Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Studienarbeit Informatik

Orientierungsberechnung mittels Multisensordatenfusion auf iOS-Endgeräten

Sebastian Engel

21.08.2012

Betreuer

Jürgen Sommer
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Engel, Sebastian:
Orientierung von Objekten bei inertialer Navigation
Studienarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 10/2011 - 08/2012

Zusammenfassung

Im Rahmen eines Navigations-Programms für Bibliotheken auf mobilen Geräten der neusten Generation beschäftigt sich diese Studienarbeit mit der Orientierung (Winkellage) des Geräts. Zur genauen Navigation in Räumen genügt die Bestimmung der Position nicht. Es ist zusätzlich relevant wie das Gerät orientiert ist. Durch diese Information ist es möglich, den Benutzer direkt zu einem bestimmten Ort, im Falle einer Bibliothek ein Buch, zu führen. Im Wesentlichen befasst sich die Studienarbeit mit der Beschaffung und Berechnung der Orientierungsdaten.

Danksagung

Vielen Dank an Jürgen Sommer, Alex Decker, Achim Fritz und Stephan Doerr für die fachliche Unterstützung Martin Lahl und Philipp Wolter für die Hilfe bei der Umsetzung und Sabrina Pfeffer für den moralischen Beistand und das Korrekturlesen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

IMU	Inertial Measuring Unit
API	Application Programming Interface
GPS	Global Positioning System
MEMS	Microelectromechanical Systems
RSSI	Received Signal Strength Indication
NFC	Near Field Communication
RFID	Radio-Frequency Identification

Kapitel 1

Einleitung

Die Arbeit gliedert sich wie folgt: Zu Beginn wird in Kapitel 2 die Entwicklung der benötigten Hard- und Software betrachtet. In Kapitel 3 wird ein Konzept zur Berechnung der Orientierung eines mobilen Geräts entwickelt. Bevor sich der genauen Implementierung, am Beispiel einer Navigations-App für Bibliotheken, in Kapitel 5 gewidmet wird, werden in Kapitel 4 die für diese Herangehensweise geeignete Plattform und Werzkeuge gewählt, die zur Umsetzung benötigt werden. Die Umsetzung erfolgt auf Basis der eingebauten Sensoren. Es soll keine Zusatzhardware zum Einsatz kommen. Es folgt eine Auswertung der Ergebnisse in Kapitel 6 mit einer Diskussion. Am Ende beschließt ein Ausblick in Kapitel 7 diese Arbeit. Die Studienarbeit entwickelt eine Diplomarbeit von Achim Fritz aus dem Jahr 2011 weiter.

1.1 Motivation

Bisher findet Navigation hauptsächlich im Freien statt – beispielsweise schon seit Langem bei Navigationssystemen für Autos. Dabei wird ausschließlich GPS verwendet. Für Navigation innerhalb von Gebäuden ist GPS aber nicht brauchbar. Es ist zu ungenau und wird durch die Wände des Gebäudes noch zusätzlich ungenauer. Bei der genauen Positionsbestimmung wird es zunehmend wichtiger, auch die Orientierung zu bestimmen. Denn innerhalb von Gebäuden und bei Positionsunterschieden von wenigen Metern ist die Information, in welche Richtung man schaut, ebenfalls interessant und liefert zusätzliche Informationen. Gerade bei einer Navigations-App für Bibliotheken ist es wichtig zu wissen, welches Regal im Moment angeschaut wird. Außerdem darf die Position eine Ungenauigkeit von einigen Zentimetern nicht überschreiten. Ziel dieser Arbeit ist es die Lage des Geräts im Raum zu berechnen. Die aus den Werten der Sensoren des Geräts berechneten Werte sollen für eine bereits bestehende praxistaugliche Navigations-App für Bibliotheken die Orientierungs-Werte beisteuern.

Kapitel 2

Stand der Technik

2.1 Beschreibung der Orientierung von Objekten im dreidimensionalen Raum

Zur Beschreibung der Orientierung von Objekten im dreidimensionalen Raum in kartesischen Koordinatensystemen gibt es mehrere Möglichkeiten. Die drei am häufigsten verwendeten werden im Folgenden vorgestellt.

2.1.1 Euler-Winkel

Bei Euler-Winkeln handelt es sich um drei Winkel, die jeweils die Rotation um eine bestimmte Achse des Koordinatensystems angeben. So kann eine Transformation zwischen zwei Koordinatensystemen, dem Labor- und dem Körperfesten-System definiert werden.

Es existieren mehrere Definitionen von Euler-Winkeln, was die Reihenfolge der Drehungen um die Achsen anbelangt. Die bekannteste ist Yaw-Pitch-Roll - zu deutsch: Roll-Nick-Gier-Winkel. Dies entspricht auch der Luftfahrtnorm (DIN 9300).

- Roll (Roll-Winkel) beschreibt die Querneigung, also die Drehung um die X-Achse.
- Pitch (Nick-Winkel) besschreibt die Längsneigung, also die Drehung um die Y-Achse.
- Yaw (Gier-Winkel) beschreibt Orientierung, also die Drehung um die Z-Achse.

Bei mobilen Geräten wie dem Apple iPhone, gibt es anders als bei Fahrzeugen, keine fest definierte Ausrichtung. Beim iPhone und iPad sind die Winkel darum so verteilt wie auf Abbildung 2.1 zu sehen.

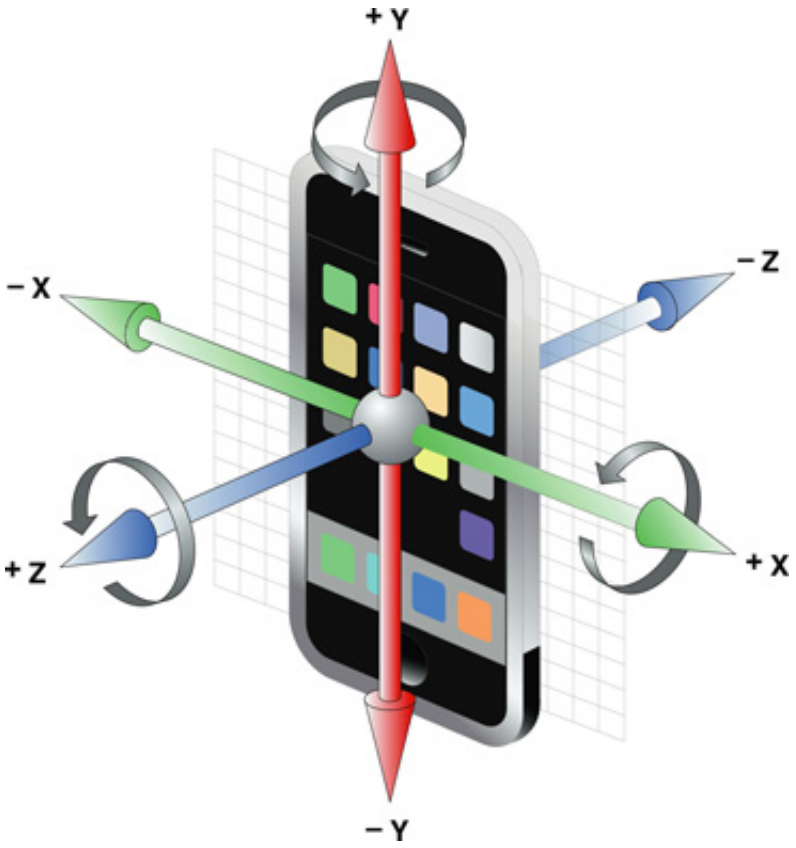

Abbildung 2.1: Roll-Pitch-Yaw [App11]

Euler-Winkel haben den Vorteil, dass sie jeder intuitiv verstehen kann. Jeder lernt Euler-Winkel in der Schule kennen. Somit kann man mit ihnen auch einfach rechnen.

Eine Drehung mit Euler-Winkeln setzt sich immer aus einer Kombination von Rotation der drei Achsen zusammen. Das heißt, eine Drehung findet nie direkt statt, sondern über mehrere nacheinander ausgeführte Rotationen der einzelnen Achsen. Das kann bei manchen Anwendungen ein Problem sein. Zum Beispiel wenn die Orientierung schneller abgefragt wird als die Teilschritte einer Drehung berechnet werden, kann es vorkommen, dass in einzelnen Key-Frames der Anwendung falsche Orientierungsdaten einfließen.

Eine Orientierungs-Angabe als Euler-Winkel würde beispielsweise wie folgt aussehen:

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} 0.016134 \\ -0.000284 \\ 1.618407 \end{pmatrix}$$

Die Werte sind in Radiant angeben. Negative Werte können zustande kommen, da die Skala von $-\pi$ bis $+\pi$ geht.

Gimbal Lock

Der große Nachteil von Euler-Winkeln ist der Gimbal Lock (engl. f. Blockade der Kardanischen Aufhängung). So nennt man es wenn zwei Achsen die selbe Drehung bestimmen. Dadurch fehlt ein Freiheitsgrad. Man kann eine bestimmte Drehung erst dann wieder durchführen, wenn man eine der beiden zusammengefallenen Achsen zurück dreht. In Abbildung 2.2 ist leicht zu erkennen, dass die innere (blau) und äußere (grün) Achse die selbe Drehung bestimmen. Es ist daher momentan nicht möglich, das Flugzeug nach vorne oder hinten zu kippen. Zuerst müsste das Flugzeug entlang der mittleren Achse um 90^o gedreht werden. [Koc08]

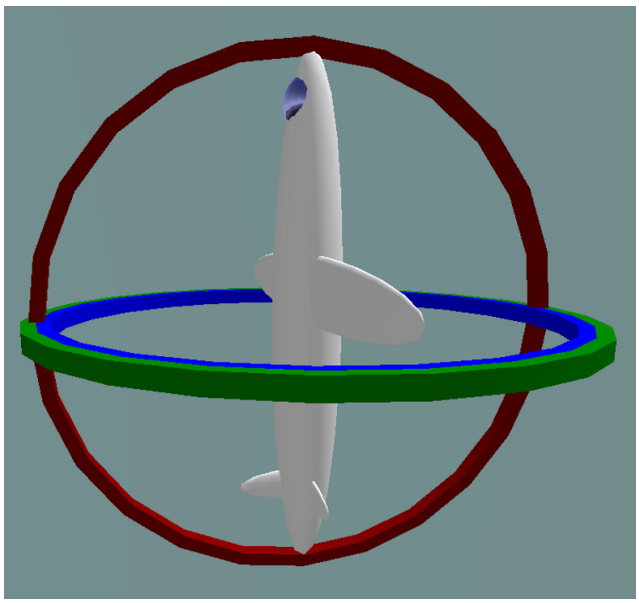

Abbildung 2.2: Gimbal Lock [Mat09]

2.1.2 Rotationsmatrizen

Eine Rotationsmatrix ist eine orthogonale Matrix, die ebenfalls die Drehung im Raum beschreibt. Sie ist als eine Hintereinanderausführung einer oder mehrerer Rotationen um beliebige Drehachsen im dreidimensionalen Raum definiert. Rotationsmatrizen sind also eine Zusammenfassung von einzelnen Rotationen mit Euler-Winkeln in ein einziges Konstrukt. Gimbal Lock kann auch mit Rotationsmatrizen auftreten. [Zel09]

Diese Rotationsmatrix entspricht einer Drehung um die x-Achse.

