An Engine enabling Location-based Mobile Augmented Reality Applications

Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany
{marc.schickler, ruediger.pryss, johannes.schobel, manfred.reichert}@uni-ulm.de

Abstract. Contemporary smart mobile devices are already capable of running advanced mobile applications with demanding resource requirements. However, utilizing the technical capabilities of such devices constitutes a challenging task (e.g., when querying their sensors at run time). This paper deals with the design and implementation of an advanced mobile application, which enables location-based mobile augmented reality on different mobile operating systems (i.e., iOS and Android). In particular, this kind of application is characterized by high resource demands. For example, at run time various calculations become neccessary in order to correctly position and draw virtual objects on the screen of the smart mobile device. Hence, we focus on the lessons learned when implementing a robust and efficient, location-based mobile augmented reality engine as well as efficient mobile business applications based on it.

1 Introduction

Daily business routines increasingly require mobile access to information systems, while providing a desktop-like feeling of mobile applications to the users. However, the design and implementation of mobile applications constitutes a challenging task [1, 2]. Amongst others, developers must cope with limited physical resources of smart mobile devices (e.g., limited battery capacity or limited screen size) as well as non-predictable user behaviour (e.g., mindless instant shutdowns). Moreover, mobile devices provide advanced technical capabilities the mobile applications may use, including motion sensors, a GPS sensor, and a powerful camera system. On the one hand, these capabilities allow for new kinds of business applications. On the other, the design and implementation of such mobile applications is challenging. In particular, integrating sensors and utilizing the data recorded by them constitute a non-trivial task when taking requirements like robustness into and scalability into account as well.

Furthermore, mobile business applications need to be developed for various mobile operating systems (e.g., iOS and Android) in order to allow for their widespread use. Hence, developers of mobile business applications must not only cope with the above mentioned challenges, but also with the heterogeneity of existing mobile operating systems, while at the same time fully utilizing their technical capabilities. In particular, if the same functions shall be provided on

different mobile operating systems, additional challenges emerge due to scalability and robustness demands.

This paper deals with the development of a generic mobile engine, that enables location-based mobile augmented reality for a variety of advanced business applications. We discuss the core challenges emerging in this context and report on the lessons learned when applying the developed engine to implement mobile business applications. Finally, existing approaches deal with location-based mobile augmented reality as well [3–6]. To the best of our knowledge, however, they do not focus on aspects regarding the efficient integration of location-based mobile augmented reality with mobile business applications.

1.1 Problem Statement

The overall purpose of this paper is to give insights into the development of the core of a *location-based mobile augmented reality engine* for the mobile operating systems *iOS 5.1 (or higher)* and *Android 4.0 (or higher)*. We denote this engine as *AREA*[1]. As a particular challenge, the augmented reality engine shall be able to display *points of interest (POIs)* from the surrounding of a user on the video camera screen of his smart mobile device. The development of such an engine constitutes a non-trivial task, raising the following challenges:

- In order to enrich the image captured by the camera of the smart mobile device with virtual information about POIs in the surrounding, basic concepts enabling location-based calculations need to be developed.
- An efficient and reliable technique for calculating the distance between two positions is required (e.g., based on data of the GPS sensor in the context of location-based outdoor scenarios).
- Various sensors of the smart mobile device must be queried correctly in order to determine the attitude and position of the smart mobile device.
- The angle of view of the smart mobile device's camera lens must be calculated to display the virtual objects on the respective position of the camera view.

Furthermore, a location-based mobile augmented reality engine should be made available on all established mobile operating systems. Realizing the required robustness and ease-of-use for heterogenous mobile operating systems, however, constitutes a non-trivial task.

1.2 Contribution

In the context of AREA, we developed various concepts for coping with the limited resources of a smart mobile device, while realizing advanced features with respect to mobile augmented reality at the same time. In this paper, we present a

[1] AREA stands for *A*ugmented *R*eality *E*ngine *A*pplication. A video demonstrating AREA can be viewed at: http://vimeo.com/channels/434999/63655894. Further information can be found at: http://www.area-project.info

sophisticated application architecture, which allows integrating augmented reality with a wide range of applications. However, this architecture must not neglect the characteristics of the respective mobile operating system. While for many scenarios, the differences between mobile operating systems are rather uncritical in respect to mobile application development, for the mobile application considered in this paper this does not apply. Note that there already exist augmented reality frameworks and applications for mobile operating systems like Android or iOS. These include proprietary and commercial engines[1] as well as open source frameworks and applications [7]. To the best of our knowledge, however, these proposals neither provide insights into the functionality of such an engine nor its customization to a specific purpose. Furthermore, insights regarding the development of engines running on more than one mobile operating systems are usually not provided. To remedy this drawback, we report on the lessons learned when developing AREA and integrating mobile business applications with it.

