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

Studienarbeit Informatik

Orientierungsberechnung mittels Multisensordatenfusion auf iOS-Endgeräten

Sebastian Engel

26.02.2012

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 - 01/2012

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, Stephan Doerr, Martin Lahl, Philipp Wolter, Markus ... und Sabrina Pfeffer.

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 dazu wie folgt: Zu Beginn wird in Kapitel 2 die Entwicklung der benötigten Hard- und Software betrachtet. In Kapitel 3 besprechen wir die Wahl eines geeigneten Konzepts. Bevor wir uns der genauen Implementierung in Kapitel 5 widmen können, wählen wir in Kapitel 4 die für unsere Zwecke geeignete Plattform und Werzkeuge, die wir zur Umsetzung benötigen, aus. Es folgt eine Auswertung der Ergebnisse in Kapitel 6 mit einer Diskussion. Ein kurzer Ausblick in Kapitel 7 beschließt diese Arbeit.

Kapitel 2

Stand der Technik

Bisher findet Navigation hauptsächlich im Freien statt. Zum Beispiel schon seit Langem bei Navigationssystemen für Autos. Dabei wird ausschließlich GPS verwendet. Für Navigation innerhalb von Gebäuden ist GPS 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.

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 und für uns relevanten 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. Für unsere Zwecke beschäftigen wir uns

mit 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.

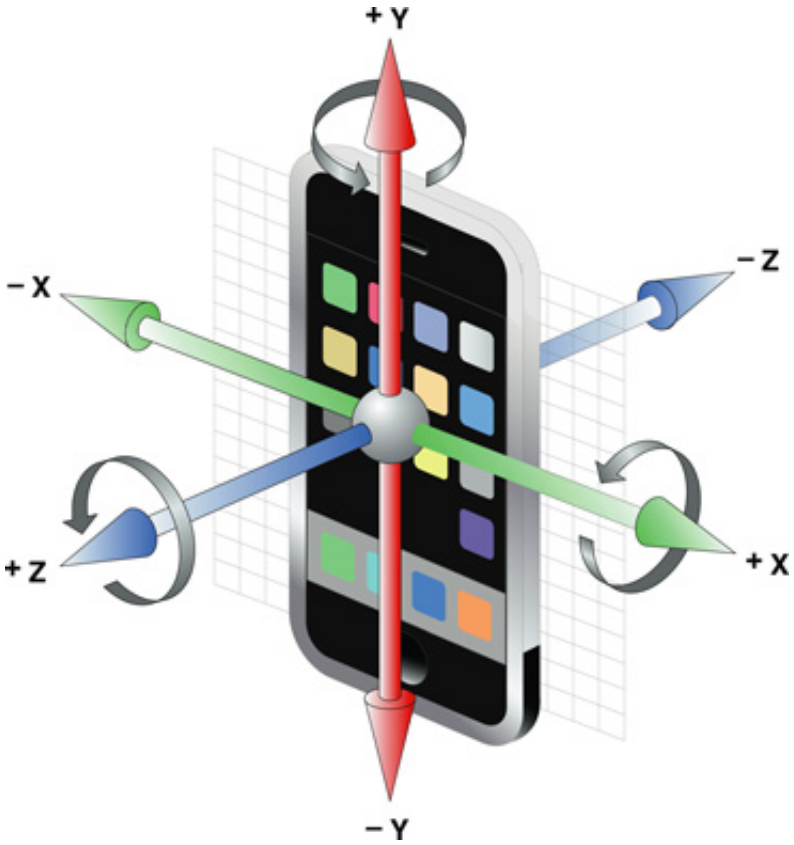

Abbildung 2.1: Roll-Pitch-Yaw [App11]

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:

$$\begin{pmatrix} 0.016134 \\ -0.000284 \\ 1.618407 \end{pmatrix}$$

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. [Wik12c] [Koc08]

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. [Wik12b]

