Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Augmented Reality für die Visualisierung von Windparks

Augmented Reality for the Visualization of Wind Farms

Masterarbeit
Informatik

12 2022

Vorgelegt von Presented by	Jiani Qu Matrikelnummer: 423889 jiani.qu@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. Thomas Noll Lehr- und Forschungsgebiet: Softwaremodellierung und Verifikation RWTH Aachen University
Betreuer Supervisor	Dr. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Contents

1 Introduction

1.1 Motivation

Due to the negative impact of fossil fuel combustion on the environment and its limited availability and non-renewable nature, switching from fossil fuels to other energy sources has become a necessity for many countries in order to sustain the increasing energy demand [24, 20].

Germany has long been the global pioneer in applying renewable energy and environmental technologies and remains focused toward its energy transition to become more climate-friendly and less dependant on fossil fuels as reported by International Trade Ministration[1]. Today, renewable energy sources are among the most important sources of electricity of Germany and the share of renewables in electricity consumption has experienced a steady growth in the past two decades according to German Federal Ministry for Economic Affairs and Energy[2], from about only 6% in 2000 to about 45% in 2020, of which wind energy accounts for the largest contribution of about a half.

However, in 2021, for the first time since 1997, the portion of renewable energy in terms of electricity consumption has not only not increased but decreased by 4.1 percent below that of the previous year (45.2 percent in 2020). In addition to the lower electricity generation from wind turbines due to the weather, the decline is also due to the overall increase in electricity consumption (by 2.4 percent compared to the previous year) in 2021. All other renewable energy sources remained around the same level as previous year and were unable to compensate for the minus in wind energy, which is responsible for the entire decline.

On top of increasing energy demands, Ukraine crisis also calls for energy independence more urgently. Currently, Germany has the goal to generate 65 percent of electricity consumption through renewables by 2030 as stated in the Renewable Energy Sources Act[3], which might be updated soon in July to enable a full supply from renewables by 2035 by a new package of laws. Due to the leading share of wind energy among all renewable energy sources, it plays an important role in expanding renewables and its capacity is expected to double to 110 gigawatts by 2030[4].

In spite of increasing public awareness on climate, the social acceptance of wind farm facilities still poses potential challenges to the further expansion of renewable energy as pointed out in a research by European Commission [8]. Two of the key factors influencing social acceptance are, for example, physical characteristics such as the size

[1] https://www.trade.gov/energy-resource-guide-germany-renewable-energy
[2] https://www.bmwk.de/Redaktion/EN/Dossier/renewable-energy.html
[3] https://www.erneuerbare-energien.de/EE/Redaktion/DE/Pressemitteilungen/2020/20201228-altmaier-eeg-tritt-wie-geplant-zum-1-januar-2021-in-kraft-zentraler-schritt-fuer-die-energiewende.html
[4] https://www.dw.com/en/ukraine-crisis-forces-germany-to-change-course-on-energy/a-60968585

and proximity of the turbines and consequently negative impacts such as noise level and visual pollution etc., and the perceived fairness in the decision-making process, namely transparency and openness with regard to the availability and quality of information provided.

In order for individuals and neighborhoods under impact of planned wind farms to form opinions as unbiased as possible in hopes of increasing social acceptance and supporting expansion of renewables, in this thesis, we aim to convey transparent information of the visual aspects of wind farms using augmented reality (AR) technologies. For this purpose, we ask two research questions:

- How should an AR application for wind farms visualization look like?
- How should such an application be evaluated?

In order to answer these question, we set out to design, develop and evaluate an application to show users a realistic 3D virtual visualization of the planned wind farm according to the exact coordinates of wind turbines, integrating to the display of real-world view as a so-called augmentation. In this way, people will be able to experience the visual impacts of planned wind farms beforehand in a very exact way.

Currently, AR applications are able to be developed on AR headsets, smartphones, and even browsers. We choose smartphones as the target device for our application for the sake of better usability and accessibility, as in a real-world use-case, users ideally should be able to carry their personal devices around and perceive the visuals of wind turbines throughout the day.

1.2 Related Work

In this section, we introduce some some background information about augmented reality, its history and current state of research that is most relevant to our project.

The term ”augmented reality” (AR) refers to a set of technologies that allows the visual perception of real-world environments to be enriched by computer-generated elements [4, 28]. With the rapid advances over the past few decades in computational power, hardware and research areas such as computer vision and computer graphics, AR has evolved from large indoor interactive facilities that failed to render real-time augmentations robustly, to small mobile devices that can be moved around freely and access digital contents almost everywhere [6, 15].

One of the earliest research prototypes of mobile augmented reality (MAR) was a *touring machine system* that was designed in 1993 for road navigation on a campus. However, due to the restricted technology at that time, the whole device has to be carried around in a huge backpack and could not understand the physical environment correctly [1]. Nowadays, not only are most AR devices wearable that can aid an individual's activities without restricting with the user's movements [29], but also many

smartphones are able to support AR contents, which makes AR content more accessible to all.

Among the main application areas of AR [4, 3], outdoor application is the category that our project falls into. Apart from applications that only adds visual hints or small interactive markers for the users to tap on to get information, there has been also more complex and hardware-intensive ones being developed. CityViewAR was for example such a mobile outdoor AR application that reproduces destructed historical buildings virtually on-site in a city-scale[14], the virtual 3D buildings are placed where they once were with help of mobile phone's camera and screen, so to say. This is similar to what we want to achieve. However, the paper pointed out that the developers encountered difficulties accurately registering virtual structures to the real world due to errors produced by the GPS sensors of at least 10-20 meters. The issue gets worse if the user tries to look at a nearby virtual building at a distance within the error range. The GPS error was deemed unavoidable, so Lee et al. provided a solution for improving usability by allowing users to use a suggested "AR viewpoint" as their location. However, this falls into the category of VR, as the entire scene is then virtual. It was also mentioned that correct visual occlusion is not implemented in this project.

In another project that was presented at the *WindEurope Summit 2016*, Grassi and Klein also took the 3D AR approach to improve social acceptance and public participation in wind farm projects, but their paper focused more on the planning aspect rather than the technical details for realising the AR contents. In fact, the technical implementation for the 3D AR visualization was hardly mentioned in the paper.

1.3 Contribution

Although Big Tech companies such as Google and Apple provide state-of-the-art AR development kits (ARCore and ARKit) for mobile end, AR visualization of wind farms still poses quite a few challenges even using these tools. According to our investigation, most existing AR apps have use cases where visualized contents are at a close distance within a couple of meters (more details in Chapter 2), whereas in our case, reasonably precise visualization of wind turbines from hundred meters or even kilometers away is expected. Consequently, depth estimation and obstacle occlusion can be difficult. Thus, we have to first find out the capability of current technologies for visualizing AR contents at great distances. We also hope to explore technical possibilities of integrating computer vision or machine learning methods into existing tools if not provided.

Our main focus of this work is on software development. We aim to improve the user experience for the app instead of only developing a mere experimental prototype, which includes, for example, smooth performance, map or directions for the wind farm site etc.

To summarize, we contribute by creating a ready-to-use AR app that realistically visualizes planned wind farms at real-world locations, which is ready to be extended for more advanced occlusion functionality and is integrated with an existing Ionic app. Our goal is to increase social acceptance of wind farm projects by providing clear and accurate information and increasing citizen involvement.

1.4 Outline

First and foremost, it is of utter importance to decide, which AR technologies are capable and suitable for developing such an app. This is a very crucial decision as it determines if our software solution will be successful. In Chapter 2, we evaluate the estimated performance of different AR technologies and take core requirements into consideration and finally choose Unity and its AR framework AR Foundation as our main development tool.

Then, in Chapter 3, we describe the process of developing our application systematically by consciously following software engineering (SE) standards. We adopt the prototyping SE model by first identifying requirements, putting forward a high fidelity UI mockup as our accepted prototype, and then the implementation is reported in detail in Chapter 4.

In Chapter 5, we evaluate the core functionality of our solution, namely AR wind farm placements, by conducting both quantitative and qualitative tests. Finally we conclude this thesis in Chapter 6 by providing a sum-up of this thesis, discussing the challenges and difficulties encountered during the project, and giving future work suggestions.

2 Choice of Technologies

In the previous chapter, we already explained that our wind farm AR application should be developed for smartphones because of better accessibility, since most people own a mobile phone today, whereas the same can't be expected for AR glasses or headsets. However, the choice of technologies is also crucial for multiple reasons such as compatibility, device support and development overhead etc.

In this chapter, we first share our investigation into a number of suitable AR frameworks for mobile end in Section 2.1, and in Section 2.2, we give a detailed overview of our chosen development platform Unity and its AR framework AR Foundation.

2.1 Evaluation of Technologies

Cao et al. conducted an extensive research into all existing mobile augmented reality (MAR) frameworks in 2021, investigating their platform supports, tracking abilities, some key features, sensors etc. Table 1 shows an excerpt of all MAR frameworks that we will discuss in detail next.

	ARCore	ARKit	AR.js	WebXR	AR Foundation
Platform support					
Android	✓	✗	✓	✓	✓
iOS	O	✓	✓	✓	✓
HTML5	✗	✗	✓	✓	✓
Unity	✓	✓	✗	✗	✓
Features					
Occlusion	✓	✓	✗	✓	✓
Sensors					
Camera	✓	✓	✓	✓	✓
LiDAR	✗	✓	✗	O	✓
IMU	✓	✓	✗	O	✓
GPS	✓	✓	✓	O	✓

Table 1: Comparison of several MAR frameworks. Features and functions that are not fully supported are marked with "O". For example, only a part of ARCore features are supported on iOS platform. As for the WebXR framework, the support for LiDAR, IMU and GPS sensors is determined by the underlying system or hardware, because WebXR works in a fashion as an interface.

There are several core features required for our application. Naturally, in order to generate virtual wind turbines on top of real-world scenes, the smartphone must have a camera. In addition, a GPS sensor is also necessary, as we need to generate the visualization based on the coordinates of the planned turbines, the position of the users holding their mobile devices, and the direction in which they are pointing their camera. This allows us to calculate the correct distance between the users and the turbines and adjust the size and position of the turbines accordingly. The framework that we choose should, of course, have support for these hardware sensors.

In addition to the basic requirements, it is important that the AR application can be integrated with an existing app that provides people with other wind farm-related information, such as a 2D map view of turbine locations, a 2D visualization of noise propagation and shadow casting, and a Q&A section. This existing app was developed

Browser	Compatibility
Chrome	✓
Edge	✓
Firefox	✗
Internet Explorer	✗
Opera	✓
Safari	✗
Chrome Android	✓
Firefox for Android	✗
Opera Android	✓
Safari on iOS	✗
Samsung Internet	✓
WebView Android	✗

Table 2: WebXR Browser compatibility[5]. Among the mainstream and more popular browsers on both desktop and mobile ends, WebXR does not support Firefox or Apple's Safari at all.

using the Ionic framework[6], which allows for the creation of apps for Android, iOS, and the web from a single code base using web technologies like HTML, CSS, and JavaScript. It is also crucial that our application can be extended to support machine learning inference for object occlusion. Other desirable features include built-in obstacle occlusion functionality in the framework, simplicity of use to reduce development overhead, and the ability to function offline, as wind farm areas are often poorly covered by 3G/4G internet.

2.1.1 AR.js

Among all the available AR frameworks, a few stand out as suitable for our requirements. AR.js[7] is a lightweight library for AR on the web that is fully based on JavaScript, which means that it does not require installation and can run on any device with a mainstream browser that supports WebGL and WebRTC. This makes it very easy to integrate into the Ionic app. AR.js is easy to use and allows for rapid prototyping, and has been adopted in several research projects[21, 22, 26, 16], but most of them use its more stable feature: marker tracking, where AR content is displayed when a marker is detected. However, this is not applicable in our case.

Since we were unable to find much information about AR.js's performance of its location-based features, we tested them ourselves by building a dummy prototype that placed a 3D cube at a specific geographical location. Unfortunately, the results were

[5]https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API#browser_compatibility
[6]https://ionicframework.com/docs/
[7]https://ar-js-org.github.io/AR.js-Docs/

rather unsatisfactory. AR.js was not performant enough, as the AR content displayed laggy and jumpy transitions when the device was moved towards or away from the content. More specifically, the content changed in size and position abruptly instead of continuously each time the location of the device changed by about 1-2 meters. In addition to this, AR.js still has many open issues related to its location-based features on their GitHub project[8] and does not support obstacle occlusion. Taking all these factors into consideration, we decided against using the AR.js library.

2.1.2 WebXR

WebXR[9] is another web solution for immersive experiences that provides compatible web browsers with access to not only AR, but also VR content. It is created by the Immersive Web Community Group, which has contributors from some BigTechs such as Google, Microsoft, Mozilla etc. WebXR is, at its core, an API that serves as an interface between web XR content and the devices on which it is displayed [6], but the functionalities and features vary between different browsers and platforms (Table 2 shows the browser compatibility of WebXR.). For example, AR content on Google Chrome browser of Android devices are powered by Google's ARCore[10]; whereas on Microsoft Edge browser of Windows Mixed Reality simulator or Hololens 2[11], AR content can only be created by a collection of JavaScript libraries such as A-Frame[12], BabylonJS[13], three.js[14] and WebGL[15]; and it is not supported by iOS devices at all[16]. This very nature of WebXR indicates too much development overhead if we want to launch our app on as many devices as possible, and that iOS, a major smartphone platform, has to be left out, which is why WebXR is also taken out of our consideration.

2.1.3 ARCore and ARKit

ARCore[17] is Google's framework for building augmented reality experiences on different platforms and devices. Its motion tracking technology identifies interesting points through the camera, and tracks movements of those points over time, which is then, along with information from the phone's inertial sensors (IMU), used for determining the position, orientation and velocity of the phone as it moves across space. This

[8] https://github.com/AR-js-org/AR.js/issues
[9] https://www.w3.org/TR/webxr/
[10] https://developers.google.com/ar/develop?hl=sv
[11] https://docs.microsoft.com/en-us/hololens/hololens2-hardware
[12] http://aframe.io/
[13] http://www.babylonjs.com/
[14] https://threejs.org/
[15] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[16] https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API#browser_compatibility
[17] https://developers.google.com/ar

is exactly what AR.js fails to do. In another prototype that we built using ARCore that is similar to the AR.js one, we confirmed the excellent quality of the geospatial AR content rendering, as the virtual content looks really "attached" to its coordinates stably in the moving camera. Positioning AR content in real-world coordinates has to utilize the Geospatial API, which is backed by Google Earth and Google Maps to achieve greater precision, so the internet connection is a must.

Some other features of ARCore include, for example, environmental understanding, surface detection, light estimation etc. What interests us most is its built-in capability of depth estimation and consequently obstacle occlusion. According to the official documentation, the algorithm can get robust, accurate depth estimations up to 65 meters away, which is still far nearer than in our use cases. A possible approach would be to use Google's ML Kit[18] along with ARCore to enhance object occlusion.

Apple's ARKit[19] has similar features as ARCore and takes depth detection to another level on some of its devices with LiDAR Scanner (laser imaging, detection, and ranging). It is also possible to integrate machine learning into ARKit with Apple's Core ML[20]. However, ARCore and ARKit's great power do depend on the device's hardware and only rather new models support these frameworks. Other than that, as shown in Table 1, ARKit supports only iOS devices, and although ARCore can also be used on the iOS platform, not all features and functions are supported. This means, we still need to develop apps for Android and iOS platforms separately, which doubles the workload.