The remainder of this paper is organized as follows: Section 2 introduces the core concepts and architecture of AREA. In Section 3, we discuss the lessons learned when implementating AREA on the mobile operating systems iOS and Android. In particular, this section discusses differences we experienced in this context. Section 4 gives detailed insights into the use of AREA for implementing real-world business applications. In Section 5 related work is discussed. Section 6 concludes the paper with a summary and outlook.

2 AREA Approach

The basic concept realized in AREA is the *locationView*. The points of interest inside the camera's field of view are displayed on it, having a size of $\sqrt{width^2 + height^2}$ pixels. The *locationView* is placed centrally on the screen of the mobile device.

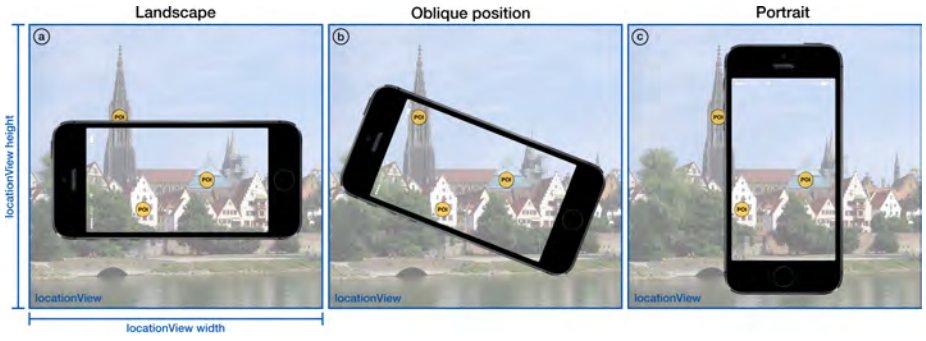

Fig. 1. locationView examples depicting its characteristics.

[1] Wikitude (http://www.wikitude.com)

2.1 The locationView

Choosing the particular approach provided by AREA's *locationView* has specific reasons, which will be discussed in the following.

First, AREA shall display *points of interest* (POIs) correctly, even if the device is hold obliquely. Depending on the device's attitude, the POIs have to be rotated with a certain angle and moved relatively to the rotation. Instead of rotating and moving every POI separately, however, it is possible to only rotate the *locationView* to the desired angle, whereas the POIs it contains are rotated automatically; i.e., the resources needed for complex calculations can be significantly reduced.

Second, a complex recalculation of the camera's field of view is not required if the device is in an oblique position. The vertical and horizontal dimensions of the field of view are scaled proportionally to the diagonal of the screen, such that a new maximum field of view results with $\sqrt{width^2 + height^2}$ pixels. Since the *locationView* is placed centrally on the screen, the camera's actual field of view is not distorted. Further, it can be customized by rotating it contrary to the rotation of the device. The calculated maximal field of view is needed to efficiently draw POIs visible in portrait mode, landscape mode, or any oblique position inbetween.

Fig. 1 presents an example illustrating the concept of the *locationView*. Each sub-figure represents one *locationView*. As can be seen, a *locationView* is larger than the display of the respective mobile device. Therefore, the camera's field of view must be increased by a certain factor such that all POIs, which are either visible in portrait mode (cf. Fig. 1c), landscape mode (cf. Fig. 1a), or any rotation inbetween (cf. Fig. 1b), are drawn on the *locationView*. For example, Fig. 1a shows a POI (on the top) drawn on the *locationView*, but not yet visible on the screen of the device in landscape mode. Note that this POI is not visible for the user until he rotates his device to the position depicted in Fig. 1b. Furthermore, when rotating the device from the position depicted in Fig. 1b to portrait mode (cf. Fig. 1c), the POI on the left disappears again from the field of view, but still remains on the *locationView*.

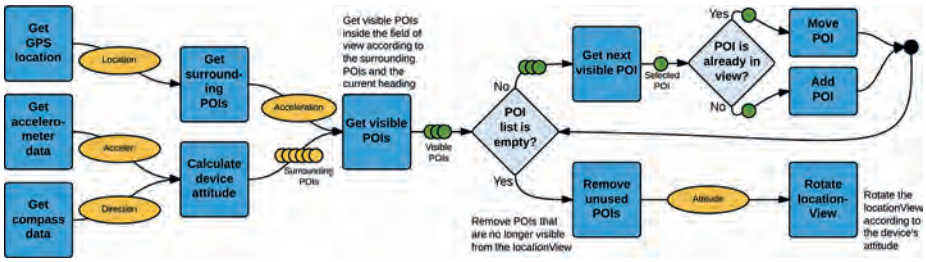

Fig. 2. Algorithm realizing the locationView.