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

2.1.3 Quaternionen

Quaternionen sind ein mathematisches Konstrukt, um Orientierung von Objekten im dreidimensionalen Raum zu beschreiben. Sie setzen sich aus einem skalaren und einem vektoriellen Teil zusammen. Der vektorielle Teil ist allein dazu da, um die Achse der durchzuführenden Drehung zu beschreiben. Der Skalaranteil gibt den Winkel der Drehung an. Es wird also für jede Rotation eine eigene Achse konstruiert entlang der gedreht wird. Dadurch gibt es keine Zwischenschritte, sondern nur eine einzige Rotation. So kann auch das Problem des Gimbal Locks garnicht erst entstehen.

$$q = \begin{bmatrix} 0.056036 \cdot \begin{pmatrix} 0.013658 \\ -0.980339 \\ 0.188705 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} w \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \end{bmatrix}$$

Ein Quaternion sieht zum Beispiel so aus, wobei w der oben angesprochene skalare Teil ist. [Koc08]

2.2 Positionsbestimmung

Die Positionsbestimmung erfolgt in diesem Fall über Bluetooth. Dazu werden in dem Raum, in dem man navigieren möchte, Bluetooth-Sender (Beacons) ausgelegt. Diese sollten möglichst gleichmäßig verteilt sein, damit eine gleichmäßige Bluetooth-Abdeckung gewährleistet ist. Die App weiß, wo sich die einzelnen Beacons im Raum befinden und empfängt die RSSI-Werte der ausgelegten Beacons und kann so die Position des Geräts bestimmen.

Dabei können mehrere Probleme auftreten. Das größte ist die Senderate der Beacons. Herkömmliche Bluetooth-Sticks sind dafür gemacht, eine ständige Verbindung zu einem Gerät aufrechtzuerhalten um Daten zu übertragen. Hier sollen keine Daten übertragen werden, sondern nur möglichst oft der RSSI-Wert der einzelnen Beacons erfahren werden. Normale Bluetooth-Sticks schaffen meistens nur eine Rate von ca. drei Sekunden. Das ist zu wenig um mit wenigen Sticks eine zuverlässige Navigation zu realisieren. Man muss entweder in relativ teure (über 100€) Bluetooth-Sender investieren die schnell sind, oder viele von den langsameren Sendern auslegen.

Ein weiteres Problem ist, dass Bluetooth leicht gestört werden kann. Personen, Wände und Bücherregale sind ein Problem bei Bluetooth. Darum kann man nicht sicher sein, dass die RSSI-Werte, die beim Gerät ankommen, die entsprechende Entfernung repräsentieren.

Um diesen beiden Hauptproblemen entgegenzuwirken, wird ein Partikelfilter eingesetzt. Mit dem Partikelfilter wird eine Wolke gewichteter Partikel erzeugt, die den aktuellen Aufenthaltsort schätzt. Anhand der aktuellsten Position, die

aus den Bluetooth-RSSI-Werten berechnet wurde, werden die einzelnen Partikel gewichtet. So kann die Ungenauigkeit der Bluetooth-Werte etwas korrigiert werden. [RAG04]

Kapitel 3

Konzept

Um die Orientierung umsetzen zu können braucht man zwei Werte: Azimut und Elevation. Azimut ist die Drehung in der horizontalen und Elevation die in der vertikalen Ebene. In Euler-Winkeln gesprochen entspricht Azimut Yaw und Elevation Pitch.

3.1 Accelerometer

Das Accelerometer ist ein Beschleunigungssensor. Er misst die Beschleunigung in alle drei Richtungen. Nach unten wirkt immer die Erdbeschleunigung von $9,81 ms^2$. Mit geeigneten Filter-Verfahren, wie dem Tiefpassfilter, kann man die Gravitation von der durch den Benutzer verursachten Beschleunigung isolieren. Mit dem ständig bekannten Gravitationswert ist es möglich die Lage des Geräts zu bestimmen. Jedoch sind diese Berechnungen nicht sehr genau. Das liegt zum einen an der Schwierigkeit der Trennung zwischen Gravitation und durch den Benutzer verursachten Beschleunigung und zum anderen an der Ungenauigkeit der in mobilen Geräten verbauten Sensoren. Das verursacht zusammen nach wenigen Sekunden einen so großen Fehler, dass das Accelerometer zu Berechnung der Orientierung nicht in Frage kommt.

Jedoch fließt das Accelerometer in die Gesamtberechnung trotzdem ein, denn weil das Accelerometer immer nach unten ausgerichtet ist und sich auch immer selbst wieder korrigiert, kann es zur Stabilisierung des Gyroskops verwendet werden.

In Abbildung 3.1 ist der Kern eines 3-Achsen-Accelerometers zu sehen.

3.2 Gyroskop

Das Gyroskop ist ein Drehratensensor. Es ermittelt die Winkel einer Drehung des Geräts entlang aller drei Achsen. Daraus kann sehr präzise und vor allem

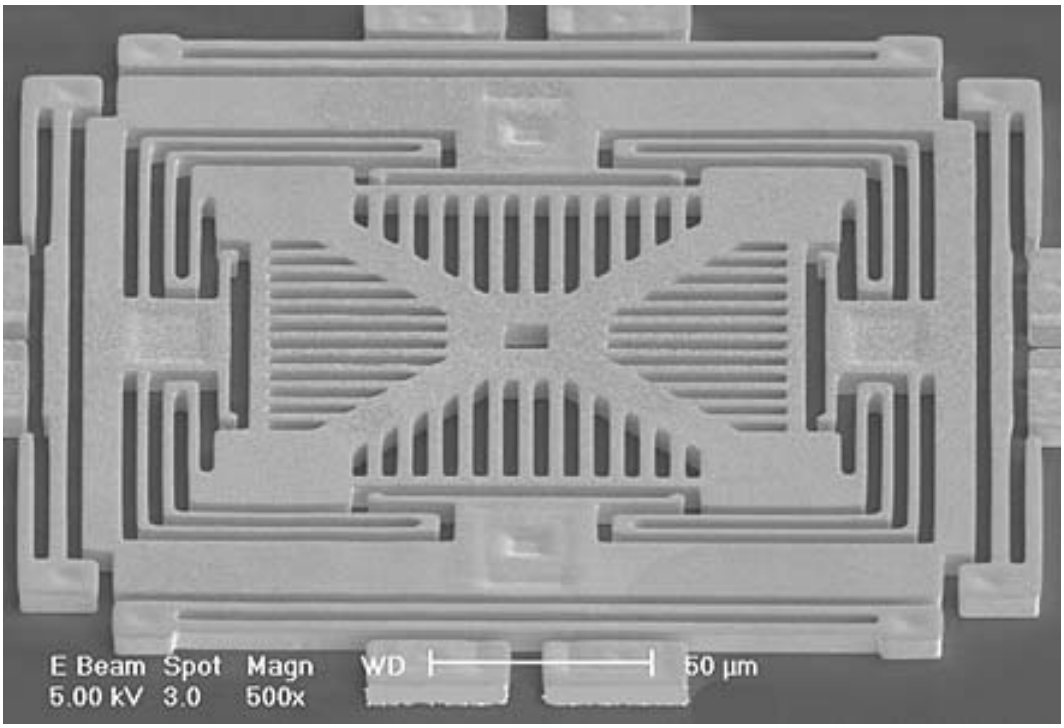

Abbildung 3.1: Accelerometer [Gui12]

flüssig die Orientierung berechnet werden. Die Orientierungs-Angabe, die man aus den Gyroskop-Daten gewinnt, ist relativ zur Position des Geräts bei Beginn der Messung. Die Gyroskop-Werte von mobilen Geräten haben meist einen merkbaren Drift. Dieser hat aber den Vorteil, dass er sehr konstant ist.

3.3 Kompass

Der Kompass bestimmt anhand des Magnetfelds der Erde die magnetische Nordrichtung. Daraus können auch alle anderen Himmelsrichtungen bestimmt werden. Mit dem Kompass kann ausschließlich Azimut bestimmt werden. Der Kompass ist immer auf Norden ausgerichtet, er richtet sich immer wieder selbst aus. So erhält man den absoluten Azimut-Wert. Der Kompass ist anfällig auf Störungen durch jegliche Magnetfelder in seiner Umgebung. Bei mobilen Geräten stellen besonders elektromagnetische Felder, wie sie im Alltag durch viele elektrische Geräte verursacht werden, ein Problem dar. Teilweise liefert der Kompass sprunghafte Werte, die nicht für eine flüssige Darstellung der Orientierung verwendet werden können.

Abbildung 3.2: Gyroskop [App12b]

3.4 Multisensordatenfusion

Keiner der gängigen Sensoren lässt sich alleine zur Berechnung der Orientierung verwenden.

