Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

MASTEROPPGAVE

Studieprogram/spesialisering: Informasjonsteknologi/Datateknikk	Vårsemesteret, 2009 Åpen
Forfatter: Ragnar Stølsmark	…… (signatur forfatter)
Faglig ansvarlig Erlend Tøssebro Veileder(e): Erlend Tøssebro	
Tittel på masteroppgaven: Innsamling og bruk av usikre data i et mobilt GIS Engelsk tittel: Collection and use of uncertain data in a mobile GIS	
Studiepoeng: 30	
Emneord: GIS Energy conservation Cell phone GPS Accelerometer	Sidetall: 73 + vedlegg/annet: 6 Stavanger, …….. dato/år

Collection and use of uncertain data in a mobile GIS

Ragnar Stølsmark

University of Stavanger

June 15, 2009

Abstract

This study looked into the possibility of saving power in mobile geographical information systems by extending the interval between GPS updates. A J2ME application has been developed to test this theory. Via a map it can guide the user to his destination. The program displays the current uncertainty of the user's position. This uncertainty spreads out until the next GPS update is received. To make the long update intervals transparent to the user, a dead reckoning scheme was included. The application tries to guess where the user is, based on different input like the previous position or data from the phone's accelerometer. Both the spreading of the uncertainty and the dead reckoning has different implementations for driving and walking. The tests of the application showed that the phone battery lasted approximately 4 times longer when going from 2 seconds GPS update intervals, to only updating when the uncertainty exceeded 500 meters in diameter. The results lead to the conclusion that extending the update intervals is a great way to achieve longer battery life for mobile applications which rely on a GPS receiver.

Acknowledgements

I would like to thank my supervisor, Erlend Tøssebro, for coming up with this interesting assignment. He also helped me with some of the difficult problems that I faced during my study.

Contents

1 Introduction 8
1.1 Outline 8
2 Background 10
2.1 Global Positioning System 10
2.2 Java Micro Edition 12
3 Choice of development phone 15
3.1 Apple iPhone 3g 16
3.2 Nokia N96 17
3.3 Sony Ericsson W760i 17
4 Overview of the Travel Application 19
4.1 Packages 20
4.2 Threads 21
5 The map 22
5.1 Raster maps 22
5.2 Vector maps 23
5.3 OpenStreetMap 24
5.3.1 OpenStreetMap ontology 24
5.3.2 OpenStreetMap XML file format and parsing 25
5.4 The Travel Application map 26
5.4.1 The ontology of the map 28
5.4.2 Drawing the map 28
6 Routing 33
6.1 Graphs and the shortest-path problem 33
6.1 Choice of shortest-path algorithm 34
6.2 Implementation of the A* algorithm 35
7 Uncertainty 37
7.1 The GPS uncertainty 37
7.2 Determining the uncertainty type 38
7.3 Walking uncertainty 41
7.4 Driving uncertainty 43
8 Dead reckoning 48
8.1 The OffroadNoRouteReckoning implementation 49
8.2 The OffroadRouteReckoning implementation 50
8.3 The RoadRouteReckoning implementation 51
8.4 The RoadNoRouteReckoning implementation 52
8.5 Intersection road choosing using the accelerometer 53
9 Tests 58
9.1 Test setup 58
9.2 Box plots 58
9.3 Testing the standby time 59
9.4 Testing the Travel Application 60
9.4.1 Measuring the effect of altering the uncertainty level 60
9.4.2 Comparing the two different routing implementations 61
9.5 GPS listener versus get location 63
9.6 Power consumption of accelerometer and GPS 65
9.7 Testing a commercial navigation application 67
10 Conclusion 70

10.1 Further work 70
Bibliography 72
Appendix 74
Appendix A Algorithms 74
Appendix A.1 The A* Algorithm 74
Appendix A.2 The update driving uncertainty algorithm 75
Appendix A.3 The reclassify algorithm 76
Appendix A.4 The calculate new position algorithm 78
Appendix A.5 The calculate position after an intersection algorithm 79

1 Introduction

In the last years it has become more common with mobile phones that support GPS. Using the GPS receiver consumes a lot of battery power, while most mobile phones still have a low battery capacity. Therefore conserving energy is an important issue when developing mobile GPS applications.

This thesis looks at the possibilities of saving power by allowing more uncertainty in a mobile GIS. GPS inherently has some uncertainty regarding the current position of the mobile device. This uncertainty will expand between GPS updates. The longer an application allows this expansion to continue, the less frequent it needs to receive GPS updates.

To test the power saving theory, a travel application for mobile phones was developed. The Travel Application is developed for Java Micro Edition. It can guide the user to a destination of his choosing. A major part of this report is dedicated to the design of the Travel Application. The application use dead reckoning to make longer update intervals acceptable to the user. The current uncertainty is always displayed on the map. The application also lets the user change how much uncertainty he tolerates.

The study has involved experiments with how different properties affect cell phone battery life. The experiments included: Testing the effect of allowing more uncertainty in the Travel Application, using an accelerometer to extend the time needed until next GPS update and using different routing implementations.

1.1 Outline

Chapter 2 covers background information about the Global Positioning System (GPS) and Java Micro Edition (J2ME). The chapter describes how GPS works, especially the concept of trilateration. The differences between J2ME and the Java Standard Edition (J2SE) are also mentioned.

In chapter 3 the process of choosing the development phone is described. The chapter includes a list of requirements such a phone had to meet.

Chapter 4 gives a quick overview of the Travel Application. It mentions how the different features are used. The program's threads and packages are also described in this chapter.

The map is the topic of chapter 5. It covers the source of the map data, OpenStreetMap. It also includes the Travel Application's map ontology. In the conclusion of the chapter the process of drawing the map on the screen is described.

Chapter 6 covers the problem of finding the shortest route to a destination. The chapter describes the A* routing algorithm. There is also a section about the implementation of this algorithm in the Travel Application.

Uncertainty is an important topic in this thesis. Chapter 7 is dedicated to all aspects of the uncertainty related to the estimate of the current position. It first gives an overview of the factors contributing to the uncertainty of GPS. It then describes the two different uncertainty schemes that the Travel Application utilizes: walking and driving uncertainty.

Dead reckoning is covered in chapter 8. The chapter first describes how the Travel Application splits the dead reckoning into 4 different algorithms based on the current situation. The chapter also includes a description of how these 4 algorithms calculates the current position of the device. The topic of choosing which road to take in an intersection, based on accelerometer data, is covered in detail.

Chapter 9 contains the tests that were done concerning battery life. The test setup is described. In addition to showing the test results, the chapter also includes some discussion.

The last chapter gives a conclusion and suggestions for further work.

2 Background

Two technologies were essential to this project.

- The **Global Positioning System** was used to find the location of the cell phone.
- The Travel Application was developed for **Java Micro Edition**.

2.1 Global Positioning System

The Global Positioning System (GPS) is a satellite based positioning system. It was developed by the U.S. for military purposes. These purposes included missile guidance and navigation for desert troops. It opened for public use in 1993, but had been in development since the 1960s [1]. A GPS receiver can find its current location in the form of longitude, latitude and altitude. This can be achieved with a horizontal uncertainty of less than 10 meters, in high-end receivers. GPS revolutionized the market for mobile GIS, for instance car navigation systems. Today, GPS receivers are found in everything from missiles to cars and cell phones.

GPS consists of 24 operational satellites [2] in medium earth orbit, approximately 20 000 km above the earth's surface. This means that the satellites are not geostationary. Because of this the number of satellites within range of the same location on earth, changes with time. The 24 satellites are spaced out so that any place on earth should be able to calculate their position all of the time. But due to obstacles, buildings, mountains, etc. and bad weather conditions, combined with the low signal strength of GPS, this is not always possible. The areas around the equator generally have better signal conditions than those near the poles.

In each satellite there are atomic clocks that maintain accurate time. This is used in the GPS receivers to calculate the distance to the satellites.

Distance = (receive time – start time)/speed of light.

The GPS receiver also knows the current positions of all the satellites. To calculate its position, the GPS receiver first calculates the distance to multiple satellites. It combines these distances with the current position of the satellites. This process is called trilateration. 2D trilateration is illustrated in Fig. 2.1. GPS trilateration is similar to 2D trilateration, but since GPS is a 3-dimensional system, spheres needs to be applied instead of circles [3]. The intersection point still needs to be calculated. The distance to the first satellite will give a possible sphere surface for the location. Applying a second satellite will lead to a circle, unless the two spheres are just touching each other in which case it will lead to a point. The third satellite will give two intersection points, so a fourth sphere is needed. This fourth sphere is the earth's surface. However to calculate a position with decent accuracy, 4 satellites are needed. This is because the fourth satellite is used to correct the receiver's clock. Without this correction an extremely expensive clock would need to be put in every GPS receiver. This is not compatible with making inexpensive GPS units. Since there is some uncertainty related to each distance estimate (see chapter 7), many GPS receivers use more than the 4 required satellites for higher accuracy.

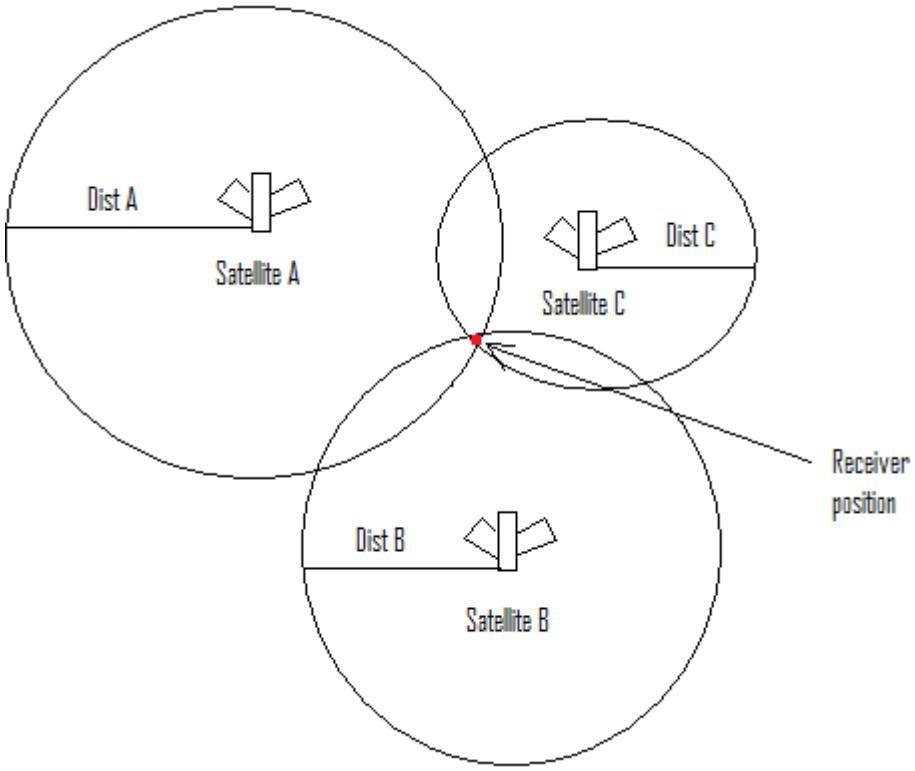

Fig. 2.1: 2D Trilateration. The receiver is located at the intersection point of the three circles.

To find the approximate location of all the satellites, every GPS satellite transmits the almanac. Since it is transmitted in intervals, the almanac takes 12:30 minutes to receive from one satellite. However the almanac data from multiple satellites can be combined for a faster reception. The almanac can be used in the receiver to avoid looking for satellites which is not visible on the horizon. The same almanac can be used for weeks, since it is just utilized to avoid wasting time on the satellites that are definitely out of reach [4]. This means that the accuracy demands are not very stringent. When the receiver has found a satellite and wants to calculate the distance to it, more accurate knowledge is required. This can be found in the ephemeris. The ephemeris is unique for each satellite. It contains highly accurate data about the satellite's orbit and position. It is transmitted once in each GPS frame. This means every 30 seconds. Every few hours it is updated to correct for minor variations in the satellite's orbit.

There are a couple of issues with GPS. Because the satellites are so far away, the signal is weak. This means it won't penetrate buildings, which renders it useless for indoor applications. It is also a case when in dense forests and other places with limited visibility.

GPS is mainly used in movable devices, since stationary ones tend to know their own location. Many of these devices run on batteries, for instance cell phones and standard portable GPS units. Battery life is a critical factor, however GPS tends to use a lot of battery power (see the test chapter for battery life tests of GPS). An application that can provide the current location with as few GPS updates as possible could be useful. This is the main focus of this thesis' Travel Application.

The first time the receiver is turned on, on a given day, is called a cold start. It then needs to download the ephemeris again. It may also need a new almanac. Downloading them could take multiple minutes, depending on timing and how many satellites the almanac can be received from. Even the 30 seconds (worst case) to get a new ephemeris has become an important factor in how quickly a receiver can get a GPS fix. The transmission speed of the

satellites is only 50 bits/s. With the introduction of GPS in cell phones, a new way to get these files has appeared. It is part of a scheme called Assisted GPS. Assisted GPS stands for the cellular network helping the GPS receiver getting a position. The current ephemeris and almanac can be downloaded, on demand, from the internet. Since the 3G network can have download speeds of approximately 1 Megabits per second, it is a much faster data source. 3G supports two-way communication, so there is no need to wait for the start of the transmission either.

Some uncertainty regarding the calculated position must be expected. This topic is covered thoroughly in the uncertainty chapter.

2.2 Java Micro Edition

Java Micro Edition (J2ME) is a version of the Java platform that is intended for small devices. It is mainly used in cell phones. Most of J2ME is a subset of the standard Java platform for PCs, J2SE. J2SE is in turn a subset of the Java enterprise edition, J2EE. This is depicted in figure 2.2.

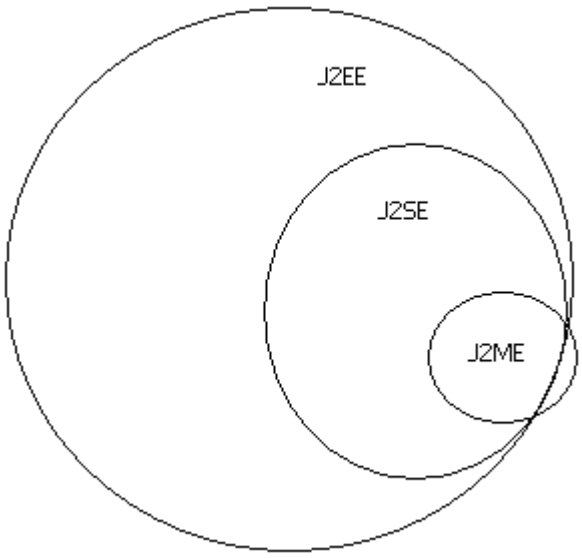

Fig. 2.2: The relationship between the different Java platforms.

J2ME is divided into two sets of APIs [5]. The most basic set is the Connected Limited Device Configuration (CLDC). It contains the basic java classes. The biggest and newest version, MIDP 2.1, also contains classes for making richer GUI's. For instance it has its own game package with support for sprites. It also guarantees support for floating point numbers. MIDP 2.1 is included in most new cell phones with Java capabilities. The other set of API's, the Connected Device Configuration (CDC), contains all the basic classes from CLDC, but not the GUI classes. The biggest difference from CLDC is that CDC has more classes, and it is meant for devices with an Internet Connection. It has packages dedicated to network communication and security. The latest CDC version, 1.1, is also implemented in all new Java phones. This is because it is hard to find a new phone that does not have either a GPRS or 3G Internet Connection.

The API's consists of many similar or identical classes to the ones in J2SE, although a few of the classes are unique to J2ME. A lot of the packages and classes have been shaved. For instance the Math class does not contain a method for finding $\sin^{-1}$. The GUI class has no

method for drawing polygons. And while you in J2SE version 6 have 54 classes in the java.util package, the corresponding number for J2ME CDC is 36. Some of the features that have been removed from J2SE are the support for enumerations and generics. The main reason for removing these features is that J2ME is intended for devices with very low memory capacity and slow processors. The memory constraints were especially stringent some years ago. Modern phones, like the Sony Ericsson W760i, have over 10 MB of available runtime memory. This will of course continue to increase, making the need for a small API decrease. That is why MIDP 3.0 is being developed. It will have more functionality, and try to make some of the optional J2ME packages standard.

J2ME contains lots of optional packages, so called Java Specification Requests (JSRs). They are optional since the devices and their capabilities are so different. One mobile phone might have a camera, while another has GPS and a powerful enough processor to handle 3D graphics. One can see the CLDC and CDC as the least common denominator, at least on new phones, and the JSRs as specialized APIs for handling common extra features. There is 19 JSRs for J2ME at the moment. In the creation and testing of the Travel Application, 3 of them were used. It was: JSR 75: File Connection and PIM, JSR 179: Location API and JSR 256: Mobile Sensor API.

J2ME has a standard database resembling storage system, called RMS. J2ME can also natively read files within the same JAR archive as the running application. However, newer phones tend to come with removable storage media, like memory cards. This makes it all the more important to have direct access to the files on these file systems. The file connection API makes this possible. Most phone implementations separate the phone memory from the memory card by using different drive letters. Otherwise accessing a file on a memory card is identical to accessing one that is stored in the phone's internal memory. Once a file connection has been opened, the java.io classes can be used to read and write data on the file. In this thesis JSR 75 was used to store J2ME emulator script files from the phone's GPS and accelerometer. The files were generated using a program that recorded real life data from these instruments. This program was developed as part of this study.

The Location API is mainly for mobile devices with GPS capabilities. It can however also take advantage of external, Bluetooth connected, GPS devices. If no GPS is present or uncertainty is of no greater concern, the API supports implementations that use cellular network trilateration or even simply the connected cell, to determine location. The API has a location provider class that gives the current location. It can also be set up to return the current location, at a specified interval, to a location listener. It uses the standard WGS-84 datum (longitude, latitude) for all location coordinates. When it returns the position it also calculates the current uncertainty associated with it. This uncertainty changes with every location update.

The Mobile Sensor API is concerned with the use of all kinds of sensors that can be attached to a mobile device. The most common are battery level, network signal and accelerometers. But the API can also be used for temperature, wind or any other sensor. It is set up very similar to the Location API. The difference is how you locate the correct sensor. In the Location API you set different criteria, for instance power consumption or maximum uncertainty, and the best location provider is chosen. In the Sensor API one locates the different sensors via strings. For instance to find any accelerometers, on a Sony Ericsson W760i, one would write:

```
SensorManager.findSensors("acceleration", SensorInfo.CONTEXT_TYPE_USER);
```

There are three different contexts, or classes, of sensors: user, device and ambient. User is intended for sensors that are connected to the user, for instance a heart rate sensor. Device sensors monitors the device, an example would be remaining battery power. Ambient sensors measure the environment, for example a thermometer. However the different

implementations of the Java Virtual Machine seem to disagree about some of the sensors. The emulator thinks an accelerometer is a device sensor, whereas Sony Ericsson W760i is certain it is a user sensor.

The API returns the data either on a get basis, or through a listener. The API also supports returning uncertainty, scale or units along with the data. This is not always implemented in the devices however, as is the case with the W760i.

3 Choice of development phone

Since this project is partly about the location uncertainty of mobile phones, a phone was needed to test the Travel Application. This was especially true since one of the most important features of the application was intended to be the ability to save battery power and still be able to navigate properly. There are emulators available for most cell phones, but these features are best tested in the real world. Because of the wide array of phones available, the following feature list over what was needed for the project, helped made the choice easier.

1. The phone must have a GPS module. This is absolutely vital, since the intention of the application is to issue a request for a GPS position when the uncertainty is too large.
2. A 3-axes accelerometer must be included in the phone. Between updates the application uses a dead reckoning scheme, assisted by the accelerometer, to determine the current position of the device. The accelerometer could for instance be used to decide if the user made a left- or right hand turn at an intersection.
3. The above features must be programmatically available to an application programmer. This also means that it must be possible to make your own applications and run them on the phone. This is standard in most modern mobile phones.
4. The phone must have a colour display to be able to have a good separation between the different items on the map. This is standard in all of the phones that support the 2 first criteria.
5. The phone should have some support for measuring the current power consumption. This is not an absolute must, since there is always the option of simply checking how long it takes before the phone is out of battery.
6. It must be possible to use the developer tools, like the emulator, on Microsoft Windows XP/Vista or Linux. This is because they are the only operating systems that are accessible on the University.
7. The price. All these features are wanted as cheap as possible.
8. J2ME is the preferred programming language.

Originally it was the intention to trilaterate the phone using the known position of 3 GSM base stations and measurements, like received signal strength (RSS), from these. This soon proved impossible, since the only information an application has access to, is the cell ID and the RSS of the cell the phone is connected to. This is because the network information is constrained within the GSM module of the phone, at least on the phones that were examined during the work with this thesis. Of course you could get an indication of the position using only the cell ID, but there would be such a great uncertainty related to this position that it would be useless in a routing application. To estimate the current position with 90% certainty one have to use a diameter of about 2200 meters when using only Cell ID in suburban areas [6].

With this list in mind, phones from different manufacturers were checked out, to find the one best suited for the project’s needs.

3.1 Apple iPhone 3g

Fig. 3.1: Apple iPhone 3G displaying a map [7].

The first phone examined was Apples new IPhone 3G [7]. This was investigated first because of its widespread popularity. It has a GPS-module, and provides assisted-GPS when there are poor signal conditions. This results in a quicker satellite fix, and can even locate the device when there is no GPS reception (Through WLAN or Cell ID). iPhone also includes an accelerometer. Both the GPS and the accelerometer are reachable for application programmers through the API.

iPhone applications must be written in the Objective-C programming language. It is a rarely used language, but that was not the deciding factor why iPhone was not chosen. The biggest problem was the fact that the software development kit (SDK) only worked on Mac OS, unless some serious tricks were performed [8]. Since spending several thousand NOK on a Mac was out of the question, iPhone 3G was dropped for this project. The price is also steep, 7488 NOK in February 2008 for an iPhone 3G 8 GB without a subscription plan [9].

3.2 Nokia N96

Fig. 3.2: The Nokia N96 [10].

The N96 is a smart phone that runs on the Symbian S60 OS [10]. It supports both GPS (also assisted) and it has an accelerometer. N96 has J2ME built-in, but it is also possible to use C++. It seems that for development on the N96, C++ is the way to go. This is mainly because of Nokia's somewhat poor java implementation. For instance you don't have access to the accelerometer data when using J2ME, since it doesn't implement the JSR-256 (sensor) API. The N96 includes WLAN support for Internet access. A positive feature with the N96 is the power consumption tools. You have tools like Performance Investigator and Nokia Energy Profiler to help determine the power consumption of a given application. The reason this phone was skipped, was mainly the poor Java support and the steep price: 4995 NOK [9].

3.3 Sony Ericsson W760i

Fig. 3.3: The Sony Ericsson W760i [11].

The W760i is a walkman (intended for music) mobile phone, with a sliding keyboard [11]. It has GPS, assisted GPS and features a 3-axes accelerometer. The phone has a 240*320 pixels colour display. W760i supports the J2ME programming language. It also features almost all possible extensions of J2ME, since it implements the JP 8.3 Sony Ericsson Java Implementation. It is a MIDP 2.0 device and some of the APIs that are supported include: file operations (JSR-75), Location and GPS (JSR-179), Accelerometer and sensors (JSR-256), in addition to support for 3D graphics and many other APIs.

W760i features a developer mode. With this turned on, it is possible to debug an application, running on the device, from Eclipse [12]. Of course the phone needs to be connected to the computer in order for this to work. The console output can also be viewed on the computer when the phone is in this mode. W760i supports service tests where one can, for instance, view the output from the accelerometer at any given time. This comes in handy

when determining how the accelerometer reacts during movement. There is also a profiler tool to help find out which methods in an application receives the most calls and should therefore be optimized for performance. Unfortunately there are no power consumption tools for developers, so for testing one will have to use the crude, which application has the longest battery life, method. The Java emulator is good, with easy integration in eclipse and functions for XML-based GPS scripts so that one can test location based applications with pre-recorded data. The same goes for the accelerometer sensor. This is an especially neat feature for GPS, since it is often the case that there is no GPS signal inside an office.

