ulm university universität ulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Bachelorarbeit
im Studiengang Informatik

Entwicklung einer Augmented Reality Engine am Beispiel des Windows Mobile Operating Systems

vorgelegt von

Robin Bird

Mai 2015

Gutachter	Prof. Dr. Manfred Reichert
Betreuer:	Rüdiger Pryss
Matrikelnummer	750747
Arbeit vorgelegt am:	15.05.2015

"Entwicklung einer Augmented Reality Engine am Beispiel des Windows Mobile Operating Systems"

Fassung vom Mai 2015

Kurzfassung

Da die Verbreitung mobiler Smartphones stark gestiegen ist, wurden Anwendungen für Smartphones entwickelt, die den Nutzern im Alltag helfen sollen. Eine populäre Technologie ist die "Augmented Reality"-Technologie. Die Realität wird von solchen Softwaresystemen durch zusätzliche Informationen künstlich erweitert.

AREA ist eine Augmented Reality Engine in Form einer Applikation für mobile Endgeräte unter verschiedenen Plattformen. Sie erweitert das Sichtfeld der Kamera durch zusätzlich eingeblendete Informationen zu Orten und Objekten. Diese Bachelorarbeit behandelt die Konzeption und Umsetzung der Augmented Reality Engine unter der mobilen Plattform Windows Phone 8.1 von Microsoft. Dabei wird in dieser Arbeit insbesondere auf die Besonderheiten der Windows Phone Plattform und der Endgeräte eingegangen. Zuerst werden spezifische Merkmale der Windows Phone Architektur erläutert, um auf diesen Erkenntnissen eine Implementierung der App zu entwickeln.

Wenn sich mehrere Objekte am selben Standort befinden, kann es zu einer Clusterbildung kommen, wodurch die Objekte nicht mehr auseinanderzuhalten sind. Die Problematik wird in dieser Bachelorarbeit ebenfalls untersucht, behandelt und eine Implementierung erläutert.

Inhaltsverzeichnis

Zu einem guten Ende gehört auch ein guter Beginn.

Konfuzius, (551 - 479 v. Chr.)

1 Einleitung

In den letzten Jahren hat sich ein neuer Trend entwickelt. Es zeichnet sich ab, dass mobile Geräte die direkt am Körper getragen werden - wie Smartphones oder Wearables - zunehmend den stationären Desktop PC ersetzen. Begründet wird dies durch mehrere Faktoren. Zum einen haben mobile Endgeräte den Vorteil, dass sie stets betriebsbereit sind und zum anderen sind sie inzwischen ständige, fast schon unverzichtbare, Begleiter des Nutzers geworden [26]. Dies hat zur Folge, dass immer mehr mobile Anwendungen entwickelt werden, die den Alltag eines Smartphone Besitzers erleichtern sollen. Exemplarisch ist ein Forschungsprojekt der Universität Ulm, bei dem ein Prototyp eines Softwaresystems entwickelt wurde, welches - prozessgesteuert - Daten von Smartphones sammeln kann. Anreiz dazu gab die Notwendigkeit der Auswertungen langer Papier-Fragebögen im medizinischen, psychologischen, wie auch pädagogischen Bereich. Eine prozessgesteuerte Sammlung und Analyse der Daten soll eine Entlastung der Wissenschaftler von manuellen Arbeiten ermöglichen [25], [21], [23], [22], [19], [11], [2]. Zu weiteren Verwendungszwecken mobiler Applikationen im Alltag gehören beispielsweise Navigationssysteme, Assistenten mit Spracherkennung oder auch Anwendungen zur mobilen Bezahlung per Smartphone.

Eine weitere Entwicklung, die sich abgezeichnet hat, ist die signifikante Verbesserung der Rechenleistung eines Smartphones. Anwendungen, die früher nur auf stationären Desktop PCs denkbar waren, können heute auch auf mobilen Geräten entwickelt werden. Zu dieser Klasse von Software gehören Echtzeitanwendungen, die mittels Sensoren Werte aus der Umgebung aufnehmen und analysieren. Ein Beispiel für diese sind medizinische Anwendungssysteme, die mittels angeschlossenen Sensoren Werte in Echtzeit zur Gesundheit des Nutzers sammeln und diese analysieren können. Eine solche Software ist die Anwendung

xFitXtreme, welche ebenfalls an der Universität Ulm entwickelt wurde [24]. Diese Applikation ermöglicht es, Sensoren zur Pulsmessung oder zur Messung der Sauerstoffsättigung des Bluts am Smartphone anzuschließen.

Ein weiterer populärer Bereich, der zum Ziel hat dem Nutzer Hilfestellung im Alltag zu bieten, sind Augmented Reality Anwendungen. Diese erkennen mittels GPS-Daten und Sensoren die Umgebung des Nutzers und blenden weitere Informationen auf dem Display ein. Die dafür notwendigen Berechnungen müssen dabei in Echtzeit und ohne Verzögerungen durchgeführt werden. Um eine gute Nutzererfahrung zu gewährleisten, müssen Ressourcen effizient und energiesparend eingesetzt werden. Damit der Nutzer die Informationen zum richtigen Objekt zuordnen kann, muss die Anzeige in der GUI ohne sichtbare Verzögerungen zu den Berechnungen erfolgen. Das bedeutet, dass die Applikation unmittelbar auf einen Ausrichtungs- oder Standortwechsel reagieren muss. Dies sind die größten Herausforderungen bei der Entwicklung einer Augmented Reality Applikation.

Die drei größten Plattformen für Smartphones sind iOS, Android und Windows Phone [28]. Für jede dieser Plattformen gibt es bereits einige kommerzielle, wie auch frei verfügbare Anwendungen zum Bereich "Augmented Reality" auf dem Markt. Einige davon werden als Beispiele in dieser Arbeit erläutert. Diese Arbeit hat das Ziel, eine Augmented Reality Applikation, "AREA", welche keine der bereits auf dem Markt verfügbaren Applikationen nutzt, für Windows Phone zu entwickeln und deren Aufbau und Funktionsweise zu erläutern.

In diesem Kapitel werden grundlegende Begriffe, wie "Augmented Reality", die Augmented Reality Engine App "AREA" und Grundlagen zur Umsetzung auf Windows Phone erläutert.

1.1 Augmented Reality

Als "Augmented Reality", im Deutschen auch als "erweiterte Realität" bezeichnet, versteht man eine Erweiterung der Wahrnehmung der Realität unter Zuhilfenahme von Softwaresystemen [1]. Im Rahmen dieser Arbeit wird als "Augmented Reality" die Überdeckung des Smartphone-Sichtfelds mit zusätzlichen Informationen bezeichnet. Diese zusätzlichen Informationen sollen dem Nutzer eine Hilfestellung bieten, um sich in der realen Umgebung besser und umfassender orientieren zu können [1].

Es gibt unterschiedliche Einsatzzwecke für Augmented Reality Anwendungen. Auch innerhalb eines professionellen Umfeldes werden diese Anwendungen genutzt. So können beispielsweise durch kombinierte Schrifterkennungs- und Übersetzungsanwendungen mit der Kamera eines Smartphones Texte vollautomatisch in eine andere Sprache übersetzt werden. Dabei ersetzt der übersetzte Text den ursprünglichen Text auf dem Display. Diese Technologie wird beispielsweise durch die Google Translator App für das Android Betriebssystem verwendet [10].

Weitere Einsatzzwecke sind unter anderem auch militärischer Natur. So werden Brillen für Soldaten entwickelt, die über ein Head-up Display verfügen, welches Informationen über potentielle Gefahren anzeigen oder auch eine virtuelle Karte über das Sichtfeld eines Soldaten legen kann [18].

Ebenso lassen sich in der Industrie zahlreiche Anwendungsmöglichkeiten für Augmented Reality Systeme finden. So nutzt Volkswagen Augmented Reality Systeme, um bei Crash-Tests von Automobilen den simulierten Schaden mit dem in der Realität entstandenen Schaden zu vergleichen [17].

Der für diese Arbeit wichtigste Einsatzzweck ist die Erkennung von "Points of Interests", im weiteren Verlauf auch "POIs" genannt, also bestimmten Interessenpunkten. Zu diesen Orten können zusätzliche Informationen angezeigt werden. Eine Aufgabe, die auf unterschiedliche Art gelöst werden kann. Zum einen hat Google eine Brille mit dem Namen "Google Glass" entwickelt, die direkt im Sichtfeld des Nutzers Informationen einblenden kann [5]. Zum anderen kann diese Funktion ebenso über ein mobiles Endgerät, wie ein Smartphone realisiert werden. Die Kamera des Smartphones wird genutzt, um dem Betrachter die Sicht auf die Realwelt anzuzeigen und anschließend werden weiterführende Informationen direkt auf dem Display über der Kameraansicht eingeblendet. Diese Informationen können beispielsweise Objekte beschreiben oder zu Navigationszwecken eine Richtung weisen.

1.2 AREA

Die "Augmented Reality Engine App", kurz AREA wurde im Rahmen einer Bachelorarbeit von Philip Geiger für das iPhone 4S, basierend auf iOS 5.1 entwickelt. Ferner wurde die Applikation für die mobile Android Plattform portiert. Das Ziel der Arbeit war es, eine Augmented Reality Engine ohne Zuhilfenahme von Teilen bereits auf dem Markt verfügbarer Augmented Reality Apps zu entwickeln [6]. Es handelt sich dabei um eine mobile Anwendung,

die mit Hilfe des GPS Sensors, Beschleunigungssensors und des Kompass' die aktuelle Position und den Blickwinkel des Nutzers erkennt, diesen auswertet, und Informationen zu Points of Interest in der Umgebung des Nutzers anzeigt. Diese POIs werden in Form eines farbigen Punktes und eines Informationsfelds über die aktuelle Kameraansicht auf den Bildschirm gelegt.

Öffnet der Nutzer die Anwendung AREA, so wird zuerst eine Kartenansicht mit den POIs der Umgebung des Nutzers angezeigt. Über einen Button kann der Nutzer den Augmented Reality Modus aktivieren. Dieser öffnet eine Kameraansicht, der die aktuelle Umgebung des Nutzers mit Hilfe der Kamera des Smartphones anzeigt. Dargestellt werden POIs durch einen farbigen Punkt an bestimmten Positionen und eine Vorschau des Namens. Die POIs überdecken dann die Kameraansicht. Abbildung 1.1 zeigt eine Vorschau dieser Kameraansicht auf der Windows Phone Plattform, deren Version im Zuge dieser Arbeit entwickelt wurde.

Abbildung 1.1: AREA: Augmented Reality Ansicht

1.3 Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der Entwicklung der AREA Anwendung für die mobile Windows Phone Plattform. Das Ziel dieser Arbeit ist die Untersuchung der Windows Phone Plattform unter Windows Phone 8.1 und die Entwicklung von AREA auf selbiger. Dafür sollen die Möglichkeiten und Besonderheiten der Plattform herausgearbeitet werden und untersucht werden, ob eine AREA Applikation mit denselben Funktionen entwickelt werden kann. Eine Implementierung soll erarbeitet werden, die eine Reihe von Anforderungen beachtet. Die AREA Applikation auf Windows Phone soll mindestens die gleichen Funktionen wie die Version für Android bzw. iOS bieten. Außerdem soll eine effiziente und modulare Programmierung erreicht werden, damit die Anwendung eine gute Nutzbarkeit und Erweiterbarkeit bietet.

Bei der Entwicklung einer Windows Phone Applikation, im Folgenden auch als "App" bezeichnet, hat der Entwickler die Wahl zwischen einer Entwicklung mit der Programmiersprache *C++* oder der Entwicklung mit dem Framework *.NET* und *XAML* [4].

Eine Entwicklung mit *C++* besitzt den Vorteil, dass der Entwickler auf die *DirectX* Programmierschnittstelle zurückgreifen kann, um grafikintensive Apps wie Spiele durch hardwarenahe Programmierung möglichst effizient zu gestalten. Ein Zugriff auf die *DirectX* Bibliotheken erfordert immer die Nutzung der Programmiersprache *C++*. Der Gewinn an Performance durch diese Entscheidung wird jedoch stark durch den Verlust des Komforts bei 2D Applikationen mit herkömmlichen *Control*s und *Event*s relativiert. Diese müssen mit *C++* selbst gezeichnet und entwickelt werden und es ist keine nutzbare Standardbibliothek für die grafische Benutzeroberfläche, die GUI, vorhanden. Die Entwicklung mit *DirectX* hat auf der Plattform Windows Phone einen entscheidenden Nachteil gegenüber dem Pendant der Desktopsysteme. Um Ressourcen zu sparen, lies Microsoft den Support der Direct2D API auf Windows Phone aus [16]. Da für die AREA-App keine 3D Operationen benötigt werden, wird in dieser Arbeit ausschließlich auf das *.net Framework* und damit auf die Sprache *C#* und *XAML* zurückgegriffen.

Bei der Verwendung von *.net* wird auf die Sprache *Visual Basic* oder *C#* in Kombination mit *XAML* gesetzt. Die logische Programmierung erfolgt mit *Visual Basic* bzw. *C#*, wohingegen die Definition der grafischen Oberfläche und deren Komponenten mit *XAML* erfolgt. *XAML* ist eine auf *XML* basierte Auszeichnungssprache, die es ermöglicht, fertige grafische Komponenten zu einer Seite

hinzuzufügen und auf einfache Weise anzuordnen. In dieser Arbeit wird *C#* und *XAML* zur Implementierung verwendet.

1.4 Aufbau der Arbeit

Die Arbeit beginnt in Kapitel 2 mit einer Untersuchung bereits auf dem Markt erhältlicher Applikationen im Bereich Augmented Reality. In Kapitel 3 wird eine Anforderungsanalyse durchgeführt, die vor allem erläutert, welche Funktionen notwendig sind, um Nutzern eine Hilfe im Alltag zu bieten. Dabei werden sowohl funktionale, nicht-funktionale Aspekte, wie auch Aspekte der Implementierung und Dokumentation beachtet. Anhand dieser Anforderungsanalyse wird in Kapitel 4 ein modulares Architekturkonzept erläutert, in dem die Klassenstruktur, wie auch die Speicherverwaltung erarbeitet wird. Außerdem wird auf die angebotenen Schnittstellen zum Datenaustausch eingegangen. Mit Hilfe des Architekturmodells wird in Kapitel 5 eine Implementierung ausgearbeitet. Diese beginnt auf der physischen Schicht der Sensoren und klärt zunächst, wie die Sensoren des Endgeräts angesprochen und ausgelesen werden. Die Sensorwerte werden dann zur Berechnung der Point of Interests verwendet, welche zuletzt auf der GUI angezeigt werden. Die Schicht der GUI wird zuletzt angesprochen. Nachdem eine Implementierung für die Windows Phone Plattform erläutert wurde, wird diese genauer analysiert und Differenzen zur AREA Anwendung für Android und iOS erklärt. Außerdem wird ein spezifischer Teil der Implementierung gezeigt. Hierbei handelt es sich um die Behandlung der Clusterbildung, falls mehrere POIs übereinander gestapelt sind. Zum Schluss wird die fertige App in Kapitel 6 mit Hilfe von Screenshots vorgestellt und die vorher spezifizierten Anforderungen in Kapitel 7 werden abgeglichen.

2

Verwandte Arbeiten

Es existieren bereits einige verwandte Arbeiten im Bereich "Augmented Reality", die ebenfalls Smartphones nutzen, um Zusatzinformationen zu Gegenständen oder Orten auf dem Display anzuzeigen. In diesem Kapitel werden zwei verwandte Augmented Reality Applikationen vorgestellt.

2.1 HERE Maps

HERE Maps der HERE Europa B.V. ist eine alternative Karten-Applikation für die Windows Phone Plattform. Zentrale Besonderheit der Software ist die sogenannte "Livesight" Funktion.

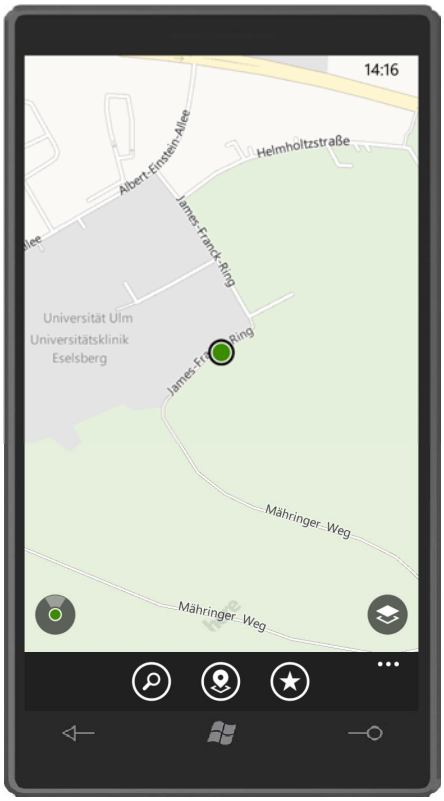

Abbildung 2.1: HERE Maps: Ansicht nach dem Start

Diese zeigt Interessenpunkte in der Umgebung des Nutzers in Form einer Augmented Reality Ansicht an [29]. Beim ersten Öffnen der Anwendung (siehe Abbildung 2.1) findet sich eine klassische Kartenansicht wieder, in der POIs in Form von blauen Symbolen in Halbkreisform in der Karte eingetragen sind.

