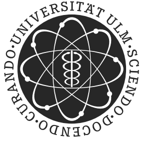


ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Entwicklung einer Augmented Reality Engine am Beispiel des iOS

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Philip Geiger
philip.geiger@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss

2012

„Entwicklung einer Augmented Reality Engine am Beispiel des iOS“
Fassung vom 25. September 2012

© 2012 Philip Geiger

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung 1

1.2 Aufbau der Arbeit 2

2 Augmented Reality **5**

2.1 Funktionsweise 5

2.2 Tracking 6

2.3 Augmented Reality in dieser Arbeit 7

3 Anforderungsanalyse **9**

3.1 Anforderungen an die Augmented Reality Engine 9

3.2 Funktionen existierender Engines 12

4 Mathematische Grundlagen **15**

4.1 Berechnung der Distanz 15

4.2 Berechnung der Himmelsrichtung 18

4.3 Berechnung des Höhenwinkels 19

4.4 Berechnung des Sichtfelds 20

5 Entwurf **21**

5.1 Archtitekturentwurf 21

5.2 Klassenstruktur 23

5.3 Kommunikationsablauf 25

5.4 Datenhaltung 26

6 Implementierung **31**

6.1 Manipulation der Kameraansicht 31

6.2 Sensoren und ihre Daten 35

6.3 Berechnungen im Controller 38
6.3.1 Berechnen von POI in der Umgebung 38
6.3.2 Berechnung des Sichtfelds 41
6.3.3 Platzierung von POI auf der Kameraansicht 43

7 Vorstellung der Applikation **47**

8 Zusammenfassung **51**

9 Ausblick **53**

A Skizzen **55**

Literaturverzeichnis **57**

1 Einleitung

Moderne Smartphones sind mittlerweile so leistungsstark, dass auf ihnen auch aufwendige Anwendungen ausgeführt werden können, wie es vorher nur auf Heimrechnern möglich war. Augmented Reality ist eine dieser Anwendungen. Diese benötigt einiges an Ressourcen, um zur Laufzeit Sensoren und GPS-Daten abzufragen. Gleichzeitig müssen in Echtzeit virtuelle Objekte auf dem Display mit einer hohen Anzahl von Bildern pro Sekunde angezeigt werden. Einige Entwickler haben sich diese Tatsache bereits zum Nutzen gemacht und Augmented Reality Engines und Applikationen für die unterschiedlichen Smartphones und Betriebssysteme wie Android und iOS entwickelt. Darunter existieren einerseits proprietäre kommerzielle, andererseits frei zugängliche OpenSource-Engines und Applikationen. Bei ersteren ist es wegen des unzugänglichen Programmcodes, bei letzteren meist wegen mangelnder Kommentierung nicht möglich, tiefere Einblicke in die Funktionsweise zu erhalten oder gar Anpassungen an diesen vorzunehmen. Aus diesem Grund wird in dieser Arbeit eine Augmented Reality Engine von Grund auf erstellt und dessen Funktionsweise erläutert.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist es, eine komplette Augmented Reality Engine für das *iPhone 4S*, basierend auf *iOS 5.1*, ohne Zuhilfenahme von Teilen bereits existierender zu entwickeln. Diese soll es ermöglichen, Interessenspunkte aus der Umgebung des Benutzers anhand seines Blickwinkels, seiner Ausrichtung und Position auf der Kamera des iPhones anzuzeigen (Abbildung 1.1). D.h., die durch die iPhone-Kamera aufgenommene Abbildung der Realität wird mit virtuellen Objekten, welche die Interessenspunkte relativ zu deren GPS-Koordinaten darstellen, erweitert. Ferner soll die Engine leicht anpassbar, in andere Anwendungen einbindbar und durch eine gute Kommentierung verständlich sein.

Die Entwicklung einer solchen Augmented Reality Engine ist gleichermaßen spannend wie komplex. Um das Bild, das durch die iPhone-Kamera aufgenommen wird, mit virtuellen Informationen zu Interessenspunkten aus der Umgebung erweitern zu können, müssen einige grundlegende Dinge über geographische Berechnungen bekannt sein. So muss unter anderem eine effiziente und verlässliche Methode gefunden werden, sodass beispielsweise die Entfernung zwischen zwei solchen Interessenspunkten anhand von GPS-Daten berechnet werden kann. Es müssen zahlreiche Sensoren zur Berechnung der Ausrichtung und Lage des iPhones bzw. des Benutzers, wie z.B. ein Beschleunigungssensor, korrekt ausgelesen werden. Weiter ist es notwendig, den Blickwinkel der Linse der iPhone-Kamera zu berechnen, um die virtuellen Objekte korrekt dem Abbild der Realität einsetzen zu können.

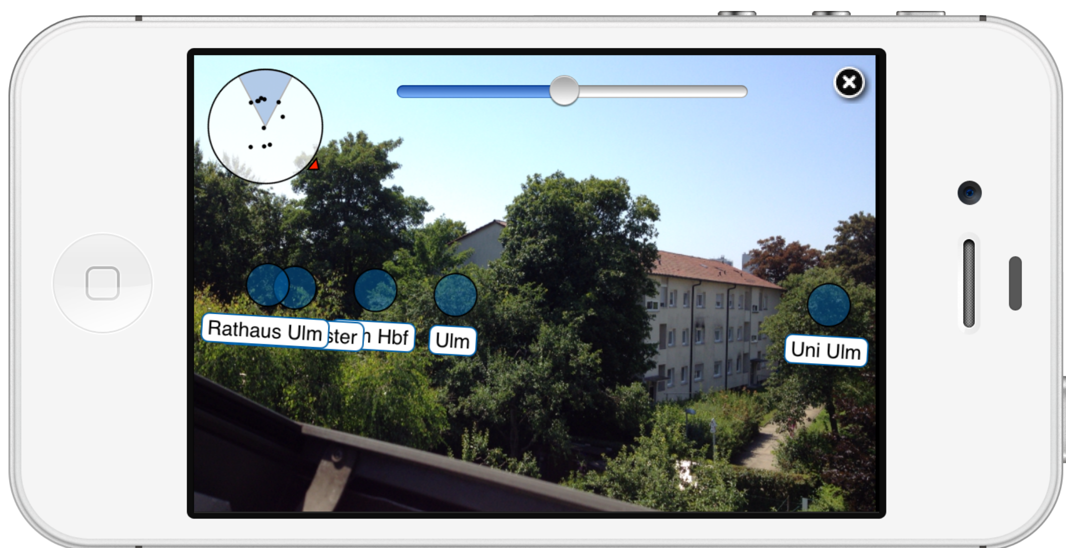


Abbildung 1.1: Ein Screenshot der fertigen Augmented Reality Engine, die in dieser Arbeit entwickelt wird. Zu sehen sind die Interessenspunkte, ein Radar und ein Regler zum Bestimmen des sichtbaren Radius.

1.2 Aufbau der Arbeit

Bevor mit der eigentlichen Entwicklung der Augmented Reality Engine begonnen wird, werden die unterschiedlichen Arten von Augmented Reality beschrieben. Danach werden die Anforderungen an die Engine gestellt und mit bereits existierenden kommerziellen oder

freien verglichen. Die Entwicklung beginnt dann damit, die mathematischen Formeln und Herangehensweisen aus der Geographie zu sammeln und zu untersuchen. Diese werden später in der Engine umgesetzt. Auf den Entwurf, in dem die grundlegende Architektur und die einzelnen Komponenten der Engine festgelegt werden, sodass diese so modular wie möglich ist, folgt die Implementierung. Dabei wird weiter auf die zuvor gelisteten Formeln der Geographie und spezifische Kommunikationsmuster der Engine anhand der Programmiersprache Objective-C eingegangen. Zuletzt wird die fertige Engine durch Anwendungsbeispiele und Screenshots vorgestellt.

2 Augmented Reality

In diesem Kapitel werden die Eigenschaften von Augmented Reality beschrieben. Hierzu gehört die generelle Funktionsweise und das sog. Tracking, das für die Bestimmung der Position des Benutzers oder des Gerätes, auf dem eine Augmented Reality Anwendung läuft, zuständig ist. Weiter werden die spezifischen Eigenschafen der Augmented Reality Engine dargestellt, die in dieser Arbeit für das iPhone anhand des iOS entwickelt wird.

2.1 Funktionsweise

Augmented Reality („Erweiterte Realität", kurz: AR) ist das Hinzufügen von zusätzlichen viruellen Informationen in das eigentliche Sichtfeld eines Benutzers. Solche Informationen bestehen meist aus Text-, 2D- oder 3D-Objekten und werden in die reale Umgebung eingebunden. Wichtig ist dabei, dass die hinzugefügten Objekte die reale Umgebung nur erweitern, nicht aber das komplette Sichtfeld belegen, d.h. die reale Umgebung weiterhin sichtbar bleibt und dadurch eine Vermischung zwischen Realität und virtuellen Objekten entstehen kann [1].

Die Funktionsweise von AR-Anwendungen lässt sich anhand von Endgeräten klassifizieren. *Head-worn displays* werden vom Benutzer aufgesetzt und die zusätzlichen Informationen werdem diesem direkt vor dem Auge eingeblendet. Ein Beispiel für ein head-worn display ist Google Glasses (Abbildung 2.1), eine Datenbrille von Google [2]. Ähnlich funktionieren auch die *projection displays*. Diese sind unter anderem aus Kampfflugzeugen (Abbildung 2.1) oder neuerdings aus Oberklasseautomobilen bekannt und projizieren Informationen wie Geschwindigkeit, Routenplanung oder Himmelsrichtung auf eine Scheibe vor dem Benutzer, ohne dass dieser seinen Blick von der eigentlichen Tätigkeit abwenden muss. Ein weiteres geeignetes Endgerät für AR-Anwendungen ist das Handheld, bzw. das Smartphone, und ist Hauptthema dieser Arbeit. Bei Smartphone-Anwendungen wird

nicht wie bei oben genannten Endgeräten das direkte Sichtfeld des Benutzers, sondern das durch eine interne Kamera aufgenommene Bild erweitert [1].

(a) Head-Up-Display eines Kampfflugzeugs

(b) Head-Worn-Display von Google

Abbildung 2.1: Zwei Beispiele für Augmented Reality Endgeräte

2.2 Tracking

AR-Anwendungen benötigen zum Erweitern des Sichtfeldes Daten aus der realen Umgebung des Benutzers. Hierzu gehören je nach Anwendung z.B. die Geschwindigkeit, die Ausrichtung, die Entfernung oder die Position des Endgeräts, des Benutzers oder von Objekten aus dieser Umgebung. Soll die Anwendung zusätzlich Informationen zu expliziten Objekten aus der realen Umgebung ausgeben, so müssen diese Objekte durch unterschiedliche Mechanismen erkennbar oder lokalisierbar gemacht werden. Im Wesentlichen gibt es für diese Aufgaben zwei Möglichkeiten, die im Folgenden voneinander differenziert werden [3].

Vision-Based Tracking Diese Art von Tracking eignet sich vorallem für das Erkennen von Objekten aus der realen Umgebung. Hierzu werden Mechanismen und Algorithmen aus der Computer-Vision verwendet, um Objekte aus der realen Umgebung zu identifizieren und für das Endgerät erkennbar zu machen. Das kann zum einen durch eindeutige Markierungen auf Objekten geschehen, welche durch die Kamera gescannt werden oder durch eine Art Echtzeitscanning, wobei das aufgenommene Bild auf markante Eigenschaften untersucht und mit Datenbanken abgeglichen wird.

Sensor-Based Tracking Hierbei werden die Informationen des Endgeräts bzw. des Benutzers und der Objekte aus der realen Umgebung anhand von GPS-Daten, Magnetfeldsensoren, Beschleunigungssensoren und Gyroskopen[1] des Endgeräts bestimmt. Anders als beim Vision-Based Tracking ist es möglich, beliebe Objekte anhand von GPS-Daten erkennen zu können, ohne dass das Bild auf Marker untersucht oder mit Datenbanken abgeglichen werden muss. Die AR-Engine, die in dieser Arbeit entwickelt wird, basiert vollständig auf Sensor-Based Tracking.

2.3 Augmented Reality in dieser Arbeit

Das Ziel dieser Arbeit ist es, eine AR-Engine für ein Smartphone zu entwickeln, mit der es möglich sein soll, beliebige Interessenspunkte aus der Umgebung auf dem Display, je nach Position des Gerätes bzw. des Benutzers und anhand von GPS-Daten, anzuzeigen. Diese basiert, wegen der Notwendigkeit möglichst viele Sensordaten über die Position und die Lage von Gerät und Interessenspunkt zu erhalten, auf Sensor-Based Tracking. Da sie für ein Smartphone entwickelt werden soll, wird nicht das eigentliche reale Sichtfeld des Benutzers, sondern eine Aufnahme des Sichtfeldes durch virtuelle Informationen erweitert.

[1] Ein Gyroskop ist ein sich schnell drehender Kreisel. Anhand des Trägheitsgesetzes und physikalischer Berechnungen kann damit ausgelesen werden, um wieviel sich ein Gerät, in Relation zur letzten Messung, in welche Richtung gedreht hat [4].

3 Anforderungsanalyse

Das Ziel dieser Arbeit ist es, eine AR-Engine zu entwickeln. Mit dieser soll es möglich sein, Interessenspunkte aus der Umgebung des Benutzers anzeigen zu lassen. Im Folgenden werden die Anforderungen an die AR-Engine aufgestellt und mit dem Funktionsumfang kommerzieller und freier verglichen.

3.1 Anforderungen an die Augmented Reality Engine

Die AR-Engine soll Interessenspunkte (im Folgenden *Points of Interest*, kurz: POI) relativ zur momentanen Position des Benutzer auf dem Display in einer Kameraansicht anzeigen. Hierzu müssen sowohl die GPS-Koordinaten des iPhones als auch die des POI bekannt sein. Es sollen nur die POI auf dem Display angezeigt werden, die dem Sichtfeld des Benutzers oder in diesem Fall dem Sichtfeld der Kamera des iPhones entsprechen. Deshalb ist es notwendig, die Höhe von iPhone und POI, den Blickwinkel relativ zum Nordpol, und die Haltung, d.h. Neigung und Rotation anhand der Achsen des iPhones, zu bestimmen. Das Auslesen dieser Eigenschaften muss in Echtzeit und mit einer hohen Genauigkeit während möglichen Bewegungen des Benutzers bzw. des iPhones geschehen. Dabei darf weder die Effizienz, noch die Stabilität vernachlässigt werden. Der AR-Engine soll es möglich sein, einen Radius durch einen Schieberegler festzulegen, bis zu dessen Grenze POI auf dem Display angezeigt werden. D.h. POI, die jenseits des Radius liegen, werden von der Anzeige ignoriert. Um alle POI aus der Umgebung im Radius unabhängig des Blickwinkels einsehen zu können, soll ein Radar mitgeliefert werden. Sowohl unabhängig vom Radius als auch vom Blickwinkel soll eine Kartenansicht die Position des Benutzers und alle POI der Umgebung anzeigen. Alle sichtbaren POI sollen in der Kameraansicht und der Kartenansicht auf Interaktionen wie Touch-Events reagieren und zusätzliche Informationen einblenden können. Es sollen Portrait- und Landscapemodus des iPhones unterstützt werden. In Abbildung 3.1 wird nochmals veranschaulicht dargestellt, welche POI in der Kameraansicht angezeigt werden sollen und welche nicht.