2.1.4 AR Foundation

AR Foundation[21] is a Unity AR framework that supports AR content in a multi-platform way that includes ARKit, ARCore, HoloLens etc. It presents an interface that allows deploying one single application across multiple devices, in other words, we would only need to write one version of Unity code, that will be implemented concretely and automatically on the target platforms. i.e. Android and iOS, and most of ARCore and ARKit's features are supported by AR Foundation. This looks very promising for us because of largely reduced development overhead. Apart from this, according to our research, Unity is a good choice for developing interactive 3D applications, as it is in its core a game engine, while using Android Studio, a native android application development platform, for example, would be much less intuitive for interactive 3D contents.

Unity also has its own machine learning library: Barracuda[22]. However, despite that all Unity platforms are supported by Barracuda's CPU inference, it doesn't have all-

[18]https://developers.google.com/ml-kit
[19]https://developer.apple.com/augmented-reality/arkit/
[20]https://developer.apple.com/documentation/coreml
[21]https://unity.com/unity/features/arfoundation
[22]https://docs.unity3d.com/Packages/com.unity.barracuda@2.0/manual/index.html

around support for GPU inference[23]. This was confirmed as part of a previous student project developing a prototype using Unity. As Barracuda can't be run on Android Unity platforms with OpenGL ES graphics API, which is mandatory for ARCore, its machine learning algorithms can only be run on CPU, which greatly increases the execution time, reduces efficiency and even often leads to app crashes.

Nevertheless, being an all-around game development engine, Unity provides the possibility to access third-party libraries and codes through plugins and even low-level native-plugins that are unavailable to Unity otherwise. Yet, diving into low-level codes or developing a plugin is not an easy task. Luckily, we discovered the possibility to import TensorFlow Lite, the mobile library of the mainstream machine learning platform TensorFlow, as a Unity plugin[24], which supports GPU acceleration with Metal on iOS/macOS, and OpenGL on Android[25].

2.1.5 Sum-up

In this chapter, we investigated several possible AR technologies for our mobile app development. Having access to the Unity prototype developed by another student group, we are able to observe the relative performances of Unity, AR.js and ARCore by only building prototypes for the latter two. We directly ruled out AR.js because of its poor performance, and WebXR because of its huge development and maintenance overhead that we can't afford to take.

There are two main operating systems for smartphones: Android and iOS. Although over 70% of smartphones are powered by Android [12], we still wish to cover both platforms, but Android's ARCore framework has only limited support for iOS, and iOS's ARKit doesn't back other systems at all. Unity's AR Foundation framework can serve as an interface for them both, which allows us to develop a single Unity application that can be exported to both platforms. Moreover, Unity allows us to integrate the AR app into the Ionic app on both platforms, and our app is extendable with potential machine learning algorithms for far-field object occlusion. So, in the end, we decided for Unity as our development platform and AR Foundation as our chosen AR framework for this project.

2.2 Overview of Unity and AR Foundation

This section contains some overview and general information about Unity and AR Foundation. We will first talk about what Unity is, its advantages and disadvantages compared to other development tools suitable for our project. Then, in subsection 2.2.2, the main mechanics of Unity will be introduced for a better understanding, as

[23] https://docs.unity3d.com/Packages/com.unity.barracuda@2.0/manual/SupportedPlatforms.html
[24] https://github.com/asus4/tf-lite-unity-sample
[25] https://medium.com/@asus4/tensorflow-lite-on-unity-4a134e43cbc6

it is a game engine different from conventional development tools which uses explicit terminologies and rules (e.g. physics engine) enabling users to concentrate on game mechanics without writing too much code to interact with the hardware. At last, we'll show AR Foundation's architecture and how it supports multi-platforms.

2.2.1 Unity Overview

Unity, or Unity Editor [26], is a 3D/2D game engine that provides built-in features essential for game development, such as 3D rendering, physics etc. It uses the object oriented language C# as the scripting language to implement game logic, and it is supported on all mainstream operating systems on computers, namely Windows, macOS and Linux. It is not an integrated development environment (IDE) itself, but comes with Visual Studio on Windows and macOS by default, and supports two other IDEs: Visual Studio Code and JetBrains Rider that are compatible with Linux as well. Generally, the IDEs only serve as Unity's external script editor, whereas most of the resource management is taken care of automatically in Unity.

Beside the reasons for choosing Unity's AR Foundation as our AR framework explained earlier, Unity itself comes with a number of advantages. Unity has a huge range of platform support from the web and mobile devices to high-end PC and consoles - especially commonly used for the mobile Android and iOS platforms, which contributes greatly to it being one of the most popular game engines.

Another reason for the popularity is that Unity is easy to use and highly user-friendly to both beginners and experts. Unity comes with a large Asset Store, where developers can upload collections of files and data from their Unity projects, or elements of projects, that are shared to the community as ready-to-use building blocks. Some powerful assets, for instance, are visual scripting tools, which enables artists or beginners that have little or even none coding skills to develop a game or application with Unity.

Also for more experienced developers, Unity Editor is designed in a way that saves tedious efforts compared to IDEs. For example, settings and dependencies of an application can be mostly managed through clicking on Unity's GUI, such that excessive setup codes or scripts are not necessary anymore. Moreover, the GUI allows intuitive drag-and-drop actions not only for organizing folder structure, but also for defining some basic game logic, which we will explain in more detail in the next subsection 2.2.2.

In a nutshell, Unity has wider platform support, faster development iteration time, smoother learning curve, larger active community, is better suited for lightweight games or applications compared to other game engines.

[26] https://docs.unity3d.com/Manual/

When compared to native mobile app development tools, i.e. IDEs for building apps exclusively for a single platform, such as Android's Android Studio and iOS's Xcode however, Unity's easy-to-use and cross-platform support attributes still remain its biggest advantages, but these are also where the shortcomings stem from. Apps developed by native IDEs have better performance and are smaller in size, because they can communicate with the underlying system directly, and have overall better accessibility. Also, being a game engine that excels in dealing with animations, Unity redraws each and every frame, whereas native Android or iOS optimizes and redraws only when needed.

2.2.2 Unity Main Mechanics

Because Unity's mechanics and workflow differ from the most conventional development tools, we introduce some of the most important concepts and features of Unity in order to achieve better understanding for later chapters of this thesis.

The most important concept in Unity Editor is GameObject. Essentially, any visible or invisible object in the game or application is a GameObject, which can represent 3D objects, properties, scenery, cameras, controllers and much more. It is a fundamental building block for Scenes. Each Scene in a game can be understood as a unique level, and for applications that don't have levels, one Scene suffices. Except for default location properties, the functionality and behavior of a GameObject is defined by its Components that determine how the GameObject looks, and what the GameObject does. A GameObject can contain any number of Components and Unity provides a lot of useful built-in Components that largely accelerate the development process.

In most cases, the target features of an application or game are more complicated than what the built-in Components are capable of – and that's where scripting comes in. Developers can create their own Components and control the GameObjects using C# scripts. By default, a new script created in Unity Editor implements a class that derives from the built-in base class "MonoBehaviour", which provides useful hooks to events, so that initializations can be done at the start and actions can be defined for each frame update for example. Such a class that derives from MonoBehaviour can be thought of as a blue print for creating a new Component type that can be attached to GameObjects.

As briefly mentioned previously, Unity provides drag-and-drop utility for some basic program logic in its GUI, mainly references and bindings. For example, when viewing a GameObject in detail, one can drag and drop its Components to reorder them or drag a script from resources folder to the GameObject to attach the script to the object. Other than this, drag-and-drop is also commonly used for grouping, ordering, linking or creating hierarchies for GameObjects, which helps to move, scale, or transform a collection of GameObjects. Besides grouping GameObjects in hierarchies, any GameObject can be stored with all its Components, property values, and child

GameObjects as a reusable asset called Prefab by simply dragging the GameObject from the Hierarchy window to the Project window where all the assets are located.

Scripting and coding in Unity is very powerful. Apart from deriving from the default MonoBehaviour class to define GameObject functionalities, scripts can do a range of other things from extending built-in classes, to getting callbacks and specifying actions for different build stages. Even code created outside of Unity in other programming languages can be included in the form of a plugin.

2.2.3 AR Foundation Architecture

AR Foundation is Unity's solution to provide a common layer for AR applications that can be ported to Android, iOS and a couple of other AR/XR platforms easily with only one code base. It is a development toolkit that is based upon the unified Unity XR plugin framework, which serves to improve Unity's multi-platform beneficence and enables direct integration. It provides an interface for Unity developers to use, but doesn't implement any AR features itself. Figure 1 shows the tech stack of Unity XR plugin framework [27].

AR Foundation has a highly modular system. As we can see, the bottom layer contains various individual provider plugins that equip Unity with access to native AR functionalities in separate packages. On top of that, XR Subsystems provide a level of abstraction between Unity application codes and platform-specific software development kits (SDK) such as ARCore and ARKit. Each subsystem defines a specific feature. For example, the plane subsystem defines an interface for plane detection. The same application code is used in Unity to interact with a detected plane on Android, iOS, or other platforms with an concrete implementation of the plane subsystem [28]. On the top, AR Foundation toolkit communicates with the subsystems and provides the main AR API for Unity applications that comes with some ready-made components and classes, e.g. ARPlaneManager and ARRaycastManager [29][17].

As of now, because AR technologies are still being researched and developed, some latest AR functionalities are not yet integrated into AR Foundation, such as Google ARCore's Geospatial API and Apple ARKit's ARGeoAnchor. Especially the Geospatial API from ARCore is important for us, because it can place wind turbines according to their real-world GPS coordinates with greater accuracy. ARCore's Geospatial API is supported on the iOS platform and also in Unity by using the ARCore Extensions package, a separate package built on top of AR Foundation. This makes it a good candidate for its use in our targeted cross-platform app to place the wind farm at geospatial locations with only one code base.

[27] https://blog.unity.com/technology/unity-xr-platform-updates
[28] https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.0/manual/arsubsystems/arsubsystems.html
[29] https://www.andreasjakl.com/ar-foundation-fundamentals-with-unity-part-1/

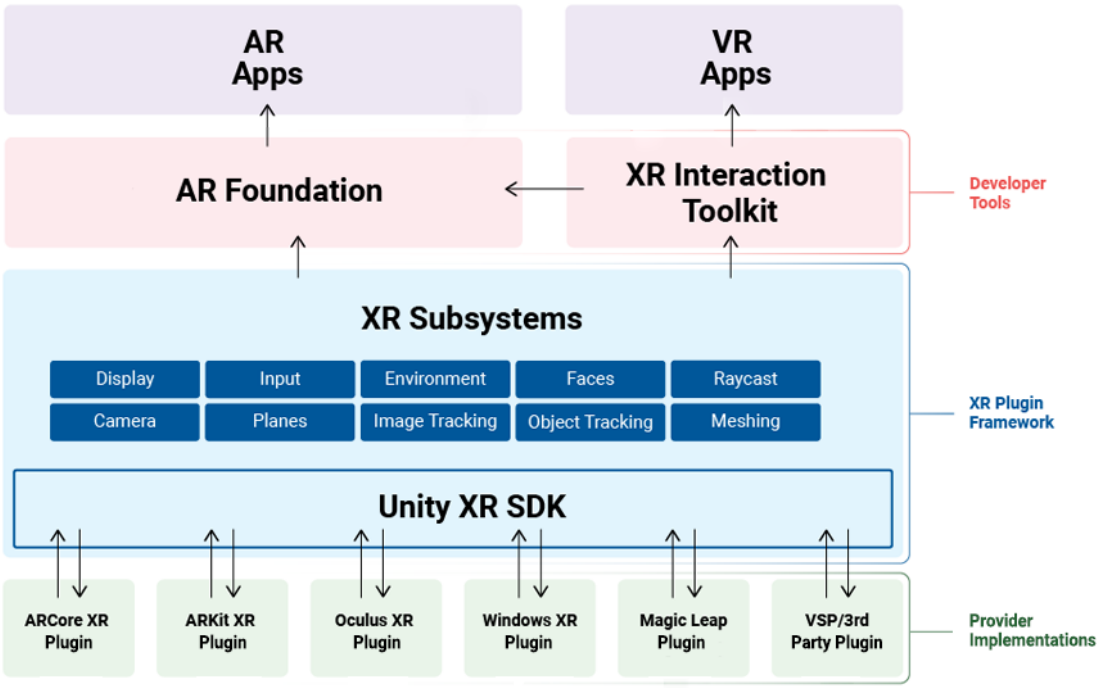

Figure 1: The Unity XR plugin framework and AR Foundation tech stack. Unity XR framework unifies individual provider plugins, with which the AR Foundation toolkit interacts.

3 Software Engineering

This chapter presents the software engineering process of our application. In order to develop a software application, it is of utter importance to first identify the requirements, which are described in Section 3.1. In Section 3.2, according to the organized requirements, we take an UI-First approach to first build an interactive prototype-like mockup that closely resembles the final design of the AR application, so that the feasibility of the design can be evaluated early on. At last, the architecture of our application is laid out in 3.3. We generally adopt the idea of the prototype model, an iterative software development life cycle model, where a prototype is built at an early stage of the project as a basis for the final product implementation. The stages after the proposed prototype being accepted are similar to the classical waterfall model (implementation -> testing -> deployment -> maintenance).

3.1 Requirements Engineering

Careful preparation and planning is essential for successful software development. In this section, we first utilize use case - a commonly used tool in software development - to identify the functional requirements of our application, and then derive the hardware and software requirements from the findings of our technology evaluation of the last chapter.

In order to better organize the requirements, we create a Requirement Trace Matrix as shown in Table 3 at the end of this section that supports the identification of all requirements, and enables us to easily check the coverage of all requirements throughout our project's development phase [7]. Moreover, documenting the requirements in this way also allows stakeholders including future developers to quickly understand the scope, objectives and goals of this application to further develop or extend on its basis.

3.1.1 Functional Requirements

Use cases describe the ways users interact with the system from their point of view. We utilize this methodology for our requirement analysis to identify, clarify and organize the outwardly visible requirements, i.e. the functional requirements of our application system [25]. Our use cases are laid out in this subsection in casual text form.

The **main scenarios** of our primary use case are as follows:

Users will see 3D virtual wind turbines on their device screen placed in real-world positions when they point their device's camera to the planned wind farm sites. No matter where the users are, they will be able to see a map of the turbines, so that they would know where to point their camera to, or in which direction they should go, in order to be near to the planned wind farm sites to see the AR visualizations.

Apart from the AR visualizations, other information such as the noise level and annual shadow cast duration from the wind farms should also be shown to the users according to their real-time position. Also, users are allowed to set wind speed and wind direction. The wind turbines and their wings will rotate accordingly.

As multiple plans for wind farms will be proposed, users will be able to switch through different scenarios. For different plans, various models of turbines and different locations and layouts may be suggested, which should also be reflected visually. Users will also be able to see additional text information about each individual wind turbine, for example the manufacturer etc.

Users should be able to enter this AR application directly through the Ionic app, and if they quit the AR view, they should be directly returned to the Ionic app.

Secondary scenarios comprise of gamification aspects of the application that contribute to making the application more attractive or making the concept of renewable energy and environmental protection more fun to know about. Examples of gamification content include the following:

At different times of a day, some other visually attractive AR events will take place near the planned wind farm sites for users to see, such as hot-air balloons flying in the sky, animals running around or a rocket launching distantly. Users can also explore around the wind farm sites by collecting randomly spawned stars at certain locations and get statistics of their star-collections.

