NTNU
Norwegian University of
Science and Technology

Using the Geographical Location of Photos in Mobile Phones

Jon Anders Amundsen

Master of Science in Computer Science
Submission date: July 2008
Supervisor: Svein-Olaf Hvasshovd, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The task is to explore different usages of geotagged photos and investigate how suitable they are for mobile devices. The new Android mobile platform should be examined to see how suitable it is as a platform for applications that involve geotagging. Prototypes of applications using geotagged photos should be implemented using the Android platform. A small user study should preferably be carried out using the prototypes to examine if people find it interesting to use location for enhancing the functionality when browsing photo collections on mobile phones.

Assignment given: 15. January 2008
Supervisor: Svein-Olaf Hvasshovd, IDI

Abstract

Digital cameras in mobile phones have become very popular in the recent years, and it is common to have large photo collections stored in the phone. Organizing these photos on the phone is still a big problem though. This study explores different ways of utilizing the location of where the photos were taken to make it easier to manage a large photo collection. Several different positioning technologies that can be used to obtain the location of where a photo was taken are presented.

Three of the application suggestions for using location information of photos were implemented as prototypes on the Android platform. Android is a new platform for mobile phones developed by Google and the Open Handset Alliance, which has been made available as a preview release for developers. A part of this study was to investigate how suitable this platform is for developing location-based software. It was found that it is very suitable, although there still are some bugs and missing features that are expected to be fixed before the final release.

The three application prototypes that were implemented were called “From Photo to Map”, “From Map to Photos” and “Who Lives Here?” The “From Photo to Map” application lets the user see a map where the location of a selected photo is visualized with a marker. The “From Map to Photos” application shows a map with markers at all of the locations where the user has taken photos. When one of the markers is selected, the photos taken at that location is shown. The “Who Lives Here?” application lets the user know which of the persons in his contact list that lives where the photo was taken.

A small user survey showed that the participants thought all of the applications could be useful, but they were not so sure if they would use them themselves. The survey also showed that most of the users were able to find photos faster when using map-based browsing in the “From Map to Photos” application than when browsing through a photo collection linearly, but several concerns about the implementation details and the use of an emulator make the exact efficiency gain very uncertain.

I would like to thank my supervisor Svein-Olaf Hvasshovd for his cooperation while working on this study. I would also like to thank the participants of the user survey for their cooperation.

Jon Anders Amundsen
Trondheim, 11 July, 2008

Contents

List of figures

1 Introduction

Digital cameras built into mobile phones have become very popular in the recent years. A lot of mobile phone users have built up large photo collections on their phones, since they always carry this camera with them. These photos will often be uploaded to a PC for viewing and sometimes archiving, but they may also be viewed and stored on the phone itself. The storage space and screen size of mobile phones are constantly increasing and opens new possibilities for what can be done on a mobile phone. Photo collections may be browsed by time and date or other metadata, they may be sorted into events or categorized into other kinds of groups. This kind of manual categorization is a very laborious task though, especially for large photo collections. It is also even harder to do on a mobile phone than on a regular PC. It is therefore desirable to have the photo collection automatically organized, so the manual categorization is less needed.

Until the last couple of years, map-based navigation and other uses of location-based data have not been practical on mobile phones, but this is changing. Larger screens, more powerful processors, better network connectivity and other built-in sensors are moving the limits of which kinds of applications that are feasible on a mobile phone. This can also be used to find new ways of interacting with photo collections on the phone. The focus of this study has been to explore different ways in which the location of where a photo was taken can be utilized to aid the mobile phone user in browsing a photo collection. This was stated more specific in the following problem definition for this study:

> *"The task is to explore different usages of geotagged photos and investigate how suitable they are for mobile devices. The new Android mobile platform should be examined to see how suitable it is as a platform for applications that involve geotagging. Prototypes of applications using geotagged photos should be implemented using the Android platform. A small user study should preferably be carried out using the prototypes to examine if people find it interesting to use location for enhancing the functionality when browsing photo collections on mobile phones."*

Some background and related research on using the location information of photos will be presented in chapter 2. The Android platform will be described in detail in chapter 3. Chapter 4 will cover several different ways of utilizing the location information of photos, while chapter 5 will describe how some of these ways was implemented, as well as describe the user study that was carried out. The results of the study will be discussed in chapter 6, and the conclusion is presented in chapter 7. There are also some suggestions of more research that should be done on the basis of this study presented in chapter 8.

2 State of the Art

The usage of the location information in photos is an area that has received an increasing amount of attention in the recent years. This chapter gives an introduction to the concept of geotagging and the most relevant technology involved, present the main research on the topic and review some of the available software for geotagging.

2.1 Geotagging

A geotagged object is an entity that has some metadata attached to it which describes the object's geographic location. In the case of a geotagged photo, the metadata describes the location of where the picture was taken. In its most basic form geotagging is not a new concept. Something as simple as writing "Vacation in Paris" or "At the family cabin" next to a photo in an old fashioned photo album can be regarded as a kind of geotagging. The idea of geotagging is therefore almost as old as photography itself, even though the word geotagging is relatively new. This very simple form of geotagging is of course useful, but with the help of modern technology a whole new world of opportunities arises.

The description of a location may be given in several different ways. As mentioned in [2], these may include commonly understood place names like Paris or the Nidaros cathedral. It may be personal place names like "the family cabin" or "my house", or it may be an address or a zip code. A location may also be described as coordinates. The most common kind of coordinates used to describe locations on the surface of the earth is latitude and longitude, and altitude above mean sea level is also often added. Other coordinate systems, like the (x, y, z) coordinates of a three dimensional Cartesian coordinate system relative to a specified origin may also be used. These kinds of coordinates are usually not as useful though, and therefore not as common. There are other ways of describing location also, but these are the most common.

Place names are often the easiest way of describing a location when humans are communicating with each other. If you ask your friends to meet you at "Marinen" to have a barbeque, it will be easy for them to understand where it is, given that they are familiar with the Trondheim area. For computers, on the other hand, place names may be difficult to interpret, so a numerical description like latitude and longitude would be much more useful. But if you asked your friends to meet you at 63° 25' 30'' N, 10° 23' 45'' E you might easily end up eating alone.

In the rest of this study the latitude and longitude representation of a location is being used, since the study concerns the usage of location information on a mobile device, which is a kind of portable computer.

2.2 Reference Systems for Coordinates on Earth

The use of latitude and longitude may seem like a very unambiguous way of describing a location, but it may in fact not be. Several different reference systems have been used throughout the world to define coordinates, and they are not completely compatible with each other. The reason for this is that until the latest few decades accurate maps were mostly made for a small area of the earth, and therefore models of the earth that best approximated that specific area was used. The earth is an irregular shape, so a model that is very accurate for one

part may be less accurate for other parts. Examples of such models that have been in use are the North American Datum, European Datum and Tokyo Datum. The difference between locations described using the different models may be as much as a kilometer. Eventually the need for a global reference system for coordinates grew, and in 1966 the first version of the World Geodetic system was defined. The latest version of this system is currently the World Geodetic system 1984 (WGS84). This is the reference system currently used by the GPS system.

WGS84 [3] is a model of the earth, published by the National Imagery and Mapping Agency, which is a part of the US Department of Defence. The model defines a reference frame for a coordinate system for describing positions on the earth. It defines the shape of the earth as an approximation of an ellipsoid with a semi-major axis (equatorial radius) of 6378137.0 meters and a semi-minor axis (polar radius) of 6356752.3142 meters.

The GPS system uses the WGS84 reference system to define latitude and longitude. Since this is the most common way of getting the coordinates of a location, this have also become the most common way of describing a location numerically. In most new maps and in Geographic Information Systems (GIS) applications it can be expected that WGS84 is used as the reference system for coordinates, unless otherwise stated. Thus, WGS84 is also the reference system used in this study.

2.3 Methods for Obtaining Location Information

There are a number of different methods for obtaining the location information that is needed when a photo is to be geotagged. It can be done manually, and it can be done automatically using several different technologies. Most of the common technologies used for automatically obtaining location can be described as either stand-alone, satellite-based, or terrestrial radio-based (Mobile Phone Location Determination and Its Impact on Intelligent Transportation Systems). A common example of a stand-alone system is dead reckoning, i.e. using sensors to track movement. An example of a dead reckoning system is to estimate the position of a vehicle moving from a known starting point using an odometer and a compass. The most common satellite-based system for obtaining location is the Global Positioning System (GPS), which is described in section 2.3.3. An example of a terrestrial radio-based system is LORAN-C [4], which consists of radio-towers transmitting navigation signals to ships, aircrafts and others. It is an old system, which has become less popular because of GPS. Although there are lots of different systems for obtaining location, only the ones that are applicable for mobile phones will be discussed in this study.

One of the aspects of geotagging that must be defined is whether to store the location of where the camera was when taking the picture or the location of what you can see in the picture. If a vague description of location was used, like the name of the city where the photo was taken, this is usually not an issue. But when using coordinates like latitude and longitude the distance between the camera and the objects in the photo may be much greater than the precision of the location information, and the photo may also cover a large area with several objects far away from each other. If the photo is geotagged manually, then it would usually be easy to define the location of one single object seen in a photo, but when there are several significant objects in a photo it might be hard to decide which object to choose. There is also currently no good way of doing this kind of geotagging automatically. Therefore, because of these complications, only the location of the camera will be considered in this study. The location of the camera can relatively easy be automatically obtained from a positioning

system built into the camera or located close to the camera. Manually deciding the location of the camera is also usually not much harder than deciding the location of an object that you can see in the photo.

2.3.1 Manual

Probably the most basic way of determining the latitude and longitude of where a photo was taken is to find the location on a map and manually read off the latitude and longitude. This can be done using an old fashioned printed map, or using an electronic map. There are several computer programs available to help in this process, where you can browse through your photo collection and click on a map to indicate where the photo was taken, or even just drag and drop the photo onto a map. The software will then store the location information for later use. Some applications that can be used for this task is described later in this chapter.

Even though this is the simplest approach to geotagging, it has several disadvantages. Firstly, it is a very laborious task that may take a lot of time if the photo collection is large. It is unlikely that most users will think that it is worth the time and effort required to do this work, at least not over time. There is also likely that human errors will occur from time to time, for instance reading the map wrong or not being able to find the location on a map at all. Another problem is that people may forget where a picture was taken, especially if there is a long time between taking the picture and geotagging it. For example if a user has been on a long trip, e.g. a round trip in China, and has been taking a lot of pictures during the trip, the photographer may not remember the exact location of where all of the pictures was taken when he comes back home.

2.3.2 Cell-ID

One of the least accurate ways of automatically obtaining the location of a mobile phone is to use the cell-ID of the base station the phone is currently connected to. This location technique has been reviewed in [5]. The idea is to use the location of the base station as an approximation of the phone's actual location, instead of finding the exact location of the phone itself. The accuracy of this approximation is proportional to the size of the network cells, i.e. the reach of each base station in the network. The cell size varies wildly as it depends on the population density in the area. An area with lots of people needs more base stations than areas with few people, and therefore the cell size is much smaller in urban than in rural areas. The accuracy in urban areas can be down to a few hundred meters, while the accuracy in rural areas will often be as high as several kilometers.

The biggest problem of implementing a positioning system based on cell-ID is that there is no public database that describes the location of all the base stations available. All base stations have a globally unique identifier, and it is available to software running on mobile phones, but only the mobile operators have access to databases that map these identifiers to the exact location of their own base stations. Some operators can provide location based on cell-ID as a service to customers, but it is usually not a free service. It would be a lot of work and probably very expensive to gain access to these services for applications that are meant to be deployed globally, since a very high amount of operators would have to be involved. These services are probably not available from all operators either. Another method of obtaining the locations of the base stations is to use publicly available databases that are being maintained by companies and organizations that see the benefits of a cheap positioning system being available all over the world. Using these databases will achieve lower coverage and accuracy,

but they are available for free. The databases are built by a community of users who have both mobile phones and GPS receivers. Areas are then mapped out by uploading GPS positions of where they are located together with the cell-ID of the base station that their mobile phone is currently connected to. The approximation of each base station's location gets better as the number of observations grow. This kind of community-built databases are available in Google Maps Mobile, in a service called My Location [6], in the Yahoo! ZoneTag Cell Location API [7], in CellDB [8], GSMLoc [9] and CellSpotting.com [10].

The element that makes this positioning method attractive is that it does not require any additional hardware at neither the mobile phone network nor the phone itself, and it can therefore be very cheap to deploy. Unfortunately, the accuracy is very low and unpredictable, so this is not an ideal location technology for use in a geotagging application.

2.3.3 GPS

The Global Positioning System (GPS) [11][12][13] is a satellite-based positioning system that provides three-dimensional positioning and accurate time to GPS users all over the world at all times. The system was developed by the United States Department of Defense, and is currently being maintained by the Global Positioning Systems Wing in the US Air Force.

The US Department of Defense started the research on a satellite-based positioning system already in the late 1950s, but getting from there to a fully functional system took a long time. The first GPS satellite was launched in 1978. Full Operational Capability was not declared until 1995 [14].

The system was designed to need at least 24 operational satellites to be functional all over the globe at all times. There are currently 31 operational GPS satellites in orbit, which provides for better accuracy and better fault tolerance of the system in case of failing satellites. All the GPS satellites contain atomic clocks that are synchronized to be able to keep time as accurate as possible. Each satellite is constantly transmitting its current time and information about its orbit in addition to other information about the satellite towards the earth.

A GPS receiver will calculate its own position based on the radio signals it receives from the satellites. Even though the clocks on all the satellites are synchronized, the GPS receiver will still see small differences in the timing information received from the different satellites. This is because the time that the signal needs to travel from a satellite to a GPS receiver is dependant on the distance between the satellite and the receiver. The exact position of the GPS receiver can be calculated using trigonometry when the positions of the satellites are known in addition to the distances between the satellites and the receiver. The clock of a GPS receiver can not be expected to be accurate and synchronized with the GPS satellites. It is therefore necessary to solve a set of equations to find not only the three dimensional position of the receiver, but also the exact time. The position can not be calculated accurately without the time also being accurate. A GPS receiver needs signals from at least four different GPS satellites to be able to calculate the three dimensional position plus the time dimension. If signals from more satellites are available, these will be used to achieve better accuracy.

The GPS satellites broadcast navigation signals on two different frequencies. One is publicly available to anyone who buys a GPS receiver. The other one is encrypted and only available to the US military.

Better and more accurate GPS receivers are constantly being developed. Some of the newer receivers, like the SiRFstarIII GSC3e/LP chip [15], can achieve an accuracy of less than 2.5 meters. According to [16], the theoretical limit for the accuracy of GPS is just a few centimeters.

The GPS system is a good system for obtaining the users current location, but it also has several drawbacks. One significant drawback is that the signals sent from the satellites are not very strong. Therefore the GPS receiver needs a nearly clear view to the area of the sky where the satellite is located to be able to receive the signal. This means that GPS receivers will usually not work inside buildings. When used in cars they have to be placed in one of the windows or use an external antenna, and they do not work inside tunnels. A work-around for car navigation systems in tunnels is to estimate the actual position inside tunnels based on the speed of the car before going into the tunnel and the time since the signal was lost. This is a kind of dead reckoning system. Another drawback is that GPS receivers have been known for giving wrong positions or no position at all when used in urban areas with lots of tall buildings or close to a tall rock wall. There are two reasons for this. The first is because a very small area of the sky is visible. It is therefore hard for the GPS receiver to receive any signals. The other reason is that the signal from the satellites gets bounced off the walls of the tall buildings before reaching the GPS receiver. This effect is called multipath. It causes the time measured for the signal traveling from the satellite to the receiver to be a bit longer than it would be if it went in a straight line. The GPS receiver may also receive the same signal more than once in this situation. Newer GPS receivers have become much better at tackling such difficult conditions, but some areas still cause problems.

A problem with GPS that is particularly important when the receiver is embedded in mobile phones is that the time it takes before the system can determine its location the first time after starting the receiver, called time-to-first-fix, can be very long. This can be from around 30 seconds to as much as several minutes [17]. When the GPS receiver knows nothing about which satellites are visible, it has to scan a large frequency range before actually finding a valid signal. It then has to download a description of each satellites orbit and position, called the ephemeris. This is a very slow transmission, only 50 bits/second. When this first initialization is done, the calculation of new locations can be done continually. The problem for mobile phones is that if the GPS receiver is running constantly, it would drain the battery very fast. The receiver therefore usually has to be started and re-initialized each time the user requires the location.

Two different kinds of GPS receivers may be used for automatic geotagging of photos: stand-alone or integrated in the camera device, which may be a mobile phone. A stand-alone GPS receiver can be used to record a timestamped track log of where the photographer has been while taking the photos. Most digital cameras store a timestamp integrated in the image files. Therefore when the images are transferred to a computer, the timestamps can be compared to the timestamps in the track log from the GPS to find the photo location. If the GPS receiver is integrated in the camera device, the location data can be stored directly in the EXIF header of the image file, and no post processing is necessary.

2.3.4 AGPS

Assisted GPS (AGPS) is a system that has been developed to solve some of the problems with integrating GPS receivers in mobile phones. The most important task of AGPS is to reduce the time-to-first-fix. This is both necessary to make a location aware system more user

friendly, but also to reduce the battery consumption. The system is based on GPS, but it utilizes the mobile phone network to enhance the performance.

The orbital positions of the navigation satellites are described in two different ways in the GPS system, called almanac and ephemeris. The almanac provides a very rough description of the position, whereas the ephemeris is used to calculate the exact satellite position. If an updated almanac is available when the GPS receiver starts up, it can instantly start receiving on the correct frequency. However the exact frequency that the GPS signals can be received on is dependent on the satellites positions, so in the absence of an updated almanac a frequency range of +/- 4.2 kHz must be scanned [18]. When the exact frequency is found, the ephemeris data has to be downloaded from one of the satellites on a 50 bits/second rate before the receivers exact location can be calculated.

In AGPS there are stationary GPS receivers that are always on and therefore always have updated information of the GPS satellites exact location. These receivers are connected to location servers, which are also connected to the mobile phone network. When an AGPS receiver is turned on it does not have to scan the frequency range and download the orbital positions from the satellites, instead this data can be downloaded from one of the location servers in the mobile phone network. There are several different standards for mobile phone networks, with a wide variety of transmission rates, but all mobile phone networks provides transmission rates that are much higher than the 50 bits/second received from the GPS satellites. Using this method can reduce the time-to-first-fix to just a second or less [19].

The AGPS system has two modes of operation. In the literature about AGPS the mobile phones are called Mobile Station (MS), and the two modes of operation are called MS-based and MS-assisted. The two modes have different uses, and both have their pros and cons.

An MS-based AGPS receiver will normally work just as a regular GPS receiver. It uses the mobile phone network to reduce the time-to-first-fix, but after that there is nothing different. The main advantage of an MS-based AGPS receiver compared to an MS-assisted one is that if a mobile phone network is not available, or if there is no location server available in the current network, the MS-based AGPS receiver can still function as a normal GPS receiver. In such cases it will have the same time-to-first-fix as regular GPS receivers.

An MS-assisted AGPS receiver, on the other hand, will offload much of the location calculation to a server in the network. The requirements of memory and processing power in the mobile phone is therefore lower, and thus the price of implementing an MS-assisted AGPS receiver in a phone is less than an MS-based one. It will also not drain the phones battery as much. The downside of MS-assisted receivers is that they transmit a lot more data over the mobile phone network. Each time the location needs to be updated, the phone must transmit data from the AGPS receiver to a location server that calculates the position. If the location information is to be used by the mobile phone itself, and not by a service in the network, the position must be transferred back to the phone.

For applications that constantly need updated information of the phones location on the phone itself, such as navigation applications, an MS-based receiver will be the best choice, since it will calculate the location on the phone and not generate any network traffic. If only seldom updates of the location are required, and the location information is not to be used on the phone itself, then an MS-assisted receiver will be a cheaper choice. This can for instance be

very useful for operators at emergency call centers that need to know where the caller is located.

In the context of a geotagging application, an MS-based AGPS receiver will be preferred, since the location information is needed on the phone itself.

2.3.5 WLAN

The availability of Wireless Local Area Networks (WLANs) have increased at an enormous rate in the recent years. They are now widespread in lots of corporations, schools and universities, in addition to people's homes. For a lot of people, WLAN is the preferred way of accessing the internet. It is available on both PCs with a WLAN adapter, as well as some of the newer and more advanced mobile phones. The wireless network clients connect to a wireless Access Point (AP) that relays connections to the regular wired network. The range of an AP is greatly influenced by the kind and size of antenna and the obstructions between the AP and the client. According to [20], the range in a typical office environment is less than 70 meters, but the range can be several times larger when there are no obstructions.

The high availability of WLANs has fostered the idea of using these networks as a positioning system for mobile network clients. The WLAN-based positioning systems can be divided into two research areas: Indoor positiong systems restricted to a small area, for instance a hospital, mall or university, and systems with wide area coverage. Most of the formal research on this kind of positioning systems has focused on indoor systems.

[21], [22] and [22] provides good descriptions of how an indoor positioning system based on WLAN APs can be realized. The basis for these systems is that the area of interest needs to have a very good WLAN coverage, and the exact locations of all the APs must be known to the system. Since the range of WLAN APs is low, these systems are only useful in relatively small areas where the WLAN coverage and AP locations can be controlled. This kind of systems can give a mean distance error as low as 1.5 meters under good conditions. Because these systems are restricted to small areas only, they are not applicable in most general geotagging applications.

Wide area WLAN positioning, on the other hand, is much more suitable for geotagging. There has not been much formal research on this, but there are several providers of WLAN positioning systems available on the internet. These systems are made possible by creating large databases of the location of WLAN APs all over the world. The databases are created by traveling around in urban areas with a laptop with both WLAN adapter and GPS, constantly scanning for new WLANs, and recording the GPS location for each AP. It is not possible to know the exact location of each AP, as required by the indoor systems, but the approximate location can be estimated based on several observations of the same AP from different locations.

The Boston-based company Skyhook Wireless, Inc. [24] has built one of the most comprehensive WLAN AP location databases available, and is constantly expanding it. Since it is a commercial company, they are able to pay drivers to scan urban areas all over the world. They claim to have tens of millions of APs in their database, have 20 meter accuracy, and cover 70% of the population in the US, Canada, Australia, Germany, France and the UK. The time-to-first-fix is less than one second. WiGLE [25] is an organization that also are creating a database of WLAN AP locations, but this is not a commercial company, and

therefore all the scanning has to be done by volunteers for free. WiGLE has currently collected the location of more than 14 million APs, by more than 800 million unique observations, since the start in 2001.

These WLAN positioning systems with wide area coverage have at least as good coverage indoors as outdoors, or maybe even better, and therefore they may be a very good addition to GPS, which has poor indoor coverage. The accuracy is also much better than systems based on the mobile phone networks cell-ID. However, GPS can not be replaced with a WLAN-based system, since these systems only work in urban areas, and the coverage in rural areas is very poor or non-existent.

2.4 Related Research on Geotagged Photos

There have been several research papers published on how to utilize the information from geotagged photos, but most of this research has been on PCs, and not mobile phones. Some research regarding mobile phones is available though. The publications that are most relevant to this study are presented here.

2.4.1 World Wide Media eXchange

The World Wide Media eXchange (WWMX) is a database of geotagged photos developed by Microsoft Research. [2] describes this database and the development of a location-aware image browsing client for PCs, and also briefly introduces some other possible applications of geotagged photos. The WWMX is a client/server system, where the server-side consists of both a central server and peer PCs. The full resolution photos are only stored on the contributor's PC, whereas the central server stores thumbnails and metadata for all the users' photos, as well as pointers to the peer PC where the full resolution photo can be obtained. The location-aware image browsing client enables the user to browse the photos in this database using a map. This systems uses an equirectangular projection of the positions, also known as unprojected latitude/longitude, which means that the latitude and longitude values of each photo are treated as (x,y)-coordinates on a 2-dimentional plane. This is a simple solution, but not the most accurate.

The paper explores several different ways of visualizing the location of geotagged photos on a map. The five visualizations described are called thumbnails, point markers, isopleths, border-dependent and media dots. The thumbnail visualization refers to showing small thumbnails of the geotagged photos at the positions where they were taken on the map. The technique is very visually appealing when the number of photos is low, but it does not scale. When the number of photos increases, it eventually becomes impossible to see the map because of all the photos in front of it, and several photos taken in the same location is impossible to separate. Thumbnail visualization also makes it difficult to read off the position of a photo accurately.

The use of point markers is, according to the paper, the most popular way of showing the location of a photo in a map. It consists of drawing a small dot or icon on the map at the location where each of the photos was taken. The density of dots on the map represents the number of images in that area, and the exact location of each photo can easily be seen. Unfortunately, this has the same problem with scaling as the thumbnail approach, so when the photo collection is large, the map becomes completely covered in markers, and photos taken in the same location will look just like one photo. However, the number of photos that can be

visualized before the map is covered in point markers is much higher than with the thumbnails.

The method called isopleth involves applying a partially transparent overlay on the map. The overlay is colored in different colors representing the varying image densities in the different areas. Isopleth solves the scaling problem, and can be a good visualization for very large image collections, but it is not good when the number of images is low. It also does not convey the fundamental discreteness of the photos.

Another method is to divide the photo collection into separate groups by using borders such as the ones between countries, cities and so on. The number of photos in each area can be visualized by an icon which is scaled according to the photo count. The drawbacks of this method are that it requires a lot of interaction with the underlying geographical map data. It is also a potential problem and that the borders that are relevant at one zoom level may not be relevant at other zoom levels.

The final visualization technique chosen by the authors of this paper is called media dots. The method consists of dividing the map into a grid of 10x10 pixel cells. The number of photos within each area is counted, and a dot with a size logarithmically proportional to the photo count is drawn in each cell. This solution scales relatively well, since several photos are aggregated into one dot. If the photo collection becomes very large, and the photo locations are uniformly geographically distributed, then large portions of the map may be covered, but it will never get as bad as with the regular point markers.

