Abschlussbericht MOSA 2012: ARTUS
Aufgaben für Tablets und Smartphones

Stefan Hoffmann
Universität Bonn
Institut für Informatik
hoffman4@cs.uni-bonn.de

Alasdair Schlünkes Collinson
Universität Bonn
Institut für Informatik
collinso@cs.uni-bonn.de

Abstract

ARTUS *ist eine Bibliothek zur Erstellung modularer physischer Aufgaben für mobile Geräte mit dem Betriebssystem Google Android.*

1. Einleitung

Seit der Ankündigung des Apple iPhone im Januar 2007 [1] und der Eröffnung des dazugehörigen AppStores im März 2008 [2] haben Smartphones und Applikationen für diese einen bemerkenswerten Aufschwung erlebt. Viele Ideen, die zuvor im Bereich der Mobiltelefone unüblich oder gar unmöglich waren, sind seitdem von Entwicklern aufgegriffen und für Smartphones genutzt worden.

Eine dieser Ideen ist das sogenannte *GeoCaching*, eine Form der Schnitzeljagd bei der GPS-Koordinaten vorgegeben werden an denen Objekte versteckt oder Aufgaben zu erfüllen sind. Smartphones, die heutzutage standardmäßig mit GPS-Sensoren ausgestattet sind, können nun nicht nur die Rolle des GPS-Empfängers übernehmen, sondern auch beispielsweise Informationen zu den jeweiligen Aufgaben geben.
Diese Idee wurde bei der Entwicklung von GEOQUEST aufgegriffen - einer App, entwickelt an der Universität Bonn, mit der nicht nur einzelne Ziele sondern komplexe Aufgabenreihen sequenziell erfüllt werden können. GEOQUEST ist beschränkt auf die Darstellung von Aufgaben und der Überprüfung von GPS-Koordinaten - die traditionellen Teile des *GeoCachings*. Die Möglichkeiten, die durch Smartphones geboten werden, sind jedoch deutlich größer - selbst sehr günstige Smartphones verfügen über eine ganze Reihe von Sensoren, die innerhalb von Applikationen genutzt werden können. Deshalb ist im Rahmen der Projektgruppe *Social and Mobile Applications* im Wintersemester 2011/12 ebenfalls an der Universität Bonn das Projekt ARTUS (Aufgaben für Tablets und Smartphones) für Google Android entstanden - eine Bibliothek von (größtenteils) sensorabhängigen Aufgaben, die beliebig kombiniert und in Projekte wie z.B. GEOQUEST eingebunden werden können.

2. Sensorauswertung

2.1. Voraussetzungen - welche Sensoren gibt es?

Moderne Smartphones verfügen wie bereits erwähnt über eine ganze Reihe von Sensoren, wovon hier die in ARTUS verwendeten aufgezählt werden sollen:

Accelerometer Der Accelerometer gehört zu den in der App-Entwicklung häufig genutzten Sensoren. Er dient dazu, die Lage des Gerätes zu messen. Die Rückgabewerte sind Beschleunigungswerte auf den x-, y- und z-Achsen (siehe Abbildung 1). [3]

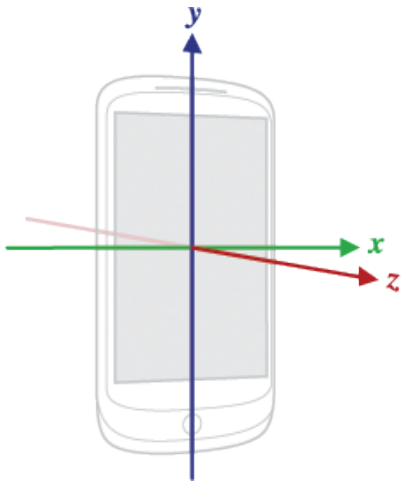

Abbildung 1: Die Achsen des Accelerometers.
Quelle: `http://developer.android.com/images/axis_device.png`

GPS-Sensor Der GPS-Sensor liefert ein Objekt des Typs `Location` zurück, aus welchem eine ganze Reihe von Informationen abgerufen werden kann. [5] Die in ARTUS genutzten Informationen sind hierbei:

- die Genauigkeit der Ortsbestimmung,
- der Abstand zu einer anderen `Location` und
- die aktuelle Geschwindigkeit der Fortbewegung

Orientierungssensor Der Orientierungssensor ist genau genommen nur ein Pseudosender, dessen Werte aus denen des Accelerometers und des Magnetfeldsensors berechnet werden. Zurückgegeben werden:

- ein Richtungswinkel
  Werte liegen hier zwischen 0 und 359, wobei 0 der (magnetische) Norden ist, 90 entspricht Osten, etc.
- ein x-Rotationswinkel
  Gibt die Rotation um die x-Achse an, Werte liegen zwischen -180 und 180.
- ein y-Rotationswinkel
  Gibt die Rotation um die y-Achse an, Werte liegen zwischen -90 und 90.

Die x- und y-Achsen liegen hierbei parallel (bzw. tangential) zur Erdoberfläche, wobei die x-Achse dabei nach Osten und die y-Achse nach Norden zeigt (siehe Abbildung 2). Dies entspricht also nicht der Gier-, Roll- und Querachse, wie man sie beispielsweise bei Flugzeugen kennt. [3] ARTUS verwendet hiervon lediglich den Richtungswinkel.

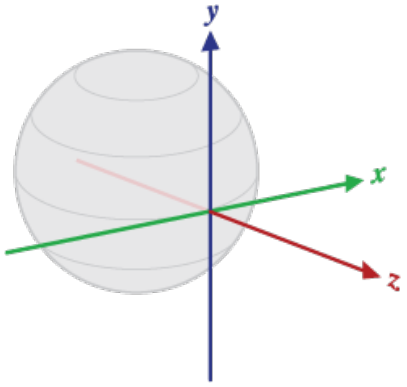

Abbildung 2: Die Achsen des Orientierungssensors, bezogen auf die Erde.
Quelle: `http://developer.android.com/images/axis_globe.png`

Touchscreen Möglicherweise der offensichtlichste Sensor bei modernen Smartphones ist der Touchscreen. Fragt man Werte von diesem ab, bekommt man bei Android ein sogenanntes `MotionEvent`-Objekt zurück, von dem man Daten abrufen kann. Am wichtigsten sind dabei meist:

- der Typ des Events (`ACTION_DOWN`, `ACTION_MOVE` und `ACTION_UP` sind für die Single-Touch-Verarbeitung ausreichend) und
- die x- und y-Koordinaten der Berührung (relativ zum Bildschirm).