Da das Gyroskop keine absoluten Winkel liefert müssen die Gyroskop-Werte, bevor man sie verwendet, einen Startwert erhalten. Auf diesen werden die weiteren Drehungen addiert. Bei Azimut nimmt man als Startwert den Kompass-Wert. Bei Elevation wird der Startwert mit Hilfe des Accelerometers berechnet. Das Accelerometer liefert auch gleichzeitig die Stabiliserung der Elevation. So kann der Drift des Gyroskops zuverlässig vermieden werden. Bei Azimut wird der Kompass als Startwert-Geber herangezogen. Zur Stabilisierung fließt der Wert des Kompasses leicht in den Gyroskop-Wert ein. Dies geschieht mit folgender Formel:

$$f(\alpha, \phi) = (\alpha * updatedAzimut + \phi * azimutCompass)/(\alpha + \phi)$$

Wobei *azimut* der jeweils neue Kompass-Wert ist und *updatedAzimut* der alte Azimut-Wert auf den die Drehung, ermittelt durch das Gyroskop, addiert wur-

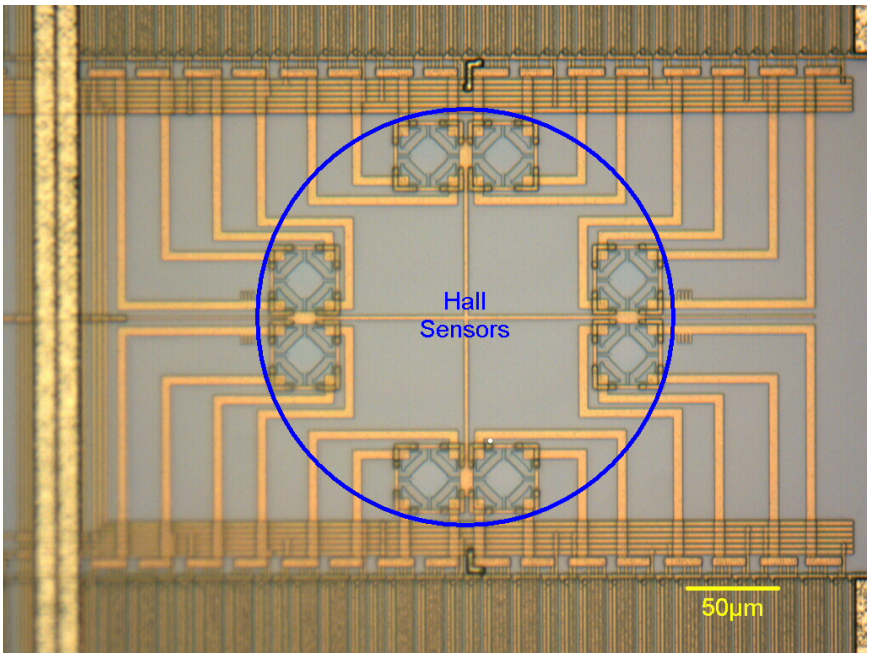

Abbildung 3.3: Kompass [DW11]

de. α und ϕ sind die Gewichtungen von $azimut$ und $updatedAzimut$. Schaubild 3.4 ist eine visuelle Darstellung dieser Vorgehensweise.

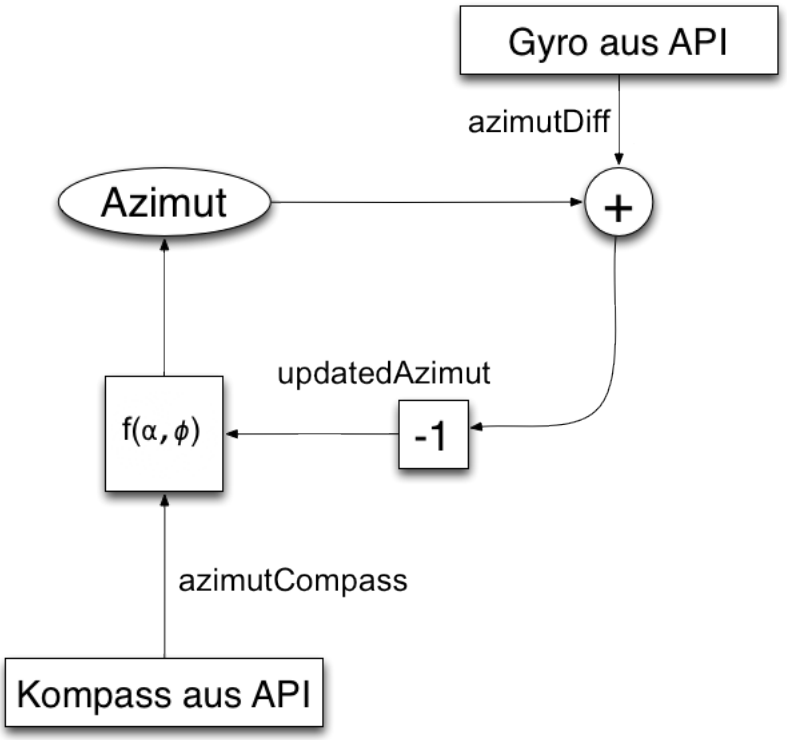

Abbildung 3.4: Konzept Multisensordatenfusion

Kapitel 4

Plattform und Werkzeuge

4.1 Plattform

Bei der Suche nach einem passenden Gerät kamen mehere Kriterien zum Tragen. Es sollte wegen der Visualisierung größer als ein Smartphone sein, aber trotzdem portabler als ein herkömmliches Notebook. Es stand also fest, dass ein Tablet am besten geeignet ist für diese Art Anwendung. Desweiteren muss das Gerät mit den oben beschriebenen Sensoren, Accelerometer, Gyroskop und Kompass ausgestattet sein.

4.1.1 Überblick am Markt befindlicher Geräte

Wirklich am Markt vertreten waren zum Zeitpunkt der Hardware-Entscheidung (Anfang 2011) nur das Apple iPad 1 und das Motorola Xoom. Das iPad war mit Accelerometer und Kompass ausgestattet, jedoch nicht mit einem Gyroskop. Das Motorola Xoom hatte alle drei IMUs verbaut. Android Version 3.0 Honeycomb erschien im Februar 2011 und war die erste Android-Version, die für Tablets ausgelegt war. [Yad11] Allerdings war diese Version des Betriebssystems anfangs Berichten zufolge instabil.

4.1.2 Wahl iPad 2

Da das iPad 2 in den Startlöchern stand entschieden wir, unsere Entscheidung noch aufzuschieben. Am 11. März 2011 wurde es vorgestellt und die Ausstattung entsprach unseren Anforderungen, da zusätzlich ein Gyroskop verbaut wurde. Zum Erscheinungszeitpunkt war Apple auch der Hersteller mit der meisten Erfahrung. Das iPad der ersten Generation war bereits ein Jahr auf dem Markt und hatte mit iOS ein ausgereiftes Betriebssystem. iOS wird auf dem Apple iPhone schon seit 2007 verwendet. Zudem war das iPad damals

der unangefochtene Marktführer. Damit war auch gewährleistet, dass für eine eventuell entstehende App für Endanwender genügend potentielle Abnehmer bereit stünden.

4.2 Werkzeuge

4.2.1 Frameworks

Zur Erstellung von iOS-Programmen steht Entwicklern Cocoa Touch von Apple zur Verfügung. Cocoa Touch ist eine Sammlung von Frameworks, die Entwickler bei der Programmierung unterstützen sollen. Grob lassen sich drei Arten von Frameworks, die in Cocoa Touch enthalten sind, nennen:

- Funktionen der Hardware
- Design-Elemente und Animationen
- Verarbeitung von Daten

Cocoa Touch ist in Objective-C implementiert. Somit wird in diesem Projekt fast ausschließlich Objective-C verwendet. [App12a]
Bei iOS sind für die Berechnung der Orientierung vor allem zwei Frameworks wichtig, die beide zum ersten Punkt der obigen Liste gehören: Core Location um den Kompass und Core Motion um das Gyroskop und das Accelerometer auszulesen.

Core Motion

Core Motion liefert Daten die, mit Bewegung zu tun haben. Das sind einerseits die rohen Daten aller drei Sensoren, die in Kapitel 3 beschrieben wurden, andererseits stellt Core Motion aber auch bereits bereinigte Bewegungs-Daten zur Verfügung. Zum Beispiel lassen sich Beschleunigung und Gravitation getrennt auslesen. Core Motion nimmt auch bereits die Stabilisierung des Elevation-Werts des Gyroskop vor. Besonders interessant ist die Klasse `CMAttitude` denn sie beinhaltet die Orientierung des Geräts zum Zeitpunkt der Abfrage. `CMAttitude` stellt diese in allen drei in Kapitel 2 beschriebenen Formen zur Verfügung.

Core Location

Core Location enthält alle Informationen, die zur Bestimmung der aktuellen Position und der Ausrichtung des Geräts nötig sind. Core Location ist auch

in der Lage, aus einem Geocode die zugehörige Stadt zu ermitteln und anders herum. Das `CMHeading`-Objekt stellt den aktuellen Kompass-Wert bereit.

4.2.2 Unity

Unity ist eine Spiele-Engine zur Entwicklung von Spielen für verschiedene Plattformen. Darunter sind PC-Betriebssyteme, mobile Betriebssyteme und Browser, aber auch herkömmliche Spiele-Konsolen. Unity ist neben der Unreal Engine eine der beliebtesten Spiele-Engines. In Unity kann man komplette 3D-Welten erstellen und es erlaubt den Export des Projekts in ein für die Zielplattform passendes Format. Im Falle von iOS wird ein Xcode Projekt erstellt. [Tec12]

4.2.3 Xcode

Xcode ist die Entwicklungsumgebung für iOS-Anwendungen – hier wird der Quellcode geschrieben und das Interface angeordnet. Außerdem kann die App, an der momentan gearbeitet wird, mit Xcode direkt auf ein angeschlossenes iOS-Gerät kompiliert werden. Am Ende wird mit Xcode auch die Datei zur Einreichung in den AppStore erstellt. [App12b]

Kapitel 5

Umsetzung

5.1 Unity-Integration

Als Ausgangspunkt liegt eine voll funktionsfähige Navigations-App für Bibliotheken vor. Sie wurde komplett in Unity umgesetzt. Die Einbindung der Orientierungsberechnung kann nicht direkt in Unity vorgenommen werden, da in Unity nur ein eingeschränkter Zugang auf die Cocoa-Frameworks, die die APIs zum Zugriff auf die Hardware zur Verfügungs stellen, möglich ist. Darum wird die App ohne den Hardwarezugriff als Xcode Projekt exportiert. Dann muss in Xcode ein Plug-in programmiert werden, das als Schnittstelle zwischen selbst geschriebenem Objective-C-Code in Xcode, der die Hardware ausließt, und Unity dient. Dazu muss erst noch in Unity das Plug-in als Eingabe-Skript erstellt werden. Dieses Eingabe-Skript ist in C# geschrieben.

```
[DllImport ("__Internal")]
private static extern CMQuaternion getDeviceMotion();
```

Listing 5.1: Plug-in in Unity

In Listing 5.1 wird also die externe Methode `getDeviceMotion()` aufgerufen. Das Plug-in erwartet den Datentyp `CMQuaternion`. Später wird ein Quaternion übergeben, also immer die aktuelle absolute Orientierung. `CMQuaternion` ist eigentlich ein Datentyp auf dem Core Motion Framework und somit Unity nicht bekannt. Darum muss erst noch ein `struct` mit der Bezeichung `CMQuaternion` definiert werden (Listing 5.2).