2.4.2 PhotoCompas

PhotoCompas is a photo retrieval system developed by researchers at Stanford University. Several papers describing the research and development of the system have been published. The most interesting papers for this study are [26] and [27]. The papers describe a method to automatically generate meaningful organizations of geotagged photo collections, based on location clustering and event clustering, where events are described as a combination of location and time. It is a method consisting of several steps, and the processing of both time and location data are integrated into the same method. The geotagged photos are first divided based on country borders, because of the assumption that people usually know which country they are in when taking photos. It has been observed that photos are usually taken in bursts [28][29], and therefore the next step in the algorithm is to process the photo collection of each country sequentially and split them into segments. The splitting is done where the geographical or temporal distance between two consecutive photos is larger than a threshold value. A geographical clustering algorithm is then executed to find several different segments of photos that are taken in the same location. Finally, consecutive segments that are located in the same place are merged. The two outputs from this algorithm are the list of geographical clusters from the penultimate step, and the list of segments from the last step representing events. The researchers wanted to eliminate the need for using a map to present the geographical clusters, and therefore an intricate way of naming these has been developed. The naming is based on a geographical dataset of administrative regions in the US. User surveys have shown that the results of this algorithm are meaningful ways of organizing photos.

The researchers behind PhotoCompas have also explored other context data that can be derived from the basic information about location and time. The location data can be used to find the time zone which a photo was taken in, and combined with the timestamp from the

camera the local time of the day can be calculated. This information can then be used to calculate the daylight status, such as day, night, dusk and dawn, and to find out which season it is. The logs from thousands of weather stations around the world are available on the internet, and so the location and time may be used to obtain the weather conditions and temperature for the time when a photo was taken. Some other context data was also explored, and a user survey was conducted to see which kinds of context data users would find valuable to use when searching for specific photos. The results showed that time of day, daylight status and season are elements of a photo that people remember well, but data such as temperature and elevation did not turn out to be very interesting. They also found that information about location is a very important aspect of remembering photos, and that it is as important as the people in the photos.

2.4.3 MediAssist

[30] describes a system for browsing personal photo collections on mobile phones utilizing the context of the photos. The photo collections used in this paper are taken using regular stand-alone digital cameras. The photos are geotagged using separate GPS devices which logs timestamped location data to a track file while photographing. This data is then integrated with the photos when they are uploaded to a PC. In addition to the context data describing time, date and GPS position, which is directly available, several other kinds of information is also derived from this. Most importantly, the location data is used to obtain human readable descriptions of the location. A gazetteer is used to translate the coordinates into descriptions on three levels: country, city/state and town. The location information of the photos is also combined with the timestamp to calculate the daylight status and the season in which each photo was taken. Thousands of weather stations are located all over the world, and both the historical and current data from many of these are available on the internet. The weather condition of each photo can therefore also be obtained by providing location, time and date. This paper describes a system which enables the user to search for photos based on time, date, country, city/state, town, daylight status, season and weather. The photos and context data are made available for mobile phones through a web-interface specially designed to be used on mobile phones. The most interesting result from this paper is the small user survey that was performed to see how long it would take the users to find a specified photo from their own photo collection using the location-based search in comparison to finding the photos only by the time they were taken. The results showed that the time-based approach took 31.2 seconds on average, while the location-based search took 17.9 seconds, i.e. using location was approximately twice as fast. The test group consisted of only 7 persons, but it still indicates that using the information of location may drastically reduce the time it takes to find photos in a large collection.

2.4.4 GeoFoto

Much of the research on the usage areas of geotagged photos revolves around productivity concerns, such as finding specific photos as fast as possible. GeoFoto [30], on the other hand, is an application for recreational use of other people's geotagged photos. The idea is to use a mobile phone to explore the area around you through pictures. The GeoFoto application starts by finding the user's current location from a GPS receiver, and retrieves photos that are geotagged close to that location. The user can then explore other photos by moving north, south, east or west using the navigation keys on the phone. This can be used to explore areas close to where the user is located, before actually going there physically. There is no limit to

how far the user can navigate using the application though, so photos from all over the world may be found using GeoFoto, if it is connected to a photo collection that is large enough.

2.5 PC Software for Geotagging

Several software solutions that utilize geotagging have become available over the last few years, in addition to the scientific research prototypes. There are two kinds of applications that are needed for geotagging to be useful. Firstly, there need to be a mechanism that generates and stores the location of the images. This may or may not be the same program or device that captures the picture. Secondly, one or more programs need to use this data and present it in a useful manner. This section presents some of the currently available software that involves geotagged photos. There are several applications that use geotagged photos on PCs, but not on mobile phones. The mobile phone software that is available is mostly for producing geotagged photos, only very few for managing them.

2.5.1 Flickr

Flickr [31] is currently one of the most popular photo-sharing websites on the internet. Users of Flickr can upload their photos to make them available on the website for other people to see. Restrictions may be applied to the photos so that only your friends and family may view them, or they can be made available for everyone. The users can browse other people's photos by several different kinds of metadata, such as owner, date, category, and so on. The most interesting feature of Flickr in regard to this study is that it supports geotagging. The users can browse a map of the world which shows markers where geotagged photos are taken. Geotagged photos may also be found by searching for places like the name of a city or country, or by browsing photos taken nearby other geotagged photos. When a user has uploaded a photo to flickr, he can choose to view a map where the location of the photo can be indicated by placing a marker. When a photo has been geotagged, it can be found by other users who are browsing the map.

2.5.2 Services from Google

Google has released a plethora of different services and applications for many different aspects of digital life the recent years. Many of these are connected to each other in an intricate and chaotic web. Some of these services and applications involve geotagging of photos.

Panoramio [32] is a website owned by Google that lets users upload photos to be shared with other users and geotag these photos using Google Maps [33]. The photos can then be browsed geographically using either Google Maps or Google Earth [34].

Picasa [35] is a desktop application for organizing personal photo collections, which is also provided by Google. It supports geotagging through the use of Google Earth. Geotagged images may be browsed using Google Earth, and it may also be used to geotag photos without location information by simply dragging the photos onto the map and dropping them on the correct location. A web-version of Picasa, called Picasa Web Albums, is also available, and enables the user to share photos with others. It is possible to browse geotagged photos in the web version also, but then Google Maps is used in stead of Google Earth. The photos are shown as thumbnails on the maps both in Google Maps and Google Earth.

Photos that have previously been geotagged, either automatically or using other software, will usually have the location information stored in the EXIF header in the image file. Both Picasa and Panoramio are able to import the location from this header so that it is not necessary to apply new geotags in this software.

2.5.3 Microsoft Pro Photo Tools

Microsoft Pro Photo Tools [36] is a desktop application for managing the metadata of image files. When images are loaded into the application, several different kinds of metadata are also loaded from the files and presented to the user. This metadata can then be viewed and, if desired, it can be changed and written back into the files. The kinds of metadata supported includes a description of the camera and camera settings such as shutter speed, ISO-value and flash mode, a list of the people in the photo, description of the photographer, different categories, a title, rating and keywords, as well as a longer textual description of the image. It also supports date and time from several different headers in the image files, and most importantly for this study, it supports geotagging. The geotags can be read from and written to the EXIF header in the image files, they can be generated from a GPS track file or they can be created by locating the photos on a map embedded in the application. The application can be used to obtain human readable location data such as country, state/region, city and street address for images where a GPS position is available. In the absence of a GPS position, the human readable location data can be used to obtain the latitude and longitude. The collection of geotagged photos can be visualized in a map using point markers. As described in [2], the use of one point marker for each photo does not scale very well, and is bad for representing several photos taken at the same location.

2.6 Mobile Phone Software for Geotagging

Most geotagging applications for mobile phones handle the task of geotagging photos taken with the built-in camera and uploading them to either a PC or a website. There are several applications available for this, and some of them will be presented here.

Shozu [37] is an application for using social websites such as YouTube [38], Facebook [39], flickr [31], blogger [40], picasa [35] and more from a mobile phone. One of the features of Shozu is that mobile phones with GPS receivers can be used to geotag photos and automatically upload them to relevant websites.

Locr [41] is a mobile application specifically for capturing geotagged photos and uploading them to the locr website. Users of the locr website may browse the photos on a map, where each photo is represented by a point marker.

Some of the newer GPS-enabled Nokia phones, such as N82, have native support for geotagging. The geotagged photos can be uploaded to Nokia's own website for sharing mobile media called Share on Ovi [42], or they may be viewed on the phone using Nokia Maps 2.0. This is one of the very few possibilities of actually using the location information on the phone itself.

2.7 Summary

This chapter have introduced the concept of geotagging, presented the WGS84 reference system for representing locations and described several different technologies for obtaining

information about location. GPS and AGPS are the most practical solutions to use for automatic geotagging, although others systems such as cell-ID or WLAN-positioning are also possible.

The related research presented here indicates that the use of location information can be very helpful when trying to find photos in large collections. It has been shown that searching for photos in large collections using location terms such as country and city may be approximately twice as fast as traditional time-based browsing. Researchers from Microsoft Research reviewed several different ways of visualizing photo locations in a map, such as thumbnails, point markers, isopleth, border-dependent and media dots, and their pros and cons have been described.

Several different software solutions for geotagging, both for creating, manipulating and visualizing location information, which is currently available, have been described. Desktop applications and websites, as well as mobile applications have been covered.

3 Android

Android [1] is a new platform for mobile phones, which is currently in the last stages of initial development. It is likely to gain popularity in the high-end market when the phones are released. Phones in this market segment are often called smartphones, to indicate that they are capable of much more than just a regular mobile phone, such as reading and writing e-mails, browsing web-pages, using office applications and running different kinds of third party applications. The currently dominant platforms for smartphones are Symbian OS [43], Microsoft Windows Mobile [44] and RIM BlackBerry [45]. The Android platform is being developed by Google and their partners in the Open Handset Alliance (OHA) [46] as a free and open alternative to these other proprietary systems.

The Android platform is a good platform for geotagging applications, as it includes both a customizable version of Google Maps that can be used for visualizing locations, a Location API for obtaining the phone's current location and it is a very open system with few limitations. Android is being developed to natively support mobile phones with a touch screen, which makes it much easier to handle a map, but it can also run on more regular phones without touch screen.

3.1 History

It is just a few months since the Android platform was first announced to the public. However, there have been speculations of Google entering the mobile phone market for several years. It all started when Google's vice-president of operations Urs Hoelzle did an interview with Siliconrepublic [47] in December 2004, where he stated that "the mobile phone market is failing to grasp the potential of the internet." One of the first reactions to this interview was by blogger Gary Price [48], writing "I wonder if GPhone is in the works."

The rumor of Google looking into the mobile phone business soon spread throughout the computer press, and the nickname GPhone was widely adopted. Several events in the coming years caused the rumors to re-appear in the press, but Google refused to say much about it. One of these events was that Google acquired a small start-up company named Android Inc. in August 2005 [49]. Not much was known about this company, other than that they were making software for mobile phones, but there were rumors that they were making an operating system for mobile phones. Other events, such as Google filing patents for mobile technologies and making business deals with companies in the mobile industry, all lead to speculations in the press of whether a GPhone was coming.

Although there were a lot of news articles written about Google's role in the future of mobile phones, it was all mostly speculations until 5 November, 2007. At that date the Open Handset Alliance (OHA) and the Android platform were announced [50]. The OHA is a group of 34 companies, including Google, T-Mobile, HTC, Qualcomm, Motorola and other companies in the mobile industry. Both software and hardware manufacturers, as well as mobile operators and commercialization companies are represented. The stated goal of the group is to "foster innovation on mobile devices and bring consumers innovative new mobile experiences." This group works together to bring Android-based mobile phones to the market.

At 12 November, 2007 the OHA released an early look of the Android Software Development Kit (SDK) [51]. This included an emulator for testing Android applications on a PC, which was made available for Windows, Mac and Linux. Several updates to this SDK has since been

released and made available for download from the official Android website [1]. There has currently not yet been released any mobile phones based on the Android platform. Therefore all the applications that are being developed for Android by programmers all over the world are still based on the emulator. According to [52], the first Android phones will be released in the fourth quarter of 2008.

3.2 Android Developer Challenge

One of the most important aspects of Android, which is supposed to set this platform apart from all the other mobile platforms available, is that it is very open and free from restrictions. This is supposed to make Android a platform for innovative solutions, and the availability of many third party applications is necessary to prove this point. It is therefore important for the OHA that there already are lots of third party applications ready when the first Android-based phones are released in the market. The early preview release of the SDK and the emulator was supposed to give people good time to get started on making Android applications well before the first phones are out.

Writing mobile applications that can only be run on an emulator might be a bit boring, so to further spark the enthusiasm for the Android platform the OHA announced the Android Developer Challenge (ADC) [53]. ADC is a competition to make the best software for Android phones, with a total of 10 million USD in price money to be split between all the winners. The ADC is split into two competitions: one competition for applications developed on the emulator, called ADC1, and another competition, called ADC2, which will be held after the first Android phones are available. The applications submitted to the ADC are examined by a total of more than 100 judges chosen by the OHA, most of which are employees of the OHA member organizations. The official judging criteria are originality, effective use of the Android platform, polish and appeal and indispensability. ADC1 opened for submissions 3 January, 2008. The deadline was 14 April, 2008. A total of 1788 applications were submitted. On 12 May, 2008 a list of what the judges thought was the 50 best applications were published at the official Android Developers Blog [54]. These 50 competitors got a little bit of the price money and the chance to further develop their application before the grand price winner is announced July 2008.

3.3 System Architecture

The Android platform is a complete software stack, reaching from the most low level modules that controls the hardware and up to the user applications. It has been described at the Android website [1], and further detailed in a presentation held at the Google I/O conference in May 2008 [55]. The platform is based on the Linux kernel and incorporates several open source software packages, in addition to libraries and applications developed specifically for Android. All the Android-specific code is planned to be released under the open source Apache License, Version 2.0 [56] when the first phones are released to the market. The system architecture has been divided into several layers, going from the operating system kernel at the bottom, to the user applications at the top, as shown in Figure 3.1.

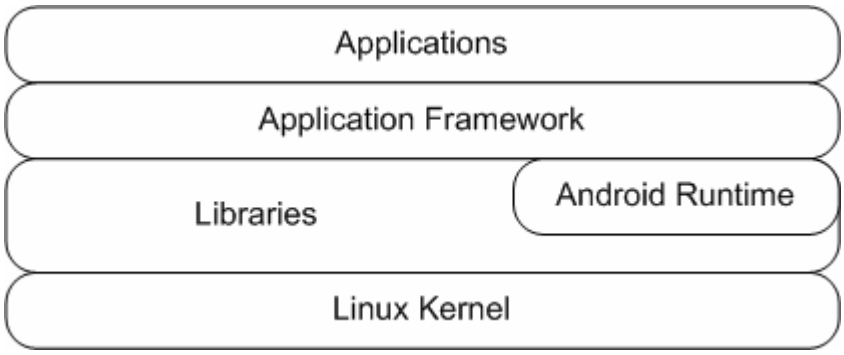

Figure 3.1: The layered architecture of the Android platform.

As mentioned, the Android platform is based on a Linux kernel, with a few enhancements that makes it more suitable for Android devices. The kernel provides basic capabilities such as memory management, process management, security features and a driver model. Although the Linux kernel has many features that fit perfectly with the needs of Android, there are some elements that are more optimized for PCs and servers than mobile devices. Mobile devices generally have slower CPUs and less memory than PCs and servers, and they have a limited battery capacity. One of the enhancements that have been made for Android is the power management. It is based on the standard Linux power management, but has been made more aggressive in Android. For instance, the CPU is turned off when not needed to avoid draining the battery. It is possible to override this though, when applications need the device to be always on. Applications can obtain either a partial WakeLock for certain parts of the device, such as the CPU or LCD screen, or a full WakeLock for the complete device. Another thing that had to be changed for Android has to do with the low amount of memory that mobile devices usually has. The system designers worked with a design criterion of low-end Android devices having approximately 64MB RAM, and only 20MB being available to applications after the system is loaded. In additions to this, there is no swap space available, so the physical RAM is all that there is. It was therefore necessary to include a module that has been called the Low Memory Killer. The task of this module is to kill the least necessary processes when the system is low on memory. This will typically be applications the user is no longer actively using, but are still running in the background. The constraints of having little memory and a slow processor also showed the need for the development of a new InterProcess Communication (IPC) system that has been called the Binder. It is a lightweight IPC solution that maintains high performance by using shared memory instead of passing data around. There are also a few other smaller enhancements that has been done on the Linux kernel, all of which are available at the Android kernel Git repository [57].

On top of the Linux kernel sits the low level libraries, which are usually written in C or C++. These include a surface manager for the display, an audio manager and an implementation of the standard C library. A media framework is available that supports processing of standard video, audio and image formats in addition to codec plug-ins. OpenGL|ES is included for creating 3D graphics. The transactional database engine called SQLite is included, and will be covered in more detail in section 3.5. The WebKit engine for processing web pages is also a part of the library layer, as well as several other libraries. These libraries will not be used directly by the applications, but through other more high level libraries in the application framework.

The part of the platform that is called the Android runtime consists of the core libraries and the Dalvik Virtual Machine (Dalvik VM). The core libraries are an implementation of most of the functionality in Java Standard Edition (Java SE) [58]. The main things that are missing are

all the GUI components, since Android provides its own libraries for this. A few other smaller features are also missing. The Dalvik VM is a virtual machine, developed specifically for Android, which executes bytecode in the Dalvik Executable format. The Dalvik VM is described in more detail in section 3.6.

The application framework is the collection of all the Android-specific libraries that are available for the applications. These come in addition to the standard Java libraries in the core libraries. The application framework contains libraries for building an Android GUI. It contains libraries for all sorts of telephony-related operations. It has a Package Manager for managing the installation of software packages, an XMPP Service for sending messages over the internet and ContentProviders for sharing different kinds of data between applications. It also provides several hardware services for controlling hardware such as Bluetooth, WiFi and USB, and special sensors such as compass and accelerometers. It also has a LocationManager, which provides a unified interface for obtaining the phone's current location from different technologies. The location may be obtained by any of the technologies described in the previous chapter, or by some new technology, and published to the application through the same LocationManager interface.

On the top of the Android software stack are the applications. These include everything that is visible to the users on the screen of the phone, in addition to background services that will only be accessed through other applications. Some of the most basic applications will be part of Android, such as the home screen, the contacts manager and the phone application, but the real power of Android is the great support for third party applications. The applications that are included uses the exact same API as the third party applications have access to, and even the built-in applications may be replaced to create an enhanced and more customized user experience.

3.4 Developing Applications for Android

Android is intended to be an open platform, where the users can run any applications they want. The great support for third party applications is one of the most important aspects the OHA is talking about when they presented Android. The applications that are developed for the Android platform are written in the Java programming language, but it is not a limited version such as Java Micro Edition (Java ME) [59], which is available on most mobile phones today. It does not adhere to any of the common Java standards, but it implements most of the functionality in Java SE. The standard GUI components of Java SE are missing, as well as some other minor features. Instead Android provides its own set of library functions for creating GUIs, in addition to lots of other useful features.

The Android Software Development Kit (SDK) provides a plug-in for developing Android applications in the Eclipse Integrated Development Environment (IDE) [60], a toolchain consisting of all the tools necessary for building Android applications without using Eclipse, an emulator for testing the applications on a PC, and some other tools. Eclipse is one of the most popular IDEs for developing Java applications, but it also supports a lot of other languages. It is an open source project, released under the Eclipse Public License, so everyone can download the software and use it for free. There is also a plethora of plug-ins available for Eclipse, which helps the developers to be more productive. Eclipse is written in Java, and is therefore available for most popular PC operating systems. The Android SDK is available for Windows XP and Vista, Linux and Mac OS X (on x86 processors only).

The Android applications are compiled into Java bytecode with a standard Java compiler, but there are more steps necessary to make them into executable Android applications. First, the .class-files created by the Java-compiler needs to be converted into Dalvik Executables, called .dex-files. This is a special bytecode format for mobile devices. Then, the .dex-files have to be packaged into Android Packages, called .apk-files. Finally, the .apk-files have to be transferred to the Android device, or the emulator. When using Eclipse, all of this happens automatically, but the SDK also provides separate tools for each of these steps. The developers are therefore able to perform these steps manually, or use other IDEs than Eclipse by integrating the Android toolchain.

All development of third party Android applications is currently done on a PC-based emulator, since there are no actual mobile phones running Android available yet. The emulator is described in more detail in section 3.8.

Android applications are built up using several different basic elements. These are Activities, IntentReceivers, Services and ContentProviders. All applications must use at least one of these elements.

The most basic element is the Activity. An Activity in Android corresponds to what usually would be called a screen in an application. It could be something like “Write an SMS”, “Pick a contact from the contact list” or “Show the video with filename xyz.avi”. All applications with a user interface that is visible on the screen need to have one or more Activities. The user interfaces of Activities consist of one or more Views, which roughly corresponds to what might be called user controls in other programming environments. There is for instance a TextView for showing text, a ListView for showing a list and an ImageView for showing images. How the Views are arranged on the screen is controlled by a Layout, which can be specified either in Java code or in XML. More information on how to create the layouts for the activities can be found in the Android reference documentation.

An Intent is a statement of what the application wants to do. For instance “I want to write an SMS”, “I want to pick a contact from the contact list” or “I want to show the video with filename xyz.avi”. Most Activities are started by an Intent, and user actions in an Activity can trigger other Activities in the same application, or in other applications. An Intent can either be explicit, i.e. directed to one specific software module, or it can be implicit, i.e. broadcasted to the whole system. The implicit events are filtered through a set of intent-filters, which will in turn trigger a set of IntentReceivers. IntentReceivers are entry-points into the different software packages for handling broadcasted Intents. Intents and IntentReceivers are somewhat similar to events and event handlers, but Intents and IntentReceivers captures the concept of “I want xyz to happen”, instead of “Xyz has happened”. By changing the intent-filters, the Activities that are launched to handle certain Intents can be replaced. Even the built-in Activities, such as browsing for a contact in the contact list, can be switched for a more customized application.

Some applications need to run in the background and do their work, without any screen interaction. This can be e.g. music players or applications that need to periodically look for updates over the network. This sort of behavior can be implemented by creating a Service. Services will typically be invoked and configured through an Activity.

The security features of the Android platform, described in more detail in section 3.7, prevents applications from accessing the files, databases and other resources belonging to

other applications. Instead, the concept of using ContentProviders to share data between applications has been introduced. By implementing a ContentProvider the applications can share specific parts of their data with the other applications and control the access to it. ContentProviders can be built on top of databases, files or any other resource that the providing application has available. Other applications can access the data by using a URI on the form content://<authority>/<path>/<ID>. The <authority> part is the fully qualified class name of the ContentProvider being accessed. The <path> part specifies what kind of data that is being requested. The <ID> part, which is optional, is used to specify one single item, as opposed to accessing the whole data collection. There are several built-in ContentProviders in the Android platform, for instance for accessing the data in the contact list. These are accessed in the same way as third party ContentProviders, except that the <authority> part is not a fully qualified class name, but rather a short and descriptive name, like “contacts”. One small problem with using ContentProviders comes up when trying to combine data from different ContentProviders. If the application that consumes the ContentProvider had direct access to database tables instead, this could be done by a simple join operation, but with ContentProviders this has to be done in the application code.

3.5 SQLite

Android has a built-in relational database engine called SQLite [61]. This is available for any software installed on the device, and is one of the preferred ways of storing user data in Android. It is also used by several internal services in Android, for instance the address book. A SQLite database created on an Android device is only available to the application that created it, because of the security mechanisms implemented in the platform, as described in section 3.7. If the data is made available to other applications on the device, it has to be published through a ContentProvider.

Most of the database systems people are used to today are using the client/server model, as shown in Figure 3.2 and Figure 3.3. Examples of this kind of systems are Oracle Database [62], Microsoft SQL Server [63] and MySQL [64]. In these systems the database management system is running as a separate process, and the client applications connect to this database server either locally on the same computer or through a network connection. SQLite is instead embedded as a module in the application that is using the database, as shown in Figure 3.4, so it reads and writes directly from the database files in the file system. SQLite is embedded in several widely adopted software applications such as Mozilla Firefox, Skype, McAfee antivirus, Sun Solaris 10, Mac OS, the iPhone and newer versions of Symbian smartphones.

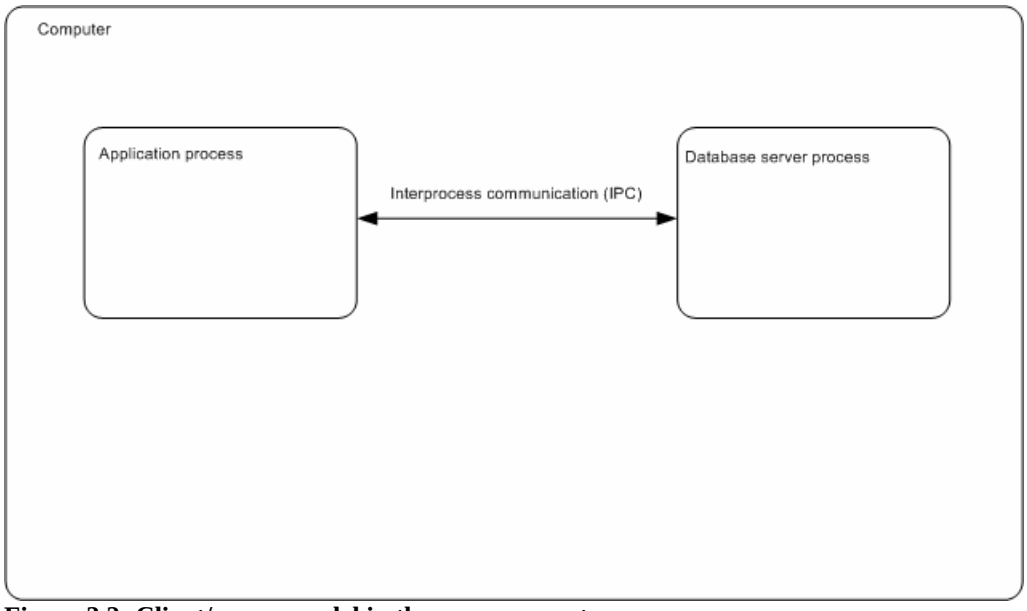

Figure 3.2: Client/server model in the same computer.

