Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 18

Turn by Turn Navigation für Android Mobilgeräte

Christoph Haag

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	14. Juni 2012
beendet am:	14. Dezember 2012
CR-Klassifikation:	J.7

Kurzfassung

Die bei einem Studienprojekt entstandene Routenplanungs-Software “ToureNPlaner” wird mit dieser Bachelorarbeit für den praktischen mobilen Einsatz angepasst. Zu diesem Zweck wird das ToureNPlaner System um Funktionen für die Turn-by-Turn Navigation erweitert. Darunter wird ein System verstanden, das anhand der per GPS ermittelten Position mittels einer Sprachausgabe Navigationsanweisungen gibt. Die Implementierung erfolgt in Form eines Client-Server Systems auf Basis einer PostGIS Datenbank.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Smartphones haben in den letzten Jahren immer größere Verbreitung gefunden. Smartphones besitzen eine Vielzahl von Möglichkeiten, die Position des Benutzers in der realen Welt zu bestimmen, unter Anderem mittels GPS Empfänger, anhand der in der Nähe befindlichen WLANs oder anhand des GSM Netzes, die auch viele ortsabhängige Applikationen verwenden. Dieser Entwicklung liegt der Wunsch der Benutzer zu Grunde, sich auch an unbekannten Orten zurechtzufinden und eine Navigationshilfe zu erhalten. Eine solche ortsabhängige Anwendung wird im Rahmen dieser Arbeit speziell für die Integration in das ToureNPlaner System entwickelt.

"ToureNPlaner" ist ein System, das aus einem Studienprojekt an der Universität Stuttgart hervorging. Es besteht aus einem Server und mehreren Clients, darunter einem Client für Android Mobilgeräte. Der ToureNPlaner Server berechnet Routen mit diversen Algorithmen, die zum Beispiel NP-schwere Probleme approximieren oder auch nur kürzeste bzw. schnellste Wege, die die Clients anschließend visualisieren.

Dem Android Client, der besonders für eine mobile Anwendung konzipiert ist, fehlen jedoch einige Funktionen, um ihn sinnvoll wie ein konventionelles Navigationsgerät einsetzen zu können. Mit dem vorgestellten System wird die Grundfunktionalität eines solchen Systems in Form der Turn-by-Turn Navigation implementiert und damit die Grundlage für weitere verwandte Funktionen gelegt. Unter der Turn-by-Turn Navigation ist ein System zu verstehen, das den Benutzer stetig über den weiteren Verlauf der gerade befahrenen Route informiert. Typischerweise wird dafür eine Sprachausgabe verwendet.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Überblick: Hier wird ein Überblick über die bisherige Funktionsweise des ToureNPlaner Systems sowie des vorgestellten Systems gegeben.

Kapitel 3 – Umsetzung der OpenStreetmap Datenbank: Hier werden die Überlegungen zur OpenStreetMap Datenbank beschrieben.

Kapitel 4 – Implementierung: Hier werden die Implementierung des Turn-by-Turn Servers sowie der Zusatzfunktionen des Androidclienten im Detail beschrieben.

Kapitel 5 – Zusammenfassung und Ausblick Hier wird das Fazit über die Arbeit sowie weitere Anknüpfungspunkte gegeben.

2. Überblick

2.1. Bisherige Funktionsweise des ToureNPlaner Systems

Der Benutzer hat die Wahl zwischen einem Webclient und einem Androidclient. In dieser Arbeit soll es nur um den Androidclient gehen.

Beim Starten des Clients fragt der Client den ToureNPlaner Server nach unterstützten Algorithmen (z.B. ShortestPath, TSP, CSP). Der Benutzer kann dann einen dieser Algorithmen auswählen und Markierungen auf einer Landkarte setzen. Diese Markierungen werden an den ToureNPlaner Server gesendet, dieser führt dann den entsprechenden Algorithmus aus und sendet das Ergebnis als Route an den ToureNPlaner Client.

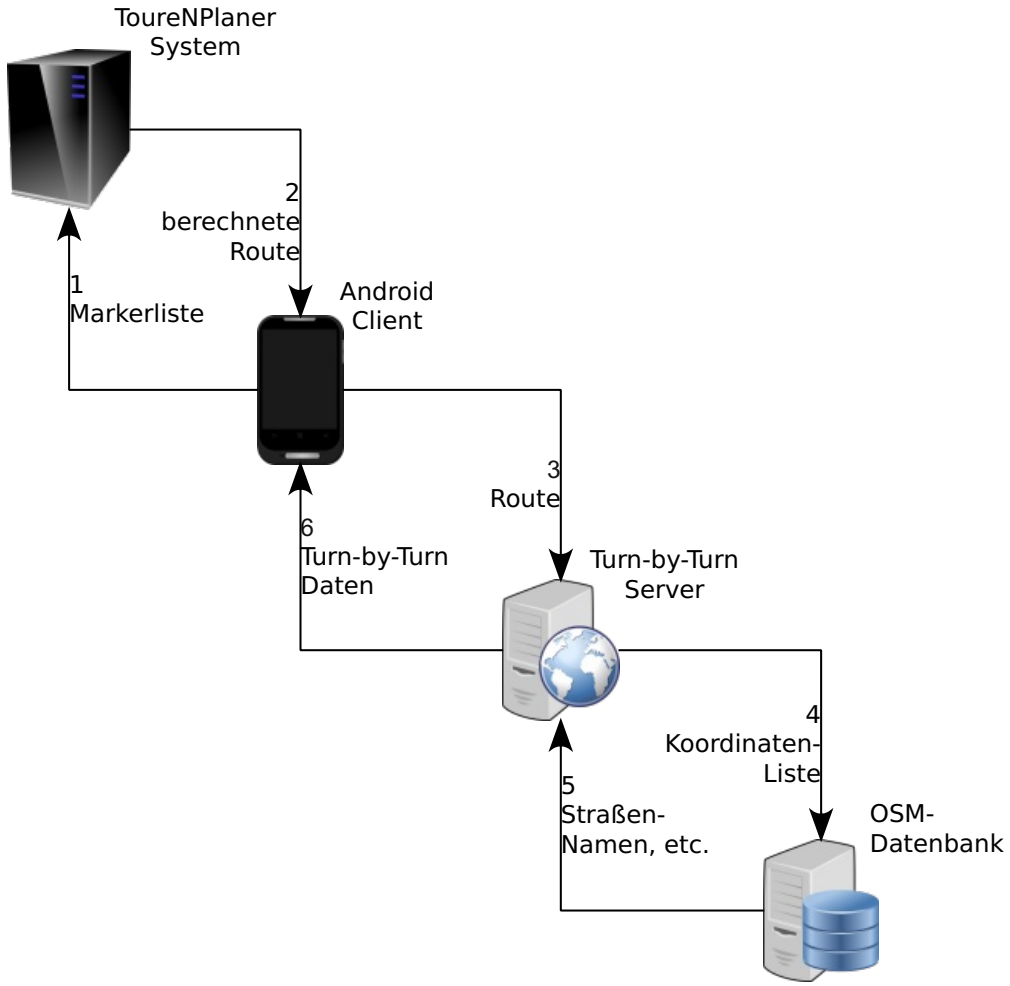
2.2. Neue Funktionalität für den Androidclient

Bisher unterstützt der Androidclient nur das Anzeigen der Route auf der Landkarte, sowie Metadaten wie die Länge der Route oder die Gesamtreisezeit. Für einen praktischen Einsatz fehlen dem Androidclienten einige Funktionen, die von einem Navigationsgerät erwartet werden können:

1. zu den Koordinaten passende Straßennamen
2. sprachliche Hinweise auf bevorstehende “interessante” Streckenabschnitte
    - frühzeitige Ansage für das Wechseln auf eine andere Straße
    - Ansage der Richtung des nächsten Straßenwechsels

2.3. Systemarchitektur

Das Turn-by-Turn System, das für die Verwendung in Verbindung mit dem ToureNPlaner System konzipiert ist, besteht aus einer Serverapplikation sowie einer Erweiterung des bestehenden Androidclienten. Sobald die Interaktion mit dem ToureNPlaner Server abgeschlossen ist und der Androidclient die Route anzeigt, soll die Möglichkeit bestehen, die Turn-by-Turn Navigationsdaten von diesem Server abzurufen. Daraufhin soll die Turn-by-Turn Navigation des Androidclienten gestartet werden.