The *third* reason for using the presented *locationView* concept concerns performance. When the display shall be redrawn, the POIs already drawn on the *locationView* can be easily queried and reused. Instead of first clearing the entire screen and afterwards re-initializing and redrawing already visible POIs, POIs that shall remain visible need not to be redrawn. Finally, POIs located outside the field of view after a rotation are deleted from it, whereas POIs that emerge inside the field of view are initialized.

Fig. 2 sketches the basic algorithm used for realizing this *locationView*[2].

2.2 Architecture

The AREA architecture has been designed with the goal to easily exchange and extend its components. The design comprises four main modules organized in a multi-tier architecture and complying with the *Model View Controller* pattern (cf. Fig. 3). Lower tiers offer their services and functions through interfaces to upper tiers. In particular, the tier ② (cf. Fig. 3) will be described in detail in Sect. 3 when discussing the differences regarding the development of AREA on iOS and Android respectively. Based on this architectural design, modularity can be ensured; i.e., both data management and other elements (e.g., POIs) can be customized and easily extended on demand. Furthermore, the compact design of AREA enables us to build new mobile business applications based on it as well as to easily integrate it with existing applications.

The tier ③, the *Model*, provides modules and functions to exchange POIs. In this context, we use both an *XML*- and a *JSON*-based interface to collect and parse POIs. In turn, these POIs are stored in a global database. Note that we do not rely on the *ARML* schema [9], but use a proprietary XML schema instead. In particular, we will be able to extend our *XML*-based format in the context of future research on AREA. Finally, the JSON interface uses a light-weight, easy to understand and extendable format developers are familiar with.

The next tier ②, the *Controller*, consists of two main modules. The *Sensor Controller* is responsible for culling the sensors needed to determine the device's location and orientation. The sensors to be culled include the *GPS sensor*, *accelerometer*, and *compass sensor*. The GPS sensor is used to determine the position of the device. Since we currently focus on location-based outdoor scenarios, GPS coordinates are predominantly used. In future work, we will consider indoor scenarios as well. Note that AREA's architecture has been designed to easily change the way coordinates will be obtained. Using the GPS coordinates and its corresponding altitude, we can calculate the distance between mobile device and POI, the horizontal bearing, and the vertical bearing. The latter is used to display a POI higher or lower on the screen, depending on its own altitude. In turn, the accelerometer provides data for determining the current rotation of the device, i.e., the orientation of the device (landscape, portrait, or any orientation inbetween) (cf. Fig. 1). Since the accelerometer is used to determine the vertical viewing direction, we need the compass data of the mobile device

[2] More technical details can be found in a technical report [8]

to determine the horizontal viewing direction of the user as well. Based on the vertical and horizontal viewing directions, we are able to calculate the direction of the field of view as well as its boundaries according to the camera angle of view of the device. The *Point of Interest Controller* (cf. Fig. 3) uses data of the *Sensor Controller* in order to determine whether a POI lies inside the vertical and horizontal field of view. Furthermore, for each POI it calculates its position on the screen taking the current field of view and the camera angle of view into account.

The tier ①, the *View*, consists of various user interface elements, e.g., the *locationView*, the *Camera View*, and the specific view of a POI (i.e., the *Point of Interest View*). Thereby, the *Camera View* displays the data captured by the device's camera. Right on top of the *Camera View*, the *locationView* is placed. It displays POIs located inside the current field of view at their specific positions as calculated by the *Point of Interest Controller*. To rotate the *locationView*, the interface of the *Sensor Controller* is used. The latter allows determining the orientation of the device. Furthermore, a radar can be used to indicate the direction in which invisible POIs are located (Fig. 5 shows an example of the radar). Finally, AREA uses libraries of the mobile development frameworks, which provide access to core functionality of the underlying operating system, e.g., sensors and screen drawing functions (cf. *Native Frameworks* in Fig. 3).

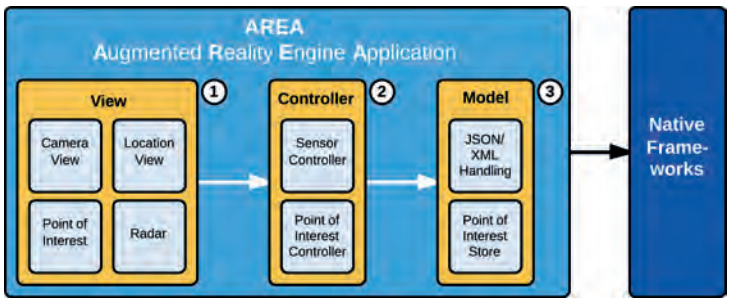

Fig. 3. Multi-tier architecture of AREA.

