LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department "Institut für Informatik"
Chair of Human-Machine-Interaction
Prof. Dr. Andreas Butz

Diploma Thesis

The Fiduciary Glove Toolkit

Johannes L. Kiemer
jkiemer@kiemer-online.de

Time needed for completion:	01. 08. 2010 to 01. 03. 2011
Supervisors:	Nicolai Marquardt M.Sc. and Prof. Dr. Saul Greenberg
Professor in charge:	Prof. Dr. Andreas Butz

Abstract

This thesis introduces the Fiduciary Glove Toolkit. It is a rapid prototyping toolkit that enables developers to differentiate between hand parts, hands and users and recognize hand postures and gestures in their applications. The toolkit is named after the Fiduciary Glove, which is its foundation. The Fiduciary Glove is an ordinary glove with fiduciary markers stuck onto 15 key hand parts. The tags are recognizable by the Microsoft Surface, each has a unique identifier. This way, the hand part and hand itself can be reliably identified when a tag touches the surface. Through assigning a glove to a particular person, user identification is possible and just as reliable. The API of the Fiduciary Glove Toolkit allows developers to easily integrate these capabilities in their applicationtogether with gestures and postures. Three utilities for the creation, adjustment and testing of glove configurations, gestures and postures complement the toolkit.

Zusammenfassung

Diese Arbeit stellt das Fiduciary Glove Toolkit vor. Das Toolkit ermöglicht eine schnelle Entwicklung von Anwendungen, die zwischen verschiedenen Teilen der Hand, Händen und Nutzern unterscheiden und statische Handstellungen sowie Gesten erkennen können. Es ist nach dem Fiduciary Glove benannt, der die Grundlage für das Toolkit darstellt. Dabei handelt es sich um einen gewöhnlichen Handschuh an dem Marker auf 15 Schlüsselstellen der Hand angebracht sind. Diese Marker können vom Microsoft Surface erkannt werden; jeder von ihnen besitzt eine eindeutige Kennung. Dadurch können der Teil der Hand und die Hand selbst zuverlässig erkannt werden, wenn ein Marker den Screen des Surface berührt. Indem ein Handschuh einer Person zugewiesen wird, ist zudem eine ebenso zuverlässige Nutzererkennung möglich. Die API des Fiduciary Glove Toolkit erlaubt es Entwicklern diese Fähigkeiten zusammen mit Gesten und Handstellungen in Anwendungen zu integrieren. Drei Utilitys für die Erstellung, Anpassung und Überprüfung von Handschuhkonfigurationen, Gesten und Handstellungen komplettieren das Toolkit.

Task definition

The hand has incredible potential as an expressive input device. But while touch recognition is a common input method for interactive surfaces, most of these technologies do not differentiate particular hand parts that caused the touches. While a few researchers are considering the whole hand as an input device for interactive tables ([35] ,[46]), the methods they use are not particularly robust. The general problem is: can we design a technology that reliably recognizes and differentiates hand parts?

I and my collaborators introduce an inexpensive and easy-to-implement solution to this problem - the Fiduciary Glove [39]. It allows the reliable identification of hand parts that touch the interactive surface.

This thesis will design, implement and evaluate a toolkit for the Fiduciary Glove. The goal of this toolkit is to facilitate rapid prototyping of applications that consider the whole hand and its various hand parts as an input device.

The toolkit will equip programmers with a packaged configuration tool, and the ability to access (a) the position and orientation of individual hand parts, (b) particular hand postures, (c) gestures, and (d) user related information associated with a particular glove. It will deliver this information primarily through an event-driven and object-oriented API. The toolkit will hide the complexity of gathering tag information and analysing relationships and movement for postures and gestures. In particular, the toolkit will incorporate a recognition engine for rudimentary postures and gestures (inspired by related work such as [53]). The toolkit will also allow the assignment of gloves to certain people. Developers will not have to struggle with low-level implementation details and will be able to concentrate on designing the actual envisioned application. If time permits, we will include mechanisms allowing programmers and end-users to define their own postures and gestures.

I will follow toolkit design guidelines - such as those described by Bloch [10] and Henning [24] - when implementing the API. For instance, this includes the concepts of self-explanatory naming or the principle of least astonishment.

A set of sample applications will be developed with the help of the toolkit, which we will use to evaluate the low threshold goal. Additionally, more comprehensive examples will be developed to demonstrate the high ceiling goal, i.e., how to use multiple toolkit features in a single application. These examples will also act as documentation that illustrates particular aspects of the toolkit API.

I will further evaluate the toolkit by two means. First, other members of the laboratory will be asked to provide an expert review of the system and its API. Second, if time permits, selected programmers will be asked to develop their own examples using our system, where they would also be asked to reflect and critique our system.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, 01. März, 2011

. . . . . .

