Simulated Presence for Location-Aware Studies

Masterarbeit

Krischan Udelhoven
Steubenstr. 64 – 45138 Essen
krischan.udelhoven@gmail.com

Bonn, den 4. Oktober 2012

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Professor Dr. Armin B. Cremers

universitätbonn

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind.

Bonn, den 4. Oktober 2012

Abstract

Location-aware applications are much more difficult to evaluate than conventional non-pervasive applications. Usually, it is not possible to restrict the evaluation environment to make the experiments predictable and controllable, but still realistic enough to provide significant results. In addition, the evaluation can be very exhausting and time consuming.

For this thesis, a special laboratory setup for the evaluation of location-aware applications was developed, which maintains the scalability of a field evaluation while keeping the options and the convenience of a laboratory. The system simulates the geographic location of study participants by tracking the movement of small toy figures in a model landscape. The system is a tangible user interface, called Presence Simulator. The simulated presence is transfered to android smartphones, on which the geographic location, stored in the device's operating system, is mocked accordingly. As a result, the evaluated applications are working with the mocked location data.

To determine the effect of the test environment, the system was evaluated in a comparative study. In the study, a location-aware task reminder was compared to a conventional to-do-list. Half of the participants attended to field experiments. The other half used the Presence Simulator. Both groups where split again into a group that used the location-aware task reminder and a group that used the todo-list to complete the same tasks. There was no significant difference in the number of forgotten tasks between the groups in the field study and the corresponding groups in the Presence Simulator based study.

Überblick

Anwendungen, die den Standort in ihr Verhalten einbeziehen sind sehr viel aufwändiger zu evaluieren als herkömmliche nicht kontextsensitive Anwendungen. Üblicherweise ist es nicht möglich die Umgebung, in der eine Anwendung evaluiert werden soll, so zu kontrollieren, dass die einzelnen Experimente unter gleichen und dennoch realistischen Bedingungen wiederholt durchgeführt weden können. Außerdem kann die Evaluation sehr mühsam und zeitaufwendig sein.

Für diese Arbeit wurde ein System zur Evaluation ortsbezogener Anwendungen im Labor entwickelt, das die Skalierbarkeit von im Freien durchgeführten Evaluationen mit den Vorteilen einer Laborumgebung vereint. Das System simuliert die geographische Position von Studienteilnehmern, indem es die Bewegung von kleinen Spielzeugfiguren in einer Modellandschaft verfolgt. Damit ist das Presence Simulator genannte System eine Art Tangible User Interface (dt. anfassbare Benutzerschnittstelle). Die simulierte Präsenz wird vom System auf Smartphones übertragen, auf welchen dann die im Betriebsystem gespeicherte geographische Position entsprechend überschrieben wird. Daher greifen auch Anwendung, die evaluiert werden, auf die simulierten Daten zu.

Für die Arbeit wurde der Presence Simulator in einer vergleichenden Studie evaluiert. In der Studie wurde ein ortsbasiertes Werkzeug zur Aufgabenerinnerung mit herkömmlichen Aufgabenlisten verglichen. Die Hälfte der Teilnehmer nahm an Feldexperimenten teil. Die andere Hälfte benutzte den Presence Simulator. Beide Gruppen wurden weiter aufgeteilt in eine Gruppe, die die ortsbasierte Aufgabenerinnerung benutzte und eine, die eine Aufgabenliste mit den gleichen Aufgaben erhielt. Es gab keinen signifikanten Unterschied in der Anzahl der vergessenen Aufgaben zwischen den Gruppen der Feldstudie und den entsprechenden Gruppen, die den Presence Simulator benutzten.

Contents

Figures And Tables

List of Figures

List of Tables

Acknowledgements

I would like to thank the following people and express my appreciation and gratitude for helping me complete this thesis.
Dr. Pascal Bihler, my advisor, for the numerous inspiring talks we had during the last seven months and the continuous support and assistance he gave to me. Without him, I would not have been able to finish this thesis.
Nora Dickel, for her personal support, great patience at all times and especially for proof-reading and correcting the thesis.
Orhan Sönmez, for helping me conduct the field study.
Last but not least I would like to thank all participants of the study, my fellow students, friends and family.

Chapter 1

Introduction

1.1 Motivation

Location-aware applications adapt passively or actively to the geographic location.

Location awareness is a specific form of context awareness where the geographic location of a device influences an applications function. An application that is actively location aware adapts its behaviour depending on the current location (e. g. an application that will forward your calls when you are out of reach of your office phone). For a passively location aware application, the location is relevant, but not critical (e. g. visual effects change, depending on the location). On the hardware level, there are several advanced technologies like the Global Positioning System (GPS) or network based localization to detect the location of a device. Location awareness is a wide-spread technology and nowadays supported by nearly all smart phones and other handheld devices.

In the past, location-aware applications were not as successful as they could have been.

Since the early days of third generation mobile and wireless systems (3G), location-aware applications have been a drive of the mobile operators' marketing efforts. Location awareness has been presented as "a crucial tool for providing the right service, at the right time, in the right locationText" [UMT00] with a lot of opportunities for obtaining substantial revenues. In 2003, *Cambridge Positioning Systems* predicted a world wide sales potential of 12 billion United States dollars for the segment in 2005 [Cam03]. As we know today, the actual sales in 2005 aggregated far below that estimate.

Nowadays, location-awareness is an established technology.

Nowadays, the situation has improved. With the release of the iPhone in 2007 by Apple, and the resulting smartphone revolution that continues to this day, now roughly one third of the German uses smart phones [Ips12]. They carry a smartphones wherever they go and use location aware services like *Qype*[1] or Google's *local*[2] quite naturally. But location-aware applications that solve more complex problems than finding the nearest restaurant or locating our friends are still rare:

[1] www.qype.com
[2] www.google.com/+/learnmore/local/

Most applications featured in overviews like "21 Awesome GPS and Location-Aware Apps for Android" by *PCWorld* [Cas12] from 2012 can be roughly categorized into applications for sight seeing and for navigational purposes.

There is a lack of innovative location-aware applications.

There is a fundamental lack of innovative location-aware applications. The reason for this is not a lack of new and creative ideas. Already in 1999, Ben Russell described the opportunities of location awareness in a wonderful techno-romantic vision. According to his headmap manifesto, smartphones are capable of making ideas like the following a reality:

- "every place has emotional attachments you can open and save [...]
- people within a mile of each other who have never met stop what they are doing and organise spontaneously to help with some task or other. [...]
- inanimate objects can become more animate (if you know where a tree is and you know when someone is walking past it you could make it burst into song)" [Rus99]

It seems that with most location aware applications something must have gone wrong on the path between idea and realization.

Privacy concerns inhibit the development of location-aware applications, as well as missing evaluation techniques

Often mentioned inhibitors for location-aware applications are unsafe technology and privacy concerns in general. [Ste04] In my thesis, I am going to address another inhibitor: The usability of location-aware applications is much harder to evaluate than the usability of conventional non-pervasive applications. In human-computer interaction (HCI), the evaluation of non-pervasive applications is an established discipline with many elaborated techniques and methods. Evaluation of location-aware applications on the other hand is still a rather untapped research field.

1.2 Problem Statement

Location-aware applications are usually evaluated in expensive field studies, which are hard to control.

What makes evaluating location-aware services so difficult? Traditional evaluation methods and metrics need a controllable test environment, and thus can only be applied with an extensive effort. Like other pervasive applications, location-aware applications are tightly attached to their environment because they react to context changes. Often, it is not possible to restrict the evaluation environment in such a way, that tests are predictable and controllable, but still realistic enough to provide meaningful results. It might be possible to recreate most indoor environments in laboratories where the variables can be controlled while observing the participants, but when the target environment of the application is placed outdoors – for example in a city – the costs and efforts will be immense. Therefore, most researchers state that expensive field evaluations cannot be avoided (e. g. [GBG04], [KSAH04] and [RCT+07]).

Field studies are exhausting and time consuming.

In addition to the suboptimal evaluation environment, field studies are very exhausting and time consuming. The participants need to be enthusiastic and tolerant – especially when it is cold outside or raining. "The scaling dimensions that characterize [ubiquitous computing] systems - device, space, people, or time - make it impossible to use traditional, contained usability laboratories." [AM00] Therefore, it is important to design the usability studies as efficient and effective as possible. For example, in order to compensate the missing tools of a fully equipped usability laboratory, several techniques like experimenter observation and pedometer usage were proposed [GBG04]. These techniques facilitate the work a lot but still the major problems are not addressed: Location aware user studies continue to be exhausting and time consuming. Because the usage of GPS consumes a lot of energy, one have to schedule intermediate pauses to recharge the batteries of the smart phones. In a laboratory setup the devices can be recharged during the experiments.

The system developed in the thesis is an attempt to solve these problems.

In my thesis I took a different approach. I designed and implemented a special laboratory setup, which maintains the scalability of a field evaluation while keeping the options of a laboratory. In order to really facilitate the researchers work this system should be as reliable and user-friendly as possible, while still providing the same results as other methods.

1.3 Contribution of this Work

The Presence Simulator simulates the presence of study participants by tracking small human like figures (LSFs)

For this thesis, a system that facilitates the performance of location-aware user studies and the evaluation of location-aware applications running on android smart phones was implemented. In order to benefit from the options of a laboratory, the system is placed in an indoor environment. The system simulates the geographic location (presence) of participants of user studies by using small human like figures. I named the system Presence Simulator (PS) and the figures Location Source Figures (LSF).

The system is a tangible user interface.

During a study an LSF is assigned to each participant. The LSFs are then placed in a model landscape depicting the environment of a corresponding field study. The position of the LSFs is tracked by the system and converted to the coordinates of the geographic location corresponding to the location on the model landscape. The participants are encouraged to change the position of their LSFs, by directly manipulating, i. e. moving them. Hence the system models a tangible user interface.

The geographic location of smartphones is mocked by the system.

For each LSF, the system generates a stream of geographic location coordinates. This stream is transmitted via a wireless connection to the smart phone of the participant the LSF was assigned to. On the participants smartphone a receiver application is continuously mocking the device's geographic location according to the streamed location coordi-

Figure 1.1: The hardware setup of the Presence Simulator - A system developed in this thesis to simulate and track the user's presence in a model landscape.

nates. This happens transparently to the application that is evaluated. Location coordinates simulated by the Presence Simulator are similar to the ones from the device's GPS receiver and thus the whole Presence Simulator is a kind of a GPS receiver simulator.

The hardware setup consist of a model landscape, a web cam mounted above the landscape, a computer with speakers, smart phones and a W-LAN router.

FThe hardware construction of the Presence Simulator built in this work is shown in Figure 1.1. The model landscape is placed on a table. In this case the landscape is just a map, but it is also possible to build a three dimensional model. Of course, that would be much more time consuming but it would also make the setting more realistic. On top of the map a playmobil® figure, serving as a LSF, is placed. The figure is 7.5 centimeters high and originally a children's toy. 1.5 meters above the center of the table, a web cam is attached to a wooden construction. The lens of the camera gives a top-down perspective of the map. The battens of the construction are tightly fixed to the table with cable ties. The web cam is connected to the computer running the Presence Simulator server software. In the background of the picture a speaker is visible, which is used to simulate traffic and other ambient noises. The wireless local area network (LAN) router, which is used to

transmit the streams of location coordinates from the computer to the smart phones lying on the table, is placed on the speaker.

The Presence Simulator was compared to a field evaluation in a comparative study.

In order to compare the system's performance with traditional field evaluation methods, and thus determine the effect of the test environment, a comparative evaluation was made. For this purpose, a location-aware android application named *ShinyNavi* was developed for evaluation. The application is a navigation system for pedestrians. Orhan Sönmez, a student from the computer science department of the university of Bonn, helped me in carrying out parts of the study. Sönmez is currently working on his bachelor's thesis about the effectiveness of location-aware task reminder tools. Therefore a task reminder was integrated by him into the navigation system and also evaluated.

A between-participants study design was used.

The participants of the study were split into two groups: One group evaluated the application by using the Presence Simulator. The subjects of the other group participated in a field evaluation. To avoid carryover effects, a between-participants study design was used. The results of the two groups are compared in Chapter 4 in order to answer the question whether there exists – for a specific study design – a significant difference between the results of evaluations based on the Presence Simulator and traditional field evaluations.

1.4 Thesis structure

Short description of the five chapters of the thesis and the appendix.

This thesis is devided into in five chapters. At the end, there is an appendix with all performance measurements and also the data that has been collected during the comparative study discussed in Chapter 4.

- *Chapter 1* served as an introduction by describing my motivation for this work, the importance of the field of location-aware software evaluation and the problems current evaluation approaches are facing. Furthermore, the steps I pursued in this work in order to overcome these problems are introduced.
- *Chapter 2* describes currently available evaluation methods for location-aware software applications in general. Particular emphasis is placed on three different evaluation approaches: Traditional field studies which are situated outside, studies where the participants act in a purely virtual environment, and studies taking place in a physical model environment. For each approach several sophisticated tricks and techniques to facilitate the researchers work are introduced. As the Presence Simulator is a tangible user interface the current state in that research field is also described.
- *Chapter 3* discusses the Presence Simulator system that has been developed in this thesis. Software design and the actual and important parts of the actual implementation are described. The chapter starts with an overview of the system's architecture to continue with a discussion of each component in detail. At the

end of the third chapter, performance measurements and the unit tests which were implemented to maintain the software quality of the system are presented.

- *Chapter 4* presents the comparative study that has been executed in this thesis in order to evaluate the Presence Simulator system. First, the location-aware application that has been implemented for the study is described, then the study design is presented and the study results are discussed.
- *Chapter 5* discusses the main conclusions of this thesis by summarising the work completed, by discussing how the research questions have been answered and by detailing future directions to extend this research area.

Chapter 2

Related work

2.1 Evaluation of Location-Aware Applications

In the last years, various methods and techniques to evaluate location-aware applications have been proposed and compared to each other. They can be roughly divided into three categories: traditional field evaluations, approaches that simulate the target environment by creating a living laboratory, and approaches that evaluate location-aware applications in a purely virtual environment (e. g. based on computer game engines). Studies placed in a living laboratory are more abstract to the participants than field studies and pure virtual environments raise the level of abstraction even more. For each category, a research paper will be discussed in the following three subsections.

Location-aware applications can be evaluated in the field, in living laboratories or, in virtual environments.

2.1.1 Field Studies

Location-aware applications are traditionally evaluated and tested in field experiments. Field experiments are quantitative experimental evaluations which are carried out in a field and, depending on their size and design, can grow up into a real field study. In their paper "Using Field Experiments to Evaluate Mobile Guides", Joy Goodman et al. describe the advantages and disadvantages of field experiments as compared to other evaluation methods. They also present a specific method of conducting field studies and discuss several evaluation measurement techniques and tools. [KSAH04]

Evaluation in the field is the traditional approach.

In comparison to evaluations conducted by experts, field experiments involve actual users. This is why, according to Goodman et al., the results of these studies are more reliable. When compared to studies in a laboratory setting, field experiments benefit noticeable from the participants exposure to the real environment. "Aspects such as lightning levels, weather, the effects of walking, the appearance of landmarks in real life and the effectiveness of location-sensing systems can have unpredictable effects on the usability and effectiveness of a device."

Real users are involved and the environment is as realistic as it can be.

[KSAH04]. Goodman et al. conclude that the only way to really find out how a location-aware application will work is to test it in its target environment.

Field studies are exhausting, time consuming and the environment is difficult to control.

According to Goodman et al., field studies are "much harder and more time-consuming than lab experiments" [KSAH04], but they found out that they are not substantially more *difficult*. The biggest challenge seems to be the controlling of confounding variables, because environmental variables like traffic or weather conditions are out of the researchers control. Careful scheduling and waiting for the right conditions before starting an experiment might help, but this would consume a lot of time and resources. Goodman et al. state that avoiding the variation of these variables would lead to unrealistic results and thus a better approach which also avoids the mentioned difficulties is to let the variables vary across conditions.

Goodman's method of running field studies.

To obtain a quantitative evaluation, the location-aware application to be evaluated is compared to another method (e. g. navigation with a navigation system is compared to navigation with a standard paper map). According to Goodman et al., the comparison can be done using between-groups, within-groups or mixed study designs. In the method proposed by Goodman et al., participants of the study are followed by an experimenter while completing a predefined list of tasks. The experimenter observes and documents the participant's behavior. After a participant has completed the list of tasks, he or she is questioned about the experiment using a questionnaire or interview.

The experimenter is responsible for taking several evaluation measurements during an experiment. Goodman et al. propose to use a range of different quantitative and qualitative measurements. Some measurements proposed by the authors are time, errors, perceived workload, comfort, and user comments and preferences.

A clear method can facilitate carrying out field studies.

Goodman et al. conclude that carrying out field evaluations can be simplified by defining a clear method. By using the method presented in the paper work will be facilitated and the study will produce a lot of quantitative and qualitative results.

2.1.2 Living Laboratory Studies

A living laboratory is a one-to-one simulation of a particular environment.

A living laboratory is a life-size simulation of a particular environment. Living laboratories are closely related to the approach I will take in my thesis. However, due to the effort and the large area needed to construct the one-to-one model, living laboratories have only been used for applications designed for indoor use.

Kjeldskov et al. evaluated a living laboratory setup.

In their paper "Is it Worth the Hassle? Exploring the Added Value of Evaluating the Usability of Context-Aware Mobile Systems in the Field" Jesper Kjeldskov et al. compare a field evaluation with a liv-

ing laboratory evaluation [KSAH04]. They also present a special video camera device for data collection in field studies.

As an example, an electronic patient record system called *MobileWard* was evaluated. For the purposes of the laboratory evaluation, three rooms of a hospital department have been rebuild inside a usability laboratory. The field evaluation was carried out in the corresponding hospital. In the laboratory, data was collected using ceiling-mounted, high quality audio and video recorders, while in the field a portable configuration was used.

Kjeldskov et al. evaluated an electronic patient record system that is used in hospitals.

In a comparative study, more usability problems were detected in the laboratory than in the field environment. In both environments context-aware problems were revealed. Carrying out the field evaluation took twice as long as the laboratory study (34 man-hours vs. 65 man-hours). Kjeldskov et al. concluded that a field evaluation has no advantages compared to a realistic laboratory study and "the lack of control undermined the extendibility of the field evaluation" [KSAH04]. However they do not claim their results to be general, because applications in other domains may have different characteristics.

The comparative study indicated that living laboratory based studies can produce similar results as field studies but can be carried out much faster.

2.1.3 Virtual Environment Studies

In their paper "Rapid User-Centered Evaluation for Context-Aware Systems" [OLMD07], Eleanor O'Neill et al. describe a system that uses the 3D graphics engine of the game Half Life 2[1] to build a purely virtual platform for the evaluation of location-aware applications.

O'Neill et al. built a virtual system for the evaluation of applications.

The platform developed by O'Neill et al. addresses the problems of field studies and tries to overcome them. By having a totally controllable virtual environment, all independent variables can be set freely. Thus, it is possible to produce exactly the same environment parameters for each experiment. Additionally, all the participant's movements and reactions can be recorded, which is obviously hard to achieve in a field study. Also the problem that field studies are exhausting for participants and researchers is solved, because all persons involved can make themselfs comfortable in front of the computers.

The virtual environment is completely controllable and all actions can be recorded.

One might argue that the high level of abstraction inherited in the system distorts the study's results but accordant to the authors the "simulated environment is sufficiently realistic to accurately convey changing physical and social context to the user through the virtual representation of the environment." [OLMD07]

According to O'Neill et al., the high level of abstraction is not a problem.

The user front-end of the platform consists of an interactive context simulator, that allows the user to navigate an avatar in a virtual environment via mouse and keyboard. The environment is defined with the help of an extended version of the Half Life 2 modeling tool. Within the

Users navigate an avatar through a 3D world, that is rendered by the Half Life 2 engine.

[1] source.valvesoftware.com/

extended modeling tool, it is possible to place different types of sensors in the virtual world. The sensors are activated at run-time by user activity and movement. On sensor activation, the system generates a message with relevant context data, which is then passed to a Java proxy gateway. The proxy mediates between the virtual environment and the application that is evaluated.

Virtual worlds of almost any size are possible and development is less time consuming.

The engine allows a flexible usage of sensors, because there is no restriction in expense and logistics like in real environments. According to the authors, with a bit of practice it is possible to rebuild 3D worlds of almost any size in a short time. The platform allows multiple services to be connected to the same virtual world. According to O'Neill et al., the platform leads to an improvement in productivity through "shorter test development life-cycles, more targeted and relevant user evaluation, a low-cost infrastructure and the facility for on-line user testing" [OLMD07].

2.2 Tangible User Interfaces

TUIs are an modern alternative to WIMP interfaces. Characteristics of TUIs are listed here.