3 Implementing AREA on Existing Mobile Operating Systems

The kind of business application we consider utilizes the various sensors of smart mobile devices and hence provides new kinds of features compared to traditional business applications. However, this significantly increases complexity for application developers. In turn, this complexity further increases in case the mobile application shall be provided for various mobile operating systems as well.

Picking up the scenario of mobile augmented reality, this section gives insights into ways for efficiently handling the POIs relevant for realizing the *locationView*

of our mobile augmented reality engine. In this context, the implementation of the *Sensor Controller* and the *Point of Interest Controller* are most interesting when studying the subtle differences one must consider in the context of the development of such an engine on different mobile operating systems.

In order to reach a high efficiency when displaying or redrawing POIs on the screen, we choose a native implementation of AREA on the iOS and Android mobile operating systems. Thus, we can make use of built-in APIs of these operating systems, and can call native functions without any translation as required in frameworks like *Phonegap*[2]. Note that efficiency is crucial for mobile business applications since smart mobile devices rely on battery power [10]. To avoid high battery usage by expensive framework translations, therefore, only a native implementation is appropriate in our context. Apart from this, most cross-platform development frameworks do not provide a proper set of functions to work with sensors [11]. In the following, we present the implementation of AREA on both the iOS and the Android mobile operating systems.

3.1 Implementing AREA on iOS

The iOS version of AREA has been implemented using the programming language Objective-C and iOS Version 7.0 on Apple iPhone 4S. Furthermore, for developing AREA, the Xcode environment (Version 5) has been used.

Sensor Controller The *Sensor Controller* is responsible for culling the needed sensors in order to correctly position the POIs on the screen of the smart mobile device. To achieve this, iOS provides the *CoreMotion* and *CoreLocation* frameworks. We use the latter framework to get notified about changes of the location as well as compass heading. Since we want to be informed about every change of the compass heading, we adjusted the heading filter of the *CoreLocation* framework accordingly. When the framework sends us new heading data, its data structure contains a real heading as well as a magnetic one as floats. The real heading complies to the geographic north pole, whereas the magnetic heading refers to the magnetic north pole. Since our coordinates correspond to GPS coordinates, we use the real heading data structure. Note that the values of the heading will become (very) inaccurate and oscillate when the device is moved. To cope with this, we apply a *lowpass* filter to the heading in order to obtain smooth and accurate values, which can then be used to position the POIs on the screen [12]. Similar to the heading, we can adjust how often we want to be informed about location changes. On one hand, we want to get notified about all relevant location changes; on the other, every change requires a recalculation of the surrounding POIs. Thus, we deciced to get notified only if a difference of at least 10 meters occurs between the old and the new location. Note that this is generally acceptable for the kind of applications we consider (cf. Section 4.1). Finally, the data structure representing a location contains GPS coordinates of

[2] Phonegap (http://phonegap.com)

the device in degrees north and degrees east as decimal values, the altitude in meters, and a time stamp.

In turn, the *CoreMotion* framework provides interfaces to cull the accelerometer. The latter is used to determine the current rotation of the device as well as the direction it is pointing to (e.g., upwards or downwards). As opposed to location and heading data, accelerometer data is not automatically pushed to the application by the *CoreMotion* framework of iOS. Therefore, we had to define an application loop polling this data every $\frac{1}{90}$ seconds. On one hand, this rate is fast enough to obtain smooth values; on the other, it is low enough to save battery power.

Basically, the data delivered by the accelerometer consists of three values; i.e., the accelerations in x-, y-, and z-direction. In general, gravity is required for calculating the direction a device is pointing to. However, we cannot obtain the gravity directly from the acceleration data, but must additionally apply a *lowpass* filter to the x-, y-, and z-direction values; i.e., the three values are averaged and filtered. In order to obtain the vertical heading as well as the rotation of the device, we then apply the following steps: First, by calculating $\arcsin(z)$ we obtain a value between $\pm 90°$ describing the vertical heading. Second, by calculating $\arctan 2(-y, x)$, we obtain a value between $0°$ and $359°$, describing the device's degree of the rotation.

Since we need to consider all possible orientations of the smart mobile device, we must adjust the compass data accordingly. For example, assume that we hold the device in portrait mode in front of us towards North. Then, the compass data we obtain indicate that we are viewing in Northern direction. As soon as we rotate the device, however, compass data will change although our view still goes to Northern direction. Reason for this is that the reference point of the compass corresponds to the upper end of the device. To cope with this issue, we must adjust compass data using the rotation calculation presented above. When subtracting the rotation value (i.e., $0°$ and $359°$) from compass data, we obtain the desired compass value, while still viewing in Northern direction after rotating the device.