In ARTUS wird der Touchscreen lediglich für die Info-Task (↪ 5.8) verwendet.

2.2. Die Suche nach verwandten Arbeiten

Zu Beginn des Projekts war ein mögliches Ziel, die Erkennung von Aktivitäten wie *Gehen*, *Rennen*, *Klettern*, *Hüpfen*, *Treppen steigen* etc., um diese als Aufgaben stellen zu können. Die Suche nach ähnlichen Ansätzen brachte eine große Vielfalt an verwandten Arbeiten zu diesem Thema hervor. Ein Großteil dieser Arbeiten verwendete einen oder mehrere Accelerometer um daraus die aktuelle Aktivität der Testperson abzuleiten. Dabei verfolgen viele Arbeiten [6, 7] einen ähnlichen Ansatz und geben die Position des Accelerometers am Körper fest vor, was für unser Vorhaben gänzlich ungeeignet war.

Auch sind nicht alle Ansätze auf Smartphones übertragbar. So wurde in [7] ein relativ komplexes Verfahren verwendet um die aktuelle Aktivität zuerst als statische oder dynamische Aktivität zu klassifizieren und anschließend einer speziellen Aktivität zuzuordnen. Während man mit diesem Ansatz eine sehr hohe Genauigkeit von 95% erreichen konnte, wurde die gesamte Berechnung auf einem PC durchgeführt, der sowohl wesentlich mehr Rechenleistung und als auch in diesem Sinne unbegrenzte Energie im Vergleich zu einem Smartphone hat. Die feste Position des Accelerometers am Handgelenk ermöglichte neben der Erkennung von Aktivitäten wie *Gehen*, *Laufen* und *Stehen* auch eine Erkennung von *Staub saugen*, *Schrubben* und *Zähne putzen*.

Eine weitere Arbeit [6] verfolgte das Ziel, die Bewegung innerhalb eines Gebäudes zu verfolgen. Man beschränkte sich dabei darauf das aktuelle Stockwerk bzw. den Wechsel in ein anderes Stockwerk zu erkennen. Dazu wurde ein Android-Smartphone am Fußgelenk der Person befestigt welches analysierte, ob die Person *still steht*, *sich bewegt*, *Treppen hinauf oder herunter steigt* oder den *Aufzug benutzt*. Die aktuelle Aktivität konnte so mit einer Genauigkeit von 84% korrekt zugeordnet werden. Dieser Ansatz kam unserer Idee schon rechte nahe, doch auch hier wurde das Smartphone stets fest am Fußgelenk getragen, so dass mit

deutlich schlechteren Ergebnissen zu rechnen war, wenn das Smartphone frei am Körper getragen werden kann.

Einen weiterer interessanter Ansatz war die Verwendung des *Nike+iPod Sport Kit* [8]. Dieser Sensor wird direkt innerhalb des Schuhs getragen und sendet verschlüsselt Informationen über den Druck, den der Fuß auf den Sensor ausübt, an ein Mobilgerät. Dieser Druck wechselt deutlich zwischen Aktivitäten wie *Stehen*, *Gehen* oder *Laufen*. Obwohl die Verschlüsselung zu diseem Zeitpunkt noch nicht überwunden wurde, konnte man mit Hilfe der verschlüsselten Daten ein Merkmal zur Auswertung generieren. Im Unterschied zu [6, 7] wird die Position des iPhones nicht festgelegt und darf somit beliebig mitgeführt werden. Ein weiterer auffälliger Unterschied ist, dass man insgesamt 124 Merkmale zur Auswertung heranzog, während [6] sich mit 16 Merkmalen begnügte und man in [7] sogar feststellte, dass mehr als 24 Merkmale keine weitere Verbesserung der Ergebnisse brachte. Schlussendlich konnte man die Aktivitäten *Laufen*, *Gehen*, *Radfahren* und *Sitzen* mit einer Genauigkeit von 97% erkennen.

Abschließend mussten wir jedoch für unser Projekt erkennen, dass die Erkennung der von uns gewünschten Aktivitäten keine leichte Aufgabe sein würde und der Erfolg ungewiss wäre, da andere Arbeiten mit festen Accelerometer-Positionnen gearbeitet oder zusätzliche Hardware benötigt haben. Diese Bedenken und der begrenzte Zeitrahmen der Projektgruppe, haben dazu geführt, dass wir uns auf das Erstellen einer Bibliothek konzentriert haben, die möglichst einfach um verschiedene Aufgabentypen erweitert werden kann.

3. Die Aufteilung in Tasks

ARTUS ist modular aufgebaut und demnach ist es sinnvoll, die verschiedenen Aufgabenteile beliebig kombinieren zu können. Dies kann man sehr gut durch das Composite Pattern [9] erreichen. Die Rolle des *Components* übernimmt hier die abstrakte Klasse `Task`. In dieser ist festgelegt, dass eine Aufgabe (d.h. eine Kindklasse von `Task`, im Weiteren als eine *Task* bezeichnet) folgende Eigenschaften hat:

- Eine Task hat einen Status, der einer der folgenden sein kann: `CREATED`, `READY`, `PAUSE_REQUESTED`, `ACTIVE`, `LOST` und `WON`. Die Task kann wie in Abbildung 3 gezeigt wie folgt die Zustände wechseln. Bei der Erstellung einer Task soll der Status im Normalfall `CREATED` sein; sobald die Task bereit zur Ausführung ist, wechselt sie den Status zu `READY`; wird sie gestartet, ist sie `ACTIVE`; im Verlustfall wechselt sie zu `LOST` und im Gewinnfall zu `WON`. Ist die Task in der Lage einen Pausenzustand anzufordern, kann sie auch den Status `PAUSE_REQUESTED` annehmen und wechselt zurück in den Status `READY` sobald die Pause nicht mehr benötigt wird. Wird eine Task pausiert, so wechselt sie in den Status `READY` und wartet auf die Aufhebung der Pause. Ausnahme: Hat die Task die Pause selber angefordert, so bleibt sie in `PAUSE_REQUESTED`. Nicht jede Task muss jeden Status annehmen können.

