Physical Widgets on Capacitive Touch Displays

Master's Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Hauke Schaper

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Kjell Ivar Øvergård

Registration date: 31.01.2013
Submission date: 15.04.2013

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, April 2013
Hauke Schaper

Contents

List of Figures

List of Tables

Abstract

Touch displays are more and more integrated in daily life, as in mobile phones, vending machines or tablets. Modern touchscreens utilize projected-capacitive touch technology. It measures changes of capacitance on the surface of the screen. Touchscreens provide a natural and intuitive interface for interactions. One drawback is the missing haptic feedback. Virtual objects yield no feedback disregarding the visual clues. Therefore eye-free manipulation is hardly feasible on planar touchscreens. Widgets, physical controls, unite the benefits of touchscreens and haptic feedback. They are placed on top of the touchscreen and paired with virtual objects. A manipulation of the physical object results in a modification of the virtual one, enabling eyes-free interaction. When using acrylic for the widgets, the labeling underneath is adapted to match the needs of the currently paired object. One advantage of capacitive widgets is the hover functionality. A touch on the widget is detected without changing its physical state. Each widget is identified through its unique marker pattern underneath. Furthermore, each control embeds a set of markers to indicate its state to the system. This thesis provides an overview of the design and construction of widgets, pointing out the constraints and limitations. In addition a framework implementation for the tracking of widgets on multi-touch tables is explained in detail. The thesis concludes with an overview of the design space for capacitive touchscreen widgets.

Überblick

Touch Bildschirme sind zunehmend in unserem täglichen Leben integriert, sei es in Form von Mobiltelefonen, Verkaufsautomaten oder Tablets. Moderne Touch Bildschirme verwenden projizierte kapazitive Touch Technologien. Diese messen Veränderungen der Kapazität auf der Oberfläche des Bildschirms. Touch Bildschirme bieten eine natürliche und intuitive Interaktionsumgebung. Ein Nachteil besteht im fehlenden haptischen Feedback. Virtuelle Objekte bieten ausschließlich visuelle Rückmeldungen. Auf ebenen Touch Oberflächen ist folglich keine blinde Benutzung möglich. Widgets, das heißt physische Steuerelemente, vereinen die Vorteile von Touch Bildschirmen und haptischem Feedback. Sie werden auf der Bildschirmoberfälche plaziert und mit virtuellen Objekten in Verbindung gebracht. Eine Manipulation der physischen Objekte bewirkt eine Veränderung der Virtuellen, was eine Interaktion ohne Sichtkontakt ermöglicht. Wird Acryl für die Widgets verwendet, werden die Beschriftungen darunter angepasst um die Bedürfnisse der gegenwärtig verbundenen Objekte zu erfüllen. Ein Vorteil von kapazitiven Widgets besteht in der Hover-Funktion. Die Berührung eines Widgets wird erkannt, ohne den physikalischen Zustand zu verändern. Jedes Widget ist durch darunterliegende, einzigartige Markermuster eindeutig identifizierbar. Weiterhin beinhaltet jedes Steuerelement eine Menge von Markern, um dem System seinen Zustand zu übermitteln. Diese Abhandlung bietet eine Übersicht über den Entwurf und die Konstruktion von Widgets, wobei Beschränkungen und Grenzen herausgestellt werden sollen. Zusätzlich wird eine im Rahmen dieses Projektes entwickelte Framework-Implementierung zur Erkennung von Widgets auf Multi-Touch-Tischen detailliert vorgestellt. Diese Masterarbeit wird mit einer Übersicht des Design Spaces für kapazitive Widgets abgeschlossen.

Acknowledgements

This Work is dedicated to Amaru and Shiva. They supported me during my work by distracting me from it.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition: *Excursus*

CONVENTION:
Conventions are conditions that are valid for a part of the thesis or the whole thesis.

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in American English.

Chapter 1

Introduction

Touch displays are more and more integrated into daily life, for example the ticket machine from Deutsche Bahn, a smart phone, a car navigation system, a modern phone booth or tablets. Touch devices are ubiquitous in our daily life. Traditional mechanical input devices are more and more replaced through modern touch systems. The technology evolves parallel to the emerge of touch devices. The three major technologies are resistive, optical and capacitive. Resistive touchscreens (Speeter [1990]) are pressure sensitive and used in PDAs, merchandise systems and home electronics like coffee machines. The advantage of resistive in comparison to capacitive is that touch is not only detected from conductive objects. Every object applying pressure on the screen is detected.

Capacitive touchscreens are emerging.

Optical tracking technologies use cameras to track touches on the surface. These systems are mostly utilized in large tabletops as Microsoft's PixelSense[1] and research projects like SLAP (Weiss et al. [2009]). SLAP uses Frustrated Total Internal Reflection (Han [2005]) and Diffused Illumination (Matsushita and Rekimoto [1997]). These techniques limit the usage of the optical tracking systems in various ways. A projector that is placed underneath the surface to project the image requires space. Most optical tracking systems require a lot of space. If these systems are moved, the projector and the tracking cameras are moved as well. These

Optical tracking systems are limited in portability.

[1]http://www.microsoft.com/en-us/pixelsense/

slight movements destroy the calibration and it has to be redone. This limits the portability of most optical tracking systems. Another drawback of Frustrated Total Internal Reflection and Diffused Illumination is the vulnerability to sunlight. Sunlight emits infrared light into the table, disturbing the touch tracking.

Capacitive touchscreens use conductivity measuring sensors to detect touches.

Devices implementing a multi-touch interaction commonly use projected-capacitive touch (Barrett and Omote [2010]). Most smart phones, tablets and other multimedia devices belong to this category. Capacitive touchscreens use conductivity measuring sensors to detect touches. One limitation of capacitive touchscreens is the affordance that only conductive elements are used for input. Graspable User Interfaces (Fitzmaurice et al. [1995]) are common in optical systems (reacTable by Jordà [2010]) and advancing on capacitive touchscreens (TUIC Yu et al. [2011], Capstones and ZebraWidgets by Chan et al. [2012]).

Widgets allow eyes-free interactions through haptic feedback.

Graspable user interfaces, referred to as widgets, enrich the interaction of multi-touch tables by adding haptic feedback (Weiss [2012]). Haptic feedback allows eyes-free interaction on touch devices. Clip-on Gadgets (Chang et al. [2012]) introduce controls, such as buttons and joysticks, that are attached to capacitive devices. The device can be operated without watching the display. Further applications are found in expensive, large scale simulators. Nowadays each machine needs its own simulator when minor functional differences are present. A simulator can be built with widgets and large scale multi-touch tabletops. Each former simulator can be built with a different arrangement of widgets. This possibility enriches the interaction by providing additional information underneath the widgets on the screen. Furthermore, it allows more flexibility in changing the setup of a simulator. The needed haptic feedback to operate heavy machinery is conserved through physical widgets.

Optical tracking systems detect widgets with the help of markers placed on the widget.

Tracking physical objects on optical screens is realized by marker detection. In SLAP (Weiss et al. [2009]) the orientation and identification is detected through the orientation and shape of the markers. This approach is not transferable to capacitive displays, as these only provide coordinates of touches, merging large areas to one spot. Different approaches are presented, such as spatial and frequency markers (Yu et al. [2011]). Each existing work covers different aspects of capacitive widgets. Some introduce sample widgets, other discuss different marker types. No work covers the whole design process including limitations and constraints. This thesis is the approach to illustrate the whole widget construction and detection process. Additionally the design space for capacitive widgets is introduced.

The hover effect is a design opportunity given by the usage of capacitive touchscreens. It is the system reaction to a touch without changing the physical state of the widget. Using optical tracking a hover effect is hardly feasible. A widget has its fixed screen occlusion. Additional markers are lowered to the surface through mechanical help, otherwise it is only realizable with the help of glass fibers, extending the tracked screen to the widget surface. Conductive materials are embedded into the widgets on capacitive displays. These materials forward the touch downwards to the system. A touch is detected by the system, without changing the widget's physical state.

The hover effect on capacitive widgets is detected without the change of the physical state of the widget.

1.1 Outline

2—"Related work" covers the fundamentals in this area and work that has already been done in this field. Capacitive touch technology is explained, presenting the basics to understand the theories in the following chapters. Furthermore, the framework used for the implementation is briefly introduced. The chapter continues with an introduction to related work in the fields of rapid prototyping, optical tracking and capacitive tracking. It concludes by a brief summary about the shown work in relation to this thesis.

3—“Widget Construction” offers an overview of the design constraints that limit the design of widgets. A three phase construction model is introduced, including a set of materials for widget construction. The chapter ends with examples of constructed widgets to deepen the understanding of the former shown theories and constraints.

4—“Framework Implementation” is separated into two major parts. On the one hand the algorithm used for the footprint generation is presented and explained. On the other hand the method for widget tracking is shown in detail.

In 5—“Design Space” the mental evolution from touch to widget is illustrated. Furthermore, the design space is outlined and examples are presented.

6—“Summary and future work” concludes this thesis by reflecting on the content and presenting an overview of possible future work.

Chapter 2

Related work

" The secret to creativity is knowing how to hide your sources. "

— Albert Einstein

Research on the interaction with physical objects has moved from optical tracking to capacitive multi-touch displays.

The idea of using physical objects to control virtual objects has been explored on various tracking technologies. Frustrated Total Internal Reflection and Diffused Illumination were used by systems like *SLAP* (Weiss et al. [2009]) and the *reacTable* (Jordà [2010]). Capacitive multi-touch displays have replaced the optical tracking methods in current research as presented in *CapStones and ZebraWidgets* by Chan et al. [2012] and TUIC by Yu et al. [2011].

This chapter covers the foundations of physical objects called widgets, the design of capacitive touch display widgets and existing related work. We explain the basics of projected-capacitive touch displays as described by Barrett and Omote [2010]. Thereafter the framework used for the implementation of the footprint generation, the tracking and displaying of the widgets is introduced. The chapter concludes with related work covering different aspects of widgets and various possibilities to track them.

Definition: *Widgets*

WIDGETS:
Widgets are physical representations of controls, such as sliders, buttons, knobs and so forth. They are used to manipulate a software system running on a multi-touch table with the help of real world objects.

2.1 Capacitive Touch Technology

Projected-capacitive touch became popular through the iPhone in 2007.

Projected-capacitive touch detection has become popular with the release of the first iPhone in 2007. It enhanced the touch technologies used until then by three main aspects: high durability, excellent optical performance and unlimited multi-touch sensing, based on the controller operating it. Projected-capacitive touch is based on capacitive sensing.

Capacitive displays are separated into coordinate systems by the sensing electrodes.

Capacitive displays measure the capacitance of a single electrode in comparison to ground. The electrodes can be arranged in different ways under the display of a device. Commonly the electrodes are arranged in a grid, each responsible for a small area of the surface. Each electrode represents a specific area in the grid and is directly wired to the controller.

Self-Capacitance touch technology detects touches by capacitance changes in electrodes.

The electrode measures the capacitance of itself and observes it for changes. All lanes of the display are measured simultaneously. If a finger touches the surface the body capacitance changes the measured value of the electrode. The electrode sends the changed value to the controller and the controller detects a touch. This method of touch sensing is called self-capacitance touch technology by Barrett and Omote [2010].

Alternative approaches use two layers of electrodes.

A preferred alternative is to arrange lanes of electrodes in two layers instead of a grid. One layer with horizontal and the other one with vertical lines. Each intersection represents a coordination instead of an area in a grid. The controller measures each lane in both layers. If one vertical and one horizontal lane have a changed measured value, the controller can compute the position of the touch from the position of the intersection.

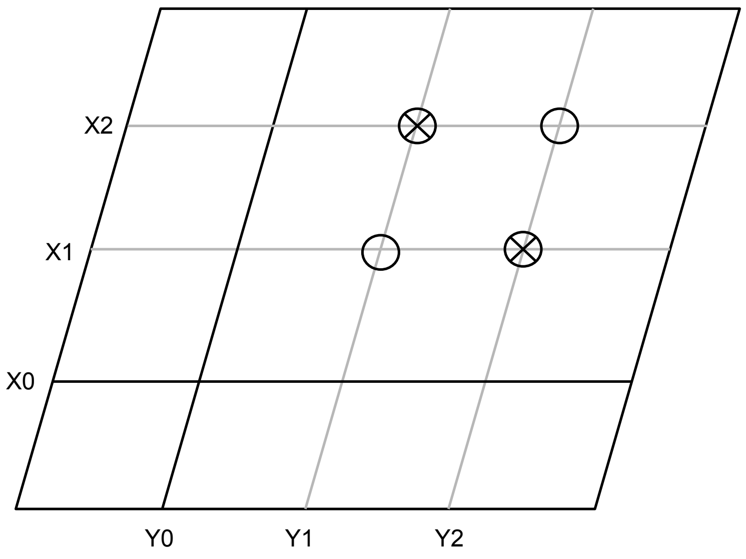

Figure 2.1: The circles mark the touches and the crossed circles symbolize the falsely detected touches in the self-capacitance approach.

Using more than one finger can result in false detection of additional touch-points.

Self-capacitance has a big drawback when using more than one finger on the surface. Multiple touches can result in a false amount of detected touches. For example, if two touches appear on two different pairs of lanes four touches would be detected. The touches may occur on the intersections $(1, 1)$ and $(2, 2)$. The controller would recognize a touch on lane x_1. The y-axis scan returns two active lanes, namely y_1 and y_2. Computing the intersections with the three touch points yield two detected touches, $(1, 1)$ and $(1, 2)$. For the touch on lane x_2 the touches on y_1 and y_2 would also appear as matches. This adds two more intersections on $(2, 2)$ and $(2, 1)$ to the detected touches. Figure 2.1 depicts the false touches as crossed circles and the correct touches as circles. Two of the four detected touches would be false results.

The iPhone introduces mutual-capacitance touch sensing technology.

Interactions with touchscreens make it desirable to use more than one finger for various operations. Scaling and rotation is simplified by using more than one finger. Accordingly the iPhone uses a modified version of the above mentioned self-capacitive sensing method. It is called mutual-capacitance by Barrett and Omote [2010].

Mutual-capacitance monitors intersections instead of whole lanes.

Mutual-capacitance uses the property of conductive elements that enables them to hold a charge if two objects are close together. The sensing is done by electrodes. Commonly they form a row - column pattern, the rows in one layer and the columns in another layer. There are different approaches like diamond patterns described in Barrett and Omote [2010]. Each charge of an intersection is monitored by the controller. In contrast to self-capacitance the controller differentiates in intersections instead of lanes when running the detection. A touch changes the charge of an intersection due to the body capacitance. The change in the charge is detected by a controller. The result is an identified touch on the intersection.

The scanning of mutual-capacitance differs from self-capacitance. In self-capacitance each electrode is directly connected to the controller. The scanning is serialized and the controller directly notices if a lane switches its state. Touches are computed based on the list of active lanes.

Mutual-capacitance uses a cyclic scanning mode to detect touches.

Contrary mutual-capacitance uses a cyclic scanning mode. The controller first checks one column and then iterates through all rows intersecting the column. It analyzes each intersection for a change in the charge. This process is repeated for every column in the display. After finishing all columns it starts over from the first column. The touches are computed by the changed charges of intersections.

Mutual-capacitance is a reliable technology for multi-touch.

Figure 2.2 depicts the example used for self-capacitance but this time for mutual capacitance. The process consists of nine different scans. Three column scans with three scans for the rows each time. This increases the load of the processing unit and the consumed time compared to self-capacitance. There this would be one scan of all lanes in parallel. Nevertheless, this scanning procedure enables a reliable multi-touch detection without false results. Only the six important scans are depicted in figure 2.2. The first scan checks lane y_1 and x_0. The charge of the intersection is unchanged. In the second picture the intersection of lane y_1 and x_1 is examined. The charge of the intersection is changed due to the body capacitance of the touching finger. The controller detects the change saving a touch for this intersection.

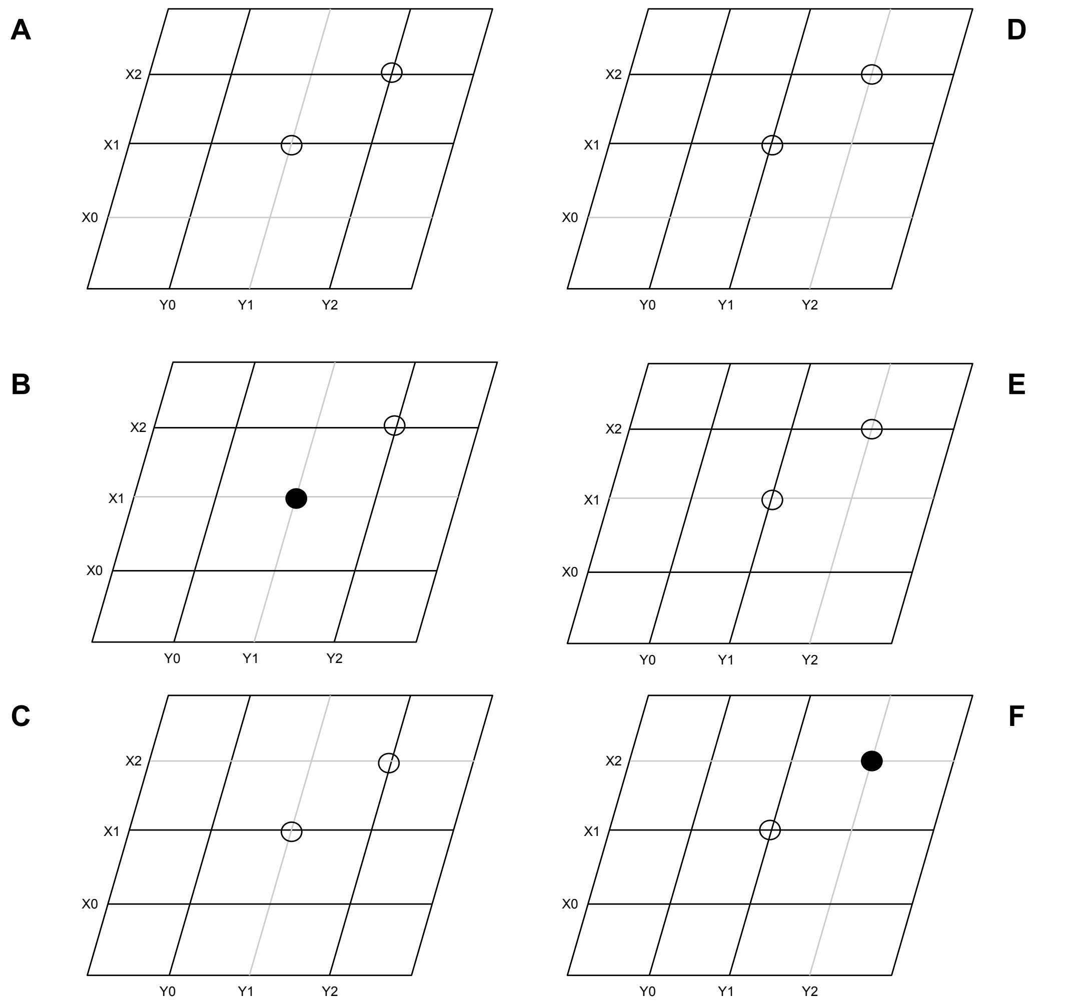

Figure 2.2: (A) shows the scanning of column one and row zero, (B) the scanning of column one and row one, (C) the scanning of column one and row two. (D)-(F) shows the same for column two. The black dot indicates that the touch is detected in this scan, the other one represents that the touch is present but not detected.

Widgets need more than one touch to be detected. With only one touch there is no differentiation between widgets because the footprint would always be the same. Therefore screens that can sense multiple touches without creating false touches are required to use widgets properly. The detection of the widget will be realized using an iPad version one. The iPad utilizes mutual-capacitance. It supports up to eleven touches at the same time. This is sufficient to detect widgets.

An iPad supports up to eleven touches simultaneously. It is used for our widget detection.