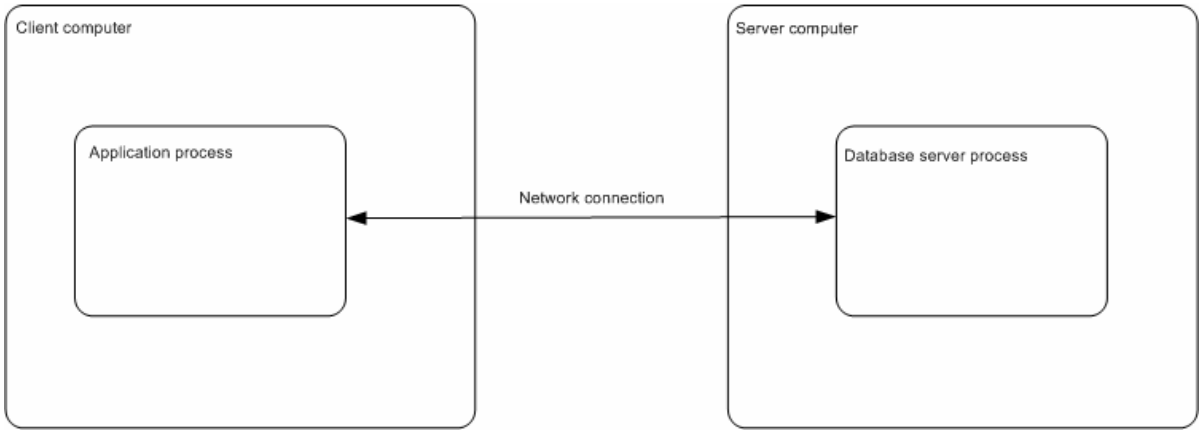

Figure 3.3: Client/server model using two different computers.

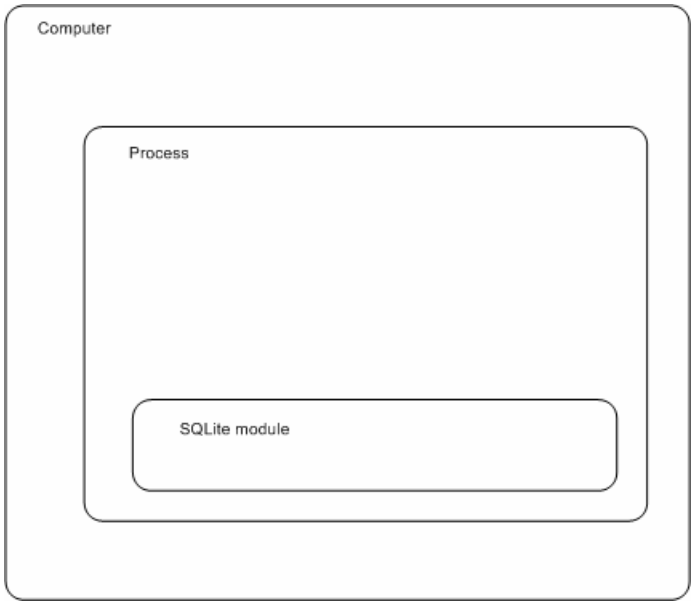

Figure 3.4: The embedded model, used by SQLite.

SQLite is a software library that can be embedded in any software that needs to store structured data. The source code of SQLite is in the public domain, so anyone can use it for whatever they want. The system is programmed in ANSI-C and does not depend on any external libraries. It can easily be recompiled to run on any platform that has a C compiler available. This is probably the most important reason for why SQLite is believed to be the most widely deployed SQL database system [65].

The four basic features that any transactional database system should have are Atomicity, Consistency, Isolation and Durability (ACID). Although SQLite is considered a transactional database system, it only supports three of these four features. Atomicity, Isolation and Durability are supported. SQLite does not enforce foreign keys, and therefore the Consistency feature is not supported. Foreign keys must be enforced by the application that SQLite is embedded in, if they are needed.

The query language used in SQLite is mostly compliant to the SQL92 standard [66], but there are a few missing pieces. The most important missing feature is probably that views in SQLite are read-only, therefore all updates and inserts has to be done directly to the tables, but there are also a few others in addition to this. For instance the ALTER TABLE command can only be used to rename tables and add columns, and is therefore incomplete. Another problem is that the support for foreign key constraints is missing. There is also no support for nested transactions, and right and full outer join is missing, even though left outer join is implemented. There are a few other small shortcomings that also prevent SQLite from being fully SQL92 compliant, but these are the most important ones.

3.6 Dalvik Virtual Machine

The details of the Dalvik Virtual Machine (Dalvik VM) were presented at the Google I/O conference in May 2008 [67]. It is a bytecode interpreter written exclusively for Android that executes all the Android applications written in the Java programming language. It does not execute Java bytecode though, but a special bytecode format optimized for mobile phones. The Java code first has to be compiled into regular Java bytecode, and then the .class- and .jar-files can be converted into Dalvik Executables, known as .dex-files, with a tool included in the SDK.

The Dalvik Executables are created to work in an environment with low system resources. The system memory may be as low as 64MB, and after the Android system is loaded, as little as 20MB might be available for the applications. There is also no swap space. The processor in these devices are usually slow, the operating frequency may be approximately 250 – 500 MHz. All mobile phones have limited battery power, and every processor instruction consumes power, so the number of instructions needs to be minimized. These requirements are different than on a PC, and therefore the regular Java bytecode is not optimized for this.

The Dalvik VM on the other hand, optimizes the bytecode to make it more suitable for the limitations of low system resources. The result is that a .dex-file takes less than half as much memory as the corresponding .class-file and has 30% fewer processor instructions, according to the developers own numbers, published in the presentation.

For the curious: The Dalvik VM is named after a place called Dalvík on Iceland.

3.7 Security

At first glance, the Android platform might look a bit insecure. The openness of the platform has been emphasized, and it has been pointed out that users are supposed to be able to run whatever software they want. Therefore, there is no certification process for Android applications, which exists for most other popular mobile platforms, to make sure that the applications do no harm to the system. When you look a bit closer under the covers, though, there are several security mechanisms that are supposed to prevent third party applications from doing too much damage to the system and to other applications.

First of all, Android applies the traditional Linux user model in a creative way. The developers realized that there is no need for several separate users on a mobile phone, since it is mostly used by just one person anyway. Instead, the user model could be used for sandboxing each application on the system. Each time a new application is installed on an Android system, a new Linux user is created for it. The underlying Linux file system permissions make sure that all the files and databases belonging to one application are kept private and inaccessible for anyone else. If data is to be shared between applications, this has to be done explicitly, for instance by implementing a ContentProvider.

Another security feature is the permissions system. Most of the critical features in the system are protected by this, so that all applications need to obtain permission before accessing them. Examples of actions that the applications need permission to do are making phone calls, accessing a GPS receiver and deleting installed software packages. An application's permissions are requested at install-time, and the user will be prompted to grant or reject each permission request separately. The idea is that the mobile phone user is supposed to use some common sense when granting permissions to avoid malware on the phone. For instance, it would be weird if a calculator application requests write access to the contacts list. Once an application is installed, it will have the granted permissions until uninstalled, without any further user interaction.

3.8 The Emulator

There are currently not being sold any devices running Android. As mentioned earlier, the first Android-based mobile phones are expected to be released in the fourth quarter of 2008. Android has been hacked onto a few devices that are currently in the market, but this is not supported by the handset manufacturers. All development of Android applications is therefore still based solely on using the emulator on a regular PC.

Most mobile phones today are based on a processor with ARM architecture, but most PCs have a processor with x86 architecture. The Android platform is naturally targeted to the ARM architecture, so to be able to run Android on a regular PC an ARM processor has to be emulated. The Android emulator is based on the open source processor emulator QEMU [68], which is released under the GNU General Public License. QEMU supports several different processor arcitectures, including ARM. The Android developers have used QEMU as a base and added a nice GUI for it, as shown in Figure 3.5. They have also provided a set of tools for making it easy to deploy, run and debug Android applications on it. The developers can interact with the emulator by using the PC keyboard, or by clicking on the virtual keyboard or phone buttons in the emulator GUI. Android has built-in support for touch-screens, so the virtual phone screen in the emulator can be interacted with by using the mouse.

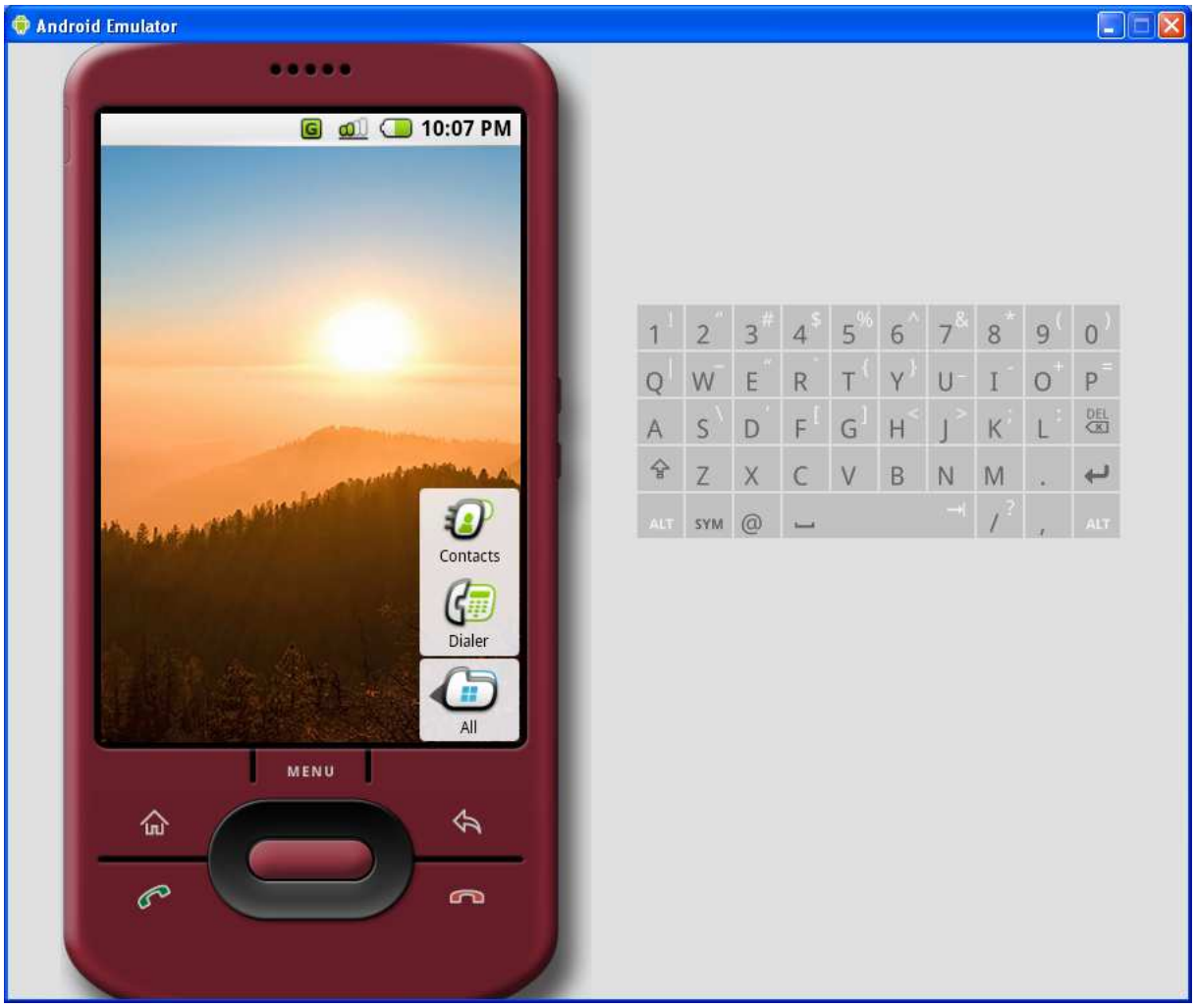

Figure 3.5: The GUI of the Android emulator running on Windows XP.

Although the API in the emulator is the same as in real phones, the results of interacting with anything outside of the phone are different from a real phone. For instance, images from the camera only shows a square in front of a chess pattern, as shown in Figure 3.6, and the location data obtained from GPS will always be in the San Francisco Bay Area.

The emulator is available for Windows XP and Vista, Linux and Mac OS X, just like the rest of the SDK.

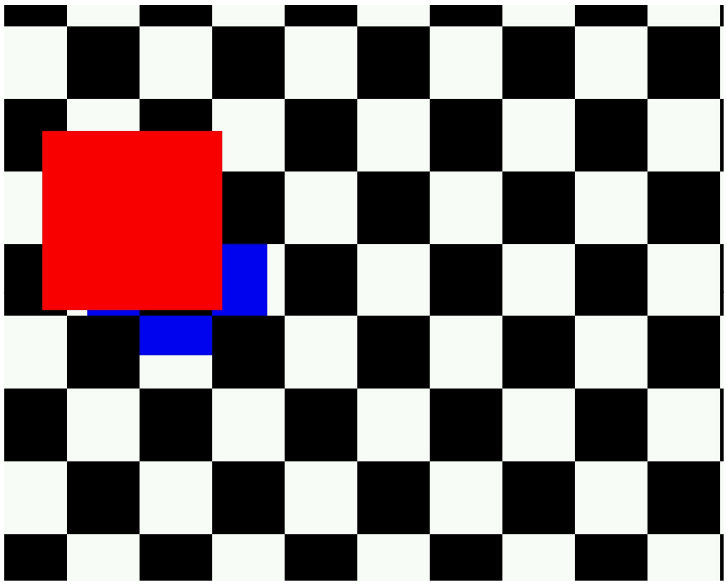

Figure 3.6: The image returned by the camera driver in the Android emulator.

3.9 Summary

Android is a mobile platform for smartphones, being developed by Google and the Open Handset Alliance (OHA). There were rumors of Google entering the mobile phone market for several years, but they remained silent until November 2007 when the OHA released an early look on the SDK. A programming contest involving large prices was also launched to gain the interest and attention from developers.

The Android platform is based on a Linux kernel, and incorporates several other open source systems, such as the 3D graphics library OpenGL|ES and the database engine SQLite. It has a virtual machine called Dalvik, which executes Dalvik Executables. These can be created from regular Java class files. The Core Libraries in Android implements most of the functionality in Java SE, but not the GUI components. The Application Framework of Android provides all the Android-specific libraries, such as GUI-libraries and the Location API, as well as libraries for accessing the underlying native libraries mentioned earlier. The Android applications are written in Java, and use both the Core Libraries, as well as the Application Framework.

There are currently no phones available for the Android platform, so all development is done on the emulator that has been released as a part of the SDK. The emulator is available for Windows XP and Vista, Linux and Mac OS X. It is a convenient way of testing applications, but it has shortcomings with most of the functionality that involves the world outside the phone itself.

4 Applications of Geotagged Photos

The location information attached to a geotagged photo can be used in a lot of different ways, only limited by the developer's imagination. Some applications are very obvious, others are more sophisticated. The most interesting applications will probably emerge from combining different sources of information.

Several different ideas for mobile applications using the location information in a geotagged photo will be explored and described in this section. Some of them are well tested concepts that are widely available throughout the internet, but may not be as widespread in the context of a mobile phone. Others are more original ideas. Most of the concepts discussed here are applicable both for personal photo collections, and for browsing other people's photo collections. How the photo collections are retrieved is not discussed here.

4.1 From Photo to Map

Let's start with one of the most basic uses of a geotagged photo: Show the location of where the photo was taken in a map. This scenario starts out with the user browsing through either all or a subset of the photos on the device. When the user has selected a specific photo, he can choose to view a map where the exact location of where the photo was taken is visualized by a marker. This function can be available by the press of a button, or via a choice in a context menu.

The usefulness of this feature will vary largely based on the user's pattern of photography. It is probably most useful when browsing photos that you did not take yourself, but it can also be valuable when the user do not know or remember where he took the picture. If the user only takes very few pictures, always uploads the photos to his PC and organizes them soon after taking them, then this kind of application will be mostly useless since he will most likely remember where his photos were taken. A very different example may be a user that has been on a long trip, taking lots of pictures along the way. When he comes back home he might not remember where all of the pictures were taken, so seeing this in a map would probably be rather helpful to him. Another useful scenario is if the picture was taken a long time ago, so the photographer might have forgotten where it was.

The high-level steps of the algorithm needed to implement this functionality are not very advanced. The algorithm is as follows:

1. Read the coordinates of the photo into memory.
2. Translate the latitude and longitude into (x,y)-coordinates of the map.
3. Center the map at the (x,y)-coordinates.
4. Adjust to an appropriate zoom level.
5. Draw the marker indicating the (x,y)-coordinates.

How difficult each step is to implement depends on the development platform. Some platforms may have lots of supporting library functions that can be used, while others may provide little support for this. The details on how to implement this on the Android platform is described in section 5.2.6.

4.2 From Map to Photos

A more sophisticated way of visualizing the photos in a map is to show a marker for all the places where the user has been taking photos. The user can then select a specific marker either by clicking on it if the phone has a touch screen, as the emulator has, or by navigating to the marker by using the d-pad or joystick on the phone, as almost all newer phones has. When the user selects a specific marker, he can choose to view and browse through the subset of his photo collection that is taken in that specific location.

The primary use of this application is when the user is searching for a photo from a specific place where he knows that he took a picture or a specific picture that he knows where he took. If the photo collection is large and not well organized, this way of finding a photo might be much faster than browsing through the whole collection sequentially.

As described in section 2.4.1, [2] discusses several methods for visualizing photo locations on a map. A quick recap of these and an evaluation of how well they will work in the context of a mobile phone are necessary here. The thumbnail visualization, where a thumbnail of each photo is drawn on the map in the location where it was taken, was considered not scalable when used on a PC. This effect is drastically increased when used on a mobile phone with a much smaller screen than a PC, since each thumbnail would have to take up a large portion of the screen if it were to be viewable at all. Using point markers, i.e. drawing small dots or icons on the map where each photo was taken, was said to be the most popular visualization technique. The problems with this were that photos taken at the same place would be rendered indistinguishable, and it also does not scale, although it can handle larger photo densities than the thumbnail approach. These problems may be somewhat increased on a smaller mobile phone screen, but not nearly as much as when using thumbnails. The paper describes a method of applying a partially transparent overlay on the map, with varying color to indicate the different photo densities in different areas. This method is not applicable for this application, as it would be very hard to implement a usable interface for selecting photos from a map with no markers. Dividing the photos by man-made borders is discussed, but it has the drawbacks of requiring much interaction with the underlying map data, and that which borders that are relevant will vary between the different zoom levels. The authors of [2] ended up using what they called media dots. The method involves dividing the map into 10x10 pixel cells, and drawing a dot in each cell which is logarithmically scaled to match the number of photos taken in that area.

Even though [2] sees media dots as the best solution, this approach still has some problems. The 10x10 pixel cells are a very artificial way of dividing the photos, and do not represent the inherent structure of the locations in the photo collection. Photos taken at approximately the same place, but with small variations may be split into several adjacent cells, and mixed with photos taken further away. The use of dots in different sizes may also not be a good fit for small mobile phone screens, where it may be difficult to see the differences.

Instead, the method used in this study takes advantage of the fact that photos are usually taken in bursts, see [28][29]. Several photos taken during a relatively short time span, will usually also have to be in a relatively restricted area (unless they are taken while traveling in a car, train, plane or something similar). This structure can be taken advantage of by grouping photos taken close to each other together. A single point marker can then be used to indicate the average location of a whole group. This approach will mitigate the problem with scaling in the other methods, since there is no need for a dedicated pointer for each photo.

One important challenge in creating this application is to automatically divide the photo collection into groups of pictures that are taken geographically close to each other. This is a kind of problem that in computer science is usually referred to as data clustering. It is a very generic method that can basically be described as dividing a set of objects into groups of objects that have something in common. The property or properties that the objects in a group have in common may be any kind of measurable attribute. The value of this property does not necessarily have to be exactly the same for all the objects in a group, but it must be close enough so that any group may be distinctively separated from all the others. An example of 12 objects divided into three clusters, determined by two different properties, is shown in Figure 4.1. The threshold for how much variance that is accepted within a group is determined by the context. The difference in value between any two objects is calculated using a function called the distance function. This function is defined explicitly for each new problem to fit the particular needs of the application. Thus, a very generic clustering algorithm may be used on any kind of data by simply replacing the distance function.

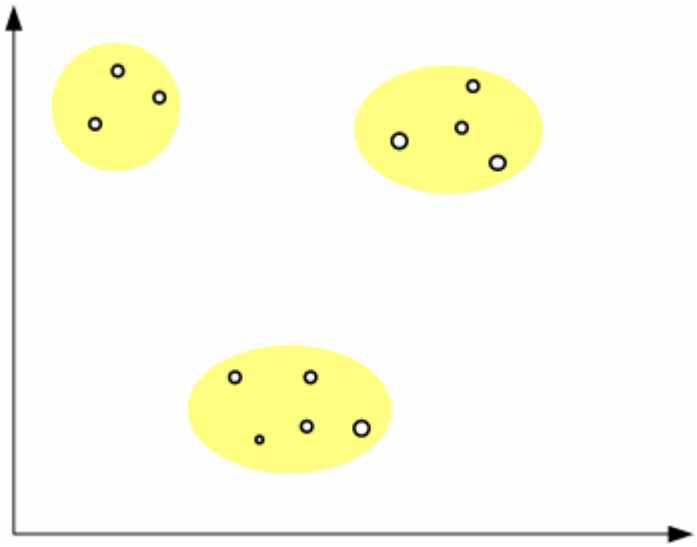

Figure 4.1: 12 objects clustered into three clusters by using two parameters.

The most common clustering algorithms can roughly be divided into hierarchical clustering and partitional clustering. There are other, more specialized clustering algorithms that don't fit into these categories, but they are not relevant for this study. The main difference between hierarchical and partitional is that the partitional algorithms try to divide the data set into a pre-defined number of partitions, whereas with the hierarchical algorithms the number of clusters may be decided after executing the algorithm or determined based on a threshold on the distance function. Hierarchical clustering was first described by in [69].

Hierarchical and partitional clustering are both very useful algorithms, but they are used to solve different kinds of problems. In this application, the number of places where the user has taken photos can not be determined before actually running the algorithm, and therefore a hierarchical clustering algorithm must be used. The number of clusters returned by a hierarchical clustering algorithm can, as mentioned earlier, either be decided directly by cutting the tree at a specified level, as shown in Figure 4.2, or it can be determined by a threshold on the value returned by the distance function. In this application the method of using a threshold on the distance is the method of choice, since the actual number of clusters is not important.

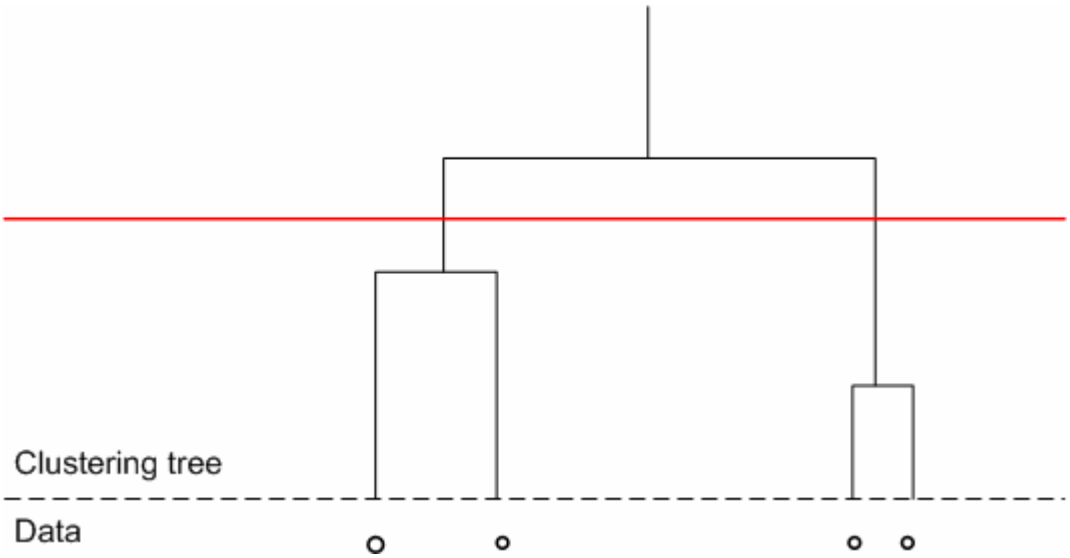

Figure 4.2: Hierarchical clustering of four objects.

The threshold value used on the distance function in this application must not be a static value, but determined by the current zoom level of the map. When the map is zoomed all the way out, so that the whole world is visible, it would be very counter-productive to have one marker for each photo. It would be impossible to select the correct marker unless the photo collection consists of just one photo for each country. For this zoom level, a very high threshold value must be used to make sure that the number of groups is not too high. When the map is zoomed further in, so that e.g. just one city is visible, the threshold value must be much smaller, so that pictures taken in different parts of the city can be divided into different groups. The composition of the groups must therefore be computed for each zoom level.

The algorithm of agglomerative (bottom-up) hierarchical clustering used in this application can basically be described as follows:

1. Create a separate group for each single point in the data collection.
2. Merge the two groups with the shortest distance between them.
3. Repeat step 2 until there is only one group or a threshold on the distance is reached.

The most important step in customizing a clustering algorithm to a specific problem is to define the distance function. In some situations this can be easy, but it can be harder or almost impossible in other situations where several different properties which are not directly comparable are involved. In this application, where the only two parameters are the latitude and longitude of the geographical coordinates, it is relatively easy, but some care must still be taken. The easiest and most inaccurate way of calculating the distance is the naïve Euclidean interpretation of the coordinates. The latitude and longitude are simply seen as coordinates in a 2-dimensional Cartesian coordinate system, so the Euclidean distance between two points is trivial to calculate. Because of the curvature of the meridians, which meet each other at the poles, this is an inappropriate way of calculating distance. It will be fairly accurate for small distances near the equator, where the curvature can be negligible, but can not be used as a general distance measure. [70] tested the naïve Euclidean method to calculate the distance between two points in Colorado which was known to be 741.7 km apart. The naïve Euclidean method yielded a distance of 933.8 km, approximately 25% more than the actual distance. Another simplified way of calculating the distance between two points is to treat the earth as a sphere, and use the method of great circle distance to calculate the destination, but this is not accurate either. To accurately calculate the distance between two points on earth, the WGS84 ellipsoid, which was described in section 2.2, has to be used.

When the groups for the current zoom level has been defined, they must be visualized on the map in some way. The easiest way of doing this is to draw a dot or a small icon at the average location for each group. Since the photos have been aggregated into groups, this will not pose any major problem with scaling. The method described earlier of using different sizes of the dots to indicate the number of photos in each group is just as applicable to this way of creating the groups as it is for the 10x10 pixel cells used with media dots, but it may still be hard to distinguish between the different sizes on a small mobile phone screen. Another method of showing the number of photos in a group could be to draw a small circle with the number written inside on the average location of the group. The circles might have to be relatively large though, for the user to be able to read the number.