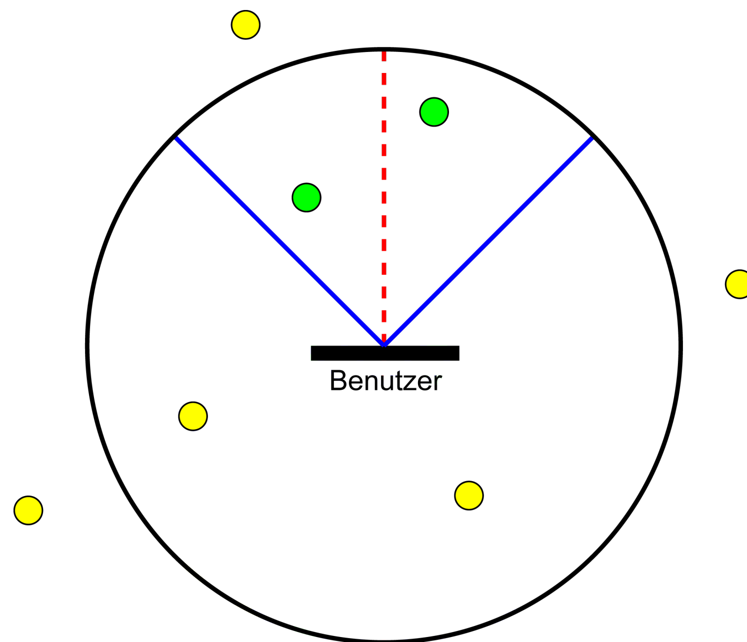


Abbildung 3.1: Schematische Darstellung zur Anzeige von POI. Zu sehen sind der einstellbare Radius (rot), das Sichtfeld (blau), POI die sowohl im Sichtfeld als auch innerhalb des Radius sind (grün), und Punkte die nicht auf der Kameraansicht angezeigt werden (gelb).

Die AR-Engine muss ohne größeren Aufwand in andere Anwendungen integriert werden können. Deshalb müssen öffentliche Schnittstellen angeboten und eine hohe Modularität erreicht werden. Dazu gehört vor allem eine einheitliche einfache Spezifikation von POI, sodass diese bei Bedarf statisch oder dymanisch hinzugefügt und entfernt werden können. Intern müssen die POI so realisiert werden, dass eine Erweiterung der Struktur keinen negativen Einfluss auf die Funktionalität hat. Eine lückenlose und verständliche Kommentieren des Programmcodes ist wichtig für zukünftige Wartungen, Erweiterungen und zum Verstehen der Funktionsweise der AR-Engine.

Die gesamten Anforderungen werden in Tabelle 3.1 nochmals strukturiert aufgelistet und verschiedenen Typen von Anforderungen zugeordnet.

Tabelle 3.1: Tabellarische Aufstellung der Anforderungen

Anforderung	Typ
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 iPhones auslesen (Beschleunigung, GPS, Magnetfeld)	funktional
Bei Bewegung des iPhones Daten und POI in Echtzeit aktualisieren	funktional
Einstellbarer Radius für die Entfernung	funktional
Radar in Kameraansicht mit POI aus der Umgebung und im Radius	funktional
Weitere Informationen bei Interaktion mit POI einblenden	funktional
Portrait und Landscape	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
POI sollen intern erweiterbar sein	Implementierung
Einfache Einbindung in andere Anwendungen (Modularität)	Implementierung
Lückenlose verständliche Kommentierung	Dokumentation

3.2 Funktionen existierender Engines

Auf dem Markt existieren bereits einige AR-Engines und Applikationen. Da die hier entwickelte Engine den existierenden in so wenig Funktionen wie möglich nachstehen soll, werden sie im Zusammenhang mit dieser Arbeit untersucht.

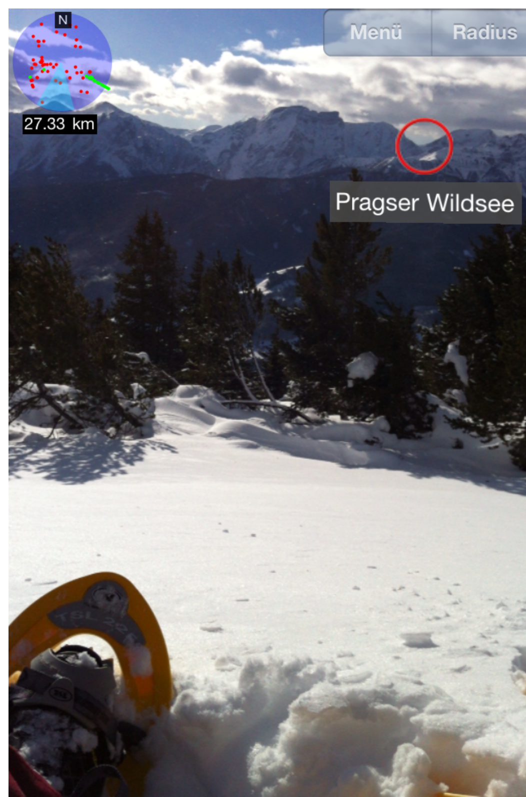


Abbildung 3.2: Screenshot von mixare

Eine weitverbreitete OpenSource-Engine ist *mixare* [5] (Abbildung 3.2). Diese steht sowohl für das iOS als auch für Android zur Verfügung. Die grundlegenden Funktionen, wie das Anzeigen von POI, dem Einstellen eines Radius und das Einsehen eines Radars, werden angeboten. Weiter kann eine Kartenansicht mit allen POI aus der Umgebung aufgerufen werden. Die POI in Kartenansicht und Kameraansicht können auf Touch-Interaktionen reagieren. *Wikitude* [6] und das mitgelieferte *ARchitect SDK* bieten im Wesentlichen die gleichen Grundfunktionen. Dasselbe gilt ebenfalls für die Applikation *Yelp* [7], eine Applikation die Restaurants und Cafés aus der Umgebung anzeigt. Alle drei Produkte erlauben es aber,

zusätzlich POI zu kategorisieren, neue hinzuzufügen und auf APIs, wie die von Twitter [8] und Google [9] zuzugreifen, um POI dynamisch je nach Position des Benutzers zu laden.

Weiteres zu fehlenden bzw. erweiterbaren Funktionalitäten wird im Ausblick am Ende der Arbeit erwähnt. Wie solche Funktionalitäten, anhand einer OpenSource-Engine und des Android Frameworks, erweitert und implementiert werden können, kann in einer bereits existierenden Bachelorarbeit nachgelesen werden [10].

4 Mathematische Grundlagen

In diesem Kapitel werden die mathematischen Grundlagen, die zur Entwicklung einer AR-Engine notwendig sind, vorgestellt. Hierzu gehört das Ermitteln der Distanz zwischen der Position des Benutzers und des POI anhand von GPS-Koordinaten. Weiter muss ein Weg gefunden werden, auszurechnen, in welcher Himmelsrichtung ein POI relativ zum Benutzer und dessen Sichtfeld liegt. Und es muss der Höhenunterschied zwischen Benutzer und POI berechnet werden, um auch hier bestimmen zu können, ob der POI im Sichtfeld liegt. Das Sichtfeld ist wiederum abhängig von den Brennweite der iPhone-Kamera.

4.1 Berechnung der Distanz

Great-circle Distance (im Deutschen meist *Orthodrome*) ist ein Verfahren aus der sphärischen Geometrie, einem Teilgebiet der Geometrie, um Entfernungen zwischen zwei Punkten auf einer gewölbten Oberfläche, wie einer Kugel, zu berechnen. Anders als eventuell zu erwarten, wird hierbei nicht die Entfernung anhand einer Geraden durch die Kugel gemessen, sondern die Entfernung entlang der Oberfläche. Formel 4.1 ist die erste Möglichkeit zur Berechnung dieser Entfernung.

$$\theta = \arccos(\sin\phi_A \sin\phi_B + \cos\phi_A \cos\phi_B \cos(\Delta\lambda)) \tag{4.1}$$

$$D = \theta * 6371km \tag{4.2}$$

Dabei ist A die Position des Benutzers und B die des POI, ϕ ist der Breitengrad, λ der Längengrad einer dieser Positionen, $\Delta\lambda$ entspricht $\lambda_B - \lambda_A$ und ist damit die Differenz zwischen den beiden Längengraden. Um den Winkel θ zwischen den beiden Punkten zu

berechnen, müssen ϕ, λ und $\Delta\lambda$ in Radiant[2] vorliegen. Für die Distanz D muss θ, ebenfalls in Radiant, noch mit dem Radius R der Sphäre, in diesem Fall dem Radius der Erde, multipliziert werden (Formel 4.2).

Obrige Formel 4.1 hat jedoch einen Nachteil. Wenn die beiden Punkte sehr nah beieinanderliegen, kommt es beim Ausführen auf einem Computer zu Rundungsfehlern. Dies liegt daran, dass der Wert der Klammer dann sehr nah an der 1 liegt, z.B. $0,99999$, und folglich beim Ausrechnen von $\arccos$ eine numerische Instabilität eintritt. Um dem Abhilfe zu schaffen, wurde die *Haversine*-Formel (Formel 4.3) entwickelt, welche eine bessere numerische Stabilität aufweist [12] und damit den Anforderungen der AR-Engine genügt.

$$\theta = 2 \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_A \cos\phi_B \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \tag{4.3}$$

Das Problem tritt jedoch immer noch auf, wenn die Punkte beinahe gegenüber auf der Kugel liegen [13]. Da solch große Distanzen in der AR-Engine jedoch nicht vorkommen werden, wird Formel 4.3 im Weiteren zur Berechnung der Distanz zweier Punkte herangezogen. Um nun die eigentliche Distanz zu berechnen, wird θ aus 4.3 in Formel 4.2 eingesetzt. Listing 4.1 zeigt den Unterschied zwischen den beiden Formeln anhand eines C-Programms. Der Wert vom Winkel θ liegt hierbei so nah an der Null, da die beiden Punkte sehr nah beieinanderliegen, dass er im Prozessor auf den Wert `1.000000` gerundet wird und dadurch mit Great-circle Distance eine ungenau Distanz berechnet wird.

[2] Der Radiant ist eine Einheit aus der Mathematik. Sie bestimmt das Verhältnis zwischen einer Bogenlänge und dessen Radius und wird anhand des Einheitskreises definiert. Die Umrechnung in das Bogenmaß, also einen Winkel in Grad, geschieht durch $Radiant * \frac{180°}{\pi}$. [11]

```c
#include <stdio.h>
#include <math.h>

#define toRad(x) ((x)*M_PI/180.0)

int main()
{
    // location one
    double lat1 = 48.4042981;
    double lon1 = 9.979349;

    // location two
    double lat2 = 48.40429881;
    double lon2 = 9.979349;

    double dLat = toRad(lat2 - lat1);
    double dLon = toRad(lon2 - lon1);
    double radius = 6371;

    // Great Circle Distance
    double angle = sin(toRad(lat1)) * sin(toRad(lat2)) + cos(toRad(lat1)) * cos(
        toRad(lat2)) * cos(dLon);
    double great_circle_distance = acos(angle) * radius;

    // Haversine Formula
    double res1 = pow(sin(dLat/2), 2) + cos(toRad(lat1)) * cos(toRad(lat2)) * pow(
        sin(dLon/2), 2);
    double haversine = 2*asin(sqrt(res1)) * 6371;

    printf("Angle: \t\t %f \nGreat Circle:\t %f km\nHaversine:\t %f km\n\n", angle,
         great_circle_distance, haversine);

    return 0;
}

Output:
Angle:          1.000000
Great Circle:   0.000134 km
Haversine:      0.000079 km
```

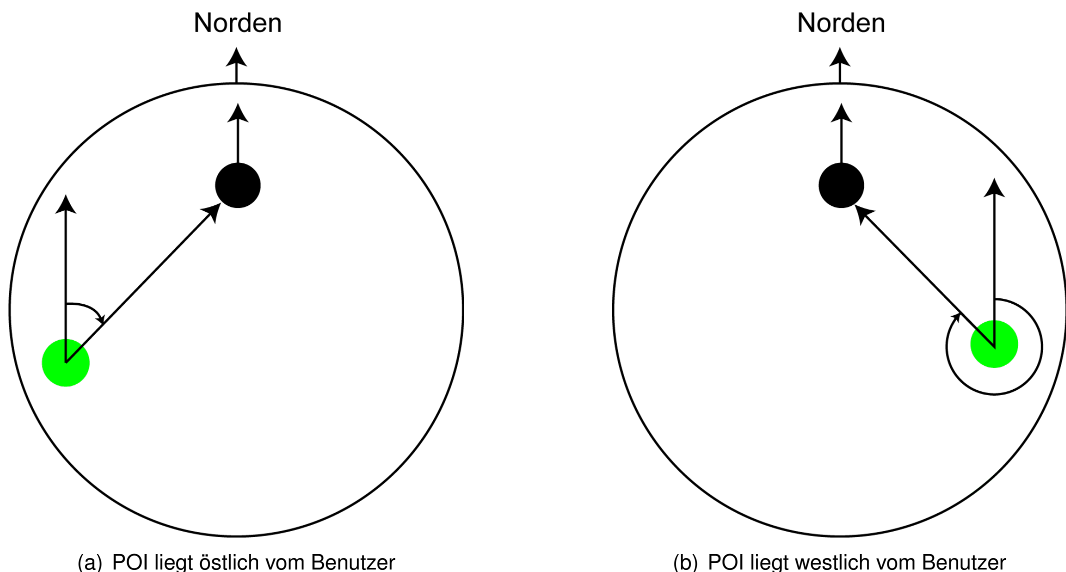
Listing 4.1: Vergleich zwischen Haversine und Great-circle distance

4.2 Berechnung der Himmelsrichtung

Auf dem Display des IPhones sollen in der Kameraansicht nur die POI angezeigt werden, die sich im Sichtfeld des Benutzers befinden. Dafür muss der Kurs zwischen der Position des Benutzers und der des POI relativ zum Nordpol berechnet werden (Abbildung 4.1). Formel 4.4 wird angewendet, um diesen Kurs zu berechnen [12][14].

$$\theta = \arctan 2(sin(\Delta\lambda)\cos\phi_B, \cos\phi_A\cos\phi_B - \sin\phi_A\cos\phi_B\cos(\Delta\lambda)) \quad (4.4)$$

Die Bezeichner A, B, ϕ und λ haben die gleichen Bedeutungen wie in Formel 4.1. Da das Ergebnis, θ umgewandelt ins Bogenmaß, das Intervall zwischen $-180°$ und $+180°$ abbildet, kann das Ergebnis noch auf das Intervall $0°...360°$ durch $(\theta + 360°)\ mod\ 360°$ transformiert werden. Durch das Ergebnis ist es später möglich, in Verbindung mit dem Kompass des iPhones, zu entscheiden, ob sich der POI im *horizontalen Sichtfeld* befindet und auf das Display gezeichnet werden muss.