2.2 Multi-Touch Framework

A widget system consists of software and hardware components.

An interaction with widgets on a multi-touch table requires different components. On the one hand there is the hardware and on the other hand the software. The hardware is represented by the multi-touch table itself and the physical objects called widgets. The software side is subdivided into the application and the tracking software.

The tracking software detects learned widgets and creates screen representations.

The tracking software has a set of tasks. First it is responsible for saving new widgets in the system and learning their footprints. The second task is to detect the learned widgets and finally it provides a foundation to visualize the widgets on screen. To fulfill these tasks it needs the touch input from the table and a way to draw visualizations on the screen.

The Multi-Touch Framework is partitioned into different functional modules.

The requirements for the tracking software are satisfied by the Multi-Touch Framework[1]. It is actively developed by the Media Computing Group at the RWTH-Aachen. The framework is separated into different functional modules, as depicted in figure 2.3.

The MultiScreen Agent handles the touch input devices and configures the output displays.

The touch input is gathered by the MultiScreen Agent. It supports a wide choice of input sources. In the latest version it supports camera tracking, used in optical based tracking systems, mouse emulation, the use of a previous defined control sequence, iPad emulation and track pad emulation. The iPad emulation installs a small application on the iPad that broadcasts the touches to the network. The MultiScreen Agent listens on the port in the network and collects every incoming touch from the iPad. The touches are gathered in the agent and collected from the TableEngine.
Furthermore, the MultiScreen Agent is used to configure the output displays. The output screen is not necessarily the same as the input device. It is possible to merge two or more screens to one big display. The rendering and computation of the display area is done by the MultiScreen Agent.

[1]`http://hci.rwth-aachen.de/multitouchframework`

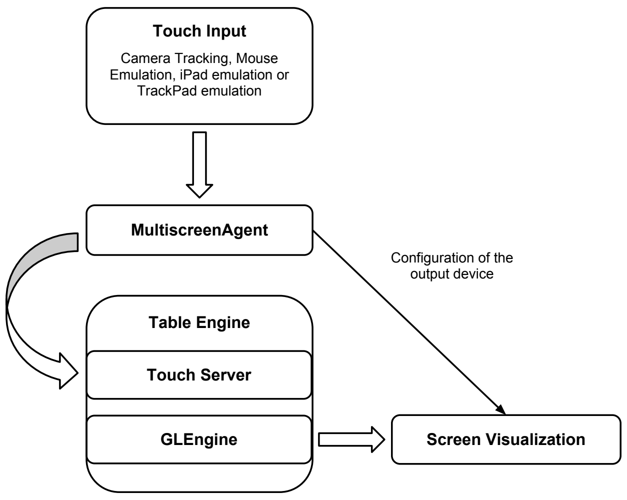
Touch Input
Camera Tracking, Mouse Emulation, iPad emulation or TrackPad emulation
MultiscreenAgent
Configuration of the output device
Table Engine
Touch Server
GLEngine
Screen Visualization

Figure 2.3: The different components of the MultiTouch framework and how they interact with each other.

The second part of the Multi-Touch Framework is the TableEngine. It is divided into two different parts: the touch server and a graphic engine. The touch server gathers the saved touches from the MultiScreen Agent and provides them for further usage. The standard settings are configured to display a dot on the display at the correct position for every touch. Any application integrating the TableEngine has access to the list of touches. The touch objects store further information, such as if it is a starting, ending or moving touch.

The TableEngines touch server provides the touches for the application.

The graphic engine called GLEngine is a framework to create graphical output. It provides basic shapes as rectangles and circles. The shapes are added to the TableEngine and then displayed. The touch server uses the GLEngine to display dots for the touch points.

The GLEngine is used to create graphical output objects.

2.3 Widgets

The second component of widget interaction on multi-touch tables is the hardware. The hardware is separated into physical objects called widgets and displays. This section introduces the concept and motivation for widgets by Fitzmaurice et al. [1995]. Additionally it provides an overview of existing related work in the field of widgets.

2.3.1 Concept of Widgets

Physical graspable user interfaces to control virtual objects.

In 1995 a new system of user interaction called *Bricks* was introduced by Fitzmaurice et al. [1995]. It is the approach of introducing graspable user interfaces (GUI) in daily work-life and show their effectiveness by a proof of concept. The idea behind it is the control of virtual objects with hardware, that inherently offers haptic feedback and can be attached and detached to virtual objects. The name "Bricks" is derived from the optical resemblance to bricks.

Fitzmaurice et al. [1995] claim the basic premise that the affordance of physical objects is richer than that of virtual handles, through direct manipulation techniques. The basic concept of Bricks is to overcome the problem of time-multiplexed systems and combine them with space-multiplexing. Time-multiplexing means that there is one control device which has different controlling functions over time. Fitzmaurice et al. provide the mouse as an example for time-multiplexing: the mouse can control only one element at a time, but can be used for different tasks. For example, windows can be resized and files or windows can be opened. This can all be done with one device, the mouse. Due to physical constraints it is only possible to perform one action at a time, which is the definition of time-multiplexing.

Time-multiplexing provides one control for all actions.

Space-multiplexing provides one control for each action.

Space-multiplexing in contrast offers one device or controller for each function, so that different actions can be performed simultaneously. They explain it using an example of a car, where you have a steering wheel, brake, clutch and gear shift, each controlling a different function, but to control the car a combination of different actions is needed at the same time. Bricks is now the attempt to overcome the time-multiplexing and the resulting interaction sequence by offering additional control devices for specific purposes.

Bricks provide specialized control devices.

The idea behind Bricks supports the theory that physical artifacts on top of the work space offer a specialized control device. This is tracked by a host computer which reads information such as position and orientation and redirects them to underlying programs. Hence the user is able to manipulate virtual objects with the bricks directly.

One brick can be used to rotate or move objects.

Fitzmaurice et al. distinguish between two interaction techniques to manipulate virtual objects, one handle and multiple handles. The one handle interaction can be used to rotate and move objects. A possible way of achieving this is the pairing of the virtual object and the brick that is placed on top of it. The virtual object mimics the behavior of the brick. If the brick is manipulated in either position or rotation, the virtual object is manipulated the same way.

Two bricks are used for complex operations.

Two or more bricks offer a wider variety of functions. A simple task could be the manipulation of virtual objects, by attaching two bricks to one object. One functions as an anchor which defines the base and the other one is used to manipulate the size of the object. By moving the non anchor brick the size of a square can be increased or even the shape can be changed. To manipulate the shape one brick is moved in a non-linear way. This behavior maps to the virtual world where the anchor functions as a holding hand and the second brick can be viewed as a hand pulling away from the other hand. The result is a manipulation of the object underneath the bricks.

Mimicking real world operations increases the intuitiveness of interactions.

2.3.2 Construction of Widgets

Prototypes are used in early development stages.

During a design process different stages are passed through. The early stages include software tests with low level prototypes. They are used to explore and test interaction concepts implemented in the software system. Two approaches for prototyping widgets are presented. *Sketch a TUI* by Wiethoff et al. [2012] is a technique to easily draw connections of widgets. Hereafter an approach by Hincapie-Ramos et al. [2011] that uses a touch mouse to sense touch input is presented.

Sketch a TUI

Sketch a TUI by Wiethoff et al. [2012] explains a method for creating early prototypes without knowledge of electronics and other fabrication techniques. They separate the design into two phases.

Objects are created from cardboard.

The first phase is to construct the body of the TUI. They provide a set of forms to create 3D objects from cardboard. Currently the library covers 26 different shapes as templates. A shape is selected and then printed onto a piece of cardboard. The lines indicate where the shape has to be cut or folded. The assembling is done by gluing the corresponding latches on the predefined areas. That process creates a shaped cardboard object. This object alone can not be used as a widget prototype. Marker and connections are missing to be sensed by the display.

Markers and connections are added with a conductive ink pen.

Phase two adds markers and connections to the widget. A pen with conductive ink as used in repairing broken circuits on boards is used to apply the conductive areas onto the cardboard. Lines and areas of conductive ink can be applied accordingly to the design of the widget and its purpose. The result is a shape formed with cardboard and enriched with conductive areas that can be used for testing software prototypes or interface concepts.

The benefits of this method are low production costs, only cardboard and a conductive pen. Furthermore, the construction process is not time consuming or complex and avoids the use of additional electronically peripheries.

Rapid Prototyping

A Microsoft Touch Mouse is modified with cloth to function as input device and widget simultaneously.

Rapid Prototyping of Tangibles with a Capacitive Mouse by Hincapie-Ramos et al. [2011] presents and describes a method of rapid prototyping touch sensitive input devices with different touch areas. They combine the widget with the touch input device by using a *Microsoft Touch Mouse*. The shape of the widget is modified by altering the shape of the mouse. The shape shifting is done by adding layers of cloth to the mouse. The layers can be plain or filled with cotton to reach a higher variety of shapes.

The Microsoft Touch Mouse offers 195 separated touch areas that can be connected to the clothing.

The surface of the mouse is touch sensible and has a grid of $13x15$ areas that can separately be detected. Each area of the grid can be connected to different capacitive areas embedded in the clothing. This adds touch sensitive areas to the surface of the clothing, and each touch area is mapped to a distinct area on the mouse. A one to one mapping can be achieved with a maximum of 195 different touch sensitive areas on the surface.

The Toki toolkit comprises and extension of the basic API for the touch mouse.

In addition to the modification of the touch sensitive mouse they present their own API, providing access to various functions of the touch mouse. One function is the extraction of touched areas from the mouse with unique identification. The API extends the basic software from the *Microsoft Touch Mouse*. The whole package containing a guide, for designing and constructing the clothing, and the extended API is called the Toki toolkit.

2.3.3 Optical Tracking

Optical tracking systems are based on image recognition and processing.

Optical tracking systems, as the name indicates, are based on camera tracking. Before capacitive displays became affordable to the research community, optical systems were broadly used. The *reacTable* by Jordà [2010] is an example for a successful optical tracking system. Another system is *SLAP* by Weiss et al. [2009]. Both systems use infrared light and tracking in combination with a beamer hidden underneath the surface. Due to the similarity between optical tracking systems only one is presented as a representative.

SLAP

SLAP uses Frustrated Total Internal Reflection and Diffused Illumination.

Silicone Illuminated Active Peripherals (SLAP) by Weiss et al. [2009] introduces a set of transparent, flexible widgets for optical based multi-touch tables. The set includes sliders, knobs, keyboards and buttons. The multi-touch table is based on Frustrated Total Internal Reflection (Han [2005]) and Diffused Illumination (Matsushita and Rekimoto [1997]). Infrared light is emitted into the surface and projected against it from underneath. The light reflected from objects and fingers is captured by an infrared camera placed under the surface.

Widgets are detected through white reflecting markers.

The widgets have markers at their bottom that can be detected by the system and an image recognition algorithm. The markers encode the position, orientation and ID of the widget through their size and positioning. The framework for touch input detection is the MultiTouch Framework[2]. Utilizing these information the system can now render a screen representation of the object on the table surface. This is done with a beamer located under the surface. With the help of this visualization the user can interact with the system by using the widget to manipulate the state of the system directly.

SLAP has a distinctive marker system.

SLAP is one important related work for this thesis, because our prototypes will be manufactured in a similar way, using acrylic and a defined setup of markers for the information transmission to the system. Contrary to optical based tracking, capacitive tracking enhances the possibilities of interaction even more. It is easier to construct widgets with complex shapes and add controls to them.

[2]`http://hci.rwth-aachen.de/multitouchframework`

2.3.4 Capacitive Tracking

In the last years capacitive multi-touch displays have become more present in the research community and replaced most of the optical tracking systems. In this section we introduce systems that use capacitive touch technology to construct widgets and communicate with system. *CapWidgets* by Kratz et al. [2011] introduce passive tangible controls. *CapStones and ZebraWidgets* by Chan et al. [2012] construct widgets similar to *SLAP*. *TUIC* by Yu et al. [2011] points out various ways of tracking tangibles on capacitive screens by using different marker types.

Optical tracking systems have been replaced by capacitive tracking systems in most research areas.

Enabling Tangible Interaction

Enabling Tangible Interaction on Capacitive Touch Panels by Yu et al. [2010] describes and presents two different approaches to sense tangible objects on capacitive touch screens. The differentiation is between spatial tags and frequency tags. The first method is the tag design based on spatial domain. Spatial refers to a unique design pattern of the markers on the widget. The system can detect the widget by analyzing the pattern formed by the markers.

Spatial sensing uses the position and arrangement of markers.

Frequency tags use the time domain instead of the spatial domain. Only one marker is required to communicate data to the system. The marker is activated and deactivated electronically to generate a frequency of touches. The time based pattern can communicate different aspects of the widget to the system.

Frequency tags communicate the data over a series of touches on the same location.

The spatial approach is limited through the fixed arrangement of the markers, and the time based approach requires a fixed amount of time to submit the data at least once. A combination of both approaches result in a reliable traceable widget. The spatial domain is used to provide the position and orientation and the frequency markers additional information.

A combination of both approaches enriches the interaction.

CapWidgets

CapWidgets by Kratz et al. [2011] present widgets specially designed for small touch displays such as smart phones or tablets. CapWidgets enrich the multi-touch surfaces by adding haptic feedback in form of controls. As the user touches the widget the system detects the widgets markers. One example widget is a rotary knob made from aluminum.

Additional controls do no always improve performance.

Furthermore they conducted a study to test the relative performance of the prototypes versus touch controls. The experiment has shown that users prefer direct touch instead of physical controls on small devices and that they do perform better with direct touch.

CapStones and ZebraWidgets

Sensing the height of a stack of blocks through marker design.

CapStones and ZebraWidgets (Chan et al. [2012]) presents two types of widgets for capacitive displays. The first one is called CapStones and consists of blocks that communicate their position and 3D arrangement to the system. Up to three blocks in height can be detected. The more blocks are stacked, the more markers on the bottom CapStone will be activated. It can show two, three or four marker and therefore the system can differentiate the various stacking heights. The height of the sensing can be adopted by adding more markers to the blocks. The blocks are only visible when they are touched, otherwise they are invisible to the system.

Touch sensitive controls to alter virtual objects.

The second widget type consists of two controls, a slider and a knob, called Zebra Dial and Zebra Slider. The controls are similar to the ones introduced by Weiss et al. [2009], except that they are for capacitive multi-touch displays. These controls also only work when being touched.

TUIC

TUIC is a technology that enables tangible interaction on capacitive multi-touch screens without the need to manipulate the underlying hardware. TUIC is the advancement of *Enabling Tangible Interaction* by Yu et al. [2010]. It enhances and refines the three approaches presented in this paper: the spatial, frequency and hybrid approach. The spatial tags consist of three positioning markers and a set of markers to encode the ID of the tag. The frequency tag has an active circuit integrated that modulate a touch frequency. In contrast to spatial tags, frequency tags have a start up delay based on the fact that the encoding is time based and it needs a fixed time to read the whole time encoded ID. The hybrid approach reduces the number of markers needed to encode the ID, but has the same start up delay as the frequency marker. Another difference between the spatial and frequency tag is, that the spatial tag has an orientation, can be moved and needs no external power supply.

TUIC enhances the spatial and frequency approach.

Tangible Drawing Tools

Using tangible drawing tools on a capacitive multi-touch display by Blagojevic et al. [2012] presents a set of widgets that supports drawing on capacitive multi-touch displays. The interesting part is not the drawing application itself but the implementation of the widgets. The detection is separated into two phases: the learning and the recognition. The first phase includes the learning of a widget. The markers are arranged alongside the arbitrary shaped drawing tools. The learning algorithm measures the distances between each pair of markers and stores it. The recognition on the other hand evaluates the present touches and searches for patterns of distances that match a learned widget.

Widgets are detected by a two phased model.

Phase one is the learning and phase two the recognition.

2.4 Summary

The field of capacitive multi-touch displays is a very active topic in Human Computer Interaction (HCI) research communities. Important existing works have been presented. This section establishes an overview of the important characteristics extracted from the previous work and which of them are enhanced.

Position and frequency markers are the two different approaches that are used to identify widgets. We only use position markers for the whole widget. In addition to all existing work, we differentiate between to types of position markers, active and passive ones. The widgets presented in the papers use active markers only. Therefore the widgets are not detected if no one is touching them. Adding passive markers makes the widget traceable even if it is not touched by a user. This bridges the gap between the virtual representation and reality - If a widget is placed on a system, the system will show the virtual representative, and if there is no virtual screen representation, than there is no widget.

Spatial markers are differentiated into passive and active.

In contrast to *CapStones and ZebraWidgets* (Chan et al. [2012]) we give detailed instructions on how to implement widgets of different types, widening the range of widgets from two to an unlimited amount of different widgets. We provide a tracking and footprint generation algorithm that can easily be adapted to any widget. Furthermore, we differentiate between two types of markers, active and passive ones. In contrast to the slider or knob introduced by Chan et al. [2012] our slider is even detected when no one is touching the widget. Additionally we provide interaction concepts of using capacitive touch technology as shown in chapter 5—"Design Space".

Yu et al. [2011] mention that spatial tags require a lot of markers to encode lots of different IDs into widgets. In our approach, instead of coding the ID into a fixed number of bits, we encode it into distances between a set of determined markers. That makes our ID encoding much more flexible and we use less markers for the same amount of information. Based on the limited number of simultaneous touches that can be detected by today's devices, this enables the possibility of placing and using more widgets on one display at the same time.

Chapter 3

Widget Construction

" Everything should be made as simple as possible, but not simpler. "

— Albert Einstein

The construction is separated into a theoretical and a practical phase.

The construction of widgets is a multi-stage process, covering theoretical and technical aspects. Before a widget can be built, a model of the widget must be created. Each widget needs to be adapted to specific design constraints and a design theory. In the first part of this chapter the design constraints for capacitive multi-touch tables are investigated. The second part covers the building process, including the presentation of suitable materials. Concluding example builds are presented.

3.1 Constraints of the Widget Design

Before building a widget, a design theory is needed.

Before something can be built a plan or blueprint is needed. Designing widgets raises questions before even starting to create the blueprints. The first question is the size of the marker used in the widget. What size can be detected? Which size is too large to be useful? Which distance is needed between the markers? Furthermore, what is the best layout to communicate information to the system? A widget needs an orientation, position and identification. All these questions have to be answered before starting to design a widget. This section investigates the different questions and provides a design theory for widgets on capacitive multi-touch displays.

3.1.1 Marker Size and Distances

The right marker size is important during the design process.

The marker size is an important factor in the design process of a widget. If the markers are too big, they will occupy more space than necessary and therefore occlude more screen space. On the other hand if the markers are to small, they will not be reliably tracked by the system. The optimal marker size is so small that it is the smallest reliably detected one, occluding the smallest, possible screen space as possible.

An experiment was conducted to gather the optimal marker size.

To determine the optimal marker size an experiment was conducted. Screws of different sizes and with different head shapes were mounted on a piece of wooden cardboard. The flat side of the head functions as the contact area an the thin side as the connection area. All connection areas were connected with a wire. Touching one screw activates all other screws, passing the capacitive change through the wire. The cardboard was placed on an iPad and the touches were visualized on a computer screen and logged. Each screw had a dedicated position on the screen, allowing a dedicated mapping of a touch and a screw. The screws were activated multiple times. Thereafter the log files were processed by an algorithm to compute the overall percentage of the touches for each screw. The threshold was a 95% rate of detection. The result was that a diameter of 1 centimeter was the smallest fitting size, reaching the threshold. Down to 8 millimeters the detection was quite good but not as reliable as required.

The result of the experiment yielded a one centimeter diameter.

CONVENTION:
All measurements, determinations and tests were performed on an iPad version one. Each capacitive touch system is designed slightly different, so it may be possible that the values have to be adapted for other systems.