For a long time human-computer interaction (HCI) was dominated by the usage of mouse and keyboard to interact with windows, icons, menus and pointers (WIMP). Since the early 1990s, interfaces were designed that differ from traditional WIMP interfaces. One type of these interface forms are tangible user interfaces (TUI). With TUIs, humans interact with computers by manipulating physical objects. According to the TUI's pioneers, Hiroshi Ishii and Brygg Ullmer, TUIs can be characterized as follows:

- "Physical representations (rep-p) are computationally coupled to underlying digital information (model)." [UI00] For example, in the Presence Simulator system the physical location source figures are coupled with the digital location source model.
- "Physical representations embody mechanisms for interactive control" [UI00]. For example, by moving the LSFs, the location source model is controlled.
- "Physical representations are perceptually coupled to actively mediated digital representations (rep-d)." [UI00] In the Presence Simulator's server the LSFs are visualized as markers on a map. Depending on the location-aware application that is evaluated, the LSFs might also be visualized on the participants smart phones.
- "The physical state of interface artifacts partially embodies the digital state of the system." [UI00] In the Presence Simulator system, there is only one type of physical artifacts and the system interprets the spatial location of each of them separately. The relation between the LSFs does not matter. This characteristic becomes important when having more complex TUIs with different physical artifacts.

TUIs have several advantages over WIMP interfaces.

Orit Shaer and Eva Hornecker present a list of strengths of TUIs in their book "Tangible User Interfaces: Past, Present, and Future Directions" [SH09]. One of the strengths is that TUIs support face-to-face collaboration by providing multiple access points to the objects and having the manipulation of objects observable and repeatable by others. Another strength is that humans are used to change their environment by manipulating physical objects. "Children learn abstract concepts through bodily engagement with tangible manipulatives." [SH09] By using a TUI, the user's thinking remains in the physical world and does not have to move to an abstract digital level, which is especially important for the Presence Simulator very important since the particularly physical action of walking though an outdoor environment is simulated.

TUIs have a scaling problem and can be tiring.

Of course, TUIs also have limitations and Shaer and Hornecker list some of them, too. For example, most TUIs cannot be scaled up to solve complex problems with a lot of different objects, data and parameters. At some point the TUI will simply be too big to be used. Another limitation is the fact that TUIs are physical interfaces which require the user to perform physical actions and thus the user will fatigue. The point of depletion can be deferred by designing the objects with human ergonomics in mind, but still, depletion will probably occur earlier than in a WIMP interface. [SH09]

Tracking systems track the movement of physical objects.

One specific type of TUIs are tracking systems. Tracking systems control virtual processes like a traditional computer mouse or the cursor keys of keyboards do. Unlike them, tracking systems are no general input devices, but designed for a specific purpose. For example there are systems for doctors to simulate difficult surgeries, or for engineers to simulate the state of work pieces. [Sta11] Several methods can be used to identify and track physical objects. Color detectors like implemented in the Presence Simulator and form detectors – e. g. QR code detectors, as described in Section 3.5.3 – are often used. But also non visual methods like RFID-chips (Radio Frequent Identification) are an option. For example Karotz, an artificial rabbit used in children's wards for social interaction has an RFID-sensor to react to artificial carrots and other objects with integrated RFID-chips. When a person that carrying an RFID-chip enters the sensor's range, Karotz can automatically send an email or an SMS. [Blo11]

The Marble Answering Machine was a seminal TUI.

Durrell Bishop's Marble Answering Machine (MAM) was an early TUI, designed in 1992. In fact, it was one of the first interfaces that interlinks the physical and the digital world by means of a TUI. [SLCJ04] The MAM was a fully functional telephone answering device. Phone calls received by the machine were physically represented by a marble. The voice message itself was digital, but the system stored the association of a specific voice message and a specific marble. Marbles had different colors so that the user could distinguish them. When the caller hung up and the telephone connection was interrupted the marble was ejected by the system. It rolled into a small hollow, which served as a marble gathering place. The hollow was open to the user, so he or she could

pick up a marble. To replay the message associated with the marble, users could place the marble in another hollow. This replay hollow had only space for one marble. In order to delete a message, the associated marble could be dropped into a hole. Although in 1992, there were also other TUIs available, the MAM was a pioneer and is referenced in nearly all introductions to TUIs.

Tangible bits bridge the gap between the virtual and the physical world.

In 1997 Ishii and Ullmer formulated a vision for TUIs in the paper "Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms". They were "inspired by the aesthetics and rich affordances of [...] historical scientific instruments" [IU97] and regret that these "beautiful artifacts" [IU97] have been replaced by personal computers. Computers provide a very general and mostly graphical interaction interface and thus the skills and work practices with physical objects that have been developed by humans during their evolution are mostly neglected. In their paper Ishii and Ullmer present several design projects that illustrate the tangible bits concept.

The metaDESK is a project for demonstrating and testing new TUIs.

One of the projects Ishii and Ullmer designed at the MIT Media Lab in Cambridge is named metaDESK. The system is an interface platform for the testing and demonstration of new tangible interaction techniques. Like the Presence Simulator developed in this thesis, it consists of a desk with movable objects (tangible bits) placed on the surface with a system tracking their location. Unlike the Presence Simulator, the metaDESK system was designed for a more general purpose and therefore is much more expensive to build. For example, the desk serves as a projection surface for a beamer, which is attached to the system's computer and placed under the desk. Depending on the users actions, the projection can be changed. The tangible bits are sensed "by an array of optical, mechanical, and electromagnetic field sensors" [IU97].

The metaDESK is a TUI allowing users to navigate through a map.

In their paper "The metaDESK: Models and Prototypes for Tangible User Interfaces" Ullmer and Ishii presented a sample application for the metaDesk called "Tangible Geospace". [UI97] In this application, a map that is projected on the desk can be panned or rotated by moving a tangible bit that looks like a model of a building and is named Phicon, on the desk's surface. The map can be zoomed by placing another Phicon in relation to the first one on the desk. An arm-mounted display shows a three-dimensional model of the map's area the display is showing. With a fiber-optic bundle named "passive lens", additional information of mapped regions can be displayed by moving the device to a specific region.

The reacTIVision project is a framework for the tracking of fiducial markers attached to physical objects.

The reacTIVision[2] software is a project that resembles the metaDESK project. The first version was developed in 2005 by Martin Kaltenbrunner at the university Pompeu Fabra in Barcelona. reacTIVision is released under the GPL license. The sytem is decribed by Kaltenbrunner in the paper "reacTIVision: a computer-vision framework for table-

[2] `reactivision.sourceforge.net/`

Figure 2.1: reactTable is a digital musical instrument and an application of the reacTIVision project. Fiducial markers at the bottom of the objects on the table are tracked by a camera underneath the table. [rea05].

based tangible interaction" [KB07]. A system that uses the reacTIVision software consists of a table with a special surface translucent for infrared light, like it is shown in Figure 2.1. Under the table, a camera and a projector is placed so that the view-port of both covers the entire table. The camera is able to receive light in the infrared range, so it can see through the table's surface. The projection from underneath the table can be seen from above. Users can move and rotate physical objects on the surface of the table. A two-dimensional fiducial marker is attached to the bottom of the objects so that the camera can observe it. The markers were designed with the help of an evolutionary algorithm. the reacTIVision software is able to track the markers position and there orientation in a real time video stream. Furthermore, the software is also able to detected finger tips on the table's surface.

Many projects are based on the reacTIVision system.

Many projects are based on the reacTIVision system. For example ToyVision, a software toolkit for the prototyping of tangible games [MCB12] or SoundStage, a surround sound mixer with a tangible interface. The goal of SoundStage was "to make ambisonic surround sound accessible and fun for kids." [Mur11]. A lot of other examples can be found at reacTIVision's vimeo video channel[3].

The Presence Simulator is not based on reacTIVision because building the system is too expansive.

The reacTIVision seems to be the perfect basis technology for the implementation of the Presence Simulator. It is has already been field-tested and proven capable of tracking objects precisely and reliably. However, its rather extravagant setup comes at a high price: Depending on the size of the system and the quality of its components, it can easily cost several thousands of Euros. Excluding pre-existing components

[3] vimeo.com/channels/reactivision/

such as the table and the computer, I spent a total of 56.11 Euros on my prototype.

Chapter 3

The Presence Simulator

In this chapter, the Presence Simulator system that has been developed for this thesis is described in detail. Already at the very beginning of the work on my thesis, it was clear that the implementation of the Presence Simulator had to meet several requirements. These requirements are described in the next section. In the subsequent sections, the software design of the system is described, starting with a top level view of the whole system. Next, the client and the server tier are discussed by presenting a top level view of the whole tier and then focusing on the components in the lower levels. The last section of the chapter focuses again on the Present Simulator's requirements, by discussing how they were met in the implementation. Also, some performance measurements and the discussion of the unit tests that sustain software quality are given in the last section.

In this chapter the Presence Simulator is discussed.

The source code of the Presence Simulator is released under the MIT license. It can be found on the CD-ROM attached in the back of this thesis and also under the link listed below the attached CD-ROM.

The source code can be found on the CD-ROM.

Users of the Presence Simulator can be divided into two groups. Researchers on the one hand are the primary users of the system. They set up the simulator by building the physical model landscape, mounting the webcam and tweaking the settings provided by the server's GUI described in Section 3.6. On the other hand, the participants of the location-aware study are also users of the system. They use the tangible interface by moving the LSFs in the model landscape. Of course their using of the evaluated smartphone application is also affected by the Presence Simulator system. Which makes the participants secondary users in this case. In this chapter, when the users are mentioned, in most cases the researchers are meant. In all other cases it is explicitly stated.

The Presence Simulator has two types of users, namely researchers and study participants.

3.1 Requirements

Requirement lists are outdated, but suitable for the purposes of a thesis.

In this section a list of requirements for the implementation of the Presence Simulator is presented. Providing such lists is a traditional way of documenting software requirements, but has gone out of fashion in modern analysis. [KE04] A more recent alternative to requirement lists are user stories. User stories describe in in a short story what the user does with the system as part of his or her job. However, developing software for the purposes of a computer science thesis is a unique setting where most constraints of business software development do not apply. For example, the requirements usually do not change during the processing period of the thesis. Therefore, a requirement list suits quite well to describe the needs of the Presence Simulator. Later, in Section 3.8, the list presented here will be revisited again to discuss how the implementation satisfies the requirements.

Functional requirements describe *what* to do, non-functional requirements, *how* to do it.

Functional requirements describe the interactions between the system and its surroundings, independent from its implementation. Functional requirements describe *what* the system must do. Non-functional requirements focus on the system it self, by defining *how* the functional requirements should be executed.

3.1.1 Functional Requirements

1. The system should be able to detect the position of little toy figures (LSFs) placed in a model landscape as pixel coordinates in a picture that is captured from above the model and is showing the whole landscape.
2. The system should be able to distinguish multiple LSFs situated in the same model landscape.
3. Users should be able to define a geographic area that corresponds to the model landscape.
4. The location of the LSFs in the model landscape should be mapped to the corresponding geographic coordinates in the defined geographic area. The mapping should be executed continuously, resulting in a stream of location updates for each LSF.
5. Users should be able to assign a specific LSF to an Android device. After the assignment, the Android device should be informed about each location update of the assigned LSF.
6. Users should be able to cancel assignments that have been made between LSFs and Android devices.
7. Location updates should be transmitted over the air to the Android device.
8. Whenever the Android device receives a new update the device's system location should be set to the coordinates specified within the update. This update mechanism should also be active when another application is running (e. g. the application to be evaluated).

3.1.2 Non-Functional Requirements

1. Reaction time should be fast. That means, a participant should not be able to notice a delay between his or her action of moving an LSF and the smartphone's response in changing the location.

2. Spatial resolution should be high. That means that even small changes of the figures position should result in changed geographic coordinates.

3. The simulated presence should be accurate. That means that the geographic coordinates corresponding to an LSF should be the ones that one expects when the figure is placed on a certain position. Also the coordinates should not fluctuate while the LSF stands still.

4. The update rate should be at least one update per second (the iPhone 4 – one of the best selling smartphones – has an update rate of up to 2Hz [Bro07]).

5. The size of the model landscape should be scalable. A landscape measuring 1 by 2 meters should be possible (which is about the size of standard desk at the university of Bonn).

6. LSFs the size of playmobil® figures (eight centimeters high and four centimeters wide) should be tracked reliably by the system.

7. The system should be easy and convenient to use for both researchers and study participants.

3.2 Design Overview

3.2.1 System Architecture

The Presence Simulator is based on a client-server model.

The implementation of the Presence Simulator is based on a client-server model. Client-server models partitions tasks in an distributed application between service providers, called servers, and service consumers, called clients.

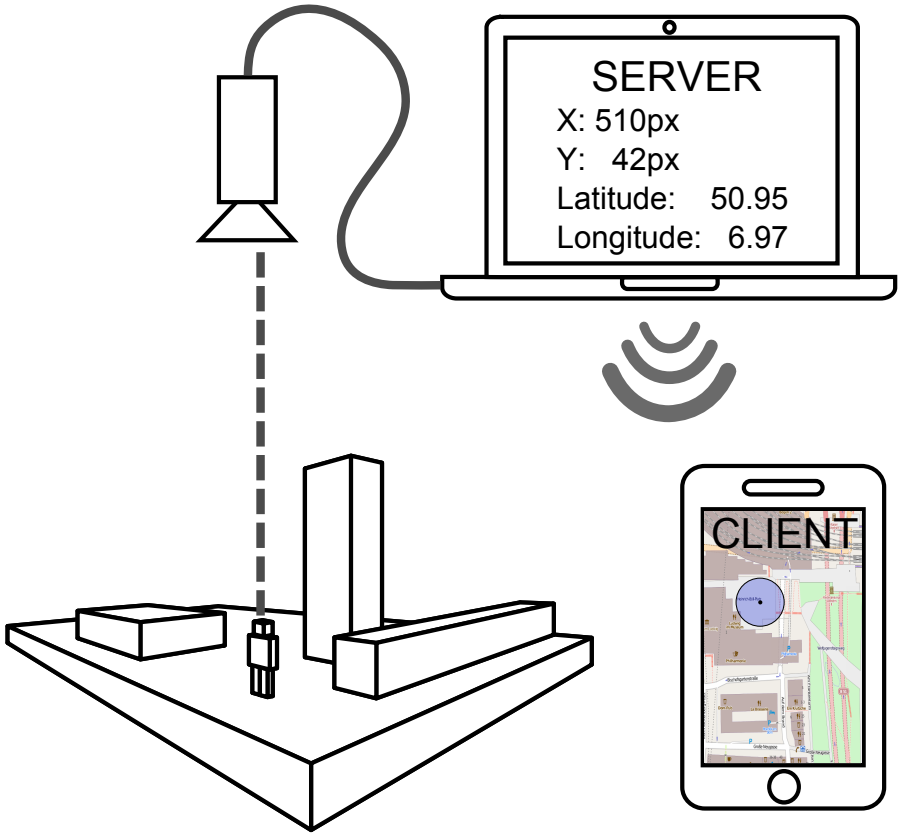


Figure 3.1: Model of the Presence Simulator system, showing a tangible user interface on the left side and the server and the Android client on the right side. The server is processing the webcam's video stream and the client is receiving location updates from the server.

Client-Server model

CLIENT-SERVER MODEL:
Systems based on the client-server model are two-layered with a specified interface between the layers. Client-Server models can be found on different levels, for example in computer networks as is case with the Presence Simulator, but also in a software running on only one device.
In the 1990s, client-server systems gained popularity at the same time as object-oriented programming languages rose. The servers in the 1990s were usually relational databases, offering data services. Nowadays, there are also other services offered by client-server systems. Still, in most cases the client is responsible for the user interface. But depending on its performance (thin versus fat client), additional application code can be executed by the client. A major advantage of the client-server model is that these systems are easy and fast to develop and usually also achieve decent performance. On the other hand, they are not very scalable, difficult to maintain, and provide a single point of failure. [Fow02]

The server tracks the location of the LSFs and transfers their coordinates to a client software running on Android smartphones.

The client-server model implemented in this thesis is depicted in Figure 3.1. On the left side of the figure, a TUI in form of a model landscape with an LSF placed in it can be seen. For the sake of simplicity, only one LSF is shown, but the system is capable of handling more than one. A participant of a study would stand next to the landscape and move the LSF. A webcam is mounted above the model landscape, pointing straight downwards. The webcam is attached to a computer. On the computer, the Presence Simulator server software is running. The server software is responsible for analyzing the video stream from the webcam in order to detect the positions of the LSFs and map these positions to geographic coordinates. The server is offering the streams of geographic coordinates as a service in the LAN. The client is an Android smartphone and is connected to the server via a wireless network connection. It consumes the updates of geographic coordinates from the server, by setting the geographic location stored in the device's Android system accordantly.

The LSFs are tracked passively only by their appearance in the webcam's video stream – no infrared emitters are used.

In the exposé of my thesis, I described the Presence Simulator system slightly different. When I began to work on my thesis, I had the intention to attach an active infrared emitter to the LSFs, pointing upwards in the direction of the webcam. Normal webcams have an infrared filter mounted in frond of the sensor, therefore this filter would have been removed and replaced by an infrared-only filter. As a result, recognizing the LSFs in the webcam's video stream would have been quite easy. However, distinguishing different figures would have been harder, because all emitters would look similar on the video frames. Also, setting up the whole system would have been much more complicated. For example, in addition to the infrared sensors, batteries would have had to be attached to the LSFs, making the figures appear less human. Fortunately, it turned out that image recognition on common non infrared video streams performs well enough for the demands of the Presence Simulator.

It would have been an option to split the server into two separate tiers.

The Presence Simulator server mainly consists of two parts: the image recognition part and the part offering the location service in the network. So, it might have been a good idea to split the server into two separate components. In the context of multi-layered architectures, the components are usually called tiers. The client-server architecture is a two-tier architecture, whereas the proposed architecture would have been a three-tier architecture.

Multi-tier architecture

MULTI-TIER ARCHITECTURE:
Multi-tier architectures are a generalization of the client-server architecture. The terms tier and layer are often used synonymously. A tier is an independent component in a distributed application. The motivation in having more layers is to reduce complexity by using a divide and conquer approach. To solve tasks from higher levels, services from lower levels are used. One level should only communicate with services from the next lower level, because otherwise complexity would increase a lot and the system would soon be unmaintainable. An example for the use of a three-tier architecture in computer networks is the extension of the client-server architecture by adding a firewall between client and server to increase security. One problem of multi-tier architectures is that the more layers are implemented, the worse the performance usually is. [Fow02]

Three reasons for using a client-server architecture instead of a three-tier architecture.

There are three reasons for implementing the Presence Simulator with a client-server architecture and not with a three-tier architecture:

- The additional layer would result in a performance drawback due to communication overhead, which is critical because the first non-functional requirement states that the system should react fast.
- Setting up the Presence Simulator would take more time, due to the fact that another tier needs to be installed and communication channels established, which is critical because the last non-functional requirement states that the system should be easy to use.
- Implementing the Presence Simulator would have taken more time, which is critical because the thesis is restricted to a six month period.

I believe these arguments outweigh the improved flexibility that a three tier architecture would provide. However, as shown in the subsections about the components of the server tier, the implemented architecture is still quite flexible and can easily be transformed into a three-tier architecture in future work.

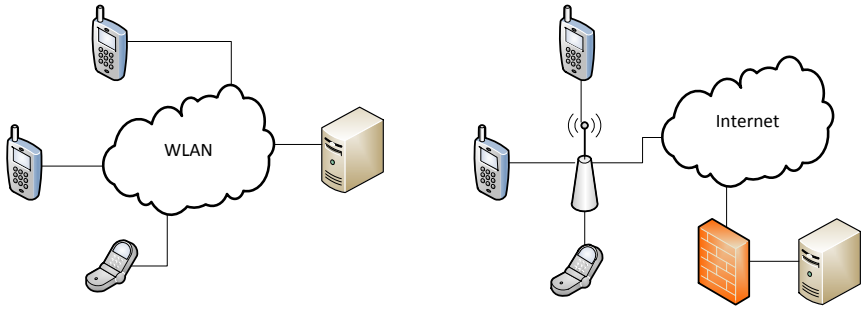
Most other architectures are not suited for the Presence Simulator.

Of course, there are more ways of designing computer software modules and the communication between them than the client-server model and its multi-tier generalization. However, most of them are either special versions of the two already mentioned – like the peer-to-peer model – or simply do not fit the requirements of the Presence Simulator – like the pipes and filters model.

3.2.2 Network Architecture

Client and server are connected via WLAN.

The network architecture defines the communication channels between the Presence Simulator's server and client. Functional requirement number seven states that location updates need to be transmitted over



(a) Client-server communication via WLAN

(b) Client-server communcation via the internet

Figure 3.2: Two ways of settings up the connection between the Presence Simulator's server and client. The WLAN setup is preferable, but external conditions might force to communicate over the internet.

the air. So, a wireless communication channel is needed. The easiest way to achieve this, is to simply connect server and Android client to the same wireless local area network (WLAN) as depicted in Figure 3.2a. All Android devices have a built-in WLAN device, as do most personal computers.