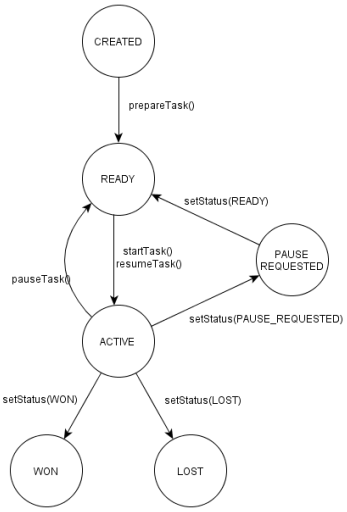

Abbildung 3: Die verschiedenen Zustände einer Task

- Jede Task hat eine öffentliche Funktion `getStatus()`, mit der der aktuelle Status abgerufen werden kann.

- Jede Task implementiert das Observer Pattern [10] und hat deshalb die Funktionen `registerTaskListener(TaskListener l)`, `removeTaskListener(TaskListener l)` und `notifyObservers()` sowie eine Liste von `TaskListener`-Objekten.

- Jede Task hat die öffentlichen Funktionen `prepareTask()`, `pauseTask()`, `resumeTask()`, `startTask()` und `stopTask()`, die in jeder Kindklasse individuell implementiert werden können und (mit Ausnahme von `prepareTask()`) auch müssen. Diese dienen zur Steuerung der Tasks.

- Jede Task hat eine `getUI()`-Funktion, die die graphische Oberfläche zurückgibt. Dies kann ein beliebiger `View` [11] sein.

- Jede Task hat außerdem eine `hasGoal()` Funktion, die Informationen darüber enthält, ob die Aufgabe gewinnbar ist. Diese Funktion muss in jeder Task überschrieben werden. Der Rückgabewert ist ein Wahrheitswert vom Typ `boolean`.

Durch diese Vorgehensweise ist die Erstellung und Einbindung einer neuen Task in das bestehende Projekt einfach.

4. Kontakt zur Außenwelt - Missionen im XML-Format

Eine Kombination oder Sequenz von Aufgaben für den Spieler wird in ARTUS als *Mission* bezeichnet. Diese Missionen werden entweder im zugewiesenen Missionsordner auf der SD-Karte gespeichert (`/[sd-verzeichnis]/missions`) oder direkt an die XmlMissionActivity (↪ 4.2) übergeben.

4.1. Definition von Missionen per XML

Der Aufbau der XML-Dateien ist so einfach wie möglich gehalten. Als Wurzelelement muss `<mission>` stehen, welches genau ein Element vom Typ `task` enthält. Weitere Task-Elemente in dieser Ebene werden ignoriert.

4.1.1. Das `<task>`-Element

Jedes `<task>`-Element steht für eine Aufgabe und muss immer das Attribut `type` enthalten; der Wert von `type` entspricht dabei dem Klassennamen der gewünschten Aufgabe. Alle Parameter, die eine Aufgabe benötigt, werden als Attribut oder weiteres Element angegeben. Im Fall der sogenannten *logischen Aufgaben* (siehe And / Or (↪ 5.6) und TaskList (↪ 5.7)) kann als Parameter wieder ein `<task>`-Element auftauchen.

Codebeispiel 1: Das Grundgerüst einer Missions-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<mission>
  <task type="TaskName" attribut1="attr1" ...>
    ...
  </task>
</mission>
```

Das folgende Beispiel verdeutlicht das Zusammenspiel der einzelnen Tasks. Der Spieler soll 20m laufen *und* dabei sein virtuelles Ei nicht fallen lassen.

Codebeispiel 2: Zusammenspiel verschiedener Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<mission>
  <task type="And">
    <task type="Eierlauf" />
    <task type="TrackDistance" distance="20" />
  </task>
</mission>
```

Eine Aufzählung dieser und aller anderen Tasks sowie genaue Bedeutungen und Beschreibungen dieser sind in Abschnitt 5 zu finden.

4.2. XmlMissionActivity - one class to rule them all

Die Klasse XmlMissionActivity übernimmt die Steuerung und Kontrolle der gestarteten Mission. Sie bildet damit die höchste Instanz in der Hierarchie von Aufgaben und vermittelt zwischen Missions- bzw. Aufgabenstatus und Spieler. Dem Aufruf per Intent wird der Dateiname inkl. Pfad der XML-Missionsdatei mitgegeben (↪ 4.1). XmlMissionActivity sorgt anschließend für das Einlesen der XML-Datei und die Instanzierung der obersten Task in der Hierarchie. Die Instanzierung möglicherweise vorhandener geschachtelter Tasks übernehmen die jeweils übergeordneten Tasks.

Ist die oberste Task instanziert, so läuft die Kommunikation nur zwischen dieser Task und XmlMissionActivity ab. XmlMissionActivity verwaltet aus seiner Sicht nur diese eine Task und gibt Statusrückmeldungen an den Spieler weiter. Umgekehrt sendet XmlMissionActivity Befehle zum Starten, Stoppen oder Pausieren an diese Task. Zu guter Letzt ist XmlMissionActivity natürlich auch dafür verantwortlich das UI der Task anzuzeigen.

5. Die Tasks

Im Folgenden werden die jeweiligen implementierten Tasks vorgestellt.

5.1. Eierlauf - „Halte das Handy gerade!“

Diese Task ist durch das traditionelle Geschicklichkeitsspiel des Eierlaufens inspiriert. Beim traditionellen Eierlauf bekommt man einen Esslöffel sowie ein Ei, einen Tennisball oder etwas Ähnliches. Man muss dann mit dem Löffelstiel in der Hand eine vorgegebene Strecke laufen und dabei das Ei auf dem Löffel balancieren.

Moderne Smartphones verfügen, wie in Abschnitt 2.1 bereits angesprochen, über einen Accelerometer, mit dem u.a. Bewegungen gemessen werden können. Mit Hilfe dieses Sensors ist deshalb eine Task entstanden, bei dem das Gerät möglichst gerade gehalten werden muss.