All of these features combined with the lowest price of 2390 NOK [9], caused this phone to be chosen as development phone.

4 Overview of the Travel Application

The Travel Application is a J2ME application for mobile phones. It is meant to be a simple program, a proof of concept to whether it is possible to build a GPS application with extra low power consumption. Its main goal is to guide a user, from where he is, to where he wants to be. This is done via routing on a map. When the user starts the program he is presented to the following menu:

Fig. 4.1: The Travel Application's main menu

Show map will display the map, with the current position of the user (only after the first GPS fix). If he has chosen a destination, the route to it will be displayed. The map screen is shown in figure 4.2.

From all subscreens it is possible to go back to the main menu. Directly in the main menu it is possible to set the uncertainty level of the application. There are four levels to choose from. A higher uncertainty level means more uncertainty is tolerated between GPS updates. This will in turn lead to longer time between updates, and therefore extend battery life.

The last option on the main menu is to choose the destination. This can be chosen multiple times while the program is running. When the user enters the screen a search field is shown. Here he can input the beginning of the destination's name. The application will then list all streets beginning with those letters, and the user can choose one of them to be the routing target.

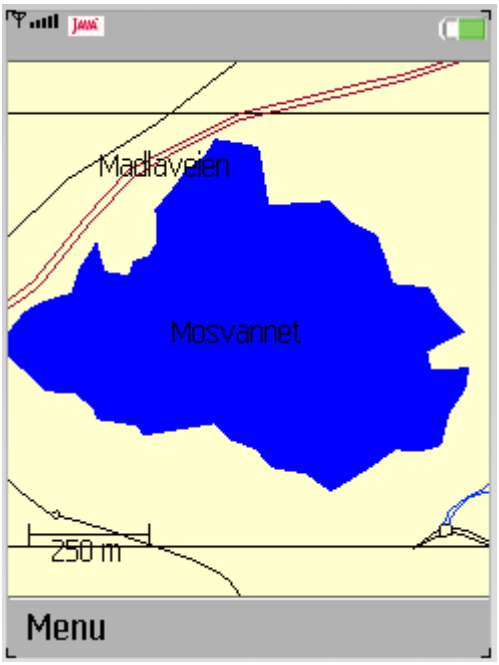

Fig. 4.2: The Map screen

Fig. 4.3: The Choose Destination screen.

4.1 Packages

The Travel Application consists of 7 separate packages, which are responsible for different parts of the program.

- The common classes package is for all the classes that are used by the other packages. Many of them describe common objects like positions and polygons. The controller, the top level object beneath the GUI, lies in this package.

- The map package has all the classes that are necessary to construct the map. It has classes for roads and buildings. It also houses the OpenStreetMap parser.
- All the classes concerned with routing from source to destination are in the routing package. Both the routing graph and the routing algorithm implementation are in this package.
- Uncertainty is an important topic for the Travel Application. That is why there is a separate uncertainty package. Its role is to calculate the next update time and spread the uncertainty with time. There are two uncertainty schemes, walking and driving, and both belong in this package. The uncertainty package is covered in chapter 7 in this report.
- Since there can be quite some time between the GPS updates, it would be unacceptable to provide the user with no position updates in between. These position updates are handled by the dead reckoning package. This package contains different classes for each reckoning scheme, for instance off road with no route. Which reckoning scheme to use is chosen based on the current situation. More details can be found in chapter 8.
- There is a separate package for device features. It handles the connection to the accelerometer and to the GPS.
- The last package is the GUI package. It contains methods for drawing the map and the menus.

4.2 Threads

The program is made up of the following execution threads:

- The main thread. This thread starts the program. It displays the main menu and constructs the controller. The controller parses the map and sets up the GPS listener to begin receiving GPS updates.
- The GUI threads. They handle any user generated events. These include pushing buttons to navigate menus and change the zoom level. These threads are automatically generated by the Java Virtual Machine.
- The repaint thread. When the map screen is shown, it repaints the map every other second. This thread also updates the position of the device and the corresponding uncertainty. It performs the dead reckoning
- The GPS listener. This is a JVM generated thread. It is generated by the GPS implementation every time the GPS position is received. It determines the uncertainty type and, based on the current situation and uncertainty level, calculates the time until the next GPS update.
- The accelerometer listener. The accelerometer listener is started by the repaint thread. It is started when one of the dead reckoning implementations tries to figure out which way to turn at an intersection. It starts at most 5 seconds before the intersection and ends 5 seconds after. The collected data can then be utilized to estimate the current position. This position determines which road the reckoning scheme chooses in an intersection.

5 The map

The Travel Application needs geographical data to work. For instance it must know where the user is placed, in relation to the road network, to be able to support routing. This data comes in form of a map. Which map to use, therefore became one of the first questions that needed an answer. There are two main categories of maps used in modern geographical information systems (GIS), raster maps and vector maps.

5.1 Raster maps

Raster maps are images. The most common sources for today's raster maps are satellite imagery and aerial photos. It could also be maps drawn on paper and scanned in on a computer. The raster map must have a scale (for instance 1:10000) and at least one known position, to be able to place a position (like a cell phone's current position) on the map. When zooming in a raster map, there are two different strategies. One is to stretch the pixels when zooming in, so that the screen shows a smaller part of the total map. This changes the scale of the map. The other strategy is to replace the image with another one that covers the same area but has a closer zoom. This strategy yields a better image at the expense of taking up more storage space. A commonly used strategy is a combination. This means that one stretches the original image until a certain zoom level is reached, at which time the image is replaced by a more detailed one.

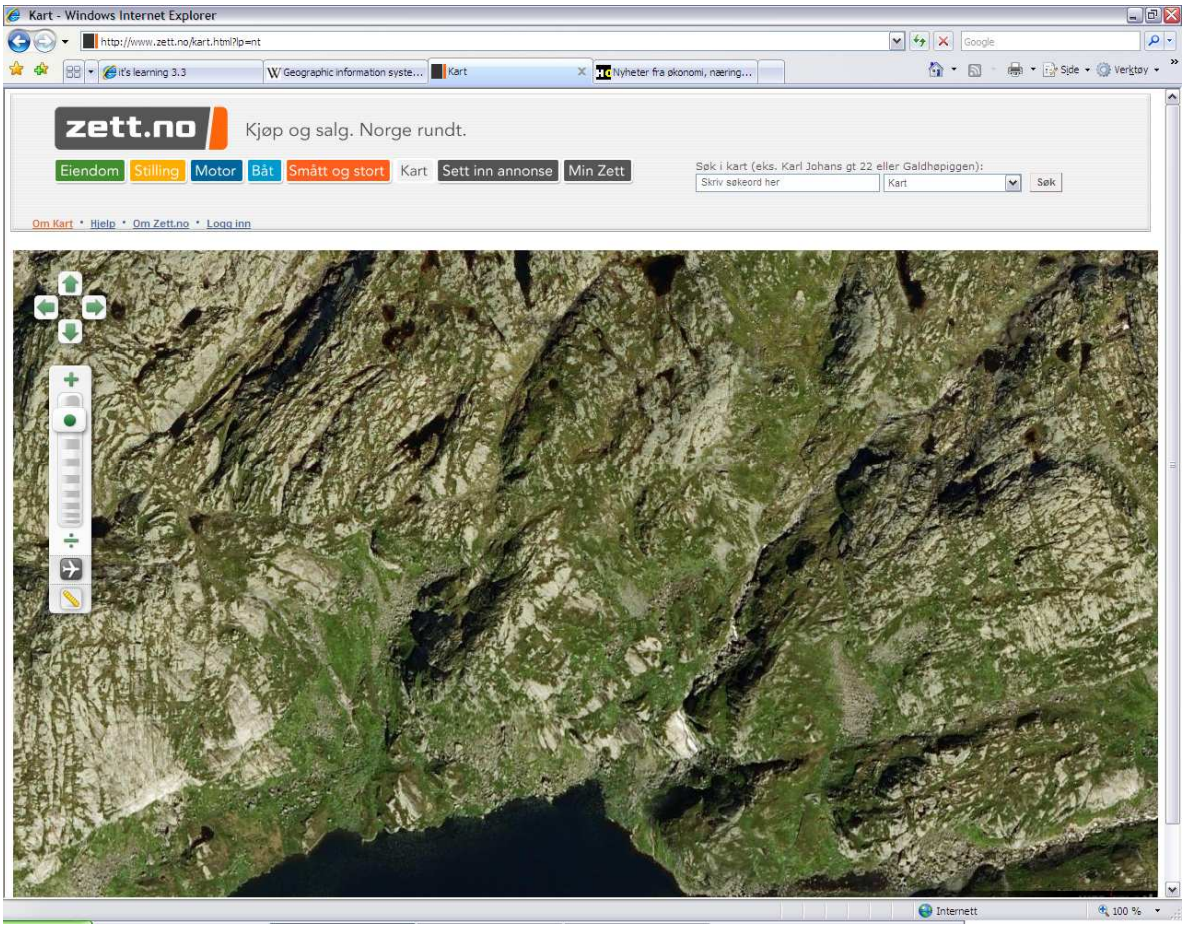

Fig 5.1: An aerial raster map image from zett.no [13].

5.2 Vector maps

Vector maps consist of geometric primitives. They use points to describe the position of points of interests, for instance shops, and museums. They use lines, or collection of lines (polylines), to represent line shaped geographic objects. These include roads, rivers, power lines and railways. The last kind of geometric objects are polygons. They are used to represent areas. It could be an estate, a lake, a building or a parking lot. These objects are stored in data structures. The map is rendered using a graphics library. Scale and zooming are handled easily. One can for instance choose to only render motorways, not secondary highways, when the current view of the map is large. The primary disadvantage of vector maps in contrast to a raster map is that it is harder to build. A raster map can be generated from satellite imagery, but to make a vector map someone has to enter the data. This can be done manually or auto-generated from raster maps using image processing techniques. Another disadvantage is that the process of rendering vector maps is more complex since one need to actually implement the drawing methods, compared to the easiness of just drawing an image.

One advantage of vector maps are that they can be easily used for applications like routing, since the roads are objects that can be accessed via the data structures. This is not possible in a raster map. Another advantage is the file size of a vector map, compared to a raster map of the same area. Since the vector map only needs to have a data about the geographical objects that may be of interest, compared to the raster which maps every square meter, the vector map can be a lot smaller. A different advantage of the vector map comes into play when a new highway is built. Since every road is its own object, the new highway simply can be added to the existing road data structure and it will appear the next time the user renders the map. Updating the map is a lot harder with raster maps since new imagery has to be captured.

The travel application should guide the user from his current location, to where he wants to go. This requires a map that can be used for routing. Ergo a vector map was needed. As an added bonus vector maps takes up less storage space, which is great since storage comes in limited supply on mobile phones.

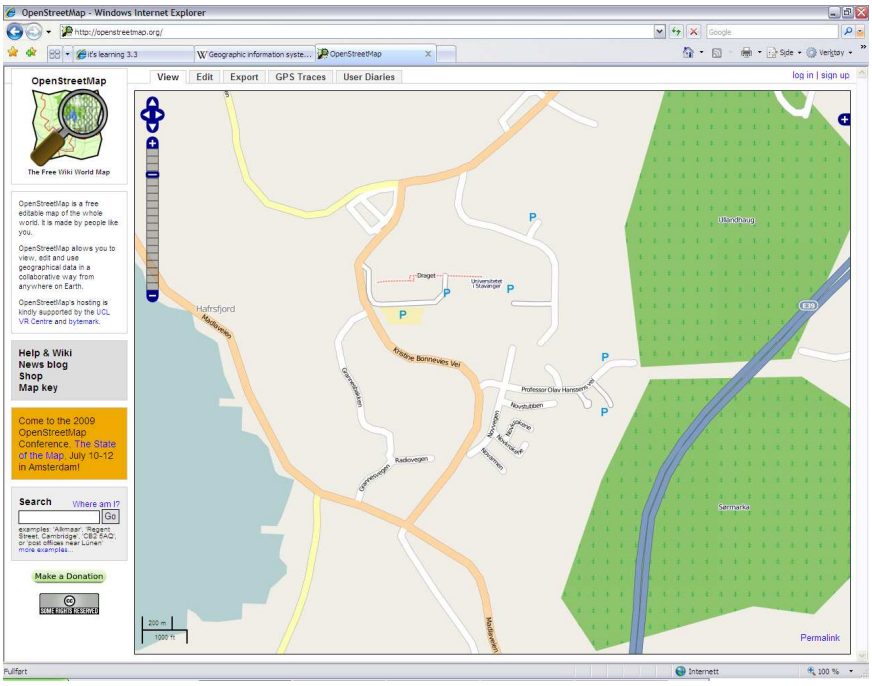

Fig. 5.2: A screenshot of a vector map from openstreetmap.org [14].

5.3 OpenStreetMap

In this project, OpenStreetMap (OSM) [14] was used as the source for geographical data. OSM was chosen for a couple of reasons. The main reason was because it is free of charge. The second reason was the easy data export. You go online and mark what rectangle of the map you are interested in. When you select export you get a well-formatted XML-file of all the data that is in your selection. Of course it is also a vector map, which is what the Travel Application needs. It uses the WGS 84 datum for the map locations, the same datum that is returned from most GPS receivers. This datum is used in the W760i as well.

OpenStreetMap is like a "Geographical Wikipedia". It is organized like a wiki, so that anyone can change the map, anytime they like. Most of the roads that exist on the map have been entered by normal users. One way to edit the map is to use a GPS device. A GPS program can be used to track a user's movement. After that these so-called GPS Traces (in GPX format) can be imported into OSM map editing software (like JOSM) on a computer. Here the roads can be classified and named. It is possible to upload the changes onto the OSM server via the map editing application. The web map will be automatically updated after some time (approx. 1 day). It is also possible to edit OSM without GPS traces. The person editing then draw the roads directly in the map editing software. Some of the data in OSM have been contributed by governments, to update entire areas with great accuracy.

A screenshot of OSM is shown on the previous page.

5.3.1 OpenStreetMap ontology

The intention in the travel application is to convert OSM data from the exported XML-file into the Travel Application's proprietary map ontology. The Travel Application has its own ontology because the OSM ontology is very comprehensive. Implementing the entire OSM ontology would take too much time. The Travel Application ontology therefore only includes the classes that are most essential to a navigation application. To be able to convert the OSM data into the Travel Application's ontology it is important to understand how the OSM data is organized. There are five standard object types which are found in almost every OSM data export:

- The OSM object is found once in each export. This is the top level object that has all the other objects as children. It includes some info about the OSM version and generator.
- Another object which is also found once per export is the Bounds. It describes the boundaries that were used as an input when creating the map. Bounds use minLon, maxLon, minLat and maxLat to do this.
- Tags are used to give information about the map objects. This can be used by application developers. All tags have a type and a value. For instance if a road has a tag of type name, and value "Grannessletta", the program is meant to interpret "Grannessletta" as the road's name. One object can have many tags. The tag types are standardized and a complete list is given on the OSM wiki [15]. Without tags there would be no difference between a road and a river.
- A very important class in OSM is the node. The map is built up by nodes. Every road, area and river is defined by the nodes that they consist of. A node has a unique id that is used to reference it. It has a longitude and latitude to indicate its location. The node also carries the name of the creator and a timestamp to indicate when it was created. It can also have some tags applied to it. For instance if it is the location of a school, it

can have a tag that says it is an amenity of type school. Another tag can then be used to indicate the school's name.

- Ways are used to indicate that certain nodes have a common connection. This could for instance be railways and roads. They have tags just like the nodes. They include a vector of node references. Each node reference contains the id of one of the nodes in the exported file. The sequence in which the node references appear, matters. For instance if the road is one way, the sequence indicates the driving direction. It also indicates which nodes are connected. A special case of the way is the area. The nodes in the sequence are corners on the boundary of the area. Here the sequence is important, since there is a rule that defines the inside of an area. The rule is: If moving from one node to another in the correct sequence, the inside is always on the left. Areas are closed, so the start node is identical to the end node.

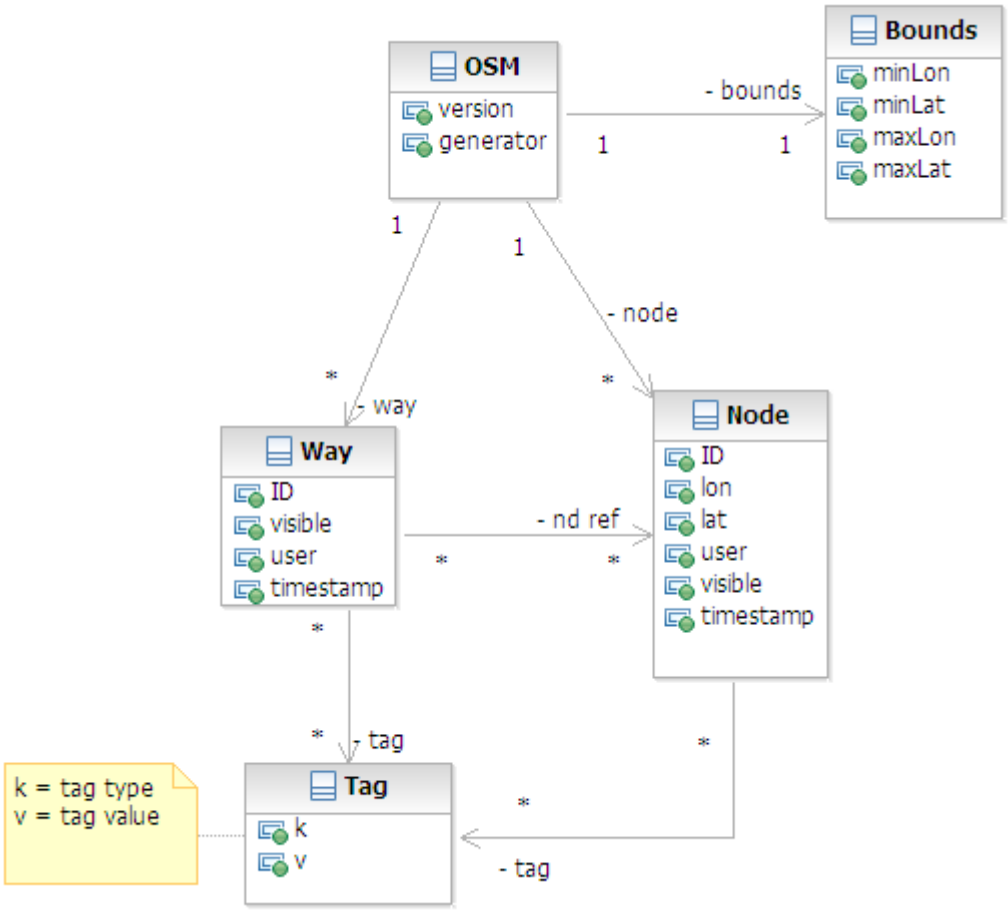

Fig. 5.3: Overview of the OSM ontology.

