UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

Development and Evaluation of the User Experience of an Augmented Reality Application for Exploration of Aerospace Objects.

Master's Thesis

Master's Degree in Software Systems Engineering and Technology

AUTHOR: Antsyferov, Daniil

Tutor: Jaén Martínez, Francisco Javier

ACADEMIC YEAR: 2023/2024

Resumen

El reciente desarrollo de diferentes enfoques de realidad extendida como la realidad aumentada y la realidad mixta hace necesario evaluar posibles enfoques de desarrollo de software que soporte dichas características y explorar las posibilidades de mejora de la experiencia de usuario. Entre los posibles usos de la realidad extendida son de especial interés las aplicaciones en las que se exploran objetos cuya disposición especial 3D es de relevancia pues en este contexto la realidad aumentada aporta beneficios indudables al situar los objetos digitales en el entorno espacial físico real. Es por ello que en este TFM se propone el desarrollo de una aplicación para la exploración de objetos aeroespaciales como planetas, estrellas, satélites. Se pone el énfasis en la realidad aumentada basada en dispositivos móviles como Android utilizando ARCore. El TFM planteado pretende identificar las ventajas y desventajas de esta aproximacion de cara a la futura implementación de sistemas de exploración de objetos espaciales utilizando realidad extendida.

Palabras clave: Android, Kotlin, Realidad Aumentada, Realidad Extendida, Realidad Mixta, ARCore, Objetos Aeroespaciales, Sistema Solar.

Abstract

The recent development of different extended reality approaches such as augmented reality and mixed reality makes it necessary to evaluate possible approaches to development of software supporting such features and explore the possibilities of enhancing the user experience. Among the possible uses of extended reality, of special interest are applications in which objects whose special 3D arrangement is relevant are explored, since in this context augmented reality provides undoubted benefits by placing digital objects in the real physical spatial environment. That is why in this TFM the development of an application for the exploration of aerospace objects such as planets, stars, satellites is proposed. The emphasis is placed on augmented reality based on mobile devices such as Android using ARCore. The proposed TFM aims to identify possible approaches as well as the advantages and disadvantages of this approach with a view to the future implementation of space object exploration systems using extended reality.

Keywords: Android, Kotlin, Augmented Reality, Extended Reality, Mixed Reality, ARCore, Aerospatiale Objects, Solar System.

Table of contents

1. Introduction ... 7
1.1. Objectives ... 8
1.2. Methodology ... 8
1.3. Structure ... 9
1.4. Relation with studies ... 10
2. Technology overview ... 11
2.1. Comparison with existing solutions ... 12
3. Problem Analysis and Design ... 14
3.1. Use Cases Diagram ... 14
3.1.1. Use cases description ... 15
3.2. Classes Diagram ... 18
3.3. Paging Sequence Diagram ... 19
3.4. User Interface Prototypes ... 21
4. Implementation of the solution ... 24
4.1. System Overview ... 24
4.2. MVVM Architecture ... 25
4.3. Modular Structure ... 26
4.4. Database Implementation ... 28
4.5. Networking Implementation ... 30
4.6. AR Implementation ... 31
4.7. Technologies ... 34
4.7.1. Kotlin ... 34
4.7.2. Jetpack Compose ... 40
4.7.3. Navigation component ... 41
4.7.4. Hilt ... 41
4.7.5. Gradle convention plugins ... 42
4.7.6. Sceneview ... 43
4.7.7. Android Studio ... 44
4.7.8. Version Control System ... 44
4.7.9. Android ... 45
5. User manual ... 46
1. Basic Functionality ... 46
2. AR and 3D Functionality ... 47
6. Future work ... 50
7. Conclusions ... 51
8. Bibliography ... 53

Table of figures

Figure 1 Use cases diagram 14
Figure 2 Classes diagram 19
Figure 3 Paging Sequence Diagram 20
Figure 4 Asteroids List and Asteroid Detail prototypes 22
Figure 5 Prototypes of 3D/AR visualizations 23
Figure 6 Room Entity class 28
Figure 7 Room DAO class 29
Figure 8 Room Database class 29
Figure 9 Ktor HttpClient setup 30
Figure 10 Ktor API class setup 31
Figure 11 ArCore Feature Tracking and World Understanding 32
Figure 12 Example of Gradle convention plugin 43
Figure 13 Home screen on tablet device 46
Figure 14 Filtering and Distance Comparison screens 47
Figure 15 3D models of the Solar System and asteroid's orbit 48
Figure 16 AR models of the Solar System and asteroid's orbit 49

Glossary

- **AR (Augmented Reality):** Enhances the real world by overlaying digital content, like images or information, onto the physical environment via devices like smartphones or AR glasses.
- **XR (Extended Reality):** An umbrella term encompassing all immersive technologies, including VR (Virtual Reality), AR, and MR, that blend the physical and digital worlds.
- **MR (Mixed Reality):** Merges real and virtual worlds, allowing digital and physical objects to interact in real-time, often requiring more advanced devices like HoloLens.
- **ISS (International Space Station):** A habitable space station orbiting Earth, serving as a research laboratory where international teams conduct scientific experiments in microgravity across various disciplines.
- **Observer Pattern:** A design pattern in software engineering where an object, called the subject, maintains a list of dependents, called observers, and notifies them of any state changes, typically used to implement distributed event-handling systems.
- **API (Application Programming Interface):** A set of rules and protocols that allows different software applications to communicate and interact with each other, enabling the integration of various systems and services.
- **NASA JPL (Jet Propulsion Laboratory):** A research and development center managed by NASA, focusing on the design and operation of robotic spacecraft for planetary exploration and other space missions.

1. Introduction

This work was inspired by an accelerating development of new software and hardware that would make users' experience with their devices more ubiquitous. The term can have many impregnations, but here, I am referring to all the different ways of presenting a 3D scene to a user as a substitution or extension of a reality. The idea has several variations, such as Virtual Reality, Augmented Reality, or Mixed Reality. With the development of this area, different approaches and ideas get mixed together and combined in a way that can easily impress wide audiences. With the latest achievements, it seems that wide adoption of some variation is bound to happen. Thus, It has raised my interest as a developer to gain more professional and technical insight into the field and put it to the test.

The project that will be built is an application that is capable of presenting a 3D model of the Solar System using Augmented Reality. This theme was chosen because it presents the exciting challenge of creating a complex scene guided by real-life data with various 3D models accurately positioned and potential for various user interactions. It is also an opportunity to provide users with an interesting perspective on our solar system. As a data source, I have chosen a NASA API that provides regular updates and a wide choice of data that can be presented to the users. Apart from just planets, users will be able to observe various asteroids orbiting around Earth and access different visualizations to better understand the Solar System. A common problem with many simple paper visualizations is that drawing a realistic model on a standard piece of paper is impossible since differences in size and distances between objects are immense. What most people have in mind when visualizing our Solar System is a very simplified model. This project is supposed to bring a more realistic perspective, and while this is still a complex problem on a tablet screen, it can be addressed through the features inherent in 3D modeling

As for Augmented Reality, the choice was dictated by it being the easiest and most widely accessible technology. It is likely to be supported on most modern-day phones and does not require purchases of expensive headsets that can quickly turn into dust collection devices. The concept of Augmented Reality relies on capturing streams of data from a normal camera, depth sensor, accelerometer, and gyroscope and using this information to combine an image of the real world with a 3D scene in a believable, responsive, and interactive way. Using this technology ensures that a user can position a model somewhere in his space and then freely interact with it, study it from different perspectives, or observe changes in real time. The goal is to build an immersive user experience that would be both engaging and educational.

This application's target audience could be anyone with a smartphone and some interest in the Solar System. For example, a parent could use the app to let their child explore the model rendered on a kitchen table. Alternatively, an exhibition could organize a collective experience where multiple users observe orbiting planets from different devices anchored to the same 3D point. This would allow users to view the same model from different perspectives and move around freely. This project is developed from scratch, and depending on the achievements of this phase, it could evolve in various directions.

1.1. Objectives

The project aims to create an application that would provide users with an engaging AR experience by presenting different 3D scenes of the Solar System. It would allow for extensive configuration and let users explore different objects, such as planets and asteroids.

The data would be loaded from NASA API. The networking should use an efficient caching model to prevent excessive requests to the server. Since NASA doesn't require authorization, it can be omitted at this stage, and for testing and development purposes, it is possible to use a free API key.

Additionally, users should be able to list and explore asteroids orbiting Earth. Information on how many different asteroids are orbiting Earth and some filtering for potentially hazardous ones could be an interesting insight to gain.

The goal is for 3D scenes to use realistic models and let users distinguish between real-life scaling and a more visually appealing one. The app should make this distinction so that users can form a complete understanding of the Solar System that is difficult to visualize using traditional sources of information but can be presented in an easy to comprehend way using AR and 3D modeling approaches.

AR experience would include not only the Solar System but also a detailed scene with Earth and some nearest objects like ISS and Moon and orbiting asteroids. This would make the application more distinctive and add a potential reason for users to return to see if any new asteroids have been detected. Since there are not a lot of changes in the Solar System and the planets' orbits can be stored as static data, it is important to provide some dynamic information that changes state over time. In this case including another scene with regular NASA updates for asteroids should prove a valuable extension to the functionality.

1.2. Methodology

Choosing the right development methodology is crucial in order to ensure the successful development of the project. It allows for organizing and prioritizing tasks, building delivery plans, estimating resources, and focusing on the most important parts of the program at the same time. Different methodologies have distinct complexity, and some are more suitable than others for certain projects.

In this case, I have decided to move forward with the Agile methodology. It is important to explain that it is not an out-of-the-box solution for all the problems but rather a set of principles and ideas to adhere to. There are many ways of implementing it, but it essentially

comes down to focusing on delivering working software and continuously evaluating the results to set the next delivery goal. (Thomas & Hunt, 2020)

During the development of this project, I started out with building essential blocks first that would be easily expandable for the added functionality in later stages. It is important to design a system that can easily adapt to new requirements since they tend to change even in thorouly planned projects. Given that the ideas for this app were open-ended initially, it was paramount to make sure that the codebase could be adapted and changed quickly over time. This requirement made it only natural to follow the Agile manifesto and design flexible software on all levels.

Following the initial stages I have been adding new functionality and discussing ideas for further development with my tutor. Thus, following a simple Agile cycle of delivering a small piece of software, evaluating the results, and deciding on the next delivery goal.

This approach made it easy to follow through with the project in terms of being able to set realistic, achievable goals at every stage and plan further advances based on gained knowledge about the domain and possible options for new functionalities. Adhering to a simple feedback loop on both communication and development levels proved to be an efficient way of delivering the application with a fair number of unknowns at the start that I was able to sort out eventually by taking advantage of gained expertise.

1.3. Structure

This work consists of seven sections. The first one presents the general introduction and overview of the project. It also provides some ideas for the scope of the project and how it is related to studies.

The second section presents an overview of the state of AR development in terms of technologies and in terms of possible applications.

The third section has a goal to define functional program requirements and suggest some high-level solutions to presented problems. It also gives an overview of how interfaces are expected to look using prototypes.

In the fourth section, readers can find detailed explanations of implemented functionality, reasoning behind some core decisions, and general information about technologies and tools that were used during development.

The fifth section provides a user manual that explains how to use the application in general and showcases some of the UX decisions that were made during development.

In the sixth section, there is an outlaid and structured explanation of some possible ideas for future development of the project,

The final seventh section presents readers with conclusions and explains how the initial objectives have been met.

1.4. Relation with studies

This project was largely inspired by the subject *Desarrollo de Interfaces Multimodales Avanzadas.* This subject has taught me some basics about how computer interfaces can be integrated with the real world and provided some fundamentals about working with AR. Upon this I have expanded my knowledge and gained skills to create an AR application.

Other related subjects from this course are *Internet de los Servicios (Ios) y de las Cosas (Iot) and Tecnologías de Gestión de Datos.* They both helped when I was developing solutions for networking and database.

Additionally, subjects from my previous years have been quite useful, the most relevant in this scope are *Interfaces persona computador* and *Introducción a los sistemas gráficos interactivos.* The first one helped me view building a smooth user experience and easily understandable interfaces. The second one provided a lot of knowledge on how 3D modeling works, which helped a lot when I was creating 3D scenes for the application.

To sum up, I have relied on knowledge gained from various subjects as well as Android development skills from my professional activities. By combining this information and learning how to work with AR on the Android platform, I was able to deliver the requested application.

2. Technology overview

In its early years, AR gained a reputation as an entertainment feature heavily employed in various games such as Pokémon Go and face mask filters on Instagram and Snapchat. However, technology is developing, and its uses are also evolving. Nowadays, AR is employed in many industries and has a multitude of applications. Both hardware and software advances have helped the technology a lot, and by employing those apps using AR can feel immersive but also be accessible on many devices. In this section, I would like to provide an overview of AR advances and explain why it is beneficial to have this technology in a professional portfolio.

The important advancement in terms of hardware is that AR is nowadays available on more than 90% of mobile devices. (Boland, 2021) On IOS, it is basically any phone later than the iPhone 6s and is approaching full coverage on Android as well. This statistic refers to WebAR, which is not the same as native and is more limited, but it still provides a perception that almost any new phone can run some sort of AR application. In practice, this demolishes a huge barrier because it makes AR available for new users out of the box without any need for additional investments. This change in AR support on mobile devices also incentivizes businesses to build engaging features around it since they finally will be able to cover a wide audience of people, which in turn keeps pushing the industry forward.

General support for AR on mobile devices is not the only thing in terms of hardware; various tech giants and new startups are interested in getting their foot in the door. A multitude of AR/VR headsets are available now; some are focused on advanced tracking features such as HoloLens, and some, such as VisionPro, seem to lean into various OS features and try to provide a full computing experience within a headset. Many smaller companies are making headsets for niche experiences as well, for example, for factory or construction workers who can provide guiding instructions, etc. In general, there are many different devices; each one is trying to stand out by providing some edge over competitors. Each generation of devices gets better, and production costs get lower. This means users can get a better AR/VR experience for lower costs, leading to more potential buyers being interested in what modern AR/VR has to offer.