3.1.2 Hardware Requirements

According to our previous AR technology evaluation and our choice of the ARFoundation framework, and thus respectively ARCore and ARKit for Android and iOS platforms, the hardware requirements are rather clear, as the device has to be supported by the underlying frameworks.

An extensive list of devices supported by ARCore can be found on the documentation site of ARCore [30], including the iOS devices that supports ARCore. The ARKit supported iOS devices are also documented in Apple's documentation archive [31]. Generally, ARCore and ARKit are only supported on more recent devices that are released after 2017.

3.1.3 Software Requirements

- Android or iOS mobile operating system
- Android 7.0 Nougat+, API level 24+

3.2 Prototyping

The "Prototype Model" is an iterative software development model, where a prototype, i.e. an incomplete version of the software application that simulates only a few aspects of the final product, is built, tested, and refined. These steps are looped until a final accepted prototype is achieved, which then serves as the basis for the final software product. Prototyping is valuable for detecting missing functionalities early on in the development phase, testing the usability and feasibility of a design solution, and thus reducing risks of a project failing.

[30] https://developers.google.com/ar/devices

[31] https://developer.apple.com/library/archive/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/DeviceCompatibilityMatrix/DeviceCompatibilityMatrix.html

Entry	Requirement	Type
1	The user shall be able to enter the AR application through the main Ionic app and go back to Ionic app from the AR application.	Functional
2	The user shall be able to see 3D wind turbines on their destined position if they point their device camera to the planned wind farm sites.	Functional
3	The user shall be able to see the position of wind turbines with respect to their current position on a map to gain orientation to the wind farm sites.	Functional
4	The user shall receive error notifications if their device position cannot be determined.	Functional
5	When the user clicks on a 3D wind turbine, information about this particular turbine will be shown to the user.	Functional
6	The user shall be able to switch through different wind farm planes to see different layouts and types of wind turbines at different locations.	Functional
7	The user shall be able to set the wind speed and wind direction and the 3D wind turbines will simulate the real-world situations accordingly with self-rotations and wing-rotations.	Functional
8	The user shall be able to see other information about the potential influences from the planned wind farms, such as noise level and annual shadow cast duration according to their current position.	Functional
9	The user shall be able to read about some facts about wind farms and renewable energy in an interesting way.	Nice-to-have
10	The user shall be able to see some optional cool 3D events or visual effects if they wish to, such as hot-air balloons in the sky etc.	Nice-to-have
11	The user shall be able to switch on or off the additional visusal effects.	Nice-to-have
12	The user shall be able to collect randomly spawned stars around planned wind farm sites when they walk close to the stars.	Nice-to-have
13	The user shall be able to see statistics of their collection of stars as feedback and motivation to collect more.	Nice-to-have
14	The user's device has to be an Android or iOS device that supports ARCore or ARKit.	Hardware
15	The user has to run the application inside an Android or iOS mobile operating system.	Software

Table 3: The Requirements Trace Matrix for our AR application. It contains all requirements from functional to non-functional requirements of this application project and serves as a requirements document to effectively communicate with all stakeholders.

For the development of highly interactive software systems, such as web and smartphone applications, prototyping, and, in particular, interface prototyping has become increasingly important, as the acceptance of such systems depends to a large degree on the quality of their user interface [5]. Bäumer et al. pointed out in their work that user interface prototypes can range from complete mockups without functional aspects to fully functional systems[5]. They classified different types of prototypes into four categories depending on how and to what degree the functionality is implemented in a prototype: presentation prototypes, functional prototypes, breadboards and pilot systems.

- **Presentation Prototypes** are built to illustrate potential solutions to a set of given requirements and are often used as part of the project proposal, therefore strongly focused on the user interface.
- **Functional Prototypes** implement strategically important parts of both the user interface and the functionality of a planned application.
- **Breadboards** are utilized for investigating technical aspects such as system architecture or functionality. They serve to inspect certain factors of special potential risks and are not intended for end users.
- **Pilot Systems** are very mature prototypes that are close resemblances of the final product regarding both interface and functionality which can already be practically applied.

A more general way to categorize prototypes is according to their fidelity. The fidelity of a prototype is determined by how it appears to the viewer rather than by how closely it resembles the final product. In other words, the degree to which the prototype accurately simulates the appearance and user experience of the final product, rather than the accuracy of the code and other features hidden from the user, is what determines the fidelity of the prototype [23, 27].

On one hand, low-fidelity prototypes involve minimal functionality and restricted interactivity and often simply sketched on paper. They are valuable in early stages of development where requirements are not fully gathered or still being evaluated, in that they can be the communication medium between users and developers and help in the brainstorming process. On the other hand, high-fidelity prototypes are almost fully interactive and allow users to interact with the user interface as if it were a real product. They define the navigation scheme of a system clearly and are much more helpful than low-fidelity prototypes for error checking [23]. Developers can use high-fidelity prototypes as a living specification of the functional and operating requirements.

Because our project has a small scale and most requirements are already specified, we started out with a low-fidelity wireframe that focused on our application's navigation logic in order to rapidly gain insights to the functional feasibility. Then, we designed a high-fidelity prototype - a mockup - with the web-based interface prototyping tool

Figma[32] that resembles the final product visually up to several icons and images. This mockup contains not only static pages, but also functional linking between buttons and pages. In this way, we finished up the prototyping phase of our application with a "Presentation Prototype" as defined by Bäumer et al.. It is not yet a "Functinoal Prototype" because the most important functionality is not implemented. Screenshots of the mockup are presented in Figure 2.

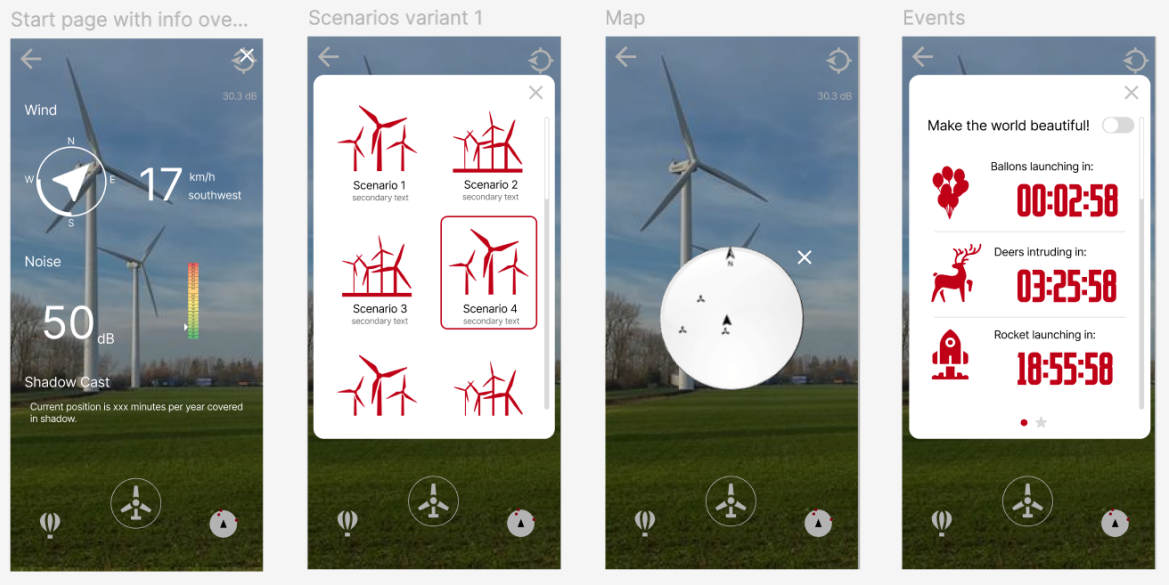

Figure 2: Several pages of the app interface mockup that almost entirely resembles the final product's UI visually.

3.3 Software Architecture

The Unity system is designed in a way that is highly compatible with a component-based architecture because of its emphasis on its core concept of game objects and Components. A component-based software architecture focuses on the decomposition of the system design into distinct individual functional or logical components[33]. There are different understandings of component-based development in academia and industry. While industry views a component as a reusable and self-deployable unit encapsulating functionality and behaviors of a part of the software system, researchers in academia usually define components as well-defined entities that are typically small with easily understandable functional and non-functional properties[10]. Either way, the most important feature of components is reusability, and the larger the components are, the greater the productivity can be achieved by reusing them. As Unity

[32]https://www.figma.com
[33]https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm#

emphasises the proper use of components, we naturally adopt the idea of component-based software engineering throughout our development. This means avoiding highly coupled inheritance relations of classes conventional in Object-Oriented Programming (OOP) and employing well-defined entity classes of the academic definition that can be attached to other object or classes with little restriction.

Moreover, for the general architecture of our application, we adopt the well-known Model-View-Controller (MVC) pattern. In the MVC pattern, *Model* stands for the database of the application, and if applicable, it also entails the data structure and management logic for the database, independent from any interface. *View* is the visual representation of the information depicted by the model. In the web/app development context, the view represents the complete user interface, including all UI elements such as menus and buttons, so it can also receive user inputs. *Controller* is the "brain" of the application that actually handles user inputs and converts it to commands for the model or view. The controller determines if a user input should change the data provided by the model and manipulates the model accordingly. So principally, the model is responsible for the view updates. However, sometimes, the controller could also apply direct updates on the view if the model isn't changed. An illustration of MVC pattern is shown in Figure 3.

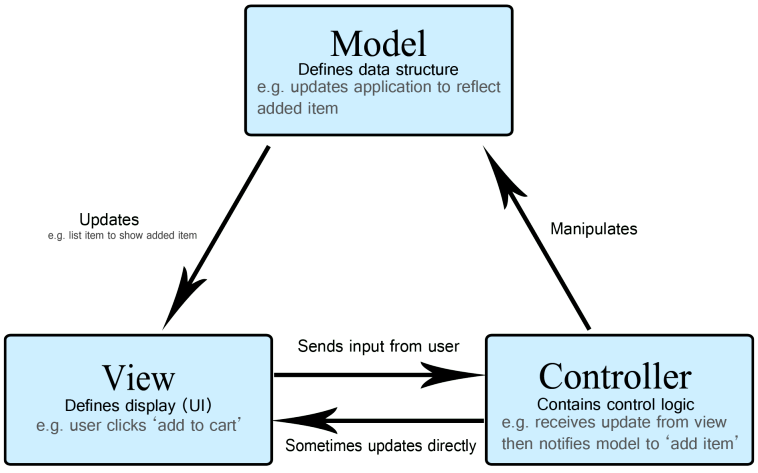

Figure 3: An illustration of the model-view-controller (MVC) pattern[34].

In the case of our application, the view is exactly our graphical user interface. As for the model, the data that we receive and build upon are static and need no changes from this application. This is due to the application being solely developed for citizens to experience the impact of planned wind farms, not for the stakeholders such as the city government to manipulate and edit the underlying data. The only manipulatable

[34]https://developer.mozilla.org/en-US/docs/Glossary/MVC

variables are the wind speed and wind direction that are temporary and only retained for the current session, which are not stored in the base database.

The controller part may consist of different controllers. We separate the responsibilities for geospatial visualizations and UI visuals. Most user interface elements that do not depend on geospatial localizations are controlled by a UI-controller, and the geospatial-relevant elements (mostly the 3D windmill models and 2D windmill sprites on maps) are managed by a geospatial-manager that places 3D objects according to coordinates stored for them in the model. To control the rotation of wind turbines and their wings according to user settings, another windmill controller is implemented as a reusable component for each windmill game object locally to easily manipulate the 3D models. This choice leads to better modularity and clearer separation of code.

4 Implementation

This chapter addresses the concrete implementation of our application. We illustrate the functional implementations according to the software requirements and design in the previous Chapter 3 in detail.

4.1 Geo-located Wind Turbines

The core objective of our application is to provide realistic AR wind turbine visualizations for citizens living near planned wind farm sites so that they can experience the visual impacts virtually. This requires placements of wind turbines to be as accurate as possible. The AR approach is a bit different as the VR approach, in that VR creates virtual objects for the whole virtual environment with no real-world visuals, whereas AR provides virtual augmentations that should be immersed into the real-world environment. So the correct placement of objects in AR is more challenging, because not only device movements and relative device orientation change, but also reference points, such as markers in the case of marker-based AR, must be recognized. In the case of geo-location-based AR, a correct rendering of AR objects additionally relies on device location based on GPS and its true heading, namely, the orientation relative to true north.

Recent smartphones have a GPS horizontal position error of under 5-10 meters most of the time under open sky, but the position accuracy worsens when the device is near buildings and trees [18], because GPS positioning relies on measuring the delay of radio signals from satellites and disruption can be caused by signals reflecting off surfaces. As for heading, it was already pointed out in numerous previous studies that the built-in compass headings produced by magnetometer has poor accuracy that can be a critical issue for AR applications [2, 18], since this sensor is easily skewed by magnetic objects.

To tackle this issue, Google's ARCore Geospatial API [35] provides an advanced solution to improve the device position and heading by using a technique called global localization which combines Google's Visual Positioning System (VPS), Street View, and machine learning [36]. VPS has a localization model consisting of trillions of 3D point clouds learned from Google Street View data around the globe. By scanning surroundings with the device camera, images are sent to Geospatial API along with GPS and orientation data, where recognizable parts of these images are identified by deep neural networks and then compared to corresponding parts of the localization model inside VPS based on given location data. A significantly more accurate position and orientation of the device is at last calculated by computer vision algorithms which was previously impossible with GPS and phone sensors alone. It's worth noting that internet is required for Geospatial API to work on a device.

In order to place Unity game objects at specific real-world locations through ARCore's Geospatial API, they have to be attached to so-called "Geospatial anchors". An anchor is created with a pose that describes the anchor's orientation and position in the world space for a single frame. World space is the coordinate space in which the camera and virtual objects are positioned, and all positions of all objects including the camera are updated each frame. Positions of Geospatial anchors are based on geodetic latitude, longitude, and altitude, and orientations are expressed in quaternions.

```
1 ARGeospatialAnchor anchor =
2     ARAnchorManager.AddAnchor(
3         latitude,
4         longitude,
5         altitude,
6         quaternion);
7
8 GameObject go = Instantiate(WindmillPrefab, anchor.transform);
```

Listing 1: Creating Geospatial anchor and instantiating objects to attach to the anchor.

As shown in Listing 1, an *ARGeospatialAnchor* is created by ARFoundation's *ARAnchorManager*. One or multiple game objects can be attached to the anchor by instantiating as children of the anchor. The instantiation takes an object, in this case, a windmill prefab which is a game object that contains the 3D mesh, colliders and materials of the windmill, and the transform of its parent the anchor, as its arguments.

However, it is very crucial, when to place the anchors. The *AR Session* is the main entry point to ARCore and other provider APIs, which controls the life cycle of an AR experience. Together with *AR Session Origin* that transforms AR coordinates into Unity world coordinates, *AR Session* can enable AR processes, such as motion tracking and environmental understanding. All anchors that are placed before *ARSession*

[35] https://developers.google.com/ar/develop/geospatial
[36] https://ai.googleblog.com/2019/02/using-global-localization-to-improve.html

is ready will be null, and thus not able to be updated to their correct position anymore. This is also because, using the Geospatial API, the device position and headings and all anchors are dependent on an *AREarth* instance, which represents the earth by abstracting it with an ellipsoid according to WGS84 specification [37] and provides the localization ability. The *AREarth* object should only be used when its *TrackingState* is *TrackingState.TRACKING*, which includes anchor instantiations [38]. The *TrackingState* is *TrackingState.PAUSED* when, for example, device tracking is lost, or if the *ARSession* is currently paused. In this case, the properties of the *AREarth* instance may be wildly inaccurate and should generally not be used.