Ein Eierlauf ist automatisch verloren, wenn das *Ei* (dargestellt als grauer Kreis) die Grenze (roter Kreis) berührt. Demnach kann ein Eierlauf allein nur verloren, nicht gewonnen werden - es bietet sich deshalb die Kombination mit einer gewinnbaren Task wie z.B. einem Timer (↪ 5.3) oder einem Entfernungsmesser (↪ 5.4) an.

Für eine sinnvolle Verwendung sei auf Codebeispiel 16 verwiesen.

Codebeispiel 3: Die Definition einer Eierlauf-Task

```
<task type="Eierlauf" sensitivity=".7" />
```

Abbildung 4: Ein Eierlauf, gekoppelt mit einem Timer

Das einzige einstellbare Attribut ist demnach `sensitivity`, welches einen `float`-Wert erwartet. Dieses Attribut ist optional - wird es nicht gegeben, wird ein Empfindlichkeitswert von $0.5f$ angenommen.

Außerdem ist zu beachten, dass der Eierlauf sofort nach dem Laden startet, die Kombination mit einer TaskList (↪ 5.7) und einer Info-Anzeige (↪ 5.8) ist also sinnvoll.

5.1.1. Logik

Die interne Logik hinter dieser Aufgabe ist sehr simpel: Reagiert wird auf Änderungen der Accelerometerwerte. Die x- und y-Werte des Accelerometers werden hierbei 10 mal pro Sekunde auf die Position des Eies addiert, wodurch die Bewegung zustande kommt; da die z-Achse bei dieser Aufgabe senkrecht zum Boden steht (vgl. Abbildung 1), ist dieser Wert für die Berechnung nicht von Bedeutung. Anschließend wird überprüft, ob sich das Ei noch innerhalb eines Radius von 25 Einheiten um den Startpunkt befindet. Ist dies nicht der Fall, berührt das *Ei* die Grenze und das Spiel ist verloren.

5.1.2. UI

Die graphische Oberfläche des Eierlaufs ist eine `View` [11] mit einem `canvas`, auf welches zwei (oder im Verlustfall 3) PNG-Dateien gemalt werden. Der `canvas` ist dabei quadratisch mit einer Seitenlänge von 95% der Bildschirmbreite. Die Bilddateien werden entsprechend skaliert, um unabhängig von der Bildschirmgröße sinnvoll dargestellt zu werden.

Die einzige Änderung zwischen Malvorgängen ist die Position des *Eies* - dazu werden durch die Funktion `public void setValues(short[] values)` regelmäßig die bereits erwähnten x- und y-Werte an die View übergeben und für die Berechnung der neuen Position des Ei-Bildes genutzt.

5.2. Spin - „Drehe dich um dich selbst!“

Die Spin-Task soll einen typischen Teil vieler Bewegungsspiele umsetzen: Man soll sich um die eigene Achse drehen. Der Orientierungssensor, der unter anderem als Kompass genutzt werden kann, gibt dem Smartphone die Möglichkeit, solche Bewegungen zu messen. Alternativ verfügen einige Geräte über ein Gyroskop [3], allerdings ist dieser Sensor zum Zeitpunkt des Schreibens in vielen neuen Geräten nicht verbaut, weshalb wir ihn aus Kompatibilitätsgründen nicht verwenden.

Das folgende Beispiel zeigt die Verwendung der Spin-Task. Da Spin, im Gegensatz zum Eierlauf, gewonnen werden kann ist eine Mission, die lediglich aus einer Spin-Task besteht, durchaus sinnvoll. Eine Kopplung mit anderen Aufgaben (z.B. einer Zeitgrenze (↪ 5.3), einer Beschränkung der Bewegung (↪ 5.4) oder auch einem Eierlauf (↪ 5.1)) ist jedoch problemlos möglich.

Codebeispiel 4: Die Definition einer Spin-Task

```
1 <task type="Spin" howoften="2" direction="left" />
```

Es gibt hier zwei wählbare Attribute: `howoften`, welches die Anzahl an Drehungen angibt, die notwendig sind um die Aufgabe zu erfüllen, und `direction`, welches die akzeptierten Drehrichtungen angibt; akzeptierte Werte sind `left` für Drehungen gegen den Uhrzeigersinn, `right` für Drehungen im Uhrzeigersinn und `both` falls beide akzeptiert werden. Diese Attribute sind jeweils optional; falls sie nicht gegeben werden wird `howoften` auf 1 gestellt und beide Drehrichtungen werden akzeptiert.

5.2.1. Logik

Der Grundgedanke hinter der Spin-Logik ist sehr simpel: Es gibt ein Array von Zeitstempeln (vgl. Abbildung 5) und jeder Eintrag des Arrays ist einem Winkelabschnitt zugeordnet. Findet die Task eine aufsteigende oder absteigende Reihe innerhalb dieses Arrays (wobei vor dem Eintrag 0 der Eintrag $n-1$ kommt und umgekehrt folgt auf $n-1$ der Eintrag 0), so gilt die Aufgabe als erfüllt (sofern beide Drehrichtungen akzeptiert werden - ansonsten wird zwischen auf- und absteigender Reihe unterschieden).

Bei diesem Vorgehen gibt es allerdings ein Problem: Nicht immer reagiert der Orientierungssensor schnell genug und so kann es vorkommen, dass im Array Werte ausgelassen werden, wodurch keine stetige Reihe zustande kommt. Um diesem Problem vorzubeugen wurden zwei Vorkehrungen getroffen:

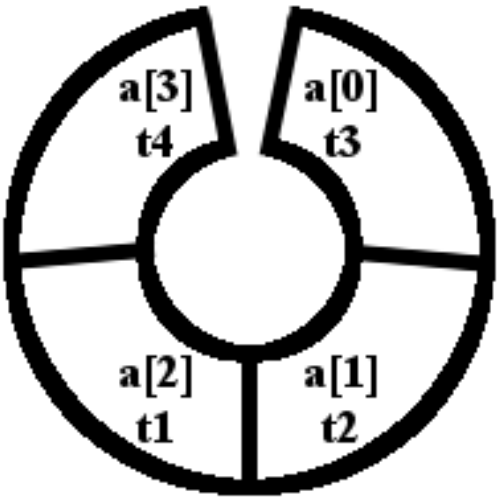

Abbildung 5: Spin: Ein (vereinfachtes) Array a zur Speicherung der Zeitstempel t1 bis t4