In terms of software, AI advances have also helped AR a lot. One of the inherent AR problems causing shaky experience and breaking immersion is that scanners tend to lose track of the real world when conditions such as lightning, textures, and user movement are not optimal. This causes abrupt loss of tracking, and 3D models get repositioned or just stuck in the AR scene. It turns out AI is quite good at filling out those gaps in world understanding that scanners have, and it can predict a lot of missed data. By now, advanced algorithms that integrate both things together have already been put in place by popular AR frameworks such as ARCore, for example. More about it is described in further sections, but the general idea of combining both technologies and improving the software part of things by leveraging AI strength has been an important step forward for the modern AR field.

In continuation, I would like to mention some of the promising modern AR applications. Those include:

- Shopping – Online stores can improve conversion by adding AR features that allow users to try on clothes or see how furniture looks in the room before buying it. In general, adding interactive product demonstrations powered by Augmented Reality helps to stand out from the competition.
- Navigation – Both indoor and outdoor navigation have promising use cases for AR applications. Outdoor navigation can eliminate the need for users to glance away from the phone screen, thus improving awareness of the surrounding world. Indoor navigation has more problems due to positioning issues, however it can be solved with beacons or visual marks. For example, AR indoor navigation can quickly lead users to the needed shelf in the store or help people reach the correct department.
- Manufacturing – It is possible to employ Augmented Reality to educate workers on the specifics of working with some devices. Or highlight specific parts of the machine that need maintenance, making the process easier. Also, remote assistance can benefit from AR integrations which will provide clear visual service instructions.
- Healthcare – AR headsets can be used during complex surgeries to provide doctors with vital patient information while also keeping their hands free.
- Automotive industry – Apart from navigation, AR technologies can help drivers with parking maneuvers or provide better insights into the condition of the car and what parts need maintenance.
- Gaming - While it is not the highlight of this section, the gaming industry is oftentimes pushing AR and VR technologies forward by constantly innovating on possible interactions to create better entertainment experiences. Many gaming companies provide one way or another for accessing AR/VR games, and with technology growing more mature, those experiences are also improving in quality.

To sum up, over the last few years, Augmented Reality become more accessible than ever by being implemented on most modern smartphones. Additionally, various hardware advances in the field of wearable headsets and software improvements, as well as AI integration, have made AR not only an entertainment tool but also a business and industry instrument. Usage of this technology can provide various benefits in many areas. Therefore, I believe it’s beneficial to familiarize oneself with the inner workings and development process of Augmented Reality applications. Moreover, this project can become an interesting example of AR application in the educational area and may be expanded upon in the future.

2.1. Comparison with existing solutions

It is important to mention some of the similar applications available in the market and to explain how my project is different and what was improved in comparison; this is the goal of this section.

After some research in the Google Play Store, I have revealed that, indeed, it already has some similar apps; however, they seem to be lacking one part or another of what this project is aimed to achieve. I could split them into two categories: ones that provide AR experience of the solar system and others that supply users with accurate dynamic information on asteroids orbiting Earth.

Examples of the first type are Planets AR and Solar System Scope. They do enable AR in a similar fashion that my app is intended to do, but the general user experience seems to be gamified and they are lacking some accurate representations when it comes to showcasing real

scaling of distances. The first one seems to use ARCore, and the other is built on Unity. Both have put a lot of work into user interfaces and provide a smooth AR experience with some ability to turn it off and just show 3D models; however, there is no dynamic information about asteroids, which may make the experience a bit flat. My application intends to solve this issue by enabling regular updates of the cached asteroid data, thus providing users with new information and experiences whenever they come back to the application.

The second type is more of a catalog application for example, Neo Tracker. It has an up-to-date list of asteroids and gives users all the information in text form. However, it doesn't do any visualizations or AR experiences. It is worth mentioning that such apps exist, but my project builds further on this idea and combines the asteroid catalog with engaging visual representations. Also, it is important to reflect that NASA not only provides an API but also has an application that is somewhat like a catalog-type app but designed to showcase a multitude of different projects. It is seemingly more complex than other mentioned apps and has a larger team of engineers working on it, thus making it difficult to compete. However, this work has some key differences, such as enabling the AR experience and focusing on detailed visualizations of asteroids, while the NASA app is more of a portfolio designed to show many things at once but with less detail.

To summarize, my project combines some of the existing ideas, and it is intended to build upon them to improve the general concept of an application that serves as a guide for different objects in the Solar system and provides AR visualizations for better impressiveness. There is also no intention to gamify the interface; despite being more attractive to children, such apps seem to lose their relatability and clarity of presentation due to a lot of visual noise. The goal is to capture users' interest through various intuitive interactions and captivating visual demonstrations while keeping all the data accessible and clear for understanding. One of the methods of achieving this is including a mode for displaying real scaling of distances. This will create some contrast between the traditional representation of the Solar system and the real situation. Thus, building a visually interesting but also scientifically accurate application is how this project is different and improves upon the existing ones.

3. Problem Analysis and Design

The goal of this section is to describe the functionalities that the system will implement and define their requirements. Also, a modeling solution is provided so that it can be followed during implementation.

3.1. Use Cases Diagram

The goal of the use cases diagram is to express the system's functional requirements. (Bittner & Spence, 2002)

In this case, the diagram presented below and the following explanation will outline the required features and functionalities that the system is expected to possess. The system will generally allow users to view and scroll the list of asteroids indefinitely due to paging implementation. Users will be able to filter this list or select one of the asteroids to view its details.

However, the most important functionality part is 3D/AR visualization of the solar system or selected asteroid that orbits the Earth. The system will automatically detect device capabilities and display a suitable visualization. Displayed models will be interactive and designed to capture users' interest in the subject.

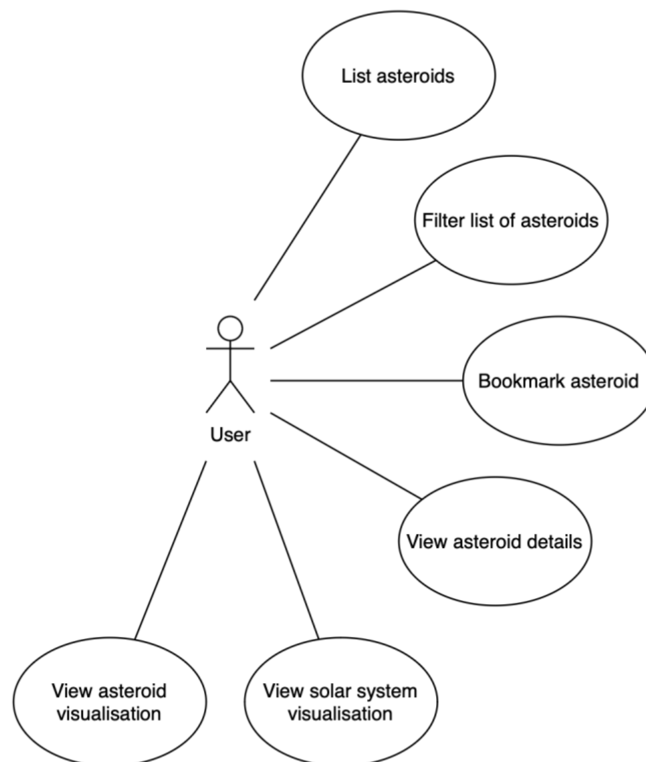

Figure 1 Use cases diagram

3.1.1. Use cases description

In this section detailed explanations are presented about the use cases found in Figure 1.

Use case	List asteroids
Actors	User
Goal	Present a list of asteroids on screen
Summary	System loads recent data from NASA API and presents it on the screen in readable form.
Preconditions	-
Postconditions	Application has cached all loaded asteroids and they are shown in the list element on the screen.

User actions		**System actions**	
1	User opens the application.	**2**	System performs a network request to load initial batch of data.
4	User scrolls downwards to the end of the displayed list.	**3**	System caches the loaded asteroids and displays them to user.
		5	System detects that user has scrolled to the bottom of the list and initializes loading of next batch of data. Then system goes to step 3.

Use case	Filter list of asteroids
Actors	User
Goal	Find asteroids that respond to certain criteria.
Summary	Application allows user to filter the presented list of asteroids to find the needed ones more easily.
Preconditions	List of asteroids has been loaded.
Postconditions	Only asteroids that correspond with the selected filters are presented on the screen.

User actions		**System actions**	
1	User opens filters menu and selects one or multiple filters among the available: Asteroid size, Danger level,	**2**	System responds to user selections in real time and updates the list of asteroids accordingly.

	Date of discovery, Asteroid name.		
3	User decides to clear one or several of the selected filters.	4	System updates accordingly as in step 2.

Use case	Bookmark asteroid
Actors	User
Goal	Bookmark a selected asteroid to avoid searching for it if user wants to view it details again sometime in the future.
Summary	Application provides a functionality to save a selected asteroid and have easy access to it later.
Preconditions	List of asteroids has been loaded and user has navigated to asteroid details screen.
Postconditions	Selected asteroid has been bookmarked or saved and is now available for review in separate tab.

User actions		System actions	
1	User clicks on an asteroid in the list and navigates to details view.	3	System adds the asteroid to bookmarked tab that allows quick access to them.
2	User clicks on a bookmark button	4	System displays a pop-up stating the success of operation.

Use case	View asteroid details
Actors	User
Goal	Get info about all the details of an asteroid provided by an API and visually estimate it size and distance from Earth and Sun.
Summary	Application provides an overview of key asteroid parameters as well as 2D visualizations of size and distance from Earth and Sun.
Preconditions	List of asteroids has been loaded and user has navigated to asteroid details screen.
Postconditions	-

User actions		System actions	
1	User clicks on an asteroid in the list and navigates to details view.	2	System displays key asteroid parameters and a clear and readable way.
3	User selects size comparison diagram.	4	System displays a 2D diagram that compares asteroid size side-by-side to Ceres, a dwarf

			planet.
5	User selects distance comparison diagram.		System displays a 2D diagram that compares distances asteroid- Earth and asteroid-Sun in an easy to comprehend way.

Use case	View solar system 3D/AR visualization
Actors	User
Goal	Present 3D/AR visualization of a solar system to the user.
Summary	Application detects whether AR is available on user device and displays 3D or AR visualization accordingly.
Preconditions	-
Postconditions	-

User actions		**System actions**	
1	User clicks on visualization button on home screen.	1	System detects AR availability on the device and presents either AR or simple 3D view of the solar system.
3	User interacts with speed control buttons or slides in a horizontal direction.	4	System adjusts the position of the planets around the Sun by making them go around faster or slower or manually adjust in case of slide interaction.
5	User clicks on a planet or Sun.	6	System displays a pop-up window with general information about selected celestial body.
7	User performs a long click on a planet or Sun, alternatively user selects one of the options in an overlay picker.	8	System positions camera behind the selected celestial body so that imitates view of the solar system from that body.
9	User selects real distances scaling and which set of planets to display with it – Mercury, Venus, Earth, Mars or Jupiter, Saturn, Uranus, Neptune.	10	System adjusts the scaling of distances between planets and Sun to correspond to real data. Due to huge discrepancy in distances, it is possible to only display either set of planets at a time.

Use case	View asteroid 3D/AR visualization
Actors	User
Goal	Present 3D or AR visualization of a solar system to the user.
Summary	Application detects whether AR is available on user device and displays 3D or AR visualization accordingly.

Preconditions	User has navigated to asteroid details screen.
Postconditions	-

User actions		**System actions**	
1	User clicks on visualization button on asteroid details screen.	**1**	System detects AR availability on the device and presents either AR or simple 3D view of the asteroid and its orbit around the Earth.
3	User interacts with the model using zoom , rotation or dragging gestures.	**2**	Apart from asteroid system displays Moon and ISS orbiting Earth. Their orbits are outlined by dotted lines so user can visually estimate them independently of where are 3D models are positioned alongside the orbit.
		4	System adjusts the visualization accordingly which allows for more careful study of the positioning of objects of interest.

To summarize this section, the reader has been presented with the general specifications that the system is intended to follow, and in further sections, more details about how it could be implemented will be provided.

3.2. Classes Diagram

This section presents the class diagram, which aims to showcase what data applications will be working with. The application will receive a list of asteroids from NASA API, and Figure 2 has a UML representation of that data.

In general, a single asteroid entity contains some simple self-explaining fields and, in addition, is composed of other entities such as estimated diameter, close approach data, and orbital data.

Among the simple asteroid fields, I would like to highlight the most important ones. Neo reference id is the identification provided by the API, name represents a user readable name that would be presented in the interface, NASA JPL URL is a link to webpage that has all the asteroid info presented in a browser format, link self can be accessed by the app to download additional asteroid information from the cursor-based API if it is needed.

Estimated Diameter, in its turn, is just a helper class that is composed of four estimations in different units but ultimately represents the same data – what NASA believes is the minimum and maximum possible size of a given asteroid. It can be used by the app to help users with filtering small or big asteroids and for size comparison visualizations.