Der eigene Standort wird durch einen farbigen Kreises angezeigt. Ein angezeigtes Textfeld oberhalb zeigt die Adresse des aktuellen Standorts an und bietet einen Button, der die Livesight Ansicht startet (siehe Abbildung 2.2).

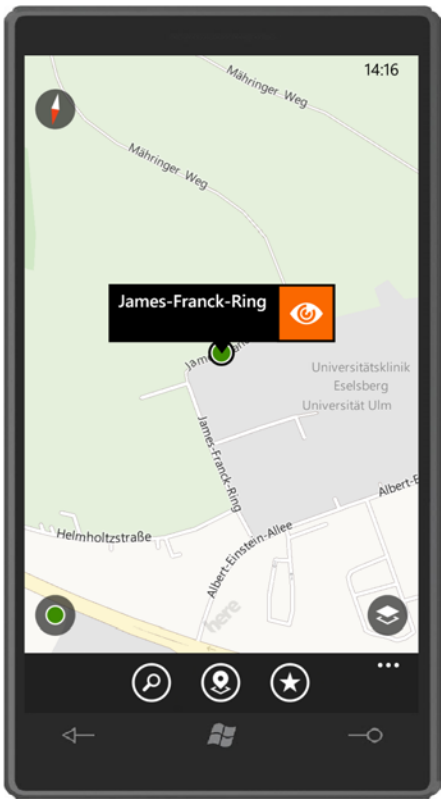

Abbildung 2.2: HERE Maps: Start der Livesight Ansicht

Ein Klick auf die Livesight-Ansicht startet eine Kameraansicht und verkleinert die Kartenansicht zu einem Kreis. Mit Hilfe des Kompass wird die Himmelsausrichtung des Nutzers festgestellt und die Karte entsprechend gedreht, so dass Interessenpunkte vor dem Nutzer in Richtung oberem Rand des Bildschirms angezeigt werden.

HERE Maps bietet im Livesight Modus zwei verschiedene Ansichten, die sich je nach Ausrichtung der Kamera voneinander unterscheiden. Ist die Kamera in Richtung Boden ausgerichtet, so zeigt die App eine runde Karte an, in die POIs in Form von farbigen Stecknadeln eingetragen sind.

Dabei dreht sich die Karte abhängig von der geografischen Ausrichtung des Kompass'. Ein graues Radarfeld, das in Abbildung 2.3 zu sehen ist, zeigt hier an, welche POIs dem Nutzer auf dem Display in der zweiten Ansicht erscheinen.

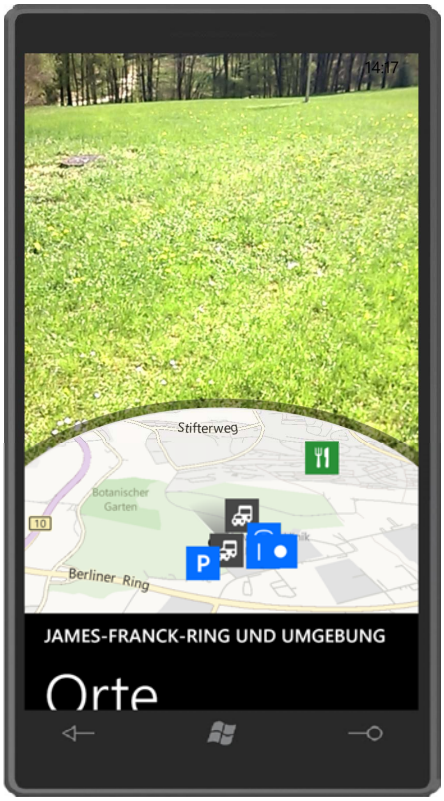

Abbildung 2.3: HERE Maps: Bodenansicht

Bei der zweiten Ansicht handelt es sich um eine Augmented Reality Ansicht. Als Hintergrund wird die Anzeige der Kamera verwendet. Die aktuellen POIs in der Umgebung des Nutzers werden in Form von Textfeldern mit dem Namen des Ortes und der Entfernung zum Ort angezeigt. Ein farbiges Symbol auf der linken Seite kategorisiert den jeweiligen POI. Beim mittig auf dem Bildschirm in Abbildung 2.4 platzierten POI handelt es sich beispielsweise um ein POI der Kategorie "Gastronomie".

Beim Bewegen des Bildschirms ändern die POIs horizontal und vertikal die Position, um die korrekte Richtung anzuzeigen. Dies geschieht jedoch nicht, wenn das Display gedreht wird. Die Software lässt sich also nur im Portraitmodus verwenden. Beim Ändern der Position und Ausrichtung lassen sich fühlbare Verzögerungen feststellen.

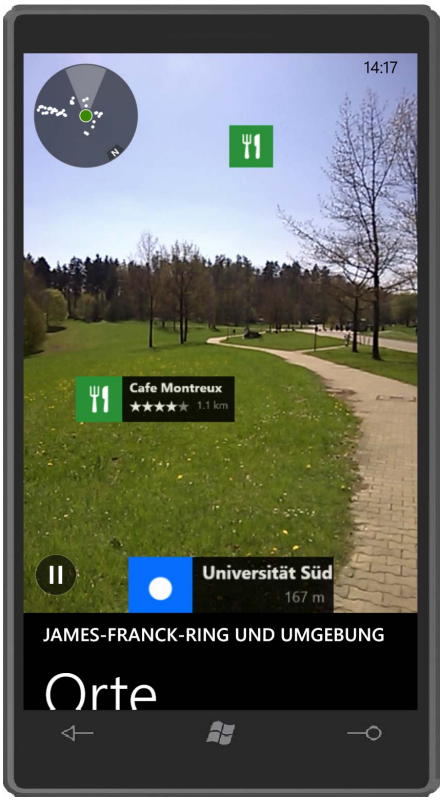

Abbildung 2.4: HERE Maps: Augmented Reality und Kategorisierung von POI's

Zwar bietet die Anwendung keine Möglichkeit, den Radius der angezeigten POIs zu ändern, jedoch ändert sich die Größe der POIs in Abhängigkeit zur Entfernung (weiter entfernte POI's erscheinen kleiner). Außerdem werden die Textfelder bei weit entfernten POIs ausgeblendet. Auf Abbildung 2.5 ist erkennbar, dass die App bei vielen aufeinanderliegenden POIs die Möglichkeit bietet, einen POI durch Klick auszuwählen. Der Name dessen wird eingeblendet und er wird etwas auf die Seite geschoben.

Ein Klick auf einen POI öffnet eine weitere Ansicht, in der Daten wie die Adresse, Fotos, Bewertungen und die Telefonnummer angezeigt werden (siehe Abbildung 2.6).

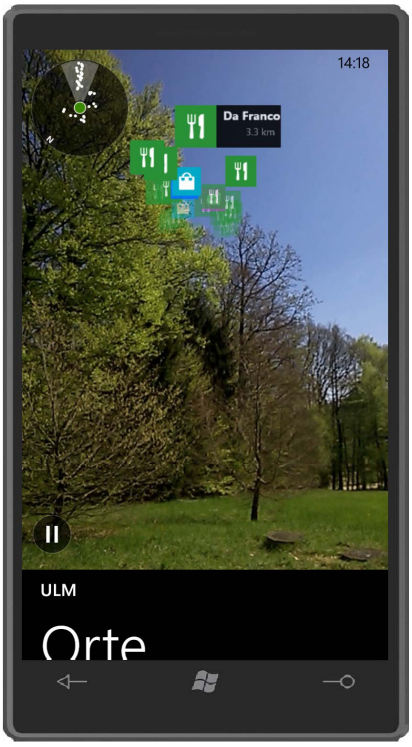

Abbildung 2.5: HERE Maps: Clusterbildung und Auswahl eines POI's

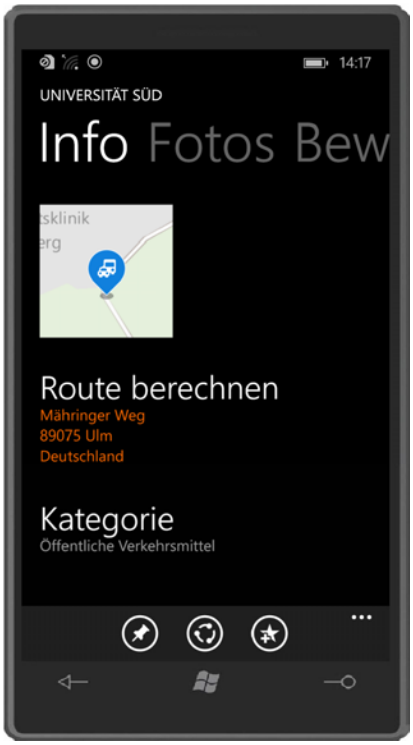

Abbildung 2.6: HERE Maps: Detailansicht eines POI's

2.2 Anatomy 4D

Anatomy 4D ist eine von DAQRI für iOS und Android verfügbare mobile Applikation, die medizinische Modelle von menschlichen Organen in einer Virtual Reality Ansicht dreidimensional anzeigen kann [3].

Beim Start der mobilen Anwendung bietet diese die Möglichkeit, verschiedene Informationsbögen zu menschlichen Organen auf einer Din A4 Seite auszudrucken. Diese Informationsbögen, wie in Abbildung 2.7, enthalten in der Mitte jeweils eine Grafik des entsprechenden Organs.

Die Hauptansicht der Anwendung startet eine Kameraansicht, in welcher nach einer Modellvorlage gescannt wird. Legt man ein ausgedrucktes Blatt mit dem Modell eines Organs auf einen Tisch und richtet die Kamera darauf aus, erkennt die App das jeweilige Organ und erstellt eine farbige und dreidimensionale Ansicht dieses Modells (siehe Abbildung 2.8). Das Modell wird exakt passend zum Umriss des Papierblatts erstellt. Die Software kann verschiedene Organmodelle unterscheiden und wählt automatisch das Richtige aus. Da Ausrichtungsänderungen des Smartphones vollständig ausgeglichen werden, bleibt das Modell immer auf der selben Stelle der ausgedruckten Seite.

Die Anwendung bietet zahlreiche Möglichkeiten an, mit dem Modell zu interagieren. Zum einen erkennt sie eine Drehung des Blatts und dreht auf dem Display auch entsprechend das angezeigte Organmodell. Ebenso werden Bewegungen und Ausrichtungsänderungen des Smartphones ausgeglichen. Indem man die Perspektive ändert, ist es möglich die Unterseite oder auch die Oberseite des Modells zu betrachten (siehe Abbildung 2.9). Durch eine "Pinch-to-Zoom"-Geste mit zwei Fingern lässt sich die Größe des Modells anpassen.

Ein Einstellungsmenü gibt dem Nutzer die Möglichkeit, verschiedene Informationsebenen eines Modells ein- bzw. auszublenden. Um einen zusätzlichen Lehreffekt zu bieten, sind einige Funktionsweisen der Organe animiert. Wählt man beispielsweise das menschliche Herz, kann man einen Herzschlagzyklus beobachten.

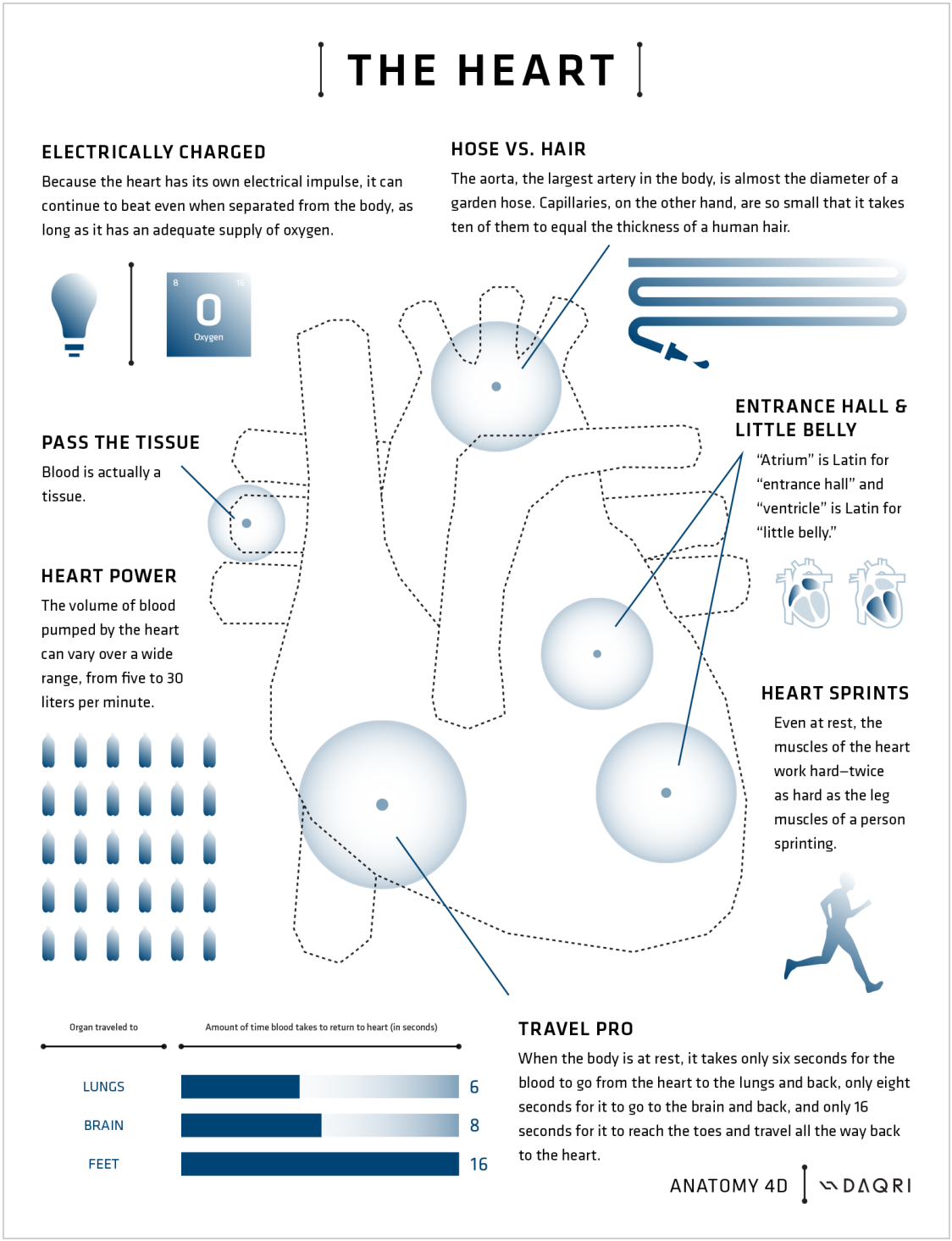

Abbildung 2.7: Anatomy 4D: Informationsbogen des menschlichen Herzens [3]

Abbildung 2.8: Anatomy 4D: Augmented Reality Ansicht und Optionen

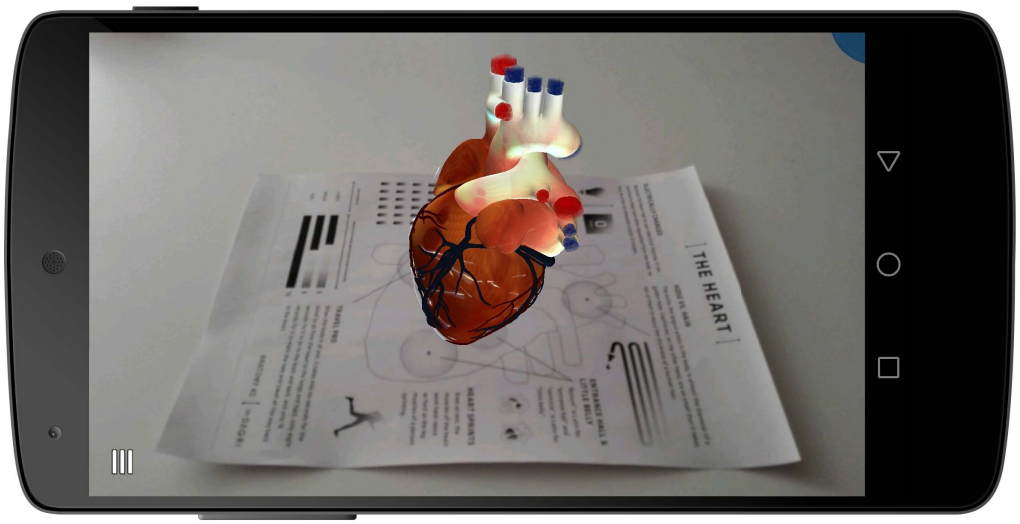

Abbildung 2.9: Anatomy 4D: Änderung der Perspektive

3

Anforderungsanalyse

Die Anforderungen für die AREA Anwendung auf der Windows Phone Plattform werden ähnlich den Anforderungen der existierenden AREA Applikation auf der mobilen Plattform iOS gewählt [6]. Falls sich mehrere Point of Interests in der selben Richtung befinden, so kann es auf dem Bildschirm zu einer Überlappung kommen. Diese Bildung von Clustern soll zusätzlich behandelt werden.

