Technical University of Crete, Greece

School of Electrical and Computer Engineering

An Outdoors Augmented Reality Application for Android, Featuring A Cultural Route Trough Old Town of Chania

Christos I. Panou

Dissertation Thesis

December 2017

Acknowledgments

This thesis is a result of a collective effort. As a token of my appreciation I would like to mention some of the contributors.

First and foremost I would like to thank my Supervisor, Associate Professor Katerina Mania for her advice, guidance and supervision throughout this whole process but also for trusting me and giving me the opportunity to undertake a project that involving imagination and creativity. Next I would like to thank, Assistant Professor Dimelli Despina and Dr.Lemonia Ragia for their invaluable knowledge in the historical context of this project, their advice and the time they invested to see this project to fruition.

I would also like to thank the staff and authors of the Wikitude SDK for their quick responses in the forum threads and of course for the provision of a free educational license to work with their framework.

Finally, I would like to thank my friends and family for their infinite support and patience.

ABSTRACT

In this thesis we present a mobile augmented reality application for cultural heritage sites in the Old town of Chania. The main focus of this work is to provide a means for the 3D on-site visualization and reconstruction of historical buildings in their past state for consumer grade mobile phones. In this application we feature three monuments, the ‘Giali Tzamisi’, an Ottoman mosque in the old harbor of Chania, the demolished Towers at the south side of the ‘Byzantine Walls’ and the ‘Saint Rocco’ Venetian chapel. Advances in mobile technology have brought Augmented Reality to the wider public by utilizing the camera, GPS and inertial sensors present in modern smartphones. Upon visiting these monuments a virtual reconstruction is matched to the user's positions and viewing angle displaying the monument in its past state while moving and exploring the area. Position tracking is performed either by utilizing the phone’s GPS or with a combination of the computer vision capabilities of the chosen AR framework and our sensor implementation. A location aware experience was designed and integrated to ensure that the information is delivered not only on functional but on usability criteria as well. Apart from the 3D reconstructions the users have access to information about various monuments throughout the city. The monuments are categorized in key Historical periods of Crete and the users are urged to explore and classify them to unlock the historical information while earning more points for themselves. The application provides a map which can be used to navigate between the spatially distributed content as well as interact with them to get directions and visit their page. The application connects to a remote database that will allow the users to document their visits as well as store data about the city concerning visit frequency, most liked monuments etc. By combining AR technologies with location aware and social aspects we aim to enhance user experience and interaction with cultural heritage sites and showcase the cultural depth of the city of Chania.

Publications

Ragia L, Dimelli D, Mania K, Panou C, (2017) Outdoors Mobile Augmented Reality Application Visualizing 3D Reconstructed Historical Monuments. The International Conference on Geographical Information Systems Theory, GISTAM 2018

Table of Contents

List of Figures

1 INTRODUCTION

Competition in the tourism industry has brought a rapid development of applications that serve and entertain the visitor. In conjunction with the promotion of monuments, cultural heritage and archaeological sites, these applications act as modern digital guides. Augmented Reality seems more than capable of being integrated in such systems by enriching the communication of information. The increased power, portability and the built-in sensors present in modern mobile phones has led to them becoming an ideal medium for Augmented Reality tourist guides.

In this work, we present the design and implementation of an MAR (Mobile Augmented Reality) application, for Android devices, that provides on-site 3D visualizations and reconstructions of historical buildings in the Old Town of Chania, Greece, superimposed over their real-world equivalent, as part of a smart AR tourist guide. Instead of the traditional static images or text presented by mobile tourist guides, we aim to enrich the sightseeing experience by providing a means to visualize the past glory of these sites in the context of their real surroundings. Taking into consideration the importance and historical value of the monuments, our approach offers the opportunity to interact with them in non-intrusive ways, thus, eliminating the need to interfere with the remains and on-going archaeological research.

Based on a mobile, personalized, location-aware experience taking place in various areas of Chania, Crete, we aim to enhance user experience and interaction with cultural heritage sites and showcase the city’s cultural wealth. The mobile AR application features a database that holds records of various monuments. The database also stores the users’ documentation of their visits and interactions in the areas of interest. User requirements gathering and AR development while located in the challenging outdoors environment of a city, pose significant technical as well as user interaction challenges when compared to generic software development, testing and evaluating process. Reliable position and pose tracking is paramount so that the 3D content is accurately superimposed on real-settings, at the exact position required and is one of the major technical problems of AR technologies. Our system features a geo-location and sensor approach which compared to optical tracking techniques allows for free user movement throughout the site, independent of changes in the building’s structure. Moreover we combined this implementation with a hybrid technique to showcase the capabilities of future and on-development technologies.

The AR mobile application developed aims to provide an easily extendable platform for future additions of digital content requiring a moderate amount of development and technical expertise. The goal is to provide a complete and operational AR experience to the end-user by tacking AR technical challenges efficiently, as well as an insight for future development in similar scenarios.

1.1 Brief Description

We present a Location-Based Augmented Reality application for Android devices that provides a gamified sightseeing experience that aims to challenge and motivate the visitors to further explore and uncover the City's underlying history. The main Goal of our work is the promotion of cultural heritage in the City of Chania, Crete. Our App consists of various screens that aim to facilitate different kinds of interactions and content presentation. As a location-based application the main functionality is incorporated in a digital map and through an Augmented Reality camera View.

The main screen of our application is the map showing the location of the user, the available Points of Interest as well as the buttons to navigate through the remaining screens of the app. The aim of the map is to help the user navigate in the city and through our spatially distributed content. This navigation can also be accomplished via the Camera view where the Points of Interests are displayed in 3D space on the camera's surface. In its initial state, the user is shown all the available monuments that contain 3D reconstructions as markers on the map. The path between them is shaded with polylines to visually integrate these points. Upon visiting the monuments the user has access to the 3D reconstructions through the AR camera. The initial monuments act as an introduction to the overall game. After the user visits a monument he is awarded a number of points and when he has visited all the available reconstructions the game enters its main state.

In this state we present all the interest areas of our system as question marks on the map. The goal is to visit all interest areas and unlock them to earn more points. To unlock a Point of Interest the user has to correctly classify it to the given key historical periods of the City of Chania. All information about the periods and the monuments can be acquired form the additional screens of the application. When a user correctly unlocks a monument he also gains access to its historical information. The aim of this approach is to urge the user to closely observe these monuments, consult the information he has already unlocked or even interact with the locals to get as much information as he can. We aim to make the user an active participant of the sightseeing experience instead of a passive spectator.

All the historical and user-specific information are stored in a remote database exposed to the mobile application via a REST Web-Service. The app requests the additional interest areas based on a location request and updates the user's progress and monuments statistics depending on the actions that take place. We can then display rankings and leaderboards of the users so we create a more challenging and competitive experience that aims to further motivate the users to explore the City and subsequently promote its cultural heritage.

1.2 Historical Context

Since the Neolithic era, the city of Chania has faced many conquerors and the influences of many civilizations through time. Byzantine, Arabic, Venetian and Ottoman characteristics are especially evident on the cultural center of the town clustered towards the old Venetian Harbor. Adverse climate conditions, modern city planning and rapid expansion have slowly compromised the state of these sites thus diminishing their historical value and original beauty. To promote and help preserve such sites, we designed a route throughout the city consisting of a selection of historical buildings, to be digitally reconstructed and presented in their past state through an AR paradigm. The final selection includes three monuments that represent key historical periods of the Town of Chania (Figure 1).

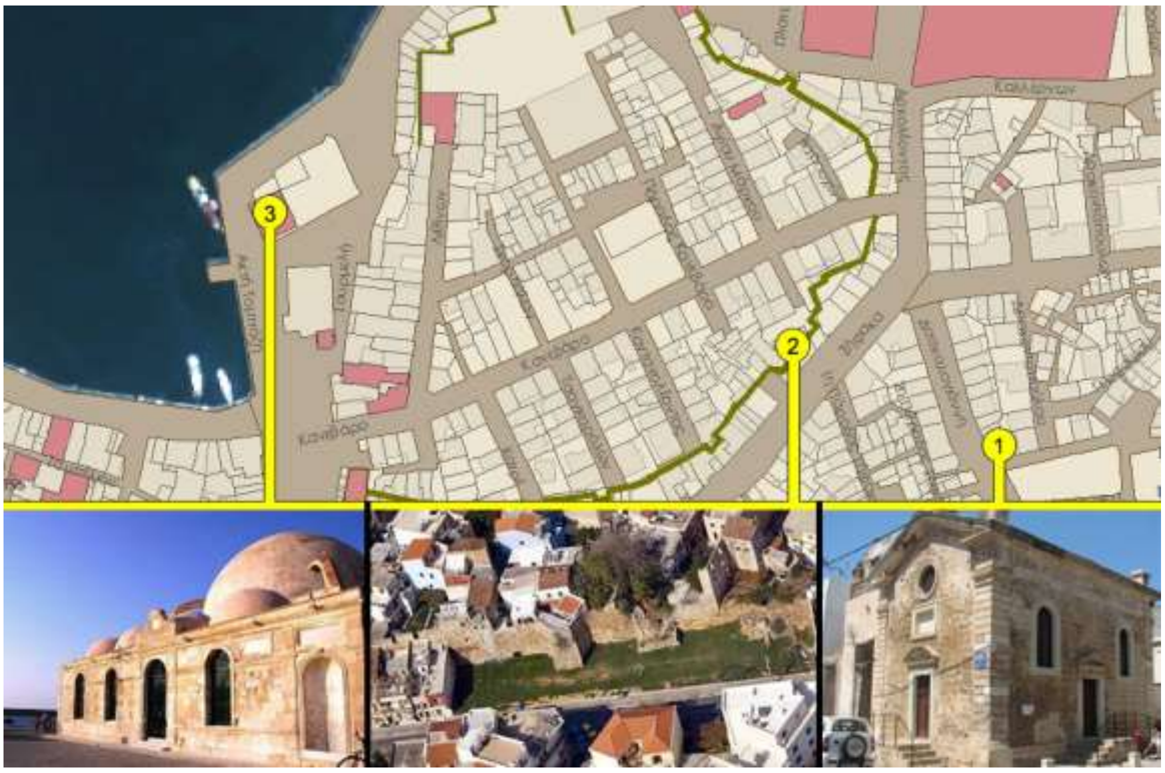


Figure 1. Overview of the selected monuments in the Old Town, Chania, Crete.

The Glass Mosque (Figure 2) is located in the Venetian Harbor of Chania and it is the first mosque built in Crete and the only surviving in the City dating from the second half of the 17th Century. Erected in honor of the first garrison commander of Chania Küçük Hasan, it is a jewel of Islamic art in the Renaissance. The mosque is a cubic building covered by a large hemispherical cupola supported by four ornate stone arches. In the western and northern part is surrounded by a covered arcade of six small cupolas, which was open to the top, as is customary in mosques, but around 1880 the arcade was covered with arched openings and neoclassic style. The mosque was used as Muslim Temple until 1923, when the last Muslims left the island in the population exchange. The small but picturesque minaret was demolished in 1920 or in 1939. During the Second World War it was quite badly damaged by bombing. After suffering many damages because of the war, it was finally restored and moved in the Archaeological Museum of Chania. Later it was used as a warehouse, folk art museum, Information

Office of Hellenic Organization of Tourism and recently became home to events and exhibitions.

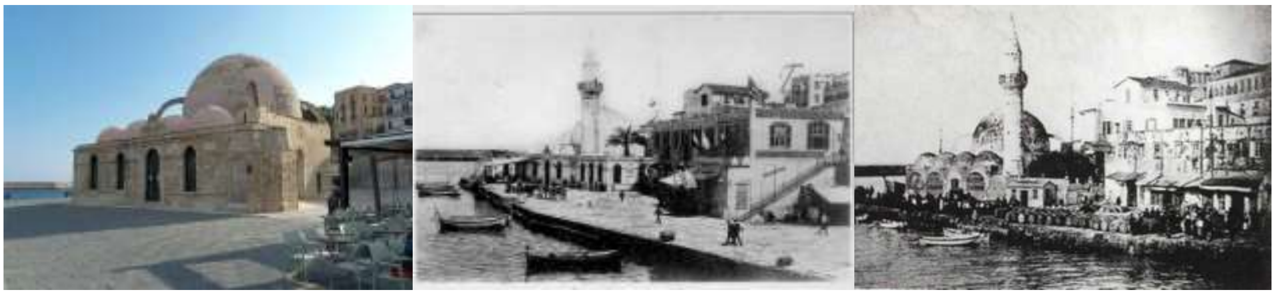


Figure 2. The Glass Mosque at its current state (LEFT). The Mosque with the now demolished Minaret (MIDDLE, RIGHT)

The Saint Rocco temple is a Venetian chapel on the northwest corner of Splantzia square that consists of two different forms of vaulted roof aisles. Although the southernmost part is preserved in good condition, the northern and oldest one has had its exterior painted over covering its stony façade, while a residential structure is built on top. (Figure 3).

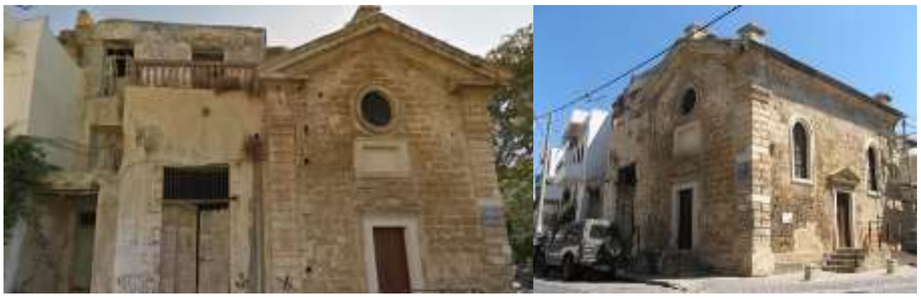


Figure 3. Front side of the temple depicting the northern and southern parts (LEFT). Southern side (RIGHT)

The Byzantine wall was built over the old fortifications of the Chydonia settlement around 6th and 7th century AD. Its outline is irregular with longitudinal axle from the East to the West, where its two central gates were located (Figure 4). The Wall consists of rectilinear parts, interrupted by small oblong or polygonal towers many of which are now partly or completely demolished (Figure 4).

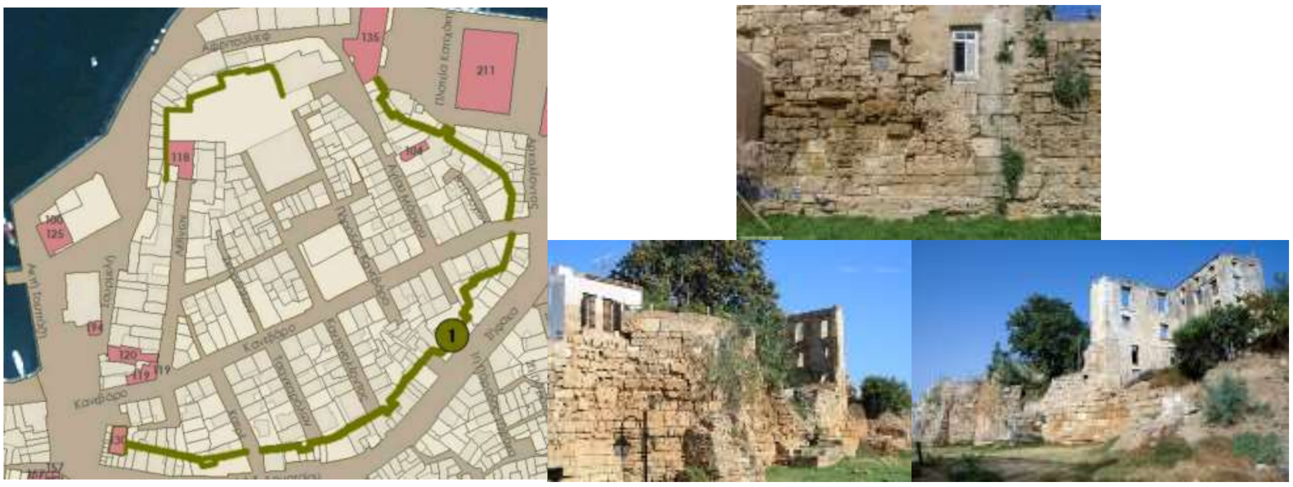


Figure 4. The whole Byzantine Wall and the part chosen to be reconstructed in Green (LEFT). The demolished and build over towers of the Wall (RIGHT)

Typical reconstruction presentation formats range from posters and maps to actual physical or digital models. The scope of this work is to virtually restore partially or fully damaged buildings and structures on historic sites and enable visitors to see them integrated with their real environment. Our aim is to design and implement a mobile AR application for Android devices that will help visitors interact with the city's monuments. We aim to deliver geo-located information to the users, in various forms, and help them document their visits. By integrating digital maps and a location based experience we aim to urge the users to further investigate interest areas in the City and uncover their underlining history.

1.3 Structure of the Thesis

In this Chapter we introduced the technologies related to our work as well as the motivation for our application. We provided a brief description of our app’s functionality and outlined the historical aspects that further motivated us.

In Chapter 2 we provide an introduction to Augmented Reality, its definition and the enabling technology. Following, we present an overview of past well known Augmented Reality Systems that relate to the cultural context of our work. Next we outline the technical challenges of Augmented Reality and describe the most common Vision Based and Sensor Based approaches that aim to tackle them, followed by a review of the available software library tools and frameworks for developing mobile AR applications. Finally we describe the testing and evaluation process of the above libraries and present our selection.

Chapter 3 presents the requirements gathering and analyzing process. We outline the requirements we set to satisfy in the beginning and present the use-cases that illustrate all the possible interactions between the users and our app.

In Chapter 4 we present the Prototyping and Storyboarding process. We present the initial design of the application’s User Interface, how we aim to incorporate the required functionality, as well as how the screens connect to each other to form a cohesive user experience. Next we present the final User Interface, explain the deviations from the initial design and finally we describe the process of creating our 3D assets to be integrated with the application.

In Chapter 5 we analyze the system architecture in detail as well as the implementation technologies. We present the implementation of the Client- Server architecture followed for this system and describe in detail their communication and structure.

Chapter 6 describes in detail the implementation of the mobile application and the REST Web-Service. We first describe the Web API that serves all the historical and user specific information to the Clients and then proceed to the mobile application where we present how the communication of the two is accomplished, how we handle, present and alter the acquired information and how we used the provided tools to create our application. Finally we provide an evaluation of our final application.

Chapter 7 is a summary of the whole experience and a suggestion for future improvements and opportunities.

2 Augmented Reality

2.1 Introduction

Augmented Reality (AR) is the act of superimposing digital artifacts on real environments. In the reality-virtuality continuum (Milgram 1994) (Figure 5), AR is a part of the broader Mixed Reality spectrum. In contrast to Virtual Reality where the user is immersed in a completely synthetic environment, AR aims to supplement reality. While early research limited the definition of AR in a way that required the use of head-mounted-displays (HMDs), a taxonomy introduced in (Azuma 1997) tried to differentiate it from the required technologies and defined that any system that; (1) combines virtual and real, (2) registers (aligns) real and virtual objects with each other and (3) runs interactively in three dimensions and in real time, is considered an AR System.

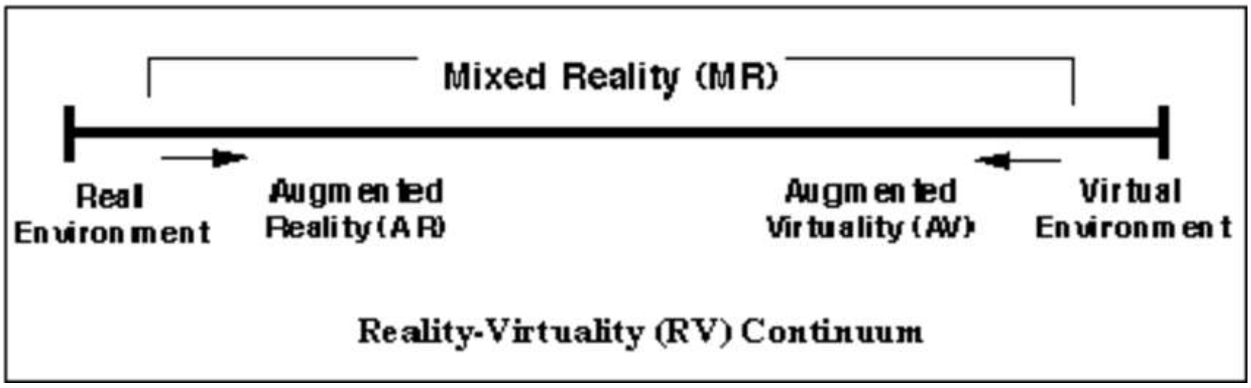


Figure 5. Milgram's Reality-Virtuality Continuum

With this definition in mind, the basic requirements of AR were laid out as: scene generation, display devices and position and pose tracking. Scene Generation refers to the rendering capabilities of the system. AR intends to blend digital information in the real world so it only needs to supplement reality. Therefore, fewer objects need to be drawn and they do not have to be realistically rendered in order to serve the purposes of an application. The display device is the way to deliver the real world combined with the digital information to the user. These devices range from the older see-through HMDs to modern HUDs, eyeglasses, contact lenses and the handheld displays of smartphone's portraying a camera view. In order for an AR system to accurately align the digital information to the real it needs an accurate computation of the user's position and viewpoint relative to the environment. While In the previous two cases AR has low requirements and high tolerance, this is not the case with pose and position tracking. Tracking approaches vary depending on intended use, indoor or outdoor environments and are the main components for accurate registration.

In comparison to older systems that used a combination of cumbersome hardware and software modules, recent advent in mobile technology has led to an integrated platform, ideal for the development of Augmented Reality experiences, often referred to as Mobile AR (MAR). MAR is a concept first conceived in the late 1990s with many systems being integrated to cultural and archaeological sites, but the use of today's modern smartphones has brought AR to an even wider audience. The presence of high processing power, cameras, inertial and GPS sensors in modern devices is capable of providing all the necessary components of an AR system in an ergonomic hand-held device.

In this thesis we focus on Mobile Augmented Reality (MAR) for Android smartphones. Although there is much research in AR and how to tackle its problems the rest of this work will be limited to technologies available to smartphones and tablets and the restrictions they apply.

2.2 History of Augmented Reality Systems

In the past years AR has been utilized for a number of applications in cultural heritage. One of the first Mobile Augmented Reality (MAR) Systems was built in 2002 for the site of Ancient Olympia (Vlahakis 2001). The system provided on-site help and Augmented Reality reconstructions of ancient ruins. The system made use of a compass, a DGPS receiver and together with the comparison of live view images from a webcam it obtained the user's location and orientation. Visitors had to carry a backpack computer which performed the calculations and wear a See-through Head Mounted Display (HMD) to display the digital Content. The mentioned components were hooked on the backpack computer making it a cumbersome MAR unit not acceptable by today's standards. In addition, the optical tracking approach requires a large number of images to be compared in real time which leads to fixed viewpoints, thus disallowing movement while viewing the reconstructions, and adds additional system delays as the communication with a central database that holds the original images is required. Despite the ergonomic restrictions, the system was very well received by the visitors as it provided a unique site-seeing experience.

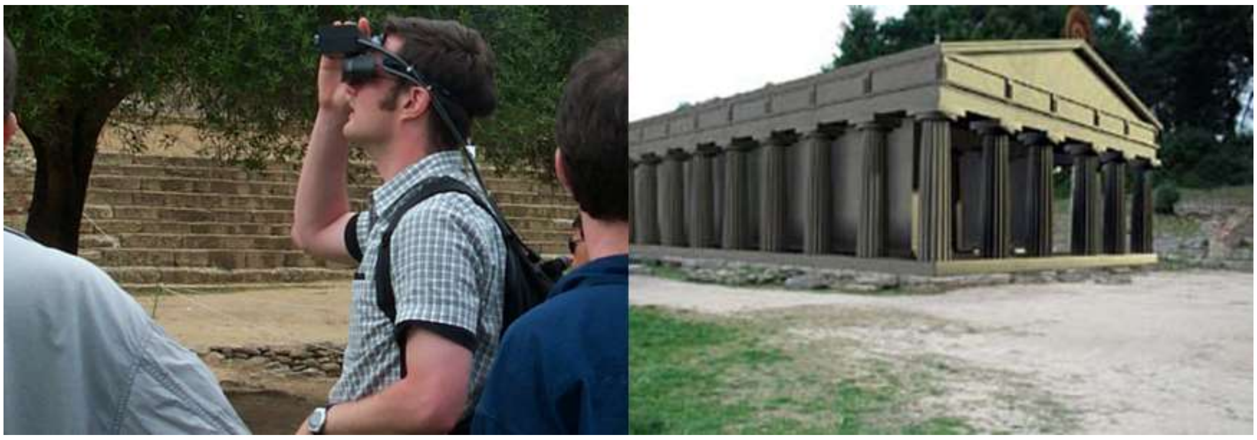


Figure 6. Mobile Unit in action (LEFT). Digital Reconstruction of temple as seen through the HMD (RIGHT)

MARCH (Choudary et. al. 2009) was a mobile Augmented Reality application developed for digitally enhancing the visits of prehistoric caves. It was developed in Symbian C++, running on a Nokia N95. It was the first attempt of a real time MAR application without the use of grey-scale markers. Instead, it was using coloured patches added to the corners of images containing prehistoric cave engravings. The system made use of the phone’s camera to detect these images and overlay them with complete drawings made from experts. The augmentations would either be available in museums or by acquiring the prepared images, uprooting the experience from its original context and presenting it in a context-less object.

With the advent of mobile devices more sophisticated AR experiences are made possible like the one for the Bergen-Belsen memorial site, (Pacheco et. al. 2014), a former WWII concentration camp in northern Germany which was burned down after its liberation. The application integrated database interaction, reconstruction modelling, and content presentation in a hand held device. The system was developed for an I-Phone. Real time tracking was performed with the device’s GPS and orientation sensors and navigation was conducted either via map or the camera. The system was superimposing the reconstructed building models on the phones camera effectively comparing past and present states. AR paradigms and location awareness in historical settings could be further employed to promote interactions with the sites and elevate the user from a passive spectator to an active participant and improve the communication of information.

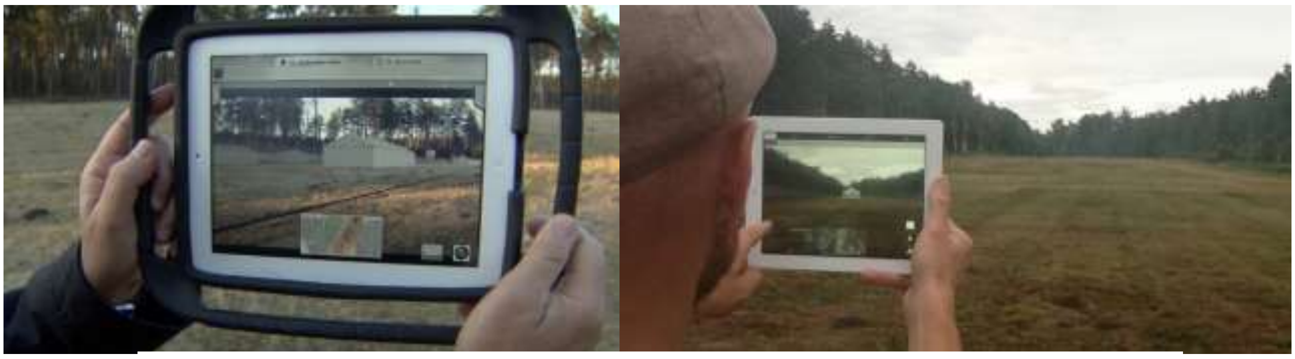


Figure 7. Augmented Reality View of the memorial site with an IPad

Focusing more on the promotion of cultural heritage in outdoor settings, VisAge (Julier et. al. 2016) was an application aiming to turn users into authors of stories and cultural histories in urban environments. The system featured an online portal where users could create their stories using routes through physical space. A story is a set of spatially distributed Points of Interest (POIs). Each POI has its own digital content consisting of images, text or audio. A viewing tool was developed for mobile tablets in Unity using Vuforia’s tracking library to overlay the digital content in the real space. The users could then follow these routes in the city and view the digital stories. Tracking was performed using natural feature detection algorithms from the camera’s feed. As per any optical approach the experiences are susceptible to environmental changes and content delivery is not guaranteed due to the lighting variations of the outdoor setting.

Further work in 3D reconstructions was shown in CityViewAR (Leeet al. 2012), a mobile outdoor AR application that was developed to allow people to explore destroyed buildings after the major earthquakes in Christchurch, New Zealand. Besides providing stories and pictures of the buildings, the main feature of the application is the ability to visualize three-dimensional models of the buildings in AR, based on a geo-location and sensor approach. The available content was displayed on a map and the reconstructions were available when at the designated location. While 2D maps offer all the required information AR navigation has shown that it can further assist users' navigation by promoting the 2D projection of the content in 3D and in the real world.