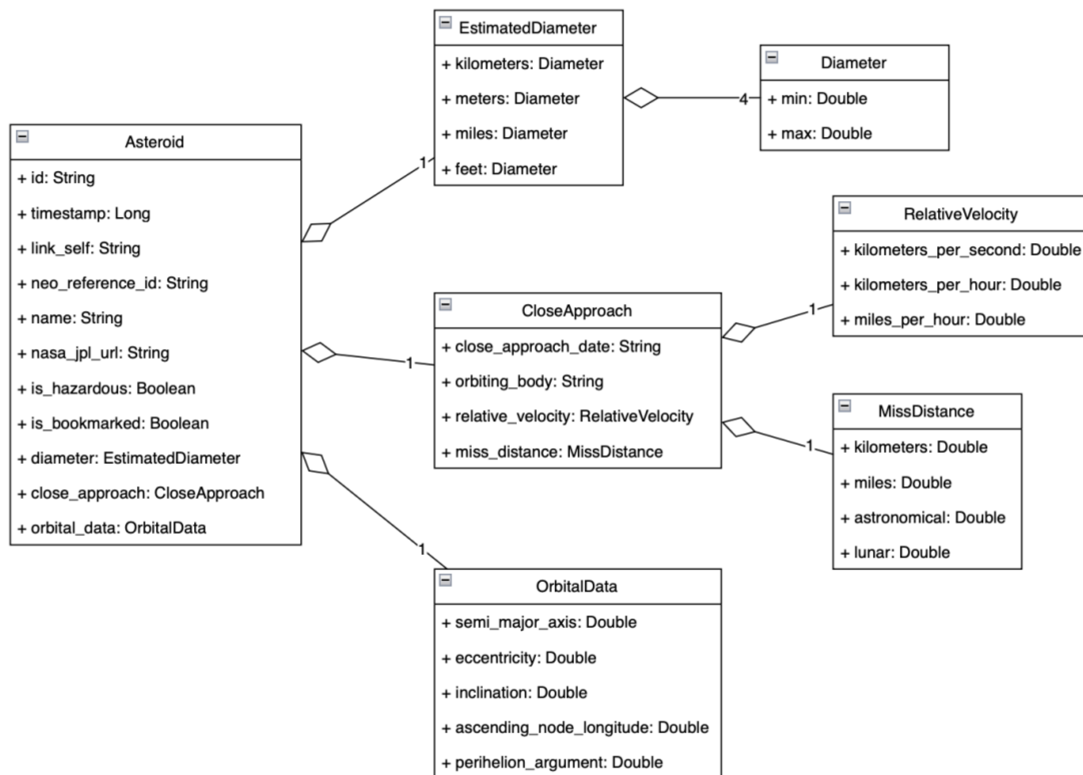

Figure 2 Classes diagram

Close approach entity provides information on what body is asteroid orbiting, usually its Earth, and additional parameters like the minimum distance between asteroid and its orbiting body or the relative speed. This data can be useful in constructing a 3D representation.

Orbital data is an entity that encapsulates all the information needed to draw a 3D orbit of an asteroid. This allows to provide users with accurate visualization after interpreting the data gathered by NASA.

Overall, API provides a lot of information on the asteroids, and during the development, I have picked the most important and potentially interesting data to present to the users. The diagram should have clarified to the reader what the application will be working with regarding input data.

3.3. Paging Sequence Diagram

In this section, the reader can find a sequence diagram in Figure 3. Its purpose is to illustrate how the application will communicate with NASA API to present a seemingly endless list of asteroids to the user. This implies that the users will be able to scroll down the list of asteroids, and the app will be loading more items progressively. To implement this, there is already pagination present in the API structure, but the application will also have pagination on the UI layer.

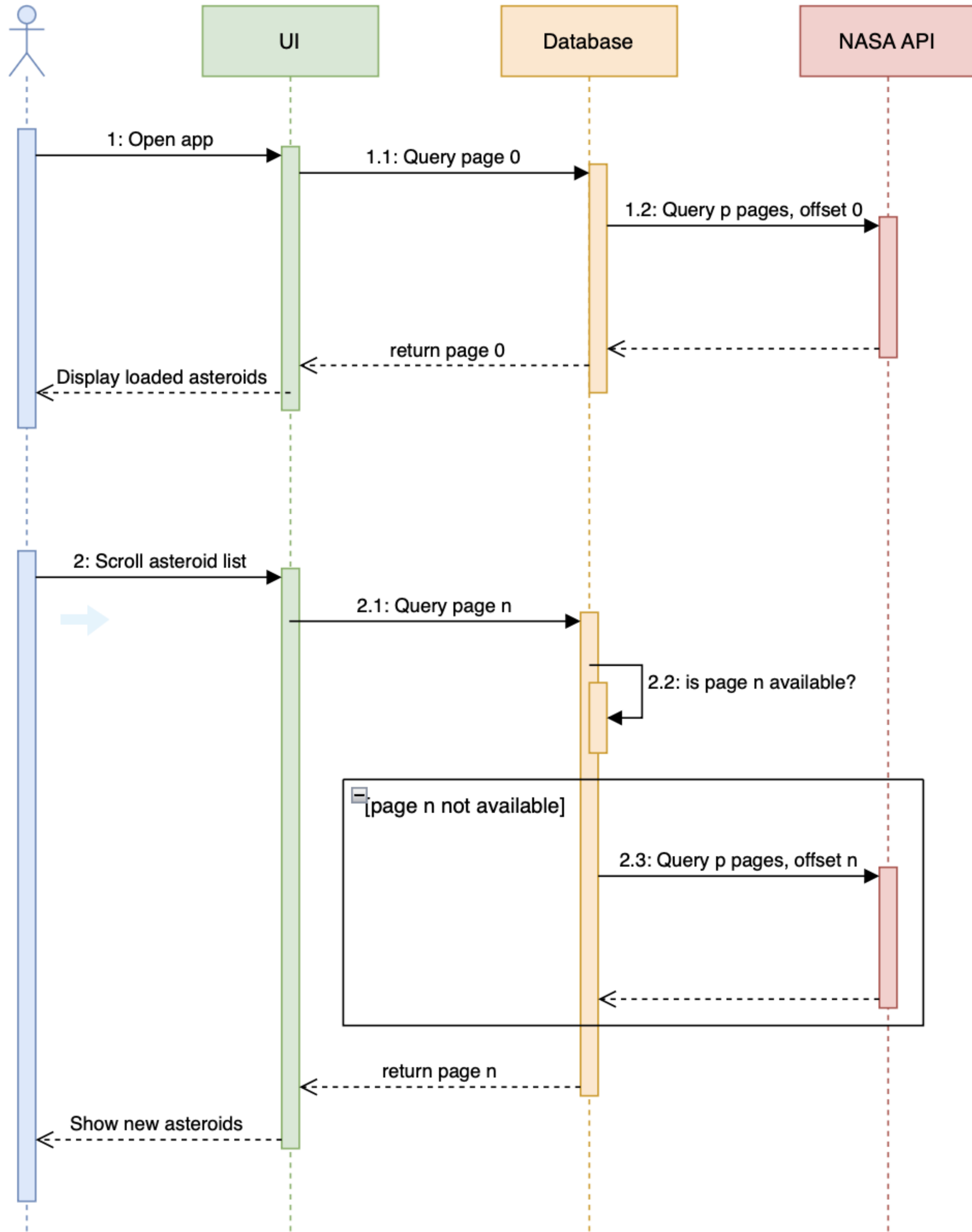

Figure 3 Paging Sequence Diagram

In the initial scenario, when the user first opens the app, the UI layer will try to load the first page, and since the database is empty, it is guaranteed to make a network request to load some asteroids. In the diagram there is a **p** number present. This is because the size of one page on UI can be relatively small; with ten to twenty items, the whole screen can be filled, and the user will have some preloaded items ready when he starts scrolling. However, with the API, it is better to request more items on one page so that network traffic is optimized. This means the

size of one page on API is bigger than one page on UI. This is why there is a p, which simply gives information on how many UI pages fit into one API page.

$$\boldsymbol{p} = \left\lfloor \frac{number\ of\ items\ in\ page\ on\ UI\ layer}{number\ of\ items\ in\ page\ on\ API\ layer} \right\rfloor$$

The result of this is that after the initial launch, the application has loaded and presented the first page of asteroids to the user and cached some pages in the databases that the user can get to by scrolling down.

When the scroll occurs, the UI layer will decide which page number it needs to show in response, which will result in a request to the DB layer for page **n**. Database, in its turn, will check if page n has been cached, and if not, then it will make an appropriate request to the API. Afterward, the page that UI has requested will be returned, and in case a network call takes place, some additional **p** pages will also be cached.

As a result, the system will optimize the device memory by loading from the database and displaying items that the user can see on the screen, plus some of those that can be reached quickly. But it will also avoid unnecessary API calls every time the user scrolls to the bottom of the list; depending on how big p is, a few calls will be skipped, and the application will read that data from the cache. As a developer, I will need to look for some optimal value of p, which I will be able to control by adjusting the page sizes that the application is working with. This will be explored in further sections.

3.4. User Interface Prototypes

In continuation, some of the initial prototypes are presented. They were used in the early stages of development to visualize ideas for the user interface and possible ways of arranging a 3D scene, which would provide users with an intuitive interface and easy-to-understand 3D models.

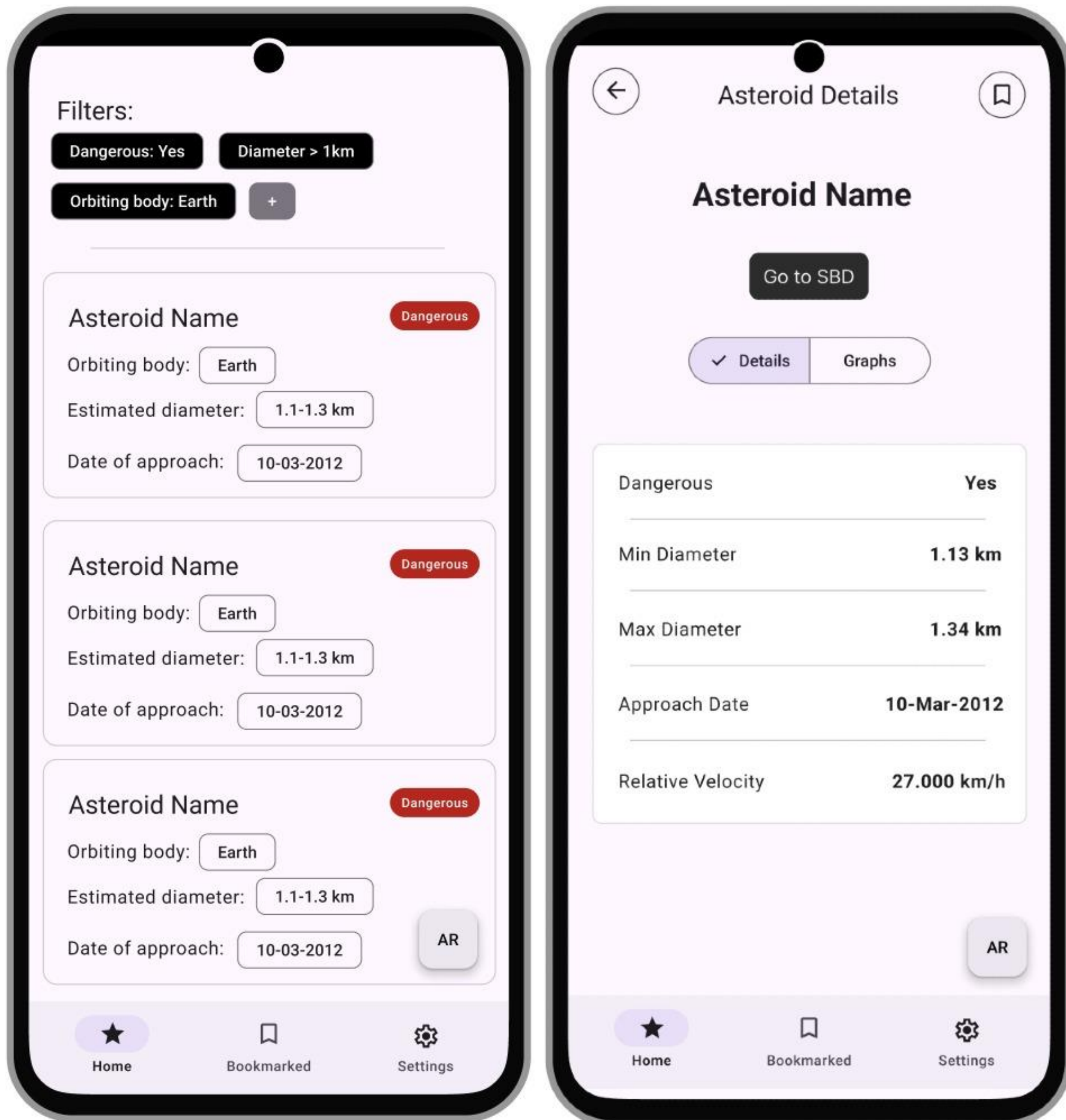

Figure 4 Asteroids List and Asteroid Detail prototypes

In figure 4, the prototypes of the home and asteroid details screens are presented. A key design decision was to incorporate a bottom navigation bar, which offers users a familiar method for navigating and switching between various sections of the application. Later in development, the navigation bar was expanded to turn into a vertical navigation rail on large screens. Additionally, both asteroid list and asteroid details panes can be viewed on tablets simultaneously, which is considered a best practice for adaptive layouts. (Natali, 2024)

More prototypes are presented in Figure 4. They provide some understanding of what the 3D visualization is going to be like. The first one is the whole solar system in general, with a fair number of controls that allow us to adjust planets' rotation along their orbits and change camera perspective.

The second is intended to be a detailed view of asteroid orbit around the Earth. A red triangle would be an asteroid, and the grey triangle would be International Space Station. Also, the Moon is depicted with a circle.