The *Update()* method, shown in Listing 2, is a Unity method that is called once every frame after application start. We check the localization state by *EarthTrackingState*. Whenever the tracking is lost, we disable all AR contents so that they can't be seen on the screen anymore to avoid undesirable visuals like objects jumping around. In addition, a message will be shown to the user to make it clear that the application is localizing, and when localization is regained, the AR contents are enabled again to be shown to the user. If the *ARSession* is not ready or the *AREarth* has not started tracking yet, the message will also be shown until the first time localization is done, and the Geospatial anchors and windmills will be placed. The array *_windmillObjects* stores references to all windmill game objects and *_firstTimePlacement* is a boolean variable that help us to instantiate the windmill game objects only once and to avoid duplicates.

4.2 Map for Orientation

For better orientation and overview to the user, we implement a map that shows the position of wind turbines that can help the user to go to wind farm sites and look in the correct direction with their devices' cameras to see the AR contents. As shown in Figure 5, the triangle fixed in the center of the map represents the user, and it always points towards the front, identical to the direction the device is heading. Each time the user moves or changes direction, the position of all windmill icons are updated. In addition, the button that opens the map is itself a minimized version of the map, supporting the orientation of the user.

Because positions of windmills in the world space are determined by the Geospatial API through real-world coordinates, in order to show them on a 2D map, the windmills have to be placed in the 3D world space first. For each placed windmill, we first instantiate two 2D graphic sprites that represent it on the two maps respectively (Listing 3). Then, the positions of 3D windmills are reflected on the 2D map space through the sprites, so that the map should look like a top-down view of the 3D. This means, the vertical Y axis of the 3D world space should be disregarded, the horizontal X axis of 3D

[37] https://en.wikipedia.org/wiki/World_Geodetic_System
[38] https://developers.google.com/ar/reference/java/com/google/ar/core/Earth

remains the X axis of 2D, and the horizontal Z axis of 3D becomes Y axis of 2D, as reflected in line 24 and 25 of Listing 4. Figure 4 illustrates the alignment of axes in Unity.

```
1  void Update()
2  {
3      bool isSessionReady =
4          ARSession.state == ARSessionState.SessionTracking;
5      var earthTrackingState = AREarthManager.EarthTrackingState;
6      if (!isSessionReady || earthTrackingState != TrackingState.
           Tracking)
7      {
8          //Lost localization during the session
9          if (!_isLocalizing)
10         {
11             _isLocalizing = true;
12             foreach (var go in _windmillObjects)
13             {
14                 go.SetActive(false);
15             }
16         }
17         LocalizationPanel.SetActive(true);
18     }
19     else if (_isLocalizing)
20     {
21         //Finished localization
22         _isLocalizing = false;
23         LocalizationPanel.SetActive(false);
24
25         //Activate all AR GameObjects or place them for the first
               time
26         if (!_firstTimePlacement)
27         {
28             foreach (var go in _windmillObjects)
29             {
30                 go.SetActive(true);
31             }
32         }
33         else
34         {
35             _firstTimePlacement = false;
36             PlaceWindmills();
37             InstantiateWindmillSprites();
38         }
39     }
40     ...
41 }
```

Listing 2: Handling localization and placing windmills after the first successful localization.

PlaceWindmills function is shown in Listing 9 and *InstantiateWindmillSprites* in List-

ing 3.

```
1  public void InstantiateWindmillSprites()
2  {
3      for (int i = 0; i < _windmillSprites.Count; i++)
4      {
5          // Sprites on the bigger map
6          GameObject WindmillSprite = Instantiate(WindmillSpritePrefab,
               SpritesContainer.transform);
7          _windmillSprites.Add(WindmillSprite);
8
9          // Sprites on the mini map button
10         GameObject MiniSprite = Instantiate(DotSpritePrefab,
               MinimapBtn.transform);
11         MiniSprite.transform.localScale = new Vector3(.5f, .5f, .5f);
12         _miniSprites.Add(MiniSprite);
13     }
14 }
```

Listing 3: Instantiating 2D windmill sprite game objects on both maps.

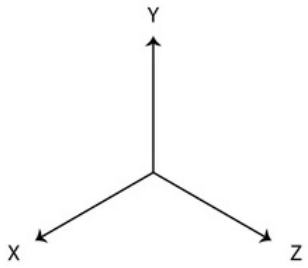

Figure 4: Unity's axes alignment.

In an AR app with an AR camera, the origin of the 3D world space is exactly the position of the device at session startup and is fixed. When the user moves around after an AR session is initialized, the device or camera position will be updated each frame relative to the established axes. The origin of the 2D screen space is at the bottom left corner of the screen, not where the user triangle sprite is placed at. To reflect the relative positions of the user and windmills on the 2D map, we first put the sprites to the center of the map (also the position of the user triangle), and then move them according to their 3D translations to the device.

Windmill sprites that are out of the map further away than 250 meters are also shown in order to give the user a rough orientation, but on the rim of the map to extinguish from those that are within the radius. They also have different graphics. Sprites on the rim appear as dots and those within the radius appear in turbine form. All windmill sprites on the minimized map button have dot form to attain the functionality but in a generalized way (because the button is rather small on screen and unsuitable for too

much information). To place sprites on the mini map button, we only have to duplicate and scale the local position of the sprites on the bigger map, i.e. their position with respect to the parent circle image objects framing the maps.

```
1  public void UpdateSpritesPosition()
2  {
3      for (int i = 0; i < _windmillSprites.Count; i++)
4      {
5          var mill = _windmillObjects[i];
6          var sprite = _windmillSprites[i];
7          var miniSprite = _miniSprites[i];
8          Vector3 translate = new Vector3(
9              (mill.transform.position.x - ARCamera.transform.position.
                   x),
10             (mill.transform.position.z - ARCamera.transform.position.
                   z), 0);
11         sprite.transform.position = spritesOrigin;
12         float dist = translate.magnitude;
13
14         if (dist < 250)
15         {
16             sprite.transform.Translate(translate);
17             if (sprite.GetComponent<Image>().sprite != windmillSprite
                   )
18             {
19                 sprite.GetComponent<Image>().sprite = windmillSprite;
20             }
21         }
22         else
23         {
24             sprite.transform.Translate(translate/dist*250);
25             if (sprite.GetComponent<Image>().sprite != dotSprite)
26             {
27                 sprite.GetComponent<Image>().sprite = dotSprite;
28             }
29         }
30
31         miniSprite.transform.localPosition = sprite.transform.
               localPosition/5f;
32     }
33 }
```

Listing 4: Updating sprites position on map in real time and change their graphics according to the distance.

After positioning the sprites in their local space, we still have to address the rotation of the camera. Instead of looping through and transforming every sprite separately, we directly rotate the parent containers around the Z axis. The Geospatial API sometimes carry out re-localizations where the detected device orientation can change, but luckily, all orientation changes are directly reflected in the AR Camera's transform.

The parent containers only have to rotate according to the degrees changed around the Y axis of the camera as shown in line 9-10 of Listing 5. The if-case in line 5 ensures that the map update will only be carried out after the placements of windmills.

```
1  void Update()
2  {
3      ...
4
5      if (!_firstTimePlacement && _windmillSprites.Count > 0)
6      {
7          UpdateSpritesPosition();
8
9          SpritesContainer.transform.Rotate(0, 0, ARCamera.transform.
               eulerAngles.y - prevHeading);
10         MinimapBtn.transform.Rotate(0, 0, ARCamera.transform.
               eulerAngles.y - prevHeading);
11     }
12     prevHeading = ARCamera.transform.eulerAngles.y;
13 }
```

Listing 5: Updating the map each frame after geospatial windmill placements.

Furthermore, we show the distance between the user and the closest windmill by directly using the built-in distance function for two vectors. As one Unity unit equals to one meter in real-life, we don't need to carry out complicated calculations for the distance using GPS coordinates anymore.

4.3 Wind Farm Scenes

For the planning of wind farms in a certain area, multiple scenarios of different construction plans of windmills are often provided for residents, which might contain different locations and layouts of wind farms and different types of wind turbines.

In order to enable users to navigate through various possible scenarios visually, we designed a button that shows a panel of all scenarios on click as illustrated in Figure 6. After clicking on a chosen scenario, the corresponding 3D windmills planned for that scenario will be rendered in the camera view and their sprites accordingly on the map.

We store the scenario data in a JSON data as depicted in Listing 6. We use the built-in *JsonUtility.FromJson* method to directly create objects from the JSON representation. This method uses Unity's serializer internally [39], which automatically transforms data structures that Unity can store and reconstruct, therefore all types created and their fields must be public and serializable. The target classes also have to be plain classes

[39] https://docs.unity3d.com/ScriptReference/JsonUtility.FromJson.html

Figure 5: Orientation map. Wind turbines outside of the map range are shown on the rim as dots. A minimized map is always shown in the lower right corner on the b

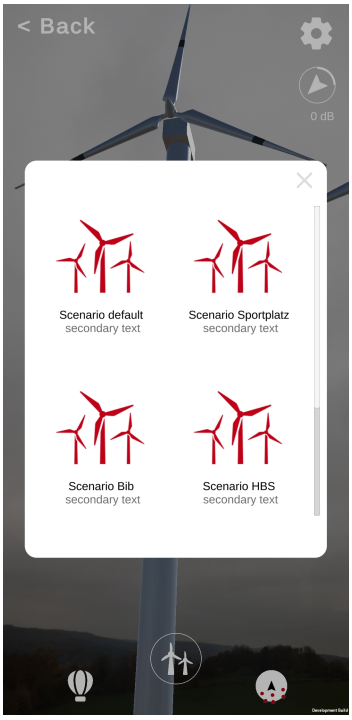

Figure 6: Wind farm scenarios panel, can be opened by the button on the bottom in the middle.

or structures and cannot be derived from *MonoBehaviour*. Thus, we declare the corresponding classes as shown in Listing 7.

```
{
    "scenarios":
    [
        {
            "nameScenario": string,
            "turbines":
            [
                {
                    "longitude": double,
                    "latitude": double,
                    "altitude": double,
                    "label": string,
                    "manufacturer": string,
                    "height_m": float,
                    "bladeDiameter_m": float,
                    "accentColor_hex": string
                },
                ...
            ],
            "noiseLines": [
                {
                    "value": int,
                    "coords": [
                        {
                            "longitude": double,
                            "latitude": double
                        },
                        ...
                    ]
                },
                ...
            ],
            "shadowLines": [
                {
                    "value": int,
                    "coords": [...]
                },
                ...
            ]
        },
        ...
    ]
}
```

Listing 6: JSON structure for storing scenarios and windmill data.

```
[System.Serializable]
public struct Windmill
{
    public double longitude;
    public double latitude;
    public double altitude;
    public string label;
    public string manufacturer;
    public float height_m;
    public float bladeDiameter_m;
    public string accentColor_hex;
    public float[] rpm;
}
[System.Serializable]
public struct LatLngCoord
{
    public double longitude;
    public double latitude;
}
[System.Serializable]
public struct ContourLine
{
    public int value;
    public LatLngCoord[] coords;
}
[System.Serializable]
public struct Scenario
{
    public string nameScenario;
    public Windmill[] turbines;
    public ContourLine[] noiseLines;
    public ContourLine[] shadowLines;
}
[System.Serializable]
public struct WindfarmScenarios
{
    public Scenario[] scenarios;
}
```

Listing 7: Wind farm scenario and windmills object type declaration in Unity.

Each wind farm scenario has a list of windmills (turbines) and a name to distinguish them from each other. All scenarios, their name and descriptions are shown to users on GUI as buttons. Each windmill also has its real-world geospatial coordinates and other relevant information that are to be shown to the user when clicking on the AR windmills (more about this in Section 4.4). We call the *JsonUtility.FromJson* method in Unity's Start method, which is called exactly once per script before any of the *Update* methods is called, to store the structured data. Then, the scenarios panel's buttons are dynamically generated according to the stored data. Also, the first scenario is stored in a global variable as the default scenario shown on launch, whose windmills will be

placed and rendered after the AR Session is ready and localization is done in Unity's *Update* method (Listing 2).

```
1  public void Start()
2  {
3      windfarmScenarios = JsonUtility.FromJson<WindfarmScenarios>(
           WindfarmJSON.text);
4      CreateScenarioBtns(windfarmScenarios);
5      _activeScenario = windfarmScenarios.scenarios[0];
6  }
7
8  public void CreateScenarioBtns(WindfarmScenarios scenarios)
9  {
10     //Add scenario buttons to the panel according to given values
11     foreach (Scenario scenario in scenarios.scenarios) {...}
12 }
```

Listing 8: Reading and storing structured wind farm JSON data and creating GUI dynamically for different scenarios in Unity's "Start" method on the run.

The windmills are initially placed by the *PlaceWindmills* method in Listing 9 that loops through all turbines in the active scenario and places each of them according to the coordinates stored in the *Windmill* object. When a user switchs scenarios, the *ReplaceWindmill* function is called, which updates the active scenario by removing previous windmill and sprite game objects and placing new ones of the new scenario.

4.4 Windmill Functionalities

The windmills have other functionalities such as the possibility to retrieve information about the windmill by clicking on the windmill, the ability to rotate according to the wind direction, and a blade rotation according to the wind speed. The wind direction and speed can be set by the user on an overlaying info panel as illustrated in Figure 8. Information about the current device position's noise level and shadow cast estimations are also shown on this panel.

The wind speed can be defined by typing numbers into the input field. We restrict the content type of the input field to integer numbers. Our JSON data file contains wing rotation data for several turbines. A small part of the required data consists of real-world data provided by authorities, and the research group intends to purchase additional data to include the correct information into the app. Each turbine has an RPM (revolution per minute) attribute, which is an array of float numbers representing the wing rotation frequency according to the wind speed. The index of each float element represents the wind speed. The use of array instead of dictionary is because of lack of support from Unity's *JsonUtility*.

```
public void PlaceWindmill(Windmill turbine)
{
    Quaternion quaternion = Quaternion.AngleAxis(180f + _winddir,
        Vector3.up);
    var anchor = ARAnchorManager.AddAnchor(turbine.latitude, turbine.
        longitude, turbine.altitude, quaternion);
    _anchors.Add(anchor);
    if (anchor != null)
    {
        GameObject windmillGO = Instantiate(WindmillPrefab, anchor.
            transform);
        windmillGO.GetComponent<WindmillInfo>().windmill = turbine;
        _windmillObjects.Add(windmillGO);
    }
}

public void PlaceWindmills()
{
    foreach (var turbine in _activeScenario.turbines)
    {
        PlaceWindmill(turbine);
    }
}

public void ReplaceWindmills(Scenario scenario)
{
    _activeScenario = scenario;
    for (int i = 0; i < _windmillObjects.Count; i++)
    {
        Destroy(_windmillObjects[i]);
        Destroy(_anchors[i]);
        Destroy(_windmillSprites[i]);
        Destroy(_miniSprites[i]);
    }
    _windmillObjects.Clear();
    _anchors.Clear();
    _windmillSprites.Clear();
    _miniSprites.Clear();
    PlaceWindmills();
    InstantiateWindmillSprites();
}
```

Listing 9: "PlaceWindmill" function for placing turbines, and "ReplaceWindmill" function for scenario switching.