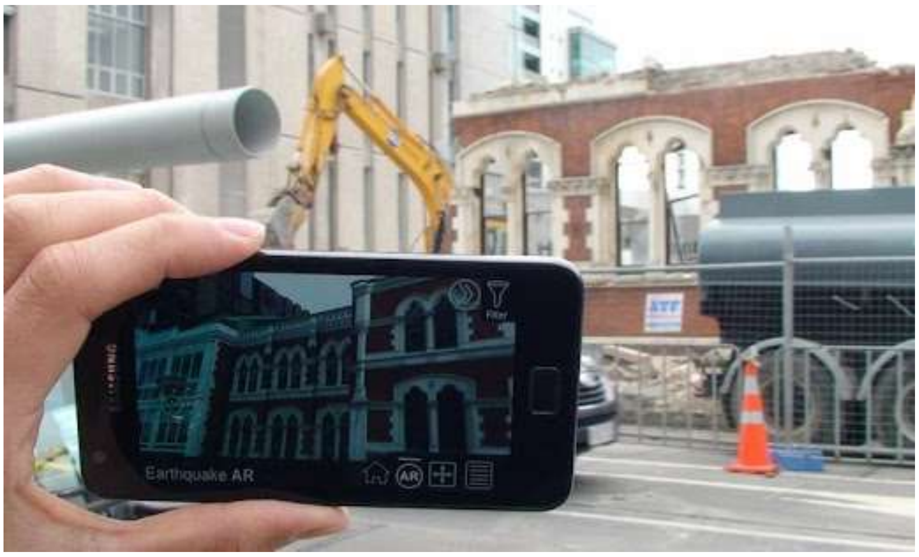


Figure 8. Reconstruction of a demolished building in Christchurch New Zealand.

2.3 The Registration Problem

Registration in an AR system is the degree in which the virtual information is accurately presented with the real environment. The objects in the real and virtual worlds need to be properly aligned with respect to each other, or the illusion that the two co-exist will be compromised (Azuma 1997). In contrast to Virtual Reality (VR) where such errors result in visual-kinesthetic conflicts, in AR such conflicts are visual-visual and easier to detect. Take for example a user wearing a VR headset that raises his/her arm to see a virtual one. If the virtual arm is off by a few centimeters, it may not be detected because the conflict is between the "sensed" position of the real arm, and the "seen" position of the virtual one. In the corresponding AR application the virtual arm should completely overlap the real one, so such an error would be easily detectable.

Registration errors are divided to dynamic and static. **Static** errors are the ones that affect the AR scene even when both user and environment are in stasis. Sources of such errors can be bad calibration of mechanical parts, incorrect tracker-to-eye ratio, field-of-view parameters, optical lens distortions, etc. Static errors depend mostly on mechanical parts and the correct initial calibration of the system and can be accounted for to a very satisfying degree. **Dynamic** errors on the other hand are the ones that take effect if either the viewpoint (user) or the annotated object, begin moving. For MAR this kind of errors are by far the largest contributors to the Registration problem and vary depending on the implementation.

In early AR systems the single most important factor for dynamic errors was end-to-end system delays. A tracker reported user movements to the system and the system should then update the digital artifacts on the screen. This computation and its delivery should precede changes in the user's pose which proved at the time to be a very difficult task. With today's hardware, system delays have been minimized and the main source for errors in registration is Pose estimation (Position and Orientation tracking). Tracking in an AR scene has proven a complicated task with no single best solution.

In order to register virtual content in the real world, the pose (position and orientation) of the viewer with respect to some "anchor" in the real world must be determined. Depending on the application and technologies used, the real world anchor may be a physical object such as a magnetic tracker source or paper image marker, or may be a defined location in space, determined using GPS or dead-reckoning from inertial tracking.

In order for an AR system to overlay the world with digital information it needs to track its position with 6 DoF (Degrees of Freedom). That means three variables for **position** and three for **orientation**. There are many factors that have enabled modern smartphone's to track their position. Inertial sensors, GPS positioning and optical sensors can provide all the necessary data for such computations. Although there are many approaches to pose estimation for AR systems, we will focus on implementations for consumer grade smartphones. Each tracking approach has its advantages depending on the use case. The main approaches are: **vision based** tracking, which relies on the device's camera and ways to process the live feed and **sensor based** that combines GPS positioning and the inertial sensors of the device.

2.4 Vision Based Implementations

Vision based tracking approaches determine camera pose using data captured from optical sensors. In the wider AR area these optical sensors can be: infrared sensors, visible light sensors and 3D structure sensors. Consumer grade devices are currently limited to visible light sensors. These cameras are particularly useful as they can both act as the real world background shown to the user as well as the screen for registering the digital content in the real world.

Vision based tracking with visible light has become increasingly popular in recent times due the minimal hardware requirements and increased computational power of modern devices. The main techniques that utilize the phone's camera can be divided into three categories: (1) **Fiducial tracking**, (2) **Natural Feature tracking** and (3) **Model based tracking**. In the next subsections we will discuss each approach as well as their advantages and limitations.

2.4.1 Fiducial Tracking

Fiducials are defined as artificial landmarks that are added to the environment to aid in registration and tracking. In the earlier days of AR these fiducials could be colored LEDs or pieces of paper. By placing them in known positions pose could be determined if enough fiducials were identified from the camera in the scene. In order to calculate the pose the detection of a minimum of 4 fiducials is required. This lead to their final form that uses a single planar fiducial (often a piece of paper) featuring a quadrilateral shape whose corners act as the 4 known points and additional information can be encoded inside the shape to allow for unique fiducials to be used in the same application.

These are known as Augmented Reality Markers (Figure 9) and served for many years as the main technique for AR applications due to their simplicity to use and high accuracy in registration. However Marker based augmented reality still requires the placement of these fiducials in the real environment which in many cases is not desirable or ever possible. Additionally as per any vision based implementation anything that compromises the visibility between the camera and the marker can make the virtual scene collapse, which further limits their use to strictly controlled environments.

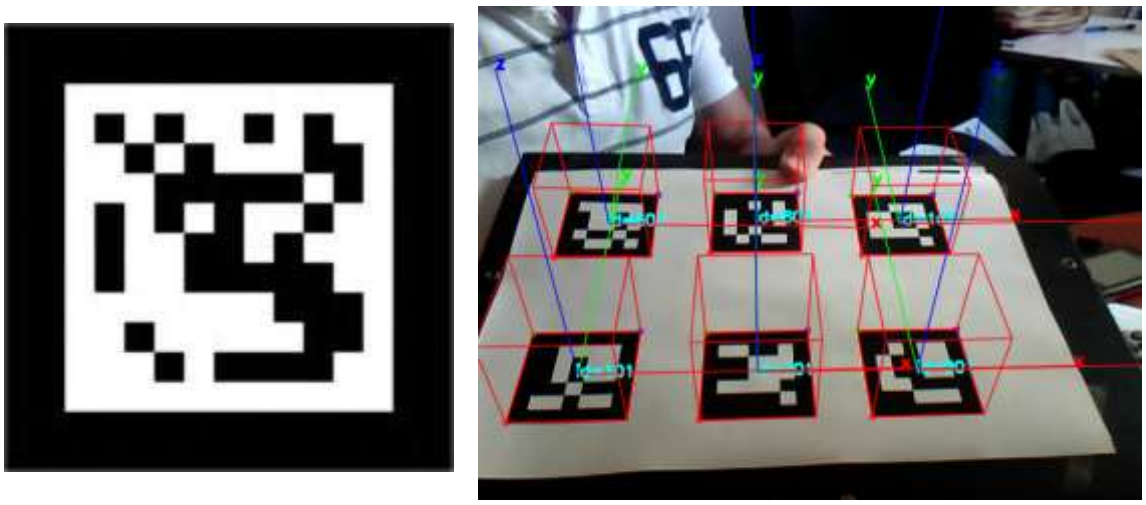


Figure 9. Example of an AR marker (LEFT). Tracking multiple markers (RIGHT)

2.4.2 Natural Feature Tracking

In contrast to marker based tracking Natural Feature tracking uses features already present in the environment. Complicated image processing algorithms are used to detect features such as corners, points or intersecting lines in captured images of the objects to be tracked, and a unique descriptor is calculated to allow for the identification of each feature. By matching features in the scene with the known ones, pose calculation is possible using similar algorithms as in marker tracking.

The most common natural feature detection algorithms are SIFT, SURF, BRIEF, ORB, BRISK and FREAK. While SIFT and SURF proved too complicated for real time tracking in mobile devices, they acted as the base for BRIEF, a much less complex and viable solution. The success of BRIEF and its integration with AR systems lead to more feature detectors which aimed to improve performance. The most notable are BRISK and FREAK with the later proven to outperform its predecessors.

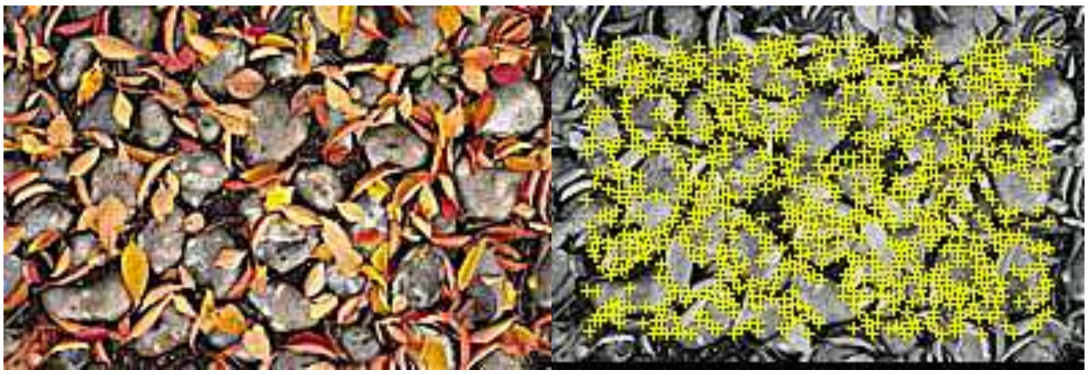


Figure 10. Sample of an Image before (left) and after feature extraction (right)

Natural feature tracking is currently the most commonly used tracking approach as it removes the intrusiveness of the AR markers and it allows for robust tracking and registration. However this approach relies heavily on the features present in the real world scene and the ability to identify them. This means that a little number of features as well as occlusions and diverse lighting conditions in a scene, can greatly diminish the system's tracking capability.

2.4.3 Model based tracking

Although not as popular as the Fiducial and Natural feature tracking these tracking techniques use known 3D structures, like a CAD model, to track real world objects. Edge filters are used to extract structure information about the scene which is then matched to primitive structure types, like lines, cubes, cylinders and circles, to provide pose estimations. Combining these tracking techniques with natural feature detection allowed for the inclusion of textures in the models which provided greater robustness in complex and variable environments.

Extracting structure information about the real world has lead AR to adopt the well-known in robotics **SLAM** concept (Simultaneous Localization and Mapping) which allows for simultaneously create and update a map of the real environment while localizing the system's position within it. These approaches elevated the AR system's capabilities from tracking planar surfaces to more complex geometries and 3D structure. The motivation for SLAM together with further optimizations for AR has led to a process named **PTAM** (Parallel Tracking and Mapping) where the tracking of the camera and mapping of the environment components where separated which improved the overall performance.

Line of sight requirements and the inability to close large loops currently limits these implementations to small environments and tracking of small objects. However these techniques and their adaptation with modern smartphones is still in its infancy and more implementations are being developed involving additional sensors and trackers that will enable smartphones to reliably track 3D space.

2.5 Sensor Based Implementations

Inertial tracking uses long range sensors like accelerometers, magnetometers and gyroscopes to calculate orientation, and combined with positions acquired from the GPS the system can calculate its pose relative to the Earth’s frame. Inertial sensors allows for orientation tracking with 3-degrees of freedom by using a 6-axis accelerometer for orientation relative to the center of the Earth, and a magnetometer for measurements relative to the North. Gyroscopes can then be employed to detect changes between relative movements. By combining this information Easting can be calculated and 3 DOF orientation estimate is available (figure 11).

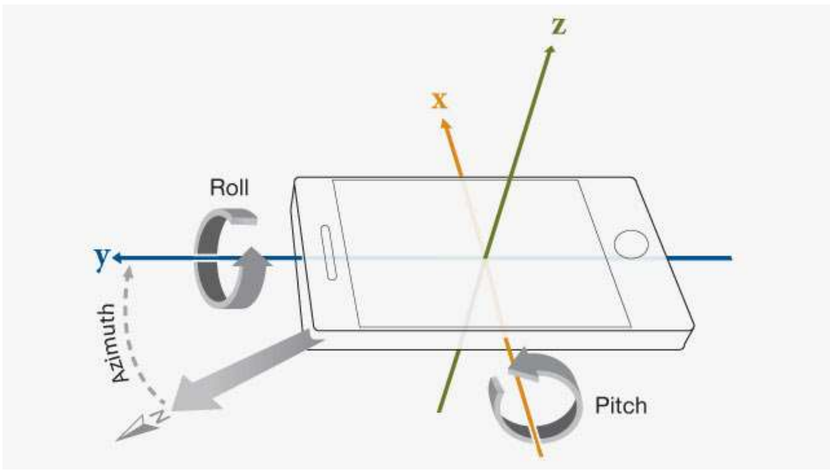


Figure 11. Mobile coordinate system and orientation relative to the Earth's frame of reference.

Although position tracking is available solely with the use of the same sensors, they are very susceptible to drift over time, especially for such estimates which can only be derived from velocity. Due to this issue positioning with inertial sensors should only be conducted with additional trackers capable of providing measurements for drift correction. For this reason sensor based systems employ other methods for position tracking. The GPS sensor provides position tracking in outdoor environments with an average accuracy of 3 meters. Locations acquired by the GPS include latitude, longitude and altitude (where available) information in the world coordinate system. Combined with the orientation estimate from the inertial sensors, 6 DOF tracking is possible relative to the Earth’s Frame.

Sensor based implementations have no range restrictions or line-of-sight requirements and carry no risk of interference from external sources. In contrast to vision based approaches, these implementations may not know where the real objects are placed in the environment and rely on a “sensed” view of the scene, with no feedback on how close the digital and the real align and thus provide less accurate registration. The combination with the GPS also limits the use to outdoor scenes where GPS reception is available.

2.6 Hybrid implementations

Hybrid tracking systems fuse data from multiple sensors to add additional degrees of freedom, enhance the accuracy of the individual sensors, or overcome weaknesses of certain tracking techniques. As mentioned above, sensor based implementations are susceptible to drift and latency due to filtering, while optical implementations have line of sight requirements and short range. This has led to many applications utilizing both vision based tracking and inertial sensors. This allowed the systems to take advantage of the low jitter and drift of the optical approaches while extending the range of the AR system through the inertial sensors that have no line of sight requirements and high update rates that ensure responsive graphical updates.

Most commonly model based tracking is combined with inertial sensor tracking. Motion estimates of the system are calculated from the inertial sensor, fusing data from accelerometers and gyroscopes, while the optical approaches provide measurements for drift correction and map the real world scene. Combining such techniques with additional depth information has opened up new possibilities for AR. The most well-known approach for such techniques is the KinectFusion system by Microsoft which uses data obtained from the Kinect's structured light depth sensor to create high quality three dimensional models of real objects and environments, and these models are also used for tracking the pose of the Kinect in the environment.

Such implementations have shown to provide pixel perfect registration and have extended the capabilities of AR systems to more complex use cases often including both indoor and outdoor environments. However the complexity of these implementations often leads to the inclusion of additional sensors and trackers like depth sensors, IR cameras, LEDs, laser pointers etc. which are not available to consumer grade mobile phones. Currently implementations that only use a camera and the inertial sensors available to the majority of mobile devices, are limited to small unknown environments and the detection of horizontal surfaces like the ground, tables etc. and do not allow for matching of predefined models, thus the digital content is manually placed by the users. Also developing with such techniques is supported by a very small number of AR SDKs and requires specialized equipment.

2.7 Augmented Reality Software Development Tools

In this section we review software library tools for developing AR applications. There are a number of available tools and each may target different application platforms like Desktop AR, Mobile AR or both. In this work we provide an overview of tools targeting Mobile AR for Android smartphones and tablets. The available tools range from low-level libraries that provide access to core functionalities like tracking and rendering, to higher-level implementations that allow the developer to focus on the AR content.

2.7.1 ARToolkit

ARToolKit is one of the most widely adopted software development tools for Augmented Reality. It is an open source and low-level tool that not only provides marker and markerless tracking of its own, it also allows developers to deploy and test custom tracking implementations. It is developed in C language and supports Windows, Linux and Mac OS X desktop operating systems. It provides full Unity3D and OpenSceneGraph support for advanced rendering capabilities. It uses computer vision techniques and supports classical square markers, 2D barcode, multimarkers, and natural feature tracking. Furthermore, ARToolKit supports any combination of the above together.

2.7.2 Vuforia

The Vuforia library developed by Qualcomm is also one of the most popular low level libraries. It is widely known for its Computer Vision capabilities as it supports the natural feature tracking of planar images, detection of cylindrical surfaces, small 3D objects, text and small boxes with flat surfaces. It has ample documentation and provides great support through its online forum. Development with the Vuforia library can either be done on the Native level with the Android NDK in C or in Unity3D with the Vuforia plugin. In this case the programmer can use the normal Unity3D visual programming and scripting interface to create rich interactive experiences.

2.7.3 Wikitude SDK

The Wikitude SDK is one of the most popular high-level AR SDKs that combines Geo-Location and Computer Vision capabilities. It provides implementations for development in Java, JavaScript and a Unity plugin similar to the Vuforia library. Due to it being a high level tool feedback from the tracking implementations is limited. The SDK's features include natural feature tracking for planar images as well as cloud recognition for big datasets. It also supports sensor and geo-location tracking and most recently the Instant tracking feature which combines sensors and image processing techniques for environmental tracking and placing of AR content. It has great documentation and it provides great support through its forum.

2.7.4 Mobile AR framework HitlabNZ

It is a framework that allows the development of Android AR apps in java. It is based on a Geo-Location approach either with relative locations (mock locations) or real Locations from the GPS. It provides its own OpenGL implementation which allows for simple touch interactions and has its own digital map implementation to display the geo-located content. The framework has ample documentation but it was developed for older versions of the Android API.

2.7.5 DroidAR

Droid AR also provides vision-based marker tracking, which is based on the OpenCV computer vision library and uses square markers similar to that of ARToolkit. It also allows for Geo-Location based AR as well as footstep recognition for indoor positioning. Unfortunately the SDK does not provide enough documentation and it is also based on older versions of Android.

2.7.6 ARCore

ARCore is a platform for building augmented reality apps on Android developed by Goggle. ARCore relies on hybrid tracking using motion detection, light estimation and environmental understanding to provide AR experiences in unknown environments. It has great documentation and currently supports devices running Android N or later. It supports a wide variety of platforms including Android, Unity3D, the Unreal Engine and the Web using the three.ar.js JavaScript library and prototype browsers for Android and iOS.

2.7.7 Tango

Tango is a tool developed by Google that targets specific AR enabled devices equipped with the KinectFusion sensors. It provides SLAM capabilities with Motion tracking, Area learning, Depth perception and Visual Positioning and development can be done either with the C/C++ API, Java API or in Unity3D. Currently it is only available for Android and specifically for devices that include the Kinect sensors.

2.8 Choosing an AR solution

The most important aspect of our AR experience is the representation of the chosen monuments in their former state. That includes replacing existing parts of those buildings, adding parts that were demolished or their complete overlap with a 3D reconstructed model, where necessary. Since our ultimate goal is to provide a complete experience to the end user, we relied on tested and working solutions for the AR experiences. Our case targets large scale buildings in wide areas and highly visited and unsupervised touristic sites. Taking this into consideration we concluded on the main restrictions our application needs to fulfill:

- We cannot interfere with the structures in any way
- The tracking technique used should ensure the delivery of the AR experience without assistance

- The AR experiences should be available to consumer-grade mobile phones with no additional external hardware components
- The AR experiences should be available from many or all, if possible, viewpoints

While Geo-Location approaches are predominantly used in outdoor environments, the accuracy of the GPS led us to also take into consideration optical approaches based on Natural Feature detection algorithms and hybrid implementations.

2.8.1 Image Recognition

AR frameworks support both sensor and optical implementations. Image recognition that relies on natural feature detection algorithms was tested using a variety of images captured from the annotated monuments on-site. For the purposes of testing these optical implementations we developed a sample application using the high-level Vuforia Unity3D plugin. The Vuforia SDK was chosen because it is highly regarded amongst computer-vision based tools, and due to the Unity plugin testing the ready to use implementations on the site requires less time.

The first step to implement the sample app was to acquire the images from the sites and upload them to the online Target Manager Tool, provided by Vuforia, to be processed through the natural feature detection algorithm and provide us with the final database of features. This database holds the collection of images we provided and their sets of features which are called "targets". The target manager is a web-based tool that enables the developers to create, store and manage these target collections as well as export them to the developing environment of their choice. After creating the target collections we export them as a unity assets package. An example set of the image targets can be seen in figure 12. These images do not represent the whole set of features but a portion of them provided by the tool to assist the developer in evaluating the targets. Each image is a target to be detected by the device in the camera's feed and tracking can be done with multiple targets simultaneously up to a maximum of 5.

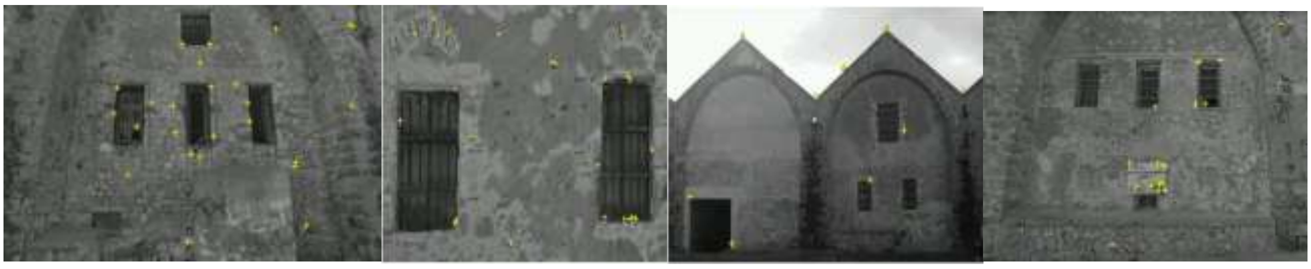


Figure 12. Facades of the buildings processed through the Vuforia target manager

The Unity game engine provides a fast and easy way to create 3D environments. Unity is notable for its ability to target games for multiple platforms including Android, Windows, macOS, iOS and Linux. Through its smart and adjustable Graphic environment it allows for the manipulation of 3D objects and scenes with minimal coding requirements. To integrate Unity with the Vuforia SDK we need to import the Vuforia unity package to the project. This package contains the components that incorporate all the Computer Vision capabilities provided by the SDK. In Figure 13 we can see an overview of the developing environment.

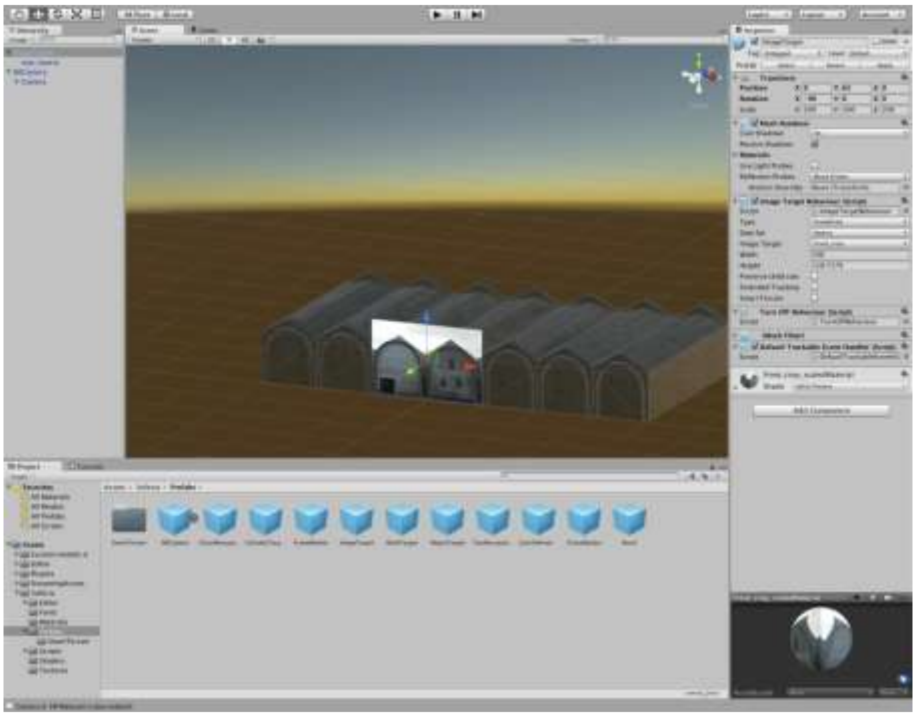


Figure 13. Unity editor with the Image target and the 3D model aligned.

The components available from the Vuforia plugin can be seen in the bottom panel. The most important component for the AR scene is the ARCamera. This is a special camera type that supports augmented reality apps for both handheld devices and digital eyewear. The image target component is where the acquired data-sets are loaded. In the 3D scene it is a simple plane depicting the target image. By making the 3D model a child of this component it specifies that it will only be rendered after detection and at the location relative to the target as specified in the scene. At this part of the process it is important to note that representation was not important, the ultimate goal is to test the tracking so the 3D model is a very basic 3D mesh and it is manually aligned with the target. Additional targets can either be added individually or with the multi-target component in the same way. After repeating the process for all the targets and buildings we can install the sample application directly from Unity to the desired device.

The main screen of the sample app is a camera surface depicting the real word. When a target of the provided set is detected in the camera's feed, the 3D model is rendered on the corresponding location on the screen. The final part is to visit each site and test the optical implementation.

While image recognition has presented great results, due to fast hardware and improved algorithms, we could not rely on it. Outdoors environments present very challenging conditions to such implementations. Building façades provided very little features and together with variations in lighting conditions, the sets of features provided to the AR system differ greatly from the real scene and thus the targets where not recognized. In the rare cases where tracking was possible, as seen in Figure 14, simple movements of the device resulted in miscalculations of the orientation. Taking also into account the cramped environment of a touristic site where many factors can compromise the visibility of the targets, image recognition did not present a realistic choice.

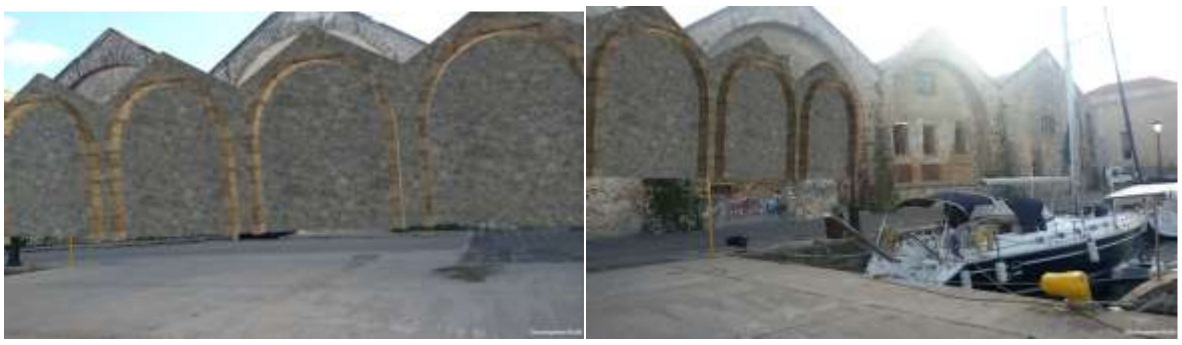


Figure 14. Image Recognition registration following a simple left-to-right motion.

2.8.2 Geo-Location and Sensors

Sensor approaches rely on the inertial sensors of the device for orientation tracking and on the GPS for positioning. Specifically for Android devices these sensors can be either be hardware components like magnetometers, accelerometers and gyroscopes or software components that rely on multiple physical components and sensor fusion to produce specific results, like the linear-acceleration sensor, orientation sensor, the gravity sensor and more. These sensors can be categorized as **motion sensors** that measure acceleration forces and rotational forces along three axes of the device, like the accelerometer and gyroscope, and **positioning sensors** that provide absolute measurements about the physical position of a device like the magnetometer and the orientation sensors. The geo-location approach requires measurements relative to earth's frame of reference. This requires the combination of motion and positioning sensors to determine the orientation and their combination with the GPS which provides latitude, longitude measurements for position.

If the virtual objects are accurately positioned and oriented in the world coordinate system, the AR system is aware of the distance between them because that model is built into it. All the digital content is associated with a geo-location which is then matched to the user's position and viewing angle. For these implementations we used the HitlabNZ mobile AR framework and the Wikitude JavaScript API. These tracking techniques do not require any preparations in the environment and their implementation relies on cheap sensors present in every modern smartphone.

The biggest problem with these implementations is the registration errors introduced by the GPS accuracy and the latency from filtering the sensor data. The AGPS present in mobile phones has an average accuracy of 3 meters. This error margin proves very impactful, especially in the case where we overlay the reconstructed parts of a monument, as the illusion completely breaks if the 3D model is registered even a meter away from the real building. The orientation calculation is based on the accelerometer and the magnetic-field sensors and a combination of these two with a gyroscope, if present. While the magnetic-filed sensor provides absolute measurements and needs no filtering, the accelerometer measures all forces that act on the device, which means that unwanted motions and mechanical noise need to be filtered out in order to isolate the force of Gravity. This Filtering process unavoidably introduces latency to the system following relative motions of the camera. For example, when a user moves the device to look higher at an overlaid building, the 3D model will be dragged along with the motion until it is significant enough to pass through the filter.

2.8.3 Hybrid Implementations

During the development process the only available tool for such implementations was the Instant tracking from the Wikitude JavaScript API. This technique uses the sensors of the device together with computer vision techniques to allow for tracking in an unknown small area, designated by the user on runtime.