The second challenge was to determine the optimal distance between to markers. If two markers are too close the system might merge them to one touch or dispose one of them. Another experiment was conducted to investigate the optimal distance between two markers. A set of screws with heads of one centimeter in diameter were placed in acrylic. They were pairwise placed with different distances between them. Each distance increased the former distance by one millimeter. The markers were activated simultaneously, logging the appearing touches. An algorithm analyzed the log files to find the pair closest to the 95% threshold. Under one centimeter of distance the touches were often merged or one of the two touches changed the position a bit every time. In conclusion the optimal distance between two markers is one centimeter or more. The centimeter is measured between the borders of the markers and not the center.

A second experiment was conducted for the optimal marker distance.

3.1.2 Marker Types

The interaction of a widget and a multi-touch system requires the communication of specific information. Without the size or position of a widget the screen representation can not be created to fit the physical counterpart. Therefore it is essential that the position of the widget is communicated to the system. The markers represent the only way to communicate with the software. Furthermore, when interacting with more than one widget the systems has to differentiate the widgets. In conclusion, each widget has to provide the following information: the position, the orientation, the size, and a unique identification.

The widget has to provide information such as position and ID.

The markers are divided into three categories, each providing a different set of information. The position markers allocate the position, size and orientation of a widget. The identification marker facilitate the unique ID of a widget. Lastly the state markers provide information about the state of the control elements on the widget.

The markers are divided into three categories: position marker, identification marker and state marker.

Position Marker

The size is computed by the distances of the borders.

The position of a widget is extracted from the touch points, if at least one marker is placed at the edge. This marker indicates the origin of the widget. The size can be calculated by computing the distances between the borders. That results in placing for markers in each corner of the widget. One will be the origin marker and the other three are used to calculate the distances for the size.

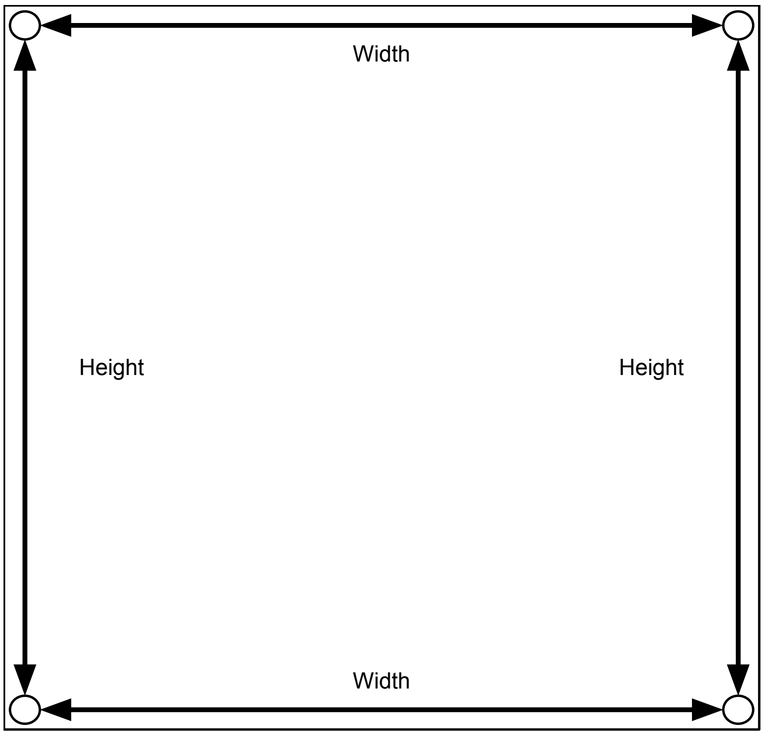

Figure 3.1: The first position marker design.

The position markers are sufficient for the position and size.

Figure 3.1 depicts a sketch of the design approach. The image illustrates that two distances are available for each size attribute. Furthermore, the design does not clarify which marker is placed at the origin. This leads to the elimination of one marker. All except the origin marker could be deleted. The top right marker was chosen for various reasons. The remaining three markers, as depicted in figure 3.2, create a coordination system with the origin marker as $(0, 0)$.

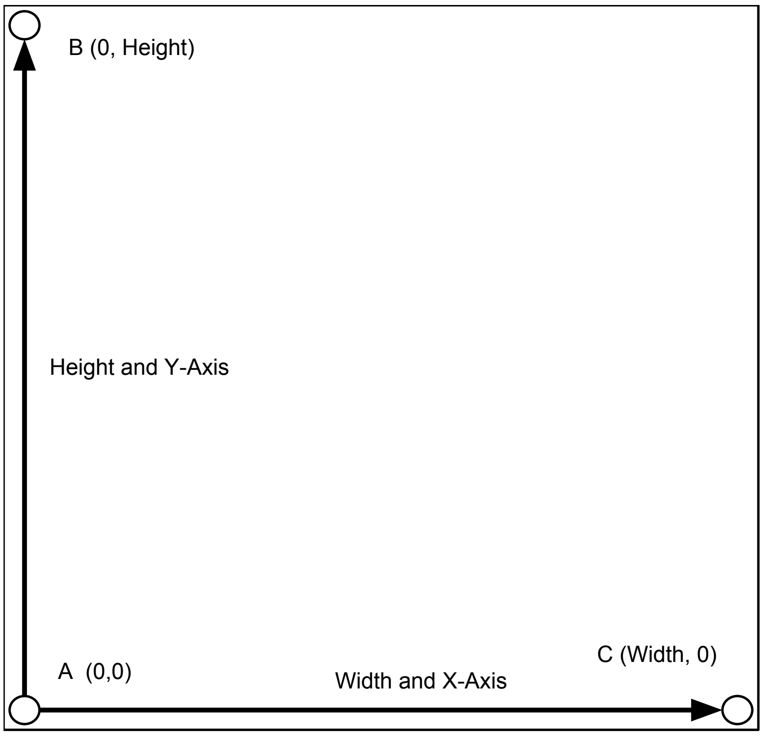

Figure 3.2: The position markers A,B and C can communicate the position and size to the system. Furthermore they create a coordination system in the size of the widget.

The orientation can be computed from the position markers by calculating the angle between the line formed by markers A and B and the x-axis of the device, as depicted in figure 3.3.

Identification Marker

Identification markers provide a unique ID.

The widget is now able to communicate position, size and orientation to the system. This information would be sufficient if only one widget is used at a time. As soon as more than one widget is placed onto the capacitive multi-touch display an identification is needed. Otherwise the system can not distinguish between different widgets. Each identification has to be unique.

An 8-Bit encoding needs 8 markers.

The first approach was the design of an eight bit encoded number, offering 256 possible IDs for the identification.

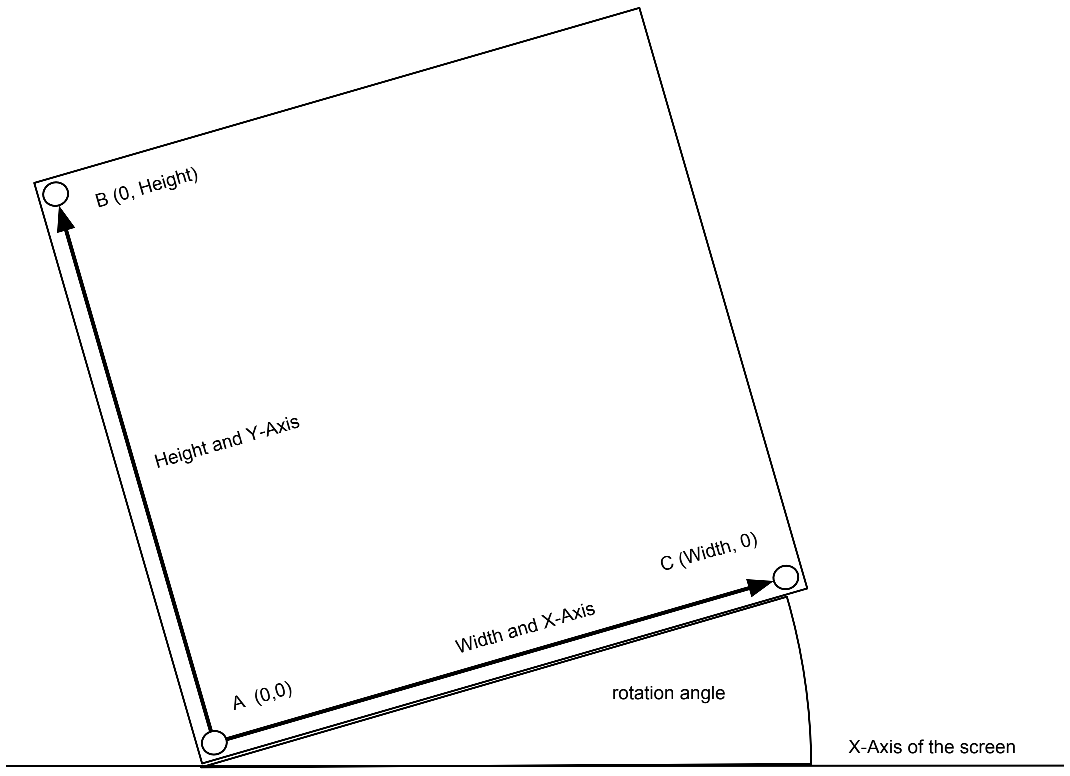

Figure 3.3: The orientation is calculated by the angle between the x-axis of the input device and the x-axis of the widget. The resulting angle is the degree of rotation that has been applied to the widget.

This approach has the drawback that eight more markers have to be added to the widget, occluding more screen space.

Each widget can be identified by a set of unique distances.

Analyzing the widget and the possibilities offered by freely placeable markers, the approach of unique distances was chosen. Identification by unique distances works as follows: each widget has a unique set of distances. They are computed between the three position markers and additional identification markers. By varying the position of the identification marker, new sets of distances can be created. The widget detection analyzes the the set of touches present on the surface, computes the distances for each point and compares the results to the given sets. Figure 3.4 depicts the usage of one additional identification marker. The unique ID of the widget is given by the three distances BD, AD and CD.

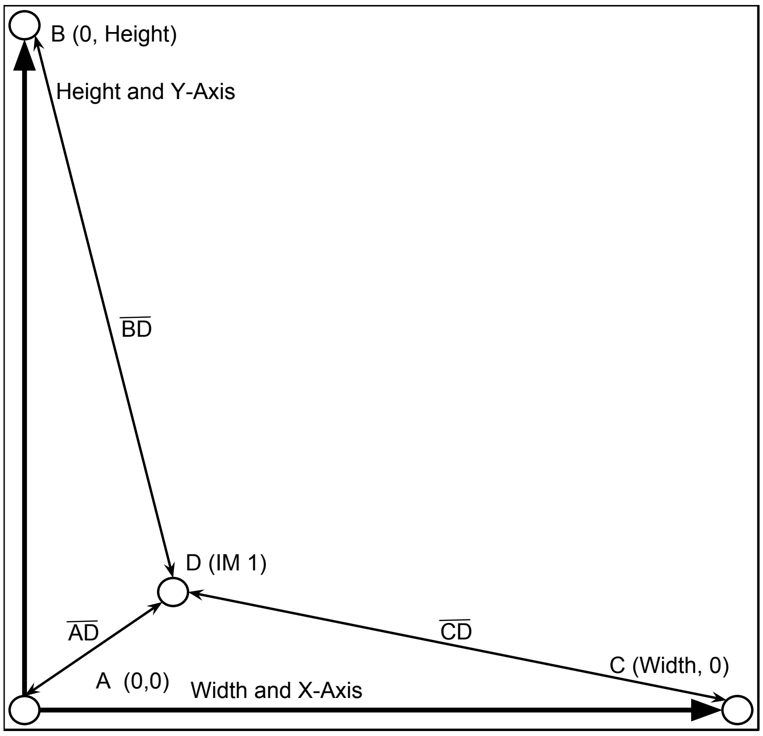

Figure 3.4: Marker D is an identification marker and can uniquely be described by the three distances BD, AD and CD.

State Marker

Each widget can be enriched with various controls, such as sliders, buttons and knobs. The existing markers are not sufficient to transmit information about the state of a control. A third category of markers is responsible for the information providing of controls. They communicate the state of a widget to the system, therefore the third category is called state marker.

State markers provide information about the state of a control.

A button has two states, either pressed or unpressed. One state marker can provide the information about the two states, by either being touched or untouched. Depending on the control it may be necessary to add more than one state marker for a control. A knob for example needs two state markers, one at the center of the knob and one for its current position. With the two markers a line can be com-

The number of state markers for a control depends on the functionality of the control.

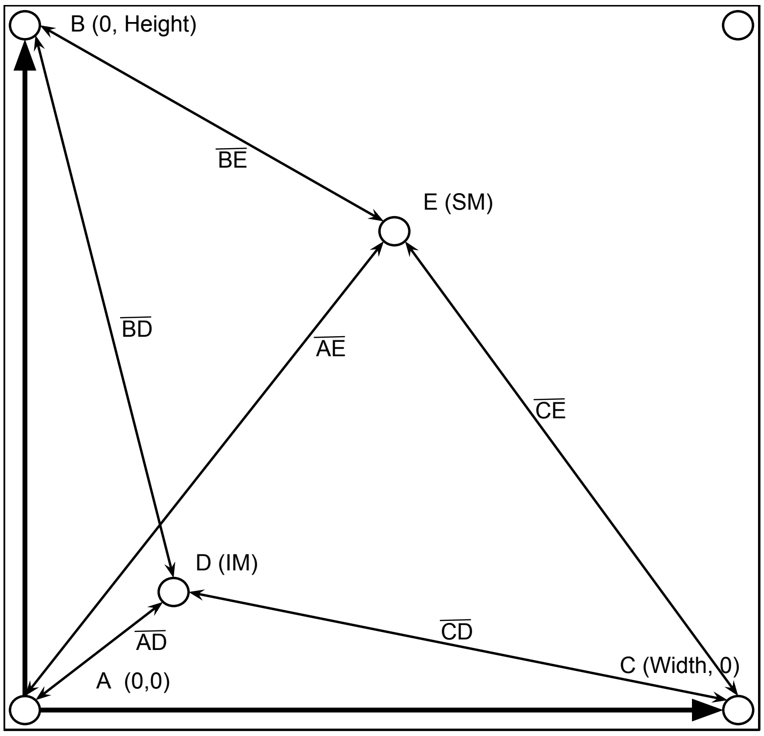

Figure 3.5: Marker E is a state marker. It can be identified by the unique distances to the position markers. The top right marker is just added for stability reasons of the physical object.

puted. The angle of the line in comparison to the device x-axis can be calculated, representing the current state of the knob.

State markers are identified analogous to identification marker.

The marker can be identified in the same way as the identification marker. The distances to the position markers as depicted in figure 3.5 are calculated and provide a unique footprint. If an additional touch, besides the position and identification marker, is present in the coordination space of the widget, the associated control is found by comparing distances.

Active and Passive Markers

Considering the interaction with widgets on a multi-touch display, widgets are often placed on it and not operated all the time. If a widget is not touched the markers will no longer be traceable by the system, due to the fact that the capacitance is not manipulated by external influences. To avoid the undesired vanishing of widgets some of the markers have to be visible all the time.

Unused widgets are not detected by the system.

ACTIVE AND PASSIVE MARKER :
Active marker are the markers, that will be visible to the system when touched. They are used for buttons and other two-state functions. Contrary passive markers are always visible to the system, independent from touch.

Definition:
Active and Passive Marker

Other systems like TUIC by Yu et al. [2011] use complex circuits to emulate touches. Since passive markers do not need to change their states, it is sufficient to have them visible all the time. Their capacitance relative to ground has to be changed permanently, ideally without modifying the display or adding powered circuits. Therefore an additional wire is attached to the widget connecting it to ground. The markers connected with ground are always visible to the system, given that the contact area of the marker is big enough, as described earlier.

Grounding a marker makes it permanently visible to the system.

This completes the design theory for our widgets. In the next section we will introduce an overview of the construction phases. Additionally a set of materials that are suitable for building widgets for capacitive multi-touch displays is discussed.

3.2 Construction of Widgets

The construction is divided into three phases.

The construction of widgets is separated in three major phases. The first phase decides the design and functionality of a widget. The second phase is about the materials that are used to build the widget. At last the third phase is the manufacturing and assembling of the single parts.

3.2.1 Phase One - Design

The design starts with setting the attributes of the widget.

The construction of a widget starts with phase one. Phase one is defined as the phase in which the design is decided. To conceptualize a widget all functionality must be considered. A few questions that should be answered in this phase are:

- How large is the widget?
- What is the shape of the widget?
- How many control elements are embedded?
- What functionality will they have?
- How many markers are needed for each function?
- Which markers are active and which are passive?

A widget design example with two buttons and one slider.

This list only gives a hint on the basic facts that need to be settled before the drawing can be done. Thereafter the construction plan for the widget is drawn. An example for phase one is the design of a widget with two buttons and one slider. The state of the buttons is either on or off. One marker for each button is sufficient to represent the two state functionality. The slider on the other hand is always visible. It needs a passive marker for the sliding element. The widget will have a rectangular shape. This provides all information to draw the construction plan of the widget.

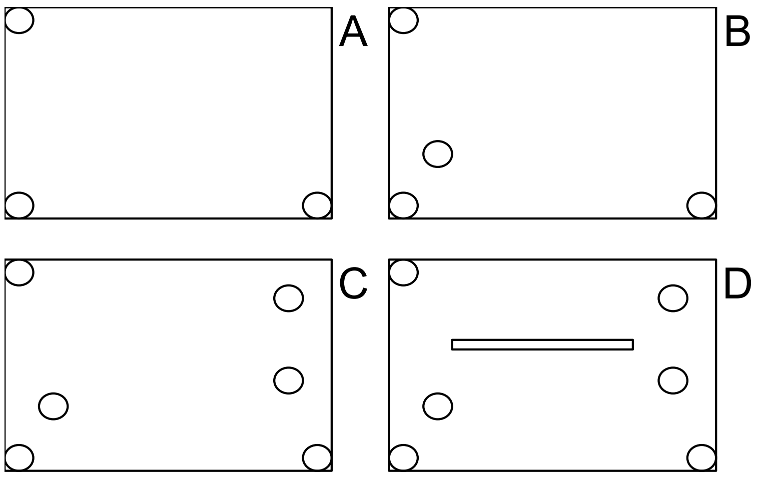

Figure 3.6: These images illustrates the design of a widget with two buttons and one slider. (A) The position markers are added, (B) identification marker, (C) button marker, (D) slider rectangle.

The first things to add are the rectangle for the shape and the three position markers, as depicted in figure 3.6 (A). The next elements are the identification markers. One identification marker is placed near the origin marker. Only one identification marker is needed, since this is a prototype that is not used with many other widgets simultaneously. The result is depicted in figure 3.6 (B).

Position and identification marker are added first.

The two buttons are placed on the right side of the widget at the same x coordination. They only differ in the y position (C). This decision is made to provide unique attributes for the two button markers. Later, in the detection process of the software, each marker needs a special attribute to identify it. This is later explained in detail in 4—"Framework Implementation". For now it is sufficient that each marker needs a unique attribute. The unique attribute for the button markers is the fact, that they are the rightmost pair of touches with an identical x position.

Each marker needs a unique attribute to be identified.

The last step is to add a rectangle where the slider marker is added. The slider is positioned left of the button markers (D). The unique attribute for the slider marker is that it is the leftmost marker, after subtracting the position and identification marker. This example illustrates the theory behind the design of a widget. First of all it is important

The work that is done before the construction saves work later on.

to know every function beforehand. Each control is then added step by step to the base widget, always thinking about a unique attribute for every added marker. Ignoring the unique attribute can yield in a lot of implementation work when writing the dedicated footprint generating function for the widget. Consider the slider is placed between the two buttons. If the slider is then moved to a specific location it would be possible to have three points at the same x coordination that only differ in their y coordinate. Therefore the unique attribute is essential to identify the markers later on.

Capacitive widgets need no offset between the widget and the surface.

In contrast to optical tracking systems like SLAP (Weiss et al. [2009]) we do not need to include an offset between the body of the widget and the table. The tracking is not based on optical perception and therefore the body of the widget is not detected as a touch, unless it is built out of conductive material.

