,29	1,82	5,26	9,17
100	1,33	1,98	9,17	50,25
1.000	1,33	1,99	9,91	90,99
10.000	1,33	1,99	9,99	99,02
100.000	1,33	1,99	9,99	99,90
1.000.000	1,33	1,99	9,99	99,99

Table 1: Increase in execution speed according to Amdahl's Law

4.4. Assessment

In order to assess the scientific deliverable of this Bachelor semester project, one should consider the different functional requirements set at the beginning:

- **FR1:** The deliverable introduced concurrency to the reader, thereby fulfilling the requirement.
- **FR2:** In addition, concurrency-related terms and technologies such as CPU, processes, threads, etc. were presented in this deliverable and the relationship between the various parts was explained. The requirement is thus met.
- **FR3:** Furthermore, the deliverable listed and explained various concepts related to concurrency such as the difference between concurrency and parallelism, and possible problems that can arise with concurrent applications. The requirement is thus met.
- **FR4:** Moreover, the process synchronization mechanisms were introduced and the importance of concurrency control was explained. The requirement is thus met.
- **FR5:** Finally, the deliverable introduces Amdahl's law and uses it to explain the importance of writing parallelizable code. Using a speed-up illustration table, the reader can see the performance benefits of writing parallel applications. This fulfils the last requirement of the scientific deliverable.

One can conclude that this deliverable is a success since all requirements were successful.

5. Prototypical Implementation of an iPad App for Drone Missions

5.1. Requirements

For the drone mission iPad app, several functional requirements have been set. These functional requirements summarize the functionalities that shall be achieved by the application by the end of the project.

- **FR1 - Login & Register:** The first requirement is that the app shall provide a login and register system. The user should be able to create his account directly on the iPad itself and be able to log into his account at any time.
- **FR2 - Mission planning:** The second requirement is that the app shall provide the full mission planning view from the given design prototype [6]. The user should be able to plan a mission and be able to save and load it afterwards. This requirement includes the ability to set an area on the map, select the vegetation indices and activities to perform and be able to select the desired drone and camera to be used during the mission.
- **FR3 - Mission history:** The third requirement is that the app shall display data from previously performed missions in a separate tab. The user should be able to analyze and view the results directly from the app.

Furthermore, the app also has one non-functional requirement which is reusability.

- **NFR1 - Reusability:** Many of the views in this app look and work alike. Reusability in this context refers to the ability to reuse the same view multiple times in different places.

5.2. Design

This section is divided into **four** different parts: the application design pattern and three parts, each corresponding to a functional requirement of the technical deliverable. Each section containing a functional requirement is further separated into frontend and backend design.

5.2.1. Application design pattern. The design pattern followed in this application is the **Model-View-ViewModel** (MVVM) architecture. Utilizing a well-known design pattern comes with several benefits, such as an increase in the maintainability of the software. MVVM has also been among the favorites of iOS developers in recent years [25]. It aims to separate the views, i.e. graphical design, from the backend logic.

Model

In MVVM, models are defined to structure the data required in the app. It provides the abstraction of the business objects encapsulating the data required in the application. For this application, we can define **nine** different models. The relationships and multiplicities between the different models can be observed in Figure 5. The models are:

1) User: defines the user using the app
2) Mission: defines the mission being planned
3) Location: defines the locations of a mission
4) Drone: defines a drone in the app
5) Camera: defines a camera in the app
6) Index: defines an index in the app
7) Activity: defines an activity in the app
8) GeoPoint: defines the GPS coordinates of a mission location
9) MissionImage: defines the mission image to be analyzed

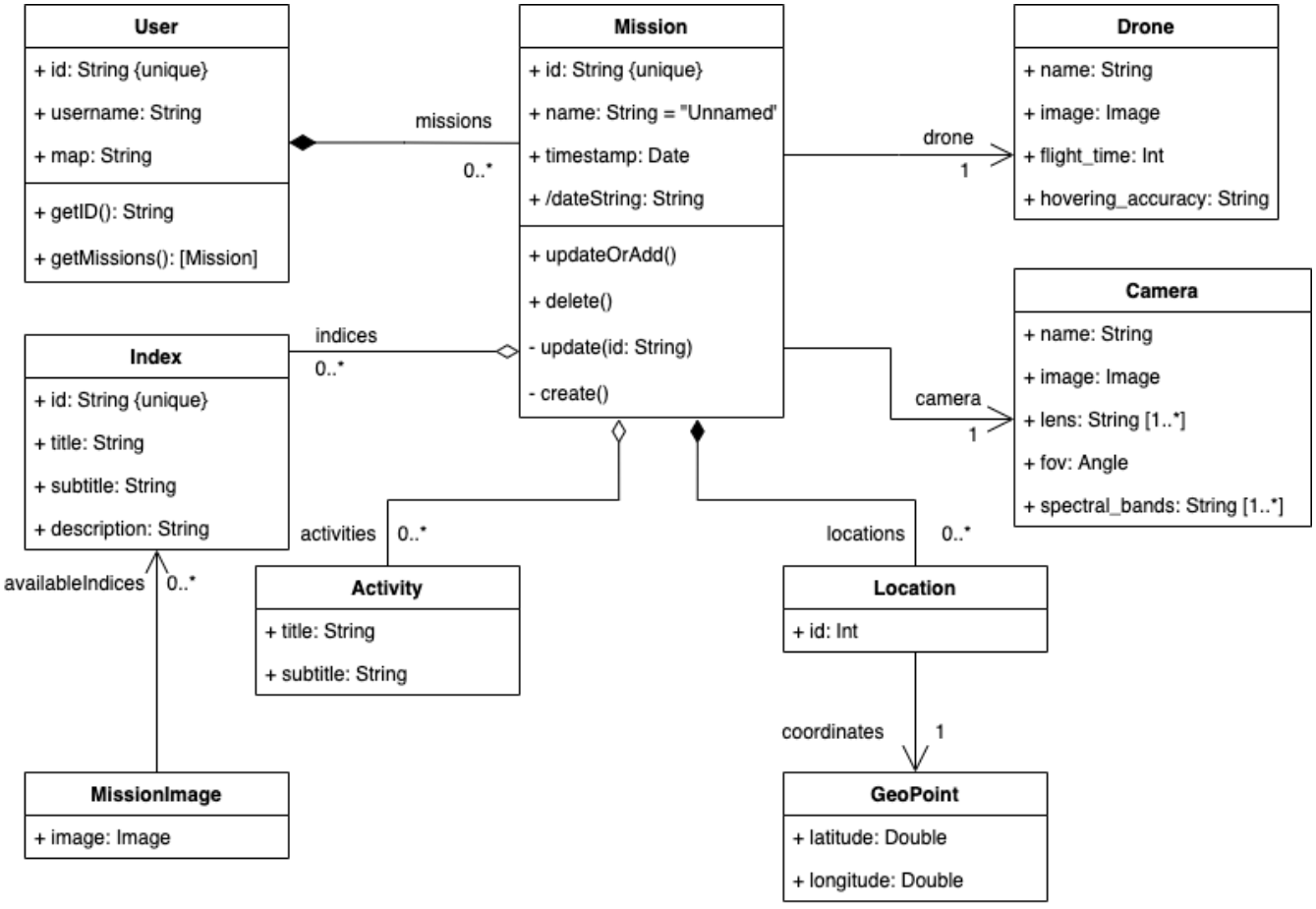

Figure 5: Design of the 9 models for the drone mission app for iPad

View
In MVVM, views contain the application's user interface (UI). They define the structure, layout, and appearance of what the user sees on the screen [26]. There are various views in this application mentioned in each of the functional requirement sections 5.2.2, 5.2.3 and 5.2.4. The design of the underlying view is shown in the frontend area of the corresponding section.

ViewModel
In MVVM, the view model has a two-way binding with the views. It holds properties to which the view can bind data to, but also contains state change listeners which can inform the view about changes [26]. Furthermore, the view model is responsible for populating and maintaining the models. Usually, there is a one-to-many relationship between the view model and the models as one view model can manage many models. It is also responsible for providing the views with the necessary app data defined in the models. Thus, view models can populate and manipulate the models, and keep views up-to-date. In this application, we have **three** main view models:

1) **LoginViewModel:** manages the login view
2) **RegisterViewModel:** manages the register view
3) **PlanningViewModel:** manages the planning views and the mission model instances

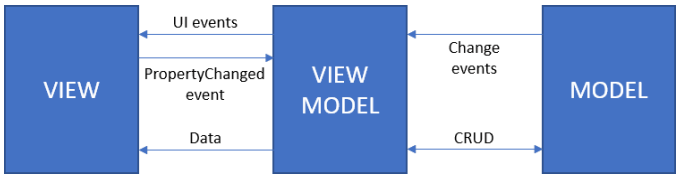

Figure 6: Model-View-ViewModel (MVVM) structure (from [27])

As we can see in Figure 6, each of the 3 components of this architecture have their own purpose and functionality.