Alles weitere erfolgt jetzt direkt in Xcode. In Xcode wird eine `*.mm`-Datei angelegt zusammen mit der zugehörigen `*.h`-Datei. In der `*.mm`-Datei erfolgt die ganze Implementierung. Damit in der bestehenden App überhaupt Daten,

```
public struct CMQuaternion {
  public double x, y, z, w;
}
```

Listing 5.2: Struct `CMQuaternion` in C#

die hier berechnet werden ankommen, muss als erstes die `getDeviceMotion()`-Methode implementiert werden:

```
static GyroscopeData* delegateObject = nil;

extern "C" {

  CMQuaternion getDeviceMotion () {

    if (delegateObject == nil) {
      delegateObject = [[GyroscopeData alloc] init];
    }

    return [delegateObject getOrientation];
  }
}
```

Listing 5.3: Methode `getDeviceMotion`

Allerdings muss in dieser `extern C` -Umgebung in C geschrieben werden. In C sind aber die API-Aufrufe nicht oder nur sehr umständlich möglich. Darum wird in Zeile 11 in Listing 5.3 eine weitere Methode `getOrientation` aufgerufen.

```
- (CMQuaternion)getOrientation {
  ...
  ...
}
```

Listing 5.4: Methode `getOrientation`

Diese Methode (Listing 5.4) enthält den eigentlichen Objective-C-Quellcode. In ihr können alle API-Aufrufe problemlos ausgeführt werden.

In Abbildung 5.1 wird der Zusammengang der Methoden und Dateien nochmal schematisch dargestellt.

[Tec12]

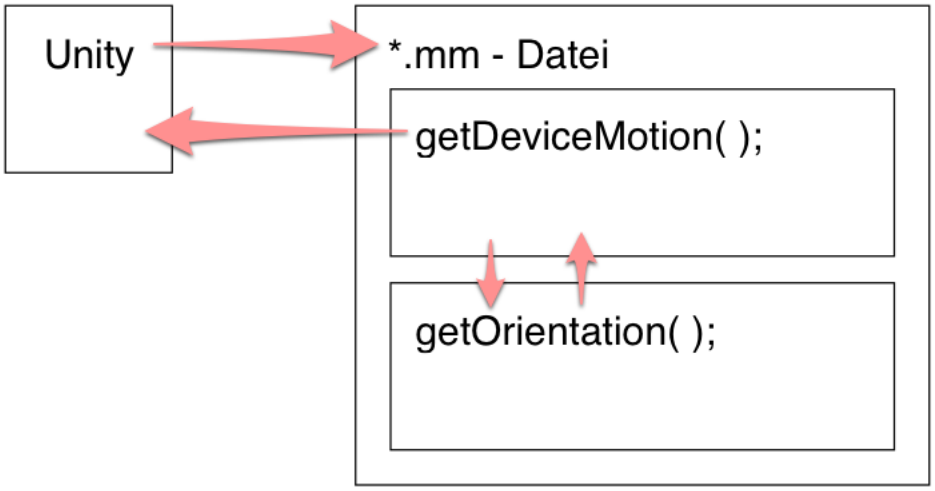

Abbildung 5.1: Unity-Integration

5.2 Vorbereitung der nötigen Daten

Um die Multidatenfusion durchführen zu können, müssen erst alle nötigen Daten ausgelesen werden. Dazu müssen `CMMotionManager`- und `CLLocationManager`-Objekte initialisert werden. [CH10] [All11]

```
// Set up locationManager
if (locationManager == nil) {
  locationManager=[[CLLocationManager alloc] init];
  locationManager.desiredAccuracy = kCLLocationAccuracyBest;
}

// Set up motionManager
if (motionManager == nil) {
  motionManager = [[CMMotionManager alloc] init];
  motionManager.deviceMotionUpdateInterval = 1.0/60.0;
}
```

Listing 5.5: `locationManager` und `motionManager` initialisieren [App12b]

In Zeile 4 in Listing 5.5 wird die Genauigkeit des `CLLocationManager`-Objekts und in Zeile 10 die Update-Frequenz des `CMMotionManager`-Objekts eingestellt. Der Kompass-Wert, der aus dem `CLLocationManager`-Objekt ausgelesen wurde, muss sehr genau sein.

Das Auslesen des Kompass-Werts findet eventgesteuert statt. Ein neuer Wert wird nur dann ausgelesen, wenn er sich vom alten Wert unterscheidet. Dazu wird die minimale Winkeländerung auf 1^o festgesetzt.

Mit dem Aufrufen der Methode `startUpdatingHeading` in Zeile 4 in Listing

```
// Start listening to events from locationManager
if ([CLLocationManager headingAvailable]) {
  locationManager.headingFilter = 1;
  [locationManager startUpdatingHeading];
}
```

Listing 5.6: `locationManager` starten [App12b]

5.6 wird hier auch gleichzeitig das eventgesteuerte Abfragen des Kompass-Werts gestartet.

Die Methode, die auf die Kompass-Änderungen (Listing 5.7) hört, liest den Kompass-Wert aus und stellt ihn in einer globalen Variable zur Verfügung.

```
- (void)locationManager:(CLLocationManager *)manager didUpdateHeading:(
      CLHeading *)newHeading {
  // Get new heading
  mHeading = newHeading.magneticHeading;

  //location specific offset depending on the 3D model
  locationOffset = 90;
  mHeading += locationOffset;

  if (mHeading > 360) {
    mHeading -= 360;
  }
  else if (mHeading < 0) {
    mHeading += 360;
  }
}
```

Listing 5.7: Azimut ermittelt durch Kompass

Es kann vorkommen, dass in dem 3D-Modell des Raums nicht an der selben Stelle Norden ist wie in der Realität an diesem Ort. Darum muss, wenn dieser Fall auftritt, ein Offset zu dem ausgelesenen Wert addiert werden. Das Resultat kann ein Wert sein, der entweder größer als 360^o oder kleiner als 0^o ist. Der Wert muss dann normalisiert werden indem 360^o subtrahiert oder addiert werden. Das Ergebnis ist der Azimut-Wert, ermittelt durch den Kompass.

Die Azimut-Änderung, ermittelt durch das Gyroskop und die Elevation, liefert das `CMMotionManager`-Objekt.

Im `CMDeviceMotion`-Objekt werden Messungen des Accelerometers und des Gyroskops zusammengefasst. Der `motionHandler` wird immer dann aufgerufen, wenn es Bewegungs-Daten des Geräts zu verarbeiten gibt. Hier ist das alle 1/60 Sekunden der Fall, weil das bei der Initialisierung des `CMMotionManager`-Objekts so festgelegt wurde.

Das Auslesen der eigentlichen Orientierungsdaten erfolgt in Zeile 6 von Listing 5.8. Wobei in diesem `CMAttitude`-Objekt alle drei Beschreibungs-Möglichkei-

```
if(motionManager.isDeviceMotionAvailable) {

  // Listen to events from the motionManager
  motionHandler = ^ (CMDeviceMotion *motion, NSError *error) {

  CMAttitude *currentAttitude = motion.attitude;
  .
  .
  .
}
```

Listing 5.8: Bewegungsdaten auslesen [App12b]

ten, Euler-Winkel, Rotationsmatrix und Quaternion, zusammengefasst sind.

```
quaternion = currentAttitude.quaternion;

if (oldQuaternion.w != 0 || oldQuaternion.x != 0 || oldQuaternion.y != 0 ||
        oldQuaternion.z != 0){
  diffQuaternion = [self multiplyQuaternions:[self inverseQuaternion:
          oldQuaternion] :quaternion];
  diffQuaternion = [self normalizeQuaternion:diffQuaternion];
}
oldQuaternion = quaternion;
```

Listing 5.9: Azimut-Änderung berechnen

Die Orientierung wird als Queternion ausgelesen und die Differenz zur letzten Orientierung gespeichert. Dies wird erreicht indem der inverse Quaternion des alten Werts mit dem Quaternion des neuen Werts multipliziert wird. Danach muss das Ergebnis noch normalisiert werden. Das geschieht in Listing 5.9 mit Hilfe der vier Methoden `quaternionMagnitude` (Listing 5.10), `inverseQuaternion` (Listing 5.11), `multiplyQuaternions` (Listing 5.12) und `normalizeQuaternion` (Listing 5.13).

```
- (float) quaternionMagnitude:(CMQuaternion)inputQuaternion {
  float magnitude = sqrt(inputQuaternion.w*inputQuaternion.w + inputQuaternion
          .x*inputQuaternion.x + inputQuaternion.y*inputQuaternion.y +
          inputQuaternion.z*inputQuaternion.z);

  return magnitude;
}
```

Listing 5.10: Methode `quaternionMagnitude`

Die interessanten Werte, Azimut und Elevation, werden aus dem Quaternion in Grad berechnet. Die Methoden `azimutFromQuaternion` (Listing 5.14) und `elevationFromQuaternion` (Listing 5.15) berechnen den entsprechenden

```
- (CMQuaternion) inverseQuaternion:(CMQuaternion)inputQuaternion {
  float magnitude = [self quaternionMagnitude:inputQuaternion];

  quaternion.w = inputQuaternion.w/magnitude;
  quaternion.x = -inputQuaternion.x/magnitude;
  quaternion.y = -inputQuaternion.y/magnitude;
  quaternion.z = -inputQuaternion.z/magnitude;

  return quaternion;
}
```

Listing 5.11: Methode `inverseQuaternion`

```
- (CMQuaternion) multiplyQuaternions:(CMQuaternion)quaternionA:(CMQuaternion)
      quaternionB {
  quaternion.w = quaternionA.w*quaternionB.w - quaternionA.x*quaternionB.x -
        quaternionA.y*quaternionB.y - quaternionA.z*quaternionB.z;
  quaternion.x = quaternionA.w*quaternionB.x + quaternionA.x*quaternionB.w -
        quaternionA.y*quaternionB.z + quaternionA.z*quaternionB.y;
  quaternion.y = quaternionA.w*quaternionB.y + quaternionA.x*quaternionB.z +
        quaternionA.y*quaternionB.w - quaternionA.z*quaternionB.x;
  quaternion.z = quaternionA.w*quaternionB.z - quaternionA.x*quaternionB.y +
        quaternionA.y*quaternionB.x + quaternionA.z*quaternionB.w;

  return quaternion;
}
```

Listing 5.12: Methode `multiplyQuaternions`

Winkel in Radian.