The provided RPM values for each turbine cover different wind speed ranges and most of them start to rotate only when the wind speed is above 3 m/s. Furthermore, RPM values for wind speed either lower than 3 m/s or above 20 m/s are often missing from the data on hand. For missing RPM values at low wind speed, we simply set them to 0 in the array. For those at high wind speed, we set the rotation frequency to the

highest provided RPM of this certain turbine, i.e. the last float value of the array. The placeholder wind speed is 0 m/s at app start.

In order for Unity to utilize the RPM values, we have to convert them into degrees per second:

$$1RPM = 360°/minute = 6°/second$$

as shown in line 13-15 in Listing 10. As the blades of turbines rotate around their intersection and each windmill rotates around themself, which means that these rotations are all local. We attach the script to each windmill prefab that contains different parts of the 3D model including the turbine blades as child game objects, so that the blades can be easily accessed and manipulated.

```
1  public class WindmillController : MonoBehaviour
2  {
3      public GeospatialManager.Windmill windmill;
4
5      private void Update() {
6          if (windmill.rpm != null)
7          {
8              float rpm = windspeed>windmill.rpm.Length-1
9                  ? windmill.rpm[windmill.rpm.Length-1]
10                 : windmill.rpm[windspeed];
11
12             if (windspeed > 0)
13             {
14                 blades.transform.RotateAround(blades.transform.
                       position, transform.forward, 6*rpm*Time.deltaTime)
                       ;
15             }
16         }
17
18         transform.localRotation = Quaternion.Slerp(
19             transform.localRotation,
20             Quaternion.Euler(0, GeospatialManager.Instance._winddir,
                   0),
21             Time.deltaTime*smooth);
22     }
23 }
```

Listing 10: Script attached to each windmill prefab that controlls the blades rotation and turbine self-rotation.

The initial wind direction is south, so the turbines should be facing north, as shown in Line 3 of Listing 9, (180 degrees of rotation around the Y axis of a normal model that is facing the +Z axis). The global variable *_winddir* indicates the orientation offset between windmills and the current true north in the world space according to the set wind direction and its initial value is 0. Table 4 shows the corresponding offsets of different wind directions.

Wind Direction	Orientation Offset
S	0°
SSW	22.5°
SW	45°
WSW	67.5°
W	90°
WNW	112.5°
NW	135°
NNW	157.5°
N	180°
NNE	−157.5°
NE	−135°
ENE	−112.5°
E	−90°
ESE	−67.5°
SE	−45°
SSE	−22.5°

Table 4: Orientation offset of wind turbines' Y axes from the true north according to the currently set wind direction.

For self-rotation according to the wind direction, we don't have real-world data about the speed of rotation, nor do we need to simulate a real-world situation where the change might happen very slowly. But we still show the user a smooth animation by using the *Quaternion.Slerp* method (line 20 of Listing 10) that spherically interpolates between the original and target headings of the turbine. The smooth factor determines the interpolation rate, so a larger value would result in a faster rotation. As animations involve updating each frame, turbine self-rotation and wing-rotation are both implemented in Unity's *Update* function.

Each time when the user clicks on a rotate button to change the wind direction, the *UIController* will update the *_winddir* variable in *GeospatialManager* (Listing 11), which is referenced as the rotation destination of the turbines.

When a user click on a 3D model of a windmill shown on their device's screen, an information panel will pop up that shows some of the data stored in the *Windmill-Controller* component attached to the windmill game object as illustrated in Figure 9.

We use ray casting to detect touches on a model. First of all, it is essential for the 3D wind turbine model to have collider components that handle physical collisions of game objects in Unity and define the shape of collideable areas. Since colliders are invisible, for optimal performance, they do not need to have the exactly shape of the model (more vertices to compute), so we assign primitive box colliders or capsule colliders to

the windmill parts whenever possible.

```
1  public class UIController : MonoBehaviour
2  {
3      public void OnWindDirRCWBtnClicked()
4      {
5          RotateWindDirBtnCW();
6          UpdateWindDirText();
7          GeospatialManager._winddir += 22.5f;
8      }
9
10     public void OnWindDirRCCWBtnClicked() {...} //Analogous to above
11 }
```

Listing 11: Handeling click events on wind direction rotation buttons.

Being called each frame in the *Update* method, the *OnScreenTouched* method acts as a listener for touch events and detects any touch input on the screen. If a user puts multiple fingers on the screen at the same time, only the first touch will be used for ray casting. Also, only the initial point of contact with the screen is considered (*TouchPhase.Began* in line 23 in Listing 12), so sliding across a windmill game object will not activate the pop up panel.

Furthermore, touches on UI elements are recognized and blocked. For example, when the user clicks on a button that is in front of and overlaps with a windmill game object, only the button should be activated without the windmill's info panel popping up at the same time. This is done by implementing an extension method for the Unity's *Vector2* type (positions on screen are 2D coordinates) as shown in Listing 12. The Unity *EventSystem* manages input events in a scene and its *RaycastAll* method utilises the *GraphicRaycaster* in our scene. The *GraphicRaycaster* is a component attached to the Canvas by default that raycasts against graphic elements such as UI images. If the raycast through the touch position hits any graphics, it means the user is pointing on UI.

If the touch is not over any UI, we shoot another ray through the physics engine to detect hits on colliders that directly outputs the first hit object. Since our windmill models are the only game objects that have colliders attached to them, a pop up panel that shows the hit object's information will be directly activated.

4.5 Noise and Shadow Cast Information

Apart from wind direction and speed that can be customized by users, the information panel (Figure 8) also shows information about the noise level and annual shadow cast minutes according to the device's position. The input data are contour lines (closed polygons) consisting of GPS coordinates that line out areas with shared values.

```
1  public static class Vector2Extensions
2  {
3      public static bool IsPointOverUIObject(this Vector2 pos)
4      {
5          PointerEventData eventPosition = new PointerEventData(
               EventSystem.current);
6          eventPosition.position = new Vector2(pos.x, pos.y);
7
8          List<RaycastResult> results = new List<RaycastResult>();
9          EventSystem.current.RaycastAll(eventPosition, results);
10
11         return results.Count > 0;
12     }
13 }
14
15 public class GeospatialManager : MonoBehaviour
16 {
17     private void OnScreenTouched()
18     {
19         if (Input.touchCount > 0)
20         {
21             Touch touch = Input.GetTouch(0);
22
23             if (touch.phase == TouchPhase.Began && !touch.position.
                   IsPointOverUIObject())
24             {
25                 Ray ray = Camera.main.ScreenPointToRay(touch.position
                       );
26                 RaycastHit hitObject;
27
28                 if (Physics.Raycast(ray, out hitObject))
29                 {
30                     GameObject go = hitObject.collider.gameObject;
31                     if (go != null)
32                     {
33                         UIController.OnWindmillGOClicked(go.transform
                               .parent.transform.parent.gameObject);
34                     }
35                 }
36             }
37         }
38     }
39
40     private void Update()
41     {
42         ...
43         OnScreenTouched();
44     }
45 }
```

Listing 12: Listening to touch events, raycasting and pop up information panel of a windmill on click.

The format of the input JSON data is as shown in Listing 6 (line 20-39) and the corresponding Unity structs in Listing 7.

Because noise data and shadow cast data are similar in their format, we process the data accordingly. We use a point-in-polygon (PNPOLY)[40] algorithm to determine which area the current position of the device is in.

To check if a point is in a polygon within a conventional Cartesian coordinate system, one can shoot a ray starting from the test point, and count how many edges it crosses. If and only if the total number of edges the ray crosses is an odd number, according to the Jordan curve theorem [9], it implies that the test point is inside the polygon.

The algorithm is as depicted in Listing 13. We have a boolean variable that denotes if a point is inside a given polygon from the data, whose initial value is false, i.e. outside by default. We then "shoot a ray towards the north pole" and check the intersection by iterating through the array of arranged coordinates, taking two coordinates each time that are adjacent to each other forming an edge of the polygon. If the longitude values of both coordinates of an edge are bigger (to the east) or smaller (to the west) than that of the device's position, an intersection is already impossible, so we directly continue with the next edge. Otherwise, we check if the ray crosses the edge based on linear equations.

Assume we check the intersection of an edge with two vertices $P(x_1, y_1)$, $P(x_2, y_2)$, and a ray shooting from point $P(x_t, y_t)$ towards and parallel with +x axis. The linear equation of the edge is $L_1 : y = k * x + b$ with $k = \frac{y_1 - y_2}{x_1 - x_2}$ and b representing the distance of the intersection of this line and Y axis to the origin. The linear equation of the ray is $L_2 : y = y_t$. The intersection is the solution to $L_1 = L_2$ and it has to be on the ray $x > x_t$. So if and only if $(y_t - y_1)/k + x_1 > x_t$, the ray intersects the edge (line 7 of Listing 13).

```
1  private bool IsCoordInPoly(double lat, double lng, GeospatialManager.
       LatLngCoord[] poly)
2  {
3      bool inside = false;
4      for (int i=0, j=poly.Length-1; i<poly.Length; j=i++)
5      {
6          if ((poly[i].longitude>lng) != (poly[j].longitude>lng) &&
7              lat < (poly[j].latitude-poly[i].latitude) * (lng-poly[i].
                   longitude) / (poly[j].longitude-poly[i].longitude) +
                   poly[i].latitude)
8          {
9              inside = !inside;
10         }
11     }
12     return inside;
13 }
```

Listing 13: Checking if a coordinate is in a polygon.

[40] https://wrfranklin.org/Research/Short_Notes/pnpoly.html

```
1  private void UpdateNoiseInfo()
2  {
3      var noiseLines = GeospatialManager._activeScenario.noiseLines;
4      var pose = GeospatialManager.pose;
5      int decibel = 0;
6
7      for (int i=0; i<noiseLines.Length; i++)
8      {
9          if (IsCoordInPoly(pose.Latitude, pose.Longitude, noiseLines[i
               ].coords))
10         {
11             decibel = noiseLines[i].value;
12             break;
13         }
14         else
15         {
16             decibel = 0;
17         }
18     }
19 }
20
21 private void UpdateShadowInfo()
22 {
23     var shadowLines = GeospatialManager._activeScenario.shadowLines;
24     var pose = GeospatialManager.pose;
25     int shadowTime;
26
27     ...// Analogous as above
28 }
```

Listing 14: Checking if the current device position is in any noise or shadow contour lines from the data.

Our noise and shadow cast data, i.e. the ContourLine objects (line 21, Listing 7), are arranged in descending order according to their values in advance. To determine noise level and shadow cast duration, we iterate through the contour lines from higher values to lower values, so that once the current position is within the bounds of a contour line, the iteration can be stopped. With the PNPOLY algorithm[41], if a current position is directly on an edge of polygon, it is not considered as in the polygon and will be assigned the next lower value if a lower value exists.

4.6 Gamification Contents

The hot air balloon button in the app is the access point to our nice-to-have gamification contents (Figure 10). In this version of our application, the nice-to-have requirements listed in Table 3 are only partially functionally implemented due to their

[41] https://wrfranklin.org/Research/Short_Notes/pnpoly.html

low priority. When a user clicks on the hot air balloon button, a multi-page panel will be shown, where the user can swipe left and right on the top and bottom parts of the panel to switch pages, and swipe up and down in the content area to scroll the page. Each page should represent a different kind of "game" the user can experience: one page for visually appealing 3D events, one page for stars collecting game, and potentially more game ideas in the future.

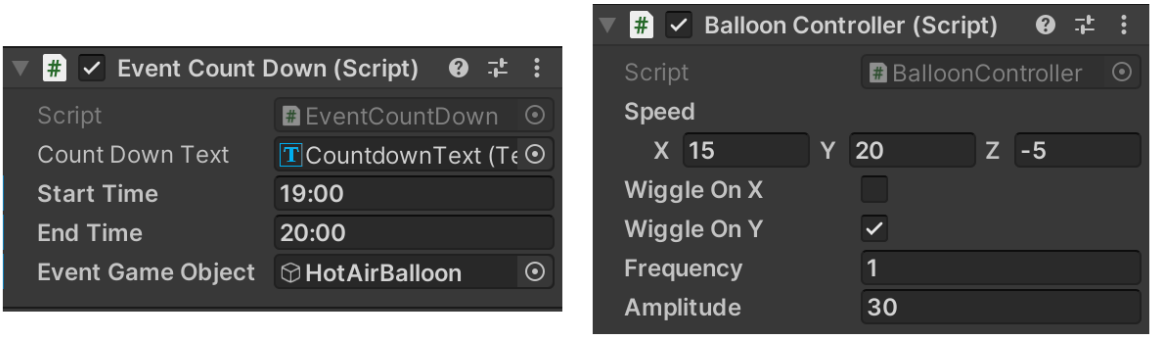

Figure 7: Serialized fields of the EventCountDown script attached to the event card prefab and the BalloonController script attached to hot air balloon game objects. EventCountDown controlls the text shown in "Count Down Text" and the visibility of the "Event Game Object" according to the given event start and end time. BalloonController controlls the general speed and direction of a balloon object and the sin-function wiggle frequency and amplitude on either X or Y axis.

The 3D events are displayed on the first page. On the top, the user can toggle the visibility of these events on and off with a toggle. The events are switched off by default at app start. Below the toggle, a list of possible 3D events are supposed to be shown. Each event contains an image icon of the event, the text description to the event, and a count-down to the time point the event is going to happen, because different events may happen at, for example, different times of a day or week. We store this combination of UI objects into a prefab that will be referred as an event card here onward. Also, an individual controller script is attached to the event card in order to calculate and show the user the countdown time, to link the event card with the event's 3D objects and to show or hide the corresponding 3D objects on screen. Figure 7 shows the serialized fields of the script, or in another word, the reconstructable fields. In this version of the app, the start and end time of an event can be directly entered by developers in the serialized fields in Unity Editor.

Currently, we only implemented a hot air balloon event as an example, because further events such as animals running around might involve purchasing 3D models with more complex animations. For the hot air balloon event, we downloaded a free model[42] and computed its animation directly with a script. One can define the start position of a

[42]https://free3d.com/3d-model/hot-air-balloon-v1–156268.html

hot air balloon in the "transform" component of its 3D model prefab. Additionally, our script act as a component where the moving speed and linear direction of the balloon can be freely adjusted. We also add an extra wiggle movement of hot air balloons by sinus curves. For example, if a balloon is moving mostly in an upwards direction, we can turn on the wiggle on the X axis, if moving horizontally, then Y axis, to make the movement more intricate. The serializable fields are as shown in Figure 7, the amplitude, frequency and axis for the sinus wave movement, and speed of the general movement can be defined individually for each hot air balloon in the editor. Because the position of the balloon is different in each frame due to the movement, it has to be updated as in Listing 15.

```
1 void Update()
2 {
3     transform.Translate(Speed * Time.deltaTime, Space.World);
4     var wiggle = Mathf.Sin(Time.realtimeSinceStartup * Frequency) *
          Amplitude;
5
6     if (WiggleOnX) transform.Translate(wiggle, 0, 0, Space.World);
7     if (WiggleOnY) transform.Translate(0, wiggle, 0, Space.World);
8 }
```

Listing 15: Update the position of a hot air balloon each frame according to serialized fields variables.