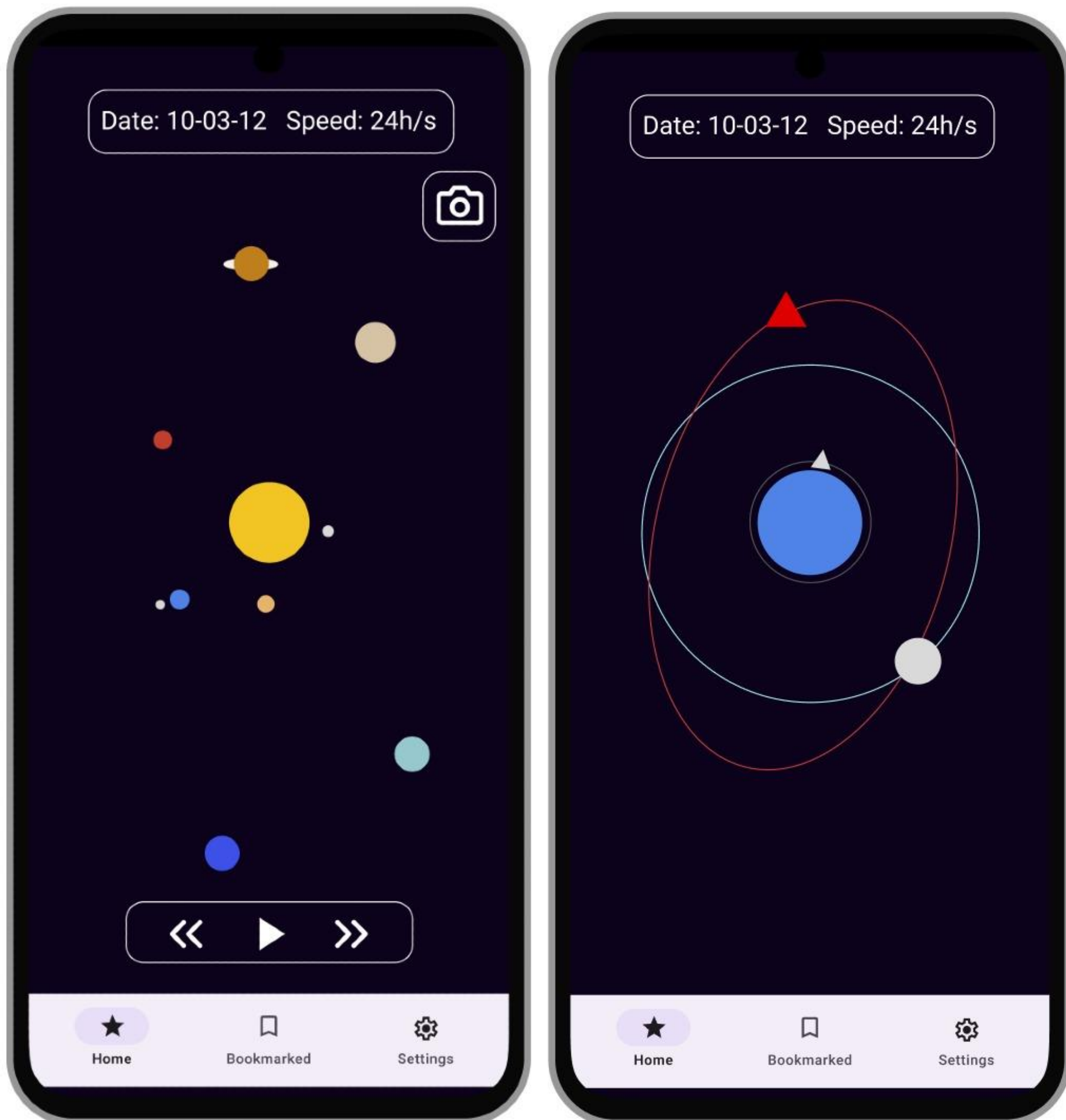

Figure 5 Prototypes of 3D/AR visualizations

Overall, the application has preserved the core elements outlined in the prototypes. However, adjustments were made during development to enhance user experience and better align with practical needs. Obviously, the presentation of 3D scenes is purely structural; in the application, suitable 3D models would be used to present planets and other bodies. The final layout of the application will be presented in the following sections.

4. Implementation of the solution

This section describes the architecture of the system and is essential for understanding how the application is structured and how its different parts work together. Moreover, this section will have an in-depth overview of every major part that is important in this scope.

4.1. System Overview

In general, the system is a client-side application that only uses available API for retrieving needed data and then handles it locally. Since the focus of the application is displaying 3D and AR scenes to the user, no dedicated server development was needed, and all the valuable logic was implemented in the scope of the Android client.

The resulting application uses MVVM architecture, which is a common approach to implementing MVC architecture in Android. It achieves decoupling of View classes from Model classes relying on the Jetpack library, which is currently encouraged by Google as the best up-to-date solution; more on this in further sections.

The application is divided into modules. The exact approach and reasoning behind this will be explained in the following section. But here, I would like to mention some key components so that the reader can form a general image of the application. The key parts of the system would be the database module, networking module, AR/3D feature module, asteroid details feature module, asteroid list feature module, and application module.

Database and networking modules provide implementations for their functionalities; they encapsulate exact implementations that are platform and library-specific. As a result, other modules within the app can work with contracts or interfaces that provide required data, and no dependencies or knowledge of the inner workings of the database or networking is required.

Feature modules that I named all belong to the UI level of the app and are split into several modules based on their logical separation, as well as some unique specific dependencies that do not need to be shared. For example, the asteroid list feature would be using a pagination library, whereas the 3D/AR module would need access to the AR Core library. Both feature modules implement very different functionalities and do not want to share named dependencies in order not to overcomplicate the system.

The app module serves as the UI layer of the application (Robert, 2018), interfacing directly with the Android framework to handle necessary system communications. It is tasked with managing features that involve user interactions or hardware access, as these operations must be conducted through the Android OS to adhere to security policies. Additionally, this module functions as the entry point for the application, being launched and overseen by the

Android system. It is responsible for initializing the app and managing its lifecycle within the Android environment.

4.2. MVVM Architecture

In software development, it is considered a positive pattern to separate the logic of drawing visual information on the screen from the logic of managing, updating, supplying, and storing this information. Traditionally, it can be achieved using the Model View Controller pattern, in short, MVC. (Gamma, et al., 1995)

However, in Android the specific implementations may vary due to various particularities of the platform. Earlier implementations relied on the Model View Presenter pattern, further called MVP. It did provide sufficient decoupling of business and UI logic so that resulting Views would be quite thin and less error-prone, while the Presenter taking on the role of controller could be easily Unit tested. The problem with this approach lay in the fact that both the View and the Presenter had to hold references of each other. This additional coupling could be dealt with through the use of interfaces. Nevertheless, the Presenter holding reference to a View was indeed a problem. In Android, Views are often implemented by Activities or Fragments and those are classes whose lifecycle is controlled by OS. Whenever Android decides to clear memory and destroy an object of Activity, for example, it should be able to do so. But in the case of MVP, after Android clears all its references to an Activity, it would still be referenced by a Presenter, which prevented it from being cleared from memory. Developers have come up with different solutions to battle this problem, but it was never completely solved until the introduction of Model View ViewModel architecture, MVVM later.

The key new component introduced in MVVM as a part of the Jetpack library by Google was a ViewModel class. It would assume the same role as the Presenter or Controller but eliminate the need to hold a reference of the View. It would pass data to the UI using the Observer pattern, which generally allows a View to subscribe for updates from ViewModel, and the latter doesn't need to know anything about its subscribers; it can simply push a new state down the pipeline while it is active. The initial implementation of this Observer pattern in Android was called LiveData. However, the latest best practices suggest using Kotlin Flows, which doesn't require any additional libraries and comes built-in with Kotlin. Obviously, this is only the case if the program is written in Kotlin, whereas in Java, LiveData is still the best option.

An additional note on this is that View can hold a reference of ViewModel, or Presenter for that matter, without any memory leak issues. These controller classes are designed to outlive View's lifecycle and thus prevent loss of data after a configuration change. This could be a change from a light to dark theme or screen rotation, for example. In short, this particular coupling is not viewed as a problem and does not cause memory leaks.

To sum up, MVVM was chosen as the best modern approach, which solves all the pressing issues and does, in fact, provide enough decoupling between Model and View so that

Android development has been able to switch to a completely new UI framework seemingly without any changes to MVVM, more on this in further sections.

4.3. Modular Structure

As previously stated, the application is built using a modular structure. Before proceeding to present the exact structure, it is important to explain to the reader what reasoning stands behind modularization and what benefits can be obtained with its use.

Regarding general code qualities such as scalability and readability, these attributes tend to diminish as the codebase grows. Therefore, developers should proactively address these issues by implementing suitable architecture to ensure the growing codebase remains manageable and maintainable.

It is worth mentioning at the start that such decisions are of a trade-off type, and there is no single solution to the problem but a range of options that have different characteristics. The developer’s job, in this case, is to choose the option that suits the given program, team, and client the best. Meaning a wide range of characteristics needs to be considered. In this chapter, the problem is discussed only from the perspective of building a maintainable and robust codebase; however, in a corporate scenario, many other issues would need consideration.

One way to address the aforementioned codebase issues is by splitting the app into distinct modules. A module can be seen as a self-contained package that can be replaced or reused as needed. In this application, dedicated Android modules are employed, following a similar concept. Here are the benefits of a modular architecture:

- **Scalability**: In large interdependent codebases, even minor changes can unintentionally affect unrelated functionality, leading to undesirable behavior. By splitting the code into modules that adhere to the principle of separation of concerns, the potential impact of changes can be contained within specific modules.
- **Encapsulation**: Modules ensure that each part of the code has minimal knowledge of other parts. Code within modules can be marked as internal or private, making it invisible outside the module. This enforces the use of abstractions, such as interfaces, and prevents other modules from relying on internal implementation details.
- **Reusability**: Modules act as building blocks for the application and can be exported and reused in other projects, reducing the development time for future applications. In a corporate environment, reusing common code can increase delivery speeds and make it easier for developers to transition between projects.
- Additional benefits of modularization include:
- **Ownership**: Clear module ownership can improve code quality, as developers or teams know they are responsible for the benefits and drawbacks of their code.
- **Testability**: Modules make it easier to test individual components, enhancing the system's robustness.
- **Reduced Build Times**: By leveraging parallel compilation and cached builds, modularization can significantly decrease build times.

While these benefits are valuable for large systems, they may not be primary concerns for smaller projects with fewer developers and shorter build times. However, achieving these benefits requires appropriate modularization tailored to the specific application.

Key concepts to consider when modularizing are granularity, coupling and cohesion:

- **Granularity** refers to the extent to which the codebase is divided into modules. Developers should find a balance between too fine-grained modules, which can introduce complexity and maintenance overhead, and too coarse-grained modules, which may not provide significant benefits and resemble monolithic structures.
- **Coupling**: This measures the degree of interdependence between modules. Low coupling is desirable, as it indicates that modules are relatively independent and changes in one module are less likely to affect others.
- **Cohesion**: This represents how functionally related the parts of a module are. High cohesion is desirable, as it makes the application logically structured and the code clearly readable.

To achieve effective modularization, two main strategies can be employed, often in combination: modularization by layers and modularization by features. (Android Developers, 2022)

- **Modularization by Layers**: Each module is responsible for a different layer of functionality. For example, a networking module handles server communication, a data module manages database operations, a domain module contains business logic, and an app module acts as the entry point for the Android application, managing the UI state.
- **Modularization by Features**: Each feature module contains all the code necessary for a specific functionality. For instance, one feature module might handle querying, creating, and displaying asteroids while another manages asteroid details. However, this approach can lead to repeated boilerplate code, as feature modules may need to reuse code from other features, challenging their independence.

A hybrid approach combines both strategies. Feature modules are further divided into layer-specific modules, with core modules providing shared access to resources like databases or network services. This allows feature modules to reuse code within the same layer, reducing coupling while maintaining modularity.

In this project, I have implemented a variation of the hybrid option, meaning data and networking modules are only split into layers, but all other UI modules are split into features. It assures that all feature modules can gain access to asteroids stored in the DB or query new ones without introducing any dependencies cycles; at the same time, feature modules are independent and know nothing about each other. This means that resulting modularization can be characterized as low granularity, low coupling, and high coherence. The drawback of such a solution is the necessity to handle a lot of boilerplate code during module setup. I will provide information on how this issue was tackled in further sections.

4.4. Database Implementation

The database was added to this application primarily to cache a large number of asteroids, which can save a lot of networking and improve responsiveness since queries to the DB are much faster than network requests for the same. In section 2, it was already explained how the database functions as a cache layer that saves an arbitrary number of pages and lets UI access them quickly or, if not available, query the API. This section provides more details on the exact database technology used.

Room is an Android persistent library that provides an abstraction layer over SQLite, which enables easy access to the database while retaining all the power of SQLite. Most of its advantages come from user-friendly annotations, which allow structure of data and queries, simplifying some tedious SQL code, but most importantly, they enable compile time verification, which means most of the simple errors can be eliminated before the application is even launched. This greatly reduces development and debugging time, which makes Room a perfect solution for the Android environment.

The library is divided into three main components which are:

- Database class, which contains all the database configuration information and serves as the access point for the app to persistent storage.
- Date Entities represent tables that will populate the database.
- Data Access Objects, further referred to as DAOs, enable CRUD operations over the saved data.

```
@Entity(tableName = "asteroids")  Daniil Antsyferov *
data class AsteroidEntity(
    @PrimaryKey(autoGenerate = false)
    val id: String,
    @ColumnInfo( name: "name")
    val name: String = "",
    @Embedded
    val estimatedDiameter: EstimatedDiameterEntity,
)
```

Figure 6 Room Entity class

In Figure 6, the reader can see an example of a simplified asteroid entity. A data class is declared as a table with *@Entity* annotation, its fields also can be annotated to indicate *@PrimaryKey* and *@ColumnInfo*. Most data classes can be configured with these annotations; a more advanced one is @Embedded, which allows the merging of different entities into one table. It makes code much easier to follow while handling all the SQL requirements.

```
@Dao  Daniil Antsyferov *
interface AsteroidsDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)  Daniil Antsyferov
    suspend fun insertAll(asteroids: List<AsteroidEntity>)

    @Query("SELECT * FROM asteroids")  Daniil Antsyferov
    fun pagingSource(): PagingSource<Int, AsteroidEntity>

    @Query("DELETE FROM asteroids")  Daniil Antsyferov
    suspend fun clearAll()

    @Query("SELECT * FROM asteroids WHERE is_bookmarked=1")  new *
    fun getBookmarkedAsteroids(): Flow<List<AsteroidEntity>>

    @Query("UPDATE asteroids SET is_bookmarked=:isBookmarked WHERE id=:id")
    suspend fun setBookmarked(id: String, isBookmarked: Boolean)
}
```

Figure 7 Room DAO class

Figure 7 presents a DAO class for the asteroid table. It is also simplified to avoid introducing too much detail, but it shows examples of different types of queries that can be made using Room annotations. It also demonstrates that Room is capable of transforming the return type of the data into Kotlin Flows or, for example, PagingSource from the paging library, proving it has tight integrations with the Android ecosystem. Additionally, queries are verified statically and compile time; if any error had been introduced, such as misspelled column name or incorrect SQL syntax, Room would detect and highlight it in the IDE.

```
@Database(entities = [AsteroidEntity::class], version = 1)  Daniil Antsyferov
abstract class AppDatabase : RoomDatabase() {
    abstract fun asteroidsDao(): AsteroidsDao  Daniil Antsyferov
}
```

Figure 8 Room Database class

Finally, there is a database class that, when instantiated, provides applications with references to the DAOs. It also contains a list of all the entities that Room should create a table and a version number used for migrations.