Bei der Elevation muss noch eine Korrektur von 90^o vorgenommen werden. Denn wenn man das iPad mit dem Display nach oben um 90^o neigt, sodass das Display zum Betrachter zeigt, befindet sich standardmäßig genau in dieser Position der Sprung von 0^o auf 360^o. Um eventuellen Problemen mit dieser Tatsache aus dem Weg zu gehen, wird der Sprung um 90^o nach oben verschoben. Dann tritt er nur auf wenn der Betrachter das iPad direkt an die Decke hält. [Koc08] [Mat12]

```
- (CMQuaternion) normalizeQuaternion:(CMQuaternion)inputQuaternion {
  float magnitude = [self quaternionMagnitude:inputQuaternion];

  quaternion.w = inputQuaternion.w / magnitude;
  quaternion.x = inputQuaternion.x / magnitude;
  quaternion.y = inputQuaternion.y / magnitude;
  quaternion.z = inputQuaternion.z / magnitude;

  return quaternion;
}
```

Listing 5.13: Methode `normalizeQuaternion`

```
- (float) azimutFromQuaternion:(CMQuaternion)quaternion {
  float azimut = atan2(2*(quaternion.w*quaternion.z+quaternion.x*quaternion.y)
          , 1 - 2*(quaternion.y*quaternion.y+quaternion.z*quaternion.z));
  return azimut;
}

azimutDiff = RADIANS_TO_DEGREES([self azimutFromQuaternion:diffQuaternion]);
```

Listing 5.14: Azimut-Wert aus Quaternion berechnen

```
- (float) elevationFromQuaternion:(CMQuaternion)quaternion {
  float elevation = atan2(2*(quaternion.w * quaternion.x + quaternion.y *
          quaternion.z), 1-2 * (quaternion.x * quaternion.x + quaternion.y *
          quaternion.y));
  return elevation;
}

elevation = -[self elevationFromQuaternion:quaternion];
elevation += M_PI/2;
elevation = RADIANS_TO_DEGREES(elevation);
```

Listing 5.15: Elevation-Wert aus Quaternion berechnen

5.3 Multisensordatenfusion

Nur der Azimut-Wert muss noch selbst berechnet werden. Das Core Motion-Framework liefert bereits einen mit dem Accelerometer stabilisierten Elevation-Wert. Darum weist der in Listing 5.15 berechnete Elevation-Wert keinen bemerkbaren Drift mehr auf.

Nun kann die Formel aus Kapitel 3.4 umgesetzt werden.

```
updatedAzimut = updatedAzimut - azimutDiff;

float alpha = 19.0;
float phi = 1.0;

//fusionate gyro estimated heading with new magneticHeading
updatedAzimut = (alpha* updatedAzimut + phi*heading)/(alpha+phi);
```

Listing 5.16: Eigentliche Datenfusion

In Zeile 1 aus Listing 5.16 wird die Azimut-Änderung auf den vorherigen Azimut-Wert angewendet. Zeile 7 ist die genaue Umsetzung der Formel aus Kapitel 3.4. `updatedAzimut` ist der zur Drehung verwendete Azimut-Wert korrigiert um die Drehung seit dem letzten Wert. `heading` ist der durch den Kompass bestimmte Azimut-Wert. Mit den beiden Steuerungsvariablen `alpha` und `phi` wird gesteuert welche Anteile die jeweiligen Komponenten der Fusion haben.

```
rotation = [self createFromAxisAngle:0 :elevation :DEGREES_TO_RADIANS(
        updatedAzimut)];
return rotation;
```

Listing 5.17: Generierung des Endergebnisses

```
- (CMQuaternion) createFromAxisAngle:(double)roll:(double)pitch:(double)yaw {
  CMQuaternion quaternion;

  float num9 = roll * 0.5f;
  float num6 = (float) sin((double) num9);
  float num5 = (float) cos((double) num9);
  float num8 = pitch * 0.5f;
  float num4 = (float) sin((double) num8);
  float num3 = (float) cos((double) num8);
  float num7 = yaw * 0.5f;
  float num2 = (float) sin((double) num7);
  float num = (float) cos((double) num7);
  quaternion.w = ((num * num3) * num5) + ((num2 * num4) * num6);
  quaternion.x = ((num * num4) * num5) + ((num2 * num3) * num6);
  quaternion.y = ((num2 * num3) * num5) - ((num * num4) * num6);
  quaternion.z = ((num * num3) * num6) - ((num2 * num4) * num5);

  return quaternion;
}
```

Listing 5.18: Generierung eines Quaternions aus Euler-Winkeln

Mit Listing 5.17 wird das Ergebnis zusammengefasst. Aus `elevation` und `updatedAzimut` wird mittels der Methode `createFromAxisAngle` (Listing 5.18) der Quaternion `rotation` generiert. Er ist die Rückgabe dieser Funktion und wird vom Plug-in in Unity erwartet.

5.4 Test-App

Um die folgenden Messungen und Grafiken für den Ergebnis-Teil zu ermitteln, wurde eine eigene Test-App erstellt. Diese teilt sich mit der Navigations-App den selben Quellcode bezüglich der Orientierung. Allerdings besteht die Test-App nur aus dem Code, der für die Orientierungsberechnungen wichtig ist, erweitert um eine Funktion, die die berechneten Werte in eine SQLite-Datenbank schreibt. Zusätzlich zeigt die App die drei relevanten Werte auch auf dem Display zur Kontrolle an. Immer wenn die App gestartet bzw. aufgerufen wird, wird eine neue Messung gestartet. Falls eine SQLite-Datei von einer vorherigen Messung noch vorhanden ist, wir diese überschrieben. Das Beenden der App durch Drücken des Home-Buttons beendet auch die Aufzeichnung der Werte. Die SQLite-Datei kann nun bequem aus iTunes heraus auf einem Computer zur Weiterverabreitung gespeichert werden.

Abbildung 5.2: Test-App

Kapitel 6

Ergebnis

Eine funktionierende Orientierung lässt sich bereits mit dem Azimut-Wert des Kompasses und dem Elevation-Wert, den das Gyroskop liefert, realisieren. Allerdings ist diese Anordnung sehr anfällig auf Deviation. Deviation ist eine Ablenkung eines Magnetkompasses, die mit elektromagnetischen Feldern in der Umgebung des Kompasses zusammenhängt. Außerdem sind die Werte des Kompasses, vor allem in älteren iOS Versionen, sehr sprunghaft.

Eine weitere funktionierende Möglichkeit besteht darin, den Kompass nur für den initialen Azimut-Wert heranzuziehen. Alle weiteren Änderungen können vom Gyroskop bestimmt werden. Jedoch hat sich in Tests ergeben, dass der Drift des Gyroskops so stark ist, dass nach wenigen Sekunden eine inakzeptable Abweichung vom korrekten Wert errechnet wird. Eine Idee war, den Gyroskop-Wert an den Kompass-Wert anzugleichen wenn das Gerät eine bestimmte Zeit lang in Ruhe war. Jedoch entsteht so immer ein Sprung wenn die Angleichung stattfindet. In der fertigen App würde das einen plötzlichen Bildausschnittssprung zur Folge haben. Dieser wäre für den Benutzer nicht nachvollziehbar.

In Grafik 6.1 wird der Azimut-Wert des Kompasses in Grün angegeben. Die Messung wurde in einer Umgebung durchgeführt, in der keine Magnetfeld-Störungen ermittelt werden konnten. Das iPad wurde über eine Dauer von $1500ms$ schnell in beliebige Orientierungen bewegt. Danach war es in Ruhe. Nach einer Phase der schnellen Orientierungsänderung weicht der mit Daten des Gyroskops ermittelte Azimut-Wert (rot) fast 150^o vom Kompass-Azimut ab. Diese Abweichung ist eindeutig zu hoch. Man erkennt auch nach ca. $3600ms$ den Angleichungssprung. In diesem Moment würde sich das 3D-Modell auf dem Gerät um 150^o drehen. Diese Grafik ist das Resultat einer einzigen Messung, also nicht allgemein gültig. Aber es sie zeigt, dass mindestens einer der beiden Sensoren falsche Daten liefern kann. Der Azimut-Wert des Kompasses zeigt gelegentlich Sprünge, der des Gyroskops wird schnell sehr ungenau. Es muss also eine Fusion stattfinden um einen möglichst genauen Wert zu erzielen.