[a]

Abbildung 2.1.: Übersicht über die Requests

[a] Quellen für die Bilder:
http://openclipart.org/detail/172335/smartphone-by-maw-172335
http://openclipart.org/detail/5159/server-cabinet-cpu-by-sagar_ns
http://openclipart.org/detail/163741/web-server-by-lyte
http://openclipart.org/detail/163711/database-server-by-lyte

Der Turn-by-Turn Server soll also anhand der Koordinaten und unter Benutzung der OSM-Datenbank bestimmen können, zu welcher Straße die jeweilige Koordinate gehört und an welchen Punkten die Straße gewechselt werden muss.

2.4. Benutzeroberfläche

Der typischen Bedienungsablauf geschieht folgendermaßen:

1. Mit einer Route, die vom ToureNPlaner Server berechnet wurde, wählt der Benutzer die Turn-by-Turn Navigation im Menü.

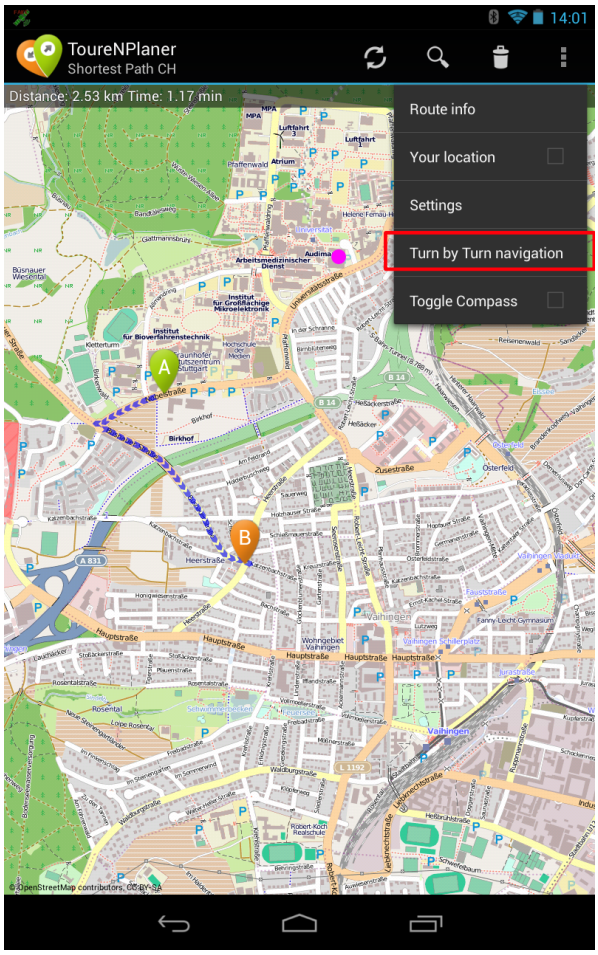


Abbildung 2.2.: Button zum Starten der Turn-by-Turn Navigation

2. Der aktuelle Standort wird als neuer Startpunkt zur Route hinzugefügt und die Turn-by-Turn Navigation beginnt. Zusätzlich wird die schon vorhandene Option “Ihre

Position" aktiviert, die bei jedem Location Update die Kartenansicht auf die GPS-Position zentriert. Jede Sprachausgabe wird zusätzlich als unaufdringliche Popup-Nachricht (Android "Toast") in Textform angezeigt.

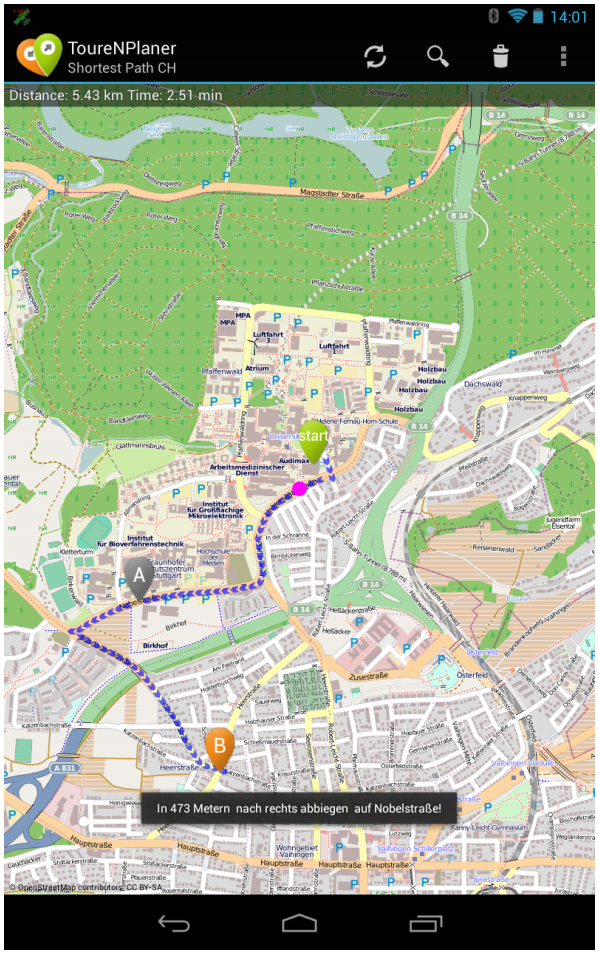


Abbildung 2.3.: Start der Turn-by-Turn Navigation

3. Wenn sich der Benutzer dem Ziel bis auf ca. 20 Meter angenähert hat, wird die Turn-by-Turn Navigation beendet.

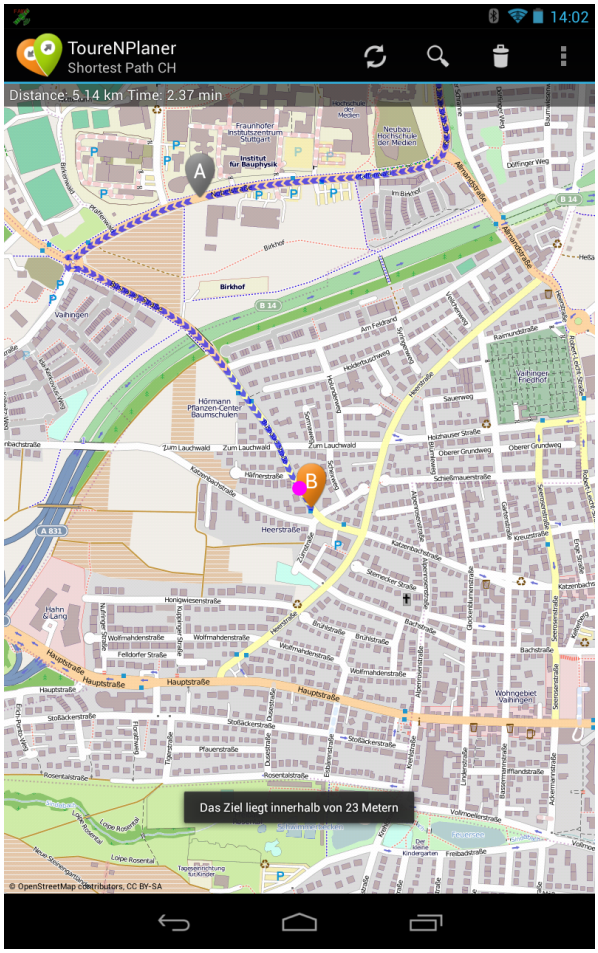


Abbildung 2.4.: Ende der Turn-by-Turn Navigation

Zusätzlich kann der Benutzer sich auf Wunsch auch noch die Route in Listenform anzeigen lassen. Straßen, deren Namen nicht bekannt sind oder die namenlos sind, werden speziell gekennzeichnet.