Overall, Room is a powerful and versatile solution for an Android Database; apart from the shown examples, it also has support for more complex queries and allows to set up migrations from older versions with a variety of strategies. Using it in this project allowed us to develop a caching solution in a reasonable time while not introducing any compromises in terms of robustness.

4.5. Networking Implementation

To implement networking in the application, I have used the Ktor library instead of the more usual choice of Retrofit. Both libraries are a valid choice; however, Ktor is a newer one, and it was valuable to gain some experience with it. Also, one of the advantages is that Ktor also provides all the needed networking functionally for the server side of the communication, which means potentially, if the developer is working on a server and a client implementation, there would be less friction when setting up the requests since the same tool would be used on all platforms.

Reference of all platforms requires some explanation; Ktor doesn't have any external dependencies and can run anywhere Kotlin can run. This was achieved because Ktor was developed specifically for usage with Kotlin, thus the first letter 'K'. As of now, Kotlin is mainly used on Android and server side. However, Kotlin Multiplatform is active development, meaning some parts of it are in alpha stages and some are in beta. Assuming this multiplatform initiative is properly released, developers using Ktor can count on having a single implementation of their networking logic between all platforms. At the moment, there is no such advantage if the developer is not willing to risk using unstable multiplatform releases but evaluating only the Android side of things, there are also no disadvantages compared to Retrofit.

```
fun provideKtorClient(): HttpClient {  Daniil Antsyferov *
    return HttpClient(Android) {
        // configure serialization
        install(JsonFeature) {...}

        // configure logging
        install(Logging) {...}
        install(ResponseObserver) {...}

        // configure auth interceptor
        install(DefaultRequest) {...}
    }
}
```

Figure 9 Ktor HttpClient setup

The setup for Ktor is fairly simple, and the first step is to create an HttpClient and configure it with the needed features. As seen in Figure 9, in the application, I have added JSON serialization, logging of requests, and configured an interceptor that would add an authorization header to every request. Instead of JSON, it is possible to use XML or ProtoBuff; in this case, I had to use JSON to deserialize the API response. Logging was added purely for utility and ease of debugging. There are more complex solutions like Chuker, but that would be excessive in the scope of this app. Finally, adding a default authorization header allowed us to avoid manually adding the same header for each request to API that needs a key for authorization.

```
class AsteroidsApi constructor(   Daniil Antsyferov *
    private val client: HttpClient
) {

    companion object {   Daniil Antsyferov
        private const val BASE_URL = "https://api.nasa.gov/neo/rest"
        private const val ASTEROIDS = "/v1/feed"
        private const val DETAILS = "/v1/neo"
    }

    suspend fun getAsteroids(   Daniil Antsyferov
        startDate: String,
        endDate: String
    ): AsteroidsResponse =
        client.get( urlString: "$BASE_URL$ASTEROIDS?start_date=$startDate&end_date=$endDate")

    suspend fun getAsteroidDetails(   Daniil Antsyferov
        asteroidId: String
    ): AsteroidDetailsResponse =
        client.get( urlString: "$BASE_URL$DETAILS/$asteroidId")
}
```

Figure 10 Ktor API class setup

The second and final step of setup can be observed in Figure 10. Here, I have configured two requests to the NASA API to get a list of asteroids in a specified timeframe and to get asteroid details. It is good to separate the base URL and path for each request since it allows swapping between base URLs easily in the future; it can be very useful during development if different environments are present, such as dev, stage, prod, etc. Each environment has a different base URL, but request paths are often the same.

Sample API class uses earlier created HttpClient to perform the defined requests. With these two steps, the networking configuration is almost complete; the only remaining task is creating classes AsteroidsResponse and AsteroidDetailsResponse. I will not demonstrate their code since they are just data classes holding the same values as Entities in the database but annotated in a different way so that Ktor can deserialize them.

Overall, the networking setup is very concise, and it is easy to appreciate the Kotlin way that Ktor is implemented. As for the API functionality, I only needed two requests to get all the required data from NASA that will be used to display asteroids. As for the planets orbiting the Sun, their orbits are not subject to change and have been provided to the application in a separate configuration file.

4.6. AR Implementation

AR functionality in the application is implemented with the use of ARCore. It is a library developed by Google for Android that can cover all the needs such as modeling a 3D scene,

working with a camera stream, detecting planes, and assigning 3D objects to a fixed position on detected surfaces. ARCore is a natural choice for the Android environment because it allows one to achieve all this functionality without the need to use external tools such as Unity; with this library, developers can keep working in a well-known system. However, the tradeoff is that ARCore may seem limited in some ways and is not the most advanced tool in terms of building a complex 3D scene. It also only works on Android, so in case of future expansion of this project on other platforms, I would consider switching to something multiplatform out of the bag. But for the current purposes of this project, ARCore fits well and has allowed the creation of visually captivating and interactive scenes.

ARCore builds an immersive user experience by combining depth estimation, light estimation, and motion tracking. It relies on complex algorithms to build an understanding of the world from these sensors and produces data that describes shapes in the scene as well as estimations of how it changes over time. This data is called features, essentially dots positioned in 3D space using camera and depth sensors, and what describes the change of these features in space is feature tracking. (Linowes, 2018)

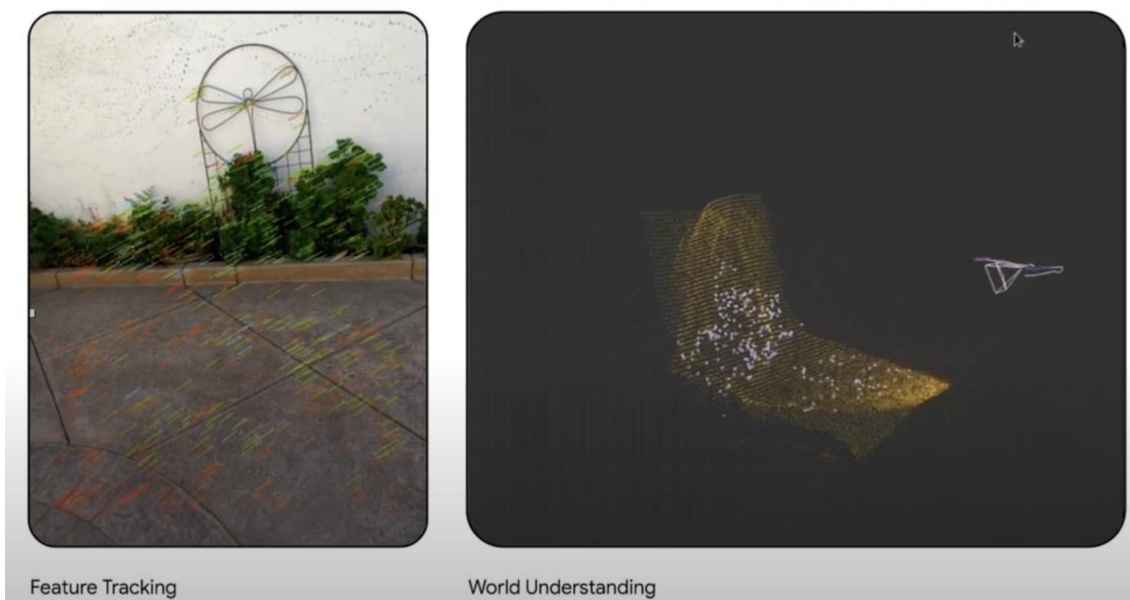

Figure 11 ArCore Feature Tracking and World Understanding

In Figure 11, there is an example of detected features over the scene. Initially, they are dots, but the camera movement library is capable of tracking their change over time, which makes them into lines. To the right, there is an example of derived world understanding that is used to compute possible locations where 3D assets can be placed in the scene.

Under the hood, ARCore uses an Inertial Measurement Unit or IMU to track motion; it combines data from the gyroscope and acceleration sensor to estimate the device's position in space and correctly adjust the features. A combination of camera, depth, and IMU data is what allows ARCore to provide a seamless user experience.

However, there are a couple of potential issues with this approach. The most obvious is the lack of light; the scene needs to be well-lit in order to detect features correctly. Another is that the texture where the device is pointed shouldn't be very monotonous. Otherwise, the algorithm just isn't able to detect enough features. This can be thought of as having enough distinctive points in the scene to make sure that various features can be generated. One more issue is lack of movement or too abrupt movement; in the first case, without understanding how the features are changing, it is harder to estimate their exact 3D position. In the second case, with too much movement, the device tends to lose track of the features and struggles to fill in the gaps, oftentimes leading to rebuilding the scene from the beginning when abrupt movement stops. These issues should be communicated to the user, and UI should lead users into correct behavior so that they ensure the environment is well-lit, move the phone around, and keep it smooth. This can be explained with various hints on the screen during the startup of the AR experience.

While the first two problems depend on the user's behavior, the issues with losing track of features and not being able to fill in the gaps can be partially tackled algorithmically. ARCore has a built-in mechanism called Anchors. They are essentially fixed points in the 3D space, and ARCore does its best to keep their position fixed after device movement or loss of visual contact. Traditionally, after the loss of camera input, ARCore would keep estimating the position of an Anchor using only IMU data. However, this approach is sensible to small accumulating errors, and that's why lately, the library added a Machine Learning component to correct the result and eliminate possible divergence.

Google has trained a model that would take previous ARCore estimations of Anchors' movement via IMU input and try to eliminate the errors that were happening. It proved to be beneficial and was seamlessly added as a feature without the need to adjust code for developers who were previously using the library.

Now when I have explained how ARCore works, I will give more details on how it is used in the application to achieve needed results. First of all, the app needs to check if the user's device supports AR or not; in case it does, the device also needs to have Google AR Services installed. The library provides a set of methods to perform these checks and redirect the users to the corresponding download pages in Play Market. In case AR is not supported, the application will just draw a 3D scene, skipping all the steps of detecting the environment and just setting the root coordinates of the model to (0,0,0).

Next, given that ARCore is configured and ready for use, the application needs to open the camera stream and start the process of detecting planes. During this process, the user will be guided with a hint to slowly move the phone around. In case the library can't detect any plane due to inadequate conditions, the user will also be notified of the exact reason and given a hint to correct this.

Once a plane or multiple ones have been detected, a grid is drawn over it to indicate to the user an approximate understanding of the world that ARCore has built. At this time, the user is

suggested to click somewhere on the plane, which will set an Anchor. After this, the application will render a model around this Anchor.

At this point, the whole scene is rendered and can be freely interacted with. Thanks to the Anchor, the user can move around the room, and the model will stay attached to where he first clicked, meaning the user can go around and see another perspective of the model.

Additionally, I have enabled various interactions using custom gestures. For example, a user can make a horizontal scroll with one finger, and it will make planets rotate forwards or backward around their orbits. This allows users to see the relative position of planets at different phases. It is worth mentioning that the user also has button controls over the planet's rotation and controls the speed with simple clicks as well. Another possible interaction is that the planets are clickable, and when selected, an additional popup shows up to give more information on the planet.

As for the 3D version of the scene, since it doesn't rely on Anchors, it has another mechanism to change the perspective, which is adjusting the camera position by selecting from a dropdown list of predetermined positions of interest, such as on top of the scene, angled, and from the perspective of each planet.

There are more interactions, all designed to build an engaging user experience and showcase the model of our solar system from different perspectives. It is also possible to switch between realistic measurements and more cinematic ones. This is needed because discrepancies in sizes and distances are so immense that fitting them all on the screen is a challenge. The intended use is for the user to interact with a cinematic version first and then get a perspective of a more realistically measured model. It should help build an understanding that most models of solar systems that are presented are oversimplified and don't necessarily provide a full picture.

To sum up, ARCore is a powerful tool in the Android environment and has provided me with all the needed tools to build a capturing user experience. Relying on its vital features, I designed a 3D model and was able to position it correctly, taking into account the real world around me; this should result in users being interested and willing to try the application. Given all of the above I consider the decision to use ARCore a correct and successful approach in this case.

4.7. Technologies

This section aims to describe technologies that were used in the project, why they were selected, and how they are implemented. Technologies that were already presented and explained before are not mentioned here.

4.7.1. Kotlin

Kotlin is a programming language that is widely used in Android development. It is a recommended language by Google and receives support for all the modern features and functionalities.

JetBrains created Kotlin to allow Android development to move forward from legacy Java code and, on top of that, to be able to set modern trends rather than follow them. Before it, there was no such instrument.

Kotlin is designed to be concise, safe, and interoperable with Java. It is able to execute whenever Java can. However, it gained most of its popularity in Android development because Google endorsed it as an official language. Nowadays more than 50% of Android developers are using Kotlin. (JetBrains, 2023)

This section will cover the distinctive basic features of the language, starting with the type system. Kotlin is a statically typed language, just like Java. The compiler is aware of the type of every entity in the code, and it can also infer types when they are not explicitly specified.