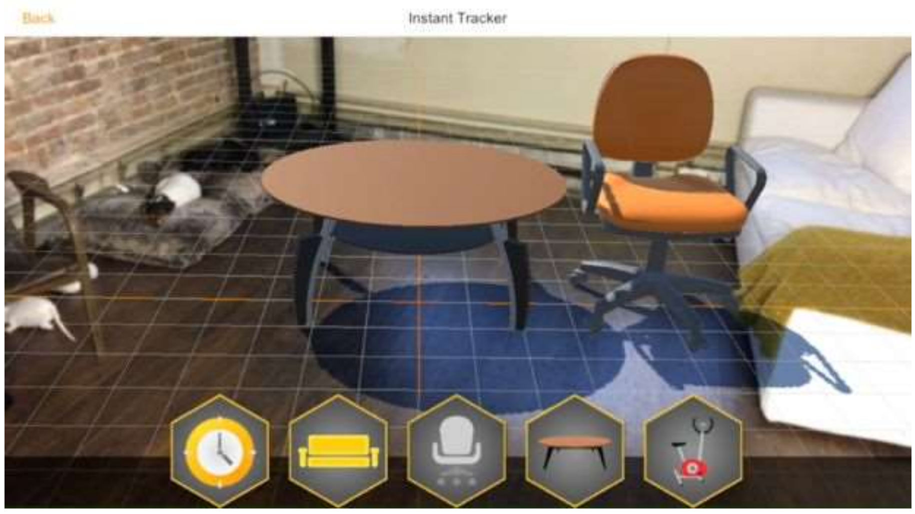


Figure 15. Wikitude Instant tracking example

The system assumes a ground surface based on the user's height and the digital content can be manually placed on it, by clicking on the screen. The user actively starts tracking with a button and the ground surface is "anchored" on features identified from the camera. This technique supports minimal movements around the scene but when the initial tracking position is off the field of view, tracking fails.

This technique enables tracking in unknown environments and not the recognition of predefined areas, so in our use case the users can place the 3D models of the buildings everywhere around them and not on top of the actual buildings. In order to limit the placement to the predefined locations we employed our own sensor implementation, based on the GPS and the orientation sensors, that aims to replicate the SDKs estimates and transfer the frame of reference from the one relative to the device to the world coordinate system. In order to do so we combined the instant tracking option with the Geo-Locations API. We used the Geo-location API to note the location of a monument on the screen with a blue circle that has a radius of the accuracy of the receiver. So if the accuracy of the GPS is at 3 meters while the user is visiting a monument, we overlay a circle with a 3 meter radius at the base of the monument. The sensor implementation is then employed to check if the user is pointing the device inside the area annotated by the circle.

When these requirements are met, the annotated circle turns green to signify the event and the tracking option is enabled to the user. To correctly draw the model on the screen we need to transfer its axes to the world coordinate system as well. Since our model is originally geo-positioned, we apply an initial rotation equal to the bearing between the user and the building.

This approach relies on the cooperation of the user. By using the GPS we have effectively transferred its error to this implementation, so to compensate for that the user has to place the center of the assumed tracking plane at the “perceived” center of the building or the model will be registered at wrong locations. Nevertheless since the receiver’s accuracy designates the available area the worst case scenario is the same as with the Geo-Location approach and the user can take action to improve the registration despite the GPS error. Moreover this approach takes advantage of the optical implementation’s tracking following relative camera movements, and the 3D model appears “anchored” on the actual building.

2.8.4 Conclusion and Our Choice

As stated in the introduction there is no single best solution to tracking and registration. Each case presents its own limitations and different approaches are better suited for each one. Vision based approaches are best suited for controlled and small environments but their performance diminishes in wide and outdoors areas where the sensor based approaches provide the best results. Tracking in AR systems is an open problem that is still being researched. Although the future seems to lie within hybrid implementations, they are currently in their infancy and most often require additional hardware components.

The most important criterion when selecting an AR solution is reliability. Graphics have little meaning when tracking is not possible. Therefore we concluded on a Geo-Location approach and the instant tracking option of the Wikitude SDK. While a case can be made for the low registration of the Geo-Location approaches, due to sensor filtering and low GPS accuracy, these implementations require fewer actions by the users and ensure that the AR experience will be delivered independent of external conditions. In contrast the Instant tracking option provides greater registration and eliminates the latency, but is susceptible to occlusions and requires more actions by the users. The decision to include both implementations was taken to provide users unaccustomed to AR applications an intuitive way of visualizing the 3D models with the Geo-Location approach, while also being able to provide a more sophisticated AR experience with the Instant Tracking. The Wikitude JavaScript API was selected due to its robust results, educational licensing-option, documentation, big community and customer service.

3 Requirements Analysis

3.1 Introduction

Before we begin to think what technologies we might need to build our app, we need to have an idea of what we wish to build. What our application is going to be about, what current needs it fulfills and what new abilities it might give to the users. After having the initial, general idea for our AR application and what we aim to do by creating it within the context of this thesis, it's time to begin mapping it out. In doing so, it will help eliminate any problems and ensure that functionality that we chose to be in the application doesn’t get missed.

Our goal was to create an application capable of providing visitors of Chania city an immersive sightseeing experience with easily accessible historical information. We aim to promote the standard ways of communicating such information by providing on-site 3D reconstructions of monuments and a location-aware experience featuring the exploration of our city’s cultural wealth.

In the next sections we provide the requirements gathering process, the use case scenarios whose aim is to identify, clarify and organize the system requirements. Because of the nature of this application the requirements gathering process was our own and did not include stakeholders and potential users of the application.

3.1.1 General Requirements

The application needs to be fully functional and provide a complete experience to the end-user. As a location based experience the physical presence in the City of Chania is required, as well as access to the City’s monuments.

The application will be available to everyone with an Android smartphone that meets the requirements of AR. The device needs to be equipped with specific sensors to support the 3D visualization in the real world and the availability of these sensors needs to be checked in order to ensure the optimal flow of the experience.

The application needs to provide a means of navigation throughout the spatially distributed content. Since all the information is delivered on a location basis, an interface that helps the user locate the available information relative to his location needs to be included.

The application is targeting any visitor or inhabitant of the City of Chania, with no age or educational requirements. The goal is to inform the user about the city in the best way possible. Bellow we outline additional functional requirements about the overall system:

The system needs to provide historical information about the city.

The system needs to showcase monument statistics.

The system needs to store and update monument data.

The system needs to store, update and delete user data.

The user needs to register to partake in the experience.

The system must provide the 3D reconstructions to unregistered users.

The user needs to be able to modify his profile.

The user needs to be able to view his progress and score.

The user needs to be able to mark places.

The user needs to be able to adjust battery settings.

The system needs to showcase player rankings.

The user must be notified about nearby monuments when the application is in the background.

The user must have control to the background processes.

The system needs access to the GPS sensor.

The application needs to have access to the phone’s camera.

The application needs internet access.

There needs to be a tutorial to help the user get acquainted with the basic functionality

3.1.2 Augmented Reality Requirements

The system must run on all android devices that meet the requirements of AR.

The system needs to provide 3D representations of the monuments in an AR paradigm.

The users must be able to change through the available representations.

The chosen monuments must be physically accessible.

The user needs to be able to select tracking methods.

The user needs to be able to Navigate through the AR camera.

The representations should be accessible from all the available viewpoints.

3.1.3 Map and Navigation Requirements

The system needs to include a map to assist in navigation.

The map needs to include annotations over the interest areas.

The annotations need to be interactive.

The map needs to annotate the user's location.

The user must be able to customize the theme of the map.

The user must be able to filter the information shown on the map.

The minimal area to interact with a monument needs to be annotated on the map.

The map needs to notify the user when he can interact with a monument.

3.2 Use Case Scenarios

In this section we provide the use case diagrams depicting a user's interaction with the system and the relationships between the user and the different use cases in which he is involved.

3.2.1 The User enters the application

The figure bellow illustrates the interactions available after the user has launched the application, and wants to proceed to the core functionalities.

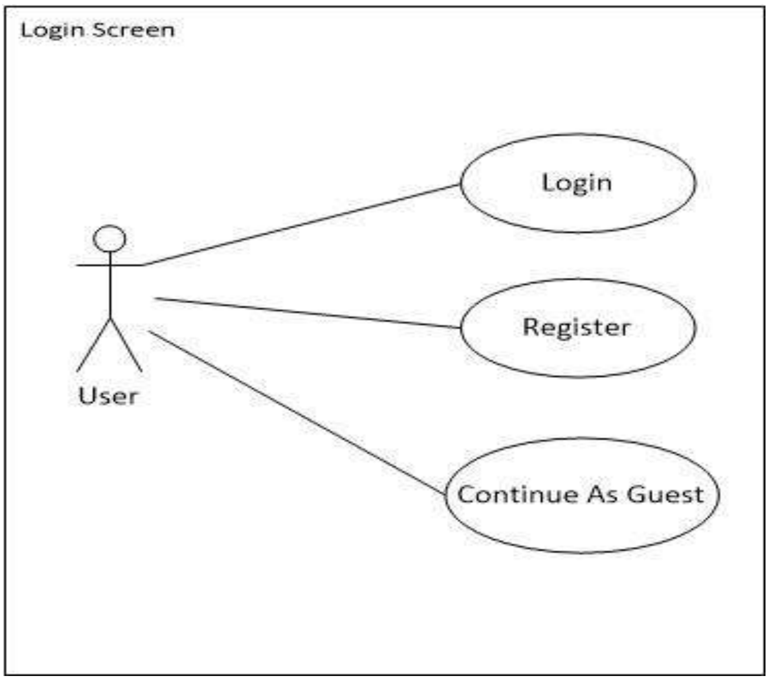


Figure 16. Login Use case

The user wants to login:
The user presses the “Login/Register” Button and enters his username/email and password to proceed with an existing account.

The user wants to create an account:
The user presses the “Login/Register” Button, changes to the register page and fills the required fields to create a new account. The fields required are a username, an email, first and last name and a password.

The user wants to proceed without an account:
The user presses the “Continue as Guest Button” to enter the application without a registered account. The user is informed by the system that the experience available to non-registered users is limited.

3.2.2 The User enters the Map

The interactions depicted below are available after the login process of the application and change depending on the option selected. The screen navigation and monument interaction cases will be described in more detail in the following subsections.

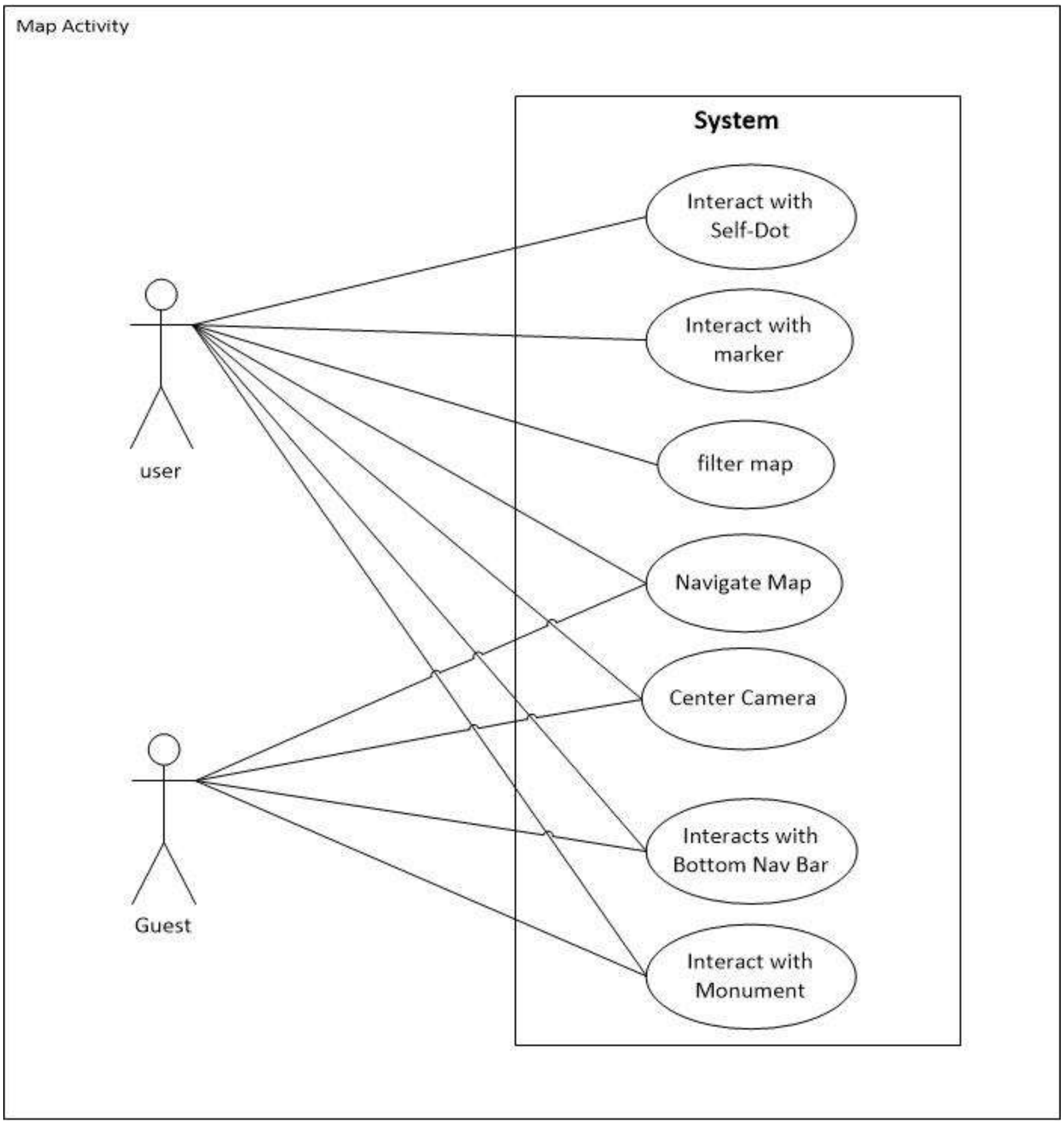


Figure 17. Map Use Case

The user wants to move the Map:

The user can use the basic touch events to navigate the map. Panning is done via one finger while rotating and tilting is done with two fingers.

The user wants to center to his location:

The user can press the “My Location” button to center the camera at his Location.

The user wants to navigate the app screens:

By interacting with the Bottom Bar the user can navigate to the rest of the screens and activities.

The user wants to interact with the self-dot:

A Registered user can get a quick overview of his stats by pressing on his location indicator on the map. The overview shows his geo-location, local, his name and score.

The user wants to interact with a marker:

When a registered user presses on a marker shown on the map an info window is displayed above it showing a thumbnail, a brief description, the monuments name and the distance between the user and the monument. Further pressing on the info window the user is transferred to the Details Page of the monument.

The user wants to filter the map:

A registered user can filter the markers shown on the map by period or by his selection.

3.2.3 The User interacts with a monument

The user's physical presence at a site is required to access the AR experiences

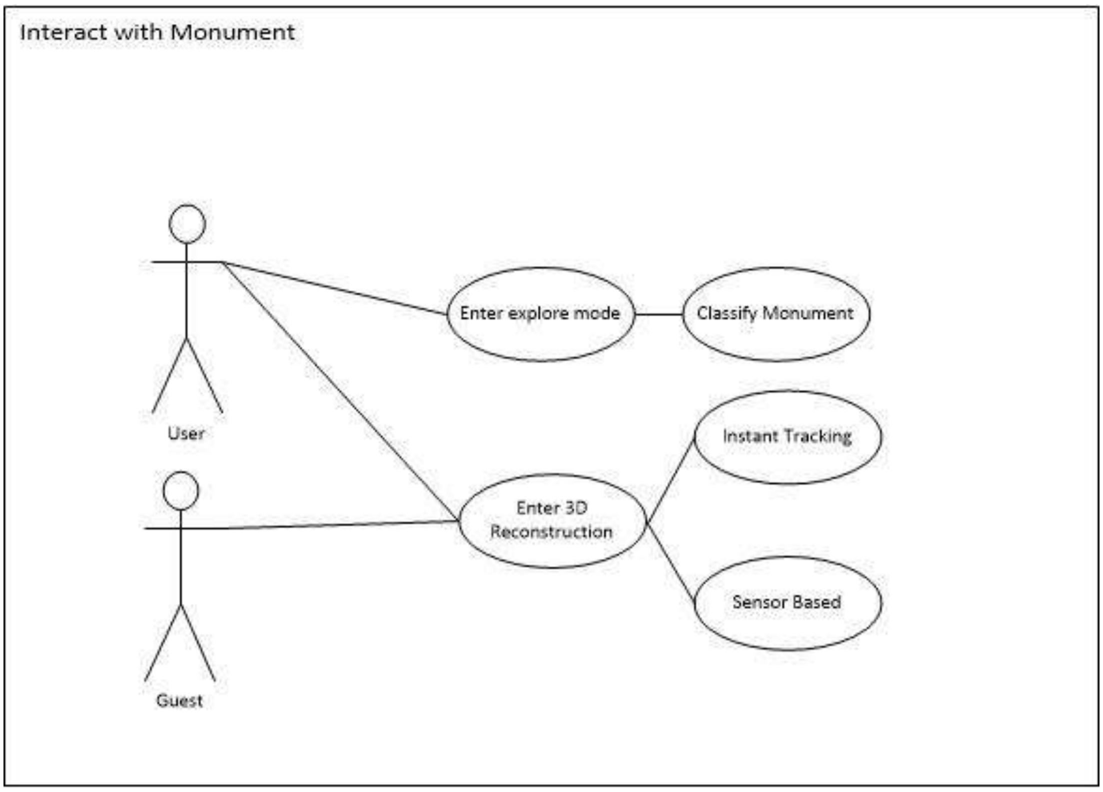


Figure 18. Interact with Monument Use Case

The user wants to classify a monument:

A registered user can enter the explore mode of a monument and proceed to classify it into one of the three given periods.

The user wants to view a 3D reconstruction:

All users can enter the 3D reconstruction of a monument if available. The users can then choose between the two tracking methods available.

3.2.4 The User navigates the app

The user wants to navigate through the available screens of the application.

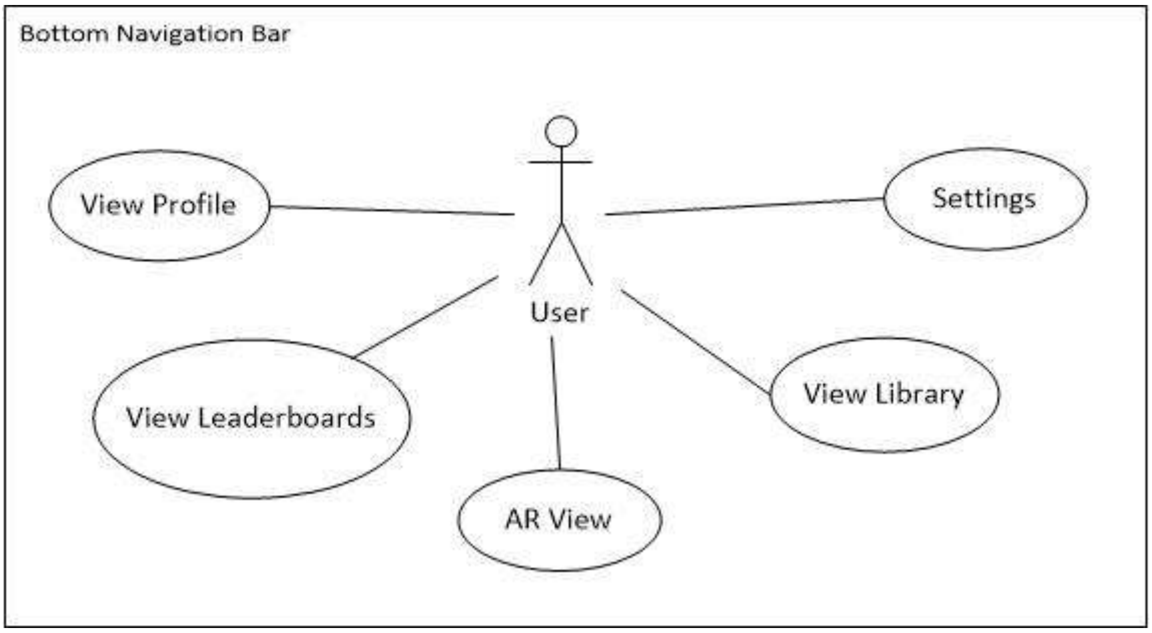


Figure 19. Screen Navigation Use Case

The user wants to see his profile:

Registered users can access their profile page by pressing the “profile” button. Guest users are informed that this page is available only to registered ones. In the profile the users can see their details, edit them, view their progress and the monuments they have marked and visited.

The user wants to see the leaderboards:

All users can proceed to the leaderboards page by pressing the “Leaderboards” Button. In the Leader Boards screen the user can swap between the players’ scores and the monuments stats.

The user wants to enter the AR Camera View:

All users can swap to the AR View from the map and proceed with 3D Navigation. This case is explained with more detail in the next subsection.

The user wants to see his collection:

Registered Users can access the Collections screen where all the available historical information is displayed. The user can then swap between periods, navigate to the monument details, get directions, and visit external links.

The user wants to change the settings:

All users have access to the settings page where they can customize the application.

3.2.5 The User enters the AR Camera View

The user will be able to experience the 3D Localization in the AR View. Contrary to a 2D Map, the content is overlaid on the phone’s camera in their real locations.

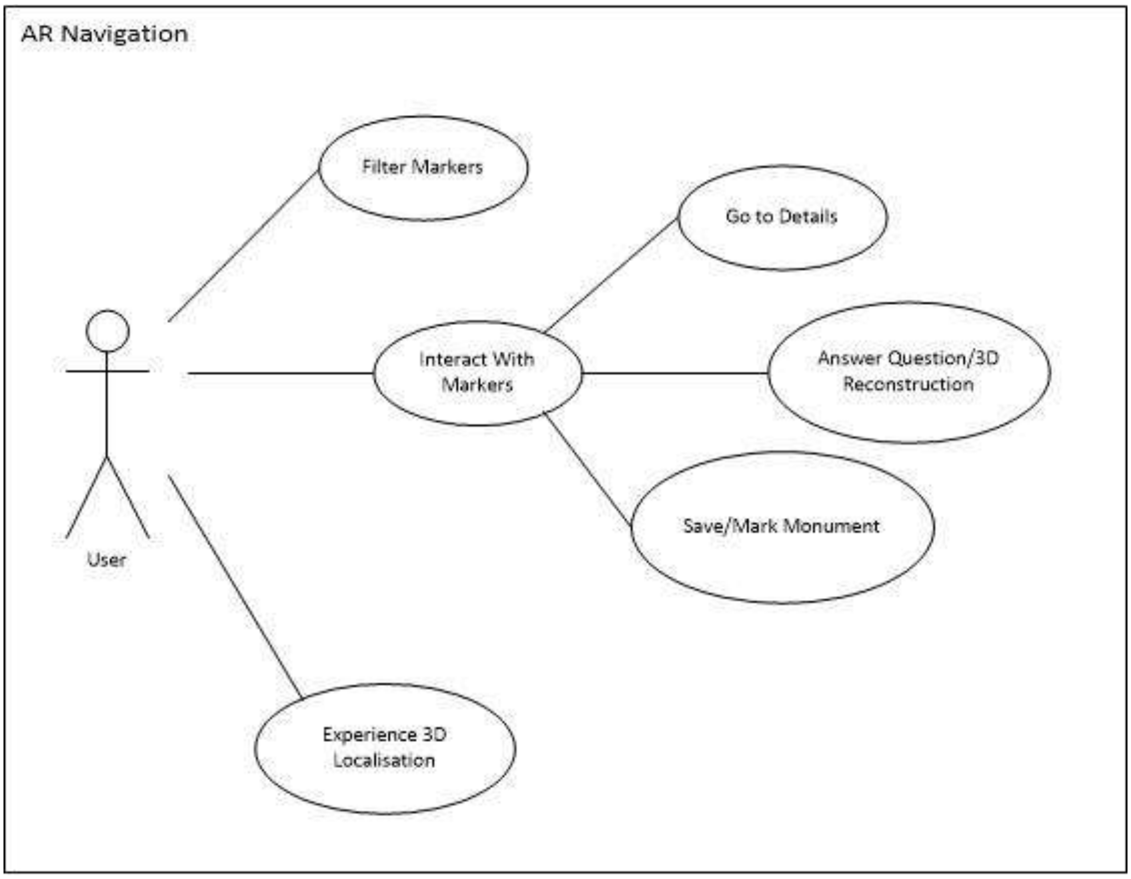


Figure 20. AR Navigation Use Case

The user wants to interact with a marker:

The user can select markers by clicking on them. Only one marker can be selected at a time. Details about the corresponding monument will be displayed at a panel at the bottom of the screen. The user can mark the monument, proceed to the corresponding AR activity or see the monuments details from buttons on said panel.

The user wants to see the 3D reconstruction or classify a monument:

The user can transfer to the AR experience by pressing the corresponding button. This transfer is only available if the user is at the monument.

The user wants to see the details of a monument:

The user can navigate to the details page of the monument by pressing the corresponding button.

The user wants to save the monument:

The user can save the selected monument by clicking on a toggle button appearing on the panel.

3.2.6 The User wants to change the Settings

The bellow use-case is available by pressing the settings option from the navigation bar. The user has access to the options depicted below.

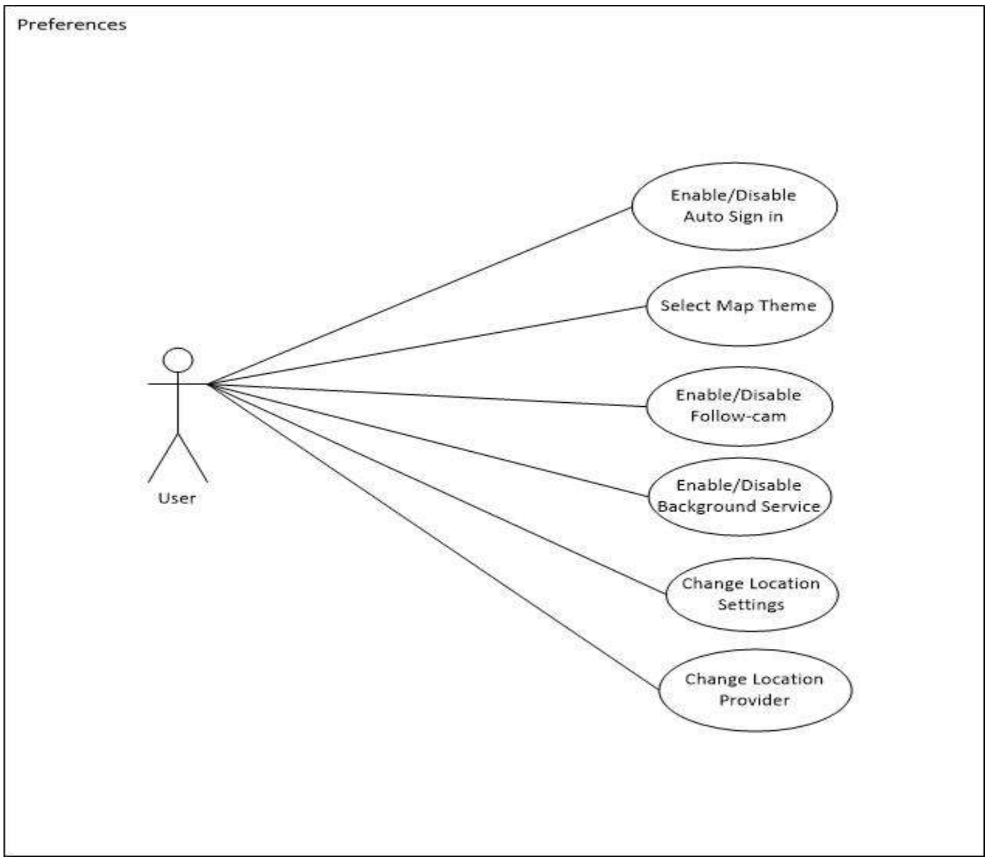

Preferences
Enable/Disable Auto Sign in
Select Map Theme
Enable/Disable Follow-cam
Enable/Disable Background Service
Change Location Settings
Change Location Provider
User

Figure 21. Settings Use Case

The user wants to enable/disable the auto-sign-in option:
The user can enable or disable the auto-sign in option.

The user wants to change the GPS location settings:
The user can choose between the available location settings profiles concerning GPS frequency and mobile network usage.

The user wants to change the theme of the map:
The user is able to change the map theme.

The user wants to disable/enable the auto-follow camera:
The user can enable or disable the auto-follow camera in the map.

The user wants to disable/enable the Background Location Service:
The user can enable or disable the background location service which is responsible for notifying the user about nearby monuments when the app is in the background.

The user wants to change location providers:
The user can swap between the GooglePlayServices Location API and the Android Location API.

4 Prototyping and Storyboarding

The initial step to start developing the application is to create some prototypes and wireframes of what the app screens and their corresponding UI might look like. Prototyping gives as a good insight on the problems that might occur and help us eliminate them before we are too deep into development. Now that we have an overview of the app's functionality and how it interacts with the user, we aim to encapsulate these aspects in our design.

As in any mobile application a clear definition of its screens and the navigation between them is required to ensure that the experience is properly delivered. Below we outline the basic aspects needed for a location based application and the way our design aims to solve them, how we decided to define our activities and how the navigation between the apps screen is accomplished.