3.2.2 Phase Two - Material

Some materials are optimized for high end prototypes - others for low cost.

During the prototyping phases for widgets we encountered a few helpful materials that can be used to construct widgets. Some are for high end widgets and some are suited for low cost prototyping. When choosing the material the design of the widget should be kept in mind - which constraints must be fulfilled when building the widget? This section will present a set of materials that have been used to build prototypes. Each material is briefly described and advantages and drawbacks are pointed out.

Body

Wood pulp board is suitable for early prototypes.

Wood pulp board was used for very early prototypes. The price of the material is very low. Furthermore, wooden pulp board is sliceable by knifes and a ruler. Design tests and usability tests that require physical hardware are the common uses cases for wooden pulp board. Using it in productive prototypes is not recommended. The wooden pulp board is quite flexible and therefore not always flat.

This results in markers not being connected to the surface if the wooden widget is placed onto the surface.

Acrylic is cheap, fixed, transparent and bendable when heated.

Acrylic is a good choice to create the body element's from. It has various advantages such as its transparency, durability, stability and it can easily be cut. Furthermore, it is possible to reshape acrylic when heating it and applying soft pressure. The transparency is a benefit in contrast to non transparent materials. It reduces screen cluttering and allows to show the screen representatives directly underneath the widget. That assures the correct mapping of widget and screen representation when using multiple widgets.
Acrylic can be cut using a basic saw, or if available, a laser cutter for high precision. Dynamic relabeling (Weiss et al. [2009]) is possible, this means assigning different functions to the same button over time, displaying the current function directly under the button.

ITO is transparent glass or acrylic with embedded transparent conductive parts.

ITO is the abbreviation for indium tin oxide, a half-conductive, mostly transparent chemical substance. This chemical is used to create transparent and conductive materials. It is applied to glass or acrylic under high temperatures and pressure. The patterns of the indium tin oxide layer can be chosen as needed to create complex conducting paths for special purposes. Transparency and conductivity make this the best material for the markers and wires when designing a completely transparent widget. Due to the high costs of this product, it is hardly feasible to buy custom made glasses with fitting conductive lanes. Therefore a completely covered sheet will be cut to fit the needs. For prototypes cheaper alternatives like normal acrylic combined with copper foil were used, and ITO is only used for high end widgets.

Marker and Connections

Magnets are good to enlarge touch areas or connect widgets.

Magnets can be used for early prototypes to enlarge the touch area on the surface. They are easy to detach and attach. Furthermore, when designing widgets that offer the possibility to be connected to one large widget, magnets are perfectly suited as connectors.

Aluminum foil in combination with wood pulp board for rapid prototyping.

Aluminum foil is a very cheap and easy obtainable material that can be used in combination with the wooden pulp board for rapid prototyping. It can be ripped or cut into the needed shape and it is sufficient for both markers and connecting lanes.

Self adhesive copper foil is used for connections.

Copper foil is an improvement to aluminum foil, it is more durable and has better conductive characteristics. There is a wide variety of copper foil with glue on one side that can easily be attached to prototypes. The glue layer is small enough for not disturbing the conductivity.

Wires are used for rapid prototypes. They can fastly be attached and detached.

Wires are another cheap and broadly available source for connection lanes. In contrast to copper and aluminum foil, wires are smaller and therefore do not occupy as much visible space on the surface as the foils. Attaching and detaching is quickly done, which makes remodeling of early prototypes cheaper and more simple.

Screws are used as markers.

Screws were used as markers during the prototyping phase. They exist in different sizes and shapes. The ones with a hexagonal head are the best fitting, they do not have a hole that can disturb the sensing of the touch due to a lack of contact. If the hole is too large, the iPad will detect the touch as a disturbance and will simply ignore it. For more information about the marker size see section 3.1.1—"Marker Size and Distances".

3.2.3 Phase Three - Engineering

Phase three is the final phase of each iteration of a widget prototype. The assembling of the widget is straight forward to the design made in phase one and the materials picked in phase two. At first the widget body is cut and all holes and other cuttings are made according to the construction plan. The markers are applied and all passive markers will be connected with conductive material, and one of them gets a longer connection. That connections is grounded. The connection of all passive markers makes it obsolete to connect every passive marker to ground.

The construction phase is an assembling of the manufactured parts.

3.2.4 Overview of the Three Phases

1. Phase - Design
   (a) Purpose of the widget
   (b) Picking the base shape of the widget
   (c) Add position and identification marker
   (d) Add shapes for the controls
   (e) Divide markers into active and passive
2. Phase - Material
   (a) Choose materials for the base
   (b) Choose materials for the marker
   (c) Choose materials for the connections
   (d) Choose materials for the controls
3. Phase - Engineering
   (a) Prepare the single elements
   (b) Assemble the elements
   (c) Add the connections

3.3 Construction Examples

This section introduces two examples of widget design and construction. A widget with one button and a slider widget are presented, including the whole manufacturing process.

3.3.1 Button

Collecting the basic facts for the widget in phase one.

The first step contains the design phase. Answering the questions provides us with the knowledge to create the construction plans. The widget's size is eight centimeters for width and height. Eight centimeters are sufficient to place the position and identification markers with one additional state marker for the button. All markers would fit on a smaller base shape, but the usability would decrease. Grabbing and touching distinct parts of a widget is harder if the widget is small. The widget has only on control element, which is a button. The button is a two state button: pressed or unpressed. Position and identification marker are passive markers and the state marker is active. All important facts are gathered, so we can continue to create the construction plan.

Placing the elements onto the base shape in phase one creating the construction plan.

Three position markers and one identification marker are placed into the quadratic shape of the base. The identification marker is located near the origin, as depicted in 3.7 (A). The button's state marker is placed in the center of the widget. Its unique attribute is, that it is the last unknown touch point after position and identification marker are detected (Figure 3.7 (B)). Additionally a marker without function is placed in the top right corner. It's purpose is to stabilize the widget when placed onto a surface and prevent dangling. This concludes the first phase.

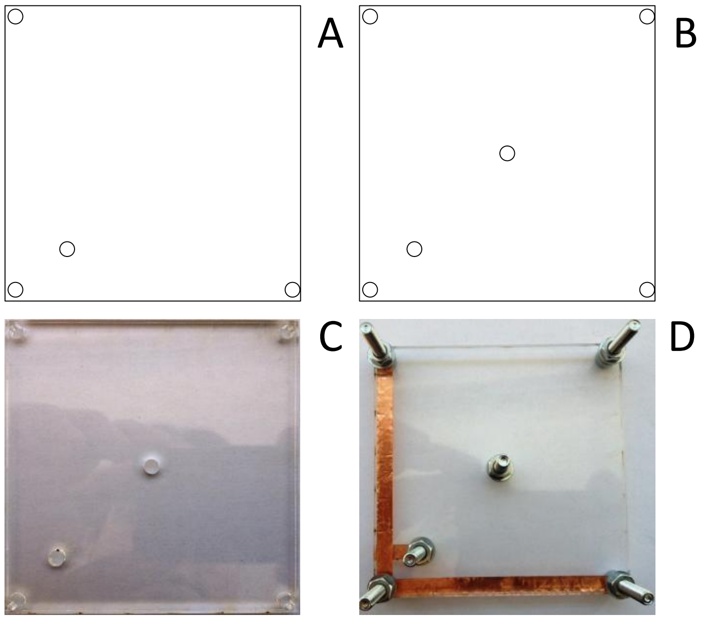

Figure 3.7: A depicts the position and identification marker. On B the state marker is added and a spot for a stabilization attachment. C shows the result of the laser cutting and D finally shows the finished widget.

The body is made of acrylic. This provides a solid but transparent body. The widget representation is displayed under the widget, therefor transparency is important. The markers consist of screws with a head of one centimeter of diameter. Connections are made from adhesive copper foil.

Phase two is the selection of materials.

The base shape is cut with a laser cutter. A laser cutter offers a high precision and fast cutting. The result is depicted in figure 3.7 (C). The three position markers and the identification marker hole are connected with copper foil. The copper foil is also slightly inserted into the holes. When the screws are inserted for the markers they establish a direct contact to the copper foil in the hole. The screws are tightened with nuts. The button marker is not connected to the other markers, since it is an active marker. The last step is to attach the grounding wire. The completed widget is depicted in 3.7 (D).

Phase three completes the construction.

This concludes the first construction example of a widget with a single button.

3.3.2 Slider

The second example is a slider with a guiding rail and a track.

The second widget example is a slider. Following the three construction phases we start with the design of the widget. The purpose of the widget is a widget with a linear slider control. A rectangle is selected as base shape for the widget. The position markers will be placed in the three corners and the identification marker close to the origin marker. The slider is more complicated to design than the button of the previous example. We want a slider that is completely made of acrylic instead of a single screw. The sliding of a screw on the display could have unwanted side effects like scratching the display.
The slider needs a guide rail on the widget, limiting the positions and fixating it on the widget. This results in the design shown in figure 3.8.

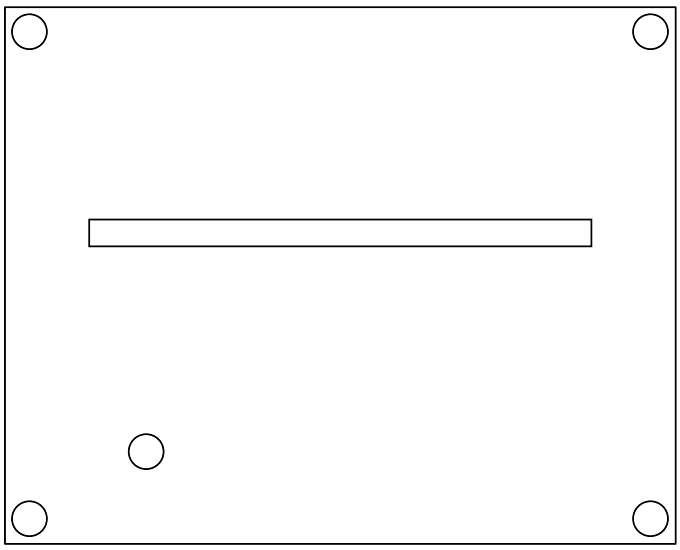

Figure 3.8: The base plate of a slider widget, with a guide rail for the slider element.

The unique attributes for the position and identification markers are the same as introduced in phase one. The slider marker's unique attribute is the fact, that it remains the last element on the widget. The position of the slider can be calculated by measuring the distances to the three position markers, this technique is called triangulation. The slider-knob is fixated to enable a smooth interaction without replacing the slider-knob all the time. The slider-knob is shaped like a cross, one part reaching down to the display, the other one up as a handle for the user. The two arms are fixated in a track, consisting of two elements. One has the width of the arms and the second one, which will be placed above, has the width of the guide rail. The three elements are depicted in figure 3.9.

The unique attributes of each marker are defined.

The slider-knob is fixating through a track on the guiding rail and its cross shape.

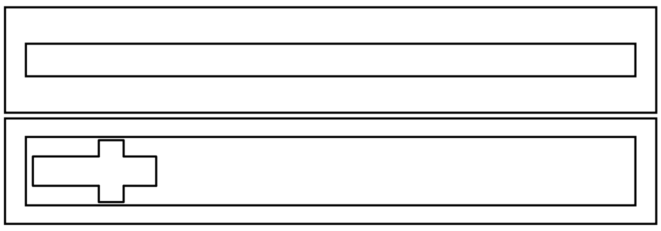
Figure 3.9: The cross is the slider marker, the two other elements will be used to fixate the control on the widget.

The next step is to select materials as described in phase two. The position and identification markers are represented by short screws, the connection is done via copper foil. The slider control will be completely made out of acrylic, adding copper foil to ensure the capacitance of the slider. The assembling result of the acrylic is depicted in figure 3.10.

The materials are picked considering the demands of the widget and elements.

The next chapter will introduce the software implementation of the framework that tracks the widgets.

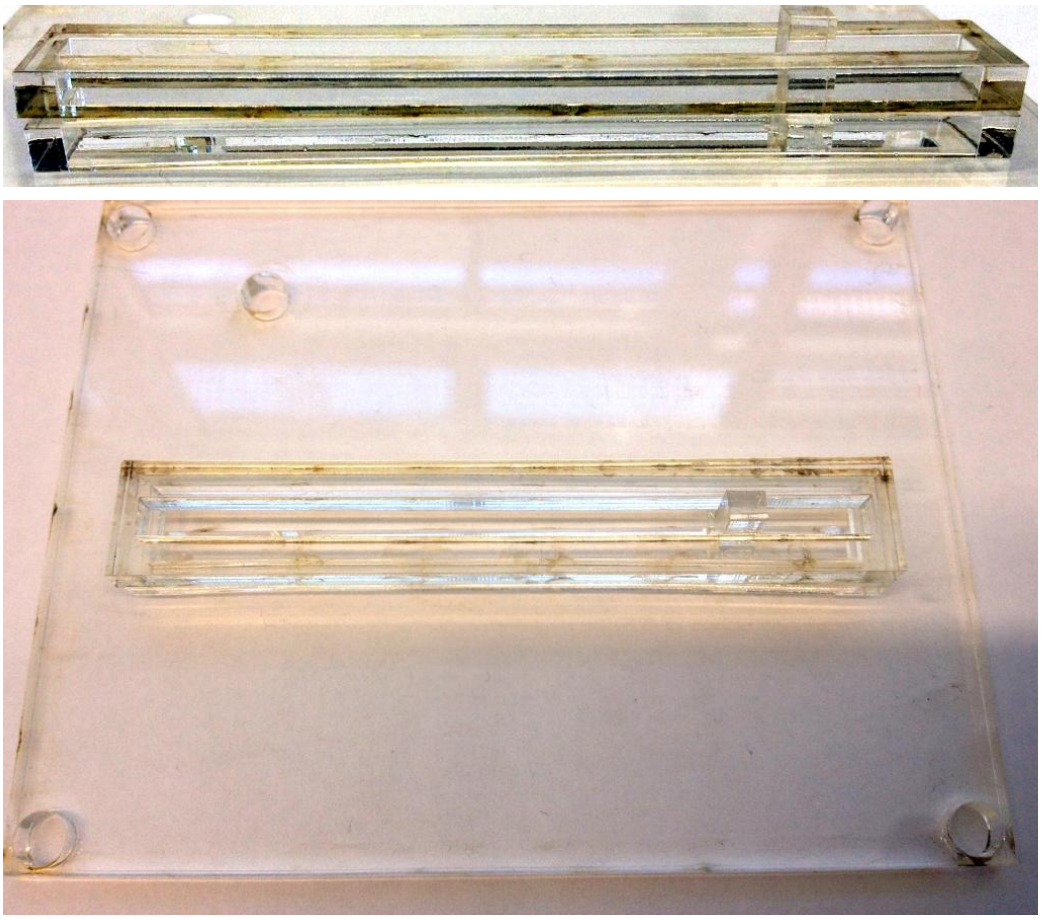

Figure 3.10: The top picture shows the slider fixation and guide rail. The second picture shows the whole widget, before attaching the markers and connections.

Chapter 4

Framework Implementation

" If we knew what it was we were doing, it would not be called research, would it? "

— Albert Einstein

The interaction between physical objects and a capacitive multi-touch table involves different parties. On the one hand there is the hardware and on the other hand there is the software. The software itself is divided into different sub-parts. One part is the *MultiTouch Framework* that provides the basic functions for touch interaction and graphical visualization. The `footprint generation` stores a unique definition for every widget, calculated from the touch point footprint. The `widget detection` uses these definitions to detect the widgets when placed on the surface and call the appropriate functions if controls are used. The `widget object` is used as a container inside the software to store the widgets. It contains the important attributes of the widget, for example the last position of each marker. The following chapter presents the different software aspects involved in a widget multi-touch table interaction.

The widget multi-touch table interaction involves a software and hardware side.

4.1 MultiTouch Framework

The MultiTouch Framework is composed out of three sub frameworks.

The *MultiTouch Framework* developed by the media computing group[1] from the RWTH-Aachen University, is the core component of the implementation. The `widget framework`, consisting of the *footprint generation* and the `widget detection`, has been integrated into the *MultiTouch Framework*. The core framework comprises three parts: the `TableEngine`, the `TouchServer` and the `GLEngine`. The `TableEngine` includes the other two frameworks and is the main entry point of the software.

4.1.1 TouchServer

The TouchServer collects and unifies touches from different sources.

The `TouchServer` is used by the `TableEngine` to communicate with different sources of touch input. It can gather touches from different capacitive displays, mouse and track-pad emulation, XML files or optical tracking systems. The `TouchServer` separates the touches in three different states: touch began, touch moved and touch ended. The semantic of each source is unified to fit one structure, independent of the source. Furthermore, it provides a dictionary for each touch state that can be used by other functions to react on touch input. The class `TSTrace` is used as representative for touches. It stores information about the state, position, size if available and historical information about the touch, like former positions and generation.

4.1.2 GLEngine

The GLEngine provides predfined shapes and a full-screen view.

The `GLEngine` is the core of the graphic processing of the framework. It provides different views that can be used by the `TableEngine`. It loads the configuration file from the `MultiScreenAgent` and adapts the view to the settings made there. It provides a canvas, fitting the size of settings, embedded in a full screen view. Furthermore, it drives the

[1] `http://hci.rwth-aachen.de/multitouchframework`

clock that refreshes the views in a specific interval. Additionally it provides a set of predefined geometries like text, rectangles, circles and many more. The basic shapes are used by a call of the shape class. It reduces the OpenGL code that has to be written.

4.1.3 TableEngine

The TableEngine distributes touches to registered objects.

The `TableEngine` combines the `TouchServer` and `GLEngine` and adds an event management. It collects touches from the touch server and distributes them to classes that are registered as an observer. This is done in the `TableEngine` class. Our `footprint generation` and `widget detection` are hooked into this class. The entry point is chosen to intercept the incoming touches before they are distributed to virtual objects. Every touch that can be assigned to a widget is removed from the distributed touches. Each touch is only used once using this concept.

The widget detection is called in the TableEngine.

The `TableEngine` framework provides a class `TEObject` which can be considered as a collection of objects that can be displayed. It consists of various other `TEObjects` or basic shapes from the `GLEngine`. It is basically a container object to create more complex objects from simple ones. Figure 4.1 shows a composition of `TEObjects` that form the screen representation of a widget. This example illustrates the nesting of `TEObjects`. The outer box is the parent `TEObject`, the top bar is another object with a text texture. The slider consists of two objects, one for the line and one for the movable slider object, that mimics the real world slider. Each button is another object with a texture. Figure 4.1 demonstrates how complex objects can be built with simple shapes.

Nesting TEObjects can create complex shapes and views.

This framework offers everything we need to implement our solution. It provides the touches from different touch screens in a unified representation, an easily usable structure to create and manipulate objects and direct access to the touches. The next section describes the widget object and what attributes are stored in it.

The MultiTouch Framework integrates all components needed to implement a widget detection.

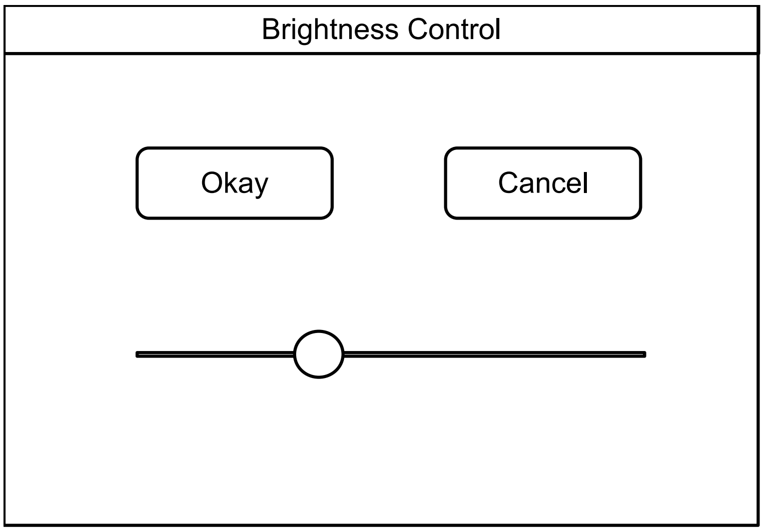

Figure 4.1: This example illustrates the nesting of `TEObjects`. The outer box is the parent `TEObject`, the top bar is another object with a text texture. The slider consists of two objects, one for the line and one for the movable slider object that mimics the real world slider. Each button is another object with a texture. This demonstrates how complex objects can be built with simple shapes.