Unlike Java, Kotlin does not have primitive types of variables to avoid cluttering the type system. However, under the hood, the compiler still optimizes these variables to its best ability. In Kotlin everything is a subtype of class Any, something similar to Java's Object class. Another major difference here is that classes are not nullable by default and require adding a '?' to their type declaration to allow null types. For example, a nullable Any class would be declared as 'Any?'. The compiler automatically checks for possible errors and highlights them in real time. Additionally, Kotlin provides useful expressions to help with managing null safety. The result of such a design is that NullPointerException is a much rarer occurrence in Kotlin, provided that the developer follows all the compiler's suggestions.

Another interesting change is function types; this allows developers to specify expected function parameters and return types and provide the corresponding lambda later. For example:

```
val func: (String) -> Int = { s -> s.length }
```

The following declaration creates a variable that has a function type with String parameter and Int return value, and then lambda is provided that has a similar signature. This system allows the passing of functions as parameters and the creation of higher-order functions. Meaning functions that accept other functions as parameters.

Kotlin also supports generics, for example:

```
fun <T> zip(a: T, b: T)
```

This function would only accept parameters of similar type T. T can be anything in this case, but with the use of generics, the function signature now requires similarly typed parameters.

Kotlin also provides a way to extend closed classes without changing them, and it is achieved with the use of Extension functions. The example could be:

fun Date.toMyFormat(): String {…}

Where Date is a closed class, and the developer is adding some functionality to it. Under the hood, it would compile a static Utils class containing the specified function. This is a useful feature since, by default, all classes in Kotlin are closed and cannot be extended unless specified otherwise. This feature allows the reduction of Java boilerplate code with identical functionality.

As for classes in Kotlin, they are similar to Java in concept but differ in some more concise syntax. For example, a primary class constructor can be declared just after the class name and provided with default values. Additional constructors can be added later in the class body; they must delegate to the primary constructor, however, since it will be called first during class instantiation. Another distinctive feature is that fields in a class are declared using *val* or *var* keywords. The first one generates a field and getter, while the second one generates a field, getter, and setter. Kotlin will also allow the overriding of default-generated access methods with custom ones.

Kotlin also has some special class types, which are described below:

- Data classes represent a commonly used type of class in Java referred to as POJOs, meaning plain old Java class. Marking a class as *data* will order the compiler to generate equals, hashCode, and toString methods as well as copy methods. This makes works with simple data holder classes much simpler and avoids boilerplate code and associated mistakes in implementing, for example, the equals method.
- Enum classes are similar to Java's and have the same purpose of creating a class with a specified and static set of instances. One distinction, however, is that when using Enums with a when clause, the compiler will require when to be exhaustive, meaning having a branch for each instance or an *else* branch. This proves useful in maintaining a clean codebase and not forgetting to add branches when adding possible values to an Enum.
- Sealed classes can be viewed as Enums for types. This means that a class marked with sealed keywords will be forced to have a fixed set of subtypes, and the compiler will be aware of it when passing through the branches. This will eliminate the need to specify the other branch and allow the developer to specify only actual subclasses in the *when* clause.

One more distinctive feature of Kotlin that I would like to explain is Delegation. This allows to declare that a field in some class is managed by another instance. For example, Kotlin has a built-in *by lazy {…}* delegate. In lambda, some initialization is provided, and it will be called only once during initial access to the field and not before. This is useful when a developer

doesn’t want to trigger too much potentially unnecessary initialization code that may never be accessed. In such a case, *lazy* delegate comes in handy. This is just a single example, and there are more delegates available, as well as an option to create your own delegates. In some cases, this is useful for solving some OOP issues that commonly appear with the usage of inheritance. (Thomas & Hunt, 2020)

Here I have only scratched the tip of an iceberg in terms of demonstrating Kotlin features. A full introduction of the language is not the goal of this work; however, it is beneficial to mention some useful features demonstrating how Kotlin delivers on its promises of concise and safe code. In the following two subsections, I would also like to explain what I consider the most important features of the Kotlin language that have largely shaped modern Android development around themselves: Coroutines and Flows.

4.7.1.1. Kotlin Coroutines

In Kotlin, Coroutines are a solution for multithreading requirements. Documentation refers to them as “lightweight threads”; however, it is an oversimplification. In reality, coroutines can be viewed as blocks of code that can be dispatched to underlying thread pools. (Laurence & Hinchman-Dominguez, 2021)

The reason for implementing Coroutines in the first place is that the previous threading model on Android was tedious to manage, error-prone, and full of boilerplate code. Coroutines provide to write better structured easily understandable, and efficient code without relying on Java threads directly or other lower implementations or any third parties.

The first thing about Coroutines is that they are non-blocking. This means that while the execution of some code block is suspended, for example, for a network request, the thread that is responsible for this coroutine can do other work and come back to the original task when available. This is possible because coroutines are executed on pre-configured thread pools for different purposes. The easiest way to bridge between normal blocking code and a coroutine is to use *runBlocking {...}.* Any code provided inside lambda would be non-blocking. To switch between thread pools, the developer can use withContext(Dispatchers.IO) {…}; this would make any code inside execute on a special thread pool for I/O operations, thus liberating the default Main thread. This is important in the context of Android because an overworked Main thread can cause UI glitches and degrade user experience significantly.

When referring to a structured aspect of coroutines, it is usually meant that coroutines keep a hierarchy tree of child and parent jobs, and when a parent is canceled, its children are also automatically canceled. This is different from default Threads behavior where each new Thread has an independent lifecycle. This behavior is paramount in modern Android development since it allows to automatically clean any excess jobs when user leaves the screen. Otherwise, memory leaks are almost inevitably produced since handling all the started Threads manually can be troublesome. The behavior of other children when one child is canceled is subject to configuration, which also allows for good flexibility. The result of this is that developers can start a lot of different coroutines using *launch* or *async* functions and expect

them to be correctly cancelled when CoroutineScope reaches the end of the lifecycle. Both above-mentioned constructor functions return a Job object, which is handled by a newly started coroutine. It can be used to observe its execution state or cancel the task.

Since I have mentioned CoroutineScope, it is important to explain it. CoroutineScope is the holder of all the information about the execution of a coroutine. It implements the above-mentioned structured concurrency mechanism, is responsible for handling the lifecycle of a coroutine, and contains configuration elements, for example, which ThreadPool uses to execute a coroutine. When creating a new coroutine parent scope is automatically inherited and is merged with parameters in a constructor function. During the merge, parameters provided in the constructor take precedence over parameters from the parent scope. As a result, each child coroutine can be configured slightly differently. However, all children are still handled by parent CoroutineScope.

Another important concept in Coroutines is *suspend* functions. I have mentioned before that a coroutine can be suspended and then resumed sometime later. Kotlin needs these suspending functions to implement the state machine that would handle these stops and starts of execution. Whenever the compiler sees a suspend function modifier, it will add a Continuation parameter to the function signature. Supposing a given function is calling several suspend functions, the main function may be paused at any of these calls. Each child suspend function would be executed through the use of Continuation object and either return immediately or be paused. If the function returns immediately, the state machine passes onto the next block of code. If the function is suspended, the underlying Thread is liberated to do other work until *resumeWith(result)* is not called internally.

As for whether a function will be suspended or not can depend on various things. For example, in the scope of this project, I'm using cached pages of asteroids to avoid calling API every time the user scrolls to the end of the list. When UI requests a new page using a suspend function, it will execute immediately if the page is already cached in memory, or it will pause if it starts a network request and waits for the result. In the second case, the user can see a loader, but the interface is still interactable and responsive since the Main thread is non-blocked by the coroutine waiting for a response from the API.

Using this approach allowed me to build a responsive user interface with complex paging logic and avoid dealing with many classical threading problems directly. This speeds up development and eliminates a lot of potential errors. Overall, the concept of Coroutines is not new; it can be traced back to as early as the 1960s, but in Android, it quickly became the most popular threading solution due to its ease of use, efficiency, and error-free code.

4.7.1.2. Kotlin Flows

Flows are an essential tool in Android development and are widely used in this project in combination with coroutines. They can be described as data streams where each step can be executed asynchronously. In this section, I will provide an introduction to hot flows and leave cold flows out of the scope. The latter was not needed for this project and can be omitted in the

scope of this section. It is also worth mentioning that by default, flows are hot, which means they will not start executing unless a terminal operator is called.

A simple flow can be constructed using the *flow{...}* function. Inside library allows the developer to call the *emit()* function various times for each value that flow is supposed to emit. The resulting type of such a flow constructor would be Flow<T>, where T is any type that the developer chooses. As I already mentioned, flows, by default, are hot and will perform any action without a terminal operator.

Operators are just a Kotlin name for functions that can be called on Flow type. They can be terminal or intermediate. An example of a terminal operator could be *collect{ value -> ...}*. This one needs to be launched in a coroutine scope and simply executes a given lambda over each value that flow emits. If the coroutine scope is canceled before the flow has emitted all its values, then the flow is canceled as well. This allows us to avoid memory leaks and deallocate resources efficiently. There are other terminal operators like *launchIn(coroutineScope:* CoroutineScope*)* that accept scope as a parameter, don't iterate over items, and simply launch the flow.

As for intermediate operators, there are various examples, and Kotlin provides a wide choice of options to build different behaviors. I will only explain the most distinctive operators that are also used in my project. First, a simple one could be:

fun <T> Flow<T>.filter(predicate: (T) -> Boolean): Flow<T>

I'm providing a simplified definition that allows us to easily notice that the filter operator is called on a flow of any type T, takes in a predicate function, and returns the flow of the same type but only contains values that have returned True after executing a predicate function. The simple use case is applying filters for asteroids on the home page. If the user only wants to see dangerous asteroids, this can be easily translated into a corresponding predicate.

Another popular example is:

fun <T,R> Flow<T>.map(transform: (T) -> R):Flow<R>

This one can be called on a flow of type T and will transform each emitted value into type R, returning a flow of the corresponding type. The transform function is responsible for mapping from one type to another. For example, in the project, this is used to transform a model of an asteroid returned by API into another model that can be stored in a database.

It is worth noting that all of intermediate operators can be chained together and possibly amount to a lot of CPU load. By default, all these operations would be executed on a ThreadPool defined as a CoroutineScope in the terminal operator. However, this is often inconvenient since flows are normally collected on the UI layer, and developers should avoid

excessive work on the Main thread. For this purpose, flows have a context-switching mechanism:

fun <T> Flow<T>.flowOn(context: CoroutineContext): Flow<T>

Calling this intermediate operator would result in all of the other operators above executing on a provided ThreadPool, for example, Dispatchers.IO, etc. This allows for a seamless switch of context without worrying about synchronization and interactions between Threads; everything is automatically managed. Another property is that each *flowOn* operator supersedes the previous context, which means a flow could be partially executed on any number of different ThreadPools. Disk operation could be done on Dispatcher.IO, and CPU-intensive work would suit better Dispatchers.Default.

To sum up flows provide many ways of handling data streams in an application while simultaneously handling a lot of work under the hood. They are essential building blocks together with coroutines in modern Android development. Their introduction has made previous libraries, such as RxKotlin or LiveData, pretty much obsolete in the context of Android development on Kotlin. Therefore, usage of these tools was a natural choice for the project and has covered all the needs that were detected during development.

4.7.2. Jetpack Compose

Jetpack Compose has emerged as a new standard library for the UI layer in Android development, designed to replace the older XML view system. Developed by Google as part of the Jetpack toolkit, its primary goal is to streamline the UI setup process by using only Kotlin code, eliminating the need for separate XML and Kotlin files.

In practice, this approach significantly simplifies development. Removing the necessity to bind Kotlin to XML allows for complex list initializations within a single file, reducing the complications of having list elements spread across multiple files, which previously made debugging and maintenance difficult.

Using this new library, applications can create intricate interfaces with animations and reuse various components by maintaining a set of independent UI elements that can be readily utilized for future enhancements.

Implementing Jetpack Compose also requires adherence to certain coding standards to achieve optimal performance. For instance, UI-related functions should be idempotent and free of side effects to maintain a consistent state. This is crucial for the library to optimize performance by skipping the redrawing of UI parts whose input parameters have not changed. Idempotent functions enable Compose to perform these optimizations effectively.

This redrawing process, known as recomposition, is a sophisticated algorithm designed to redraw only the UI elements affected by changed variables, thus maximizing performance. It encourages developers to avoid incorporating I/O or long-running operations within the scope of recomposition, as frequent and efficient execution is necessary to ensure optimal performance.

Overall, utilizing the Compose library ensures that the developed applications are flexible, easily adaptable, and expandable as needed. To summarize, the Compose library offers significant advantages over XML views, facilitating the creation of readable, reusable code that avoids unwanted side effects and is easy to maintain.

4.7.3. Navigation component

Navigation has always been a critical aspect of Android development. While some proprietary solutions have introduced abstraction layers between the application navigation system and direct Android navigation commands, these solutions have not seen widespread adoption. The introduction of an official navigation library has addressed many pressing issues, providing a unified standard for implementation across all applications.

This navigation library has also been integrated with the Compose UI toolkit, becoming the primary solution for app navigation. Although work is ongoing to introduce more advanced features, the library already covers all essential functionality.

At the core of this library is the NavHostController, which serves as the entry point for navigation. All screen transition operations are handled through this class. To process navigation requests, the NavHostController requires a navigation graph where all destinations are registered and entry points for rendering corresponding screens are defined.

Previously, developers could use an interactive navigation graph builder in the XML implementation. However, this feature has been removed from the Compose version, necessitating the construction of the navigation graph using Kotlin. To manage the complexity that can arise when an application has many screens, there are convenient solutions for dividing a single graph into multiple subgraphs based on logical criteria. This approach maintains navigational code manageability while preserving the benefit of a single entry point, as all subgraphs are registered in the main graph for proper setup.

The use of the NavHostController has facilitated efficient navigation within the application, allowing adjustments to business needs without directly issuing commands to the Android system, thereby avoiding potential complications and drawbacks.