Illustrative example
One can see the interaction between the model, view and view model using an illustrative example. On Figure 7 one can see an example of a user executing a UI event to create an annotation on the map. The view model then creates and saves the annotation by communicating with the Location-model, which creates an instance of the annotation on the map. The view model will then receive the created annotation, display it and update the map for the user.

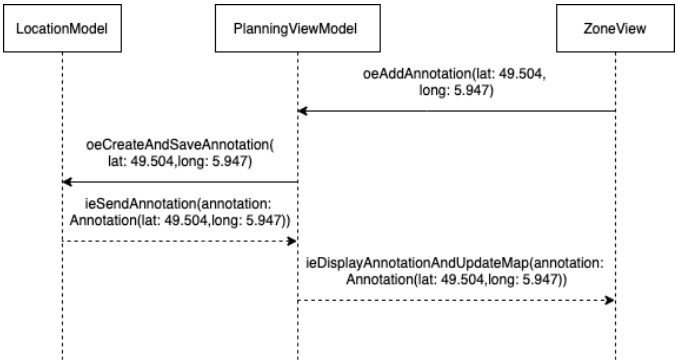

Figure 7: Use case instance: illustrative example

5.2.2. FR1 - Login & Register. The first functional requirement of the app is the login and register views. The corresponding frontend and backend designs are outlined below.

Frontend design (graphical)
The login and register views are designed in the same way: half of the screen is covered by a placeholder image from the DJI website and the other half contains the entry fields for the user. The login page contains an email and password field allowing users to enter their credentials to log in. Furthermore, the password field contains an icon allowing users to see their password instead of a masked version. As an additional feature, users can request a password reset if they forget their password. At the bottom of the page, users can find the *Sign-in* button along with a button asking if the user does not have an account, which redirects the user to the registration page when clicked. The login view can be seen in Figure 8. The register page contains text fields for the user name and email address. Similar to the login page, the register view contains a password field with the possibility of toggling between the masked and unmasked version. Finally, users can find the *Create account* button at the bottom of the page. The register view can be seen from Figure 9 to Figure 12.

Backend design
The login and register views both share a common session manager with the rest of the application. This session manager provides the necessary authentication functions, such as the *sign-in*, *create user* and *send password reset* to the login and register view. Nevertheless, both views are linked to a separate view model **LoginViewModel** and **RegisterViewModel**, respectively. This means that the data entered in the text fields of both views are kept separated from one another. Thus, both views are handled as independent views and do not share any information. Both view models only contain textual properties that are bound to the text fields in the view, no authentication handling methods are placed in the view models.

5.2.3. FR2 - Mission planning. The second functional requirement of the app is the mission planning views. The

corresponding frontend and backend designs are outlined below.

Frontend design (graphical)
The mission planning views are composed of **four** independent views: zone to scan view, monitoring view, drones and cameras view, and the summary view. These views allow users to plan, create and finally save the mission. To manage the saved missions, a button called *Manage missions* is placed in the navigation bar on the top right corner. This button lists all the saved missions of the user and gives the possibility to load or delete a specific mission. This behaviour can be seen from Figure 19 to Figure 22. Let's focus on the four main views below.

- **Zone to scan:** The zone to scan view displays a map stretching through the entire available space. Users can define the desired area by placing annotations on the map, each representing a GPS location. When the amount of annotations is at least three, a polygon is drawn on the map connecting all the annotations. The area of the polygon is calculated in square meters and shown in the bottom left corner. This view can be seen in Figure 13.
- **Monitoring view:** The monitoring view allows the user to select the indices they want to capture and the activities they want to have activated during the mission. The view consists of 2 boxes, one for the indices and one for the activities. Each box then contains the respective data in a list allowing the user to select the desired elements. This view can be seen in Figure 14.
- **Drones and cameras view:** The drones and cameras view is similar to the monitoring view. The difference is that the user can select the drone and camera to use during the mission. The list of drones and cameras is a fixed list with the available material at the University of Luxembourg. This view can be seen in Figure 15.
- **Summary view:** The summary view is once again made up of 2 boxes next to each other. In the left box, the user can see a smaller version of the map from the zone to scan view. This small map is enough to highlight the selected area such that the user can be sure everything is correct. In the right box, the user can see a summary of the selected indices, activities, and the selected drone and camera for the mission. Ultimately, the user can find the save and start the mission button at the bottom of the box. This view can be seen from Figure 16 to Figure 18.

To navigate between the different views, the user can use the navigation bar at the top of the screen or the *Next step* button on the bottom right corner of each view.

Backend design
The mission planning views share the same instance of the **PlanningViewModel**. This means that since the PlanningViewModel contains a Mission class instance called *currentMission*, the four views are bound to this *currentMission*-property, allowing them to read and update the mission accordingly. The zone to scan view can update the locations and coordinates related to the zone, the monitoring view can update the indices and activities of the mission, and so on. The PlanningViewModel also contains a *loadMission* method which can load a mission into the app. Furthermore, the view model keeps track of the currently displayed tab of the planning views.

5.2.4. FR3 - Mission history. The third functional requirement of the app is the mission history views. The corresponding frontend and backend designs are outlined below.

Frontend design (graphical)
The view of the mission history has been made very simple. A box with the relevant information is displayed for each mission carried out. First, we have the header in each box. The header contains the mission name on the left and the mission location and time on the right. Then we have two horizontal scrollers that contain the images for the orthomosaic and the activity images. When the user clicks on an image, a new popup will open containing the image itself on the left and a new box on the right with various indices that can be applied to the image. The user can then select the desired index and save the image. The mission history view can be seen in Figure 23 and Figure 24.

Backend design
The mission history view accesses the missions of the current user instance and creates a new mission history line for each mission. Before displaying the missions, the view checks whether the missions have already been loaded from Firebase and displays a temporary text *Loading missions ...* if this is not the case. It is assumed for the mission location that the selected area is so small that each annotation is in the same district. Therefore, in order to find the mission location, the first annotation GPS coordinates are used to determine the location using the reverse geocode location.

5.3. Production

5.3.1. FR1 - Login & Register. The first functional requirement of the app is the login and register views. The corresponding frontend and backend productions are outlined below.

Frontend production (graphical)
The main element behind the login and register views is the SwiftUI GeometryReader. The GeometryReader is defined by Apple as *a container view that defines its content as a function of its own size and coordinate space* [28]. In other words, we can define element dimensions relative to the available screen size. Thus, we used GeometryReader to divide the screen in half and place a placeholder image from DJI on the right-hand side of the screen. The content of the remaining left-hand side of the screen was then fully centered in the available space. Here, the email and password fields were created using the TextField and

SecureField elements respectively. As an additional feature to the password field, a button was added which toggles between the TextField and SecureField elements, making the password visible when desired by the user. Finally, the *Sign in* button was created using the systemBlue background color. Clicking on this button will remove the focus from whatever input field is currently focussed and send a user authentication request to Firebase. The register page is the exact same thing as the login page, except for the possible inputs. On the register page, the user must input his full name, email address, and password in order to create an account. The font used in both login and register views, as well as the entirety of the application, was the **SF-Pro** font created by Apple.

Backend production
Both the login and the register view are linked to their respective ViewModels LoginViewModel and RegisterViewModel. Each ViewModel contains the corresponding TextField and SecureField data using **@Published** attributes so that the views can read and update the underlying values. Both views share a session manager instance with the rest of the application, which contains the authentication functions that are used when the user tries to log in, register, or recover their password. These functions are therefore the functions *login*, *register* and *passwordReset*, which communicate with Firebase when executed. Depending on the result of the functions, the views are updated accordingly by displaying pop-ups with error messages, or the user is simply logged in without feedback if everything is OK.

5.3.2. FR2 - Mission planning. The second functional requirement of the app is the mission planning views. The corresponding frontend and backend productions are outlined below.

Frontend production (graphical)
The NavigationBar component is placed on the top part of the screen and its title is set to *Planning: missionName*. On the right-hand side of the NavigationBar, a NavigationBarItem which is a button called *Manage missions* was added. By clicking on this button, a custom sheet is opened where all the missions stored in the user model are listed using a *List* element combined with a ForEach. The user can load a mission by clicking on the row or delete a mission by swiping to the left, thus the standard Apple behaviour on lists. Each of the four main views was created using a VStack containing the PlanningViewHeader (HStack of 4 buttons corresponding to the 4 tabs) and the view itself below. Let's focus on these views below.