In Grafik 6.2 wird die Multisensordatenfusion (blau) mit dazu genommen –

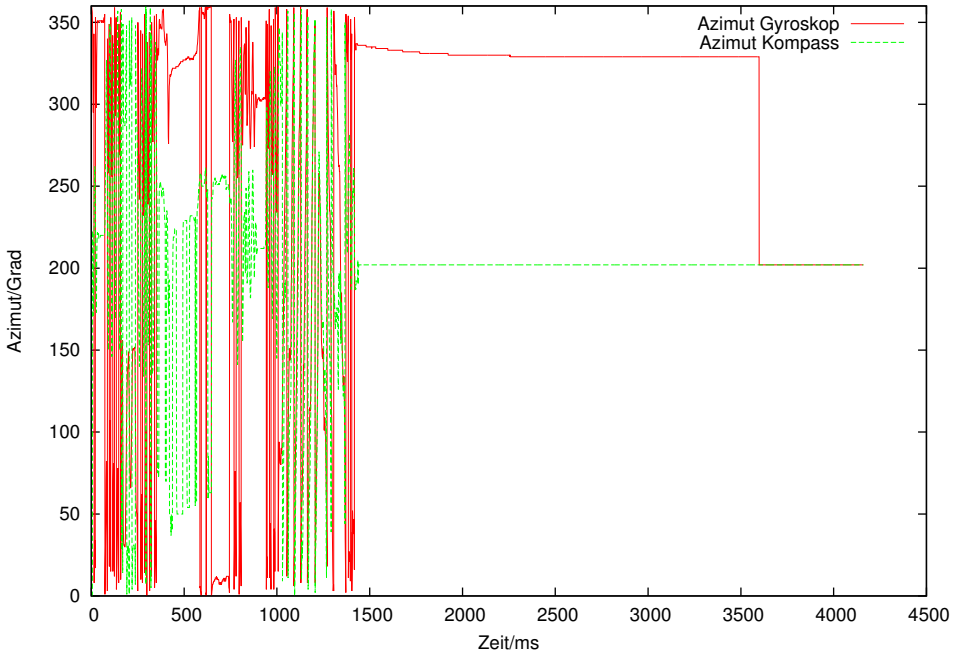

Abbildung 6.1: Azimut-Wert des Gyroskops im Vergleich zum Azimut-Wert des Kompasses

wieder in der selben Umgebung. Es ist zu erkennen, dass sie deutlich näher am Wert des Kompasses liegt. Die Kompassbewegungen werden durch die Multisensordatenfusion geglättet, sodass eine sanfte Bewegung des 3D-Modells entsteht, ohne zu stark von der realen Bewegung des Geräts abzuweichen. Diese Messung wurde mit den Steuerungsvariablen `phi` = 1 und `alpha` = 19 vorgenommen. In Tests hat sich das Mischungsverhältnis von 1:19 bewährt. Bei einem Verhältnis von ca. 1:10, also einem höheren Kompass-Anteil, ist die Glättung der Sprünge des Kompasses durch den Azimut-Wert des Gyroskops nicht groß genug. Bei einem wesentlich höheren Anteil des Gyroskopswerts mit einem Mischungsverhältnis von ca. 1:80 ist die Fehlerkorrektur in der Ruhe zu sehr bemerkbar. Die Driftkorrektur durch den Azimut-Wert des Kompasses ist in diesem Fall nicht groß genug. Wenn das Gerät schnell bewegt wird summieren sich Rechenfehler. Die Korrektur kann wegen den schnellen Bewegungen nicht zum Tragen kommen. Wenn das Gerät danach in Ruhe ist, kann man beobachten wie sich die Orientierung wieder langsam dem korrekten Wert annähert. Dies sollte aber im Optimalfall nicht zu sehen sein. Nur Bewegungen die tatsächlich stattfinden, sollten auch im 3D-Modell sichtbar sein. So findet eine als unnatürlich wahrgenommene Bewegung statt, wodurch die Synchronisierung zwischen Realität und 3D-Modell auf dem Gerät für den Benutzer verloren geht.

Um verlässlichere Aussagen machen zu können habe ich mehrere gleichartige

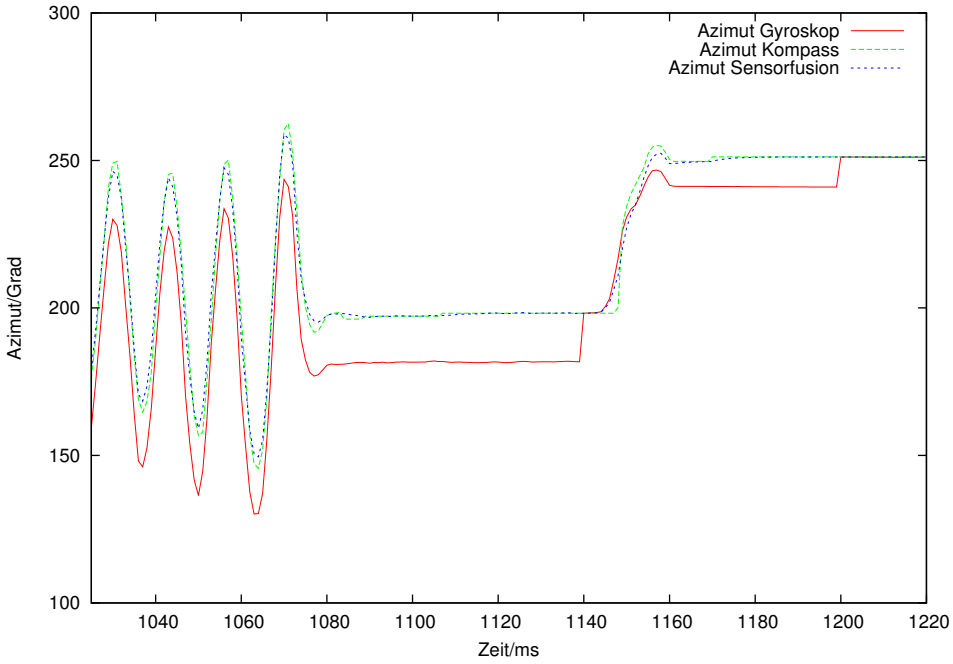

Abbildung 6.2: Azimut berechnet durch Sensorfusion

Messungen durchgeführt und dann die Durchschnittswerte verglichen. Bei diesen Messungen wurde ein iPhone 4S mit iOS 5 verwendet. Pro Messung wurde eine Kreisbahn abgefahren und die drei Azimut-Werte wurden aufgezeichnet. Grafik 6.3 zeigt den Mittelwert aus 20 Messungen. Die Zacken in den Linien in der Mitte ist der Tatsache geschuldet, dass der Sprung von 0^o zu 360^o nicht bei jeder Messung zur selben Zeit stattfindet. Darum kommen im Schnitt solche starken Sprünge zustande. Diese sollen aber in der Beurteilung nicht weiter stören. Es ist zu erkennen, dass der Kompass (blau) wie oben erwähnt Sprünge liefert. Dies ist aber durch die Mittelung über 20 Messungen stark abgeschwächt. Trotzdem ist besonders zwischen 0 und $200ms$, sowie zwischen 1000 und $1200ms$ ein unterschiedlicher Charakter der Linien zu erkennen. Die rote Linie der Sensorfusion ist deutlich weicher und gleichmäßiger. Die Sprünge des Kompasses sind in Grafik 6.4 sehr gut zu erkennen (grüne Linie). Dies ist wieder eine Visualisierung einer einzigen Messung. Die Sprünge sind bei jeder Messung unterschiedlich, so, dass sich kein aussagekräftiger Durchschnittswert als Grafik darstellen lässt. Der Verlauf der Linien des Kompass-Azimut-Werts aller 20 Messungen entspricht aber in seiner Intensität Grafik 6.4. Teilweise bleibt der Kompass bei einem Wert regelrecht *hängen*. Genau diese Fehler werden mit der Sensorfusion (rot) sehr erfolgreich ausgeglichen.

Auch Messungen mit Störfeld wurden durchgeführt. Grafik 6.5 zeigt den Durchschnitt von fünf Messungen. Hierbei wurde an dem ruhenden iPhone ein Störfeld vorbei geführt. Schön zu erkennen ist, dass sich das Gyroskop

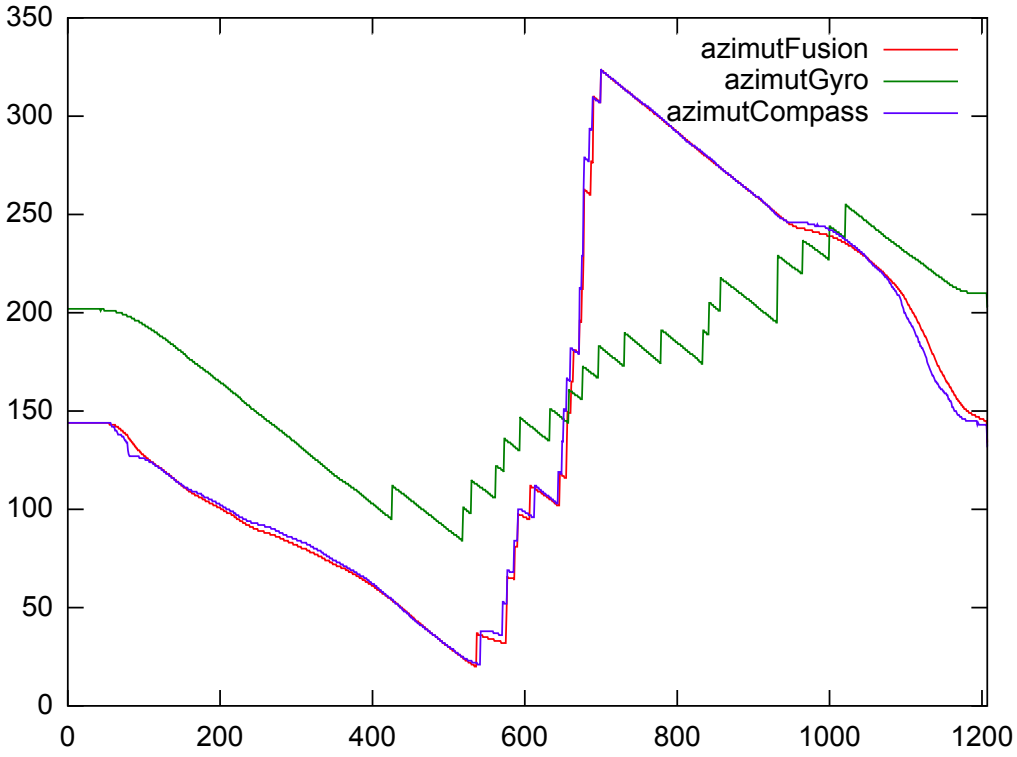

Abbildung 6.3: Gleichmäßige Bewegung in einer Kreisbahn

(grün) völlig ruhig, also korrekt, verhält. Die kurzzeitige Magnetfeldstörung findet zwischen 40 und 100ms statt. Der Azimut-Wert des Kompass (blau) springt von einer Messung zur nächsten im Schnitt 8 Grad auf einmal. Die Sensor-Fusion (rot) kann diese Deviation etwas ausgleichen. Jedoch schwankt die rote Linie trotzdem noch etwas. Es ist aber zu beachten, dass die Werte der Sensor-Fusion maximal 3^o abweichen und die des Kompasses maximal 6^o. Der hier verwendete Fusions-Algorithmus kann allerdings nur kurze Magnetfeld-Störungen ausgleichen. Diese kommen häufiger vor als länger andauernde. Zum Beispiel durch Mobiltelefone und sonstige Funkwellen. Dieser Fusions-Algorithmus ist dazu gedacht, das visuelle Erscheinungsbild weicher, also angenehmer und natürlicher zu gestalten.