Point of Interest Controller As soon as the *Sensor Controller* has collected the required data, it notifies the *Point of Interest Controller* at two points in time: (1) when detecting a new location and (2) after gathering new heading and accelerometer data. When a new location is detected, the POIs in the surrounding of the user must be determined. For this purpose, we use an adjustable radius (see Fig. 5 for an example of such an adjustable radius). Using the latter, a user can determine the maximum distance she shall have to the POIs to be displayed. By calculating the distance between the device and the POIs based on their GPS coordinates, we can determine the POIs located inside the chosen radius and hence the POIs to be displayed on the screen. Since only POIs inside the field of view (i.e., POIs actually visible for the user) shall be displayed on the screen, we must further calculate the vertical and horizontal bearing of the POIs

inside the radius. Due to space limitations, we do not describe these calculations in detail, but refer interested readers to a technical report [8].

As explained in [8], the vertical bearing can be calculated based on the altitudes of both the POIs and the smart mobile device (the latter can be determined from the current GPS coordinates). The horizontal bearing, in turn, can be computed with the *Haversine* formula by applying it to the GPS coordinates of the POI and the smart mobile device. In order to avoid costly recalculations of these surrounding POIs in case the GPS coordinates do not change (i.e., movings are within 10m), we buffer POI data inside the controller implementation.

The heading and accelerometer data need to be processed when a notification from the *Sensor Controller* is obtained (Section 3.1.1). Then it needs to be determined which POIs are located inside the vertical and horizontal field of view, and at which positions they shall be displayed on the *locationView*. Recall that the *locationView* extends the actual field of view to a larger, orientation-independent field of view (cf. Fig. 4a). First, the boundaries of the *locationView* need to be determined based on the available sensor data. In this context, the heading data provides the information required to determine the direction the device is pointing. The left boundary of the *locationView* can be calculated by determining the horizontal heading and decreasing it by the half of the maximal angle of view (cf. Fig. 4a). In turn, the right boundary is calculated by adding half of the maximal angle of view to the current heading. Since POIs also have a vertical heading, a vertical field of view must be calculated as well. This can be accomplished analogously to the calculation of the horizontal field of view, except that the data of the vertical heading is required instead. Finally, we obtain a directed, orientation-independent field of view bounded by left, right, top, and bottom values. Then we use the vertical and horizontal bearings of a POI to determine whether it lies inside the *locationView* (i.e., inside the field of view). Since we use the *locationView* concept, we do not have to deal with the rotation of the device, i.e., we can normalize calculations to portrait mode since the rotation itself is handled by the *locationView*.

The camera view can be created and displayed by applying the native *AVFoundation* framework. Using the screen size of the device, which can be determined at run time, the *locationView* can be initialized and placed centrally on top of the camera view. As soon as the *Point of Interest Controller* has finished its calculations (i.e., it has determined the positions of the POIs), it notifies the *View Controller* that organizes the view components. The *View Controller* then receives the POIs and places them on the *locationView*. Recall that in case of a device rotation, only the *locationView* must be rotated. As a consequence, the actual visible field of view changes accordingly. Therefore, the *Point of Interest Controller* sends the rotation of the device calculated by the *Sensor Controller* to the *View Controller*, together with the POIs. Thus, we can adjust the field of view by simply counterrotating the *locationView* using the given angle. The user will only see those POIs on his screen, which are inside the actual field of view; then other POIs will be hidden after the rotation, i.e., they will be moved out of the screen (cf. Fig. 1). Related implementation issues are discussed in [8].

3.2 Android Mobile Operating System

We also developed AREA for the Android mobile operating system. This section gives insights into the respective implementation and compares it with the iOS one. Although AREA's basic architecture is the same for both mobile operating systems, there are subtle differences regarding its implementation.

Sensor Controller For implementing the *Sensor Controller*, the packages *android.location* and *android.hardware* can be used. The *location package* provides functions to retrieve the current GPS coordinate and altitude of the respective device; hence, it is similar to the corresponding iOS package. Additionally, the Android location package allows retrieving an approximate position of the device based on network triangulation. Particularly, if no GPS signal is available, the latter approach can be applied. As a drawback, however, no information about the current altitude of the device can be determined in this case. In turn, the *hardware package* provides functions to get notified about the current magnetic field and accelerometer. The latter corresponds to the one of iOS. It is used to calculate the rotation of the device. However, the heading is calculated in a different way compared to iOS. Instead of obtaining it with the location service, it must be determined manually. Generally, the heading depends on the rotation of the device and the magnetic field. Therefore, we create a rotation matrix using the data of the magnetic field (i.e., a vector with three dimensions) and the rotation based on the accelerometer data. Since the heading data depends on the accelerometer as well as the magnetic field, it is rather inaccurate. More precisely, the calculated heading is strongly oscillating. Hence, we apply a lowpass filter to mitigate this oscillation. Note that this lowpass filter differs from the one used in Section 3.1.1 when calculating the gravity.