4.2 The Widget Object

The widget class is the core data structure of the widget detection framework.

The widget object and therefore the widget class is one of the core components of the widget detection system. It stores the data during the footprint generation and supplies the widget detection with all necessary data to detect and identify the widgets. It is a highly customizable class to reflect the different aspects of the various widget types.

The widget object provides different attributes.

General information about the widget are stored in the widget object, for example the name or the position of all markers. Furthermore, it offers attributes to store the orientation and the screen object it is bound to. The orientation is recalculated in every execution of the widget detection. Together with the orientation the size of the rectangle is calculated and stored. The bound or paired object is only set if the widget is being paired to an on screen object, otherwise it is null.

In addition to the general information attributes the widget object provides arrays for each kind of marker. For the minimal functionality it includes the position marker and the identification marker array. The identification marker array is an unsorted collection of dictionaries. In contrast to the identification marker array, the position marker array has a logical ordering. The first entry needs to be the origin position marker, the bottom left corner marker of the widget. The second entry is reserved for the top left corner position marker and the third entry is for the bottom right marker. This ordering is necessary to refer to these position markers, knowing which one is which, in order to calculate the orientation and the underlying rectangle without errors. This ordering could be replaced by an algorithm that checks the position of every marker when calculating data based on the ordering. It could check the position of the marker for the same unique attributes as explained later in chapter 4.3.1—"Position Marker Detection".

The markers are stored in arrays. Each marker is a dictionary.

The position marker, in contrast to the other marker arrays has an ordering.

Each entry of a marker array is a dictionary. A dictionary is a collection of (key, value) pairs. It is a convenient and efficient way to address data with an arbitrary key. Each key in a dictionary must be unique, therefore the addressing is absolute.
The value object type can change for each object and is not limited to one object type per dictionary. The design of the dictionaries is the same for position and identification markers and can differ for other types of markers, depending on the information for the marker that are important to store. The dictionary for position and identification marker consists of two (key, value) pairs. In one pair, namely the *position* key, the information about the last trace is stored. This is done by storing the `TSTRace` as value. The second pair consists of the key *distances* and an array with all distances to the other markers. The relations are depicted in figure 4.2.

Each marker is a dictionary. The distances are stored as arrays.

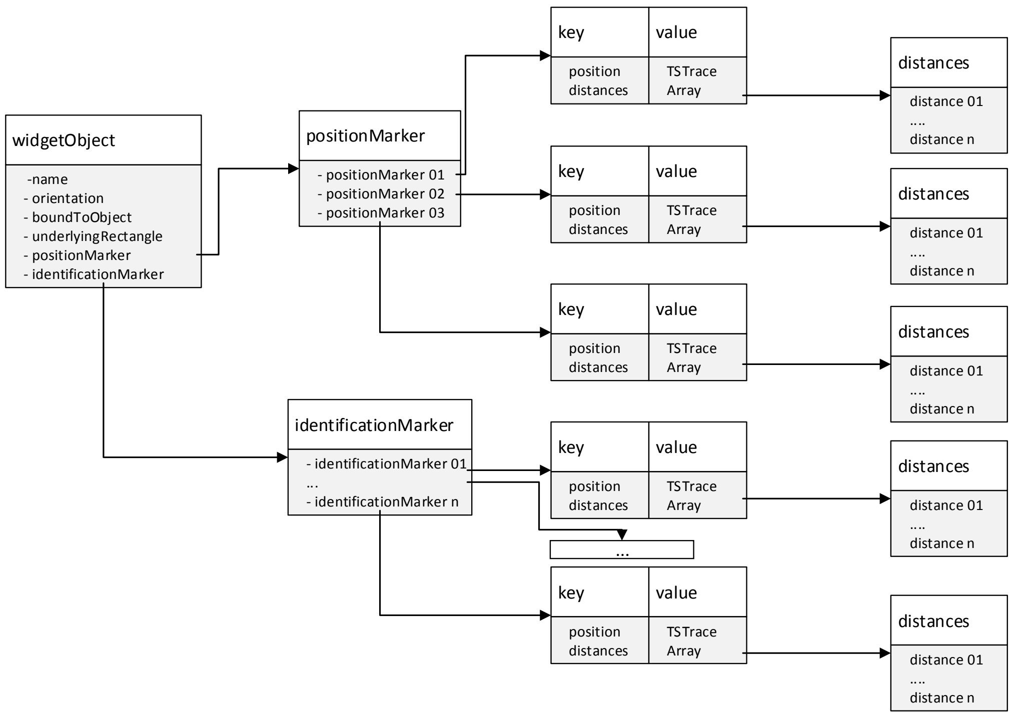

Figure 4.2: This figure shows the composition of a widget object with the focus on the marker storage. Every marker category is saved in an array. This array contains all markers of the specific type. Each marker consists of a dictionary in which the position and distances are stored. All distance elements in the dictionary are arrays, holding all distance values.

4.3 Footprint Generation

The footprint is created for every widget.

The footprint is the arrangement, including the distances, of all markers. The footprint generation creates a footprint for widgets. It is stored in a file that can than be used by the widget detection. Every widget needs a footprint file, otherwise the widget detection is not able to detect it.

The widget detection intercepts the touches before they are passed to virtual objects.

The footprint generation, if enabled, is hooked into the update routine for touches in the `TableEngine`. Before another component of the framework gets the chance to handle the touches the routine for the widget detection is called, including the footprint generation. It ensures that none of the touches created from a widget is detected as a normal touch and side effects are minimized.

All current touches are stored in a list. Ended touches are removed from the list.

The widget detection receives a dictionary with touches in three states: beginning, moving and ending. Only touches that are beginning or have been moved are interesting for the tracing of widgets. The dictionary entries of these touches are inserted into a list, ignoring the two different states. The detection does not differentiate between these two states, only the present touches are used to compute footprints, independently if the touch has been there before or not. The important data is the current set of touches. Touches of the third state, the ending state, are removed from the list of present touches.

Capacitive screens do not recognize shapes and orientation of markers.

Systems like SLAP (Weiss et al. [2009]) use the size and shape of a marker for identification and orientation of the widget. Since most of the capacitive screens are not able to detect the size or shape of the touch, we can not rely on this information. Basically the capacitive screen handles each touch as a small touch area on the screen, independent of size and shape. Capacitive screens detect touches only on intersections of sensing lanes. If the lanes are close enough together it is possible to detect shapes by analyzing the touch patterns. As it is now, the touch screens use grids that are not precise enough to extract shapes. This means we only receive the information where the touch has occurred, consequentially we need to take a different approach. We consider the incoming touches as a point cloud and extract the different markers with different criteria, as described in the different subsections.

The footprint generation needs the number of expected touches for each widget.

A footprint can only be generated if all markers are present and detected on the screen. Otherwise the result of the footprint generation would describe another widget. The footprint generation waits for the expected number of touches to appear before it generates an output. The expected touches can be calculated beforehand for every widget. For position and identification marker it is one touch for one marker. They are passive markers, allowing the constant tracking of the widget. This results in the preliminary formula $expectedTouches = PM_{count} + IM_{count} + x$ to calculate the expected touches. X denotes the missing touches from the state markers.

The touch count has to be calculated for each widget individually.

The touches for the state marker can not be generalized for every control, because each control is designed in a different way. A button needs one state marker whereas a rotary knob needs two state markers. Additionally a button consists of one active marker and the rotary knob of one active and one passive marker. Furthermore, a control can be designed in different ways. For example, a slider can be realized in at least two different ways. It either has one state marker and the position is calculated by triangulation from the coordinate space created by the position markers. Alternatively it consists of three markers – one for the slider and one for the start and end of the range. In the second approach only the distances between $start_{current\_position}$ and $end_{current\_position}$ have to be measured to calculate the position of the slider.

A control can be designed in different ways.

It denotes that each state marker type is calculated differently. The introduction of new control types implies a new state marker type and the extension of the calculation routine. The final formula to calculate the expected touches is $expectedTouches = PM_{count} + IM_{count} + \sum SM_{count}$.

The markers count is provided as arguments to the constructor of the footprint generation.

The count of each marker is is passed to the function to calculate the expected touches. The constructor has to be adapted every time new state marker types are added. An example constructor call:

```
initWithPMCount: (int)pmc IMCount: (int)imc
ButtonCount: (int)bmc SliderCount: (int)smc
DoubleSliderCount: (int)dsc KnobCount:(int)kmc
DemoMarkerCount: (int)demomc widgetName:
(NSString*)name.
```

The call provides the number of state markers of the different categories. If a state marker is not present on the current widget a zero is passed as argument. However, position and identification marker may not be zero. The position marker count is always three and the identification marker argument always greater or equal than one. The footprint generation can compute the number of expected touches from the arguments. A call

The sum of expected touches is calculated with the passed arguments.

```
-(id)initWithPMCount: 3 IMCount: 2 ButtonCount: 3
SliderCount: 2 DoubleSliderCount: 0 KnobCount:0
DemoMarkerCount: 0 widgetName: @"DemoWidget"
```

results in $expectedTouches = 3 + 2 + ((3 * 1) + (2 * 3)) = 14$.

The algorithm for generating the footprint waits for the correct number of touches to be in the list of touches. If the number of touches matches the calculated sum, the single detection functions are called. A widget file is generated if every marker has successfully been detected. The algorithm to detect the markers can be simplified as follows:

The algorithm generates a file when every marker has been detected.

```
WHILE no widget is detected DO
   IF calculated marker count = touch count
   THEN
     Position Marker Detection
     Identification Marker Detection
     State Marker Processing
     IF all markers have been detected
     THEN
       Storing the Widget as XML File
     ELSE
       do nothing
   ELSE
     do nothing
END
```

4.3.1 Position Marker Detection

Before the actual footprint generation starts with the position marker detection, the number of present touches is compared to the calculated amount of expected touches. If the number of touches is greater than expected, more than one widget or additional touches are present. The algorithm does not continue with the generation. If it continued it could lead to unpredictable results. Which of the touches does belong to the widget and which does not? This question can not be answered by the program.

The algorithm continues only if the right count of touches is present.

On the other hand if the count of touches is smaller, one or more marker are not working correctly or are not touched to be visible. If the touches fit the number of expected touches the algorithm proceeds, otherwise it compares the counts again when the amount of touches changes.

An incorrect number of detected touches can indicate a defect widget.

CONVENTION:
A widget is always placed onto the surface for the footprint generation in its original orientation. The origin marker is placed in the bottom left corner.

Each position marker is detected by its unique attribute.

The first step of the footprint generation is the extraction of the position markers from the list of touches. Position markers, as described before, are in three corners: top left, bottom left and bottom right. They have unique attributes that are used to identify each position marker as depicted in figure 4.3. The marker in the top left corner has the unique attribute that its y coordinate is the maximum of all touches that belong to the widget. The bottom right position marker has the maximum of all x coordinates and the marker in the bottom left corner has the minimum of all x and y coordinates.

Detected markers are added to the dictionary of the widget.

A successfully detected touch is removed from the list of touches as every touch can only be assigned to one marker. Each point is stored in a `TSTrace` inside a dictionary and added to the widget object. Hereafter, if all position markers are identified and the corresponding touches have been deleted, the algorithm continues with the identification marker detection.

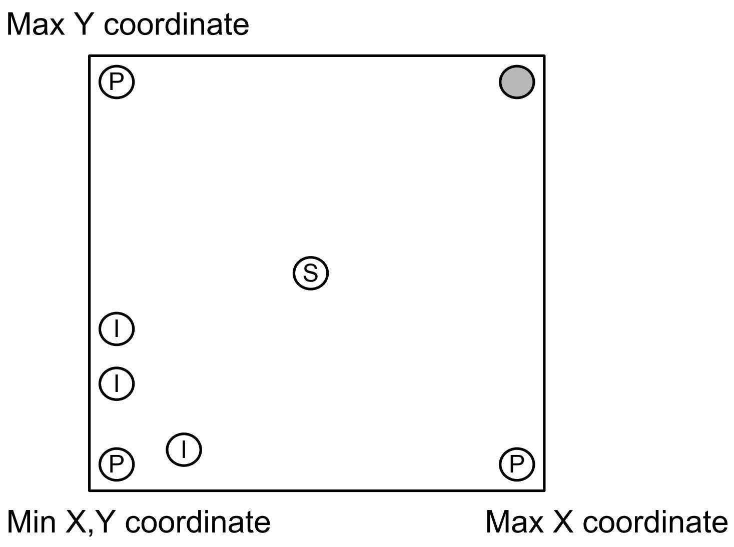

Figure 4.3: Each Position Marker has a unique attribute in terms of a maximum or minimum coordinate.

4.3.2 Identification Marker Detection

The point cloud from figure 4.3 is reduced from seven $(3PM + 3IM + 1SM)$ to four touch points as depicted in figure 4.4 by the grayed out markers. The next step in the footprint generation algorithm is the identification marker detection.

A distinct area near the origin marker is reserved for the identification markers. As presented in figure 4.4 the three identification markers have the touch points with the smallest x and y coordinates. An intuitive approach is the search for the smallest x and y coordinate within the point cloud. This may lead to different results, depending on the order of the search. One marker can have a smaller y coordinate and another marker has a smaller x coordinate. Depending on the search order they would be found in a different sorting. If only the identification markers are found, the arrangement is meaningless, but in a few cases it would be possible to find elements that are no IM. Placing a control element to the far left side of a widget and all identification markers right would yield the state marker as a IM when searching for the smallest x coordinate first.

IM can not be searched by either looking for the smallest x or y value.

Instead of searching for a marker with the smallest coordinates on each axis a geometrical aspect is used to determine the identification marker. The area of the widget can be divided into two areas A and B as depicted in figure 4.4. One area contains the identification marker and the other area all state markers.

The widget area is divided into two areas.

Each sum of the x and y coordinate in area B is now larger than every sum of area A. Based on the fact that area A has the form of a triangle with a right angle and equal side lengths. Each point on the dividing line has the same sum of the x and y coordinate. Starting from $(0, 100 = 100)$ over $(20, 80) = 100$ to $(60, 40) = 100$ ending in $(100, 0) = 100$. The numbers are only chosen to demonstrate the concept. They may vary depending on the widget, but the basic idea is the same. Every point below the line in area A has a smaller sum than 100, otherwise it would be in area B.

Instead of taking single coordinates the sum of x and y is used.

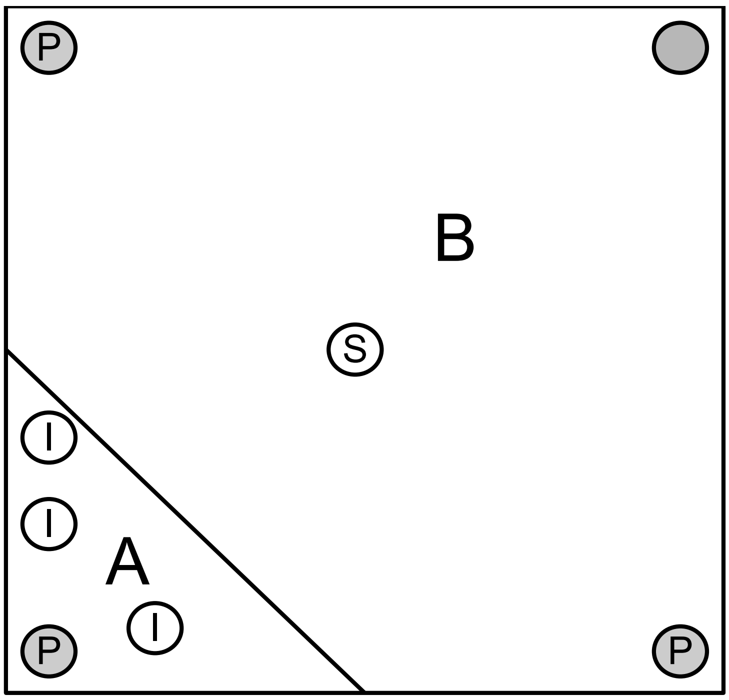

Figure 4.4: The markers that have been detected so far are grayed out. The widget space is separated into two areas. (A) is the area with the identification markers and (B) holds the state markers.

Each IM is detected by its sum and stored within the widget object.

The algorithm iterates as often as IMs were specified through the remaining points in the cloud to find the touches with the smallest x and y sum. The order of the points is irrelevant, since they are only used to calculate distances between each other and the position markers to uniquely identify the widget. Each identification marker is stored in a `TSTrace` and added to the widget object.

Unique distances are used to identify a widget.

The widget is identified by calculating the distances for each position marker compared to all identification markers and other position markers. In the given example this would result in $3 * 5 = 15$ distances, three position markers compared to five other markers. The distances are stored in a dictionary with names describing which distance they describe as depicted in figure 4.1. Table 4.1 summarizes the unique attributes of each position and identification marker.

Marker Name	Unique Attribute
PM 2	Maximum Y Coordinate
PM 3	Maximum X Coordinate
PM 1	Minimum Y AND X Coordinate
IM	Smallest Sum of X and Y Coordinate

Table 4.1: Unique attributes of each PM and IM.

4.3.3 State Marker Processing

State marker detection can not be unified.

State markers, in comparison to position and identification markers, differ from widget to widget, depending on their function. The process of identifying state markers can not be unified with one specific algorithm. Instead we introduce the main concept and present an example.

Semi unique attributes become unique after removing touches in previous stages.

The tracking should already be considered during the design phase. Each state marker needs a unique or semi-unique attribute to be identified. Semi-unique means that the attribute of the marker becomes unique when processing other markers, reducing the number of points within the cloud. This is similar to detection of identification markers. They can only be detected in a reliable way after processing the position marker. The attributes of the identification marker are only unique when the position marker points are removed from the cloud. Otherwise the origin position marker has the same attribute as a identification marker.

The unique attribute of a button is actually semi-unique.

An example is a button which is a one state control. One state marker is added to represent the button on the widget. The buttons semi-unique attribute is, that it is the last remaining point within the cloud. This attribute becomes unique in terms of fitting points in the cloud, when all position and identification markers are detected and removed. Thereafter only and exactly one point fits the semi-unique attribute, making it unique.

State markers are identified by unique distances.

Following the example of the button widget, the distances between the state, position and identification markers are computed. The distances are stored within the widget object in the button marker array. If a new touch appears during a widget is placed on a capacitive display, the distances between the new touch and the detected PM and IM is calculated and then compared to every state marker of the widget. Each state marker is identified with unique distances – two state markers on one position can not exist.

The state marker detection is divided into three phases.

The detection of the state markers is summarized in three steps. The first step is the definition of unique or semi-unique attributes for every state marker during the design phase. The second phase includes the implementation of a function that searches for the unique attributes of the newly added state marker type. A function for the example with the button is a function that returns the last point in the list of touches. Complex unique attributes afford more effort to implement a search function. The final step is the calculation of the distances between the state marker and the position and identification marker.

A rotary knob is much more complex to detect.

A more complex example is a rotary knob. It consists of two markers: one is the center of the knob and the other one is the position of the rotary arm. Both markers are passive and therefore visible all the time. Passive markers allow the system to read the initial state of a widget when it is placed on the surface and every time the value is needed. During the detection the knob has to be rotated to differentiate between the knob and the rotary arm marker. The knob marker is fixed whereas the rotary arm marker changes its position. The unique attribute of the knob marker is the fixation to one position. Two touches remain in the point cloud and the one not changing its position is the knob marker. The rotary arm marker is the last remaining touch within the point cloud.

