ulm university universität uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

The Conception and Realization of a Mobile Windows Phone Location-based Augmented Reality Application

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Patrik Miguel Gonçalves
patrik.goncalves@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss

2016

Fassung 15. November 2016

© 2016 Patrik Miguel Gonçalves

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX 2_ε

Abstract

It is considered very evolved having Augmented Reality (AR) to be used in an application. This raises the need to have good AR frameworks and AR engines to facilitate the development. Most of the available engines and frameworks are either hard to understand, due to poor documentation or do not provide a sufficient insight, or are proprietary, which force the developer to pay for it. This thesis introduces a location-based AR engine from scratch, which is in its dynamic structure easy to understand and to integrate it in any custom application. The usage of user controls and the possibility to extend the available classes provide a good basis to individualize the engine. This engine is based on the original AREA for iOS[1] and uses advanced calculations to enhance performance. This engine is made for Windows Phone 8.1 using C# with XAML(Extensible Application Markup Language) to create the UI.

Acknowledgment

I would like to thank my friends and family for the great support. In particular I want to thank my father Antonio, Nicole Koch and my tutor Rüdiger Pryss and his colleagues who made this thesis possible.

Table of Contents

1 Introduction **1**

1.1 Focus of this thesis 1

1.2 Structure of the paper 2

2 AREA **3**

2.1 Location-based Augmented Reality 3

2.2 Matrix transformations 5

3 Similar work **7**

3.1 HERE maps 7

3.2 Current available tools for AR 9

4 Requirements for the engine **11**

4.1 Functional and non-functional requirements 11

4.2 Tabular form of the requirements 13

5 The architecture of AREA **15**

5.1 Structure of AREA 15

5.2 Graphic frameworks 17

5.3 Additional frameworks 18

5.4 Threading model in AREA 18

5.5 Database 19

6 Implementation details **21**

6.1 The used UI components 21

6.2 Sensors 22

6.3 The update process 23

6.4 Radius changes 23

6.5 Updating the UI 25

6.6 Mathematical calculations 26

7 Demonstration of the application 27

8 Comparison of the requirements 31

9 Summary and outlook 35

1 Introduction

Nowadays it is considered very evolved, when an organization or a company have an Augmented Reality Application to represent themselves. Since the first Virtual and Augmented Reality system named *The Sword of Damocles* in 1960's by Ivan Sutherland [2] the possibility to create virtual or manipulate existing worlds have greatly increased. For decades Augmented Reality (AR) systems were just for research purposes and was exclusive for some persons due to the high resource demanding systems. Nowadays many users use a smartphone or any similar device, which mostly contain a camera and a gyroscope that are needed for any basic AR application. This opened the possibility to create relatively cheap AR applications for a high number of costumers, without needing a special device to run it. This possibility emerges a need to create predefined frameworks and engines to facilitate developing AR applications without needing to reinvent the wheel.

1.1 Focus of this thesis

This thesis focuses on the development of a location-based AR engine without relying on existing AR technologies and building it from scrap. This makes it independent on proprietary software and gives a detailed insight on the underlying mechanics and the possibility to modify the applied logic for an individual application. The engine can then be used in an individual application and with its dynamic structure it is still being able to be modified to the developer's needs. This engine is based on the Bachelor thesis made by Philip Geiger[1], which implemented an AR engine for iOS. In this thesis optimized calculations are provided to gain more performance. This engine uses the mounted

camera of the device and projects points of interest(POI) in real-time on the current frame. With this arrangement it gives then the impression of POI being fixed in the real-world, which are made visible with the device. To set an example of this AR engine, we implement it in an Augmented Reality Engine Application, which shows an example application on which this engine could run. So the core features of this engine are made visible.

1.2 Structure of the paper

This paper begins with the introduction on AREA (Augmented Reality Engine Application) and its basic ideas behind it. AREA is based on the work of [3][4][5][6][7][8][9][10][11]. The term *location-based Augmented Reality* and using matrices for calculations are explained in chapter 2. Any similar work is discussed and evaluated in chapter 3. Next, the specified functionality and other criteria for this engine are defined. These are listed as functional and non-functional requirements in chapter 4. In this thesis the AR engine is mainly implemented using the programming language C#. The architecture, frameworks and the used databases for this engine is explained in chapter 5. Furthermore we point out some implementation details in chapter 6 and explain them. The implementation of an example application of this AR engine is in chapter 7 presented with screenshots to visualize the potential of this AR engine. Finally we evaluate the requirements in chapter 8 and conclude in chapter 9.

2

AREA

Before we submerge in the realization of the Augmented Reality (AR) engine, we need to establish a working knowledge of the used terminology and explain the mechanics behind the engine.

2.1 Location-based Augmented Reality

Augmented Reality describes the method of adding virtual information inside a real-world environment. These kind of information usually are visual or auditive nature, which "augment" or enhance this environment. An AR system is not confused with a virtual reality system, which in contrast is set in an artificial environment. The most common practice for AR is to project in real-time textual or graphical information on a display showing the real-world environment. Two modern use-cases for AR systems are shown in figure 2.1 and 2.2. The first example shows a head mounted display in a car to aid the driver with additional information. The second example shows a pilot's view in the cockpit, where among other information the plane's heading, altitude and velocity are displayed. Both examples have the task to provide with additional information about the current state of the vehicle, or more general, of the device. This information is context based, so e.g. the pilot knows the inclination of the plane relative to the horizon.

When we assume a certain information is bound to a specific point in a coordinate system, we call it a point of interest (POI). If POI are set in a global coordinate system,

[1] http://www.hakvoort.de/picserver1/userdata/1/21172/VifXwhikB/head_up_display.jpg
[2] https://ictmagic.files.wordpress.com/2011/04/cockpit-display.jpg

fig. 2.1: A common head up display in a car[1]

fig. 2.2: A view of a cockpit in a plane[2]

only those which are looked at can be seen. An AR system can display the visible POI, while hiding those, which aren't. This setup is what we call location-based Augmented Reality.