Setting up an ad-hoc WLAN can be done fast, but is not supported by all devices.

A easy way to set up a WLAN is the usage of an ad-hoc network. Ad-hoc WLANs do not need a WLAN access point. Instead, the network nodes communicate directly to each other in a peer-to-peer communication style. Most WLAN drivers should be able to set up or connect to an ad-hoc network, but especially older Android devices like the Motorola Defy, that has been used in the evaluation of the Presence Simulator, might lack this functionality. The ad-hoc mode also decreases the computer's performance compared to a normal access point based WLAN. Therefore, when there is already a WLAN access point available, it is usually a good idea use it.

When WLAN is not available but the server is accessible from the internet, the smartphone's 3G connection can be used.

Sometimes WLAN communication is not an option. Maybe the WLAN is also used for private data and one does not want to compromise the network by connecting an Android device to it. Sometimes an ad-hoc network is hard to configure and maybe connection cannot be established. When the Presence Simulator server is accessible from the internet, there is another way to set up a connection between client and server. It is possible to connect to the server's internet protocoll (IP) address and port, so Android smartphones can use their built-in 3G network device to connect to the internet and can therefore connect to the server without using WLAN. This connection type is shown in Figure 3.2b. Of course, 3G network connections are often not as reliable and fast as WLAN connections, so this type should only be used when WLAN is not an option.

3.3 Basis Technology and Frameworks

The server is based on the .NET framework, the client on the Android platform.

In this section, the two most basic frameworks that were used for the implementation of the Presence Simulator are introduced: The .NET framework [Mic] and the Android platform [Gooa]. Because the .NET platform is not very popular at universities, also a rough introduction to C# – which is one of the .NET programming languages – is given. Java on the other hand is so well known in the research community that I will not introduce it here, although it is used by Android.

A Java based Presence Simulator prototype performed worse than the .NET based one.

Why is the .NET platform used? Because the client software needs to run on an Android device, it was inevitable to program the client in Java. Of course, my first idea was to write the server's code in Java, too. However, I faced some difficulties in communicating with the webcam and with image processing while working on a Java based prototype of the Presence Simulator. Out of curiosity, I decided to give the .NET platform and C# a try – I had no experience with the platform. Soon, I found a really useful image processing library called AForge [Kira]. The first prototype turned out to perform much better than the Java version and so I stuck to it.

3.3.1 .NET Framework and C#

.NET is a software platform developed by Microsoft and released in 2002.

The .NET platform is a software framework as well as a runtime environment developed by Microsoft during the last ten years. The first version was released in 2002. At the time of writing, the current stable release is .NET 4.0, which was released in April 2010. The .NET platform provides a healthy ecosystem. A lot of third party tools, libraries and frameworks can be found on the internet.

The CLR is comparable to the JVM.

The runtime environment of the .NET platform is called Common Language Runtime (CLR). .NETs CLR is comparable to the Java virtual machine, because it is an interpreter which executes intermediate code. Like the Java bytecode, which can be compiled from several different languages, .NETs Common Intermediate Language (CIL) can be generated from different higher level languages, like C# or VB.NET.

Performance can be gained by using unmanaged code.

Technologies like reflections, garbage collection or just in time compiling has been adapted by Microsoft from the Java platform. But there are also a lot of differences to Java. One fundamental difference is that .NET programs can consist of code managed by the CLR (managed code) as well as unmanaged code like calls to Microsoft's Component Object Model (COM). In contrast to Java, .NET supports pointers on the memory space and operations on them. Programmers need to be aware of the risks that pointer-operations come with, and therefore, code with pointer-operations needs to be declared in an unsafe code block. By bypassing the features of the CLR, a lot of performance can be gained. The image analyzing parts of the Presence Simulator are making extensive use of these features.

As mentioned, CLR code can be generated from several higher level languages. The server part of the Presence Simulator is written in one of them: C#. C# is a object oriented programming language with static and strong typing. As mentioned, pointers can be used in unsafe code blocks.

C# is a .NET language with static and strong typing.

In C#, pointers on methods are supported. They are called delegates. Unlike pure method pointers, delegates also contain a pointer to the object the method belongs to as an implicit parameter. C# supports generics, anonymous methods, generators and partial classes (classes whose definition is split into multiple pieces across multiple files).

C# supports delegates.

The .NET framework runs primarily on Microsoft Windows. But there exists an open source, cross-platform implementation named Mono[1] of C# and .NET's CLR that is compatible with the .NET framework. All third party libraries used by the Presence Simulator are said by their developers to be fully operational with the Mono environment. Therefore, it should be possible to make the Presence Simulator runnable under Linux or MacOS with minimal efforts.

Mono is an open source and cross-platform implementation of the .NET framework.

There are a lot of software editors and Integrated Development Environments (IDEs) for the .NET languages. The most commonly used by far is Microsoft's Visual Studio[2]. It is a full-blown IDE with a lot of features. Like Eclipse for Java, it provides programmers with nearly everything they need.

Visual Studio is the standard IDE for .NET.

3.3.2 Android

Android is an operating system and a software platform for mobile devices. It is developed by the Open Handset Alliance[3] and its main member Google. Android is free and open source. As a smartphone OS, Android had a market share of 68.1% in the second quarter of 2012 [IDC12].

In Q2 2012 Android had a market share of 68.1%.

Android is based on the Linux kernel. Therefore, the memory and process management as well as the interface for multimedia playback and network communication provided by the Linux kernel are used. Of course, the kernel used in Android is not a vanilla kernel, but a modified one with added hardware drivers and tweaks suitable for mobile devices.

Android is based on a modified Linux kernel.

Software for Android is written in Java, but Android is not using a standard Java Virtual Machine (JVM). The JVM in Android is developed by Google and called Dalvik Virtual Machine (DVM). It is an JVM optimized for mobile devices and incompatible to other JVMs. As part of

The DVM is a JVM optimized for mobile devices.

[1] www.mono-project.com/
[2] www.microsoft.com/visualstudio/
[3] www.openhandsetalliance.com/

the Android software platform, Google provides a well documented and comprehensive software development kit.[4]

Android is not as open as Google may want us to believe.

Due to its relatively open nature, Android is very popular in the research community and the industry [RLMM09]. That is the reason why the Presence Simulator's client is implemented for Android. However, Android is not as open as Google may want us to believe. As Joe Hewitt, a Mozilla developer states: "Until Android is read/write open, it's no different than iOS to me. Open source means sharing control with the community, not show and tell." [Hew10]

For the Presence Simulator, Android is open enough.

For this project, Android is open enough. For example, in contrast to Apples iOS, Android allows programmers to simulate (mock) the devices location by implementing a test location provider and registering it to the system location services (see Section 3.8.2). However, during implementation, some problems were faced. For example: While mocking the device's location was no problem, mocking the device's orientation is currently not possible.

3.4 Server Design Overview

The Presence Simulator comes with a lot of features that enhance usability.

In a basic and most requirement satisfying version, the Presence Simulator's server is mainly responsible for detecting the positions of the LSFs in the video stream, mapping these positions to geographic coordinates and transferring them to the client. However, in its basic version, the server would not be very useful. A lot of features which highly increase usability would be missing. Therefore, I added additional features to also satisfy the last requirement that states that the system should be easy and convenient to use. Some of the more advanced features that were implemented are an auto calibration system, a location source track recorder, and an easy and quick to use interface. These features are discussed in detail in Section 3.5. The current section concentrates on the underlying software design that greatly simplified the implementation of the server's additional features.

The Presence Simulator's server is based on an MVC architecture.

The Presence Simulator's server is based on a Model-View-Controller (MVC) architecture. This architecture was first formulated in the 1970s by Trygve Reenskaug at Xerox PARC, where a lot of research on human computer interaction has been done. [Wei08] Originally the architecture was meant for designing computer programs with user interfaces. In the following years, the architecture was extended, so nowadays it is also used in other contexts.

The MVC architecture separates information and representation.

The MVC architecture is a flexible architecture, allowing the system to be easily extended later on. This flexibility is achieved separating the information from its representation. The MVC architecture is depicted in figure 3.3. Subsystems of applications that implement the MVC architecture are divided into three different categories:

[4] `developer.Android.com/sdk/`

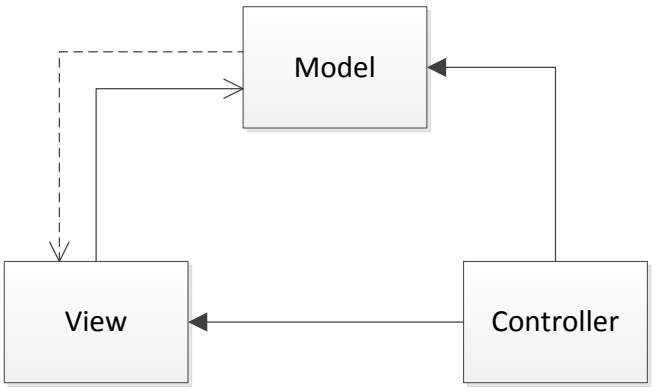


Figure 3.3: The Model-View-Controller architecture enables a convenient way of extending software systems.

- The model is responsible for the knowledge of the application domain and the notification of the views when there has been a change in the model's state, so the view can update the model's representation. In the Presence Simulator the model consists of a list of location sources, with data about the position and corresponding geographic location of the LSFs. The location source component is discussed in Section 3.5.1.
- When the view is notified by the model, it requests the information from the model that is needed for the representation. The Presence Simulator has several views implemented. For example, in the server's user interface, there is a map view where for each location source, a marker is drawn. But the connection to the Android client is also implemented as a view of a location source model. The views are discussed in detail in Section 3.5.3, 3.5.4, and 3.5.1.
- The controller contains the logic that is responsible for manipulating the model (i. e. by updating the geographic location stored in the model). In most MVC architectures, the controller can also manipulate the views (i. e. when a text view is scrolled, the view is changed by the controller). However, control mechanisms like this are not implemented in the current version of the Presence Simulator. Controllers can be found in the Detector component, discussed in Section 3.5.2 and in the Recorder component, discussed in Section 3.5.5.

The observer pattern can be used to decouple the model from the view.

When the model is changed and the views are notified, does this imply that the model needs knowledge of all the views, so whenever a new view is implemented the programmer needs to adapt the model, too? There is a technique to avoids this: By adding an abstraction layer between model and view, the model only needs to be aware of an abstract view type and not of all concrete view types that are implemented. This is indicated with a dashed line between model and view in Figure 3.3.

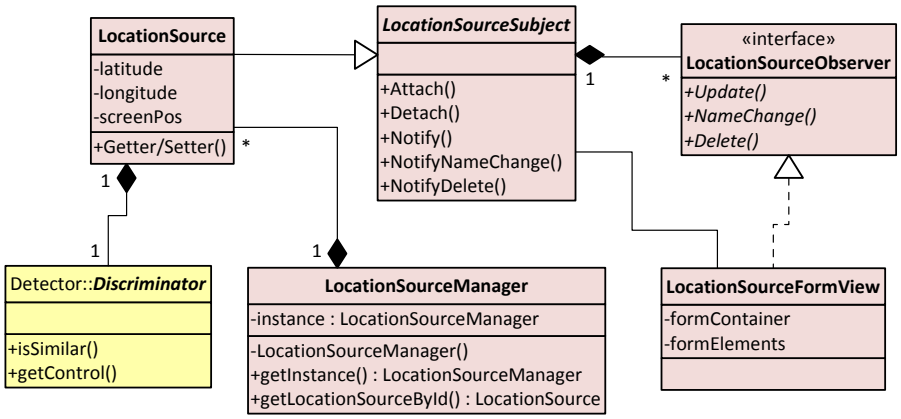


Figure 3.4: UML class diagram of the server's Location Sources component. The yellow colored class is a dependency form the Detectors component.

The abstraction layer can be implemented with the help of the observer pattern, which is described in Section 3.5.1.

3.5 Server Components

In the following subsections the five components of the Presence Simulator's server are described.

3.5.1 Location Sources

The *LocationSource* class models the LSFs.

An instance of the *LocationSource* class models an LSF. It is the combination of the position of the LSF in a frame of the video stream in terms of pixel coordinates as well as the corresponding geographic coordinates, which are restricted by the spatial area defined by the user. For convenience and usability, the model also has a name property and a unique identifier.

LocationSource objects are serializable.

LocationSource objects are serializable. This means, that they can be transformed into a string (serialized) without loosing information. Later on, the string can be transformed backwards (deserialized) to rebuild the same object. This feature is necessary to transmit location sources via a network connection – as used by the server's Network component described in Section 3.5.4 – or save them to a file – as used by the server's Recorder component described in Section 3.5.5.

LocationSource objects have a *Discriminator* to distinguish them.

In addition to the simple data properties, location sources also have a property of type *Discriminator*. *Discriminators* are used by location source detectors to discriminate location sources detected in the video stream. That's why the *Discriminator* interface defines a method for testing whether two discriminators are similar to each other. The concrete *Discriminator* type that is attached to the *LocationSource* depends on the used location source detector (see Section 3.5.2).

According to the second functional requirement, the server should be able to handle and distinguish multiple location sources at the same time, so that study participants can control several LSFs in the model landscape or studies with multiple participants at once are possible. Therefore, a class called *LocationSourceManager* was implemented which bundles several *LocationSources* into one object, by managing them in a list. The manager has some handy methods to select a specific *LocationSource* by a *Discriminator* or by an identification string. The manager is also responsable for shutting down all *LocationSources* when the application is closed by the user. Because the server only needs one *LocationManager*, the class is implemented as a singleton. Singleton classes can be instantiated only once.

The *LocationSourceManager* manages multiple *LocationSources*.

SINGLETON PATTERN:
The singleton pattern is a creational pattern to ensure that there exits only one object of the class the pattern is applied to. The singleton was introduced by Gang of Four in their book "Design Patterns. Elements of Reusable Object-Oriented Software" [GHJV95]. The pattern is usually implemented by having a private constructor and adding a static method that instantiates the class' object when the method is called for the first time (lazy instantiation). The instance is then saved as a private and static member.
In C#, the member and its getter method are usually implemented as a property. One problem with this implementation is that – at least in C# – it is not thread-safe. When multiple threads enter the property at the same time, multiple instances of the singleton might be created. As the Presence Simulator' server is multi-threaded, this should be avoided. Luckily, it can be, by having a static instantiation instead of a lazy one. In order to achieve this, all singletons in the Presence Simulator are marked as *sealed* to prevent that derivations of the singleton class are able to add instances. The property is marked as *readonly*, so assignments can only be made during the static initialization. [MS]

Singleton Pattern

Figure 3.4 shows the class diagram of the location source component with the *LocationSource* class and the *LocationSourceManager* class. The abstract class *LocationSourceSubject* that is generalizing the *LocationSource* class is part of the observer pattern. The observer pattern allows the *LocationSource* model to notice subscribed views whenever the model's data gets updated, without having to know all concrete view types. The model only needs to know the *LocationSourceObserver* interface. There are three different methods for notification implemented: One is for informing the observers about changes in the geographic coordinates, one is for announcing a name change and one is for announcing that the *LocationSource* object is about to be deleted, so observers can adapt to this (e. g. by closing a network connection to the client).

LocationSource, *LocationSourceSubject* and *LocationSourceObserer* are implementing the observer pattern.

Observer Pattern

OBSERVER PATTERN:
The observer pattern is a behavioural pattern introduced by the Gang of Four. [GHJV95] The pattern enables a subject to notify other dependent objects, called observers, about changes. The subject is offering a service for observers to subscribe and unsubscribe them selfs to the updates. The subject is not depending on all concrete subscribed observers, only on the observer interface. The observers are implementing an update method which is called by the subject when changes are emerging.
In the Presence Simulator, the pull model of the observer pattern is implemented. That means that observers pull the updated information they need from the subject within their update method. For that reason, observers need to maintain a reference to a subject object.
In contrast to the pull modell, in a push model the information is passed as a parameter to the observers. Therefore, all observers get the same detailed information, whether they want it or not.

The *LocationSourceFormView* is a concrete observer that visualizes LSs in the GUI.

A concrete *LocationSourceObserver* is already implemented in the Location Sources component. It is a very basic view named *LocationSourceFormView*. It represents a *LocationSource* within the user interface of the server by visualizing it as a text view of the geographic coordinates, a text box for the name and a control for the discriminator. The name text box also serves as a controller, because it enables users to change the *LocationSource's* name property.

3.5.2 Detectors

Detectors recognize LSFs in the video stream by applying computer vision algorithms.

"The basic idea behind [object tracking] is to define marking attributes (patterns, shapes, etc.) on the given object, use image processing techniques to collect spatial data from the camera stream and finally to calculate the 3D position based on the 2D image(s)." [PSKS11] Detectors are responsible for detecting the position of LSFs in terms of pixel coordinates within the single frames of the video stream provided by the webcam. The detectors implemented in the current version of the Presence Simulator apply computer vision algorithms in order to separate the region with the LSF from the rest of the image and also to distinguish several found LSFs from each other. For the computer vision parts, the Presence Simulator uses the *AForge.Imaging* and the *AForge.Video* library [Kira].

The AForge.NET framework is designed for developers in the field of computer vision.

The *AForge.NET* framework is written in C#, released under the GNU Lesser General Public License and designed for developers in the field of computer vision and artificial intelligence. The framework is in constant progress and continuously improved. The *AForge.Video* library provides easy access to webcams. The webcam's video stream is processed efficiently and fast by multiple threads. The *AForge.Imaging* library is providing processing routines and filters for images.

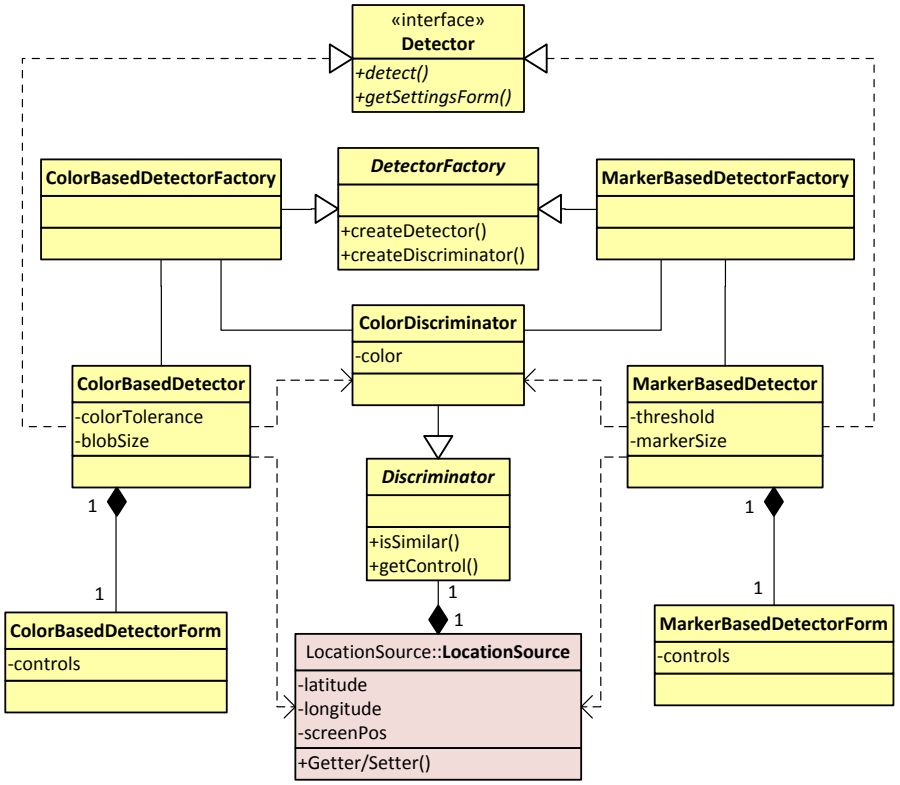


Figure 3.5: UML class diagram of the server's Detector component. The violet colored class is a dependency from the Location Sources component.

The *VideoSourcePlayer* class of the *AForge.Video* library is used to access the video stream of the webcam. Users of the Presence Simulator are able to select the webcam device and its resolution (see Section 3.6). When the *VideoSourcePlayer* is initialized during the start of the Presence Simulator, a listener for new video frames is registered to it. From thereon, the listener is called by the *AForge.Video* library each time a new frame is received from the webcam. The *AForge.Video* library is managing a thread pool for the video processing. When the listener takes more time to process a video frame than the period between two subsequent frames, a new thread is launched to process the new frame. The *AForge.Video* library automatically skips frames when processing speed becomes critically slow. However, on the rather old computer that was used in the evaluation of the Presence Simulator, a delay between the capturing of a frame by the webcam and the finalization of its processing was experienced (see Section 3.8.2).

The *VideoSourcePlayer* class provides efficient access to the webcam by managing multiple threads.