The second example illustrates that the algorithm can not be standardized.

Another approach to generate a footprint for a knob is the usage of fixed positions. The knob has to be placed on the display in the same position every time a footprint is generated. For example, the arm is always left of the center. The unique attributes depend on the position of the touches. The knob is always the rightmost remaining marker and the rotary arm is always the leftmost. This illustrates the diversity in implementing the footprint generation for state marker and amplifies the non standardization of the algorithm.

4.3.4 Storing the Widget

The data gathered in the footprint generation is used by the widget detection. If the program is shut down the temporary saved data from the widget object is lost. A permanent way to store the footprint of a widget is the storage in a file in a nonvolatile memory.

Footprints are stored in files on the hard disk.

The data hold in a widget object consists predominantly of markers. Each marker has a type and an array with distances. We choose the XML-File format to save the footprint. The structure of the file is closely related to the footprint generation. A node structure is created for each marker type. Listing 4.1 presents an example of a fully generated footprint of a widget with one identification and state marker. The node trees are visible within the file. One for each position, identification and state marker. Each tree has sub entries for every marker of this category. For example, there are entries PM01 to PM03 and each one represents one position marker. Inside of each specific marker there are the unique distances to each position and identification marker. The name of the widget is provided in the file name. Further information such as size and orientation are computed from the position markers during runtime.

A XML-File holds the widget data in a tree structure.

```
<Widget>
    <PositionMarkerList>
        <PM0>
            <distance>115.663307</distance>
            <distance>0.000000</distance>
            <distance>374.386169</distance>
            <distance>366.267670</distance>
        </PM0>
        <PM1>
            <distance>288.321014</distance>
            <distance>374.386169</distance>
            <distance>0.000000</distance>
            <distance>525.632019</distance>
        </PM1>
        <PM2>
            <distance>314.162384</distance>
            <distance>366.267670</distance>
            <distance>525.632019</distance>
            <distance>0.000000</distance>
        </PM2>
    </PositionMarkerList>
    <ButtonMarkerList>
```

```
        <BM0>
            <distance>242.297333</distance>
            <distance>341.646027</distance>
            <distance>364.539429</distance>
            <distance>181.245697</distance>
        </BM0>
    </ButtonMarkerList>
    <IdentificationMarkerList>
        <IM0>
            <distance>0.000000</distance>
            <distance>115.663307</distance>
            <distance>288.321014</distance>
            <distance>314.162384</distance>
        </IM0>
    </IdentificationMarkerList>
</Widget>
```

Listing 4.1: The XML File for a Widget with one button and one identification marker

This concludes the footprint generation, we have shown how the design constraints are integrated into the footprint generation. An algorithm for the marker extraction and identification has been presented, followed by a file structure for widget data.

4.4 Widget Detection

The widget detection loads footprints and searches for a widget within the point cloud.

The widget detection loads a list of widget footprints, specified by the user. These widgets are constantly searched among the incoming touches. If a pattern matches a footprint the corresponding widget object is updated. The current position for each marker is inserted, the orientation and size are calculated. For each loaded footprint a `TEObject` is created that represents the virtual representation. The size and position are updated each time a widget is detected. The widget detection loads the specified footprints. For every footprint the whole point cloud of incoming touches is analyzed to find a match for the origin position marker. If the origin marker is found the other two position markers are searched. Only if all position markers are detected the algorithm continues to search for the remaining markers. If less than three position markers are found, the algorithm continues with the next footprint.

The search algorithm stops if not all position markers are found.

The algorithm in pseudo code:

```
LOAD footprints
receive point cloud
WHILE touches exists DO
   ITERATE over loaded footprints
      search for the origin marker
      IF a candidate is found
      THEN
         search for second and third PM
         IF all PMs are found
         THEN
            search IM
            search SM
            compute Size, Orientation
            update virtual representation
            update widget object
      ELSE
         do nothing
END
```

4.4.1 Loading the Footprint

The stored footprint is read by the `WidgetManager`. The position and identification markers are processed. Every footprint file has three position marker elements and at least one identification marker element. The information provided by the footprint file are converted into arrays and dictionaries and added to a widget object. Thereafter the footprint file is analyzed for other state marker types. Through the tree structure of the XML-file the top nodes can directly be accessed, identifying the marker type. The state markers are processed depending on the type. If the structure is the same as the basic type-distance structure it is processed analogous to the position markers. Customized types can require additional functions to parse the footprint data. Each marker is added to a corresponding marker list within the widget object. Finally the name of the widget object is set to the file name of the footprint. On a successful load the `WidgetManager` returns a widget object, containing all markers in the associated arrays.

Each marker type is processed independently.

4.4.2 Position and Identification Marker Detection

A widget is considered as detected if all passive markers are present.

The `WidgetManager` passes each loaded footprint as a widget object to the widget detection. Receiving an updated touch point cloud triggers the detection algorithm. Each loaded footprint is compared to the current point cloud, searching a candidate for the origin marker. When no origin marker is found, no widget can be found. A widget needs all three position markers. The origin markers servers as a first criteria if it is possible that a widget can be detected within the point cloud. The other position markers are searched one by one, if the origin marker is detected. As soon as one position marker is not found, the current widget is skipped and the next one is analyzed. If all three position markers are present the algorithm continues with the identification markers. If no state marker is a passive marker, the widget is flagged as found if all identification markers are found. Thereafter the widget object is updated. If some state markers are passive markers, than they are added to the minimum marker that must be found for a successful detection. An unsuccessful run flags the object as undetected.

The origin marker is searched by iterating over all touches and comparing the measured to the footprint distances.

The origin marker search starts with the first touch in the point cloud and computes the distances to every other point in the cloud. Based on the loaded footprint a set of distances for the origin marker exists. Every new computed distance is compared to the set of required distances. If a distance is a match, the distance is deleted from the given set. If all distances are deleted from the current set, the point is considered as the origin marker. A match is based on the difference between the measured and the stored one. Since the measurements appear not to be as accurate as needed we have added an offset of three percent. The offset of the comparison adds a range to the compared values. If one distance is in the range of the other distance +- 3% it is a match ($distance_1 < distance_2 * 1.03 \wedge distance_1 > distance_2/1.03$). If the distance array is not empty after comparing all points, the next point is set as starting point. This procedure is repeated until a widget is found or every point has been checked.

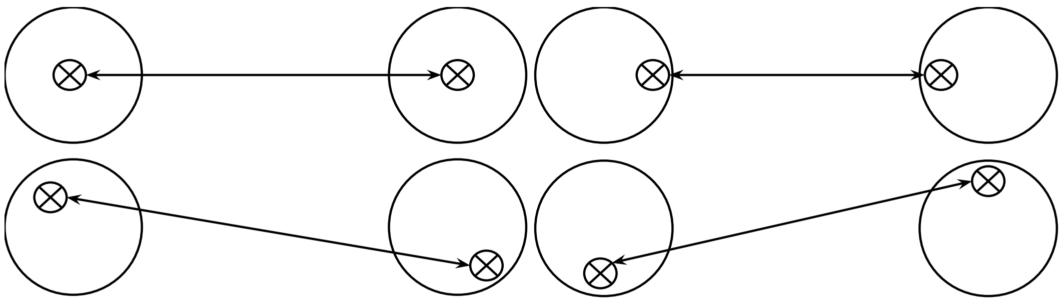
Figure 4.5: Approximation of the center of a touch affecting the calculated distance between the two markers. The distance in each case differs from the others. The center is depicted by the cross inside of a circle, the large circle represents the whole marker size.

The accuracy of the distances depends on the fineness of the display sensor grid.

The accuracy is influenced by the capacitive multi-touch display controller. Depending on the part of the marker that is connecting to the intersection of the grid the distances may vary. If a set of two markers with a fixed distance is placed onto a capacitive screen multiple times, the difference differs from time to time. This behaviour is depicted in figure 4.5.

4.4.3 State Marker Detection

State marker are identified by their unique distances.

The state marker detection can extract a list of possible state markers of the widget object. If a new touch appears the distances to the position and identification markers are compared to the distances of each state marker. A match indicates which state marker is activated and the system can call the implemented function for that control. The detection itself is split into the different control element types. Each type of state marker has a proper function, since it is not possible to generalize the detection. Even the distinction between active and passive markers can not be generalized, because it is different for every design.

Each state maker type is detected by a proper function.

The state marker detection iterates through the different kinds of state markers. Empty state marker arrays are ignored, only if a marker is present it is processed. The processing of a state marker calls the corresponding detection method for that marker type. The method analyzes the point cloud to find the state marker. The detection function differs from type to type. They can be identical, most commonly the control elements differ too much to be unified.

A button widgets point cloud has either one or zero remaining touches.

If a widget has an embedded button, the point cloud, after removing the points from the position and identification markers, is either empty or has exactly one point within the coordination space of the widget. An empty point cloud indicates that the button is not pressed. Contrary if one point remains within the point cloud, the detection algorithm computes the unique distances to each position and identification marker. Thereafter the result is compared to the stored distance set of the button state marker. A match indicates a pressed button and the corresponding action is called.

The common approach uses unique distances.

Some marker require different approaches.

The common approach to find state markers is the comparison of the distances from the remainders in the cloud to the given distance sets for each marker category. One exception that we encountered is the position of the slider marker or the position of the rotary knob marker. For these two kinds of state markers exists no predefined set of distances to the position and identification markers. A set can only be precomputed if the marker has a fixed position, but the two markers can be moved according to the design. These markers are detected through a different approach. The slider state marker is placed between two passive markers and the distances between the border markers and an existing point between is measured. The point between the slider markers is the slider knob and the distances are used to compute the absolute position of the slider.

4.4.4 Screen Representation

The widget detection creates a foundation for the screen representations.

The screen representation is only partially integrated into the widget detection. The actual screen representation differs from application to application. The widget detection only provides a foundation for it. Each attribute for a screen representation is computed and stored within the widget object. For each loaded widget a screen object is created. An example representation is depicted in figure 4.6. This object is a `TEObject` composition. The object is flagged as visible if the widget is detected and if it is not detected, the visibility is set to hidden. This means if the widget is detected the object is shown on screen. If it is not altered there is a plain rectangle in the size and orientation of the widget. The application itself is responsible for decorating the object to the needs of the program and the controls of the widget. The implementation may vary among different applications.

The final layout is done by the application using the framework.

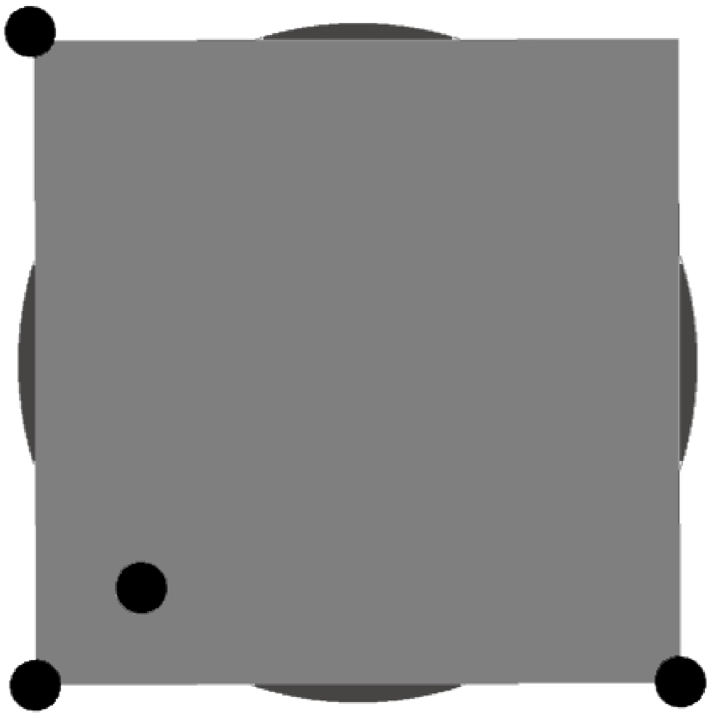

Figure 4.6: The widget detection provides a draft screen representation for each loaded footprint. This model represents a widget with an underlying menu. The hidden menu is depicted by the circle below the rectangle. The dots represent the detected markers of the widget.

Chapter 5

Design Space

" 'Facts, facts, facts,' cries the scientist if he wants to emphasize the necessity of a firm foundation for science. What is a fact? A fact is a thought that is true. But the scientist will surely not recognize something which depends on men's varying states of mind to be the firm foundation of science. "

— Gottlob Frege

Capacitive touch screens provide new interaction concepts. The range of sensors is extended by conductive materials offering new possibilities. A user can "touch" a display with the help of a conductive wire from a distance. This principle is used when creating widgets for capacitive screens. A user does not need to touch the table directly, he can touch it through other objects. These objects called widgets can limit the number of actions or enforce specific ones by applying physical constraints and affordances. Widgets can be used to utilize different functions. The interaction with widgets is divided into explicit and implicit touch on one or multiple touch area controls.

5.1 From Touch to Widget

Traditional touchscreen interaction uses a finger to create touches.

Capacitive multi-touch displays allow the detection of a human body touch. A user touch is the connection of the skin and the surface of the capacitive display. The body part, commonly a finger, changes the charge of the capacitance at a specific point on the surface as explained in 2.1—"Capacitive Touch Technology". The traditional touchscreen interaction is performed with one or more fingers on the surface of a multi-touch table directly as depicted in 5.1.

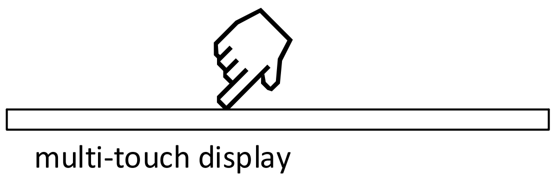

Figure 5.1: The traditional touchscreen interaction is performed with one or multiple fingers directly on the display.

The concept of capacitive multi-touch displays allows us to extend the range of a finger touch by adding conductive elements. The capacitive change that is caused by a finger can be transported through conductive materials. A screw for example is similar to a finger in its shape. Conductivity and resemblance in shape is not enough to generate a touch. If a screw is placed on a capacitive display, the display does not detect a touch. Neither is the charge of the intersection at this position changed. The screw alone is only extending the range of the sensors at the contact area. When a finger touches the screw the charge of the sensors is changed. The capacitive change is passed through the screw downwards to the display. This sensor range extension is true for all conductive materials.

A screw can extend the range of the display sensors.

The next step is the combination of two conductive materials. A screw is placed on the surface and a wire is attached to the screw. A user touching the screw or the wire creates a touch. The position where the user touches the wire does

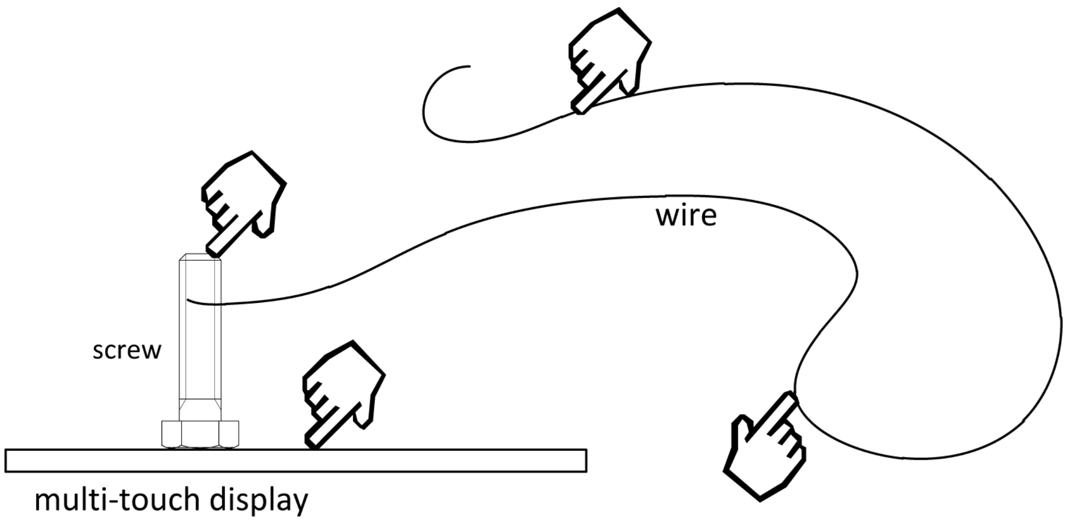

Figure 5.2: A touch is detected at every position the hand touches the surface, screw or wire. The wire and screw are conductive and forward the touch, extending the reach of the finger.

not matter. Even if the wire is two meter long and a user touches the far end of the wire, a touch is detected at the position of the screw. The wire transports the change to the screw, which passes it to the display, as depicted in 5.2. This scenario can be advanced with a series of conductive materials. The conductive materials can be embedded into other non conductive materials such as plastic. The embedding in other non conductive materials does not change the capacitive change when touching the conductive parts of the construction.

Combining conductive elements increases the range extension of the sensors.

The advantage is the removal of direct touch interaction through the user. Instead of touching the display itself a user touches an object that passes the touch downwards to the display. A new concept of user interaction on multiple touch screen is given through the use of physical objects. These objects are called widgets. Widgets facilitate new possibilities in the interaction space. Touchscreens try to limit user actions by providing only a few visible elements on the screen or ignoring touches within specific areas. With the help of widgets, these constraints are directly integrated into the physical appearance. Users are forced to perform specific actions through physical affordances (Norman [2002]) or some actions are blocked through physical

A user touches a widget instead of the screen to interact with the system.

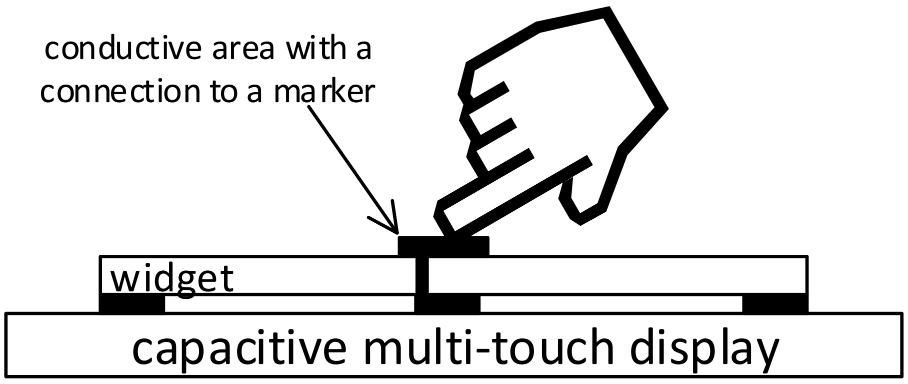
conductive area with a connection to a marker
widget
capacitive multi-touch display

Figure 5.3: A user can only touch one specific area of the touchscreen through the widget. The remaining screen is protected by the widget's non-conductive body, limiting the actions a user can perform.

constraints. Figure 5.3 depicts an example where a user can only touch one specific area of a touchscreen through a widget.

Affordances guide the user to specific actions.

Physical affordances utilize the shape of an object to indicate functions or interactions. If an object with a round handle on top is provided, it affords an interaction with the handle. A rotary handle affords rotating it and a button affords a press. Using affordances a user is guided through the interaction with the system.

Physical constraints limit the number of possible actions.

Physical constraints limit the number of interactions. In figure 5.3 the user can only touch one area of the widget and that area is mapped to a dedicated area on the touchscreen. The rest of the screen underneath the widget is blocked by the widget's non-conductive body.

Widgets provide haptic feedback.

Furthermore, widgets provide haptic feedback. A physical rotary knob provides the chance to operate it without watching. Through the haptic feedback the user knows in which state the control is, and turning the knob provides the haptic feedback that it is actually rotated. A virtual screen object provides visual feedback only. Without looking at the screen a user does now know if his action has altered the state of the control.

Our design space of a widget is divided into the distinction between one area and multiple area controls and explicit and implicit touch. After providing two examples of widget interactions on touchscreens we introduce the concept of explicit and implicit touches.