(a) POI liegt östlich vom Benutzer

(b) POI liegt westlich vom Benutzer

Abbildung 4.1: Schematische Darstellung zur Berechnung des Kurses. Der POI ist in (a) und (b) auf gleichem Längen- und Breitengrad. Die Position des Benutzers ist jedoch unterschiedlich, sodass sich der Kurs vom POI in (a) und (b) unterscheidet.

4.3 Berechnung des Höhenwinkels

Das Sichtfeld der iPhone-Kamera ist nicht nur in der Breite begrenzt, sondern auch in der Höhe. Es müssen also ebenfalls die Höhenunterschiede zwischen dem Benutzer und den POI berechnet werden, um damit entscheiden zu können, ob sich der POI im *vertikalen Sichtfeld* befindet. Dazu wird eine Herangehensweise benötigt, die einen Winkel im Bogenmaß zwischen $-90°$ und $+90°$ als Ergebnis liefert. Dadurch kann, wie später gezeigt wird, im Zusammenhang mit der Neigung des iPhones ermittelt werden, welcher Bereich auf dem Display abgebildet werden kann. Abbildung 4.2 veranschaulicht die Formel 4.5 zur Berechnung dieses Winkels.

$$\theta = \sigma * \frac{180}{\pi} \arctan\left(\frac{\Delta h}{d}\right), \sigma \in \{-1, 1\} \tag{4.5}$$

Dabei ist Δh der Höhenunterschied zwischen der Position des Benutzers und der des POI, d die Distanz zwischen beiden. Zu beachten ist, dass σ von diesem Höhenunterschied abhängig ist. Es ist $\sigma = 1$, wenn der POI höher liegt als der Benutzer, $\sigma = -1$ sonst. Durch die Berechnung von $\arctan\left(\frac{\Delta h}{d}\right)$ wird der Winkel in Radiant zwischen Hypothenuse und Ankathete (s. Abbildung 4.2) berechnet. Dieser muss letztlich noch in das Bogenmaß umgerechnet und mit σ multipliziert werden. Aus dieser Formel geht dann ein Höhenwinkel aus dem Intervall $0°...90°$ hervor, wenn der POI über dem Benutzer liegt, aus $-90°...0°$, wenn dieser unter dem Benutzer liegt.

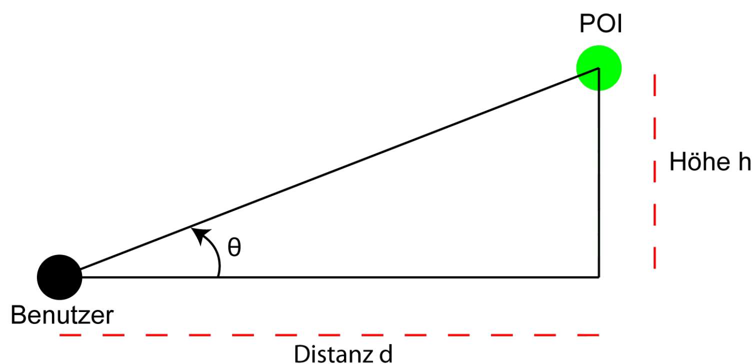


Abbildung 4.2: Veranschaulichung von Formel 4.5 zur Berechnung des Höhenwinkels zwischen Benutzer und POI

4.4 Berechnung des Sichtfelds

Bis zu diesem Punkt kann anhand der Formeln entschieden werden, ob sich ein POI im horizontalen und vertikalen Sichtfeld befindet. Es ist aber noch nicht bekannt, wie sich die Größe des Sichtfelds vom iPhone und dessen Kamera bestimmen lässt. Berechnet werden kann das durch Formel 4.6 [15].

$$\alpha = 2 \arctan\left(\frac{B}{2f}\right) \tag{4.6}$$

B ist die Bildgröße, genauer die Größe des *Bildsensors*, f die Brennweite der Kameralinse. Leider werden weder in den Spezifikationen des iPhones, noch auf einer offiziellen Seite von Apple, Informationen über diese Parameter geliefert. Durch eine intensive Onlinerecherche konnten der Bildsensor mit eine Größe von $4,592$ x$3,45\ mm^2$ und die Brennweite der Linse auf $4,28\ mm$ ausgemacht werden [16, 17, 18, 19].

$$2 \arctan\left(\frac{4,592}{2 * 4,28}\right) * \frac{180}{\pi} \approx 56,4225 \tag{4.7}$$

$$2 \arctan\left(\frac{3,45}{2 * 4,28}\right) * \frac{180}{\pi} \approx 43.9026 \tag{4.8}$$

Daraus folgt, sofern sich das iPhone im Landscapemodus (dabei sind die längeren Seiten oben und unten) befindet, ein horizontales Sichtfeld von gerundet $56°$ (Rechnung 4.7) und ein vertikales von gerundet $44°$ (Rechnung 4.8) aufweist.

Nun ist einerseits die Größe des Sichtfeldes der iPhone-Kamera bekannt, andererseits kann anhand obiger Formeln entschieden werden, ob sich ein POI im vertikalen Sichtfeld, im horizontalen Sichtfeld und in welcher Entfernung zum Benutzer befindet. Diese Berechnungen werden eine zentrale Rolle bei der Implementierung und der Funktionsweise der AR-Engine übernehmen.

5 Entwurf

In diesem Kapitel wird der Entwurf der Augmented Reality Engine Application (kurz: *AREA*) vorgestellt. Dabei wird ein generelle Architekturentwurf entwickelt, ein Klassendiagramm entworfen und die Kommunikation der einzelnen Komponenten angesprochen. Da eine einheitliche Schnittstelle für die POI angeboten werden muss, wird ebenfalls die Datenhaltung genauer betrachtet.

5.1 Archtitekturentwurf

Die Architektur von AREA besteht im Wesentlichen aus drei großen Komponenten. Ein Controller verwendet die im vorherigen Kapitel vorgestellten mathematischen Formeln. Er ist zum einen für das Auslesen der Sensoren des iPhones zuständig. Dazu gehört das Bestimmen der momentanen Positionen des Benutzers über den GPS-Sensor und das Ermitteln der horizontalen Blickrichtung über Kompassdaten. Die vertikale Blickrichtung wird über den Beschleunigungssensor anhand von drei Achsen bestimmt. Zum anderen bestimmt der Controller anhand der Sensordaten, ob sich ein POI im Sichtfeld befindet und an welcher Stelle auf dem Display, d.h. an welcher x- und y-Koordinate, dieser gezeichnet werden muss. Im Model wird die Datenhaltung der POI und eine einheitliche Schnittstelle für diese realisiert. Die Schnittstelle besteht aus einem XML-Schema und einem passenden XML-Parser. Dadurch ist es möglich, POI plattformunabhängig auszutauschen und zu erweitern. Eine View beinhaltet die unterschiedlichen Elemente, die auf dem Display angezeigt werden sollen. Dies sind im Wesentlichen zwei Darstellungen für die POI. Eine für die Kameraansicht und eine für die Kartenansicht. Ebenfalls findet sich hier eine Darstellungsform des Radars für die Kameraansicht. Eine vierte Komponente, welche nicht explizit zu AREA gehört, ist eine Sammlung von Bibliotheken und Frameworks des iOS. Diese werden benötigt, um Funktionen wie das Auslesen der Sensoren oder das Zeichnen auf dem Display verwenden zu können. Benutzt wird diese zusätzliche Schicht von allen internen AREA-Komponenten.

Diese vier Komponenten sind in einer Schichtenarchitektur angeordnet (Abbildung 5.1), sodass unten liegende Schichten ihre Dienste und Funktionen den darüber liegenden durch Schnittstellen anbieten. Durch diesen Architekturaufbau wird Modularisierung gewährleistet, sodass sowohl die Datenhaltung als auch die unterschiedlichen Elemente der View auf Bedarf angepasst und erweitert werden können. Zusätzlich ist es durch die Kompaktheit von AREA möglich, beliebige Anwendungen auf ihr aufzubauen, oder selbst in bestehende einzugliedern.

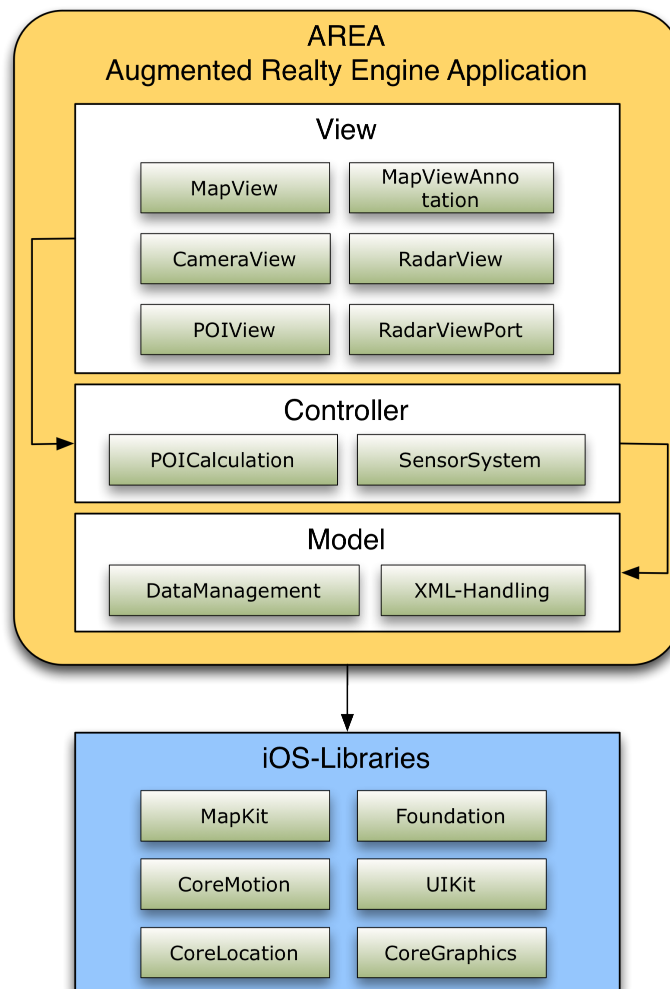


Abbildung 5.1: Schichtenarchitekturaufbau von AREA. Zu erkennen sind die drei Schichten von AREA und die zusätzliche Schicht von iOS-Libraries

5.2 Klassenstruktur

Im Klassendiagramm von AREA können die Module der Komponenten näher betrachtet werden (Abbildung 5.2). Die Klassenstruktur ist so aufgebaut, dass es ohne weiteres möglich ist, Anwendungen in AREA zu integrieren oder AREA in einer bereits existieren Anwendung zu verwenden. Dazu wird nur eine Referenz auf die Klasse `ViewController` benötigt. Das Initialisieren der anderen Komponenten und Module wird dann automatisch von der AREA getätigt.

Wie im Architekturentwurf bereits beschrieben, ist der Controller für das Auslesen der Sensoren und die Berechnung von POI zuständig. Die Klasse `SensorController` implementiert das Protocol `CLLocationManagerDelegate`, damit GPS- und und Kompassdaten abgefragt werden können. In dieser Klasse läuft des Weiteren eine endlose Anwendungsschleife, in der die Daten des Beschleunigungssensors *gepollt* und sowohl dessen Daten als auch die des GPS- und Kompasssensors an den `LocationController` weitergegeben werden. Dazu muss die Klasse das Protocol `SensorControllerDelegate` implementieren. Hier wird dann entschieden, ob sich POI im Sichtfeld des Benutzers befinden und an welcher Stelle des Displays diese gezeichnet werden müssen. `ViewController` wird über diese Berechnungen und die zu zeichnenden POI über das zu implementierende Protocol `LocationControllerDelegate` informiert. `ViewController` ist des Weiteren dafür zuständig, die komplette View zu laden. Hierzu muss auf die Kamera anhand des `UIImagePickerController` zugegriffen werden. Anhand dessen `CamerOverlayView` ist es dann möglich, die `RadarView`, `RadarViewPortView` und die `LocationView` auf der Kameraansicht zu platzieren und anzuzeigen. Letztere ist die grafische Darstellung von POI in der Kameraansicht. Das Modell besitzt neben `Store` und `Location` noch einen `XMLParser`. Dieser ist für das platformunabhängige Manipulieren und Laden von POI zuständig.

AREA, wie es in dieser Arbeit entwickelt wird, soll laut den Anforderungen eine Kartenansicht anbieten. Dies ist ein gutes Beispiel für die leichte Integration von Anwendungen. Der `MapViewController` ist für die Kartenansicht zuständig und wird vom iOS zur Verfügung gestellt. Um von diesem auf die Kameraansicht zu wechseln, muss lediglich `ViewController` referenziert werden. Um den Rest kümmert sich AREA vollständig autonom.