- **Zone to scan:** This view is a ZStack containing a MapView component stretching through the available height and width. On top of the map, a small text view was placed on the bottom left corner indicating the area in square meters of the selected area on the map as well as a *Next step*-button on the bottom right corner.
- **Monitoring view:** This view is a 2 column HStack. Each column represents a box that contains the indices or activities. For both boxes, we go through all indexes and activities with a ForEach and create buttons that contain the name and description as well as an indicator that shows whether the respective index or activity is selected or not. A click on the button selects or deselects the element.
- **Drones and cameras view:** Similar to the monitoring view, this is a 2-column view showing all available drones and cameras. For each drone and camera, the respective image is also embedded on the left side of the button which allows the item to be selected or deselected by placing all of the information in another HStack.
- **Summary view:** This view is again a 2-column HStack view with 2 boxes. The left panel shows a summary of the selected area using the MapView component with all the annotations. This instance of the MapView component is informational only and the zoom, rotate and drag functions have been disabled. In the right box, a VStack contains a list of all selected elements of the mission, separated by a divider. Finally, at the bottom of the page, the user will find the buttons *Save* and *Start the mission*.

Backend production
The four mission planning views share the same view model which contains the necessary data to handle the mission planning. The view model has a **@Published** property named *currentMission* containing an instance of type Mission, which corresponds to the current mission. Since the four views are bound to this property, all can update and modify the content of the current mission, which allows for the simple updating, addition, and deletion of missions through Firebase. By loading a mission, the *currentMission* property is replaced by a new instance of type Mission containing all the data loaded from Firebase. Moreover, the PlanningViewModel keeps track of the current tab with a **@Published** property called *currentTab* accessible by all the views. Since every view has access to the information stored in the view model, the *Zone to scan*-view can pass the mission location points of type **GeoPoint** to the MapView, creating a custom annotation at the exact same GPS location. Finally, by handling long-press gestures on the MapView, the user can create an annotation and add the GPS location to the current mission by creating a **GeoPoint** instance containing the corresponding latitude and longitude and appending it to the mission locations.

5.3.3. FR3 - Mission history. The third functional requirement of the app is the mission history views. The corresponding frontend and backend productions are outlined below.

Frontend production (graphical)
The main element of the mission history view is a simple ScrollView. Then, inside a VStack, we iterate over all the user missions and create a dedicated box for every mission. Every box starts with a HStack, containing the mission name

on the left, and using a Spacer() element we can place the time and location on the right. Then, for the orthomosaic and activities images, a horizontal ScrollView was added with some placeholder images. By clicking on an image, a new popup is opened which blurs the background. The image which was selected is placed on the left-hand side of the popup, while the box with the available indices is placed on the right-hand side. This box was created using the same techniques as the monitoring view boxes.

Backend production
To get the user missions, the *getMissions()*-method from the current User class is used. This method returns a list of the missions from the user which is used in the ForEach of the missions history view. When the user clicks on an image, a new instance of **MissionImage** is created, which contains the image itself and the available indices that were selected during the planning of the mission. This MissionImage instance is used to create the popup with the image on the left-hand side and the indices on the right-hand side.

5.4. Assessment

In order to assess the technical deliverable of this Bachelor semester project, one should consider the different functional and non-functional requirements set at the beginning:

- **FR1:** The final version of the application presented in this deliverable includes a fully functional login and register system. Thus, the requirement is met.
- **FR2:** In addition, the application offers several views that allow a mission to be planned. One can customize the entire mission by selecting the desired area on the map, select the indices and acitivies to perform as well as the drone and camera to be used. The requirement is thus met.
- **FR3:** Finally, the last functional requirement was also met, as the previously performed missions can be viewed and analyzed directly in the app.
- **NFR1:** The non-functional requirement **reusability** was met in the application, as several views were broken down into smaller components. Reducing the size of each view increases reusability as the view can be reused in various other places in the app. This results in a lower total amount of code and time required to build the app.

One can conclude that this deliverable is a success since all requirements were successful.

Acknowledgment

I would like to thank my Tutor, Benoît Ries, for the help and support during the entire semester. His advice and encouragement were fundamental to the success of the entire project.

6. Conclusion

This paper presents a Bachelor semester project whose goal is the prototypical implementation of an iPad app for drone missions and presenting concurrency in software applications. The scientific part focuses on the presentation of concurrency in software applications, which includes, among other things, the concepts, technologies, and methods used to create concurrent applications, while the technical part is the implementation of the iPad app *Aerial Systems*. All the requirements for the scientific and technical deliverable of this project have been met, making this project a successful one.

Finally, this project enabled me to deepen my knowledge of computer infrastructures by researching on different hardware components that reside in a computer. Additionally, this project gave me a better understanding of the concepts and importance of concurrency. While doing the technical deliverable, I was also able to further improve my iOS development skills in Swift and SwiftUI by facing the creation of more sophisticated UI views in the app.

7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a precheck by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission,

with or without compensation) in one's own name ('ghost-writing')
8) Using another person's unpublished work without attribution and permission ('stealing')
9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Backlinko, "How many people use Facebook in 2021?" Sep 2020. [Online]. Available: https://backlinko.com/facebook-users

[2] F. De Jesus Matias, "Goodness Groceries - Symmetric key encryption and iOS security measures," June 2021, bachelor Semester Project.

[3] (2021) SwiftUI. [Online]. Available: https://developer.apple.com/xcode/swiftui/

[4] (2020) DJI Mobile SDK. [Online]. Available: https://developer.dji.com/mobile-sdk/

[5] (2021) Firebase. [Online]. Available: https://firebase.google.com/

[6] T. De Jesus, "Design and Usability Study of a Graphical User Interface for Resiliency of Ecosystems Monitoring," June 2019, bachelor Semester Project.

[7] A. Polyvyanyy and C. Bussler, *The Structured Phase of Concurrency*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 257–263. [Online]. Available: https://doi.org/10.1007/978-3-642-36926-1_20

[8] J. B. Pedersen and P. H. Welch, "The symbiosis of concurrency and verification: teaching and case studies," *Formal Aspects of Computing*, vol. 30, no. 2, pp. 239–277, Mar 2018. [Online]. Available: https://doi.org/10.1007/s00165-017-0447-x

[9] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2013.

[10] B. Schauer, "Multicore Processors - A Necessity," Sep 2008.

[11] B. Venu, "Multicore Processors - An overview," Oct 2011. [Online]. Available: https://arxiv.org/pdf/1110.3535.pdf

[12] J. Roberts and S. Akhter, *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Richard Bowles, Apr 2006.

[13] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen, *Introduction To Concurrency In Programming Languages*. Crc Press, 2009.

[14] J. Fernandez Gonzalez, *Mastering Concurrency Programming with Java 9*. Packt Publishing Ltd., 2017.

[15] A. Thomasian, "Concurrency Control: Methods, Performance, and Analysis," *ACM Computing Surveys*, vol. 30, no. 1, March 1998.

[16] R. Agarwal and S. D. Stoller, "Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables," July 2006.

[17] A. Kiessling, *An Introduction to Parallel Programming with OpenMP*, Apr 2009. [Online]. Available: https://www.roe.ac.uk/ifa/postgrad/pedagogy/2009_kiessling.pdf

[18] Lamport, Leslie, "Turing Lecture: The Computer Science of Concurrency: The Early Years," Jun 2015. [Online]. Available: https://cacm.acm.org/magazines/2015/6/187316-turing-lecture-the-computer-science-of-concurrency/fulltext#R5

[19] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Commun. ACM*, vol. 8, no. 9, p. 569, sep 1965. [Online]. Available: https://doi.org/10.1145/365559.365617

[20] N. Navet, *Computing Infrastructure 1: Lecture 6*, 2020.

[21] Intel, "What Is Hyper-Threading?" 2021. [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

[22] (2021, Oct) Illustration of concurrent and parallel execution of 3 processes. [Online]. Available: https://www.guru99.com/cpu-core-multicore-thread.html

[23] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "Combining Static Analysis and Testing for Deadlock Detection," pp. 4–6.

[24] D. L. Parnas, "On a solution to the cigarette smokers' problem," 1972.

[25] "Understanding The Most Popular iOS Design Patterns in Swift," May 2019. [Online]. Available: https://iosapptemplates.com/blog/mobile-app-development/ios-design-patterns-swift

[26] D. Britch, "The Model-View-ViewModel Pattern," Jul 2021. [Online]. Available: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[27] "Gunnar Peipman - Programming Blog," Mar 2019. [Online]. Available: https://gunnarpeipman.com/inotifypropertychanged/

[28] Apple, "GeometryReader - Apple Developer Documentation," 2021. [Online]. Available: https://developer.apple.com/documentation/swiftui/geometryreader

8. Appendix

8.1. Application source code

The entire application source code for this project can be found in this Github repository: **https://github.com/Flavio8699/Aerial-Systems**.