Hauptfunktion der mobilen Anwendung soll es sein, Points of Interest relativ zur Position des Nutzers auf dem Bildschirm anzuzeigen. Hierbei sollen diese die Kameraebene überdecken, so dass sich die POIs einer direkten Richtung oder einem Objekt zuordnen lassen können. Dies impliziert, dass Objekte, die nicht im Sichtfeld der Kamera liegen, auch nicht auf dem Display angezeigt werden. Es soll möglich sein, durch eine unterschiedlich angezeigte Größe der POIs Entfernungsunterschiede mit einem Blick zu erkennen, d.h. weiter entfernte Objekte erscheinen auf dem Bildschirm kleiner.

Um diese Funktionen zu realisieren, muss das Smartphone Zugriff auf Sensoren ermöglichen, um Daten wie Standort, Ausrichtung, Lage, Höhe des Smartphones und Standort und Höhe der POIs zu erhalten. Beispielsweise muss der Kompass verwendet werden, um den Blickwinkel in Grad, relativ zum geografischen Nordpol zu berechnen. Diese erhaltenen Daten müssen verwertet werden um zu bestimmen, ob ein POI im Umkreis des Nutzers auf dem Bildschirm angezeigt werden soll. Abbildung 3.1 soll verdeutlichen, welche POIs ein- bzw. ausgeblendet werden. Außerdem muss die Position des Objekts auf dem Bildschirm berechnet werden. Um auch ausgeblendete POIs zu erkennen, sollen alle Objekte im Umkreis auf einem zusätzlichen Radar angezeigt werden.

Die Performance der Berechnungen spielt eine sehr große Rolle, da die Software verzögerungsfrei arbeiten soll und sie eine Ausrichtungs-, bzw. Positionsänderung sofort erkennen soll. Die angezeigten POIs müssen dann entsprechend in ihrer Position auf dem Bildschirm geändert werden. Diese Berechnungen sollen also in Echtzeit, mit hoher Genauigkeit und unter Berücksichtigung der Stabilität der

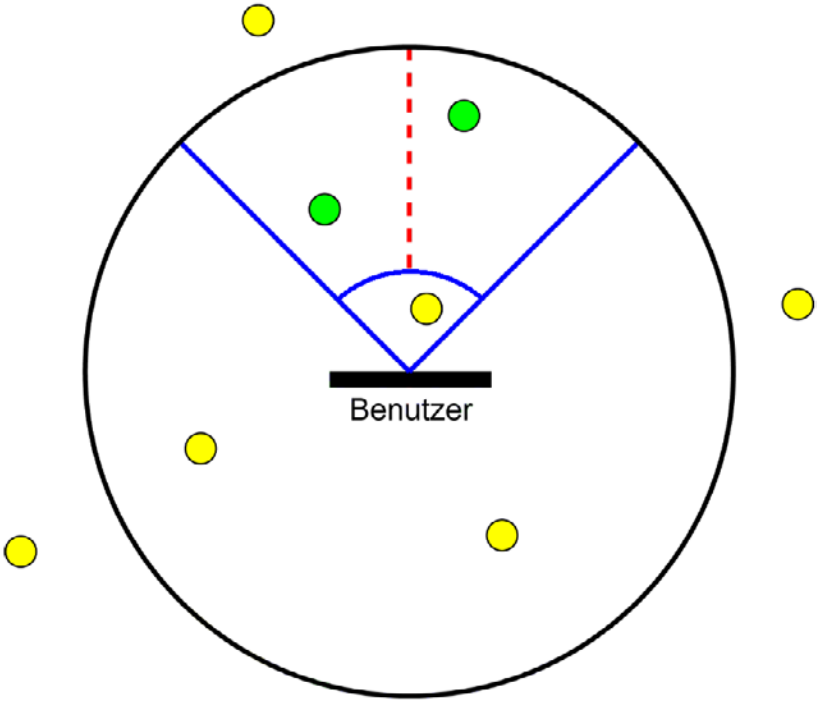

Abbildung 3.1: Darstellung der angezeigten POIs. POIs, die sich innerhalb des minimalen und maximalen Radius befinden, außerdem im Blickwinkel liegen (grün). POIs, die nicht im Augmented Reality Modus angezeigt werden (gelb), eingestellter Radius (rot), Blickwinkel (blau) [6]

Applikation durchgeführt werden.

Bei einem Klick auf einen POI sollen weitere Informationen, wie Entfernung, Höhe und Name zu diesem angezeigt werden. Um zu verhindern, dass zu viele POIs angezeigt werden und die Übersichtlichkeit bzw. Interagierbarkeit leidet, soll es möglich sein einen minimalen bzw. maximalen Radius festzulegen. Innerhalb dieser Grenzen sollen POIs auf dem Bildschirm angezeigt werden bzw. ausgeblendet werden. Zur Einstellung der Radien soll es zwei Schieberegler geben, die bis zu einer voreingestellten Grenze betätigt werden können. Deren eingestellten Werte dürfen sich keinesfalls überschneiden. Befinden sich viele POIs an derselben Position, leidet darunter die Übersichtlichkeit.

Um trotzdem beide POIs voneinander unterscheiden zu können und sich weitere Informationen anzeigen lassen zu können, soll die Anwendung eine Behandlung dieser Clusterbildung integrieren. Mehrere POIs, die sich überschneiden, werden also auf die Seite geschoben, um einen Blick auf den Namen beider zu ermöglichen.

Es sollen alle Ausrichtung des Smartphones möglich sein, um eine permanente Interaktionsmöglichkeit mit der Software zu gewährleisten. Des Weiteren sollen dadurch POI-Namen ständig lesbar bleiben. Insbesondere sollte der Portraitmodus und der Querformatmodus unterstützt werden.

Die AR-Engine soll leicht in Projekte einzubinden sein und es sollen einzelne Module übernommen werden können. Hierfür müssen öffentliche Schnittstellen und eine Modularität, wie eine Standardisierung angeboten werden. Module sollen ebenfalls einfach erweiterbar sein. Eine lückenlose Kommentierung und eine Modularität soll eine bequeme Wartbarkeit, Erweiterbarkeit und Verständlichkeit ermöglichen.

Alle Anforderungen werden in Tabelle 3.1 nochmals genau aufgelistet:

Anforderung	Typ der Anforderung
POI auf Kameraansicht anzeigen	funktional
POI auf Kartenansicht anzeigen	funktional
POI in Kameraansicht nur anzeigen, wenn im Sichtfeld	funktional
POI in Kameraansicht und Kartenansicht sollen auf Interaktionen reagieren	funktional
Sensordaten zur Positionierung des Windows Phones auslesen (Beschleunigung, GPS, Magnetfeld)	funktional
Bei Bewegung des Smartphones, Daten und POI in Echtzeit aktualisieren	funktional
Einstellbarer Minimalradius für die Entfernung	funktional
Einstellbarer Maximalradius für die Entfernung	funktional
Größe des angezeigten POI in Abhängigkeit zur Entfernung ändern	funktional
Vermeidung von Clusterbildung	funktional
Radar in Kameraansicht mit POIs aus der Umgebung und im Radius	funktional
Weitere Informationen bei Interaktion mit POI einblenden	funktional
Portrait und Landscape	funktional
Dialogfelder an Ausrichtung anpassen	funktional
Hohe Effizienz von Berechnungen	nicht-funktional
Hohe Effizienz beim Zeichnen der Anzeige	nicht-funktional
Hohe Stabilität	nicht-funktional
Hohe Genauigkeit	nicht-funktional
Gute Wartbarkeit	nicht-funktional
Einheitliche Spezifikation von POI	Implementierung
POIs sollen intern erweiterbar sein	Implementierung
Einfache Einbindung in andere Anwendungen (Modularität)	Implementierung
Lückenlose, verständliche Kommentierung	Dokumentation

Tabelle 3.1: Anforderungsanalyse und Klassifizierung [6]

4

Architektur

In diesem Kapitel wird der Architekturentwurf von AREA vorgestellt. Zum einen wird die Klassenstruktur erläutert, hierbei wird insbesondere darauf eingegangen, wie eine Modularisierung bei AREA erreicht wurde. Außerdem wird das Datenhaltungsprinzip der Applikation betrachtet. Diese wurde vereinheitlicht und eine Schnittstelle entwickelt.

Zum Zwecke der Modularisierung und der Erweiterbarkeit wurde AREA auf Basis eines Model-View-Controller Konzepts entwickelt. Für die Model Schicht wurde ein zentraler Speicher entwickelt, welcher geladene POIs einheitlich speichert. Dieser ist leicht anpassbar, wodurch die Struktur der POIs erweitert werden kann. Außerdem gibt es eine Schnittstelle, die genutzt werden kann, um der Engine POI-Daten bereitzustellen. Zum einen können Daten über ein *XML*-Dokument geladen werden, welches von einem *XML*-Parser gelesen wird. Zum anderen können Daten über eine Schnittstelle geladen werden, die *JSON*-Dokumente annimmt. Dies bietet durch seine Leichtgewichtigkeit insbesondere dann Vorteile, wenn Daten über das Internet geladen werden müssen. Die Schnittstellen bieten Unabhängigkeit von der Plattform. So besitzen beispielsweise alle AREA-Apps für iOS, Android und Windows Phone dieselben Schnittstellen und können eine gemeinsame Datenquelle verwenden.

Zu den Views zählen im Wesentlichen die Basis-Kartenansicht, die beim Start der App angezeigt wird, wie auch die Augmented Reality Ansicht. Zentraler Bestandteil ist die *Location View*, eine View, welche größer als der eigentliche Bildschirm ist und über die Views für die angezeigten POIs geladen werden. Ebenso gibt es Views für den Radar und die Anzeige des Blickwinkels über dem Radar.

Die Controller übernehmen das Einlesen von Sensordaten, wie auch die Berechnung, welche POIs sich in der *Location View* momentan befinden bzw. welche sich nicht mehr in Sichtweite des Nutzers befinden. Um diese Funktionen zu erfüllen, wurden die Controller zweigeteilt in einen *SensorController* und in einem *LocationController*.

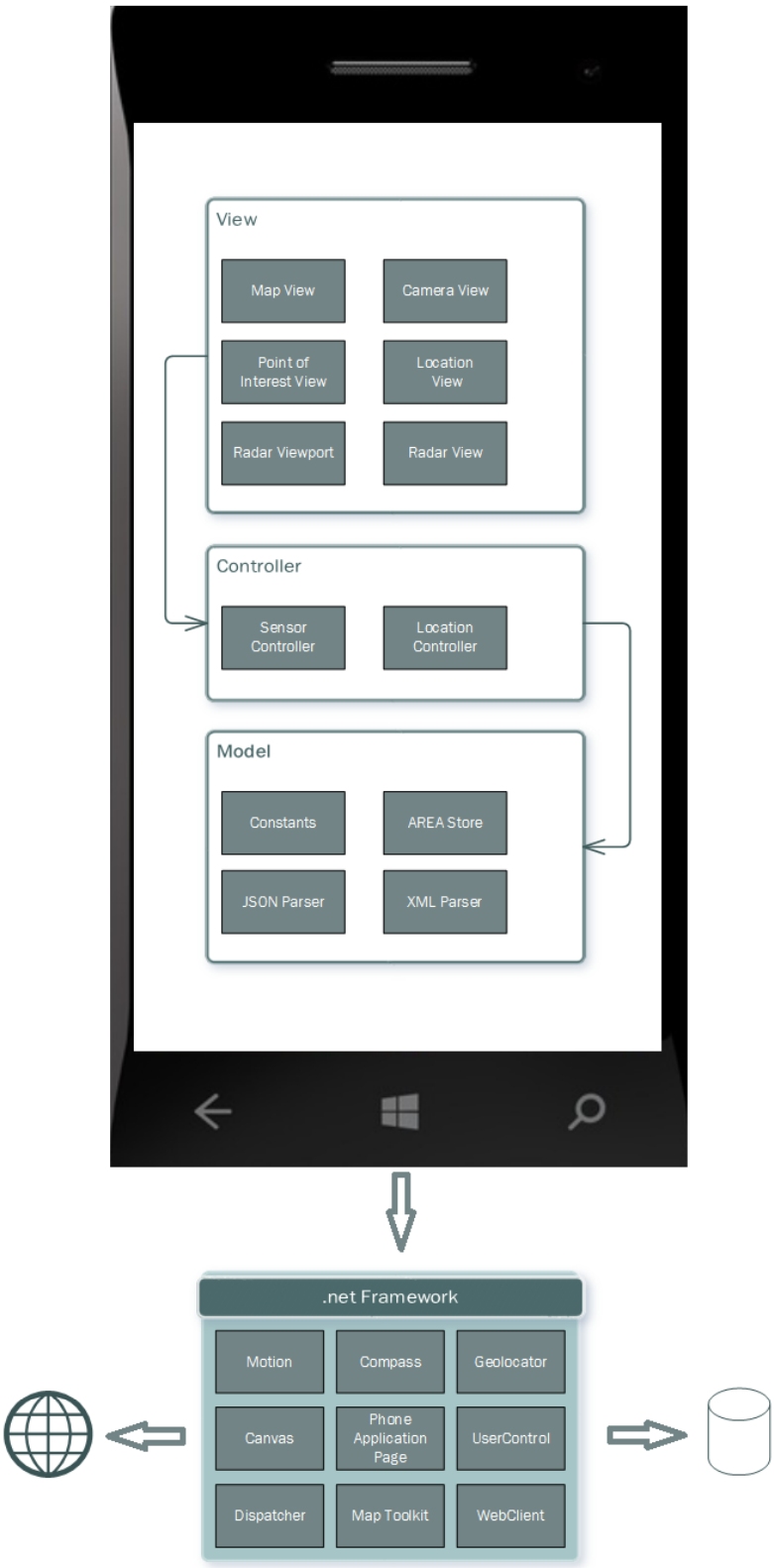

Abbildung 4.1: Mehrschichtenarchitektur von AREA auf Windows Phone

4.1 Klassen

Abbildung 4.2 zeigt eine Übersicht über die Klassenstruktur in Windows Phone. Durch die Modularität kann die Engine leicht in andere Anwendungen übernommen werden. Hierfür muss nur eine Instanz der *CameraView*-Klasse erstellt werden. Diese registriert ein *LocationListener* Interface und initialisiert die restlichen Komponenten automatisch.

Es existieren zwei Basisviews, die unabhängig voneinander agieren. Die Kartenansicht wird zu Beginn der Applikation gestartet, um eine Karte mit allen POIs in der Umgebung anzuzeigen. Die zweite View ist die *CameraView*-Klasse, welche eine Vorschau der Kamera startet, die *LocationView* initialisiert und über den Bildschirm legt. Ebenso registriert sie einen *AREALocationListener*. Weitere Views sind die *AREARadarView* und die *AREARadarViewPortView*. Diese bieten eine Radaransicht mit den POIs im Umkreis des Nutzers, respektive eine Sicht, welche POIs sich momentan im Blickfeld des Nutzers befinden. Die *POIView*-Klasse stellt einen POI in Form eines Kreises und eines Textfelds mit dem Namen des entsprechenden POI's dar. Diese werden als Kinder der *LocationView* registriert bzw. entfernt.

Der *AREALocationListener* ist ein Interface, welches von der *CameraView* implementiert wird. Der *AREALocationController* wiederum implementiert einen *AREASensorListener*. Die Zuständigkeit des *AREALocationControllers* liegt in der Berechnung der Bildschirmpositionsdaten von gespeicherten POIs. Er gibt die sichtbaren POIs an die *CameraView* Klasse weiter, welche POIs in die *LocationView* einträgt oder löscht. Auf der untersten Schicht steht der *AREASensorController*. Dieser ist zuständig für den Empfang und die Verarbeitung der Sensordaten. Er implementiert die *Motion* API, *Compass* API und die *GeoLocator* API. Daraus werden die Rohdaten des GPS-Sensors, des Beschleunigungssensors und des Kompasses gelesen. Nachdem die Daten entsprechend verarbeitet wurden, um den Standort des Nutzers, den Blickwinkel und die Lage des Smartphones zu berechnen, werden diese an den *AREALocationController* weitergegeben.