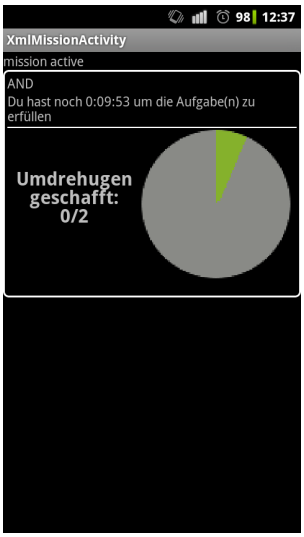

Abbildung 6: Eine Spin-Task gekoppelt mit einer zeitlichen Grenze

1. Es wurde bei der Registrierung des Sensors durch `SENSOR_DELAY_FASTEST` [4] die schnellstmögliche Lieferung von neuen Werten beantragt und
2. für fehlende Werte gibt es einen Interpolationsmechanismus. Dieser trägt fehlende Werte ein, wenn sie sinnvoll erscheinen - falls zwischen zwei aufeinanderfolgenden Werten klar einer oder zwei fehlen.

Durch diese Maßnahmen werden Drehungen meist richtig erkannt.

5.2.2. UI

Die graphische Oberfläche der Spin-Task ist etwas komplexer als die des Eierlaufs - sie besteht aus einem horizontalen `LinearLayout` [12] in welches eine `TextView` [13] und eine selbstdefinierte `PicView` eingebettet sind.

Die `PicView` ist dabei ähnlich wie die Oberfläche des Eierlaufs - es werden wieder Bilddateien auf ein `canvas` gemalt. Insgesamt werden 5 Bilddateien genutzt - eine für den Hintergrund (der graue Kreis) und vier für die Winkel 22,5°, 45°, 77,5° und 90°. Diese Winkel werden dann jeweils um 90°, 180° oder 270° gedreht, um die Winkel 112,5° bis 360° darzustellen. Dieses Vorgehen wurde gewählt, da eine Drehung um einen Winkel, der kein Vielfaches von 90° ist, ein Bild verzerren kann, die Dateigröße und Arbeitsspeicherauslastung aber dennoch minimiert werden sollte.

5.3. Timer

Der Timer ist ein Task ohne Sensorenabhängigkeit. Stattdessen wird hier die interne Uhr des Mobilgerätes ausgewertet und genutzt um zeitliche Grenzen für Aufgaben zu setzen, wobei es zwei unterschiedliche Modi gibt.

5.3.1. Modus `limit`

Der `limit`-Modus bietet eine obere zeitliche Grenze - innerhalb einer gegebenen Zeit muss eine Aufgabe also erfüllt werden. Beispiele für solche Aufgaben wären *Laufe 100m in unter einer Minute* oder *Drehe dich innerhalb von 10 Sekunden zwei Mal um dich selbst*. Nach Ablauf der gegebenen Zeit gilt die Aufgabe als verloren.

Die Syntax ist hier sehr simpel:

Codebeispiel 5: Timer: Eine obere zeitliche Grenze

```
1 <task type="Timer" mode="limit" time="0:0:10" />
```

Das Attribut `time` nimmt also eine Zeitangabe im Format `h:m[m]:s[s]`.

5.3.2. Modus `countdown`

Der `countdown`-Modus ist im Gegensatz zu `limit` eine Task, die gewonnen wird sobald die vorgegebene Zeit erreicht ist. Damit werden Aufgaben wie z.B. *Lasse das Ei 2 Minuten lang nicht fallen* möglich.

Die Syntax ist fast identisch zu `limit`:

Codebeispiel 6: Timer: Vorgabe der notwendigen Dauer

```
1 <task type="Timer" mode="countdown" time="0:2:0" />
```

Auch hier nimmt `time` eine Zeitangabe im Format `h:m[m]:s[s]`.

5.4. TrackDistance

TrackDistance bietet verschiedene Möglichkeiten, um mit Hilfe des GPS-Sensors Entfernungen zu messen. Die erreichbare Genauigkeit der GPS-Positionen hängt stark von der Umgebung und sogar dem Wetter ab. Alle Modi bieten daher die Möglichkeit, über das optionale Attribut `accuracy` die gewünschte Genauigkeit in Metern anzugeben. Bei fehlendem Attribut wird eine Genauigkeit von 15m erwartet. Die Aufgabe wird nicht starten ehe die geforderte Genauigkeit des GPS-Signals erreicht ist und wird - falls nötig - die Mission pausieren, sollte die Signalqualität während der Aufgabe wieder abnehmen. Die Genauigkeit sollte in Verbindung mit den Zielen oder Grenzen betrachtet werden. Liefert der GPS-Sensor nur eine Genauigkeit von 15m, ist es nicht sinnvoll bei den Entfernungen Werte von unter 15m zu verwenden, da die ermittelte Position des Spielers ohne eine Bewegung seinerseits um diesen Wert schwanken kann.

Es stehen insgesamt vier Modi für diese Aufgabe zur Verfügung, die im folgenden genauer erläutert werden.

5.4.1. Modus `pathlength`

Der Spieler muss eine bestimmte Strecke zurücklegen.

Codebeispiel 7: TrackDistance im Modus pathlength

```
<task type="TrackDistance" mode="pathlength"
      distance="20" accuracy="30"/>
```

Dieser Modus erwartet als einziges zusätzliches Attribut `distance` mit einer Entfernungsangabe in Metern. Sobald diese Strecke zurückgelegt wurde, ist die Aufgabe für den Spieler gewonnen.

5.4.2. Modus `minDistanceFromLocation`

Der Spieler muss einen festgelegten Mindestabstand von einer Referenzposition erreichen.