As soon as other magnetic devices are located nearby the actual mobile device, the heading is distorted. To notify the user about the presence of such a disturbed magnetic field, which leads to false heading values, we apply functions of the hardware package. Another difference between iOS and Android concerns the way the required data can be obtained. Regarding iOS, location-based data is pushed, whereas sensor data must be polled. As opposed to iOS, on Android all data is pushed by the framework, i.e., application programmers rely on Android internal loops and trust the up-to-dateness of the data provided. Note that such subtle differences between mobile operating systems and their development frameworks should be well understood by the developers of advanced mobile business applications.

Point of Interest Controller Regarding Android, the *Point of Interest Controller* works the same way as the one of iOS. However, when developing AREA we had to deal with one particular issue. The *locationView* manages the visible POIs as described above. Therefore, it must be able to add child views (e.g., every POI generating one child view). As described in Section 3.1, on iOS we simply rotate the *locationView* to actually rotate the POIs and the field of view. In turn, on Android, a layout containing child views cannot be rotated the same

way. Thus, when the *Point of Interest Controller* receives sensor data from the *Sensor Controller*, the x- and y-coordinates of the POIs must be determined in a different way. Instead of placing the POIs independently of the device's current rotation, we utilize the degree of rotation provided by the *Sensor Controller*. Following this, the POIs are rotated around the centre of the *locationView* and they are also rotated about their centres (cf. Fig. 4b). Using this approach, we can still add all POIs to the field of view of the *locationView*. Finally, when rotating the POIs, they will automatically leave the device's actual field of view.

(a) Illustration of the new maximal angle view and the real one.

(b) Rotation of a POI and field of view.

Fig. 4. Usage of *locationView*

3.3 Comparing the iOS and Android Implementaion

This section compares the two implementations of AREA on iOS and Android. First of all, it is noteworthy that both implementations support the same features and functions. Moreover, the user interfaces realized for AREA on iOS and Android, respectively, are the same (see Fig. 5).

Realizing the locationView The developed *locationView* and its specific features differ between the Android and iOS implementations of AREA. Regarding the iOS implementation, we are able to realize the *locationView* concept as described in Section 2.1. On the Android operating system, however, not all features of this concept have worked properly. More precisely, extending the device's current field of view to the bigger size of the *locationView* worked well. Furthermore, determining whether a POI lies inside the field of view, independent of the current rotation of the device, worked well. By contrast, rotating the *locationView* with its POIs to adjust the visible field of view as well as moving invisible POIs out of the screen has not worked on Android as expected. As a

particular challenge we faced in this context, a simple view on Android must not contain any child views. Therefore, on Android we had to use the *layout* concept for realizing the described *locationView*. However, simply rotating a layout does not work on all Android devices. For example, on a Nexus 4 device this worked well by implementing the algorithm in exactly the same way as on iOS. In turn, on a Nexus 5 device this led to failures regarding the redraw process. When rotating the layout on Nexus 5, the *locationView* is clipped by the camera surface view, which is located behind our *locationView*. As a consequence, to ensure that AREA is compatible with a wider set of Android devices, running Android 4.0 (or higher version), we applied the adjustments described in Section 4.2.

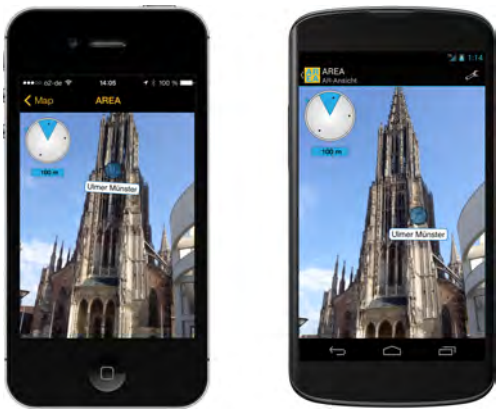

Fig. 5. AREA's user interface for iOS and Android.

Accessing Sensors The use of sensors on the two mobile operating systems differs. This concerns the access to the sensors as well as their preciseness and reliability. Regarding iOS, the location sensor provides both GPS coordinates and compass heading. This data is pushed to the application by an underlying iOS service. In turn, on Android, the location sensor solely provides data of the current location. Furthermore, this data must be polled by the application. Heading data, in turn, is calculated through the fusion of several motion sensors, including the accelerometer and magnetometer.

The accelerometer is used on both platforms to determine the current orientation of the device. However, the preciseness of the provided data differs significantly. Compiling and running AREA on iOS 6 results in very reliable compass data with an interval of one degree. Compiling and running AREA on iOS 7, leads to different results compared to iOS 6. One one hand, iOS 7 allows for a higher resolution of the data intervals provided by the framework due to the use of floating point data instead of integers. On the other, delivered compass data is partially unreliable. Furthermore, in the context of iOS 7 compass data tend to oscillate within a certain interval when moving the device. Therefore, we

need to apply a stronger lowpass filter in order to compensate this oscillating data. Furthermore, on Android the internal magnetometer, which is required for calculating the heading, is vulnerable to noisy sources (e.g., other devices, magnets, or computers). Consequently, delivered data might be unreliable and thus the application must wait until more reliable sensor data becomes available.