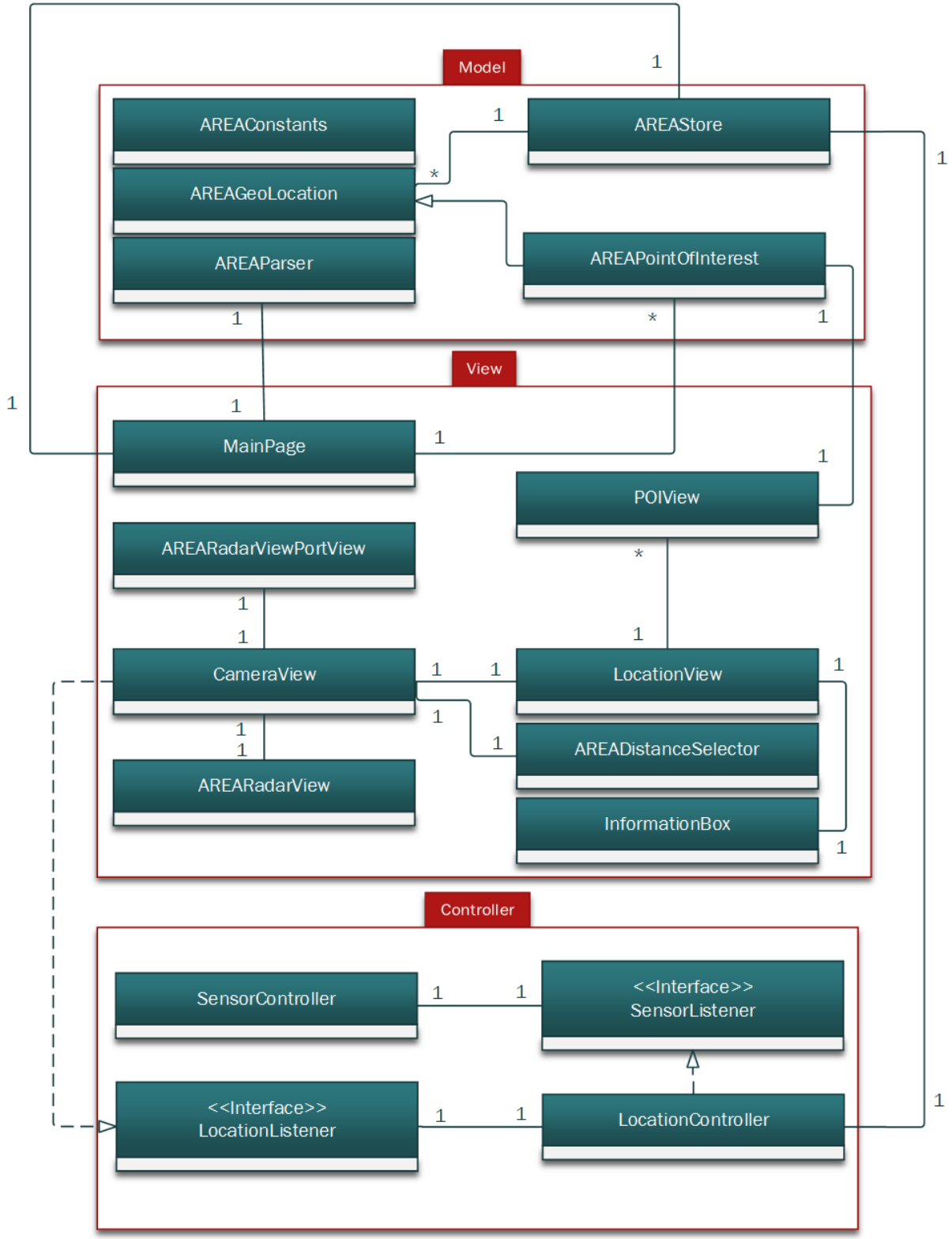

Abbildung 4.2: Klassendiagramm von AREA auf Windows Phone

4.2 Persistierung

AREA für Windows Phone bietet einige Schnittstellen, einen zentralen Speicher und eine einheitliche Datenstruktur, um POIs zu verwalten.

Für die Datenhaltung wird eine Klasse *AREAStore* verwendet, welche einen zentralen Speicher für POIs bietet. Auf diesen wird vom *AREALocationController* zugegriffen, welcher POIs aus dem Store lädt und Berechnungen zur Position durchführt. Als Datenstruktur stehen zwei Klassen zur Verfügung. *AREAGeoLocation* bietet eine einheitliche Struktur für Standortdaten. Sie enthält als Informationen eine GPS-Koordinate vom Typ *GeoCoordinate* und besteht aus der Höhe, dem Längen- und dem Breitengrad eines Standorts. Außerdem werden Werte gespeichert, die vom *AREALocationController* berechnet werden. Dazu gehört die berechnete Distanz ausgehend vom Standort des Nutzers und die horizontale bzw. vertikale Abweichung vom Blickwinkel des Nutzers. Ein weiterer Wert stellt ein Punkt vom Typ *Point* dar, der die Position angibt, an der das POI auf dem Bildschirm gezeichnet wird. Zuletzt wird die Rotation des POI in Grad gespeichert. Dieser Wert wird genutzt, um Ausrichtungsänderungen am Smartphone auszugleichen. Die Klasse *AREAPointOfInterest* erbt die Klasse *AREAGeoLocation*. Zusätzlich enthält sie weitere Informationen zu einem Standort, wie den Namen und eine Beschreibung. Diese Modularität hat den Vorteil, dass Erweiterungen ohne Änderung des Basiskonzepts durchgeführt werden können. Ein *AREAStore* kann beliebig viele *AREAGeoLocation*s enthalten. Eine weitere Klasse, die *AREAConstants*-Klasse speichert Konstanten, die zur Initialisierung der App verwendet werden.

Abbildung 4.3 illustriert das Datenhaltungsmodell als ein Entity-Relationship-Model. Übergeordnet steht der zentrale Speicher *AREAStore*, in dem alle POIs gespeichert werden.

Als Schnittstellen zum Abruf und zur Speicherung neuer POIs werden ein *XML*-Parser und ein *JSON*-Parser bereitgestellt. Der *XML*-Parser kann ein *XML*-Dokument, dessen Schema in Listing 4.2 dargestellt ist, auslesen und die enthaltenen POIs im *AREAStore* speichern. Dabei werden alle Werte berücksichtigt, die über POIs schon vor der Laufzeit bekannt sein müssen. Ein Beispiel eines *XML*-Dokuments, das ein POI-Element enthält, lässt sich in Listing 4.1 einsehen. Eine weitere Schnittstelle bietet ein *JSON*-Parser, der unter anderem dazu verwendet werden kann, um POI-Daten von einem Server zu laden. Genutzt wird er in AREA dafür, die Points of Interest in der Umgebung des Nutzers vom Google Server zu laden. In Listing 4.3 wird das JSON Schema einer POI-Liste, die der JSON Parser verarbeiten kann, vereinfacht dargestellt.

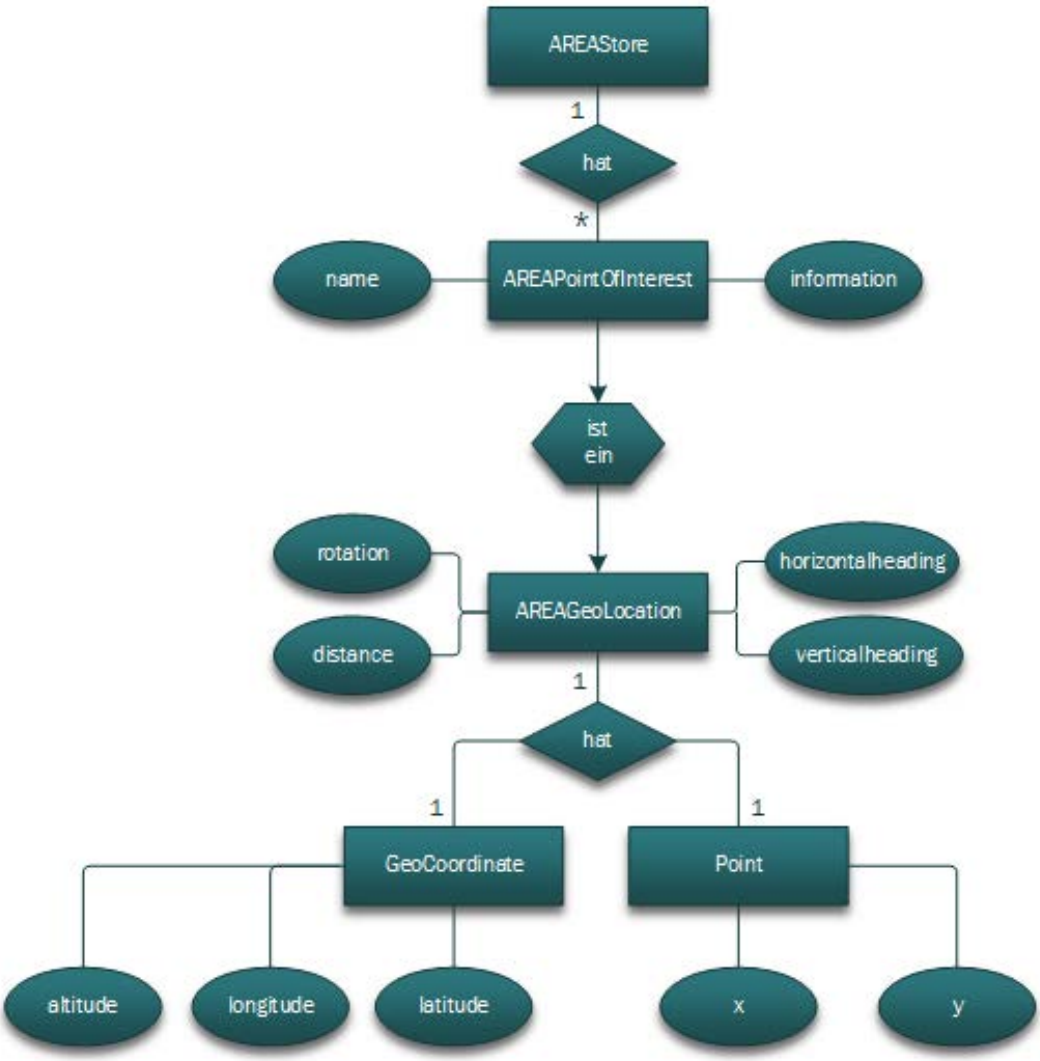

Abbildung 4.3: Entity Relationship Modell eines POI

Soll ein POI erweitert werden, so muss das *XML*-Dokument selbst und die Struktur der Datenhaltung geändert werden. Außerdem müssen der *XML*-Reader und der *JSON*-Parser angepasst werden.

```
<location>
  <latitude>48.398054</latitude>
  <longitude>9.99077529</longitude>
  <altitude>480</altitude>
  <name>Stadthaus Ulm</name>
  <information>Das Stadthaus Ulm wurde vom New Yorker Architekten
    Richard Meier entworfen.</information>
</location>
```

Listing 4.1: Definition einer Location in XML

```xml
<?xml version="1.0"? encoding="UTF8">

<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="locations">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="location" maxOccurs="unbounded" minOccurs=
    "0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:float" name="latitude"/>
              <xs:element type="xs:float" name="longitude"/>
              <xs:element type="xs:float" name="altitude"/>
              <xs:element type="xs:string" name="name"/>
              <xs:element type="xs:string" name="information"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 4.2: XML Schema einer POI Liste

```json
{
 "results": {
      "id": "/results",
      "type": "array",
      "items": {
        "id": "/results",
        "type": "object",
        "properties": {
          "geometry": {
            "id": "/results/geometry",
            "type": "object",
            "properties": {
              "location": {
                "id": "/results/geometry/location",
                "type": "object",
                "properties": {
                  "lat": {
                    //Latitude
                    "id": "/results/geometry/location/lat",
                    "type": "number"
                  },
                  "lng": {
                    //Longitude
                    "id": "/results/geometry/location/lng",
                    "type": "number"
                  }
```

```
                    }
28                  },
                    [...]
30                }
              },
32            [...]
              "name": {
34              //Name des POI
                "id": "/results/19/name",
36              "type": "string"
              },
38            [...]
}
```

Listing 4.3: JSON Schema einer POI Liste

4.3 Windows Phone GUI

In diesem Kapitel soll die Grundstruktur der Entwicklung mit dem *.net Framework* für Windows Phone unter Wahl der Programmierung mit *C#* und der Auszeichnungssprache *XAML* erläutert werden.

Durch die Wahl des *.net Frameworks* kann das GUI Design vom logischem Programmcode mit *C#* getrennt werden. Microsoft bietet eine Vielzahl an vordefinierten *Controls* und *Events*, die vom Entwickler nutzbar sind. Eine grafische Anwendungsseite besteht immer aus einem Basis Container und Containern oder *Controls*, welche angefügt werden können. Als Layoutcontainer stehen unter anderen die Elemente in Tabelle 4.1 bereit. In dieser Arbeit spielt der *Canvas*-Container eine große Rolle, da hier Elemente frei positioniert werden können [4].

Container	**Eigenschaften**
Grid	ähnlich einer Tabelle; Positionierung in Zeilen und Spalten
StackPanel	automatische horizontale oder vertikale Positionierung
Canvas	Positionierung mit absoluten Koordinaten
WrapPanel	Positionierung in horizontaler oder vertikaler Ausrichtung zeilen- oder spaltenweise Anordnung

Tabelle 4.1: Layoutcontainer des .net Frameworks[4]

Vordefinierte *Control*s oder auch selbst geschriebene *Control*s können den Containern in *XAML* als Kindknoten hinzugefügt werden. Es können hier auch Eigenschaften, wie Position, Seitenabstand, Größe etc. festgelegt werden. *Event*s

können hier ebenfalls direkt mit einem *EventListener* verknüpft werden. Auf alle Elemente kann auch mit logischem Code zugegriffen werden. Zwischen einzelnen Seiten einer Applikation erfolgt die Navigation mittels Hyperlinks oder der *NavigationService*-Klasse. Es besteht außerdem die Möglichkeit, UI-Elemente an eine Datenquelle zu binden. Diese muss vom Typ DependencyProperty sein. Genutzt wird dies in der AREA Anwendung für Windows Phone unter anderem dafür, um die Points of Interest in die Karte der Kartenansicht zu binden. Dabei werden die POIs automatisch in der Karte an der richtigen Position eingetragen [15].

5

Implementierung & Implementierungsaspekte

Die Implementierung von AREA für Windows Phone wurde mit der Programmiersprache C# mit Bibliotheken für das Windows Phone 8.1 Betriebssystem auf einem Nokia 920 umgesetzt. Als Entwicklungsumgebung wurde Microsoft Visual Studio Professional 2013 verwendet. Visual Studio ist eine Entwicklerumgebung von Microsoft, mit welcher sowohl Desktopanwendungen in Visual Basic, C# und C++, wie auch Web- und Windows Phone Anwendungen entwickelt werden können.

Dieses Kapitel soll die grundlegenden Konzepte der Implementierung auf Windows Phone darstellen, die Differenzen zu iOS und Android aufzeigen, sowie einen Einblick in einige spezifische Aspekte der Implementierung geben. Zu den grundlegenden Bausteinen gehört die Sensorik und die Darstellung von POIs auf dem Bildschirm. Die Basis bildet sich aus dem Einlesen von Sensorwerten, das vom *AREASensorController* gesteuert wird. Der auf den Rohdaten aufbauende *AREALocationController* berechnet die Darstellung von POIs auf dem Bildschirm und gibt die Informationen an die GUI weiter. Beide Funktionalitäten werden in den folgenden Kapiteln näher betrachtet.

5.1 Sensorik

Das Herzstück von AREA ist der *AREASensorController*, welcher Sensoren anspricht und die erhaltenen Daten verarbeitet. Zum Zugriff auf die Sensoren werden von Microsoft mehrere Schnittstellen bereitgestellt. Mit Hilfe der *Motion*-Klasse kann auf den Beschleunigungssensor zugegriffen werden. Die *Compass*-Klasse kann genutzt werden, um Daten des Kompass' zu erhalten und Positionsdaten können vom *GeoLocator* erhalten werden. Bei allen drei Sensoren handelt es sich um Push-Sensoren. Bei einer Änderung der Werte wird die *CurrentValueChanged* Methode aufgerufen.

In der *startSensoring*-Methode (siehe Listing 5.1) werden die Sensoren initialisiert und sie beginnen, ihre Daten zu liefern.