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

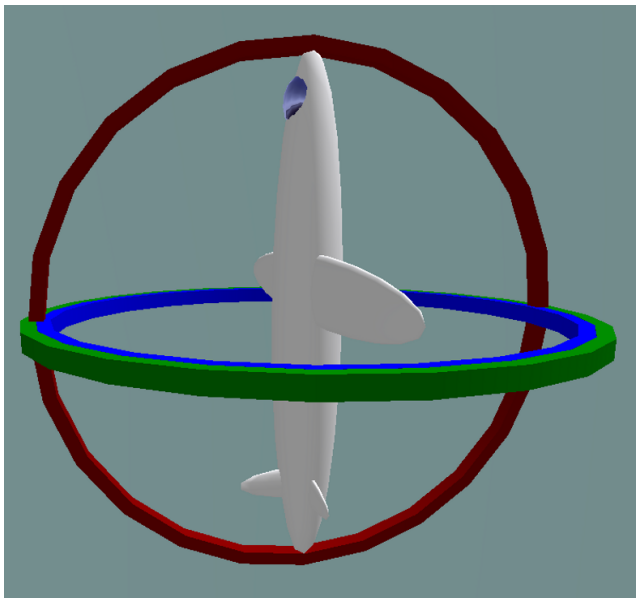

Abbildung 2.2: Gimbal Lock [Wik09]

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.

2.2 Positionsbestimmung

Die Positionsbestimmung erfolgt in unserem 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. Bei unserem Anwendungsfall wollen wir keine Daten übertragen, sondern nur möglichst oft den RSSI-Wert der einzelnen Beacons erfahren. 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. Perso-

nen, 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. [Wik12d]

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.

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 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.

3.4 Multisensordatenfusion

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

Da das Gyroskop keine absoluten Daten 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:

$$f(\alpha, \phi) = (\alpha * updatedHeading + \phi * heading)/(\alpha + \phi)$$

Wobei *heading* der jeweils neue Kompass-Wert ist und *updatedHeading* der alte Azimut-Wert auf den die Drehung, ermittelt durch das Gyroskop, addiert wurde. α und ϕ sind die Gewichtungen von *heading* und *updatedHeading*. Schaubild 3.1 ist eine visuelle Darstellung dieser Vorgehensweise.