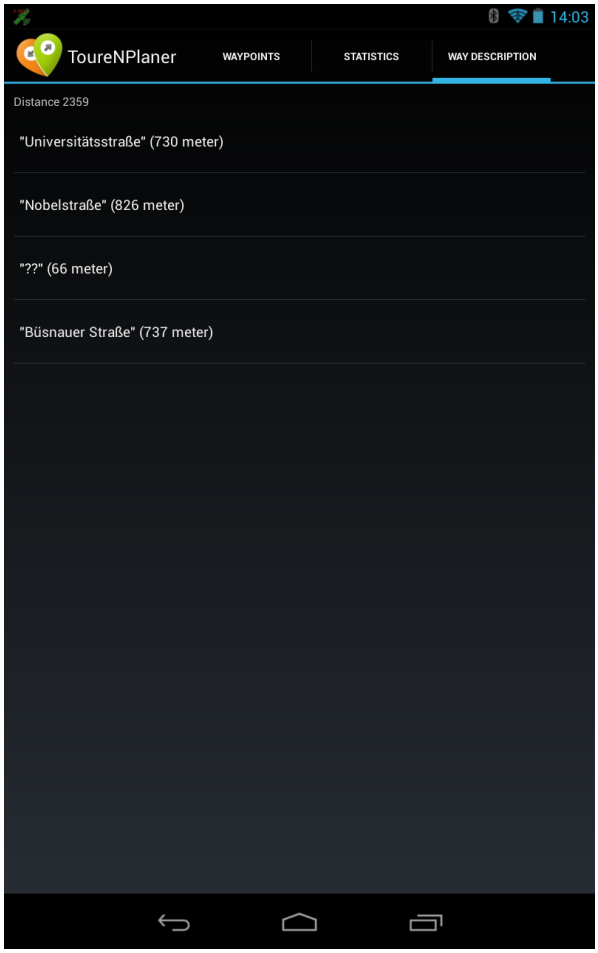


Abbildung 2.5.: Ablauf der Route in Listenform

(Die Screenshots in diesem Kapitel wurden mit Hilfe der App “Fake GPS location”[fakegps] erstellt).

3. Umsetzung der OpenStreetmap Datenbank

3.1. Übersicht

Um für eine vorgegebene Liste von Koordinaten die entsprechenden Straßennamen herauszufinden, wird auf jeden Fall ein Zugriff auf die OpenStreetmap Daten benötigt. Die vorgestellte Serverlösung besteht aus einer Serverapplikation sowie einer Datenbank, welche die benötigten OSM-Daten hält.

Als Datenbank wird für diese Arbeit die für geographische Aufgaben verbreitete Kombination von PostgreSQL mit den PostGIS Erweiterungen [postgis] verwendet.

3.1.1. Verwendung der OSM Daten in der PostgreSQL Datenbank

Das Programm "Osmosis" legt jeweils die OSM-Knoten, die OSM-Wege und die Relation der Knoten zu den Wegen in einer eigenen Tabelle ab. Ein Query für den nahesten OSM-Knoten zu gegebenen Koordinaten, der Teil einer Straße ist (OSM: Der Tag "highway" ist gesetzt), verwendet zwei JOINs.

Das Beispiel 3.1 berechnet den OSM Knoten, der die geringste Distanz zu den Koordinaten (48.8089539, 9.1851113) hat und der zu einem "highway" gehört. Mit der LIMIT-Angabe kann bestimmt werden, wie viele "naheste" Knoten berechnet werden.

Listing 3.1 Nachbarknoten-Berechnung 1

```
SELECT nodes.id,ways.tags::hstore -> 'name'
FROM nodes JOIN way_nodes ON nodes.id = way_nodes.node_id JOIN ways ON way_nodes.way_id =
    ways.id
WHERE ways.tags::hstore ? 'highway'
ORDER BY geom <-> ST_GeomFromEWKT('SRID=4326;POINT(9.1851113 48.8089539)')
LIMIT 1;
```

3.2. Optimierungen

Folgende weitere Optimierungsmöglichkeiten wurden evaluiert:

3.2.1. Prepared Statements

Ein Speedup kann erwartet werden, da die Datenbank Prepared Statements im Vorfeld schon optimieren kann.

Zum Beispiel ist schon bekannt, welche JOINS durchgeführt werden müssen. Im Test hat diese Möglichkeit aber keine sichtbaren Verbesserungen erbracht, da die Queries relativ einfach sind und daher auch schon in der direkten Form schnell optimiert werden können.

Listing 3.2 Erstellen eines Prepared Statements

```
PREPARE nns1(text) AS
SELECT nodes.id,ways.tags::hstore -> 'name'
FROM nodes JOIN way_nodes ON nodes.id = way_nodes.node_id JOIN ways ON way_nodes.way_id =
    ways.id
WHERE ways.tags::hstore ? 'highway'
ORDER BY geom <-> ST_GeomFromEWKT($1)
LIMIT 1;
```

Ein Beispiel-Query mit Benutzung eines solchen Prepared Statements ist in 3.3 gegeben.

Listing 3.3 Beispielnutzung eines Prepared Statements

```
EXECUTE nns1('SRID=4326;POINT(9.1851113 48.8089539)');
```

3.2.2. neue Tabelle mit genau den benötigten Spalten

Wird eine neue Tabelle gemäß 3.4 erstellt, in der die Information aus den drei Tabellen "nodes", "ways" und "way_nodes" kombiniert werden, müssen keine JOINs mehr durchgeführt werden. Leider werden dabei aber sehr viele Informationen dupliziert abgespeichert.

Listing 3.4 Erstellen einer kombinierten Tabelle

```
SELECT DISTINCT nodes.id as node_id,nodes.tags as node_tags, ways.id AS
    way_id,way_nodes.sequence_id,ways.tags as way_tags,ways.nodes as way_nodes
INTO highway_nodes
FROM nodes JOIN way_nodes ON nodes.id=way_nodes.node_id JOIN ways ON way_nodes.way_id=ways.id
WHERE ways.tags::hstore ? 'highway';

# knoten sind evtl. in Kreisen (Kreisverkehre, etc.) am anfang und am ende eines ways
ALTER TABLE highway_nodes ADD PRIMARY KEY(node_id, way_id, sequence_id);

CREATE INDEX idx_nodes_geog ON highway_nodes USING GIST(geog);

VACUUM ANALYZE;
```

Wie in 3.5 zu sehen ist, werden die Queries dadurch einfacher, dass die JOINs sowie die Überprüfung auf die "highway" Tags entfällt.

Listing 3.5 Beispielquery an eine kombinierte Tabelle

```
SELECT id,geom FROM highway_nodes ORDER BY geom <->
    ST_GeomFromEWKT('SRID=4326;POINT(9.1851113 48.8089539)') LIMIT 10;
```

3.2.3. unnötige Daten aus nodes, ways und way_nodes entfernen

Die komplette OpenStreetmap Datenbank enthält viele Daten, die für die Turn-by-Turn Navigation nicht benötigt werden. Mit der in 3.6 beschriebenen Methode werden die nodes, ways und way_nodes mit jeweils nur den relevanten Spalten und Zeilen in neue Tabellen geschrieben:

Listing 3.6 Entfernen irrelevanter Spalten Zeilen aus allen Tabellen

```
SELECT DISTINCT ways.id, ways.tags, ways.nodes INTO highway_ways FROM ways WHERE
    ways.tags::hstore ? 'highway';
CREATE INDEX idx_highway_ways_id ON highway_ways(id);
ALTER TABLE highway_ways ADD PRIMARY KEY(id);

SELECT DISTINCT way_nodes.way_id, way_nodes.node_id, way_nodes.sequence_id INTO
    highway_way_nodes FROM way_nodes INNER JOIN highway_ways ON way_nodes.way_id =
    highway_ways.id;
CREATE INDEX idx_highway_way_nodes_node_id ON highway_way_nodes(node_id);
ALTER TABLE highway_way_nodes ADD PRIMARY KEY(node_id);
ALTER TABLE highway_way_nodes ADD FOREIGN KEY(way_id) REFERENCES highway_ways(id);

SELECT DISTINCT nodes.id, nodes.tags, nodes.geom INTO highway_nodes FROM nodes INNER JOIN
    highway_way_nodes ON nodes.id = highway_way_nodes.node_id;
ALTER TABLE highway_nodes ADD CONSTRAINT id_unique UNIQUE (id);
CREATE INDEX idx_highway_nodes_geom ON highway_nodes USING GIST(geom);
ALTER TABLE highway_way_nodes ADD FOREIGN KEY(node_id) REFERENCES highway_nodes(id);

GRANT ALL on highway_nodes, highway_way_nodes, highway_ways TO osm;

VACUUM ANALYZE;
```

Werden die originalen Tabellen "nodes", "ways" und "way_nodes" entfernt, schrumpft das Datenbankverzeichnis von über 40 Gigabyte auf nur rund 12 Gigabyte. Dies kann von vielen modernen Computern komplett im RAM gehalten werden. Zudem entfällt die Prüfung für die Bedingung, dass der Knoten in einem "way" mit einem "highway"-Tag sein muss, da nur noch solche Knoten in der Datenbank stehen. Ein Beispiel-Query ist in 3.7 gegeben.

Listing 3.7 Beispiel-Query an eine Tabelle mit ausschließlich "highway"-Daten

```
SELECT highway_nodes.id,highway_ways.tags::hstore -> 'name'
FROM highway_nodes JOIN highway_way_nodes ON highway_nodes.id = highway_way_nodes.node_id
    JOIN highway_ways ON highway_way_nodes.way_id = highway_ways.id
ORDER BY geom <-> ST_GeomFromEWKT('SRID=4326;POINT(9.1851113 48.8089539)')
LIMIT 1;
```

3.2.4. alle Nearest Neighbour Suchen in einem Query vereinen

Für eine sinnvolle Anwendung muss für jede gegebene Koordinate ein Query der obigen Form ausgeführt werden. Man kann hier das Faktum nutzen, dass die Datenbank eine Suche auf dem Index ausführt, indem man sie nach allen Koordinaten gleichzeitig suchen lässt, statt jeweils sequenziell neue Suchläufe zu starten. Anschließend müssen die Ergebnisse aber geeignet konkateniert werden.

3.2.5. Vorberechnung der drei nächsten Nachbarn für jeden Knoten

Angesichts der gemessenen Zeitdauern war die Berechnung hierfür leider zu lange, um praktikabel zu sein. Der Aufwand für den Speicherplatz wäre aber vertretbar.

3.2.6. Fazit und Benchmarks

Das Aufbrechen der Normalform mit dem Erstellen einer einzigen Tabelle mit kombinierten Informationen erhöht sogar den Speicherbedarf und ist nicht zu empfehlen.

Das Löschen der Zeilen, die für systemirrelevante Informationen enthalten, ist für viele Systeme die wichtigste Optimierung, da es den Speicherbedarf um einen Faktor von rund $\frac{2}{3}$ reduziert.

# of neighbours	shortest/s	longest/s	total/s
1	0.0006	0.0359	4.6079
2	0.0009	0.0393	6.5801
3	0.0011	0.0419	6.1541
4	0.0006	0.0247	4.7222
5	0.0008	0.0363	7.0394
6	0.0011	0.0374	6.6814
7	0.0006	0.03	4.5051
8	0.0006	0.0413	6.4914
9	0.0007	0.0344	5.3854
10	0.0008	0.0391	6.0156
11	0.0008	0.0389	5.7161
12	0.0006	0.0359	5.74
13	0.0009	0.0355	6.6325
14	0.0006	0.0315	4.8616
15	0.0005	0.0328	5.5751
16	0.0011	0.0365	7.1228
17	0.0006	0.0378	5.4899
18	0.0015	0.0386	7.2358
19	0.0007	0.0409	6.8643
20	0.0007	0.0395	5.2538

Tabelle 3.1.: Suche der n nächsten OSM Knoten für jeweils 500 zufällige Koordinaten in Deutschland

Das Bestimmen der nächsten Nachbarknoten anhand gegebener Koordinaten dauert im Durchschnitt zwischen 9 und 15 Millisekunden, wie Tabelle 3.1 zeigt. Die Anzahl der nahesten Nachbarknoten spielt dabei eine kleinere Rolle spielt als andere, eher zufällige Faktoren.

# of neighbours	shortest/s	longest/s	total/s
1	0.0005	0.0398	4.9419
2	0.0006	0.0359	6.8653
3	0.0005	0.0414	5.9007
4	0.0011	0.0387	5.5615
5	0.0008	0.0397	7.3728
6	0.0005	0.0404	6.9991
7	0.0008	0.0354	5.2451
8	0.0008	0.0321	4.5407
9	0.0006	0.0391	5.5981
10	0.0012	0.0381	5.97
11	0.0006	0.0383	6.2596
12	0.0007	0.038	6.4901
13	0.0005	0.0353	5.4079
14	0.0005	0.0412	5.2648
15	0.0006	0.0395	5.4513
16	0.0009	0.0378	6.3207
17	0.0012	0.0391	7.2876
18	0.001	0.0389	7.1082
19	0.0011	0.0381	7.8879
20	0.0006	0.0394	6.8739

Tabelle 3.2.: Suche der n nächsten OSM Knoten für jeweils 500 zufällige Koordinaten in Deutschland mit Prepared Statements

Das Verwenden von Prepared Statements hat sich als unnötig herausgestellt. Wie Tabelle 3.2 zu entnehmen ist, unterscheiden sich die Abfragezeiten weniger als die zufälligen Variationen. Zudem ist eine Kombination von Prepared Statements mit dem UNION Schlüsselwort schwierig zu realisieren.

# coordinates	length/km	time/seconds
837	111,792	2,3533205986
948	59,895	1,950186491
1036	157,606	2,0537598133
1093	206,95	2,4053983688
1206	195,368	2,5820646286
1319	122,509	4,7641582489
1572	341,349	2,9500787258
1600	357,919	3,2501907349
1701	228,239	4,9144062996
1860	237,571	4,2341644764
1932	287,966	6,967851162
2213	258,386	5,7023663521
2291	420,362	6,2478981018
2378	287,238	8,2169313431
2411	457,205	6,3952562809
2452	166,965	8,4971950054
2460	460,434	4,3653018475
2574	422,535	8,4685773849
2634	507,537	7,3914530277
3107	507,066	6,3629086018
3176	410,502	8,7928514481
3187	314,559	12,6563251019
3345	515,444	8,6899342537
3393	539,892	7,3720471859
3416	409,639	10,2732479572
3417	518,383	11,1047837734
3444	455,369	11,1362421513
3469	454,145	10,2163584232
3484	608,873	12,1567780972
3746	494,936	10,8390653133
3923	436,937	8,4241147041
3943	439,481	11,2028839588
4099	661,526	14,9066317081
4718	816,934	14,9269952774
4956	768,749	15,1636908054
5550	823,367	14,0813224316
5824	595,857	18,7359137535

Tabelle 3.3.: Berechnung der Turn-by-Turn Daten für zufällige Koordinatenpaare in Deutschland

Tabelle 3.3 zeigt die Gesamtberechnungszeit für die Turn-by-Turn Daten zufälliger Routen in Deutschland. Die Berechnungszeit skaliert wie erwartet ungefähr linear mit der Anzahl der Koordinaten pro Route. Die Anfragen an die Datenbank dominieren deutlich alle anderen Berechnungen, hier besteht also die größte Verbesserungsmöglichkeit im Bereich der Performance. Diese Berechnung ist auf dem unten genannten System hautpsächlich durch die Geschwindigkeit des Prozessors begrenzt.

Alle Benchmarks in diesem Abschnitt wurden auf einem Notebook mit einer Intel i7 3632qm Quadcore CPU mit einer Taktfrequenz von 3.1 GHz und 32 Gigabyte Arbeitsspeicher mit einem Takt von 1600MHz durchgeführt. Die eingesetzte PostgreSQL Version ist 9.2.2.