```
public bool startSensoring()
{
  //Motion Sensors
  motion = new Motion();
  motion.CurrentValueChanged += new EventHandler<
    SensorReadingEventArgs<MotionReading>>(motion_CurrentValueChanged
    );
  motion.Start();

  //Location
  geoLocator = new Geolocator();
  //Threshold for Location Updates: 5 meters, 500 milliseconds
  geoLocator.MovementThreshold = 5;
  geoLocator.ReportInterval = 50;
  geoLocator.DesiredAccuracy = PositionAccuracy.High;
  geoLocator.PositionChanged += new Windows.Foundation.
    TypedEventHandler<Geolocator, PositionChangedEventArgs>(
    geoLocator_PositionChanged);
  compass = new Compass();

  //Compass
  compass.CurrentValueChanged += new EventHandler<
    SensorReadingEventArgs<CompassReading>>(compass_ValueChanged);
  compass.Start();
  return true;
}
```

Listing 5.1: AREASensorController.cs: Start der Sensoren

Der wichtigste Sensor ist der Beschleunigungssensor, mit dessen Daten die Haltung des Smartphones berechnet werden kann. Diese wird verwendet, um in der *AREALocationController*-Klasse das vertikale Sichtfeld zu berechnen und die Rotation des Bildschirms zu erkennen. Die Drehung des Bildschirms wird verwendet, um die einzelnen POIs und Dialogfenster zu rotieren, damit der User die Inhalte in jeder Ausrichtung lesen kann. Außerdem müssen die Kompasswerte an die Haltung (Portrait/Landscape) angepasst werden. Dies wird später genauer erläutert.

Zur Berechnung der Haltung des Smartphones werden die Werte dreier Achsen ausgelesen und verarbeitet. Um den Beschleunigungssensor zu starten, muss eine Instanz der Klasse *Motion* erstellt werden und ein *EventHandler* für das Ereignis *CurrentValueChanged* hinzugefügt werden. Im *EventHandler* selbst werden die Rohdaten verarbeitet. AREA benötigt zwei berechnete Werte, um die Haltung des Smartphones zu erhalten. Zum einen wird der Pitch benötigt, also die horizontale Drehung des Displays, zum anderen die vertikale Haltung.

Da nur die Ausrichtung des Endgeräts bestimmt werden muss, dürfen Werte nicht durch Bewegungen des Geräts, sondern nur durch die Beschleunigung der vorhandenen Gravitation bestimmt werden. Hierfür liefert die Klasse Werte der Gravitation, die bereits gefiltert wurden. Es muss darauf geachtet werden, dass die Methode *CurrentValueChanged* asynchron aufgerufen wird. Da durch die Sensorwerte Änderungen an der GUI durchgeführt werden, muss die Methode an die GUI delegiert werden. Dies muss bei allen Sensoren beachtet werden. Die vollständige Methode lässt sich in Listing 5.2 betrachten.

```
private void motion_CurrentValueChanged(object sender,
    SensorReadingEventArgs<MotionReading> e)
{
  //calculate the rotation of the device in degree
  double temp = Math.Atan2(-e.SensorReading.Gravity.Y, -e.
    SensorReading.Gravity.X);
  pitch = temp * -180 / Math.PI;

  //calculate the vertical heading
  verticalHeading = Math.Asin(e.SensorReading.Gravity.Z)  * 180 /
    Math.PI ;

  //notify the listeners
  dispatcher.BeginInvoke(() => notifySensorsChanged());
```

Listing 5.2: AREASensorController.cs: Übergabe neuer Sensorwerte an die GUI

Der Kompass wird durch die Erstellung einer Instanz der Klasse *Compass* initialisiert. Beim Kompass muss beachtet werden, dass er durch bestimmte Umweltbedingungen unzuverlässig werden kann und rekalibriert werden muss. Das Feld *IsDataValid* enthält einen bool'schen Wert der angibt, ob die Daten des Kompass' zuverlässig verwendet werden können. Falls die Werte unzuverlässig sein sollten, muss eine Kalibrierung durchgeführt werden. Hierfür wird in der GUI ein Overlay *calibrationStackPanel* angezeigt, welches den Vorgang illustriert. Die asynchrone Methode *compass_Calibrate* stößt diesen an und liefert den Faktor der Genauigkeit mit. Wenn er unter einen festgelegten Schwellenwert fällt, wird das Overlay wieder ausgeblendet.

Um die aktuelle Abweichung des Blickwinkels vom geografischen Nordpol zu berechnen, dürfen nicht die Rohdaten der Kompasswerte übernommen werden. Die Klasse *Compass* bietet ein Feld *TrueHeading*, welches die Berechnung des geografischen Nordpols aus den Rohdaten übernimmt. Die so ausgelesenen Werte können in AREA verwendet werden. Um das korrekte *horizontalHeading* zu berechnen, muss allerdings zusätzlich die Haltung des Displays miteinbezogen werden. Beim Wechseln in den Landscape bzw. umgedrehten Landscape Modus, wechselt der Kompass auf eine andere Achse.

Deshalb muss die Displayorientierung ausgelesen werden und zum Winkel des Kompass' in der Restklasse 360, 90° im Landscape Modus bzw. 270° im umgedrehten Landscape Modus addiert werden. Um Kompasssprünge zu vermeiden, wird der Wert zum Schluss in der Methode *smoothedHeading* geglättet.

Die Positionsdaten werden durch einen *Geolocator* abgefragt. Hierzu wird zuerst eine Instanz desselben erstellt und ein *Eventhandler* hinzugefügt, wobei mehrere Dinge beachtet werden müssen. Eine der Anforderungen von AREA ist eine höchstmögliche Präzision. Der Schwellenwert des GPS Sensors sollte also so niedrig wie möglich sein. Dagegen sollten die Effizienz der Berechnungen und die Performance so groß wie möglich sein. Diese Anforderungen stehen im Widerspruch, weil ein niedriger Schwellenwert viele Neuberechnungen erzeugt. Es muss also ein Kompromiss für den Schwellenwert gefunden werden, der hier bei einer Entfernung von 5 Metern bzw. einem Zeitintervall von 50ms greift. Der Wert *MovementThreshold* legt den Schwellenwert für die Entfernung in Metern fest, der Wert *ReportInterval* den zeitlichen Schwellenwert in Millisekunden. Um eine möglichst hohe Genauigkeit der Standortbestimmung zu erreichen, muss zusätzlich der Wert *DesiredAccuracy* auf *PositionAccuracy.High* festgelegt werden. Auffallend ist, dass bei einer niedrigen Genauigkeit keine Angaben zur Standorthöhe gemacht werden können. Diese wird jedoch zwingend benötigt, um eine korrekte vertikale Position von POIs zu bestimmen. Die gesetzte Einstellung garantiert, dass die GPS Sensoren zur Standortbestimmung verwendet werden und auch die Höhe über dem Meeresspiegel geliefert wird.

Die Aktualisierung der Werte in der GUI läuft für jeden Sensor ähnlich ab. Meldet ein Sensor neue Werte, so wird über den *Dispatcher* entweder die Methode *notifySensorChanged*, *notifyLocationChanged* oder *notifyAccuracyChanged* aufgerufen. Diese Methoden geben die Änderung an den registrierten *AREASensorListener* weiter. Daraufhin werden die weiteren Berechnungen im *AREALocationController* durchgeführt, welcher das *AREASensorListener* Interface implementiert. Für den *Geolocator* lässt sich der Vorgang beispielhaft in Listing 5.3 und Listing 5.4 einsehen.

```
[...]
currentLocation = new GeoCoordinate(args.Position.Coordinate.
    Latitude, args.Position.Coordinate.Longitude, stdalt);
dispatcher.BeginInvoke(() => notifyLocationChanged());
[...]
```

Listing 5.3: AREASensorController.cs: Übergabe einer neuen Position an die GUI

```
1 public void notifyLocationChanged()
  {
3   foreach (AREASensorListener obj in listeners)
    {
5     obj.onLocationChanged(currentLocation);
    }
7 }
```

Listing 5.4: AREASensorController.cs: Benachrichtigung der Listener

5.2 Berechnung und Anzeige von Point of Interests

Bis zur Anzeige von POIs auf dem Bildschirm müssen zwei Schritte durchlaufen werden. Als erstes wird im *AREALocationController* anhand der übergebenen Sensordaten berechnet, welche POIs auf dem Display angezeigt werden. Außerdem wird deren Position berechnet. Die Liste der POIs wird dann der GUI übergeben und diese aktualisiert die POI Elemente.

5.2.1 Berechnungen der Controller

Die Berechnungen im *AREALocationController* erfolgen innerhalb mehrerer Schritte. Zuerst werden die POIs in der Umgebung des Nutzers abgerufen und zwischengespeichert. Die Sensordaten werden verwendet, um ein Sichtfeld zu berechnen. Dieses wird verwendet, um für jeden POI zu prüfen, ob er sich im Sichtfeld des Nutzers befindet. Zum Schluss werden für POIs im Sichtfeld des Nutzers die genauen Koordinaten auf dem Display berechnet und die berechneten Daten der GUI zur Anzeige übergeben [6].

Der *AREALocationController* implementiert das *AREASensorListener* Interface und erhält so Informationen über neue Sensordaten. Bei jeder Änderung der Positionsdaten wird im *AREALocationController* die *onLocationChanged* Methode aufgerufen. Diese erhält vom *AREASensorController* auch die neuen Positionsdaten. Daraufhin wird in der Methode berechnet, welche POIs sich innerhalb des eingestellten maximalen Radius *maxDistance* und innerhalb des minimalen Radius *minDistance* befinden. Alle POIs in der Umgebung werden so in einer Schleife geprüft. Befindet sich ein POI innerhalb der angegebenen Distanz, so wird im nächsten Schritt der horizontale und vertikale Kurs, *horizontalHeading* und *verticalHeading*, vom aktuellen Standort aus berechnet. Hierfür werden die Methoden *calculateVerticalHeading* und *calculateHorizontalHeading* aufgerufen und der Standort des POI, der Standort des Nutzers, und beim vertikalen Kurs zusätzlich die Distanz übergeben.

Die Berechnungen erfolgen analog zu den Berechnungen der iOS-Version. Die Werte werden im *AREAStore* zu den POIs gespeichert bzw. aktualisiert. Außerdem werden alle POIs, die sich innerhalb des Radius befinden, in der Liste *surroundingLocations* gespeichert. Anschließend werden die Listener, welche das *AREALocationListener* Interface implementieren, also die *CameraView* über die Änderung informiert und die *surroundingLocations*-Liste wird übergeben. Die *CameraView* nutzt die erhaltenen Daten, um den angezeigten Radar zu aktualisieren. Alle POIs, welche sich in Reichweite befinden, werden auf dem Radar entsprechend ihrer Position eingetragen.

Die Methode *onSensorsChanged* des *AREALocationController* wird aufgerufen, wenn sich die Sensordaten, also die Ausrichtung des Smartphones ändert. Ihr werden vom *AREASensorController* die drei Werte *horizontalHeading*, *verticalHeading* und *pitch* übergeben. In der Methode wird zuerst das Sichtfeld berechnet. Anhand des Sichtfelds kann dann die Platzierung und die Rotation des POI auf dem Display errechnet werden. Die Daten werden innerhalb des jeweiligen POI-Objekts gespeichert und alle sichtbaren POIs in die Liste *visibleLocations* geschrieben. Diese wird durch Aufruf der *onUpdateHeadingLocations* Methode an die *CameraView* übergeben. Ebenso wird die Methode *onUpdateHeadingAndPitch* mit Übergabe von *pitch* und *heading* aufgerufen. Die *CameraView* nutzt die Werte *pitch* und *heading*, um den Radar in die korrekte Ausrichtung zu rotieren und die Dialogfelder der GUI entsprechend der Drehung des Displays anzupassen.

5.2.2 Anzeige der Kameravorschau

Die eigentliche Anzeige der POIs wird von der *CameraView* Klasse gesteuert. Diese besteht aus mehreren Teilen. Der erste Teil stellt eine Vorschau der Kameraansicht bereit, die den grafischen Hintergrund bildet, der zweite Teil bildet die *LocationView*, in welcher die POI Elemente hinzugefügt werden. Außerdem enthält die Klasse einen *AREARadarView*, einen *AREARadarViewPortView* und das *CalibrationStackPanel*.

Nach Laden der AREA Ansicht wird eine Vorschau der Kamera gestartet. Hierfür werden einige von Microsoft bereitgestellte Klassen verwendet. Es müssen zuerst zwei Objekte *CaptureSource* und *VideoCaptureDevice* erstellt werden. Mit *captureSource.VideoCaptureDevice* = *videoCaptureDevice*; wird die gewünschte Kamera auf das neu erstellte Objekt gesetzt. Zuvor wird die Standardkamera, also die rückseitige Kamera, durch die Methode *CaptureDeviceConfigurati-*

on.GetDefaultVideoCaptureDevice() ausgewählt. Eine grafische Darstellung der Kameravorschau lässt sich in einem *VideoBrush* Objekt realisieren.

Das erstellte Objekt muss mit der Methode *SetSource()* noch mit der Kameraquelle verbunden werden. Mit der Methode *captureSource.Start()* lässt sich die Kameravorschau starten.

Um die grafische Ansicht auf dem Bildschirm anzuzeigen, muss das *VideoBrush* in einen GUI Container geladen werden. Dieser wurde im *XAML*-File der *CameraView* Klasse als ein *Rectangle* Objekt erstellt (siehe Listing 5.5). Wird die Ansicht wie beschrieben gestartet, so realisiert man, dass die Kamera um 90° gedreht ist. Um die Kameraansicht an das Display anzugleichen, muss eine manuelle Rotation um 90° vorgenommen werden. Es wird ein *CompositeTransform* Objekt benötigt, bei welchem die *Rotation* Eigenschaft auf 90° festgelegt wurde.

Die komplette Initialisierung der Kameravorschau lässt sich in Listing 5.6 betrachten.

```
<!--Camera viewfinder -->
<Rectangle
  x:Name="viewfinderRectangle"
  Width="480"
  Height="770"
  HorizontalAlignment="Center"
  Canvas.Left="0"
/>
```

Listing 5.5: CameraView.xaml: Definition des Kamerafensters im XAML File

```
public void InitializeVideoRecorder()
{
  if (captureSource == null)
  {
    // Create the VideoRecorder objects.
    captureSource = new CaptureSource();
    videoCaptureDevice = CaptureDeviceConfiguration.
    GetDefaultVideoCaptureDevice();
    captureSource.VideoCaptureDevice = videoCaptureDevice;

    // Initialize the camera if it exists on the phone.
    if (videoCaptureDevice != null)
    {
      // Create the VideoBrush for the viewfinder.
      videoRecorderBrush = new VideoBrush();

      //Do a Transform of 90 degrees of the Videobrush in order to
    get right orientation
      CompositeTransform rotateTransform = new CompositeTransform();
```

```
        rotateTransform.Rotation = 90;
19      rotateTransform.CenterX = 0.5;
        rotateTransform.CenterY = 0.5;
21      videoRecorderBrush.RelativeTransform = rotateTransform;
        videoRecorderBrush.SetSource(captureSource);
23
        // Display the viewfinder image on the rectangle.
25      viewfinderRectangle.Fill = videoRecorderBrush;
        // Start video capture and display it on the viewfinder.
27      captureSource.Start();
      }
29    else
      {
31      MessageBox.Show("Your device doesn't have a camera. Augmented
    Reality mode is not support for your device.");
      }
33  }
}
```

Listing 5.6: CameraView.cs: Start der Kameravorschau

5.2.3 Radar und Sichtfeld

Um den Radar und den RadarViewPort anzuzeigen, müssen von beiden Instanzen erstellt werden und diese dem GUI Container der *CameraView* hinzugefügt werden. Der GUI Container der *CameraView* ist das *Canvas*-Objekt *LayoutRoot*. Ein *Canvas* als Container hat den Vorteil, dass Objekte frei positioniert werden können und auch über die eigentliche Bildschirmgröße hinausreichen können, was später für die *LocationView* benötigt wird. Hinzugefügt werden die Objekte durch die Methode *LayoutRoot.Children.Add()*. Dies muss für beide GUI-Elemente durchgeführt werden. Um den Radar korrekt anzuzeigen, muss er sich immer in Richtung geografischen Nordpol mit dem Nutzer mitdrehen. Dies kann durch ein *RotateTranform* Objekt realisiert werden, das in der Methode *onHeadingAndPitchChanged* um den korrekten Drehungswinkel aktualisiert wird. Es muss beachtet werden, dass das Zentrum der Rotation auf die durch zwei geteilte Höhe und Breite des Radars gesetzt wird. Die Rotation des *AREARadarViewPortView*'s wird analog umgesetzt, nur dass hierfür der Wert *pitch* statt *heading* verwendet wird. Die Realisierung der Rotation wird in Listing 5.7 beschrieben. Außer dem *AREARadarView* und dem *AREARadarViewPortView* wird ein Dialogfenster *calibrationStackPanel* hinzugefügt. Dessen Sichtbarkeit wird als unsichtbar gesetzt und erscheint nur dann, wenn die Kompassdaten vom *AREASensorController* als unzuverlässig gemeldet wurden.