When the groups are created and a marker is drawn on the map, it is relatively easy to make the application show all the photos in the group when the user selects a marker on the map. If the user zooms close enough, the markers may indicate just one single photo, since small variations in coordinates is likely to occur. The user will then be presented with only that single photo when the marker is selected.

4.3 Who Lives Here?

The use of maps is probably the most obvious way of utilizing location information, but there are a lot of other ways of using this information as well, that may be even more interesting and useful. One such application is to show the user which of his contacts that lives where his photos were taken. When the user can't remember where a photo was taken, and it's not possible to find out by looking at the photo, he can easily find out if it was taken where some of his contacts live by the press of a button or a choice in the context menu. This serves the same purpose as showing the location of where a photo was taken in a map, so both applications may be used when the user wants to find out where a photo was taken.

Even though this application covers the same need as the map application described in section 4.1, the format of the answer that the user gets is significantly different. Which of these applications that is most useful for the user is determined by the situation. The application described here is completely useless for photos taken while hiking in the woods, but in that situation the map application is an appropriate presentation of the location. Photos that are taken while visiting one of your friends, on the other hand, are the ones that will be suitable for this application, and the name of the person who lives there is expected to be much faster and easier to interpret for the user than a map with a marker on it. The situation where this application will be most useful though, is probably when looking at pictures taken by someone else, since the user then will have no way of knowing where the photo was taken.

The location of where a geotagged picture was taken will be stored as latitude and longitude, but it can not be expected that the user will bother to store the coordinates of where his contacts lives manually. The location of where your contacts live must in some way be automatically translated into latitude and longitude coordinates that can be compared to the coordinates of the photos. The most basic way of determining the coordinates of the contacts will be to start out by requiring the user to manually store the address of each contact in the contact list. There exist several web services on the internet that are able to translate addresses into coordinates [71][72]. When the addresses of all the contacts are known, one of these web services can be used to retrieve the coordinates.

There will usually be a slight variance between the coordinates stored in a geotagged photo and the coordinates retrieved for an address, since the coordinates can be given with higher accuracy than the size of a building. Small deviations must therefore be accepted when comparing the locations of photos to the addresses of contacts. To compare the photo location to the contact locations, the distance from the photo to each of the addresses must be calculated. As discussed in the last section, the WGS84 ellipsoid must be used to get an accurate answer. Each of the distances must be compared to a threshold value to decide if the photo was taken at that address or not.

The algorithm for finding which of the contacts that lives where a specific photo was taken can be described as follows:

1. Read the location of the photo into memory.
2. Retrieve the coordinates of the address for each of the contacts in the contact list.
3. For each of the addresses, calculate the distance between the photo and the address.
4. If the distance is less than the threshold, add the contact to the list of matches.

This algorithm will cover the case where several contacts live together. There are however some problems with it. The fixed size threshold value can be really problematic, since the size of people's properties varies wildly. A very low threshold value of 20-50 meters would be appropriate for people living in small apartments, whereas a much larger value would be needed for people living at farms. Unfortunately, the web services that convert addresses to coordinates have no information about the size of the properties, so there is no good way of determining this. Geographical data that describes which areas are considered urban and which are considered rural might be used to make a guess on property size, but it would be very inaccurate. Another problem is that people living at different floors in the same apartment building can not be distinguished from each other.

4.3.1 Phone Directory

The biggest problem with the "Who Lives Here?" application is that it requires the user to manually store the home address of all the contacts in the contact list. A typical mobile phone user will usually just store the name and phone number of his contacts. Other information, like home address, is currently not very useful in a mobile phone, and it is therefore expected that very few people will take the time to enter this manually. This problem can be mitigated by the use of a public phone directory available as a web service, since most phone directories usually also contains the address of the phone owner. The problem of several people having the same name can be avoided, since the phone number is already available on the phone. The name and phone number pair will uniquely identify one specific person. The address of this contact can then be returned to the mobile phone and used in the same way as if it was stored manually.

4.3.2 Social Networking Websites

The popularity of websites for social networking has exploded the last few years. Examples are Facebook [39], MySpace [73] and LinkedIn [74]. A large portion of the population use one or more of these services, and many have more contacts on these sites than they have on their mobile phone. Most of the social networking sites let the user publish a profile with personal information, such as address, phone number and so on. Most of the sites also provide some sort of API that lets developers integrate the sites into other applications. The data from these sites could therefore potentially be used to enhance the functionality of the "Who Lives

Here?" application. Instead of just looking through the contact list on the phone, the application could retrieve the user's contact list from the social networking websites and use these in the exact same way. If a contact on a social networking site has published an address, this can be used directly with the web services that translate to coordinates, just like with a regular phone contact. If only a phone number is available, and no address, the phone directory can be used, just like described above. This addition could increase the portion of a photo collection that matches a contact's address.

4.4 From Contact List to Photos

The "Who Lives Here?" application described in the last section can be just as interesting if it is reversed. Instead of finding which contacts that live where a specific photo was taken, it can be useful to find all photos taken where a specific contact lives. It can be used for the same basic problem as finding a photo with a map, namely to find a photo that the user knows where was taken. If the user remembers taking the photo where one of his friends lives, it would be much easier to just browse through the contact list to find his friend, than navigating to the right location on a map. Photos that are not taken where anyone in the contact list lives are not accessible at all using this approach though.

4.5 Social Gathering Places

A lot of the places where people meet to socialize are provided and taken care of by companies. This may be pubs, bars, restaurants, cafés, concert scenes, schools and many others. When people describe these places to each other, it will usually be by the name of the company. Given that the person is familiar with the place, this is the easiest way of describing the location. Using address, coordinates or any other description of location will be less intuitive.

When a picture is taken at one of these places, its location will therefore also most intuitively be described by the name of the company. If this idea is taken into the domain of an image browsing application, it might be useful if the application could tell the user for instance that "This photo was taken at Rick's café in Trondheim." It is often dark and very crowded at some of these locations, and it may therefore be difficult to recognize where it was taken. The use of a company name can be much faster and easier to interpret for the user than for instance a map, as described in section 4.1.

A comprehensive database of companies and their locations is needed to implement this application. It is hard to find such a database that has global coverage, but databases for specific countries are possible to obtain. For a lot of countries, this information is made available on the web by companies such as YELLOWPAGES.COM LLC [75] for the US and Gule Sider by Eniro Norge AS [76] for Norway. A lot of the companies in such databases will not be relevant for this kind of application, so it will have to be filtered to obtain only the most interesting categories. When a list of addresses for the relevant companies is available, the exact location for these companies, described by latitude and longitude, may be obtained and used in the same way as for the contacts, as described in section 4.3. The same web services for translating addresses into coordinates can be used, and the same general algorithm as used with the contact list to find geographical matches can be used, by just replacing the contact location by company location instead. The usage of a threshold on the distances also still holds. The algorithm for this application would then be as follows:

1. Read the location of the photo into memory.

2. Retrieve the coordinates of the address for each of the companies in the database.
3. For each of the addresses, calculate the distance between the photo and the address.
4. If the distance is less than the threshold, add the company to the list of matches.

This of course has the same pros and cons as when used on contacts. If several companies are located at the same place, all of them will be found. The problem of using a static threshold value for all companies is still problematic, so for example large outdoor concert scenes will cause problems.

4.6 Current Location

The goal of this application is to use the coordinates of where the user is currently located as a basis for browsing a subset of the photo collection. Some times a user may be at a place where he remembers having taken pictures some time ago. By utilizing the user's current position, obtained by some kind of positioning system in the mobile phone, the geotagged photos that were taken in this location can be retrieved. The same algorithm as used earlier for retrieving objects that are within a certain distance threshold can be used here too, but in this case it might be beneficial for the user to manually adjusting the threshold value through the user interface, instead of using a predefined static value. If the user is not exactly at the place where a certain photo was taken, the threshold value can be increased until the correct photo is found. When the algorithm returns too many photos, the user can lower the threshold value if he is in the exact same place where the photo was taken. It is important to be careful when adding more adjustable parameters to the user interface though, as this might often confuse the user. This is especially true on a mobile phone, where the space on the screen is limited. It is probably not a problem in this case though, as this will be the only adjustable value.

4.7 Same Location as Other Photos

This feature is about finding more photos that are taken in the same location as the photo the user is currently viewing. This can be used to find several photos of the same object, possibly from different view angels, different seasons and different weather. For instance, if the user is viewing a photo of the Nidaros cathedral, this feature can be used to find other photos of the cathedral. Since the photos may be taken at different events, with a long time span separating them, the traditional way of sorting photos by date and time is of little help. Photos taken at the same location can be found using a map, as described in section 4.2, but it would be a lot of unnecessary extra work to use this approach when the location is already available. If the result set of this location-based search is very large, it can be beneficial to list the photos by event, using a time-based clustering algorithm [29].

The same algorithm as described earlier can be used too to determine the photos which are taken close to the currently viewed photo. The ability to choose threshold value for the distance might be interesting in this application also, but the concern of reduced usability still prevails.

4.8 Related Geotagged Information

Even though a picture is supposed to tell more than a thousand words, it is still often interesting to get more information about the content of the photo. For instance, when the user is viewing a photo of the Nidaros cathedral, he might be interested in obtaining more

information about the cathedral. Unfortunately, the research on image recognition techniques has not come so far as to be able to recognize any object in a picture. Instead, the location of where the photo was taken can be used for making good guesses, if a database of geotagged information is available. This is of course not a perfect solution, for instance a close-up photo of a bird, which was incidentally taken outside the Nidaros cathedral, would be recognized as the cathedral and not the bird. Even though this problem exists, it might be a good enough approximation in many cases.

The database of geotagged information could come from many different sources. For instance, news articles from a news archive might be interesting, as well as articles from an encyclopaedia. Unfortunately, there are not many sources of information that currently geotag their content. The most interesting exception is Wikipedia [77], the currently largest and most popular encyclopaedia with user-generated content, available for free on the internet. The number of articles that have been geotagged is constantly growing, and has currently reached more than 200 000. Unfortunately Wikipedia does not, at least not yet, provide any way of searching for articles based on coordinates. Instead, the database of geotagged articles is made available for download from the project webpage of Wikipedia-World [78]. This can be used by interested developers to create applications that utilize the geotagged information.

One big obstacle for making this a user friendly system has to do with the size of the object being described in the article. For instance, it might be important to distinguish between articles describing countries, cities and buildings. Even though the coordinates for these might be the same, the scale is not. If a user is viewing a photo of the Nidaros cathedral, he is probably not interested in an article about Norway. The databases of geotagged information might contain information about scale, such as the Wikipedia database does, but this is not available for photos that have been automatically geotagged. It might be reasonable to assume though, that most private photos, at least those taken with a mobile phone camera, will usually show just a small area, and not large regions like whole cities or countries. Thus, removing or filtering out articles about large-scale objects from the database before doing a location-based search might be favorable. The method for identifying geotagged articles that are close enough to be relevant for a specified photo will be very similar to the one described for the earlier applications.

4.9 Navigation to Photo Location

Sometimes a user might look at a photo and recognize where it was taken, but not know how to get there. This can happen if the photo was taken in an area that the user is not familiar with, or if the user is viewing photos that were taken by someone else. In these cases, the user might want to get to the location of the photo, especially if the photo shows a sight or landmark that is usually visited by tourists. This can be made possible by combining the geotagged photos with a navigation application, with similar functionality as the GPS navigation units that has become extremely popular in cars and boats. It can either be a stand-alone navigation application that just receives the target coordinates from the geotagged photo, or the navigational data can be an integrated part of an image browsing application. The Android platform has support navigation as a part of the integrated Google Maps application and framework.

One of the most interesting scenarios where this kind of functionality can be useful is providing services for tourists. A tourist information office or website might provide a collection of geotagged photos of the most interesting places for tourists to visit in the area.

When the collection is loaded into the phone, the user can request directions to each of the places while browsing the photos.

4.10 Summary

Several different usages of geotagged photos have been discussed in this chapter. Mobile phones have until the recent years not been a good enough platform for these kinds of applications, but this is currently changing. Some of the application suggestions described here are well tested on PCs, others are not. Most of them have not spread to mobile phones yet. The concept of visualizing geotagged photos using a map is well known in the PC world, as described in section 2.5. Connecting the location of photos to contacts or social gathering places, as well as other geotagged information has not been well explored.

5 Methodology

To be able to answer what was stated in the problem definition, several of the application suggestions described in the previous chapter had to be implemented. This was necessary both to gain a better understanding of the capabilities of the Android platform, and to be able to carry out a realistic user survey. Unfortunately, there was not enough time to implement them all, so three applications that were regarded as most interesting was chosen. They have not been developed to a level where they are ready to be released to the public, but only as proofs of concepts to be able to answer the fundamental questions of this study. This chapter will elaborate further on what the study is about, provide details on how the application suggestions were implemented, and describe how the user survey was carried out.

5.1 Problem Elaboration

As stated in the problem definition, the objectives of this study is to explore different usages of geotagged photos, assess how suitable the Android platform is for applications involving geotagged photos, and to carry out a small user survey to investigate if mobile phone users find these kinds of applications interesting to use. Several concepts for how geotagged photos can be used were explored in the previous chapter, and some of these have been investigated further to gain a deeper understanding of how they can be implemented on the Android platform. The platform has several built-in features that are concerned with the location of the mobile phone, and therefore fits perfectly into geotagging applications. These features had to be studied to find out how they could be used to implement the different prototypes.

A very interesting part of this study though, is the user survey. A small user group is asked to try the different prototypes that are implemented, and to answer some questions about them. This is important to not only find out how the applications can be implemented, but also if people find these kinds of applications useful and intuitive.

5.2 Implementation of Prototypes

As mentioned earlier, there was not enough time to implement all of the application suggestions described. Therefore, only the three applications that were regarded most interesting were chosen for prototype implementation. These are the applications called “From Photo to Map”, “From Map to Photos” and “Who Lives Here?”, as described in the previous chapter. The details of how these prototypes were implemented are described here, as well as how the underlying data is created and stored. Some general concerns regarding development for mobile phones, not specific for the Android platform, are also discussed here.

5.2.1 Challenges of Mobile Development

Although the recent years have shown huge improvements in the technology of mobile phones, they are still very limited devices compared to regular PCs. This has to be considered when developing mobile applications. Mobile phones today still have limited resources such as slow processors and little memory, even though they are much better than they were just a few years ago. Unlike desktop computers, mobile devices are powered by batteries, so more work for the processor means that the battery drains faster. Mobile devices also have much smaller screens and more awkward input methods than PCs, which makes it much more

important to focus on implementing simple user interfaces that are easy to use and have only as many user choices as absolutely necessary.

The popularity of fast and always-on internet connections for PCs has strongly influenced the way software is developed, and software developers today can usually assume that most PCs have a good quality connection to the internet most of the time. The situation for mobile phones, on the other hand, is varies a lot. High speed connections are available in some areas, but in other areas only slow connections are available, although this is constantly being improved. The cost of transferring data to and from a mobile phone has also been high. Mobile subscriptions that allow unlimited data transfers for a fixed price have recently become available, but are not very widespread yet, although the popularity is rising. Some newer and more advanced mobile phones also have a built-in WLAN adapter, which can provide free internet access where this is available. In this study, it is assumed that a high quality internet connection is available.

Software development on mobile phones has traditionally meant using Java ME, which is available on most mobile phones. This is a very limited version of Java, and the kinds of applications that can be implemented using this are restricted. There is a long list of standardized extensions to Java ME called Java Specification Requests (JSRs), but the support for these varies a lot between the different phone models. The more advanced mobile phones available that goes in the category smartphones usually does not have so much restrictions though, as applications for these platforms can be developed using other technologies than Java ME. For instance, applications for Symbian OS can be programmed in C++ or Python, and Windows Mobile applications can be programmed using either native C++ or any language supporting the .net framework, such as C#. The customized Java-support in the Android platform is no where near the limited Java ME, but closely resembles Java SE, and does not put a lot of restrictions on the kinds of applications that can be implemented.

5.2.2 Using the Android Emulator

An emulator that makes it possible to run applications for mobile phones on the development PC is available for most mobile platforms supporting third party applications. It makes development and debugging of mobile software much easier, as there is no need to deploy the application to the phone each time a minor change is done. The developers don’t even need access to the phones they are developing applications for. Although the applications should always be tested on real hardware before they are released, of course.

The Android emulator is very well integrated with the Eclipse IDE plug-in, and it is very easy to test and debug Android applications on the emulator from within the IDE. For the user survey, on the other hand, the use of an emulator might have some drawbacks. The experience of using the mobile applications by manipulating the virtual phone with the mouse and keyboard is fundamentally different than having an actual phone in the hand. This might influence the results of the user survey. The concepts of the applications are the most important thing to investigate here though, so a slight awkwardness in the usability might not be that important.

Other problems with the Android emulator is, as mentioned in section 3.8, that the camera only shows a square in front of a background with a chessboard pattern, and that the GPS in the emulator only simulates that the phone is traveling back and forth on a predefined route in

the San Francisco Bay area. There are other limitations too, but they are not relevant for the applications described here.

5.2.3 Storing and Accessing Location Information in Image Files

The first step to implementing these geotagging applications is to find a good way of storing the location information. The Android developers seem to be planning to implement a way to store geotagged photos in the platform itself, as a ContentProvider for this has been defined in the class android.provider.MediaStore.Images. Unfortunately, the ContentProvider does not work yet, and there is no information available on how it will work or when it will be done. A way of storing the location information must therefore be created by the application developer. The solution must be fast and effective, so that retrieving the location information from all the photos in large collections, as needed when the location of all photos is to be visualized in a map, does not cause too much delay. The location information can be stored in several different ways, but the two most promising ways are either to store the information in the Exchangeable Image File Format (EXIF) [79] header of the image file itself or to use a separate database in the mobile phone to store the locations.

The EXIF header is a standardized format for storing many different types of metadata about a photo inside the image file itself. This can be information such as the time and date, the camera model, the camera settings used to capture this exact picture, and most importantly for this study, the geographical location for where the photo was taken. The information in the EXIF headers in the image files can be read and written using a third-party Java library called Sanselan [80], which is an open-source project released under the Apache Licence, Version 2.0 [56]. There are two main advantages of using the EXIF header. The first is that since the location information is stored in the file itself, it will follow the file wherever it is copied to. If the file is transferred from the mobile phone to a PC or uploaded to a server, the location information will be available there too. The other advantage is that it is a standardized format, supported by a large portion of applications that handle geotagged photos. The location information can therefore also be easily used by other applications when the photos are transferred out of the phone. The main disadvantage with using the EXIF header is speed. As long as the location information for only single photos is requested, there is no problem. When the location information from a large number of photos are to be aggregated, on the other hand, the application might get very slow if the information must be read from a large number of different files.

The other alternative is to store the location information for each of the photos in a database, separated from the image files. The photos could be stored as regular files, and the database would store the filename of each file to be able to connect the location information to the correct photos. As described in section 3.5, the Android platform has the SQLite database engine built in, and it could be utilized for this purpose. Databases are made for handling large datasets, so the main advantage of using a database for storing the location information would be speed. There is unfortunately no standardized way of storing the location information of a geotagged photo in a database, and the image files themselves would have no information about the location. If the images were to be transferred to another device, the location information would be lost. A system could be made to interface between the database and the receiving system, but there is no standardized way of doing this.

The portability problems of using a database is a major concern, but the performance of storing the location information distributed into each separate file might also be problematic.

To be able to make a decision, a benchmark test comparing the performance of the two methods had to be performed. The scenario being tested was to read the location information for 100 photos. There were created 100 image files with the location information embedded in the EXIF header, and a simple database table was created in SQLite that contained 100 coordinates. First the image files were tested. The test program looped through all the image files, reading all the coordinates into an array using the Sanselan library. The test was run five times while the execution time was measured. The average result was 27.8 seconds, with very low variance. Then the database method was tested. The test program used the built-in library from the Android Application Framework to read the 100 coordinates from the database table into an array. This test was also run five times, and the execution times were measured. The result was 0.2 seconds, also with very low variance. As expected, the database solution was extremely much more efficient than using the files, as shown in Figure 5.1. The point of this test was to see how big the difference is, and it turned out to be major. The difference is also expected to grow larger as the size of the photo collection increases.

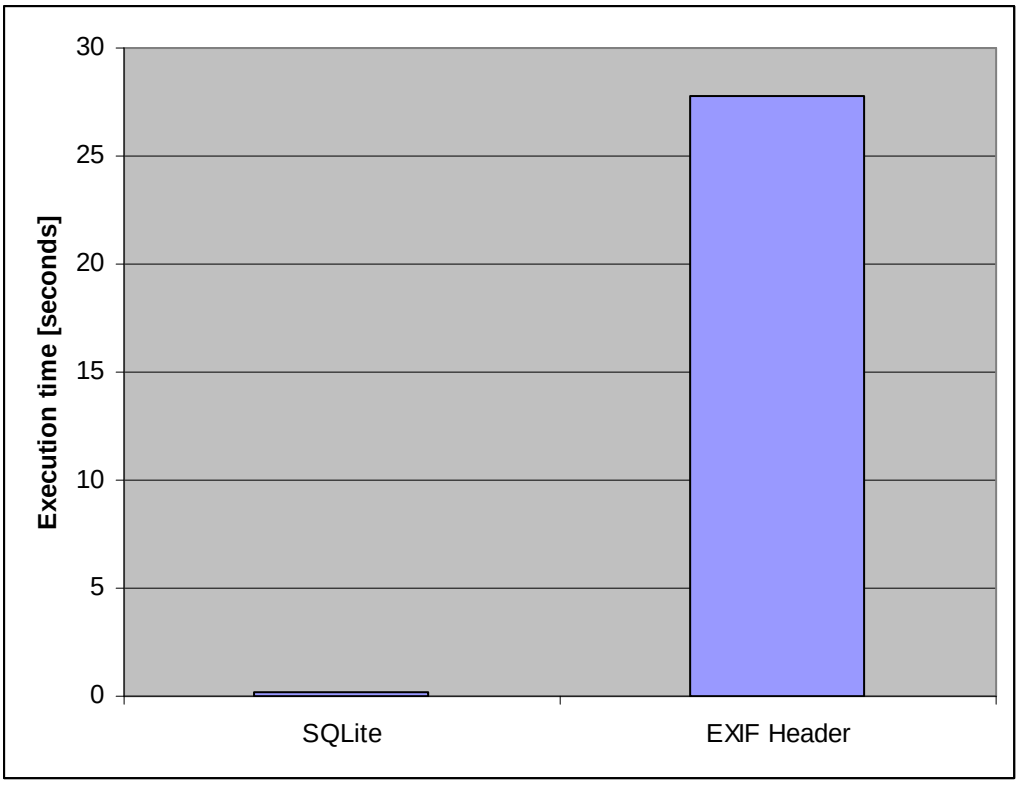

Figure 5.1: Execution time for reading coordinates of 100 geotagged photos.

Although the database storage is superior in performance, the portability of the geotagged photos is still a big concern. A possible solution is to combine the two methods, by storing the location information both in the EXIF header in the image files and in a database table. The database can thereby be used as a fast caching mechanism of the metadata in the files. This combined solution has the advantages of both the two original suggestions, and none of the disadvantages. If the image file is transferred to other devices, the location information will still be intact, at the same time as the database can be used to get high performance when the location of many photos is needed. When the location for just a single photo is needed, both methods can be used. The delay will be negligible anyway, so the method that is considered easiest to implement in the current context can be used.

The combined storage solution has been used to create a ContentProvider for geotagged photos, to make the geotagged photos available to the applications on the mobile phone that will use them. It is simply named ImageProvider. ContentProviders will usually contain methods to query, insert, update and delete the underlying data. Only the method for querying data was needed for the prototypes implemented here, so the other three methods were omitted. If the ImageProvider was to be released to real users, and not just used as a proof of concept in a user survey, the methods for insert, update and delete would also have to be implemented. This would be trivial though. When querying the ImageProvider, both the metadata and the actual image file data is made available to the application consuming the data from the provider. How the data is further utilized is completely up to the consumer application. The source code of the ImageProvider is given in appendix E.

The database table created for the ImageProvider is shown in Figure 5.2. The fields for filename, latitude and longitude do not need any further explanation, but the fields image_original, image_screensize and image_thumbnail will be described in the following sections. Other fields for storing other parts of the metadata from the EXIF header, such as time and date, may be desirable to store in the database for other applications than the ones described here. This has not been implemented, but it would be an easy extension.

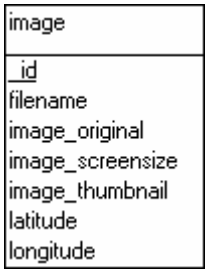

Figure 5.2: Database table used to store location information of geotagged photos.

An Activity was added to the ImageProvider application to make it possible to read metadata from image files in a specified folder and store it in the database. The activity was called ProviderController. It uses the Sanselan library to read the EXIF header in the files, and the SQLite library in the Android Application Framework to store the data directly in the database. Since this activity is a part of the same application package as the ImageProvider, they both have direct access to the database.

The algorithm for importing the location information from the EXIF headers is as follows:

1. Retrieve a list of all the files in the image folder.
2. Retrieve a list of all the records in the database.
3. Compare the list of image files to the list of records.
4. If an image file is not in the database, read the metadata from the file, create a thumbnail in the thumbnail folder and store the information as a new row in the database.
5. If a row in the database points to a file that does not exist, delete the row from the database.

In the future, the earlier mentioned built-in ContentProvider described in android.provider.MediaStore.Images might be used instead of this ImageProvider, but it is currently not implemented in Android yet. It seems like the plan is for the provider to be able to store the location of images, but it is not yet known how the location data will be stored. A new benchmark will therefore have to be performed if the provider becomes available to see if

it is fast enough for the uses specified earlier, or if the custom ImageProvider developed in this study must still be used.

5.2.4 Creation of a Geotagged Photo Collection

An important aspect of geotagging, although not the main focus of this study, is how to create geotagged photos. Several different positioning technologies and software for geotagging were presented in chapter 2. There are applications that automatically geotag photos taken with a mobile phone available for other mobile platforms, but none is available for Android yet.