4. Implementierung

4.1. Übersicht

Im Folgenden wird die Implementierung des Systems beschrieben. In der vorgestellten Lösung ist im Server vergleichsweise wenig Funktionalität implementiert, während im Android-Client die Hauptarbeit für die Turn-by-Turn Navigation erledigt wird. Eine Lösung, in der die Hauptfunktionalität im Server implementiert ist, wäre aber auch möglich.

Der Normalablauf der Turn-by-Turn Navigation sieht folgendermaßen aus:

1. Der Client sendet eine Liste mit Koordinaten an den Turn-by-Turn Server.
2. Der Turn-by-Turn Server verarbeitet die Liste mit den Koordinaten.
   a) Der Turn-by-Turn Server ermittelt mit Hilfe der OpenStreetmap Daten, zu welcher Straße jede einzelne Koordinate gehört.
   b) Der Turn-by-Turn Server erstellt eine neue Liste der Koordinaten, die nach Straßen strukturiert ist.
3. Der Turn-by-Turn Server sendet das Ergebnis an den Client.
4. Der Client führt die Turn-by-Turn Navigation aus.
   a) Der Client wartet auf ein Positionsupdate vom Android System.
   b) Der Client berechnet die Position bzw. den Fortschritt auf der gewünschten Route.
   c) Der Client ermittelt anhand der Position die Entfernung zu sich eventuell ändernden Straßenbezeichnungen, die für eine Meldung an den Benutzer interessant sind.
   d) Der Client gibt Meldungen über sich eventuell ändernde Straßenbezeichnungen mittels Sprach- und/oder Textausgabe aus, wenn sie sich in einer Entfernung von weniger als 800 Metern in Fahrtrichtung befinden.
5. Ist das Ziel erreicht, beendet der Client die Turn-by-Turn Navigation.

Listing 4.1 gekürztes Beispiel für den Aufbau einer Anfrage an den Turn-by-Turn Server

```
[
   [
      [
         484869841,
         81500874
      ],
      [
         484868803,
         81500028
      ],
      [
         484863980,
         81502720
      ],
      [
         484855924,
         81507245
      ]
   ],
   [
      [
         484851963,
         81509695
      ],
      [
         484837542,
         81521774
      ],
      [
         484832065,
         81525348
      ]
   ]
]
```

4.2. Datenbankzugriff und Protokoll

Das ToureNPlaner System verwendet JSON zum Datenaustausch, daher verwendet das vorgestellte System ebenfalls JSON zum Datenaustausch.

Der Turn-by-Turn Server benötigt nur die Liste mit Koordinaten. Diese werden per POST HTTP-Request als JSON Objekt oder Array gesendet.

In dem JSON Objekt befindet sich ein Array von Koordinaten-Arrays. Jedes dieser Koordinaten-Arrays beginnt bei einem Marker, den der Benutzer mit dem ToureNPlaner Client gesetzt hat und endet beim nächsten Marker.

Der Client sendet die Koordinaten als Integer, indem jeweils die echten Koordinaten mit 10^7 multipliziert werden und die Nachkommastellen abgeschnitten werden.

Der Server sollte folgende Daten zurückliefern, um eine Turn-by-Turn Navigation zu erlauben:

- Eine Liste von Straßenabschnitten, wobei jeder Straßenabschnitt mit dem Straßennamen annotiert ist
- Eine Liste der Koordinaten, die einen Straßenwechsel notwendig machen bzw. andere interessante Punkte wie Kreisverkehre, etc.
- eventuell interessante Tags der Straßen

Die Turn-by-Turn Daten werden in einem Objekt namens "streets" gesendet.

Da die Turn-by-Turn Route eine Abfolge von Straßen ist, enthält "streets" ein Array mit Straßen.

Jeder Eintrag im Array ist ein Objekt, das aus Straßenname und einem Array aus Koordinaten-Informationen besteht.

Die Koordinaten-Informationen sind: Latitude, Longitude sowie die Abweichung der jeweiligen Koordinate aus der Datenbank von der Koordinate, die der ToureNPlaner Server gesendet hat.

Außerdem enthält das "streets" Objekt noch ein Array "failed", in dem die Koordinaten stehen, die der ToureNPlaner gesendet hat, und die der Turn-by-Turn Server nicht in seiner Datenbank gefunden hat.

4.3. Implementierung des Servers

Der vorgestellte Turn-by-Turn Server ist in Python 3 mit dem "bottle" [bottle] Framework implementiert.

4.3.1. Queries zur Datenbank

Die Strukturierung nach Teilwegen entlang der vom Benutzer gesetzten Marker ist für die Turn-by-Turn Berechnung irrelevant. Der Turn-by-Turn Server iteriert ein Mal über alle Koordinaten in allen Listen und Sublisten. Für jede dieser Koordinaten erzeugt der Turn-by-Turn Server ein SQL Query, das eine bestimmte Anzahl von nahesten OSM-Knoten von der gegebenen Koordinate zurückliefert.

Das Berechnen des nahesten Knotens ist nicht ausreichend, da in den OpenStreetmap Daten viele Knoten aufeinander oder sehr nahe beieinander liegen können.

Diese vielen Queries können zu einem großen Query konkateniert werden, indem sie mit "UNION" verbunden werden.

Das übergebene Literal "index" wird dann dazu verwendet, aus dem Ergebnis die ursprüngliche Reihenfolge der Koordinaten wiederherzustellen, da UNION (zumindest bei der

Listing 4.2 Beispiel für den Aufbau einer Antwort des Turn-by-Turn Servers

```
{
   "streets" : [
      {
         "coordinates" : [
            {
               "ln" : 9.1947089,
               "lt" : 48.7994631,
               "deviation" : 0.0111205851517824
            },
            {
               "ln" : 9.194664,
               "lt" : 48.7996483,
               "deviation" : 0
            },
            {
               "ln" : 9.1946046,
               "lt" : 48.7998347,
               "deviation" : 0
            },
            {
               "ln" : 9.1937786,
               "lt" : 48.8010088,
               "deviation" : 0
            }
         ],
         "name" : "Rosensteinstrasse"
      },
      {
         "coordinates" : [
            {
               "ln" : 9.1562795,
               "lt" : 48.8173162,
               "deviation" : 0
            },
            {
               "ln" : 9.1557504,
               "lt" : 48.8169861,
               "deviation" : 0
            }
         ],
         "name" : "Josef-Waibel-Weg"
      }
   ],
   "failed" : [
      []
   ]
}
```

Listing 4.3 SQL-Query, das die nahesten OSM-Knoten zu einem gegebenen Punkt findet; Werte in eckigen Klammern sind vom Benutzer einzusetzen

```
(SELECT [index], highway_nodes.id, ST_Y(highway_nodes.geom), ST_X(highway_nodes.geom),
    highway_ways.id, highway_ways.tags -> 'name', highway_ways.tags -> 'ref',
    highway_ways.nodes, highway_way_nodes.sequence_id, highway_ways.tags,
    ST_Distance('POINT(*lon* *lat*)',highway_nodes.geom::geography)
FROM highway_nodes JOIN highway_way_nodes ON highway_nodes.id = highway_way_nodes.node_id
    JOIN highway_ways ON highway_way_nodes.way_id = highway_ways.id
ORDER BY highway_nodes.geom <#> ST_GeomFromEWKT('SRID=4326;POINT([lon] [lat])')
LIMIT [NN_NUM])
```

Implementierung von PostgreSQL) keine Garantie über die Reihenfolge der konkatenierten Ergebnisse abgibt.

Da die einzelnen Queries unabhängig voneinander sind und keine Schreibvorgänge vorgenommen werden, sind sie sehr einfach parallel ausführbar. PostgreSQL nimmt dies leider nicht automatisch vor.

Die vorgestellte Implementierung verwendet mehrere Datenbankverbindungen und bearbeitet pro Datenbankverbindung einen Teil des Queries und konkateniert jeweils die Ergebnisse.