The listener passes a reference to the new video frame to the detector, which then analyzes the frame. In the current version of the Presence Simulator, two detectors with different characteristics are implemented. Both are implementing the *Detector* interface as shown in Figure 3.5. Users are able to select the *Detector* they want to use and programmers are able to extend the Presence Simulator by implementing

Concrete *Detectors* have to implement the *Detector* interface.

new detectors. This flexibility is achieved by the implementation of a abstract factory pattern.

Factories are responsible for the creation of *Detectors* and compatible *Discriminators*.

A factory is an object that is used to create other objects. An abstract factory is used for the creation of related sets of objects. The *DetectorFactory* interface defines two methods: one for the creation of *Detector* objects and one for the creation of *Discriminator* objects. Methods like these are called method factories. As mentioned above, discriminators are used by detectors to distinguish LSFs. Because there might be detectors that use special features to detect LSFs, using a generic discriminator won't work. For both detector implementations, there is a concrete *DetectorFactory*. The concrete factory's methods create the specific detector and the discriminator that is used by this detector.

Method Factory Pattern

METHOD FACTORY PATTERN:
The method factory pattern is a creational pattern that controls the creation of objects by adding an abstraction layer. According to the Gang of Four, the essence of the pattern is to "define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses." [GHJV95] The pattern is often used by frameworks, because it allows subclasses to select the concrete type of objects that should be created. The main flow of control is defined by the framework but the user application can define what is happening in a specific situation by plugging in the desired concrete method factory.

Users have to choose between a color based and a marker based LSF detector.

The two detectors which are implemented in the current version of the Presence Simulator are the *ColorBasedDetector* and the *MarkerBasedDetector*. They implement the *Detector* interface, that specifies a method for the detection of LSFs and a method that returns a graphical user interface in which by users can tweak the detector. The *ColorBasedDetector* detects the LSFs by means of their color. The detector is fast, but it needs a large contrast in color between the LFSs and the model landscape. Also, the lighting conditions should be as constant as possible (e. g. the model landscape should not be placed in front of a window). The *MarkerBasedDetector* on the other hand is robust to changes in lighting, but it requires the LFSs to "wear" special two dimensional markers on top of them – which might be no problem when Matchbox® cars are used as LFSs, but when it comes to Playmobil® figures, it might not feel natural for study participants.

Color Based Detector

Tracking objects by their color is a well known approach.

Tracking objects by their color is not a new approach. For example a color tracking algorithm was already implemented in the 1990s by McKenna, Raja and Gong [MRG99]. Nowadays, computers are fast enough to easily apply the method in real time on the frames of a high resolution video stream. In 2008 Andrew Kirillov developed a system

for the Lego Mindstorm NXT robotics kit[5] which allows color based object tracking [Kir08]. The color tracking algorithm implemented in the Presence Simulator is based on his work.

Algorithm 1 Algorithm for the detection of LSFs in video frames based on their color.

Require: An image of the current webcam frame and a list of location sources.

Ensure: The location sources of the list have updated latitude and longitude coordinates.

```
for all locationSource in locationSourceList do
  if typeof locationSource is ColorDiscriminator then
    frame = Clone(currentFrame)
    ColorFilter.applyInPlace(frame, locationSource.Discr.Color)
    grayImage = GrayScaleFilter.apply(frame)
    blob = BlobCounter.getLargestBlob(grayImage)
    if blob.Size is within the size specified by the user then
      locationSource.LatLng = Map.ConvertToLatLng(blob.Center)
    end if
  end if
end for
```

The algorithm iterates through all available location sources.

Algorithm 1 shows a pseudo code version of the color tracking algorithm. When the *ColorBasedDetector* is called by the new video frame listener, the current video frame is passed to it. The *ColorBasedDetector* has access to the *LocationSourceManager* and is therefore able to control all the *LocationSources* created by the user. The algorithm's outer loop iterates through the available *LocationSources*. For each *LocationSource*, the current video frame is cloned, because some instructions in the loop's body performed on the image are irreversible.

A AForge color filter is applied to the cloned video frame.

On the cloned video frame, an AForge color filter is applied. The filter sets all pixels outside of a specified RGB color range to black. The color range is defined by the *Discriminator's* color of the current *LocationSource* and the color tolerance value is set by the user. To get good results from the detector, the scene filmed by the webcam needs to be uniformly illuminated, so that the color values of the LSFs do not depend on their location in the scene.

The image is grayscaled and blobs are detected. Blobs of a certain size are assumed to be LSFs.

To reduce complexity, the resulting frame is then converted to a gray scale image and the AForge *BlobCounter* is applied. The *BlobCounter* extracts stand alone regions (blobs) that have a uniform color differing from the background. It is based on a connected-component labeling algorithm [CLRS09]. The *BlobCounter* might return several blobs, depending on the level of noise in the video frame. Therefore, only the largest blob is chosen by the *ColorDetector*. If the blob's dimensions fit into the maximum and minimum values set by the user, the *ColorDetector* assumes that the blob corresponds to the current *LocationSource*.

[5] `mindstorms.lego.com`

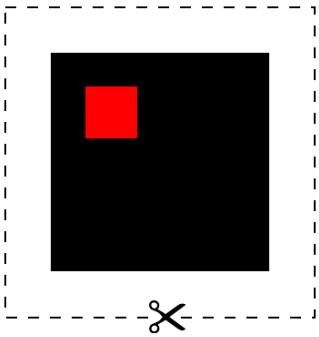


Figure 3.6: A marker the *MarkerBasedDetector* is able to detect. Ready to cut out and stick on location source figures.

The pixel coordinates of the blob's center are then converted to latitude and longitude coordinates by making use of the Map component (see Section 3.5.3). The geographic coordinates are then used to update the *LocationSource*.

Marker Based Detector

The *MarkerBasedDetector* detects optical glyph like patterns.

The marker based detector tries to locate optical glyph like patterns in frames of the video stream. Optical glyphs are grids with the same number of columns and rows. Each cell of the grid is either black or white. To distinguish the glyph from its background, the first and last row and column is always black and the glyph is printed on white paper. The cells of the inner rows and columns form a pattern which is used to detect the orientation of the glyph and to distinguish glyphs from each other. Optical glyphs are often used in augmented reality to add 3D animation to a video stream.

Normal glyphs are to large, but black squares on white background,with a small colored region, work.

In AForge, there is already a Glyph Recognition and Tracking Framework (GRATF) implemented [Kirb]. During the initial tests I made, it turned out that the glyphs need to be much larger than playmobile® figures to make the system operate precisely and reliably. However, the detection of simple black squares on a white background without the grid pattern inside works well, even when the square is sized like the hat of a playmobile® figure. That's why I replaced the grid pattern of the glyph with a black area and added a small colored square, halfway between the center of the black square and one of its corners. The resulting marker is shown in Figure 3.6. The colored square is used by the detector to detect the orientation of the marker and to distinguish markers from each other – each marker has a square colored in a different color.

The color tolerance value can be set high.

Because the detector is using color to distinguish LSFs, a color filter is applied, like it is in the *ColorBasedDetector*. But the color tolerance value can be set much higher, because only as many colors need to be distinguish as there are location sources and not as there are colors in the video frame.

Algorithm 2 Marker based algorithm for the detection of LSFs in video frames.

Require: An image of the current webcam frame and a list of location sources.

Ensure: The location sources of the list have updated latitude and longitude coordinates.

```
frame = Clone(currentFrame)
grayImage = GrayScaleFilter.apply(frame)
gradientImage = SobelFilter.apply(grayImage)
thresholdImage = ThresholdFilter.apply(gradientImage)
blobs = BlobCounter.processImage(thresholdImage)
for all blob in blobs do
  if blob.Size is within the size specified by the user then
    if blob is square and contrast to its background is large then
      color = getBlobColor(blob, currentFrame)
      discriminator = new ColorDiscriminator(color)
      LsManager.updateLs(discriminator, blob.Center)
    end if
  end if
end for
```

Frames are cloned and grayscaled, then a sobel-operator is applied.

A pseudo code version of the algorithm used by the *MarkerBasedDetector* is shown in Algorithm 2. First, the current frame is cloned and a grayscale filter is applied on the clone. Then, the sobel-operator is applied to the grayscaled image. The sobel-operator detectes edges in images by computing a gradient image of the image intensity function. This is done efficiently by shifting a 3x3 convolution kernel over the image computing the maximum difference between pixels in four directions around the processing pixel [KVB88]. Because the contrast of the marker's black square to the white background is very high, the edges of the square are highlighted in the gradient image.

The gradient image is binarized and blobs are detected.

The gradient image is then binarized with a threshold filter. The threshold can be set by the user in the detector's settings. Then, the AForge *BlobCounter* is applied to the monochrome image to get all stand alone regions of white color. The blobs that are returned by the *BlobCounter* are filtered again, so that only the blobs remain that fit into the maximum and minimum size dimensions defined by the user.

The remaining square blobs are assumed to be LSFs and are distinguished by the small colored region.

At this stage, there are still blobs in the list that are not squares, so these need to be filtered out. This is achieved by detecting the blob's corners and checking whether they fulfill the criteria of a square. When they do not, they are filtered out. The *MarkerBasedDetector* assumes that the remaining blobs correspond to LSFs. In order to distinguish them, their coordinates are transferred back to the original video frame and the color of their small colored region is extracted. This color is used to create a new *ColorDiscriminator* and together with the blob's pixel coordinates, the discriminator is passed to the *LocationSourceManager*. The

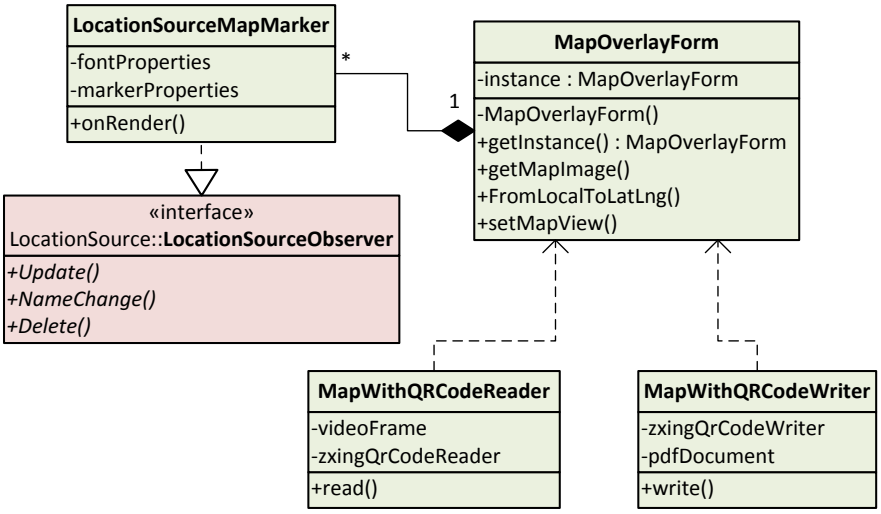


Figure 3.7: UML class diagram of the server's Map component. The violet colored class is a dependency from the Location Source component.

LocationSourceManager takes care of converting the pixel coordinates to geographic coordinates and updating the right location source.

The *MarkerBasedDetector* is able to detect the orientation of LSFs.

One advantage of the marker based approach is that it is able to detect the orientation of the marker. The Presence Simulator in its current version is not making use of this feature, but it would be an option for future works. In fact, implementing the orientation feature only failed because in the Android framework it is currently not possible to mock orientation data.

3.5.3 Map

The Map component maps pixels to geographic coordinates and visualizes LSs on a map.

The primary task of the server's Map component is to map pixel coordinates of detected location sources from a webcam video frame to points in a geographic coordinate system. The geographic coordinates can then be used by location source controllers like the detectors discussed in the previous subsection to update the location source's presence. The secondary task of the component is to visualize location sources on a map view in the server's GUI.

GEOGRAPHIC COORDINATE SYSTEM:
Geographic coordinate systems assign a unique set of numbers and/or letters to every location on the earth . There are several systems which differ in the used measurements and accuracy. In this thesis, the commonly used coordinates latitude and longitude are used. A system quite similar to the latitude-longitude system has already been used by Greek scientists more than two thousand years ago. [Ord10]
The latitude coordinate specifies the north-south position of a point on the surface of the earth. It is the angular distance of the point from the equator, measured at the center of the earth. So, the point can be either 90° north or south of the equator. The letter N (North) or a S (South) is added to the degree value. Figuratively speaking, the earth is sliced into discs parallel to the equator. [Ord10]
The longitude coordinate specifies the east-west position of a point on the surface of the earth. It is the angular distance of the point from the arbitrarily but consistently chosen north-south-line called prime meridian. The standardized prime meridian extends through Greenwich, Great Britain. Points on the prime meridian have longitude zero. Points east of the prime meridian have a degree value between 0° East and 180° East, point west of the prime meridian a value between 0° West and 180° West. Again figuratively speaking, the earth is sliced into wedges like an apple. [Ord10]

Geographic coordinate system

In order to perform the mapping, the spacial area that corresponds to the video frame needs to be defined by the user. In the server's GUI, this can be done by adjusting a map view which is drawn congruently above the video frame view showing the model landscape. Thus pixel coordinates of the video frame can easily be mapped to the geographic coordinates of the point in the map with the same pixel coordinates.

The spacial area corresponding to the landscape model is defined using a map view.

To draw the map, the server is making use of a library called *GMap.NET* [rad]. *GMap.NET* is a powerful, open source .NET control which enables the usage of maps from various providers, like Google[6], Bing[7] or OpenStreetMap[8]. In this thesis, the maps from OpenStreetMap are used in order to avoid license problems, but changing the map provider can be done very easily and even a custom tile server can be used. With the help of the *GMap.NET* control, the map can be panned, zoomed and rotated.

For the map, the 3rd party library GMap.NET is used.

The class that encapsulates the map control is called *MapOverlayForm* and is shown in Figure 3.7. It is a WinForms form (see Section 3.6). It is not a control embedded in the main form, because in WinForms, there is no real transparency for controls – only for forms. Transparency is needed to blend the map above the webcam view. As shown in the class diagram, the *MapOverlayForm* class is implemented as a singleton. That

The *MapOverlayForm* encapsulates the map in a separate form.

[6] `maps.google.com/`
[7] `www.bing.com/maps/`
[8] `www.openstreetmap.org/`

Figure 3.8: Map area saved in a PDF file for later use. The openstreetmap map provider is used here. Map position, scale and orientation are encoded in a QR Code which is used by the Presence Simulator's auto calibration system.

is because there is no need for multiple map views and location source controllers can access the mapping functionality of the map more easily.

LSs are drawn as markers on top of the map.

Besides showing and adjusting map tiles, the GMap.NET control is also capable of managing layers for markers which are drawn on top of the map. The Map component uses this feature by drawing makers for each *LocationSource* at the geographic location that is stored in the *LocationSource*. The shape of the makers is defined in the *LocationSourceMapMarker* class, which is implementing the *LocationSourceObserver* interface as shown in the class diagram. Thus, each time a *LocationSource* changes its location or its name, the map marker is automatically updated.

An auto calibration mechanism helps to align the map view.

Adjusting the map view in such a way that it corresponds exactly to the physical model landscape captured by the webcam takes some time. Therefore, the Map component has an auto calibration mechanism implemented. The mechanism does not work with any model landscape, only with those built upon a map printed especially for this purpose.

The map has a QR Code on it that encodes the position and scale of the map.

Such a special map is shown in Figure 3.8. There is a Quick Response (QR) Code in the upper left corner of the map. QR Codes are two dimensional bar codes which can be used to store data. In the QR Codes used by the Presence Simulator, the spatial position and the scale of the maps are stored.

QR CODES:
QR Codes are a type of two-dimensional barcode that was first used in the automotive industry in the 1990s. The code consists of black squares arranged on a white background. Nowadays, QR Codes are popular in various fields such as commercials and mobile computing. [Heg10]
QR Codes are standardized by the ISO/IEC. There are six different versions of QR Codes which differ in size and the amount of data that can be stored in the code. The QR Codes used by the Presence Simulator are version three QR Codes. They consist of a 29x29 matrix of back and white squares, surrounded by a four squares thick white border that is called the quiet zone. [Heg10]
The matrix is divided into different areas. On the upper two corners and the lower left corner the cells of the matrix form larger black squares. These squares are called *positions*, because they are used to detect the position of the code. Between the *positions*, there are checked rows and columns which are used to define the timing. On the lower right corner, there is a smaller square like structure that is used to detect the alignment of the code. There are also some smaller areas for format information and version information. The rest of the matrix is used for the data. Data is stored using an error correction algorithm. Therefore, QR Codes can be read even when some parts of it are not visible or the image quality is not optimal. [Heg10]

QR Codes

The *MapWithQRCodeWriter* class is responsable for generating a PDF file that contains a picture of the map and the QR Code. The class is initialized with an image of the map currently shown by the *MapOverlayForm* class, the desired output paper format and the name of the destination file. The QR Code is generated using the .NET version of Google's *zXing* ("Zebra Crossing") library. [ZXi] *zXing* is an open source library to generate and detect different types of bar codes.

The *MapWithQRCodeWriter* generates a PDF file with a map and the respective QR Code in it.

In order to safe the spatial position, the latitude and longitude coordinates of the map point which is situated at the upper left *position* of the QR Code are used. To safe the map's zoom level, more effort is required, because the scaling of the map depends on the distance between the webcam and the printed map – which may be unknown at the time of generating the PDF file. That is why the distance in meters between the map points which are situated at the upper left and upper right *position* of the QR Code are used to encode the zoom level. The map is always saved north up, so rotation is saved implicitly.

The scaling is saved relatively, because the distance between webcam and printed map can vary. Rotation is saved implicitly.

The map image with the QR Code is embedded into a PDF file using the *PDFSharp* library [emp09]. The PDF format is used, because it is better suited for files that are to be printed out than a normal image format.

The PDFSharp library is used to generate the PDF file.

The *MapWithQRCodeReader* class is responsible for the actual auto calibration. When the auto calibration is started by the user, the current

The auto-calibration used the QR Code.

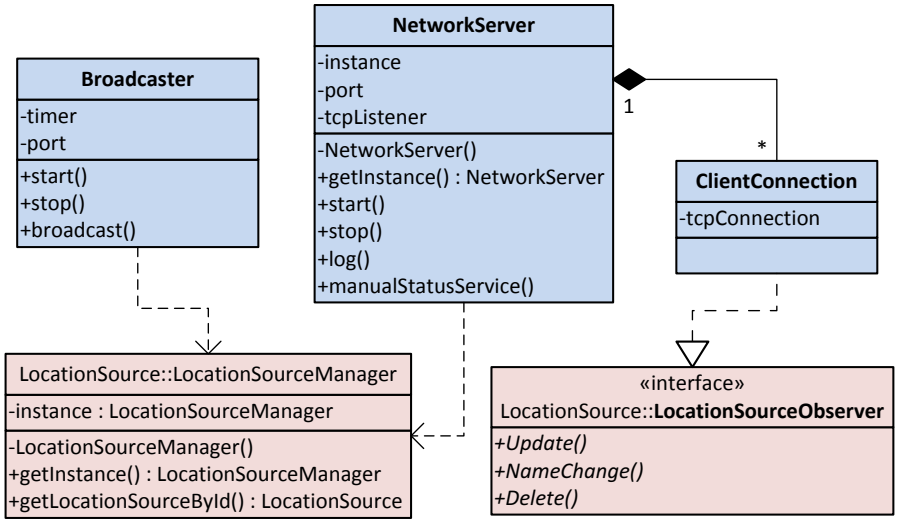


Figure 3.9: UML class diagram of the server's Network component. The violet colored classes are dependencies from the Location Source component.

webcam frame is analyzed with the help of the *zXing* QR Code detector. When the detector was able to decoded a QR Code, the position of the map in the *MapOverlayForm* singleton is set to the latitude and longitude values that were stored in the QR Code. In order to set the zoom level, the *MapWithQRCodeReader* takes the two geographic points of the *MapOverlayForm's* map, that correspond to the upper two *position* points of the QR Code. The zoom level is then set so that the distance between the two geographic points equals the distance stored in the QR Code. The rotation of the *MapOverlayForm* is set so that the line defined by the two geographic points is parallel to the map's upper border.

3.5.4 Network

The Network component sends location updates to the clients.

The Network component is primarily responsible for transferring the changing latitude and longitude coordinates of *LocationSources* to a client software running on Android smartphones. The client software that receives the updates and mocks the device's location is discussed in Section 3.8.2.

Because of the limited capacities of smartphones, the location updates can not be broadcasted.

My first idea on how to implement the Network component was to simply broadcast all location source updates in the local network. More than one location source might be available at the same time, so additionally to the location update, the broadcast message would also contain the location source's ID. However, when there are many location sources available and the update frequency is high, the client might get flooded with messages – which is a problem, especially since the hardware of smartphones is not that capable. That is why broadcasting the updates is not possible.

To avoid the flooding problem, the Network component has to ensure that the client only receives location updates for one location source, namely the one the client is registered to. The network component achieves this by handling a Transmission Control Protocol (TCP) connection to each client separately.

Each smartphone is connected via a TCP connection.