```
public void onHeadingAndPitchChanged(double pitch, double heading)
{
[...]
//Rotate the Radar
rotateRadarView.Angle = -heading;
rotateRadarView.CenterX = radarView.width / 2;
rotateRadarView.CenterY = radarView.height / 2;
radarView.RenderTransform = rotateRadarView;

//rotate the Radar View Port
rotateRadarViewPort.Angle = pitch + 90;
rotateRadarViewPort.CenterX = radarView.width / 2;
rotateRadarViewPort.CenterY = radarView.height / 2;
radarViewPort.RenderTransform = rotateRadarViewPort;
[...]
}
```

Listing 5.7: CameraView.cs: Rotation des Radars

5.2.4 Anzeige der POIs in der LocationView

Um die POI-Objekte auf dem Bildschirm anzuzeigen, wird eine weitere Klasse verwendet, die *LocationView*. Die *LocationView* ist eine Klasse, die von *Canvas* erbt. Die POI Objekte können so nach freien Koordinaten gesetzt werden. Die *LocationView* ragt über die eigentliche Größe des Bildschirms hinaus. Sie wird zentriert in der Mitte des Bildschirms platziert.

Die Wahl, eine weitere Klasse als GUI Container zu verwenden liegt darin begründet, dass die App in allen möglichen Ausrichtungen verwendet werden soll. Dafür hat die *LocationView* eine quadratische Form und die Seitenlänge muss mindestens der Länge der Diagonalen des Bildschirms entsprechen. Die Diagonale des Bildschirms als Kantenlänge der *LocationView* wird deshalb gewählt, damit alle POIs, welche in beliebigen Ausrichtungen des Displays sichtbar sind, auch gezeichnet werden können. Befindet sich das Smartphone beispielsweise im Portraitmodus, ist die Breite des Blickwinkels sehr begrenzt. Möglicherweise liegt ein POI direkt neben dem Display auf der linken Seite. Durch Drehen des Smartphones in den Landscape Modus, wird dieses POI dann sichtbar [20], [7], [8].

Beim Drehen des Smartphones sollen die POIs nach der Anforderungsanalyse stets lesbar bleiben, d.h. sie sollen sich mitdrehen. Die Drehung eines *POIView*'s kann durch ein *RotateTransform* Objekt gelöst werden.

Dieses wird bei jeder Positionsänderungen um den entsprechenden Drehungswinkel angepasst, der im POI als Rotation gespeichert ist. (siehe Listing 5.8)

Ein weiterer wichtiger Grund, weswegen die *LocationView* zur Anwendung kommt, ist die Wiederverwendbarkeit von *POIViews*. Die *LocationView* bietet die Möglichkeit, POIs zentral zu speichern und auch dann zu behalten, falls ein POI aufgrund der Rotation des Smartphones nicht mehr im Blickfeld des Nutzers ist. Dies entspricht einem signifikanten Performance Vorteil, da POIs nur einmal gezeichnet werden müssen und dann nur noch deren Position und Rotation geändert werden muss. Wird das Smartphone gedreht und ein POI verschwindet aus dem Sichtfeld, so bleibt es trotzdem bestehen. Gerät das POI wieder in das Blickfeld, so muss die GUI nicht erneut gezeichnet werden, sondern nur dessen Position geändert werden.

Da die *LocationView* eine Unterklasse von *Canvas* ist, kann die Position von POIs geändert werden, indem die *Canvas.SetLeft* und *Canvas.SetTop* Methoden verwendet werden. Diesen wird ein Wert übergeben, welcher dem Abstand vom linken Rand bzw. oberen Rand des Displays entspricht. Wenn POIs neu in das Sichtfeld hinzugefügt werden oder verschwinden, so müssen *POIViews* dem Basis-Container hinzugefügt bzw. von ihm entfernt werden. Hinzufügen lassen sie sich mit der *Children.Add()* Methode, entfernen mit der *Children.Remove()* Methode. Das Aktualisieren der Position und der Rotation geschieht in der *updatePosition()* Methode innerhalb der *LocationView* (siehe Listing 5.8).

```
1 public void updatePosition(AREAPointOfInterestView poiView)
  {
3   //Set position on Screen
    Canvas.SetLeft(poiView, poiView.poi.point.X);
5   Canvas.SetTop(poiView, poiView.poi.point.Y);
    //Set rotation on Screen
7   RotateTransform rotateTransform = new RotateTransform();
    rotateTransform.Angle = poiView.poi.rotation;
9   poiView.RenderTransform = rotateTransform;
  }
```

Listing 5.8: MyRelativeLayout.cs: Änderung der Position und Rotation eines POI's

Die *LocationView* Klasse steuert nur die Positionierung der POIs und fügt POIs hinzu bzw. entfernt diese. Die Datenhaltung und Steuerung, welche POIs hinzugefügt bzw. entfernt werden, oder bei welchen nur die Position geändert wird, erfolgt in der *onHeadingWithLocationsChanged* Methode innerhalb der *CameraView* Klasse. Die Methode erhält vom *AREALocationController* eine Liste aller aktuellen *AREAPointOfInterests* und überprüft dann, welche POIs sich bereits in der *LocationView* befinden und bestehen bleiben bzw. welche sich

dort befinden, aber entfernt werden müssen. Am Ende werden die Positionen aller POIs aktualisiert. Eine genauere Beschreibung der Vorgehensweise der Methode lässt sich im Unterkapitel 5.4 zur Clusterbehandlung finden.

5.3 Differenzen zu iOS/Android

Durch gegebene Möglichkeiten und Einschränkungen der unterschiedlichen Plattformen wurden verschiedene Lösungsansätze für die selbe Problemstellung erforderlich. In diesem Kapitel sollen einige Besonderheiten der Windows Phone Plattform im Vergleich zu iOS und Android aufgezeigt werden. Es werden für jeden Fall die Varianten der Implementierung auf den Plattformen dargestellt.

5.3.1 Drehung der POIs

Bei der AREA Version für iOS wird die *LocationView*, welche die POIs enthält, selbst gedreht. Das bedeutet, das die *POI*-Views nicht gedreht werden, sondern die *LocationView* an einem Stück mit allen enthaltenen Objekten [6].

Listing 5.9 zeigt die Methode, die in iOS für die Drehung der *LocationView* verwendet wird.

```
1 -(void)didUpdatePitch:(double)pitch
  {
3   // In order to display the locations in a horizontal position, the
      locationView must be rotated when the device is hold in a
    oblique position.
    self.locationView.transform = CGAffineTransformMakeRotation(pitch)
      ;
5 }
```

Listing 5.9: AREAViewController.m: iOS; Rotation der LocationView

Die Rotation auf der Windows Phone Version erfolgt ähnlich der für Android. Alle POIs bleiben zwar initialisiert, wenn sich der *pitch*-Wert des Smartphones ändert, werden also nicht neu gezeichnet, aber die Rotation wird für jeden POI selbst gesetzt. Dies geschieht mit Hilfe eines *RotateTransform* Objekts. Für das Drehen eines *POI*-View's ist die *LocationView* Klasse zuständig. Listing 5.10 zeigt die Drehung in der *updatePosition* Methode.

```
public void updatePosition(AREAPointOfInterestView poiView)
{
  [...]
  //Set rotation on Screen
  RotateTransform rotateTransform = new RotateTransform();
  rotateTransform.Angle = poiView.poi.rotation;
  poiView.RenderTransform = rotateTransform;
}
```

Listing 5.10: MyRelativeLayout.cs: Rotation eines POI's

5.3.2 Positionssteuerung der POI-Views

Die Positionierung der POIs auf GUI Ebene erfolgt auf den drei Plattformen Android, iOS und Windows Phone jeweils unterschiedlich.

Bei der Variante für das iPhone erfolgt die Positionierung innerhalb der *AREAPointOfInterestView*. Die View selbst beinhaltet die Methode *updateFrame*, in welcher die Positionen aktualisiert werden.

Auch bei der Version für Android wird innerhalb einer *View* die eigene Position derer aktualisiert. Dies geschieht durch die Methoden *this.setX()* und *this.setY()* (siehe Listing 5.11).

```
public void updatePosition() {
    this.setX(poi.getPoint().x);
    this.setY(poi.getPoint().y);
    this.setRotation((float)poi.getRotation());
}
```

Listing 5.11: AREAPointOfInterestView.java: Android; Update der Position eines POI [6]

Die Positionierung und Rotation auf Android wird ausschließlich über die *LocationView* gesteuert. Diese fungiert als Container für alle enthaltene *POI*-Views. Da es sich anbietet, die Steuerung dort zu implementieren, wo die POIs gespeichert werden, wird die Position, wie auch Rotation in der *LocationView* aktualisiert. Ebenso werden Dialogfelder mit weiteren Informationen zu einem POI in der *LocationView* erstellt.

```
public void updatePosition(AREAPointOfInterestView poiView)
{
  //Set position on Screen
  Canvas.SetLeft(poiView, poiView.poi.point.X);
  Canvas.SetTop(poiView, poiView.poi.point.Y);
  //Set rotation on Screen
  [...]
}
```

Listing 5.12: MyRelativeLayout.cs: Windows Phone; Update der Position eines POI

5.3.3 Positionierung in verschiedenen Ebenen

Android bietet die einfache Möglichkeit, durch den Aufruf der Methode *view.bringToFront()* ein GUI Element auf die oberste Ebene zu bringen. Das heißt, falls eines oder mehrere Elemente durch ein anderes der GUI überdeckt wurde, so überdeckt dieses nach Aufruf der Methode die anderen. Die Ebenen der dritten Dimension werden also zwischen den Objekten getauscht. Dies wird in der Android Version dafür verwendet, POIs abhängig ihrer Entfernung in verschiedene Ebenen zu bringen. Dafür werden die POIs nach Entfernung sortiert. Startend bei der weitesten Entfernung werden die Views nacheinender auf die oberste Ebene gesetzt. Nach Durchlaufen der Schleife überdecken nähere Objekte die weiter entfernten. Auch iOS bietet diese Möglichkeit durch den Zugriff auf die Eigenschaft *layer.zPosition* eines *UIViews*. Listing 5.13 zeigt, wie die Ebenen der Views auf iOS angepasst werden, nachdem sie nach ihrer Entfernung sortiert wurden.

```
-(void)didUpdateHeadingWithLocations:(NSArray *)locations
{
  [...]
  int i = 0;
  for(UIView *view in subviews)
  {
      view.layer.zPosition = i++;
  }
}
```

Listing 5.13: AREAViewController.m: iOS; Ändern der Ebene eines POI Views [6]

Für Windows Phone gibt es keine vergleichbare Methode. Die einzige Möglichkeit, GUI Elemente nachträglich auf die oberste Ebene zu bringen, ist sie zu entfernen und neuzuzeichnen. Da dies bei jeder Datenänderung geschehen muss, führen die vielen Neuzeichnungen zu einem spürbaren Performanceverlust. Deshalb wurde in der Windows Phone Version bewusst auf diese Funktionalität verzichtet.

Sollten sich POIs verschiedener Entfernungen überdecken, so bildet sich ein Cluster, in dem mehrere Objekte übereinandergestapelt sind. Dies bringt die Problematik mit sich, dass die Interagierbarkeit und Lesbarkeit erheblich erschwert werden. Deshalb wurde in dieser Arbeit eine Clusterbehandlung als Alternative integriert, welche in Kapitel 5.4 genauer erläutert wird.

5.3.4 Auflösung der Geräte

Besonders die mobile Plattform Android ist bekannt dafür, dass viele verschiedene Endgeräte ohne einer standardisierten Auflösung verfügbar sind [9], [27]. Dies stellt Entwickler vor erhebliche Herausforderungen, da es oft wichtig ist, möglichst viele Geräte zu unterstützen.

Microsoft geht einen anderen Weg und erlaubt nur einige standardisierte Auflösungen [14]. Insgesamt sind ab der Versionsnummer "Windows Phone 8" vier verschiedene Displayauflösungen für die Gerätehersteller wählbar. Die logischen Auflösungen werden für den Entwickler skaliert, wodurch der Entwickler nur noch zwei virtuelle und gerenderte Auflösungen unterstützen muss. Die unterschiedlichen Auflösungen mit ihren Skalierungen werden in der nachfolgenden Tabelle 5.1 aufgelistet.

Auflösung	**Auflösung**	**Skalierungsfaktor**	**Skalierte Auflösung**
WVGA	480 x 800	1.0x Skalierung	480 x 800
WXGA	768 x 1280	1.6x Skalierung	480 x 800
720p	720 x 1280	1.5x Skalierung	480 x 853
1080p	1080 x 1920	1.5x Skalierung	480 x 853

Tabelle 5.1: Verschiedene Auflösungen für Windows Phone Geräte

Wie sich aus der Tabelle auslesen lässt, bleibt die Breite bei allen vier Auflösungen bei 480 Pixeln. Einzig die Höhe kann zwischen 800 und 853 Pixeln variieren. Um alle möglichen Auflösungen zu unterstützen, reicht es aus, den Skalierungsfaktor auszulesen. In der Windows Phone Anwendung *AREA* wird die entsprechende Pixelhöhe und Pixelbreite in der *AREAConstants*-Klasse gespeichert. Durch Abfrage des Skalierungsfaktors kann die Bildschirmgröße und Bildschirmbreite wie in Listing 5.14 gespeichert werden.

```
public static void setUpConstants()
{
[...]
if (App.Current.Host.Content.ScaleFactor == 100)
  {
    // WVGA
    kDeviceHeight = 800;
    kDeviceWidth = 480;
  }
  else if (App.Current.Host.Content.ScaleFactor == 160)
  {
    // WXGA
    kDeviceHeight = 800;
    kDeviceWidth = 480;
  }
  else if (App.Current.Host.Content.ScaleFactor == 150)
  {
    // 720p || 1080p
    kDeviceHeight = 853;
    kDeviceWidth = 480;
  }
//Useable Amount of Pixels -> substract 5\% of screen Height for
    Statusbar
kDeviceHeight = kDeviceHeight -  kDeviceHeight * 0.05;
}
```

Listing 5.14: AREAConstants.cs: Windows Phone; Speichern der Bildschirmgröße

Wie sich erkennen lässt, muss von der Pixelanzahl der Bildschirmhöhe noch 5% abgezogen werden, da diese Höhe von der Statusleiste des Smartphones beansprucht wird.

5.4 Clusterbehandlung

Befinden sich zwei oder mehrere POIs unterschiedlicher Entfernung in derselben Richtung, so kann es vorkommen, dass sich diese gegenseitig überdecken und die Bedienbarkeit und/oder Lesbarkeit eingeschränkt wird. Abbildung 5.1 zeigt eine solche Clusterbildung in einem Extremfall. Eine Implementierung zur Behandlung der Clusterbildung wird in diesem Kapitel beschrieben.

Um dem Nutzer einen möglichst guten Überblick über die POIs in der Umgebung zu geben und eine hohe Interaktivität zu gewährleisten, wurden in dieser Arbeit drei Teile zur Lösung der Clusterbildung integriert. Der erste Teil gibt dem Nutzer im Vergleich zur iOS und Android Version zusätzlich zur Einstellung des Maximalradius, die Möglichkeit einen Minimalradius auszuwählen. Dadurch ist es möglich einen exakteren Umkreis der angezeigten POIs zu definieren.

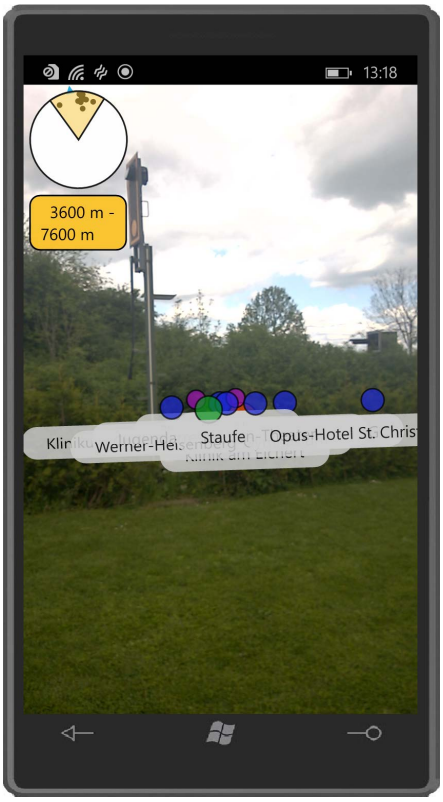

Abbildung 5.1: AREA: unbehandelter POI-Cluster

Beispielsweise können so besonders nahe POIs ausgeblendet werden und eine Clusterbildung in vielen Fällen vollständig vermieden werden.

Falls sich verschiedene POIs immer noch überdecken sollten, so werden diese im zweiten Teil etwas auseinander gezogen. Hierfür wird die Position eines überlappenden POI's etwas in Richtung Horizont verschoben. Dies gewährleistet, dass der Name des POI lesbar bleibt und der Nutzer den von ihm gewünschten POI antippen kann, um weitere Informationen zu erhalten.

Im dritten Teil wird die Größe und Farbe der POI-Views abhängig von ihrer Entfernung geändert. Dadurch kann der Nutzer schneller erkennen, welche POIs näher bzw. weiter von ihm entfernt sind. Die Einstellung eines gewünschten Radius' kann so erleichtert werden.

Um die Einstellung eines minimalen Radius' zu ermöglichen, muss zuerst eine Änderungen an der Datenstruktur vorgenommen werden. In der *AREAConstants* Klasse wird ein Wert *kStdMinDistance* vom Typ *double* hinzugefügt. Eine Variable *minDistance* wird im *AREALocationController* angelegt.

Daraufhin muss die GUI erweitert werden, um die Auswahl des minimalen Radius zu ermöglichen. Wie in Listing 5.15 wird ein weiterer *Slider* zum *AREADistanceSelector* hinzugefügt.