4.1.1 Map Screen

To navigate Through the City the user can either use the Map or the AR Cam activities. The map activity is the main screen of our application which facilitates the core of the functionality. In this screen the user gets an overview of his location and the City. In its initial state we only show a Route featuring the AR presentations (Figure 22). When the user reaches a monument in the Route the map zooms to his location and the minimal area needed to interact with the monument is shown around the marker (Figure 22). When the user enters the highlighted area, the marker is then active and he can transfer to the AR screen. The user can navigate in free and guided mode. In the guided mode the map's camera follows the user's location at all times. Panning, zooming and rotating are still enabled but when a new location is received the camera returns to the user's position. In free mode the user can move the camera to any location.

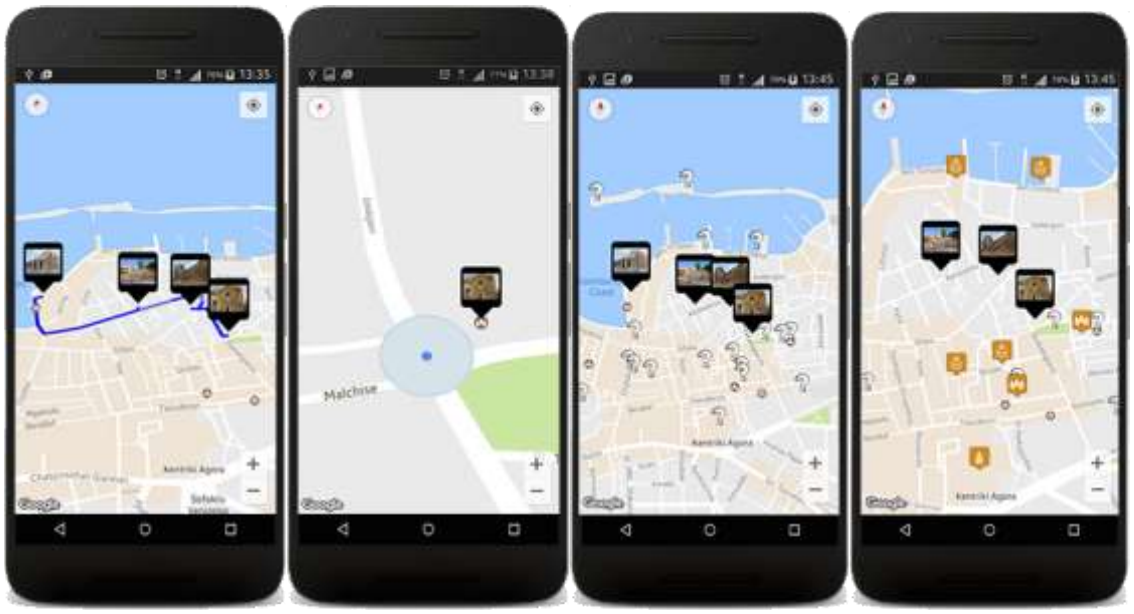


Figure 22. Map Activity prototypes. From Left to Right: Initial State, monument interaction, unknown areas revealed, unlocked areas.

4.1.2 AR Navigation Screen

The user will be able to switch from the map to the AR camera displaying the locations of interest as labels in the camera's surface (Figure 23). This way the user can get a representation of the content in the real environment. The same functionality as the map applies to the AR camera as an alternate means of navigation. Switching between the map and the AR camera can be done from either screen.

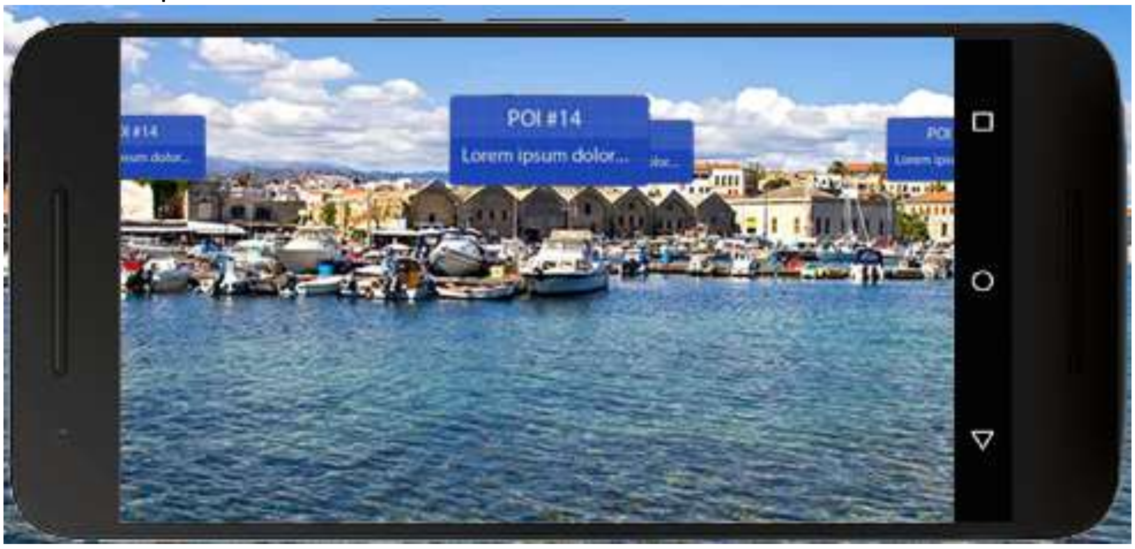


Figure 23. AR Navigation showing 2D labels on top of the real monuments

4.1.3 3D Reconstructions

The AR View is the screen where we overlay the digital content on the camera's surface (Figure 24). The main aspect of it is the reconstruction of the monuments in 3D and the display of historical information in text. The additional information will either be available by clicking on the model or constantly shown in a frame on the screen. The representation starts from the furthest point in time while it gradually moves to the present.

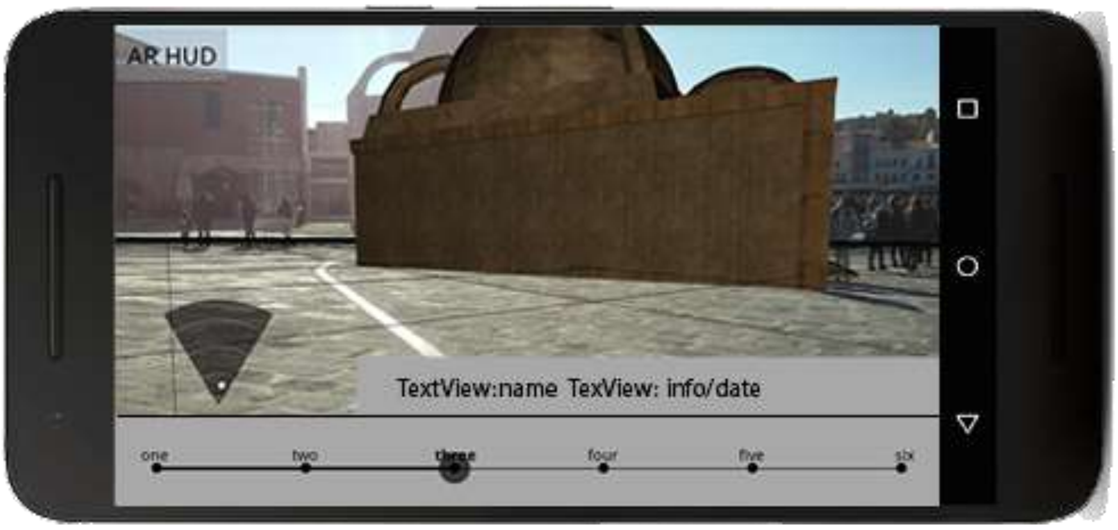


Figure 24. 3D representation in AR

To simulate the passage of time the user can progress through it with the use of 2 buttons (forward, backward) or a seek bar, that change the period and in response the digital content (3D model, text). When the user completes the presentation he is returned to the map. The presentation varies for each site and the information available to us. The user will be able to access the AR presentation anytime he is visiting a given landmark.

4.1.4 Classification in AR

The user is transferred to this screen when visiting an unknown location. Similar to the AR navigation, when the user transfers to this screen a 2D label is shown, on top of the given landmark, prompting the user to sort it to one of the provided historical periods (Figure 25). When the user chooses correctly the label changes to represent the period and the user unlocks the historical information while he gains points for himself.

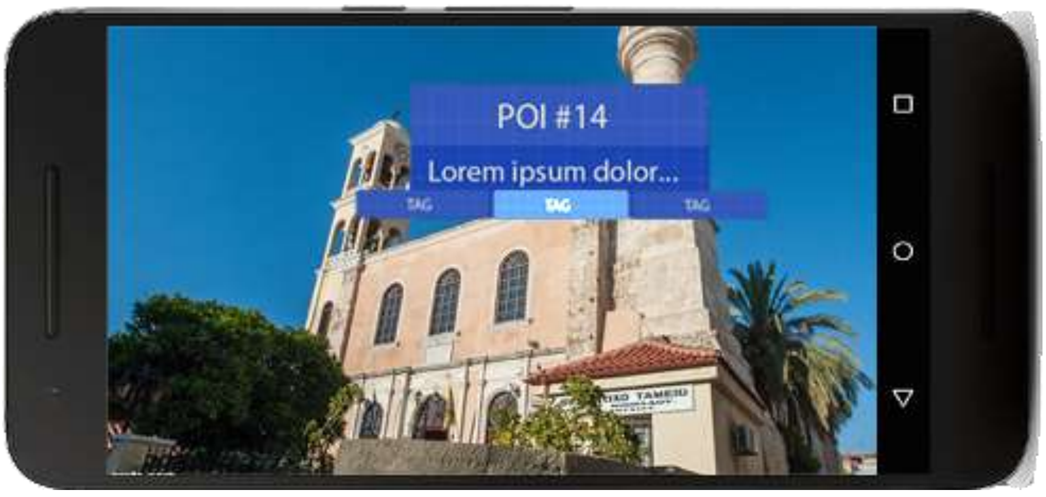
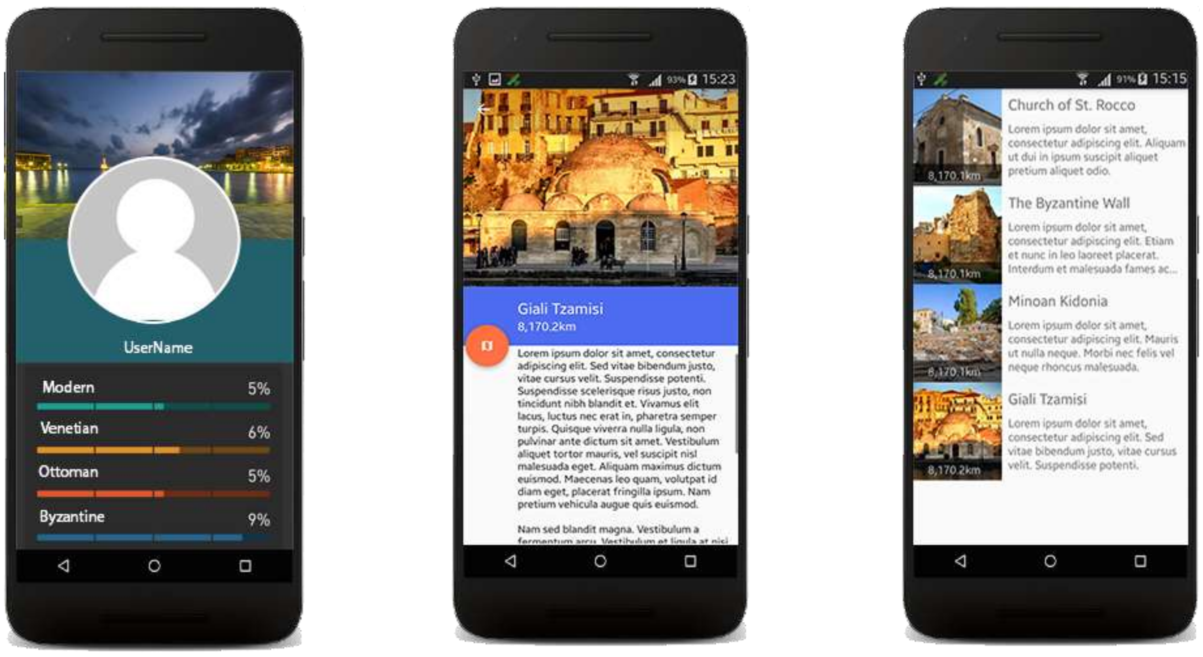


Figure 25. Classification in AR. Selecting the middle choice

4.1.5 Overview Pages

In the details page the user can get an overview of each attraction, historical information, images and links to external sources (Figure 26). The user can transfer to this screen by the map, the AR view and the list view. The list view is the view holding all the monuments. The user can sort them by period, by status or by his selection. Each plate in the list represents a monument with a thumbnail on the left and the corresponding text on the right.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae bibendum justo, vitae cursus velit. Suspendisse potenti. Suspendisse scelerisque risus justo, non tincidunt nibh blandit et. Vivamus elit lacus, luctus nec erat in, pharetra semper turpis. Quisque viverra nulla ligula, non pulvinar ante dictum sit amet. Vestibulum aliquet tortor mauris, vel suscipit nisl malesuada eget. Aliquam maximus dictum euismod. Maecenas leo quam, volutpat id diam eget, placerat fringilla ipsum. Nam pretium vehicula augue quis euismod.

Nam sed blandit magna. Vestibulum a fermentum arcu. Vestibulum et ligula at nisi

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ut dui in ipsum suscipit aliquet pretium aliquet odio.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam et nunc in leo laoreet placerat. Interdum et malesuada fames ac...

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris ut nulla neque. Morbi nec felis vel neque rhoncus malesuada.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae bibendum justo, vitae cursus velit. Suspendisse potenti.

Figure 26. Profile Page (Left), Monument Details (Middle), Monument List (Right)

Finally in the Profile screen the user gets an overview of his information and he has access to all the historical information he has unlocked and to what remains hidden. He can track his progress for each period and he can also visit the details page of each monument he has unlocked.

4.1.6 App Navigation

In the next figure we layout the basic navigation between the above screens. For the initial design we use a wide implementation rather than a deep one, making every view easily accessible from the main screen. The main screen of the application is the Map View. At any time the user can change between the Map view and the AR view while the app navigation remains intact. The user has access to the profile page and the list page through buttons in the main screen.

If the user is at a specific location he can initiate the corresponding AR experience. The details page is available through the map, if marker is selected, after the completion of any AR experience, and through the list view where all the available monuments are displayed. The screens' hierarchy is preserved when navigating backwards.

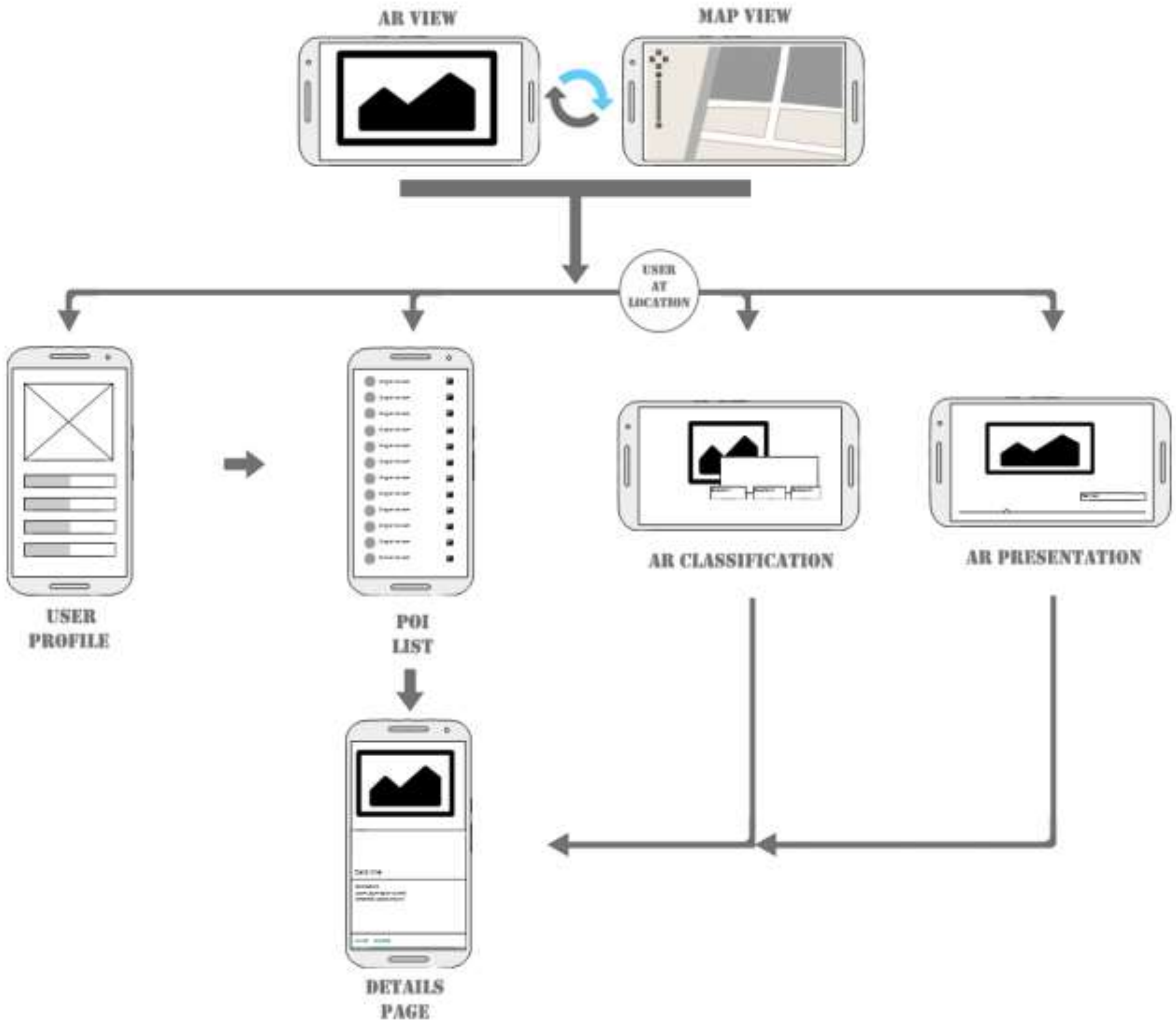


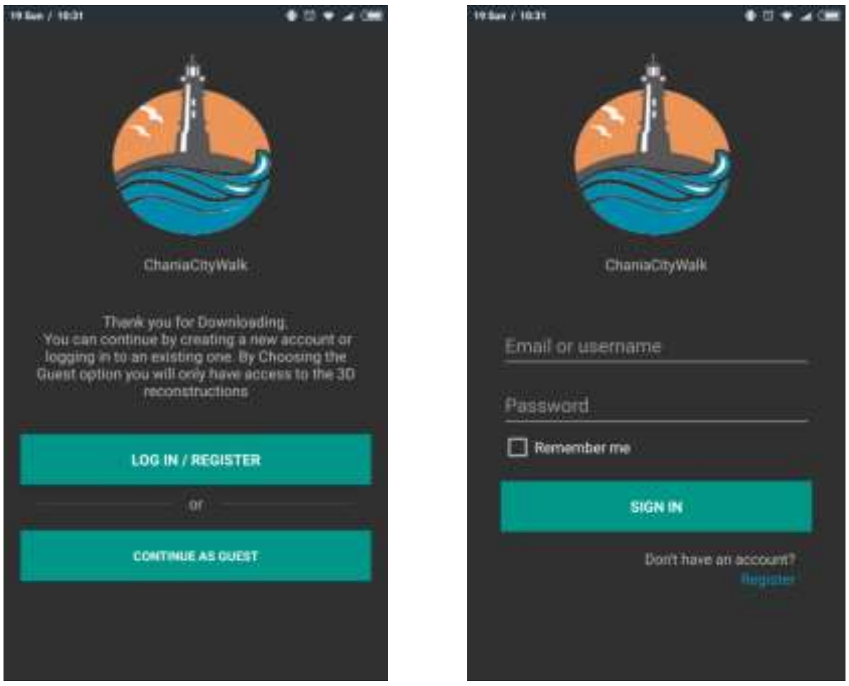
Figure 27. Navigation between the available screens

4.2 Final screens and User Experience

In this section we present the final screens of the application and explain in detail the flow of the experience while we justify any deviations from the initial design.

4.2.1 Login/Splash Screen

Upon the activation of the application the user is welcomed in a splash screen. In this screen the user can choose whether to create an account or login to an existing one, or continue as a guest and have access only to the 3D reconstructions (Figure 28).


ChaniaCityWalk

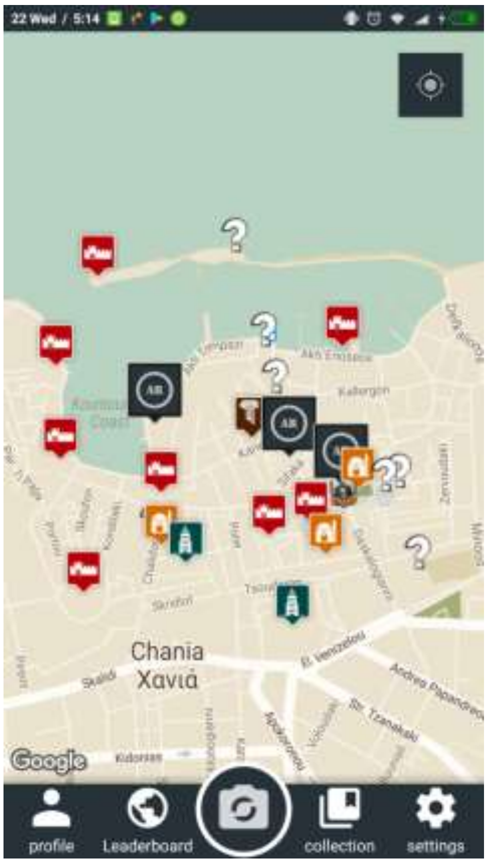
Thank you for Downloading.
You can continue by creating a new account or logging in to an existing one. By Choosing the Guest option you will only have access to the 3D reconstructions

Figure 28. Splash screen (Left), Login/Register form (Right)

If the user chooses the login/register option, the form changes so the user can enter the required credentials. The user can swap between the login and register form from the corresponding highlighted text at the bottom of the screen. After the login process the user transfers to the main screen.

4.2.2 Map Screen

In this screen all POIs are displayed on the map in their corresponding geo-locations. By clicking on a marker the user can see the info window of the POI containing its name, a thumbnail and the distance between them (Figure 29). By clicking the Info window itself the user is transferred to the Details Page of that monument. From the bar at the bottom of the screen, the user can navigate to the remaining screens of the application. These include his profile, the leader-boards, the Collection and the preferences, and the AR navigation available from the middle round button.



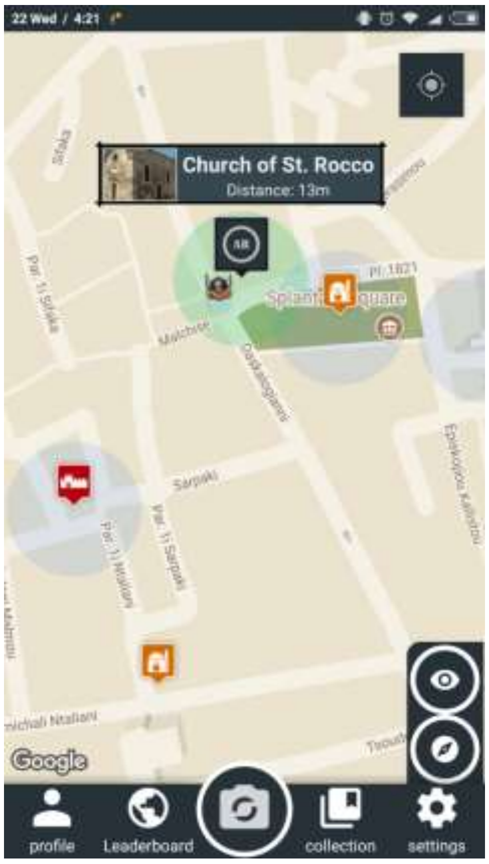


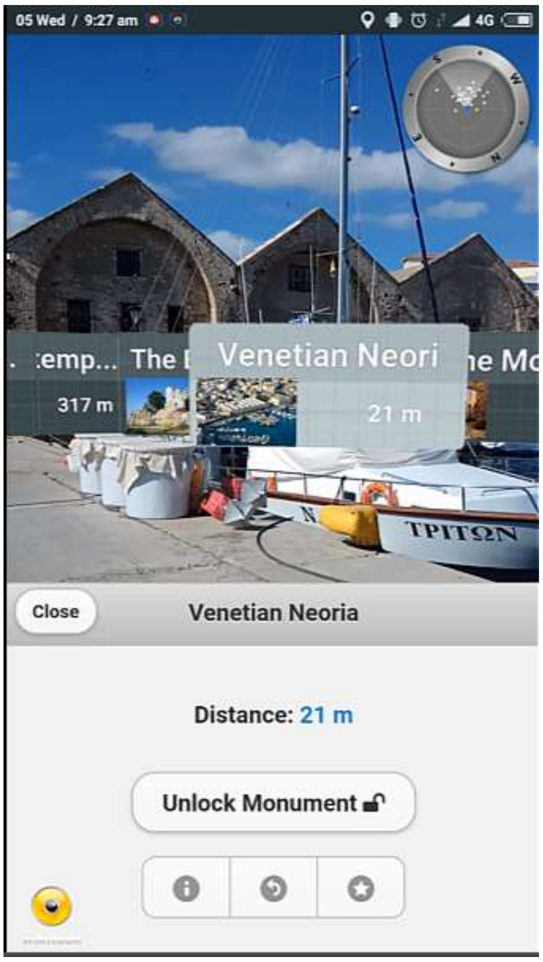
Figure 29. Map Screen showing explored and unexplored areas (left), monument interaction (Right)

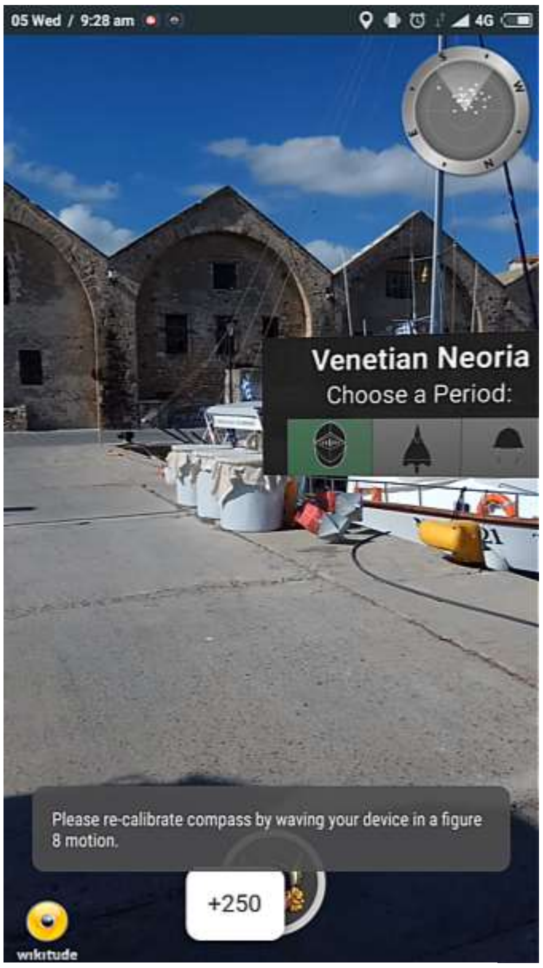
In the initial state the user is only shown the available 3D reconstructions, AR markers in Figure 29. In order to interact with a monument the user needs to be inside the highlighted area as shown in the above figures. When the user enters the area the color changes to note the activation of that monument. If that monument has available 3D reconstructions a sidebar is revealed with two buttons used to transfer AR activities with the two different tracking options.

After visiting these monuments and viewing them in AR, the rest of the POIs locations are unlocked and displayed on the map, question marks in Figure 29. The goal is to visit them and classify them to the provided historical periods based on their architectural characteristics and on clues obtained in the Collection page and from the already visited monuments.

4.2.3 AR Navigation Screen

In this screen the content is displayed in the real world as 2D labels, which contain basic information about the POI. By clicking on a label a bottom drawer appears which holds a brief description if the monument has been explored, and allows for more interactions (Figure 30). Users can save the POI for later reference, access the reconstruction if available, or return to the map with the camera centered on the selected monument. If the label represents an unexplored area the user can transfer to the AR classification page, shown in Figure 30. Radar with all the available locations is shown on the top right corner of the screen.





Please re-calibrate compass by waving your device in a figure 8 motion.

Figure 30. AR navigation screen. Selecting a marker (left). Classifying the monument (right)

4.2.4 3D Reconstruction Screen

The 3D reconstructions are the main feature we aimed to provide (Figures 31, 32). In this screen a reconstructed 3D model of the monument is overlaid on the camera and the GPS and inertial sensors are exploited to display the monument on its real location. The users can freely move around the real site to view the monuments from all available angles. They can access the slider, available from the bottom right button, to change the representation. In the current state of the application the user can change between the whole models and the reconstructed parts.