For each sensor, the respective documentation should be studied to use it in a proper and efficient manner. In particular, the large number of devices running Android constitutes a challenge with respect to the deployment of AREA on these devices. Finally, Android devices are often affected by distortions of other electronic hardware and, therefore, delivered data might be unreliable.

Altogether, these subtle differences indicate that the development of mobile business applications, which make use of the technical capabilities of modern smart mobile devices, is far from being trivial for application developers.

4 Validation

This section gives insights into the development of business applications based on AREA and the lessons we learned from this. AREA has been integrated with several business applications. For example, the [13] application, which has been realized based on AREA, can be used to provide residents and tourists of a city with the opportunity to explore their surrounding by displaying points of interests (e.g., public buildings, parks, and event locations). When implementing respective business applications based on AREA, one can take benefit from the modular design of AREA as well as its extensibility.

For developing *LiveGuide*, the following two steps were sufficient: first, the appearance of the POIs was adapted to meet UI requirements of the customers. Second, the AREA data model need to be adapted to an existing one. When developing applications like *LiveGuide*, we gained profound practical insights regarding the use of AREA.

Release Updates of Mobile Operating Systems Both the iOS and Android mobile operating systems are frequently updated. In turn, respective updates must be carefully considered when developing and deploying an advanced mobile business application like AREA. Since the latter depends on the availability of accurate sensor data, fundamental changes of the respective native libraries might affect the proper execution of AREA. For example, consider the following scenarios we needed to cope with in the context of an Android operating system update (from Android 4.2 to 4.3). In Android 4.2, the sensor framework notifies AREA when measured data becomes unreliable. By contrast, with Android 4.3, certain constants (e.g., *SENSOR_STATUS_UNRELIABLE*) we had used before were no longer known on the respective devices. To deal with such an issue, the respective constant had to be replaced by a listener (*onAccuracyChanged*).

As another example consider the release of iOS 7, that led to a significant change of the look and feel of the entire user interface. In particular, some of the user interface elements we customized in the deployed version of the *LiveGuide*

applications got hidden from one moment to the other or did not react to an user interaction anymore. Altogether, adjusting mobile applications in the context of operating system updates might cause considerable efforts.

Overlapping POIs In the context of the *LiveGuide* business application we also explored scenarios for which certain POIs located in the same direction overlap with each other, making it difficult for users to precisely touch them. To cope with this issue, we have designed specific concepts for detecting clusters of POIs and offering a way for users to interact with these clusters. Fig. 6 illustrates one of the realized concepts. Again, the modular design of AREA enabled us to implement these extensions efficiently.

(a) Overlapping POIs. (b) Alternative visualization.

Fig. 6. Concept for handling overlapping POIs.

5 Related Work

Previous research related to the development of a location-based augmented reality application, which is based on GPS coordinates and sensors running on *head-mounted* displays, is described in [14, 15]. In turn, [16] applies a smart mobile device, extended with additional sensors, to develop an augmented reality system. Another application using augmented reality is described in [7]. Its purpose is to share media data and other information in a real-world environment and to allow users to interact with this data through augmented reality. However, none of these approaches addresses location-based augmented reality on smart mobile devices as AREA. In particular, no insights into the development of such mobile business applications are provided.

The growing market of smart mobile devices as well as their increasing technical maturity has also motivated software vendors to realize *augmented reality software development kits* (SDKs) [17–19]. In addition to these SDKs, there exist applications like *Yelp*[3] that use additional features of augmented reality to assist users when interacting with their surrounding.

[3] Yelp (http://www.yelp.com)

Only little work can be found dealing with the engineering of augmented reality systems itself. As an exception, [20] validates existing augmented reality browsers. However, neither software vendors nor academic approaches related to augmented reality provide insights into the way a location-based mobile augmented reality engine can be developed.

6 Summary

This paper gives insights into the development of the core framework of an augmented reality engine for smart mobile devices. We further show how business applications can be implemented based on the functions provided by this mobile augmented reality engine. As discussed along selected implementation issues, such an engine development is challenging.

First, a basic knowledge about mathematical calculations is required, e.g., formulas to calculate the distance and heading of points of interest on a sphere in the context of outdoor scenarios. Furthermore, profound knowledge about the various sensors of smart mobile devices is required from application developers.