The second page of the multi-page panel is a placeholder of the star-collecting game, for which only a part of the interface is implemented.

4.7 Integration with the Ionic Project

The main WindFarm app is developed with the Ionic framework and our AR application serves as a part of the main app. Ionic is a development toolkit for building cross-platform applications from a single code base written with web development technologies. In this project, we focused on the integration for the Android platform. The integration consists of two critical steps: merging the two projects into one Android native project, and bridging the web interface of the Ionic project and the Unity AR app.

In order to merge the applications, we first have to export both of them into native Android projects to create an intersection point, and then, our AR application is plugged into the main app as a library.

By default, native Android projects are handled by the Gradle build system and the main programming language is Java. To import the Unity project into the Ionic project, we open the exported native Ionic project in the Android Studio IDE and edit its *settings.gradle* file. The exported Android native Unity project has two modules,

launcher and *unityLibrary*. The former is a simple launcher wrapper, and the latter contains our AR application contents. We only need to import the latter, as our AR application will not be launched as an standalone app but only by the Ionic app. We import the *unityLibrary* module by directly pointing to its folder path (Listing 16). In this way, any future changes to the Unity project will be automatically recognized.

Furthermore, we discovered that the Unity project not only has a minimum requirement for the Gradle version[43], but also might be incompatible with the newest versions depending on the Unity Editor version. So we might have to adjust the Gradle plugin and distribution version of the main (Ionic) project in the project structure. Currently, we developed with the Unity Editor 2021.3.5f1 and set the Gradle plugin version to 7.1.3 and distribution version 7.3.1. Unity settings in the *gradle.properties* file should also be copied over to the main project.

Then, we have to implement the *unityLibrary* module as a dependency in the Ionic app level *build.gradle* file in order to be able to call it (Listing 17). Because the dependencies of the two exported projects might partly conflict with each other or have duplicates, we have to comment out the conflicting implementation references in the

```
1 include ':unityLibrary'
2 project (':unityLibrary').projectDir = new File('{localPath}/unity/
    unityLibrary')
```

Listing 16: Import the Unity project by importing its "unityLibrary" sub-folder. These lines are added to the "settings.gradle" file of the native Ionic app.

```
1 repositories {
2     flatDir{
3         dirs '{localPath}/unity/unityLibrary/libs', 'libs'
4     }
5 }
6 dependencies {
7     implementation project(':unityLibrary')
8 }
```

Listing 17: Add the "unityLibrary" dependency in the app level "build.gradle" file and link to the module's "libs" folder.

unityLibrary level of *build.gradle* file and manually delete the .jar files from the *libs* folder to resolve the errors.

After solving dependency problems, we move on to creating a bridge between the two projects. Capacitor[44] is a cross-platform native runtime that is installed with newer versions of Ionic by default. Only with Capacitor, were we able to export and deploy the Ionic project natively. We want to be able to open the AR app by clicking on a

[43]https://docs.unity3d.com/Manual/android-gradle-overview.html
[44]https://capacitorjs.com/docs

button in the Ionic app's interface as listed in the requirements (Table 3) and this can be done by creating a Capacitor plugin which enables Javascript to interface directly with native APIs.

As shown in Listing 18, our plugin is a Java class extending the Capacitor Plugin class. The "name" attribute inside the *@CapacitorPlugin()* annotation indicates the name of the plugin to be registered. The frontend Javascript also accesses the plugin according to this name and will be able to call the methods annotated with *@PluginMethod*. An Android Intent is an abstract description of an operation to be performed and is often used for binding codes in different applications[45], as it can be used to launch activities from different sources. The *UnityPlayerActivity* of a Unity Android application is responsible for basic interactions between the Android operating system and the application[46], and is our accessing point to the AR app. What we do in the *startAR* method is basically wrapping the Unity native functions in an Intent to start the Unity app.

```
1  @CapacitorPlugin(name = "UnityActivity")
2  public class MainUnityActivity extends Plugin {
3
4      @PluginMethod
5      public void startAR(PluginCall call) {
6          Intent arIntent = new Intent(this.bridge.getWebView().
               getContext(), UnityPlayerActivity.class);
7          arIntent.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
8          this.bridge.getWebView().getContext().startActivity(arIntent)
               ;
9      }
10 }
```

Listing 18: Capacitor plugin that acts as a wrapper for native functions to be called by Javascript. This UnityActivity plugin enables the Ionic frontend Javascript to start the Unity AR app.

To register the plugin, we need to override the *onCreate* method of the exported Ionic *MainActivity* class (Listing 19). This is where the Ionic app starts.

```
1  public class MainActivity extends BridgeActivity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          this.registerPlugin(MainUnityActivity.class);
5          super.onCreate(savedInstanceState);
6      }
7  }
```

Listing 19: Register plugins in the MainActivity class of the exported Ionic project.

[45]https://developer.android.com/reference/android/content/Intent
[46]https://docs.unity3d.com/Manual/AndroidUnityPlayerActivity.html

To call the plugin method from the frontend interface, we have to also register the plugin there according to the natively registered name (Line 2 of Listing 20), and then simply call the *startAR* plugin method.

```
1  import { registerPlugin } from '@capacitor/core';
2  const UnityActivity = registerPlugin('UnityActivity');
3
4  export default {
5      methods: {
6          async startARApp() {
7            UnityActivity.startAR({message: '...'});
8          }
9        }
10 }
```

Listing 20: Frontend code for registering the plugin and using the plugin method.

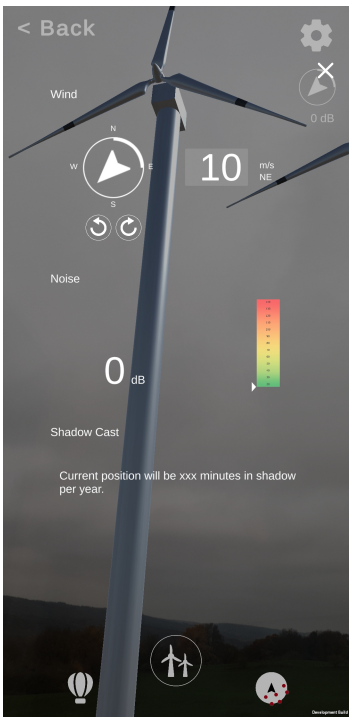

Figure 8: Information panel as an see-through overlay, so that the user can see the camera feed and AR contents, especially immediate changes after setting wind speed and direction. This panel contains buttons and input fields for wind attributes, as well as noise and shadow cast information about the current position.

Figure 9: Information pop-up window shown after clicking on a windmill object.

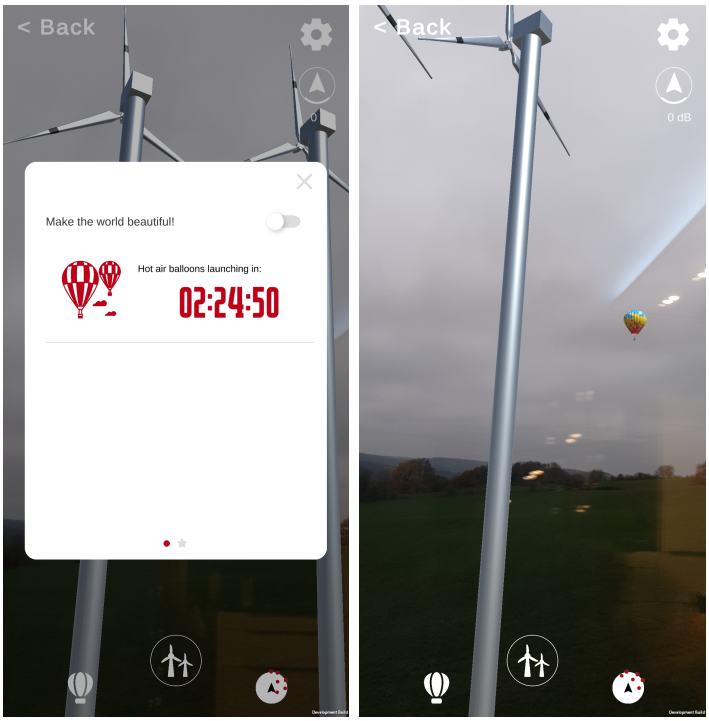

Figure 10: The screenshot on the left shows the game panel that is opened by the balloon button on the bottom left side of the screen, the one on the right shows the hot air balloon event. When ”Make the world beautiful” is toggled on, 3D events will happen at appointed time indicated by the event count down on the panel.

5 Evaluation

In this chapter, we evaluate our application in two ways. First, in Section 5.1, we evaluate the precision of the geospatial placement functionality of ARCore's Geospatial API that we implemented in our application. The quality of geospatial placements depend on both the detected position and orientation of the device. We conducted quantitative studies to evaluate these two factors. In the second test, described in Section 5.2, we compare augmented wind turbines with existing real-world turbines visually by placing the augmentations exactly according to established turbine data. This allows us to evaluate our application qualitatively.

5.1 Quantitative Evaluation of the Geospatial API for AR Content Placement

The Geospatial API provides advanced solutions for detecting devices' position and heading direction with the VPS system as mentioned in Section 4.1. With the added benefit of CV and ML methods through cloud computing services, we observed the Geospatial API exhibiting overall rather good performance. In this test, our goal is to validate our observation quantitatively and find out if Geospatial API really provides better results for placing geo-located virtual content. We want to evaluate the quality of the orientation estimations and understand to what extent and under which conditions the position detection is superior to using GPS data alone. By conducting a thorough evaluation, we can better understand the capabilities and limitations of the Geospatial API for placing virtual content in the real world.

5.1.1 Position Detection

In this evaluation, we aim to determine the precision of the Geospatial API's position detection and compare it to GPS and the Location Service provided by our development tool Unity.

According to Google's documentation [47], Google Street View has three photo types: photo sphere, photo path and street view. They provide the initial raw data for the point clouds of the localization model that the VPS depends on.

- **Photo Sphere** is the type of single 360°panorama photos taken individually without connecting to other nearby photos through navigation.
- **Photo Path** is where multiple non-360°photos near each other are connected through navigation.

[47] https://support.google.com/maps/answer/10443241?visit_id=638060097983000160-2966048377&p=sv_imagekey&rd=1

- **Street View** is where multiple 360°photos are connected through navigation. Google's Street View cars drive through almost all streets in countries and cities that allow such image data collection. Private Street View Studio users can also upload photos.

Due to the different number of photos and the navigation information, the quality of point clouds learned from the three photo types can obviously vary. We also want to observe how the photo types influence the positioning accuracy of the Geospatial API.

Thus, we hand-picked 11 test points around Aachen and noted their latitude/longitude coordinates according to two criteria: the Street View type, and distinct visual cues on Google Maps with satellite view. Three test points are captured by street view photos, two are along photo paths, three near photo spheres, and three near no photo references. All test points are under open sky. In order to determine the precision of all positional data, we have to position ourselves at the test points as exactly as possible to minimize the error contribution from ourselves to actually measure the offsets produced from the data. So all test points are easily distinguishable from the environment, for example, the center of a manhole cover, the corner of a building etc. The coordinates of the test points are acquired by right-clicking on Google Maps and they have 15 decimal degrees of accuracy. For our test, we decided for 6 decimal degrees of accuracy for our test points input, corresponding to about 0.1 meter of distance for both longitude and latitude [48].

Our test application places an object using the test point's latitude/longitude coordinates and the first detected device altitude by the Geospatial API, as the altitude information provided by Google Earth is too inaccurate. The test application logs the device's positional data estimated from both the Geospatial API and Unity's Location Service. It also logs the horizontal offset distance between the test point's coordinates and each estimated coordinates calculated by the haversine formula as shown in Equation 1[49], where d is the distance between the two points, r is the radius of the earth, $lat1$, $long1$, $lat2$, $long2$ are the latitude and longitude values of the first and second point respectively. Additionally, the vertical offset (difference between initial and current estimated altitude) and overall offset of the placed object are also logged. The GPS values are collected by our device's built-in compass application.

$$d = 2r \cdot \arcsin\sqrt{\sin^2\left(\frac{lat2 - lat1}{2}\right) + \cos lat1 \cdot \cos lat2 \cdot \sin^2\left(\frac{long2 - long1}{2}\right)} \quad (1)$$

For the data collection, we place ourselves with the device directly at each test point. Standing still, we then turn on our test application. After the first estimation values

[48] https://en.wikipedia.org/wiki/Decimal_degrees
[49] https://en.wikipedia.org/wiki/Haversine_formula

are registered, we rotate around ourselves slowly for 360 degrees for the Geospatial API to capture more surrounding information. After that, we wait for another 10 seconds without moving to make sure the values are stable, save the values, restart the test application and repeat the same procedure for 5 times on two days separately. On one day the sky was clear and the other almost completely covered by clouds. The GPS data from the built-in compass is also taken 10 times, each with a re-estimation.

Table 5 shows our test results. We can observe that the Geospatial API has the highest accuracy for estimating in the horizontal space, and the Unity Location Service the worst. On average, the Geospatial API produces a horizontal position error of under only 3 meters, the GPS about 20 meters and the Unity Location Service about 80 meters. The GPS data results are mostly relatable to existing researches [18, 19] stating that a level of accuracy around 20 meters is commonly due to multipath errors, mainly influenced by landscape characteristics. Test point 2, 7 and 8, having an average error of over 20 meters, are for example near tall buildings or surrounded by leaf-on trees.

Test point 11 is an interesting position, as both the lowest and largest error from the GPS were measured there and no other test point has such a large difference of over 250 meters. This point is on the edge of an open field, near to only a few leaf-off trees. The errors from the Unity Location Service on this point is also very high, all positional estimation are more than 200 meters off, with an average of over 300 meters. From the results of the Location Service, a total of five points have an average error of more than 50 meters (test point 4, 5, 6, 9, 11). Of those points, point 4 is on an outdoor parking lot, point 6 is on an empty sports field and point 9 is in a park. Point 5 is also worth noting, as the errors measured with a clear sky were all around only 6 meters, but then on a cloudy day all above 1 km, and the results are independent of the internet.

We are very surprised by the results produced by Unity's Location Service, as it is very far-off from the actual GPS data of the same device. We looked into this issue online and found some reports of same large errors on the Unity forum, but this issue is still unresolved.

As for the performance of the Geospatial API, we observed the best results for test point 1 to 3 that are captured in photos of the "street view" type. Among all the samples taken for these three test points, the largest horizontal error is under 3 meters, the average horizontal errors are under 1 meter and the average overall errors are about 2 meters only. However, no distinct difference can be observed between test points of the type "photo path", "photo sphere" or "none".