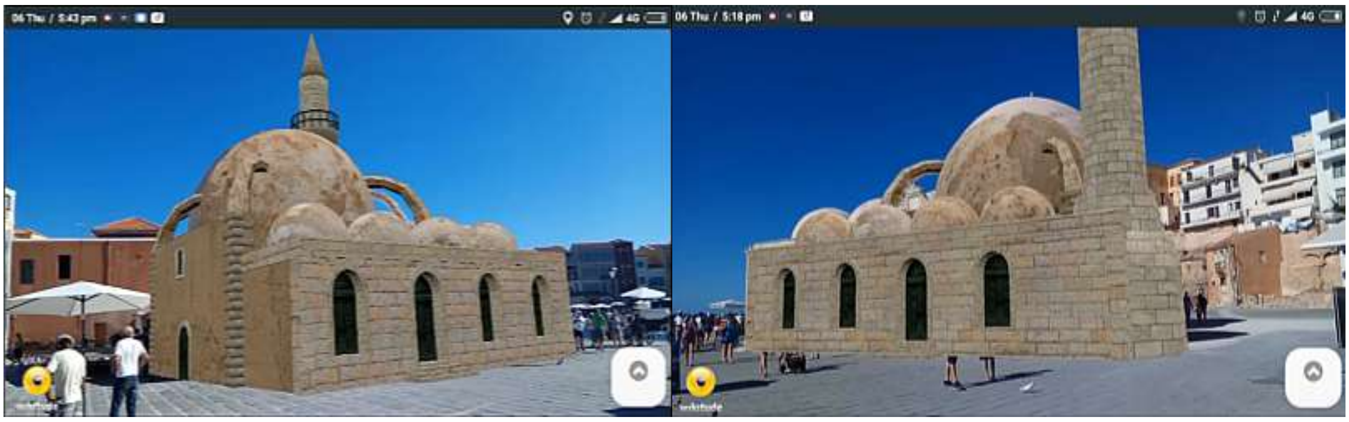


Figure 31. 3D reconstruction of the Glass mosque featuring the now demolished minaret, as seen by the mobile's camera

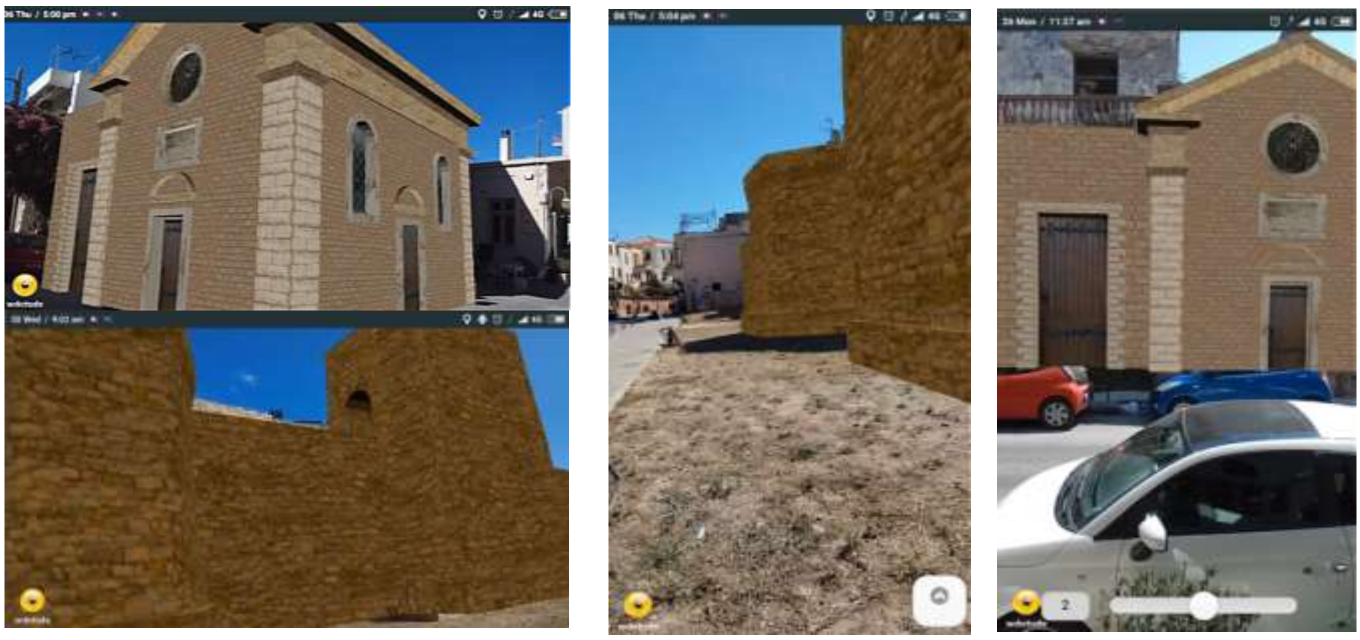
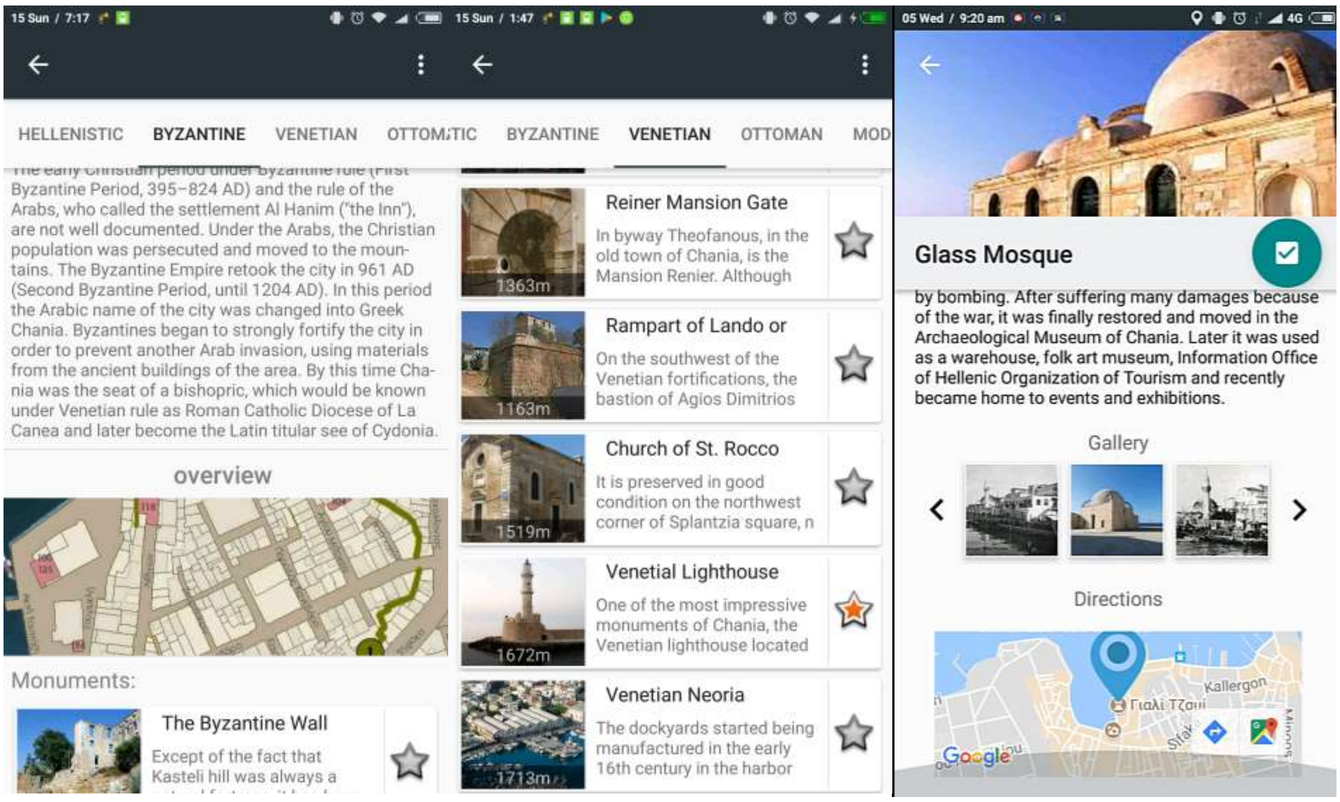


Figure 32. Reconstructions of the demolished towers of the Byzantine Wall and the facial restoration of the Rocco temple

4.2.5 Collections Page

The Collections page is where a collection of all the historical information is displayed. It consists of a view pager containing all periods in chronological order; the user can swap right and left to change through the available periods. Each page has a historical briefing, an image showing the active area for that period and a list with all the monuments that have been correctly classified. The locked monuments are contained in a separate list at the end of the pager. The users can see the monument specific information by selecting the items on the list (Figure 33).



The early Christian period under Byzantine rule (First Byzantine Period, 395–824 AD) and the rule of the Arabs, who called the settlement Al Hanim ("the Inn"), are not well documented. Under the Arabs, the Christian population was persecuted and moved to the mountains. The Byzantine Empire retook the city in 961 AD (Second Byzantine Period, until 1204 AD). In this period the Arabic name of the city was changed into Greek Chania. Byzantines began to strongly fortify the city in order to prevent another Arab invasion, using materials from the ancient buildings of the area. By this time Chania was the seat of a bishopric, which would be known under Venetian rule as Roman Catholic Diocese of La Canea and later become the Latin titular see of Cydonia.

The Byzantine Wall
Except of the fact that Kasteli hill was always a

Reiner Mansion Gate
In byway Theofanous, in the old town of Chania, is the Mansion Renier. Although

Rampart of Lando or
On the southwest of the Venetian fortifications, the bastion of Agios Dimitrios

Church of St. Rocco
It is preserved in good condition on the northwest corner of Splantzia square, n

Venetial Lighthouse
One of the most impressive monuments of Chania, the Venetian lighthouse located

Venetian Neoria
The dockyards started being manufactured in the early 16th century in the harbor

Glass Mosque
by bombing. After suffering many damages because of the war, it was finally restored and moved in the Archaeological Museum of Chania. Later it was used as a warehouse, folk art museum, Information Office of Hellenic Organization of Tourism and recently became home to events and exhibitions.

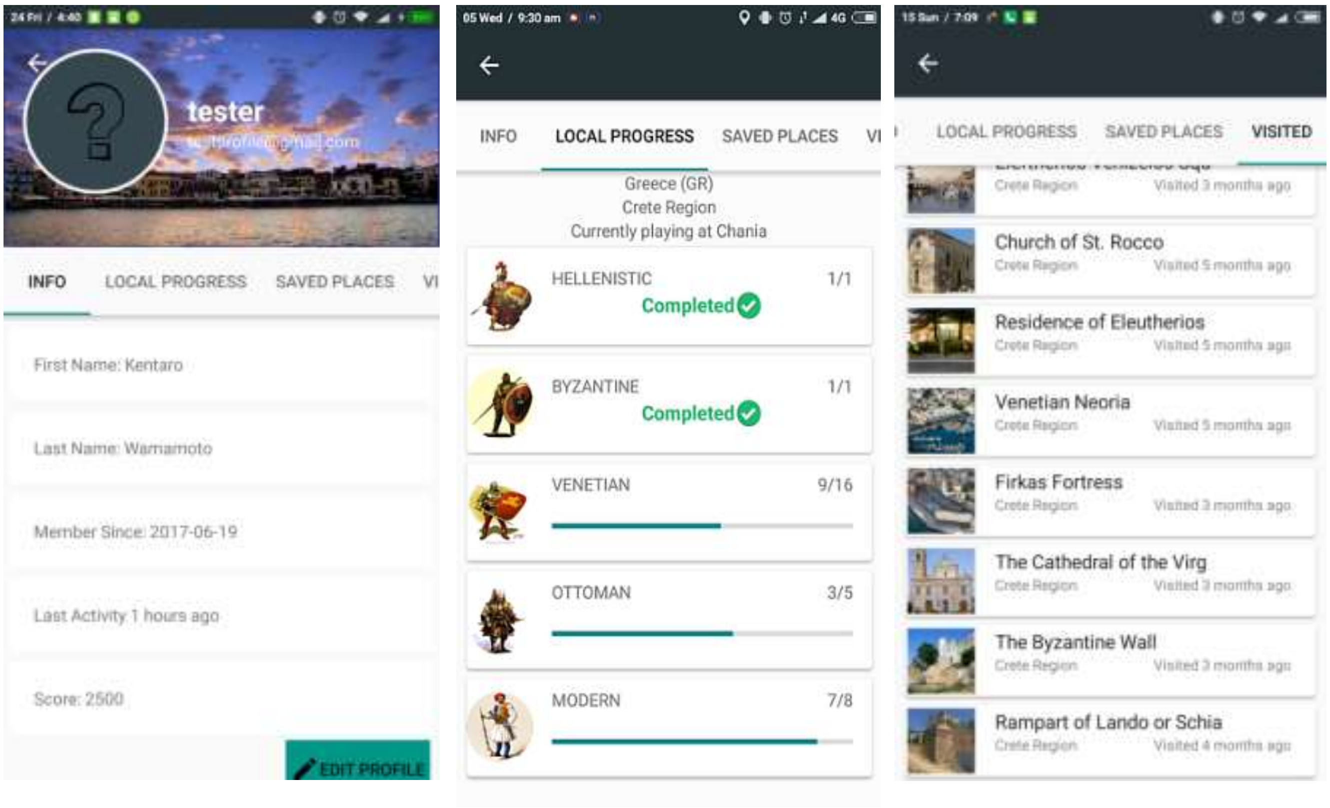
Figure 33. Collections Screen showing the historical information of each period and the list of its monuments (Left). Details screen of the Glass Mosque (Right)

By clicking an item plate the user transfers to the Details activity where all the monument specific information is displayed. In the gallery region of the screen the user can swipe right and left swipe through the stored photos or bring them at full screen by clicking on them. He can mark and save a monument, from the button at the name plate and get directions to the monument's location from the bottom map image.

4.2.6 Profile Screen

The profile and leader-board page follow the same structure as the Collections (Figure 34). In the profile page the user can swipe through pages containing his information, his local progress and the lists of saved and visited places. By clicking on the progress plates visible from the local progress tab, he can transfer to the Collections for the selected period and by selecting a monument he can transfer to its details page. In his information tab he can swap to the edit profile form where he can edit all the demographic information provided when registering, including his password.

The leaderboard page can be used to get information about the City. The user’s position is highlighted in a list showing the current standings. He can then compare his progress and results with that of the other visitors (Figure 34).


tester

INFO | LOCAL PROGRESS | SAVED PLACES | VI

First Name: Kentaro

Last Name: Wamamoto

Member Since: 2017-06-19

Last Activity 1 hours ago

Score: 2500

EDIT PROFILE

INFO | LOCAL PROGRESS | SAVED PLACES | VI

Greece (GR)
Crete Region
Currently playing at Chania

HELLENISTIC	Completed ✓	1/1
BYZANTINE	Completed ✓	1/1
VENETIAN		9/16
OTTOMAN		3/5
MODERN		7/8

LOCAL PROGRESS | SAVED PLACES | VISITED

Eleftheriou Venizelou Squ	Crete Region	Visited 3 months ago
Church of St. Rocco	Crete Region	Visited 5 months ago
Residence of Eleutherios	Crete Region	Visited 5 months ago
Venetian Neoria	Crete Region	Visited 5 months ago
Firkas Fortress	Crete Region	Visited 3 months ago
The Cathedral of the Virg	Crete Region	Visited 3 months ago
The Byzantine Wall	Crete Region	Visited 3 months ago
Rampart of Lando or Schia	Crete Region	Visited 4 months ago

Figure 34. Profile View showing local progress (left) and visited monuments (right).

4.3 Modeling the Past

In order to record the past state of the selected monuments, old photographs, historical information and estimates from experts were utilized. The 3D models visualizing their past state will be presented in real size superimposed over the real-world monument and must be in proportion with their surroundings. Therefore, accurate measurements of their structure are necessary. Due to the lack of schematics and plots, we relied on data derived from online mapping repositories which provide outlines and height. In order to ensure historical accuracy and avoid the communication of false information, the final models and their reconstructed parts are in abstract form, depicting only the main structural elements of each monument.

4.3.1 Data Acquisition

The outlines of the three monuments were acquired from OpenStreetMap (OSM). By selecting specific areas of the monuments on the map, we can then export a .osm file that contains the available information concerning that area, including building outlines and height, where available. This file is essentially an xml file including all OSM raw data including roads, nodes, tags etc. The file is then imported into OSM2World, a Java application whose aim is to produce a 3D scene of the underline data, (Figure 35).

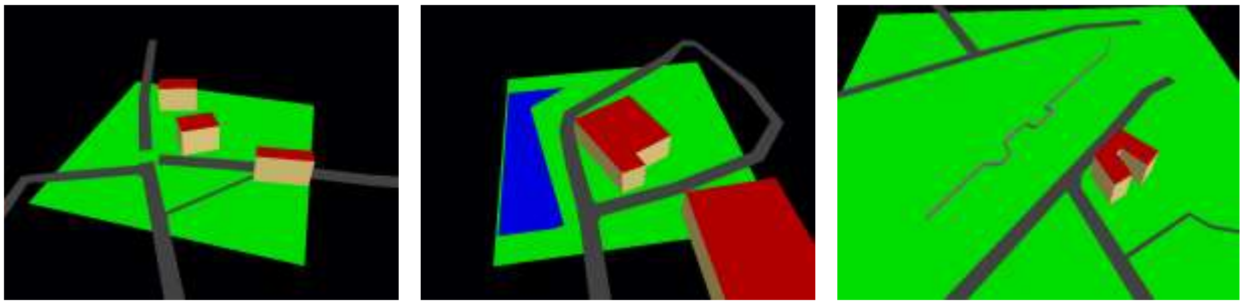


Figure 35. 3D Scene of the OSM Data as produced by the OSM2World. St.Rocco in the middle (Left), Glass Mosque (Middle), The Byzantine Wall (Right)

These representations are basic triangulated meshes of the outlines raised to reach the height value for each building. As evident from the Byzantine Wall (Figure 35) height data are not always available and in the other two cases it is not clear if the domes and roofs have been taken into consideration. The decision was made to continue with these models as a basis and any disproportions would be corrected after the on-site testing. The models were then exported to .obj format and imported to the Blender 3D modelling software.

4.3.2 3D modeling

With the basic structures of the buildings and information about the reconstructions we proceed to create the final 3D mesh that will be used in the application. The modelling process was focused on preserving a low vertex count as complex geometry compromises interactive framerate in systems with low processing power such as mobile phones. In the initial stage we only created the demolished parts, fig11, to be overlaid on the real buildings.

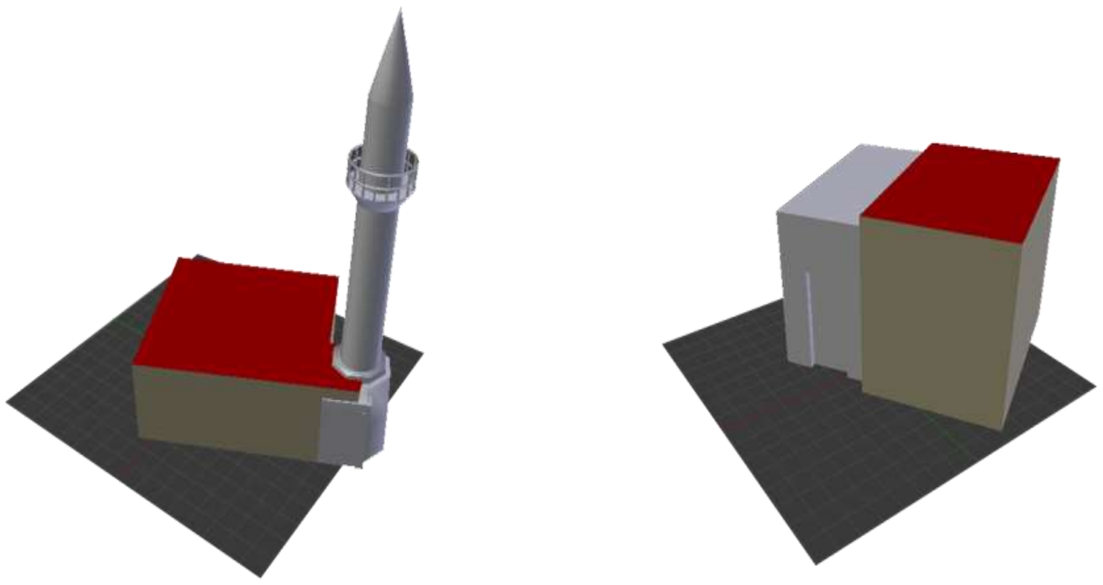


Figure 36. The 3D mesh of the reconstructions in scale with the existing buildings

On site testing showed that the three meters average accuracy of the GPS receiver and the constraint viewpoint in some sites (St.Rocco for example) completely breaks the illusion as the parts were not registered accurately with the environment. Instead we created the whole buildings with their reconstructed parts, to completely overlap the real ones. The most important aspect is to keep the reconstructions in proportion. The final scale and size will be accounted for in Google sketch-up while we position the final model in the world coordinate system. The final models can be seen at Figure 37. These models constitute the whole buildings as they used to be. Reference images were used for modelling the existing parts while more details were added to keep them consistent with the real ones. This detail work, especially in the domes and railings of the Glass Mosque(a) and the columns in Saint Rocco(c), unavoidably heightens the poly count but the final results were deemed satisfactory. The Byzantine wall(b) being the most abstracted, resulted in 422 vertexes. The Glass Mosque counts 7,614 and the Saint Rocco temple 4,919 vertexes.

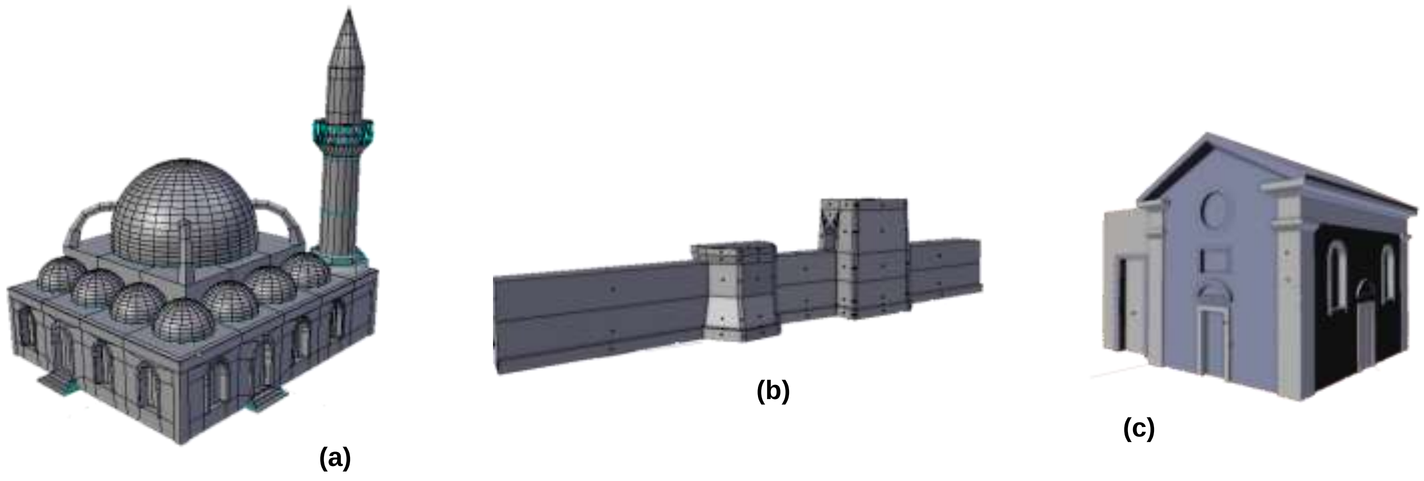


Figure 37. The final 3D meshes (a)Glass Mosque, (b)Byzantine Wall, (c)St. Rocco temple

4.3.3 Texture and Lighting

Texture mapping is a method of adding photorealism and detail to a flat surface without adding extra geometry. A texture map is a bitmap image that is applied to the surface of a polygon creating a high fidelity visual result. Images from the monuments were used as references to locate the surface materials. The texture mapping procedure followed a multi-material approach were different materials are assigned to different parts of the buildings. Each material is then assigned images to apply to the surfaces. Due to the lack of information the actual texture of the reconstructed parts is unknown so the aim is to more accurately represent the compositing material rather than the actual surface. At this point of the procedure, an account of the rendering capabilities of the AR framework needs to be made. The framework supports only a power of 2 png or jpeg single material texture map. That means that we cannot include bumps, normal maps and multi-textures to more accurately represent the surfaces.

The materials that compose the entire texture set are baked into one image that will serve as the final texture. UV mapping is the process of unwrapping the 3D shape of the model into a 2D map. This map contains the coordinates of each vertex of the model placed on an image. The materials that were assigned to the surfaces are then baked onto the image that forms the final texture. Taking into account that the monuments will be displayed on a mobile phone screen in real size, we needed high resolution textures. This unfortunately raises the final size of the texture files, however, the process results in a high quality visual result. The final texture resolution is 2048x2048 pixels (Figure 38). In order to light the scene, we used a simple hemi light provided by 3D the modelling software. The hemi light is a 180-wide uniform and shadow-less light. Although the AR framework supports other light sources that produce shadows and make the scene more realistic, we cannot adjust its position during run-time. This would lead to misaligning shadows and wrongly lit surfaces between the real environment and the 3D models. The hemi lamp provides lighting from all angles and does not produce shadows that would impair the experience.

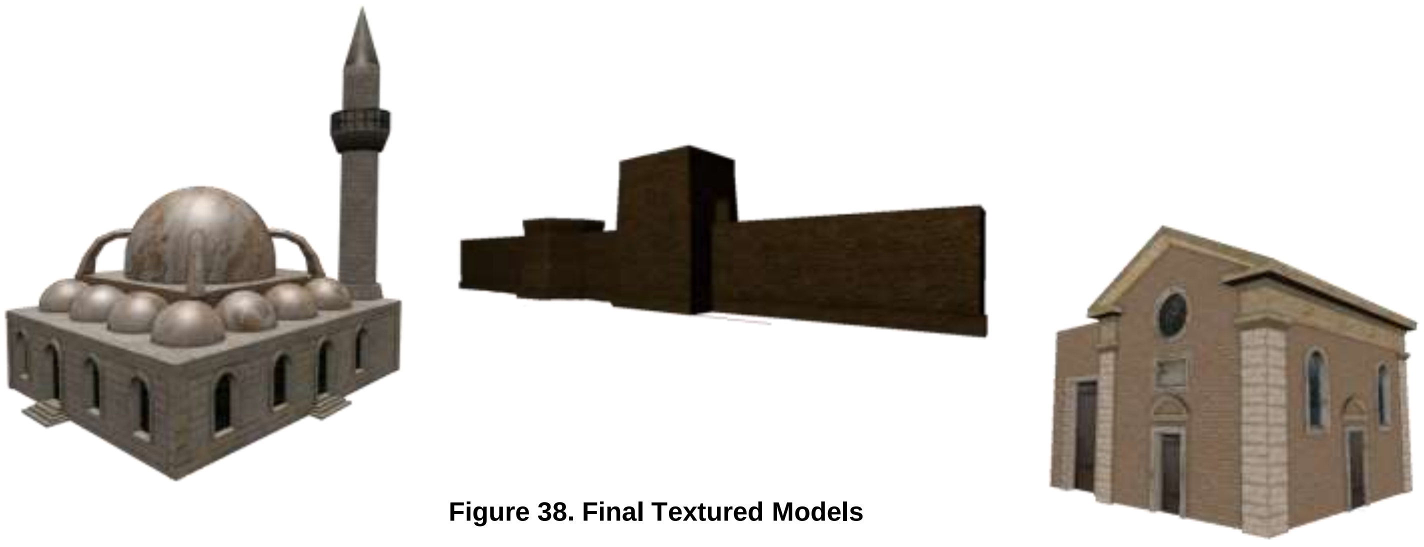


Figure 38. Final Textured Models

4.3.4 Geo-Positioning

In order for the reconstructions to be accurately displayed combined with real-time viewing of the real world, an initial transformation and rotation needs to be applied. The models were exported in .dae format and imported into Google Sketch-up. The area of the monuments provided by Google Maps is projected on a ground plane. We then position the monument on its counterpart on the map. Given that the proportions of the monuments are in line, the final model is scaled to fit on the outlines. The location of the monument is then added to the file and provided to the framework.

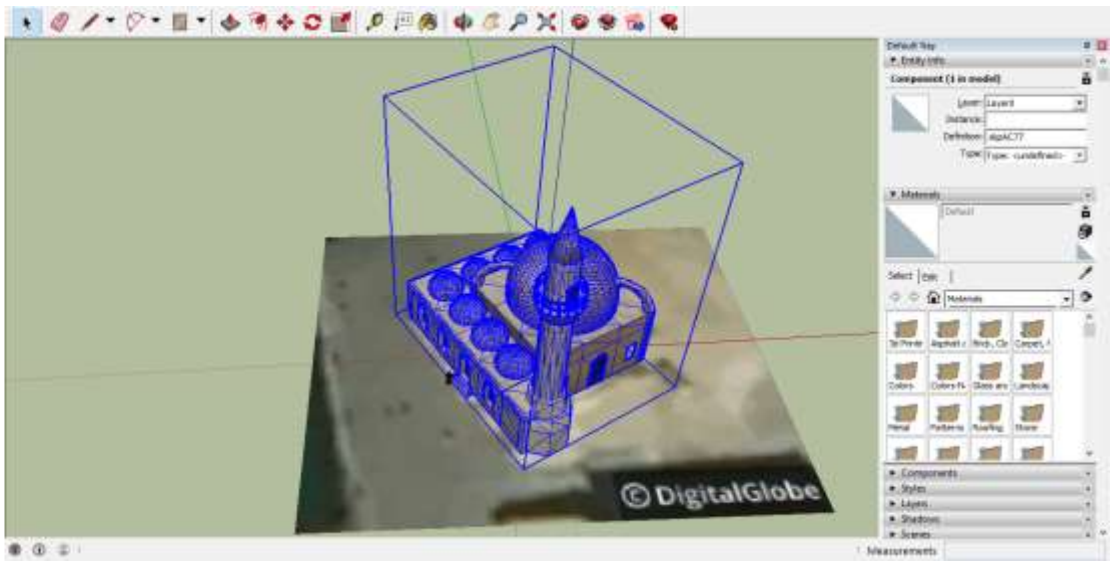


Figure 39. Positioning the model at its geo-location with

In order to include the model in the AR framework, it needs to be in the Wikitude 3D file format (.wt3). This is a compressed binary format used and introduced by Wikitude, for fast loading on mobile devices. To convert the file to this format we use the Wikitude 3D Encoder, a desktop application provided by the SDK. Currently it can only convert models from the .fbx format, which is used to provide interoperability between different digital content creation applications. After the conversion the final wt3 file, which includes the models and their textures in a single file, is deployed with the assets of the application.