The first approach of this study was to create an Android application that could capture an image from the camera, retrieve the location of the phone from the GPS receiver and store the location in the EXIF header of the image file. The Sanselan library was used to write the EXIF header. Unfortunately, as described in section 3.8, the data retrieved from the camera and GPS in the emulator is not very interesting. The emulator is able to access web services on the internet, and therefore it would be possible to connect a webcam and a GPS receiver to the host computer, and publish the data from these devices through a local web server. It would be very awkward though, to run around with a laptop, webcam and GPS receiver to create a collection of geotagged photos. It would also take a lot of time to create a collection of a reasonable size for use in the user survey. The survey also would be most interesting if the collection had photos from several different countries and continents, which would not have been possible if they were to be created using the emulator. It was therefore decided to not use the Android geotagging application.

Instead of creating geotagged photos using the emulator, two photo collections were created in two different ways. First, a small collection consisting of only seven photos was made for testing the concept. It was created by selecting photos from a private photo collection and manually geotagging them by using PC software, as described in section 2.5. The collection was then transferred to the emulator and stored in the ImageProvider. For some of the tests in the user survey though, a much larger photo collection was needed. It was therefore created an Android application that downloaded publicly available geotagged photos from flickr and stored them in the ImageProvider. A photo collection of 176 photos taken all over the world was created using this tool.

5.2.5 Implementation of a Location-Aware Image Browser

The three application suggestions that were selected for prototype implementation have been combined into one integrated location-aware image browser. The application's main layout, as shown in Figure 5.3, has been based on the example code of an image gallery that comes with the Android SDK. The photos used in the example code were hard-coded resource files that were compiled into the application. For the location-aware image browser the photos instead had to be retrieved from the ImageProvider at runtime. The user can switch between the different photos using the directional keys on the phone. A context menu was added to provide access to the three new functionalities added, as shown in Figure 5.4.

In the first attempt, the original full-size image data was used directly both for the large-scale view of the selected photo and for the small thumbnail-sized photos near the bottom of the screen. This caused two big problems that made the application unusable. The first problem was that the image browser was extremely slow. It took a long time to start up, and switching

from one photo to the next took several seconds. The other problem is related to the current version of the Android platform. The garbage collection mechanism in the Dalvik VM does not work very well yet. If a series of large objects are created in the Java code, and the reference to each object is removed before the next one is created, the garbage collector is expected to delete each of the unreferenced objects when necessary. When the memory available to the applications fills up with unreferenced data, the garbage collector should remove this to make space for new data. Unfortunately, the garbage collector in Dalvik seems to just work periodically and not kick in when the memory fills up. Instead, the whole application is simply killed, without any warning or error messages. This issue must be expected to be fixed in the final release of the Android platform though, so the main problem of using the original full-size image data is speed.

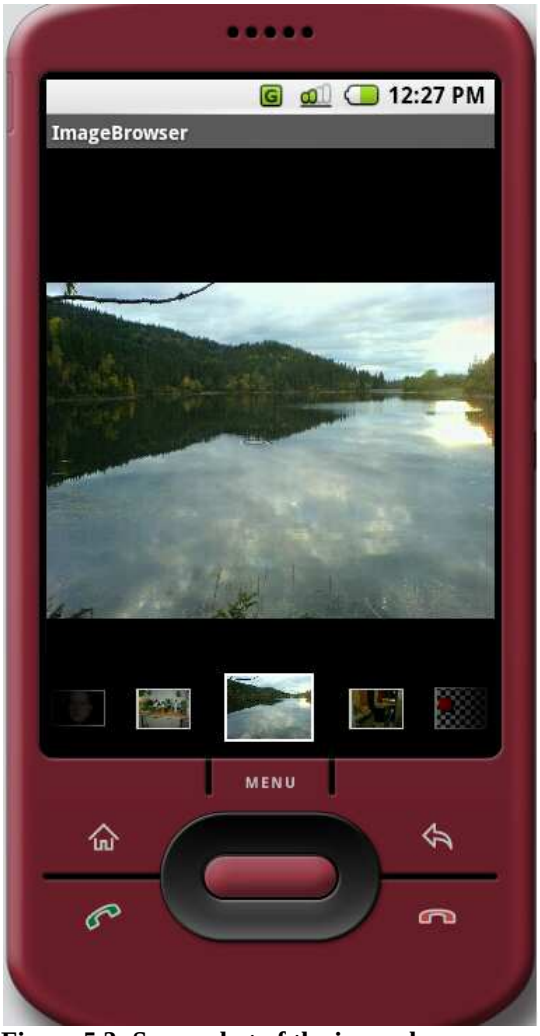

Figure 5.3: Screenshot of the image browser.

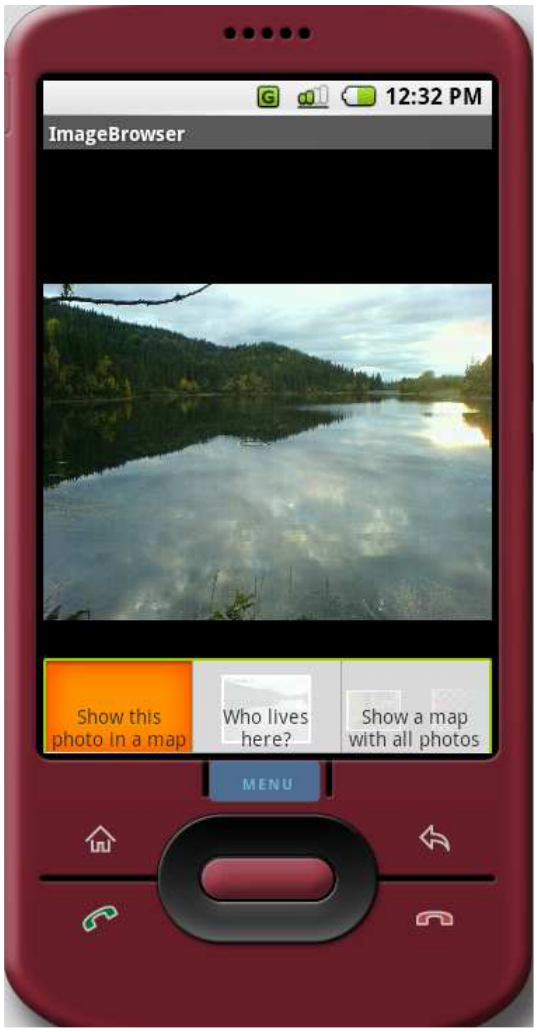

Figure 5.4: Screenshot showing the context menu of the image browser.

To avoid both the performance problem and the application crash, the size of the image data used had to be reduced. A photo taken by most kinds of digital cameras, both integrated in mobile phones and stand-alone cameras, will usually consist of significantly more pixels than the screen of a mobile phone can show. It is not uncommon today that mobile phone cameras produce pictures of 2-3 megapixels. The resolution of mobile phone screens is of course varying, but the resolution of the emulator screen can be used as an indicator. The default emulator resolution is 320x480 pixels, approximately 0.15 megapixels. The thumbnail-sized photos at the bottom of the screen in the image browser are much smaller than this too. It was therefore decided that when the photos are imported into the ImageProvider, two image copies with lower resolution should be created. One with a maximum width of 320 pixels and another with a maximum width of 60 pixels were needed to fit the full-screen view and the thumbnail view. The heights of these low-resolution images had to be calculated from the aspect ratio of the original image. Using these two image files instead of the original full-sized image resolved both the described problems. The image browser became very responsive, and the garbage collector got time to do its job before the memory filled up.

Storing two copies of each photo will of course need more storage space than just storing the original, but this is considered a minor problem when compared to the advantage gained. Most new and advanced mobile phones have large storage capacities, provided by the support of memory cards. Currently the memory cards for mobile phones are available in sizes up to 8GB. A JPEG compressed photo taken with a mobile phone camera will vary in size, but

usually take up somewhere between 200kB and 2MB. The two low-resolution image files combined will need approximately 5-10% of the storage space needed by the original image, which is considered to be negligible.

The next sections will describe the implementation of the three different usages of the location information in the geotagged photo collection. The source code of the complete location-aware image browser is provided in appendix D.

5.2.6 Implementation Details of "From Photo to Map"

The first feature that has been implemented is the one called "From Photo to Map", which enables the user to see a marker on a map where a specified photo was taken. The feature was described in section 4.1. The user starts by browsing the photos in the location-aware image browser. When he has found a photo that he wants to find the location of, he can open the context menu shown in Figure 5.4 and select "Show this photo in a map" to open the map Activity. The map activity shows a map of the area where the photo was taken, with the exact location indicated by a marker, and the zoom adjusted to a suitable level. The zoom level has been hard-coded and will therefore be the same for all photos. An example of a map with a marker is shown in Figure 5.5. The circle shown on the end of the marker is not a part of the marker, but simply indicates the center of the screen to make it easier to aim when zooming.

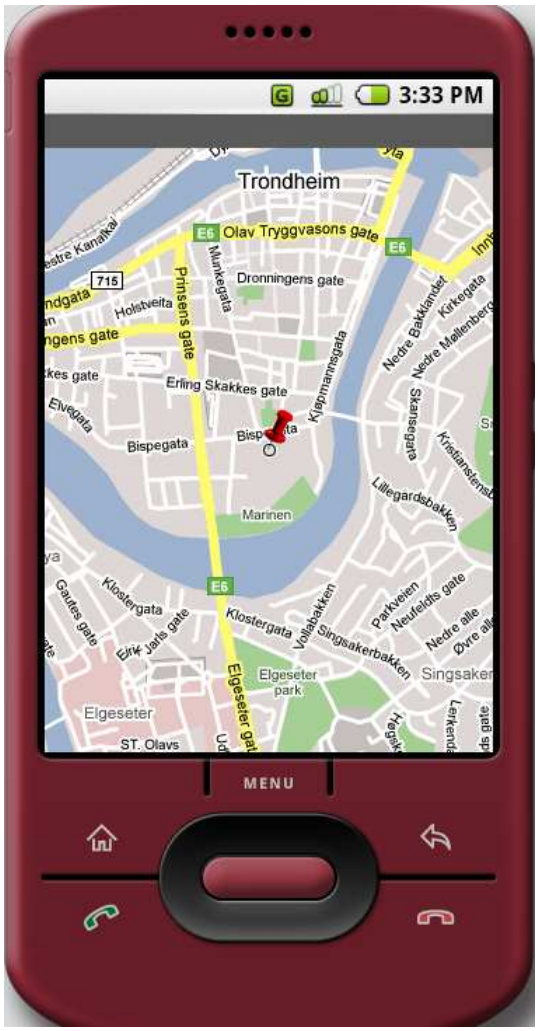

Figure 5.5: Screenshot of the map with a point marker.

On the emulator the map can be panned around by moving the mouse pointer around on the screen while holding the left mouse button down. This is equivalent to moving the finger around on the screen on a real phone. By clicking and holding the mouse button down without moving the mouse, or by holding the finger still on a real phone, the partially transparent zoom controller appears. It was decided not to have it constantly visible in order to save space on the screen for the map. This is not a problem on a regular PC, but on the small screen of a mobile phone the space on the screen needs to be used more carefully. The zoom controller is shown in Figure 5.6.

Figure 5.6: Screenshot showing the zoom controller of the map.

As described in the Android chapter, the Activities in Android applications are built up by Views. It was also mentioned that Google Maps, as shown in these screenshots, are a built-in part of the Android platform. Google Maps is made available to the developers as a View, simply called MapView. Features of the MapView, such as panning and zooming, can be controlled programmatically by calling methods on a MapController obtained from the MapView.

The web-based version of Google Maps used on PCs has a built-in feature for placing a marker onto the map in a specified coordinate. This feature is unfortunately not available in the Android implementation of Google maps. What is available though is an OverlayController, which can be obtained from the MapView. The OverlayController can be

used to draw any kind of graphics onto the map. The developer does not need to worry about the screen coordinates of where the overlays are drawn, as the framework contains functionality for converting from latitude and longitude. In this application, the OverlayController has been used to draw a point marker icon stored in a resource file, which is compiled into the application package.

5.2.7 Implementation Details of "From Map to Photos"

The second implemented feature is the one called "From Map to Photos", described in section 4.2. It lets the user start with a map where all the places where photos have been taken are shown with a point marker. If the user clicks on one of the point markers, he is shown all the photos taken at that location. Photos taken close to each other are aggregated into single point markers. How close the photos have to be to be grouped into one marker depends on the current zoom level of the map. When the map is zoomed all the way out so that the whole world is visible, photos from large areas, often spanning several countries, will be grouped together. When the map is zoomed close in, only photos taken at the exact same location will be grouped together. Three different zoom levels are shown in Figure 5.7, Figure 5.8 and Figure 5.9, illustrating that the number of groups are different when the whole world, a country and just a city is visible. The feature is available via the context menu in the image browsing application, as shown in Figure 5.4, by selecting "Show a map with all photos".

Figure 5.7: From Map to Photos, showing the whole world.

Figure 5.8: From Map to Photos, showing a country.

Figure 5.9: From Map to Photos, showing a city.

When the application is started, the map will automatically be adjusted to show all the photo locations. If photos in the collection are taken all over the world, the whole world will be visible. If the collection consists of photos from only one city, then just that city will be visible. The pan and zoom controls that the user needs to navigate the map are the same as described in the previous section.

The hardest implementation challenge here is the clustering algorithm. An overview of the steps in the algorithm as defined by [69] was given in section 4.2. A more detailed description of the implementation is given here. At the beginning, each photo is given its own cluster, so the number of photos, N, equals the number of clusters. Then, the distances between each of the clusters need to be calculated. As described earlier, the WGS84 ellipsoid has to be used to accurately calculate the distance between two coordinates. Fortunately for Android developers, this distance calculation is included in the Location API, so using it is as easy as simple function call. The distances are stored in an NxN matrix, so that the value placed in (x, y) is the distance between photo x and photo y. The matrix is then traversed to find the shortest distance between any two clusters, and those two clusters are merged. In the matrix, the corresponding rows and columns are deleted, and a new row and column are added for the new cluster. The distance from the new cluster to all other clusters must be calculated and inserted into the matrix. The distance between two clusters consisting of more than one photo can be defined in several ways, the most important being single-linkage and complete-linkage. Single-linkage means using the shortest distance from any photo in the first cluster to

any photo in the other, while complete-linkage means using the greatest distance. If complete-linkage was used, all the photos in one cluster would have to be very close to each other. Single-linkage was chosen, to be able to catch situations such as a series of photos taken alongside a road. When the new distances are added to the matrix, a new minimum must be found. The process of merging clusters, removing old rows and columns and adding new, calculating new distances and finding a new minimum is repeated until either all photos are merged into the same cluster or a threshold value on the minimum distance is met.

In the first attempt of using this algorithm, a new distance threshold was calculated each time the zoom level changed, and the clustering algorithm was executed until this threshold was met. This worked smoothly on the small test collection of only seven photos, but when a larger collection was used the inherent scalability problems in the clustering algorithm became very obvious. The clustering algorithm has a time complexity of $O(n^2)$, which means that each time the number of photos doubled, the execution time is multiplied by four, and this was confirmed by testing the algorithm with photo collections of different sizes. It was hoped that this effect would be negligible for the relatively small size of an average photo collection, but this was proven wrong. For the photo collection of 176 photos, it took approximately five minutes to generate the clusters. It is of course totally unacceptable to have a delay of five minutes each time the zoom level of a map is changed.

It was realized that there is no need to regenerate the clustering tree for each time the zoom level is changed. Instead, the clustering algorithm can be executed once until all photos are merged into the same cluster. This clustering tree can then be kept in memory together with an array containing the minimum distances that caused the cluster merges for each level. When the zoom level of the map changes, the corresponding composition of clusters can be obtained by going into the correct level in the clustering tree. This was used to enhance the performance of the application, so that no noticeable delay from calculating clusters occurs when changing zoom levels, although the cluster tree still takes five minutes to build when the application is started. This was not considered a problem for using the application as a proof of concept in the user survey though, so this was the method used. If the application were to be released to normal users, the loading time problem would have to be mitigated somehow. Suggestions for how this could be done are discussed in chapter 8.

5.2.8 Implementation Details of "Who Lives Here?"

The third and last application suggestion that was implemented was the one called "Who Lives Here?" It is described in section 4.3. It is available from the context menu in the image browser, shown in Figure 5.4, by selecting "Who lives here?" If any of the contacts lives where the currently selected photo was taken, the user is shown a list of these, as shown in Figure 5.10. Otherwise a message telling the user that none of the contacts lives where the photo was taken is shown. The two possible extensions described in the sub-sections 4.3.1 and 4.3.2 has not been implemented, so this application requires the user to store the address of each of the contacts manually.

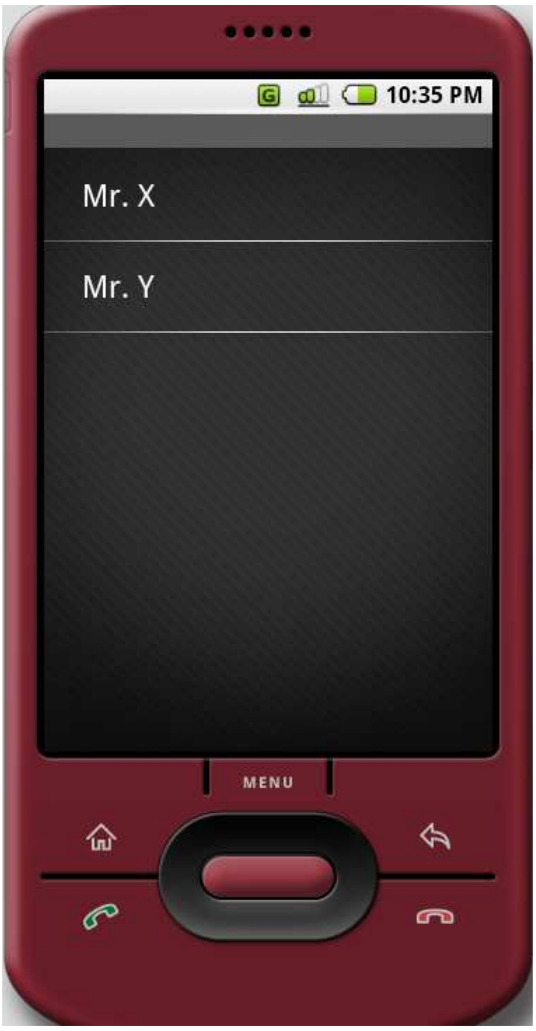

Figure 5.10: Who Lives Here?, showing two contacts.

The developers of Android plan to include the functionality of translating street addresses into coordinates as a part of the Android platform. The class has been created as android.location.Geocoder, but the functionality is not implemented yet. As mentioned in section 4.3, there exist several web services that are able to perform this task. The one provided by Google was used in this prototype implementation.

As explained in section 4.2, the distance from the selected photo to the address of each of the contacts must be calculated. Therefore, the coordinates for every contact in the contact list must be retrieved. To use a web service is a relatively slow process, especially when it has to be called many times in a row. If the coordinates for all the contacts were to be obtained from the web service each time the application needed them, it would induce a huge delay and the application would feel extremely unresponsive. Thus, the coordinates need to be cached locally at the phone. The contact list in Android, which can be access through a ContentProvider, is an extensible data structure that supports many other kinds of data than the ones available through the built-in contact list application. One of the possibilities is to store the location for each contact. This is currently not in use by the Android platform, and can therefore be used to store the cached coordinates of the address. There is no documentation on how the location field in the contact list is supposed to be used though, so for an application released to users it would be wise to store the cached coordinates in a separate database and use foreign key references to the contact list for the other contact information. Otherwise there might be a conflict with other applications or services using the

same data fields in another way. This is not a concern for this prototype implementation though, so the coordinates are stored directly in the contact list. The current address of each contact is also copied to a field assigned for extra data in the location structure, so that it can be detected if the address is changed manually.

The algorithm to retrieve all the contacts that lives where a photo was taken, including the caching mechanism, can be summed up as follows:

1. Retrieve a list of all contacts that does not have the coordinates cached.
2. Obtain the coordinates for each of these contacts from the web service and store them locally in the contact list ContentProvider.
3. Retrieve a list of all coordinates stored in the contact list ContentProvider.
4. Loop through all the coordinates.
5. If a coordinate is within a specified distance threshold from the current photo, add the ID to a match list.
6. Loop through the match list of IDs and retrieve the contact information for each of them.

The question of how large the distance threshold should be, and if a static threshold is appropriate, was discussed in section 4.3. In this prototype a static threshold of 100 meters was used. Not much work has been done in determining what a good value for this would be. 100 meters is just a guess that works well with the test data, but more tests must be done to find a value that will make the application more accurate.

5.3 User Survey

In order to answer some of the core questions of this study, a small user survey had to be carried out. A small user group of 10 volunteers was created for this. The user group consisted of people between 19 and 27 years old, who are comfortable with using new and mobile technology in everyday life. Both sexes were represented in the group.

The persons in the group were first given a brief introduction to Android before being showed the location-aware image browser. This was done to make sure that the general user interface and navigation in Android was not an obstacle to understanding the application. The users were then given just a short introduction to what this study is about, before being asked to play around and explore the application themselves for a little while. In the end a detailed explanation of the application was given. The users were then asked to fill out a form with some questions, both regarding the three implemented features and some of the other application suggestions that were not implemented. The original Norwegian form given to the users can be seen in appendix A, and an English translation is given in appendix B. Each question was to be answered by a number between 1 and 5, where 1 means completely disagree and 5 means completely agree. The questions involve how useful they think each of the features is and how easy they are to use.

In addition to the questions, the users were also asked to participate in an efficiency test to see how fast they could find a specified photo using the map interface compared to browsing through all the photos linearly. The large photo collection of 176 photos was used in this test. The users were asked to find a photo of the Norwegian parliament building as fast as possible, first by using the map, then by browsing through the photo collection linearly. The parliament building was chosen because everyone knows where it is and most people can locate it on a map. If they don’t know the exact location on the map, at least they know that it is in Oslo and

can find that city on the map. For the linear browsing, the photo was placed in the middle of the collection, so that half of the pictures had to be browsed through before finding the correct one. This was based on the assumption that the average position of all the photos is in the middle, and this would therefore represent the average case.

5.4 Summary

The most important steps that had to be performed to answer the questions of the problem definition was described in this chapter. Three applications called “From Photo to Map”, From Map to Photos” and “Who Lives Here?” was selected for prototype implementation. A method for creating a geotagged photo collection was created, as well as a ContentProvider for storing and accessing the geotagged photos. A location-aware image browser was developed, which includes and implementation of all the three application prototypes described. A user survey also had to be carried out to find out how mobile phone users respond to the ability of using the extra information of location when interacting with a photo collection.

6 Results

The prototypes described in the previous chapter have been implemented, and the user survey has been conducted. This chapter presents some of the experiences acquired from working with the Android SDK, an analysis of the answers from the user survey and the results of the efficiency test of the map-based browsing.

6.1 Experiences from Using the Android SDK

The implementation of the prototype applications described in this study has required the developer to acquire a firm understanding of the currently available SDK for the Android platform. There have been both good and bad experiences. The most important aspects are described here.

In general, Android seems like a very good platform for developing both applications for geotagged photos and other location-aware applications. The platform includes a mapping solution and the Location API for easily acquiring the current location of the phone. Throughout the Android API there are many hints that make it obvious that location-awareness has been an important thing on the minds of the developers. Much of the functionality that involves location has not been implemented yet, but it is important to remember that the currently available SDK is described as a preview release, and that the platform is not finished yet. Most of the problems encountered with Android are things that most likely will be changed before the final release.

The biggest problem encountered was the garbage collector in the Dalvik VM that did not kick in before the whole process was killed when it used too much memory. This made the prototype development much more difficult. This is something that must be expected to be fixed before the final version is released though.

Another problem has been with the Android emulator sometimes crashing and hanging during startup. All the settings and user files for the emulator then had to be deleted before the emulator would start again. This might not be an issue with the Android platform itself though, as it might just as well be an issue with the emulator. This is also something that could be fixed before the final release.

A major annoyance that is unlikely to be changed has to do with the combination of data from more than one ContentProvider. Often times the ContentProviders are just an interface layer on top of a database table. An example of this is the contact list system, where several different ContentProviders grants access to different kinds of data about a contact stored in different database tables. The underlying database tables, and therefore also the data available through the ContentProviders, follows ordinary database design guidelines and uses foreign key constraints to connect the data in different tables to each other. When the database is queried directly, the data from different tables can be combined using simple join operators. When the only possible access is through the ContentProviders, on the other hand, the combination of data from different sources must be done in the Java code. This may in some cases be much more complex than combining directly in an SQL query.

There have also been some other smaller annoyances and bugs, but several features have been fixed by SDK updates, and much can still be changed before the platform is finished.

6.2 Answers from the User Survey

All the users in the survey group seemed very interested in the system developed, and they all eagerly answered all the questions they were given. The raw numbers are provided in appendix C, so only an analysis of them is appropriate here. Some of the results were very conclusive, while others were not so clear. For all the questions, the median of the ten answers has been used as a basis for the analysis.

For the application called “From Photos to Map”, the users mostly think it is useful, and they definitely agree that it is very intuitive to use. They generally do not think it would have been much easier to use on a PC, and they think that the application is suitable in the context of a mobile phone. Some think that they would use this application themselves, others do not. The median of the scores on this is 4, so the users seem slightly positive to using this kind of application.

When it comes to the application “From Map to Photos” the users are a bit more skeptical. The answers of the usefulness and intuitiveness of this application are positive, but not as enthusiastic as with the previous one. The users also say that it would have been easier to use on a PC. When asked if they would use this application themselves, the median score was 3.5, which must be regarded as inconclusive.

The “Who Lives Here?” application was regarded very intuitive to use, and also relatively useful. They also mostly agreed that it would not be easier to use on a PC, and that the application was suitable for a mobile phone. This application received the highest score of the three implemented ones when asked if they would want to use the application themselves. Both the suggested extensions to this application, i.e. obtaining addresses from a phone directory and using the contact list from social websites, was considered very useful additions to this application.

When asked about the reversed kind of the “Who Lives Here?” application, i.e. obtaining all the photos taken where a specified contact lives, the users mostly agreed that it was useful, but was not so sure if they would use it themselves. The same results are also received when they are asked about getting help to navigate to the place where a photo was taken and to get information from Wikipedia regarding the photo location.

In general, the users answered more positively when asked about the usefulness of a feature than they did when asked if they would use it themselves. Some of the reason for this might be that only the technology was presented to the users, but not much was said about possible usage scenarios. People that are interested in technology might think that a feature is cool and that it might be useful, without being able to think of situations where it would be useful to them.

6.3 Efficiency of Map-Based Browsing

A very interesting part of the user survey was to see if the users could find photos faster by using a map instead of browsing for the photos linearly. The results are shown in Figure 6.1. It can be seen that the efficiency of using the map varies a lot between the different users. The overall results though, indicate that photos can be found faster using a map for most people.