8.2. Application screenshots

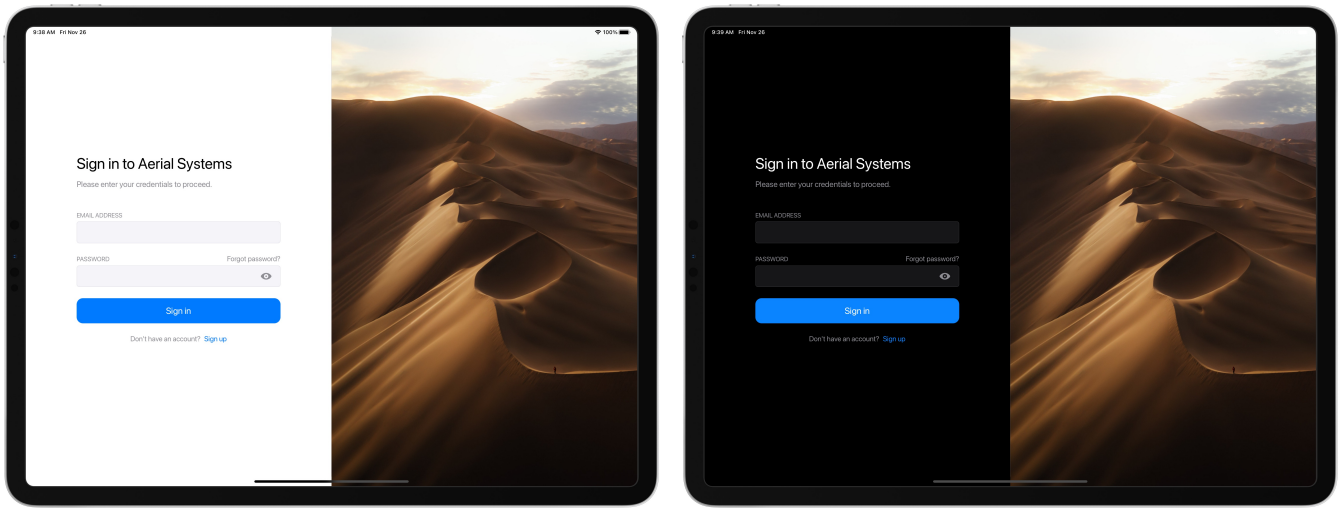

Figure 8: Graphical user interface - iPad App for Drone Missions - View 1

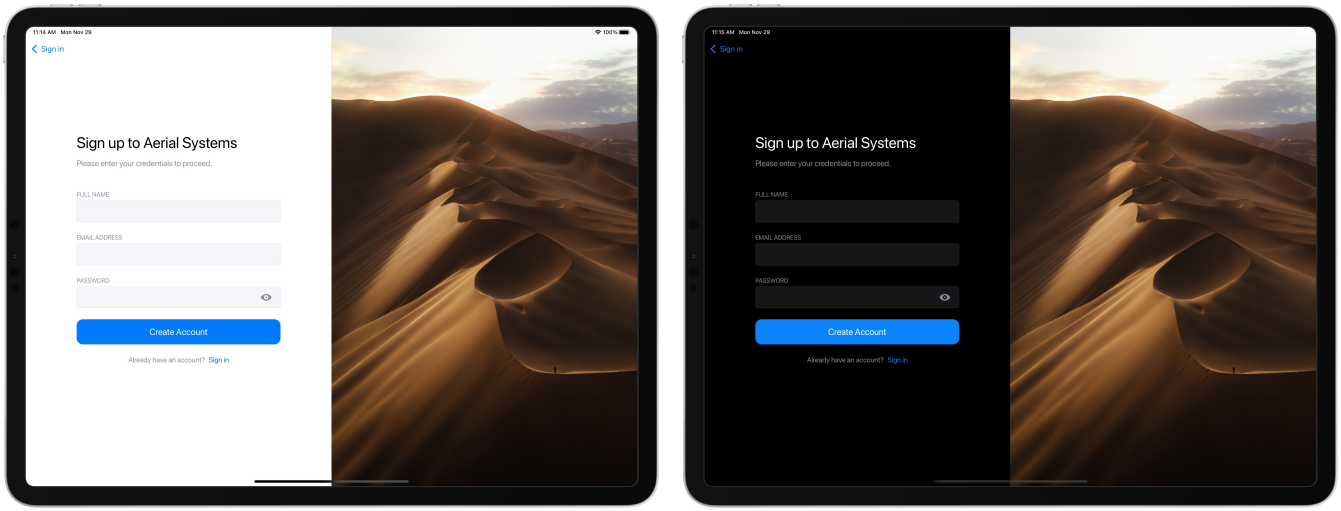

Figure 9: Graphical user interface - iPad App for Drone Missions - View 2

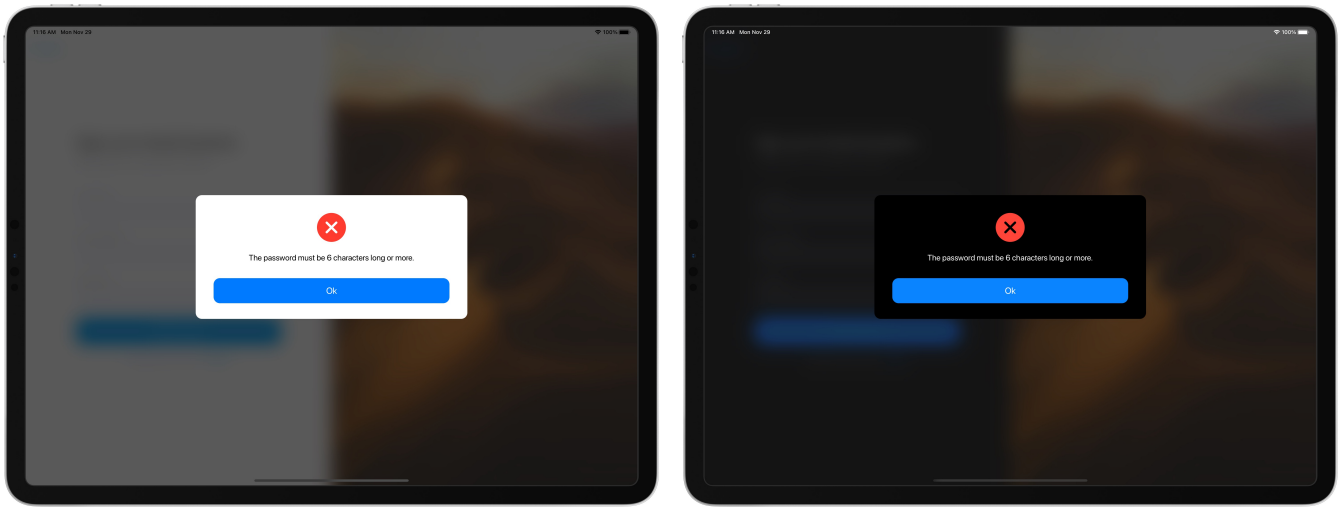

Figure 10: Graphical user interface - iPad App for Drone Missions - View 3

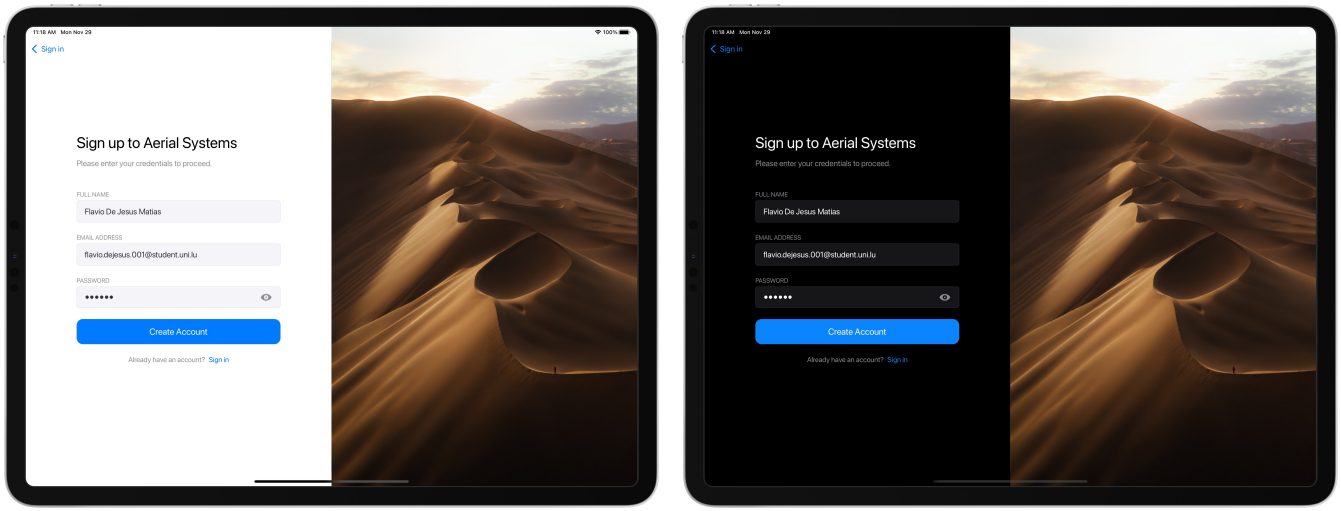