```
[...]
<TextBlock Name="textDistance1" TextWrapping="Wrap" Text="Current
    minimum Radius selected: " Margin="15,5,0,0" FontSize="16"
    HorizontalAlignment="Center"/>
<Slider Name="sliderDist1" BorderThickness="15,10,15,0" Margin="
    15,10,15,0" ValueChanged="sliderDist1_ValueChanged"/>
[...]
```

Listing 5.15: AREADistanceSelector.xaml: Definition eines Sliders zur Einst. des Minimalradius

Der maximal einstellbare Wert des minimalen Radius darf den eingestellen maximalen Radius nicht überschreiten. Um dies zu gewährleisten, wird der höchste einstellbare Wert des minimalen Radius nach Änderung des Maximalradius angepasst (siehe Listing 5.16).

```
private void sliderDist2_ValueChanged(object sender,
    RoutedPropertyChangedEventArgs<double> e)
{
[...]
sliderDist2.Value = maxDistance;
if (sliderDist2.Value < sliderDist1.Value)
{
    sliderDist1.Value = sliderDist2.Value;
}
sliderDist1.Maximum = AREAConstants.kMaxDistance >= maxDistance ?
    maxDistance : AREAConstants.kMaxDistance;
[...]
}
```

Listing 5.16: AREADistanceSelector.cs: Einstellung des Sliderwertebereichs

Im *AREALocationController* muss eine Überprüfung erfolgen, ob die gespeicherten POIs innerhalb beider eingestellten Radien liegen. Bei einer Positionsänderung wird die Methode *onLocationChanged* im *AREALocationController* aufgerufen. In dieser werden alle im *AREAStore* gespeicherten POIs auf ihre Distanz, vom Nutzer aus gesehen, geprüft. Hier wird eine Überprüfung wie in Listing 5.17 hinzugefügt, die auf den minimalen Radius prüft.

```
public void onLocationChanged(GeoCoordinate location)
{
  [...]
  foreach (AREAGeoLocation loc in store.store)
  {
    GeoCoordinate locLocation = loc.location;
    // calculate the distance
    float distance = (float)currentLocation.GetDistanceTo(
    locLocation);
    if (distance <= maxDistance && distance >= minDistance)
    {
      [...]
    }
  }
  [...]
}

```

Listing 5.17: LocationController.cs: Prüfung, ob POI innerhalb eingestellter Radien liegt [13]

Um die sich noch überlappenden POIs zu verschieben, wird zuerst eine Methode in der *LocationView* hinzugefügt, welche einen übergebenen POI verschieben kann (siehe Listing 5.18). Dafür wird von der Y-Koordinate des *POIViews* ein kleiner Wert subtrahiert, um die View auf dem Display nach oben zu verschieben.

```
//Repositioning of a POI if it overlaps with another one
internal void repositionPOI(AREAPointOfInterestView view)
{
  view.poi.point = new Point(view.poi.point.X , view.poi.point.Y -
    60);
  //Set position on Screen
  Canvas.SetLeft(view, view.poi.point.X);
  Canvas.SetTop(view, view.poi.point.Y);
}


```

Listing 5.18: MyRelativeLayout.cs: Verschieben überlappender POI's

Die Steuerung, welche POIs von der *LocationView* entfernt, zu ihr hinzugefügt oder aktualisiert werden, übernimmt die Methode *onHeadingWithLocations-Changed* der *CameraView* Klasse. In diese muss eine zusätzliche Funktion implementiert werden, welche die auf dem Display angezeigten POIs auf Überlappung prüft (siehe Listing 5.19). Für diese Überprüfung wird die Distanz zwischen den Koordinaten zweier POIs berechnet. Ist diese kleiner als die Größe des angezeigten Kreises addiert mit einer Konstante, so liegen diese übereinander

und ein POI muss verschoben werden.

Anzumerken ist, dass jeder POI mit allen anderen POIs verglichen werden muss und eine aufwändige Distanzberechnung durchgeführt werden muss. Um die Performance bei dieser Berechnung zu verbessern, werden die POIs mit Hilfe eines *IComparer* Objekts zuerst nach Größe der x-Koordinate sortiert. Die Klasse lässt sich in Listing 5.20 betrachten. Nach der Vorsortierung wird überprüft, ob die x-Koordinate des ersten Views größer ist, als die x-Koordinate des zweiten addiert mit einer Konstante. Falls dies zutrifft, muss nicht auf Überlappung geprüft werden, da der Abstand in der Horizontalen groß genug ist. In dem Fall wird der Vergleich mit dem nächsten Element der inneren Schleife fortgeführt. Es wird außerdem geprüft, ob die x-Koordinate des zweiten Views schon größer ist als die des ersten addiert mit einer Konstante. Trifft dies zu, so ist auch der Abstand aller folgenden POIs nicht mehr zu überprüfen und der Vorgang wird mit einem weiteren POI aus der äußeren Schleife fortgesetzt.

```
public void onHeadingWithLocationsChanged(List<AREAGeoLocation>
    locations)
{ [...]
//Order the subviews by their X Coordinate. This is for Scanning for
     Overlapping POI's. To improve the performance, only POI's
//in the list with a specific threshold are compared to each other
List<AREAPointOfInterestView> sortedList = new List<
    AREAPointOfInterestView>(subviews);
xPointComparer xSort = new xPointComparer();
sortedList.Sort(xSort);
for (int i = 0; i < sortedList.Count; i++)
{ AREAPointOfInterestView view1 = sortedList.ElementAt(i);
  //Check if there are Views overlapping
  for (int j = i; j < sortedList.Count; j++)
  {
    AREAPointOfInterestView view2 = sortedList.ElementAt(j);
    if (view1.poi.point.X > view2.poi.point.X + 120)
      continue;
    if (view2.poi.point.X > view1.poi.point.X + 120)
      break;
    // calculate distance of two views
    float distance = (float)Math.Sqrt((Math.Pow((view1.poi.point.X -
     view2.poi.point.X), 2)) + (Math.Pow((view1.poi.point.Y - view2.
    poi.point.Y), 2)));
    //If there are POI View Circles overlapping, repositioning of
    view2
    if (distance < (view1.stdCircleSize + view2.stdCircleSize + 50))
      view2.repositionPOI();     [...]

```

Listing 5.19: CameraView.cs: Überprüfung auf überlappende POI's [13]