5 System Architecture

5.1 Client-Server Architecture

In the previous chapter we analyzed the requirements that our application needs to meet. Now that the basic functionality is laid out it is time to start mapping out the implementation of our overall system. From early on in the process it was clear that our system could not only consist of a stand-alone mobile application.

Our overall implementation is based on client-server architecture. This decision was made to provide an easily expendable platform that will allow the seamless inclusion of additional sites and Cities, as well as providing useful statistics about the users and the monuments, without interfering with the application. Moreover it ensures that the data can reach additional platforms, other than the Mobile Application, that will allow for the future development of tools that e.g. create and handle additional content.

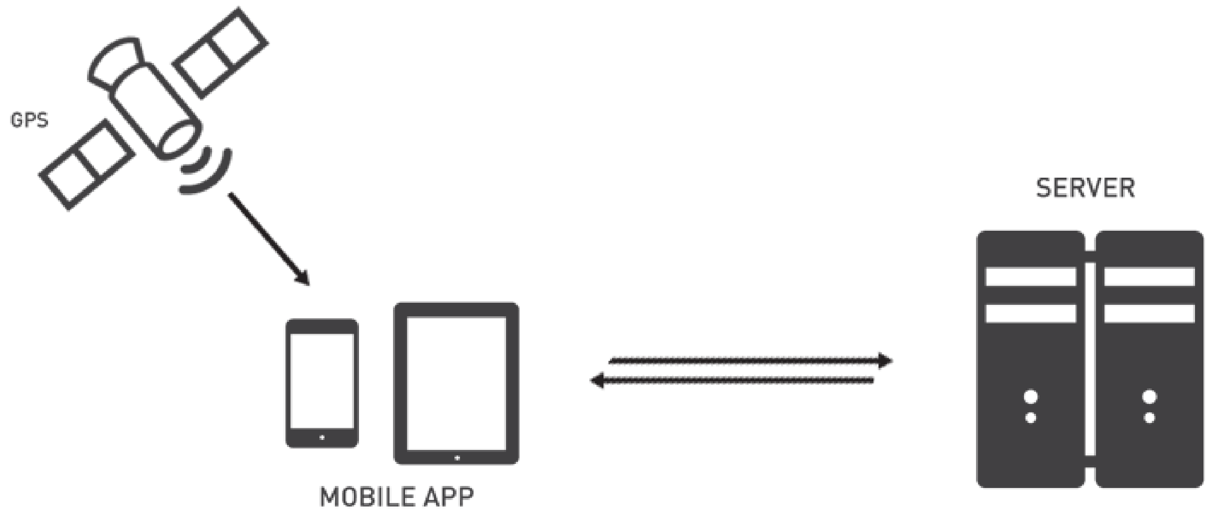


Figure 40. System Overview

In our design the server features a database with all the historical and user specific information and is responsible for serving them as requested by the mobile client. A web based API facilitates the communication of the two and a registration to the system is necessary to provide personalized information. A concept that needs to be addressed at this point is that the application facilitates a location based experience. This means that for the information to be of any use to the client, the latter’s physical presence at any interest area needs to be established. So the mobile client request all the information based on its location. The server concludes if the location corresponds to any content and responds accordingly. In the next sections we will explain in detail the System's design shown in Figure 41 and then proceed to its implementation.

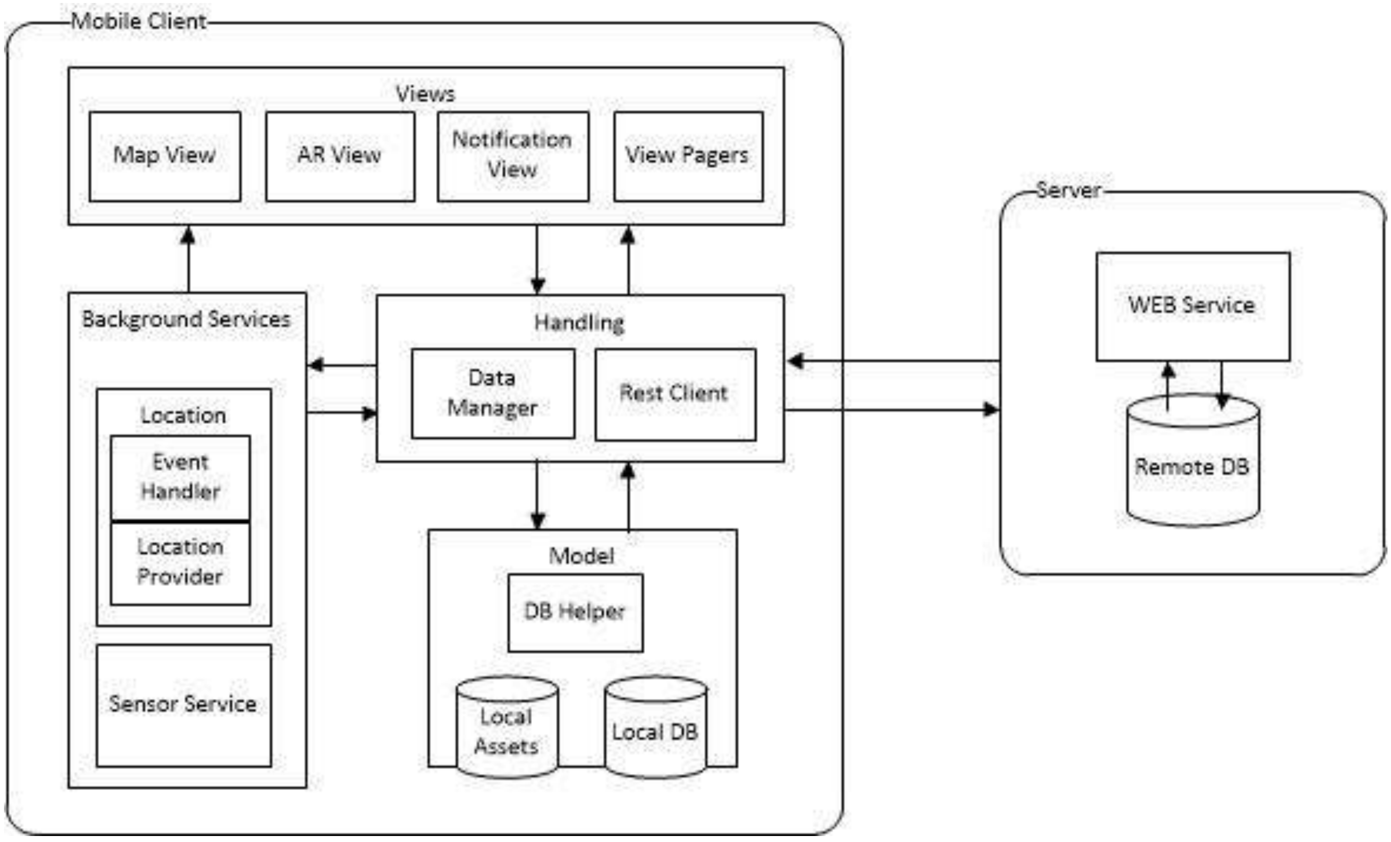


Figure 41. System Architecture

5.2 Mobile Application

One of the main challenges we faced in designing our MAR application was the lack of established guidelines in the application and integration of AR technologies in outdoor heritage sites. Since our experience takes place in outdoors areas, where internet connectivity is limited and predominately based on data usage, we developed a local database that will cache the information acquired by the server. Our aim was to create a stable platform that will be able to provide all the necessary information and interactions to its users while keeping it consistent with the server.

Our design is based on three main layers (Figure 41) and the background services that provide the GPS and sensor data. The views layer is where the interactions with the users take place. Together with the background location service they act as the main input points to the system. The events that take place are forwarded to the handling layer which consists of two modules. The Data Manager which is responsible for interacting with the local content and communicating with the views and the Rest Client which is responsible for handling requests, to and from, the server. The model layer consists of a local database and basic helper modules to interact with it as well as parse the obtained JSON files. The actions flow from the Views layer and the Background Service to the lower levels. Responding to a user event or a location update, a call is made to the handling layer which will access the model to return the requested data.

5.2.1 Handling Layer

The Handling layer is the most important of the three, all interactions, exchange of information and synchronization passes through this layer. The Rest Client provides an interface for receiving and sending information to the remote database, as requested by the other layers, while the Data Manager is managing the local content. All information received from the server are parsed and stored in a local Data Base to minimize internet usage. These two components are responsible for keeping the two data structures consistent. So if any component needs to request any data form the server it will make a call to the Rest Client, after the data is received the client will employ the Data Manager to update the local database and then the initiating call will resolve. The handling layer acts as the intermediate between the Views and the Model and provides all the possible ways to interact with the underlying data.

5.2.2 Views Layer

The views layer consists of basic user-interface components facilitating all the possible interactions with the users. It is responsible for updating the user-interface after changes. This layer handles all the input events from the users and does what is appropriate to serve them. The components of this layer are the Android activities which consist of the layout files that structure the UI and bound it to event patterns, and the Java classes that handle these events. This layer requests all the needed information from the handling layer.

Each activity is a standalone screen in our application and is responsible for creating its own UI and handling the corresponding events. The Map View is a fragment containing a 2D map developed with the Google Maps API. It displays the user's location, as obtained by the background service, and the points of Interest (POIs) as markers on the map. The AR views are where the AR experiences take place. It is a Web view with a transparent background overlaid on top of a camera surface. It displays the 3D models while it receives location updates from the background service and orientation updates from the underlying sensor implementation. All interactivity is handled in JavaScript. The View pagers are framework specific UI elements that display lists of the POIs, details for each POI, user leader-boards and user profiles. Finally the Notification View is used when the application is in the background and aims to provide control over the location service. It is a permanent notification on the system tray where the user can change all preferences of the location strategy and start, stop, or pause the service at will.

5.2.3 Model Layer

The Model layer consists of standard storing units and handlers to enable parsing JSON files obtained from the server and interact with the local DB. The local DB acts as a cache for the monument information acquired by the server user specific information, and additional variables needed to ensure the optimal flow of the application. The local assets, including the 3D models and the html, JavaScript files required by the Wikitude API, are stored in this layer and provided to the AR experiences as requested. The SQLite Helper is the component responsible for directly interacting with the local storage and offers an interface to the Handling layer, containing all available operations (INSERT, SELECT, UPDATE, DELETE).

5.2.4 Background Services

The Background Services are responsible for obtaining the information from the hardware components of the device, and serve them to the requesting views. There are two services in our application; the location service and the sensor service. The location service is responsible for supplying the locations obtained by the GPS to the Map View and AR Views to update their UI. The location Provider is the component responsible for obtaining the locations and offers the option to swap between the Google Play Services API and the Android Location API, two different location strategies. In order to offer control over battery life and data-usage the users can customize its frequency settings from the preferences. The Event Handler is the component responsible for serving the location events to the registered views. The user’s location is continuously compared to that of the available POIs and if the corresponding distance is in an acceptable range the corresponding event is fired to the listening Views. The aim of the standalone background service is to allow users to roam freely in the city while receiving notifications about nearby POIs. If the application is in the background a notification is issued leading to the Views.

5.3 Server

The aim of the application server is to provide an online storage unit of historical and user specific information. To facilitate client-server communication we used the Representational state transfer (REST) architectural style for providing a web-based API for the client applications. The data can be queried to provide more personalized information to the users and useful statistics about the city.

The Web API exposes its resources via unique custom defined URLs hold by the mobile client. Each key entity in the database schema (Figure 42) is mapped to a relative path from the base URL of the server, and for each entity identified with the URL the client uses different types of HTTP request methods (GET, PUT, POST, and DELETE). By accessing these URLs and defining the http method, the clients can perform all CRUD (Create, Read, Update and Delete) operations on the underlying data. All information is transferred in JSON (JavaScript Object Notation). Below we outline the key entities of our database schema and an explanation of their relationships:

Player Table

The system supports the registration of new users. Minimal requirements for this action are the email, username and password as well as some demographic information. The **visits** and **places** tables are used to record all the interactions of the user with the **scenes**. Each user has a collection of monuments that he has visited or saved. The **player_plays_in_levels** table holds the score of each player for each level and is used to provide level specific leaderboards.

Scene Table

The **scene** table holds all the records about the monuments. Each monument is uniquely identified with an auto-incremented id. For each monument the system records its name, description, latitude and longitude values as well as relative paths to its images. Similar to the players table, the **visits** and **places** entities enable as to keep track of additional statistics about each the monument. Each monument can be a member of only one level (explained below) and relates to only one historical period.

Period Table

The **period** table holds all the information about the historical periods used to classify the monuments. Each period has a name, description, paths to its images and ended and started dates. The period table is used to classify the monuments and provide additional historical information about the **levels**.

Levels Table

The **levels** table is used to identify the playable areas that our system supports. Each level has a Location described in latitude and longitude values, as well as a radius (*'bound'*) that sets its boundaries. Each monument in our database corresponds to a playable area and each area can relate to a number of periods defined by the **level_has_periods** table. In the current state the database only holds information about The City of Chania, Greece, but it can be easily expanded to additional areas.

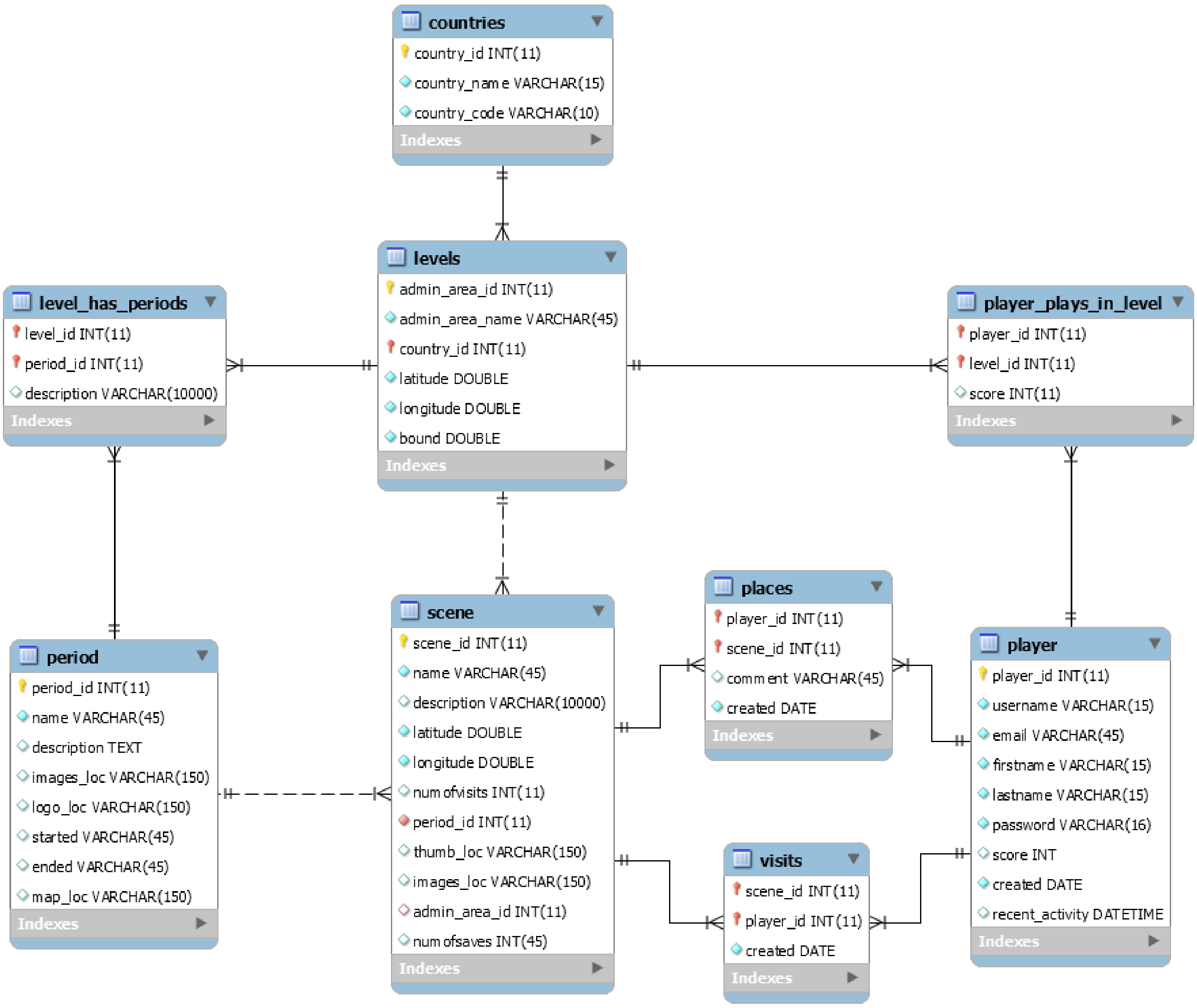


Figure 42. Database Schema

6 Implementation

6.1 Introduction

In this chapter we will describe in detail the implementation of the overall system. We will start with the server implementation that features the Rest Web Service and proceed to the mobile application.

6.2 Server implementation

For the purposes of this Thesis we used an **Apache Tomcat** server running on a local machine. The tomcat server hosts our Rest API which is developed with **Jersey**, a reference implementation of the **JAX-RS** annotation language API. The system's database was developed with **MySQL**.

Apache Tomcat
The Tomcat Server is an open source web server and servlet container developed by the Apache Software Foundation. It implements the Java Servlet and the JavaServer Pages (JSP) specifications from Sun Microsystem, and provides a "pure Java" HTTP web server environment for Java code to run in. In the simplest configuration Tomcat runs in a single operating system process. The process runs a Java virtual machine (JVM) and every single HTTP request from a browser to Tomcat, is processed in the Tomcat process. Apache Tomcat includes tools for configuration and management, but can also be configured by editing XML configuration files.

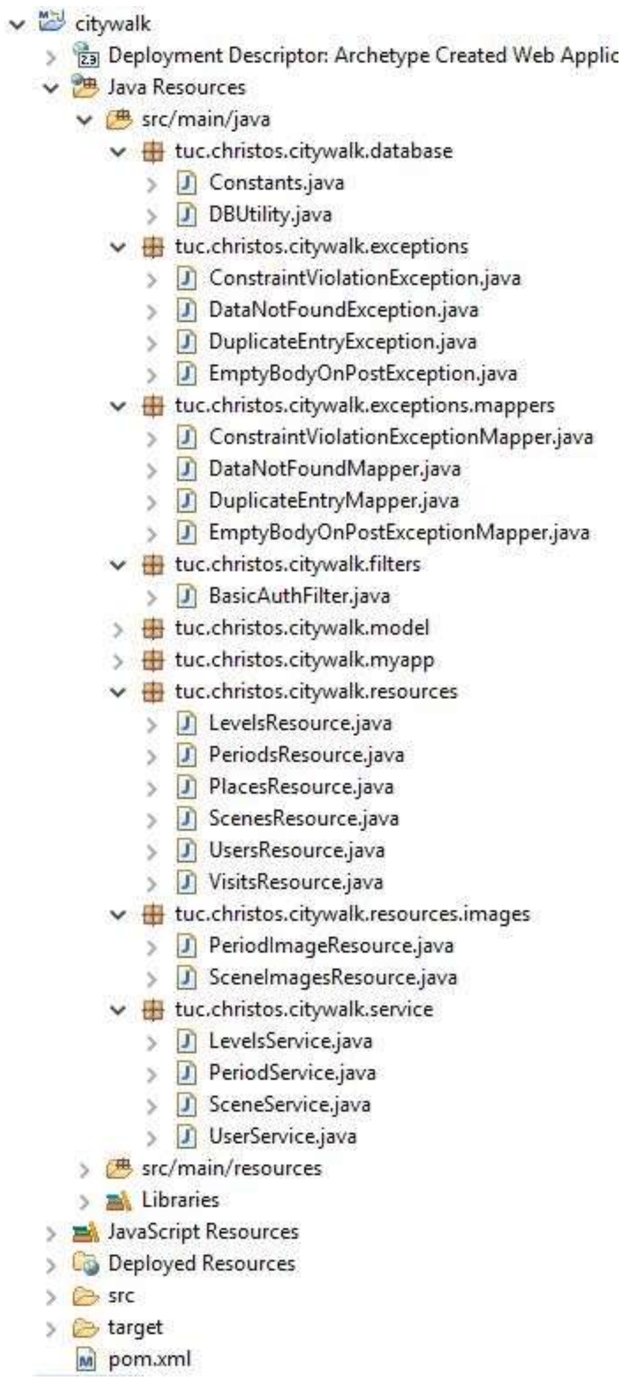
Jersey
The Jersey RESTful Web Services framework is an open source, production quality framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS Reference Implementation. It provides its own API that extends the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.

MySQL
MySQL is a relational database management system (RDBMS) that uses Structured Query Language (SQL), the most widely used language for adding, accessing, and processing data in a database.

Our web API maps its location based on a relative path from the Server's web address. So if the server's address is *'http://citywalk.duckdns.org'* and the API's path is *'citywalk/arapp'* then the Base URL will be '*http://citywalk.duckdns.or/citywalk/arapp'.* Every resource available will increment the Base URL to specify its location. From this point on, every HTTP request made to the server will be mapped to a specific resource of our API which in turn will query the database to perform the requested operations.

6.2.1 Project Structure

The web Service was developed in Java with the Eclipse IDE and the Jersey framework. The project structure can be seen bellow:



```
citywalk
  Deployment Descriptor: Archetype Created Web Applic
  Java Resources
    src/main/java
      tuc.christos.citywalk.database
        Constants.java
        DBUtility.java
      tuc.christos.citywalk.exceptions
        ConstraintViolationException.java
        DataNotFoundException.java
        DuplicateEntryException.java
        EmptyBodyOnPostException.java
      tuc.christos.citywalk.exceptions.mappers
        ConstraintViolationExceptionMapper.java
        DataNotFoundMapper.java
        DuplicateEntryMapper.java
        EmptyBodyOnPostExceptionMapper.java
      tuc.christos.citywalk.filters
        BasicAuthFilter.java
      tuc.christos.citywalk.model
      tuc.christos.citywalk.myapp
      tuc.christos.citywalk.resources
        LevelsResource.java
        PeriodsResource.java
        PlacesResource.java
        ScenesResource.java
        UsersResource.java
        VisitsResource.java
      tuc.christos.citywalk.resources.images
        PeriodImageResource.java
        SceneImagesResource.java
      tuc.christos.citywalk.service
        LevelsService.java
        PeriodService.java
        SceneService.java
        UserService.java
    src/main/resources
    Libraries
  JavaScript Resources
  Deployed Resources
  src
  target
  pom.xml
```

Figure 43. Web-Service structure

In this section we provide an explanation for the packages and classes in the above figure. The **resources** and **service** packages are the main components of the project as they handle the requests from the client and execute the interactions with the database. The **database** package contains the classes that hold the required information needed by the JDBC driver to establish a connection with the database. The resources will be explained in detail in the following section. Each **service** class implements the database interactions for the corresponding resource. It employs the database to execute specific queries and acquires the result sets. It parses the results to the known POJOS and returns them to the Resources.

The **exceptions** package contains all the exceptions that we throw during runtime. The **exception mappers** map these exceptions to Response objects that contain a JSON object with the HTTP status code and a message explaining what went wrong. Each exception mapper is annotated with the `@Provider` annotation to be discoverable by the JAX-RS runtime during the provider scanning phase. When an exception is thrown it will search all registered providers for anyone that handles the specified type of exception. The aim of exception handling is to override the default functionality that responds in HTML. We want the Response objects to have a meaning for the client and to be easily parsed. For example if a client tries to register with an already taken username/email, we throw a *DuplicateEntryException*. The Mapper will then respond to the client with a JSON object that contains a 409 status code and a message that the provided credentials are already registered. The client can easily handle this message and inform the user.

The ***BasicAuthFitler*** facilitates the authentication of our system. This class implements the *ContainerRequestFilter* interface and is applied globally to all incoming requests that have been matched to a particular resource by JAX-RS runtime. It intercepts requests whose URI include a prefix specified by us and checks for authentication. Any path containing the '*secure*' prefix will be intercepted by the *BasicAuthFilter* and depending on the authentication result it will either block or allow the request. When blocking a request we respond with a JSON object similar to the exception mappers. When the authentication is ok execution will proceed to the specified resource.

6.2.2 Web-Service Resources

The main functionality in our system is provided by the resources package. Each class in this package represents a resource mapped to a specific URL. For each resource there is a method for all HTTP requests (GET, PUT, POST, DELETE). An example of a resource file is given below for the **Users Resource**:

```
@Path("/users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class UsersResource {

	//Retrieves a collection of all the users
	@GET
	public Response getUsers(@Context UriInfo uriInfo){
		//Queries the database and returns a List of users
```

```
        List<Profile> users = UserService.getAllProfiles();
        //for each user create a link for self, places and visits
        for(Profile profile: users){
            profile.getLinks().clear();
            profile.addLink(new Link(uriForSelf(uriInfo,profile),"self"));
            profile.addLink(new Link(uriForPlaces(uriInfo,profile),"places"));
            profile.addLink(new Link(uriForVisits(uriInfo,profile),"visits"));
        }

        GenericEntity<List<Profile>> gProfiles = new
    GenericEntity<List<Profile>>(users){};

        return Response.ok(gProfiles).build();
    }
```

The getUsers() method is mapped to a GET request at the relative path of the containing class. We first retrieve a collection of User profiles from the database and then iterate through the list and add links to the related resources of a user. Finally we create a Response object that contains the List of the retrieved users and respond to the client.

```
    //Creates a new user in the database
    @POST
    public Response postUser(@Context UriInfo uriInfo, Profile profile){
        //Queries the database create a new user
        Profile prof = UserService.addProfile(profile);

        if(profile == null)
            return Response.status(Status.INTERNAL_SERVER_ERROR)
                        .build();

        prof.getLinks().clear();
        prof.addLink(new Link(uriForSelf(uriInfo,profile),"self"));
        prof.addLink(new Link(uriForPlaces(uriInfo,profile),"places"));
        prof.addLink(new Link(uriForVisits(uriInfo,profile),"visits"));

        URI uri = uriInfo.getAbsolutePathBuilder()
                        .path("secure")
                        .path(prof.getUsername())
                        .build();

        return Response.created(uri)
                        .entity(prof)
                        .build();

    }
```

The postUser() method is mapped to the POST requests at the relative path of the class and is used for registering a new user to the system. We first insert the provided profile to the database and then retrieve it, add the related links and respond to the client. The provided credentials are checked in the UserService method to ensure uniqueness.

```
    //Authenticates user and retrieves his data
    @GET
    @Path("/secure/{username}")
    public Response getUser(@Context UriInfo uriInfo,@PathParam("username") String 
username){
          //Queries the database and returns the user
          Profile profile = UserService.getProfile(username);

          profile.getLinks().clear();
          profile.addLink(new Link(uriForSelf(uriInfo,profile),"self"));
          profile.addLink(new Link(uriForPlaces(uriInfo,profile),"places"));
          profile.addLink(new Link(uriForVisits(uriInfo,profile),"visits"));


          URI uri = uriInfo.getAbsolutePathBuilder()
                              .path(String.valueOf(profile.getUsername()))
                              .build();

          return Response.ok(uri)
                         .entity(profile)
                         .build();
    }
```

The getUser() method is mapped to GET requests at the class's path plus a provided username or email. This method is used to retrieve a specific user profile and requires authentication, evident from the 'secure' prefix. Similar to the previous methods this one retrieves a profile from the database, adds the links and responds to the client.

```
    //Authenticates user and updates info
    @PUT
    @Path("/secure/{username}")
    public Response updateUser(@Context UriInfo uriInfo, 
@PathParam("username")String username, Profile profile){
          //Queries the database to update the specified user
          Profile uProfile = UserService.updateProfile(profile, username);

          uProfile.getLinks().clear();
          uProfile.addLink(new Link(uriForSelf(uriInfo, uProfile),"self"));
          uProfile.addLink(new Link(uriForPlaces(uriInfo, uProfile),"places"));
          uProfile.addLink(new Link(uriForVisits(uriInfo, uProfile),"visits"));

          URI uri = uriInfo.getAbsolutePathBuilder()
                              .path(String.valueOf(uProfile.getUsername()))
                              .build();

          return Response.ok(uri)
                         .entity(uProfile)
                         .build();
    }
```

The updateUser() method is mapped to GET requests at the class's path plus a provided username or email. This method is used to update a user profile and requires authentication. Similar to the previous methods this one retrieves a profile from the database, adds the links and responds to the client.

```
    //Deletes the user from the database
```

```
	@DELETE
	@Path("/secure/{username}")
	public Response deleteUser(@Context UriInfo uriInfo,
@PathParam("username")String username){
		//Queries the database to delete the specified user
		UserService.deleteProfile(username);

		return Response.status(Status.NO_CONTENT)
				.build();
	}
```

The deleteUser() method is mapped similar to the previous ones and is used to delete a specific profile from the system. A request to this method requires authentication as well.

```
	@Path("/secure/{username}/visits")
	public VisitsResource getVisits(){
		return new VisitsResource();
	}

	@Path("/secure/{username}/places")
	public PlacesResource getPlaces(){
		return new PlacesResource();
	}

}
```

To clear the above code a little bit we need to get into the JAX-RS annotations. The `@Path` annotation identifies the URI path that a resource class or class method will serve requests for. For an annotated class the base URI is the application path, and for an annotated method the base URI is the effective URI of the containing class. The path specified with the annotation is appended to the Base URI. The `@GET`, `@PUT`, `@POST`, `@DELETE` annotations map a method to the specific HTTP request type. So in our case a GET request for '*http://citywalk.duckdns.or/citywalk/arapp/**users***' will be handled by the **getUsers()** method etc.