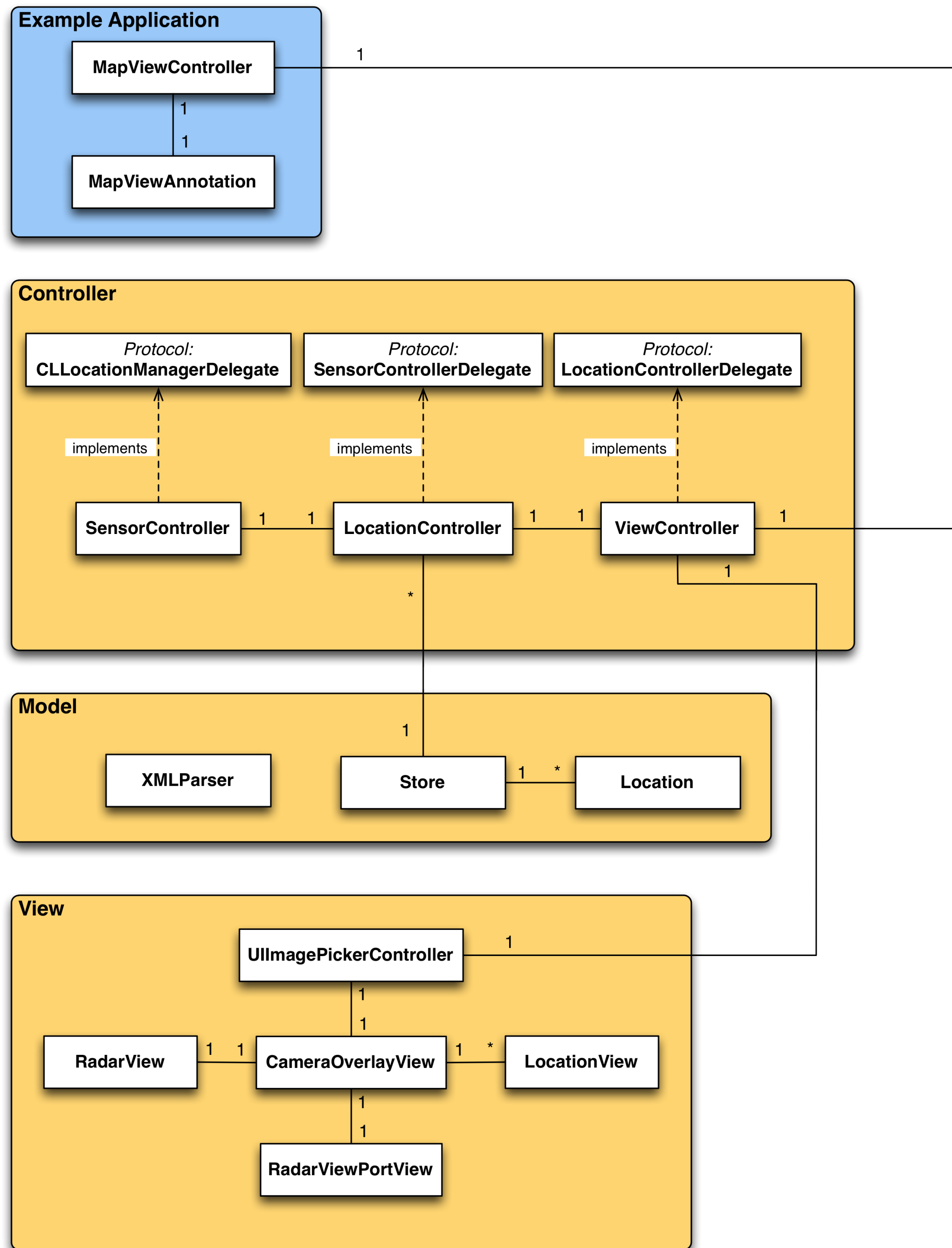


Abbildung 5.2: Klassendiagramm von AREA. Orange dargestellt ist die eigentliche AR-Engine. Blau die Kartenansicht

5.3 Kommunikationsablauf

Das Herzstück von AREA ist der Controller. In diesem wird die Position, die Blickrichtung, das Sichtfeld und die Position auf dem Display von sichtbaren POI berechnet. Der Kommunikationsablauf, der dabei zwischen `SensorController`, `LocationController` und `ViewController` abläuft, lässt sich durch Kommunikationsdiagramme gut veranschaulichen.

Wenn der Benutzer seine Position ändert (Abbildung 5.3), wird am `SensorController` eine Funktion aufgerufen, welche den `LocationController` dazu auffordert, die POI aus dem `Store` zu extrahieren, welche von der neuen GPS-Koordinate ausgehend im aktuellen Radius liegen. Die POI, die dieses Kriterium erfüllen, werden im `LocationController` zwischengespeichert, da sie später noch gebraucht werden. Der `LocationController` wiederum veranlasst den `ViewController` dazu, durch einen Redraw den Radar mit den POI aus der Umgebung zu befüllen. Es wird also immer dann, wenn der Benutzer seine Position ändert, eine neue Liste von POI aus der Umgebung, innerhalb eines bestimmten Radius, ermittelt, zwischengespeichert und auf den Radar gezeichnet.

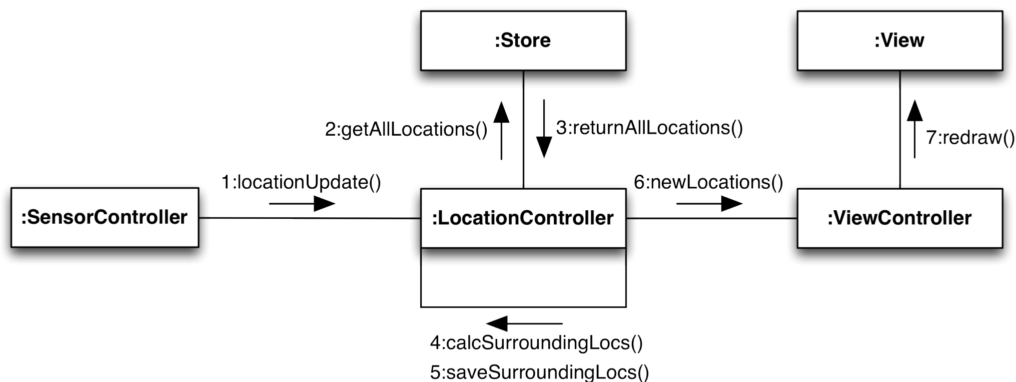


Abbildung 5.3: Kommunikationsdiagramm nach der Positionsänderung des Benutzers

Ähnlich läuft die Kommunikation ab, wenn neue Daten des Beschleunigungssensors und des Kompasses abgefragt wurden (Abbildung 5.4). Im `SensorController` läuft eine Anwendungsschleife, die in kurzen Zeitabständen den Beschleunigungssensor und den Kompass *pollen*. Immer wenn dies geschehen ist, werden dem `LocationController` diese Daten übermittelt. Dieser ist dafür zuständig, anhand der Daten das aktuelle Sichtfeld zu

berechnen. Wenn die Berechnung abgeschlossen ist, benutzt er die zwischengespeicherten POI und extrahiert aus diesen wiederum die POI, die sich im Sichtfeld befinden. Für diese wird dann ein Punkt mit x- und y-Koordinate berechnet, der bestimmt, an welcher Position des Displays die POI gezeichnet werden müssen. Darauf wird der `ViewController` darüber informiert, dass sich die POI im Sichtfeld geändert haben, worauf er einen Redraw der Kameraansicht veranlasst.

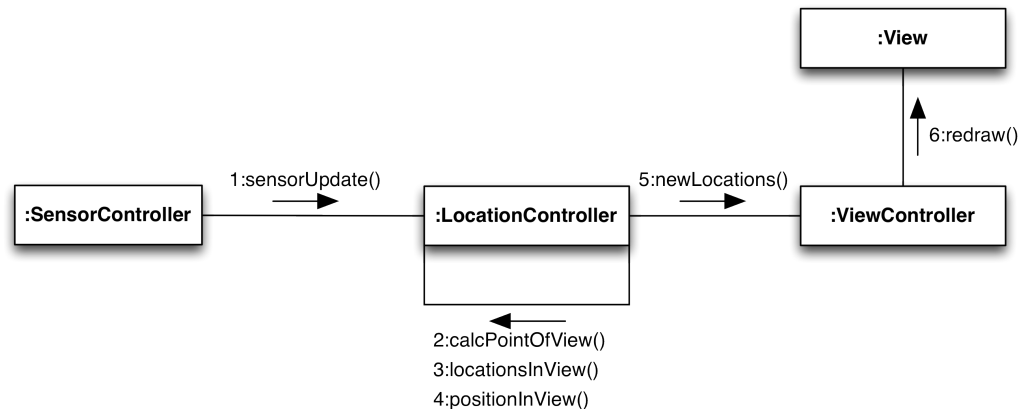


Abbildung 5.4: Kommunikationsdiagramm für einen Durchlauf der Anwendungsschleife zur Abfrage von Beschleunigungssensor und Kompass

5.4 Datenhaltung

Bisher wurde mehrere Male erwähnt, dass AREA eine einheitliche und einfache Schnittstelle zum Hinzufügen von POI anbieten soll. Bevor jedoch darauf eingegangen werden kann, muss beschrieben werden, welche Attribute in einem POI gespeichert werden müssen.

Ein POI benötigt als erstes einige Standardinformationen, die bereits vor der Laufzeit bekannt sein müssen. Hierzu gehören ein Name, ggf. zusätzliche Informationen und eine GPS-Koordinate. Letztere wird benötigt, um einen POI überhaupt lokalisieren zu können. Die GPS-Koordinate besteht aus Längengrad, Breitengrad und Höhe. Es fehlen jedoch zusätzliche Informationen, die erst zur Laufzeit berechnet werden können. Wichtig sind diese unter anderem, um den POI an der richtigen Stelle des Displays zeichnen zu können. Dazu gehört ein Punkt, der aus einer x-Koordinate und einer y-Koordinate besteht. Dieser

Punkt kann anhand der horizontalen und vertikalen Abweichung von der Blickrichtung des Benutzers berechnet werden, welche ebenfalls im Datenobjekt vermerkt werden. Zu guter Letzt wird die Distanz zwischen Benutzer und POI benötigt, um auch hier entscheiden zu können, ob der POI für den Benutzer sichtbar ist.

In Abbildung 5.5 werden die Attribute, die ein POI besitzen muss, nochmals in einem Entity-Relationship-Model dargestellt. Dabei ist auch zu erkennen, dass POI-Datenobjekte in einem POI-Store gesammelt werden.

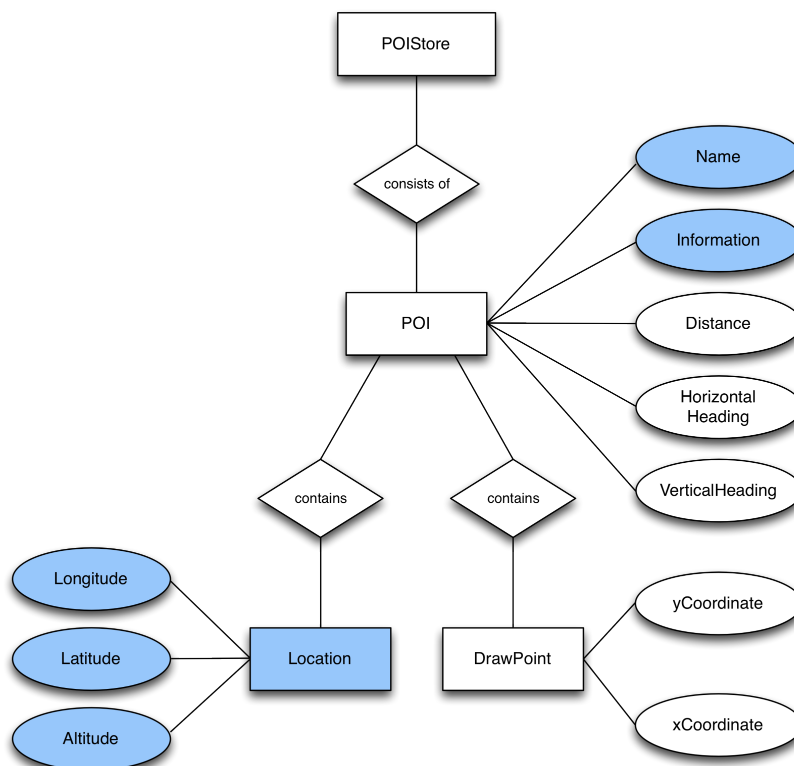


Abbildung 5.5: ER-Diagramm eines POI. Blau dargestellt sind hier die Attribute, die vor dem Starten der Anwendung bekannt sein müssen. Weiß diese, die erst zur Laufzeit berechnet werden können

Da nun bekannt ist, welche Attribute in einem POI vor der Laufzeit bekannt sein müssen, kann eine plattformunabhängige Schnittstelle zum Austausch und Laden erstellt werden. Da das XML-Format hinsichtlich Plattformunabhängigkeit und der Darstellungsform sehr gut dafür geeignet ist, basiert die Schnittstelle der POI von AREA auf XML-Dokumenten.

Um die Eindeutigkeit der Schnittstelle zu gewährleisten, wird ein XML-Schema definiert (Listing 5.1), anhand dessen XML-Dokumente validiert werden können. Ein Beispiel für ein valides XML-Dokument, das zwei unterschiedliche POI beinhaltet, wird in Listing 5.2 dargestellt.

```
1  <?xml version="1.0"? encoding="UTF8">
2
3  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
4  targetNamespace="http://www.area.com"
5  xmlns="http://www.area.com elementFormDefault="qualified">
6
7  <xs:element name="locations">
8     <xs:complexType>
9        <xs:sequence>
10          <xs:element name="location" minOccurs="0" maxOccurs="unbounded">
11             <xs:complexType>
12                <xs:sequence>
13                   <xs:element name="latitude" type="xs:decimal">
14                   <xs:element name="longitude" type="xs:decimal">
15                   <xs:element name="altitude" type="xs:decimal">
16                   <xs:element name="name" type="xs:string">
17                   <xs:element name="information">
18                   <xs:simpleType>
19                      <xs:restriction base="xs:string">
20                         <xs:minLength value="0"/>
21                      </xs:restriction>
22                   </xs:simpleType>
23                </xs:sequence>
24             </xs:complexType>
25          </xs:element>
26       </xs:sequence>
27    </xs:complexType>
28 </xs:element>
29 </xs:schema>
```

Listing 5.1: XML-Schema zum Datenaustausch von POI

```
<xml version="1.0" encoding="UTF8">
<locations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="locationsSchema.xsd">
   <location>
      <latitude>48.4042981</latitude>
      <longitude>9.979349</longitude>
      <altitude>479</altitude>
      <name>Ulm</name>
      <information>In Ulm, um Ulm und um Ulm herum.</information>
   </location>
   <location>
      <latitude>48.3906042</latitude>
      <longitude>10.0060398</longitude>
      <altitude>380</altitude>
      <name>Neu-Ulm</name>
      <information>Bayrische Nachbarstadt von Ulm.</information>
   </location>
</locations>
```

Listing 5.2: XML-Dokument eines POI

Durch die Struktur des internen Datenobjekts eines POI und das XML-Dokument zum Datenaustausch, ist eine Erweiterung der Datenstruktur ohne weiteres möglich. Dabei muss nur darauf geachtet werden, dass hinzugefügte Attribute, die vor der Laufzeit bekannt sein müssen, wie z.B. ein zusätzliches Attribut zur Kategorisierung, sowohl im XML-Dokument und im XML-Schema, als auch im internen Datenobjekt erweitert werden.