5.2 Scenarios for Widget Interactions

Every multi-touch system using widgets has to deal with two major usability challenges. On the one hand there is the limited screen estate, that is even further occluded by the widgets. The second challenge is the pairing of physical objects to virtual counterparts. We introduce a solution approach for each of that challenges for capacitive multi-touch screens.

5.2.1 Screen Occlusion

Transparency of widgets reduces screen occlusion.

One of the main challenges when working on a multi-touch screen is the limited screen estate. We only have a limited amount of available pixels to display information. This problem is boosted if we need additional space to display the screen representations of the widgets and their menus. The first step to reduce the screen estate occupied by graphical representations is the transparency of the widgets. This enables the display of the widgets representation directly underneath the widget instead of occluding additional space around it.

Screen representations of widgets are displayed underneath the widget.

In SLAP by Weiss et al. [2009] the menus are always visible around a paired widget. This reduces the visible screen space used to display information or system based objects. The widgets screen representation should always be visible to give a feedback if the widget is detected. Furthermore, the space under a widget, even if this is transparent, can hardly be used for anything else. It will always be covered by the widget reducing the visibility of the screen space beneath it. Based on feedback and visibility agree, that the space beneath a widget is reserved for screen representation. That leaves the display of the menus of a widget as a starting point to reduce the screen occlusion. To illustrate the amount of screen estate that can be covered by menus, we designed a mock-up of a possible multi-touch table display, shown in figure 5.4. This example clarifies the need to reduce the screen occlusion resulting from the widgets.

Screen occlusion can be reduced by adapting the menus of a widget.

On the one hand, we want to reduce the screen space occupied by the menus. On the other hand we need the menus to operate the different controls of a widget, since the functionality can change based on the object the widget is paired with. This changing functionality can be represented by dynamic relabeling, as introduced by Weiss et al. [2009].

On screen menus are mandatory.

Dynamic relabeling describes the possibility to display different menus and even widget representations based on the functions the widget can offer. For example if we have a knob widget we can pair it to a video object, than the rotation of the knob could change the volume of the system. The menu around the widget should display a graphical hint that the volume is being controlled and at which level it currently is. This could be done by a static rendering of the menu for exactly that widget. But what happens if we pair this knob with a picture? We would still have the fixed rendered screen menu that displays a volume control. It would make no sense to change the volume of a picture. This is where the dynamic relabeling comes into play. Instead of displaying the volume control change the menu on the fly, showing a menu that is suitable to the paired object. In our example this could be a control to change the brightness of the paired picture object.

Dynamic relabeling adapts the labeling to changing functionality.

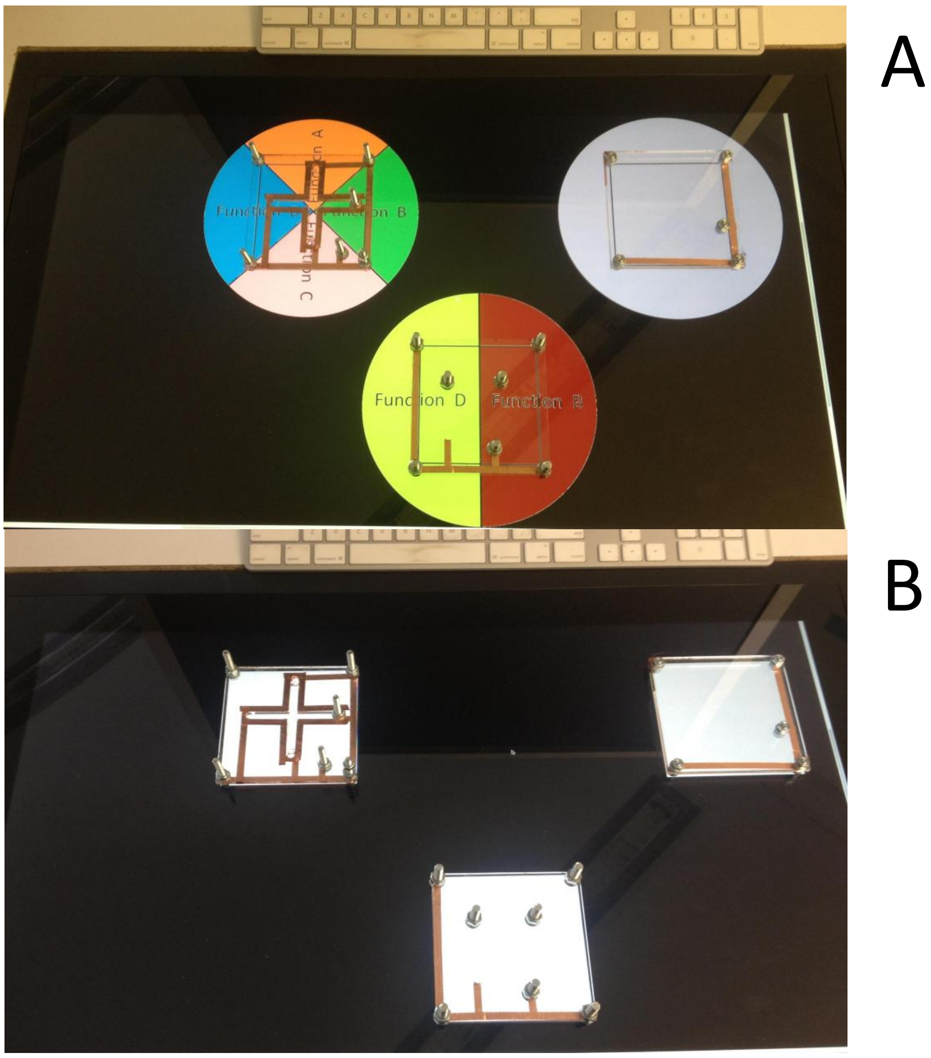

Figure 5.4: Figure A shows the surface of a table with widgets and their active menus around them. Depending on how they are placed, these three widgets already occlude a lot of screen estate. Figure B shows the same table with the same widgets but without the menus to illustrate the difference in occluded screen space.

Three State Model for Widgets

We introduce a solution by presenting a three state model for widgets. The three state model differentiates between the three possible states of a detected widget. Figure 5.5 provides an overview of the three states and the transitions between them and figure 5.6 visualizes the different states.

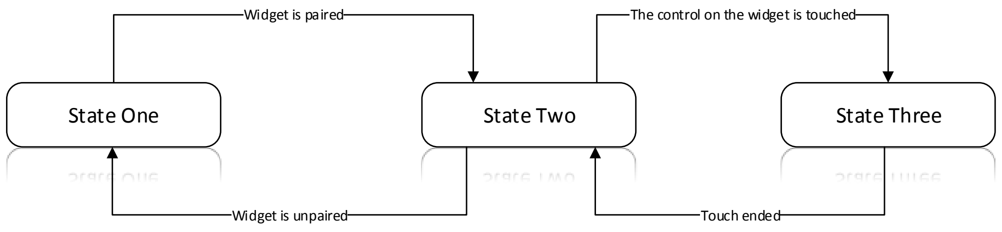

Figure 5.5: This diagram depicts how the states can interchange and which action is required to achieve the change.

In state one a widget is not paired and has no visible menu.

State One is defined as the state in which a widget is after initially placing it on the multi-touch display. In this state the widget is not paired with any object. The widget only requires its screen representation directly underneath the widget. Presenting the menu would be redundant because there is no information that could be displayed in it due to the fact that it is not paired with an object. Therefore the screen occlusion is limited to the space occupied by the real-world representation on the table itself and the virtual representation underneath it.

In state two the widget is paired and the menu is partly hidden providing a visual clue.

State Two is reached after pairing the widget with an on-screen object. The pairing gives the widget itself a meaning. In contrast to earlier states the widget now has an object on which it can perform actions, based on the type of the object. How the pairing can be done and how it is visualized is shown in section 5.2.2—"Pairing". To give feedback on the state of the widget, additionally to its screen representation and to show the possible functions the menu will be activated. Since noone is interacting with the widget anymore the menu can be hidden. It is not hidden completely, so that there still is a visible clue that something is present in addition to the widget object to avoid surprises when reaching state three. Therefore a small border part of the menu is still visible and the rest is hidden "under" the widget. If a user touches the control state three is reached.

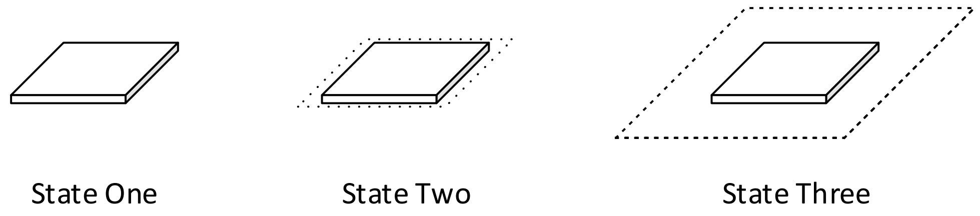

Figure 5.6: The three states of our three state model. State one is the unpaired widget, not showing any menu. State two is a paired widget with a hint that there is more under the widget, depicted by a small visible area of the widgets menu. State three is the quasimode state in which the menu becomes visible. The quasimode is triggered by a user touching the controls of the widget or other predefined areas.

State Three is the final state of our three state model. It reflects the state in which a widget is paired and a user is touching the control, indicating an interaction. This touching of the control has been defined by Raskin [2000] as quasimode. The quasimode is a mode in which the user constantly has to perform a physical action to maintain the active state. In this case the action is touching the control, entering a quasimode in which the full menu is be shown. The quasimode in this case is used to indicate the system that a user is most likely going to interact with the widget. Therefore it is mandatory that the user gets sufficient feedback about his actions, reflected by the widget's menu. The third state is only valid as long as the user touches the control of the widget. If the touch is revoked the user no longer needs the visual feedback of the widget, since he finished his actions. Therefore ending the touch results in the transition to state two again.

In state three the menu of the paired widget is displayed.

This concludes our three state model. We have shown that the screen occlusion can be reduced by using the three state model.

5.2.2 Pairing

Physical objects can be paired with virtual ones, establishing a link between the two.

The interaction of physical widgets with virtual objects affords a link between these two objects. Without a link it can not be differentiated which object is currently being manipulated by a widget. To establish the link a pairing action is needed. Weiss et al. [2009] present a pairing gesture where the the border of a widget and an object are simultaneously tapped. The border has a width of about two centimeters, making it easy to tap it. This pairs and unpairs the two objects. Unpaired objects and widgets are tagged with a colored border. This border is the pairing area for the widget. It implies a loss of screen space for the space of the colored area. Two paired objects are linked with a colored line to indicate the connection.

The colored border of the widget is used to pair the widget with a virtual object. In optical tracking the border adds a significant area to the widget which can directly be touched. Touching the widget itself has no effect using visual tracking technologies. A dedicated button that can lift a marker from the surface is necessary. Otherwise a touch can not be communicated to the surface without technical controls. Therefore it is necessary to enlarge the area of the widget with a part that is touch sensitive. This limitation is omitted with capacitive touch. There it is possible to have a marker that is constantly connected to the surface and can switch states. The marker is not detected if it is not touched. Therefore it is possible to dedicate a small area of the widget that needs to be touched to initialize the pairing gesture.

The widget marker in the top right corner is the dedicated pairing button.

To avoid additional markers and occupying additional space on the widget the marker in the top right corner is chosen as pairing button. The advantage is that the marker is already there to stabilize the widget and that it is placed in a corner. The controls of a widget are placed in the center region of the widget. This minimizes the amount of accidental touches of the pairing button. However, depending on the widget design an additional marker is possible. It can be placed anywhere on the widget. Even the border of the widget is a possible position for a touch area.

A meaningful pairing gesture is simple in terms of execution and memorizing. Additionally it needs to be designed in a way that makes it difficult to accidentally pair objects. The pairing gesture of SLAP (Weiss et al. [2009]) is a time based operation. The user touches the colored border of a widget and the virtual object itself at the same time. The touch has to be maintained for a predefined time interval. A successful pairing is indicated by a color change of the border and an additional line that connects the two paired objects.

A pairing gesture is easy to remember but protects from accidental use.

We developed a pairing gesture that is based on the idea of SLAP (Weiss et al. [2009]) – a synchronous bimanual gesture that interacts with both participants. The widget is touched at the dedicated touch area. This indicates the wish to pair this widget with another object. To avoid unintended pairings a time based component is added. The user has to move his finger over the object for a small amount of time to confirm the pairing. The movement is added to remove false pairing. If a widget is close to a virtual object and the user is holding the widget at the border, it may occur that one finger is touching the surface. This accidental touch could be on a virtual object. Triggering the pairing button on the widget would then pair the widget with the object close to it because it is constantly touched. Therefore an intentional interaction component was added to the pairing gesture. Figure 5.8 illustrates the sliding gesture, whereas figure 5.7 shows the initial setup.

Two objects are paired by touching the pairing marker and sliding over the virtual object.

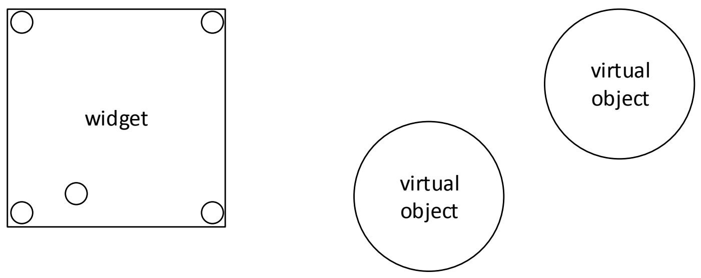

Figure 5.7: A widget with two virtual objects, all are unpaired and placed on a multi-touch display.

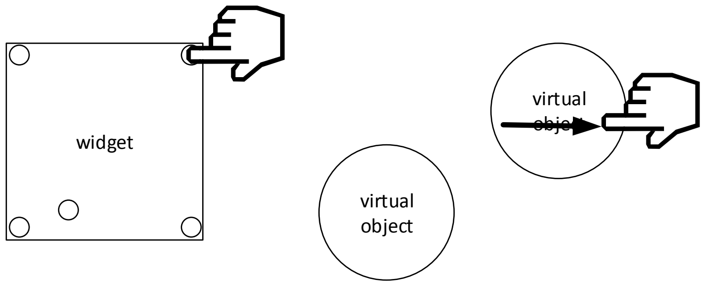

Figure 5.8: The pairing gesture performed on the widget and the object on the right side. The gesture includes touching the dedicated area on the widget and sliding over the object. If both is done simultaneously the objects will be paired.

Colored borders indicate affiliations.

It is important to provide feedback after establishing a connection between two objects. If paired, both objects receive a colored and very small border in the same color. Each pair has a unique color. This indicates the link between exactly these two objects. If more than one object is paired with a widget, all paired objects receive the same color scheme. A successful pairing is depicted in figure 5.9.

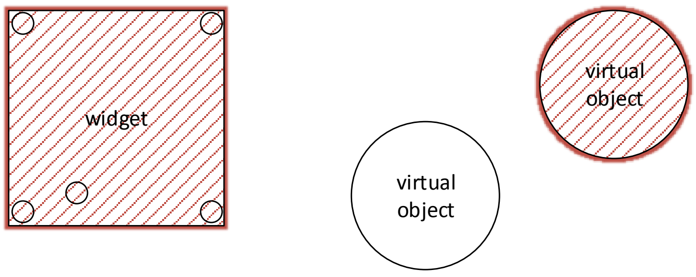

Figure 5.9: A successful pairing creates a small colored border around the widget and the object. This indicates a pairing and the correlation.

Touching the pairing area highlights all connected virtual objects.

On a table with a few widgets and 20 or more virtual objects it is natural to lose track of the paired combinations. To get a highlight of the objects that are paired to a widget, a function was added. When the pairing area of the widget is touched the connections are visualized by a glowing line in the current color scheme. The line connects all virtual objects with the widget, as depicted in figure 5.10.

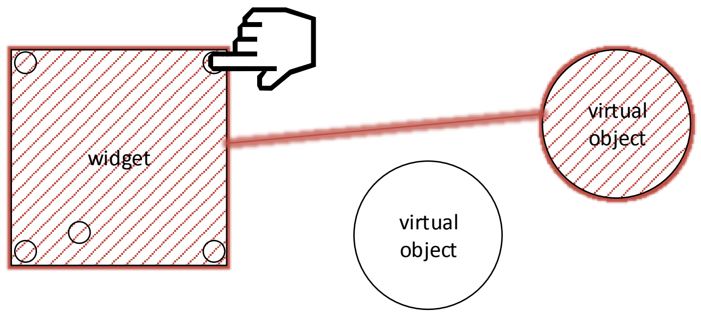

Figure 5.10: Touching only the pairing area on the widget highlights every connected object and creates a glowing connection line between them.

5.3 Explicit and Implicit Touches

Explicit touches are deliberate interactions.

Touching a touch sensitive area on a widget is categorized into explicit and implicit touches. Explicit touches are selective interactions with a control or area. The intention is to touch the area to trigger a function of the control or widget. Explicit touches can be enforced by physical affordances like shapes or highlighted areas. If a control is recognizable as such, a user can deliberately interact with it.

Implicit touches are unconscious user interactions with the system.

Implicit touches on the other hand are touches that happen unknowingly. A widget might have an area that is not recognizable as a touch sensitive area. A user interacts with the widget and touches the area unconsciously. That kind of touch is an implicit touch. However, accidental touches and implicit touches are not the same. Accidental touches are unwanted touches with no deeper meaning. Implicit touches are obscured user interactions with the system.

5.3.1 Explicit Touch Example - Pressing a Button

Pressing a button is an explicit touch.

An explicit touch example is almost every interaction with a control on a widget. The user decides to touch a control on the widget, resulting in an explicit touch. For example a one button widget is paired with a virtual object and the user wants to call a function for that virtual object. He deliberately touches the button on the widget. Accidental touches of the button are not considered as an explicit touch. An explicit touch requires an intention of the user.

5.3.2 Implicit Touch Example - Detect Removal of a Widget

Removing a widget can cause problems on the software side.

The interaction with large scale multi-touch tabletops requires different operations. Actions like removing a widget from the table have a specific action sequence. The user grabs the object and lifts it from the table. Depending on the object the widget is paired with, a removal can have an immense impact on the system. To evade unwanted behavior that can result from removing a widget or performing other operations, touch areas on the widget can be used to anticipate the action. For example, if a user is manipulating an object, the widget detection tracks the state of the widget. Grabbing the object results in additional touches on the screen. Lifting the object removes the touch-points from the surface. The touch-points created from the hand grabbing the object and the missing points from the widget could cause unwanted errors. These errors are avoided if the system is informed about the removal of the widget before it is executed.

Anticipating the removal avoids errors.

An additional touch area can predict the removal.

Detecting the incoming action can be done by applying additional touch areas with markers to the widget. Considering the example of removing a widget from the display a user grabs the object before he can lift it. The grabbing can be detected if the border of the widget can sense the touch. One additional layer of a conductive material around the border can detect this action.
Figure 5.11 depicts how such a widget might look like. The black bar is the additional conductive layer that is con-

nected to a dedicated marker on the widget. If this marker becomes active, the system can anticipate that the widget is most likely being removed in the soon. Therefore it can stop tracking the object or calling the dedicated functions as long as the button is active.

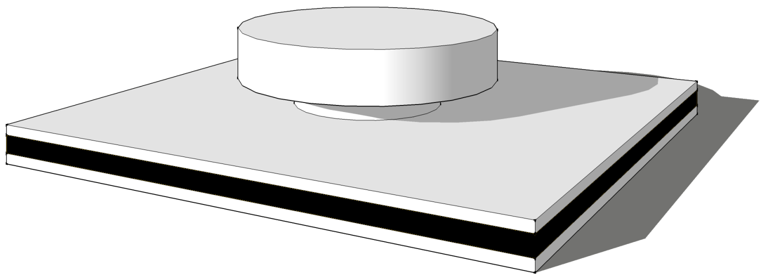

Figure 5.11: A knob widget with an additional layer of conductive material around the border, depicted by the black bar. This bar is connected to a dedicated marker. If this marker is active the system detects that the border is touched.