The `@Produces`, `@Consumes` annotations specify the media type that the methods can produce and accept. Our implementation only communicates with JSON. JAX-RS automatically maps top level Java classes with the `@XmlRootElement` annotation to JSON using JAXB. Java Architecture for XML Binding (JAXB) is an XML-to-Java binding technology that simplifies the development of web services. For example when a POST method is received and mapped to the *postUser()* method, JAXB automatically parses the provided JSON file that contains the user information, to the Profile Java Bean specified in the model package (Figure 43). The profile object is then provided to the UsersService to create the entry in the database. We then add the links to the profile and create a Response object that contains the newly created account.

The **Visits** and **Places** resources are specified as Sub-Resources of the UsersResource. This means that the root resource partially processes a request and provides another resource to process the remainder. The *getVisits(), getPlaces()* methods are resource locators that provide the object capable of processing the request. The HTTP request types are mapped to specific methods in the subresource's class. So a GET request for '*http://(Base URI)**/users/secure/Chris/visits***' will be handled by a method of the **VisitsResource** mapped to the GET request type.

The same process follows for the remaining resources:

LevelsResource:

The Levels Resource serves all the required information to describe a playable area. The implemented functionality includes checking a client's location for content and serving that content if present. The resource receives a GET request that contains latitude longitude values and queries the database for all available areas to derive if the client is inside a level's boundaries. Get requests can either be used to retrieve collections of levels or specify a level by its unique id or by providing latitude and longitude values. A level's data cannot be modified by the users.

ScenesResource:

The Scenes Resource serves all the scene information from the database. The scenes can be acquired by specifying their unique id or retrieve collections of them for period and/or level. Same as the levels Resource the scene information cannot be modified by the users so the API only supports GET requests. The images for each scene are exposed as their own resource through the Scene's URIs. The database holds a path relative to the server's deployed resources folder. When a Get request is made for an image the **SceneImageResource** will query the database to acquire the relative path and check if there are any images present. If there are we respond with a JSON array that holds all the URIs for the images. If a specific image is requested we create a Response object with the requested image in jpeg format.

PeriodsResource:

The Periods Resource serves all the scene information from the database. The periods can be acquired by specifying their unique id or retrieve collections of them for a given level. Same as the levels Resource a period's information cannot be modified by the users so the API only supports GET requests. The images for each Period are exposed as their own resource through the Period's URIs and the implementations follows that of the Scene Images.

6.3 Android Application

In this section we provide the full implementation of the Android Application. We will start by laying out the basics of Android development to clear out some terms that will be used frequently in this section. Then we will describe the most vital parts of the application. The application was developed in the Android Studio with Java and targets any phone or tablet that runs Android 4.4 (API level 19) and higher.

6.3.1 Basics

The essential components for building an Android App are **Activities**, **Services**, **Broadcast receivers** and **Content providers**. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. Each component is an entry point through which the system or a user can enter the application.

To better describe our implementation we will focus on the **Activities** and **Services**. An activity is the entry point for interacting with the user. It represents a single screen with a user interface. Each activity is independent and can be initiated by other apps if allowed. A collection of activities forms the cohesive user experience of our application. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle. In order for an Activity to start another, it issues Intent to the system specifying the target activity's class and passing some arguments if needed. The activities are arranged in a stack in the order in which each one was opened. This back stack allows for backwards navigation amongst the screens. The user interface for an Android app is built using a hierarchy of layouts (ViewGroup objects) and widgets (View objects). Layouts are invisible containers that control how its child views are positioned on the screen. Widgets are UI components such as buttons and text boxes. Android provides an XML vocabulary for ViewGroup and View classes, so most of our UI is defined in XML files. Each activity defines its layout in its own file.

A **Service** is a component that runs in the background to perform long running operations or to perform work for remote process. Another component, such as an activity, can start the service and let it run or Bind to it in order to interact with it. The main purpose of the Service is to perform these operations without blocking the user interaction with an activity.

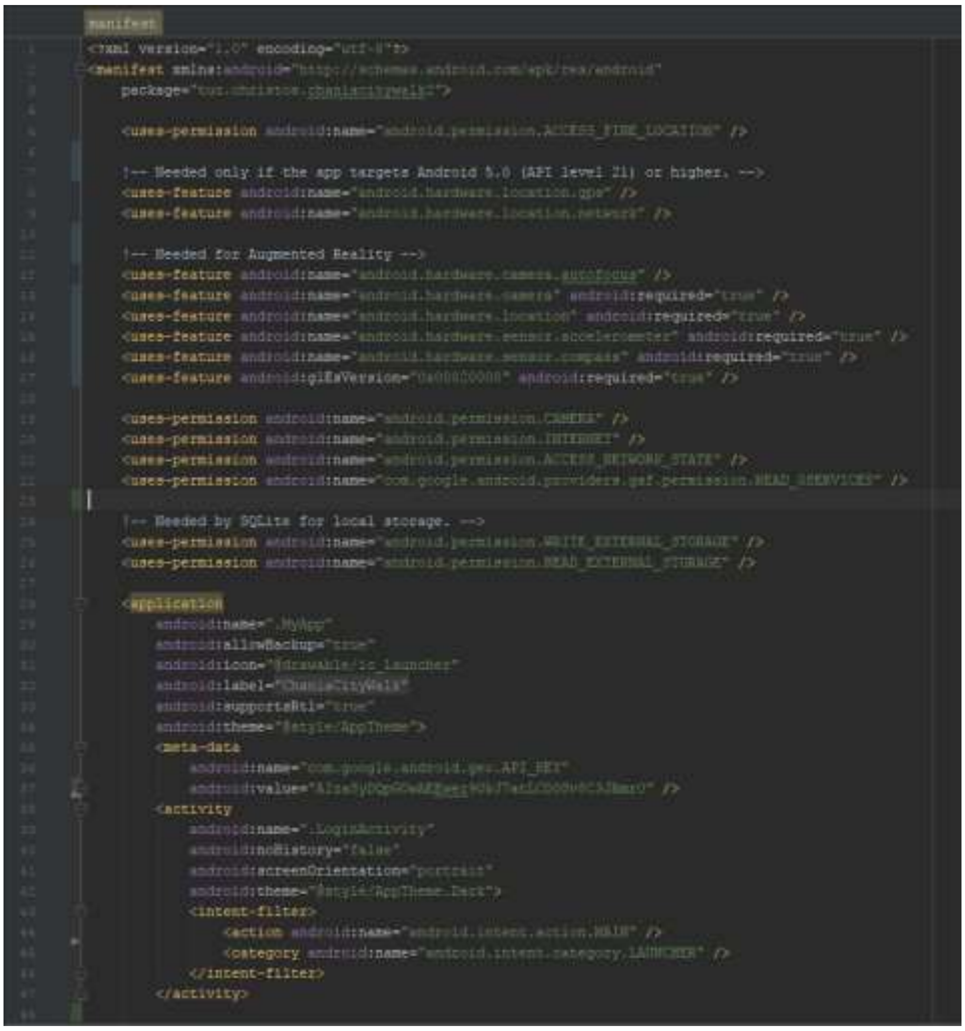
The **Activities** of our application are the:

- **LoginActivity**
- **MapsActivity**
- **ARActivity**
- **ProfileActivity**
- **CollectionsActivity**
- **SceneDetailsActivity**

- **LeaderboardsActivity**
- **NotificationActivity**
- **SettingsActivity**

And the implemented services are the **SensorService** and the **LocationService.**

Before the Android system can start an app component, the system must know that the component exists by reading the app's manifest file. In this file we declare all the components of our application, we declare the hardware features and permissions needed by the Augmented Reality Activities and we declare other APIs that our app needs to be linked with. A part of our app’s manifest file can be seen below. The intent Filter identifies the activity we wish to be launched when the user launches the application.



```
manifest
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="[illegible]">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Needed only if the app targets Android 5.0 (API level 21) or higher. -->
    <uses-feature android:name="android.hardware.location.gps" />
    <uses-feature android:name="android.hardware.location.network" />

    <!-- Needed for Augmented Reality -->
    <uses-feature android:name="android.hardware.camera.autofocus" />
    <uses-feature android:name="android.hardware.camera" android:required="true" />
    <uses-feature android:name="android.hardware.location" android:required="true" />
    <uses-feature android:name="android.hardware.sensor.accelerometer" android:required="true" />
    <uses-feature android:name="android.hardware.sensor.compass" android:required="true" />
    <uses-feature android:glEsVersion="0x00020000" android:required="true" />

    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />

    <!-- Needed by SQLite for local storage. -->
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

    <application
        android:name=".MyApp"
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="ChaniaCityWalk"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="[illegible]" />
        <activity
            android:name=".LoginActivity"
            android:noHistory="false"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.Dark">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

Figure 44. Manifest File

Now that we have laid out the fundamental components of our application we will proceed to the implementation of each one. The following sections will quite often reference the architecture of the application shown at Figure 41, and described in the previous chapter.

6.3.2 Location Service

The location Service is a bound Service that runs throughout the use of the application. It is created at the start and persists even if no activity is in the foreground, depending on the user settings. Upon creation, or when another component needs to interact with the service, it returns a Binder object to provide access to its public methods. The service creates the Location Provider and Location Event handler that provide the location updates and location events accordingly.

The **Location Provider** creates a *GoogleApiClient* object and a Request specifying location interval and accuracy settings and implements the *LocationListener* interface to receive locations acquired by the API. When a new Location is received it will forward that location to the service and the *Event Handler*. It provides the following methods so that the service can control it following its own lifecycle changes: *connect(), disconnect(), Start(), Stop(), StartLocationUpdates(), StopLocationUpdates(), setLocationMode(), registerListener(), removeListener().*

The **Location Event Handler** receives a List of scenes with their locations and checks if the user has entered or left the active area of any of them, which we will call a *GeoFence*. The *GeoFence* is defined as a circle with a radius of 20 meters. To fire the location events we check if the distance between the user and a monument is less than the radius. It receives Location updates from the *LocationProvider* and forwards any location triggers to the Service. The methods it provides are the: *updateSceneList(), requestGeoFences(), setLocationEventListener(), removeLocationEventListener().*

The Service implements two interfaces and registers itself to each of these two components to receive the Location updates and triggers, the *LocationEventListener* and *LoationCallbacksListener* interfaces. The *LocationEventListener* specifies three methods: *userEnteredArea*(), *userLeftArea*(), *drawGeoFences*() and the *LocationCallbacksListener* specifies the *handleNewLocation*() method. So when a new Location is received the *LocationProvider* will call the *handleNewLocation*() methods of the service and the *LocationEventHandler*. The *LocationEventHandler* will then check that location for any triggers while the Service will forward it to the Listening Activities to change their UI. In order for an activity to receive location updates it needs to bind to the service and implement the *ServiceListener* interface. This interface defines all the methods that can be called by the Location Service.

From the time the service is created it will check its lifecycle based on the bound activities, meaning that if an activity unbinds from the Service, because another one has started or the user exits our application, we check with the Activity manager to see if any activity of our process is still in the Foreground. If not we destroy or keep the service alive based on the User settings. When the application is in the background the Service will issue notifications on the System's Tray when a *GeoFence* has been triggered. These notifications contain an Intent Wrapper with a Pending Intent object that effectively gives permission to the Notification Manager to start our activities when the notification is pressed.

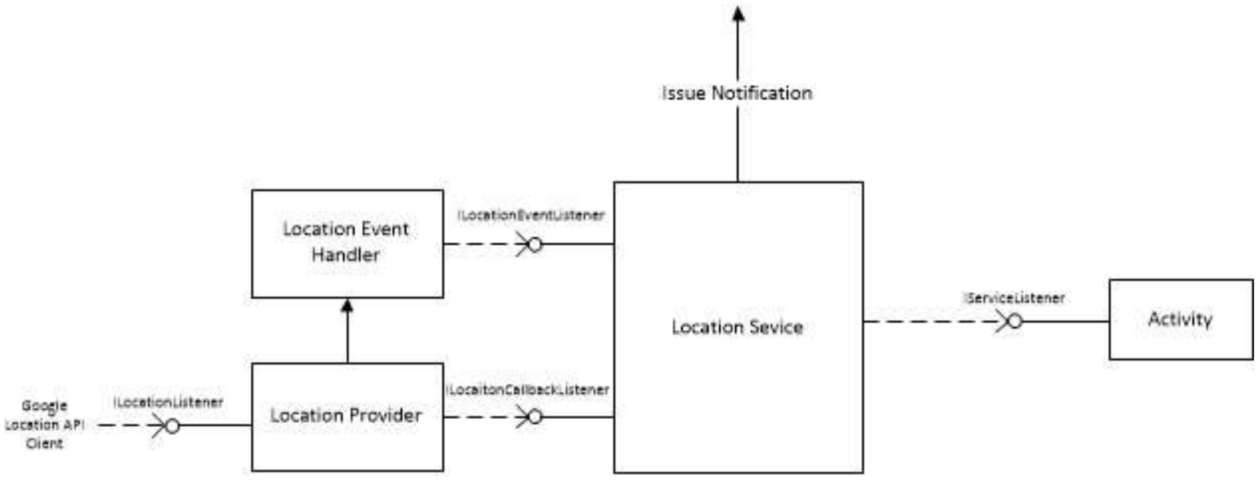


Figure 45. Location Service diagram

The Location Service is also responsible for detecting when the user leaves a playable area. When a Location is outside the current Level’s boundaries or the local db has not been updated in a while, it will employ the *RestClient* to request a new Level. More about the communication with the remote database and the local storage is described in the next section.

6.3.3 Data Handling and Local Storage

As described in the previous section each activity runs isolated from the rest. So we needed a persistent storage unit to keep the downloaded content and provide it to all the activities. This local storage unit is a database developed with **SQLite**. The most important aspect of our local database was to keep it consistent with the remote. As described in the previous chapter the two components responsible for this, are the **DataManager** and the **RestClient** (see figure 41 in the previous chapter). The *RestClient* provides an implementation for each GET, POST, PUT and DELETE requests of our Web API. When a component needs to either get or change information from the remote database it will employ the *RestClient*. The downloaded content is stored in the local db and provided to the activities and services from the *DataManager*.

Following the initialization process the first activity to access the remote database is the Login Activity. If the user logs in to an account or registers to the system the activity will use the *RestClient* to either authenticate and GET a user profile or POST a new user. The workflow then goes to the *LocationService* which requests a playable area given the device’s location. When downloading content from the server the *RestClient* will first employ the *DataManager* to store that content in the local db, and then respond to the calling component. If something goes wrong while connecting to the server or saving to the local db the operation will be aborted. We then notify the user for the error and last request location and timestamp in the local database. The locationService will then make a new request based on these values. In Figure 46 below we provide a diagram to illustrate a successful level request.

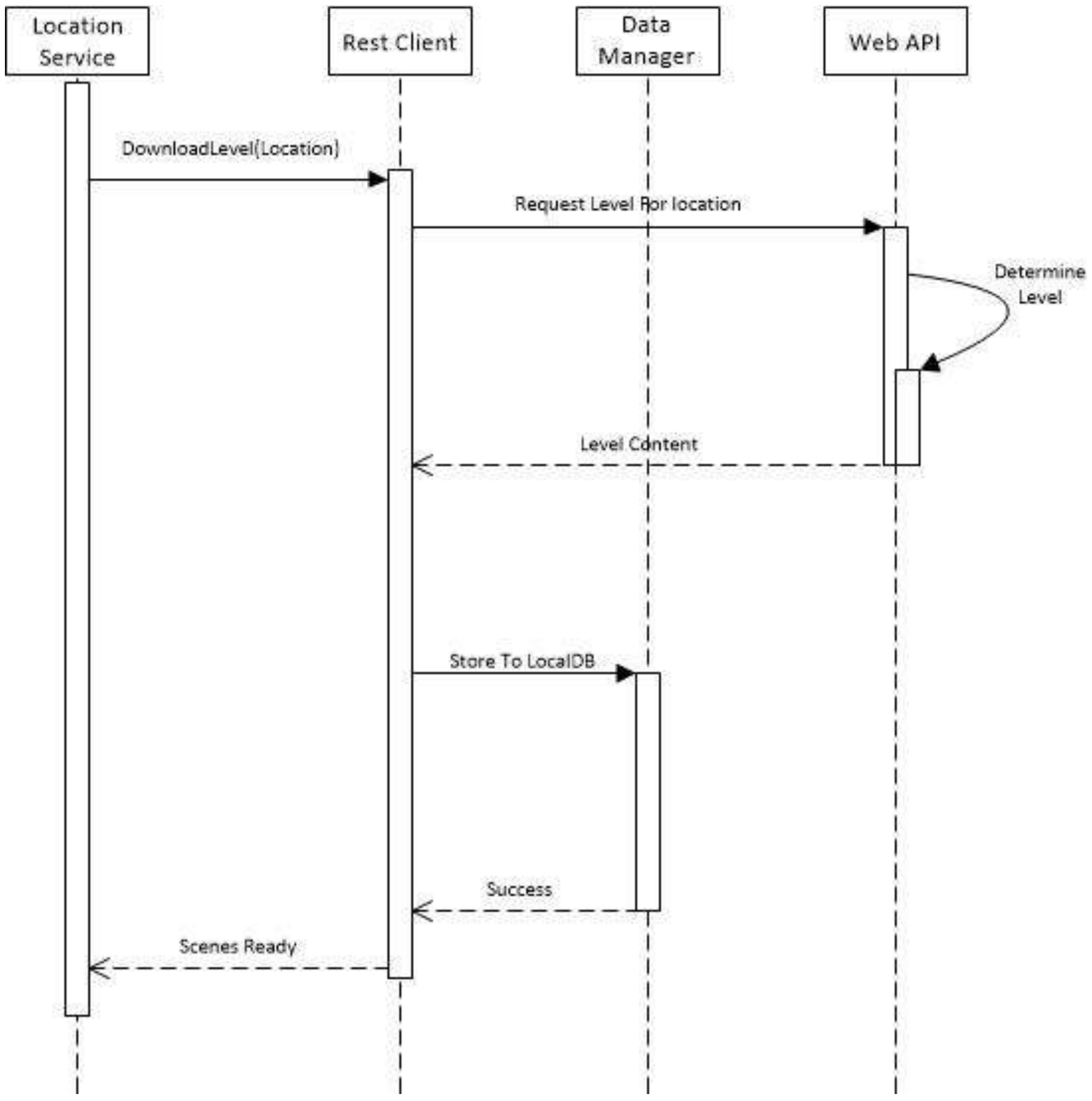


Figure 46. downloading process sequence diagram

In contrast to downloading content, when uploading something to the remote db we follow the reverse order. When a user edits his info, makes a new Visit or saves a new Place the component performing the action will first employ the *DataManager*. The *DataManager* then calls the *RestClient* to update the remote database and depending on the response it will either update the local db or notify the user.

The local database holds a copy of the remote one for the specific *Level* the user is in, its content (*Periods* and *Scenes*), and the logged in user (with *Visits* and *Places*). It also holds some application specific information to ensure the optimal flow of the application, like the last known location and timestamp that we updated the Level Content to ensure it remains in sync with any changes in the server's database. Since our db is initialized all the components access the information only through the *DataManager*. So all components first validate the state of the local content and then make requests to the client component.

The *DataManager* and the *RestClient* each follow the singleton pattern. When an activity is created it will acquire a reference to their instances and call their methods. Since all communication is done with Asynchronous Tasks to not block the UI thread, all communication between the calling activity or component is done with interfaces.

6.3.4 Activities

The most fundamental aspect of an android activity is its lifecycle within Android. Each activity changes states while a user navigates through, out of, and back to our app. The Activity class provides a core set of six callbacks: *onCreate*(), *onStart*(), *onResume*(), *onPause*(), *onStop*(), and *onDestroy*(). Android invokes each of these callbacks as an activity enters a new state. For example when the user starts our application the system will call the *onCreate*() method of the *LoginActivity* and it will enter the **created** state. This is where we initialize our handling components, services, class scope variables and reference the UI objects. The system will then call the *onStart()* and *onResume()* methods in quick succession. The activity then enters the **resumed** state which is when the user can interact with it. The activity will stay in this state until the user navigates to another activity of our application or another app altogether. So our focus lies on handling the system callbacks and the user interactions through the UI.

The key activities in our application are the *MapsActivity* and the *ARActivity*. These two activities are the most dependent on the components described above and incorporate the core functionality of our application.

6.3.5 MapsActivity

The maps activity is where we hold the map that displays all the available scenes of our application as markers on their geo-location. The map component is implemented with the *GoogleMapsAPI*. In the *onCreate*() callback we get a reference to all the UI components, get a reference to the Handler objects described above, initiate the fragment that will hold the map, and start the Background *LocationService*. The *mapsActivity* is registered as a listener to the service so that we can receive the location updates and change the content on the screen. The map component adds an additional Callback that specifies when it is ready to receive content. In this callback we initialize all the listeners that will handle the touch events on the map objects (markers, polylines, infowindows etc.) and draw the map. To draw the map we get a list of all the scenes from the *DataManager* and draw a marker for each based on if it is visited, if it has AR reconstructions and based on the period it belongs to. The marker holds a HashMap with every scene id mapped to its marker to identify the touch events. When the user presses on a marker we display an info window with the monument's name, the distance between the user and the monument and a thumbnail.

After the map is drawn and the activity is in its resumed state the user interacts with it by clicking on the markers and moving around in the City. The service will report location updates by invoking the methods of the *IServiceListener* interface implemented by this activity. The interface specifies the following methods: *drawGeoFences(), handleNewLocation(), UserEnteredArea(), UserLeftArea(), regionChaned().*

The *drawGeoFences*() method specifies all the scenes for which the locationService will send triggers. The map receives an array of scene ids, draws a circle for each one and then maps the two to another HashMap.

The *handleNewLocation*() method is called to update the user's Location. The map receives a Location object which has latitude, longitude, altitude and accuracy values. We then move the user's marker to this location and draw a circle around it to represent the accuracy of the GPS receiver.

The *UserEnteredArea*() method is called to note that the user has entered a GeoFence. The map will then animate the color and alpha of the triggered fence to note the event to the user. If the scene has an AR reconstruction the map will also display a tray holding the buttons to navigate to the AR activity.

The *UserLeftArea*() method is called when a user exits a *GeoFence*. The map will then change its color to the original and hide the buttons tray if it was visible.

The *regionChanged*() callback is invoked when the user leaves the Level he was playing in or enters another. In this method we clear the map of all markers, circles and lines and draw the new Content if available.

The maps activity acts as the main activity of our application. The user can navigate through our app from the bar drawn at the Bottom of the screen.

User Interface

The user interface for an android activity is defined in its own XML file. The XML file defines a view hierarchy that consists of containers and widgets. A container can hold widgets and additional containers. There are a number of available components provided by Android with different specifications. To populate the views we define an id for each object and get a reference at the *onCreate*() callback. We can then manipulate that *View* or *Viewgroup* through our activity's code. For this activity we used a *Framelayout* as the root container. This *ViewGroup* object draws its Childs on top of one another, so the order in which they are declared matters and follows a LIFO fashion (The last View will be drawn on top of the others).

The first view of the container is a fragment object where the map will be drawn. On top of the map we put a *LinearLayout* for the AR buttons tray, which appears depending on location events, and then a *LinearLayout* for the Navigation Bar at the bottom of the screen where the users can navigate through the remaining activities. All clicks and touch events are handled in the activity's code.

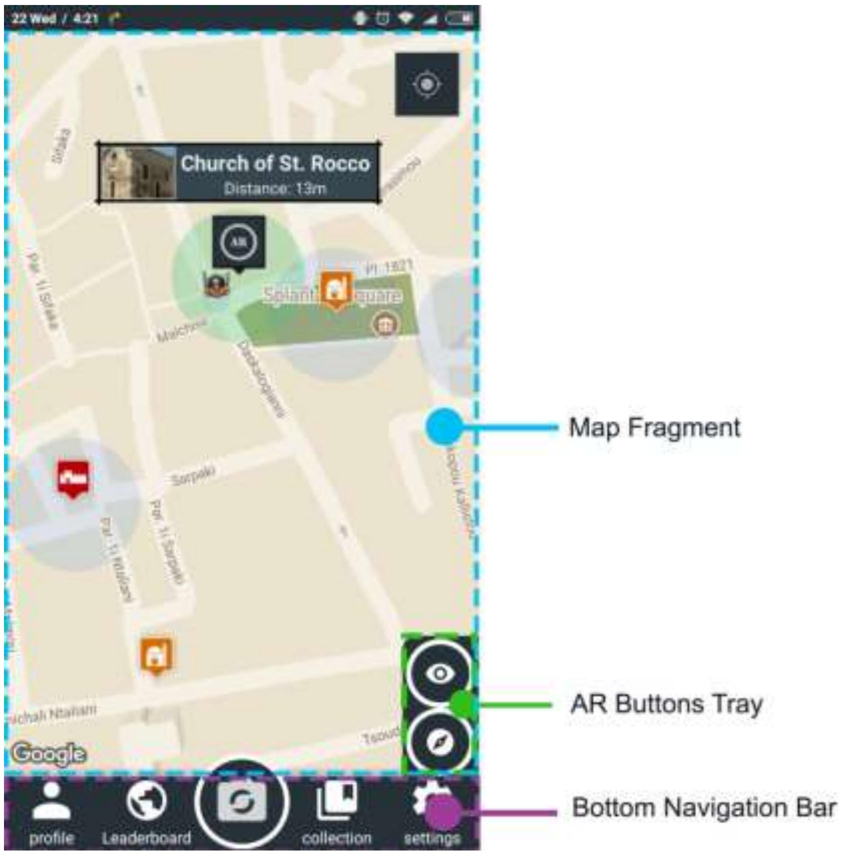


Figure 47. Maps Activity User Interface

The xml vocabulary provided by android allows the creation of custom defined shapes and selectors to define how different states of the buttons will be drawn. The round buttons in the above figure are defined using such shapes to derive form the standard rectangular buttons and improve the aesthetics of our application. The info windows and the AR markers were designed using 9 patch drawables which can stretch depending on the content they show, without changing their initial design. Another important aspect of the Map component is the camera. The map view is modeled as a camera looking down on a flat plane. The position of the camera (and hence the rendering of the map) is specified by the following properties: target (latitude/longitude location), bearing, tilt, and zoom. By adjusting these properties we animate the camera at certain points of the activity.

6.3.6 ARActivity

The Augmented Reality activity was implemented based on the Wikitude JavaScript API. The Wikitude SDK is based on web technologies (HTML, JavaScript, CSS). To integrate the web view with Android the SDK provides us with a specific view component called ***ARchitectView*** which we add to the activity's Layout. We can then load ordinary HTML pages, located in our assets folder, that utilize the API to create objects in Augmented Reality. The first step is integrating the API with the activity's lifecycle. During the *onCreate()* call we need to initialize the *ARchitectView* object and create an interface that communicates with the pages. All information is transferred in *JSON*. Meaning that we need to parse any arguments we wish to send or receive from the AR experiences.

The AR activity is started from the Map activity with an Intent. That intent contains a key-value pair specifying the AR experience (html page) we wish to load. As described in chapter 4 we designed three AR experiences the **ARNavigation**, **3DModelAtGeoLocation** and **InstantTracking**. Each one is a separate html page that we load at the creation of the *ARActivity*. After initialization the activity binds to the *LocationService* to receive Location updates. The acquired locations are sent to the *architectView* to draw the AR objects on the screen. So we have a single native AR activity that runs each AR page and is responsible for initializing each one in a different manner depending on the intent passed on by the Maps Activity. Creating each AR experience follows the standard web development process. The UI includes a 3D scene for the AR objects and standard 2D elements designed with *JQuerry* to provide additional control over the content. We will refer to each experience as a World from this point on.

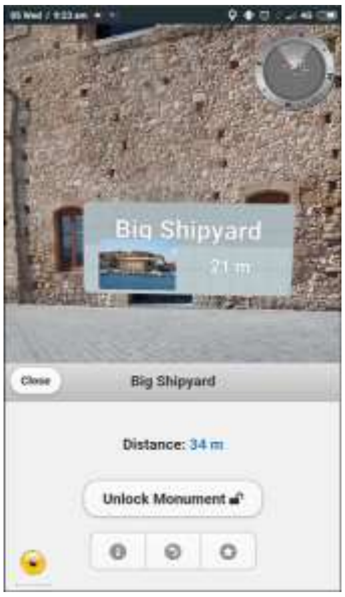
AR Navigation

The AR navigation page displays all scenes in the camera screen at their geo-locations. After the page is loaded we provide a *JSON* array with all the scene information from the native code. The array is then parsed and for each scene we create a marker object to be displayed on the screen. The marker object provides its own logic to animate its changes between selected and deselected states using *AR.PropertyAnimations*. Bellow we provide the method called from the Activity to parse and create a marker for each Scene (POI-Point Of Interest):