Figure 11: Graphical user interface - iPad App for Drone Missions - View 4

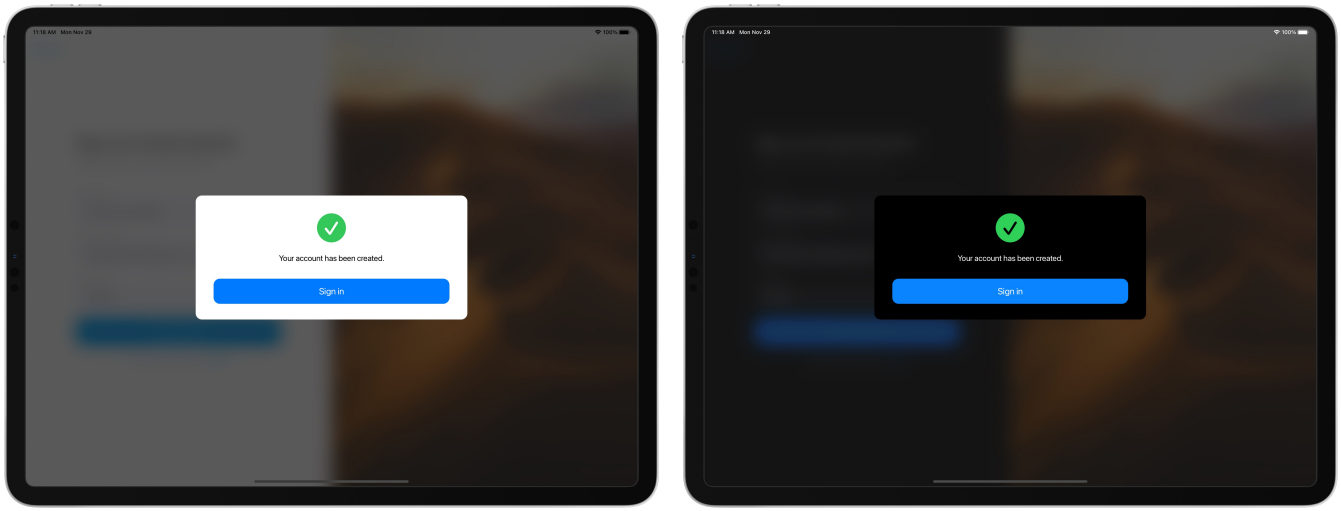

Figure 12: Graphical user interface - iPad App for Drone Missions - View 5

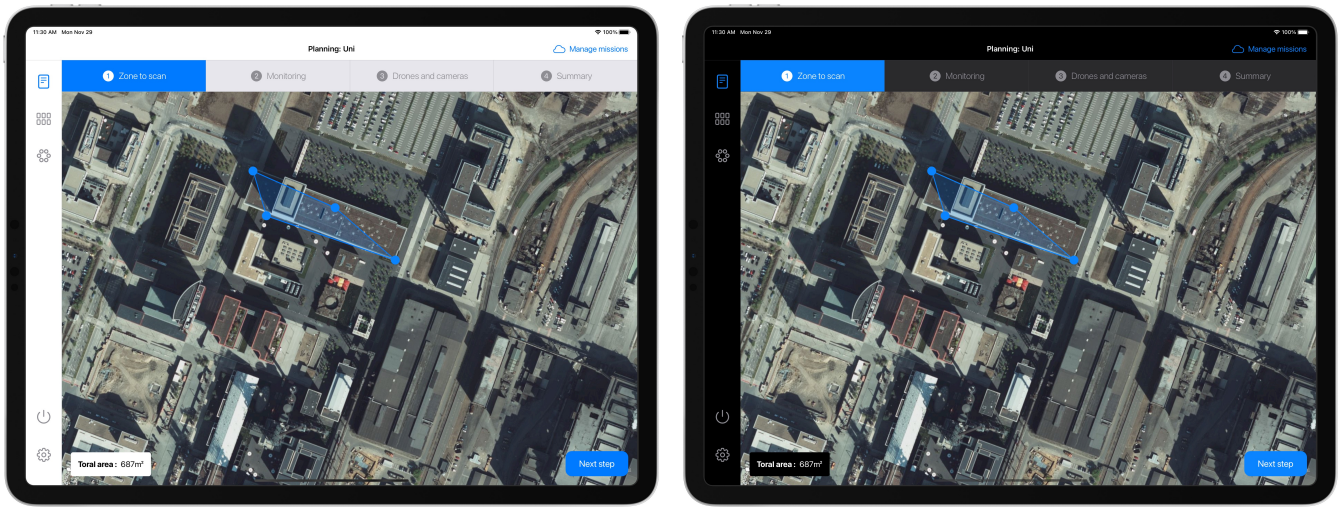

Figure 13: Graphical user interface - iPad App for Drone Missions - View 6

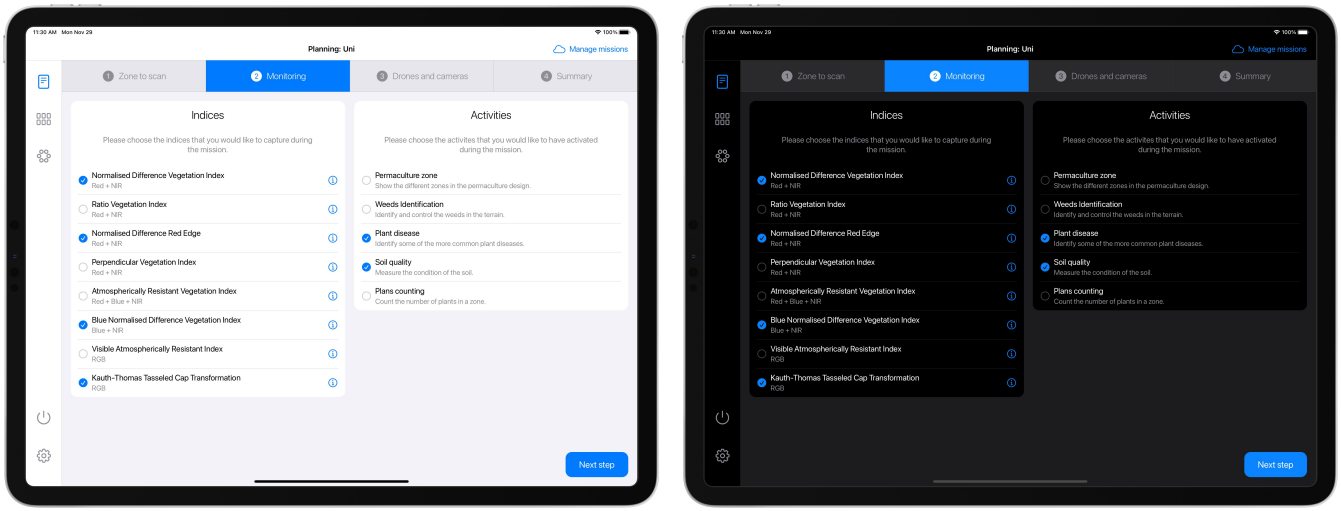

Figure 14: Graphical user interface - iPad App for Drone Missions - View 7

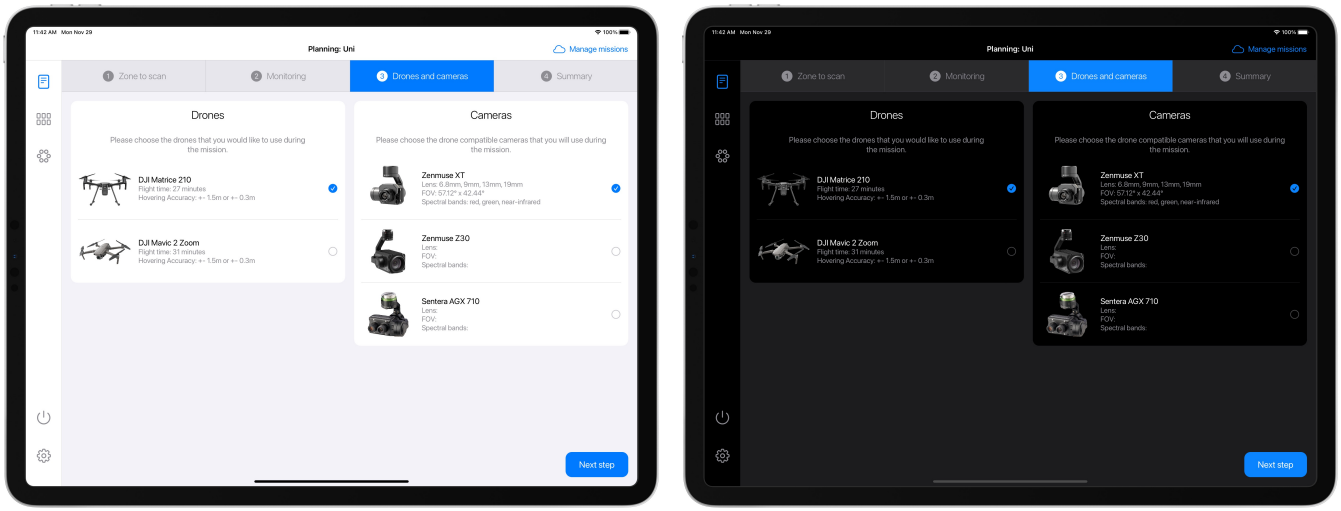