2.2 Matrix transformations

The usage of matrices in computer graphics are nowadays a common practice. In mathematics the multiplication of a matrix with a vector results in a modified vector. If we now assume, that a matrix stores information about this modification, we can then define various matrices, which represent different transformations. These transformations could then represent translation, rotation and scaling. If more than one modification is done for a specific vector, we can combine them by multiplying the matrices together. With this knowledge it is possible to store whole set-ups within one matrix, without needing to recalculate it. Multiplying many vectors with a constant matrix can massively improve the overall performance of a system. Knowing this we can then define a rotation matrix, representing the heading and a view-projection matrix, representing the current visible field of view. The transformation of a vector can be interpreted as a modification of the position for a point. Further we assume, that the origin of a coordinate system is the position of the device and all POI represented by a point are relative to this origin. This simplifies further calculations as a multiplication of the POI's position with the rotation matrix results in a rotation around the origin by keeping the same distance to the device.

3

Similar work

First, the app *HERE maps* is introduced and explained. Second, we discuss the current available engines and frameworks for AR systems.

3.1 HERE maps

A good example of a location-based AR application for Windows Phone 8.1 is the phone's standard map app *HERE Maps*. This app provides a map and is able to show POI near the user's current position. If the user taps on the green radar symbol, the app automatically aligns the top part according to the users course and allowing therefore to facilitate reading the map. When the GPS position of the device is precise enough, the user can turn on the *LiveSight*, which is the core feature of this app. This feature implements an AR view displaying the queried POI around the user. It can be turned on by either tapping on the current position (green dot) or in the map options and selecting the eye symbol. *LiveSight* uses the device's current heading and shows the POI accordingly where it would be on the screen. The figure 3.1 describes an example view while the *LiveSight* is turned on. With this a user can determine the direction of a POI without needing a map to look at. When the user heads the device down the map will overlay on the background and showing the POI in the vicinity on a map. This is presented in figure 3.2.

The poi are organized accordingly to the distance to the user, meaning poi near the user are more likely be on top of other poi. In addition the POI further away are displayed vertically on top of the POI less further away. This gives the impression, that these POI

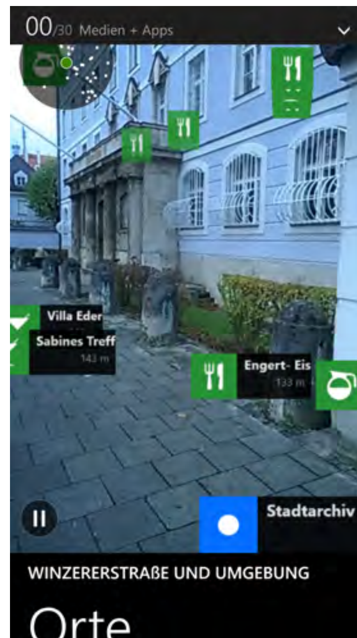

fig. 3.1: The augmented reality view of HERE Maps

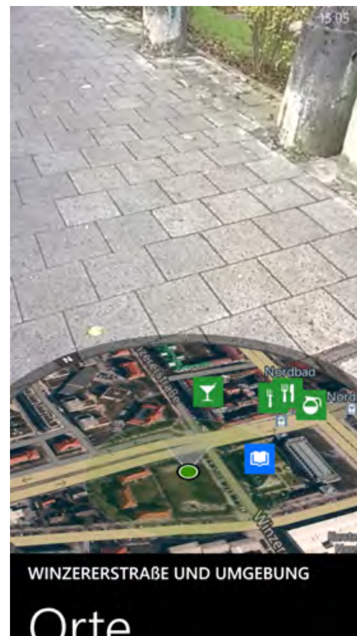

fig. 3.2: The augmented reality and map view together of HERE Maps

fig. 3.3: Details page of HERE Maps with additional information for a POI

are positioned on a higher terrain, which they aren't in reality. Although it makes it more easy to select a certain poi near the user. If a user selects a poi, details for this poi are shown in fullscreen as in figure 3.3 demonstrated.

On the downside it is not possible to filter for searched poi or to change the search radius. It simply just shows a fixed number of POI in the vicinity of the user. This application only works in portrait mode. If the user changes the orientation of the device the displayed POI won't move accordingly to the device's heading.

3.2 Current available tools for AR

As discussed in section 2.2 using matrices it is possible to manipulate points and graphics on a screen. Therefore any graphic library, which, dependent of the device's heading, that projects graphics on a real-world video stream is already considered an AR application. Therefore graphic libraries like OpenGL for C++ or Mono for .NET can be used to create AR systems. Nevertheless they are rarely used, because an AR engine is needed to

be built from scratch. When we talk about AR, the oldest and possible most prominent framework is ARToolKit [12]. It is an open-source library to implement AR in applications on Microsoft Windows, Linux, Mac OS X, Android and iOS. Due to the high popularity of ARToolKit and it being free, there have been various modifications and ports to it to increase the possible use-cases.

4

Requirements for the engine

To achieve the goal of implementing an AR engine we need to set some requirements to ensure the success of this engine. This chapter lists the functional and non-functional requirements and summarizes them into a table.

4.1 Functional and non-functional requirements

This engine should be able to interpret POI and display them on the screen. To do so, it determines the current position of the user, usually by querying the current GPS position and compares it with the POI position. Using mathematical calculations it is then displayed according to devices heading. Only POI, which are in the camera's field of view are to be shown and all others are to be hidden. Furthermore this engine should support landscape, portrait and any opaque mode. The latter is e.g. a view between the landscape and the portrait mode.