6 Implementierung

Die Implementierung von AREA wurde mit der Programmiersprache Objective-C auf Basis von iOS 5.1 für das iPhone 4S umgesetzt. Als Entwicklungsumgebung wurde Xcode [20] in der Version 4.4.1 verwendet. Xcode wird von Apple zur Programmierung von MACs und Handhelds zur Verfügung gestellt und beinhaltet einige Features, die die Entwicklung von Applikationen um einiges vereinfachen.

Im Folgenden wird auf die wichtigsten Konstrukte bei der Implementierung von AREA eingegangen. Hierzu gehört die Manipulation der Kameraansicht und das Anzeigen von POI. Im Controller, dem Herzstück von AREA, wird auf das korrekte Abfragen von Sensoren, das Interpretieren von Daten, das Berechnen des Sichtfelds und auf die Positionsbestimmung von POI auf dem Display eingegangen. Da es eine Anforderung ist, eine lückenlose gute Kommentierung durchzuführen, können die hier nicht erwähnten Konstrukte leicht im Programmcode nachgelesen und nachvollzogen werden.

6.1 Manipulation der Kameraansicht

Auf der Kameraansicht des iPhones sollen ein Radar und die POI als benutzerdefinierte `UIViews` angezeigt werden. Hierfür muss ein `UIImagePickerController` instanziiert und angepasst werden, welcher für die Steuerung der Kamera zuständig ist und bereits von iOS zur Verfügung gestellt wird. Listing 6.1 ist ein Auszug aus der `init`-Methode der Klasse `AREAViewController`. Diese Klasse ist unter anderem, wie im Entwurf bereits angemerkt, für das Steuern der einzelnen View-Komponenten zuständnig. Die Kameraansicht besitzt normalerweise eine `UINavigationBar`. Da die Kamera aber über das komplette Display reichen soll, wird diese ausgeblendet. Dadurch entsteht ein schwarzer leerer Balken am oberen Ende des iPhones, welcher durch eine Streckung des Kamerabildes entfernt werden muss. Das passiert mit Zeile 7, wobei die beiden Konstanten `CAMERA_TRANSFORM_X` und `CAMERA_TRANSFORM_Y` den Wert 1.24299 besitzen. Um

nun auf der Kameraansicht zeichnen zu können, muss dem `UIImagePickerController` noch ein benutzerdefiniertes Overlay in Zeile 10 hinzugefügt werden. Auf dieses Overlay können dann der Radar, ein Slider für die Einstellung des Radius, und die POI gezeichnet werden.

```
1   self.picker = [[UIImagePickerController alloc] init];
2   self.picker.sourceType = UIImagePickerControllerSourceTypeCamera;
3   self.picker.showsCameraControls = NO;
4   self.picker.navigationBarHidden = YES;
5   self.picker.wantsFullScreenLayout = YES;
6   // without this the camera layer would not cover the entire screen
7   self.picker.cameraViewTransform = CGAffineTransformScale(self.picker.
        cameraViewTransform, CAMERA_TRANSFORM_X, CAMERA_TRANSFORM_Y);
8
9   // initiate the overlay for the UIImagePickerController
10  self.overlayView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 480)];
11  self.overlayView.opaque = NO;
12  self.overlayView.backgroundColor = [UIColor clearColor];
13  ...
14  self.locationView = [[UIView alloc] initWithFrame:CGRectMake((IPHONE_WIDTH-
        VIEW_SIZE)/2, (IPHONE_HEIGHT-VIEW_SIZE)/2, VIEW_SIZE, VIEW_SIZE)];
15  ...
16  [self.overlayView addSubview:self.locationView];
17  self.picker.cameraOverlayView = self.overlayView;
```

Listing 6.1: Initialisierung der iPhone-Kamera und Erstellen eines benutzerdefinierten Overlays

Die POI, die sich im Sichtfeld des Benutzers bzw. der iPhone-Kamera befinden, sollen auf diesem Overlay angezeigt werden. Anstatt die POI direkt auf dem Overlay zu zeichnen, wird eine zweite View namens `locationView` mit einer Größe von `580x580` Pixeln mittig auf das Overlay gelegt. Dieses Vorgehen hat mehrere Gründe.

Zum einen soll AREA die POI auch dann korrekt anzeigen, wenn das iPhone etwas schräg gehalten wird. Dazu müssen die POI, je nach Lage des iPhones, um einen gewissen Winkel gedreht und relativ zur Rotation verschoben werden. Anstatt nun jeden einzelnen POI zu drehen, und relativ zur Rotation neu zu positionieren, ist es möglich, nur die `locationView`, welche die POI beinhaltet, um den gewünschten Winkel zu rotieren. Dadurch drehen sich die POI automatisch mit und wichtige Ressourcen, die später zur Berechnung benötigt werden, können eingespart werden.

Die Größe von `580x580` Pixel wird benötigt, um sowohl POI zu zeichen, die im Portrait-, Landscapemodus und bei einer Schieflage dazwischen sichtbar sind. In Abbildung 6.1 ist als weißes Quadrat die `locationView` dargestellt. Wie man sieht, ist sie größer als das eigentliche iPhone-Display mit seinen `320x480` Pixeln. Das Sichtfeld wird deswegen um einen gewissen Faktor vergrößert, sodass alle POI, die entweder im Portrait-, Landscapemodus oder irgendeiner Rotation dazwischen im Sichtfeld liegen, auf der `locationView` eingezeichnet werden. Es ist weiter ein POI zu erkennen, der sich im Landscapemodus außerhalb des momentanen Sichtfelds befindet, aber dennoch auf die `locationView` gezeichnet wird. Wird das iPhone in Richtung Portaitmodus rotiert, so ändert sich das momentane Sichtfeld in der Höhe und der Breite. Da die POI jedoch unabhängig von der Rotation des iPhones auf das Quadrat gezeichnet werden, muss dieses in diesem Fall einfach gleichmäßig in die entgegengesetzte Richtung rotiert werden, damit der POI sichtbar wird.

Diese Vorgehen hat den zusätzlichen Vorteil, dass bei der Berechnung des Sichtfelds nicht beachtet werden muss, in welcher Rotation sich das iPhone befindet. Das vertikale und horizontale Sichtfeld wird im Verhältnis zur Diagonalen des Displays skaliert, sodass ein neues maximales Sichtfeld in der Größe `580x580` Pixeln entsteht. Da die `locationView` zusätzlich mittig eingesetzt wird, wird das tatsächliche vom Benutzer einsehbare Sichtfeld der iPhone-Kamera nicht verfälscht, und kann je nach Rotation des iPhones durch Gegenrotation angepasst werden.

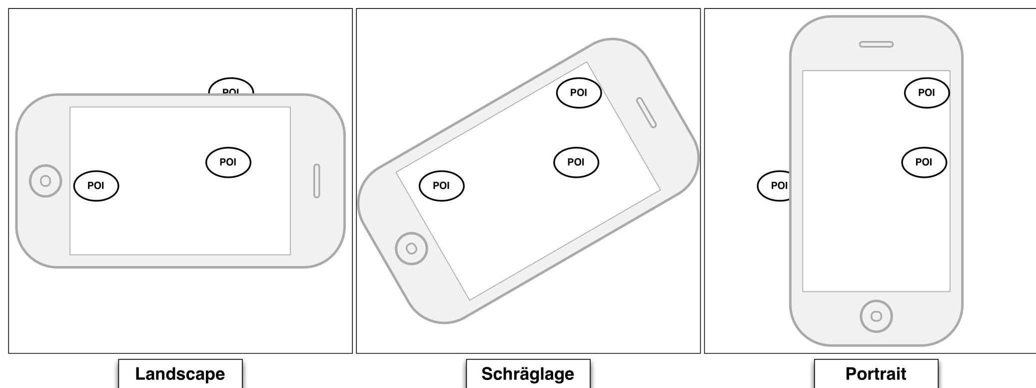


Abbildung 6.1: Darstellung der locationView (weißes Quadrat). Zu sehen sind die POI die in das neue maximale Sichtfeld eingezeichnet werden, aber im echten Sichtfeld nur in bestimmten Rotationen sichtbar sind.

Ein letzter Grund hat wieder mit Performance zu tun, und ist in Listing 6.2 zu sehen. Wenn das Display neu gezeichnet werden muss, können die bisher auf der `locationView` befindlichen und damit bisher sichtbaren POI sehr leicht abgefragt und wiederverwendet werden. Anstatt das komplette Display zu leeren, und alle POI neu zu initialisieren sowie zu zeichnen, werden die POI, die noch immer sichtbar sein sollen, verschoben, die nicht mehr sichtbaren werden gelöscht und nur die, die neu hinzukommen, werden frisch initialisiert.

In Tabelle 6.1 werden die Gründe, die für eine solche `locationView` sprechen, nochmals aufgelistet.

```
1  -(void)didUpdateHeadingLocations:(NSArray *)locations
2  {
3      // array with new visible locations
4      NSMutableArray *array = [NSMutableArray arrayWithArray:locations];
5      // iterate over the subviews (the AREALocationViews) of the locationView
6      for(AREALocationView *view in self.locationView.subviews) {
7          if([array containsObject:view.location]) {
8              // if the location (subview) exists also in the new locations,
9              // just update the frame (no redraw necessary -> performance)
10             [array removeObject:view.location];
11             [view updateFrame];
12         } else {
13             // otherwise remove the subview from its superview
14             [view removeFromSuperview];
15         }
16     }
17
18     // the locations that were not yet visible (a subview in locationView) must be
           initiated
19     // and added to the locationView
20     for(AREALocation *loc in array) {
21         AREALocationView *view = [[AREALocationView alloc] initWithLocation:loc];
22         [self.locationView addSubview:view];
23     }
24 }
```

Listing 6.2: Ressourcensparender Redraw-Prozess der POI

Tabelle 6.1: Gründe für die separate locationView

Grund	**Nutzen**
$580\ x\ 580$ Pixel	Zeichnen der POI unabhängig von der Rotation
Rotation der POI	Alle auf einmal, anstatt jeden einzeln rotieren
Berechnung von POI im Sichtfeld	Unabhängig von der Rotation des iPhones
Zugriff auf POI	Wiederverwendbarkeit von POI im Redrawprozess

6.2 Sensoren und ihre Daten

Das korrekte Auslesen der Sensoren ist die Aufgabe von `AREASensorController`. In Listing 6.3 ist die `init`-Methode einsehbar. Hier wird zum einen ein `CMMotionManager`, der für das Auslesen des Beschleunigungssensors notwendig ist und zum anderen ein `CMLocationManager` initialisiert, welcher für das Auslesen des GPS-Sensors und des Kompasses zuständig ist.

Der Beschleunigungssensor wird benötigt, um die momentane Haltung des iPhones anhand von drei Achsen berechnen zu können (Abbildung 6.2) [21]. Die Haltung wird später benötigt, um die `locationView`, wie oben gezeigt, korrekt zu rotieren, den Kompass anzupassen und vorallem das vertikale Sichtfeld zu bestimmen. Da dessen Daten gepollt werden müssen, wird in Zeile 7 ein Zeitintervall zur Abfrage definiert. Dieses Intervall sollte so klein wie möglich sein, da es sonst zum Nachziehen, Ruckeln und zu Ungenauigkeiten bei Berechnungen kommen kann (weiteres dazu in Kapitel 6.3). Weiter müssen die Kompass-Daten zur Bestimmung der Blickrichtung und der GPS-Sensor zur Positionsbestimmung abgefragt werden. Diese Sensoren *pushen* ihre Daten jedoch an Methoden, die im *Protocol* `CLLocationManagerDelegate` definiert sind. Da AREA laut Anforderungen eine sehr hohe Genauigkeit und Verlässlichkeit anbieten soll, müssen am `CMLocationManager` einige Einstellungen vorgenommen werden. Zum einen sollen alle Kompassdaten, egal wie klein die Differenz zum davor gemessenen Wert war, abgefragt werden. Hierzu wird in Zeile 10 der `headingFilter` auf die Konstante `kCLHeadingFilterNone`, was dem Wert

Null Grad entspricht, gesetzt. Ähnliches gilt für die Genauigkeit des GPS-Sensors. In Zeile 12 wird der `distanceFilter` auf die Konstante `kCLDistanceFilterNone` gesetzt, wodurch alle Positionsänderungen abgefragt werden und in Zeile 14 wird `desiredAccuracy` auf eine Konstante gesetzt, sodass der GPS-Sensor die beste Genauigkeit liefert.