It is to be noted that the vertical error we measured are actually only an indicator of how stable the altitude estimations are. We observed a very stable estimation of horizontal position from the Geospatial API. The first latitude/longitude estimations remained mostly stable throughout the session, but the estimated altitude changed a lot just as shown in table with an average of more than 7 meters. This observation tells

				GPS			Unity Location Service			Geospatial API								
Test Point	Street View	Latitude	Longitude	Horizontal Offset			Horizontal Offset			Horizontal Offset			Vertical Offset			Overall Offset		
				Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
1	streetview	50.778007	6.078776	11.6	5.1	23.9	20.2	7.02	34.3	0.9	0.2	2.9	0.6	0.0	1.4	1.2	0.2	3.0
2	streetview	50.777520	6.078851	22.5	9.8	39.7	18.1	15.18	21.9	0.8	0.2	1.5	0.8	0.2	1.6	1.2	0.3	2.2
3	streetview	50.771641	6.090788	19.9	6.9	29.8	4.7	1.73	4.7	0.2	0.1	0.3	3.7	0.0	11.0	3.8	0.1	11.0
4	photopath	50.775880	6.039089	2.1	1.4	3.3	80.7	59.5	95.3	3.7	0.5	5.8	8.7	0.7	13.9	9.4	1.4	15.8
5	photopath	50.775803	6.043076	3.1	1.0	4.9	169.3	6.1	1477.8	1.8	0.2	3.8	12.0	0.5	20.0	11.5	2.2	20.2
6	photosphere	50.778040	6.069551	11.7	10.4	12.6	80.1	59.2	117.7	2.9	1.1	5.3	10.4	0.9	18.1	10.5	3.1	18.3
7	photosphere	50.776415	6.083611	49.1	39.9	74.8	9.77	2.0	26.0	4.9	4.6	5.1	0.3	0.1	0.4	4.9	4.6	5.1
8	photosphere	50.778500	6.079637	32.2	25.4	44.1	40.9	35.1	49.8	3.0	0.3	12.3	17.1	0.1	21.4	18.0	1.5	24.7
9	none	50.778372	6.062884	11.1	4.5	31.6	143.1	56.6	281.2	1.6	0.8	3.2	8.8	7.7	9.4	9.0	8.4	9.5
10	none	50.776150	6.069349	9.6	3.0	40.7	4.06	2.6	5.6	5.8	1.7	12.0	12.5	8.4	26.4	14.5	9.7	26.4
11	none	50.775080	6.035335	67.9	0.7	284.0	317.2	229.3	485.7	2.3	2.0	2.4	5.4	5.5	4.9	5.8	5.5	6.0
Mean				21.9			80.7			2.5			7.3			8.2		

Table 5: Position detection test results for 11 test points around Aachen. For each test point, we show the horizontal offset between the test points coordinates and measured coordinates from GPS, Unity Location Service and Geospatial API. The Geospatial API places an object at the test point's given latitude and longitude, and the altitude is the detected device altitude when placing the object. The vertical offset thus indicates how stable is the altitude estimation of the API. The overall offset calculates the general distance of the placed object from the actual test point. The distance unit is meter. The grey cells notes the minimum and maximum values of all horizontal offsets measurements from one source.

	Point 1		Point 2			Geospatial API		
Test Site	Latitude	Longitude	Latitude	Longitude	Bearing	Heading	Heading Acc.	Error
1	50.778966	6.06902	50.779136	6.069905	73	74.3	3.6	1.3
2	50.779008	6.062531	50.779322	6.062801	28.3	30.1	1.6	1.8
3	50.782115	6.069474	50.782452	6.068933	314.3	-47.5	6.1	1.8
4	50.778007	6.078777	50.777886	6.078595	223.3	-137.3	1.3	0.6
5	50.777567	6.078771	50.777522	6.078848	132.4	131.2	2	1.2
6	50.775885	6.042464	50.775883	6.0422	269.2	-87.7	4.2	3.1

Table 6: Orientation estimation test results for 6 test sites around Aachen. For each test site, we selected two points and recorded their latitude and longitude values. The ground truth is the bearing of point 2 referencing point 1 and was calculated using the haversine formula. The Geospatial API places an object at each point. The estimated heading and heading accuracy are collected by measuring the direction of the device at point 1 facing point 2, reported by the API. The values are sampled 5 times for each site and we recorded the average. The error column shows the absolute degree offset between the ground truth bearing and the average reported heading.

us that the virtual objects should be shown on screen to users only after the positional values has become stable.

It's also worth pointing out that during our test, the internet was unstable near the country border and the Geospatial API could not be used about half of the times. It directly returned an internal error reported by *Earth.EarthState* and cannot be recovered without a restart of the application.

To sum up, we observed a better accuracy of horizontal position estimation from the Geospatial API according to geospatial coordinates than both GPS and Unity Location Service. The VPS system improves the localization of the Geospatial API for about 5 times better for test points of "street view" type with an average horizontal error of under 1 meter, whereas test points of other types have on average around 3 meters of error.

5.1.2 Orientation Estimation

The orientation of a device, as returned by the Geospatial API, is known as the "heading". Heading normally refers to the direction that an object is moving or facing, measured in degrees from a reference direction. The Geospatial API uses the true north as the reference direction and the heading ranges from $-180°$ and $+180°$, with 0 indicating the device facing true north, and positive values indicating clockwise directions from north until south. Bearing, on the other hand, refers to the direction of an object from a specific point of reference with respect to true north and ranges between $0 - 360°$.

To evaluate the precision of the Geospatial API's orientation estimation, for each test site, we selected two reference points to form a line. Similar to the previous study, all

test points are under open sky, and the coordinates are acquired again using Google Maps. As the ground truth, we calculate the bearing from point 1 to point 2 with the haversine formula as shown in Equation 2, where θ is the bearing angle in radians, $\triangle$ is the difference of the longitudes of the two points, $lat1$ and $lat2$ are the latitude values of point 1 and point 2 respectively.

$$\theta = \operatorname{atan2}\left(\sin \triangle long \cdot \cos lat2, \cos lat1 \cdot \sin lat2 - \sin lat1 \cdot \cos lat2 \cdot \cos \triangle long\right) \quad (2)$$

To minimize the potential for error caused manually by ourselves, we attempt to align the device camera with the line connecting the two points as accurately as possible. To help with this, we added a fine vertical line to the UI of the test application that is positioned exactly in the middle of the screen. The application places a 20 cm thin and 100 m high semi-transparent magenta cylinder at each of the two reference points according to the previously acquired coordinates with the Geospatial API. For data collection, we position ourselves first roughly at a close distance of several meters from point 1 facing the direction of point 2, and turn on the test application. Then, we move around and adjust our position at a normal walking speed until the two cylinders are aligned with the middle line on the screen as closely as possible and log the heading given by the API. In the best case scenario, the line overlaps completely with the vertical center line of both cylinders. Our device is held in portrait mode with the camera facing forward the whole time when the application is on. The position of the device relative to the two points are illustrated in Figure 11 and the screenshot of a successful alignment is shown in Figure 12.

We collected data from six sites, with five data samples collected at each site. During our tests, we noticed that the reported heading accuracy from the Geospatial API was often quite large at the start of the application, and the cylinders appeared to be at wrong positions. Therefore, each time we had to walk around to align the camera with the cylinders, but we noticed that the reported accuracy always improved during this process.

This improvement is significant, as shown in Table 6. The error column calculates the absolute difference of the bearing and the average heading by first transforming heading values into the same range as the bearing. (Negative values plus 360 degrees.) The performance of the Geospatial API appears to be generally good, with relatively small differences between the haversine bearing and the heading estimations provided by the API. In most cases, the heading accuracy estimated by the Geospatial API is more pessimistic than our measured error. But overall, the largest average error of the six sites is about 3 degrees, suggesting good quality of the estimations.

However, we have to point out that only test site 4 and 5 are located on crowded streets, both of these sites are on streets with full Google Street View coverage (type street view), and none of the other test sites is near many physical obstructions. Test

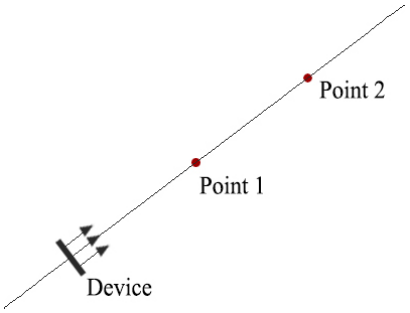

Figure 11: The position of our device for orientation estimation data collection. We stand on the connecting line through both points and the device faces the point 2.

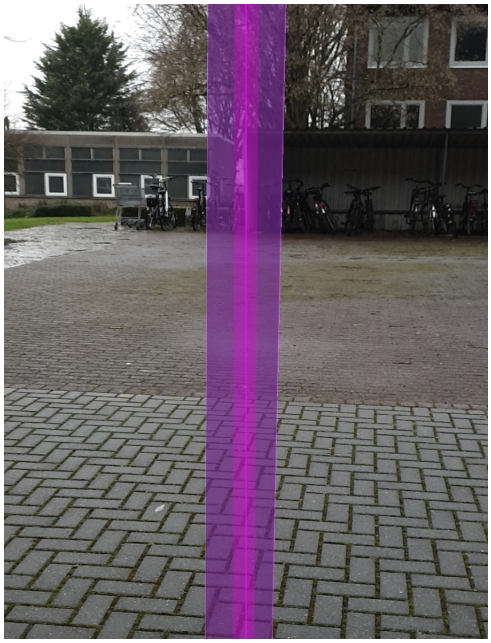

Figure 12: Screenshot of a successful alignment example of the device and the two cylinders at a test site. The line in the middle of the screen, the long-thin form and the transparent color of the cylinders help us to better align on the line.

sites 1 and 2 are located near tall buildings on one side, but there are few obstructions on the other side. These factors could have contributed to the good results.

This test also suggests that the localization provided by the Geospatial API can be improved if the user walks around. One possible explanation for this is that when the device is moving, the Geospatial API can use data from the accelerometer, gyroscope, and other sensors to estimate the device's movement and orientation, which can improve the accuracy of the location estimation.

Based on both quantitative tests, we can conclude that the Geospatial API indeed provides good results. Its performance for positioning is also superior to both GPS and the Unity Location Service, but only when the device is connected to the internet.

5.2 Comparing AR and Real-World Wind Farms

For our second test, we found information about nearby established wind farms from the NRW State Office for Nature, Environment and Consumer Protection[50] and selected the wind farm near Laurensberg as our test site. Figure 13 shows the geographical information of our test site. We preprocessed the information of the selected 12 turbines denoted in cyan to fit our JSON format (Listing 6) and created a new scenario. The turbines are selected according to the distance to our planned movement path for testing, namely, within 3 kilometers.

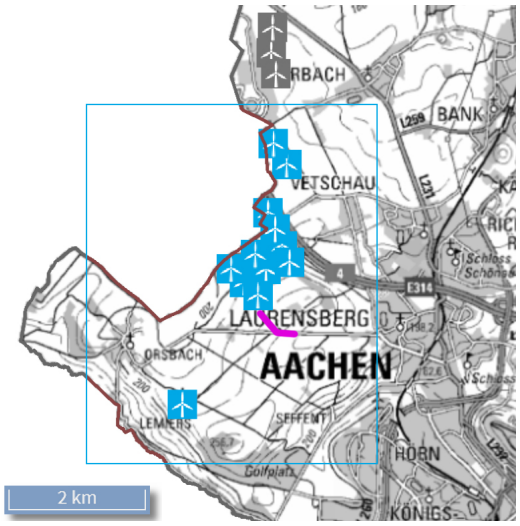

Figure 13: Real world wind turbines at the test site. All nearby turbines positions are noted with a wind turbine icon, but only the cyan-colored ones are visualized for our test scenario that are within 3 kilometers from our walking path. The magenta path denotes roughly our test path.

[50] https://www.energieatlas.nrw.de/site/bestandskarte#

The turbine information provided by the state includes turbine id, manufacturer, type of turbine model, power, hub height, diameter, horizontal position etc. The height of our 3D turbine model is adjustable and can reflect the real-world height realistically. However, altitude information is not given in this database. During our development process, we already found out that the altitude of a position provided by Google Earth and that measured by the Geospatial API can have a large difference ranging from 2 meters to more than 10 meters. As horizontal position error is very noticeable on screen, especially when the virtual objects are floating in the sky, we added a simple functionality to adjust the altitude of all virtual turbines by typing in altitude values manually.

Figure 14 shows screenshots taken during our test after manually adjusting the altitude of virtual turbines. The left screenshot illustrates turbines at a far distance of about 2 kilometers away from the device camera, and the turbines on the right are middle-distanced about 500 to 800 meters away. The white turbines are real-world turbines captured by the camera and the gray ones in front of them are virtual 3D models rendered on top of the camera feed. From the screenshots, we can see that the generated turbines and real turbines almost overlap with each other, and the hub height of the turbines are also rather correct. The reason why some of them appear higher than the real-world references are either because of the wrong altitude, (we set all turbine altitude to the same value for simplicity, but the terrain is not completely flat), or because the real turbines are partly occluded by the terrain.

We also walked to the base of a turbine to observe a close-distanced visualization as shown in Figure 15. The horizontal accuracy of the device pose reported by the Geospatial API is around 2 meters, and the vertical accuracy about 1.5 meters. But by pointing the camera to the bottom of the turbine, we would say that visually, the virtual turbines looks about only 0.5 to 1 meter away from the real one.

However, we observed some unrealistic movements of the placed 3D objects. Figure 16 shows two screenshots taken 1 second apart while we weren't moving the device. From inspecting the values on the debug console, we found out that the current pose position or heading given by the Geospatial API changed in these occasions. The re-position happened smoothly instead of abruptly, so the objects looked as if they were dancing around. A re-estimation of the heading direction has more impact on the visualization. Also, because the test location is around the borders and in the suburbs, the internet was unstable. For more than half of the time, our application couldn't work due to the internal error from *EarthState*.

In any case, it is important for the application to work when internet connection is unstable or unavailable. A hybrid solution of combining the Geospatial API with an improved manual GPS method should be implemented in the future, which will be addressed again later in Section 6.3.