Second, resource and energy consumption must be addressed. Since smart mobile devices have limited resources and performance capabilities, the points of interest should be displayed in an efficient way and without delay. Hence, the calculations required to handle sensor data and screen drawing must be implemented efficiently. The latter is accomplished through the concept of *locationView*, that allows increasing the field of view by reusing already drawn points of interest. In particular, the increased size allows the AREA engine to easily determine whether a point of interest is inside the *locationView* without need to consider the current rotation of the smart mobile device. In addition, all displayed points of interest can be easily rotated.

Third, we argue that an augmented reality engine like AREA must provide a sufficient degree of modularity to allow for a full and easy integration with existing applications as well as to implement new applications on top of it.

Fourth, we have demonstrated how to integrate AREA in a real-world business application (i.e., *LiveGuide*) and utilize its functions in this context. The respective application has been made available in the Apple App and Android Google Play Stores showing a high robustness. Finally, we have given insights into the differences between Apple's and Google's mobile operating systems when developing AREA.

Currently, AREA can only be applied in outdoor scenarios due to its dependency on GPS. In future research AREA shall be extended to cover indoor scenarios as well. In this context, we will consider Wi-Fi triangulation as well as Bluetooth 4.0 beacons to be able to determine the indoor position of the device.

References

1. Geiger, P., Schickler, M., Pryss, R., Schobel, J., Reichert, M.: Location-based mobile augmented reality applications: Challenges, examples, lessons learned. 10th Int'l Conf on Web Inf Sys and Technologies (WEBIST 2014) (2014) 383–394

2. Pryss, R., Mundbrod, N., Langer, D., Reichert, M.: Supporting medical ward rounds through mobile task and process management. Information Systems and e-Business Management (2014) 1–40
3. Fröhlich, P., Simon, R., Baillie, L., Anegg, H.: Comparing conceptual designs for mobile access to geo-spatial information. Proc 8th Conf on Human-computer Interaction with Mobile Devices and Services (2006) 109–112
4. Carmigniani, J., Furht, B., Anisetti, M., Ceravolo, P., Damiani, E., Ivkovic, M.: Augmented reality technologies, systems and applications. Multimedia Tools and Applications **51** (2011) 341–377
5. Paucher, R., Turk, M.: Location-based augmented reality on mobile phones. IEEE Conf Comp Vision and Pattern Recognition Workshops (2010) 9–16
6. Reitmayr, G., Schmalstieg, D.: Location based applications for mobile augmented reality. Proc 4th Australasian Conf on User Interfaces (2003) 65–73
7. Lee, R., Kitayama, D., Kwon, Y., Sumiya, K.: Interoperable augmented web browsing for exploring virtual media in real space. Proc of the 2nd Int'l Workshop on Location and the Web (2009)
8. Geiger, P., Pryss, R., Schickler, M., Reichert, M.: Engineering an advanced location-based augmented reality engine for smart mobile devices. Technical Report UIB-2013-09, University of Ulm, Germany (2013)
9. ARML: Augmented reality markup language. `http://openarml.org/wikitude4.html` (2014) [Online; accessed 07/05/2014].
10. Corral, L., Sillitti, A., Succi, G.: Mobile multiplatform development: An experiment for performance analysis. Procedia Computer Science **10** (2012) 736 – 743
11. Schobel, J., Schickler, M., Pryss, R., Nienhaus, H., Reichert, M.: Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. Int'l Conf on Web Information Systems and Technologies (2013) 509–518
12. Kamenetsky, M.: Filtered audio demo. `http://www.stanford.edu/~boyd/ee102/conv_demo.pdf` (2014) [Online; accessed 07/05/2014].
13. CMCityMedia: City liveguide. `http://liveguide.de` (2014) [Online; accessed 07/05/2014].
14. Feiner, S., MacIntyre, B., Höllerer, T., Webster, A.: A touring machine: Prototyping 3d mobile augmented reality systems for exploring the urban environment. Personal Technologies **1** (1997) 208–217
15. Kooper, R., MacIntyre, B.: Browsing the real-world wide web: Maintaining awareness of virtual information in an AR information space. Int'l J Human-Comp Interaction **16** (2003) 425–446
16. Kähäri, M., Murphy, D.: Mara: Sensor based augmented reality system for mobile imaging device. 5th IEEE and ACM Int'l Symposium on Mixed and Augmented Reality (2006)
17. Wikitude: Wikitude. `http://www.wikitude.com` (2014) [Online; accessed 07/05/2014].
18. Layar: Layar. `http://www.layar.com/` (2014) [Online; accessed 07/05/2014].
19. Junaio: Junaio. `http://www.junaio.com/` (2014) [Online; accessed 07/05/2014].
20. Grubert, J., Langlotz, T., Grasset, R.: Augmented reality browser survey. Technical report, University of Technology, Graz, Austria (2011)