5.3.2 OpenStreetMap XML file format and parsing

The OSM export function returns an XML-file called map.osm. On the next page there is a simple example of what it might look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.5" generator="OpenStreetMap server">
  <bounds minlat="58.92893" minlon="5.68252" maxlat="58.94095748" maxlon="5.7"/>
  <node id="266570823" lat="58.9339598" lon="5.6993738" user="Rolf Marthin Nilsen" visible="true"
timestamp="2008-05-23T22:43:16+01:00"/>
  <node id="266571376" lat="58.9342421" lon="5.6995141" user="Rolf Marthin Nilsen" visible="true"
timestamp="2008-05-23T22:48:14+01:00"/>
  <node id="266571377" lat="58.9346229" lon="5.6996605" user="Rolf Marthin Nilsen" visible="true"
timestamp="2008-05-23T22:48:15+01:00"/>
  <node id="266571378" lat="58.9351869" lon="5.6998953" user="Rolf Marthin Nilsen" visible="true"
timestamp="2008-05-23T22:48:15+01:00"/>
  <way id="24512596" visible="true" timestamp="2008-05-23T22:48:15+01:00" user="Rolf Marthin
Nilsen">
    <nd ref="266570823"/>
    <nd ref="266571376"/>
    <nd ref="266571377"/>
    <nd ref="266571378"/>
    <tag k="name" v="Kringkasterveien"/>
    <tag k="created_by" v="Merkaartor 0.10"/>
    <tag k="highway" v="unclassified"/>
  </way>
</osm>
```

Listing 5.1: A simple example of a map.osm file.

To translate the contents of this file into map objects that can be used in the Travel Application, it needs to be parsed. Since it is an XML-file it was easiest to use the XML-parser that is included in J2ME.

The parser is called SAXParser. SAX stands for Simple API for XML. It takes an XML-file and a default handler as input. To use it one must make a class that extends the default handler. The default handler only contains empty methods. The parser reads through the XML-document sequentially from the top. As it comes over tags and symbols in the document, it calls the corresponding method in the default handler. The methods that had to be implemented were start- and endElement, together with startDocument.

The great thing about the structure of the OSM XML-file is that all the nodes appear before the ways. This makes it possible to be certain that when you stumble upon an nd ref in the file, the corresponding node exists in your data structure.

Since the actual object type of the way is indicated by tags, the application has to first make a node collection when it comes across a way. Then it adds all the referenced nodes to this collection. And when the parser comes across a tag that belongs to a certain class, it makes a new instance of that class. This instance is given all the nodes of the node collection. When the parser comes across a </way> tag it can add it to the correct data structure in the map, according to the class. For instance roads were added to the road collection.

5.4 The Travel Application map

The Travel Application gets the map.osm XML-file from the application's JAR archive. It could also have read it from the file system (the phone memory or memory card). The archive was chosen because it is easier to have it in the JAR file when running the application in an emulator. To emulate the phone's file system the emulator requires a location on the hard drive to act as a file system root. This has to be pointed out manually.

Another way to get the map would be to download it on demand from the OSM server. This means that you would only download the map of an area when you were close to it. Since the W760i has both a 3G and a GPRS Internet connection, this would be possible. The Travel Application does not include download on demand for a couple of reasons:

1. Download on demand is more complex than downloading the map in advance. Since the core part of the project was regarding the uncertainty and battery saving of the mobile GIS, this would not add enough value to the project, at least not enough to defend the added implementation time.
2. Download on demand is not always better than download in advance. If you want to look at the map when you are outside of cell coverage, this is possible with download in advance. However if you are downloading on demand you need an Internet connection. Using the connection also drains battery power, which the Travel Application is meant to preserve.
3. Download on demand is more expensive than download in advance. This is because with download in advance you use a PC with an Internet connection to get the map. This is free. Download on demand on the other hand generates traffic over the mobile network, where you often pay per MB for the traffic. This would add an unnecessary cost to the testing phase of the project. This fact is even more important if one would want to use the Travel Application abroad. Using the mobile network for data traffic could be very expensive.

5.4.1 The ontology of the map

Below is a model of the map ontology.

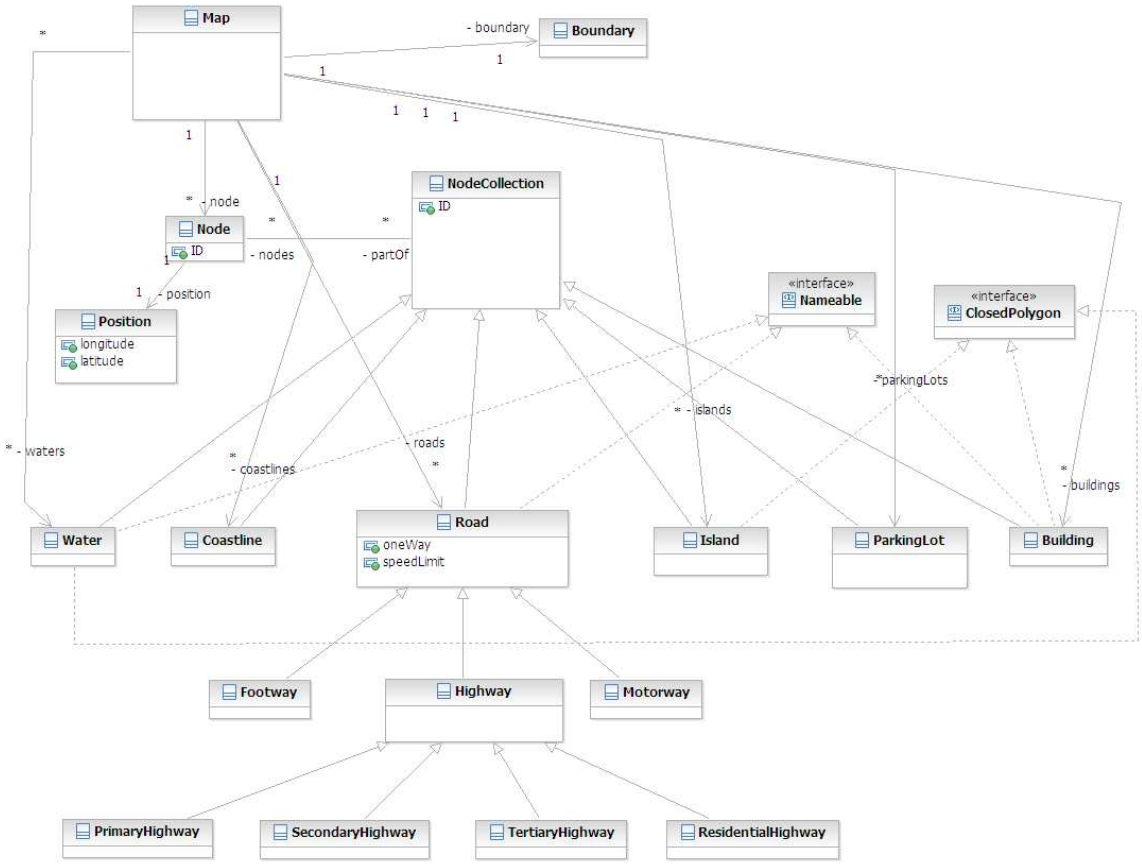

Fig. 5.4: The map ontology.

The Nameable interface is used to indicate that the implementing objects could have a name. The ClosedPolygon interface is intended for those objects in OpenStreetMap that is listed by their boundaries.

5.4.2 Drawing the map

To draw the map on the mobile screen, the J2ME MIDP 2.0 Graphics API was used. This is a 2D Graphics API. The greatest difference between it and the normal J2SE 2D Graphics API, is its simplicity. For instance there are no predefined colours in J2ME, one has to use the RGB values. There is also no native support for rotation of the drawing objects around a point. In J2SE one can draw smooth segmented lines using B-splines, this option is not included in J2ME. This could have been a useful function when drawing the roads.

My greatest concern with the J2ME Graphics API was the lack of a method for filling arbitrary polygons with colour. In the Travel Application map there are a lot of polygons, for example waters, buildings and parking lots. In J2SE there is a function to fill a polygon given its edges. Since the Travel Application stores the edges of the polygons, it would have been easy to use this function. To work around this issue, the Travel Application implements its own fill polygon method.

The fill polygon method exploits the fact that any simple polygon can be divided into a finite number of triangles [16]. A simple polygon is a polygon that has no crossing lines between the edges and no holes.

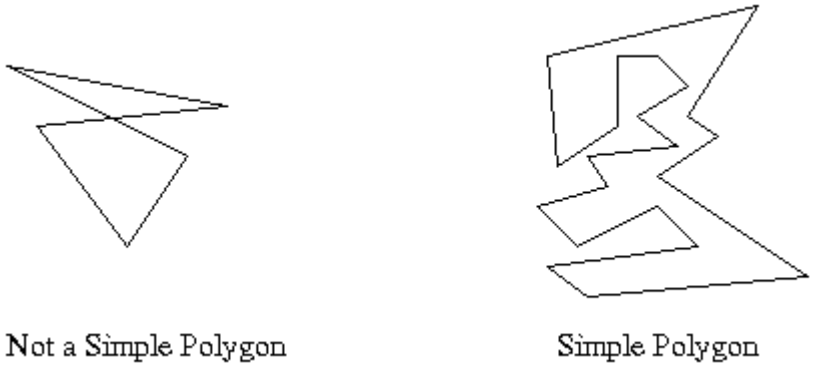

Fig. 5.5: The difference between a simple and a not simple polygon [4].

The method that was used to find these triangles is called ear cutting. The theory behind ear cutting is that a simple polygon which has the correct triangle (ear) removed, is still a simple polygon. The exception to this rule is if there are no more ears to remove, in which case the whole polygon has been cut away. It is the removal of the ears that has lead to the name ear cutting. The process is illustrated beneath.

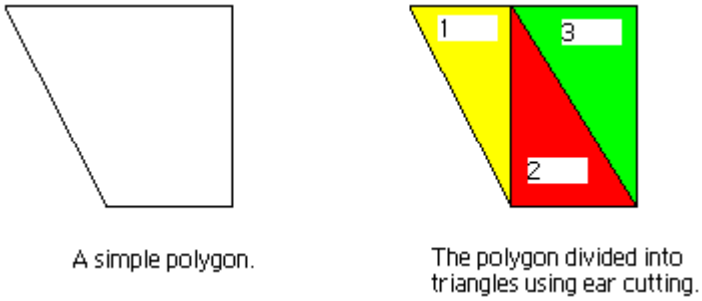

Fig. 5.6: Ear cutting.

In the Travel Application the brute force ear cutter found in [16] was used, with only minor modifications to make it fit. Finding the triangles turned out to be quite a processing intensive task. For instance drawing Mosvannet using this method took 140 milliseconds. Since the triangles don’t change during the execution of the program, the best option is to calculate them only the first time the polygon is drawn and then store them in memory for later use. When this improvement was implemented, the time needed for drawing Mosvannet, after the first time, was reduced to 30 ms.

Fig. 5.7: Mosvannet from OpenStreetMap [14].

In the J2ME Graphics API when something new is drawn, what is previously drawn on the screen is drawn over. This makes the sequence the objects are drawn in essential. The Travel Application uses the following sequence (The last objects are drawn last):

1. Draw the background blue to indicate the ocean.
2. Fill the land off-white using the coastlines as a guide.
3. Draw the coastlines black.
4. Draw the boundaries of the Open Street Map source file
5. Fill any lakes and waters that might be on the map with blue colour.
6. Fill islands off-white.
7. Draw roads in the colour specified by the road's class.
8. Fill buildings red.
9. Fill parking lots greyish blue.
10. Draw the route from the mobile phone to the destination green, if there is a destination.
11. Draw the uncertainty of the mobile device position blue. It draws it either as a circle or as a dotted line along the roads, depending on the type of uncertainty.
12. Fill the centre triangle blue. This is the triangle that indicates the current position and course of the mobile device.
13. Draw the texts that should be displayed. For instance street and lake names.
14. Draw the scale ruler black. This is used to indicate the scale of the map on the screen, compared to the real world, at the current zoom level.

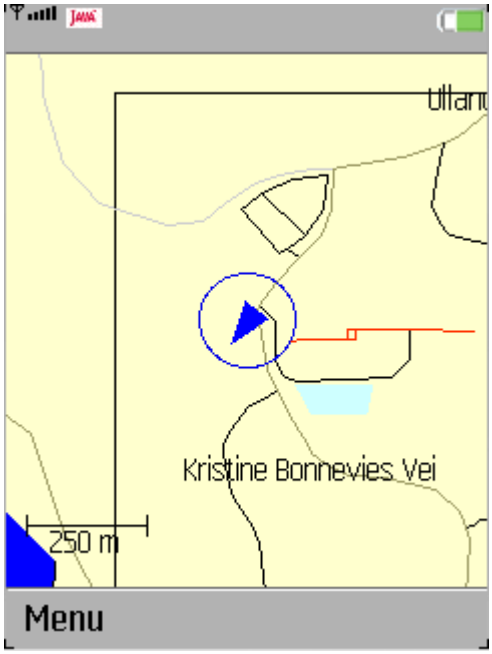

Fig. 5.8: Screenshot of the finished map.

Drawing the roads and routes were pretty straightforward. The method used was to draw simple lines between the nodes. One thing that needed to be taken into consideration for all the drawing was the relationship between longitude and latitude. One unit of latitude is not equally long, in meters, as one unit of longitude. If the longitude and latitude values were directly translated into x and y coordinates, the result would be a distorted image. First they need to be scaled according to the number of meters per longitude or latitude. The meters per latitude are fairly constant at about 111280 meters. The meters per longitude however, changes with latitude. Using [17] the meters per longitude was found to be around 57415 meters in Stavanger. In the travel application this is just implemented as static variables that need to be changed when replacing the map with a different latitude one. For a more dynamic approach one could have used a table or implemented a meter per longitude calculator.

One of the hardest things to get right when drawing the map is the coastlines and the separation between sea and land. A special case is when there are no coastlines included in the downloaded map section. This could mean that you are in the middle of the ocean, or that you are on dry land, but not close to the shore. In this case the Travel Application draws land if there are roads in the map. If not it draws blue ocean. The coastline in OSM is defined as a sequence of line segments with land on the left side and water on the right. When downloading a map of an area, you won't get the coastline for the entire world. You only get the part of the coastline that is inside, or close to, the area's bounding rectangle. The Travel Application needs to make one or more simple polygons out of the land, to be able to draw it. This means that one will need an algorithm to possibly "fill in the blanks", when the downloaded coastlines don't provide enough information to make the land form a simple polygon. This is done in two ways. If the entire downloaded coastlines are contained inside the screen, the four corners of the screen are added to the coastlines to make the land into a polygon. If it extends beyond the screen some faraway corners are used (currently the corners of a square with 50 km sides and a middle point located at the centre of the screen).

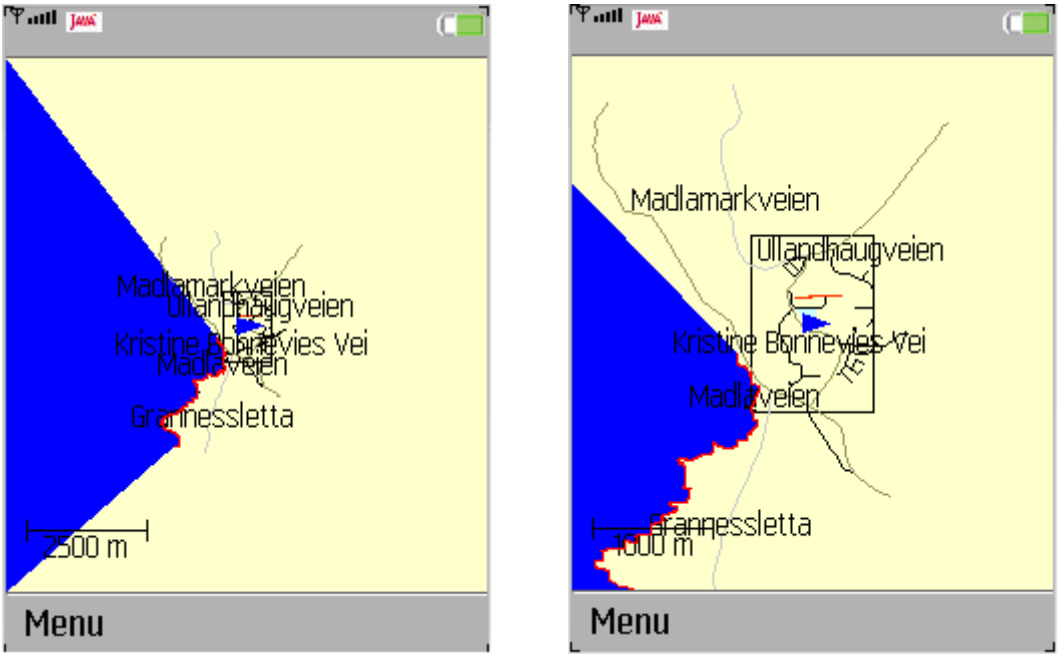

Fig. 5.9: Screenshot comparison of screen corners and faraway corners. The blue ocean stretches to the corner of the screen on the right hand screenshot. In the other one it stretches to a corner far from the centre of the screen. The part of the coastline that was defined in the map is marked in red.

6 Routing

Routing is an important function in the Travel Application. It is not always easy to spot the fastest route to a destination only by looking on the map. But by following the result calculated by a routing algorithm, one will have no trouble choosing the correct path.

When implementing the routing function it was important to choose the right shortest-path algorithm. Since there are already developed some good ones, an existing algorithm was chosen. This is more efficient and most likely leads to a better solution than developing a new shortest-path algorithm just for the Travel Application. To be able to pick the best algorithm, one first has to understand the shortest-path problem.

6.1 Graphs and the shortest-path problem

A graph consists of nodes and edges that connect the nodes. If some of the edges can only be traversed in one direction, we talk about a directed graph. This is exactly what we have in the Travel Application's road network. The road nodes are the nodes. They are linked together by roads. They are the graph's edges. The roads can be one-way, so we are talking about a directed graph.

If the edges are associated with weights we have a weighted graph. The weight indicates the cost of using an edge. In the Travel Application the weight represents the time it takes to travel between two nodes, using the road indicated by that edge.

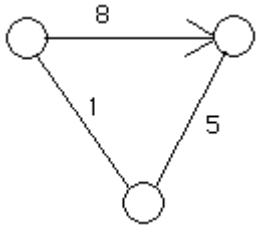

Fig. 6.1: A simple weighted, directed graph.

The single pair shortest-path problem is to find a path, from a source node, to a destination node, such that the sum of all the weights in the path is minimal. This is almost exactly what is needed to be able to give driving directions in the Travel Application. The difference is that there are a set of destinations, instead of just one. In the Travel Application it is only possible to route on a street level. This is because house numbers are lacking from OSM. Since a street consists of multiple nodes, there will be more than one destination. But through some small simplifications, one can apply the single pair shortest-path problem.

1. First choose the geographically closest node, from the source node, on the destination street, as the destination node.
2. After one has calculated a route to the destination node, cut off the route at the first node that is on the destination street.

This will give you the shortest route in most cases. Problems occur if there for instance are shortcuts from the source node, to the other end of the destination street. But in my tests this has yet to occur.

On the next page one can see an illustration of the shortcut problem. The circle represents the source position. The destination street is indicated in red. The X marks the

"node" which will be chosen as the destination, due to its small geographical distance from the source. The fastest road to this destination goes via the blue road. But since the destination originally was the entire street, it would have been better to choose the green road.

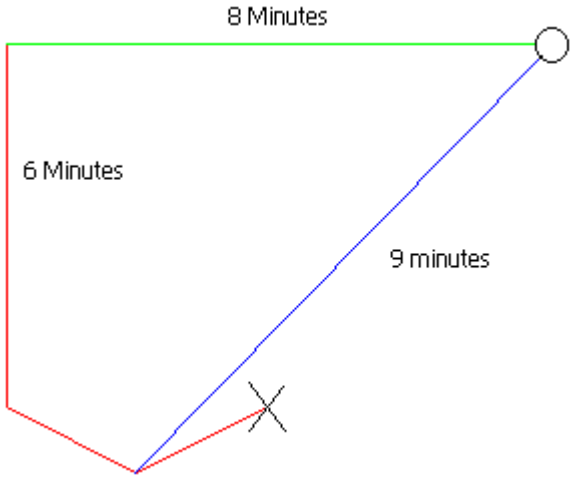

Fig. 6.2: The problem of using the closest geographical node as a destination.

6.1 Choice of shortest-path algorithm

A shortest-path algorithm is an algorithm that solves the shortest-path problem in graph theory. The following algorithms were considered for use in the Travel Application:

- Dijkstra's
- Bellman-Ford
- A*

Dijkstra's algorithm calculates the shortest path between a source node and every other node in a graph. It is then easy to find the shortest route to the destination street. Simply choose the path with the lowest calculated time consumption to a node on that street. This is however not an optimal solution. It calculates the path to all the other nodes in the entire graph, when the Travel Application only needs the fastest path to one of the possible destination nodes.

Bellman-Ford is an algorithm that calculates the shortest path from a source node to all the other nodes in a graph. The difference from Dijkstra's algorithm is that it supports negative weights on the edges. The downside is that it runs slower than Dijkstra's [18, 19]. It has a time complexity of O(VE), compared to a good implementation of Dijkstra's at O(E+V log V). E is the number of edges in the graph and V is the number of vertices. In the Travel Application the weights stand for time consumed when using the road represented by the edge. This cannot be negative, so there was no call for the Bellman-Ford solution.

The A* Algorithm computes the shortest path from a single source, to one or many destinations in a graph. It tries to search along the best path first. The next node to be examined is determined using a heuristic function, f(x). f(x) is the sum of two other functions, g(x) and h(x). The unexamined node with the lowest f(x), is considered the node with the highest potential and is therefore picked next. g(x) is a measurement of the actual shortest distance from the source node to node x. In the Travel Application this will be a measurement of the time used to reach x from the initial node. h(x) is a best case estimate of the distance from x to the destination. If there are multiple destinations it calculates h(x) for all the destinations and uses the best of them. Since the Travel Application cares about time, h(x) is the shortest possible time it takes to travel from x to the destination. This is the straight line distance divided by the highest speed limit (the motorway speed limit of 90 km/h). The great

thing about using f(x), is that the algorithm will check the most promising direction first. If the destination is reached in 100 seconds using one path, the examination of other paths can stop when their f(x) exceeds 100 seconds. These two features combined ensure that it is possible to reach a solution for the problem very efficiently.

Since the Travel Application has all the features needed to use A* and it is the fastest of the considered algorithms, A* was chosen to be implemented in the program.

6.2 Implementation of the A* algorithm

To be able to calculate the shortest path, the routing graph with nodes and edges must be created. In the Travel Application this is created from all the roads in the map. It is only done once per program execution. Each node on the roads is represented as a route node in the graph. To determine neighbourship the application takes advantage of the sequence of the nodes in OSM. The sequence is combined with the one way property of the road. This means that if node Y follows node X in the sequence, both will have each other as neighbour. If the road is one way however, only X will have Y as neighbour. If a node exists in more than one road (this indicates an intersection in OSM) it needs no special treatment, the neighbours from the second road is just added to the same route node.
Here is a UML diagram representing the structure of the routing part of the Travel Application:

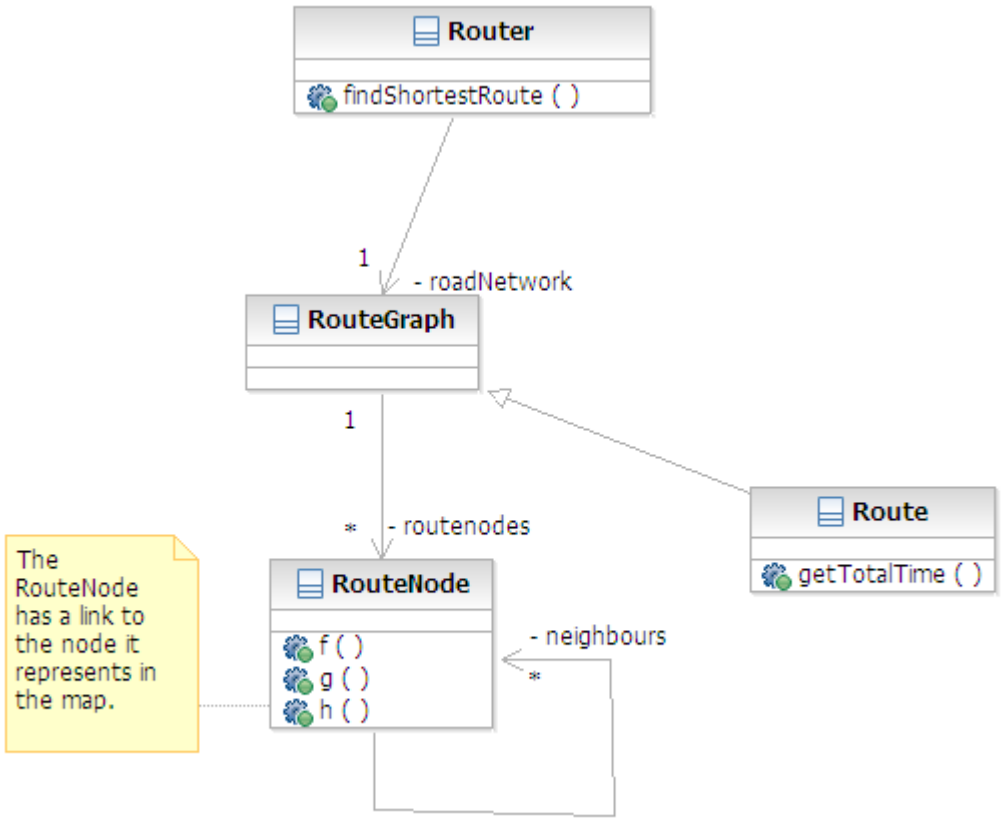

Fig. 6.3: The routing package.

The route graph represents the graph. The route node class represents the nodes. When combining the neighbour relationships, one gets the edges. A path in the graph is a route in the Travel Application. It extends the route graph since it also is a collection of route nodes.

The router contains the A* algorithm. To calculate the shortest route it requires a street name as a destination and the current position of the cell phone as a source. Since the cell phone position is not included as a node in the graph, it needs to be inserted with the right neighbour. Choosing the right neighbour is done using the f() function. The node with the lowest f-value given the phone's position as a source (or route parent) and walking in a straight line towards the node, is chosen as a neighbour. When one is outside the road, the router always assumes movement occurs at a walking speed (6 km/h). When on a road, it assumes that travel happens according to the road's speed limit. The pseudo code for the A* algorithm can be found in appendix A.1.

Implementing A* was easy since the routing graph and nodes were already made. The routing function was first implemented by finding the shortest path to the geographically closest node on the destination street. This is what is described on page 30. A specific routing then took 16 ms (From the University of Stavanger to Grannessletta). At a later time the routing was extended to calculate the shortest path to the entire destination street. This could be done by making the destination a set of route nodes instead of just one. First the set included all the nodes in the street, but then it took 160 ms to route (Grannessletta has 40 nodes). Since longer routing time equals more power consumption (through processing), it is important to reduce it as much as possible. This time can be greatly reduced by taking advantage of the following fact: The destination node will be in an intersection. The only exception is if one is standing so close to the destination street that the first node one uses to reach the destination, is the destination. This situation can easily be handled by adding the destination street's geographically closest node to the source, into the destination set, if it is not already there. With this improvement the time needed to route went down to 32 ms (Grannessletta has 2 intersections). It was still twice the time needed for the original routing. In the testing phase both the single destination and the improved street destination routing were tested. This was to see how much shorter the battery would last with the improved routing and then to determine if it was worth it, given the added accuracy.

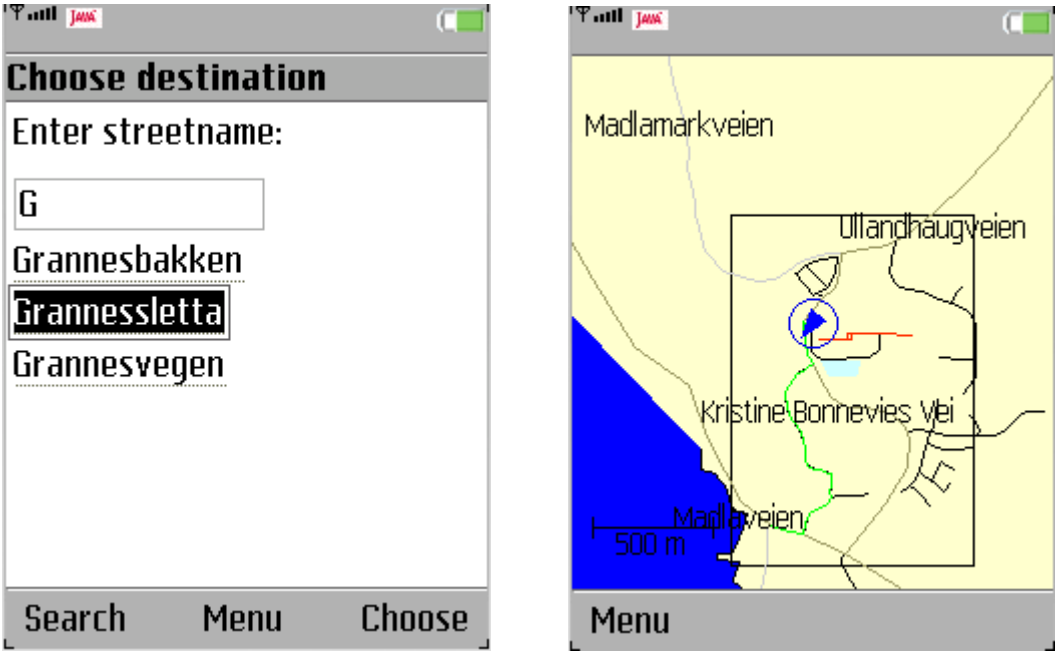

Fig. 6.4: Routing to Grannessletta in the Travel Application.

7 Uncertainty

The intention of the Travel Application is to be conservative in relation to power consumption. It tries to achieve this by extending the intervals between each GPS update. The drawback of this technique is that it adds some uncertainty regarding the current position. For instance if you have not received any GPS updates for the last 3 minutes, you can have travelled quite far. If you are doing 90 km/h, you could have travelled up to 4500 meters in one direction. This makes handling the uncertainty important, so that the application can determine which time the next update should occur.

7.1 The GPS uncertainty

There is some uncertainty about the position of the cell phone even if the application has just received a GPS update. This is due to the inaccuracy of GPS. In the Travel Application the concern has been the horizontal accuracy (longitude, latitude) of GPS. GPS can also determine altitude, but this is associated with a greater inaccuracy. The following factors contribute to the GPS error [21]:

- **Satellite geometry.** In GPS you estimate your position based on the distance from your location to at least 4 satellites. If some of these satellites are close to each other, the area of your possible location grows larger. This is because of the circular uncertainty areas around the satellites have a longer overlap. This is shown below with the blue area indicating the possible location for receiver A. Since the satellites are not geostationary, the error caused by satellite geometry will change with time. Also a GPS receiver can calculate a position with signal from as little as 4 satellites, but the best receivers can connect to 20 satellites. The maximum number of satellites observed while testing the W760i was 8. Having extra satellites available can help reduce the uncertainty. The error caused by satellite geometry cannot be measured directly in meters, but rather works as an amplifier to the other error sources.

Fig. 7.1: Good versus bad satellite geometry [20]. The good on the left.