To enhance the usability of the application it should be also possible to visualize POI, which are not currently inside the camera's field of view. Therefore a radar is to be implemented and all surrounding POI are displayed on it, relative to the users current heading.

To determine which POI are shown, a user should be able to interact with a slider to adjust a maximum search radius. Hence all POI with a distance less or equal to the user are queried and displayed on the screen. Therefore all POI greater than the maximum radius are to be removed from the screen and ignored in further calculations.

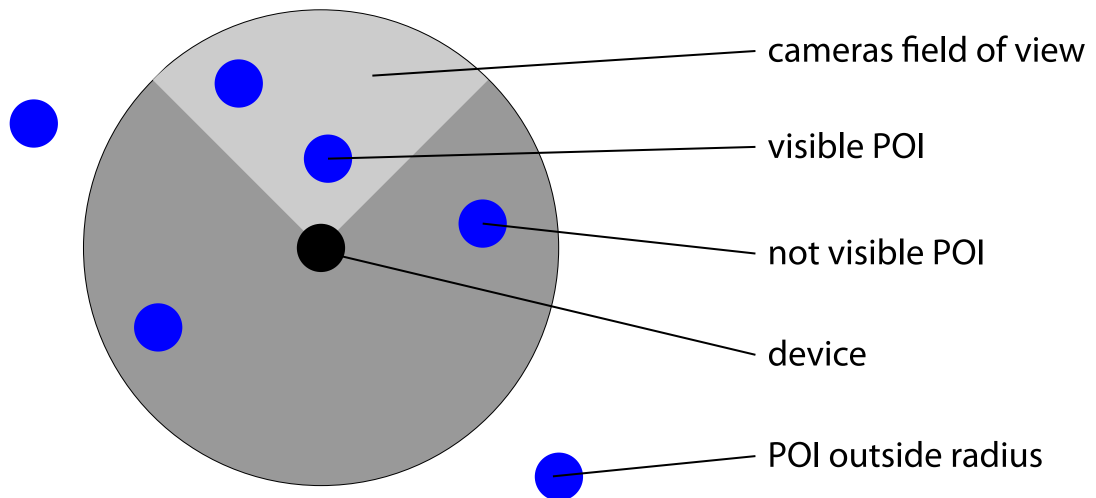

fig. 4.1: A possible set up of POI around the user

A common setup of POI is described in figure 4.1. It describes, which POI are visible, hidden or outside the search radius. It pictures from the aerial view down to the user. The light gray area represents the field of view.

In addition the engine should offer an interface to query for POI from a database. This database can be either a local database or a custom method to get the POI data from a remote server.

The current heading of the device is to be calculated using the available sensors of the device. The sensors could include a magnetometer to determine the view direction according to the geographical north pole, a gyroscope to calculate the devices rotation and a GPS to determine the users position in a global coordinate system. We assume that in most cases small changes of the position will not change the list of POI and therefore a threshold is used. Furthermore we assume, that the usage of this engine will be in stationary mode so GPS updates don't need to be very frequently. This results in less queries needed to obtain new data.

All calculations and changes to the POI should be done in real-time and react if the position or data change.

The POI displayed on the current frame should react to any user interactions and be able to show further information for the selected POI.

Furthermore all the previous mentioned functional requirements need to meet the following non-functional requirements.

All calculations in the background and updates on the user interface (UI) need to be very efficient. This supports a high responsiveness from the system without noticeably having a low frame rate. We assume a maximum number of 100 UI components are handled. Too many POI at the same time on a screen are not very likely as it lowers the overview. These calculations and changes should be very precise and offer a high stability. Additional it needs to have a high maintainability and documentation of the source code. To encourage the usage of this engine it is crucial to offer a extensibility for custom code, like custom POI or the ability to change the look and feel of the UI. In addition to that the architecture should be designed in a way, that an integration of the engine in new or existing applications should be facilitated.

4.2 Tabular form of the requirements

The following table 4.1 summarizes all requirements set for this engine. It consists of listed requirements and whether they're functional or non-functional as a type.

requirements	type
Show POI on screen	functional
Show POI correctly in landscape, portrait and opaque mode	functional
Hide POI not currently on the cameras field of view	functional
POI need to react to interactions	functional
Sensor data need to be acquired to determine devices position and heading	functional
POI data need to be acquired from a database	functional
POI calculations need to be updated in real-time	functional
Maximal search radius needs to be adjusted	functional
Show POI on the radar	functional
Additional information is displayed when POI is selected	functional
Efficient calculations	non-functional
Efficient updates on the UI	non-functional
High stability	non-functional
High precision	non-functional
High maintainability	non-functional
Extensibility for custom code	non-functional
Ability to embed engine in other applications	non-functional
Good documentation of code	non-functional

tab. 4.1: Table of functional and non-functional requirements

5 The architecture of AREA

In the last chapter we laid out the requirements to create an AR engine. With this we define a design for this engine and implement it in AREA.

5.1 Structure of AREA

The structure of AREA can be divided into four parts: UI, models, data providers and a core controller. As visualized in figure 5.1 the core controller connects all other three parts together and implements the core logic for the engine. Custom implementations can be made within the model, UI(user interface) and data provider parts.

The main task of the `AREACoreController` class is to prepare the raw data for the user interfaces and thus to unload the workload on the UI-Thread. It uses the heading and the position to calculate the current position on the screen for each POI and decides whether a POI is inside the camera field of view. The results are then used to display or modify existing POI's. This is an essential part for a fast and responsive UI in any Windows Phone Application.