Codebeispiel 8: TrackDistance im Modus minDistanceFromLocation

```
<task type="TrackDistance"
      mode="minDistanceFromLocation" radius="50"
      latlong="50.751557,7.096937" />
```

Über das Attribut `radius` wird die Entfernung in Metern angegeben, die sich der Spieler mindestens von der Referenzposition entfernen muss. Die Position kann dabei über das Attribut `latlong` festgelegt werden. Erwartet wird eine Positionsangabe über geographische Breite (latitude) und Länge (longitude). Ist kein `latlong`-Attribut gegeben, so wird die aktuelle Position zu Beginn der Aufgabe als Referenzposition verwendet. Sobald sich der Spieler außerhalb des Radius um die Referenzposition befindet, ist die Aufgabe gewonnen.

5.4.3. Modus `reachLocation`

Der Spieler muss eine Position erreichen bzw. sich ihr bis zu einen festgelegten Radius nähern.

Codebeispiel 9: TrackDistance im Modus reachLocation

```
<task type="TrackDistance" mode="reachLocation"
      radius="50" latlong="50.751557,7.096937" />
```

Das Attribut `radius` gibt einen Radius (in Metern) um die Zielposition `latlong` vor. Sobald sich der Spieler innerhalb des Radius um die Zielposition befindet, ist die Aufgabe gewonnen.

5.4.4. Modus `stayAtLocation`

Der Spieler darf sich nur innerhalb eines Radius um die gegebene Position bewegen. Verläßt er den Kreis, ist die Aufgabe verloren.

Codebeispiel 10: TrackDistance im Modus stayAtLocation

```
<task type="TrackDistance" mode="stayAtLocation"
      radius="50" latlong="50.751557,7.096937" />
```

Das Attribut `radius` bestimmt den Radius, in dem sich der Spieler um die Referenzposition `latlong` aufhalten muss. Fehlt die Positionsangabe, so wird die aktuelle Position zu Beginn der Aufgabe als Referenzposition festgelegt. Sobald sich der Spieler außerhalb des Radius um die Referenzposition befindet, ist die Aufgabe verloren.

5.5. TrackSpeed

TrackSpeed bietet mehrere Möglichkeiten an um Geschwindigkeiten in Aufgaben zu verwenden. Die Geschwindigkeit wird dabei durch den GPS-Sensor ermittelt. Bedingt durch das Funktionsprinzip bei der Bestimmung der Geschwindigkeit, ist die ermittelte Geschwindigkeit wesentlich genauer als die ermittelte Position. TrackSpeed bietet genau wie TrackDistance (↪ 5.4) das Attribut `accuracy` an, um die geforderte Genauigkeit in Metern festzulegen. Auch hier ist der Standardwert 15, falls der Parameter fehlt. Ein wichtiger Unterschied zur Positionsbestimmung ist hier, dass auch bei schlechtem GPS-Empfang die ermittelte Geschwindigkeit, im Gegensatz zur ermittelten Position, nur wenig schwankt. Steht der Spieler still, so ist auch die ermittelte Geschwindigkeit 0 oder sehr klein im Betrag. Alle Modi erwarten als einziges Attribut `speed`, welches die gewünschte Zielgeschwindigkeit in $\frac{m}{s}$ (Meter pro Sekunde) bestimmt.

5.5.1. Modus `reachSpeed`

Ziel ist es, eine bestimmte Geschwindigkeit zu erreichen. Sobald die Geschwindigkeit erreicht wurde, ist die Aufgabe

gewonnen. Auf diese Weise kann z.B. ein kurzer Sprint des Spielers gefordert werden.

Codebeispiel 11: TrackSpeed im Modus reachSpeed

```
<task type="TrackSpeed" mode="reachSpeed"
      speed="4.16" />
```

5.5.2. Modus `stayBelowSpeed`

Dieser Modus verfolgt ein teilweise entgegengesetztes Ziel zum Modus `reachSpeed` (↪ 5.5.1):
Der Spieler darf die vorgegebene Geschwindigkeit nicht überschreiten, sonst ist die Aufgabe verloren. Auf diese Weise kann der Spieler beispielsweise gezwungen werden, sich sehr langsam zu bewegen oder die Benutzung anderer Fortbewegungsmittel (z.B. Bus und Bahn) kann unterbunden werden.

Codebeispiel 12: TrackSpeed im Modus stayBelowSpeed

```
<task type="TrackSpeed" mode="stayBelowSpeed"
      speed="4.16" />
```

5.5.3. Modus `stayAboveSpeed`

Auch dieser Modus ist eine Begrenzung für den Spieler und verbietet es ihm unter eine bestimmte Geschwindigkeit zu fallen. Vom Spieler kann so also beispielsweise verlangt werden eine Strecke zu laufen, ohne zwischendurch eine Pause machen oder das Tempo reduzieren zu dürfen.

Codebeispiel 13: TrackSpeed im Modus stayAboveSpeed

```
<task type="TrackSpeed" mode="stayAboveSpeed"
      speed="2.78" />
```

Da der Spieler zu Beginn der Aufgabe die geforderte Geschwindigkeit eventuell noch nicht erreicht hat, wird die Grenze erst aktiv sobald die Geschwindigkeit einmalig erreicht wurde. Da findige Spieler dies ausnutzen könnten und die Aktivierungsgeschwindigkeit dauerhaft unterschreiten könnten, ist es unter Umständen sinnvoll die Aufgabe mit einer TaskList (↪ 5.7) zu kombinieren und vorher TrackSpeed im Modus `reachSpeed` (↪ 5.5.1) zu starten - dies zwingt den Spieler die Geschwindigkeit auch wirklich zu erreichen.

5.6. And / Or

Diese beiden Klassen entsprechen den logischen Operatoren *UND* und *ODER*. Sie sind zwei von drei `LogicalTasks` - einer abstrakten Klasse, deren Kindklassen eine Liste von Tasks beinhalten. Durch And, Or und TaskList (↪ 5.7) ist es also möglich, verschiedene Aufgaben zu verbinden, die im Weiteren als *Subtasks* bezeichnet werden. Dabei können alle drei dieser `LogicalTasks` eine beliebige Anzahl an Subtasks enthalten. Diese werden zeitgleich ausgeführt, wobei die graphische Oberfläche von And bzw. Or eine Kombination der Oberflächen der Subtasks durch ein vertikales `LinearLayout` [12] ist.