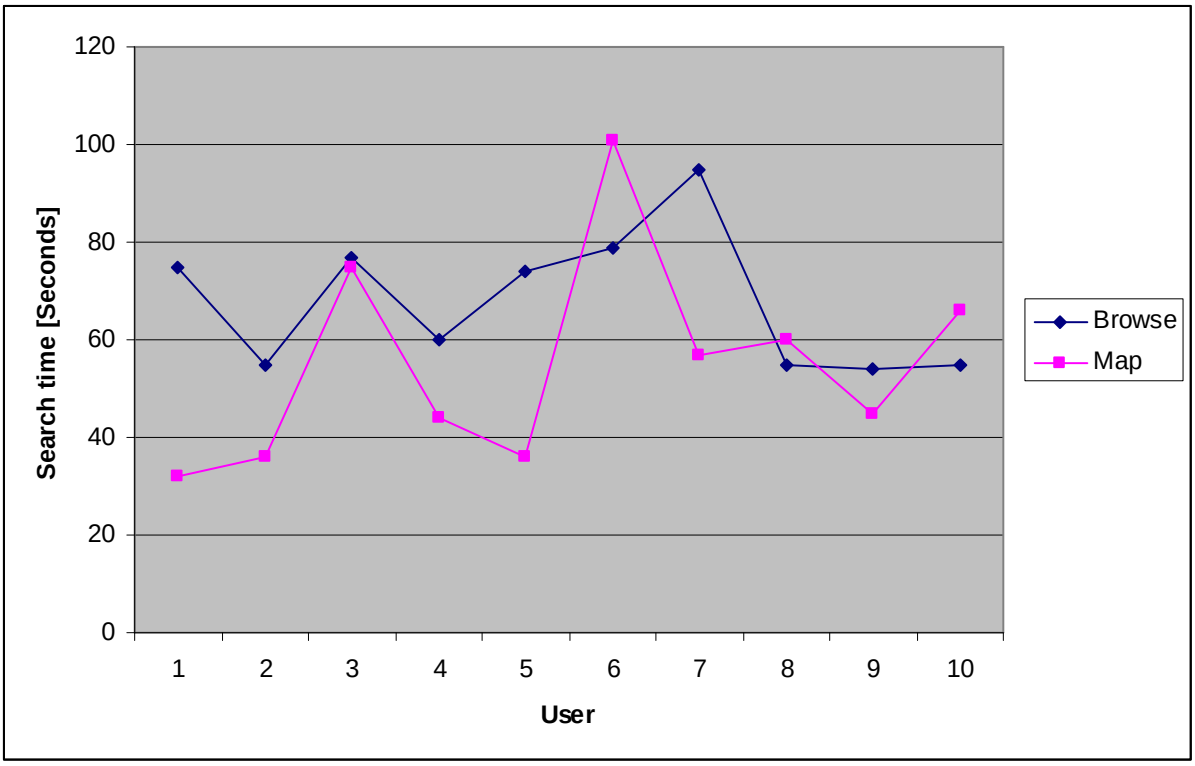

Figure 6.1: Search time by using map-based vs linear browsing.

Out of the ten users in the survey, seven found the specified photo faster by using the map than when using linear browsing. It should be noted that when user 6 performed this test, Google Maps was loading very slow on the emulator. This was caused by problems with the internet connection or the Google servers, not the emulator or the application. Because of this, the result of user 6 should be disregarded, and therefore seven out of nine users found the photo faster with a map. It is also interesting to see the difference in the average search time for the two methods. The results of user 6 is still disregarded, therefore the average search time using a map is 50 seconds, while using linear browsing takes 66 seconds, i.e. it takes 32% longer to use the linear browsing. This is illustrated in Figure 6.2.

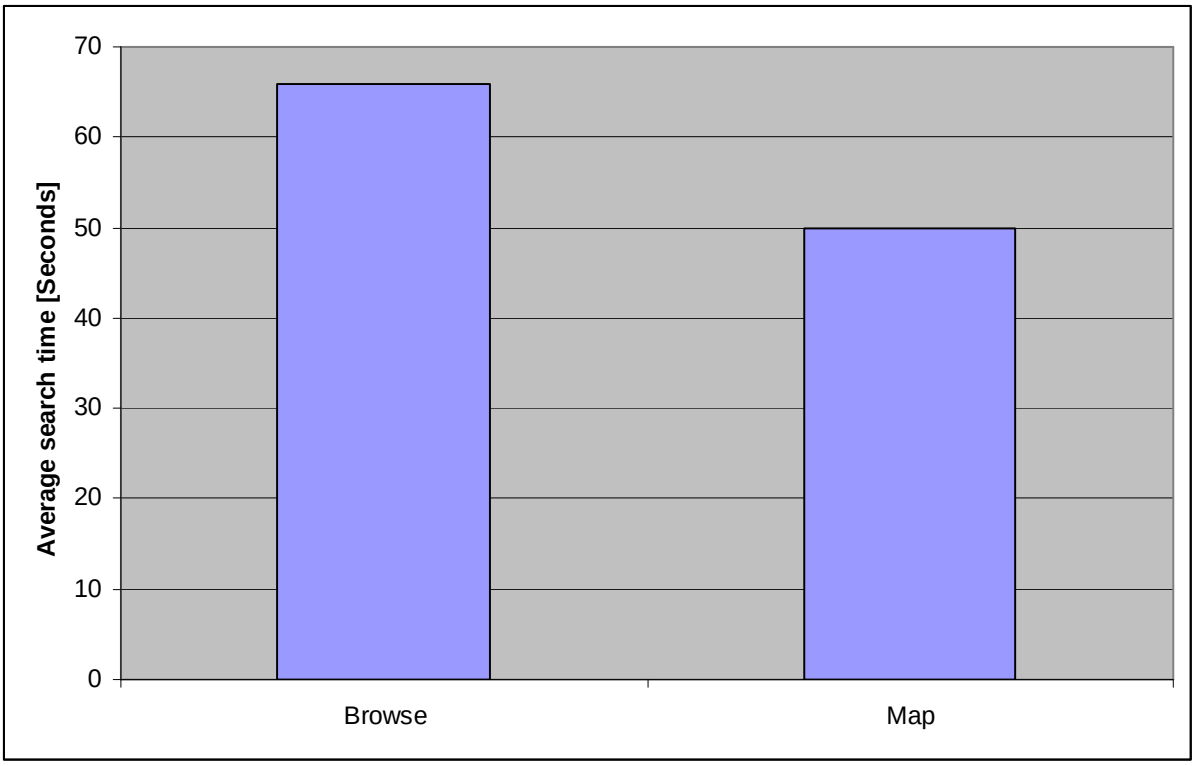

Figure 6.2: Average search times for map-based and linear browsing.

It was observed during the user survey that some of the users found both the application itself as well as the interface of the emulator easier to use than other users. This also affected how efficient their map-based browsing was. Most of the users pointed out that it was a bit awkward to navigate the map, especially the zoom controller. It was chosen not to have the zoom dialog always visible because of space constraints on the mobile screen, but usability concerns may require the zoom dialog to always be there. If the zoom controller was easier to use, some of the users would have found the photo faster.

Some users noted that it was impossible to know how close it was necessary to zoom, since the markers did not convey any information of the number of images they represent. This could have been solved by the suggestion mentioned earlier, of replacing the marker icon with a circle with the number of photos written inside. This also would have made some of the users find the photo faster, as they zoomed in much more than actually needed before clicking on the marker.

It should also be noted that the map-based approach might have been more effective on an even larger photo collection. The search time using the linear browsing is expected to increase linearly with the number of photos, while the search time using the map is expected to grow much slower. Finally, it should be pointed out that there are more effective ways of linearly browsing a photo collection than going one by one photo, as in this application. Showing a grid of thumbnails on the screen might be enough to evaluate the content of the photos, even on a small mobile phone screen.

7 Conclusion

This study has explored different uses of geotagged photos on mobile phones based on the Android platform. Three of the application suggestsions, namely “From Photos to Map”, “From Map to Photos” and “Who Lives Here?” have been implemented prototypes of. Both these and some of the other suggestions were examined by giving them to a group of volunteers, who answered some questions after.

It was found that Android is a very good platform for implementing applications that involve geotagged photos. The platform still has some problems, but most of these are expected to be fixed before the final release.

The users in the user group turned out to have a very positive response to applications using location information and thought that the applications in the survey were useful. They were a little bit more skeptical towards using the applications themselves though.

In the efficiency test between map-based and linear browsing, the results were in favor of the map-based approach. On average, the users spent 50 seconds on finding a photo using the map, while it took 66 seconds if they were to browse through the photo collection linearly. Several points that could affect the efficiency of both the map-based approach as well as the linear browsing were discussed, so the results presented here are not conclusive.

8 Further Work

This study explored several possible applications of geotagged photos on mobile phones, but unfortunately, there was only enough time to implement a few of them. All the other application suggestions are also interesting, and deserve to be studied in more detail.

Some of the users participating in the survey indicated that navigating the map was awkward, and would be easier on a regular PC with keyboard and mouse. This was only a simple prototype, and the map navigation could probably have been made much easier if there was time for it. It might also be easier if there was an actual mobile phone, and not just the emulator. In a real phone, hardware buttons could be used to zoom and pan the map, instead of using the touch screen.

The implementation of the “From Map to Photos” application has a big problem with the time it takes to generate the cluster tree. The creation procedure of the tree was moved from each time the zoom level changes to the start of the application. This was a acceptable solution for the user survey, but not for a real product. It should be investigated further if other, more lightweight clustering algorithms can handle this problem more efficiently without too large side effects. The cluster tree generation procedure can also be moved further, so that the user does not have to wait each time the application is started. The simplest solution would be for the application to store the cluster tree in a file, and reload the file each time the application is restarted. When the photo collection is changed, the cluster tree would have to be regenerated, which would still force the user to wait. A more user friendly solution might be to integrate the generation of the cluster tree into the ImageProvider, so that each time a new photo is added to the collection, parts of the cluster tree could be rebuilt in a low-priority background thread. This might be a bit messy though, as the cluster tree can not be regarded as a fundamental part of a geotagged photo collection, but rather an application specific database of metadata.

9 References

[1] http://code.google.com/android

[2] *Geographic Location Tags on Digital Images.*
Kentaro Toyama, Ron Logan, Asta Roseway.
Proceedings of the eleventh ACM international conference on Multimedia.

[3] *NIMA Technical Report TR8350.2, "Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems", Third Edition.*
http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html

[4] http://www.navcen.uscg.gov/loran/Default.htm

[5] *Cell-ID location technique, limits and benefits: an experimental study.*
Emiliano Trevisani, Andrea Vitaletti.
Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04).

[6] http://www.google.com/mobile/gmm/mylocation/

[7] http://developer.yahoo.com/yrb/zonetag/locatecell.html

[8] http://www.celldb.org/

[9] http://gsmloc.org/

[10] http://www.cellspotting.com/

[11] http://www.losangeles.af.mil/library/factsheets/factsheet.asp?id=5311

[12] http://www.losangeles.af.mil/library/factsheets/factsheet.asp?id=5325

[13] http://www.navcen.uscg.gov/gps/default.htm

[14] http://www.navcen.uscg.gov/gps/geninfo/global.htm

[15] http://www.sirf.com/products/gps_chip3e.html

[16] *Navstar GPS and GLONASS: global satellite navigation systems.*
Professor P. Daly.
Electronics & Communication Engineering Journal, Volume: 5, Issue: 6

[17] *Standardization of Mobile Phone Positioning for 3G Systems.*
Yilin Zhao.
IEEE Communications Magazine, Volume: 40, Issue: 7

[18] *Mobile Phone Location Determination and Its Impact on Intelligent Transportation Systems.*
Yilin Zhao.
IEEE Transactions on Intelligent Transportation Systems, Volume: 1, Issue: 1

[19] *Geolocation and Assisted GPS.*
Goran M. Djuknic, Robert E. Richton.
Computer, vol. 34, no. 2

[20] *Measured Performance of 5-GHz 802.11a Wireless LAN Systems.*
James C. Chen, Jeffrey M. Gilbert.
White paper, Atheros Communications

[21] *Positioning with IEEE 802.1lb Wireless LAN.*
A. Kotanen, M. Hannikainen, H. Leppakoski, T.D. Hamalainen.
14th IEEE Proceedings on Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. Volume: 3

[22] *Indoor Positioning Techniques Based on Wireless LAN.*
Binghao Li, James Salter, Andrew G. Dempster, Chris Rizos.
1st IEEE International Conference on Wireless Broadband and Ultra Wideband Communications

[23] *LOCADIO: Inferring Motion and Location from Wi-Fi Signal Strengths.*
John Krumm, Eric Horvitz.
First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)
[24] http://www.skyhookwireless.com/
[25] http://www.wigle.net/
[26] *Automatic Organization for Digital Photographs with Geographic Coordinates.*
Mor Naaman, Yee Jiun Song, Andreas Paepcke, Hector Garcia-Molina.
International Conference on Digital Libraries. ICDL 2004
[27] *Context Data in Geo-Referenced Digital Photo Collections.*
Mor Naaman, Susumu Harada, QianYing Wang, Hector Garcia-Molina, Andreas Paepcke.
Proceedings of the 12th annual ACM international conference on Multimedia
[28] *Temporal event clustering for digital photo collections.*
Matthew Cooper, Jonathan Foote, Andreas Girgensohn, Lynn Wilcox.
ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), Volume 1 , Issue 3
[29] *Time as essence for photo browsing through personal digital libraries.*
Adrian Graham, Hector Garcia-Molina, Andreas Paepcke, Terry Winograd.
Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries
[30] *Mobile Access to Personal Digital Photograph Archives.*
Cathal Gurrin, Gareth J. F. Jones, Hyowon Lee, Neil O'Hare, Alan F. Smeaton, Noel Murphy.
Proceedings of the 7th international conference on Human computer interaction with mobile devices & services
[31] http://flickr.com/
[32] http://www.panoramio.com/
[33] http://maps.google.com/
[34] http://earth.google.com/
[35] http://picasa.google.com/
[36] http://www.microsoft.com/prophoto/
[37] http://www.shozu.com/
[38] http://youtube.com/
[39] http://www.facebook.com/
[40] http://www.blogger.com/
[41] http://locr.com/
[42] http://share.ovi.com/
[43] http://www.symbian.com/
[44] http://www.microsoft.com/Windowsmobile/
[45] http://www.blackberry.com/
[46] http://www.openhandsetalliance.com/
[47] http://www.siliconrepublic.com/news/news.nv?storyid=single4143
[48] http://blog.searchenginewatch.com/blog/041201-120143
[49] http://www.businessweek.com/technology/content/aug2005/tc20050817_0949_tc024.htm
[50] http://www.openhandsetalliance.com/press_110507.html
[51] http://www.openhandsetalliance.com/press_111207.html
[52] http://online.wsj.com/article/SB121418837707895947.html?mod=googlenews_wsj
[53] http://code.google.com/android/adc.html
[54] http://android-developers.blogspot.com/

[55] *Anatomy & Physiology of an Android.*
Patrick Brady.
Google I/O 2008
[56] http://www.apache.org/licenses/LICENSE-2.0
[57] http://git.android.com/
[58] http://java.sun.com/javase/
[59] http://java.sun.com/javame/
[60] http://www.eclipse.org/
[61] http://www.sqlite.org/
[62] http://www.oracle.com/database/
[63] http://www.microsoft.com/sql/
[64] http://www.mysql.com/
[65] http://sqlite.org/mostdeployed.html
[66] http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
[67] *Dalvik VM Internals.*
Dan Bornstein.
Google I/O 2008
[68] http://bellard.org/qemu/
[69] *Hierarchical clustering schemes.*
Stephen C. Johnson.
Psychometrika, Volume 32, Number 3
[70] On Geodetic Distance Computations in Spatial Modeling.
Sudipto Banerjee.
Biometrics, Volume 61, Number 2
[71] http://code.google.com/apis/maps/documentation/services.html#Geocoding_Direct
[72] http://developer.yahoo.com/maps/rest/V1/geocode.html
[73] http://www.myspace.com/
[74] http://www.linkedin.com/
[75] http://www.yellowpages.com/
[76] http://www.gulesider.no/
[77] http://wikipedia.org/
[78] http://de.wikipedia.org/wiki/Wikipedia:WikiProjekt_Georeferenzierung/Wikipedia-World/en
[79] http://www.exif.org/
[80] http://www.fightingquaker.com/sanselan/

Appendix A: User Survey – Original Norwegian version

Alle spørsmål skal besvares med tallene 1 – 5, der 1 = helt uenig og 5 = helt enig.

Del 1: Implementerte funksjoner

Funksjon 1: Se posisjonen til hvor et bilde ble tatt i et kart **Svar:**

Denne funksjonen er nyttig. _____

Denne funksjonen er intuitiv å bruke. _____

Denne funksjonen hadde vært enklere å bruke på en PC
med større skjerm, samt tastatur og mus. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Funksjon 2: Kart som viser posisjon for alle bilder

Denne funksjonen er nyttig. _____

Denne funksjonen er intuitiv å bruke. _____

Denne funksjonen hadde vært enklere å bruke på en PC
med større skjerm, samt tastatur og mus. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Funksjon 3: Hvem bor her?

Denne funksjonen er nyttig. _____

Denne funksjonen er intuitiv å bruke. _____

Denne funksjonen hadde vært enklere å bruke på en PC
med større skjerm, samt tastatur og mus. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Del 2: Ikke-implementerte funksjoner

Funksjon 4: Samme som funksjon 3, men kobler seg til telefonkatalogen på internett for å slå opp adressen til kontakter man ikke har lagret adressen for.

Denne funksjonen ville gjort funksjon 3 mer nyttig. _____

Jeg ville benyttet denne funksjonen dersom jeg også benyttet funksjon 3. _____

Funksjon 5: Samme som funksjon 3, men i tillegg til kontaktlisten på mobiltelefonen hentes også adresser fra vennelisten på nettsider for sosiale nettverk (f.eks. Facebook).

Denne funksjonen ville gjort funksjon 3 mer nyttig. _____

Jeg ville benyttet denne funksjonen dersom jeg også benyttet funksjon 3. _____

Funksjon 6: Brukeren ser gjennom kontaktlisten sin og ber om å få opp alle bilder som er tatt hjemme hos en spesifikk kontakt.

Denne funksjonen virker nyttig. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Funksjon 7: Når man har valgt et bilde, så kan man få hjelp til å navigere til stedet der bildet ble tatt (slik som navigasjonssystemer for bil fungerer i dag).

Denne funksjonen virker nyttig. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Funksjon 8: Wikipedia har lagret posisjon for en del artikler. Når man ser på et bilde så kan man få opp link til artikler som omhandler noe som er i nærheten av der bildet ble tatt.
(F.eks. et bilde av Nidarosdomen kan gi link til Wikipedia-artikkel om Nidarosdomen)

Denne funksjonen virker nyttig. _____

Denne funksjonen passer godt for bruk på en mobiltelefon. _____

Jeg ville benyttet denne funksjonen dersom jeg hadde en Android-telefon. _____

Appendix B: User Survey – English translation

All questions must be answered by the numbers 1 – 5, where 1 = completely agree and 5 = completely disagree.

Part 1: Implemented functions

Function 1: See the position of where a photo was taken in a map	**Answer:**
This function is useful.	_____
This function is intuitive to use.	_____
This function would be easier to use on a PC with larger screen, a keyboard and mouse.	_____
This function is a good fit for use on a mobile phone.	_____
I would use this function if I had an Android phone.	_____

Function 2: Map that shows position for all photos

This function is useful.	_____
This function is intuitive to use.	_____
This function would be easier to use on a PC with larger screen, a keyboard and mouse.	_____
This function is a good fit for use on a mobile phone.	_____
I would use this function if I had an Android phone.	_____

Function 3: Who lives here?

This function is useful.	_____
This function is intuitive to use.	_____
This function would be easier to use on a PC with larger screen, a keyboard and mouse.	_____
This function is a good fit for use on a mobile phone.	_____
I would use this function if I had an Android phone.	_____

Part 2: Non-implemented functions

Function 4: Same as function 3, but connects to the phone directory on the internet to look up the address of contacts which does not have the address stored.

This function would have made function 3 more useful. _____

I would use this function if I also used function 3. _____

Function 5: Same as function 3, but in addition to the contact list on the phone addresses are also collected from the friends list on social networking websites (e.g. Facebook).

This function would have made function 3 more useful. _____

I would use this function if I also used function 3. _____

Function 6: The user looks through the contact list and requests to bring up all photos taken at the home of a specific contact.

This function seems useful. _____

This function is a good fit for use on a mobile phone. _____

I would have used this function if I had an Android phone. _____

Function 7: When you have selected a photo, you can get help to navigate to the place where the photo was taken (like navigation systems for cars currently work).

This function seems useful. _____

This function is a good fit for use on a mobile phone. _____

I would have used this function if I had an Android phone. _____

Function 8: Wikipedia have stored the position for some articles. While looking at a photo you can get a link to articles dealing with something close to where the photo was taken. (For instance a photo of the Nidaros cathedral may give you a link to a Wikipedia article concerning the cathedral)

This function seems useful. _____

This function is a good fit for use on a mobile phone. _____

I would have used this function if I had an Android phone. _____

Appendix C: User Survey – Results

	User 1	User 2	User 3	User 4	User 5	User 6	User 7	User 8	User 9	User 10	Average	Median
Function 1												
Question 1.1	3	5	5	5	3	3	5	5	3	4	4,1	4,5
Question 1.2	4	5	5	5	5	5	5	5	5	3	4,7	5
Question 1.3	5	3	3	2	5	1	1	2	4	2	2,8	2,5
Question 1.4	5	5	4	5	5	5	5	5	3	5	4,7	5
Question 1.5	2	3	3	5	3	4	5	4	5	4	3,8	4
Function 2												
Question 2.1	3	4	4	4	5	5	4	4	3	4	4	4
Question 2.2	4	4	5	4	3	4	5	4	5	3	4,1	4
Question 2.3	5	5	4	4	5	4	4	1	4	3	3,9	4
Question 2.4	5	3	3	4	3	4	3	3	4	5	3,7	3,5
Question 2.5	2	3	4	4	3	5	4	3	3	5	3,6	3,5
Function 3												
Question 3.1	4	4	5	3	2	3	4	4	2	4	3,5	4
Question 3.2	4	5	5	5	4	5	5	5	5	4	4,7	5
Question 3.3	3	3	2	1	5	1	1	1	4	2	2,3	2
Question 3.4	5	5	5	5	4	5	5	5	4	5	4,8	5
Question 3.5	3	5	5	3	1	3	5	5	4	5	3,9	4,5
Function 4												
Question 4.2	5	5	5	5	3	4	5	5	5	5	4,7	5
Question 4.2	5	5	5	5	3	4	5	4	5	5	4,6	5
Function 5												
Question 5.1	5	4	5	5	1	4	5	5	5	5	4,4	5
Question 5.2	5	4	5	5	1	4	5	5	3	5	4,2	5

Function 6

Question 6.1	5	5	3	4	3	5	5	4	2	5	4,1	4,5
Question 6.2	5	5	3	5	3	4	3	5	3	5	4,1	4,5
Question 6.3	3	5	3	4	2	4	4	2	3	5	3,5	3,5

Function 7

Question 7.1	5	3	5	3	4	5	3	3	5	5	4,1	4,5
Question 7.2	5	5	2	4	5	5	4	3	4	5	4,2	4,5
Question 7.3	5	3	2	3	3	5	3	3	5	5	3,7	3

Function 8

Question 8.1	5	5	4	4	4	5	3	4	5	5	4,4	4,5
Question 8.2	5	5	3	3	5	3	2	3	5	5	3,9	4
Question 8.3	5	5	3	3	2	3	1	5	5	4	3,6	3,5

Appendix D: Source Code of ImageBrowser

ContactList.java

```
package no.ntnu.idi.android.imagebrowser;

import android.app.ListActivity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.BaseColumns;
import android.provider.Contacts;
import android.view.View;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
import android.widget.Toast;

public class ContactList extends ListActivity {

  public static final String PERSON_ID_LIST = "personIdList";
  private ListAdapter adapter = null;

  @Override
  protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    Bundle extras = getIntent().getExtras();
    int[] personIDs = (extras == null) ? null :
      (int[])extras.getSerializable(PERSON_ID_LIST);

    if (personIDs == null || personIDs.length == 0) {
      Toast.makeText(this, "No matching contacts.",
        Toast.LENGTH_SHORT).show();
      return;
    } else {
      Uri peopleUri = Contacts.People.CONTENT_URI;

      StringBuilder sb = new StringBuilder();
      // The "people." is an ugly hack to get around a bug in the current
      // version of Android.
      sb.append("people." + BaseColumns._ID);
      sb.append("=");
      sb.append(personIDs[0]);
      for (int i = 1; i < personIDs.length; i++) {
        sb.append(" OR ");
        sb.append("people." + BaseColumns._ID);
        sb.append("=");
        sb.append(personIDs[i]);
      }
      String selection = sb.toString();

      String[] projection = new String[]{BaseColumns._ID,
        Contacts.People.NAME};

      Cursor cursor = this.managedQuery(peopleUri, projection,
        selection, null);
      if (cursor == null) {
```

```
        Toast.makeText(this, "Query error.", Toast.LENGTH_SHORT).show();
        return;
      }

      adapter = new SimpleCursorAdapter(this,
          android.R.layout.simple_list_item_1,
          cursor, new String[]{Contacts.People.NAME},
          new int[]{android.R.id.text1});
      setListAdapter(adapter);
    }
  }

  @Override
  protected void onListItemClick(ListView l, View v, int position, long id)
{
    if (adapter != null) {
      Cursor cursor = (Cursor)adapter.getItem(position);
      int idColumn = cursor.getColumnIndex(BaseColumns._ID);
      int personID = cursor.getInt(idColumn);

      Uri peopleUri = Contacts.People.CONTENT_URI;
      peopleUri =
peopleUri.buildUpon().appendPath(Integer.toString(personID)).build();
      Intent i = new Intent(Intent.VIEW_ACTION);
      i.setData(peopleUri);
      startActivity(i);
    }
  }

}
```