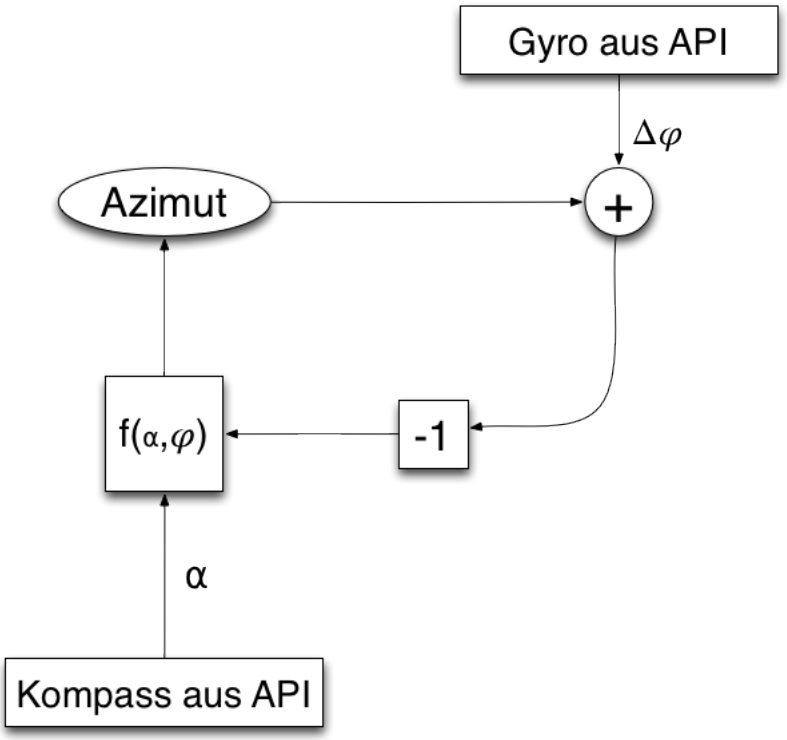

Abbildung 3.1: Konzept Multisensordatenfusion

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. [Wik12a] 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 Erwartungen, da es 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 verwenden wir auch hauptsächlich Objective-C. [App12a]
Bei iOS sind für unsere Zwecke 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 für unseren Anwendungsfall ist die Klasse `CMAttitude`. 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. Für uns ist die Klasse `CMHeading` interessant. Sie 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. Unity 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. In Xcode wird der Quellcode geschrieben und das Interface angeordnet. Mit Xcode kann die App, an der momentan gearbeitet wird, direkt auf ein angeschlossenes iOS-Gerät kompiliert werden. Am Ende wird mit Xcode auch die Datei zur Einreichung in den AppStore erstellt.

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 Zugriff auf die Cocoa-Frameworks möglich ist. Die App wird als Xcode Projekt exportiert. Dann muss in Xcode ein Plug-in Programmiert werden, das als Schnittstelle zwischen selbst geschriebenem Objective-C-Code in Xcode 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

Hier wird also die externe Methode `getDeviceMotion()` aufgerufen. Das Plug-in erwartet den Datentyp `CMQuaternion`. Wir übergeben später einen Quaternion, also immer die aktuelle absolute Orientierung. `CMQuaternion` ist eigentlich ein Datentyp auf dem Core Motion Framework und somit Unity nicht bekannt. Darum musste erst noch ein `struct` mit der Bezeichung `CMQuaternion` definieren.

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 die hier berechnet werde ankommen muss als erstes die `getDeviceMotion()`-

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

Listing 5.2: Struct `CMQuaternion` in C#

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 eine weitere Methode `getOrientation` aufgerufen.

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

Listing 5.4: Methode `getOrientation`

Diese Methode enthält den eigentlichen Objective-C-Quellcode. In ihr können alle API-Aufrufe problemlos ausgeführt werden.
[Tec12]

5.2 Vorbereitung der nötigen Daten

Um die Multidatenfusion durchführen zu können müssen erst alle nötigen Daten auslegesen werden. Dazu müssen `CMMotionManager`- und `CLLocationManager`-Objekte initialisert werden.

```
1  // Set up locationManager
2  if (locationManager == nil) {
3    locationManager=[[CLLocationManager alloc] init];
4    locationManager.desiredAccuracy = kCLLocationAccuracyBest;
5  }
6
7  // Set up motionManager
8  if (motionManager == nil) {
9    motionManager = [[CMMotionManager alloc] init];
10   motionManager.deviceMotionUpdateInterval = 1.0/60.0;
11 }
```

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

In Zeile 4 wird die Genauigkeit des `CLLocationManager`-Objekts und in Zeile 10 die Update-Frequenz des `CMMotionManager`-Objekts eingestellt. Der Kompass-Wert den wir aus dem `CLLocationManager`-Objekt auslesen 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 setzen wir die minimale Winkeländerung auf 1^o fest.

```
1  // Start listening to events from locationManager
2  if ([CLLocationManager headingAvailable]) {
3    locationManager.headingFilter = 1;
4    [locationManager startUpdatingHeading];
5  }
```

Listing 5.6: `locationManager` starten [App12b]

Mit dem Aufrufen der Methode `startUpdatingHeading` in Zeile 4 wird hier auch gleichzeitig das eventgesteuerte Abfragen des Kompass-Werts gestartet.

Die Methode die auf die Kompass-Änderungen hört ließt den Kompass-Wert aus und stellt ihn in einer globalen Variable zur Verfügung.

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 man, wenn dieser Fall auftritt, einen Offset mit dem ausgelesenen Wert addieren. 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. Jetzt haben wir die Azimut ermittelt durch den Kompass.

Die Azimut-Änderung ermittelt duch 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 aufge-

```
- (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

```
if(motionManager.isDeviceMotionAvailable) {

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

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

Listing 5.8: Bewegungsdaten auslesen [App12b]

rufen, wenn es Bewegungs-Daten des Geräts zu verarbeiten gibt. Hier ist das alle 1/60 Sekunden der Fall, weil wir das bei der Initialisierung des `CMMotionManager`-Objekts so festgelegt haben.