TCP is a widely spread network protocol used by a lot of network applications. TCP provides a reliable and ordered delivery of data in form of a stream of packages. TCP is connection oriented. That means that before transferring data between two TCP endpoints, a connection needs to be established and after the transfer the connection needs to be closed. That is exactly what is needed to avoid the flooding problem. Endpoints of TCP connections are defined by the Internet Protocol (IP) address of the device the endpoint is running on and a port number.

TCP provides a reliable and ordered delivery of packages by managing connection.

Another option would have been the usage of the User Datagram Protocol (UDP), which would have needed the implementation of some sort of subscribing and unsubscriping mechanism. That would have been costlier to implement than the TCP approach.

A connectionless UDP transfer would be costlier.

Figure 3.9 shows a class diagram of the Network component. The *NetworkServer* class implements a TCP listener. A TCP listener is a thread that listens on a specified port for incoming connections. When a client connects to the listener, the listener starts a new connection and passes it to a connection handler. The connection handler is also running in its own thread. The *NetworkServer* is also responsible for logging and shutting down active connections when the Presence Simulator application is closed by the user. The class is implemented as a singleton.

The *NetworkServer* is implementing a TCP listener that passes incoming connections to a handler.

The connection handler is implemented in the *ClientConnection* class. The class implements the *LocationSourceObserver* interface (see Section 3.5.1). So, each time the latitude and longitude coordinates of the *LocationSource* that the *ClientConnection* is subscribed to are changed, the *ClientConnection* is informed.

The handler delivers location updates to the client.

But how does the *ClientConnection* know to which location source it should subscribe to? When the TCP connection is passed from the TCP listener to the *ClientConnection*, the *ClientConnection* first waits for the client to send a message containing a serialized version of the location source the client has selected. The *ClientConnection* then subscribes itself as a *LocationSourceObserver* to the selected location source. After this message is received, the *ClientConnection* is ready to send location updates.

On connection initialization the client sends the ID of the LS he wants to subscribe to to the server.

Location updates are serialized in form of a GPS Exchange Format (GPX) file. GPX is based on XML. It is a open format that is used by many programs to exchange geographic data [Top]. Below, a sample GPX file is shown.

Location updates are encoded in a GPX file.

```
1  <?xml version="1.0" encoding="UTF-8"
2    standalone="no" ?>
3  <gpx version="1.1" creator="Presence Simulator">
4    <trk>
5      <trkseg>
6        <trkpt lat="50.7340106171797" lon="
             7.10656642913818" />
7        <trkpt lat="50.7340106171650" lon="
             7.10643768310547" />
8        <!-- [...]-->
9      </trkseg>
10   </trk>
11 </gpx>
```

For each location update, a *trkpt* tag with the coordinates is sent.

The first five lines of the GPX file are sent right after the client's initial message is received. Lines similar to the sixth and seventh line are sent whenever the position of the location source has been changed by a location source controller. When the connection to the client is closed – e. g. when the Presence Simulator's server is closed by the user – the last three lines are sent and the client is thereby informed that the TCP connection is about to be closed.

To avoid flooding, the update frequency is limited.

Sometimes, the geographic position of location sources is updated in a very high frequency. The client might not benefit from this. In fact, the client might get flooded with updates. Therefore, the update frequency can be limited in the *ClientConnection* class. By default one update is sent per second. When the location sources are updated in a higher frequency, some updates are dropped. , An important question has not been discussed yet: How does the client know which location sources are available on the server? Let's first look at the more convenient way to transfer this knowledge from the server to the client.

To select a location source, the client needs to know which ones are available.

The *StatusBroadcaster* periodically broadcasts a list of location sources.

The *StatusBroadcaster* class periodically (by default every five seconds) broadcasts an XML file with a list of currently available location sources. In addition to the list of location sources, the IP address and the port of the TCP connection listener are also broadcasted. Thus, it is not necessary to enter this data on the client's device manually. The client simply listens for broadcast messages and thereby gets informed about all available location sources and how to connect to them.

Some Android devices cannot receive broadcast messages.

Unfortunately, not all Android devices are able to receive broadcast messages. In theory, they should all be capable of this, but as various people in Google's Android developer group have reported[9], at least some devices running on Android version 2.2 and lower might lack this feature.

The list of LSs can be fetched alternatively via a network service.

When an Android device is not able to receive broadcasted status messages, it is possible to fetch the status message manually, by connecting to a special service offered by the *NetworkServer* class. The service is

[9] `groups.google.com/group/Android-developers`

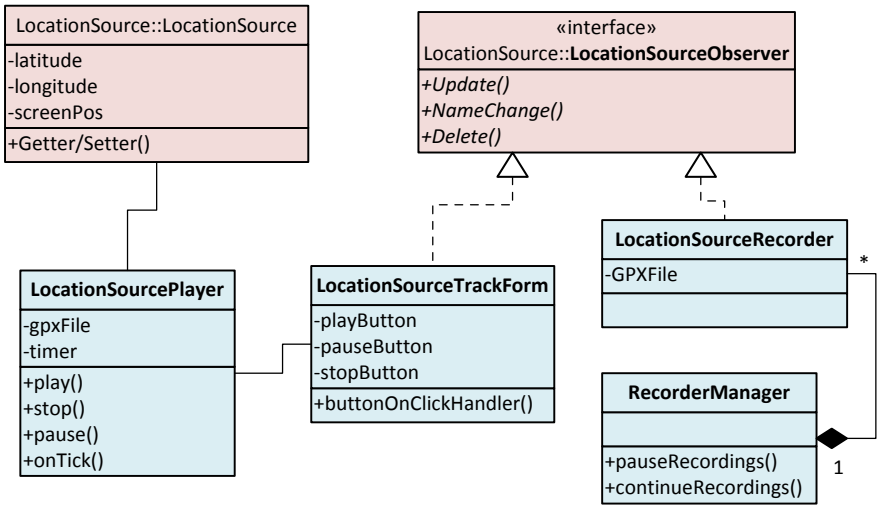


Figure 3.10: UML class diagram of the server's Recorder component. The violet colored classes are dependencies from the Location Source component.

implemented as a TCP listener. When a clients connects to the service, a status message equal to the one broadcasted by the *StatusBroadcaster* is sent to the client and afterwards the connection is closed. Of course, in order to use the service, the user needs to know the IP address and the port of it and has to enter the data in the client's GUI.

3.5.5 Recorder

The Recorder component of the server is responsible for recording tracks of location sources as well as replaying them. A *LocationSources* controller is implemented, because when a track is replayed, the latitude and longitude coordinates of *LocationSources* are set. For the track recording, a *LocationSource* view is implemented.

The Recorder component records LS tracks and replays them.

By default, the tracks of all location sources are recorded. When a *LocationSource* class is created, the *LocationSourceRecorder* class is instantiated, too. The *LocationSourceRecorder* class implements the *LocationSourceObserer* interface and thus objects of the class are informed whenever the *LocationSource* object they are subscribed to is updated. Updates can then be saved as track points.

The *LocationSourceRecorder* is implemented as an LS observer.

Like in the *ClientConnection* class of the Network component, the track is encoded in a GPX XML file (see Section 3.5.4). But in contrast to the *ClientConnection's* GPX file, timestamps of track points and the ID and name of the location source are encoded as well. The GPX files are saved on the hard disc in the "tracks" sub-folder of the Presence Simulator's server application. When the user changes the name of a location source at some point during the recording process, this is saved too, as shown in the following GPX file in line 12. Names and IDs are

Tracks are encoded as GPX files and stored on the hard disc drive.

saved as XML comments within the GPX file, because encoding them as XML tags would have contradicted the GPX XML schema [Top].