```
loadPoisFromJsonData: function loadPoisFromJsonDataFn(poiData) {


   PoiRadar.show();
   PoiRadar.setMaxDistance(800);
   $('#radarContainer').unbind('click');
   $("#radarContainer").click(PoiRadar.clickedRadar);

   World.markerList = [];

   World.markerDrawable_idle = new AR.ImageResource("assets/marker_idle_colored.png");
   World.markerDrawable_selected = new
AR.ImageResource("assets/marker_selected_colored.png");

   World.markerDrawable_idle_q = new
AR.ImageResource("assets/marker_idle_stretch.png");
   World.markerDrawable_selected_q = new
AR.ImageResource("assets/marker_selected_stretch.png");



   if(poiData.length == 0){
       World.updateStatusMessage('No Scenes in your Area');
       return;
```

```
    }
    // loop through POI-information and create an AR.GeoObject (=Marker) per POI
        for (var currentPlaceNr = 0; currentPlaceNr < poiData.length; currentPlaceNr++)
{
            var singlePoi = {
                "id": poiData[currentPlaceNr].id,
                "latitude": parseFloat(poiData[currentPlaceNr].latitude),
                "longitude": parseFloat(poiData[currentPlaceNr].longitude),
                "title": poiData[currentPlaceNr].name,
                "period_id": poiData[currentPlaceNr].period_id,
                "description": poiData[currentPlaceNr].description,
                "visited":poiData[currentPlaceNr].visited,
                "saved":poiData[currentPlaceNr].saved,
                "hasAR":poiData[currentPlaceNr].hasAR,
                "thumbnail":poiData[currentPlaceNr].thumb_uri
            };
            World.markerList.push(new Marker(singlePoi));
        }
    World.initiallyLoadedData = true;
    World.updateStatusMessage(currentPlaceNr + ' places loaded');
}
```

First we create the images that represent a marker with the *AR.ImageResource* and a path to an image at the local assets folder. Then we loop through the Array and create a marker for each scene. A marker is an *AR.GeoObject* with a specific geo-location and a selection of drawables to be drawn on the screen (*TextDrawables*, *ImageDrawables* etc.). More on the *GeoObjects* will be explained at the *ModelOnGeoLocaiton* World. The AR Navigation screen was designed to be very similar to the Maps Activity. It receives the same location events and responds accordingly. Moreover here the users can classify and unlock the unknown areas. When a *GeoFence* is triggered we hide all the other markers and the user can proceed to the classifying process. When initiated the *GeoFence* marker is replaced with a Question object. This object is very similar to the marker object with some additional drawables to represent each period and additional logic to verify the selections (Figure 48). When the user makes a wrong choice the period's button turns to red and an animation deducting his points is shown at the bottom of the screen on top of his icon. If the user makes the correct choice we hide the question object and animate the AR scene to the original view with all the markers.



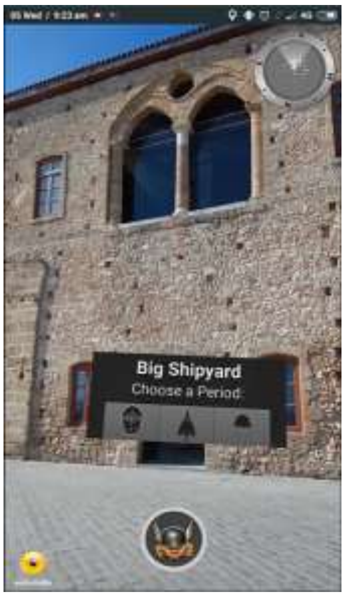


Figure 48. AR Navigation World UI

The bottom panel is a JQuery panel that we slide up and down when a user selects or deselects a marker. Inside the panel are all the Buttons to the other activities and AR experiences as desribed in chapter 4.

ModelAtGeoLocation

The *ModelAtGeoLocation* world is where we present the 3D reconstructions. It can be loaded from either the Map activity or the AR Navigation world. Upon creation we pass an *ARScene* object. At this point we need to make the clarification of the simple scenes and the AR scenes in our system. The *ARScenes* are the ones that include 3D reconstructions. To allow the reconstructions to be available offline through the Guest Login and due to the size of the models, all their assets and 3D models are stored locally on the device. In the assets folder we hold a *JSON* file with the *ARScenes* which we parse on execution and load the content. The additional properties of an AR scene are; a list with the paths to the 3D models, and some Boolean flags. Bellow we present the instantiation of this World:

```
getScene: function getSceneFn(args) {
      World.modelList = [];
      for(var i =0; i < args.models.length; i++){
          var ar = {
            "path": args.models[i].path,
            "latitude": args.models[i].latitude,
            "longitude": args.models[i].longitude
          };
          World.modelList.push(ar);
      }
      var singlePoi = {
       "id": args.id,
       "latitude": parseFloat(args.latitude),
       "longitude": parseFloat(args.longitude),
       "title": args.name,
       "description": args.description,
       "num": args.num
      };
      World.scene = singlePoi;
World.createModelAtLocation();
  },
```

The first function we call from the Native code is the *GetScene*(), passing the *ARScene* object. We then parse the Array to get all the models for the given scene and store it a list. This list contains the path of each model to the assets folder and the geo-location to be drawn. Then we call the *creatModelAtLocation*() function to create the overlays.

```
createModelAtLocation: function createModelAtLocationFn() {
    World.objList = [];

   for(var i = 0; i < World.modelList.length; i++){

           var model = new AR.Model(World.modelList[i].path, {
               scale: {
                   x: 1.0,
                   y: 1.0,
                   z: 1.0
               },
               onLoaded: this.worldLoaded,
               verticalAnchor: AR.CONST.VERTICAL_ANCHOR.BOTTOM
```

```
            });

            var location = new AR.GeoLocation(World.modelList[i].latitude,
World.modelList[i].longitude);

            var indicatorImage = new AR.ImageResource("assets/indi.png");
            var indicatorDrawable = new AR.ImageDrawable(indicatorImage, 0.1, {
                verticalAnchor: AR.CONST.VERTICAL_ANCHOR.TOP
            });

            World.objList.push(new AR.GeoObject(location, {
                drawables: {
                   cam: [model],
                   indicator: [indicatorDrawable]
                },
                enabled:false
            }));
        }

},
```

We then loop through the models list to create an *AR.GeoObject* for each one. To create a *GeoObject* we pass in the location (*AR.GeoLocation*) to be rendered, and the drawables to draw. These include a 3D model (*AR.Model*) and an indicator to point to the model if it is not in the field of view of the camera. To change through the available representations we provide a slider at the bottom of the screen designed with JQuery. The user can hide and show the slider from the button at the bottom right, so it doesn't block the view of the camera.

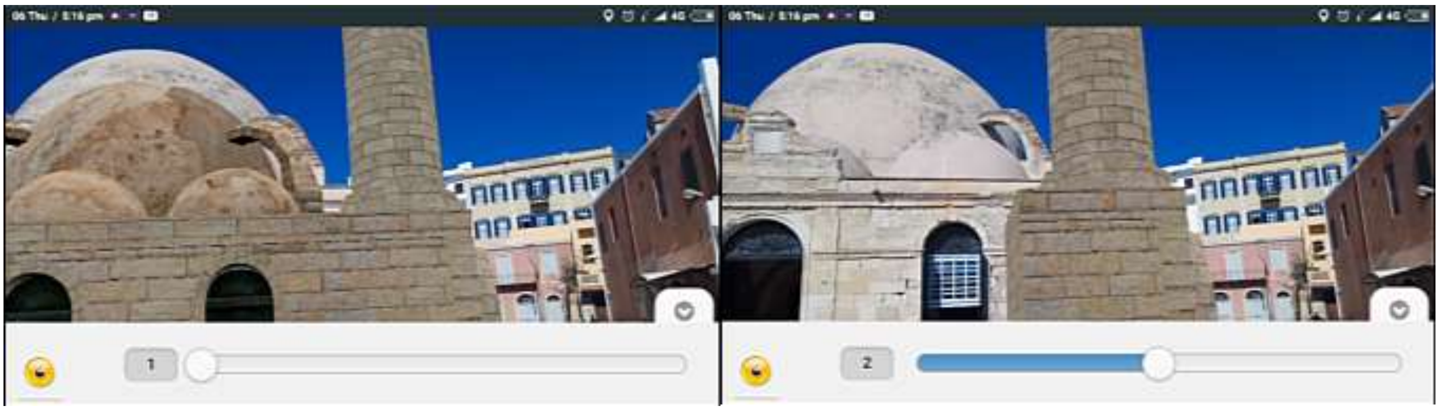


Figure 49. Changing 3D models

InstantTracking

The instantTracking world provides a second tracking method for the 3D reconstructions. This is the only part of our application that employs the *SensorService (Figure 41)*. The activity starts the service and binds to when loading the html file. All communication is done with the *SensorServiceListener* interface similar to the Location Service.

As explained in chapter 2 the instant tracking method is a SLAM implementation that allows for tracking in unknown small areas. It assumes a ground plane based on a height value. To illustrate this plane it overlays a crosshair on the screen that is positioned at the 0,0 point of the plane. Tracking is started and stopped on demand by the user and is dependent on the current area the camera is pointing at. The problem with this method is that there is no way of knowing where the user is pointing the camera, or his position relative to the real buildings. So in order for the 3D models to appear on top of the real buildings we needed a way to estimate where the crosshair is pointing at relative to the world coordinate system. This is where our implementation comes in. The following Figure will help communicate the case.

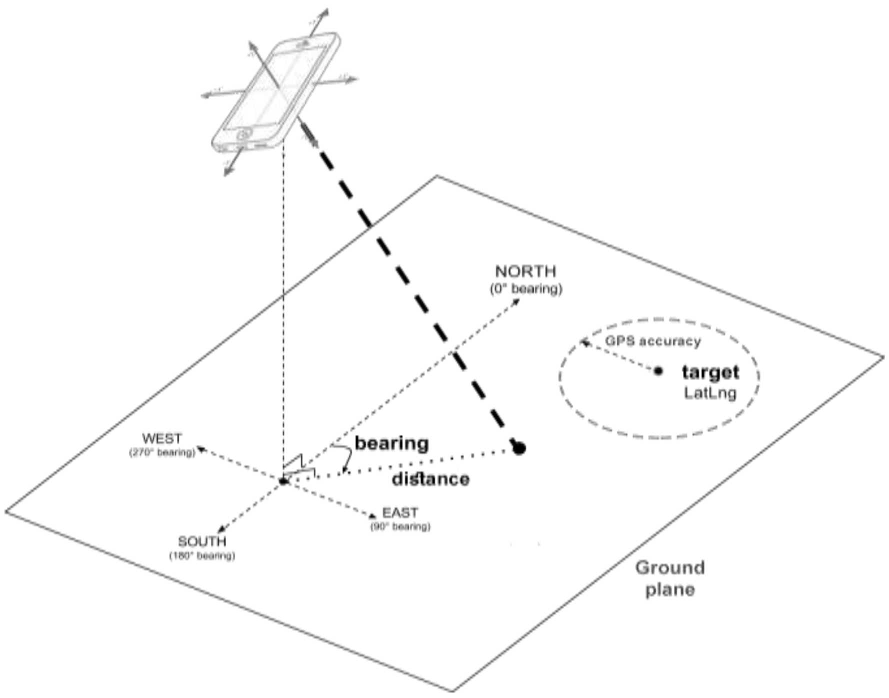


Figure 50. Instant tracking

The inclined dashed line between the device and the ground represents the crosshair's position. The user can move the crosshair on the ground by looking around with the device. The "*target*" circle in the above figure represents an *AR.GeoObject* we draw on the screen on the geo-location of the monument. The point is to place the crosshair inside that circle and at the center of the real building, to start tracking. Our sensor implementation is constantly calculating the bearing of the device and the crosshair's distance (noted in the above figure), and reports them to the AR World. We then compare the reported values with the already known values of the Distance and bearing between the user and the target. Bellow we present the JavaScript code that makes the comparison:

```
calcPointingPosition: function (){
    var degrees = World.location.acc/World.modelDistance * 180 / Math.PI;
    if( World.cHInit && World.modelInit ){

        if( (( World.modelBearing-degrees<= World.cHBearing && World.modelBearing
+degrees >= World.cHBearing ) ||
( World.modelBearing - degrees +360 <= World.cHBearing  && World.modelBearing +
degrees +360 >=  World.cHBearing ))
```

```
            && Math.abs(World.modelDistance - World.cHDistance) <= World.location.acc
){
            World.inPosition = true;
            document.getElementById("tracking-start-stop-button").disabled = false;
            World.controlObjectModel.enabled = false;
            World.controlObjectTriggeredModel.enabled = true;

        } else if(this.tracker.state === AR.InstantTrackerState.INITIALIZING){
                document.getElementById("tracking-start-stop-button").disabled = true;
                World.controlObjectModel.enabled = true;
                World.controlObjectTriggeredModel.enabled = false;
        }
    }
},
```

When these criteria are met we change the circle color to note the event and enable the start-tracking button. When pressed tracking starts and we apply an initial rotation to the model equal to the bearing and show it on the screen. The user can change the crosshair's position to better register the model to its real counterpart. In this way he can compensate for the GPS inaccuracies.

The sensor implementation is based on the **rotation vector** and **gravity** sensors of the device. Below is the part that computes the crosshair's distance based on the gravity sensor:

```
//compute gravity vector magnitude
double gravityMagnitude = Math.sqrt(gravity[0]*gravity[0] + gravity[1]*gravity[1] +
gravity[2]*gravity[2]);
//angle between the gravity vector and the x-z plane in Radians
double alpha = Math.acos(gravity[2]/gravityMagnitude);
//distancce between the users position and the location the camera is Pointing at
double distance = Math.tan(alpha)*deviceHeight;
```

```
                        Gx = gravity[0], Gy = gravity[1] , Gz = gravity[2]
```

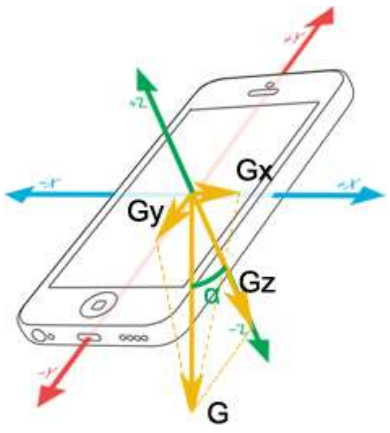


Figure 51. Gravity vector on the axes of the phone

The orientation of the device is given form the following:

```
@Override
public void onSensorChanged(SensorEvent event) {
    switch (event.sensor.getType()) {

        case Sensor.TYPE_ROTATION_VECTOR:
            float[] rotationMatrix = new float[9];
            System.arraycopy(event.values, 0, rotationVector, 0, 3);
```

```
                SensorManager.getRotationMatrixFromVector(rotationMatrix,rotationVector);
                SensorManager.remapCoordinateSystem(rotationMatrix, SensorManager.AXIS_X,
                     SensorManager.AXIS_Z, rotationMatrix);
                SensorManager.getOrientation(rotationMatrix,orientation);
                break;

            case Sensor.TYPE_GRAVITY:
                // copy new data into gravity array and calculate orientation
                System.arraycopy(event.values, 0, gravity, 0, 3);
                break;
        }
        calculatePosition();
}
```

The *onSensorChanged* function is a callback invoked by the *SensorManager*. When starting the service we get a reference to the *SensorManager* system service and register the sensors we wish to receive updates, together with a value for the sampling rate. This value is only a hint to the system and the actual delivery of events may vary. Nonetheless we provided the *SENSOR_DELAY_UI* predefined rate. Since the sampling period is not a part of our calculations the actual interval is of no importance, given that we receive regular updates. In the following Figure we present the instant tracking process.

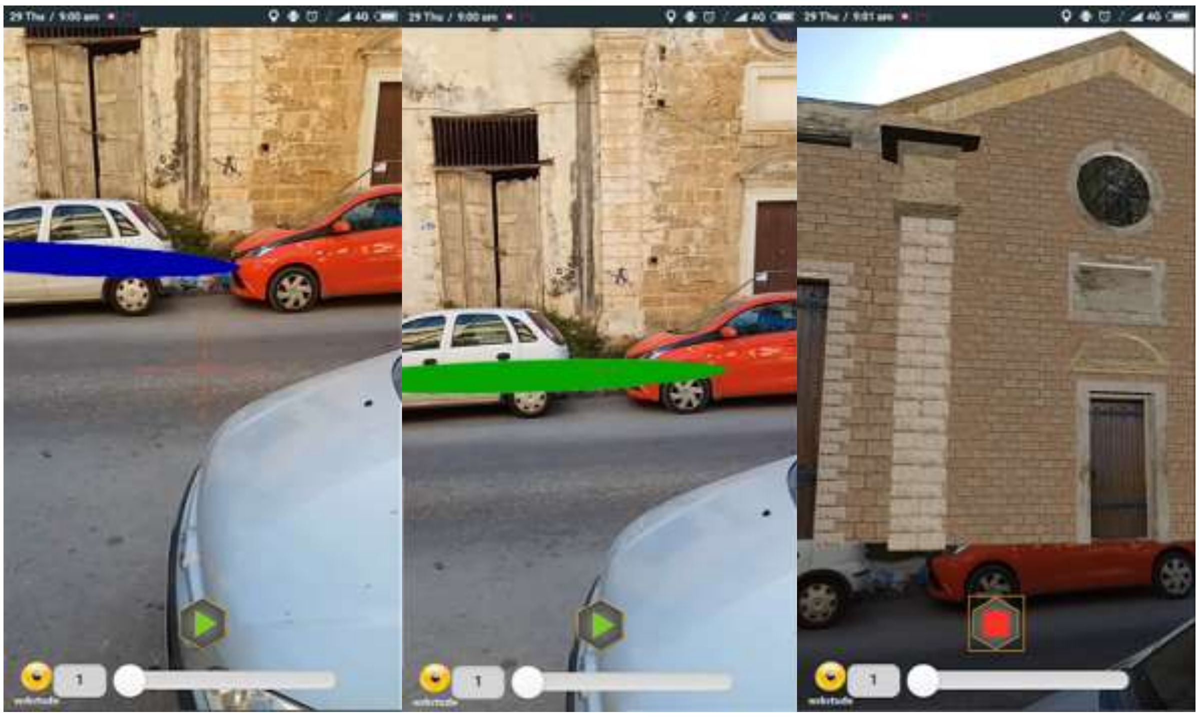


Figure 52. Instant tracking Crosshair (Left), validation (Middle), tracking (Right)

6.4 Evaluation

To evaluate the resulting application we conducted field tests with two different devices. The devices were: a Samsung Galaxy S3 Neo (Android 4.4, RAM 1.4 GB, 1.2 GHz quad-core, Wi-Fi, GPS, Geo-magnetic sensor, accelerometer, gyroscope) and a Xiaomi

Redmi note 4x(Android 6.0, RAM 4GB, 2GHz octa-core, GPS, Wi-Fi, Geo-magnetic sensor, accelerometer, gyroscope). While both devices were expected to perform similarly, there was a big difference concerning the accuracy of their GPS receivers. The S3 Neo presented an average accuracy of 8 meters while the Redmi Note presented the expected 3 meters accuracy. In wider more open areas both devices performed better with the S3 Neo at 3 meters accuracy and the Redmi Note at 1.4. Although the accuracy provided with a Geo-Location represents the confidence level of the receiver and is not a precise representation, the results remained consistent with the acquired values. This indicates that depending on the deployed device the resulting registration of the AR experiences may vary independent of the underlying implementation. As for the orientation tracking the two did not present any differences and both implementations performed as expected and described in Chapter 2.

As expected since both devices are equipped with the same sensors, during the instant tracking testing they performed the same with the S3 Neo accuracy issues carrying over to this implementation as well. A known issue about the instant tracking method is the inconsistency between our calculation of the device’s orientation and the one calculated by the API. This measurement has been observed to vary up to 5 degrees when the Geo-magnetic sensor reports low accuracy values. This error is minimized by re-calibrating the sensor by moving the device in an “eight” figure motion. The user is informed with a Toast when the sensor accuracy is low. Another known issue with this AR experience happens when the users leave the application during tracking, if the phone rings for example. When they return to the AR camera view and reinitiate tracking the models either appear corrupted, and the user needs to navigate back to another activity and reinitiate this one, or the activity will freeze and the user will return to the map.

In order to accurately evaluate the resulting application and if it achieves the intended functionality, we were constantly checking with potential users. The more persistent comment was for the AR camera view during navigation. Most users found that the use of the camera instead of the map for limited their movements and perception of their surroundings and refrained from using it but only to locate specific sites and to classify the monuments. In the classification process the AR camera proved useful as it helped locate the specified monument. Moreover the AR camera together with the constant usage of the GPS for extended time periods resulted in high battery consumption. Following these comments we made the AR camera view a standalone activity instead of a replacement to the map in the main activity.

Other important features implemented by following user comments are the gallery and the save option in the monument details page. The implemented functionality allowed users to save monuments from either the list in the collections screen or from the info window when pressing on the marker at the map screen. The info window functionality was mostly missed and the users found it useful to be able to save a monument to their places when browsing through its details, so it was transferred to the details page while the list functionality remained as is. For the gallery option the initial design contained a link which when pressed the users transferred to a full screen view of the images. The users suggested that a gallery region should be included that shows all the images in the details page and when an image is pressed it will then proceed to the full screen view of the images. The implemented region follows a carousel fashion where the users can swipe right and left to browse through all the images, and when an image is pressed we bring it at full screen.

As per the reception of the Augmented Reality experiences the results were very promising. Most users had never been acquainted with a similar application and were very excited to see the reconstructions. Although the registration problem was commented from most users, the geo-locations approach proved really easy-to-use and intuitive and was primarily used. The instant tracking method proved more challenging for an unaccustomed audience, but after an initial explanation and some guidance the users got used to it and proceeded to experiment with placing the models in the annotated area to better overlay it over the building.

7 Conclusion

7.1 Summary

In this thesis we presented the design of a mobile Augmented Reality application aimed for consumer-grade mobile phones with the ultimate goal of increasing the synergy between visitors and cultural heritage sites. In addition to exploring well known application screens and web content we offer a novel approach for visualizing historical information on-site. By employing 3D reconstructions through AR we aimed to enhance user-experience in heritage environments and bridge the gap between digital content and the real environment. Our design was focused on providing an expandable platform that can easily envelop more sites requiring little preparation and will enable future experts to display their digitized collections using different forms of data presentation.

During this process one of the most valuable lessons we benefit from is the unpredictable problems and obstacles someone can face and how he can manage to overcome them. Augmented Reality is a field that has just reached the wider public. Mobile AR especially, where the processing power and sensor availability is limited, makes the originally envisioned result even harder to reach. The experiences someone wants to create are tightly correlated to the available technologies. Each use case varies greatly leading the developer to review and reestablish the initial requirements. Nonetheless there is much research being done and Augmented Reality has evolved greatly with the addition of more sophisticated algorithms and specialized hardware.

Although outdoors Mobile Augmented Reality presents several challenges on a technological aspect (concerning localization and registration), it already seems more than capable of providing novel experiences to a wide audience. The availability and technological advances of modern smartphones allows for an ideal integration of the technology that can enhance the understanding of historical datasets and the generation of more meaningful experiences.

7.2 Future Work

The field of Augmented Reality is a quickly evolving one where new technologies are rising with high frequency so our application should follow this progress. The techniques used in this application should not be taken for granted as tracking and registration in AR are far from solved. Changing to a low-level Augmented Reality SDK, like the AR-toolkit, would allow us to experiment with Augmented Reality in a more fundamental manner by providing access to its core functionalities. New technologies, like the wide-are tracking system presented in (Ventura and Hollerer 2012), could then be experimented with, to see if they better fit our case. Content delivery is also a very important aspect for creating an engaging experience. Plugins that allow the development of AR applications with game engines like Unity3D and the Unreal engine would allow the generation of more sophisticated interactions and high fidelity graphics, and would greatly improve the user experience.

A Web-Based authoring tool where users and experts could create and publish content would be a great feature for the overall system. Experts could create new AR scenes by uploading media assets and placing them on a map, while users would be able to browse through the geo-located information and even create and modify their own content. By reducing the size of the assets to an acceptable value the AR experiences could then be available to the Mobile Units over the internet.

In its current state the application does not support many interactions between the users, apart from the overall rankings. Extending the platform to include comments, likes, shares etc. and adding an additional communication layer would increase their interest to the heritage sites and would add an extra motivation for the app's use.

Gamification of heritage sites has become of increasing value as it engages visitors and allows for new means of interacting with cultural heritage information. Augmented and Virtual reality, have received even more attention from the field since they present the perfect tool to elevate the interaction with historical datasets from the standard ways of communication. Combined with scavenging and treasure hunts, location-aware storytelling etc. they would add to an even more immersing experience and increase visitor involvement and engagement.

8 References

Azuma R T (1997) A survey of augmented reality. Presence, 6(4):355–385

Liarokapis, F, Brujic-Okretic V, Papakonstantinou S (2006) Exploring Urban Environments using Virtual and Augmented Reality, Journal of Virtual Reality and Broadcasting. GRAPP 2006 Special Issue, Digital Peer Publishing, 3(5): 1-13

Lee J Y, Lee S H, Park H M, Lee S K, Choi J S, Kwon J S (2010) Design and implementation of a wearable AR annotation system using gaze interaction. Consumer Electronics (ICCE), 2010 Digest of Technical Papers, pp.185–186

Arvanitis T N , Petrou A, Knight J F, Savas S, Sotiriou S, Gargalakos M, Gialouri E. (2009) Human factors and qualitative pedagogical evaluation of a mobile augmented reality system for science education used by learners with physical disabilities. Personal and Ubiquitous Computing 13(3):243–250

Zhou F, Been-Lirn H D, Billinghurst M (2008). Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR. In Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality(ISMAR '08). IEEE Computer Society, Washington, DC, USA, 193-202

Azuma R T, Baillot Y, Behringer R, FeinerS, Julier S MacIntyre B (2001). Recent advances in augmented reality. Computer Graphics and Applications, IEEE, 21(6):34–47

Piekarski W, Thomas B (2002) Arquake: the outdoor augmented reality gaming system. Communications of the ACM, 45(1):36–38

Kutter O, Aichert A, Bichlmeier C, Traub J, Heining SM, Ockert B, Euler E, Navab N (2008) Real-time volume rendering for high quality visualization in augmented reality. In International Workshop on Augmented environments for Medical Imaging including Augmented Reality in Computer-aided Surgery (AMI-ARCS 2008), New York,USA

Fielding R T (2000) Architectural styles and the design of network-based software architectures. PhD thesis, University of California

Milgram P, Kishino F (1994) A taxonomy of mixed reality visual displays. IEICE Transactions on Information Systems, E77-D(12):1321-1329

Milgram P, Takemura H, Utsumi A, Kishino F (1994) Augmented reality: A class of displays on the reality-virtuality continuum. Proceedings of Telemanipulator and Telepresence Technologies, 2351(34):282-292

Nagakura T, Sung W (2014) Ramalytique: Augmented Reality in Architectural Exhibitions. Proceedings for Conference on Cultural Heritage and New Technologies (CHNT)

Niedmermair S, Ferschin P (2011) An Augmented Reality Framework for On-Site Visualization of Archaeological Data. In Proceedings of the 16th International Conference on Cultural Heritage and New Technologies, 636-647

Vlahakis, Karigiannis J, Tsotros M, Gounaris M, Almeida L, Stricker D, Gleue T, Christou I T, Carlucci R, Ioannidis N (2001) Archeoguide: First results of an augmented reality, mobile computing system in cultural heritage sites," Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage, VAST '01, (New York, NY, USA), pp. 131-140, ACM, 12, 13

Choudary O, Charvillat V, Grigoras R, Gurdjos P (2009) MARCH: mobile augmented reality for cultural heritage, MM '09: Proceedings of the 17th ACM international conference on Multimedia

Pacheco D, Wierenga S, Omedas P, Wilbricht S, Knoch H, Paul F M J (2014) Spatializing experience: a framework for the geolocalization, visualization and exploration of historical data using VR/AR technologies, Verschure April 2014 VRIC '14: Proceedings of the 2014 Virtual Reality International Conference

Julier S J, Schieck A F, Blume P, Moutinho A, Koutsolampros P, Javornik A, Rovira A, Kostopoulou E (2016) VisAge: augmented reality for heritage June 2016 PerDis '16: Proceedings of the 5th ACM International Symposium on Pervasive Displays

Střelák D, Škola F, Liarokapis F (2016) Examining User Experiences in a Mobile Augmented

Reality Tourist Guide, PETRA '16: Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments

Amin D, Govilkar S (2015) Comparative Study of Augmented reality SDKs". In: International Journal on Computational Sciences & Applications (IJCSA) 5

Ventura J, Hollerer T. (2012) Wide-area scene mapping for mobile visual tracking. In Mixed and Augmented Reality (ISMAR), IEEE International Symposium.