- **Satellite orbits.** Slight shifts in the satellite orbits from what is stated in the ephemeris. This means that the satellites are not located exactly where the phone thinks they are. Regularly controlled so that the error caused by this is no more than 2 m.

- **Multipath effects.** The phone always assumes that the signal has travelled in a straight line from the satellite. However it could be that the signal instead has bounced of something solid on the way. This means that the signal will travel further and be more delayed than it was supposed to. This will again lead to the phone thinking it is further away from the satellite than it actually is. Multipath effects are most common in narrow streets with tall buildings. The typical error is a few meters.
- **Atmospheric effects.** When entering the atmosphere, the signals will travel at a slower speed. The receivers take this change into account when they calculate their position. However they do not take into consideration the changes in this speed drop. Sometimes it might be higher due to variable effects like strong solar winds. Military receivers can correct for this effect since they use two different frequencies. This gives them some meters less error. The amount of water vapour in the troposphere will also affect the arrival time of the signal. The error caused by atmospheric effects could be around 5 meters.
- **Clock inaccuracies and rounding errors.** Although the clocks are synchronized, there will still remain some timing inaccuracy. Also rounding of numbers will cause a small error in the receiver. These effects combined give an error of approx. 3 meters.

Altogether these factors sum up to a theoretical error of ca. 15 m. A good civilian receiver today can have an uncertainty radius of about 20 m.

Before USA turned it off selective availability was the greatest source of error in GPS. Selective availability was intended to give the US an advantage in case of a war. The way it works is that only those who know the codes get the detailed information. Among its intentions was to prevent hostile missiles using GPS for guidance. Selective availability could cause an error of up to 100 meters in civilian receivers. When GPS became popular, this feature was turned off on May, 2. 2000.

During testing with the Sony Ericsson W760i, most of the results were within 30 meters of the actual position. There were however some outliers. Some positions had an error of a little bit over 100 m.

J2ME's Location API has a couple of methods in the QualifiedCoordinates class dedicated to handling the error of a given position [21]. GetHorizontalAccuracy() returns the horizontal (longitude, latitude) accuracy in meters. It assumes the actual error follows a normal distribution. It returns the standard deviation of this error under the current conditions (number of satellites, etc.) This means that approximately 68 % of the returned results (if the same test is done many times) would lie within a circle with the returned radius, around the given position. GetVerticalAccuracy() is an analogue method for the altitude error. Of course here the uncertainty is not a circle, but rather an interval.

The Travel Application takes advantage of the getHorizontalAccuracy() method. The application uses it to set the initial radius of the uncertainty circle. This is done after each GPS update. If the device doesn't implement this method, for instance if 0 is returned, the radius is set to 100 meters just to be on the safe side. If one is working with an unaltered version of the J2ME emulator, 0 is returned. This can however be changed in the emulator's property file. The W760i implements getHorizontalAccuracy() correctly.

7.2 Determining the uncertainty type

The Travel Application separates the spreading of the uncertainty into two different main types: Walking uncertainty and driving uncertainty. The walking uncertainty is applied when the user moves slowly. It spreads in a circle. When the user moves too fast to be walking, the driving uncertainty is used. If there are any roads inside the initial GPS uncertainty circle, the

driving uncertainty will spread out using those roads. The speed of the spreading will be determined by the speed limits. The spreading algorithm assumes that one stays in the same uncertainty class until the next update. This means that if one is walking and starts to drive, one might end up outside the uncertainty circle for a brief period of time. This is not a major concern since the transition will be detected at the next update anyway.

The application decides which uncertainty type should be applied based on the speed of the device. It calculates the distance from the last GPS update to the current, and divides this on the elapsed time. If the straight line speed is greater than the estimated maximum walking speed (10 km/h), the user is considered to be driving. Else he is classified as walking. The Travel Application assumes that the user is not running or cycling. The Travel Application will consider both running and cycling as driving. The uncertainty will then spread too fast. The best way to support running and cycling is to implement two new uncertainty types. They would have to use a higher maximum speed than walking. For instance: 20 km/h for running and 60 km/h for cycling. However the cycling uncertainty should in any case not spread faster than the speed limit. When cycling, the uncertainty should spread along the roads. The running uncertainty could however be more like a fast walking uncertainty. Separating cycling from driving would be an issue, especially on roads with low speed limits.

Using only the previous and current GPS position to estimate the speed of the device, could sometimes be a problem. This is especially true when the interval is small. For instance: Given two returned positions, A and B, with an interval between them of 2 seconds. Assume that the device has not moved between the updates. Also assume that the horizontal accuracy is 30 meters. Given that A and B lies on opposite borders of the uncertainty circle, this leads to the following situation:

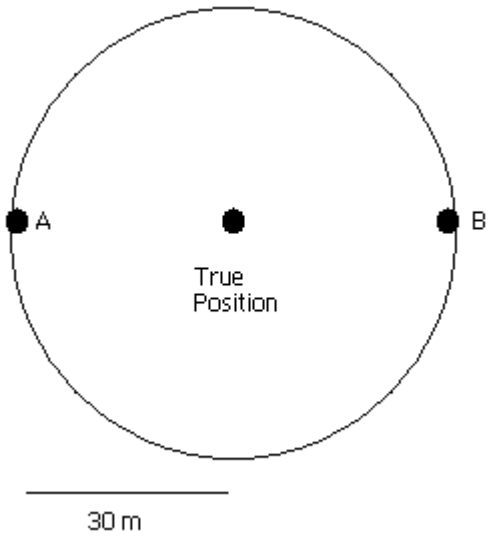

Fig. 7.2: The problem of determining speed using small update intervals.

Calculating the speed based on these two updates will give: (2*30 m) / 2 s = 30 m/s = 108 km/h. The real speed is 0. This problem can be handled by adding the calculated speed to a weighted average of the latest speeds. Then one can instead let this weighted average decide which uncertainty type to use. This has not yet been fully implemented in the Travel Application. The main reason is that in testing just using the previous GPS update has proved sufficient. The average update interval in the Travel Application is around 20-30 seconds. And that is not on the highest uncertainty settings. To apply this to the problem above: 60 m / 30 s = 2 m/s = 7.2 km/h => It would still be classified as walking.

To calculate the speed one needs at least 2 GPS positions. This means that some kind of start up process is needed before determining the uncertainty type for the first time. The simplest would be to just wait for 2 updates before calculating the speed. The Travel Application waits for 30 updates before determining the uncertainty type. During the start up process it uses a GPS update interval of 2 seconds. This gives a start up time of 1 minute before the speed is calculated. The reason this delay has been included, is to make sure that the user has reached a speed he will be travelling at for some time. This helps avoid wrong uncertainty classification. Here is a typical use case for the Travel Application, which illustrates the problem:

The user is sitting in his parked car. He starts the Travel Application. After that he enters his destination into the program. When he sees the route, he starts to drive.
If the start up process had begun determining uncertainty type straight away, he would be considered to be walking, since his speed was 0. When we wait 1 minute, he will be driving. In the meantime the Travel Application does not use the dead reckoning scheme, and just display the uncertainty as a circle with the related GPS error as a radius. Here the intervals are so small that it could be useful to employ a weighted average to determine the speed after the start up. The Travel Application implements a weighted average with exponentially decreasing weights. This means that the newer speeds are given more importance than the older speeds. The following formula was used to calculate the weighted mean speed:

$$\overline{v} = \sum_{i=1}^{m} w_i v_i$$

v is the weighted mean. There are m samples. w_i is the normalized weight for speed sample i. The sum of all normalized weights is 1. x_i is speed sample i. x_1 is the newest sample, x_m is the oldest.
w_i is given by this equation:

$$w_i = \frac{w^{i-1}}{V_1}$$

w is a constant between 0 and 1. It indicates how fast the weights shall decrease with age. The lower it is, the faster the weights decrease. In the Travel Application this is set to 0.8. If there are 30 samples in total, the 3 latest samples account for almost 50% of the mean. V_1 is the sum of all the unnormalized weights. It can be calculated as follows:

$$V_1 = \sum_{i=1}^{m} w^{i-1} = \frac{1 - w^m}{1 - w}$$

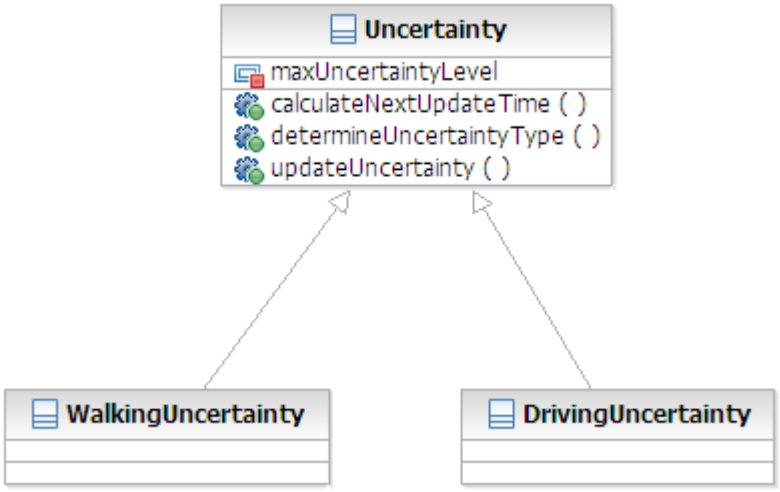

Fig. 7.3: An overview over the most important classes in the uncertainty package.

Fig. 7.3 shows the relation between the uncertainty class and its two subclasses. The maxUncertaintyLevel variable indicates how far the uncertainty should be allowed to spread before issuing an update request. This is used when calculating the next update time. The calculateNextUpdateTime and updateUncertainty methods have an empty implementation in the Uncertainty class. They are overridden by their respective methods in the driving and walking uncertainty. This makes it work like a finite state machine with 3 states: Undecided (start up), walking and driving.

The uncertainty type is the first thing to be determined after each GPS update. As previously mentioned this happens only after the start up process has finished. It returns either a walking or a driving uncertainty based on the current speed of the device.

The calculateNextUpdateTime method is called after each GPS update (not during start up). It returns the number of seconds until the next update. It is called after the uncertainty type is determined.

UpdateUncertainty is called for each repaint of the map in the GUI (every 2 seconds when the map is shown). This is another feature that is not used until after the start up period is over. It increases the uncertainty according to the uncertainty type. This method is used to give an indication of the current uncertainty to the user.

7.3 Walking uncertainty

The walking uncertainty is intended for users walking freely around the landscape. It starts out as a circle with the GPS' horizontal accuracy as a radius. The radius of the circle then expands with the maximum walking speed (10 km/h). Fig. 7.4 and 7.5 shows the logic of this spreading scheme through an example.

Fig. 7.4: The initial uncertainty at time t_0.

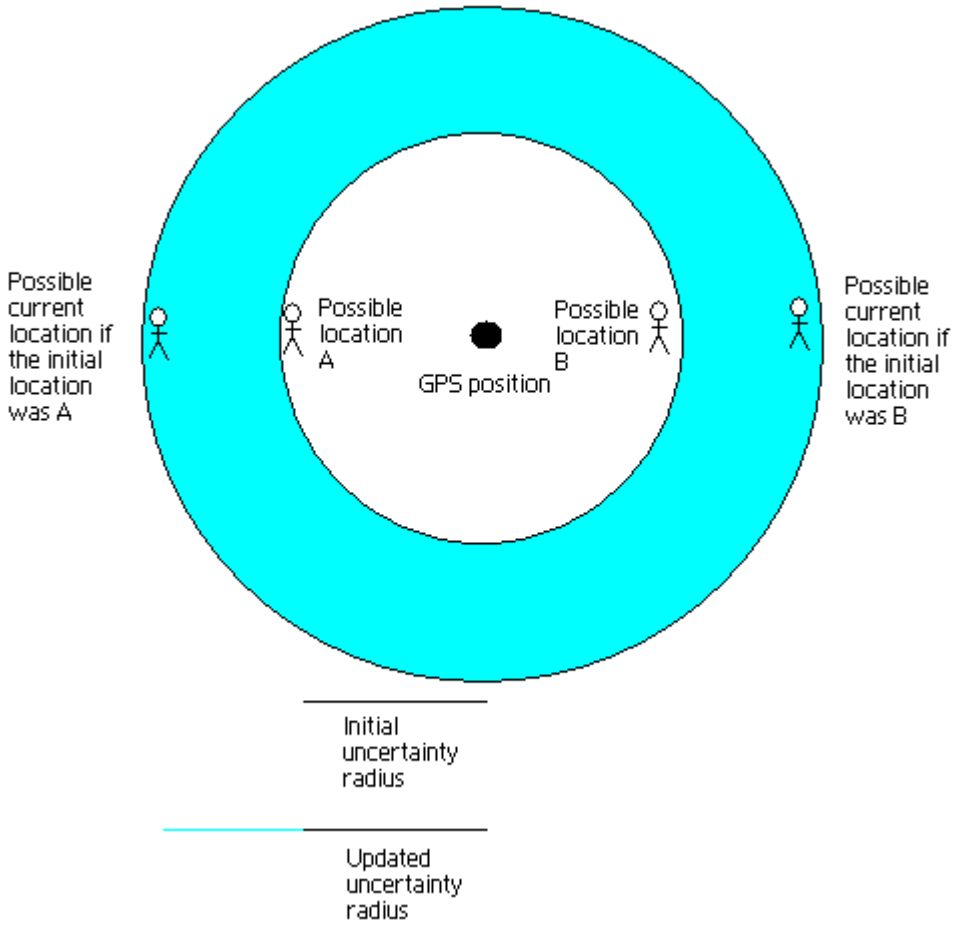

Fig. 7.5: The updated uncertainty at time t_1.

The initial uncertainty radius is s meters. The maximum walking speed in m/s is v. The updated uncertainty radius at time t_1 is then:

$$\text{Updated radius} = s + (t_1 - t_0) * v$$

To calculate the time until next update, the Travel Application finds out when the uncertainty diameter will reach the maximum allowed diameter, d, indicated by the current uncertainty level. This means solving the following equation:

Seconds until next update = (½ * (d/v)) – (s/v) (The time it takes to walk the maximum radius – the time it takes to walk the initial radius)

The maximum walking uncertainty levels are:

Level 1: 100 m diameter
Level 2: 200 m
Level 3: 500 m
Level 4: 1000 m

The uncertainty levels have been chosen such that on the lowest level it should only take a few seconds between each update. Level 2's diameter is twice as large as level 1's, but due to the fact that the GPS uncertainty instantly eats up a lot more of the first level's diameter, the time between updates are multiplied by a factor of approximately 4. More than 1000 meters of uncertainty will make the application useless for routing, so the maximum level was set accordingly. It would be useless both because of the high uncertainty and the long update intervals.

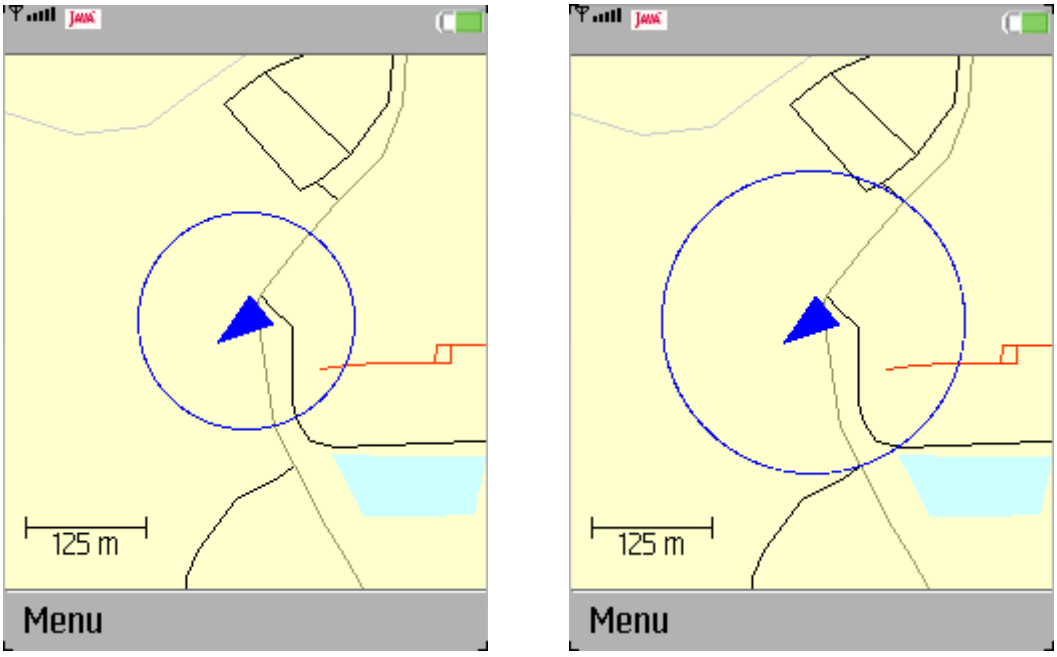

Fig. 7.6: Screenshots of the walking uncertainty increasing with time. The uncertainty decreases again after an update

7.4 Driving uncertainty

The driving uncertainty is far more complex than the walking uncertainty. The walking uncertainty only expands as a circle, while the driving uncertainty spreads along the roads. The speed of the spreading is also not constant. It is dependent on the speed limit of the roads. The previous statements are true only in one of the driving uncertainty modes. The driving uncertainty separates between two different cases. Driving on a road on the map and driving on an uncharted road. The separation is based on the initial uncertainty circle. If it does not intersect any roads or there are no road nodes inside the uncertainty circle, the device is considered to be driving on an uncharted road. Else it is driving on a road that is on the map.

If the driving uncertainty is in uncharted road mode it behaves almost identical to the walking uncertainty. The uncertainty still spreads as a circle with constant speed. The initial uncertainty is identical. The difference is the speed of the spreading. The maximum known speed limit of any road known by the application is used as spreading speed. Currently it is the speed limit on motorways (90 km/h). The reasoning behind this is that the user is moving at a speed which is over the maximum walking speed. Ergo he must be driving on some kind of road. However the map indicates no road within the uncertainty circle. This means he drives on an unknown road. If the road is unknown it may be anything from a residential highway, to a motorway. Since the uncertainty circle should always include the worst case, as long as it is a legal speed, it must take the motorway into consideration. The worst case uncertainty scenario is the following: The user is driving. His estimated GPS position was as wrong as it could possibly be. He is driving in a straight line directly away from the GPS position. The road he is travelling on is an uncharted motorway. This is illustrated in Fig. 7.7.

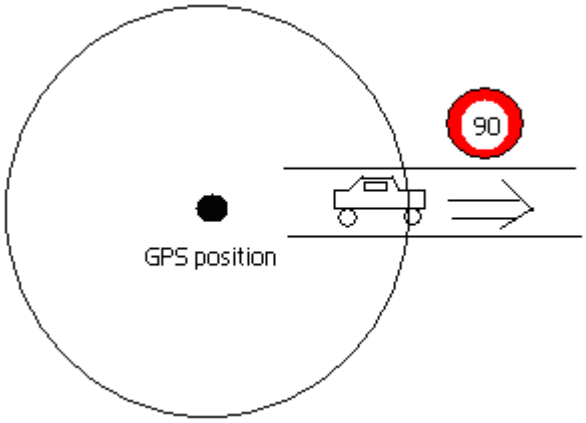

Fig. 7.7: The worst case uncharted road scenario.

To be prepared for this, the application must always use the motorway speed limit as the spreading speed of the uncharted road uncertainty. The other difference from walking uncertainty is that the driving uncertainty has higher uncertainty levels than the walking uncertainty. Because the uncertainty spreads much faster, this could make the user tolerate a little more uncertainty.

The driving uncertainty levels are:

Level 1: 150 m diameter
Level 2: 300 m
Level 3: 800 m
Level 4: 1400 m

The modelling of the other driving uncertainty, when the user is driving on roads which exist on the map, was among the most difficult algorithms to design in the entire application. To illustrate how this uncertainty spreads it is best to give an example. Fig. 7.8 and 7.9 shows the uncertainty spreading across a network of roads.

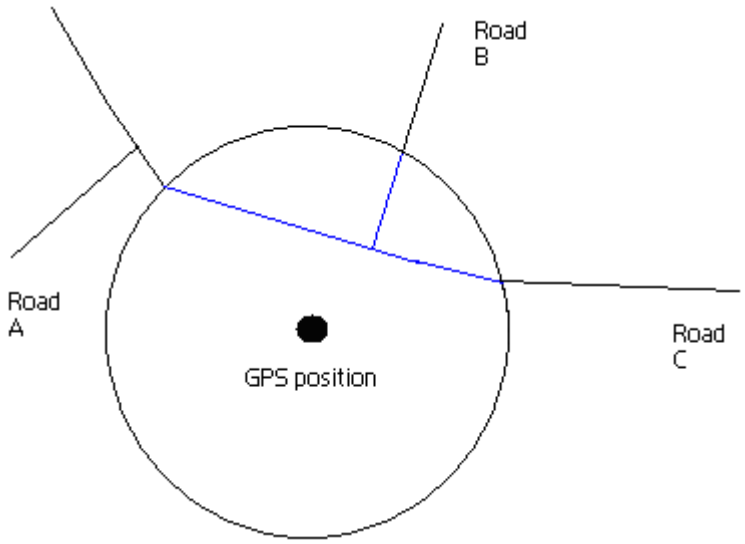

Fig. 7.8: The initial uncertainty at time t_0.

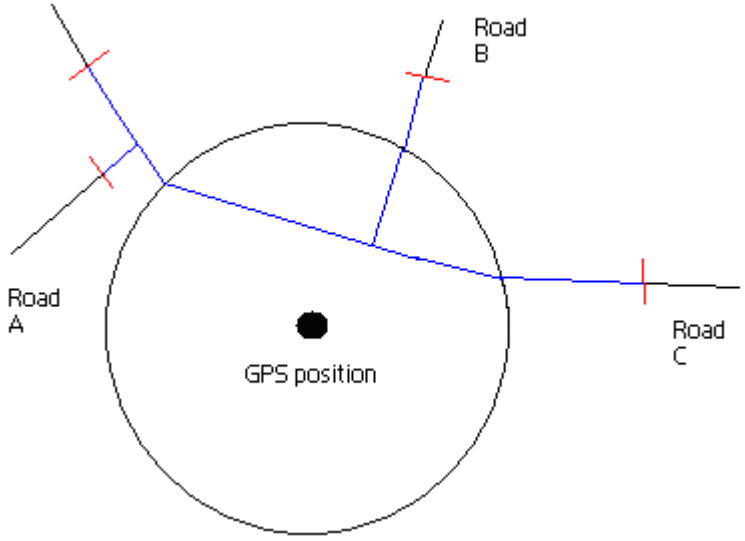

Fig. 7.9: The uncertainty at time t_1.

The speed limit on roads A and B is 30 km/h. The speed limit on the main road, road C, is 60 km/h. This is why the uncertainty has spread twice as long via road C, than road B, from t_0 to t_1. The red lines on fig. 7.9 indicate the uncertainty boundaries. The uncertainty starts initially as all parts of the roads that are inside the GPS uncertainty circle. It then spreads out as fast as it legally can. This means following the speed limit on the roads. It should not spread against the driving direction on a one-way street. This is accomplished using the routing graph as a representation of the roads. It can only spread to neighbours in the routing graph. If it comes to an intersection it should spread out along all connected roads. The driving uncertainty assumes that a user, who is driving, does not start to walk until the next GPS update.

To determine when to update next, the spreading is simulated second by second. The simulation stops, and the time until next update should be set accordingly, if it after a simulation second satisfies one of the two following criteria:

- If the geographical distance between any of the road nodes inside, and including, the uncertainty boundary, is bigger than the diameter declared in the current driving uncertainty level.

- If the number of intersections inside the uncertainty boundaries, are higher than the maximum number of intersections declared in the current driving uncertainty level.

The numbers of intersections in the different uncertainty levels are:

Level 1: 1
Level 2: 3
Level 3: 5
Level 4: 7

Adding the number of intersections into the next update equation makes it possible to demand a higher accuracy in places with many intersections. This makes the Travel Application more accurate when it is needed the most. The levels where chosen using much of the same logic as the walking uncertainty. The lowest level is as accurate as it can get, issuing an update for every intersection the uncertainty comes across. The levels have a steady increase in uncertainty up to the maximum. 7 intersections is a lot when the application's main task is to guide the user to make correct turns.

The algorithm for updating the uncertainty starts by finding the route nodes inside the initial uncertainty circle. This is done by checking the distance between the nodes and the GPS position. Then the initial boundary nodes need to be found. Calculating the intersections between the uncertainty circle and the roads will lead to the discovery of the initial boundary nodes. A boundary node is a subclass of a route node. It has an inside and an outside neighbour. An important note here is that boundary nodes are created and destroyed dynamically. They are not the same as the permanent nodes in the routing graph. When the boundary nodes are found, the expansion starts towards the outside neighbours. Each time an outside neighbour is passed, new boundary nodes are created along each possible path. They will have the most recently passed node as an inside neighbour. The created boundary nodes will be temporary boundary nodes. A temporary boundary node is a boundary node that also has remaining time as a variable. If it is meant to update the uncertainty, this time is initially set to the time that has passed since the uncertainty was last updated. The time remaining is reduced as new temporary boundary nodes are created. This makes it possible to stop the expansion at a given time. The update uncertainty algorithm can be found in appendix A.2

The calculateNextUpdateTime is essentially the same. The only difference is that the time remaining is set to 1 second initially. This makes it possible to check if the uncertainty is too big after the first second. If it is, the application stops simulating and returns 1 second as the time until next update. Else the time remaining is set to 1 second again and the step by step simulation continues.

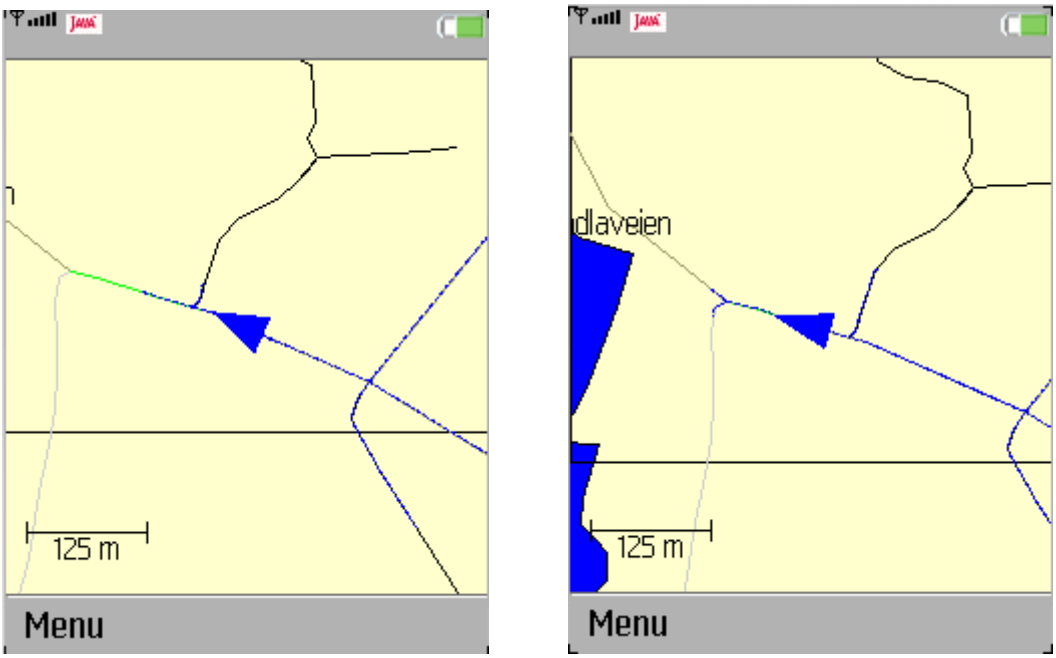

Fig. 7.10: The driving uncertainty expanding.

8 Dead reckoning

Dead reckoning is the process of finding the current position based on previous location measurements and some additional information. This technique was used by sailors to determine the current location of the ship, when they were sailing the ocean without any landmarks in sight. They estimated the course they had held and measured their speed using crude devices. Combining this with the time since their last known certain location, they could find their current position on a map. A modified version of this technique is used in the Travel Application. It is important to give the user the illusion of movement in between GPS updates. Dead reckoning makes it possible to have an acceptable screen update interval, approx. every other second, while still maintaining a long battery life by seldom checking the GPS. In the Travel Application common intervals between the GPS updates lie in the range of 10 seconds to 6 minutes, depending on uncertainty level.

Since it is such an essential part of the Travel Application, dead reckoning is implemented in its own module. It is designed as a finite state machine. This means that the dead reckoning will change behaviour based on environmental input. An overview of the dead reckoning module is shown in figure 8.1:

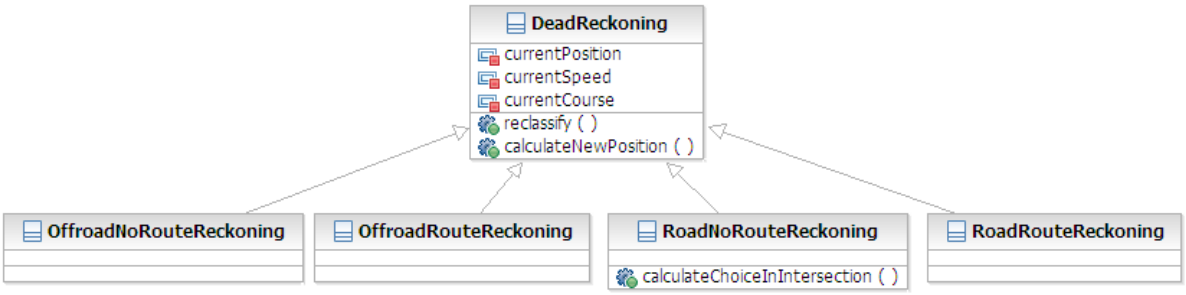

Fig. 8.1: Overview of the dead reckoning package.

When the user is walking, one of the off-road classes does the reckoning. It chooses between the two different off-road options based on whether or not the user has requested routing to a destination. The off-road classes reckon the device continues at the same speed as it had since the last update. The direction will follow the route. If there is no route, the direction is estimated based on the course between the two latest GPS updates.

If the user is considered to be driving, all four reckoning schemes can be utilized. When the user is driving on a road that doesn't exist on the map, one of the off-road classes is chosen. Based on routing requests, the reclassify method again chooses between route and no route. It is important to know that the user can have a route to the destination without being on the road. When calculating the route, the location of the device is inserted as the source route node. The other nodes on the route will always be on the road.

RoadNoRouteReckoning or RoadRouteReckoning is chosen when the user drives on a road that exists on the map. These schemes reckon that the user travels along the road while following the current speed limit. Again the existence of a route determines which one is chosen.

The calculateNewPosition method is the most important method in the dead reckoning package. It has the current time as input. Based on the current time it updates the current position, speed and course. This method has an empty implementation in the DeadReckoning class. All the subclasses have their own version of it. The implementation of this method is in essence the difference between them.

The RoadNoRouteReckoning class has a method that doesn't exist in the other subclasses. CalculateChoiceInIntersection is used to choose between multiple possible paths in intersections. A road is chosen by analyzing data collected from the phone's accelerometer.

Reclassify is the bulk of the DeadReckoning class. It is called by the GUI's run method. However it is only called after the start up phase is finished, because the guessing is not needed until the GPS update intervals start increasing. Reclassify is only invoked in the first run loop after each GPS update. The method's task is to choose which reckoning class to use until the next update. The input it needs is:

- **The current uncertainty type.** Is the user driving or walking? Is he on an uncharted road, or is it possible to take advantage of the application's knowledge of the road network?
- **The current route.** If there is a shortest route calculated, there is reason to believe that the user could be following this route.
- **The previous route.** This is used to see whether the user is closer to the intended target now, than at the previous GPS update time. If not, it is an indication that he is ignoring the routing information. Another use for the previous route is to determine if the routing target has changed. This most likely means that the user has changed the destination manually. He may then be more likely to follow the newly calculated route.

The algorithm for the reclassify method is listed in appendix A.3

8.1 The OffroadNoRouteReckoning implementation

The OffroadNoRouteReckoning class has the simplest guessing scheme. It is constructed for a situation where there is no route or roads to follow. It calculates the current speed based on the distance and time between the two latest GPS updates. Just after the start up phase it uses the weighted average speed instead. The class also uses the last two updates to estimate the course of travel. It maintains constant speed and course until the next update. The following equations must be solved to find the current position:

Current longitude = previous longitude + ((time since previous update * current speed * cos(current course))/ meter per longitude)

Current latitude = previous latitude + ((time since previous update * current speed * sin(current course))/ meter per latitude)

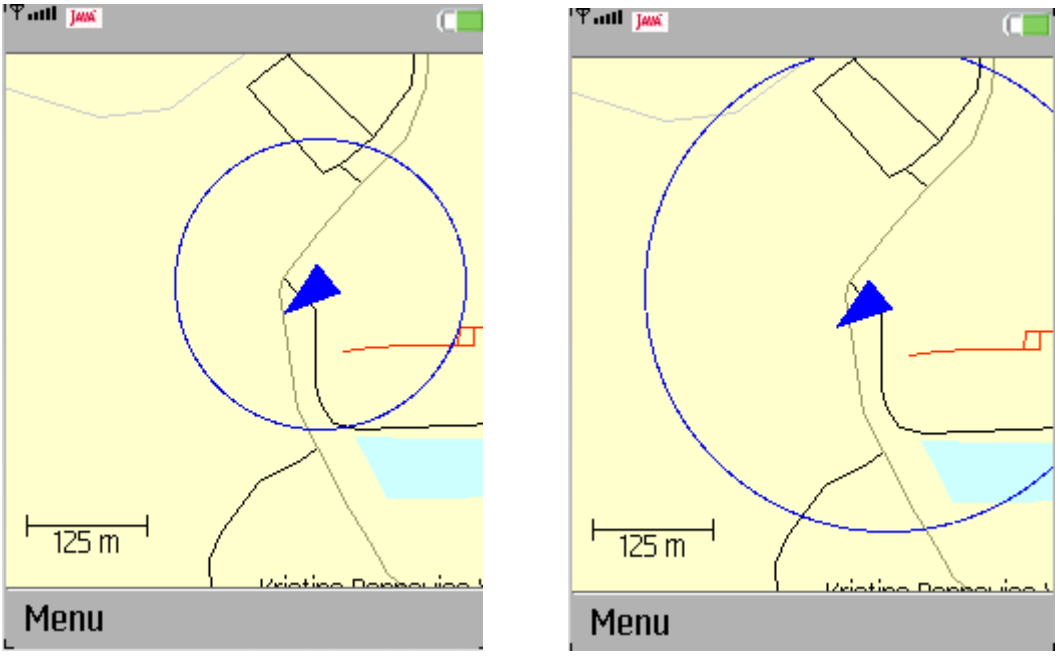

Fig. 8.2: Screenshots of the OffroadNoRouteReckoning. Notice how it travels in a straight line along the arrow's direction.

8.2 The OffroadRouteReckoning implementation

OffroadRouteReckoning is intended for situations where the user is considered to be following a route, but is not influenced by any known speed limits. This could be because he is walking or because he is driving on an uncharted road. It uses the current route and speed to determine the updated position. Its estimate is based on holding a constant speed along the calculated route. The speed is still determined via the two latest GPS updates. Since the position traverses along the route, the position update algorithm can borrow heavily from the road spreading of the driving uncertainty. It is actually somewhat simpler, since there is only one possible road to take at each intersection. A new feature in the reckoning solution is the TempPosition. It has the same role as the temporary boundary node in the driving uncertainty, without the inside neighbour. The algorithm is listed in appendix A.4.

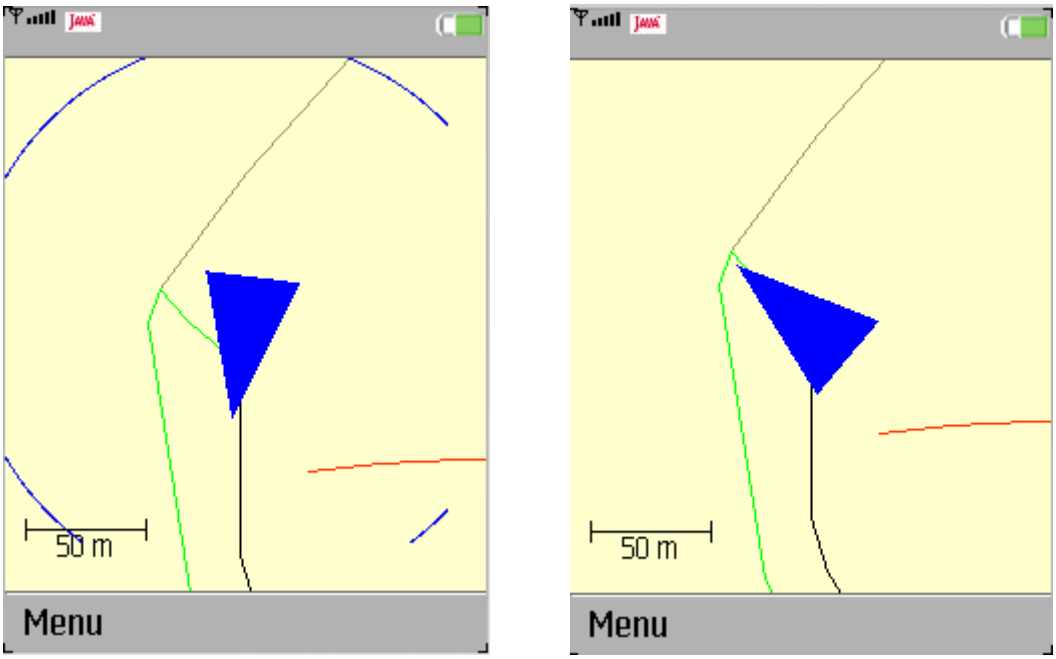

Fig. 8.3: Screenshots of the OffroadRouteReckoning. The arrow turns around the corner in the route.

8.3 The RoadRouteReckoning implementation

The road route reckoning scheme is only used when the user is driving on a road. It follows the current route. This makes the position update algorithm very similar to the OffroadRouteReckoning. The big difference is that the current speed is dynamic. It is determined by the different speed limits along the route. This will give a good estimate on the current position during normal traffic. If the driver either drives too fast or too slow, it could be a bit deceiving. When there is a traffic jam, the traffic will a lot of the time move so slow that it will be classified as walking. The Travel Application will then use the OffroadRouteReckoning and the speed will be determined as low.

The RoadRouteReckoning has a snap to road feature. The OffroadRouteReckoning use the last GPS position as a starting position. The RoadRouteReckoning instead finds the point on the road that is closest to the GPS position. This is then used as the initial position.

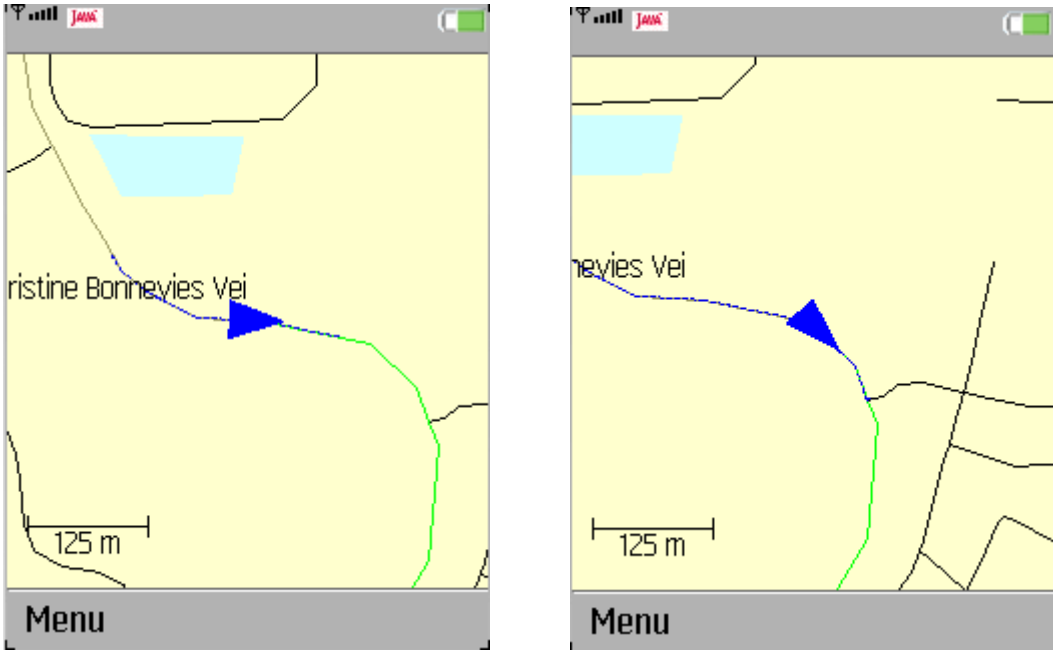

Fig. 8.4: Screenshots of the RoadRouteReckoning. Notice how the starting position is not in the middle of the uncertainty due to the snap to road feature.

8.4 The RoadNoRouteReckoning implementation

RoadNoRouteReckoning has the most complex implementation of the calculateNewPosition method. It is meant for situations where the user is driving on the road, but either has no route or shows no signs of following the route he has been given. It has a couple of similarities with RoadRouteReckoning. The same snap to road feature is implemented in both classes. RoadNoRouteReckoning also use the speed limit on the current road section to determine the movement rate. The big difference appears in how they handle intersections and how they decide the initial course.

The initial course is easy to decide when the user is following a route. It is the angle from the position where the user is snapped to the road, to the next road node on the route. The problem for RoadNoRouteReckoning is that there exists no valid route. The issue is then to choose which road node should be considered the next node. RoadNoRouteReckoning's solution to the problem is drawn in figure 8.5

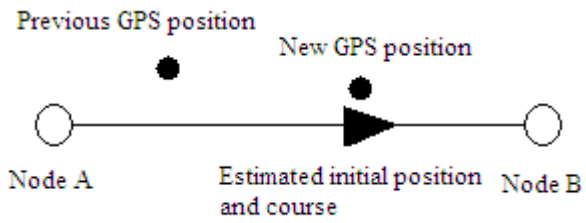

Fig. 8.5: Determining the initial course without a route.

To choose between node A and B, the Travel Application calculates the course from A to B and from B to A. The closest angle, to the course between the GPS positions, is chosen as initial course. It is sort of like an angular snap to road function. In figure 8.5 Node B has been chosen as the next node.

The RoadRouteReckoning implementation does not even notice when it arrives an intersection. It only follows the calculated route. The RoadNoRouteReckoning has to make a choice. Should it make a left, right or maybe even a u-turn? If it is a major intersection, there could also be further possibilities. The first version of the algorithm used speed limits to decide which road to take in an intersection. It always continued moving along the road with the highest speed limit. The reckoning algorithm also kept note of the last node it had visited, to avoid u-turns. This is a very crude solution. A typical car journey starts out at a residential highway, continues to a secondary highway, follows along a primary highway or motorway, then goes back on a secondary highway, just to end up at a residential highway. This means that the algorithm will work well for the first part of the trip. The second half however, will have many wrong choices.

8.5 Intersection road choosing using the accelerometer

The W760i phone has an accelerometer. In the improved version of the RoadNoRouteReckoning the data collected from the accelerometer is used to decide which road to choose in intersections. The accelerometer measures acceleration in 3 dimensions. The X-axis goes from left to right across the screen. The Y-axis runs from the bottom to the top of the screen. The Z-axis is directed out from the screen. The direction of the axes is shown in fig. 8.6.

Fig. 8.6: The W760i accelerometer axes.

The acceleration is measured in G or gravitational units. If an axis returns the value 1000, this means that it is being exposed to a positive acceleration of 9.81 m/s^2, or 1 G. When the phone is not moving, the vertical axis will always measure an upwards acceleration of 1 G. This is because it measures force and a certain force is needed to keep the accelerometer in place. If the phone is free falling towards the earth, without any wind drag, the accelerometer

would show zero vertical acceleration. This has a positive side effect. It makes it very easy to find the vertical axis. It is just to look for the axis that measure closest to ± 1 G.

The accelerometer only measures acceleration. The Travel Application needs to transform the measured acceleration into change in position. The angle from the intersection to the new position can then be estimated. The road with angle from the intersection, closest to the estimated angle, is chosen by the application. The current position will then continue down this road.

For each time the calculateNewPosition method is called, the time until next intersection is calculated. This is not 100% accurate, as both the current position and the remaining time until the next intersection is estimated based on the current road's speed limit. If the time is below 5 seconds, the accelerometer data collector is started. It collects data until 5 seconds after passing the intersection. The accelerometer in the W760i has an update frequency of 20 Hz. This means that after 10 seconds the data collector will have approximately 200 data readings, per axis, from the accelerometer. The collector transforms the acceleration data into m/s^2 using the following formula:

Acc [m/s^2] = (input Acc[1000G] / 1000)*G

The application uses the J2ME Mobile Sensor API to read the accelerometer. It registers itself as a data listener with the accelerometer's sensor connection. The collector implements the DataListener interface and receives the data, 20 times per second, through the dataReceived-method.

When the current position of the reckoning class reaches the intersection, it waits there until the data collector is finished. After the collector is finished, and a road has been chosen, the current position is moved t seconds down that road with the current speed set to the speed limit. t = the number of seconds the current position has been standing still at the intersection. The process is described in fig 8.7:

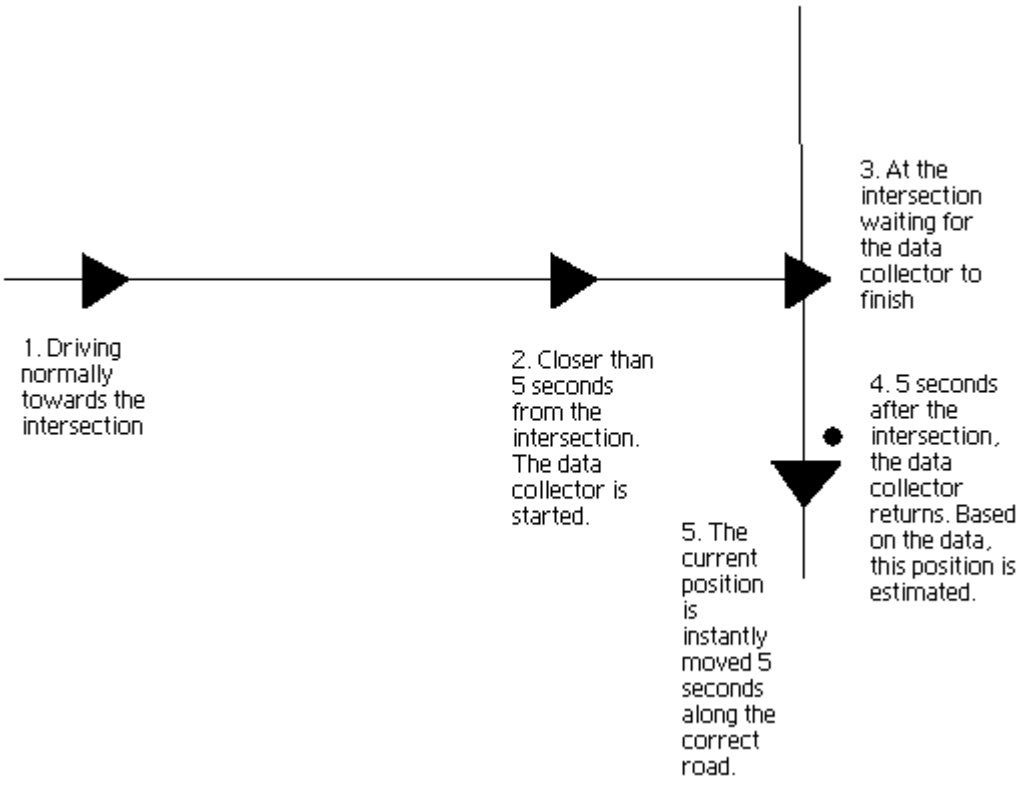

Fig. 8.7: The RoadNoRouteReckoning intersection process.

When the data has been collected, the problem is how to estimate the current position based on the previous position and the data from the accelerometer? To be able to calculate the new position one must first make some assumptions. One must assume the initial speed and course. In the Travel Application the initial speed is set to the speed limit of the road before the intersection. The initial course is assumed to be the course the user was estimated to have at the start of the data collection. The Travel Application assumes that one of the axes is approximately vertical. The more askew the phone is, the worse the guessing will be. The fourth assumption is that the traffic flows steady through the intersection. If the user must stop in a traffic jam, he will use too much time reaching the intersection and the data collecting will be over before he has had a chance to make the turn. The final assumption is that the user brakes before the intersection and accelerates afterwards. With these assumptions one can start to find the role of each axis. There are three possible roles:

- **The vertical axis.** This axis is largely ignored in the calculation of the new position. Its most important function is that if it is negative, the other axes will be flipped. Flipped means that an acceleration that would normally be regarded as a left, is considered to be a right. The same applies to acceleration and deceleration in the speed direction.
- **The tangential axis or the speed axis.** This determines the tangential acceleration of the device. If it has a positive acceleration, it is considered an increase in speed.
- **The centripetal axis or the angle axis.** This is connected to the angular acceleration. A positive acceleration means a left hand turn. It is not directly connected however. The formula for determining the angular acceleration given the centripetal acceleration is:

  $a_{angular} = (a_{centripetal} * a_{tangential})/(v_{tangential}^2)$

Figure 8.8 shows the acceleration axes of the cell phone in a turn.

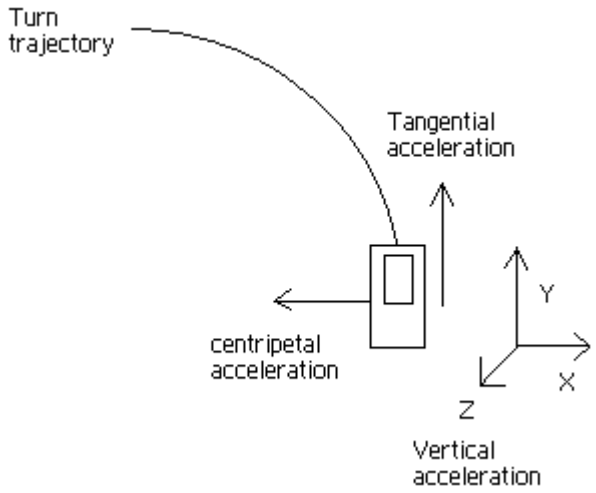

Fig. 8.8: The different acceleration axes during a left hand turn.

One must first find the vertical axis **t**o find which of the accelerometer axes is tangential, centripetal or vertical. Since none of the other axes will be close to 9.81 m/s^2, this is quite easy to distinguish. The highest acceleration measured by the other axes during testing were about 3 m/s^2.

Now there are two axes left. To find the tangential axis, the sum of the acceleration's absolute value, before and after the intersection, is calculated. The axis with the highest sum is chosen to be the tangential axis. The acceleration data that is recorded during the interval [calculated intersection time – 2 seconds, calculated intersection time + 2 seconds] is overlooked. This is because in this period the user might be turning around the corner and the centripetal acceleration will most likely be larger than the tangential. The tangential axis is considered to be flipped if the sum of acceleration data, not the absolute value, before the intersection is higher than after. This is because acceleration before the intersection and braking afterwards is unnatural.

With only one axis to choose from, the last axis has to be the centripetal axis. The Travel Application must know if the axis is flipped. This helps to avoid the situation where a left hand turn would be interpreted as a right hand one, just because the phone was held upside down. The following process is used to determine if the centripetal axis is flipped:

An analysis of which axes are vertical and tangential determines the initial flipped status. This is because of the way the phone axes are defined.

- Z-vertical, Y-tangential: X-axis flipped.
- Y-vertical, Z-tangential: X-axis not flipped.
- X-vertical, Z-tangential: Y-axis flipped.
- Z-vertical, X-tangential: Y-axis not flipped.
- Y-vertical, X-tangential: Z-axis flipped.
- X-vertical, Y-tangential: Z-axis not flipped.
- If the vertical axis is flipped, the flipped status is inverted.
- If the centripetal axis is flipped, the flipped status is inverted again.

With all the axes correctly flipped and ready, it is time to start calculating the current position. This is done in steps. Each accelerometer data reading is one step. The angular and tangential acceleration is handled separately. The acceleration is considered to be constant between each step. This means that the standard physical formulas for motion with constant acceleration can be applied. The angular speed is set to 0 initially. The algorithm used to calculate the position after the intersection is listed in appendix A.5.

After the application has calculated the position, the angle from the intersection, to all its neighbour nodes, is estimated. Since the Travel Application uses the neighbours from the routing graph, it is impossible to choose to drive against a one-way street. The angle from the intersection to the calculated position is called α. The neighbour that has angle from the intersection closest to α is chosen. The current position is then moved accordingly down that neighbour's road.

There are of course many sources of error when using this method. One could be stuck in a traffic jam, turn to change position in the roadway before the intersection or take the corner without braking. Due to lack of a vehicle to test it with, there were not done many tests to see if the method guessed correctly. The ones that were done seemed to be working ok, although it did not guess right every time. When it guessed wrong, it was mainly because the axes were wrongly picked. This was in turn because the corners could be taken almost at full speed. When taking the corner at full speed, there is not a lot of tangential acceleration before and after the intersection. This causes high probability of choosing the centripetal axis as the tangential axis.

Using the accelerometer has one clear advantage over using GPS every time one has passed an intersection. It does not need a GPS signal to work. It can be used in underground road systems and in narrow city streets with poor GPS conditions. Another possible advantage is the power consumption. In the test chapter this is examined more closely.

Please note that it is not possible to only use the accelerometer for dead reckoning. This is because of all the assumptions that have to be made to find the correct axes. For instance to find the tangential axis, the dead reckoning algorithm assumes that the user is braking before an intersection. If he is not near an intersection this becomes impossible. Another problem arises with the fact that the accelerometer does not register rotation around its own axes. So even if we have made all the right assumptions at the beginning, if the user flips his phone all the assumptions will be wrong. Also when using the acceleration to determine the current position, the uncertainty becomes squared because the values have to be integrated twice. This is not good in the long run, but it would probably work well enough for the short time between updates.

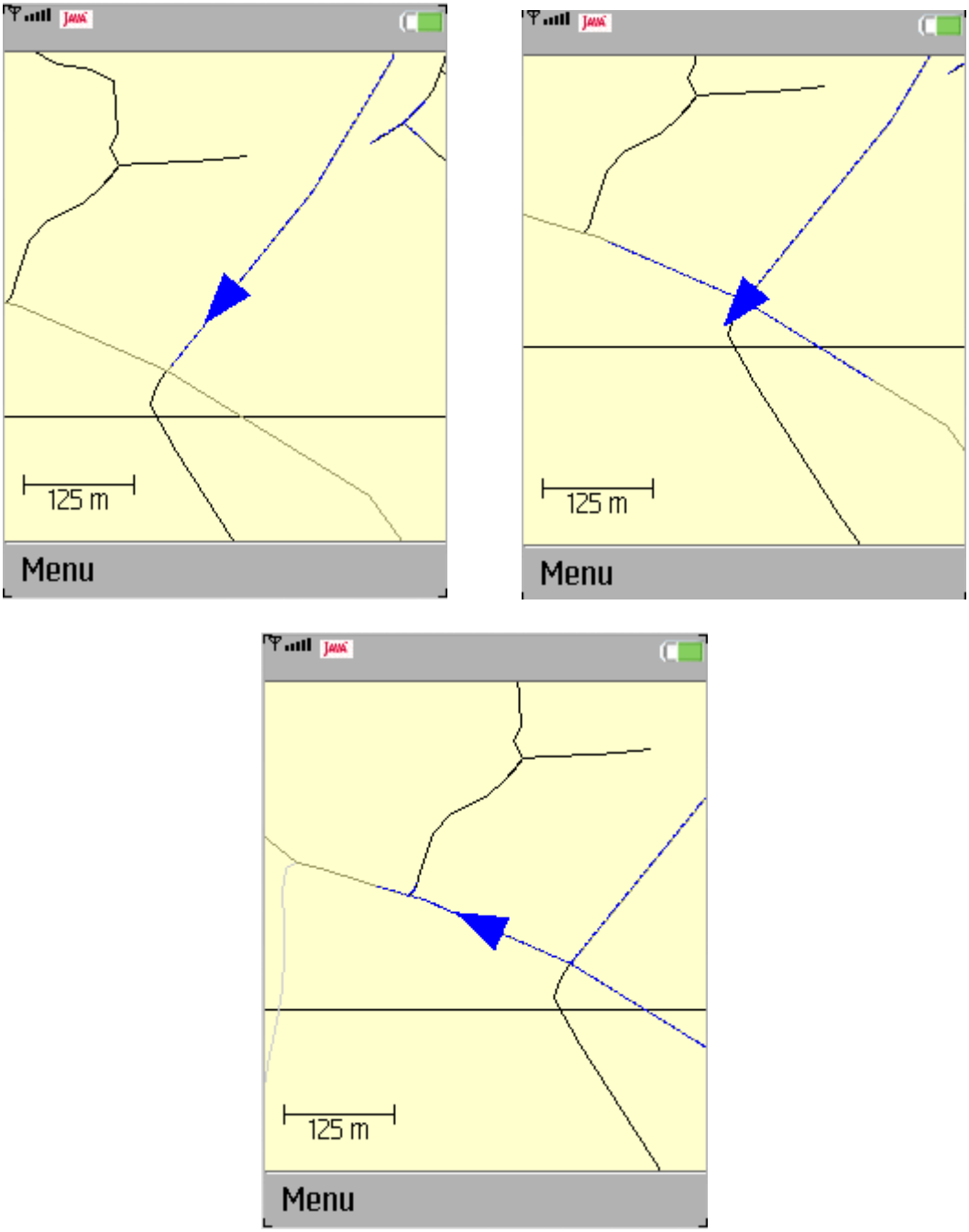

Fig. 8.9: Screenshots of RoadNoRouteReckoning at an intersection.

9 Tests

The main reason behind the methods implemented in the Travel Application is to extend battery life. To see if the theories worked in real life, a series of battery life tests were performed. Some of them were done on the Travel Application. Others were run on specially written programs, to test specific hypotheses.

9.1 Test setup

A two months old Sony Ericsson W760i phone was used in all the tests. They were all performed in the following manner:

- The phone's battery was completely discharged.
- The battery was charged for exactly 3 minutes.
- The phone was turned on. The phone vibrates straight afterwards. The time was measured from this event.
- If an application was tested, it was started as soon as possible.
- The phone was not moved during the tests.
- The time was measured until the phone turned itself off.
- All tests were repeated 10 times.

Since being connected to a 3G network drains the battery faster than a regular GSM network, this feature was disabled. This could otherwise have been a factor, since not all the test locations had 3G coverage. If any special events occurred, for instance the reception of an SMS, the result was discarded. The phone has a flight mode where the GSM network is turned off. This could have prevented these disturbances. However it was not applicable to these tests. This was due to the fact that the flight mode turns off all external communications. It means that the GPS module won't work, making it impossible to perform most of the tests.

9.2 Box plots

Box plots are graphs that give a good indication of central tendency and variation in data sets. They do not require the data to follow any known distribution. This is great for the battery life tests, since the distribution of the results are hard to identify. For instance the standby time had a tendency to only turn itself off at the different hour marks. If it had passed 7 hours, it almost always lasted until 8 hours had passed. This was because each hour the low battery message was displayed. Since this took a lot of energy (the phone lit up, vibrated and made a sound), the low battery message was almost always what pushed the phone over the shut down edge. This was analogue to what was experienced with the GPS tests. Here the shut downs mostly occurred right after a GPS update.

Box plots are also great for spotting outliers among the data samples, detecting skewness and comparing results.

An example of a box plot can be seen in figure 9.1. The box plots in this chapter were constructed using MATLAB. The features included in a box plot may vary, but here is what MATLAB includes [22]:

- The top of the box is the 75^{th} percentile or 3^{rd} quartile.
- The bottom end of the box is the 25^{th} percentile or 1^{st} quartile.

- The red line in the middle is the median
- The lines extending from the top and bottom of the box are called whiskers. The maximum length of the whiskers is 1.5 times the distance from the 1st to the 3rd quartile. However they never extend beyond the most extreme samples.
- Any samples outside the whiskers are considered outliers and marked with a red +.
- A 95% confidence interval for the median is given by the notches inside the box. Steep notches indicate an accurate median measurement.

9.3 Testing the standby time

The first tests measured the standby time. The phone was turned on and left untouched until it switched itself off. The most important feature of this test was to establish if 3 minutes of battery charging were enough. The test result is displayed in figure 9.1.

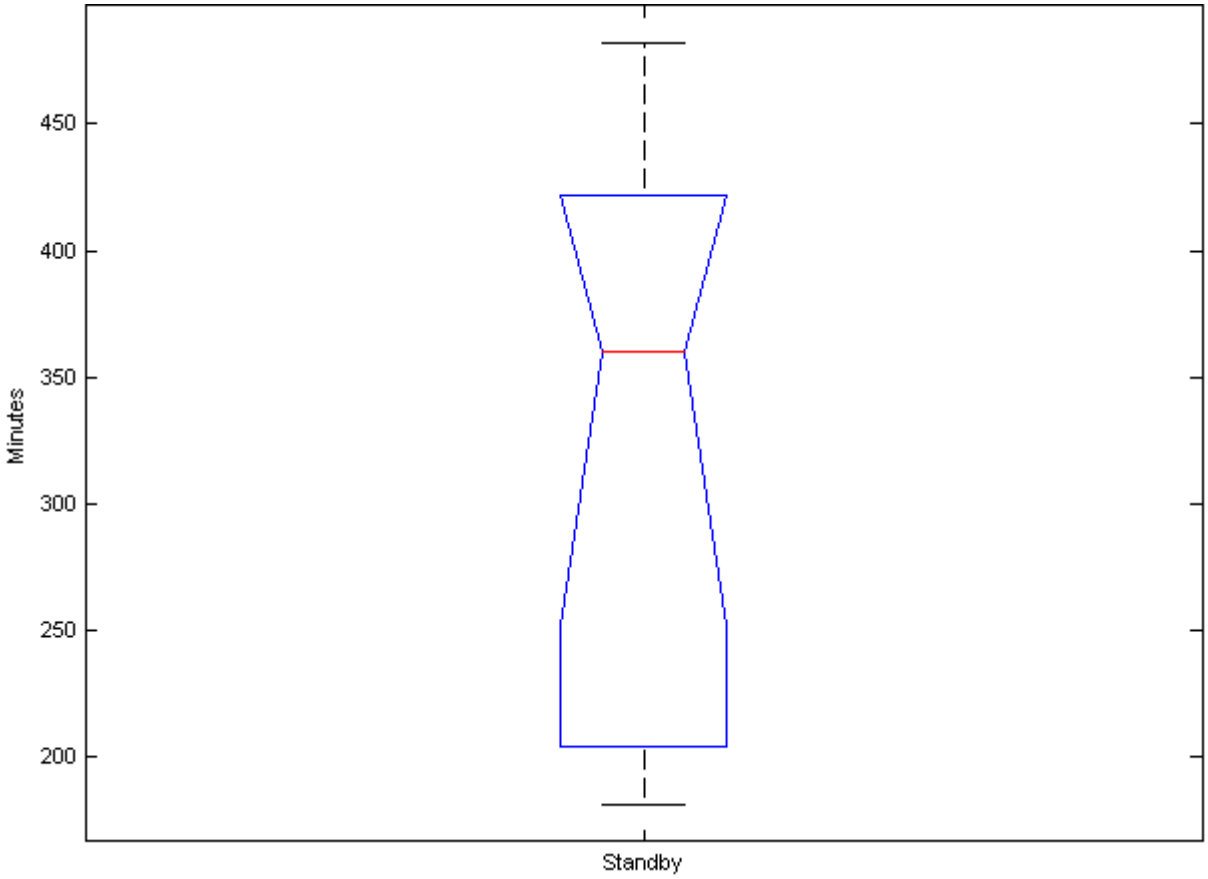

Fig. 9.1: Box plot of the standby time

The high degree of variance is due to the fact that the tests were performed at several locations. The best results were in areas with very good network signal strength. This is because the phone turns up the transmit power in areas with poor coverage. Most of the standby tests were done early, but a couple of them were performed at the end of the testing phase. The later tests showed similar results to the earlier ones. This gives an indication that the testing had not caused any significant battery wear.

The data set's mean value was 332 minutes or 5:32 hours. With such a long standby time, it is clear that 3 minutes of charging should be enough to get valid results.

9.4 Testing the Travel Application

9.4.1 Measuring the effect of altering the uncertainty level

5 different tests were performed to measure the difference in battery life when changing the uncertainty level. 4 of them were performed using the standard Travel Application, but at different uncertainty levels. The last one was performed with a slightly modified version. The modification was that the start up phase never ended. Normally the start up phase ends after 30 GPS updates. This means that there was no reckoning involved in the modified version. It received GPS updates every other second. In all other aspects, screen update intervals, etc., it was identical to the other version. All tests were performed on the University of Stavanger with routing to the nearby road Grannesvegen. The results are shown in figure 9.2.

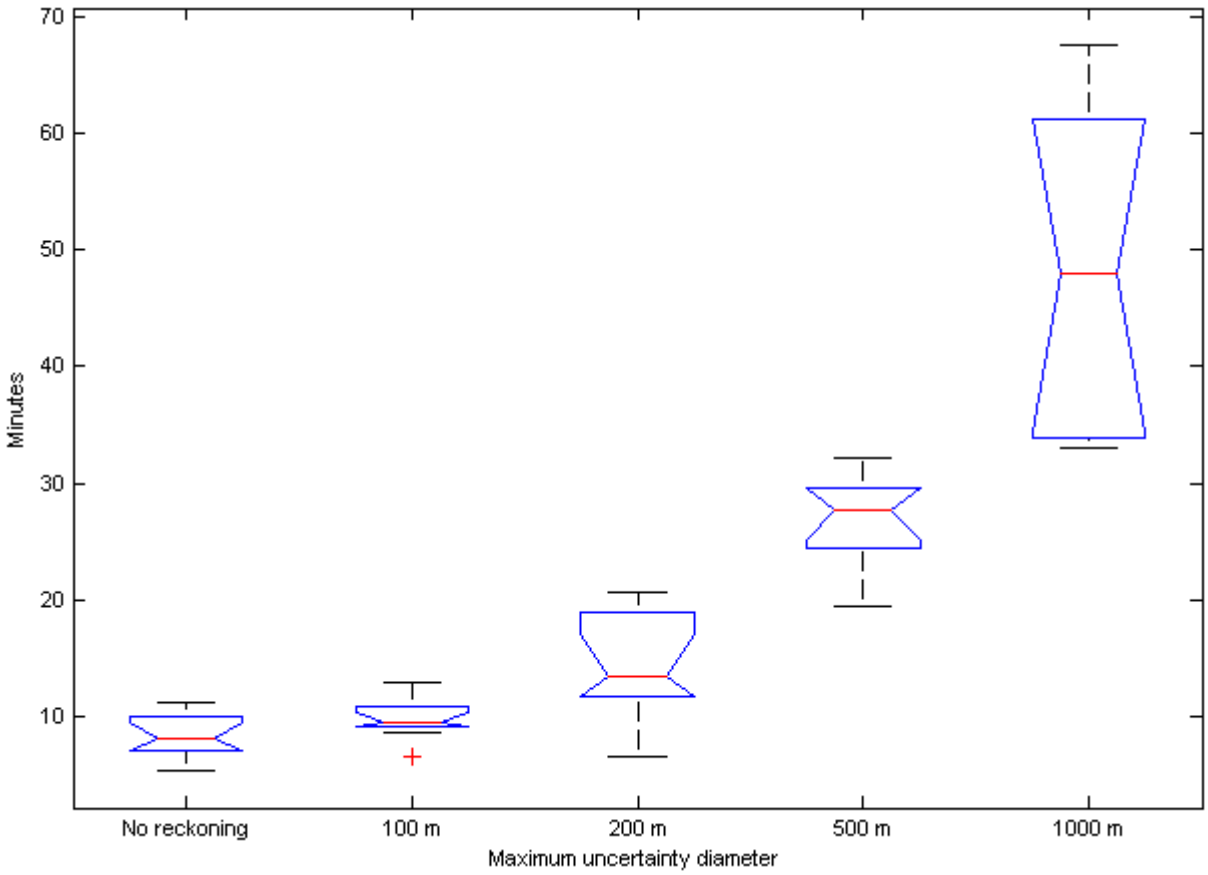

Fig. 9.2: Battery life at different uncertainty levels.

The means are shown in table 9.1. They are shown along with the increase, in percent, from the battery life measured when not using dead reckoning. The increase from the previous uncertainty level is also shown.

Maximum uncertainty	Mean battery life [minutes]	Increase from no reckoning	Increase from previous level
No reckoning	8:22	0 %	N/A
100 m	9:47	16.8 %	16.8 %
200 m	14:01	67.5 %	43.3 %
500 m	27:01	223.0 %	92.8 %
1000 m	47:48	471.3 %	76.9%

Table 9.1: Comparison of the mean battery life for different uncertainty levels.

The results show a clear increase in battery life when more uncertainty is tolerated. This becomes even clearer when the start up phase is removed. It took at least 90 seconds, from the phone was turned on, until it got the first GPS update. It then received 30 updates with 2 seconds intervals. This period of 2 minutes and 30 seconds is identical for all the tests. It is only after this period the application starts to calculate update intervals, and use dead reckoning. In the following table the start up period has been omitted:

Maximum uncertainty	Mean battery life [minutes]	Increase from no reckoning	Increase from previous level
No reckoning	5:52	0 %	N/A
100 m	7:17	24.0 %	24.0 %
200 m	11:31	96.2 %	58.2 %
500 m	24:31	318.0 %	113.0 %
1000 m	45:18	672.2 %	84.8 %

Table 9.2: Comparison of the mean battery life for different uncertainty levels. The start up phase has been removed.