Die in der `Task`-Klasse spezifizierten Funktionen werden, dem Composite Pattern [9] entsprechend, größtenteils so implementiert, dass And / Or diese in den verschiedenen Subtasks aufruft. Lediglich `hasGoal()` bildet hier eine Ausnahme, da es sich hier um eine Abfrage handelt, die von unterschiedlichen Subtasks unterschiedlich beantwortet werden kann. Die Lösung dieses Problems: Falls eine der Subtasks gewinnbar ist, wird auch And / Or als gewinnbar gehandhabt. Diese Vorgehensweise ist deshalb sinnvoll, weil bei einer Task, die nicht gewonnen werden kann, der Zustand *nicht verloren* bei einem And gleichbedeutend mit *gewonnen* ist und bei einem Or die Task gewonnen ist sobald sich eine Subtask als *gewonnen* meldet.

Die Verwendung von And und Or sieht man in Codebeispiel 17.

5.7. TaskList

TaskList ermöglicht es mehrere Tasks nacheinander auszuführen. Sobald die aktuell aktive Subtask gewonnen wurde, wird die nächste Subtask gestartet. Das setzt natürlich voraus, dass alle Subtasks, bis auf die letzte Subtask, gewonnen werden können - eine Task die kein Ziel hat, würde ein Vorankommen in der Liste unmöglich machen. Da TaskList den aktuellen Zustand der jeweils aktiven Subtask widerspiegelt, verhält sich TaskList nach außen hin wie ein einzelner Task. Entsprechend bestimmt die aktuell aktive Subtask, ob TaskList ein Ziel hat oder nicht.

Das folgende Beispiel soll diesen Zusammenhang etwas verdeutlichen.

Codebeispiel 14: Die Definition einer TaskList

```
<?xml version="1.0" encoding="UTF-8"?>
<mission>
  <task type="And">
    <task type="TrackDistance" mode="pathlengt" distance
        ="20" />
    <task type="TaskList">
      <task type="TrackDistance" mode="pathlengt"
          distance="5" />
      <task type="Eierlauf" />
    </task>
  </task>
</mission>
```

Wie zu erkennen ist, enthält die Task `TaskList` als Unterelemente die gewünschten Subtasks. Attribute gibt es bei dieser Task nicht.

Von dem Spieler wird in diesem Beispiel erwartet, dass er insgesamt eine Strecke von 20m läuft. Nach den ersten 5m ist die erste Subtask von TaskList gewonnen und die Subtask Eierlauf startet. Der Eierlauf wiederum hat nun kein Ziel und somit auch TaskList nicht. Sobald jedoch die

restlichen 15m mit dem Ei gelaufen wurden, ist die Mission gewonnen. Hinter Eierlauf könnte noch eine weitere Subtask stehen; sie würde jedoch nie gestartet werden.

5.8. Info - „Was muss ich tun?“

Die Info-Task ist dazu gedacht, Informationen vor der Ausführung anderer Tasks darzustellen. Sie kann aber beispielsweise auch dazu verwendet werden, den Start einer anderen Task (z.B. eines Eierlaufs (↪ 5.1)) zu verzögern bis der Nutzer bereit ist.

Die Funktionsweise ist einfach: Es wird ein Text und (optinal) ein Bild eingeblendet und bei einer Berührung des Touchscreens ist die Aufgabe gewonnen. Sinnvollerweise wird ein Info-Bildschirm also in einer TaskList (↪ 5.7) dargestellt.

Die Nutzung des Touchscreens ist mitweilen uneindeutig, da es (wie bereits in Abschnitt 2.1 erwähnt) verschiedene Typen von Events gibt. Legt man den Finger auf den Bildschirm wird dies durch ein `ACTION_DOWN`-Event gekennzeichnet, eine Bewegung wird durch ein `ACTION_MOVE` signalisiert und das Anheben des Fingers löst ein `ACTION_UP` aus. Allerdings gibt es beim Berühren des Bildschirms meist kleine Bewegungen, wodurch schnell ein `ACTION_MOVE`-Event auftritt obwohl der Nutzer oftmals nur berühren und loslassen wollte. Das wird dann problematisch, wenn gescrollt werden soll - was bei längeren Aufgaben-Beschreibungen sinnvoll (und bei allen Tasks möglich) sein soll. Aus diesem Grund zeichnet die Info-Task die Berührungspunkte bei einem `ACTION_DOWN` oder `ACTION_MOVE`-Event auf und vergleicht sie bei einem `ACTION_UP`-Event mit der aktuellen Position - nur wenn die Bewegung sehr gering ist wird eine *Tap* statt einer *Scroll*-Aktion angenommen.

Definiert werden solche Info-Tasks durch den folgenden Code:

Codebeispiel 15: Die Definition eines Info-Tasks

```
1 <task type="Info" image="figurines.jpg" height="1.1"
2       text="Bitte den Bildschirm berühren um die Aufgabe
              zu starten."/>
3 <task type="Info" image="figurines.jpg" width="1.1">Die
      Aufgabe ist geschafft.
4   Das war einfach, oder?
5 </task>
```

Wie man sieht, gibt es zwei alternative Möglichkeiten, Text zu definieren: entweder mit dem Attribut `text`, was für Text ohne manuelle Zeilenumbrüche völlig ausreichend ist, oder als Inhalt des `<task>`-Tags.
Außerdem gibt es noch die (optionalen) Attribute

- `image`, mit dem ein Bild eingebunden werden kann. Der Dateipfad wird dabei als relativ zur Missionsdatei interpretiert oder, bei voller Pfadangabe, absolut angegeben und die Größe wird im Standardfall automatisch auf die Bildschirmgröße angepasst.

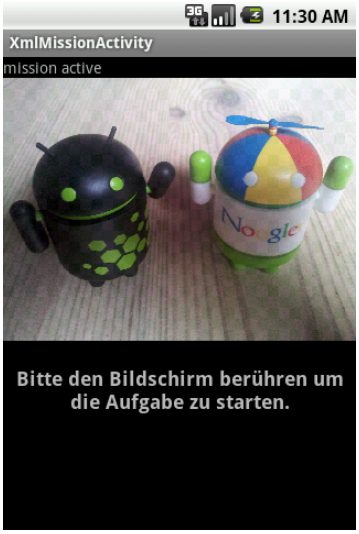

Abbildung 7: Ein Info-Bildschirm

- `height`, mit dem das Bild relativ zur automatischen Größe skaliert werden kann. Die Werte sind hierbei vom Typ `float`, wobei 1 hier 100% entspricht. Wird dieses Attribut ohne `width` angegeben, so wird das ganze Bild so skaliert, dass die Höhe der gewünschten Skalierung entspricht.