4.3.2. Verarbeitung der Ergebnisse der Queries

Die Antwort des Turn-by-Turn Servers soll nach Sequenzen von Straßennamen strukturiert sein. Dabei kann ein Straßenname auch mehrfach vorkommen, sowohl als ein- und dieselbe Straße, als auch eine andere Straße, die zufällig den selben Namen trägt (bespielsweise existiert eine "Hauptstraße" sicherlich in vielen Städten).

Der Turn-by-Turn Server bearbeitet das Ergebnis des Datenbankqueries also sequenziell nach der Reihenfolge, die durch die Variable [index] gegeben ist. Dabei wird eine Liste von Straßennamen erstellt, wobei jedem Eintrag eine Liste von entsprechenden Koordinaten zugeordnet ist, gemäß dem Beispiel 4.1.

Umgang mit Inkonsistenzen

Grundlegende Abwägungen zum Vorgehen Beim Umgang mit OpenStreetmap Daten sind mehrere Dinge zu beachten:

- Die OpenStreetmap Daten sind über mehrere Versionen hinweg nicht stabil.
- Die OpenStreetmap Daten sind eventuell fehlerhaft und unvollständig (z.B. fehlende Straßenstücke, fehlende Straßennamen, fehlerhafte Tags).
- Die OpenStreetmap Daten sind eventuell nicht "wohlgeformt" (z.B. Duplikate in den Straßen, die verschieden markiert sind).

Zusätzlich muss bedacht werden, dass für die Generierung des ToureNPlaner - Graphs einig Tricks angewandt werden, um die Daten aufzubereiten, die in einem Datensatz resultieren können, der sich von den originalen OpenStreetmap Daten unterscheidet.

Das vorgestellte System soll möglichst unabhängig vom ToureNPlaner System funktionieren. Daher wird die Aufbereitung des Graphs für das ToureNPlaner System nicht dupliziert. Stattdessen wird versucht, eine Lösung zu finden, die möglichst gut zur Anfrage passt, aber trotzdem nur auf den originalen OpenStreetmap Daten arbeitet.

Eine Grundannahme des Turn-by-Turn Servers ist, dass die anfragenden Systeme besser aufbereitete Daten verwenden, als die unveränderten OpenStreetmap Daten. Wenn möglich, soll also versucht werden, die Koordinaten der Anfrage zu erhalten, auch wenn das bedeutet, dass hin und wieder Informationen aus den OpenStreetmap Daten nur teilweise beachtet oder gar ignoriert werden.

Zu einer gegebenen Koordinate wird in der Datenbank kein Knoten in unmittelbarer Nähe gefunden Gemäß der beschriebenen Annahme wird versucht, die vom Client erhaltenen Koordinaten zu erhalten. Die Koordinaten werden "in Fahrtrichtung" der Reihe nach betrachtet, daher kann als Behelf, wenn es nicht die erste betrachtete Koordinate ist, auf den Straßennamen der zuletzt betrachteten Koordinate zurückgegriffen werden. Dies verbessert zum Beispiel Kurven, in denen das ToureNPlaner System mehr Zwischenkoordinaten benutzt als in den OpenStreetmap Daten verfügbar sind.

4.4. Implementierung im Client

In diesem Abschnitt wird die Implementierung der diversen Funktionen beschrieben, die von einem Turn-by-Turn-Navigationsgerät erwartet werden.

4.4.1. Der Weg vom aktuellen Standort zur Route

Je nach Anwendungsbereich sind unterschiedliche Methoden sinnvoll, den aktuellen Standort in die Route zu integrieren. Bei einer TSP Route hat der Benutzer möglicherweise einen Anfahrtsweg, der nicht in das Problem mit aufgenommen werden soll. Bei einem Kürzester-Weg Problem ist es wahrscheinlicher, dass der Benutzer die Orte auswählt, die durchfahren werden sollen. Im vorgestellten System ist nur die Möglichkeit implementiert, dass der aktuelle Standort als erster Knoten zur Liste der Knoten hinzugefügt wird, so dass der Benutzer auf dem kürzesten Weg zum ersten Knoten seiner gewählten Route geleitet wird.

4.4.2. Position der aktuellen GPS-Koordinaten auf der Straße

Um eine sinnvolle Navigation grundsätzlich zu ermöglichen, muss das Navigationsgerät die aktuelle Position des Benutzers kennen. Ein GPS-Empfänger wird für das vorgestellte System als gegeben und aktiv vorausgesetzt.

Umgang mit Ungenauigkeit Da ein GPS-Empfänger unpräzise ist, muss immer mit Näherungen gearbeitet werden. Location Updates mit sehr großer Ungenauigkeit werden im vorgestellten System komplett ignoriert, da sie für eine Turn-by-Turn Navigation ungeeignet sind.

Außerdem werden Location Updates ignoriert, die den Ungenauigkeitsradius um mehr als 20 Meter vergrößern und dabei eine Positionsänderung von weniger als 20 Metern aufweisen.

Entfernung von und Position auf der Route Die simpelste Methode, die aktuelle Position und Entfernung von der Route zu ermitteln, ist das Finden der nahesten Koordinate, die zu der vom Turn-by-Turn Server gelieferten Route gehört. Dies kann auf langen, geraden Straßen problematisch sein, da dort die Wegpunkte spärlich verteilt sind. Hier können Ungenauigkeiten von mehreren Dutzenden Metern auftreten.

4.4.3. Ansage der nächsten Straße

Eines der wichtigsten Features der Turn-by-Turn Navigation besteht darin, dem Benutzer rechtzeitig mitzuteilen, wann der nächste Straßenwechsel vorgenommen werden muss.

Das vorgestellte System meldet bei jedem Location Update die Entfernung zum nächsten Straßenwechsel, sofern diese kleiner 800 Meter ist. Außerdem erfolgt die Meldung nur, wenn seit der letzten Meldung eine kleine Zeitspanne verstrichen ist.

Kreuzungen beispielsweise interessiert den Benutzer nicht nur der Name der nächsten Straße, auf die er wechseln muss, sondern auch, in welche Richtung er abbiegen muss. Es gibt mehrere Möglichkeiten, diese Richtungsänderung zu bestimmen. Eine Möglichkeit wäre die Bildung des Kreuzproduktes dreier Koordinaten. Das vorgestellte System berechnet stattdessen den Winkel zwischen dem letzten Wegabschnitt auf der Straße, auf der der Benutzer sich gerade befindet, und dem ersten Wegabschnitt auf der nächsten Straße in Grad.

Der Winkel relativ zu Nord lässt sich mit dem Algorithmus 4.1 berechnen:

Die Richtung des Abbiegens kann dann mit zwei solchen Winkeln mit der Methode 4.2 bestimmt werden:

Liegt der Winkel “helper” hier zwischen 0 und 180 Grad, liegt eine Linkskurve vor; liegt er zwischen 180 und 360 eine Rechtskurve.

Algorithmus 4.1 Berechnung des Winkels relativ zu Nord anhand zweier Koordinaten

```
latitude1 = source_latitude * (Math.PI / 180)
longitude1 = source_longitude * (Math.PI / 180)
latitude2 = target_latitude * (Math.PI / 180)
longitude2 = target_longitude * (Math.PI / 180)
deltaLongitude = longitude2 - longitude1
y = sin(deltaLongitude) * Math.cos(latitude2)
x = cos(latitude1) * sin(latitude2) - sin(latitude1) * cos(latitude2) * cos(deltaLongitude)
bearing = atan(y, x)
angle_current = (bearing * (180 / Math.PI) + 360 ) % 360
```

Algorithmus 4.2 Hilfsberechnung zur Bestimmung der Richtung eines Winkels

```
helper = (last_direction_current_street - first_direction_next_street + 360) % 360;
```

Da es auch noch die zwischen diesen Winkeln liegenden Möglichkeiten des Wendens und des Geradeausfahrens gibt, werden die Winkel für die Kurven auf die Bereiche [10,170] Grad und [190,350] Grad beschränkt.