The difference from no reckoning to the other tests would have been bigger if the tests had all been done in perfect GPS conditions. Some were done when the weather was poor, which made receiving a GPS signal harder. Others had poor satellite geometry, which resulted in greater initial uncertainty and therefore shorter update intervals. When testing with 1000 m uncertainty the phone displayed an extra, low battery warning. This caused it to turn itself off after about an hour. Had this warning not been displayed, the 1000 m uncertainty level would have had a longer mean battery life.

Applications with different requirements should use different uncertainty levels. For instance when a user is on a hiking trip, charging the battery can be difficult. This means that, to preserve battery power, a hiking application should use a high uncertainty level, for example 500 meters. Especially since an approximate position might be considered more than enough in large alpine areas. Since power outlets are readily available, having up to date position information is more important than preserving battery life in an urban routing application. In that case a more appropriate uncertainty level lies somewhere below 200 meters.

9.4.2 Comparing the two different routing implementations

Two different routing implementations were developed for the Travel Application. The first one made a simplification by just routing to the geographically closest node on the destination street. This gives the shortest route in most cases. However, as explained in the routing chapter, there are some exceptions. That was why another routing implementation was designed. It calculated the shortest route to any of the possible destination nodes on the street.

This implementation always calculates the correct shortest route. But it comes at a price. It requires more processing power, which in turn means shorter battery life. The two versions were compared, to see if the difference would be large enough to make a significant impact in real life.

The tests in section 9.4.1 were performed with the routing implementation that always returns the correct route. To be able to compare results, one more test application was developed. It was identical to the Travel Application, but it used the other routing implementation. This was then tested with all the other parameters identical to the previous tests. The test still had Grannesvegen as the routing destination, a road with 10 nodes and an intersection at each end. The maximum uncertainty was set to 1000 m. This was because the higher the uncertainty level, the more effect has the routing algorithm. Longer update intervals means the routing algorithm is run more times per GPS update. The routing is done every other second at all uncertainty levels. The comparison between the routing implementations is shown in figure 9.3.

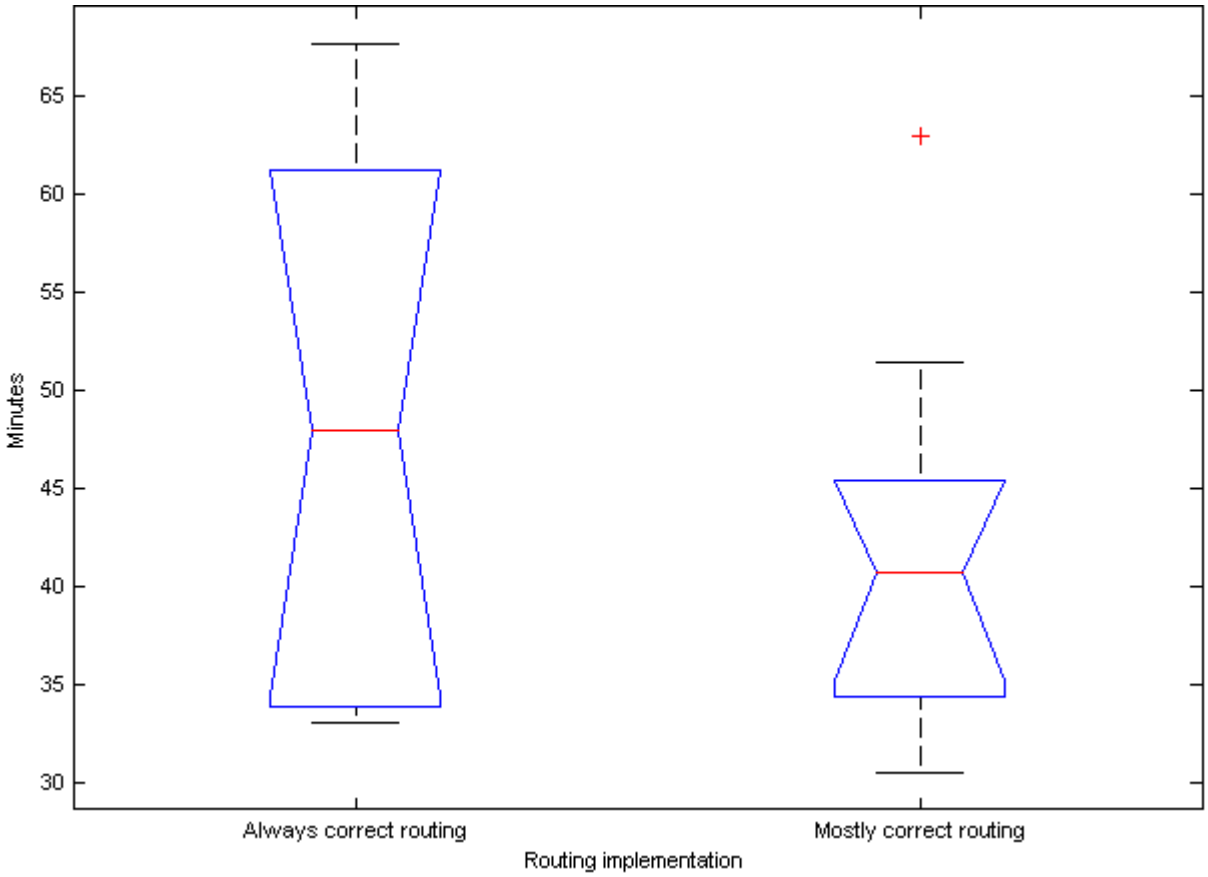

Fig. 9.3: Comparison between the two routing implementations.

As the figure shows, the mostly correct routing gives no significant advantage in battery life over the always correct routing. The battery life is actually lower, but this is probably just a coincidence. The mostly correct routing's median is inside the 95 % confidence interval for the always correct routing's median. Because of the high uncertainty, more tests are needed to get a statistically significant result. The means are 47:48 for always correct and 41:53 for mostly correct. The mostly correct routing could have beaten the always correct, had the destination street been further away from the source, or if the destination street had included more intersections. It would probably still be very close, since the GPS updates stands for the bulk of the power consumption. Since the difference in battery life is so small, it is definitely worth using the always correct routing.

9.5 GPS listener versus get location

The J2ME Location API supports two ways of getting location updates. If one sets up a listener, it will receive updates at a specified rate. It is also possible to get new location updates on a per request basis. The Travel Application uses the listener method. This method was chosen because it is more reliable, since the GPS implementation knows it shall deliver a GPS location at a certain time in the future. The GPS can then begin getting an update in advance. This makes it easier for the GPS to deliver updates on time. If the GPS is called on a per request basis, it has no knowledge of the request prior to the actual method call. There are two problems with using a listener for the Travel Application.

- The next update time has to be calculated after a GPS update. This means anything from a quick calculation for the walking uncertainty, to an extensive simulation when the uncertainty spreads along the roads. This could be skipped if get location was used instead of a listener. The update uncertainty method could then check if the current uncertainty was higher than the current maximum uncertainty level. If so, it would issue a GPS update request. Since these requests can take some time to process, the maximum uncertainty levels would need to be lowered accordingly.
- The biggest problem is the fact that one gets a double call when setting the update interval. The implementation returns a GPS update instantly after the interval has been set. It then waits the specified time before returning a new update. This causes half of the GPS requests to be unnecessary. The second double call is just ignored in the Travel Application. It could have been used together with the first to decrease the uncertainty. Since the second call happens instantly after the first, the combined uncertainty would be:

  $$\text{First call} \cap \text{Last call}.$$

  This is shown in fig. 9.4. It could however not be used successfully to extend the update interval. This is because setting the update interval again would result in just another instant call. The impact of this extra call on battery life would be greater than the benefit of a somewhat longer update interval.

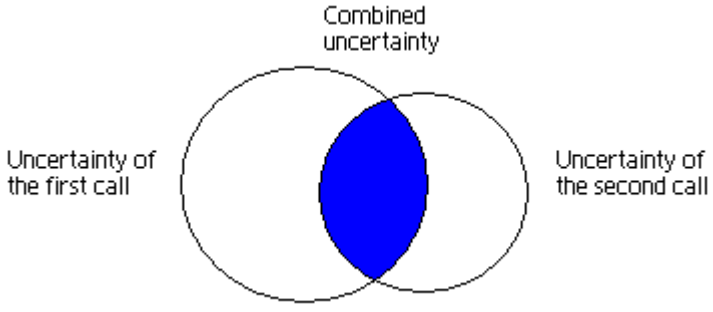

Fig. 9.4: The effect of a double call on initial uncertainty.

Timing is a big problem when using requests instead of a listener. The problem increases with higher uncertainty levels. If the device has lost the signal to some of the satellites it used for the previous request, it needs to find replacements. This could take more than 1 minute. The requests have no guarantee for when they will return a location. Ergo we would often see an uncertainty higher than the maximum uncertainty level (unless we use a large buffer, which is not desirable, since it would cause shorter update intervals). For instance consider a location

request being issued when the uncertainty exceeds 60 meters. Sometimes the uncertainty diameter when the update request is finished would be 140 meters, but if the request is processed faster it might be 70 meters.

Two test applications were made to examine the effect of the listener's double call on battery life. One was a listener which had a 30 second update interval. Every time it received an update that was not a double call, it would set the listener's update interval to 30 seconds again. This provoked the double call. When the second call was received it did nothing. This behaviour was performed repeatedly until the phone's battery was empty. The other test application requested a GPS update through the location API's get-method. The application's main thread then slept for 30 seconds before issuing another request. This process was repeated until the phone turned itself off. In figure 9.5 the test results are compared.

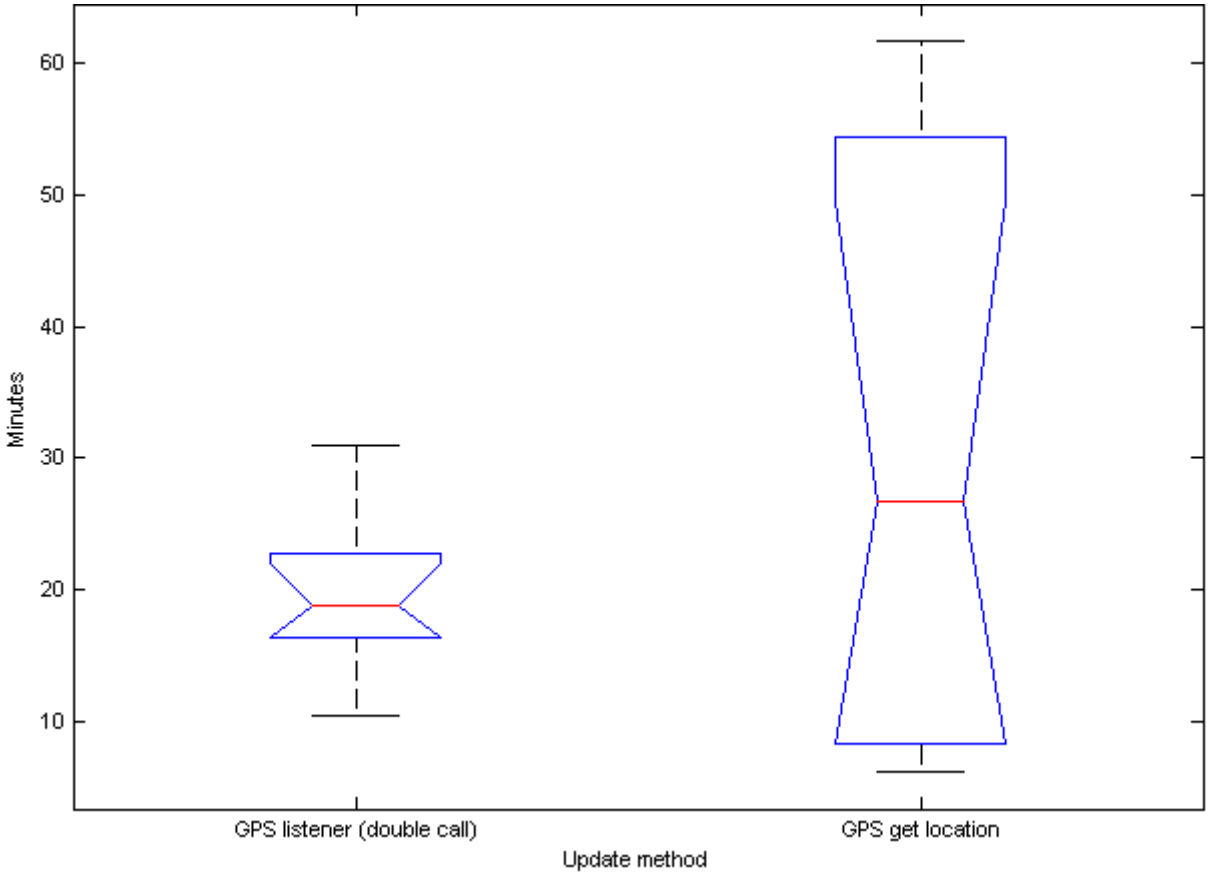

Fig. 9.5: GPS listener versus GPS get location.

Due to the extreme variance of GPS get location, it is difficult to conclude which is the best update method. GPS get had results ranging from 6 to 61 minutes. GPS listener ranged from 10 to 30 minutes. The mean of the listener is 19:32 and the getter is 30:55. The median of the GPS listener is within the 95% confidence interval of GPS get.

By breaking down the test results based on GPS signal condition, the results gets more interesting. GPS get had poor conditions 5 times. The results then ranged from 6-10 minutes. The rest of the time the conditions were good and the results ranged from 47-61 minutes. The GPS listener had poor conditions 6 times, with corresponding results from 10-18 minutes. Under good conditions 4 tests were made, with results in the range of 21-30 minutes. The listener seems to outperform the getter during poor conditions. However during good condition the getter has a longer battery life. Two features may explain this behaviour.

- During perfect conditions it is no problem to get a GPS signal. The double call effect gives the GPS listener a clear disadvantage over the getter. This is illustrated in the highest test results for the two tests. The getter's 61 minutes is twice as long as the listener's 30.
- During poor signal conditions the GPS must struggle to find satellites. More of this struggle has to be repeated by the getter. This is because it does not know when the next request will be issued. This gives the listener an edge in poor conditions.

Poor conditions can be measured in an application via two variables. The time it takes to get the first GPS fix and the uncertainty radius related to the GPS position. Further testing is needed to find the condition threshold where the GPS listener uses the same amount of power as the getter. If this threshold is found, it can be implemented into applications. When the conditions are worse than the threshold, the listener could be used. When the threshold is passed, the update method could be dynamically switched.

9.6 Power consumption of accelerometer and GPS

The accelerometer is used in the Travel Application to determine which way the user has chosen at an intersection. This could also have been performed by issuing an extra GPS update every time the user had passed an intersection. Two different test applications were made to inspect if using the accelerometer could result in a longer battery life. The first application was constantly listening to the accelerometer at a rate of 20 Hz. It did nothing when it received data. The other application was a GPS listener with 10 seconds between each update. This application had an empty locationUpdated method. 10 seconds was chosen since that is approximately how long the accelerometer is used at each intersection. This test tries to determine which method leads to the longest battery life when faced with constant, back to back, intersections. In figure 9.4 the results are displayed.

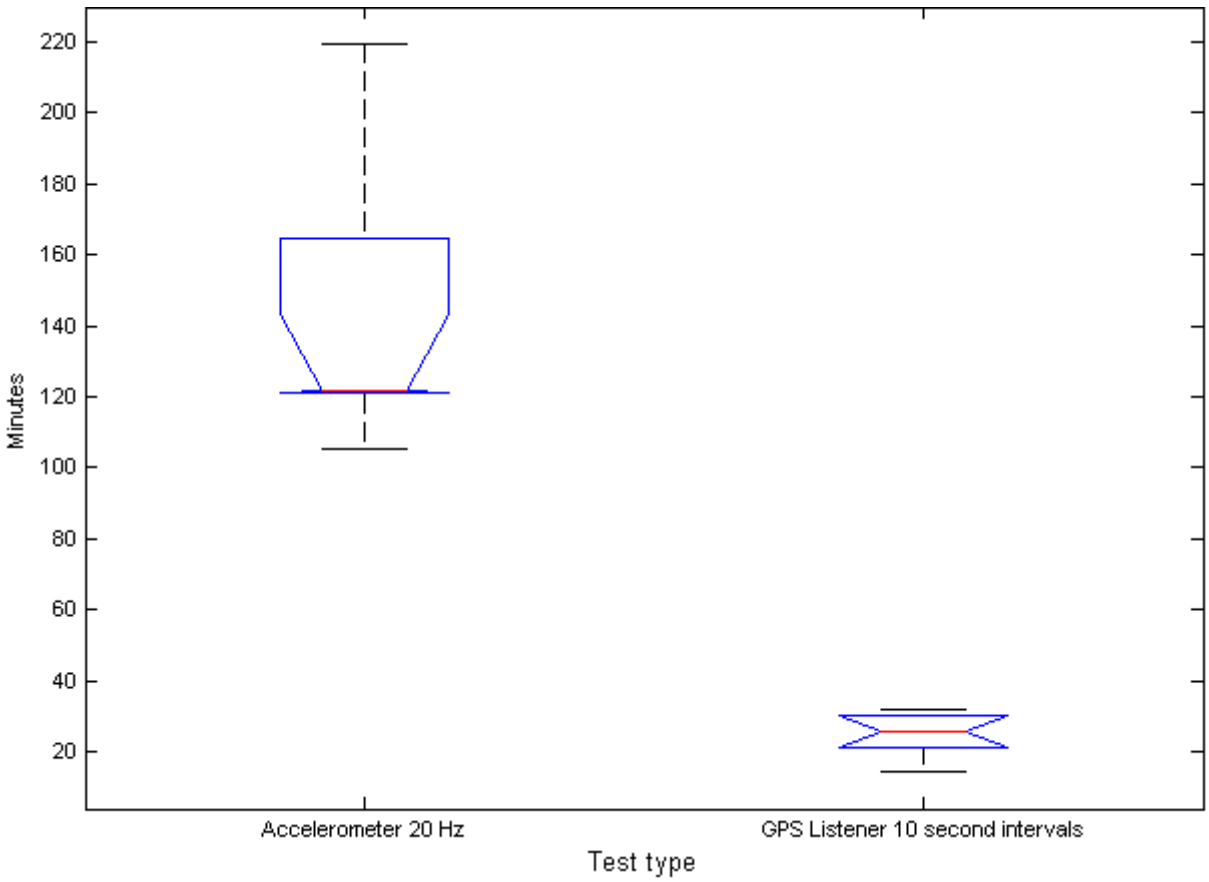

Fig. 9.4: Battery life while using the accelerometer compared to GPS.

The mean battery life was 138:29 for the accelerometer test and 24:55 while using GPS. The box plot shows that the accelerometer data were skewed. The data set is very bottom heavy, but has some long lasting results.

The accelerometer lasts about 5.6 times longer than the GPS listener. This indicates that it can be better, from an energy saving perspective, to use the accelerometer in intersections. However this depends on how long it is until the next scheduled GPS update. If a GPS update is scheduled one second after the intersection, it is better to request the update at once, and not having to use the accelerometer. This is illustrated in figure 9.5.

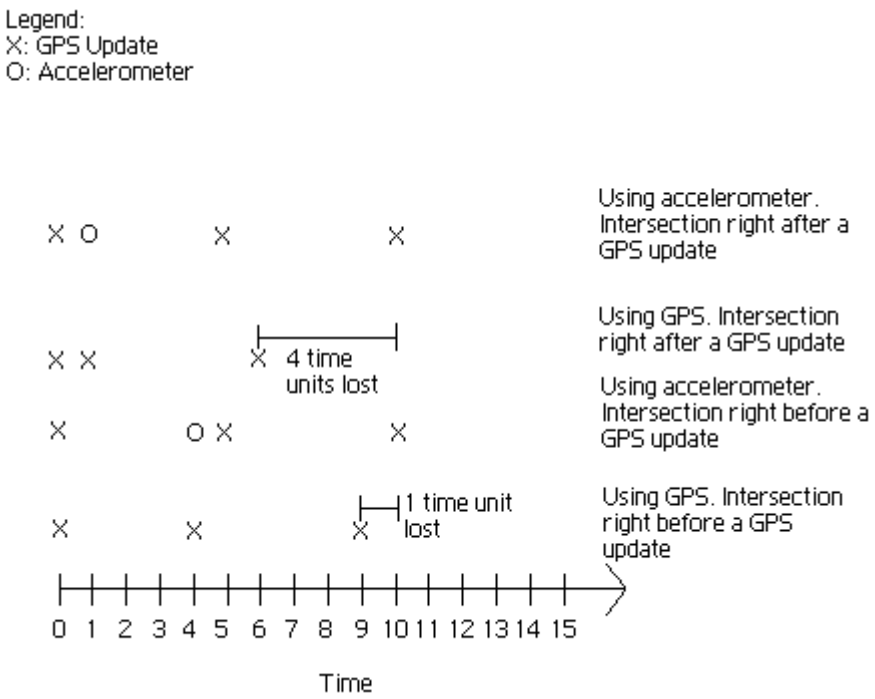

Fig. 9.5: The advantage of using an accelerometer and it's relation to the time until the next scheduled GPS update.

The figure use a getter scheme (no double calls) for receiving GPS updates. The result would however be similar for a listener. The explanation of the figure is that the battery initially has enough power left for a little bit more than two GPS updates and an accelerometer intersection reading. This means it will turn itself off when the third GPS update is requested. The figure shows that the longer the time between the intersection and the next GPS update, the more useful it is to utilize the accelerometer. More research is needed to find exactly the time until next intersection where using the accelerometer is equivalent, from an energy perspective, to using the GPS receiver. This could then be implemented into the Travel Application as a threshold, to dynamically choose the most energy conserving solution.

9.7 Testing a commercial navigation application

The Sony Ericsson W760i comes with a preinstalled navigation application called Wayfinder Navigator. As a standard it uses 3D maps when routing. These are downloaded, on demand, from the Internet. When choosing the routing destination, the address search is also performed online. Wayfinder Navigator gives step by step voice directions, in multiple languages. When the routing function is used, the phone's background light never turns itself off. During testing it seemed to have GPS update intervals of approximately 2 seconds. All these factors add up to the conclusion that energy conservation was probably not on the agenda when Wayfinder Navigator was developed.

Fig. 9.6: Screenshot of Wayfinder Navigator[23]

The Wayfinder Navigator tests were performed in exactly the same manner as the Travel Application tests. There was however one small difference. The routing was performed to Grannesveien. This turned out to be a different road than the one used in the Travel Application, Grannesvegen. But this probably had no effect on the result, since they were quite close and approximately the same length. During the testing of the Travel Application, different destinations were tried without impact on the results. Grannesveien and Grannesvegen are marked in figure 9.7.

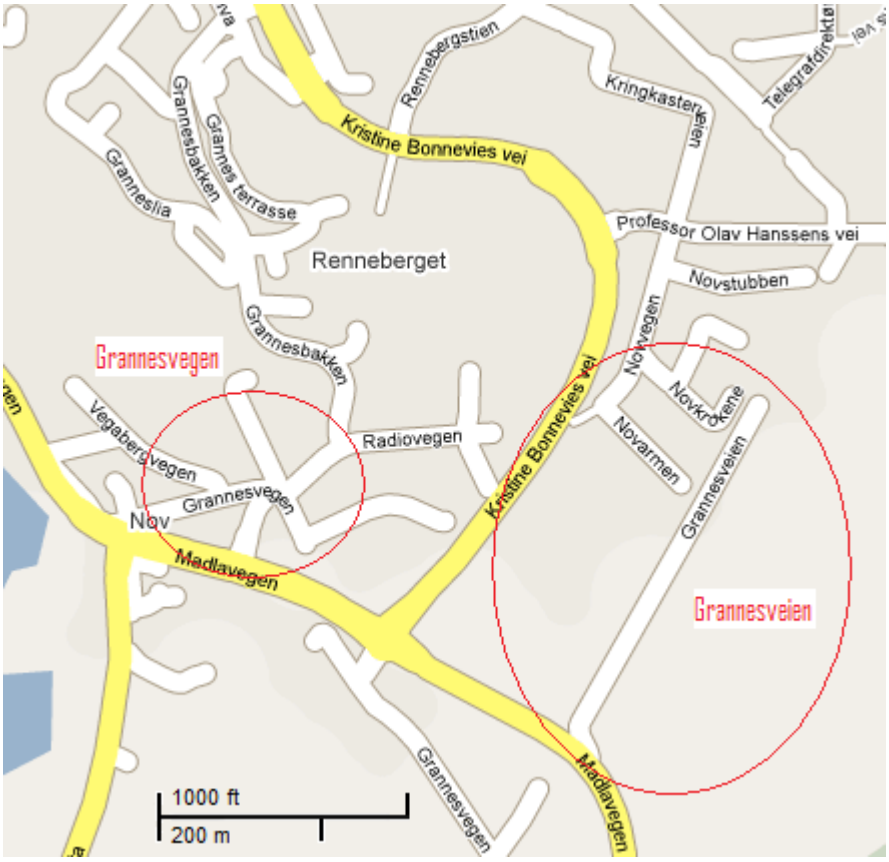

Fig. 9.7: Grannesvegen and Grannesveien in Google Maps [24].

In some of the tests with Wayfinder Navigator, the application did not manage to calculate a route before the phone turned itself off. Figure 9.8 compares the test results to the results of the Travel Application.

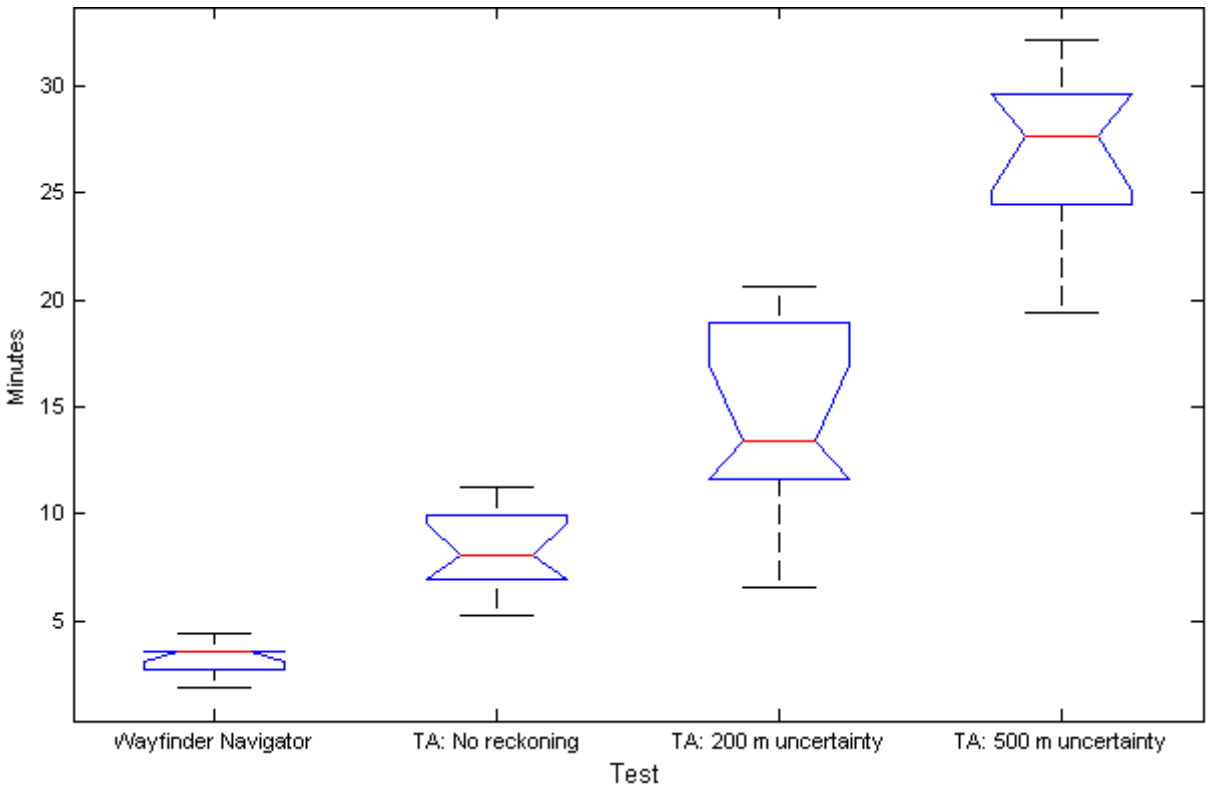

Fig. 9.8: Wayfinder Navigator compared with the Travel Application.

Wayfinder Navigator had an average battery life of 3:14 minutes during the tests. Even with no reckoning, the Travel Application had approximately 2.6 times longer average battery life. By extending the uncertainty level to 500 meters, the battery lasted 8.4 times longer. This shows that designing an application for low power consumption can lead to a substantial gain in battery life.

There is a possibility in Wayfinder Navigator to download a section of the world map in advance. This feature was not tested, but would probably have lead to a slight increase in the application's average battery life.

10 Conclusion

The introduction of uncertainty and dead reckoning to a mobile GIS makes it possible to have longer GPS update intervals. When accepting a 500 m uncertainty diameter, the test phone's battery lasted approximately 3 times longer than with no reckoning. Although the result may vary between phones, the increase is so large that tests done with other phones probably would lead to the same conclusion. This shows that the GPS update intervals are a critical factor to a mobile GIS application's power consumption. Developers should take this into account during the design of programs for mobile GPS receivers. The need for accurate information ought to be weighed against the need to preserve battery power.

A crucial element in a mobile GIS, that determines how long the update intervals can be, is the choice of dead reckoning scheme. A well developed reckoning algorithm can make longer update intervals transparent to the user. It can also make the application less dependent on a constant GPS signal. As the study shows, use of movement indicators with lower power consumption than the GPS module, like an accelerometer, can be successfully implemented as a part of such a scheme. The data the application has available must be considered when planning a reckoning algorithm. How to cleverly use that information to determine the user's current position is the key to successful dead reckoning. When developing mobile GIS applications with low power consumption, a lot of effort should be put into this part of the program. Dead reckoning has much room for improvement and can prolong battery life while conserving the user experience.

10.1 Further work

The following topics are suggestions for further research on uncertainty, GPS and dead reckoning algorithms. Also mentioned are some improvements that could be implemented in the Travel Application.

- **Accelerometer versus GPS in intersections:** This is regarding the Travel Application's dead reckoning scheme. Choosing which road to follow after an intersection is always done by the accelerometer in the RoadNoRoute reckoning algorithm. However if the intersection comes very close to the next scheduled GPS update, advancing the GPS update would use less power. There is a threshold between using the accelerometer and advancing the GPS update. The easiest way to find an estimate of this threshold is to simulate the situation using a discrete event simulator. Knowing the relationship between the cost of a GPS update and the cost of using the accelerometer, one or more simulation models could be constructed. For instance a Petri Net graph could be used to describe these models. Once the models have been established, running the simulations to find the equilibrium point is straightforward.
- **Effect of GPS signal condition on different update methods:** There are two different update methods for a J2ME GPS application, getter and listener. In terms of power consumption, getter is the best when signal conditions are good, while the listener outperforms the getter in bad conditions. If the condition threshold between getter and listener was found, this could be used to dynamically switch update method in GPS applications. The threshold can be described in terms of GPS uncertainty or time required to get an update. Further testing is needed to find this value.
- **Update now button in the Travel Application:** In the higher uncertainty levels the time between GPS updates can be in the order of minutes. Therefore a neat feature

would be to have an update now button. This could for instance be useful when the user approaches an unfamiliar intersection. It would be easy to implement in the Travel Application.

- **Making the Travel Application scalable:** To make the Travel Application compatible with larger maps it is essential to improve the scalability of the map drawing algorithms. With the current implementation it would take many seconds to draw a map of a country like Norway. This is of course not acceptable. What needs to be done is to add map layers so that the level of detail goes down when the user zooms out. A normal mobile phone has limited memory. This makes it important to avoid having the entire map loaded into memory at any time. Ergo memory aware map handling algorithms are needed. The drawing of the coastlines also needs to be perfected for the Travel Application to work well for larger maps.
- **Perfecting the dead reckoning:** There are many things that can be done to perfect the dead reckoning scheme in the Travel Application. When driving, the current implementation always assumes the driver follows the speed limit. Instead it could have analyzed the driving behaviour of the user and decided if he was a slow or fast driver. For instance if he over time holds a speed of 10 km/h below the current speed limit, the dead reckoning would move the current position at a slower rate than usual. This will lead to a more accurate estimate. This feature could also help during rush hours. If a driver suddenly drives very slow compared to his normal driving pattern, the dead reckoning could go into rush hour mode. The current position would then move very slowly, at least until the next intersection.

Bibliography

1. http://www.rand.org/pubs/monograph_reports/MR614/MR614.appb.pdf - *Pace S., Frost G. P., et. Al.* – The Global Positioning System – Rand Publishing 1995 – Retrieved on 2009-05-05.
2. http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html - *Dana P. H.* – Global Positioning System overview – Retrieved on 2009-05-05.
3. http://electronics.howstuffworks.com/gadgets/travel/gps.htm - *Brain M., Harris T.* – How stuff works article about GPS receivers – Retrieved on 2009-05-05.
4. http://www.how-gps-works.com/glossary/ - How GPS works. Retrieved on 2009-05-05.
5. http://java.sun.com/javame/index.jsp - The official home page of J2ME from SUN Microsystems – Retrieved on 2009-05-06.
6. http://to.swang.googlepages.com/ICC2008LBSforMobilessimplifiedR2.pdf - *S. Wang, J. Min, B. K. Yi* Location based Services for Mobiles: Technologies and Standards, International Conference on Communication (ICC) 2008, Beijing, China Retrieved on 2009-02-10.
7. http://store.apple.com/no/browse/home/shop_iphone/family/iphone - Apple iPhone 3G product page. Retrieved on 2009-02-10.
8. http://www.maxhorvath.com/documents/programming_for_iphone_using_l/eclipse-iphone-cdt.pdf - *P. J. Cabrera* Programming for iPhone in windows or linux Retrieved on 2009-02-10.
9. www.prisguide.no – A Norwegian site for comparing prices among web shops. Retrieved on 2009-02-10.
10. http://www.forum.nokia.com/devices/N96 - Nokia N96 device details on forum Nokia. Retrieved on 2009-02-10.
11. http://www.sonyericsson.com/cws/products/mobilephones/overview/w760i?cc=no&lc=no – Sony Ericsson W760i Product page. Retrieved on 2009-02-10.
12. http://developer.sonyericsson.com/docs/DOC-1734 - Sony Ericsson developer wiki about on-device debugging. Retrieved on 2009-02-10.
13. http://www.zett.no/kart.html?lp=nt – Aerial raster map. zett.no. Retrieved on 2009-03-31.
14. http://www.openstreetmap.org – Open Street Map a free editable vector map. Retrieved on 2009-03-31.
15. http://wiki.openstreetmap.org/wiki/Tags - List of OSM tags. Retrieved on 2009-04-01.
16. http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Ian//cutting_ears.html – *Ian Garton.* About simple polygon ear cutting. Retrieved on 2009-04-14
17. http://www.chemical-ecology.net/java/lat-long.htm - *John A. Byers.* Surface distance calculator
18. http://www.itl.nist.gov/div897/sqg/dads/HTML/dijkstraalgo.html - *National Institute of Standards and Technology*. Time complexity of Dijkstra's algorithm. Retrieved on 2009-04-29.
19. http://www.itl.nist.gov/div897/sqg/dads/HTML/bellmanford.html - *National Institute of Standards and Technology*. Time complexity of the Bellmann-Ford algorithm. Retrieved on 2009-04-29.
20. http://www.kowoma.de/en/gps/errors.htm - *Dr. Anja Köhne und Dr. Michael Wößner*. Sources of errors in GPS. Retrieved on 2009-04-21.
21. http://jcp.org/aboutJava/communityprocess/final/jsr179/index.html - J2ME JSR-179 Location API. Retrieved on 2009-04-22.

22. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/index.html?/access/helpdesk/help/toolbox/stats/boxplot.html - The MATLAB manual's article about box plots. – MathWorks – Retrieved on 2009-05-08.
23. http://www.wayfinder.com/?id=8618 – Product page for Wayfinder Navigator - Wayfinder UK – Retrieved on 2009-05-18.
24. http://maps.google.com/ - Google Maps – Retrieved on 2009-05-18.
25. http://en.wikipedia.org/wiki/A* - Wikipedia article about the A* search algorithm with pseudo code. Retrieved on 2009-04-20.

Appendix

Appendix A Algorithms

Appendix A.1 The A* Algorithm