- `width`, mit dem die Bildbreite relativ zur automatischen Größe verkleinert oder vergrößert werden kann. Die Werte sind hierbei vom Typ `float`, wobei 1 hier 100% entspricht. Wird dieses Attribut ohne `height` angegeben, so wird das ganze Bild so skaliert, dass die Höhe der gewünschten Skalierung entspricht.

Werden sowohl das Attribut `height` als auch das Attribut `width` angegeben, so werden die Höhe und Breite separat skaliert.

6. Fazit

Das Ziel des Projektes ARTUS war es, für Applikationen wie z.B. GEOQUEST eine Bibliothek zu schaffen, die es Entwicklern erlaubt schnell und einfach physische Aufgaben zu stellen und die Erfüllung dieser zu überprüfen.

Es gibt, wie sich herausgestellt hat, einige Einschränkungen, die für Probleme während der Entwicklung neuer Tasks sorgen können; ein solches Problem ist die Zeit, die der GPS-Sensor benötigt um einen ausreichend genauen Fix zu bekommen. Dadurch ist die schnelle Folge zweier GPS-Tasks z.T. schwierig, wie im Abschnitt über den TrackSpeed Modus *stayAboveSpeed* (↪ 5.5.3) klar geworden ist. Ebenfalls ein Problem ist die teilweise Ungenauigkeit der Sensoren - ob es sich nun um den Orientierungssensor handelt, der z.T. zu langsam Werte liefert oder um

den GPS-Sensor, der nur eine ungenaue Position angeben kann.

Dennoch, entstanden ist eine nützliche Sammlung von einzelnen, miteinander kombinierbaren Aufgaben. Diese Sammlung kann größtenteils leicht erweitert werden, da die Anforderungen an neue Tasks klar geregelt sind und nur sehr wenige Einschränkungen bestehen. Ideen für zukünftige Erweiterungen wären beispielsweise die Erkennung von Schritten (Stichwort: *Pedometer*) oder anderen ähnlichen Aufgaben, beispielsweise *Treppen steigen* oder *Hüpfen*. Während diese aufwendiger sind als die bestehenden Tasks ist es möglich, diese in das bestehende System einzubauen. Somit bleiben für die Zukunft viele Türen offen.

Abbildung 8: Das Icon von ARTUS

A. Anhang: Beispiele

Codebeispiel 16: Praesentation.xml - ein vollständiges Beispiel einer Aufgabenstellung

```
<?xml version="1.0" encoding="UTF-8"?>
<mission>
  <task type="TaskList">
    <task type="Info" image="figurines.jpg" width=".8">
        Dies ist die Präsentation des Projektes ''ARTUS
        ''.
Spezifikationen:
- Zeitgrenze 5 Minuten
- 3 mal um dich selbst drehen
- dann 30m laufen oder
- einen Radius von 15m nicht verlassen
- und dabei den weißen Kreis den roten Kreis nicht
    berühren lassen
    </task>
    <task type="And">
      <task type="Timer" mode="limit" time="0:5:0"/>
      <task type="TaskList">
        <task type="Spin" howoften="3"/>
        <task type="And">
          <task type="TrackDistance" mode="pathlength"
                distance="30" />
          <task type="TrackDistance"
                mode="stayAtLocation" radius="15" />
          <task type="Eierlauf" sensitivity=".7" />
        </task>
      </task>
    </task>
  </task>
</mission>
```

Codebeispiel 17: Ein Eierlauf (mit Standart-Empfindlichkeit) über eine Zeit von 5 Minuten ODER eine Strecke von 500m

```
<?xml version="1.0" encoding="UTF-8"?>
<mission>
  <task type="And">
    <task type="Or">
      <task type="Timer" mode="countdown" time="0:5:0"/>
      <task type="TrackDistance" mode="pathlength"
          distance="500" accuracy="20" />
    </task>
    <task type="Eierlauf" />
  </task>
```

B Anhang: Verzeichnisse

Literatur

[1] http://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html, abgerufen am 23. Februar 2012

[2] http://www.apple.com/pr/library/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta.html, abgerufen am 23. Februar 2012

[3] http://developer.android.com/reference/android/hardware/SensorEvent.html, abgerufen am 23. Februar 2012

[4] http://developer.android.com/reference/android/hardware/SensorManager.html#SENSOR_DELAY_FASTEST, abgerufen am 28. Februar 2012

[5] http://developer.android.com/reference/android/location/Location.html, abgerufen am 24.Februar 2012

[6] Parnandi et al., Coarse In-Building Localization with Smartphones, MobiCASE 2009, LNICST 35, pp. 367-378, 2010

[7] J.-Y. Yang et al., Using acceleration measurements for activity recognition: An effective learning algorithm for constructing neural classifiers, Pattern Recognition Letters 29, pp. 2213-2220, 2008

[8] Saponas et al., iLearn on the iPhone: Real-Time Human Activity Classification on Commodity Mobile Phones, CSE Technical Report, 2008.

[9] Kniesel, G.: Das Composite Pattern. In: Einführung in die Softwaretechnologie, Wintersemester 2011, Kapitel 6: Entwurfsmuster (23. November 2011)

[10] Kniesel, G.: Das Observer Pattern. In: Einführung in die Softwaretechnologie, Wintersemester 2011, Kapitel 6: Entwurfsmuster (23. November 2011)

[11] http://developer.android.com/reference/android/view/View.html, abgerufen am 24. Februar 2012

[12] http://developer.android.com/reference/android/widget/LinearLayout.html, abgerufen am 26. Februar 2012

[13] http://developer.android.com/reference/android/widget/TextView.html, abgerufen am 26. Februar 2012

Quellcode

Abbildungsverzeichnis