The model part provides all necessary classes to store all important data for the engine. Customized POI's can be inherited from the base class `POIData`, which represents one POI. All POI together are stored within the container class `POICollection`. Those two classes are the primary classes regarding any POI information. The third class `DeviceData` contains all other information, which are independent of a specific POI. Moving along to the the data providers, their main task is to provide the core controller with sensor data or to query for POI from a local or remote database.

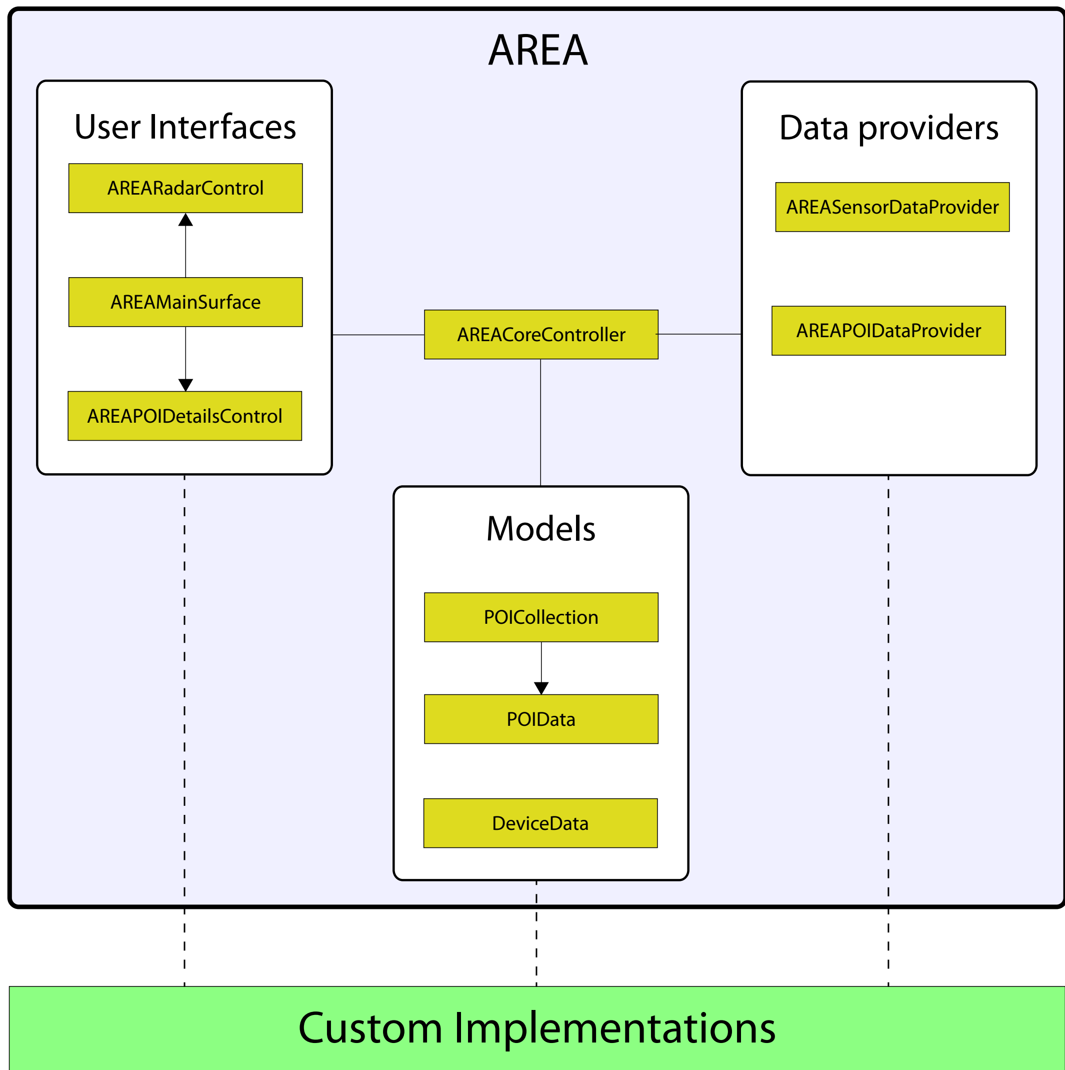

fig. 5.1: Structure of AREA

The UI part uses the processed data from the core controller and draws the UI components to visualize the POI. The class `AREAMainSurface` takes a major role in AREA as it is the entry point to integrate AREA in an application. After creating an instance of this user control, it automatically initializes the core controller to start using the AR engine.

5.2 Graphic frameworks

In this thesis we implement the AR engine on the Windows Phone 8.1 platform. To do so, we use the programming language C# using the .NET Silverlight Framework. On Windows Phone it is possible to use various graphic libraries such as Mono, XNA Framework or native C++ using DirectX. Mono is based on the XNA Framework and provides a simple cross-platform framework. The license for this framework basically is open-source, but parts of it may be restricted by Microsoft patents and therefore it difficult to decide whether to use it or not. In addition to this, the Windows Presentation Foundation(WPF) which is an essential framework for Windows Phone applications, is not supported. The developer needs to define it's own UI graphics, due to not been able to use XAML(Extensible Application Markup Language) in their application. Thus it is not compatible with Windows Phone. Moving to the XNA Framework, it is uses the advantages of DirectX to create a fast and native graphic-based application. However the development has been halted and in the future the importance of XNA as a graphic library will diminish and is therefore not of importance. All Windows Phone devices have an implemented DirectX library to draw graphics on the screen. Using the programming language C++ it is possible to create a fast and powerful graphic engine to handle many calculations per frame. The disadvantages are, that developers have to define their own user controls, making it more complex and having a different look and feel in contrast to a standard Windows Phone 8.1 App. A workaround to this is to combine DirectX and WPF, making it possible to use XAML with the code behind in C# and still be able to use all the advantages as in a native C++ application. The .NET framework provides an interface to establish a communication between those programming languages. We decided against it, because the number of POI and therefore the number of calculations per frame does

not exceed a fixed number. Having more than 100 moving UI components distributed on a screen is less likely to happen as the visibility of the app decreases. Therefore we use XAML to implement UI components and to create a fast and native UI with the Windows Phone proper look and feel. The advantages of using XAML is due to the fact, that the event handling is already implemented and the elements can easily be referenced by storing them in a variable.