4.7.4. Hilt

Hilt is a library used for dependency injection (DI), a programming technique to supply classes in an application's hierarchy with their necessary dependencies (Android Developers, 2022). For instance, classes implementing use cases require instances of data sources to function, but ideally, they should only be aware of the interfaces of these data sources, not their actual implementations. Without DI, use case classes would need to instantiate data sources themselves, creating unwanted dependencies on those classes. To avoid this, dependencies are provided as constructor parameters, ensuring use case classes only interact with interfaces, not concrete implementations.

Several methods exist for implementing DI. One method is manual implementation, where the developer must create and manage all the code for supplying class instances. In a growing codebase, this can become increasingly problematic as dependency graphs expand and become unmanageable.

Another approach used in Android is requesting needed dependencies from other classes. For example, the Context class has the *getSystemService()* function to provide various system services to the application. While convenient for Android APIs, this manual method still faces the complexity issues of the first solution.

A more efficient method is using a dedicated DI library. Hilt, the latest recommended library for DI in Android, simplifies the process. It integrates with Android Studio, helping developers trace dependencies through the IDE's user interface.

Overall, the Hilt library offers straightforward tools for managing the complexities of DI, making it an essential tool for developing modern Android applications.

4.7.5. Gradle convention plugins

Gradle convention plugins are not a library but rather a tool that is part of Gradle. It helps to improve the situation by maintaining the multimodule project. As I have already explained in section 4.3, adding a lot of modules to the project adds a lot of boilerplate code. The problem severity gets worse with more modules added. Additionally, apart from just having to copy and paste heavy chunks of configuration code for each module, it gets difficult to introduce changes since the same update needs to be done in each and every related module.

In this project I have relied on these Gradle convention plugins to define configuration for each type of module and share it across the whole project. Overall, convention plugins can provide the following advantages:

- Sharing common configuration logic between modules.
- Making project maintenance an easier task.
- Improving readability of modules configuration.
- Facilitating automation.

```kotlin
internal fun Project.configureKotlinAndroid(
    commonExtension: CommonExtension<*, *, *, *, *, *>,
) {
    commonExtension.apply {
        compileSdk = 34

        defaultConfig {
            minSdk = 21
        }

        compileOptions {
            // Up to Java 11 APIs are available through desugaring
            // https://developer.android.com/studio/write/java11-minimal-support-table
            sourceCompatibility = JavaVersion.VERSION_11
            targetCompatibility = JavaVersion.VERSION_11
            isCoreLibraryDesugaringEnabled = true
        }
    }

    configureKotlin<KotlinAndroidProjectExtension>()

    dependencies {
        add("coreLibraryDesugaring", libs.findLibrary("android.desugarJdkLibs").get())
    }
}
```

Figure 12 Example of Gradle convention plugin

In Figure 12, the reader can observe an example of a plugin that configures the usage of Kotlin language. Normally this configuration would need to be repeated over every module using Kotlin. With this feature, I was able to define the configuration once and use it for all modules. This made the multi-modular solution a viable choice in terms of required maintenance effort. If I ever need to update some parameter in this configuration, it will be enough to update in once in this file and it will automatically propagate through all the modules.

In conclusion, Gradle convention plugins are a very powerful tool for multinodular projects and allow the support of better architecture with minimal overhead.

4.7.6. Sceneview

Sceneview is a helper library for creating 3D and AR scenes based on ARCore and Google filament. It is based on a now deprecated open-sourced Google library, Sceneform, which served the same purpose. After Google dropped support for it, Sceneview is now maintained by the community.

One of its biggest advantages is that it provides integration with Jetpack Compose, which means that developers can create complex AR scenes without building extensive bridges into legacy code but simply add desired functionality to the modern style code.

The use of this library in the project enabled me to create 3D and AR scenes faster, make them more complex, and ultimately introduce more functionalities by taking advantage of integrations with Jetpack Compose.

4.7.7. Android Studio

The IDE used for developing the solution is Android Studio. Based on IntelliJ IDEA, Android Studio is distributed free of charge and offers an advanced set of features specifically designed for Android development.

Android Studio supports both Java and Kotlin languages, allowing developers to translate Java code into Kotlin. It integrates with version control systems (VCS) and can be customized with a wide array of available plugins.

A key feature of Android Studio is its emulator, which can be configured to simulate a wide range of physical devices and Android OS versions. Additionally, Android Studio provides special device inspectors, enabling developers to view database contents, network requests, and resource usage. These tools help analyze the performance of applications and identify potential bottlenecks or errors.

Android Studio also includes Gradle, a build automation tool responsible for managing dependencies and building APKs. Gradle handles all necessary signing steps for submitting an APK to the Google Play Market.

Overall, Android Studio is a vital tool for Android development, offering comprehensive functionality to efficiently develop applications.

4.7.8. Version Control System

GitHub was selected as the version control system for this project. It enables the creation of public repositories for free, allowing for code storage and version history maintenance. Additionally, GitHub supports the configuration of automated scripts to run Gradle for APK assembly, execute tests, and distribute application versions.

Using GitHub provides several advantages:

- Version Control: It maintains a detailed history of changes, facilitating collaboration and tracking modifications over time.
- Collaboration: Multiple developers can work on the same project simultaneously, using features like branches and pull requests to manage changes and integrate contributions smoothly.
- Continuous Integration: GitHub Actions can be set up to automate workflows, such as building the application, running tests, and deploying new versions.
- Code Review: Built-in tools for code review help maintain code quality and facilitate peer reviews before merging changes into the main branch.
- Backup and Accessibility: Storing code in a GitHub repository ensures it is securely backed up and accessible from anywhere.

Overall, GitHub provides a robust platform for managing code, enhancing collaboration, and automating development workflows, making it an essential tool for modern software development.

4.7.9. Android

Android is an operating system developed by Google for mobile devices like phones, tablets, TVs, and wearables. Built on Linux, it is available for free. Many manufacturers opt to supply their devices with customized versions of Android, a practice allowed by its open distribution policy. However, this can lead to security concerns, such as delayed critical updates and issues unique to specific manufacturers' versions.

This variability poses challenges for developers, as the range of devices and OS versions is much broader compared to IOS. To ensure proper functionality across different devices, it is advisable to test device-specific features on various devices.

Android's architecture provides an abstraction layer over hardware, enabling smooth communication with the device. The OS also ensures applications have the necessary permissions to access hardware components. Android includes native C++ libraries and the Dalvik Virtual Machine (DVM), an optimized version of the Java Virtual Machine (JVM) for mobile devices.

Despite the potential issues, Android remains one of the most accessible and secure operating systems, provided users keep their OS versions up-to-date and follow basic security practices. Developing applications for Android allows for a wide user reach, maximizing the potential audience.

5. User manual

This section aims to present a user manual that would explain how to use the application in general and give an overview of how all the use cases have been implemented.

1. Basic Functionality

At first, I will showcase some basic functionality so that It is clear how to use the app in general.

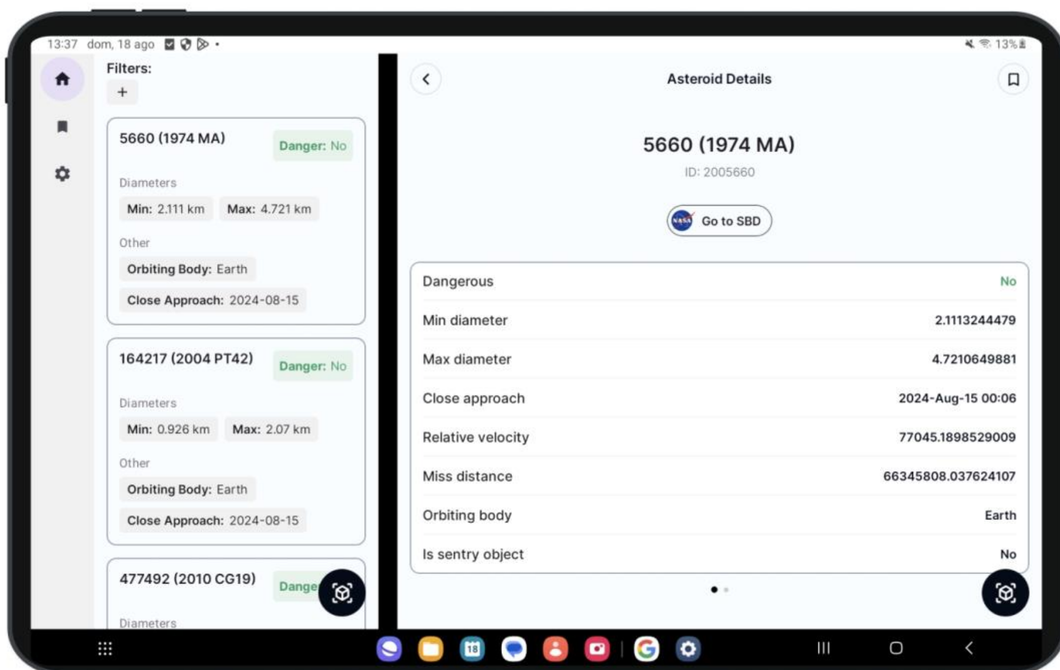

Figure 13 Home screen on tablet device

In Figure 13, the reader can observe the Home screen open on a tablet device. I have specifically selected a larger device for this figure since it showcases an adaptive layout for tablets. Instead of a naïve approach of just making things wider on the tablet screen, I organized the space efficiently, and the user can get more information than on a mobile device. This home screen features a pane with a list of asteroids and another pane with asteroid details. On mobile devices, only one pane can be selected at a time, but the tablet allows it to show both panes at once. This makes the application much more usable on tablet devices and allows users to interact with it efficiently.

Asteroids list pane presents the user with a list of asteroids and also controls to open the Solar System visualization in the bottom right corner and another button to add filters to the asteroid selection. The asteroid details page presents all the detailed asteroid info and as well a

button to open asteroid orbit visualization. In the top left corner, users can find navigation controls that allow them to switch between the Home screen, bookmarked asteroids, and settings.

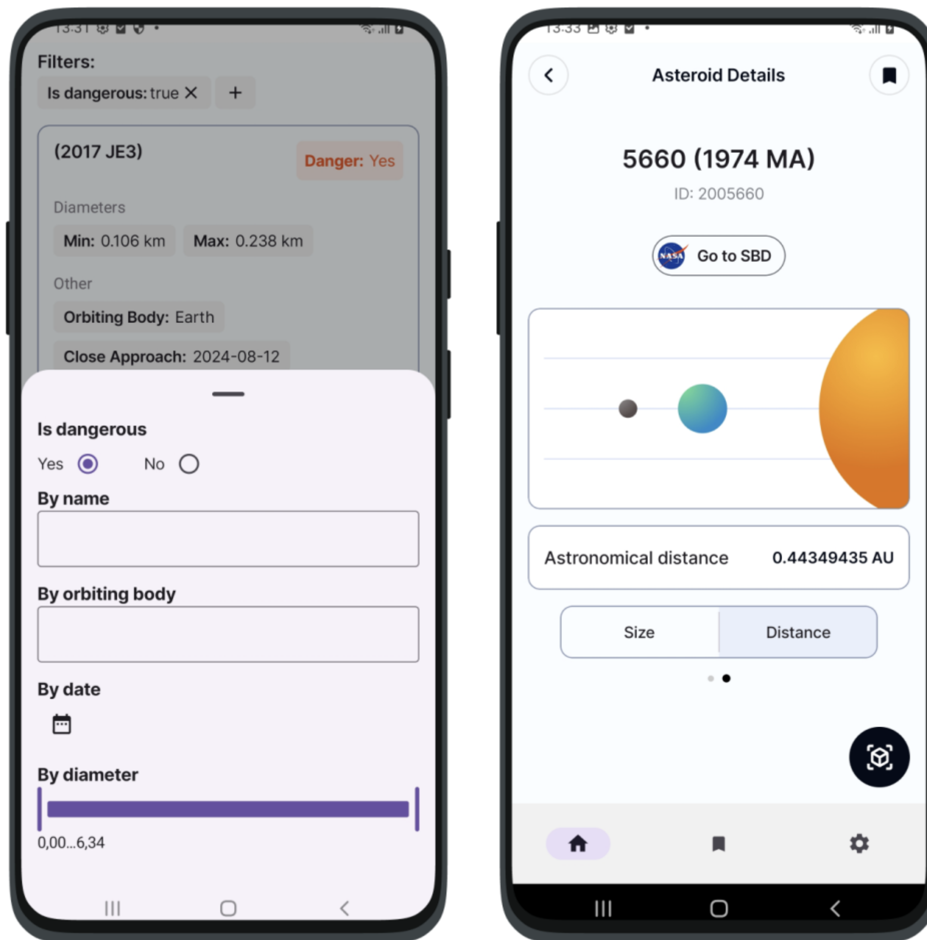

Figure 14 Filtering and Distance Comparison screens

In Continuation, figure 14 presents filtering controls that allow the user to select which asteroids to show in the list based on various parameters and a chart that visualizes to the user comparative distances between a selected asteroid, Earth, and the Sun. Filtering popup can be accessed by clicking the plus button on the Home screen, and any selected filters immediately apply and take effect. They can be removed by clicking the cross button on the corresponding filter chip. The comparison chart can be accessed by swiping to the left on the asteroid details screen.

2. AR and 3D Functionality

The following figures will showcase the AR and 3D functionality of the application.