EnhancedMapView.java

```
package no.ntnu.idi.android.imagebrowser;

import android.content.Context;
import android.view.MotionEvent;

import com.google.android.maps.MapView;

public class EnhancedMapView extends MapView {

  private OnTouchEventListener listener = null;

  public EnhancedMapView(Context context) {
    super(context);
  }

  public void setOnTouchEventListener(OnTouchEventListener listener) {
    this.listener = listener;
  }

  @Override
  public boolean onTouchEvent(MotionEvent ev) {
    if (listener != null) {
      if (listener.onTouchEvent(ev)) {
        return true;
      } else {
        return super.onTouchEvent(ev);
      }
    }

    return super.onTouchEvent(ev);
  }

  public interface OnTouchEventListener {
    public boolean onTouchEvent(MotionEvent ev);
  }

}
```

Geocoder.java

```
package no.ntnu.idi.android.imagebrowser;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.InetSocketAddress;
import java.net.MalformedURLException;
import java.net.Socket;
import java.net.URL;
import java.net.URLEncoder;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;

import android.app.Activity;
import android.content.ContentValues;
import android.database.Cursor;
import android.location.Location;
import android.net.Uri;
import android.provider.BaseColumns;
import android.provider.Contacts;

public class Geocoder {

  private static final String API_KEY = "INSERT PRIVATE API KEY HERE";
  private String partialUrl =
"http://maps.google.com/maps/geo?output=csv&key=" + API_KEY + "&q=";
  private Activity activity = null;

  public Geocoder(Activity activity) {
    this.activity = activity;
  }

  public Location getLocationFromAddress(String address) {
    try {
      Socket socket = new Socket();
      socket.connect(new InetSocketAddress("maps.google.com", 80));

      OutputStream os = socket.getOutputStream();
      BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(os));
      String encodedAddress = URLEncoder.encode(address, "UTF-8");
      bw.write("GET /maps/geo?output=csv&key=" + API_KEY
        + "&q=" + encodedAddress);
      bw.write("\nHost: maps.google.com\n\n");
      bw.flush();

      InputStream is = socket.getInputStream();
      BufferedReader br = new BufferedReader(new InputStreamReader(is));
      String s = br.readLine();

      socket.close();

      String[] lineParts = s.split(",");
      if (lineParts[0].equals("200") && lineParts[1].equals("8")) {
```

```
      Location location = new Location();
      location.setLatitude(Double.parseDouble(lineParts[2]));
      location.setLongitude(Double.parseDouble(lineParts[3]));
      return location;
    } else {
      return null;
    }

  } catch (UnknownHostException uhe) {
    return null;
  } catch (IOException ioe) {
    return null;
  } catch (Exception e) {
    return null;
  }
}

public List<Integer> findContactsNearLocation(Location location,
    int maxDistance) {
  List<Integer> matchingContacts = new ArrayList<Integer>();

  // Retrieve all contacts
  Uri peopleUri = Contacts.People.CONTENT_URI;
  Cursor peopleCursor = activity.managedQuery(peopleUri,
    new String[]{BaseColumns._ID}, null, null);

  if (peopleCursor != null && peopleCursor.first()) {
    int idColumn = peopleCursor.getColumnIndex(BaseColumns._ID);
    do {
      int personID = peopleCursor.getInt(idColumn);
      Uri contactMethodsUri = peopleUri.buildUpon()
        .appendPath(Integer.toString(personID))
        .appendPath("contact_methods").build();
      Cursor contactMethodsCursor = activity.managedQuery(
        contactMethodsUri, new String[]{Contacts.ContactMethods.KIND,
          Contacts.ContactMethods.DATA,
          Contacts.ContactMethods.AUX_DATA},
        Contacts.ContactMethods.KIND + "="
        + Contacts.ContactMethods.LOCATION_KIND
        + " OR (" + Contacts.ContactMethods.KIND + "="
        + Contacts.ContactMethods.POSTAL_KIND
        + " AND " + Contacts.ContactMethods.TYPE + "="
        + Contacts.ContactMethods.POSTAL_KIND_HOME_TYPE + ")",
          null);

      if (contactMethodsCursor != null) {
        if (contactMethodsCursor.first()) {
          int kindColumn = contactMethodsCursor
            .getColumnIndex(Contacts.ContactMethods.KIND);
          int dataColumn = contactMethodsCursor
            .getColumnIndex(Contacts.ContactMethods.DATA);
          int auxDataColumn = contactMethodsCursor
            .getColumnIndex(Contacts.ContactMethods.AUX_DATA);
          Location cachedLocation = null;
          String cachedAddress = null;
          String currentAddress = null;
          do {
            int kind = contactMethodsCursor.getInt(kindColumn);
            if (kind == Contacts.ContactMethods.POSTAL_KIND) {
              currentAddress =
                contactMethodsCursor.getString(dataColumn);
```

```
            } else if (kind == Contacts.ContactMethods.LOCATION_KIND) {
              String locationString = contactMethodsCursor
                .getString(dataColumn);
              String[] locationParts = locationString.split(",");
              if (locationParts.length == 2) {
                cachedLocation = new Location();
                cachedLocation.setLatitude(
                  Double.parseDouble(locationParts[0]));
                cachedLocation.setLongitude(
                  Double.parseDouble(locationParts[1]));
              }
              cachedAddress = contactMethodsCursor
                .getString(auxDataColumn);
            }
          } while (contactMethodsCursor.next());

          if (currentAddress != null) {

            if (cachedAddress != null &&
                !cachedAddress.equals(currentAddress)) {
              // Retrieve new location
              Location newLocation =
                getLocationFromAddress(currentAddress);
              cachedLocation = newLocation;
            } else if (cachedAddress == null) {
              // Retrieve new location
              Location newLocation =
                getLocationFromAddress(currentAddress);

              // Insert new location
              ContentValues values = new ContentValues();
              values.put(Contacts.ContactMethods.PERSON_ID, personID);
              values.put(Contacts.ContactMethods.KIND,
                Contacts.ContactMethods.LOCATION_KIND);
              values.put(Contacts.ContactMethods.DATA,
                newLocation.getLatitude() + ","
                + newLocation.getLongitude());
              values.put(Contacts.ContactMethods.AUX_DATA,
                currentAddress);
              activity.getContentResolver().insert(contactMethodsUri,
                values);

              cachedLocation = newLocation;
            }

            // check if the person lives near the given location
            if (location.distanceTo(cachedLocation) <= maxDistance) {
              matchingContacts.add(personID);
            }
          }
        }
      }
    } while (peopleCursor.next());
  }

  return matchingContacts;
 }
}
```

GeotaggedBitmap.java

```
package no.ntnu.idi.android.imagebrowser;

import android.graphics.Bitmap;
import android.location.Location;

public class GeotaggedBitmap {
  private Bitmap bitmap;
  private Location location;

  public GeotaggedBitmap(Bitmap bitmap, Location location) {
    this.bitmap = bitmap;
    this.location = location;
  }

  public GeotaggedBitmap(Bitmap bitmap, double latitude, double longitude){
    this.bitmap = bitmap;
    Location location = new Location();
    location.setLatitude(latitude);
    location.setLongitude(longitude);
    this.location = location;
  }

  public Bitmap getBitmap() {
    return bitmap;
  }

  public void setBitmap(Bitmap bitmap) {
    this.bitmap = bitmap;
  }

  public Location getLocation() {
    return location;
  }

  public void setLocation(Location location) {
    this.location = location;
  }
}
```

GeotaggedImageGroup.java

```
package no.ntnu.idi.android.imagebrowser;

import java.util.ArrayList;
import java.util.List;

import android.location.Location;

public class GeotaggedImageGroup {
  private List<GeotaggedImageID> imageIDs = null;

  public GeotaggedImageGroup() {
    imageIDs = new ArrayList<GeotaggedImageID>();
  }

  public GeotaggedImageGroup(GeotaggedImageID imageID) {
    this();
    add(imageID);
  }

  public void add(GeotaggedImageID imageID) {
    imageIDs.add(imageID);
  }

  public void add(GeotaggedImageGroup images) {
    for (GeotaggedImageID image : images.getImageIDs()) {
      add(image);
    }
  }

  public Location getAverageLocation() {
    Location averageLocation = new Location();

    Location location;
    double latitudeSum = 0.0;
    double longitudeSum = 0.0;
    for (GeotaggedImageID image : imageIDs) {
      location = image.getLocation();
      latitudeSum += location.getLatitude();
      longitudeSum += location.getLongitude();
    }

    averageLocation.setLatitude(latitudeSum / imageIDs.size());
    averageLocation.setLongitude(longitudeSum / imageIDs.size());

    return averageLocation;
  }

  public List<GeotaggedImageID> getImageIDs() {
    return imageIDs;
  }

  public double calculateMinDistanceTo(GeotaggedImageGroup otherGroup) {
    List<GeotaggedImageID> otherImageIDs = otherGroup.getImageIDs();
    double minDistance = Double.MAX_VALUE;
    Location location1, location2;
    double distance;
    for (GeotaggedImageID imageID : imageIDs) {
      location1 = imageID.getLocation();
      for (GeotaggedImageID otherImageID : otherImageIDs) {
```

```
        location2 = otherImageID.getLocation();
        distance = location1.distanceTo(location2);

        if (distance < minDistance)
          minDistance = distance;
      }
    }

    return minDistance;
  }
}
```

GeotaggedImageID.java

```
package no.ntnu.idi.android.imagebrowser;

import android.location.Location;

public class GeotaggedImageID {
  private int imageID;
  private Location location;

  public GeotaggedImageID(int imageID, Location location) {
    this.imageID = imageID;
    this.location = location;
  }

  public int getImageID() {
    return imageID;
  }

  public void setImageID(int imageID) {
    this.imageID = imageID;
  }

  public Location getLocation() {
    return location;
  }

  public void setLocation(Location location) {
    this.location = location;
  }
}
```

ImageAdapter.java

```
package no.ntnu.idi.android.imagebrowser;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import no.ntnu.idi.android.imageprovider.ImageProvider;
import android.app.Activity;
import android.content.ContentUris;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Matrix;
import android.net.Uri;
import android.provider.BaseColumns;
import android.view.View;
import android.view.ViewGroup;
import android.view.ViewGroup.LayoutParams;
import android.widget.BaseAdapter;
import android.widget.Gallery;
import android.widget.ImageView;

public class ImageAdapter extends BaseAdapter {

  private Activity activity;
  private ImageConsumer imageConsumer;

  public ImageAdapter(Activity activity, ImageConsumer imageConsumer) {
    this.activity = activity;
    this.imageConsumer = imageConsumer;
  }

  public int getCount() {
    return (imageConsumer == null) ? 0 : imageConsumer.getCount();
  }

  public Object getItem(int position) { return position; }

  public long getItemId(int position) { return position; }

  public View getView(int position, View convertView, ViewGroup parent) {
    ImageView i = new ImageView(activity);
    i.setImageBitmap(imageConsumer.loadBitmap(position,
        ImageConsumer.ImageSize.THUMBNAIL));
    i.setAdjustViewBounds(true);
    i.setLayoutParams(new Gallery.LayoutParams(
        LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
    i.setBackground(android.R.drawable.picture_frame);
    return i;
  }

  public float getAlpha(boolean focused, int offset) {
    return Math.max(0.2f, 1.0f - (0.2f * Math.abs(offset)));
  }

  public float getScale(boolean focused, int offset) {
    return Math.max(0, offset == 0 ? 1.0f : 0.6f);
  }
}
```

ImageBrowser.java

```
package no.ntnu.idi.android.imagebrowser;

import java.util.List;
import android.app.Activity;
import android.content.Intent;
import android.graphics.drawable.BitmapDrawable;
import android.location.Location;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.view.Menu.Item;
import android.view.ViewGroup.LayoutParams;
import android.widget.AdapterView;
import android.widget.Gallery;
import android.widget.ImageSwitcher;
import android.widget.ImageView;
import android.widget.Toast;
import android.widget.ViewSwitcher;

public class ImageBrowser extends Activity
  implements ViewSwitcher.ViewFactory, AdapterView.OnItemSelectedListener {

  public static final String IMAGE_IDS = "image_ids";
  private ImageSwitcher imageSwitcher = null;
  private ImageConsumer imageConsumer = null;
  private Location currentLocation = null;
  private static final int MENU_MAP = Menu.FIRST;
  private static final int MENU_LIVES_HERE = Menu.FIRST + 1;
  private static final int MENU_MAP_ALL = Menu.FIRST + 2;

    @Override
    public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      setContentView(R.layout.imageswitcher);

      imageSwitcher = (ImageSwitcher)findViewById(R.id.imageSwitcher);
      imageSwitcher.setFactory(this);
      Bundle extras = getIntent().getExtras();
      int[] imageIDs = (extras == null) ? null :
        (int[])extras.getSerializable(IMAGE_IDS);
      if (imageIDs != null) {
        imageConsumer = new ImageConsumer(this, imageIDs);
      } else {
        imageConsumer = new ImageConsumer(this);
      }
      Gallery gallery = (Gallery)findViewById(R.id.gallery);
      gallery.setAdapter(new ImageAdapter(this, imageConsumer));
      gallery.setOnItemSelectedListener(this);
    }

  @Override
  public void onItemSelected(AdapterView parent, View view,
      int position, long id) {
    // Load the new image into the ImageSwitcher
    GeotaggedBitmap geotaggedBitmap =
      imageConsumer.loadGeotaggedBitmap(position,
      ImageConsumer.ImageSize.SCREENSIZE);
    currentLocation = geotaggedBitmap.getLocation();
    imageSwitcher.setImageDrawable(
```

```
      new BitmapDrawable(geotaggedBitmap.getBitmap()));
  }

  @Override
  public void onNothingSelected(AdapterView arg0) { }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(0, MENU_MAP, "Show this photo in a map");
    menu.add(0, MENU_LIVES_HERE, "Who lives here?");
    menu.add(0, MENU_MAP_ALL, "Show a map with all photos");
    return super.onCreateOptionsMenu(menu);
  }

  @Override
  public boolean onOptionsItemSelected(Item item) {
    switch (item.getId()) {
    case MENU_MAP:
      Intent mapIntent = new Intent(this, ImageMap.class);
      mapIntent.putExtra(ImageMap.LOCATION, currentLocation);
      startActivity(mapIntent);
      break;
    case MENU_LIVES_HERE:
      Geocoder geocoder = new Geocoder(this);
      List<Integer> contactList =
        geocoder.findContactsNearLocation(currentLocation, 100);

      if (contactList.size() <= 0) {
        Toast.makeText(this, "None of your contacts lives here.",
          Toast.LENGTH_SHORT).show();
      } else {
        int[] contactIDs = new int[contactList.size()];
        int i = 0;
        for (Integer contactID : contactList) {
          contactIDs[i++] = contactID;
        }
        Intent contactListIntent = new Intent(this, ContactList.class);
        contactListIntent.putExtra(ContactList.PERSON_ID_LIST, contactIDs);
        startActivity(contactListIntent);
      }
      break;
    case MENU_MAP_ALL:
      Intent mapAllIntent = new Intent(this, ImageMap.class);
      startActivity(mapAllIntent);
      break;
    }

    return super.onOptionsItemSelected(item);
  }

  @Override
  public View makeView() {
    ImageView i = new ImageView(this);
        i.setBackgroundColor(0xFF000000);
        i.setScaleType(ImageView.ScaleType.FIT_CENTER);
        i.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
          LayoutParams.FILL_PARENT));
        return i;
  }
}
```

ImageConsumer.java

```
package no.ntnu.idi.android.imagebrowser;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

import no.ntnu.idi.android.imageprovider.ImageProvider;
import android.app.Activity;
import android.content.ContentUris;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.location.Location;
import android.net.Uri;
import android.provider.BaseColumns;

public class ImageConsumer {

  public enum ImageSize {
    ORIGINAL,
    SCREENSIZE,
    THUMBNAIL
  }

  private Activity activity;
  private int[] imageIDs;

  //private Uri imageUri = ImageProvider.CONTENT_URI;
  private Uri imageUri = ImageProvider.FLICKR_CONTENT_URI;

  public ImageConsumer(Activity activity) {
    this.activity = activity;
    this.imageIDs = retrieveImageIDs();
  }

  public ImageConsumer(Activity activity, int[] imageIDs) {
    this.activity = activity;
    this.imageIDs = imageIDs;
  }

  public int getCount() {
    return (imageIDs == null) ? 0 : imageIDs.length;
  }

  private int[] retrieveImageIDs() {
      Cursor cursor = activity.managedQuery(imageUri,
        new String[]{BaseColumns._ID}, null, null, BaseColumns._ID);
      if (cursor == null) {
        return new int[0];
      }

      int[] imageIDs = new int[cursor.count()];

      int idColumn = cursor.getColumnIndex(BaseColumns._ID);
      int i = 0;

      if (cursor.first()) {
        do {
          // put id into correct position in array
```

```
        imageIDs[i++] = cursor.getInt(idColumn);
      } while (cursor.next());
    }

    return imageIDs;
  }

public GeotaggedImageID[] retrieveGeotaggedImageIDs() {
  return retrieveGeotaggedImageIDs(null, null, null, null);
}

public GeotaggedImageID[] retrieveGeotaggedImageIDs(Double minLatitude,
    Double maxLatitude, Double minLongitude, Double maxLongitude) {

  Cursor cursor;
  if (minLatitude != null && maxLatitude != null && minLongitude != null
      && maxLongitude != null) {
    cursor = activity.managedQuery(imageUri,
      new String[]{BaseColumns._ID,
        ImageProvider.LATITUDE, ImageProvider.LONGITUDE},
      "latitude > ? AND latitude < ? "
      + " AND longitude > ? AND longitude < ?",
      new String[]{minLatitude.toString(), maxLatitude.toString(),
        minLongitude.toString(), maxLongitude.toString()},
      BaseColumns._ID);
  } else {
    cursor = activity.managedQuery(imageUri,
      new String[]{BaseColumns._ID,
        ImageProvider.LATITUDE, ImageProvider.LONGITUDE},
      null, null, BaseColumns._ID);
  }

    if (cursor == null) {
      return new GeotaggedImageID[0];
    }

    GeotaggedImageID[] imageIDs = new GeotaggedImageID[cursor.count()];

    int idColumn = cursor.getColumnIndex(BaseColumns._ID);
    int latitudeColumn = cursor.getColumnIndex(ImageProvider.LATITUDE);
    int longitudeColumn = cursor.getColumnIndex(ImageProvider.LONGITUDE);
    int i = 0;

    if (cursor.first()) {
      do {
        // put id into correct position in array
        int imageID = cursor.getInt(idColumn);
        double latitude = cursor.getDouble(latitudeColumn);
        double longitude = cursor.getDouble(longitudeColumn);
        Location location = new Location();
        location.setLatitude(latitude);
        location.setLongitude(longitude);
        imageIDs[i++] = new GeotaggedImageID(imageID, location);
      } while (cursor.next());
    }

    return imageIDs;

}

  public Bitmap loadBitmap(int position, ImageSize imageSize) {
```

```
Uri specificImageUri = ContentUris.appendId(imageUri.buildUpon(),
  imageIDs[position]).build();
Cursor cursor = activity.managedQuery(specificImageUri,
  null, null, null);

if (cursor.first()) {
  int imageColumn;
  if (imageSize == ImageSize.THUMBNAIL) {
    imageColumn = cursor.getColumnIndex(ImageProvider.IMAGE_THUMBNAIL);
  } else if (imageSize == ImageSize.SCREENSIZE) {
    imageColumn =
      cursor.getColumnIndex(ImageProvider.IMAGE_SCREENSIZE);
  } else {
    imageColumn = cursor.getColumnIndex(ImageProvider.IMAGE_ORIGINAL);
  }

  String path = cursor.getString(imageColumn);

  try {
    InputStream is = activity.getContentResolver()
      .openInputStream(Uri.parse("file:" + path));
    Bitmap bitmap = BitmapFactory.decodeStream(is);
    is.close();
    return bitmap;
  } catch (FileNotFoundException e) {
    return null;
  } catch (IOException ioe) {
    return null;
  }
}

return null;
}

public GeotaggedBitmap loadGeotaggedBitmap(int position,
    ImageSize imageSize) {
Uri specificImageUri = ContentUris.appendId(imageUri.buildUpon(),
  imageIDs[position]).build();
Cursor cursor = activity.managedQuery(specificImageUri,
  null, null, null);

if (cursor.first()) {
  int imageColumn;
  if (imageSize == ImageSize.THUMBNAIL) {
    imageColumn = cursor.getColumnIndex(ImageProvider.IMAGE_THUMBNAIL);
  } else if (imageSize == ImageSize.SCREENSIZE) {
    imageColumn =
      cursor.getColumnIndex(ImageProvider.IMAGE_SCREENSIZE);
  } else {
    imageColumn = cursor.getColumnIndex(ImageProvider.IMAGE_ORIGINAL);
  }
  int latitudeColumn = cursor.getColumnIndex(ImageProvider.LATITUDE);
  int longitudeColumn = cursor.getColumnIndex(ImageProvider.LONGITUDE);

  String path = cursor.getString(imageColumn);
  double latitude = cursor.getDouble(latitudeColumn);
  double longitude = cursor.getDouble(longitudeColumn);

  try {
    InputStream is = activity.getContentResolver()
      .openInputStream(Uri.parse("file:" + path));
```

```
        Bitmap bitmap = BitmapFactory.decodeStream(is);
        is.close();
        return new GeotaggedBitmap(bitmap, latitude, longitude);
      } catch (FileNotFoundException e) {
        return null;
      } catch (IOException ioe) {
        return null;
      }
    }

    return null;
    }
}
```

ImageMap.java

```
package no.ntnu.idi.android.imagebrowser;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;

import android.content.Intent;
import android.location.Location;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.widget.Toast;
import android.widget.ZoomDialog;
import android.widget.ZoomSlider;

import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import com.google.android.maps.OverlayController;
import com.google.android.maps.Point;

public class ImageMap extends MapActivity implements
MapView.OnLongPressListener, EnhancedMapView.OnTouchEventListener,
ZoomSlider.OnZoomChangedListener {

  public static final String LOCATION = "location";
  private static final int MIN_SPACING = 25;

  private MapView mapView = null;
  private MapController controller = null;
  private OverlayController overlayController = null;
  private Location location = null;
  private ImageConsumer imageConsumer;
  private GeotaggedImageID[] imageIDs = null;
  private Collection<GeotaggedImageGroup> currentGroups = null;
  private MapPinOverlay[] overlays = null;
  private boolean initialized = false;
  private ArrayList<ArrayList<GeotaggedImageGroup>> clusterTree = null;
  private ArrayList<Double> levelMinimums;

  @Override
  protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    mapView = new EnhancedMapView(this);
    mapView.setOnLongPressListener(this);
    ((EnhancedMapView)mapView).setOnTouchEventListener(this);
    controller = mapView.getController();
    overlayController = mapView.createOverlayController();
    setContentView(mapView);

    Bundle extras = getIntent().getExtras();
    location = (extras == null) ? null :
      (Location)extras.getParcelable(LOCATION);

    if (location != null) {
      float lat = (float)location.getLatitude();
      float lon = (float)location.getLongitude();
```

```
    controller.animateTo(new Point((int)(lat * 1000000),
      (int)(lon * 1000000)));
    controller.zoomTo(15);
    overlayController.add(new MapPinOverlay(this, location.getLatitude(),
      location.getLongitude()), false);
  } else {
    // Retrieve list of all images
    imageConsumer = new ImageConsumer(this);
    imageIDs = imageConsumer.retrieveGeotaggedImageIDs();

    // Find zoom span
    double minLatitude = Double.MAX_VALUE;
    double maxLatitude = Double.MIN_VALUE;
    double minLongitude = Double.MAX_VALUE;
    double maxLongitude = Double.MIN_VALUE;

    Location location;
    for (GeotaggedImageID imageID : imageIDs) {
      location = imageID.getLocation();
      if (location.getLatitude() > maxLatitude)
        maxLatitude = location.getLatitude();
      if (location.getLatitude() < minLatitude)
        minLatitude = location.getLatitude();
      if (location.getLongitude() > maxLongitude)
        maxLongitude = location.getLongitude();
      if (location.getLongitude() < minLongitude)
        minLongitude = location.getLongitude();
    }

    Point averageLocation =
      new Point((int)(((minLatitude + maxLatitude) / 2) * 1000000),
      (int)(((minLongitude + maxLongitude) / 2) * 1000000));
    controller.animateTo(averageLocation);
    controller.zoomToSpan((int)((maxLatitude - minLatitude) * 1000000),
      (int)((maxLongitude - minLongitude) * 1000000));

    // Group locations
    generateClusterTree(imageIDs);
  }
}

@Override
public void onWindowFocusChanged(boolean hasFocus) {
  super.onWindowFocusChanged(hasFocus);

  if (location == null && hasFocus && !initialized) {
    generateOverlays();
    initialized = true;
  }
}

@Override
public boolean onLongPress(View v, float x, float y) {
  ZoomDialog zoomDialog = new ZoomDialog(this);
  zoomDialog.setParams(0, mapView.getMaxZoomLevel(),
    mapView.getZoomLevel(), this, true);
  zoomDialog.show();

  return true;
}
```

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  if (event.getAction() == MotionEvent.ACTION_UP && overlays != null) {
    for (int i = 0; i < overlays.length; i++) {
      if (overlays[i].coversPixelCoordinate((int)event.getX(),
          (int)event.getY())) {
        GeotaggedImageGroup group =
          (GeotaggedImageGroup)currentGroups.toArray()[i];
        Intent browseIntent = new Intent(this, ImageBrowser.class);
        int[] imageIDs = new int[group.getImageIDs().size()];
        int j = 0;
        for (GeotaggedImageID imageID : group.getImageIDs()) {
          imageIDs[j++] = imageID.getImageID();
        }
        browseIntent.putExtra(ImageBrowser.IMAGE_IDS, imageIDs);
        startActivity(browseIntent);
        return true;
      }
    }
  }

  return false;
}

public void onZoomChanged(ZoomSlider zoomSlider,
    int oldZoom, int newZoom) { }

public void onZoomCompleted() {  }

public void onZoomIn(ZoomSlider zoomSlider, int oldZoom, int newZoom) {
  controller.zoomTo(newZoom);

  if (location == null) {
    generateOverlays();
  }
}

public void onZoomOut(ZoomSlider zoomSlider, int oldZoom, int newZoom) {
  controller.zoomTo(newZoom);

  if (location == null) {
    generateOverlays();
  }
}

private void generateOverlays() {
  overlayController.clear();

  double metersPerPixel = calculateMetersPerPixel();
  double threshold = metersPerPixel * MIN_SPACING;
  currentGroups = findGroupsCached(threshold);

  overlays = new MapPinOverlay[currentGroups.size()];
  int i = 0;
  for (GeotaggedImageGroup group : currentGroups) {
    Location location = group.getAverageLocation();
    MapPinOverlay overlay = new MapPinOverlay(this,
        new Point((int)(location.getLatitude() * 1000000),
        (int)(location.getLongitude() * 1000000)));
    overlayController.add(overlay, false);
    overlays[i++] = overlay;
```