5.4 Controls with Multiple Touch Areas

All of the previous widget examples provide controls with exactly one touch area per control. Capacitive touch technology facilitates the use of multiple touch ares for one control. The sensing areas have to be separated through a thin non-conductive layer. Each area is connected to its own marker at the bottom side of the widget. Multiple areas enhance the functionality of a control element. Furthermore, multiple control areas can be used to replace mechanical control elements. A rotary knob can be replaced by a circle of independent touch areas.

Multiple touch-sensitive areas can be embedded into a control handle.

The following two examples illustrate the use of multiple touch areas on a control and as a replacement for controls.

5.4.1 Rotary Knob with Two Granularities

Traditional rotary knobs provide only one granularity.

A traditional rotary knob provides a linear control element. By rotating the knob values can be changed. However, a knob has one important drawback. The granularity is fixed to one value. To change the granularity additional buttons can be added, or a second knob can be stacked on top of the first one as presented in *CapStones and ZebraWidgets* (Chan et al. [2012]). We present a knob that offers two granularity modes without additional knobs or buttons.

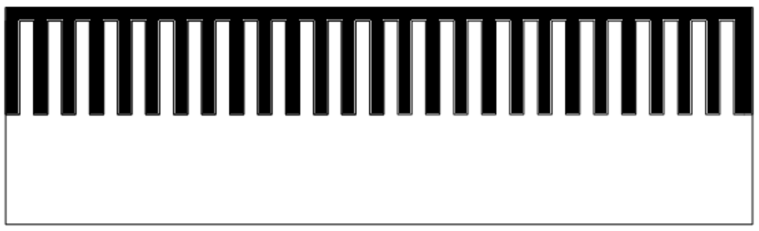

Figure 5.12: The black area is a conductive area separated from the white conductive area. This pattern enables a two mode interaction on a control.

Two toothed touch areas provide two granularities.

The two mode rotary knob has two touch sensitive areas embedded in the handle, as depicted in figure 5.12. Both areas are separated through a thin non-conductive area. This avoids activating both areas when one is touched. Each area is connected to a marker, the white area to the base marker, indicating the center of the knob. The black area is connected to an additional marker close to the center marker. If a user operates the knob by holding it at the lower end of the handle only the white touch area is activated. The normal granularity is applied to the rotary operations of the knob. The granularity can be changed by holding the upper area of the handle. Touching the upper end activates the black and the white touch area. Both markers are activated and the rotary operations are performed with a different granularity than before.

The additional touch area mimics a mode button.

The additional touch area works analogous to a mode button that can be pressed during a rotation. The advantage is that the user is not aware of pressing additional buttons and does not need to use more fingers to perform the operation. The widget can be extended to more than two modes by adding more toothed areas. They can also be placed in different alignments to activate different mode switches depending on the touched region.

5.4.2 Four State Slider

The second application of multiple touch sensitive areas is the replacement of mechanical controls. This example explains how a traditional slider can be replaced by an arrangement of different touch sensitive areas.

A combination of different touch areas can replace hardware controls.

A four state slider requires four markers. Each state is encoded by a combination of active markers. State one uses the first marker, state two the first two markers, state three the first three and state four uses all markers. In dependency to the active markers the system detects the state of the slider. The hardware arrangement is depicted in 5.13. The lanes are shifted, adding a new lane in every area for a state.

A four state slider requires four markers.

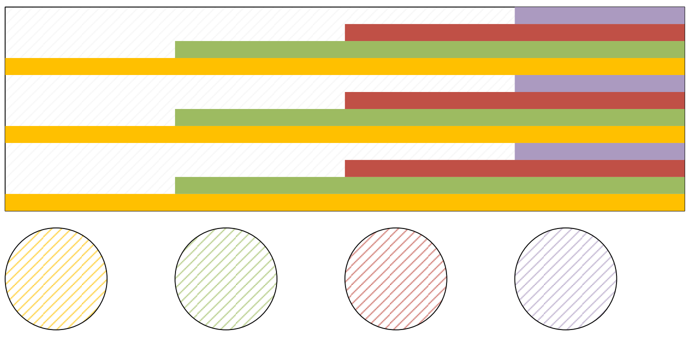

Figure 5.13: A slider design without any mechanics. The sliding works through different and independent touch areas. It has four states combined through the four touch areas. In this example all four marker below the slider are inactive.

When a user touches the first part of the slider, the first marker is activated as depicted in 5.14 (A). It is important that the lanes are small enough, so that all lanes of the area are touched, regardless of the touching position. If the lanes are too big, a user might touch area three but only hitting

The size of the lanes is important.

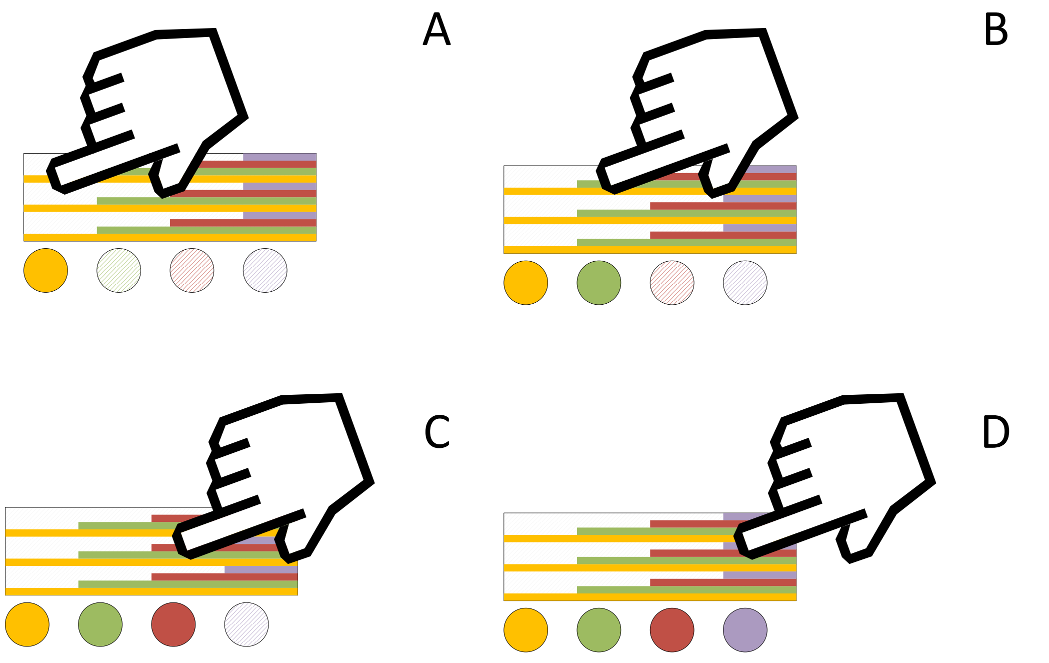

Figure 5.14: (A) A user is touching the first part of the slider. Only one touch area is present in this area. Therefore only one marker is active. (B) The touch moved to part two of the slider with two different touch areas. The touch activates two markers. (C) Three different areas are touched, activating three markers. (D) All four markers are touched at the same time. Therefore all four markers are active.

two of the three lanes. As the finger slides from area one to two the second lane is touched, activating marker two (B). When the user reaches area four all four lanes and markers are active.

The lane height limits the number of states.

The four state slider can be extended to as much states as technical realizable. A size of half a millimeter for a lane extends a slider with one centimeter height to around 16-18 states, depending on the isolation size between the lanes.

5.5 Limitations

Just like any technology capacitive multi-touch sensing has its limitations. One basic limitation is the state model of capacitive widgets. A marker has only two states. Either it is detected or it is not touched and undetected by the system. There is no state between these two. A marker can not be half activated or deactivated. Touching a marker activates it completely. Furthermore, if two markers are connected both are activated if one is touched. Without mechanical components it is not possible to activate only one of the two, or in alternating order. Connections are static and continuously alter the behavior of the connected elements. Passive markers can not change their type to active without remodeling the widget or implementing a mechanical switch that interrupts the connection. However, linking the switch to a button press or other controls may offer new challenges.

A marker has two fixed states: on or off.

Connections are static.

The second limitation arises from the connection problem of two markers. If two markers are placed close to each other and a finger is slid from one marker to the other, both markers are activated due to the connection through the finger. When designing the four state slider marker we thought about different approaches. The first design included a bit wise encoding of the state with three markers for three bit encoding. The seven different states are illustrated through table 5.1.

A finger acts as a connection between markers.

State	Marker A	Marker B	Marker C
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 5.1: 3-Bit encoding of different states

The technical realization is directly derived from table 5.1. Seven areas are required to encode the seven states. Area one implements marker C, area two only marker B and area three both markers and so forth. Examining the transition of state one to state two a finger connects marker B and C. Therefore, the transition is detected as state three of the slider. The same error occurs in the transition of state six and seven.

The transition is falsely detected as another state.

Additional markers are required to mark the transitions between the states. We identified two approaches. The first approach is to use one marker for the transitions. Independent of the selected state, if the transition marker is detected the state does not change. It works well if the last state is saved and used during the active phase of the transition marker. The only error that may occur is if a user initially touches the slider on a transition. Which state is being touched? If the sequence 0 1 1 and active transition marker is received two states are possible. The first state is the transition from one to two where both areas in addition to the transition are touched, so either state one or two is active. The second state is the transition before or after state three when only the third state in addition to the transition marker is touched. This happens shortly after the transition to three or shortly before the transition to state four. This problem is handled by the second approach.

The current state is ambiguous.

The second approach uses a separate marker for every at risk transition. The three bit encoded slider has two at risk states, between states one and two as well as between six and seven. The extra markers identify the current transition and a current state is estimated from the transition marker. A new problem arises from the extra markers. Each additional bit doubles the number of transition markers. However, this method requires less markers than the four state slider.

Every additional bit doubles the number of transition markers.

Wires were introduced as material for connections between markers. One important aspect when using wires or other long materials is that at some point the material starts to generate a touch, even if it is not grounded or physically touched. We formed the hypothesis that a long wire starts to function as an antenna. However, long connections are not suited for active markers.

The last limitation is the limited physical space. A common widget has the size of eight to ten centimeters in square. Each marker has the size of one centimeter and has a distance between markers of one centimeter. If a widget is filled with markers an eight centimeter widget can hold up to 16 markers, four rows and four columns. The number is sufficient for simple constructions, but the position and identification markers require at least four of the 16 available markers.

The widget size limits the amount of usable markers.

Chapter 6

Summary and future work

"There are truths which are not for all men, nor for all times."

— Voltaire

This work contributes to the field by introducing design constraints and limitations for capacitive touchscreen widgets during the construction phase. Furthermore, algorithms for the footprint generation and widget detection have been presented. None of the currently existing works covered the aspects of design constraints and algorithms for the detection. At last we described and explained a design space for capacitive touchscreen widgets.

In this chapter we summarize the thesis and point out areas for future work.

6.1 Summary

The marker size and distance are design constraints.

We have shown which design constraints must be fulfilled when designing physical objects for capacitive touchscreens. The marker size and distance of one centimeter were gathered by conducting experiments on an iPad version one. Smaller sizes reduce the accuracy of touch detection. Larger markers reduce the usable space on a widget.

Detection, position and identification is realized with three marker categories.

Three types of markers were introduced: Position markers communicate the position and size of a widget to the touchscreen. They are located on three corners of a widget. Identification markers add a unique ID to each widget. The unique ID is constructed by the unique distances between the identification markers and the position markers. The last type describes the state markers which are responsible for the tracking of control elements placed on a widget. State markers are divided into different subcategories, depending on the used controls. Furthermore, a distinction between active and passive markers has been made. Active markers are touch activated and passive markers are grounded, to be detected at all times.

The three phase model divides the construction process into three phases.

The next section presented an overview of the physical construction of a widget and a set of suitable materials, used for different stages of a widget. The three phase model divided the construction cycle into the design, material picking and engineering phase. Two examples, a button and a slider have been presented, illustrating the three phase model.

Unique attributes are used to differentiate between the markers.

A highly adaptable algorithm for the footprint generation creates a storable file for every widget. The unique and semi-unique attributes have been introduced in this context. The markers are processed and divided into the appropriate categories and saved in the XML file format. The widget detection loads these files and tracks widgets on a capacitive touchscreen.

The overview of the design space indicates the benefits of widgets in contrast to normal touch input. We have shown how widgets facilitate physical constraints by limiting the touchable area that is reachable for a user. The widget blocks the occupied touch sensitive area, and only selected areas are connected to the surface. Single and multiple touch areas have been introduced. They allow more complex control elements and the division of a widget in different areas. A control element can utilize different touch areas, implementing various modes. Furthermore, we presented the difference between explicit and implicit touch. Explicit touch is an unconscious user interaction with the system, whereas implicit touch is goal-oriented. Explicit and implicit touches are enforced through physical affordances and constraints in the widget's design. Finally we divided the markers of a widget into two categories: active and passive markers. Active markers are activated when a user touches them – passives marker are permanently detected. Table 6.1 summarizes the different design space aspects.

Widgets limit the reachable area on the surface.

Markers are divided into active and passive markers.

Category	Subcategory
Touch Area	Single Area
	Multiple Area
Touch Modes	Explicit
	Implicit
Marker Types	Active
	Passive

Table 6.1: Overview of the design space for physical widgets

6.2 Future Work

During the work on this thesis we encountered different aspects that are out of scope. This section offers an overview of the future work that can be done in the field of capacitive touchscreen widgets.

6.2.1 Data Extraction from the Hardware

The touchscreen is a blackbox.

The touch input data we used to track and detect our widgets was provided from the iPad's operating system. During the single steps of the implementation we noticed that the iPad is a black box to us. We place a widget on top and receive some touch locations. How the positions are computed, or how disturbances are filtered out is completely unknown to us. The controller might provide different touch information than the operating system. The stages in which larger touch spots are merged to smaller ones, at which thresholds touches are ignored or even the fact that sometimes touches disappeared after a time are non-transparent.

Extracting the touches directly from the controller can yield more precise results.

Therefore, a possible work for future systems implementing capacitive touchscreen widgets is the direct touch point extraction from the controller. The controller should yield high resolution point clouds of the touches instead of merging them to one touch at the calculated center. Sensors are placed tight enough, allowing the computation of marker shapes if all sensor results are present. The shape can be used to identify widgets or to calculate the orientation, saving widget space by using less markers. Additionally much smaller markers could be detected and the distance between to markers can be decreased with more precise raw data.

Experiments have to be conducted to validate that the controller yields more precise data, or if the controller is the one applying the restrictions for the widget design.

6.2.2 Further Application Areas

Up till now we have only considered the technical realization and design of a widget system. Operating a widget on a multi-touch table affords a meaningful screen representation. An intuitive mapping between the physical object and the virtual screen representation is essential. The content under the widget is not limited to traditional images or graphics. Everything can be visualized on the virtual space. Therefore a set of appropriate screen representations can be invented and tested with the help of a user study. Typically the representation depends on the application using the widget system. Nevertheless a guideline for the design is essential for a successful user interaction.

Design guidelines for screen representations are essential.

Bibliography

Gary Barrett and Ryomei Omote. Projected-capacitive touch technology. *Information Display*, 26, No. 3:16–21, 2010.

Rachel Blagojevic, Xiliang Chen, Ryan Tan, Robert Sheehan, and Beryl Plimmer. Using tangible drawing tools on a capacitive multi-touch display. In *Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers*, BCS-HCI '12, pages 315–320, Swinton, UK, UK, 2012. British Computer Society. URL `http://dl.acm.org/citation.cfm?id=2377916.2377959`.

Liwei Chan, Stefanie Müller, Anne Roudaut, and Patrick Baudisch. Capstones and zebrawidgets: sensing stacks of building blocks, dials and sliders on capacitive touch screens. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 2189–2192, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1015-4. doi: 10.1145/2208276.2208371. URL `http://doi.acm.org/10.1145/2208276.2208371`.

Tzuwen Chang, Neng-Hao Yu, Sung-Sheng Tsai, Mike Y. Chen, and Yi-Ping Hung. Clip-on gadgets: expandable tactile controls for multi-touch devices. In *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services companion*, MobileHCI '12, pages 163–166, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1443-5. doi: 10.1145/2371664.2371699. URL `http://doi.acm.org/10.1145/2371664.2371699`.

George W. Fitzmaurice, Hiroshi Ishii, and William A. S. Buxton. Bricks: laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 442–449, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1. doi: 10.1145/223904.223964. URL `http://dx.doi.org/10.1145/223904.223964`.

Jefferson Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 115–118, New York, NY, USA, 2005. ACM. ISBN 1-59593-271-2. doi: 10.1145/1095034.1095054. URL `http://doi.acm.org/10.1145/1095034.1095054`.

Juan David Hincapie-Ramos, Morten Esbensen, and Magdalena Kogutowska. Rapid prototyping of tangibles with a capacitive mouse. In *Proceedings of the 11th Danish HCI Research Symposium*, 2011. URL `http://jhincapier.files.wordpress.com/2010/02/toki-diy-toolkit.pdf`.

Sergi Jordà. The reactable: tangible and tabletop music performance. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, pages 2989–2994, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-930-5. doi: 10.1145/1753846.1753903. URL `http://doi.acm.org/10.1145/1753846.1753903`.

Sven Kratz, Tilo Westermann, Michael Rohs, and Georg Essl. Capwidgets: tangile widgets versus multi-touch controls on mobile devices. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 1351–1356, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. doi: 10.1145/1979742.1979773. URL `http://doi.acm.org/10.1145/1979742.1979773`.

Nobuyuki Matsushita and Jun Rekimoto. Holowall: designing a finger, hand, body, and object sensitive wall. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, UIST '97, pages 209–210, New York, NY, USA, 1997. ACM. ISBN 0-89791-881-9. doi: 10.1145/263407.263549. URL `http://doi.acm.org/10.1145/263407.263549`.

Donald A. Norman. *The design of everyday things*. Basic Books, [New York], 1. basic paperback ed., [nachdr.] edition, 2002. ISBN 0-465-06710-7.

Jef Raskin. *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-37937-6.

Thomas H Speeter. A tactile sensing system for robotic manipulation. *The International Journal of Robotics Research*, 9 (6):25–36, 1990.

Malte Weiss. *Bringing Haptic General-Purpose Controls to Interactive Tabletops*. PhD thesis, RWTH Aachen University, 2012.

Malte Weiss, Julie Wagner, Yvonne Jansen, Roger Jennings, Ramsin Khoshabeh, James D. Hollan, and Jan Borchers. Slap widgets: bridging the gap between virtual and physical controls on tabletops. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 481–490, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518779. URL `http://doi.acm.org/10.1145/1518701.1518779`.

Alexander Wiethoff, Hanna Schneider, Michael Rohs, Andreas Butz, and Saul Greenberg. Sketch-a-tui: low cost prototyping of tangible interactions using cardboard and conductive ink. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction*, TEI '12, pages 309–312, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1174-8. doi: 10.1145/2148131.2148196. URL `http://doi.acm.org/10.1145/2148131.2148196`.

Neng-Hao Yu, Li-Wei Chan, Lung-Pan Cheng, Mike Y. Chen, and Yi-Ping Hung. Enabling tangible interaction on capacitive touch panels. In *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 457–458, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0462-7. doi: 10.1145/1866218.1866269. URL `http://doi.acm.org/10.1145/1866218.1866269`.

Neng-Hao Yu, Li-Wei Chan, Seng Yong Lau, Sung-Sheng Tsai, I-Chun Hsiao, Dian-Je Tsai, Fang-I Hsiao, Lung-Pan Cheng, Mike Chen, Polly Huang, and Yi-Ping Hung. Tuic: enabling tangible interaction on capacitive multi-touch displays. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 2995–3004, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979386. URL `http://doi.acm.org/10.1145/1978942.1979386`.

Index

Typeset April 12, 2013