Es bleibt festzuhalten, dass zu Beginn meiner Arbeiten gerade erst das iPhone 4 erschienen ist und zum ersten Mal ein Gyroskop in einem iOS-Gerät verbaut wurde. Das iPad 2, mit dem wir hauptsächlich arbeiteten, wurde ein leicht anderes Gyroskop-Modell als beim iPhone 4 verwendet. Damals war zusätzlich iOS 4 ziemlich frisch erschienen. Zum ersten Mal konnten über die CoreMotion API Gyroskop-Daten ausgelesen werden. Die Apple-eigene Sensorfusion war daher noch sehr unausgereift. Inzwischen sind 1,5 Jahre vergangen, iOS 6 steht

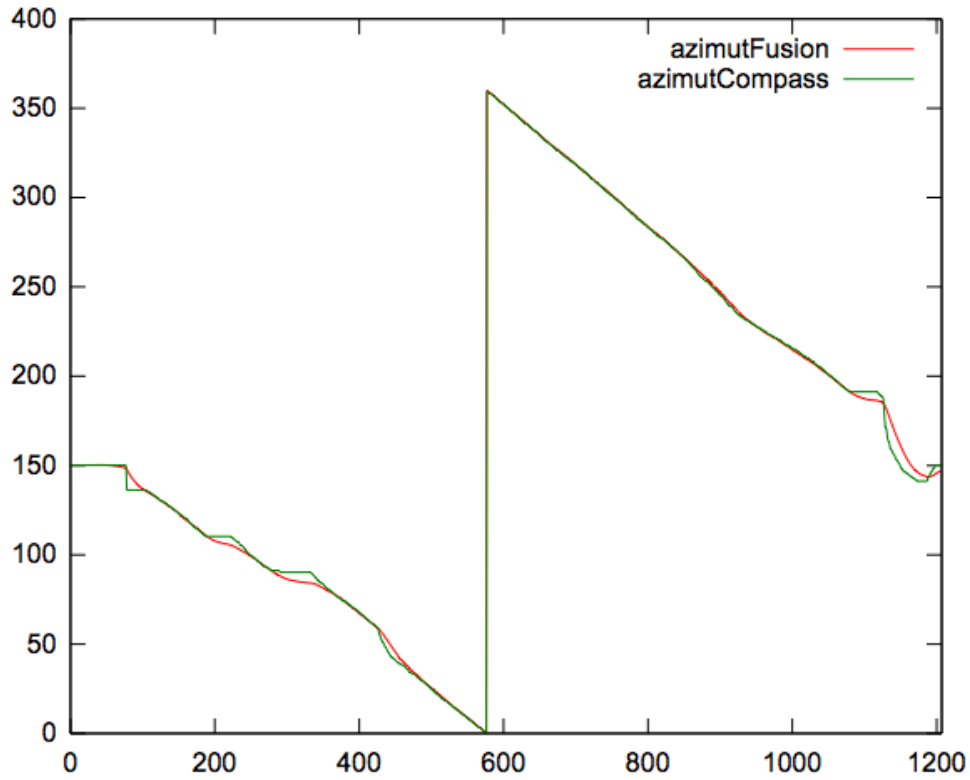

Abbildung 6.4: Beispiel sprunghafter Kompasswerte

in den Startlöchern. Meine letzten Entwicklungsschritte fanden bereits unter eine Beta-Version von iOS 6 statt. In dieser Zeit haben sich Augmented Reality Apps als Erweiterung bestehender Angebote etabliert. Apple selbst hat ebenfalls merklich Entwicklungsarbeit für die CoreMotion API aufgewendet. Der Kompass ist weitaus weniger sprunghaft und auch viel weniger anfällig auf Magnetfeld-Störungen. Apple hat gewissermaßen teilweise im selben Gebiet weiterentwickelt wie ich. Das ist technischer Fortschritt. Trotzdem ist es weiterhin sinnvoll nicht ausschließlich auf den Kompass-Wert der CoreMotion API zu vertrauen. Dieser ist immernoch sprunghaft, wenngleich dies auch viel minimaler ausfällt als noch vor einigen Monaten.

Trotz des relativ einfachen Verfahrens, welches nur etwas Feinjustierung hinsichtlich der jeweiligen Hardware nötig hat, kann ein brauchbares Ergebnis erzielt werden. Wenn die verfügbaren Sensoren richtig kombiniert werden sind die Resultate sehr stabil. Daran haben auch die Verfahren teil, die Apple schon in seinen Cocoa touch Frameworks anwendet. Leider hat man als einfacher Entwickler keinen Einblick darin. Apple erläutert an keiner Stelle der API-Dokumentation, wie die Daten, die die API ausgibt, zustande kommen. Die errechnete Orientierung ist praxistauglich und kann in jede iOS-App einfach

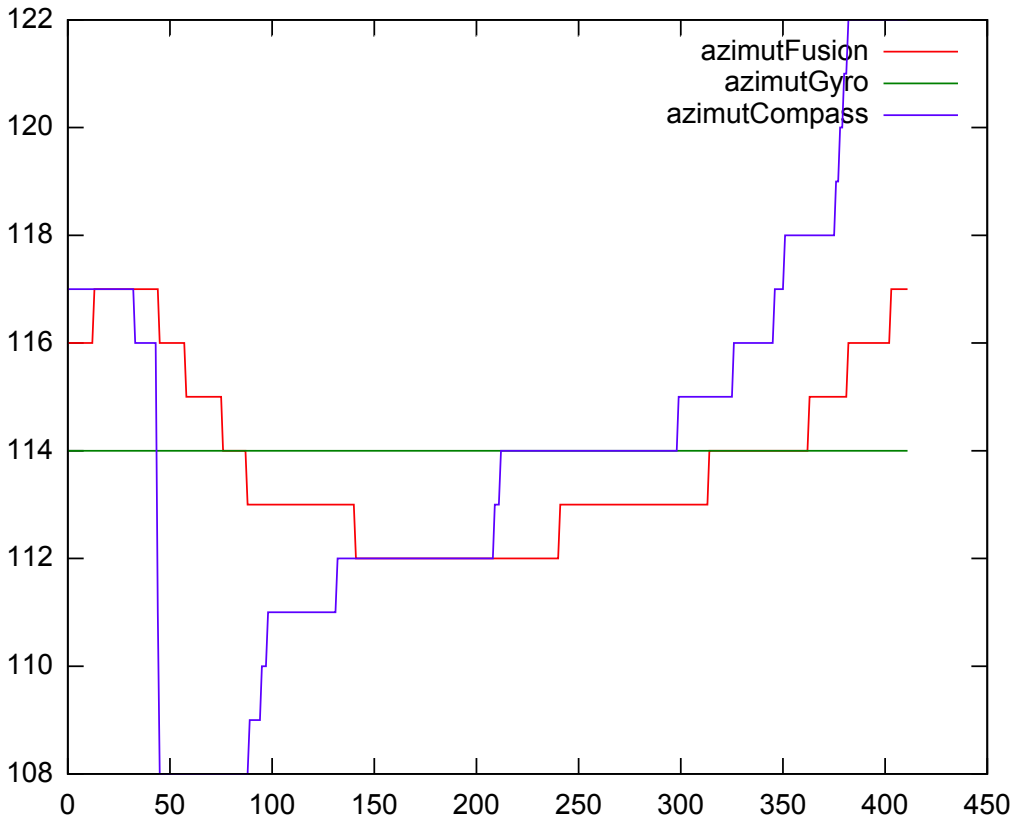

Abbildung 6.5: Beispiel Störfeld

implementiert werden.

Für den gleichzeitig entwickleten Prototypen einer Navigations-App für Bibliotheken konnte die Orientierung begeisteuert werden. Dadurch wird für den Endnutzer eine intuitive zusätzliche Information hinzugefügt. Die Umgebung in der sich das Gerät befindet wird sehr realistisch und genau dargestellt. Durch die entwickelte Sensorfusion kann eine besonders natürliche Wirkung erzielt werden.