5.3 Additional frameworks

To efficient calculate the position of every POI we use matrix transformations using the `Matrix` and `Vector3` namespaces from the XNA Framework, which are still widely used. It provides a range of helper classes to calculate and create matrices and to multiply them with vectors. As for the sensors we use the Motion API to create get the rotation matrix of this device. The Compass class is needed to determine the course of the device. The `Geolocator` class provides with the GPS coordinates. For AREA we implement a JSON database and aid the (de-)serialization process using the Json.NET framework by Newtonsoft. A JSON string is interpreted with this framework by using a predefined class structure, which represents the same structure as the data string.

5.4 Threading model in AREA

The architecture of AREA defines two major threads: A *UI-Thread* and a *controller-Thread*. The UI-Thread generates and manipulates new and existing UI components. This thread is part of the .NET framework and updates independently when changes to the UI are detected. To update the UI components from a different thread the *Dispatcher* of the UI-Thread is called. The dispatcher handles the queue of work of this thread, while the UI-Thread processes these tasks. The controller-Thread manages the control of the update loop. In fixed intervals a update is called, which then processes all information for the UI and calls the dispatcher to update it accordingly on the screen. This includes

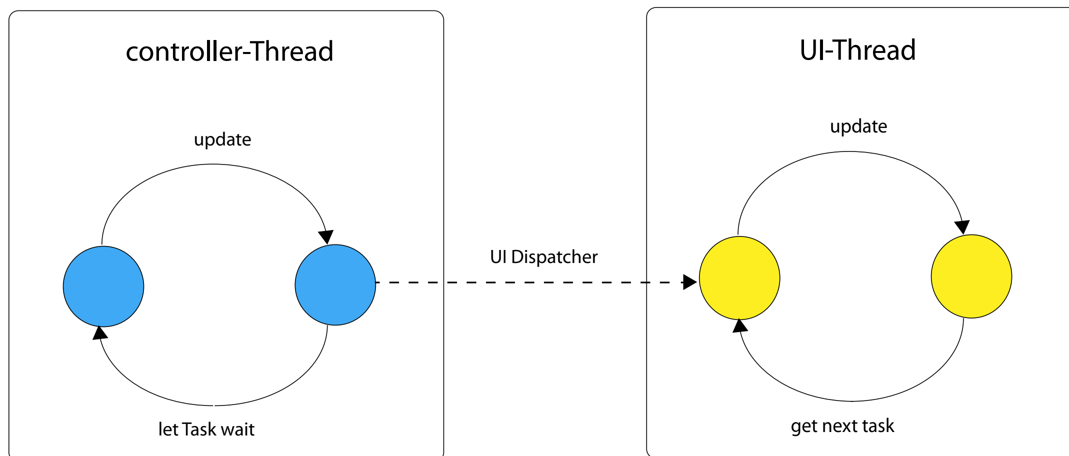

fig. 5.2: Threading structure of AREA with the thread's life loops

POI position updates on screen and radar. A graphical representation of this threading model is illustrated in figure 5.2.

5.5 Database

Although the structure of AREA is very dynamic, we already offer default interfaces to a JSON database and to query from a remote server using a HTTP GET request, which response is then processed as a JSON file. To facilitate the dynamic structure of JSON databases all attributes in the models `POIData` and `POICollection<POIData>` are tagged with a Json.NET own attributes. For this example application, a database of POI are queried from the open source database OpenStreetMap(OSM) and stored within a local database.

The base class `POIData` stores the latitude, longitude and altitude information of a POI. These are then converted to a local Cartesian coordinate system ENU(Earth-North-Up) and stored in the position attribute of a POI. The origin is centered on the position of the device and facilitating so further calculations.

6

Implementation details

We describe in this chapter some selected details of the implementation of AREA. First, we lay out the used UI components. Second, we discuss the details of the used sensors. Third, we explain the update process. Forth, we lay out the process for radius changes. Fifth, we explain details of each UI update. Lastly, we lay out the used mathematical expressions.

6.1 The used UI components

To show the POI on the screen we use components derived from the `UIElement` base class. The most important are summarized in the following table 6.1.

To integrate AREA in an application the `AREAMainSurface` is added to a page. It contains all other components. Moving on, the `AREAPOIControl` is a graphical representation of a POI, which is added to the drawSurface. A radarPoint is the representation of the position of a POI on the radar. Detailing information of a POI is shown within the

name	type of class	remarks
AREAMainSurface	UserControl	main user control
AREAPOIControl	UserControl	represents a visible POI
AREARadarControl	UserControl	represents the radar
POIDetailsUserControl	UserControl	is shown, when a POI is selected
drawSurface	Canvas	all visible POI are drawn here
radarPoint	Ellipse	represents a POI on radar

tab. 6.1: Table of important UI components and their usage

AREAPOIDetailsControl after the user selects it. We decided to use user controls so any developer using this engine can simply add this user control like any other UI element and still be able to make individual changes to it. In addition to that the XAML and its C# code are stored in a separate file. Nevertheless a default layout is implemented, which automatically adjusts to different sizes.