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--locationSourceId:
    9c73d3cf-2e58-4067-86a2-d5f669780229-->
<!--locationSourceName: Clark Kent-->
<gpx version="1.1" creator="PresenceSimulator">
  <trk>
    <trkseg>
      <trkpt lat="50.7340106171797"
             lng="7.10656642913818">
        <time>634794907699836396</time>
      </trkpt>
      <!-- locationSourceName: Superman -->
      <trkpt lat="50.7340106171797"
             lng="7.10643768310547">
        <time>634794907710386999</time>
      </trkpt>
      <!-- [...] -->
    </trkseq>
  </trk>
</gpx>
```

The timestamps of single track points are encoded as ticks.

Timestamps are encoded as ticks. A single tick represents 100 nanoseconds. The value saved as a time stamp is the number of ticks that have elapsed since 00:00:00, January 1, 0001. Because the update frequency of the location sources can be quite high and this accuracy is usually not needed, some updates are dropped. By default, only one update per second is written to the XML file.

Recording can be paused with the help of the *RecorderManager*.

The user might want to pause the track recording for example during the setting up of the map overlay. Therefore, the *RecorderManager* class, which is implemented as a singleton keeps track of the different instances of the *LocationSourceRecorder* class. When the user pauses the recording, the manager sets all recorders on pause. Recording is continued when the user uncheckes the pause function in the server's GUI.

The *LocationSourcePlayer* replays saved GPX files, by making use of XPath.

Instances of the *LocationSourcePlayer* class are able to replay a saved GPX file, by controlling a *LocationSource* object. The player can change the object's name property and latitude and longitude coordinates accordantly to the data encoded in the file. The GPX file is loaded as an *XPathDocument* object with the help of the XML tools provided by the .NET framework.

The pace of the replay is similar to the pace while recording. A timer is used to manage the durations.

XPath is an XML query language developed by the W3-Consortium [BBC+10]. With XPath, it is possible to navigate through a XML Domain Object Model (DOM) by selecting nodes which fulfill specific criteria. The *LocationSourcesPlayer* selects all "trkpt" (track point) nodes ordered by their occurrence in the document. After all the nodes are selected, a timer is started. When the interval managed by the timer is elapsed, an event is fired. The interval is defined by the duration between the timestamps of two subsequent track point nodes.

When the timer fires the event, the next track point node is pulled from the list and the timer's interval is updated to the duration between the current track point and the next one. Then, the latitude and longitude properties of the *LocationSource* object the player is controlling are set accordantly to the values stored in the track point node.

When a new track point is read, the LS is updated and the timer is reset.

There is a special GUI control for *LocationSource* objects controlled by a *LocationSourcePlayer*. The control allows the user to start, stop and pause the playback.

There is a GUI element for *LocationSourcePlayers*.

Of course, *LocationSource* objects controlled by a *LocationSourcePlayer* also have a discriminator. To make sure that these location sources are not controlled by a *LocationSourceDetector*, a dummy discriminator which always returns false is used.

A dummy discriminator avoids interference with LS detectors.

3.6 Server User Interface Design

The server's user interface is implemented with the help of the Windows Forms (WinForms) graphical application programming interface (API) which is part of the .NET framework [MSD]. WinForms basically wraps the Windows API in managed .NET code and thus provides access to native Microsoft Windows interface elements like text boxes, combo boxes or windows – which are called forms in WinForms. The WinForms interface has also been implemented in Mono, so making use of it is not restricted to Mircosoft Windows.

The server's GUI is based on WinForms.

Since .NET 3.0, there is an alternative API for rendering GUIs, called Windows Presentation Foundation (WPF). WPF is using an XML based language for defining GUIs, like it is also done in Android. This way, a greater separation between the design of the GUI and the actual program code is enforced. However, some of the third party libraries the Presence Simulator uses are not offering WPF support and therefore, the Presence Simulator sticks to the older WinForms.

WPF is a modern alternative to WinForms, but it is not supported by all of the 3rd party libraries the server is using.

Most of the actions provided in the Presence Simulator's GUI are implemented as commands. The command pattern encapsulates a single action in a command object. As a result, there is a decoupling between the action and the action invoker, allowing other objects to manipulate an action or the combination of several commands to construct a more complex action [GHJV95]. The overhead introduced by the command pattern is rather large, but it is by far outweighed by the gained flexibility.

Most of the server's GUI actions are implemented as commands.

The server's GUI design is inspired by the Ribbon GUI interface introduced in Microsoft Office 2007. In a Ribbon GUI, the main interface of a program consists of multiple toolbars (ribbons) which are placed on tabs in a tab bar. This is a well known design pattern, already used for programs in the 1990s. Nevertheless, Microsoft is currently attempting to patent the ribbons interface [HBMS04] and has already set up a web

The GUI design is inspired by the Ribbon GUI.

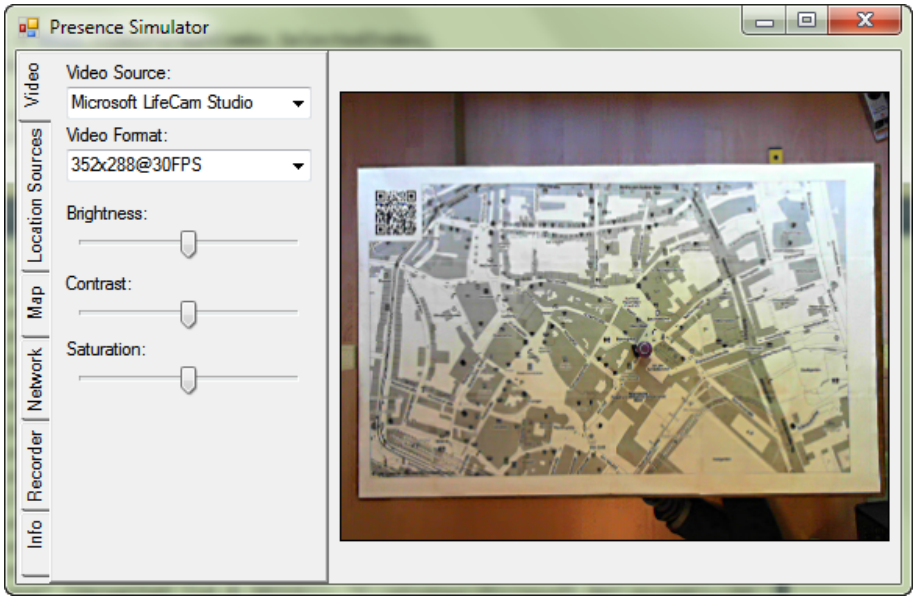


Figure 3.11: Video settings GUI of the Presence Simulator for controlling the webcam device. On the right side, the video stream of the selected webcam is shown.

page[10] for acquiring licenses for it. Therefore, I decided to make no use of Microsoft's ribbons library for the .NET framework. Instead, I used the controls provided by the WinForms API to built a ribbon like UI. I implemented the toolbars as tabs in a tab view on the left side of the application's window.

The video settings on the left side of the GUI allows users to select a webcam device. On the right side of the GUI, the video stream is shown.

Figure 3.11 shows the window that is shown when the user starts the Presence Simulator. As mentioned, there is a ribbon like menu structure on the left side of the window. On the right side, there is a larger webcam view. The list of available webcams and their resolution capacities is fetched with the help of the AForge library. By default, the webcam view gets its video stream from the first available webcam, but of course, the user can change this in the drop down lists shown in the video settings menu. When the user changes the webcam resolution, the window size is adjusted automatically, so that it is always large enough for the webcam view.

The brightness, contrast and saturation of the video stream can be adjusted.

The three sliders in the video settings menu right below the video device settings can be used to modify brightness, contrast and saturation of the video stream. This feature is implemented with the help of AForge image filters. The filters are applied on all video frames before they are used by the location source detector component. Therefore, by tweaking the settings, detection quality can be increased. However, the filters lower the video frame rate and are therefore deactivated when the sliders are in the middle position – which is the default position.

[10] msdn.microsoft.com/en-us/office/aa973809.aspx

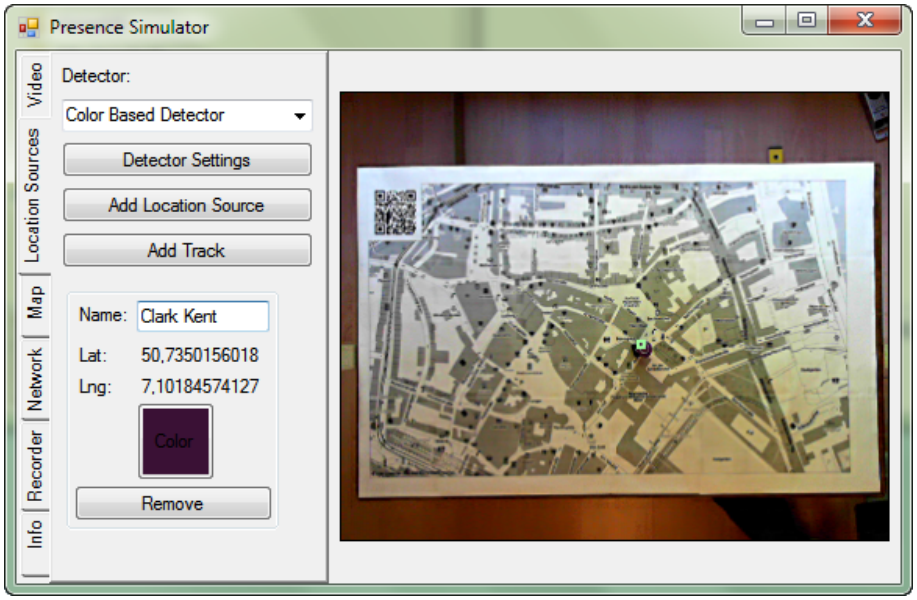


Figure 3.12: GUI for managing Location Sources of the Presence Simulator. In the webcam view on the right, a detected LSF is highlighted.

The LS settings enable users to select a location source detector.

In Figure 3.12 the location source menu tab is activated. The drop down list at the top of the menu enables the user to select the location source detector that is used by the server. In the current version, users can select either the color based detector or the marker based detector. By default, the color based detector is selected. As mentioned in the discussion of the server's Detector component (see Section 3.5.2), detectors need to have a detector settings form. This form is shown when the user clicks on the "Detector Settings" button.

LSs can be added or removed and saved tracks can be replayed.

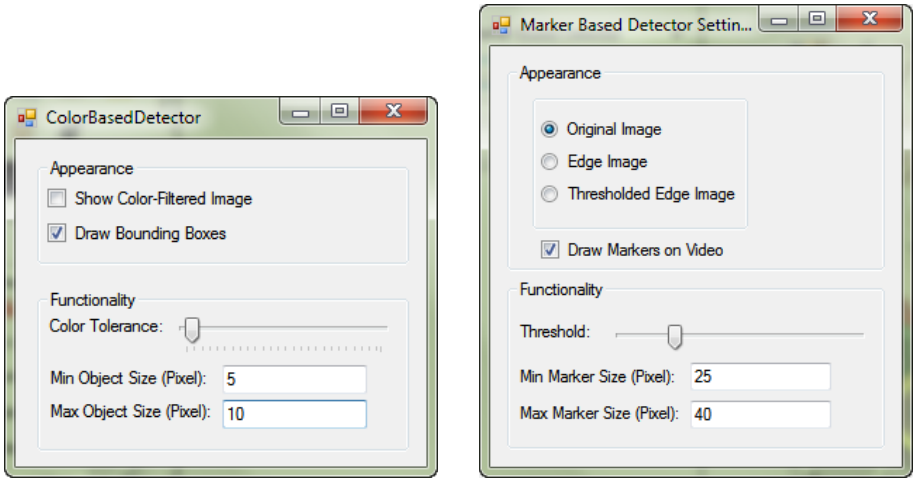
The two buttons below the "Detector Settings" button are for adding location sources that are either controlled by the selected detector ("Add Location Source") or by a *LocationSourcePlayer* ("Add track"). When the user clicks on the "Add track" button, a file selection dialog is opened, allowing the user to select the track of the location source he or she wants to replay. Below the buttons is a scroll view, where views of location sources are added automatically. The scroll view in Figure 3.12 shows a *LocationSourceFormView* with a *ColorDiscriminator* control.

The *ColorBasedDetector* can visualize detected LSFs.

In the video stream view on the right side of Figure 3.12, a tiny green square is visible. This square is drawn on the video frames by the *ColorBasedDetector* to indicate that an LSF has been detected at this position. *LocationSourceMapMarkers* are not visible in the figure, because they are drawn on the map view, which is deactivated in the figure.

Both detectors have their own setting form, allowing users to tweak the detection quality.

Figure 3.13 shows the setting forms of the color based detector and the marker based detector. In both forms, there are settings that only affect appearance and settings that may increase or decrease detection quality. Appearance settings affect how the video frames are drawn in the GUI, they do not influence detection quality. For example, the user can



(a) *ColorBasedDetector* settings form **(b)** *MarkerBasedDetector* settings form

Figure 3.13: Presence Simulator GUI forms for controlling visual and functional settings of LSF detectors.

select to show only video frames filtered by the sobel filter, instead of the unmodified ones, or the detected location sources can be visualized with green squares directly in the video frame. Functionality settings on the other hand do influence detection quality. In the settings form of the *ColorBasedDetector*, the color tolerance can be set. In the settings form of the *MarkerBasedDetector*, the threshold of the binarizer can be set and in both setting forms the maximum and minimum size of the LSFs can be specified in pixels.

The map overlay is a separate form with adjustable opacity.

Figure 3.14 shows the map menu and the video stream view. In the figure, the map overlay is activated. The map overlay is actually a separate form that is positioned on top of the video stream view. Whenever the user is moving the main window or the video resolution is changed, the map overlay is repositioned and resized automatically. Additionally, the map overlay is not shown as a separate form in the Windows task bar. For the user, it looks like the form is part of the main form. The reason for implementing the map like this is that the map needs real and adjustable transparency so that it can be fitted to the model landscape shown in the underlying video view. The WinForm API does not support real transparency for controls, but forms can be drawn transparently. The opacity of the map can be adjusted by the user with the opacity slider in the map menu.

The map view can be zoomed, rotated and panned, so it can be aligned congruently to the physical model shown in the video view.

The map view can be zoomed and rotated with the zoom and the bearing slider. Due to a bug in the *Gmap.NET* library, there is no smooth zooming. Only one of the 18 zoom levels provided by *OpenStreetMap* can be selected. When the mouse courser is on the map view, the map can also be zoomed using the mouse wheel. The location of the map can be changed with drag and drop mouse gestures on the map view. When the user wants to calibrate the system, he or she has to use these

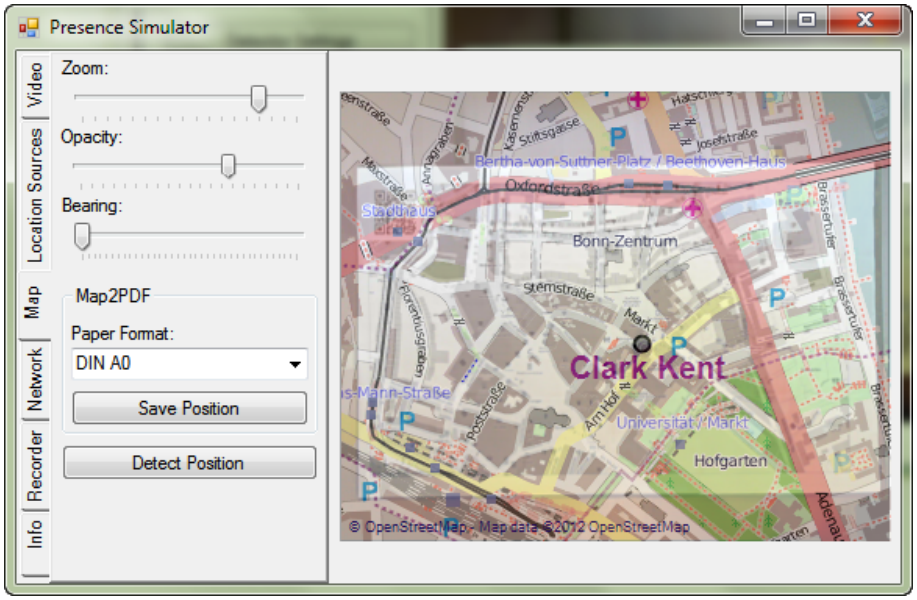


Figure 3.14: Presence Simulator GUI with the map settings and tools for the auto calibration system on the left side, and a semi-opaque map view on the right. Underneath the map view, the webcam's video stream view is visible.

settings to align the map view congruently to the physical landscape model shown in the video view.

The auto calibration speeds up the alignment of the map view.

Sometimes, the calibration may take some time and therefore the auto calibration mechanism described in Section 3.5.3 can be used by clicking the "Detect Position" button. Of course, in order to successfully calibrate, the video view needs to capture a map with a QR Code printed on it. When the user wants to save the current position as a map with a QR Code, he or she can selected a paper format in the drop down menu shown on the left of Figure 3.14 and click the "Save Position" button. A file selection dialog will then be shown and the map that is currently drawn by the map view will be saved with a QR Code on it in the selected PDF file.

Markers of location sources are drawn on the map.

There is a *LocationSourceMarker* drawn in the marker overlay of the map view shown in Figure 3.14. As mentioned in Section 3.5.3, the markers are location source observers. Therefore, their position and name is updated automatically whenever the location source is changed by a location source controller.

The network menu shows a log and enables the user to turn off the network functionality.

In Figure 3.15, the network menu is shown. There are two check boxes on top of the menu. With the first check box, the whole network functionality can be turned off. Without the network functionality, Android clients are not able to receive status messages and cannot connect to the location source update streams. With the second check box, only the broadcasting of status messages can be turned of. The largest area of the network menu is used by a text view with a scrollbar. The text

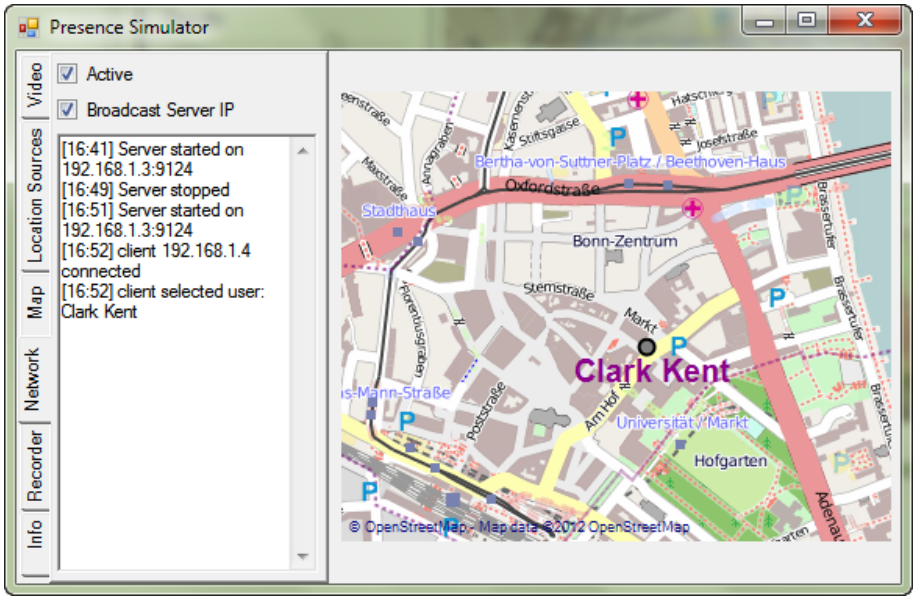


Figure 3.15: Presence Simulator GUI with activated network settings on the left and a fully opaque map view on the right side.

box is used for logging. For example, when an Android client connects to the server, this is logged in the text view. Additionally, the figure is showing a map view with 100% opacity, so the video view is completely hidden by the map.

The recorder menu enables users to turn of track recording.

The recorder and the info menu tab are not depicted because there is actually not much to show. In the recorder menu, there is only one check box for setting the track recording functionality on pause. The info menu tab shows a credits list and a button to reset all setting values to default.

The application settings are stored in an XML file.

The application settings are automatically stored in a settings XML file by using the application settings manager feature of Visual Studio. With the help of this feature, the settings are persistent even when the user restarts the application. Setting files are similar to property files in Java, but they are much better integrated in the IDE. For example, in Visual Studio it is possible to link the value of a GUI element with a property within three mouse clicks.

3.7 Android Client

This section describes the Android client that mocks the device's location.

In this section, the Android client of the Presence Simulator is described. The client enables users to subscribe to an update stream of a selectable location source. The corresponding server component which provides the update stream is described in Section 3.5.4. Whenever an update is received, the client sets the Android system's values for the geographic location accordantly.

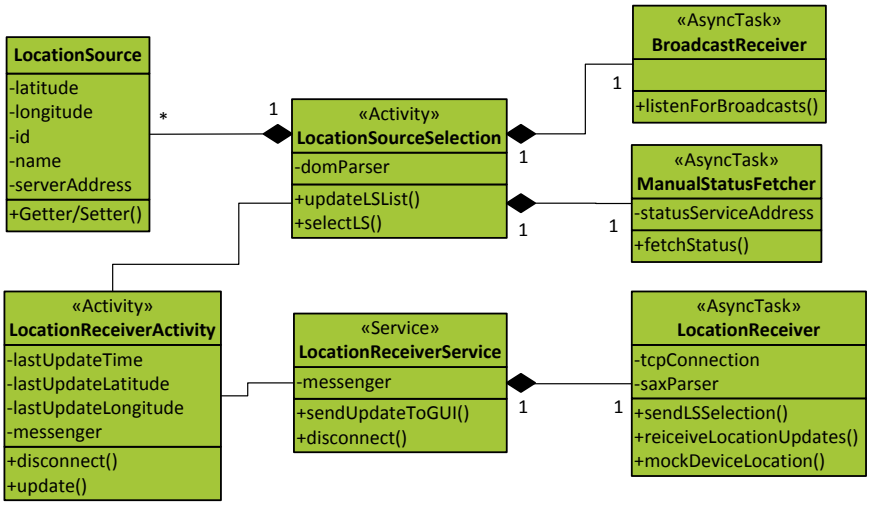


Figure 3.16: UML class diagram of the Android client. The application starts with the *LocationSourceSelection Activity*.

The architectural design of the client is shown in the Figure 3.16. The main starting point of the application is the *LocationSourceSelection Activity* class which is shown to the user right after the application is loaded. An *Activity* in Android is like a window in full screen mode on a desktop computer, but unlike it, an Android *Activity* should focus on only one thing at a time, because of the limited screen size of mobile devices [Gooa]. The *LocationSourceSelection* enables the user to select a location source. Figure 3.17a shows the *Activity*. There is a list view in the upper part of the *Activity* and each row of the list shows the name and the server address of a location source. The user can register to the location updates of a location source by taping on the corespondent row.

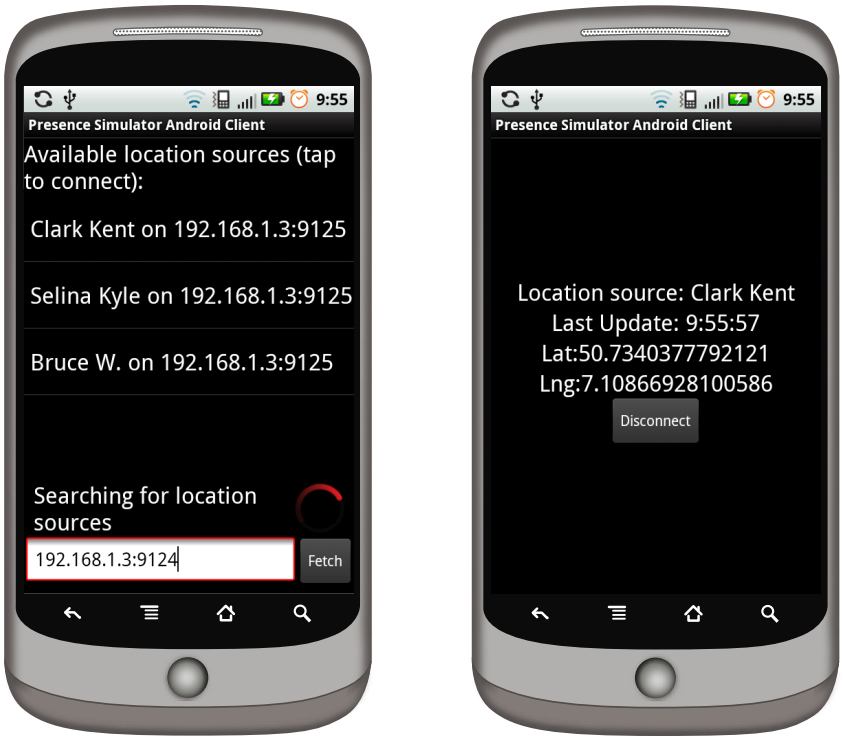
The *LocationSourceSelection Activity* of the client enables users to select an LS.

In order to provide the location source list to the user, the client needs to fetch the list of available location sources from the Presence Simulator's server. As described in the Network component of the server (see Section 3.5.4), there are two ways to fetch available location sources. The more convenient one is to listen to the status messages the server is broadcasting. The more robust one is to connect to the status service offered by the server. For both ways, an *AsyncTask* has been implemented in the client. The abstract *AsyncTask* class is provided by the Android framework and enables the easy use of threads. An *AsyncTask* basically encapsulates a thread. Tasks running for a long period of time – e. g. the download of data from a server via a network connection – should be implemented as *AsyncTasks*, so that they do not block the user interface thread [Gooa].

The list of LSs is fetched by the client with *AsyncTasks*.

The *BroadcastReceiver* class is responsible for listening to broadcast messages. It is an *AsyncTask* and is started in the constructor of the *LocationSourceSelection Activity*. A status message broadcasted by the server consists of an XML file containing a list of serialized location sources

The *BroadcastReceiver* receives the server's broadcast status messages.



(a) GUI of the LocationSelection Activity, showing a list of available LSs.

(b) GUI of the LocationReceiver Activity, showing the latest location update received for the selected LS.

Figure 3.17: The Android client's GUI.

and also the address of the location source update service. Once a status message is received, the *LocationSourceSelection* activity is informed by the *AsyncTask* so that it can update the user interface and fill the list view with the new information. A domain object model (DOM) parser is used to decode the data stored in the XML file. The XML DOM provides methods to traverse an XML tree easily and access, insert or delete nodes at a specific location. The DOM parser needs full access to the whole XML file before the DOM can be used. The Android DOM parser is provided in the W3C DOM package of the platform [Goob].

The *ManualStatusFetcher* can pull status messages from the server without listening to broadcasts.

The *ManualStatusFetcher* class enables the user to connect manually to the status service offered by the server. It is launched by the *LocationSourceSelection Activity* once the user has entered the address of the server's status service and tapped the "Fetch" button, which is shown in Figure 3.17a at the bottom. When the *ManualStatusFetcher* has successfully received the status message and the connection has been closed by the server, the message is passed to the *LocationSourceSelection Activity* and processed in the same way as a broadcasted status message.

The client's list of LSs is continuously synchronized.

The *LocationSourceSelection Activity* manages a list of known location sources. When a new status message is received, the data in the list is updated. So when a location source is renamed, deleted or newly created on the server, the location source list of the client is synchronized.

It is also possible to have location sources from multiple servers. This might be an interesting option when evaluating an application with several teams in multiple rooms. After each update, a small message is shown to the user, describing what was updated. The Android framework is providing a method for displaying messages called "toasts" [Gooa].

When the user has selected a LS, the *LocationReceiverActivity* is responsible for receiving location updates.

After the user has selected a location source, the *LocationSourceSelection Activity* launches the *LocationReceiverActivity Activity*. The Android framework then automatically places the launched *Activity* in the foreground. The *LocationReceiverActivity Activity* displays the latest location update received for the selected location source, as shown in Figure 3.17b. In order to continuously receive the update stream, a TCP connection to the update service of the server is established.

The TCP connection to the server is encapsulated in a background service.

When the user is switching to another application – e. g. the application that is evaluated – the client application is paused by the system. Usually, on pause no updates can be received. Even worse, when the device gets low on memory, the system might stop the client completely. To avoid these problems, the connection to the server is established within a *Service* and not in the *Activity*. A *Service* is a "facility for the application to tell the system about something it wants to be doing in the background (even when the user is not directly interacting with the application)" [Gooa]. However, a *Service* is not a process and also not a thread. Therefore, the *LocationReceiverService* launches right after the start an *AsyncTask* named *LocationReceiver* that maintains the connection to prevent the blocking of the *Service's* thread by the connection.

The location update stream is decoded with a SAX parser.

The *LocationReceiver* maintains the TCP connection to the server. At the beginning of the connection, the ID of the selected location source, encoded in a small XML file, is sent by the client, so the server can answer with the correct update stream. The location updates are encoded by the server in a GPX XML file, as described in Section 3.5.4. The GPX file received by the client is parsed with the help of the Simple API for XML (SAX). In contrast to the DOM parser used by the *LocationSourceSelection Activity* the SAX parser is a sequential access parser. The DOM parser needs to know the whole XML document while parsing, whereas the SAX parsers operates on each XML piece sequentially. After a track point tag has been decoded, an event is fired by the parser.

When a location update is received, the GUI is updated and the device's location is mocked.

When such an event is fired, the *LocationReceiver* updates the geographic location stored in the device's operation system. This is done by calling *setTestProviderLocation* method of the Android platform's *LocationManager* with a *Location* object corresponding to the received location as a parameter. The Presence Simulator's client fakes a GPS receiver. In order to have the Android OS accept the faked location, the user has to enable "mock locations" in the debug settings of the OS.