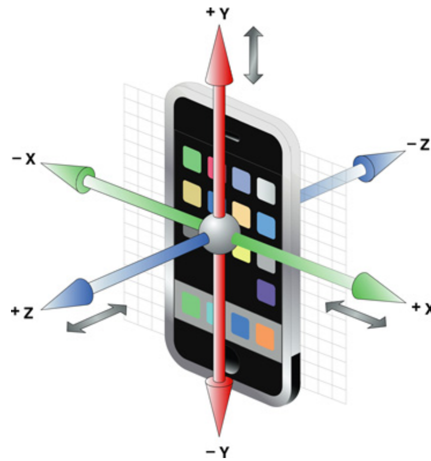


Abbildung 6.2: Die drei Achsen des Beschleunigungssensors.

```
1  -(id)init
2  {
3      if(self = [super init]) {
4          _motionManager = [[CMMotionManager alloc] init];
5          // for reliable accerleration data, the update frequency must be really
               high.
6          // 1/90.0 seconds is heigh enough, so no lagging will be visible
7          self.motionManager.accelerometerUpdateInterval = 1.0/90.0;
8          _locationManager = [[CLLocationManager alloc] init];
9          // no heading filter so all heading updated will be received
10         self.locationManager.headingFilter = kCLHeadingFilterNone;
11         // location updates every 10 meters
12         self.locationManager.distanceFilter = kCLDistanceFilterNone;
13         // with the highest possible accuracy. ATTENTION: High battery usage!
14         self.locationManager.desiredAccuracy =
               kCLLocationAccuracyBestForNavigation;
15         self.locationManager.delegate = self;
16     }
17     return self;
18 }
```

Listing 6.3: init-Methode von AREASensorController

In Listing 6.4 wird das Auslesen der Sensoren gestartet. Da die Daten des Beschleunigungssensors *gepollt* werden müssen, wird in Zeile 7 eine `NSOperationQueue` erstellt. In dieser werden die Daten in einem separaten Thread abgefragt. Da nur die Gravitation zur Berechnung der Haltung des iPhones interessant ist, wird in dieser Queue ein *low-pass filter* implementiert [21] und das Ergebnis in die Instanzvariablen `_xAcc`, `_yAcc` und `_zAcc` gespeichert. Im Entwurf von AREA wurde beschrieben, dass `AREASensorController` eine Anwendungsschleife implementiert. Diese Schleife wird in Zeile 16 initalisiert, mit einem Zeitintervall versehen und ebenfalls gestartet. In der Anwendungsschleife werden alle Daten, also Beschleunigungs-, GPS- und Kompassdaten gesammelt und an ein `delegate`, in diesem Fall an `AREALocationController`, der für die Berechnung zuständig ist, gesendet. Letztere muss hierzu das Protocol aus Listing 6.5 implementieren. Die erste Methode wird aufgerufen, wenn sich die Position anhand von GPS-Daten geändert hat, die zweite in jedem Durchlauf der Anwendungsschleife. In diesen Methoden werden dann die Berechnungen von POI durchgeführt und daher werden sie im nächsten Kapitel genauer beschrieben.

```
1  -(void)startSensoring
2  {
3      [self.locationManager startUpdatingLocation];
4      [self.locationManager startUpdatingHeading];
5
6      // this queue is polling the acceleration data.
7      self.motionQueue = [[NSOperationQueue alloc] init];
8      [self.motionManager startAccelerometerUpdatesToQueue:self.motionQueue
           withHandler:^(CMAccelerometerData *data, NSError *error) {
9          _xAcc = (data.acceleration.x * 0.1) + (_xAcc * (1.0 - 0.1));
10         _yAcc = (data.acceleration.y * 0.1) + (_yAcc * (1.0 - 0.1));
11         _zAcc = (data.acceleration.z * 0.1) + (_zAcc * (1.0 - 0.1));
12     }];
13
14     // this is the main-loop of the sensor controller. every 1/30.0 seconds the
           sensor data will be collected and sent to the delegate.
15     // In addition every 1/30.0 seconds the screen will be redrawn
16     self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0/30.0 target:self
           selector:@selector(updateSensors) userInfo:nil repeats:YES];
17 }
```

Listing 6.4: Starten der Sensorabfrage und der Anwendungsschleife

```
1 // Protocol to delegate updates of sensors to a delegate thats uses this raw data
2 @protocol AREASensorControllerDelegate
3 -(void)didUpdateToLocation:(CLLocation *)newLocation;
4 -(void)didUpdateBearingX:(double)newBearingX andBearingY:(double)newBearingY
      andBearingZ:(double)newBearingZ andHeading:(double)newHeading;
5 @end
```

Listing 6.5: AREASensorControllerDelegate

6.3 Berechnungen im Controller

Im Folgenden Abschnitt werden die Berechnungen im `AREALocationController` vorgestellt. Im Wesentlichen handelt es sich dabei um das Sammeln von POI aus der Umgebung des Benutzers, der Berechnung des Sichtfelds anhand der Sensordaten und den Formeln aus Kapitel 4 und der korrekten Platzierung von POI auf dem Display.

6.3.1 Berechnen von POI in der Umgebung

Wenn im `AREASensorController` eine neue Position des Benutzers eingegangen ist, müssen die POI, die sich innerhalb eines bestimmten Radius um den Benutzer befinden, neu gesammelt und berechnet werden. `AREALocationController` wird darüber, durch die im Protocol aus Listing 6.5 definierte Methode `didUpdateToLocation:`, informiert und bekommt dabei die neue Position mit übergeben. Das Vorgehen wird in Listing 6.6 dargestellt.

Für jeden POI, welche sich gesammelt im `AREAStore` befinden, wird zuerst überprüft, ob er sich innerhalb eines bestimmten Radius um die neue Position befindet. Dies geschieht in Zeile 7 durch die Haversine-Formel (Formel 4.3) aus Kapitel 4. Da das iOS diese Methode jedoch schon intern implementiert, muss diese nicht nochmals ausprogrammiert werden. Ist die Distanz zwischen dem POI und der aktuellen Position kleiner oder gleich groß wie der Radius (`maxDistance`), so werden für diesen POI der horizontale (Zeile 11) und vertikale (Zeile 13) Kurs berechnet und im Datenobjekt des POI zusammen mit der Distanz gespeichert. Schließlich wird `didUpdateLocations:inDistance:fromLocation:` aus

dem Protocol `AREALocationControllerDelegate` am `delegate`, in diesem Fall implementiert von `AREAViewController`, aufgerufen und darüber informiert, dass ein bestimmter Redraw notwendig ist. Es werden weiter alle POI, die sich im Radius befinden, in Zeile 17 intern zwischengespeichert. Diese werden zur Berechnung des Sichtfeldes benötigt.

```
1  -(void)didUpdateToLocation:(CLLocation *)newLocation
2  {
3      self.currentLocation = newLocation;
4      NSMutableArray *locations = [NSMutableArray array];
5
6      for(AREALocation *loc in self.store.store) {
7          double distance = [loc.location distanceFromLocation:newLocation];
8          if(distance <= self.maxDistance) {
9
10             // the horizontal heading with values between 0 and 359 degrees
11             double horizontalHeading = [self
                   calculateHorizontalHeadingFromLocation:newLocation toLocation:loc.
                   location];
12             // the vertical heading with values from +90 (top) to -90 (bottom)
                   degrees
13             double verticalHeading = [self calculateVerticalHeadingFromLocation:
                   newLocation toLocation:loc.location withDistance:distance];
14             loc.distance = distance;
15             loc.horizontalHeading = horizontalHeading;
16             loc.verticalHeading = verticalHeading;
17             [locations addObject:loc];
18         }
19     }
20     ...
21     // calculation is finished. So we call the delegate method with the current
           distance, the surrounding locations and the current user location
22     [self.delegate didUpdateLocations:self.surroundingLocations inDistance:self.
           maxDistance fromLocation:newLocation];
23 }
```

Listing 6.6: Berechnung von umgebenden POI und deren vertikale und horizontale Abweichung

In Listing 6.7 und Listing 6.8 sind die Berechnungen für den horizontalen und vertikalen Kurs der POI dargestellt. Dabei handelt es sich im Wesentlichen um die Umsetzung der

Formeln 4.4 und 4.5 aus Kapitel 4 in Programmcode. Erstere liefert einen Wert zwischen $0°$ und $359°$, letztere einen Wert zwischen $-90°$ und $+90°$. Da es sich bei den Ergebnissen um Gleitkommazahlen handelt, das Ergebnis der horizontalen Kurses jedoch an genannten Wertebereich durch Modulo-Rechnung in Zeile 11 angepasst werden muss, wird die Funktion `fmod(double, double)` verwendet. Durch diese ist es möglich, den Modulooperator auch auf Gleitkommazahlen anzuwenden.

```
1  -(double)calculateHorizontalHeadingFromLocation:(CLLocation *)loc1 toLocation:(
       CLLocation *)loc2
2  {
3      double lat1 = radians(loc1.coordinate.latitude);
4      double lon1 = radians(loc1.coordinate.longitude);
5      double lat2 = radians(loc2.coordinate.latitude);
6      double lon2 = radians(loc2.coordinate.longitude);
7      double deltaLon = lon2-lon1;
8      double x = sin(deltaLon) * cos(lat2);
9      double y = cos(lat1) * sin(lat2) - sin(lat1) * cos(lat2) * cos(deltaLon);
10     double heading = atan2(x, y) * 180.0 / M_PI;
11     return fmod(heading + 360.0, 360.0);
12 }
```

Listing 6.7: Berechnung des horizontales Kurses

```
1  -(double)calculateVerticalHeadingFromLocation:(CLLocation *)loc1 toLocation:(
       CLLocation *)loc2 withDistance:(double)distance
2  {
3      double locHeight = loc2.altitude;
4      double myHeight = loc1.altitude;
5      double dHeight = 0.0;
6      int sign = 0;
7      if(locHeight >= myHeight) {
8          dHeight = locHeight - myHeight;
9          sign = 1;
10     } else {
11         dHeight = myHeight - locHeight;
12         sign = -1;
13     }
14     double verticalHeading = dHeight/distance;
15     return sign * degrees(atan(verticalHeading));
```

Listing 6.8: Berechnung des vertikalen Kurses

6.3.2 Berechnung des Sichtfelds

Nach jedem Durchlauf der Anwendungsschleife werden `AREALocationController` die aktuellen Daten von Kompass und Beschleunigungssensor zugeschickt. Das bedeutet, dass sich der Kurs und die Haltung des iPhones geändert haben können und damit eine neue Berechnung des Sichtfelds durchgeführt werden muss.

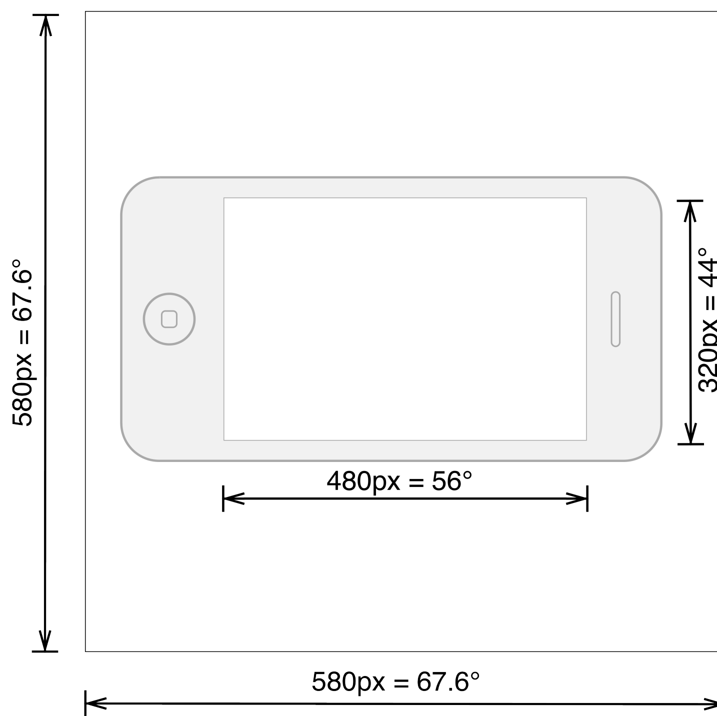


Abbildung 6.3: Darstellung von maximalem und echtem Blickwinkel.

Dazu ist es allerdings vorerst notwendig, den maximalen Blickwinkel in der Höhe und der Breite zu bestimmen, da ,wie weiter oben bereits gezeigt wurde, die POI auf einer `580x580` Pixel großen Fläche gezeichnet werden. D.h. der eigentliche Blickwinkel der iPhone-Kamera von $56°$ in der Breite und $44°$ in der Höhe muss proportional zur neuen Fläche vergrößert werden. Da es egal ist, ob diese Umrechnung anhand des horizontalen oder des vertikalen Blickwinkels umgesetzt wird, wurde hier folgende Rechnung durchgeführt:

$$\theta = \frac{Seitenlänge_{neu}}{Seitenlänge_{alt}} * Blickwinkel_{alt} = \frac{580px}{480px} * 56° = 67,6° \tag{6.1}$$

Der maximale Blickwinkel hat damit eine Höhe und eine Breite von $67,6°$. In Abbildung 6.3 wird nochmals verdeutlicht, dass der Benutzer weiterhin nur den echten Blickwinkel von $56°$ und $44°$ einsehen kann und der maximale Blickwinkel nur für die interne Berechnung und das einfache Rotieren der POI verwendet wird.

```
-(void)didUpdateBearingX:(double)newBearingX andBearingY:(double)newBearingY
    andBearingZ:(double)newBearingZ andHeading:(double)newHeading
{
    // calculate the rotation of the device in degrees referenced to portrait mode
    double deviceRotation = atan2(-newBearingY, newBearingX);
    deviceRotation = deviceRotation - M_PI/2;

    // same to calculate the real heading relative to top, whatever in which
        orientation the device is
    double headingOffset = fmod((deviceRotation*180.0/M_PI)+360.0, 360.0);
    double heading = fmod(newHeading + headingOffset, 360.0);
    double leftAzimuth = fmod(heading - DEGREES_IN_VIEW/2.0 + 360.0, 360.0);
    double rightAzimuth = fmod(heading + DEGREES_IN_VIEW/2.0 + 360.0, 360.0);

    double verticalHeading = degrees(asin(newBearingZ));
    double topAzimuth = fmod(verticalHeading+DEGREES_IN_VIEW/2.0, 180.0);
    double bottomAzimuth = fmod(verticalHeading-DEGREES_IN_VIEW/2.0, 180.0);

    ...
}
```

Listing 6.9: Berechnung des Sichtfeldes

Über den maximalen Blickwinkel ist es nun möglich, das aktuelle Sichtfeld anhand der Daten vom `AREASensorController` zu berechnen. In Listing 6.9 ist der Programmcode für dieses Vorhaben dargestellt. Die Kompassdaten und damit der horizontale Kurs ist aber nur dann korrekt, wenn sich das iPhone im Portrait-Modus befindet. Wird das Gerät jedoch rotiert, z.B in den Landscape-Modus, so werden die Kompassdaten von iOS noch immer so gehandhabt, als befinde es sich im Portrait-Modus. Um dieses Problem zu umgehen, könnte in diesem konkreten Fall einfach ein Wert von $90°$ zum eigentlich Kurs hinzu addiert werden. Da aber kontinuierliche Werte benötigt werden, wird in Zeile 4 und 5, anhand der Gravitationswerte der x- und y-Achse des Beschleunigungssensor, die aktuelle Rotation errechnet [22]. Abbildung 6.4 stellt dar, wie der echte Kurs aussehen muss. In Zeile 8 wird dieser Wert ins Bogenmaß umgerechnet und normalisiert.

Der echte Kurs, in Abhängigkeit der Rotation des iPhones, wird dann in Zeile 9 definiert. Das daraus resultierende aktuelle horizontale Sichtfeld lässt sich dann in den Zeilen 10 und 11 berechnen, wobei die Konstante `DEGREES_IN_VIEW` den Wert aus Formel 6.1 besitzt. Hierbei ist die linke Grenze des Sichtfelds der Kurs $Kurs - 67,6°/2$, definiert durch `leftAzimuth` und die rechte Grenze der Kurs $Kurs + 67,6°/2$, definiert durch `rightAzimuth`. Beide werden auf Werte zwischen $0°$ und $359°$ normalisiert. Da die POI zusätzlich einen vertikalen Kurs besitzen, muss ebenfalls ein vertikales Sichtfeld berechnet werden. Dies geschieht analog zur Berechnung des horizontalen Sichtfeldes, mit dem Unterschied, dass dabei die Daten der z-Achse des Beschleunigungssensors benötigt und Werte zwischen $-90°$ und $+90°$ berechnet werden (Zeilen 13-15).