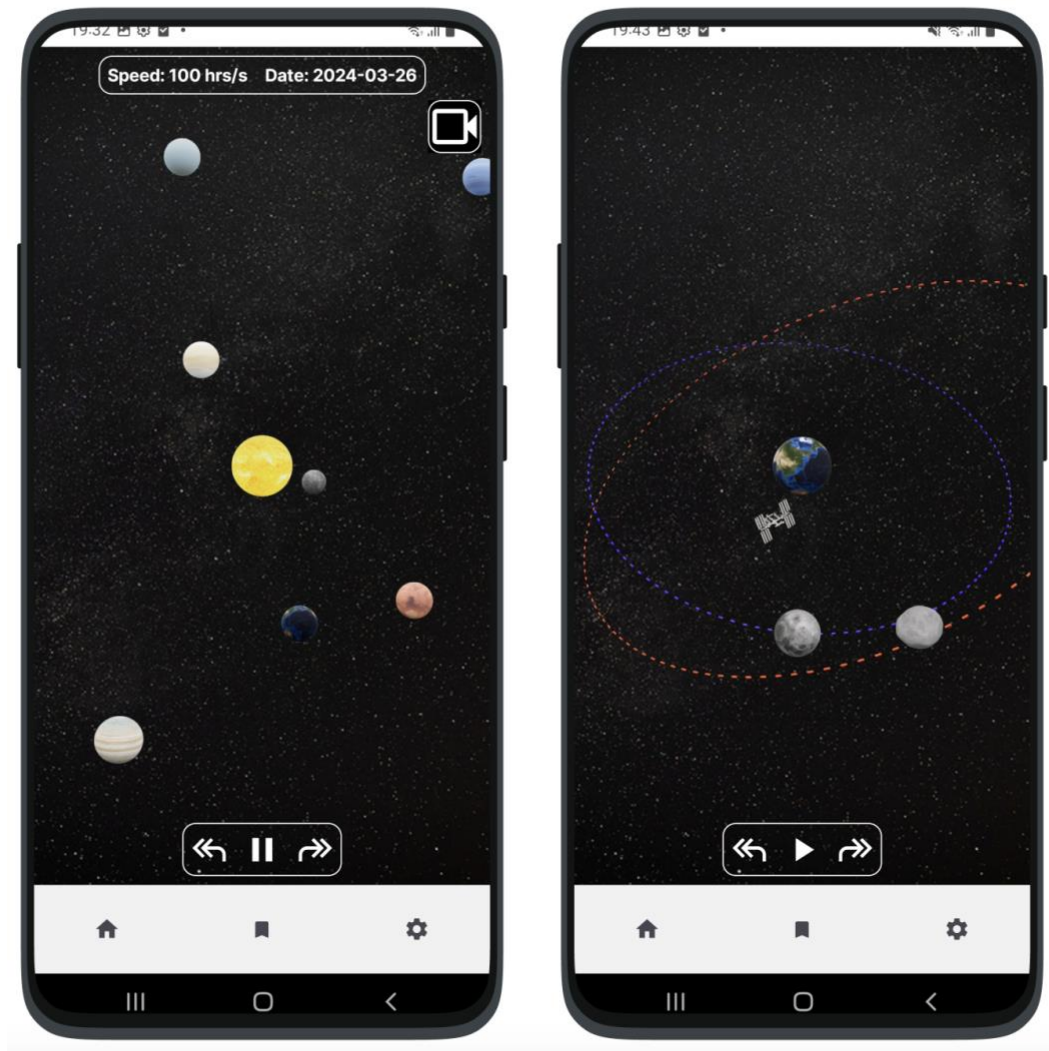

Figure 15 3D models of the Solar System and asteroid's orbit

Figure 15 presents the fallback scenario when the user's device doesn't have AR support and the application is just drawing 3D scenes instead. To the left, there is a solar model; it has controls to manipulate the planet's rotation speed; it can also be done with horizontal swipes. Additionally, there is a camera control that slides out and lets users change the angle and perspective of the camera. To the right, there is a model representing an asteroid orbit around the Earth; it also has the Moon and ISS. In this case, the user can control the camera freely and inspect the scene from any direction.

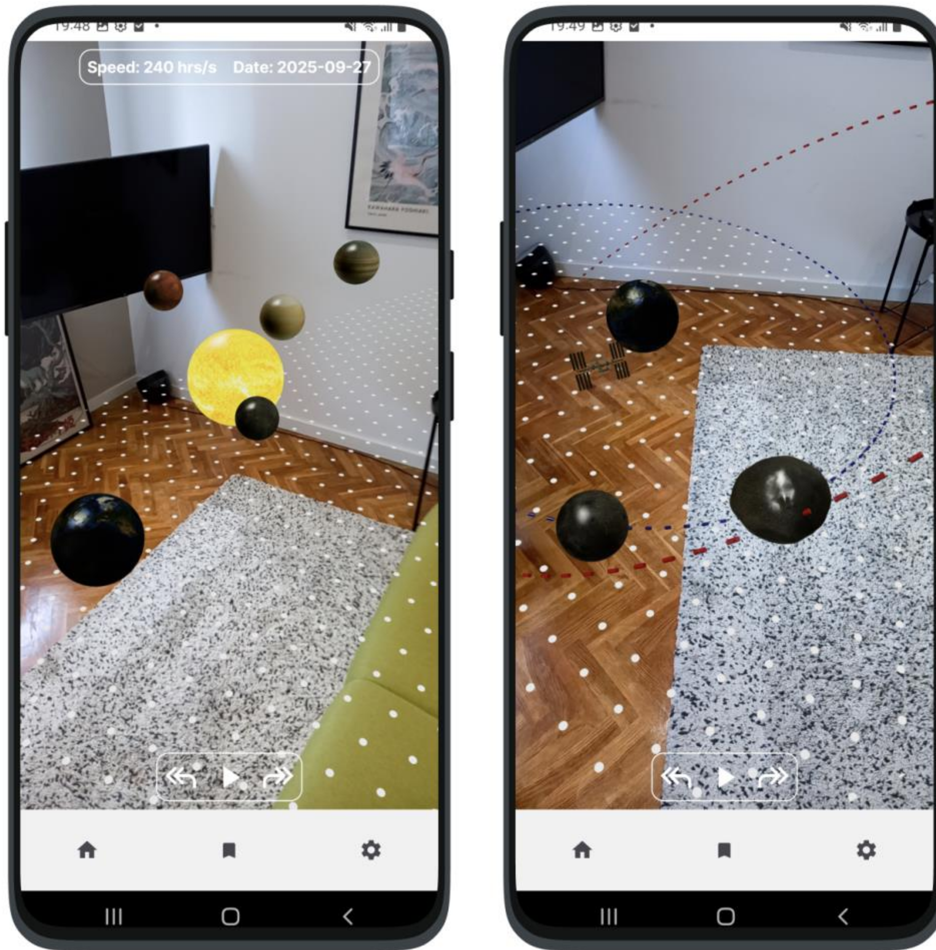

Figure 16 AR models of the Solar System and asteroid's orbit

Finally, figure 16 presents an example of AR scenes that users can find in the application. Their functionality and structure largely correspond with their respective 3D counterparts. However, there are some differences present. Users can see a dotted plane that represents ARCore's understanding of the surface that it is drawing upon. With more movement this dotted plane can get updated since ARCore keeps evaluating the scene and is able to refine its understanding with more input data. Also, camera controls are disabled in AR mode since the user can inspect the models from any side by moving the device around. The rendered scene is anchored to where the user first clicked when a plane was detected, meaning that when the user moves around, 3D models stay in the same place. This allows for natural interactions like looking behind the model or over it.

To sum up, the application presents a simple and efficient interface that fits a lot of potentially interesting information to the user. The main functionality of AR visualization is very accessible, and just one click away. In case the device doesn't support AR scenarios, there is a fallback scenario that covers the rest of the users. The presented 3D scenes showcase models of the Solar System and asteroid's orbit around the Earth which makes for simple and understandable visual representation.

6. Future work

In this section, I will present some ideas for possible future development of the project. It will be divided into three categories: general improvements and expansions achievable with the current technology stack, specific ARCore features that could be implemented on top of the existing ones, and long-term improvements alongside possible development directions.

First of all, some obvious enhancements to the application could be made entirely within the current technological stack. It could be beneficial to add more detailed scenes for each planet and to fill them with more objects, such as satellites and moons. Currently, only Earth-detailed scenes, which feature the Moon, ISS, and an asteroid of choice, are available. However, adding similar scenes for each planet seems a logical addition. Some planets have many moons, meaning an exact 3D scene could be difficult to construct, but it would be possible to set some upper limit and let the user decide which moons are to be visualized. In case there is some ongoing NASA mission on the planet, for example, Mars Exploration Rovers, it is also possible to indicate its location and let users access some data about the mission. Overall, these improvements could enrich the user experience and make the application more engaging while also giving users a reason to visit the app and see possible updates. These features could be implemented the same way the rest of the app is done, which is quite feasible but does not have a lot of value as technical improvements.

Second, the promising feature could be enabling cloud anchors, allowing multiple users to share the same AR experience and potentially collaborate. ARCore allows the hosting of an anchor in the cloud after it was locally created; it can be persisted and accessed later. Next, the host can share the anchor identifier with other users, which can each get that same anchor from the cloud. ARCore also is doing synchronization in real time which means if one user interacts with the model, it will be reflected on each device observing the same model. With this feature, a guide or a teacher could potentially present the model of the Solar System to a group of people who could see changes or interactions performed by the host user. This would allow us to highlight some details and provide better guidance with such an engaging presentation. Adding this functionality would not require a complete overhaul of the system, but the management of cloud anchors and their sharing between users would need to be set up. It could be done with some out-of-the-box solutions such as Firebase, or a custom server could be developed.

Finally, apart from the previously mentioned features, it could be beneficial to make this application available on other platforms as well. However, this is a more difficult task. While ARCore, in general, is available for different platforms such as IOS, Unity, and Web, it would only be similar in core concepts, and most of the code would need to be rewritten entirely. In the case of IOS and the Web, the benefit would be a larger audience for the app, which is obviously good. In the case of some wearable devices, their support could provide users with a more engaging experience, better tracking, and more refined interactions. While this multiplatform scenario seems beneficial, in my opinion, it would be better to continue expanding the Android project further. The direction and goals of the development are clear, and the application already has a stable and active user base.

7. Conclusions

In this section, I will present the conclusion that can be made after completing the development of the project and relate it to the initial objectives.

During the development of this work, I have produced an AR-capable application that boasts not only complex 3D scenes but is also built with modern Android architecture in mind. Modularity ensures that the code base is scalable, and more features can be added seamlessly without affecting the existing ones. Additionally, using Gradle convention plugins allows adding new modules much easier, without duplicating large blocks of code. On top of that, the application has a networking model implemented using a pagination approach with local cache. This ensures that the device doesn't waste resources on excessive queries and loads only needed pages when they are absent from cache storage. On the side of the user interface, I have used an infinite scroll approach, which gives the user a modern interface that allows for the whole pagination system to function seamlessly under the hood.

The first AR feature that users can access is a 3D model of the Solar System. I was able to build both an accurate realistic representation and a cinematic one. This allows us to choose between the two and puts it into perspective since it is easier to notice inaccuracies in the traditional, cinematic model when comparing it directly to the more scientific model. The data for planets' orbits is statically stored in the application; there was no need for an API here; however, if certain future expansions are implemented, more objects with dynamic data may be added. The scene itself is interactive and animated, aimed to increase engagement and capture the user's interest. In case the user is using an older device that doesn't support ARCore, the application will automatically fall back on just rendering the same scene but only in 3D, without the AR experience. In this case, the user has predefined camera positions to see the models from different perspectives. However, there is no free movement in this fallback mode.

The second AR feature is the scene that presents Earth and a rotating asteroid; in this case, it was beneficial to add an asteroid trajectory because the resulting orbits tend to differ a lot. This is beneficial for educational purposes since it lets users evaluate how close or how far an asteroid can get from Earth while rotating in its orbit. On top of that, this scene features the Moon and ISS; they also have their trajectories drawn as a part of the scene. The data for this scene is dynamic and directly benefits from the caching solution. Each asteroid has a different orbit, which the application gets from NASA API. This makes it interesting to observe the orbit of some dangerous asteroids and improves user engagement.

During the development of this project, I have acquired valuable experience with designing and developing AR applications. I learned to solve various related problems and account for small details that are normally not visible initially but become apparent during development. I have made my best effort to ensure that the application runs on the widest range of Android devices possible by including fallback scenarios when AR functions are unavailable.

On top of that, while conducting the research for suitable themes for AR applications and corresponding technologies, I have learned more about what can be done with Augmented Reality in general and utilized those ideas to improve this project and to build a more complex AR experience.

In conclusion, this project resulted in an AR application with different complex scenes and interactive elements that have potential applications in education. During the development I have acquired practical knowledge about how Augmented Reality works in modern applications and successfully applied it drive the project to its result.

8. Bibliography

Robert, M. C., 2018. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* s.l.:Prentice Hall.

Bittner, K. & Spence, I., 2002. *Use Case Modelling.* s.l.:Addison-Weasley Publishing Company.

Gamma, E., Vlissides, J., Helm, R. & Johnson, R., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.:Addison-Wesley Publishing Company.

Laurence, P. O. & Hinchman-Dominguez, A., 2021. *Programming Android with Kotlin.* 1st ed. s.l.:O'Reilly Media Inc..

Thomas, D. & Hunt, A., 2020. *The Pragmatic Programmer.* 2nd ed. s.l.:Pearson Education, Inc..

Linowes, J., 2018. *Augmented Reality with ARCore: Create AR apps for Android using Google's ARCore.* 1st ed. s.l.:Packt Publishing.

Boland, M., 2021. *AR Insider.* [Online] Available at: https://arinsider.co/2021/07/12/are-90-of-smartphones-ar-ready/ [Accessed 3 August 2024].

Natali, S., 2024. *Medium.* [Online] Available at: https://proandroiddev.com/adaptive-compose-layouts-86b7f1e51338 [Accessed 10 April 2024].

JetBrains, 2023. *Kotlin.* [Online] Available at: https://kotlinlang.org/docs/android-overview.html [Accessed 6 March 2024].

Android Developers, 2022. *Guide to Android app modularization.* [Online] Available at: https://developer.android.com/topic/modularization [Accessed 4 February 2023].