```
class xPointComparer : IComparer<AREAPointOfInterestView>
{
  public int Compare(AREAPointOfInterestView a,
    AREAPointOfInterestView b)
  {
    if (a.poi.point.X > b.poi.point.X)
        return 1;
    else if (a.poi.point.X < b.poi.point.X)
        return -1;
    return 0;
  }
}


```

Listing 5.20: xPointComparer.cs: Vergleichsklasse, um POI's nach x-Koordinate zu sortieren

Die letzte Anpassung erfolgt auf GUI Ebene. Zur genaueren Erkennbarkeit der entsprechenden Entfernung von POIs, wird die Größe der POI-Punkte abhängig der Entfernung angepasst. Das heißt, weiter entfernte Objekte erscheinen kleiner, nähere erscheinen größer. Außerdem werden die POI-Kreise entsprechend ihrer Entfernung unterschiedlich gefärbt. Dies geschieht nach Vorschlag in der Bachelorarbeit von Julia Müller [13]. Die nächstliegenden POIs sind gelb gefärbt. Umso weiter die POIs entfernt sind, umso dunkler wird deren Farbe. Sie geht über orange nach rot, wechselt dann in einen Grünton und geht schließlich über blau zu lila über. Die Größe des POI-Punkts wird um bis zu 50% verringert. Die Auswahl der Größen und Farben erfolgt im Verhältnis zu den ausgewählten Radien in sechs Stufen. Dafür werden durch Berechnung der Differenz des maximalen und minimalen Radius Stufenweiten gesetzt. Für jede Distanz wird geprüft, in welcher Stufe sich der POI befindet und eine Größe bzw. Farbe bestimmt. Die Anpassung und Berechnung der Werte erfolgt in der *circleFillAndSize* Methode. Diese wird bei der Initialisierung eines *POIView*'s aufgerufen. Am Ende müssen noch die Punkte der POIs korrekt in die Mitte des POI Namens gesetzt werden und der vertikale Abstand zum Textfeld der Höhe des Punkts angeglichen werden. Abbildung 5.2 zeigt dieselbe Umgebung wie die zu Beginn des Kapitels vorgestellte Abbildung 5.1 mit implementierter Clusterbehandlung. Es lässt sich leicht feststellen, dass die Bedienbarkeit und Lesbarkeit der einzelnen POIs erheblich erleichtert wurde.

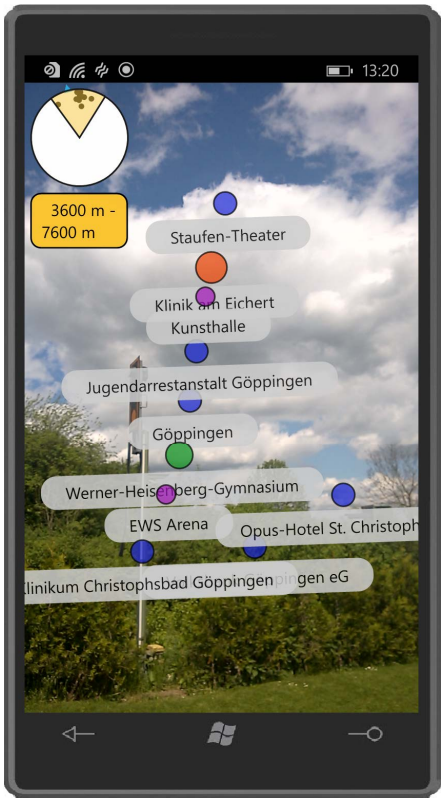

Abbildung 5.2: AREA: behandelter POI-Cluster

6

Vorstellung der mobilen Anwendung

In diesem Kapitel wird die fertig entwickelte Engine und die darauf basierende Applikation AREA für Windows Phone vorgestellt. Hierfür werden einige Screenshots aus der Anwendung gezeigt und diese erläutert.

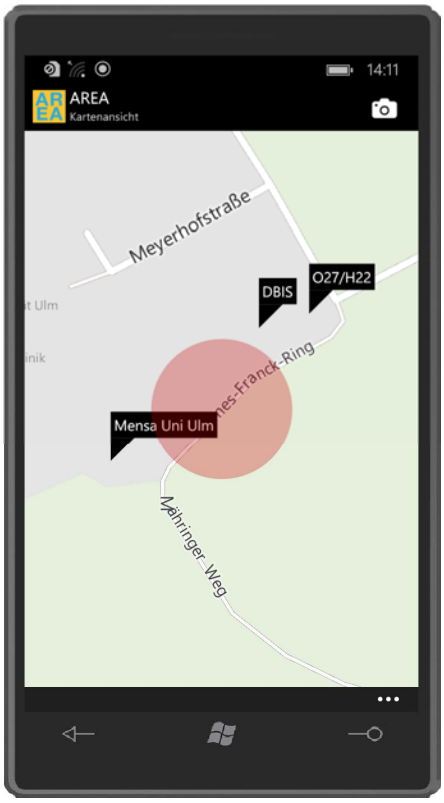

Abbildung 6.1: AREA: Kartenansicht

Abbildung 6.1 stellt die Kartenansicht dar, die direkt nach dem Öffnen der App erscheint. Der Standort des Nutzers wird durch einen roten, halbtransparenten Punkt dargestellt. Dessen Radius variiert mit der empfangenen Genauigkeit des Standorts. Der Zoom der Karte wird ebenfalls in verschiedenen Stufen gesetzt, abhängig von der Genauigkeit des Standorts. Der Nutzer erhält eine Übersicht aller POIs, die ihn umgeben. Die Namen der POIs werden in Form schwarzer

Textfelder mit einer Art Stecknadelkopf dargestellt. Die Stecknadelköpfe stellen den genauen Standort auf der Karte dar. Möchte der Nutzer weitere Informationen zu einem POI bekommen, kann er mit diesem interagieren. Ein Klick auf einen POI öffnet einen Dialog, der weitere Informationen zum POI anzeigt. Es handelt sich dabei um ein Freitextfeld mit einer genaueren Beschreibung und dem Namen des Ortes.

Abbildung 6.2: AREA: Kartenoptionen

In der unteren Sidebar (siehe Abbildung 6.2) lässt sich die Ansicht zwischen einer Karten- und einer Satellitenansicht umschalten. Ebenso gibt es hier die Möglichkeit, die Position des Nutzers erneut zu bestimmen. Bei einem Klick auf den Standort-Button wird der rote Standortkreis neu gezeichnet und die Karte auf den aktuellen Standort des Nutzers neu zentriert. Die Zoom-Stufe der Karte wird durch diesen Vorgang nicht beeinflusst, damit der Nutzer seine eingestellte Detailstufe beibehalten kann.

Der Augmented Reality Modus lässt sich durch einen Klick auf den Kamerabutton in der oberen, rechten Ecke starten. Abbildung 6.3 zeigt den gestarteten AREA Modus. Dieser ist der Hauptbestandteil von AREA.

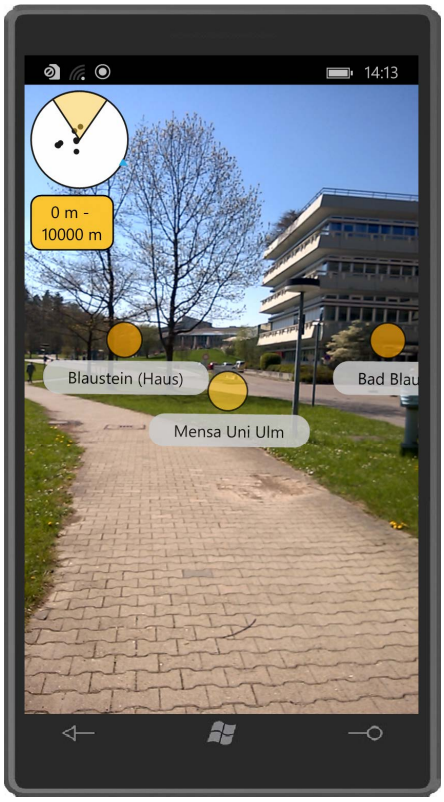

Abbildung 6.3: AREA: Augmented Reality Ansicht

Er besteht aus drei Teilen. Eine Ansicht der aktuellen Aufnahme der Kamera bildet die Basisebene und erstreckt sich über den gesamten Bildschirm. Zweiter Bestandteil sind die grafischen Zeichnungen von POIs, welche sich ebenfalls über den gesamten Bildschirm erstrecken und die Kameraansicht überdecken. Dritter Bestandteil ist der Radar, welcher den Platz in der oberen linken Ecke des Bildschirms beansprucht.

Die Basis des Radars bildet ein weißes, rundes Objekt. Auf diesem werden in Form von kleinen, schwarzen Punkten POIs in der Umgebung angezeigt. Der Radar dreht sich mit der Ausrichtung des Kompass, d.h. die obere Seite des Radars zeigt immer in die Richtung, in die der Nutzer blickt. Ebenfalls werden die POIs mitgedreht. POIs, die sich vor dem Nutzer befinden, werden auf der oberen Seite des Radars gezeichnet. POIs, die sich hinter dem Nutzer befinden, auf der unteren Seite. Der Mittelpunkt des Radars entspricht der Position des Nutzers. Die POIs werden in Verhältnis zur Entfernung des Nutzers auf dem Radar gezeichnet. Weiter entfernte Objekte bewegen sich auf dem Radar vom Mittelpunkt weg in Richtung äußerer Rand.

An der Außenseite des Radars befindet sich ein kleiner, dreieckiger Pfeil, der in Richtung geografischer Nordpol zeigt. Auch dieser dreht sich mit dem Blickwinkel des Nutzers. Ein weiteres Merkmal ist ein gelbes Blickfeld, in welchem erkennbar ist, welche Objekte sich abhängig vom Winkel der Kamera des Smartphones, momentan im Blickfeld des Nutzers befinden. Dieses Blickfeld gibt ebenfalls die Drehung des Bildschirms an, d.h. es zeigt immer in Richtung Horizont.

Unterhalb des Radars befindet sich ein Textfeld, welches den Minimalradius und den Maximalradius des Radars angibt. Ein Klick auf das Textfeld oder auf den Radar öffnet ein Dialogfenster, in welchem der Nutzer mittels zweier Schiebebalken die gewünschten Radien einstellen kann. Die Einstellung beeinflusst, welche POIs in der Umgebung des Nutzers auf dem Bildschirm und auf dem Radar angezeigt bzw. ausgeblendet werden. Es werden nur POIs gezeichnet, welche sich innerhalb der eingestellten Distanzen befinden. Das Dialogfeld dreht sich mit der Rotation des Bildschirms mit, kann also in jeder möglichen Ausrichtung verwendet werden (siehe Abbildung 6.4).

Abbildung 6.4: AREA: Änderung des minimalen und maximalen Radius

Die POIs selbst werden durch verschiedenfarbige Kreise auf dem Bildschirm angezeigt. Dabei befinden sich diese sowohl horizontal, als auch vertikal an der Stelle, an der sich das angezeigte Objekt in der Realität befindet. Falls der Nutzer sich dreht, so wird auch die Position eines POI verändert. Verschiedene Größen, wie auch Farben der Kreise geben die unterschiedliche Entfernung des POIs wieder. Umso weiter ein POI entfernt ist, umso kleiner wird es auf dem Bildschirm angezeigt, und umso stärker geht seine Farbe in den violetten Bereich über. Nahe Objekte werden hingegen in einem hellen Rot oder Gelbton angezeigt (siehe Abbildung 6.5).

Abbildung 6.5: AREA: Anzeige von unterschiedlichen Entfernungsstufen

Sollten sich POIs überlappen, d.h. verschieden entfernte POIs befinden sich in der selben Richtung, so wird ein POI, von zweien, oder mehreren jeweils nach oben verschoben. So ist es möglich, mit jeden POI zu interagieren und die Namen aller POI zu erkennen (siehe Abbildung 6.6).

Wie auch im Kartenmodus lässt sich ein Dialogfeld (siehe Abbildung 6.7) durch Klick auf einen POI anzeigen. In diesem befinden sich Felder mit weiteren Informationen, der Höhe und der Distanz des POIs. Auch dieses Dialogfeld lässt sich in jeder Bildschirmausrichtung verwenden.

Abbildung 6.6: AREA: Clusterbehandlung

Abbildung 6.7: AREA: Dialogfeld mit weiteren Informationen eines POI's

7

Abgleich der Anforderungen

Vergleicht man die Anforderungsanalyse mit dem in dieser Arbeit entstandenen Ergebnis, so kommt man zu dem Schluss, dass die meisten der Anforderungen erfüllt wurden.

Die Grundfunktionen, wie die Anzeige der POIs in der Karten- und Kameraansicht sind erfüllt. Ebenfalls wurden Performanceprobleme eliminiert und die Performance kann somit als sehr gut bezeichnet werden. Die Daten und Positionen werden in Echtzeit und zuverlässig aktualisiert. Es besteht die Möglichkeit, mit POIs zu interagieren und weitere Dialogfelder anzuzeigen. Diese lassen sich nicht nur im Portrait- und Landscapemodus verwenden, sondern in jeder beliebigen Lage des Bildschirms.

Die ausgelesenen Sensordaten sind präzise und zuverlässig. Jedoch mussten die Werte des Kompass manuell an die Bildschirmausrichtung angepasst werden, da das ausgelesene geografische Heading von der bereitgestellten Compass API je nach Bildschirmausrichtung verändert wird.

Bei der Bildung von Clustern werden die POIs übereinander angezeigt. Dies funktioniert in den meisten Fällen zuverlässig, verschlechtert bei vielen aufeinanderliegenden POIs jedoch etwas die Performance, da die Verschiebung der Position direkt in der GUI vorgenommen wird. Durch Vermeidung von unnötigen Berechnungen wurde die Performance verbessert.

Bei den nicht-funktionellen Anforderungen lässt sich eine sehr hohe Stabilität und Performance bei der Anzeige feststellen. Die Genauigkeit und die Effizienz der Berechnungen lässt sich als gut bezeichnen.

Die Implementierung achtet auf einheitliche Spezifikationen und Erweiterbarkeit. Eine Modularität zur Einbindung in andere Anwendungen ist gesichert. Ebenso ist eine verständliche Kommentierung und eine gute Wartbarkeit vorhanden.

Tabelle 7.1 gibt einen Überblick über die Anforderungen und stellt eine Bewertung der Erfüllung dieser Anforderungen dar. Die beste Bewertung stellt hier das "++" dar, die schlechteste Bewertung das "–".

Anforderung	**Bewertung**
POI auf Kameraansicht anzeigen	++
POI auf Kartenansicht anzeigen	++
POI in Kameraansicht nur anzeigen, wenn im Sichtfeld	++
POI in Kameraansicht und Kartenansicht sollen auf Interaktionen reagieren	++
Sensordaten zur Positionierung des Windows Phones auslesen (Beschleunigung, GPS, Magnetfeld)	+
Bei Bewegung des Smartphones, Daten und POI in Echtzeit aktualisieren	++
Einstellbarer minimaler und maximaler Radius für die Entfernung	++
Größe des angezeigten POI in Abhängigkeit zur Entfernung ändern	++
Vermeidung von Clusterbildung	+
Radar in Kameraansicht mit POIs aus der Umgebung und im Radius	++
Weitere Informationen bei Interaktion mit POI einblenden	++
Portrait und Landscape	++
Dialogfelder an Ausrichtung anpassen	++
Hohe Effizienz von Berechnungen	+
Hohe Effizienz beim Zeichnen der Anzeige	++
Hohe Stabilität / Hohe Genauigkeit / Gute Wartbarkeit	++ / + / +
Einheitliche Spezifikation & Erweiterbarkeit von POI	++
Einfache Einbindung in andere Anwendungen (Modularität)	++
Lückenlose, verständliche Kommentierung	+

Tabelle 7.1: Bewertung der Anforderungserfüllung

8

Zusammenfassung & Ausblick

Ziel dieser Arbeit war es, eine Augmented Reality Engine basierend auf der AREA App für iOS und für Android, für die Windows Plattform zu entwickeln.

Dabei sollten keine Teile bereits existierender Augmented Reality Engines verwendet werden. Es waren einige Herausforderungen zu bewältigen, die vor allem durch spezifische Eigenheiten der Windows Plattform entstanden. Die grundlegenden mathematischen Berechnungen konnten aus der bereits existierenden AREA Engine für iOS übernommen werden. Es mussten unter anderem Berechnungen in der Geographie durchgeführt werden. Dazu gehörten Formeln, wie die Distanzberechnung zweier Punkte oder der Kurs eines Punktes relativ zum Nordpol [6].

Um diese Berechnungen durchführen zu können, mussten die Daten der Sensoren des Smartphones abgefragt werden. Auf der Windows Phone Plattform handelt es sich hierbei um reine "Push"-Sensoren, d.h. die Sensoren melden Änderungen selbst. Dabei ist es wichtig, Änderungen in der GUI mit den Änderungen der Daten zu synchronisieren, da die Daten asynchron übermittelt werden. Außerdem musste für jeden Sensor jeweils eine geeignete Genauigkeit festgelegt werden, um unnötige Neuberechnungen in der GUI zu vermeiden.

Da die Ressourcen auf einem Smartphone sehr begrenzt sind, ist es allgemein wichtig, dass die Aktualisierung von GUI Komponenten möglichst ohne das erneute Zeichnen der Komponenten durchgeführt wird. Auf Grund dessen werden einmal gezeichnete POIs wiederverwendet und verschwinden nur, falls sie sich nicht mehr im Bereich der darübergelegten *LocationView* befinden.

Um eine hohe Modularität zu gewährleisten, wurde eine modulare Klassenstruktur für die AREA Anwendung auf Windows Phone verwendet. Bei dieser musste vor allem die Funktionsweise der Sensorenschnittstelle berücksichtigt werden. Die entstandene Engine lässt sich durch ihre Modularisierung leicht anpassen, erweitern oder in andere Apps integrieren. Die Funktionen zum Datenaustausch, wie *XML* und *JSON* wurden auf allen Plattformen auf ein einheitliches Schema

spezifiziert und bieten so eine gemeinsame Schnittstelle.

8.1 Ausblick

Bei der für Windows Phone entwickelten AR-Engine AREA gibt es noch an einigen Stellen Möglichkeiten zur Verbesserung oder Erweiterung.

Beim Ansprechen der Sensoren wurde direkt auf die Werte des Beschleunigungssensors und auf die des Kompasses zugegriffen. Möglich wäre es, die von Microsoft bereitgestellte kombinierte Motion API zu verwenden, welche es ermöglicht auch auf die Daten des Gyroskops zuzugreifen. Durch Nutzung der bereitgestellten Daten wäre es möglich, den Kompass zur Initialisierung zu verwenden und dann auf die bereitgestellte Rotationsmatrix der kombinierten Motion API zurückzugreifen [12]. Hiermit ließe sich insbesondere eine höhere Genauigkeit erreichen, wie auch vermeiden, dass der Kompass in bestimmten Umgebungen beginnt, unzuverlässige Daten und "springende" Werte zu liefern. Es muss jedoch beachtet werden, dass das Gyroskop auf der Windows Phone Plattform noch keine hohe Verbreitungsrate aufweist. Deshalb kann es nur als eine Alternative zum herkömmlichen Kompass verwendet werden.

Ein weiterer Punkt, den man noch optimieren könnte, wäre die Darstellung von übereinandergestapelten POIs. Diese werden durch die implementierte Clusterbehandlung zwar übereinander angezeigt, jedoch funktioniert dies nicht in allen Fällen. Zur Erkennung, ob zwei POIs sich überlappen, wird der Abstand zwischen den grafischen POI Punkten auf dem Bildschirm verwendet. Falls sich zwei POIs auf derselben Höhe befinden und horizontal verschoben sind, ein POI jedoch einen langen Namen hat, so können sich die Namen der POIs immer noch überlappen. Die korrekte Interaktion mit dem gewünschten POI ist – durch Klick auf den Punkt - dennoch gewährleistet.

Wie auch in der Version für iOS angemerkt [6], könnte man eine Art Richtungsweiser in der Augmented Reality Ansicht verwenden. Dieser würde bei leerem Bildschirm anzeigen, wo sich der nächste POI befindet. Da der Radar nur einen Blick aus der Vogelperspektive bietet, wäre dies insbesondere sinnvoll, falls sich ein POI über oder unter dem Sichtfeld befindet.

Literaturverzeichnis

[1] Artikel, Wikipedia: Erweiterte Realität. (Abgerufen am 06.04.2015). http://de.wikipedia.org/wiki/Erweiterte_Realität

[2] Crombach, A. ; Nandi, C. ; Bambonye, M. ; Liebrecht, M. ; Pryss, R. ; Reichert, M. ; Elbert, T. ; Weierstall, R.: Screening for mental disorders in post-conflict regions using computer apps - a feasibility study from Burundi. *XIII Congress of European Society of Traumatic Stress Studies* (2013)

[3] DAQRI: Anatomy 4D. (Abgerufen am 06.04.2015). http://daqri.com/project/anatomy-4d/

[4] Ehlert, R. ; Woiwode, G. ; Debus, J.: Windows Phone 8, Grundlagen und Praxis der App-Entwicklung. *dpunkt.verlag* (2013)

[5] Engadget: Google Glass learns how your friends dress, picks 'em out in a crowd. (Abgerufen am 06.04.2015). http://www.engadget.com/2013/03/08/google-glass-clothes-insight/

[6] Geiger, P.: Entwicklung einer Augmented Reality Engine am Beispiel des iOS. *Bachelorarbeit an der Universität Ulm* (2012)

[7] Geiger, P. ; Pryss, R. ; Schickler, M. ; Reichert, M.: Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices. *Technical Report. University of Ulm, Ulm* (2013)

[8] Geiger, P. ; Schickler, M. ; Pryss, R. ; Schobel, J. ; Reichert, M.: Location-based Mobile Augmented Reality Applications: Challenges, Examples, Lessons Learned. *Web Information Systems and Technologies - 10th International Conference* (2014)

[9] Google: Android API Guide. (Abgerufen am 02.05.2015). http://developer.android.com/guide/practices/screens_support.html

[10] Google: Translator. (Abgerufen am 06.04.2015). https://play.google.com/store/apps/details?id=com.google.android.apps.translate&hl=de

[11] Isele, D. ; Ruf-Leuschner, M. ; Pryss, R. ; Schauer, M. ; Reichert, M. ; Schobel, J. ; Schindler, A. ; Elbert, T.: Detecting adverse childhood experiences with a little help from tablet computers. *XIII Congress of European Society of Traumatic Stress Studies* (2013)

[12] Microsoft: How to use the combined Motion API for Windows Phone 8. (Abgerufen am 06.04.2015). https://msdn.microsoft.com/en-us/library/windows/apps/hh202984%28v=vs.105%29.aspx#BKMK_

CreatingaSilverlightbasedAugmentedRealityApplication

[13] Müller, J.: Konzeption und prototypische Implementierung eines Verfahrens zur POI Clusterbehandlung innerhalb einer Augmented Reality Anwendung. *Bachelorarbeit an der Universität Ulm* (2014)

[14] MSDN: Multi-resolution apps for Windows Phone 8. (Abgerufen am 02.05.2015). https://msdn.microsoft.com/en-us/library/windows/apps/jj206974%28v=vs.105%29.aspx

[15] MSDN: XAML concepts for Windows Phone 8. (Abgerufen am 06.05.2015). https://msdn.microsoft.com/en-us/library/windows/apps/jj206948%28v=vs.105%29.aspx

[16] MSDN: Supported Direct3D APIs for Windows Phone 8. (Abgerufen am 08.05.2015). https://msdn.microsoft.com/en-us/library/windows/apps/jj207010%28v=vs.105%29.aspx

[17] Noelle, S.: Stereo augmentation of simulation results on a projection wall by combining two basic ARVIKA systems. *Proceedings International Symposium On Mixed And Augmented Reality* (2002)

[18] Open-Signal: Fragmentation of Android devices. (Abgerufen am 02.05.2015). http://opensignal.com/reports/fragmentation.php

[19] Readwrite: Military-Grade Augmented Reality Could Redefine Modern Warfare. (Abgerufen am 06.04.2015). http://readwrite.com/2010/06/11/military_grade_augmented_reality_could_redefine_modern_warfare

[20] Ruf-Leuschner, M. ; Pryss, R. ; Liebrecht, M. ; Schobel, J. ; Spyridou, A. ; Reichert, M. ; Schauer, M.: Preventing further trauma: KINDEX mum screen - assessing and reacting towards psychosocial risk factors in pregnant women with the help of smartphone technologies. *XIII Congress of European Society of Traumatic Stress Studies* (2013)

[21] Schickler, M. ; Pryss, R. ; Schobel, J. ; Reichert, M.: An Engine Enabling Location-based Mobile Augmented Reality Applications. *Web Information Systems and Technologies - 10th International Conference* (2014)

[22] Schobel, J. ; Pryss, R. ; Reichert, M.: Using Smart Mobile Devices for Collecting Structured Data in Clinical Trials: Results From a Large-Scale Case Study. *28th IEEE International Symposium on Computer-Based Medical Systems* (2015)

[23] Schobel, J. ; Ruf-Leuschner, M. ; Pryss, R. ; Reichert, M. ; Schickler, M. ; Schauer, M. ; Weierstall, R. ; Isele, D. ; Nandi, C. ; Elbert, T.: A generic questionnaire framework supporting psychological studies with smartphone technologies. *XIII Congress of European Society of Traumatic Stress Studies* (2013)

[24] Schobel, J. ; Schickler, M. ; Pryss, R. ; Maier, F. ; Reichert, M.: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. *10th Int'l Conference on Web Information Systems and Technologies* (2014)

[25] Schobel, J. ; Schickler, M. ; Pryss, R. ; Nienhaus, H. ; Reichert, M.: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. *Web Information Systems and Technologies - 9th International Conference* (2013)

[26] Schobel, J. ; Schickler, M. ; Pryss, R. ; Reichert, M.: Process-Driven Data Collection with Smart Mobile Devices. *Web Information Systems and Technologies - 10th International Conference* (2015)

[27] Search-Engine-Watch: Mobile Now Exceeds PC: The Biggest Shift Since the Internet Began. (Abgerufen am 06.05.2015). `http://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began`

[28] Statista: Marktanteile der mobilen Betriebssysteme am Absatz von Smartphones in Deutschland von Januar 2012 bis März 2015. (Abgerufen am 10.05.2015). `http://de.statista.com/statistik/daten/studie/225381/umfrage/marktanteile-der-betriebssysteme-am-smartphone-absatz/in-deutschland-zeitreihe`

[29] Windows-Phone-Store: HERE Maps. (Abgerufen am 06.04.2015). `http://www.windowsphone.com/de-de/store/app/here-maps/efa4b4a7-7499-46ce-aa95-3e4ab3b39313`

Abbildungsverzeichnis

Listings

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sinngemäße Übernahmen aus anderen Werken sind als solche kenntlich gemacht und mit genauer Quellenangabe (auch aus elektronischen Medien) versehen.

Ulm, den 15.05.2015 Robin Bird