6.2 Sensors

The sensor reading is implemented in the class SensorDataProvider. The Motion class from the Motion API provides the rotation matrix, which represents the current heading of the device. An ordinary rotation matrix composed of the rotation angles around the x-,y- and z-axis with the corresponding pitch, yaw and roll values, provides the wrong rotation matrix. This rotation matrix works only in portrait mode and a change of the device's orientation concludes in a wrong rotation matrix. The Motion class considers these rotations, so any heading and orientation is possible. The GPS position is updated by default every second and with a moving threshold of five meters, but can be modified if needed. If a GPS signal is available, the new position is saved. The current listing 6.1 describes the motion and GPS update.

```
1
2  private void Motion_CurrentValueChanged(object sender,
3                  SensorReadingEventArgs<MotionReading> e){
4          AREACoreController.device.RotationMatrix =
5                  motion.CurrentValue.Attitude.RotationMatrix;
6          }
7
8  private async void UpdateLocation(){
9          Geoposition gPos = (await gps.GetGeopositionAsync());
10   var geoPoint = new Geopoint(gPos.Coordinate.Point.Position,
11                 AltitudeReferenceSystem.Ellipsoid);
12   AREACoreController.device.Latitude =
13                 geoPoint.Position.Latitude;
```

```
14    AREACoreController.device.Longitude =
15                   geoPoint.Position.Longitude;
16    AREACoreController.device.Altitude =
17                   geoPoint.Position.Altitude;
18  }
```

Listing 6.1: SensorDataProvider

6.3 The update process

The update process runs on a separate thread. The control-Thread calls in fixed intervals the `AREACoreController.Update()` method to redraw the current frame. Every update begins to calculate the view frustum, which is to determine if a POI is inside the current field of view and if it's shown or hidden. The current heading, or more precise, the rotation matrix is used to generate a rotated view-projection matrix, which is multiplied with every POI to move them around the origin accordingly. We don't need to translate the vectors, because we have a camera centered coordinate system represented in ENU coordinates. The core controller calls the static methods `UpdatePOI()`, `UpdateRadarPOI()` and `HidePOI()` of the UI classes to change the UI.

6.4 Radius changes

The maximal search radius can be adjusted by adjusting the value of the slider. Any changes on the value, may it be by an user or by programming code, will fire an event. The values represent the radius in meters and range from 2 to 50000 meters by default. The listing 6.2 shows the `changeRadius(int value)` static method of the core controller. It asynchronously calls the `POIDataProvider` to get an updated version of the POI collection. Only when the query is completed the results are processed. Further on, every POI is then converted to ENU coordinates and the distance to the device is calculated. The latter is used to sort the list accordingly their distance to the device. This

is used to have near POI be drawn over further POI. Lastly the view-projection matrix is updated.

```
internal static void changeRadius(int radius){
  poiCollection = null;
  Task.Run(async () =>{
    await dataProvider.queryNewData(device.Longitude,
    device.Latitude, device.Altitude, radius);
    foreach (var poi in poiCollection.list){
      if(useRandomAltitude){
        //simulate height differences
        poi.Altitude =
          testSensors.Altitude - 30 + random.Next(60);
      }
      poi.Position = AREAMath.ConvertGPSToENU(poi.Latitude,
        poi.Longitude, poi.Altitude, testSensors.ECEF);
      poi.Distance = poi.Position.Length();
    }
    //sort POI by distance from device
    poiCollection.list.Sort((x, y) =>{
    if (x.Distance < y.Distance) return 1;
    else return -1;
    });
    createViewProjectionMatrix(radius);
  });
}
```

Listing 6.2: Radius changes

6.5 Updating the UI

As mentioned in section 6.3 the corresponding static methods are called to update the UI components. The POI showed on the screen are displayed on a `Canvas` element, which we name drawSurface. POI drawn on this canvas use the `AREAPOIControl`, which can be individualized as the logic and UI is separated from the drawSurface. To update the UI we use the `BeginInvoke()` method of the dispatcher to access this canvas as showed in listing 6.3.

```
1 public static void UpdatePOI(POIData poi){
2     dispatcher.BeginInvoke(() => { mainSurface.UpdateUI(poi); });
3 }
```

Listing 6.3: Update POI on the drawSurface

The main surface represents an instance of the current `AREAMainSurface`. The actual calculation are done within the `UpdateUI(POIData poi)` method. The listing 6.4 shows the code, which runs on the UI-Thread. If the list of visual `AREAPOIControl`s does not contain an entry for the given user control, it is generated and a reference is added to the POIData. The position of each user control is changed with the prior made calculation of the controller-Thread. To reduce the workload on the processors of the device we reuse the POI already displayed on the screen and only recalculate the current position. When a POI is not within the current view, it will be removed by calling the `HidePOI(POIData poi)` method.

```
1 void UpdateUI(POIData poi){
2         if (visualPOIs == null || poi == null)
3                 return;
4         if (!visualPOIs.Contains(poi.userControl))
5                 CreatePOI(poi);
6         MovePOI(poi.userControl, poi.screenX, poi.screenY);
7 }
```

Listing 6.4: UpdateUI method on the UI thread

6.6 Mathematical calculations

The mathematical calculations to create and multiply matrices with each other or with vectors are borrowed from the XNA Framework. All other calculations are outsourced and collected in the static helper helper class `AREAMath`. It provides methods to translate GPS coordinates to ECEF and ENU coordinates. To translate a GPS coordinate into an ECEF coordinate the WGS 84 ellipsoid[13] is used. This ellipsoid is the most used reference ellipsoid to approximate the earth's surface. The shape of this ellipsoid defines two constants: the eccentricity and the semi-major axis. The conversion between ECEF and ENU coordinates needs a reference point and the point itself in ECEF coordinates. In this case the device itself is the reference point and all points are positioned relative to the device's position. Therefore to calculate the ENU coordinates all POI have first to be converted in ECEF coordinates. To display the points on the radar and the drawSurface the modified ENU coordinates are used to convert them into radar and screen coordinates.