Communication is done with messages.

The *LocationReceiverActivity* needs to be informed by the *LocationReceiverService* when an update is received, so that the GUI can be updated.

The communication between *Service* and *Activity* is done by using a *Messenger*. The *Messenger* class provided by the Android framework allows message-based communication between different processes.

3.8 System Integration

Discussion on how the non-functional requirements are met by the implementation.

In this section, it is discussed to what extend the implementation of the Presence Simulator meets the requirements listed in Section 3.1. As described in the last section in detail, the system meets all the functional requirements by design. Therefore, only the non-functional requirement are discussed in the following list.

1. Whether the system's *reaction time* is fast enough was evaluated in the performance measurements discussed in Section 3.8.2. It turned out that the system is so fast that there is only a small delay between the movement of an LSFs and the location update by the client, even when it is running on older hardware.
2. The *spatial resolution* depends on the selected video resolution of the webcam. The higher the resolution of the webcam, the higher the spatial resolution. For most cases, a resolution of 960x544 should be sufficient.
3. When the map view is calibrated precisely, the *simulated presence* is accurate. Due to a bug in the GMap.NET library (see Section 3.6) smooth zooming is currently not supported. Therefore, it might be necessary to calibrate the hight of the webcam manually.
4. The *update rate* was measured as described in Section 3.8.2. It turned out that the update rate can easily be higher than one update per second.
5. The *size of the model landscape* is only limited by the viewing range of the webcam and the distance between webcam and model landscape. A model landscape of the size of one by two meters is no problem.
6. *LSFs of the size* of playmobile® figures can be tracked reliably by the system as long as the detectors are adjusted correctly and the webcam's resolution is set to at least 800x448 pixels (see Section 3.8.2).
7. No usability tests were made, so it cannot be stated whether the system is *convenient* to use or not.

3.8.1 Software Testing

Unit and integration tests ensure software quality.

In order to increase the software quality and reduce the amount of bugs in the Presence Simulator, several unit and integration tests were written. Unit tests test single classes, whereas integration tests test the interaction of components.

Examples of unit tests are the tests of the location source detector classes, in which the LSF detection method is called by passing a single video frame image. The image passed to the method shows one LSF with known coordinates and the test verifies whether the detected coordinates match the known ones.

Detectors are tested with unit tests.

An example of an integration test is the test of the location update service implemented in the *NetworkServer* class, in which a *LocationSource* object is instantiated and added to the *LocationSourceManager*. The *NetworkServer* is started and a TCP connection to the location update service is established, by sending the ID of the constructed *LocationSource*. To test whether the update service is working properly, the coordinates of the *LocationSource* object are changed and it is verified that the update is transfered over the TCP connection.

The test of the location update service is an integration test.

3.8.2 Performance Evaluation

In order to determine whether the Presence Simulator's reaction time is fast enough, so that study participants are not distract by a delay between their movement of the LSF and the smartphone's response in changing the system location values, some performance measurements were carried out. The detailed results of each measurement can be found in Appendix A.

The measurements ensure that the system operates fast enough.

In the measurements, the received location updates per second were measured by counting the number of updates within a 20 second period received at the Android client. For this purpose, the update brake for avoiding flooding of the client was deactivated. During the 20 second period one – respectively two – LSF(s) was (were) moved from one side of the webcam's view field to the other. Five different webcam video resolutions were tested. The server software was running on an Intel Pentium dual core T4500 with 2.30GHz – which is at the time of writing a low end device. The used webcam was able to capture with at least 30 frames per second (FPS) on resolutions below 800x448 pixel, with at least 15 FPS on resolutions up to 960x544 pixel and with at least 10 FPS on resolutions above that. All measurements were performed three times in a row. The average values are shown in Figure 3.18.

The measurement setup is described.

The marker based detector was not able to detect markers at the lowest resolution. At a video resolution of 960x544, there was a small but noticeable delay (less than one second) between the users action and the received updates. At a video resolution of 1280x720, the delay was disturbing (more than two seconds). Therefore, higher resolutions were not tested. I assume, a faster processor would enable the usage of higher resolutions with a smaller delay.

There is a delay on higher resolutions.

On resolutions below 800x448, detection quality was not optimal: The color based detector had some false positive detections and the marker based detector some smaller dropouts. Therefore, a resolution of

800x448 or 960x544 pixels are recommended.

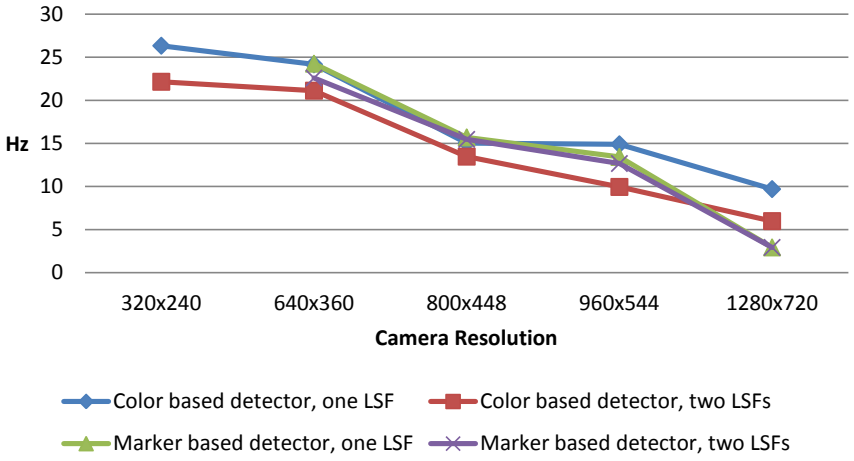


Figure 3.18: Average received location updates per second at the Android client.

800x448 or – when a small delay is tolerable – 960x544 pixels is recommended.

The marker based detector does not suffer much from increasing the number of LSFs.

There is almost not difference between the results of the marker based detector measurements with one LSF and with two LSFs. The color based detector however is losing performance. That is because the color based detector applies the color filters and the other operations once for each LSF, whereas the marker based detector applies its operations only once per frame.

Performance was acceptable.

Overall, the performance was acceptable. Even on the older hardware used for the measurements, the performance was good enough to allow a smooth and reliable operation.

Chapter 4

ShinyNavi - A Comparative Study

In order to determine whether a Presence Simulator based study can produce results similar to the ones from a corresponding field study, I carried out a comparative study with the help of Pascal Bihler and Orhan Sönmez. Sönmez is currently working on his bachelor thesis about location-aware task reminder tools. For his work, he needed to determine whether a location-aware reminder is superior – in terms of the number of forgotten tasks – to a simple to-do-list, with the help of a research study.

A comparative study based on a study for Sönmez's bachelor thesis was conducted.

The comparative study in this thesis is piggybacked on Sönmez's study, by conducting it twice: Once in a field environment and once in a lab environment using the Presence Simulator. The study design and the results of the studies are compared in this chapter.

Sönmez's study was conducted in a field and in a lab environment.

Because the participants' distraction is a critical factor in Sönmez's study, they were not openly asked to evaluate a location-aware reminder tool. Instead they had to evaluate a navigation system for pedestrians guiding the participants through a predefined route and were given a list of tasks to fulfill along the route to make the experience more realistic.

The location-aware reminder was integrated in a navigation system.

The navigation system developed for the study is called ShinyNavi and is presented in Section 4.2. It is remotely controlled by a human and therefore is not a navigation system ready to use for real navigation purposes.

The navigation system ShinyNavi is remotely controlled.

4.1 Study Design

24 participants attended the comparative study, nine of which were female and 15 male. The youngest participant was 11 years old and the oldest 75 years. There were participants with no smart phone know-

Demographics of the participants.

ledge at all as well as average and experienced smart phone users. None of the participants had any experience with the Presence Simulator.

A between-subjects study design was used. Half of the participants attended to a field study, the others used the PS.

To avoid carryover effects, a between-participants study design was used. The participants were split into two equal groups, each with similar demographic characteristics. The first group – named P – attended the field study, whereas the second group – named PS – attended a similar study that took place in a lab environment with the help of the Presence Simulator.

Both groups carried out Sönmez's study.

Both groups carried out Sönmez's study and were therefore split again into two equal subgroups with similar demographic characteristics: P-T, P-L, PS-T and PS-, T being the group with nothing but the to-do-list to help them, and L being the group with the location-aware reminder. All participants had to accomplish 10 tasks distributed along a predefined, circular track of 1.74 kilometers through the inner city of Bonn. The participants were guided by the ShinyNavi application.

A list of the tasks that had to be accomplished.

The tasks could only be accomplished at certain places along the track. Here is a list of the ten tasks:

1. Count the number of windows of the university's main building.
2. Please pick up a *UNICUM* magazine from the University.
3. What is the name of the kebab shop on *Strockenstraße*?
4. On weekdays after 9:00 pm, when do the buses run from the bus stop *Bonn Markt*.
5. How much is a small *Americano* coffee at *Starbucks*?
6. At the post office on *Münsterplatz*, when is the mailbox emptied on Saturdays?
7. Please pick up a pamphlet from the tourist information of the city.
8. At a ticket machine for local transport, please look up the price level for a trip to *Wesseling*.
9. What is the *Sub* of the day at *Subways*?
10. Please take a picture of the head statue on *Martinsplatz*.

The L group participants were using the location-aware reminder. The T group participants only had a to-do-list.

The participants in the two L groups were notified by the ShinyNavi application when ever they reached a place where they had to accomplish a task. Additionally, they also had the option to view all tasks in a to-do-list. The participants in the two T groups were not notified by the application. They only had the to-do-list as a reminder. For the to-do-list, a third party application named "My ToDo List"[1] was used.

A continuous distraction was simulated with math tasks.

To simulate a continuous distraction, the participants had to solve addition and subtraction problems with numbers up to 1000. The math

[1] `play.google.com/store/apps/details?id=nz.co.guevara.mytodolist`

tasks were asked continuously and in oral form on the way through the city. They were repeated if requested by a participant.

The participants were briefed at the beginning of each experiment.

At the beginning of an experiment, the respective participant was shorty briefed. In the briefing the participant was told that the study was about the effectiveness of a navigation system for pedestrians and that there were ten tasks along a predefined track were to accomplish to make the experience more realistic. Also it was mentioned that a continuous distraction would be simulated with addition and subtraction tasks. The evaluation of the navigator was emphasized as the most important part of the study. A smartphone with the ShinyNavi application installed and the to-do-list was handed out to the participant and it was explained how to control the software.

A questionnaire was filled out at the end.

At the end of each experiment, a questionnaire was handed out, in which the participants were asked about their gender, age, smartphone experience and how appropriate the tasks and routing was. In the lab study they were also asked about the authenticity of the simulation. After the participant had filled it out, he or she was debriefed, by revealing that the navigation system was only mocked and that the study was actually about comparing a location-aware task reminder to a conventional to-do-list, by counting the number of forgotten tasks. In the lab study, they were also informed about the piggyback.

4.1.1 Field Study

Participants were accompanied by two researchers.

In the field study each participant was accompanied by Sönmez and me on his or her track. One researcher controlled the ShinyNaviController application that was used to send directions and tasks to the ShinyNavi application as well as recording observation data. The researcher controlling the application was also responsible for stating the math tasks. The other researcher payed attention to the traffic in order to avoid dangerous situations.

4.1.2 Presence Simulator Based Study

The design of the PS based study was similar to the field study.

The Presence Simulator based study was similar to the field study except that the participants did not walk through the real inner city of Bonn, but moved an LSF in a model of the same area. Also there was only one researcher present during the experiments.

The LSF had to be moved very slowly.

The briefing at the beginning was extended by an introduction to the Presence Simulator, the model landscape and how to move the LSF in the model. In order to avoid that participants finish the track in a very short time, they were told to move the LSF as slowly as possible. It was explained, that the navigation application needs the slow movement to work properly – which is not true, but sounds reasonable.

It was explained how to use the PS.

Depending on the gender of the participant, a male or a female playmobil® figure was used as an LSF. The participants were told to

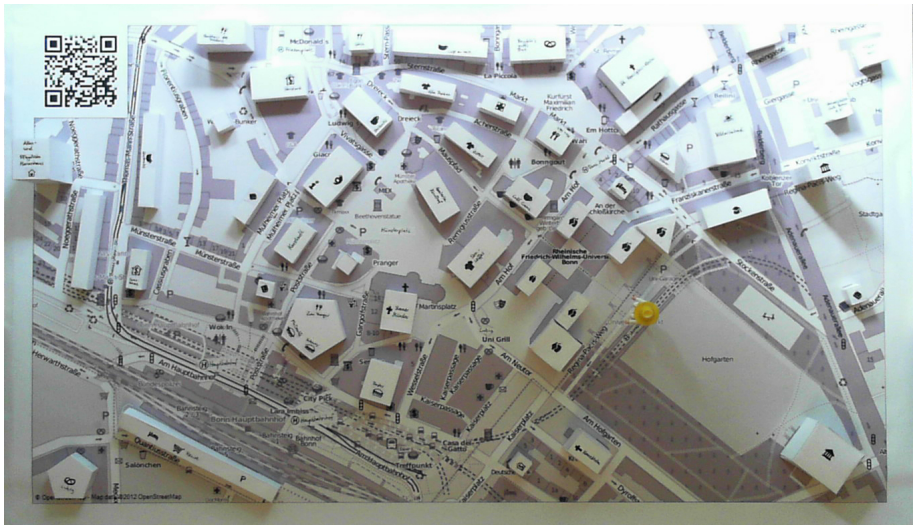


Figure 4.1: Model of the inner city of bonn with an LSF standing in front of the university.

inform the researcher when the LSF reached a place where a task could be completed. The participants were then provided with the material needed to accomplish the task (e. g. a photo of the kebab shop on *Strockenstraße* or a batch of different pamphlets).

The model landscape was built on top of a printed map.

The model landscape that was constructed for the study is shown in Figures 4.1 and 4.2. It consist of a printed DIN A0 map showing a larger area than needed for the track to avoid having to move the LSF along the corners of the map.

Paper prisms roughly indicate the location of important buildings.

Some buildings were modeled with paper prisms. On the paper prisms, small pictographs to indicate shops and public buildings were drawn. Also, on most paper prisms the names of the shops (e. g. *Starbucks*) were written. Some paths through buildings were not indicated on the map. In these cases, the dashed lines were added to indicate the way.

Traffic noise was played.

In order to simulate ambient noise, a recording of traffic noise was played in a continuous loop.

4.2 ShinyNavi Implemenation

The navigation system was implemented by applying the Wizard-of-Oz scheme.

The implementation of a real navigation system for pedestrians with an integrated location-aware task reminder would have been very expensive. Therefore, the elaborate parts were done by a researcher who, at the right moment, sent a direction or task reminder command to a client application running on the smartphone that was handed out to the participant. The researcher acted as the man behind the curtain. By doing this, the same experience as with a real implementation was provided to the participants, but without the efforts of actually implementing it. Such experiments are often called Wizard-of-Oz experiments, because

Figure 4.2: Close shot of the model of the inner city of Bonn with an LSF standing near the *Lubig* bakery.

they are similar to "what happened to Dorothy in the Wizard of Oz" [GW85].

The application that sends the commands is called *ShinyNaviController*. It is controlled by one of the researchers that accompany the participants on their route. Besides methods for sending direction commands and task reminders, the application also offers methods for logging the participants behavior and a list of math tasks. The application was developed for android and is shown in Figure 4.3b.

The *ShinyNaviController* can send direction and task command and is used for logging the participants behavior.

Figure 4.3a shows the *ShinyNavi* application that was used by the study participants. There is a map view in the background that shows the current location of the participant. In the figure, the application has received a left command during the last 10 seconds, therefore the road sign is visible in the foreground. When a task reminder is received, a text box with the task's description is shown. Whenever a command is received, the smartphone vibrates.

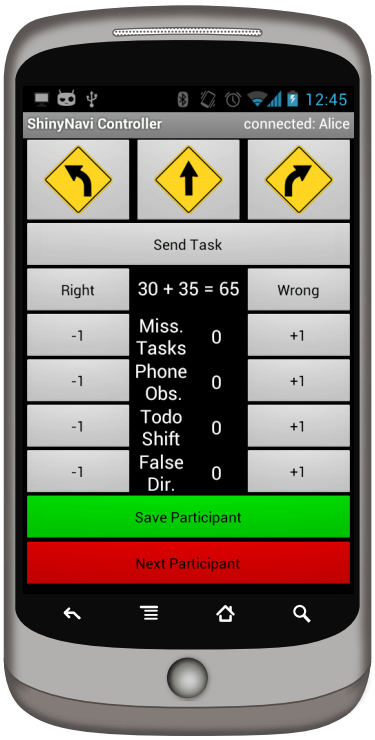
The *ShinyNavi* application was used by the participants.

In a prototype of *ShinyNavi* and the *ShinyNaviController*, the applications were connected via the Android Cloud to Device Messaging (C2DM) service, offered by Google.[2] However, it turned out that C2DM is not reliable enough for the study. Sometimes, it took several seconds until a message was received and some messages were even lost. Therefore, I implemented a bluetooth version of the applications and that fixed the problem.

The applications are connected via bluetooth.

[2] `developers.google.com/android/c2dm/`

(a) When a command is received by the ShinyNavi application, an overlay is drawn on top of the map.



(b) The ShinyNaviController enables the sending of directions and task commands as well as the logging of the participants' behavior.

Figure 4.3: The ShinyNavi application used by the study participants and the ShinyNaviController application used by the researchers.

The source code of *ShinyNavi* and the *ShinyNaviController* can be found on the CD-ROM attached in the back of this thesis.

4.3 Study Results

There was no significant effect of the environment on the number of accomplished tasks.

A one-way between subjects analysis of variance (ANOVA) was conducted to compare the effect of the test environment (field vs. lab) on the number of accomplished tasks (0-10) for participants of the two groups that had only a to-do-list as a reminder (P groups). The participants with the highest and lowest accomplished task count in each subgroup were excluded from the analysis (P04, P09, PS05 and PS11). There was no significant effect of the test environment on the number of accomplished tasks for the two conditions [$F(1,6) = 0.871$, $p = 0.387$].

The participants in the L groups accomplished all tasks.

Because all participants in the two groups with the location-aware task-reminder (L groups) managed to accomplish all ten tasks, there was no significant effect of the environment on the number of accomplished tasks for the L groups, either.

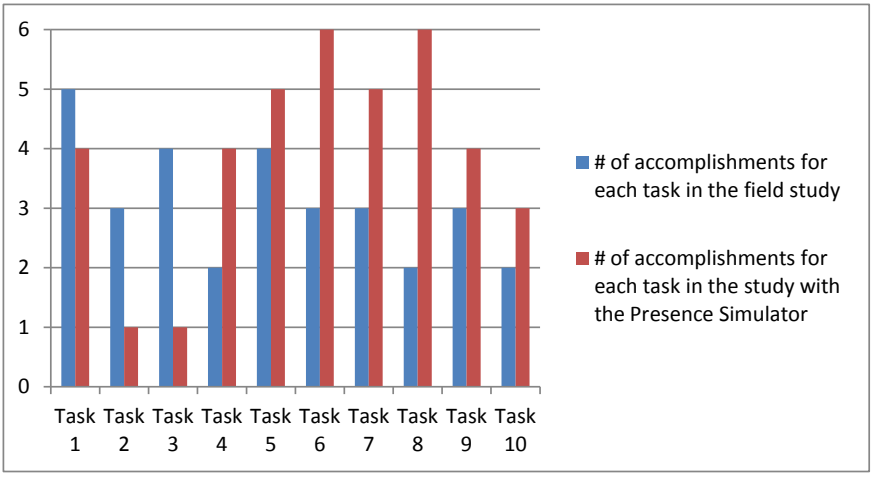


Figure 4.4: Accumulations of task accomplishments per task by participants in the groups that had only the to-do-list as a reminder (T groups). The groups had a size of six participants.

There was a difference in the results of the lab and field study when the completion of single tasks is compared.

Figure 4.4 shows how many participants accomplished each task in detail. The accomplishments of the participants of the field study are compared to the ones of the participants of the lab study. It is obvious, that most participants using the Presence Simulator forgot the second and third task. The second task was to pick up a *UNICUM* magazine from the University. In the field study, the participants were guided through a room full of magazine racks. Presumably the room was not simulated adequately in the model landscape. The third task was to find out the name of the kebab shop on *Strockenstraße*. In the field study, the shop emitted a strong kebab odor. Maybe the lack of this odor in the model landscape is responsible for the fact that only one participant of the Presence Simulator based study accomplished the third task.

There was no significant effect of the environment on the number of detours from the track.

A one-way between subjects ANOVA was conducted to compare the effect of the test environment (field vs. lab) on the number of times the participants went astray from the track. Again, the participants with the highest and lowest count were excluded from the analysis (P3, P12, PS02 and PS07). There was no significant effect of the environment on the number of detours from the track for the two conditions [$F(1,18) = 4.366$, $p = 0.051$].

The model landscape should be accessible from all sides.

During the study with the Presence Simulator, I observed that some participants had problems in deciding which way to move when they were not looking in the same direction as the LSF. Therefore, I advice to place the model landscape in the center of the lab room, so that the participants can access it from all sides.

The test environment effected the time participants needed for the track.

A one-way between subjects ANOVA was conducted to compare the effect of the test environment (field vs. lab) on the time it took the participants to complete the track. The participants with the longest and shortest time were again excluded from the analysis (P1, P11, PS01 and

PS08). There was a significant effect of the environment on the time the participants needed to complete the track for the two conditions [$F(1,18) = 63.189$, $p < 0.001$].

Depending on the study, methods for extending the duration of the experiments should be added.

Although the participants of the Presence Simulator based study took more time than I expected, they were still much faster than the participants in the field environment. Therefore, when a study is carried out where time is critical, methods for extending the period of time the participants spend on an experiment with the Presence Simulator should be added. In other cases, faster experiments imply more participants in fewer time.

There was no significant effect of the environment on the percentage of incorrectly answered math tasks.

A one-way between subjects ANOVA was conducted to compare the effect of the test environment (field vs. lab) on the percentage of incorrectly answered math task. The participants with the highest and lowest percentage values were excluded from the analysis (P10, P12, PS01 and PS05). There was no significant effect of the environment on the percentage of incorrectly answered math tasks for the two conditions [$F(1,18) = 0.728$, $p = 0.405$].

The participants rated the authenticity of the simulation rather high.

In the questionnaire handed out to the participants of the group using the Presence Simulator, the participants were asked to rate the authenticity of the simulation on a scale between 0 for very unrealistic and 9 for very realistic. On average the participants rated the authenticity with a 6.2 (mean = 6.2, median = 6).

Chapter 5

Summary and Further Work

5.1 Summary and Contributions

Missing evaluation techniques inhibit the success of location-aware applications.