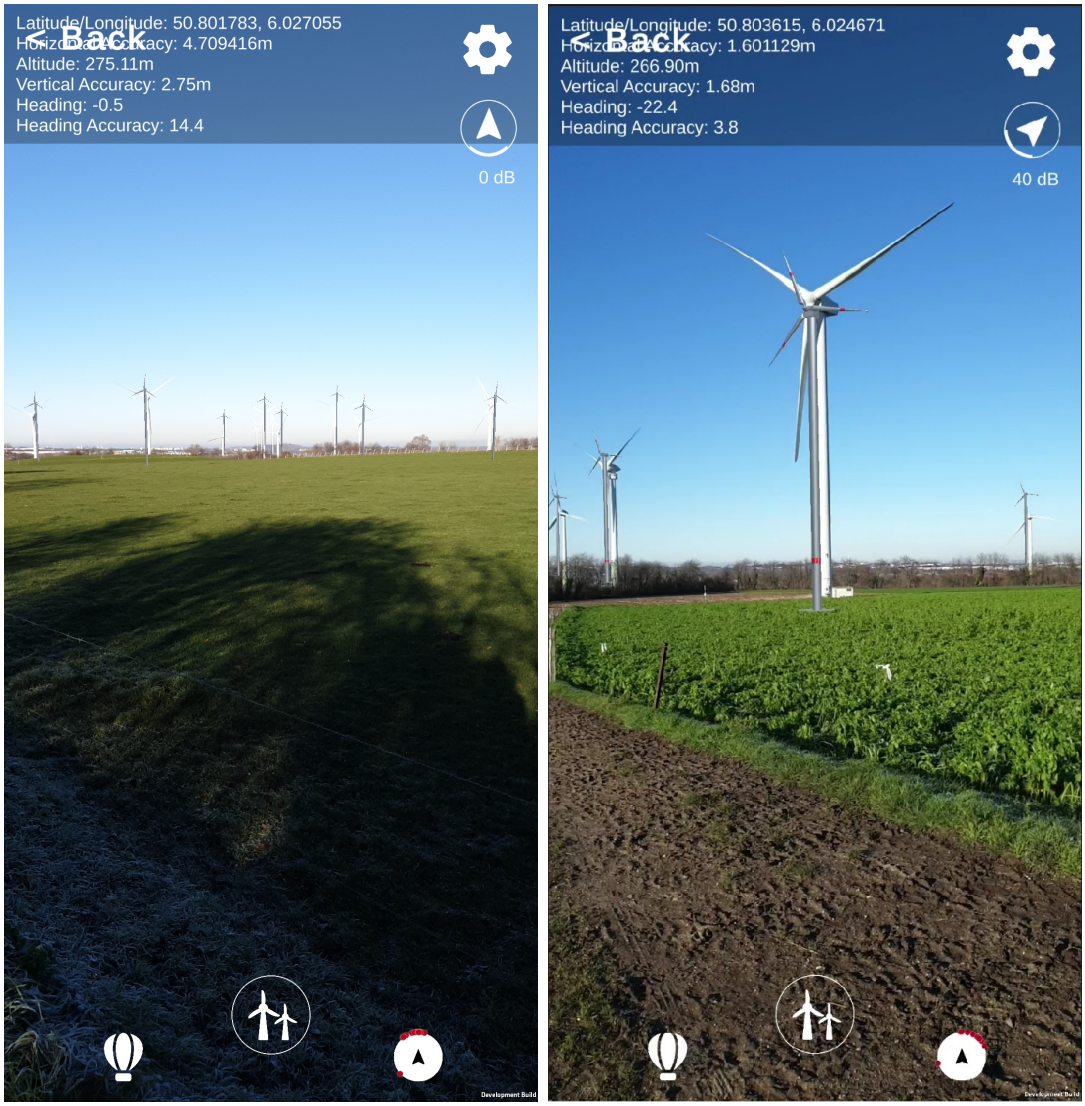

Figure 14: Screenshots taken during the test. The left one shows a far-distanced view, the right one a middle-distanced view. The white wind turbines are real world turbines. The gray turbines are virtually rendered on top of the camera feed. The metrics shown above are information from the Geospatial API about the current pose of the device only for debugging purpose of a development build.

Figure 15: Screenshot of a virtual turbine generated in place of a real turbine at a close distance.

Figure 16: Partial screenshots of the same turbines from a far distance with 1 second of time offset during testing. Because of a re-estimation of the device heading, the virtual turbine was horizontally moved on screen.

6 Conclusion

In this project, we developed an AR application for planned wind farms, visualizing wind turbines according to real-world GPS coordinates realistically in hope to increase the social acceptance of wind farm projects by nearby citizens. Having some projects and work on the wind farm subject laid out by the research group, we engineered our application carefully respecting both the architecture and usability aspects, so that our work can serve as an effective linking point for previous and future work. In this chapter, we summarize the contribution of our work done during research, development and evaluation phases in Section 6.1. In Section 6.2, we review the difficulties and challenges encountered in this project and finally, we lay out the improvements that should or can be done in the future in Section 6.3.

6.1 Conclusion

Aiming to build a mobile AR application that shows realistic visualizations of planned wind farms, in this thesis, we kept two research questions in mind with respect to the design, implementation and evaluation of such an application.

The first question is, how should a mobile AR application for wind farms visualization look like? To answer this general question, we can split it into further sub-questions: (1) Which functionalities should such an application have? (2) How should its user interface look like? (3) Is it possible with current technologies? (4) How to implement the functionalities with current technologies?

We looked into relevant researches in Section 1.2 in Chapter 1, but among the work that mentioned the application of AR, we found only few of them related to geo-located content positioning. In Chapter 2, we further inspected a set of promising MAR development frameworks extensively. Thereby, we are able to answer the sub-question (3) with a yes and lay out the foundation for a successful implementation. We selected Unity and its AR Foundation framework as our development tool mainly according to the following criteria: performance, multi-platform support, development cost, extensibility for occlusion with machine learning and integrability with an existing multi-platform app from the research group.

We proceeded with the application development systematically by following established software engineering standards. In Chapter 3, we adopted the prototype software engineering model, collected the requirements for the application that answers the sub-question (1), laid out a high-fidelity presentation prototype that represents the final UI closely, answering the sub-question (2), and chose the MVC pattern as the basis for our code architecture. Chapter 4 describes our functional implementations in detail that answers (4). By proposing and developing an example mobile AR application visualizing wind farms, our first research question is answered with a feasible approach.

The most important function of our AR application is placing wind turbines at real-world coordinates, so the correct estimation of the device position and orientation is vital. We evaluated the performance of the position and orientation estimation of our solution in Chapter 5 both quantitatively and qualitatively to find out how well the virtual wind farm can resemble a real wind farm. Quantitatively, we measured the precision of the positional and orientation data provided by the Geospatial API and compared its positional data to those provided by Unity's Location Service and the GPS data from the device. Qualitatively, we visualized an existing wind farm and compared the virtual objects with the reality visually. By that, the second research question: "how should an AR application for wind farms visualization be evaluated" is answered partially, as an evaluation of the whole application requires extensive user studies. We will propose a usability study next in Section 6.3.

6.2 Challenges and Difficulties

During the development of our application, we encountered quite a few challenges, a big part of which is because of the scarcity of information we can get for some specific problems, for example the making of a hybrid application by integrating our Unity app with the Ionic app. Of the little information we could find on this topic, most of them are outdated, which we discovered only after a great number of trial-and-errors.

Similarly, ARCore's Geospatial API is a very new feature, its plugin for AR Foundation was even only published after we started this project, so even less information can be found. In the previous half of our project, its official documentation was even partially incomplete. Fortunately it was improved with time.

Generally, we also experienced longer iterations of feature implementation, testing and debugging cycles when developing AR application for smartphones with AR Foundation, or ARCore specifically, because only certain Android devices fulfill the hardware requirements and we have to build and export the application to device each time to inspect the changes made to the code. This can take between 2 minutes to more than half an hour with Unity, slowing down the development process largely.

6.3 Future Work

In this section, we first propose a usability study for our AR application, as it provides insights into how well the system accomplishes its goal and how it can be improved and further answers our research question one. After that, we discuss further future work.

Usability Evaluation Study Proposal

We propose a usability study according to the usability principles laid out by Ji et al. for our AR application [13], in which its intuitiveness, effectiveness, efficiency, feedback support, and users subjective satisfaction should be evaluated.

- Intuitiveness: Users should have a nearly effortless understanding of the navigation and controls of the application. This metric encompasses familiarity, predictability and consistency. The user interface should adhere to general conventions and comply with user's expectations. The icons, terms and layouts in the application should be consistent. Process structures or the chain of actions for tasks should also follow a uniform principle.
- Effectiveness: This is the most important aspect of the usability study. The central question is: how well can users complete their tasks with the provided application system? Suitable functionalities should be provided for users to carry out their tasks so that they won't be at a loss of what to do because of the lacking of information, or be overloaded with excessive information.
- Efficiency: Ideally, users should accomplish a task in minimal time and the system should also react and execute a certain action as quickly as possible. In our AR wind farm visualization case, this also entails for example how quickly the system needs to identify users position and place augmentations accordingly, or how quickly the scenes can be switched.
- Feedback: The application must keep users informed of events actively. This includes actions taken by the system, change of state etc. When an error occurs, a notification explaining the reason should be provided concisely, and if applicable, also the solution to the error.
- Subjective Satisfaction: Users should rate their experience with the application and the tasks they carried out, their feedback and advises should also be collected.

The use cases we described in Section 3.1 can be referenced for tasks for participants. We also suggest the evaluation study to be carried out in form of a usability testing, where observers (we) watch, listen and take notes of participants completing their tasks, because our AR application involves not only clicking and touching on screen, but also more involved actions such as how users hold the device and move around are of interest to us.

Other Future Work

The careful selection of our development tool and the software architecture that we laid out in this project prepared it to be, first of all, extended by obstacle occlusion functionalities. As mentioned earlier, obstacle occlusion is a very important factor for realistic visualizations and should definitely be implemented soon. This is currently being researched by another student project.

Also, because of the internet requirement from the Geospatial API, the use of the app can be restricted, especially when in the most cases, wind turbines are located in the suburbs where internet are sometimes unstable or unavailable. So, it is of utter importance to develop an offline solution. By comparing the horizontal position errors of the Geospatial API, Unity Location Service, and GPS, we can directly rule out Unity Location Service because of its bad and inconsistent performance. A reasonable solution would be to build a plugin for Unity to read device native GPS data, and develop the offline object placement functionality around them.

Apart from this, we chose tools and frameworks that also support the iOS platform. Provided the development hardware, namely a PC with the macOS operating system and an iPad or iPhone as the test device, our AR application can be deployed to the iOS platform with just a few additions to the code base.

The nice-to-haves listed in our requirement matrix (Table 3) can also be fully implemented in the future.

References

[1] M. Arango, L. Bahler, P. Bates, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S.-Y. Wuu. The touring machine system. *Commun. ACM*, 36(1):69–77, jan 1993. ISSN 0001-0782. doi: 10.1145/151233.151239.

[2] Shiri Azenkot, Richard E. Ladner, and Jacob O. Wobbrock. Smartphone haptic feedback for nonvisual wayfinding. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, page 281–282, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309202. doi: 10.1145/2049536.2049607.

[3] Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre. Recent advances in augmented reality. *IEEE Comput. Graph. Appl.*, 21(6):34–47, nov 2001. ISSN 0272-1716. doi: 10.1109/38.963459.

[4] Eleonora Bottani and Giuseppe Vignali. Augmented reality technology in the manufacturing industry: A review of the last decade. *IIE Transactions*, 51:284–310, 03 2019. doi: 10.1080/24725854.2018.1493244.

[5] Dirk Bäumer, Walter Bischofberger, Horst Lichter, and Heinz Züllighoven. User interface prototyping - concepts, tools, and experience. pages 532–541, 01 1996. doi: 10.1109/ICSE.1996.493447.

[6] Jacky Cao, Kit-Yung Lam, Lik-Hang Lee, Xiaoli Liu, Pan Hui, and Xiang Su. Mobile augmented reality: User interfaces, frameworks, and intelligence, 2021.

[7] Jane Cleland-Huang, A. Zisman, and O. Gotel. *Software and Systems Traceability*. 10 2012. ISBN 978-1-4471-2238-8. doi: 10.1007/978-1-4471-2239-5.

[8] European Commission, Joint Research Centre, G Ferraro, and G Ellis. *The social acceptance of wind energy : where we stand and the path ahead*. Publications Office, 2017. doi: doi/10.2789/696070.

[9] R. Courant, H. Robbins, and I. Stewart. *What is Mathematics?: An Elementary Approach to Ideas and Methods*. Oxford Paperbacks. Oxford University Press, 1996. ISBN 9780195105193.

[10] Ivica Crnkovic. Component-based software engineering — new challenges in software development. *Software Focus*, 2(4):127–133, 2001. doi: https://doi.org/10.1002/swf.45.

[11] S Grassi and T M Klein. 3d augmented reality for improving social acceptance and public participation in wind farms planning. *Journal of Physics: Conference Series*, 749:012020, sep 2016. doi: 10.1088/1742-6596/749/1/012020.

[12] Maral Jamalova and Milán Constantinovits. The comparative study of the relationship between smartphone choice and socio-economic indicators. *International Journal of Marketing Studies*, 11:11, 07 2019. doi: 10.5539/ijms.v11n3p11.

[13] Yong Gu Ji, Jun Ho Park, Cheol Lee, and Myung Hwan Yun. A usability checklist for the usability evaluation of mobile phone user interface. *International Journal of Human–Computer Interaction*, 20(3):207–231, 2006. doi: 10.1207/s15327590ijhc2003_3.

[14] Gun Lee, Andreas Duenser, Seungwon Kim, and Mark Billinghurst. Cityviewar: A mobile outdoor ar application for city visualization. pages 57–64, 11 2012. ISBN 978-1-4673-4663-4. doi: 10.1109/ISMAR-AMH.2012.6483989.

[15] Lik-Hang Lee and Pan Hui. Interaction methods for smart glasses: A survey. *IEEE Access*, 6:28712–28732, May 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2831081.

[16] Ziyao Li, Hou Shu, Wei Song, Xinrui Li, Lin Yang, and Shan Zhao. Design and implementation of traditional kite art platform based on human-computer interaction and webar technology. In *2021 2nd International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, pages 209–212, 2021. doi: 10.1109/ICHCI54629.2021.00051.

[17] J. Linowes. *Augmented Reality with Unity AR Foundation: A practical guide to cross-platform AR development with Unity 2020 and later versions*. Packt Publishing, 2021. ISBN 9781838982966.

[18] Krista Merry and Pete Bettinger. Smartphone gps accuracy study in an urban environment. *PLOS ONE*, 14(7):1–19, 07 2019. doi: 10.1371/journal.pone.0219890.

[19] Esmond Mok, Guenther Retscher, and Chen Wen. Initial test on the use of gps and sensor data of modern smartphones for vehicle tracking in dense high rise environments. pages 1–7, 10 2012. ISBN 978-1-4673-1908-9. doi: 10.1109/UPINLBS.2012.6409789.

[20] Vaughn Nelson and Kenneth Starcher. *Wind Energy: Renewable Energy and the Environment*. 11 2018. ISBN 9780429463150. doi: 10.1201/9780429463150.

[21] Nenad Petrović, Vasja Roblek, Merab Khokhobaia, and Ineza Gagnidze. Ar-enabled mobile apps to support post covid-19 tourism. In *2021 15th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)*, pages 253–256, 2021. doi: 10.1109/TELSIKS52058.2021.9606335.

[22] Maša Radenković, Valentina Nejkovic, and Nenad Petrovic. Adopting ar and deep learning for gamified fitness mobile apps: Yoga trainer case study. 10 2021.

[23] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *Interactions*, 3(1):76–85, jan 1996. ISSN 1072-5520. doi: 10.1145/223500.223514.

[24] R. Saidur, M.R. Islam, N.A. Rahim, and K.H. Solangi. A review on global wind energy policy. *Renewable and Sustainable Energy Reviews*, 14(7):1744–1762, 2010. ISSN 1364-0321. doi: doi.org/10.1016/j.rser.2010.03.007.

[25] Geri Schneider and Jason P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201309815.

[26] Xiaoyan. Sun, Shiyuan. Gu, Linfu. Jiang, and Yingfei. Wu. A low-cost mobile system with multi-ar guidance for brain surgery assistance. In *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pages 2222–2225, 2021. doi: 10.1109/EMBC46164.2021.9630928.

[27] TS Tullis. High-fidelity prototyping throughout the design process. In *Proceedings of the Human Factors Society 34th annual meeting*, page 266. Human Factors Society Santa Monica, 1990.

[28] D.W.F. van Krevelen and R. Poelman. A survey of augmented reality technologies, applications and limitations. *International Journal of Virtual Reality*, 9(2):1–20, Jan. 2010. doi: 10.20870/IJVR.2010.9.2.2767.

[29] Nancy J Wei, Bryn Dougherty, Aundria Myers, and Sherif M Badawy. Using google glass in surgical settings: Systematic review. *JMIR Mhealth Uhealth*, 6(3): e54, Mar 2018. ISSN 2291-5222. doi: 10.2196/mhealth.9409.