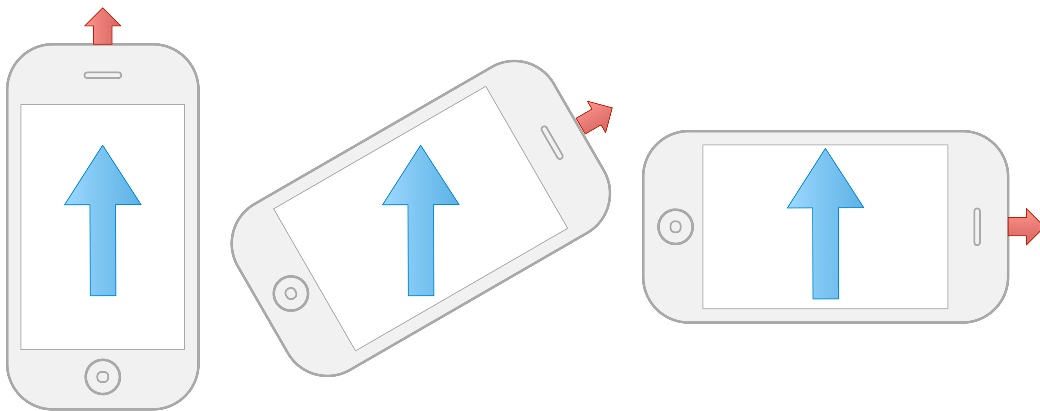


Abbildung 6.4: Anpassung der Kompassdaten an die Rotation des iPhones. Roter Pfeil ohne, blauer Pfeil mit Anpassung. Die Referenzrichtung wird an die Rotation angepasst.

6.3.3 Platzierung von POI auf der Kameraansicht

Nachdem das horizontale und vertikale Sichtfeld berechnet worden sind, muss als nächstes überprüft werden, ob sich die POI aus der Umgebung, die weiter oben gesammelt und im `AREALocationController` zwischengespeichert wurden, in diesem befinden. Im Wesentlichen funktioniert dies wie folgt. Der horizontale Kurs des POI wird mit der linken und der rechten Grenze des Sichtfeldes verglichen. Ist der Kurs des POI dann größer als die linke Grenze und kleiner als die rechte Grenze, so befindet sich der POI im horizontalen Sichtfeld. Hierbei muss aber zusätzlich eine Fallunterscheidung durchgeführt werden, da ein Kompass einen Übergang von $359°$ auf $0°$ vorweist. Das bedeutet, dass gesonderte

Betrachtungen durchgeführt werden müssen, wenn die linke Grenze des Sichtfelds größer als $291,4°$ oder die rechte Grenze kleiner als $67,6°$ ist. Diese Werte kommen durch das maximale horizontale Sichtfeld von $67,6°$ zustande. Abbildung 6.5 stellt diese Fallunterscheidung anschaulich dar. Damit sich der POI im Sichtfeld befindet, muss dieser ebenfalls im vertikalen Sichtfeld liegen. Anders als beim horizontalen Sichtfeld, wird hier keine Fallunterscheidung benötigt, und es reicht zu vergleichen, ob der vertikale Kurs des POI kleiner als die obere Grenze und größer als die untere Grenze ist. In Listing 6.11 ist das Vorgehen im Programmcode dargestellt. In den Zeilen 5 und 15 ist die Fallunterscheidung für das horizontale Sichtfeld zu sehen.

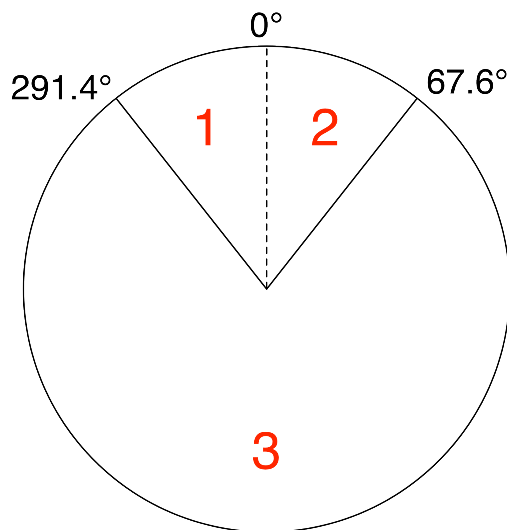


Abbildung 6.5: Die drei Fallunterscheidungen des horizontalen Sichtfelds.

Wenn sich nun ein POI im Sichtfeld befindet, müssen für ihn eine x- und eine y-Koordinate berechnet werden, damit dieser korrekt auf das Display gezeichnet werden kann. Dabei muss ebenfalls eine Fallunterscheidung durchgeführt werden. Je nach dem, ob sich der horizontale Kurs des POI in der 1., 2. oder der 3. Fläche in Abbildung 6.5 befindet, muss die Umrechnung des Kurses in eine x-Koordinate unterschiedlich vonstatten gehen. Befindet sich der POI in der 1. oder 3. Fläche, so muss einfach die Differenz zwischen dessen horizontalen Kurs und der linken Grenze berechnet und mit der Konstante `PIXEL_DEGREE` multipliziert werden (Zeilen 7 und 16). Die Konstante bestimmt, wieviele Pixel einem Grad auf dem Display entsprechen und berechnet sich aus $\frac{480}{56}$, also Displaybreite durch den echten horizontalen Blickwinkel. Durch die Differenz wird die Zahl an Grad bestimmt, um

die der POI vom linken Rand aus verschoben werden muss. Befindet sich der Punkt aber in der 2.Fläche, so wird zwischen der linken Grenze und dem horizontalen Kurs des POI der Nullpunkt übersprungen. Deswegen wird zuerst von $360°$ die linke Grenze abgezogen. Damit erhält man den Abstand der linken Grenze zum Nullpunkt. Anschließend wird der horizontale Kurs des POI addiert und mit oben genannter Konstante multipliziert. Bei der Berechnung der y-Koordinate müssen keine Fälle unterschieden werden, da das Intervall von $-90°$ bis $+90°$ reicht. Die Koordinate wird durch die Differenz der oberen Grenze mit dem vertikalen Kurs des POI und dem anschließenden Multiplizieren mit der Konstante bestimmt.

Sobald die Berechnungen abgeschlossen sind, wird `AREAViewController` darüber informiert. Dieser muss dazu das Protocol aus Listing 6.10 implementieren. Die Methode in Zeile 2 wird gerufen, wenn sich die Position des Benutzer geändert hat und die Punkte in der Umgebung neu berechnet wurden, die Methoden in den Zeilen 3-5 werden nach jeder Neuberechnung des Sichtfelds, d.h. also nach jedem Durchlauf der Anwendungsschleife vom `AREASensorController` aufgerufen. Diese liefern dem `AREAViewController` Informationen darüber, um wieviel Grad die `locationView` rotiert werden muss, welche POI wo auf dieser gezeichnet werden müssen und um wieviel Grad sich der Kompass bzw. der Radar rotiert werden müssen, dass diese auch nach einer Rotation korrekt angezeigt werden.

```
1 @protocol AREALocationControllerDelegate
2 -(void)didUpdateLocations:(NSArray *)locations inDistance:(CLLocationDistance)
      distance fromLocation:(CLLocation *)location;
3 -(void)didUpdateHeadingLocations:(NSArray *)locations;
4 -(void)didUpdateHeadingWithRotation:(double)rotation;
5 -(void)didUpdateDeviceOrientationToAngle:(double)radianAngle;
6 @end
```

Listing 6.10: AREALocationControllerDelegate

```
1  -(void)didUpdateBearingX:(double)newBearingX andBearingY:(double)newBearingY
       andBearingZ:(double)newBearingZ andHeading:(double)newHeading
2  {
3      ...
4      for(AREALocation *loc in self.surroundingLocations) {
5          if ((rightAzimuth <= DEGREES_IN_VIEW || leftAzimuth >= 360-DEGREES_IN_VIEW
               ) && loc.verticalHeading <= topAzimuth && loc.verticalHeading >=
               bottomAzimuth) {
6              if (leftAzimuth <= loc.horizontalHeading) {
7                  loc.point = CGPointMake((PIXEL_DEGREE) * (loc.horizontalHeading-
                       leftAzimuth), (PIXEL_DEGREE) * (topAzimuth - loc.
                       verticalHeading));
8                  [locations addObject:loc];
9              }
10             else if((rightAzimuth >= loc.horizontalHeading) && loc.verticalHeading
                    <= topAzimuth && loc.verticalHeading >= bottomAzimuth) {
11                 loc.point = CGPointMake((PIXEL_DEGREE) * (360 - leftAzimuth + loc.
                       horizontalHeading), (PIXEL_DEGREE) * (topAzimuth - loc.
                       verticalHeading));
12                 [locations addObject:loc];
13             }
14         }
15         else if(leftAzimuth <= loc.horizontalHeading && loc.horizontalHeading <=
               rightAzimuth && loc.verticalHeading <= topAzimuth && loc.
               verticalHeading >= bottomAzimuth){
16             loc.point = CGPointMake((PIXEL_DEGREE) * (loc.horizontalHeading-
                   leftAzimuth), (PIXEL_DEGREE) * (topAzimuth - loc.verticalHeading))
                   ;
17             [locations addObject:loc];
18         }
19     }
20     [self.delegate didUpdateDeviceOrientationToAngle:deviceRotation];
21     [self.delegate didUpdateHeadingLocations:locations];
22     [self.delegate didUpdateHeadingWithRotation:newHeading];
23 }
```

Listing 6.11: Berechnung der Koordinaten der POI auf dem Display

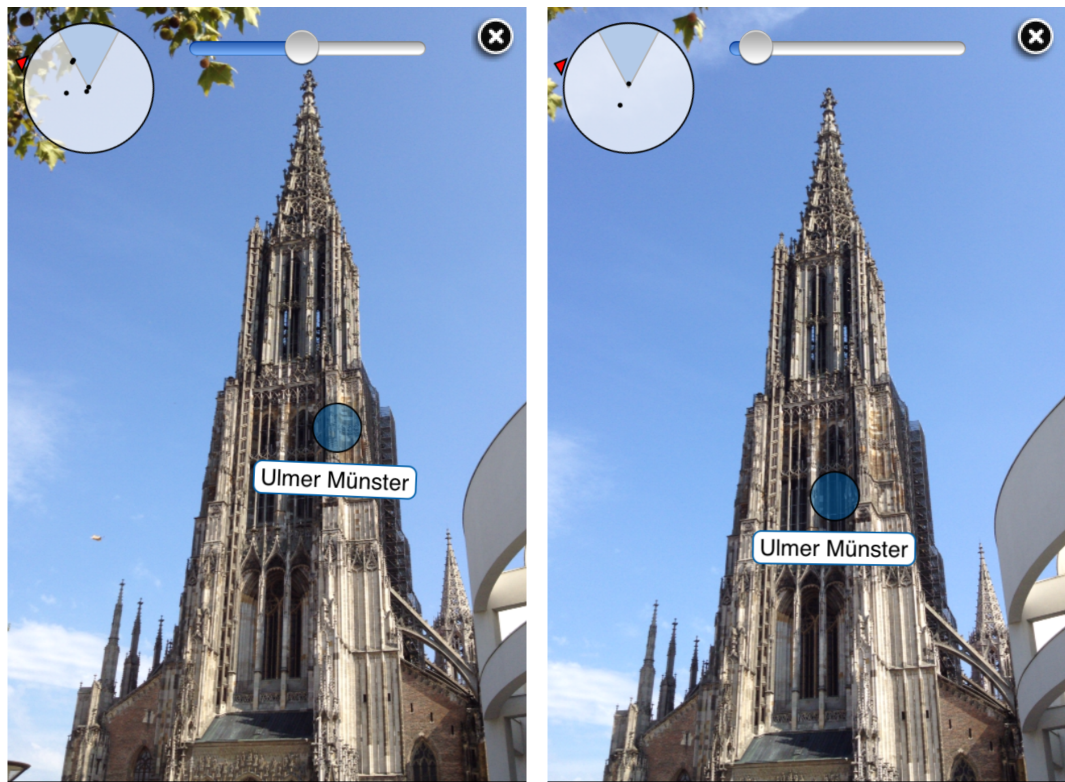
7 Vorstellung der Applikation

In diesem Kapitel wird die fertig entwickelte Engine und die darauf aufbauende Applikation vorgestellt. Dazu werden einige Screenshots aus der laufenden Anwendung abgebildet, erklärt was darauf zu erkennen ist und die Funktionen, die der Benutzer ausführen kann, vorgestellt.

Abbildung 7.1: Kartenansicht von AREA

In Abbildung 7.1 ist die Kartenansicht von AREA abgebildet. Auf der Kartenansicht werden alle zuvor hinzugefügten POI aus der Umgebung und die Position des Benutzers angezeigt. POI werden dabei als rote Stecknadeln auf die Kartenansicht gelegt. Möchte der Benutzer weitere Informationen über einen solchen POI erhalten, wie z.B. den Namen, so kann dieser einfach auf die Stecknadel tippen. Die Informationen werden anschließend als Annotation über der Stecknadel angezeigt. Die eigene Position kann ein Benutzer über den blauen Punkt einsehen. Über die Schaltfläche „Locate Me" ist es möglich, die Kartenansicht so verschieben zu lassen, sodass sich die Position des Benutzers mittig auf dem Display befindet. Gleichzeitig wird dabei auf einen fest definierten Radius (in diesem Fall 20 km)

herausgezoomt, damit der Benutzer die bestmögliche Übersicht erhält. In die eigentliche Kameraansicht kann über die Schaltfläche mit der Kamera gewechselt werden.