Das auslesen der eigentlichen Orientierungsdaten erfolgt in Zeile 6. Wobei in diesem `CMAttitude`-Objekt alle drei Beschreibungs-Möglichkeiten, 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. Dazu wurden die vier Methoden `quaternionMagnitude`, `inverseQuaternion`, `multiplyQuaternions` und `normalizeQuaternion` geschrieben.

```
- (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`

```
- (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`

Interessant für uns sind Azimut und Elevation. Darum berechnen wir die beiden Werte in Grad aus dem Quaternion. Die Methoden `azimutFromQuaternion` und `elevationFromQuaternion` berechnen den entsprechenden Winkel in Radian.

```
- (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

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, so dass 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 verschieben wir den Sprung um 90^o nach oben. Dann tritt er nur auf wenn der Betrachter das iPad direkt an die Decke hält.

Jetzt haben wir die Azimut-Änderung und die Elevation, basierend auf Gyroskop-Daten errechnet.

[Koc08] [Wik12e] [Mat12] [CH10] [All11]

```
- (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

Hier müssen wir uns, danke Core Motion nur noch um den Azimut-Wert kümmern. Core Motion hat den Elevation-Wert aus dem Gyroskop bereits mit dem Accelerometer stabilisiert. Darum weißt der in Listing 5.15 berechnete Elevation-Wert keinen bemerkbaren Drift mehr auf.

Nun können wir die Formel aus Kapitel 3.4 umsetzen.

```
updatedAzimut = updatedAzimut - azimutDiff;                                    1
                                                                               2
float alpha = 19.0;                                                            3
float phi = 1.0;                                                               4
                                                                               5
//fusionate gyro estimated heading with new magneticHeading                    6
updatedAzimut = (alpha* updatedAzimut + phi*heading)/(alpha+phi);              7
```

Listing 5.16: Eigentliche Datenfusion

In Zeile 1 aus Listing 5.16 wird die Azimut-Änderung auf den vorherigen Azimut-Wert angwendet. Zeile 7 ist die genaue Umsetzung der Formel ausl 3.4. `updatedAzimut` ist der vorherige 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.

Kapitel 6

Ergebnis

Eine funktionierende Orientierung lässt sich bereits mit Kompass und dem Pitch-Wert den das Gyroskop liefert realisieren.

Kapitel 7

Ausblick

NFC [Wik12f], iPad 3, Android, Kalman-Filter, Bewegungsberechnung, Augmented Reality

Mit das Wichtigste natürlich!

Hier gilt es beides, die Info-Seite der Arbeit sowie die Bio-Seite zu diskutieren!!

Take your time for writing the discussion, it is the most important chapter of your thesis.

Mindestens 5 Seiten lang.

Ausblick kann auch ein extra Kapitel werden, wenn man das will.

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.

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

[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].

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

[Wik09] Wikipedia. File:gimbal lock.png. http://en.wikipedia.org/wiki/File:Gimbal_lock.png---Wikipedia{,}TheFreeEncyclopedia, 2009. [Online; Stand 24. Februar 2012] This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

[Wik12a] Wikipedia. Android (betriebssystem) — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/wiki/Android_(Betriebssystem), 2012. [Online; Stand 4. März 2012].

[Wik12b] Wikipedia. Drehmatrix — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Drehmatrix&oldid=99490688, 2012. [Online; Stand 24. Februar 2012].

[Wik12c] Wikipedia. Gimbal lock — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Gimbal_lock&oldid=478135314, 2012. [Online; Stand 24 Februar 2012].

[Wik12d] Wikipedia. Particle filter — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Particle_filter&oldid=477198335, 2012. [Online; Stand 24 Februar 2012].

[Wik12e] Wikipedia. Quaternion — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Quaternion&oldid=479916302, 2012. [Online; Stand 7. März 2012].

[Wik12f] Wikipedia. Rfid — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=RFID&oldid=100331792, 2012. [Online; Stand 4. März 2012].