Location awareness is an established technology, in which the geographic location of a device is influencing an applications function. In the past, location-aware applications were not as successful as they could have been. Often, privacy concerns are mentioned as an inhibitor. The expensive evaluation process of location-aware applications is rarely addressed.

The PS maintains the scalability of field evaluations while keeping the options and convenience of a laboratory.

Location-aware applications are usually evaluated in expensive field studies, which are hard to control, exhausting and time consuming for researchers and participants. The system developed in this thesis tries to solve these problems. The system is called Presence Simulator (PS). It is a special laboratory setup, which maintains the scalability of a field evaluation while keeping the options and the convenience of a laboratory. The system simulates the geographic location of study participants by tracking the movement of small toy figures, called Location Source Figures (LSFs), in a model landscape. In the current version of the Presence Simulator, playmobil® figures are used as LSFs. The simulated presence is transfered to android smartphones, on which the geographic location of the device's operating system is mocked accordingly. As a result, the applications to be evaluated in research studies using the Presence Simulator work with the mocked location data.

Living laboratories and virtual simulations are alternatives, but with either limited scaling abilities or a high level of abstraction.

Besides field studies involving actual users and an environment that is as realistic as it can be, one-to-one simulations of a particular environment called living laboratories and completely virtual simulations have been proposed in related works for the evaluation of location-aware applications. This thesis tries to overcome the limited scaling abilities of living laboratories and the high level of abstraction that is inherent in a virtual environment, with the implementation of a tangible user interface.

The PS is based on a client-server model. LSFs are tracked passively with the help of a web cam.

The Presence Simulator is based on a client-server model. A web cam mounted above the model landscape is attached to the computer on which the server software is running. The server tracks the location of the LSFs and maps their position in the model landscape to geographic coordinates. LSFs are tracked passively only by their appearance. The geographic coordinates are transfered as an XML stream of location updates via a TCP connection to a client software running on android smartphones

The five components of the PS's server.

The Presence Simulator's server is based on a MVC architecture and consits of the following five componentens:

- The *Location Sources* component models LSFs by saving their simulated geographic coordinates and defines a *Discriminator* interface to distinguish multiple LSFs.
- *Detectors* recognize LSFs in the video stream by applying computer vision algorithms. In the current version of the Presence Simulator, there is the *ColorBasedDetector*, recognizing LSFs by their color, and the *MarkerBasedDetector*, that is able to detected a special type of two dimensional marker. The *ColorBasedDetector* is sensitive to changes in lighting. The *MarkerBasedDetector* is more tolerant to changes in lighting, but markers on playmobil® figures might distract the users.
- The *Map* component maps pixel coordinates to geographic coordinates, enables the user to define the spacial area that corresponds to the model landscape, and visualizes LSFs in a map view that is part of the server's GUI.
- The *Network* component offers a service to register to the location updates of a specific location source in the network and sends location updates to the smartphones on which the client software is running.
- The *Recorder* is responsible for recording the tracks of location sources by saving them to a GPX file. Saved GPX files can be replayed and it is possible to stream the replay to the client.

The PS client runs on android, connects to the server and mocks the device's geographic location.

The Presence Simulator's client was developed for android. It connects to the server and has a GUI for selecting a specific LSF and registering to its update stream. The received updates are used to mock the devices geographic location. The main parts of the client software are running in the background, so updates are continuously received, even when other applications are in the foreground.

A comparative study was conducted.

To determine the effect of the test environment, the Presence Simulator was evaluated in a comparative study. In the study, the effectiveness of a location-aware task reminder was compared to a to-do-list: First in a field environment and then using the Presence Simulator. Half of the participants attended to field experiments. The other half used the Presence Simulator. Both groups were split again into a group that used the location-aware task reminder and a group that used a to-do-list for accomplishing the same tasks.

There was no significant effect on the number of accomplished tasks.

There was no significant difference in the number of accomplished tasks between the groups in the field study and the corresponding groups in the Presence Simulator based study. The participants using the Presence Simulator were significantly faster than the ones in the field and thus the Presence Simulator based study was carried out faster than the field study.

There was no significant difference in the number of detours between lab and field study.

Because the participant's level of distraction affects the number of forgotten tasks, the location-aware task reminder and the to-do-list were not evaluated directly but embedded in a navigation system for pedestrians. There was no significant effect of the test environment (field vs. lab) on the number of detours from the track.

The PS based study produced results similar to the field study.

The conducted Presence Simulator based study was able to produce results similar to the results of the conducted field study, but was done faster. Because it was situated in a controllable lab environment, all tests were performed under the same conditions and no downtimes for recharging the smartphones were needed.

5.2 Further Work

A list of ideas, techniques and methods that can be implemented in further work.

The Presence Simulator that has been developed in this thesis was build up from scratch and due to the limited amount of time, I was not able to implement and evaluate all the ideas that came to my mind. The Presence Simulator has been designed with extendability in mind and I hope that it will be extended by others in the future. Therefore, this section presents a list of features, techniques and methods that can be implemented in future work.

- Only one study to compare the usage of the Presence Simulator in a location-aware user study to a traditional field approach has been carried out so far. This is why no general statements about the validity of the results of studies using the Presence Simulator can be made. Therefore, it is advisable to carry out more comparative studies.
- The color based detector and the marker based detector are based on well known image recognition methods.The usage of a more recent method (like the Scale-invariant feature transform) might increase performance and/or detection quality, for example to enable the tracking of Lego® figures, which are much smaller than playmobil® figures.
- An implementation of the Presence Simulator based on the reacTIVision project (see Section 2) would make the web cam mounting construction on top of the table obsolete and the system could benefit from the additional features of a reacTIVision table (e. g. the touch interface).
- Adding a beamer to the system would allow the landscape model to be dynamic (e. g. live traffic data could be projected on a map).

This is also an advantage of the reacTIVision project, because it already has a beamer integrated.

- The 3D printing technology is a emerging and interesting research field in computer science. The Presence Simulator would greatly benefit from the possibility of printing out 3D landscape models and have them detected automatically by the system.
- The Presence Simulator can be used to evaluate the usability of applications, but the usability of the Presence Simulator itself has not yet been evaluated in a study.
- So far only the geographic location is simulated. To increase the authenticity of the simulation it might be an option to simulate orientation, altitude and velocity as well. The marker based detector is already capable of recognizing the marker's orientation, but this knowledge is currently not used by the client.
- The client software has only been implemented for android. Implementing a client for iOS and other mobile operating systems would enable the usage of the Presence Simulator on these platforms as well.
- The playback of ambient sounds is currently not integrated into the Presence Simulator. As a result, it might happen that the LSF stands in a park but the noise of cars is played. To avoid such situations, the sound should depend on the location.

Appendix A

Detector's Performance Measurments

The following tables lists the results of the performance measurements of the marker based and the color based detector. All measurements where taken at the android client right afterthe location updates where received and decoded. The values are the number of updates received within a 20 seconds period devided by 20. Thus the measurement unit is hertz (Hz). The Presence Simulator's server software was executed on an Intel Pentium dual core T4500 with 2.30GHz – which is at the time of writing a low-end device.

At 320x240 the markers could not be detected. At 960x544, there was a small but noticable delay (<1s) between the users action and the received updates. At 1280x720, the delay was irritating (>2s). At resolutions in between, performance was acceptable.

	Color based detector, one location source (Hz)		
Resolution	Measurement 1	Measurement 2	Measurement 3
320x240	24.75	26.55	27.7
640x360	23.45	25.5	23.55
800x448	15	15.05	15.05
960x544	14.7	14.9	15.1
1280x720	9.75	9.55	9.8

Table A.1: Color based detector performance measurements with one active location source.

	Color based detector, two location sources (Hz)		
Resolution	Measurement 1	Measurement 2	Measurement 3
320x240	21.5	23.8	21.15
640x360	21.05	21.35	20.95
800x448	15.35	15.1	14.95
960x544	9.9	10.0	9.95
1280x720	6.05	5.95	5.95

Table A.2: Color based detector performance measurements with two active location sources.

	Marker based detector, one location source (Hz)		
Resolution	Measurement 1	Measurement 2	Measurement 3
320x240	n/a	n/a	n/a
640x360	25.65	22.5	24.6
800x448	15.75	15.7	15.65
960x544	13.45	13.35	13.5
1280x720	2.9	2.7	3.2

Table A.3: Marker based detector performance measurements with one active location source.

	Marker based detector, two location sources (Hz)		
Resolution	Measurement 1	Measurement 2	Measurement 3
320x240S	n/a	n/a	n/a
640x360	22.2	23.65	21.9
800x448	15.35	15.4	15.65
960x544	12.3	12.65	13.05
1280x720	3.25	2.65	2.9

Table A.4: Marker based detector performance measurements with two active location sources.

Appendix B

Data Recorded in the Comparative Study

	Task (0 = missed. 1 = completed.)									
Participant	1	2	3	4	5	6	7	8	9	10
P02	0	1	1	0	1	1	0	0	1	0
P04	1	0	1	1	1	0	1	1	1	0
P05	1	1	1	1	1	1	1	0	0	0
P09	1	0	0	0	0	0	0	0	0	0
P10	1	0	0	0	0	0	0	1	1	1
P12	1	1	1	0	1	1	1	0	0	1
PS01	0	0	0	1	0	1	1	1	1	1
PS03	1	0	0	1	1	1	0	1	1	1
PS05	1	0	0	0	1	1	1	1	0	0
PS07	1	1	0	0	1	1	1	1	0	1
PS09	0	0	0	1	1	1	1	1	1	0
PS11	1	0	1	1	1	1	1	1	1	0

Table B.1: Task distribution of participants in the to-do-list group (T). Participants in the location-aware reminder group (L) managed to complete all tasks. P = field study participant. PS = Presence Simulator participant.

Participant[1]	Phone Observations[2]	App Switches[3]	Wrong Directions	Wrong Math Tasks	Right Math Tasks	Time [min][4]
P01 (L)	43	0	1	10	72	29.88
P02 (T)	43	9	1	7	100	28.43
P03 (L)	49	0	0	10	50	20.93
P04 (T)	50	8	0	10	82	27.07
P05 (T)	44	8	1	7	69	26.62
P06 (L)	39	0	0	9	83	25.67
P07 (L)	41	0	0	8	54	26.07
P08 (L)	52	0	0	13	66	23.52
P09 (T)	36	0	0	13	73	21.83
P10 (T)	32	4	0	4	133	26.08
P11 (L)	41	0	0	3	65	20.25
P12 (T)	32	10	2	16	35	21.27

Table B.2: Data recorded during the field study with the ShinyNaviController.

[1] L = Location-aware reminders. T = To-do-list only.
[2] Number of times the phone has been viewed.
[3] Number of switches between the to-do-list application and ShinyNavi.
[4] Time without briefing and debriefing.

Participant[5]	Phone Observations[6]	App Switches[7]	Wrong Directions	Wrong Math Tasks	Right Math Tasks	Time [min][8]
PS01 (T)	32	8	1	3	52	27.30
PS02 (L)	38	0	0	5	46	15.27
PS03 (T)	28	9	0	4	26	18.78
PS04 (L)	31	0	2	7	29	13.60
PS05 (T)	30	6	2	13	42	18.78
PS06 (L)	44	0	2	4	50	15.67
PS07 (T)	36	8	3	4	26	14.78
PS08 (L)	29	0	2	5	36	13.20
PS09 (T)	31	7	1	6	54	19.35
PS10 (L)	42	0	0	12	52	16.53
PS11 (T)	38	11	0	4	58	16.83
PS12 (L)	28	0	0	8	30	15.45

Table B.3: Data recorded during the Presence Simulator study with the ShinyNaviController.

[5] L = Location-aware reminders. T = To-do-list only.
[6] Number of times the phone has been viewed.
[7] Number of switches between the to-do-list application and ShinyNavi.
[8] Time without briefing and debriefing.

Participant[9]	Age	Sex	Smartphone Experience[10]	The Navigation was...[11]	The Tasks where...[12]	Exhaustion[13]	Mental Exertion[14]
P01 (L)	31	♂	8	7	9	6	4
P02 (T)	75	♂	0	9	4	9	5
P03 (L)	29	♀	8	8	5	6	5
P04 (T)	21	♀	8	7	6	4	9
P05 (T)	15	♂	8	9	9	7	4
P06 (L)	27	♂	7	8	0	4	6
P07 (L)	23	♀	0	8	8	8	7
P08 (L)	52	♂	7	7	1	7	8
P09 (T)	32	♂	8	7	n/a	8	9
P10 (T)	54	♂	0	7	5	6	7
P11 (L)	34	♂	1	9	9	9	9
P12 (T)	33	♀	0	4	4	9	9

Table B.4: Data recorded in the final questionnaire of the field study.

[9] L = Location-aware reminders. T = To-do-list only.
[10] 0 = No experience. 9 = Much experience.
[11] 0 = Confusing. 9 = Expedient.
[12] 0 = Disruptive. 9 = Appropriate.
[13] 0 = KO. 9 = Fit.
[14] 0 = Very high. 9 = Not at all.

Participant[15]	Age	Sex	Smartphone Experience[16]	The Navigation was...[17]	The Tasks where...[18]	Exhaustion[19]	Mental Exertion[20]	Authenticity of the Simulation[21]
PS01 (T)	26	♀	6	7	7	2	2	6
PS02 (L)	25	♀	9	7	9	4	2	6
PS03 (T)	43	♀	1	8	9	5	3	5
PS04 (L)	11	♂	7	9	9	6	4	8
PS05 (T)	13	♂	2	7	8	9	6	8
PS06 (L)	53	♂	3	8	7	5	3	6
PS07 (T)	25	♀	2	4	6	2	3	5
PS08 (L)	34	♂	0	8	0	8	5	5
PS09 (T)	30	♂	5	8	9	7	7	7
PS10 (L)	62	♀	5	7	5	7	8	7
PS11 (T)	63	♂	4	7	9	7	5	6
PS12 (L)	26	♂	9	8	7	5	1	5

Table B.5: Data recorded in the final questionnaire of the Presence Simulator study.

[15] L = Location-aware reminders. T = To-do-list only.
[16] 0 = No experience. 9 = Much experience.
[17] 0 = Confusing. 9 = Expedient.
[18] 0 = Disruptive. 9 = Appropriate.
[19] 0 = KO. 9 = Fit.
[20] 0 = Very high. 9 = Not at all.
[21] 0 = Very unrealistic. 9 = Very realistic.

Bibliography

[MS] MSDN. Implementing Singleton in C#. `msdn.microsoft.com/en-us/library/ff650316.aspx` (visited on August 29, 2012).

[AM00] G. D. Abowd, E. D. Mynatt. Charting Past, Present and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction* 7:29–58, 2000.

[BBC+10] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation, World Wide Web Consortium, December 2010.

[Blo11] S. R. Blom. The CareRabbit business model: Can innovation get any cuter? Master's thesis, January 2011.

[Bro07] Broadcom Corporation. BCM4750 Product Brief. Technical report, Irvine, California, USA, November 2007.

[Cam03] Cambridge Positioning Systems (bought by Cambridge Silicon Radio). Operators Forecast $12bn Location Based Services Revenues - But Demand Low Cost Enabling Technology. Technical report, Cambridge Positioning Systems, Ltd, 2003.

[Cas12] L. Cassavoy. 21 Awesome GPS and Location-Aware Apps for Android. August 2012. `www.pcworld.com/article/260112/21_awesome_gps_and_locationaware_apps_for_android.html` (visited on August 25, 2012).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[emp09] empira. PDFsharp library for processing PDF. 2009. `www.pdfsharp.com` (visited on September 1, 2012).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GBG04] J. Goodman, S. Brewster, P. Gray. Using Field Experiments to Evaluate Mobile Guides. In Schmidt-Belz and Cheverst (eds.), *HCI in Mobile Guides, workshop at Mobile HCI 2004*. Glasgow, UK, September 2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gooa] Google Inc. Android Developers. `developer.android.com` (visited on August 30, 2012).

[Goob] Google Inc. Android W3C DOM API. `developer.android.com/reference/org/w3c/dom/package-summary.html` (visited on August 30, 2012).

[GW85] P. Green, L. Wei-Haas. The Wizard of Oz: A Tool for Rapid Development of User Interfaces. June 1985.

[HBMS04] J. M. Harris, A. M. Butcher, D. A. Morton, J. C. Satterfield. Command user interface for displaying selectable software functionality controls. September 2004. US 2006/0036965 A1. http://www.google.de/patents/US20060036965

[Heg10] M. Hegen. *Mobile Tagging: Potenziale von QR-Codes im Mobile Business*. August 2010.

[Hew10] Hewitt, Joe (joehewitt). "Until Android is read/write open, it's no different than iOS to me. Open source means sharing control with the community, not show and tell.". October 2010. `twitter.com/joehewitt/status/27878912110` (visited on August 28, 2012).

[IDC12] IDC Corporate USA. Android and iOS Surge to New Smartphone OS Record in Second Quarter, According to IDC. August 2012. Press Release.

[Ips12] Ipsos OTX MediaCT. Unser mobiler Planet: Deutschland - Der mobile Nutzer. May 2012. `services.google.com/fh/files/blogs/our_mobile_planet_germany_de.pdf` (visited on August 25, 2012).

[IU97] H. Ishii, B. Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '97, pp. 234–241. ACM, New York, NY, USA, 1997.

[KB07] M. Kaltenbrunner, R. Bencina. reacTIVision: a computer-vision framework for table-based tangible interaction. In *Proceedings of the 1st international conference on Tangible and embedded interaction*. TEI '07, pp. 69–74. ACM, New York, NY, USA, 2007.

[KE04] D. Kulak, EamonnGuiney. *Use Cases: Requirements in Context*. Addison-Wesley, 2004.

[Kira] A. Kirillov. AForge.NET Framework. `www.aforgenet.com` (visited on September 1, 2012).

[Kirb] A. Kirillov. Glyph Recognition And Tracking Framework. `code.google.com/p/gratf/` (visited on September 1, 2012).

[Kir08] A. Kirillov. Lego Pan Tilt Camera and Objects Tracking. November 2008. `www.codeproject.com/Articles/31104/Lego-Pan-Tilt-Camera-and-Objects-Tracking` (visited on September 1, 2012).

[KSAH04] J. Kjeldskov, M. B. Skov, B. S. Als, R. T. Hoegh. Is it Worth the Hassle? Exploring the Added Value of Evaluating the Usability of Context-Aware Mobile Systems in the Field. Pp. 61–73. Springer-Verlag GmbH, 2004.

[KVB88] N. Kanopoulos, N. Vasanthavada, R. L. Baker. Design of an image edge detection filter using the Sobel operator. *IEEE Journal of Solid-State Circuits* 23(2):358–367, April 1988.

[MCB12] J. Marco, E. Cerezo, S. Baldassarri. ToyVision: a toolkit for prototyping tabletop tangible games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. EICS '12, pp. 71–80. ACM, New York, NY, USA, 2012.

[Mic] Microsoft Corporation. Microsoft .NET Framework. `www.microsoft.com/net/` (visited on September 1, 2012).

[MRG99] S. J. McKenna, Y. Raja, S. Gong. Tracking colour objects using adaptive mixture models. *Image and Vision Computing* 17(3-4):225 – 231, 1999.

[MSD] MSDN. Windows Forms Reference. `msdn.microsoft.com/en-us/library/dd30h2yb.aspx` (visited on August 30, 2012).

[Mur11] B. P. Murphy. SoundStage. May 2011.

[OLMD07] E. O'Neill, D. Lewis, K. McGlinn, S. Dobson. Rapid user-centred evaluation for context-aware systems. In *Proceedings of the 13th international conference on Interactive systems: Design, specification, and verification*. DSVIS'06, pp. 220–233. Springer-Verlag, Berlin, Heidelberg, 2007.

[Ord10] Ordnance Survey. A guide to coordinate systems in Great Britain. December 2010. `www.ordnancesurvey.co.uk` (visited on September 1, 2012).

[PSKS11] T. Pallos, G. Sziebig, P. Korondi, B. Solvang. Multiple-Camera Optical Glyph Tracking. *Advanced Materials Research* 222:367–371, April 2011.

[rad] radioman [Pseudonym]. GMap.NET WinForms Control version 1.6.

[RCT+07] Y. Rogers, K. Connelly, L. Tedesco, W. Hazlewood, A. Kurtz, R. E. Hall, J. Hursey, T. Toscos. Why it's worth the hassle: the value of in-situ studies when designing Ubicomp. In *Proceedings of the 9th international conference on Ubiquitous computing*. UbiComp '07, pp. 336–353. Springer-Verlag GmbH, 2007.

[rea05] The Design and Evolution of Fiducials for the reacTIVision System. 2005.

[RLMM09] R. Rogers, J. Lombardo, Z. Mednieks, B. Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., 1st edition, 2009.

[Rus99] B. Russell. know your place - headmap - location aware devices. 1999. `technoccult.zippykidcdn.com/wp-content/uploads/library/headmap-manifesto.pdf` (visited on August 25, 2012).

[SH09] O. Shaer, E. Hornecker. Tangible User Interfaces: Past, Present and Future Directions. *Found. Trends Hum.-Comput. Interact.* 3(1-2):1–137, 2009.

[SLCJ04] O. Shaer, N. Leland, E. H. Calvillo-Gamez, R. J. K. Jacob. The TAC paradigm: specifying tangible user interfaces. *Personal Ubiquitous Comput.* 8(5):359–369, September 2004.

[Sta11] T. Stapelkamp. *Interaction- und Interfacedesign: Web-, Game-, Produkt- und Servicedesign Usability und Interface als Corporate Identity*. X.media.press / publishing. Springer-Verlag GmbH, 2011.

[Ste04] C. Steinfield. *The development of location based services in mobile commerce*. Pp. 177–197. Springer-Verlag GmbH, 2004.

[Top] TopoGrafix. GPX: the GPS Exchange Format.

[UI97] B. Ullmer, H. Ishii. The metaDESK: models and prototypes for tangible user interfaces. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. UIST '97, pp. 223–232. ACM, New York, NY, USA, 1997.

[UI00] B. Ullmer, H. Ishii. Emerging frameworks for tangible user interfaces. *IBM Syst. Jornal* 39(3-4):915–931, July 2000. http://www.research.ibm.com/journal/sj/393/part3/ullmer.html

[UMT00] UMTS Forum, Information and Communication Technologies Group. Enabling UMTS / Third Generation Services And Applications. Technical report, UMTS Forum, London, October 2000.

[Wei08] M. Weisfeld. *The Object-Oriented Thought Process*. Addison-Wesley Professional, 3rd edition, 2008.

[ZXi] ZXing Team. ZXing barcode processing library. `code.google.com/p/zxing/` (visited on September 1, 2012).

Glossary

3G	3rd Generation (of mobile telecommunication technology)
API	Application Programming Interface
CIL	Common Intermediate Language
CLR	Common Language Runtime
COM	Component Object Model
DOM	Domain Object Model
DVM	Dalvik Virtual Machine
GNU	GNU's Not Unix
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
HCI	Human-Computer Interaction
IDE	Integrated Development Environment
IP	Internet Protocol
JVM	Java Virtual Machine
LAN	Local Area Network
LS	Location Source
LSF	Location Source Figure
MAM	Marble Answering Machine
MVC	Model View Controller
PS	Presence Simulator
QR	Quick Response
RFID	Radio Frequent Identification
RGB	Red Green Blue (color model)
SAX	Simple API for XML
TCP	Transmission Control Protocol
TUI	Tangible User Interface
UDP	User Datagram Protocol
W3C	World Wide Web Consortium
WLAN	Wireless Local Area Network

Source-code and documentation of
the Presence Simulator is available online at
`www.applesandoranges.eu/masterthesis/software.zip`
Username: uni
Password: uhg592z

Typeset October 2, 2012