Figure 15: Graphical user interface - iPad App for Drone Missions - View 8

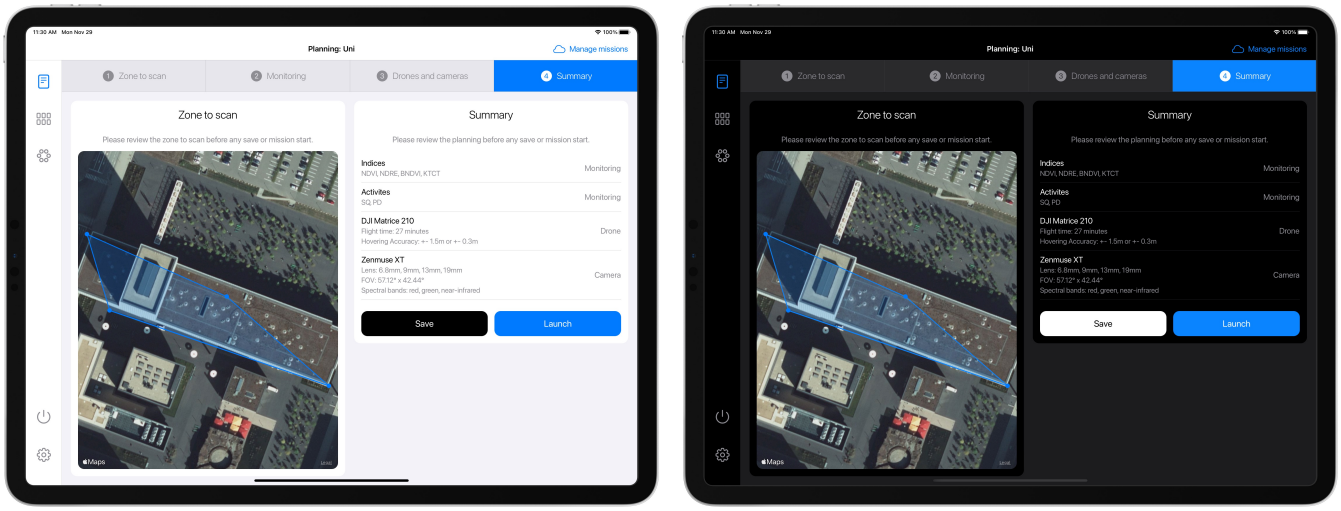

Figure 16: Graphical user interface - iPad App for Drone Missions - View 9

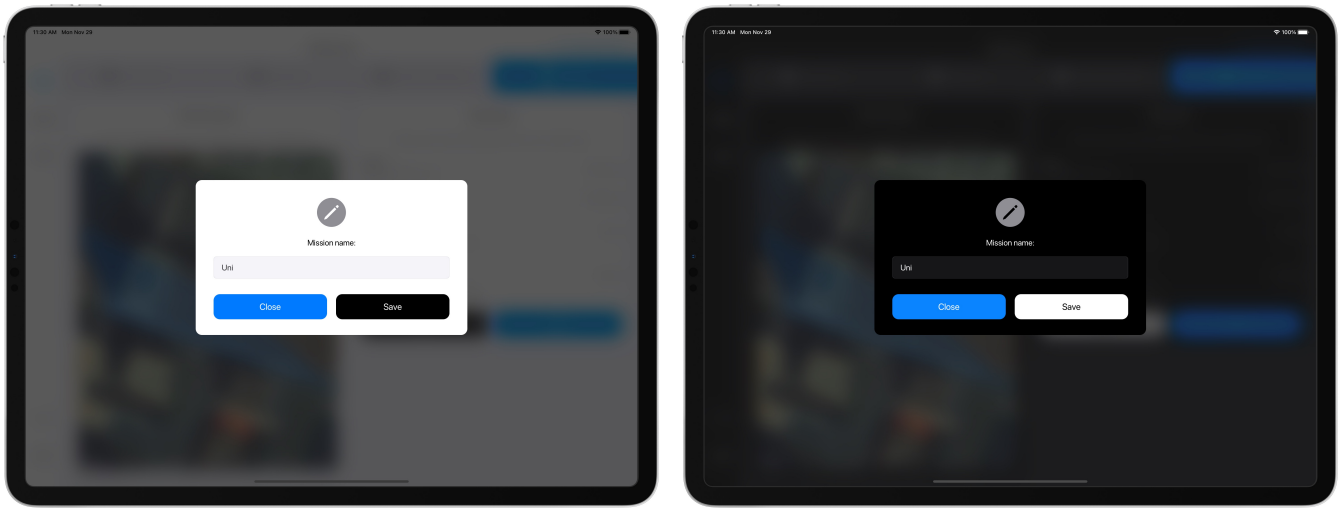

Figure 17: Graphical user interface - iPad App for Drone Missions - View 10

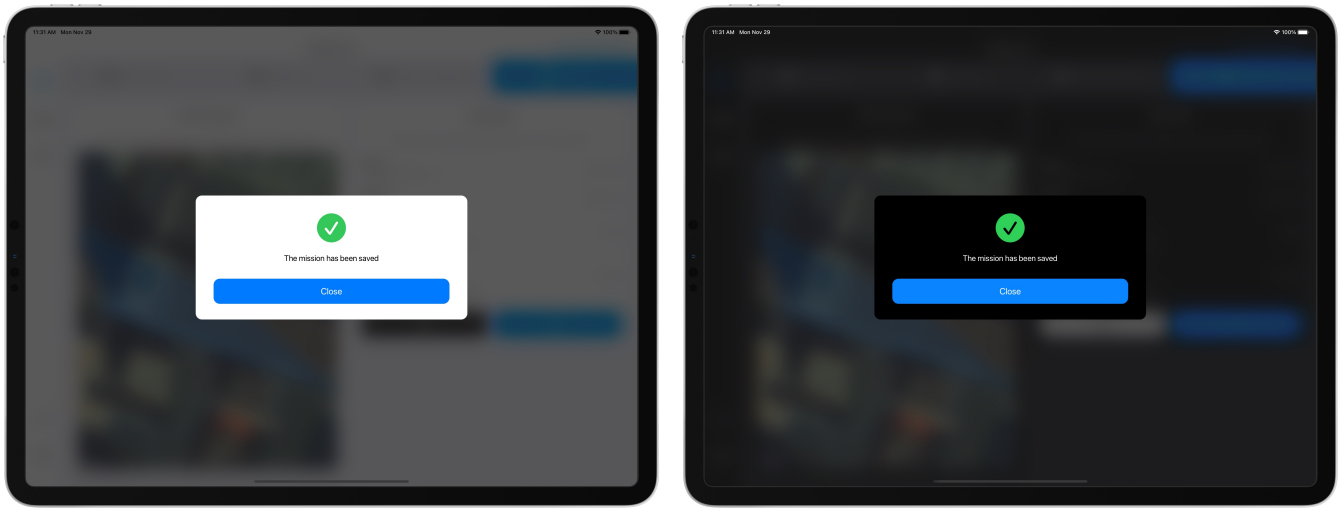

Figure 18: Graphical user interface - iPad App for Drone Missions - View 11

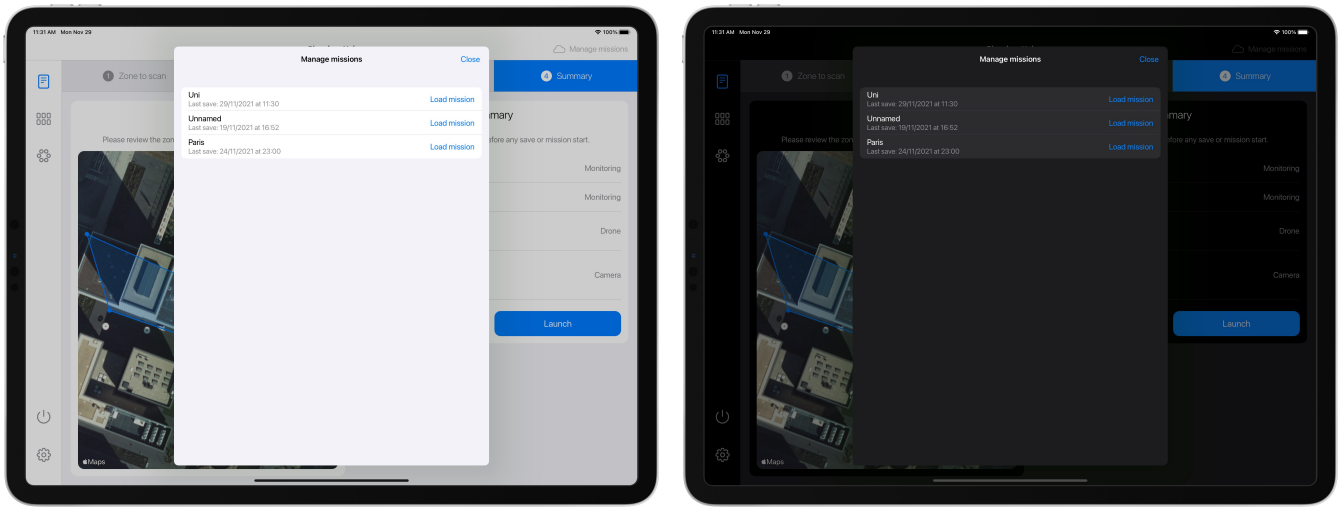