Ein Winkel zwischen 170 Grad und 190 Grad bedeutet, dass der Übergang nahezu geradlinig verläuft, ein Winkel kleiner 10 Grad oder größer 350 Grad bedeutet, dass eine Wendung vorliegt.

Problem: leichte, lange Kurven Oft entspricht der Winkel zwischen dem letzten Straßenstück der einen Straße und dem ersten Straßenstück der nächsten Straße nicht der generellen Richtung der Straße vor bzw. nach der Kurve, da die Kurven über eine Entfernung von einigen Metern mit mehreren kleinen geraden Stücken mit immer stumpfer werdendem Winkel abgebildet werden. Daher sollte der Straßenverlauf für jeweils einige Meter vor und hinter dem Straßenübergang beachtet werden. Bei Betrachtung realer Straßen ist hier eine Entfernung von bis zu rund 25 Metern sinnvoll. Hier bietet sich die einfache Methode an, jeweils die Änderung der Winkel auf den letzten bzw. ersten Winkel aufzuaddieren, um die Gesamtrichtungsänderung zu approximieren.

Dies wird mit dem Algorithmus 4.3 erreicht.

Da mit Winkeln relativ zu Nord gerechnet wird und die Winkel zum Bestimmen der Richtungsänderung in den Bereich [0,360] Grad geschoben werden, kann die Änderung des Winkels addiert werden, ohne auf negative Werte Rücksicht nehmen zu müssen.

Anmerkung: Fehlende Straßennamen Für manche Straßenabschnitte sind in den OpenStreetmap Daten keine Straßennamen verfügbar. Dies betrifft zum Beispiel kurze Auffahrten, die auch in der Realität keinen Namen haben. Das vorgestellte System setzt in diesem Fall den entsprechenden Straßennamen auf den String “eine unbekannte Straße”.

Algorithmus 4.3 Berechnung der Gesamtänderung der Richtung in einer Kurve

```
initialisiere angle_current mit der Richtung der letzten beiden Koordinaten auf der
    aktuellen Stra\ss{}e
lastdirection = angle_current
currentdelta = 0
tempdist = 0
initialisiere temp_coordinate mit der vorletzten Koordinate auf der aktuellen Stra\ss{}e
do {
      tempdist += distance(temp_coordinate, temp_coordinate.previous)
      tempdirection = direction (temp_coordinate.previous, temp_coordinate)
      currentdelta += tempdirection - lastdirection;
      lastdirection = tempdirection;
      temp_coordinate = temp_coordinate.previous
} while (tempdist < 25 Meter && (temp_coordinate != first(temp_coordinate)));
angle_current +=currentdelta;
```

4.4.4. Sprachausgabe

Die Grundfunktionalität eines Navigationsgerätes kann mit dem Zusammensetzen von relativ wenigen Strings implementiert werden, die für die jeweilige Ausgabe passend konkateniert werden.

Für die Sprachausgabe wird die Android Text-to-Speech API verwendet. Einem TextToSpeech Objekt der Android API können Strings übergeben werden, die das Android System automatisch in gesprochene Sprache konvertiert. Für dieses TextToSpeech Objekt kann auch die verwendete Sprache direkt gesetzt werden, sofern das entsprechende Sprachpaket installiert ist.

Das vorgestellte System sagt die folgenden Ereignisse an:

- In welche Richtung der Benutzer beim nächsten Straßenwechsel fahren soll:
  - links (“In 169 Metern nach links abbiegen auf Richterplatz”)
  - rechts
  - geradeaus (“In 143 Metern gerade aus fahren auf Neue Weinsteige”)
  - wenden (z.B. U-Kurve)
- Es weist den Benutzer darauf hin, wenn er sich auf der Straße befindet, auf der sein Ziel liegt (“Das Ziel liegt auf dieser Straße in 423 Metern”).
- Es weist den Benutzer darauf hin, wenn er sich innerhalb von 20 Metern vor seinem Ziel befindet (”Sie erreichen Ihr Ziel in 15 Metern“).

5. Zusammenfassung und Ausblick

5.1. Zusammenfassung

Das Ziel der vorliegenden Bachelorarbeit war die Implementierung der Turn-by-Turn Navigation in das ToureNPlaner Projekt. Zu diesem Zweck wurde eine neue Serverapplikation implementiert und der Android-Client aus dem ToureNPlaner Projekt erweitert.

Der Source Code des Turn-by-Turn Servers befindet sich auf github. [server]

Im Gegensatz zum ToureNPlaner System wurde einer der Kernteile des Systems - die Suche nach dem nächsten OSM-Nachbarknoten zu gegebenen Koordinaten - nicht selbst implementiert. Stattdessen wurde eine PostgreSQL/PostGIS Datenbank verwendet. Dabei hat sich herausgestellt, dass die Performance nicht den Erwartungen entsprechen konnte. Je nach Anzahl der Koordinaten der Route dauert das Berechnen der nächsten Nachbarn zu allen gegebenen Koordinaten einer Route zwischen wenigen Sekunden (bis ca. 4000 Koordinaten bzw. ca. 500 Kilometern unter 10 Sekunden), kann aber ab ca. 2500 Kilometern rund eine Minute dauern. Dies kann durch schnellere bzw. mehr Prozessoren ausgeglichen werden, besser wäre aber eine Beschleunigung der PostgreSQL Datenbank, entweder durch eine Verbesserung der PostgreSQL Datenbank selbst oder eventuell durch das Optimieren der Konfiguration der PostgreSQL Datenbank.

Obwohl das System an Performanceproblemen leidet, ist es doch für die meisten praktischen Einsätze geeignet. Für die seltenen sehr langen Routen kann vorerst auch eine etwas längere Wartezeit in Kauf genommen werden.

5.2. Ausblick

5.2.1. Verbesserungsmöglichkeiten

Abweichungen von der Route während der Fahrt Unter Umständen folgt der Benutzer nicht immer der berechneten Route. Dies kann viele Gründe haben: Ein leerer Tank bedingt einen Umweg zu einer Tankstelle, eine Straße ist gesperrt und es müssen Umwege gefahren werden, etc. Ein Navigationsgerät sollte den Benutzer unterstützen, so dass der Benutzer zu jeder Zeit die Route zu den ausgewählten Markern wieder zurückfindet.

Dies kann umgesetzt werden, indem ein neuer Shortest-Path-Request ausgeführt wird, sobald sich der Benutzer zu weit von der Route entfernt hat. Dabei muss die Genauigkeit

berücksichtigt werden, da Android auch Location mit mehreren hundert Metern Ungenauigkeit erzeugt. Zusätzlich sollten schon besuchte Marker ausgeschlossen oder entfernt werden.

Genauigkeit der Entfernung von und Position auf der Route erhöhen Die im vorgestellten System implementierte Methode für die Bestimmung des Fortschritts auf der Route ist relativ ungenau. Eine etwas genauere und komplexere Möglichkeit ist das Betrachten von jeweils zwei aufeinanderfolgenden Koordinaten. Damit kann der kleinste Abstand zwischen deren Verbindungslinie und den aktuellen Koordinaten ausgerechnet werden. Diese Abstandslinie schneidet die Verbindungslinie an einem Punkt, der besser geeignet ist, die aktuelle Position auf der Route zu abzuschätzen.

Die Schätzung kann noch weiter verbessert werden, wenn die Genauigkeit der aktuellen Koordinaten beachtet wird. Diese ist durch einen Kreis um die aktuellen Koordinaten gegeben, der (bei Android) mit 68%-iger Wahrscheinlichkeit angibt, dass die tatsächlichen Koordinaten innerhalb dieses Kreises liegen. Die Genauigkeit der Koordinaten sollte als Gewichtungsfaktor für eine Heuristik eingesetzt werden.

Merken der schon besuchten Straßenabschnitte Speziell bei Routen, die vom ToureNPlaner System erzeugt werden, kann es vorkommen, dass der Benutzer auf einer Straße erst in einer Richtung zu einem Marker fährt und dann auf der selben Straße wendet und zurück fährt.