(a) Mittlerer Radius. (b) Kleiner Radius

Abbildung 7.2: Kameraansicht. Zu sehen sind Radar, Schieberegler für den Radius und die virtuellen Objekte (POI)

Die Kameransicht kann in Abbildung 7.2 betrachtet werden. Diese besteht aus den Teilen und Konzepten, die in den vorherigen Kapiteln vorgestellt wurden und ist das eigentliche Thema dieser Arbeit. In der oberen linken Ecke wurde der Radar samt Kompassnadel platziert. Die Kompassnadel wird als kleiner roter Pfeil dargestellt und zeigt immer Richtung Norden, sodass sich der Benutzer leichter orientieren kann. Im Radar selbst bestehen die POI aus schwarzen Punkten. Hierbei handelt es sich um die POI, die sich in einem bestimmten Radius um den Benutzer befinden. Dieser Radius kann durch den Schieberegler am oberen Bildrand eingestellt werden und hat eine Reichweite von 1 bis 20 Kilometern. Sobald durch diesen ein neuer Radius eingestellt wurde, ändern sich die angezeigten POI. Um die Orientierung des Benutzers weiter zu verbessern, wurde am Radar das Sichtfeld

der iPhone-Kamera angebracht. Dreht sich der Benutzer um die eigene Achse, so dreht sich der Kompass mit und die POI im Sichtfeld ändern sich. Die virtuellen Objekte, welche die, durch die Kamera aufgenommene, Realität erweitern und anzeigen, wo sich POI im Sichtfeld befinden, werden durch blaue Punkte und den Namen des POI dargestellt. Die POI werden auf der Kameraansicht so platziert, dass sie sowohl horizontal als auch vertikal an der richtigen Position liegen. Das bedeutet, dass das iPhone ggf. nach oben gerichtet werden muss, um höhere Punkte, wie das *Ulmer Münster* sehen zu können. In Abbildung 7.3 wird weiter gezeigt, dass die POI auch dann korrekt angezeigt werden, wenn das iPhone rotiert wird. Zu bemerken ist, dass sich das Interface ebenfalls mitdreht. Um zur Kartenansicht zurück zu wechseln, kann die Schaltfläche mit dem *X* in der rechten oberen Ecke gedrückt werden.

Abbildung 7.3: Kameraansicht im Landscape-Modus

Ähnlich wie in der Kartenansicht, ist es auch in der Kameraansicht möglich, weitere Informationen über einen POI einblenden zu lassen. Durch den Druck auf einen beliebigen sichtbaren Punkt wird dem Benutzer eine Informationsbox (Abbildung 7.4) angezeigt. Auf dieser wird die Distanz zum POI, dessen Höhe über Normalnull und weitere Informationen in Textform eingeblendet.

Abbildung 7.4: Informationsbox. Wird eingeblendet, wenn in der Kameraansicht auf einen POI gedrückt wird

8 Zusammenfassung

Das Ziel dieser Arbeit war es, eine Augmented Reality Engine von Grund auf selbst und ohne Zuhilfenahme Teile bereits existierender Engines zu entwickeln. Wie gezeigt wurde, handelt es sich dabei um ein komplexes Vorhaben. Es muss zuerst ein grundlegendes Verständnis über mathematische Berechnungen aus der Geographie vorhanden sein. Dazu gehören Formeln zur Distanzberechnung und der Berechnung des Kurses eines beliebigen Punktes relativ zum Nordpool. Weiter ist es notwendig, die unterschiedlichen Sensoren des iPhones zu kennen. Insbesondere unterscheiden sich diese anhand der Datenabfrage. So muss auf der einen Seite der Beschleunigungssensor *gepollt* werden, auf der anderen Seite aber *pushen* GPS-Sensor und Kompass ihre Daten. Ein weiterer wichtiger Punkt ist der Ressourcen- und Energieverbrauch. Da das iPhone nur eine begrenze Leistung besitzt, die POI aber ohne störendes Ruckeln auf der Display angezeigt werden sollen, mussten die Berechnungen, das Abfragen der Sensoren und das Anzeigen von POI so effizient wie möglich implementiert werden. Letzteres wurde vor allem durch das Wiederverwenden von bereits existieren POI und durch das Vergrößern des Sichtfeldes umgesetzt, da das Neuzeichnen des gesamten Displays und aller sichtbaren POI sonst zu viel Zeit in Anspruch nimmt und es in Folge eventuell zum Performance-Einbruch kommt. Die Vergrößerung des Sichtfelds brachte zudem den Vorteil, dass die Positionierung und Berechnung von POI unabhängig von der eigentlichen Ausrichtung des iPhones geschehen und zugleich eine einfache Rotation der View durchgeführt werden konnte. Die AR-Engine sollte außerdem eine hohe Modularität aufweisen, sodass es ohne großen Aufwand möglich sein soll, die Engine zu modifizieren, Anwendungen auf dieser aufzubauen oder diese selbst in existierende Anwendungen zu integrieren. Hierzu war es nötig, einen geeigneten Architektur- und Klassenentwurf zu erstellen, wobei auch die interne Kommunikation der einzelnen Komponenten nicht zu vernachlässigen war.

Tabelle 8.1 gibt einen Überblick über die definierten Anforderungen und wie gut diese in AREA umgesetzt wurden.

Tabelle 8.1: Tabellarische Aufstellung der Anforderungen.

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 iPhones auslesen (Beschleunigung, GPS, Magnetfeld)	++
Bei Bewegung des iPhones Daten und POI in Echtzeit aktualisieren	+
Einstellbarer Radius für die Entfernung	++
Radar in Kameraansicht mit POI aus der Umgebung und im Radius	++
Weitere einblenden bei Interaktion mit POI Informationen	+
Portrait und Landscape	++
Hohe Effizienz von Berechnungen	+
Hohe Effizienz beim Zeichnen der Anzeige	+
Hohe Stabilität	++
Hohe Genauigkeit	+
Gute Wartbarkeit	+
Einheitliche Spezifikation von POI	++
POI sollen intern erweiterbar sein	++
Einfache Einbindung in andere Anwendungen (Modularität)	++
Lückenlose verständliche Kommentierung	+

9 Ausblick

Die hier entwickelte AR-Engine AREA lässt sich an manchen Stellen noch verbessern bzw. erweitern.

Hierzu gehört unter anderem das Handhaben mit den Sensordaten. Sowohl GPS-, als auch der Kompass können relativ ungenau sein, wenn sich das iPhone zwischen hohen Häuserschluchten befindet. Dies fällt vor allem dadurch auf, dass der Kompass hin und her springt. Zumindest das Ruckeln des Kompasses, und dem daraus resultierenden Ruckeln der Kameraansicht, kann durch die Nutzung des internen Gyroskops verbessert werden. Dazu muss das Gyroskop mit einem Referenzvektor, in diesem Fall in Richtung des geographischen Nordens, versehen werden. Durch die Nutzung des Gyroskops werden die Kompassdaten nur noch zum Initialisieren benötigt, womit das Ruckeln abgeschaltet werden könnte.

Ein weiterer Punkt ist das Anzeigen von POI in der Kameraansicht. Wenn mehrere POI hintereinanderliegen, so ist es nur möglich, mit dem vordersten zu interagieren. Es wäre also eine Funktion denkbar, mit der eine solche Ansammlung von POI kurzzeitig auseinandergezogen werden kann. Das Gegenteil tritt ein, wenn gar kein POI auf dem Display angezeigt wird, also sich kein POI im Sichtfeld befindet. Hier könnte ebenfalls eine Funktion angeboten werden, die kurzzeitig durch Pfeile anzeigt, in welche Richtung das iPhone bewegt werden muss, um POI zu sehen. Die ist vor allem dann von Vorteil, wenn sich POI über oder unter dem Sichtfeld befinden, da der Radar nur eine Vogelperspektive anbietet und dieser das vertikale Sichtfeld ignoriert.

Zu guter Letzt fehlt bisher die Anbindung an das Internet. Die AR-Engine besitzt zwar bereits eine einheitliche Schnittstelle, um POI einfach austauschen zu können, diese wurde im Laufe dieser Arbeit jedoch nicht direkt mit anderen Systemen genutzt.

Es gibt also noch manche Dinge, die erweitert oder verbessert werden könnten. Das Ziel, eine AR-Engine voll funktionsfähig von Grund auf zu entwickeln, wurde jedoch vollständig erfüllt.

A Skizzen

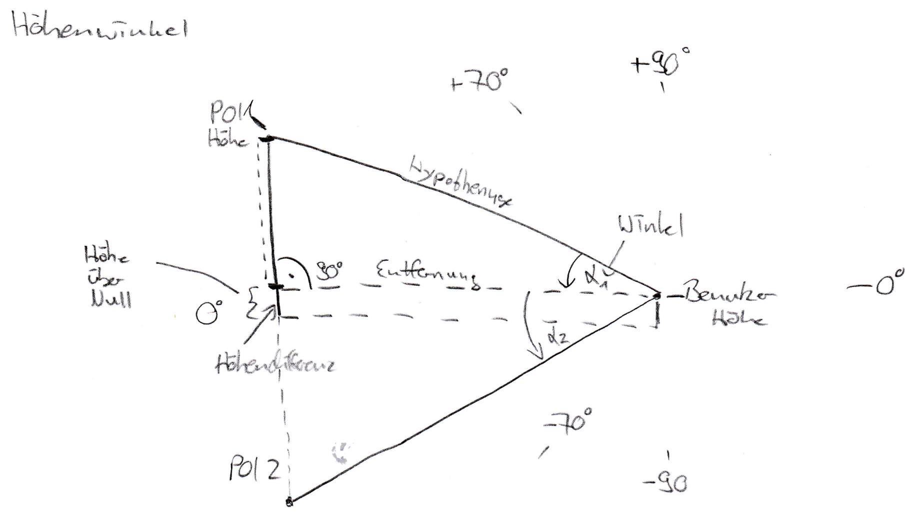


Abbildung A.1: Skizze zum Höhenwinkel

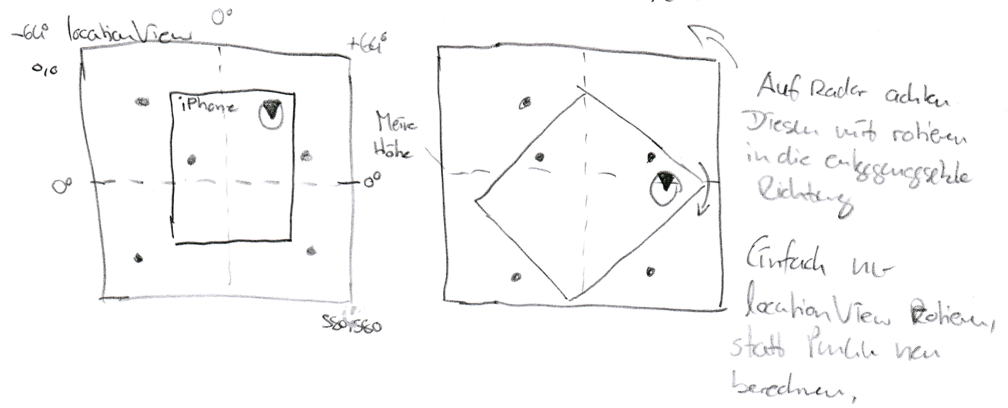


Abbildung A.2: Skizze zur locationView

Literaturverzeichnis

[1] AZUMA, R. ; BAILLOT, Y. ; BEHRINGER, R. ; FEINER, S. ; JULIER, S. ; MACINTYRE, B.: Recent advances in augmented reality. In: *Computer Graphics and Applications, IEEE* 21 (2001), Nr. 6, S. 34–47

[2] *Project Glass*. http://en.wikipedia.org/w/index.php?title=Project_Glass&oldid=507726354, . – [Online; accessed 03.09.2012]

[3] ZHOU, F. ; DUH, H.B.L. ; BILLINGHURST, M.: Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR. In: *Mixed and Augmented Reality, 2008. ISMAR 2008. 7th IEEE/ACM International Symposium on* Ieee, 2008, S. 193–202

[4] *Gyroscope*. http://en.wikipedia.org/w/index.php?title=Gyroscope&oldid=513782595, . – [Online; accessed 23.09.2012]

[5] *mixare*. http://code.google.com/p/mixare/, . – [Online; accessed 04.09.2012]

[6] *Wikitude*. http://www.wikitude.com, . – [Online; accessed 04.09.2012]

[7] *Yelp*. http://www.yelp.com, . – [Online; accessed 04.09.2012]

[8] *twitter API*. https://dev.twitter.com, . – [Online; accessed 04.09.2012]

[9] *google places API*. https://developers.google.com/places/documentation/, . – [Online; accessed 04.09.2012]

[10] SCHWARZ, F.: *Untersuchung der Augmented Reality Technik am Beispiel des Android Frameworks*, Universität Ulm, Bachelorarbeit, 2011

[11] *Radian*. http://en.wikipedia.org/w/index.php?title=Radian&oldid=509360352, . – [Online; accessed 23.09.2012]

[12] SINNOTT, R.W.: Virtues of the Haversine. In: *Sky and telescope* 68:2 (1984), S. 158

[13] BULLOCK, R: *Great Circle Distances and Bearings Between Two Locations*. http://www.dtcenter.org/met/users/docs/write_ups/gc_simple.pdf, 06 2007. – [Online; accessed 04.09.2012]

[14] VENESS, C.: *Calculate distance, bearing and more between Latitude/Longitude points*. http://www.movable-type.co.uk/scripts/latlong.html, . – [Online; accessed 04.09.2012]

[15] HOHNER, M.: *Formelsammlung Kameras*. http://www.mhohner.de/formulas.php?lg=d, . – [Online; accessed 05.09.2012]

[16] HOTPAW2: *iPhone 4 Camera Specifications - Field of View / Vertical-Horizontal Angle*. http://stackoverflow.com/questions/3594199/iphone-4-camera-specifications-field-of-view-vertical-horizontal-angle, . – [Online; accessed 05.09.2012]

[17] LUMO, F.: *Apple iPhone 4 camera specs*. http://falklumo.blogspot.de/2010/06/apple-iphone-4-camera-specs.html, . – [Online; accessed 05.09.2012]

[18] BAUTCH, M.: *Digitale bildgebende Verfahren*. http://de.wikibooks.org/wiki/Digitale_bildgebende_Verfahren:_Aufnahme, . – [Online; accessed 05.09.2012]

[19] CUJO30227: *I Need iPhone 4 Camera Specifications - Field of View / Vertical-Horizontal Angle*. http://www.iphonedevsdk.com/forum/iphone-sdk-development/57664-i-need-iphone-4-camera-specifications-field-view-vertical-horizontal-angle.html, . – [Online; accessed 05.09.2012]

[20] APPLE: *Xcode 4 Downloads and Resources - Apple Developer*. https://developer.apple.com/xcode/, . – [Online; accessed 10.09.2012]

[21] APPLE: *Event Handling Guide for iOS: Motion Events*. https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html, . – [Online; accessed 10.09.2012]

[22] ALLAN, Alasdair: *Basic Sensors in iOS: Programming the Accelerometer, Gyroscope, and More*. O'Reilly Media, 2011. – 47 S.

Name: Philip Geiger Matrikelnummer: 700561

Erklärung

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

Ulm, den ....

Philip Geiger