Figure 19: Graphical user interface - iPad App for Drone Missions - View 12

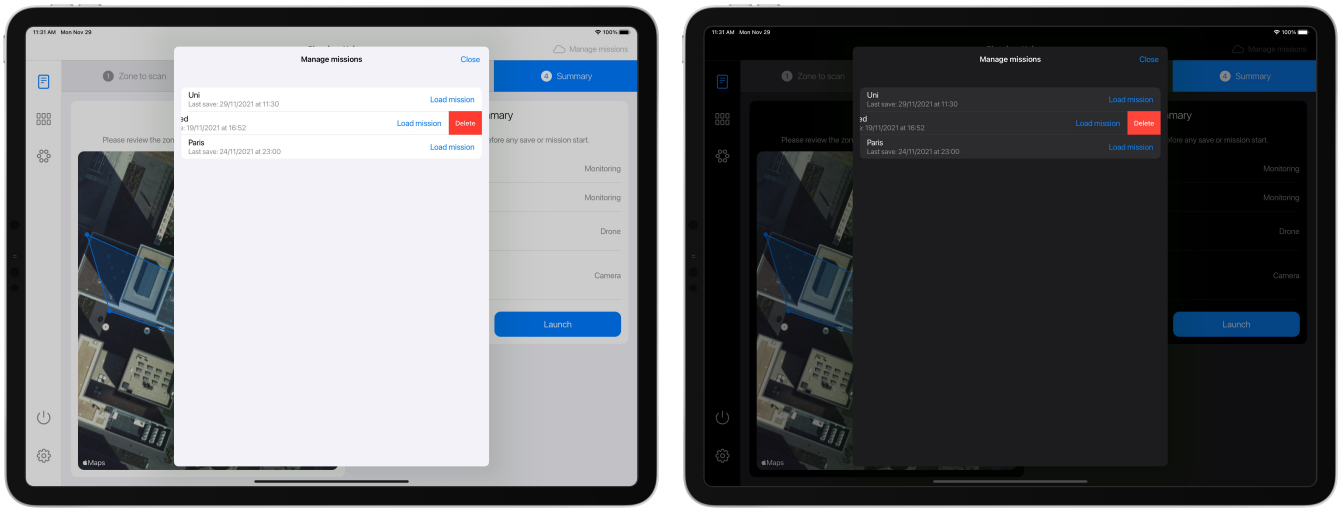

Figure 20: Graphical user interface - iPad App for Drone Missions - View 13

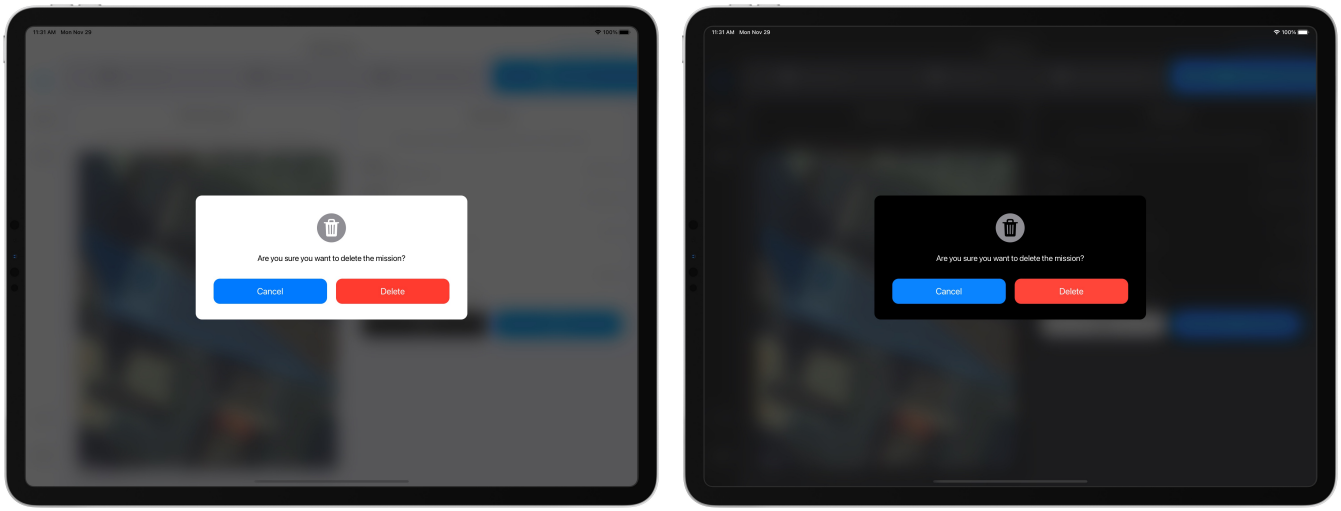

Figure 21: Graphical user interface - iPad App for Drone Missions - View 14

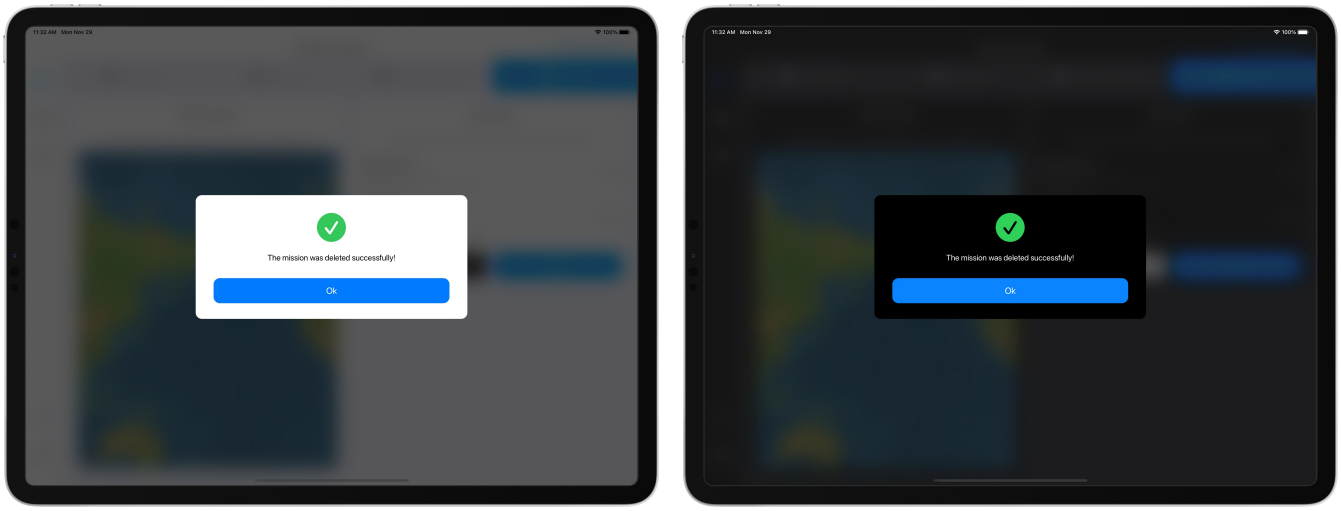

Figure 22: Graphical user interface - iPad App for Drone Missions - View 15