In diesem Fall reicht es nicht aus, nur den nahesten Routenabschnitt zu suchen, da auch noch der "Hinweg" vom "Rückweg" unterschieden werden muss. Eine Lösung für dieses Problem ist das "Markieren" aller schon besuchten Koordinaten. Diese Lösung adressiert jedoch noch nicht das Problem, dass das GPS aus unterschiedlichen Gründen für eine unvorhersehbare Zeit keine brauchbaren Daten liefert. Auch hier sollte also eine Heuristik eingesetzt werden, die auch weitere Faktoren wie die aktuelle Fahrtrichtung, die aus den letzten beiden besuchten Koordinaten berechnet wird, beachten kann.

Einbeziehung von interessanten Punkten für die Ansage Die Einbeziehung von naheliegenden interessanten Punkten ("Points of Interest") ist eine intuitive Methode, dem Benutzer das Zurechtfinden weiter zu erleichtern. In Frage kommen dafür sowohl einfache Eigenschaften der Straße, wie zum Beispiel Bahnübergänge oder Brücken, die eventuell in den Tags eines OSM Knotens gespeichert sind, als auch naheliegende, gut sichtbare Punkte wie Kirchen, Sehenswürdigkeiten, Parks, etc.

Grammatikalisch möglichst korrekte Sprachausgabe Bei der Vielzahl an unterschiedlichen Straßennamen gibt es viele Feinheiten zu beachten. Zum Beispiel verhindert das Vorkommen von auf "-Weg" endenden Straßennamen sowie das Vorkommen von auf "-Straße" endenden Straßennamen, dass der männliche oder weibliche Artikel verwendet wird, jedenfalls ohne den Kontext (den Straßennamen) zu beachten. Andere Sonderfälle wie "Arnulf-Klett-Platz"

oder “In den Riedwiesen” verkomplizieren eine grammatikalisch korrekte Richtungsangabe. Eine Lösung hierfür muss also ein relativ umfangreiches Mapping von Straßennamen und jeweils deren Bestandteilen auf grammatikalische Richtigkeit enthalten.

A. Anhang

A.1. Einrichten der PostgreSQL Datenbank

Vorausgesetzt werden:

- ein aktuelles (Dezember 2012) Archlinux System
- OpenStreetmap Daten für das entsprechende Land. Die Daten für diese Arbeit wurden von geofabrik.de [geofabrik] bezogen

A.1.1. Tuning der Datenbank für große Datenmengen

PostgreSQL ist nicht für die Arbeit mit großen Datenmengen wie den OpenStreetmap Daten voreingestellt. Mit den Änderungen in A.1 oder ähnlichen Änderungen in `/var/lib/postgres/data/postgresql.conf` kann aber die Performance gesteigert werden:

Listing A.1 PostgreSQL Einstellungen für erhöhte Performance bei großen Datenmengen

```
max_connections = 100
shared_buffers = 512MB
max_connections = 30
work_mem = 64MB
maintenance_work_mem = 1GB
synchronous_commit = off
checkpoint_segments = 32
checkpoint_timeout = 15min
checkpoint_completion_target = 0.9
default_statistics_target = 1000
autovacuum = off
fsync = off
```

Eventuell erlaubt die Konfiguration des Systems keine hohen Werte für shared_buffers. In diesem Fall muss der Kernel Wert für *shmmax* in `/etc/sysctl.conf` wie in A.2 angegeben erhöht werden:

Listing A.2 sysctl.conf Einstellungen für große Buffer

```
# 8 GB Maximum size of shared memory segment (bytes)
kernel.shmmax=8589934592
#Total amount of shared memory available (bytes or pages)
kernel.shmall=4194304
```

A.1.2. Import der OpenStreetmap Daten in die Datenbank

Für den Import von OpenStreetmap Daten in eine Datenbank sind einige Scripte und Programme verfügbar.

- *osm2pgsql* erzeugt eine brauchbare Datenbank. Es ist aber nicht klar, ob die OSM-Daten wie IDs von Knoten und Kanten korrekt erhalten bleiben, da die Standardeinstellung hauptsächlich zum Rendern von "Tiles" verwendet wird.
- *osm2postgresql* ist ein "all-in-one" Script, das eine Komplettlösung einrichtet, inklusive einer neuen Installation von PostgreSQL.
- *osm2pgrouting* ist ein Programm, das die Daten für einen Routenplaner auf Postgis-Basis importiert.
- *osmosis* ist ein umfassendes Tool zum Arbeiten mit OSM-Daten und bietet unter anderem die Funktionalität, die OSM XML oder PBF Daten direkt in eine PostGIS Datenbank zu schreiben.

Osmosis ist gut gewartet und wird aktiv weiterentwickelt. Im Test hat es eine gut nutzbare Datenbank erstellt, die diverse Features der PostgreSQL Datenbank wie "hstore" für die Tags benutzt.

Nachdem die angegeben Voraussetzungen erfüllt sind, werden die Daten mit der Methode A.3 importiert, indem die Osmosis Anleitung [postgissetup] angepasst und erweitert wird.

Listing A.3 Import der OSM Daten in die Datenbank

```
# das Standard-Template soll im utf8 Zeichensatz sein
sudo -u postgres psql
UPDATE pg_database SET datistemplate = FALSE WHERE datname = 'template1';
DROP DATABASE template1;
CREATE DATABASE template1 WITH TEMPLATE = template0 ENCODING = 'UNICODE';
UPDATE pg_database SET datistemplate = TRUE WHERE datname = 'template1';
exit

createuser osm
createdb -E UTF8 -O osm gis
createlang plpgsql gis

# postgis einrichten
psql gis < /usr/share/postgresql/contrib/postgis-2.0/postgis.sql
psql gis < /usr/share/postgresql/contrib/postgis-2.0/spatial_ref_sys.sql

# zusaetzliche Projektionen in spatial_ref_sys einfuegen:
# http://wiki.openstreetmap.org/wiki/Mapnik#Invalid_projection_in_pgSQL
wget http://svn.openstreetmap.org/applications/utils/export/osm2pgsql/900913.sql
sudo -u postgres psql -f 900913.sql -d gis

# wenn hstore fuer die Tags verwendet werden soll: pgsnapshot_schema
psql gis -c "CREATE EXTENSION hstore;"
psql gis -f /usr/share/osmosis/script/pgsnapshot_schema_0.6.sql
# sonst arrays f\"ur Tags: pgsimple_schema
# psql -d gis -f /usr/share/osmosis/script/pgsimple_schema_0.6.sql

# schnelles dumpen in Dateien, die direkt in die Datenbank "kopiert" werden koennen:
JAVACMD_OPTIONS="-Djava.io.tmpdir=/var/tmp" osmosis --read-pbf file="baden-wuerttemberg.osm.pbf"
     --buffer --write-pgsql-dump directory=pgsqldump

cd pgsqldump
# verwende die editierte pgsnapshot_load_0.6.sql, die dem Server beiliegt
psql -U osm -d gis -f ./pgsnapshot_load_0.6.sql
```

Literaturverzeichnis

[bottle] Bottle web-framework. http://bottlepy.org/docs/dev/. (Zitiert auf Seite 27)

[fakegps] App "fakegps" im google play store. https://play.google.com/store/apps/details?id=com.lexa.fakegps. (Zitiert auf Seite 16)

[geofabrik] Osm ausschnitt download von geofabrik.de. http://download.geofabrik.de/. (Zitiert auf Seite 39)

[postgis] Postgis erweiterung für postgresql. http://postgis.refractions.net/. (Zitiert auf Seite 17)

[postgissetup] Postgis setup anleitung. http://wiki.openstreetmap.org/wiki/Osmosis/PostGIS_Setup. (Zitiert auf Seite 40)

[server] Turn-by-turn server. https://github.com/ToureNPlaner/TurnByTurn. (Zitiert auf Seite 35)

Alle URLs wurden zuletzt am 12. 12. 2012 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Christoph Haag)