7 Demonstration of the application

We present this engine by implementing an example app. The base application contains a slider, a radar with points in it, a background video stream and POI, symbolized by rectangles. It displays the category and the distance to the POI in meters. The main camera on the back of the device captures a video stream of the real-world and it is placed on the background canvas. All other UI components overlay this background video. While the slider and the radar are anchored on the device's screen, the POI are fixed on the video stream and therefore giving the impression, that those are a part of the real-world. The device acts then as a see-through display, which is an essential part of an AR application. The figure 7.1 shows an implementation of AREA.

This app searches for POI in the vicinity of the user and displays the POI according to their relative position to the device. If the user turns the device in the direction of a POI, the POI appears on the screen and moves according to the movements of the user. Details for every POI can be visualized by pressing on them. In this example the details span over the whole display and the POIs on the background are hidden. In figure 7.2 the details view of a POI is presented. The usage of the OSM database makes it rather difficult to predict which tags are existent and if those are even set. Thus we decided to show the most used tags like name, amenity(category), address, GPS information and some random text to simulate additional information.

On the upper left the radar is displayed. The center symbolizes the current user's position and around it the position of the POI, dependent of the current course of the device. It can be seen as a view from top looking down to the user. In addition, the distance from the center is equivalent to the actual distance to the POI in relation to the maximum radius. The light part of the radar stands for the field of view and thus the dark part for

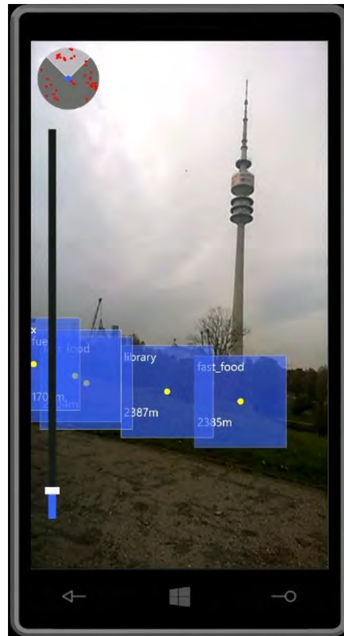

fig. 7.1: Portrait view of AREA

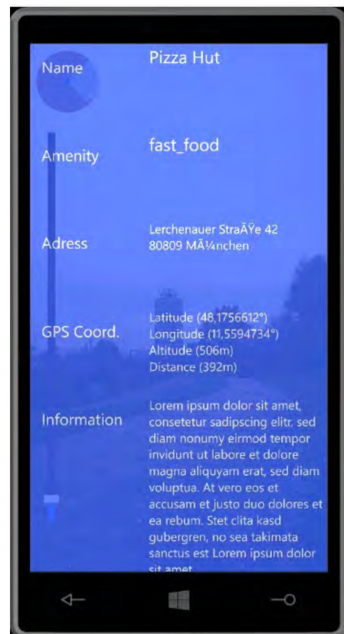

fig. 7.2: A POI details view

fig. 7.3: App in landscape mode

the area and the POI, which are currently not visible through the device. A change of the device's orientation, will automatically change the field of view of the radar.

Bellow the radar, the slider is displayed. The user can increase and decrease the search radius of the engine by dragging the slider up or down or by simply tapping on the desired part of the slider. The numerical value pops out after the value has changed. A smaller set value removes those POI, which have a greater distance to the user than the radius and therefore reducing the number of visible POI. The unit of the value is their equivalent in meters.

The engine reacts to any orientation changes. The radar will align with the device's orientation. Further on, the POI will be also moved on the screen, but the position on the real-world video stream will stay the same. This gives us the impression, that the POI are not fixed to the device, but rather to the real-world. The orientation of the POI's content, although won't change. The changes of the layout while in landscape mode is visualized in figure 7.3.

fig. 7.4: App in an opaque mode

As laid out, the POI are also drawn correctly in any opaque view, meaning even while changing between portrait and landscape mode, they smoothly adapt, rather than abrupt. The view in an opaque mode can be seen in figure 7.4.

8

Comparison of the requirements

A successful project can be measured, if the requirements set in chapter 4 are met and how well they are implemented.

The engine is able to read POI from a database, may it be local or remote. It translates the GPS coordinates of a POI to a local coordinate system using the GPS coordinates of the device. The position of a POI in ENU coordinates can then be reinterpreted into screen coordinates using the rotation matrix from the sensor data.

The engine decides which POI are shown on the screen and which are hidden. Any POI within the search radius is additionally shown on the radar to keep track in which direction the hidden POI are. The engine calculates correctly the movement of the POI in portrait or landscape mode and any position in between, although the content of the UI components won't change.

The POI can be interacted with tapping on them and opening a details panel to show additional information about this POI. The slider changes the current maximum radius of the displayed POI. An user can interact with the slider to change the current maximum radius.

The update loop of the controller-Thread is updated in fixed intervals. Testing the app with 100 POI equally distributed around the device kept the App still responsive, although moving with the device too fast around lowered the framerate noticeably. While monitoring the running application the measured average framerate is around 40-60 frames per second, although the Windows Phone limits it at 60 frames per second.

The responsiveness of the app keeps the UI changes to seem fluid and therefore resulting in efficient calculations. The engine produces a stable application with precise

calculations. The structure of the Engine provides a good maintainability and extendability to change present code and attributes or add new features by deriving existing classes.

The usage of user controls and separating so the engine code from the application code facilitates the integration of the engine in custom applications.

Finally the documentation of code provides a good understanding of the used techniques in the source code.