```
    }
  }

  private double calculateMetersPerPixel() {
    // Must use width, since the height of the View is not fully utilized
    // when zoomed out all the way. Height would be easier since 1 minute
    // of arc is defined as 1852 meters

    Point mapCenter = mapView.getMapCenter();

    // Work-around for bug in MapView
    // Reported as Android issue #736
    int longitudeSpan;
    if (mapView.getZoomLevel() == 1) {
      longitudeSpan = 360000000;
    } else {
      longitudeSpan = mapView.getLongitudeSpan();
    }

    Location westBoarderLocation = new Location();
    westBoarderLocation.setLatitude((double)mapCenter.getLatitudeE6()
      / 1000000);
    westBoarderLocation.setLongitude(((mapCenter.getLongitudeE6()
      - (longitudeSpan / 2)) % 180000000) / 1000000);
    Location eastBoarderLocation = new Location();
    eastBoarderLocation.setLatitude((double)mapCenter.getLatitudeE6()
      / 1000000);
    eastBoarderLocation.setLongitude(((mapCenter.getLongitudeE6()
      + (longitudeSpan / 2)) % 180000000) / 1000000);

    double distance = westBoarderLocation.distanceTo(eastBoarderLocation);
    double metersPerPixel = distance / mapView.getWidth();

    return metersPerPixel;
  }

  private void generateClusterTree(GeotaggedImageID[] imageIDs) {
    clusterTree = new ArrayList<ArrayList<GeotaggedImageGroup>>();
    levelMinimums = new ArrayList<Double>();
    HashMap<Integer, GeotaggedImageGroup> imageIDMap =
      new HashMap<Integer, GeotaggedImageGroup>();
    int numImages = imageIDs.length;
    double[][] distanceMatrix = new double[numImages][numImages];

    double distance;
    Location location1, location2;
    ArrayList<GeotaggedImageGroup> clusterList =
      new ArrayList<GeotaggedImageGroup>();
    GeotaggedImageGroup groupToCache = new GeotaggedImageGroup();
    for (int i = 0; i < numImages; i++) {
      imageIDMap.put(i, new GeotaggedImageGroup(imageIDs[i]));
      clusterList.add(new GeotaggedImageGroup(imageIDs[i]));

      location1 = imageIDs[i].getLocation();
      for (int j = i + 1; j < numImages; j++) {
        location2 = imageIDs[j].getLocation();

        distance = location1.distanceTo(location2);

        distanceMatrix[i][j] = distance;
        distanceMatrix[j][i] = distance;
```

```
    }
  }
  clusterTree.add(clusterList);

  double minDistance;
  int minDistanceID1, minDistanceID2;
  do {
    // find min distance
    minDistanceID1 = -1;
    minDistanceID2 = -1;
    minDistance = Double.MAX_VALUE;
    for (int i = 0; i < numImages; i++) {
      for (int j = i + 1; j < numImages; j++) {
        if (distanceMatrix[i][j] < minDistance) {
          minDistance = distanceMatrix[i][j];
          minDistanceID1 = i;
          minDistanceID2 = j;
        }
      }
    }

    if (minDistanceID1 >= 0 && minDistanceID2 >= 0) {
      //  put content of group minDistanceID1 into group minDistanceID2
      ((GeotaggedImageGroup)imageIDMap.get(minDistanceID2))
        .add((GeotaggedImageGroup)imageIDMap.get(minDistanceID1));

      //  remove group minDistanceID1
      imageIDMap.remove(minDistanceID1);

      //   set row and column minDistanceID1 to Double.MAX_VALUE
      for (int i = 0; i < numImages; i++) {
        distanceMatrix[i][minDistanceID1] = Double.MAX_VALUE;
        distanceMatrix[minDistanceID1][i] = Double.MAX_VALUE;
      }

      //  recalculate row and column minDistanceID2
      for (int i = 0; i < numImages; i++) {
        if (i != minDistanceID2 && distanceMatrix[i][0]
            != Double.MAX_VALUE) {
          distance = ((GeotaggedImageGroup)imageIDMap
            .get(minDistanceID2)).calculateMinDistanceTo(
              (GeotaggedImageGroup)imageIDMap.get(i));
          distanceMatrix[i][minDistanceID2] = distance;
          distanceMatrix[minDistanceID2][i] = distance;
        }
      }

      // Loop through all groups to store in cached tree
      clusterList = new ArrayList<GeotaggedImageGroup>();
      for (GeotaggedImageGroup group : imageIDMap.values()) {
        groupToCache = new GeotaggedImageGroup();
        for (GeotaggedImageID imageID : group.getImageIDs()) {
          groupToCache.add(imageID);
        }
        clusterList.add(groupToCache);
      }
      clusterTree.add(clusterList);
      levelMinimums.add(minDistance);
    }
  } while (minDistanceID1 >= 0 && minDistanceID2 >= 0);
}
```

```
  private Collection<GeotaggedImageGroup> findGroupsCached(
      double threshold) {
    int level;
    for (level = 0; level < levelMinimums.size(); level++) {
      if (threshold < levelMinimums.get(level)) {
        level++;
        break;
      }
    }

    return clusterTree.get(level);
  }
}
```

MapPinOverlay.java

```
package no.ntnu.idi.android.imagebrowser;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Paint;
import com.google.android.maps.Overlay;
import com.google.android.maps.Point;

public class MapPinOverlay extends Overlay {

  private final android.graphics.Point PIN_HOTSPOT =
    new android.graphics.Point(5,29);

  private Bitmap mapPin = null;
  private Paint paint = null;
  private Point location = null;
  private int x = 0;
  private int y = 0;
  private android.graphics.Point screenCoord;

  public MapPinOverlay(Context context, double latitude, double longitude){
    mapPin = BitmapFactory.decodeResource(context.getResources(),
      R.drawable.mappin_red);
    paint = new Paint();
    location = new Point((int)(latitude * 1000000),
      (int)(longitude * 1000000));
  }

  public MapPinOverlay(Context context, Point location) {
    mapPin = BitmapFactory.decodeResource(context.getResources(),
      R.drawable.mappin_red);
    paint = new Paint();
    this.location = location;
  }

  @Override
  public void draw(Canvas canvas, PixelCalculator calculator,
      boolean shadow) {
    super.draw(canvas, calculator, shadow);
    int[] pixelCoord = new int[2];
    calculator.getPointXY(location, pixelCoord);
    screenCoord = new android.graphics.Point(pixelCoord[0] - PIN_HOTSPOT.x,
      pixelCoord[1] - PIN_HOTSPOT.y);
    x = pixelCoord[0] - PIN_HOTSPOT.x;
    y = pixelCoord[1] - PIN_HOTSPOT.y;
    canvas.drawBitmap(mapPin, x, y, paint);
  }

  public boolean coversPixelCoordinate(int x, int y) {
    if (x > screenCoord.x && x < screenCoord.x + mapPin.getWidth()
        && y > screenCoord.y && y < screenCoord.y + mapPin.getHeight()) {
      return true;
    }
    return false;
  }

}
```

Appendix E: Source Code of ImageProvider

BitmapHelper.java

```
package no.ntnu.idi.android.imageprovider;

import android.graphics.Bitmap;
import android.graphics.Matrix;

public final class BitmapHelper {

  /*
   * Resize a bitmap to the specified scale.
   * Values < 1 makes the bitmap smaller, values > 1 makes it bigger.
   */
  public static Bitmap resizeBitmap(Bitmap bitmapOrg, float scale) {
    int width = bitmapOrg.width();
        int height = bitmapOrg.height();

        Matrix matrix = new Matrix();
        matrix.postScale(scale, scale);

        Bitmap resizedBitmap = Bitmap.createBitmap(bitmapOrg, 0, 0,
          width, height, matrix, true);
        return resizedBitmap;
  }

  /*
   * Resizes a bitmap to the specified width.
   * The aspect ratio is not changed, so the new height will
   * be calculated based on the provided width.
   */
  public static Bitmap resizeBitmap(Bitmap bitmapOrg, int newWidth) {
    int width = bitmapOrg.width();
    int height = bitmapOrg.height();

    float scale = ((float) newWidth) / width;

    Matrix matrix = new Matrix();
    matrix.postScale(scale, scale);

    Bitmap resizedBitmap = Bitmap.createBitmap(bitmapOrg, 0, 0, width,
      height, matrix, true);
    return resizedBitmap;
  }
}
```

DatabaseHelper.java

```
package no.ntnu.idi.android.imageprovider;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {

  private static final String DATABASE_NAME = "images";
  private static final int DATABASE_VERSION = 6;

  @Override
  public void onCreate(SQLiteDatabase db) {

    db.execSQL("CREATE TABLE images ("
        + "_id INTEGER PRIMARY KEY AUTOINCREMENT,"
        + "filename TEXT UNIQUE,"
        + "image_original TEXT,"
        + "image_screensize TEXT,"
        + "image_thumbnail TEXT,"
        + "latitude REAL,"
        + "longitude REAL);");

    db.execSQL("CREATE TABLE flickr_images ("
        + "_id INTEGER PRIMARY KEY AUTOINCREMENT,"
        + "filename TEXT UNIQUE,"
        + "image_original TEXT,"
        + "image_screensize TEXT,"
        + "image_thumbnail TEXT,"
        + "latitude REAL,"
        + "longitude REAL);");

  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
    db.execSQL("DROP TABLE IF EXISTS images");
    db.execSQL("DROP TABLE IF EXISTS flickr_images");
    onCreate(db);
  }

  public SQLiteDatabase openDatabase(Context context) {
    return openDatabase(context, DATABASE_NAME, null, DATABASE_VERSION);
  }

}
```

FlickrPhotoHandler.java

```
package no.ntnu.idi.android.imageprovider;

import java.util.ArrayList;
import java.util.List;

import org.xml.sax.Attributes;
import org.xml.sax.Locator;

import android.location.Location;

public class FlickrPhotoHandler implements org.xml.sax.ContentHandler {

  private List<String> filenames = new ArrayList<String>();
  private List<String> urls = new ArrayList<String>();
  private List<Location> locations = new ArrayList<Location>();

  public void startElement(String uri, String localName,
      String qName, Attributes atts) {
    if (qName == "photo") {
      String id = atts.getValue("id");
      String secret = atts.getValue("secret");
      String server = atts.getValue("server");
      String farm = atts.getValue("farm");
      String isPublic = atts.getValue("ispublic");
      String latitude = atts.getValue("latitude");
      String longitude = atts.getValue("longitude");

      if (isPublic.equals("1")) {
        if (!latitude.equals("0") && !longitude.equals("0")) {
          String baseUrl = "http://farm" + farm + ".static.flickr.com/"
            + server + "/";
          String filename = id + "_" + secret + ".jpg";
          filenames.add(filename);
          urls.add(baseUrl + filename);
          Location location = new Location();
          location.setLatitude(Double.parseDouble(latitude));
          location.setLongitude(Double.parseDouble(longitude));
          locations.add(location);
        }
      }
    }
  }

  public List<String> getFilenames() {
    return filenames;
  }

  public List<String> getUrls() {
    return urls;
  }

  public List<Location> getLocations() {
    return locations;
  }

  // Required to implement interface
  public void   characters(char[] ch, int start, int length){}
  public void   endDocument(){}
  public void   endElement(String uri, String localName, String qName){}
```

```
  public void   endPrefixMapping(String prefix){}
  public void   ignorableWhitespace(char[] ch, int start, int length){}
  public void   processingInstruction(String target, String data){}
  public void   setDocumentLocator(Locator locator){}
  public void   skippedEntity(String name){}
  public void   startDocument(){}
  public void   startPrefixMapping(String prefix, String uri){}

}
```

ImageProvider.java

```
package no.ntnu.idi.android.imageprovider;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.content.Context;
import android.content.Resources;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;

public class ImageProvider extends ContentProvider {

  private static final String URI_AUTHORITY =
    "no.ntnu.idi.android.imageprovider";
  private static final String URI_PATH = "images";
  private static final String FLICKR_URI_PATH = "flickr";
  private static final int URI_IMAGES = 1;
  private static final int URI_IMAGE_ID = 2;
  private static final int FLICKR_URI_IMAGES = 3;
  private static final int FLICKR_URI_IMAGE_ID = 4;

  public static final Uri CONTENT_URI = Uri.parse("content://"
    + URI_AUTHORITY + "/" + URI_PATH);
  public static final Uri FLICKR_CONTENT_URI = Uri.parse("content://"
    + URI_AUTHORITY + "/" + FLICKR_URI_PATH);
  public static final String IMAGE_ORIGINAL = "image_original";
  public static final String IMAGE_SCREENSIZE = "image_screensize";
  public static final String IMAGE_THUMBNAIL = "image_thumbnail";
  public static final String LATITUDE = "latitude";
  public static final String LONGITUDE = "longitude";

  private UriMatcher uriMatcher;
  private SQLiteDatabase db;

  /*
   * @returns true if the provider was successfully loaded, false otherwise
   */
  @Override
  public boolean onCreate() {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(URI_AUTHORITY, URI_PATH, URI_IMAGES);
    uriMatcher.addURI(URI_AUTHORITY, URI_PATH + "/#", URI_IMAGE_ID);
    uriMatcher.addURI(URI_AUTHORITY, FLICKR_URI_PATH, FLICKR_URI_IMAGES);
    uriMatcher.addURI(URI_AUTHORITY, FLICKR_URI_PATH + "/#",
      FLICKR_URI_IMAGE_ID);

    DatabaseHelper dbHelper = new DatabaseHelper();
    db = dbHelper.openDatabase(getContext());
```

```
    return (db == null) ? false : true;
  }

  @Override
  public int delete(Uri uri, String selection, String[] selectionArgs) {
    return 0;
  }

  @Override
  public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
    case URI_IMAGES:
      return "vnd.android.cursor.dir/vnd.ntnu.image";
    case URI_IMAGE_ID:
      return "vnd.android.cursor.item/vnd.ntnu.image";
    default:
      throw new IllegalArgumentException("Unknown URI: " + uri);
    }
  }

  @Override
  public Uri insert(Uri uri, ContentValues values) {
    return null;
  }

  @Override
  public Cursor query(Uri uri, String[] projection, String selection,
      String[] selectionArgs, String sortOrder) {

    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

    switch (uriMatcher.match(uri)) {
    case URI_IMAGES:
      queryBuilder.setTables((String)getContext()
        .getResources().getText(R.string.db_table));
      break;
    case URI_IMAGE_ID:
      queryBuilder.setTables((String)getContext()
        .getResources().getText(R.string.db_table));
      queryBuilder.appendWhere("_id=" + uri.getPathSegments().get(1));
      break;
    case FLICKR_URI_IMAGES:
      queryBuilder.setTables((String)getContext()
        .getResources().getText(R.string.flickr_db_table));
      break;
    case FLICKR_URI_IMAGE_ID:
      queryBuilder.setTables((String)getContext()
        .getResources().getText(R.string.flickr_db_table));
      queryBuilder.appendWhere("_id=" + uri.getPathSegments().get(1));
      break;
    default:
      throw new IllegalArgumentException("Unknown URI: " + uri);
    }

    Cursor cursor = queryBuilder.query(db, projection, selection,
      selectionArgs, null, null, sortOrder);
    return cursor;
  }

  @Override
  public int update(Uri uri, ContentValues values, String selection,
```

```
      String[] selectionArgs) {
    return 0;
  }
}
```

ProviderController.java

```
package no.ntnu.idi.android.imageprovider;

import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;

import org.cmc.sanselan.ImageReadException;
import org.cmc.sanselan.ImageWriteException;
import org.cmc.sanselan.Sanselan;
import org.cmc.sanselan.formats.jpeg.JpegImageMetadata;
import org.cmc.sanselan.formats.jpeg.exifRewrite.ExifRewriter;
import org.cmc.sanselan.formats.tiff.TiffImageMetadata;
import org.cmc.sanselan.formats.tiff.write.TiffOutputSet;
import org.xml.sax.SAXException;

import android.app.Activity;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.location.Location;
import android.os.Bundle;
import android.util.Xml;
import android.view.Menu;
import android.view.Menu.Item;
import android.widget.TextView;
import android.widget.Toast;

public class ProviderController extends Activity {

  private static final int MENU_RESCAN = 1;
  private static final int MENU_FLICKR = 2;
  private static final int THUMBNAIL_WIDTH = 60;
  private static final int SCREENSIZE_WIDTH = 320;

  private SQLiteDatabase db;
  TextView counterTextView;

  @Override
  protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    DatabaseHelper dbHelper = new DatabaseHelper();
    db = dbHelper.openDatabase(this);

    counterTextView = new TextView(this);
```

```
    setContentView(counterTextView);

    updateCounter();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(0, MENU_RESCAN, "Rescan image folder");
    menu.add(0, MENU_FLICKR, "Rescan flickr");

    return super.onCreateOptionsMenu(menu);
  }

  @Override
  public boolean onOptionsItemSelected(Item item) {
    if (item.getId() == MENU_RESCAN) {
      loadImagesFromFile();
      updateCounter();
      Toast.makeText(this, "Done!", Toast.LENGTH_SHORT).show();
    } else if (item.getId() == MENU_FLICKR) {
      loadImagesFromFlickr();
      Toast.makeText(this, "Done!", Toast.LENGTH_SHORT).show();
    }

    return super.onOptionsItemSelected(item);
  }

  private void updateCounter() {
    Cursor cursor = db.query("images", new String[]{"_id"}, null,
      null, null, null, null);
    counterTextView.setText("Number of images in DB: " + cursor.count());
  }

  private void loadImagesFromFile() {
    // Get a list of all files in the subdirectory called images
    File imagesDirectory = this.getDir((String)getResources()
      .getText(R.string.images_directory), Context.MODE_PRIVATE);
    File thumbnailsDirectory = this.getDir((String)getResources()
      .getText(R.string.thumbnails_directory), Context.MODE_PRIVATE);
    File[] dirFiles = imagesDirectory.listFiles();

    // Get a list of all images in db
    Cursor cursor =
      db.query((String)getResources().getText(R.string.db_table),
        new String[]{"filename"}, null, null, null, null, null);
    int numRows = cursor.count();
    String[] dbFilenames = new String[numRows];
    int filenameColumn = cursor.getColumnIndex("filename");
    if (cursor.first()) {
      int i = 0;
      do {
        dbFilenames[i++] = cursor.getString(filenameColumn);
      } while (cursor.next());
    }

    boolean found;
    // For each file in the subdirectory called images
    for (File file : dirFiles) {
      found = false;
      for (int i = 0; i < numRows; i++) {
        //if (filename.equals(dbFilenames[i])) {
```

```
  if (file.getName().equals(dbFilenames[i])) {
    found = true;
    break;
  }
}


// if file doesn't exist in db
if (!found) {
  // Make thumbnail and store in a subdirectory called thumbnails
  try {
    String filename = file.getName();
    String thumbnailFilename = filename.substring(0,
      filename.lastIndexOf('.')) + "_thumb.jpg";
    String screensizeFilename = filename.substring(0,
      filename.lastIndexOf('.')) + "_screen.jpg";

    InputStream inputStream = new FileInputStream(file);
    Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
    inputStream.close();
    inputStream = new FileInputStream(file);
    Double latitude = null;
    Double longitude = null;
    try {
      JpegImageMetadata metadata = (JpegImageMetadata)Sanselan
        .getMetadata(inputStream, filename);
      if (metadata != null) {
        TiffImageMetadata exif = metadata.getExif();
        if (exif != null) {
          TiffImageMetadata.GPSInfo gpsInfo = exif.getGPS();
          if (gpsInfo != null) {
            latitude = gpsInfo.getLatitudeAsDegreesNorth();
            longitude = gpsInfo.getLongitudeAsDegreesEast();
          }
        }
      }
    } catch (ImageReadException ire) {
    } catch (IOException ioe) {
    }
    inputStream.close();

    Bitmap thumbnail = BitmapHelper.resizeBitmap(bitmap,
      THUMBNAIL_WIDTH);
    File thumbnailFile = new File(thumbnailsDirectory,
      thumbnailFilename);
    OutputStream outputStream = new FileOutputStream(thumbnailFile);
    thumbnail.compress(Bitmap.CompressFormat.JPEG, 100,
      outputStream);
    outputStream.close();
    thumbnail = null;

    Bitmap screensize = BitmapHelper.resizeBitmap(bitmap,
      SCREENSIZE_WIDTH);
    File screensizeFile = new File(thumbnailsDirectory,
      screensizeFilename);
    outputStream = new FileOutputStream(screensizeFile);
    screensize.compress(Bitmap.CompressFormat.JPEG, 100,
      outputStream);
    outputStream.close();
    screensize = null;
```

```
            bitmap = null;

            // Add new row to db
            db.execSQL("INSERT INTO "
              + (String)getResources().getText(R.string.db_table)
              + "(filename, "
              + ImageProvider.IMAGE_ORIGINAL + ", "
              + ImageProvider.IMAGE_SCREENSIZE + ", "
              + ImageProvider.IMAGE_THUMBNAIL + ", "
              + ImageProvider.LATITUDE + ", "
              + ImageProvider.LONGITUDE + ") VALUES('"
              + filename + "', '" + imagesDirectory.getAbsolutePath() + "/"
              + filename + "', '"
              + thumbnailsDirectory.getAbsolutePath() + "/"
              + screensizeFilename + "', '"
              + thumbnailsDirectory.getAbsolutePath() + "/"
              + thumbnailFilename + "', "
              + ((latitude == null) ? "null" : "'" + latitude + "'") + ", "
              + ((longitude == null) ? "null" : "'" + longitude + "'")
              + ");");

          } catch (FileNotFoundException e) {
            continue;
          } catch (IOException ioe) {
            continue;
          }
        }
      }
    }

    private void loadImagesFromFlickr() {
      File imagesDirectory = this.getDir((String)getResources()
        .getText(R.string.flickr_images_directory),
          Context.MODE_PRIVATE);
      File thumbnailsDirectory = this.getDir((String)getResources()
        .getText(R.string.flickr_thumbnails_directory),
          Context.MODE_PRIVATE);

      String urlString =
"http://api.flickr.com/services/rest/?method=flickr.interestingness.getList
&per_page=500&extras=geo&date=2008-05-26&api_key="
        + getResources().getText(R.string.flickr_API_key);

      try {
        URL url = new URL(urlString);
        HttpURLConnection connection =
          (HttpURLConnection)url.openConnection();
        if (connection.getDoInput()) {
          InputStream is = connection.getInputStream();
          BufferedReader br = new BufferedReader(new InputStreamReader(is));
          StringBuilder sb = new StringBuilder();
          String line;
          while ((line = br.readLine()) != null) {
            sb.append(line);
          }
          br.close();

          Xml result = new Xml();
          FlickrPhotoHandler handler = new FlickrPhotoHandler();
          result.parse(sb.toString(), handler);
```

```
List<String> filenames = handler.getFilenames();
List<String> urls = handler.getUrls();
List<Location> locations = handler.getLocations();

if (urls.size() != locations.size())
  return;

for (int i = 0; i < urls.size(); i++) {
  String filename = filenames.get(i);
  String imageUrl = urls.get(i);
  Location location = locations.get(i);

  String thumbnailFilename = filename.substring(0,
    filename.lastIndexOf('.')) + "_thumb.jpg";
  String screensizeFilename = filename.substring(0,
    filename.lastIndexOf('.')) + "_screen.jpg";

  url = new URL(imageUrl);
  connection = (HttpURLConnection)url.openConnection();
  if (connection.getDoInput()) {
    is = connection.getInputStream();
    TiffOutputSet outputSet = new TiffOutputSet();
    outputSet.setGPSInDegrees(location.getLongitude(),
      location.getLatitude());
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    new ExifRewriter().updateExifMetadataLossless(is, baos,
      outputSet);
    is.close();

    File imageFile = new File(imagesDirectory, filename);
    OutputStream outputStream = new FileOutputStream(imageFile);
    byte[] imageBytes = baos.toByteArray();
    baos = null;
    outputStream.write(imageBytes);
    outputStream.close();


    Bitmap bitmap = BitmapFactory.decodeByteArray(imageBytes, 0,
      imageBytes.length);
    imageBytes = null;

    Bitmap thumbnail = BitmapHelper.resizeBitmap(bitmap,
      THUMBNAIL_WIDTH);
    File thumbnailFile = new File(thumbnailsDirectory,
      thumbnailFilename);
    outputStream = new FileOutputStream(thumbnailFile);
    thumbnail.compress(Bitmap.CompressFormat.JPEG, 100,
      outputStream);
    outputStream.close();
    thumbnail = null;

    Bitmap screensize = BitmapHelper.resizeBitmap(bitmap,
      SCREENSIZE_WIDTH);
    File screensizeFile = new File(thumbnailsDirectory,
      screensizeFilename);
    outputStream = new FileOutputStream(screensizeFile);
    screensize.compress(Bitmap.CompressFormat.JPEG, 100,
      outputStream);
    outputStream.close();
    screensize = null;
```

```
            bitmap = null;

            db.execSQL("INSERT INTO "
              + (String)getResources().getText(R.string.flickr_db_table)
              + "(filename, "
              + ImageProvider.IMAGE_ORIGINAL + ", "
              + ImageProvider.IMAGE_SCREENSIZE + ", "
              + ImageProvider.IMAGE_THUMBNAIL + ", "
              + ImageProvider.LATITUDE + ", "
              + ImageProvider.LONGITUDE + ") VALUES('"
              + filename + "', '" + imagesDirectory.getAbsolutePath()
              + "/" + filename + "', '"
              + thumbnailsDirectory.getAbsolutePath() + "/"
              + screensizeFilename + "', '"
              + thumbnailsDirectory.getAbsolutePath() + "/"
              + thumbnailFilename + "', "
              + ((location == null) ? "null" : "'" + location.getLatitude()
              + "'") + ", "
              + ((location == null) ? "null" : "'"
              + location.getLongitude() + "'") + ");");
          }
        }

      }
    } catch (MalformedURLException mue) {
    } catch (IOException ioe) {
    } catch (SAXException se) {
    } catch (ImageReadException ire) {
    } catch (ImageWriteException iwe) {
    }
  }
}
```