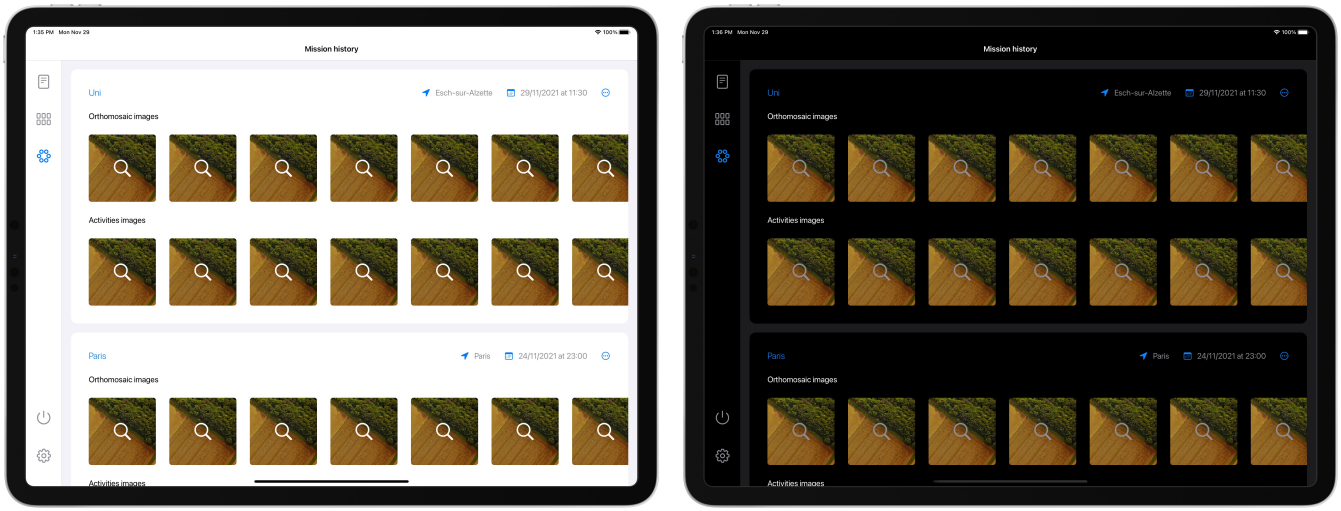

Figure 23: Graphical user interface - iPad App for Drone Missions - View 16

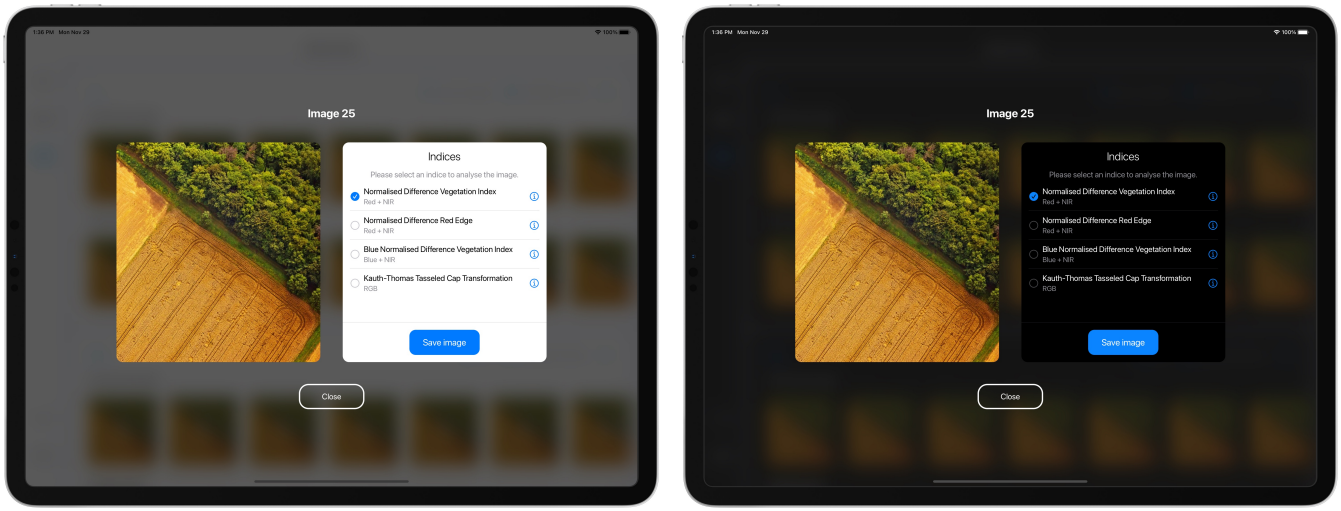

Figure 24: Graphical user interface - iPad App for Drone Missions - View 17

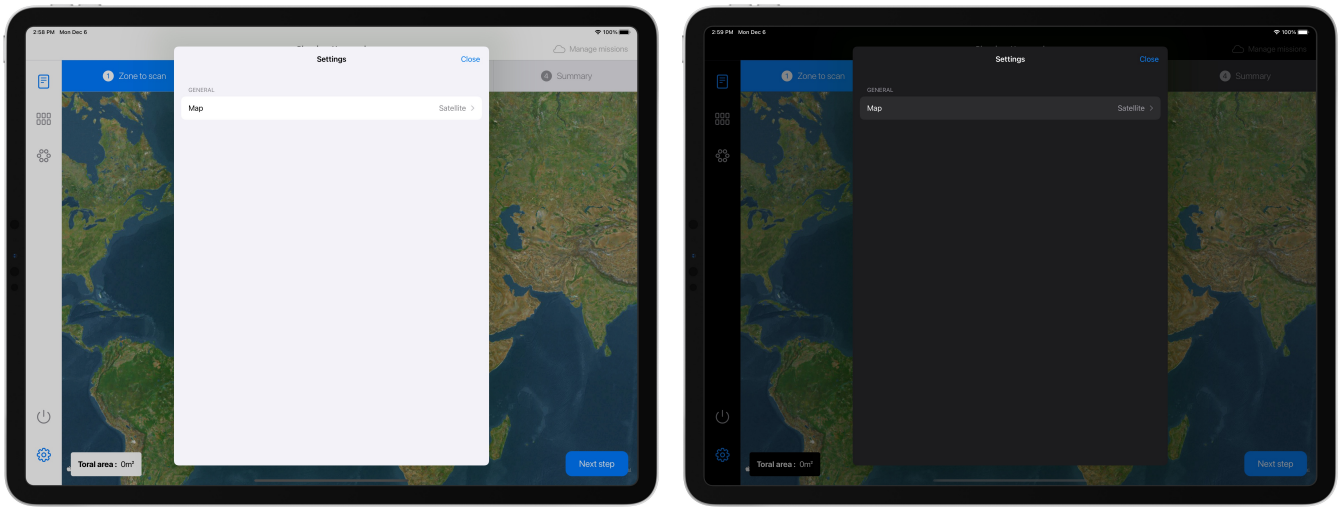

Figure 25: Graphical user interface - iPad App for Drone Missions - View 18

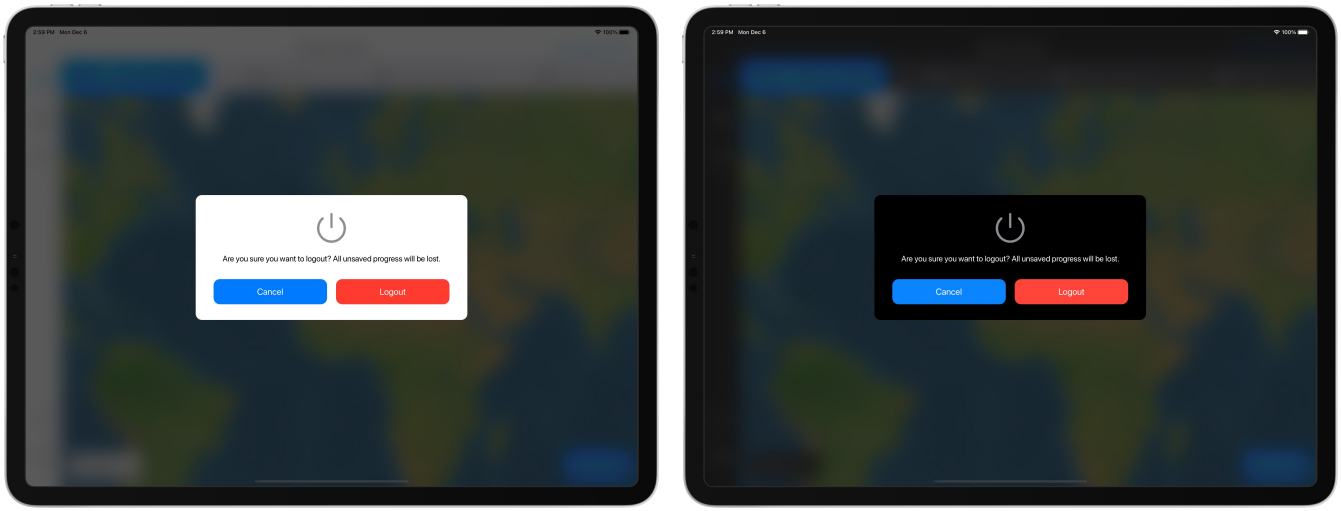

Figure 26: Graphical user interface - iPad App for Drone Missions - View 19