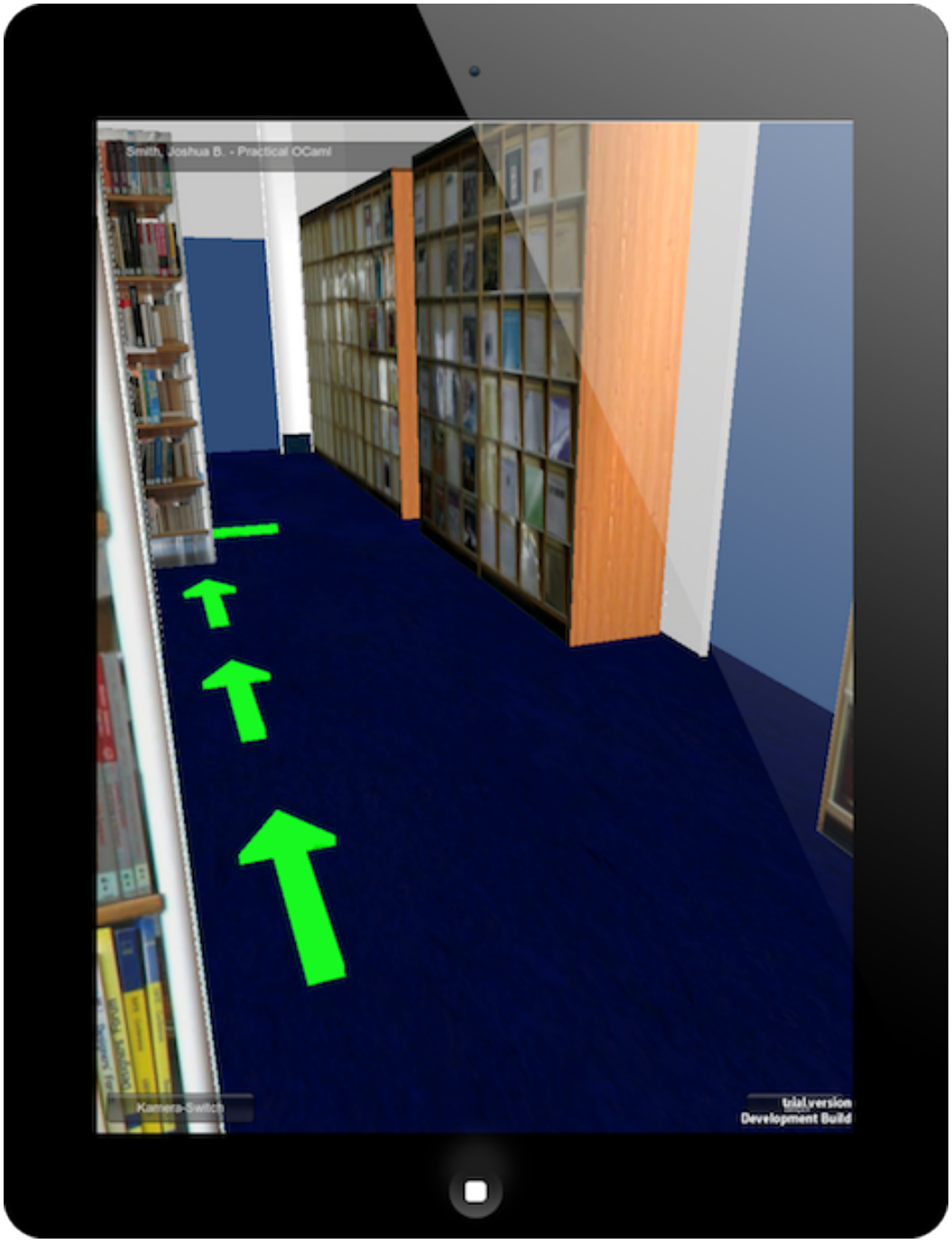

Abbildung 6.6: Fertige Ansicht des 3D-Modells in der App

Kapitel 7

Ausblick

Die naheliegendste Weiterentwicklungsmöglichkeit ist mit Sicherheit Augmented Reality. Das bedeutet, man verzichtet auf das vollständige 3D-Modell um die Bibliothek nachzubilden und nutzt stattdessen die Kamera des Geräts. Auf das Kamera-Bild kann man dann die Navigationspfeile, Markierungen und sonstige Informationen einzeichnen. Augmented Reality hat aber den Nachteil, dass die Genauigkeit nochmal deutlich höher sein muss, als wenn man mit einem 3D-Modell arbeitet. Denn Ungenauigkeiten fallen viel stärker auf, da der abgebildete Raum immer der reale Raum ist. Beim 3D-Modell ist es weniger auffällig wenn die zugrundeliegenden Berechnungen nicht ganz korrekt sind, weil man zwischen Navigation und Realität noch das 3D-Modell als Puffer hat. Das 3D-Modell ist dennoch nach wie vor nötig. Es wird wie bisher auch gebraucht, um die Position der Bücher in den Regalen zu vermerken und um die Wege zu berechnen. Es ist nur unsichtbar, das heißt es fällt auch der Wartungsaufwand bei sich ändernden Regalinhalten weg.

Die erforderliche Genauigkeit ist mit der hier angewendet Sensorfusion noch nicht am Ende ihrer Möglichkeiten. Der Kalman-Filter ist in der Lage, die Ungenauigkeit der Sensoren weiter herauszurechnen. Da die Genauigkeit der Position auch eines der größten Probleme bei der Berechnung der Orientierung ist, würde der Kalman-Filter auch bei der Positionsbestimmung hilfreich sein. Aufgrund der IMUs könnte mit Hilfe des Kalman-Filters auch eine Position bestimmt werden. Diese kann man mit den RSSI-Werten der Bluetooth-Sender und dem Partikelfilter fusionieren und so die Position weiter verbessern. Diese Berechnungen sind allerdings sehr schwierig, da, aufgrund der Geräte-Größe, die Sensoren immer merkbare Fehler produzieren werden. Es muss das richtige Mischungsverhältnis der verschiedenen Sensoren und Verfahren gefunden werden. Das beste Mischungsverhältnis kann aber zusätzlich auch von der aktuellen Art der Bewegung abhängig sein. Dies stellt eine zusätzliche Schwierigkeit dar. Die Entwicklung eines wirklich guten Kalman-Filters für Orientierungs- und Positionsdaten setzt eine große Menge theoretisches Vorwissen und vor allem Test-Zeit voraus. Dies wäre im Rahmen einer Studienarbeit leider nicht

möglich gewesen.

Wenn das Kamerabild sowieso schon für Augmented Reality verwendet wird, könnte man die Positionsgenauigkeit zusätzlich durch Bilderkennung unterstützen. Ähnlich wie bei modernen Autos, die anhand der Fahrbahnmarkierung und der Leitplanken versuchen die Spur zu halten, könnten die Bilddaten mit einem internen 3D-Modell abgeglichen werden. So könnte geprüft werden, ob man sich wirklich gerade an der berechneten Stelle befindet. Wenn diese Überprüfung ständig stattfinden kann, kann sie sogar laufend in die Positions- und Orientierungsberechnung einfließen. Allerdings nur wenn es Sinn macht. Böden, Wände oder auch Regalansichten sehen oft sehr ähnlich aus, sodass keine verlässliche Aussage darüber getroffen werden kann wo man sich befindet. Das Problem bei der Bildverarbeitung in Echtzeit ist der große Rechenaufwand. Es müsste ca. 20 mal in der Sekunde ein komplettes Bild analysiert werden. Das ist eine sehr große Herausforderung für die mobilen Prozessoren. Dennoch könnte auch eine weniger häufige Bildanalyse hilfreich sein. Es könnte dazu dienen die anderen Berechnungen zu verifizieren.

Um die Stabilität der Orientierungsberechnung weiter zu verbessern, könnte eine Magnetfeld-Störungs-Erkennung realisiert werden. Dazu müsste die Orientierung des Kompasses beobachtet werden und wenn eine, verglichen mit den Orientierungs-Daten der anderen Sensoren, unnatürliche Bewegung auftritt, eine Art Reaktions-Schwellenwert greifen. Das heißt, wenn der Kompass eine Drehung feststellt, aber das Gyroskop völlig ruhig bleibt, ist anzunehmen, dass eine Störung durch ein Magnetfeld vorliegt. Die genaue Umsetzung wird aber ebenfalls schwierig sein, denn die Fehler beider Sensoren dürfen nicht als unterschiedliche Drehung erkannt werden. Apple hat dieses Problem bereits selbst erkannt. Die API liefert auch schon deutlich stabilere Werte als zu Anfang.

Nützlich für die Navigation in Bibliotheken ansich könnte die NFC-Technik werden. Diese kleinen Chips sind schon heute keine Seltenheit in Bibliotheks-Büchern. Zum einen können sie zur Orientierung dienen. Über die gefundenen Bücher in der Umgebung kann der Standort relativ genau berechnet werden und als weiterer Fusionsinput für die Positionsbestimmung dienen. Zum anderen kann man mit NFC auch eine automatische Bibliotheks-Inventur durch normale Benutzer realisieren. Die App kann beim normalen Gang durch die Bibliothek die Bücher in der Umgebung erkennen und zumindest ihre ungefähre, auf ca. drei Meter genaue, Position bestimmen. Diese würde auf dem Bibliotheks-Server mit der korrekten Position abgeglichen. Wenn der Unterschied zu groß ist würden die Mitarbeiter der Bibliothek informiert. In der Zwischenzeit wird der neue, aber falsche Standort auf dem Server als aktuelle Position des Buches gespeichert, damit auch andere Bibliotheks-Besucher das Buch, trotz falscher Position, mit der Navigations-App finden können. Diese Technik ist bei wenigen Geräten bereits eingebaut. Allgemein steht sie aber,

wenn man die Patent-Anmeldungen der Hersteller der letzten Jahre verfolgt, auf jeden Fall in den Startlöchern. [Ker11]

Um den Nutzerkreis einer solchen App zu vergrößern ist es auch empfehlenswert, die App für andere Geräte und Systeme neben iOS, wie zum Beispiel Android und Windows Phone, zu entwickeln. Das Grundgerüst der App sollte dank Unity leicht auf andere Plattformen portierbar sein. Nur die API-Aufrufe müssen an das jeweilige System angepasst werden.

Literaturverzeichnis

[All11] A. Allan. *iOS 4 Sensor Programming*. O'Reilly, Sebastopol, 2011. Rough Cuts.

[App11] Apple. Datei:acceleration_axes.jpg. http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html, 2011. [Online; Stand 4. Januar 2012].

[App12a] Apple. Cocoa touch – ios technology overview – apple developer. https://developer.apple.com/technologies/ios/cocoa-touch.html, 2012. [Online; Stand 5. März 2012].

[App12b] Apple. ios developer center – apple developer. https://developer.apple.com/, 2012. [Online; Stand 7. März 2012].

[CH10] J. Conway and A. Hillegass. *iPhone Programming*. Big Nerd Ranch, Atlanta, 2010.

[DW11] St.J. Dixon-Warren. Motion sensing in the iphone 4: Electronic compass. http://www.memsjournal.com/2011/02/motion-sensing-in-the-iphone-4-electronic-compass.html, 2011. [Online; Stand 14. März 2012].

[Gui12] D. F. Guillou. Packaging mems: New manufacturing methodology substantially reduces smart mems costs. http://archives.sensorsmag.com/articles/1203/20/main.shtml, 2012. [Online; Stand 14. März 2012].

[Ker11] Christian Kern. *RFID für Bibliotheken*. Springer, Berlin, 2011.

[Koc08] Thomas Koch. Rotationen mit quaternionen in der computergrafik, 2008. Diplomarbeit, Fachhochschule Gelsenkirchen.

[Mat09] MathsPoetry. File:gimbal lock.png. http://en.wikipedia.org/wiki/File:Gimbal_lock.png, 2009. [Online; Stand 24. Februar 2012] This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

[Mat12] MathWorks. Mathworks — block reference (aerospace blockset™). http://www.mathworks.de/help/toolbox/aeroblks/f2-16051.html#f2-23009, 2012. [Online; Stand 7. März 2012].

[RAG04] B. Ristic, S. Arulampalam, and N. Gordon. *Beyond the Kalman Filter: Particle Filters for Tracking Applications.* Artech House, 2004.

[Tec12] Unity Technologies. Unity – 3d game engine. http://unity3d.com/, 2012. [Online; Stand 4. März 2012].

[Yad11] Manish Yadav. History of android. http://www.tech2crack.com/history-android/, 2011. [Online; Stand 21. August 2012].

[Zel09] A. Zell. Robotik i, 2009. Vorlesung, Universität Tübingen.