The following table 8.1 summarizes the requirements and every entry is rated with any of the following markers: ++(very good), +(good), +/-(neutral), -(poor), –(bad).

requirements	rating
Show POI on screen	++
Show POI correctly in landscape, portrait and opaque mode	+
Hide POI not currently on the cameras field of view	++
POI need to react to interactions	++
Sensor data need to be acquired to determine devices position and heading	++
POI data need to be acquired from a database	++
POI calculations need to be updated in real-time	+
Maximal search radius needs to be adjusted	++
Show POI on the radar	++
Additional information is displayed when POI is selected	+
Efficient calculations	+/-
Efficient updates on the UI	+/-
High stability	+
High precision	+
High maintainability	++
Extensibility for custom code	++
Ability to embed engine in other applications	+
Good documentation of code	+

tab. 8.1: Comparing the requirements

9 Summary and outlook

The goal of this thesis is the development of a location-based Augmented Reality Engine Application (AREA) for Windows Phone 8.1. The challenge is the implementation of a AR engine from scratch, without using any available AR frameworks.

The engine is an updated version of the original AREA for iOS [1]. The calculations supporting the engine are done with matrix multiplications, which represent various transformations of points and are more efficiently than presented in the original work [14]. The usage of matrices reduces redundant calculations, which allows us to create more complex AR systems, due to the saved CPU workload.

To get this data the data providers query for data, which is then delivered to the core controller. More specific the `POIDataProvider` searches in a database or a remote server for POI near the user and returns them to the controller. As for the sensors, the sensor data provider manages the devices sensors and pushes in fixed intervals the updated values to the controller. To do so it implements a compass, the motion API and obtaining the GPS coordinate of the device. To minimize the GPS updates a threshold is used and therefore unnecessary database queries are minimized. We use this, because in most cases the data won't change, if the GPS position minimally changes and we assume, that the engine will mainly be used in stationary mode. This results in less queries and saves the resources of the device.

To gain extensibility for this engine, a high dynamic structure is used to easily add new features. The usage of user controls, allows an easy integration of AREA in individual applications. The integration of `AREAMainSurface` triggers the initialization of the engine.

The possibility to use JSON files or to create connections to remote servers boosts the usage of individual data structures and therefore to be independent of the engine's logic. This database could be then platform-independent, as seen in the example app using the OSM database.

The future development of AREA can include various improvements as follows.

The contents of the POI can change regarding of the device's heading and be rotated to match the device's orientation. This may improve the readability of the displayed text. Additionally the size of the of the POI could be modified, so POI nearer to the user are bigger than POI further away. To improve the visibility even further, POI having similar directions could be clustered. These clusters merge various POI into one POI and facilitate the selection of a specific POI. If a cluster is selected the POI within are then listed or visualized apart from the others. The content of the radar could be improved by adding a map of the vicinity to better locate the POI. Additionally a full screen map could be shown, if the radar is selected.

References

[1] Geiger, P.: Entwicklung einer augmented reality engine am beispiel des ios. Bachelor thesis, University of Ulm (2012)

[2] Sutherland, I.E.: A head-mounted three dimensional display. In: Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I. AFIPS '68 (Fall, part I), New York, NY, USA, ACM (1968) 757–764

[3] Weilbach, M.: Einsatz von opengl in area. Bachelor thesis, Ulm University (2016)

[4] Inanc, E.: Conception and implementation of a location-based augmented reality kernel. Bachelor thesis, Ulm University (2015)

[5] Bird, R.: Entwicklung einer augmented reality engine am beispiel des windows mobile operating systems. Bachelor thesis, Ulm University (2015)

[6] Ullrich, U.: Realisierung und evaluierung einer mobilen augmented reality engine mit opengl für android. Bachelor thesis, Institute of Databases and Information Systems (2015)

[7] Schäuffele, S.: Integration von "location-based mobile augmented reality tasks" in eine business process management umgebung. Bachelor thesis, University of Ulm (2014)

[8] Schwab, F.: Konzeption und realisierung einer markererkennungsengine für augmented reality applications auf mobilen geräten. Bachelor thesis, University of Ulm (2013)

[9] Geiger, P., Schickler, M., Pryss, R., Schobel, J., Reichert, M.: Location-based mobile augmented reality applications: Challenges, examples, lessons learned. In: 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps. (2014) 383–394

[10] Schickler, M., Pryss, R., Schobel, J., Reichert, M.: An engine enabling location-based mobile augmented reality applications. In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). Number 226 in LNBIP. Springer (2015) 363–378

[11] Pryss, R., Geiger, P., Schickler, M., Schobel, J., Reichert, M.: Advanced algorithms for location-based smart mobile augmented reality applications. Procedia Computer Science **94** (2016) 97–104

[12] Kato, H., Billinghurst, M.: Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In: Proceedings of the 2Nd IEEE and ACM International Workshop on Augmented Reality. IWAR '99, Washington, DC, USA, IEEE Computer Society (1999) 85–94

[13] Committee, D.M.A.W.G.S..D., Smith, R.: Department of defense world geodetic system 1984: its definition and relationships with local geodetic systems. Technical report, Department of Defense (1987)

[14] 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, Ulm University, Ulm (2013)

figures

2.1 A common head up display in a car 4
2.2 A view of a cockpit in a plane 4

3.1 The augmented reality view of HERE Maps 8
3.2 The augmented reality and map view together of HERE Maps 8
3.3 Details page of HERE Maps with additional information for a POI 9

4.1 A possible set up of POI around the user 12

5.1 Structure of AREA 16
5.2 Threading structure of AREA with the thread’s life loops 19

7.1 Portrait view of AREA 28
7.2 A POI details view 28
7.3 App in landscape mode 29
7.4 App in an opaque mode 30

tables

4.1 Table of functional and non-functional requirements 14

6.1 Table of important UI components and their usage 21

8.1 Comparing the requirements 33

Name: Patrik Miguel Gonçalves Matrikelnummer: 707841

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Patrik Miguel Gonçalves