```
function A*(start,goal)
     closedset := the empty set % The set of nodes already evaluated.
     openset := set containing the initial node % The set of tentative
nodes to be evaluated.
     g_score[start] := 0 % Distance from start along optimal path.
     h_score[start] := heuristic_estimate_of_distance(start, goal)
     f_score[start] := h_score[start] % Estimated total distance from start
to goal through y.
     while openset is not empty
         x := the node in openset having the lowest f_score[] value
         if x = goal
             return reconstruct_path(came_from,goal)
         remove x from openset
         add x to closedset
         foreach y in neighbor_nodes(x)
             if y in closedset
                 continue
             tentative_g_score := g_score[x] + dist_between(x,y)
             tentative_is_better := false
             if y not in openset
                 add y to openset
                 h_score[y] := heuristic_estimate_of_distance(y, goal)
                 tentative_is_better := true
             elseif tentative_g_score < g_score[y]
                 tentative_is_better := true
             if tentative_is_better = true
                 came_from[y] := x
                 g_score[y] := tentative_g_score
                 f_score[y] := g_score[y] + h_score[y]
     return failure

 function reconstruct_path(came_from,current_node)
     if came_from[current_node] is set
         p = reconstruct_path(came_from,came_from[current_node])
         return (p + current_node)
     else
         return the empty path
```

Listing A.1: Pseudo code for the A* algorithm. [25]

Appendix A.2 The update driving uncertainty algorithm

```
updateUncertainty(currentTime){
  Fill the empty hash table exploredNodes with the currentInsideNodes
  //The currentInsideNodes will initially be the nodes inside the uncertainty circle
  Fill the vector temporaryBoundaryNodes with the currentBoundaryNodes.
  //The currentBoundaryNodes are stored between updates and are initially calculated when the next
  //update time is estimated
  Set the time remaining in those BoundaryNodes = currentTime – previousTime;
  Make an empty vector nextBoundaryNodes, which represents the boundaryNodes at time =
  currentTime
  While (there are more temporaryBoundaryNodes){
    Remove the first temporaryBoundaryNode
    Calculate the distance to its outside neighbour
    If(it can not cover the distance in its the remaining time given the road's speed limit){
      Make a new BoundaryNode at the place where the temporaryBoundaryNode ended up when the
      time expired
      Add this BoundaryNode to the nextBoundaryNodes
    }
    Else{
      Put the outside neighbour in the explored nodes
      Set tmpTimeRemaining = the time remaining when reaching the outside neighbour
      Set avoidNeighbour = temporaryBoundaryNode.insideNeighbour
      While(outside neighbour has more neighbours){
        If(the neighbour = avoidNeighbour){
          Continue
        }
        Else{
          Make a new TemporaryBoundaryNode tmpBN.
          Set its inside neighbour = outside neighbour
          Set its position = outside neighbour's position
          Set its outside neighbour = the neighbour
          Set its time remaining = tmpTimeRemaining
          Add tmpBN to the temporaryBoundaryNodes
        }
      }
    }
  }
  Fix the boundaryNodes inside references in the case where the inside neighbour is another
  boundaryNode.
  Set the currentBoundaryNodes = nextBoundaryNodes
  Add the exploredNodes to the currentInsideNodes
  Set the previousTime = currentTime
}
```

Listing A.2: The update driving uncertainty algorithm.

Appendix A.3 The reclassify algorithm

```
DeadReckoning reclassify(currentRoute, newest GPS location, current uncertainty){
  If the accelerometer is collecting data, shut it down.
  If( the uncertainty is a DrivingUncertainty){
    If(the user is driving on an uncharted road){
      If(there is no currentRoute ){//currentRoute = null
        Return a new OffroadNoRouteReckoning object
      }else if(there is no previous route){
        Return a new OffroadRouteReckoning object
      }else if(currentRoute has a different target than the previous route){
        Return a new OffroadRouteReckoning object
      }else if(currentRoute is closer to the target than the previous route){
        Return a new OffroadRouteReckoning object
      }else{
         //the user has the same target, but is farther away from it than last time,
         //ergo he must not be following the route
         Return a new OffroadNoRouteReckoning object
      }
    }else{//driving on a road
      If(there is no currentRoute ){//currentRoute = null
        Return a new RoadNoRouteReckoning object
      }else if(there is no previous route){
        Return a new RoadRouteReckoning object
      }else if(currentRoute has a different target than the previous route){
        Return a new RoadRouteReckoning object
      }else if(currentRoute is closer to the target than the previous route){
        Return a new RoadRouteReckoning object
      }else{
         //the user has the same target, but is farther away from it than last time,
         //ergo he must not be following the route
         Return a new RoadNoRouteReckoning object
      }
    }
  }
  If(the uncertainty is a walking uncertainty){//This is treated essentially the same way as driving on an
  //uncharted road
    If(there is no currentRoute ){//currentRoute = null
      Return a new OffroadNoRouteReckoning object
    }else if(there is no previous route){
      Return a new OffroadRouteReckoning object
    }else if(currentRoute has a different target than the previous route){
      Return a new OffroadRouteReckoning object
    }else if(currentRoute is closer to the target than the previous route){
      Return a new OffroadRouteReckoning object
    }else{
       //the user has the same target, but is farther away from it than last time,
       //ergo he must not be following the route
       Return a new OffroadNoRouteReckoning object
    }
  }
  //The algorithm continues on the next page
```

```
  //if the method hasn't returned yet, the uncertainty is neither of the above. This means we are
  //still in the start up phase
  //The dead reckoner then needs to update its previous route (this is done in the constructor for the
  //for the subclasses. It should return itself, since it is a no guessing implementation.
  Set previous route = currentRoute
  Return this
}
```

Listing A.3: The reclassify algorithm.

Appendix A.4 The calculate new position algorithm

```
calculateNewPosition( currentTime){
  Make a new TempPosition object , tmpPos,  with the current position and time remaining =
  currentTime – previous time.
  //The nextNode is initially set to the first node in the route (not where the user is at the start)
  //The current speed is constant and calculated initially as the speed held between the two latest
  //updates
  While(tmpPos != null and nextNode != null){//When it reach the target it will stop moving
  //Use a while loop since it can pass by multiple nodes between updates.
    Calculate the distance from tmpPos to nextNode.
    Calculate the distance covered in the tmpPos time remaining, given the current speed.
    If(distance > distanceCovered){
      Move tmPos distanceCovered meters towards the nextNode.
      Set currentPosition = tmpPos.position.
      Set currentCourse = the course from tmpPos to nextNode.
      Set tmpPos = null
    }else{//The user passed the nextNode within the timeRemaining
      Calculate a new timeRemaining, which takes the previous time remaining and subtracts the time
      Used to reach the nextNode
      Set tmpPos = a new tmpPos with position = nextNode and timeRemaing = the new timeRemaining
      If(nextNode is not the routing target){
        Set nextNode = the node after the nextNode in the route.
      }else{//the user has arrived at the routing target
        Set currentPosition = nextNode.position
        Set nextNode = null;
      }
    }
  }
}
```

Listing A.4: The OffroadRouteReckoning's calculate new position algorithm.

Appendix A.5 The calculate position after an intersection algorithm

```
calculateTheNewPosition(){//Assume the data has been collected and the axes has been found
  Set currentPosition = start position before the intersection accelerometer data collecting
  Set currentSpeed = starting speed, the speed limit before the intersection
  Set currentAngle = the estimated angle at the starting time.
  Set currentAngleSpeed = 0;
  Loop(over all collected data){
    currCentAcc = 0;
    If(centripetalAxis is flipped){
      currCentAcc = -collected centripetal acceleration
   }else{
     currCentAcc = collected centripetal acceleration
   }
   currTangAcc = 0;
   If(tangential axis is flipped){
     currTangAcc = - collected tangential acceleration
   }else{
     currTangAcc = collected tangential acceleration
   }
   currAngularAcc = (currTangAcc*currCentAcc)/(currSpeed * currSpeed)
   t = the time between this and the next data was collected.
   currentAngle = currentAngle + currentAngleSpeed * t + ½ * currAngularAcc * t *t
   currLon = (currentPosition.longitude * METERPERLON + currentSpeed * t *
   cos(currentAngle) +
   ½ * currTangAcc * t * t * cos (currentAngle))/METERPERLON
   currLat = (currentPosition.latitude * METERPERLAT + currentSpeed * t *
   sin(currentAngle) +
   ½ * currTangAcc * t * t * sin (currentAngle))/METERPERLAT
   currentPosition = a new Position object with longitude = currLon and latitude = currLat
   currentSpeed = currSpeed + currTangAcc * t
   currentAngleSpeed = currentAngleSpeed + currAngularAcc * t
 }
}
```

Listing A.5: The RoadNoRouteReckoning's calculate new position after an intersection algorithm.