Acknowledgements

First of all, I would like to thank Prof. Dr. Andreas Butz for supporting me with my studies abroad. Also, your lectures are one of the best I've ever attended and helped me a lot throughout this thesis, my studies and projects in general.

This thesis, my journey to Calgary, and to Saarbrücken, would not have been possible without Prof. Dr. Saul Greenberg. I can never thank you enough for this opportunity. Your advice - not only in terms of this thesis - was most valuable to me. You are truly an inspiring person.

Next, I would like to express my immense gratitude to Nicolai Marquardt whose supervision was flawless. You pinpointed the right way when I could not even make out the rough direction. Your idea for the Fiduciary Glove has been the initial spark for this thesis. Thank you for everything.

I extend my great appreciation to all members of the Interactions Lab. I couldn't imagine a better place to do research and have fun at the same time. I'm glad to have you as my friends.

I would also like to thank my parents who supported me throughout my whole life. I wouldn't be where I am now without you. I love you.

Last but not least, I thank you, Julie, for eight amazing years. I love you, too.

Publications

Materials, ideas, and figures from this thesis have appeared previously in the following publications:

- Marquardt, N., Kiemer, J. and Greenberg, S. (2011)
  Expressive Interaction with a Surface through Fiduciary-Tagged Gloves. SurfNet - Technology of the Future - Today!, Vol 2, Issue 1, page one, January/February 2011, Newsletter.

- Marquardt, N., Kiemer, J. and Greenberg, S. (2010)
  What Caused That Touch? Expressive Interaction with a Surface through Fiduciary-Tagged Gloves. In Proceedings of the ACM Conference on Interactive Tabletops and Surfaces - ACM ITS'2010. (Saarbruecken, Germany), ACM Press, pages 139-142, November 7-10.

- Marquardt, N., Kiemer, J. and Greenberg, S. (2010)
  What Caused That Touch? The Video. Research report 2010-965-14, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, June.

Contents

1 Introduction

1.1 Context

The hand is indisputably an extraordinary tool. However, its full potential as an expressive input device hasn't been leveraged by most commercial technologies. Devices with touch input do not discriminate what caused the touch. At best, some assume that people tend to touch and gesture with their hands: single touch by fingertip, two-touch by two fingertips, and multi-touch by multiple fingers and whole-hand postures (e.g., [52]). In contrast, we assert that more interaction power can be expressed if the system knows what hand part is causing the touch.

We are not alone in this assertion: others have developed a variety of algorithms and technologies - some for domains other than surface interaction - to recognize particular fingers, or hand postures, or to distinguish between the hands of one person or between multiple people. Perhaps the most common method to recognize bare hands uses vision. Kruger's VideoPlace [33] tracked silhouettes, and used that to determine hand postures. Newer techniques recognize bare fingers [35], and can even track translation and rotation parameters associated with particular bare hand gestures [46]. To ease the vision recognition task, [51] had a person attach different colored tape to his fingertips in order to help the system identify particular fingers. The Microsoft Surface visually identifies fingertip blobs, where it uses that shape to return the fingertip position and orientation. Dang [15] uses these blobs to distinguish if finger touches are from the same or different hands, while Freeman [19] offers further distinction of hand postures via vision techniques.

Another method is to use specialized hardware to distinguish hand parts. The MERL DiamondTouch surface [17] can identify which touch came from which person. M. Wu and Balakrishnan used the signal shapes returned from the DiamondTouch to recognize fingers and postures [54]. Benko's approach [9] to derive which hand part caused a touch was to attach muscle sensors to the underarm. After recording the signals for touches with certain hand parts. these samples built the basis for the later recognition. Holz and Baudisch [26] presented a solution that is capable of identifying a user via fingerprint recognition.

In other domains, gloves facilitate the tracking of hands and hand parts. Within virtual reality environments, [48] surveys a multitude of ways to track gloves: optical tracking, markers, silhouette analysis, magnetic tracking, optical fibers, while [51] uses colored gloves to discriminate parts of the hand.

We built upon these attempts. In particular, our method knows what hand parts are touching the surface, and which and whose hand the touches where caused by. Importantly, it differs from prior work as we offer a very inexpensive, simple, yet highly accurate method: gloves tagged with fiduciary markers. We show how expressive interactions can be created given the certainty of this knowledge. While this approach requires people to wear gloves, it allows interaction designers to rapidly prototype and explore meaningful interfaces. This can occur well before research solves the problem of accurate barehanded touch identification.

1.2 Fiduciary-Tagged Gloves

A Fiduciary-tagged glove [39] consists of an ordinary glove and a set of fiduciary tags that are stuck onto 15 key hand parts of this glove (cf. Figure 1.1). The Microsoft Surface recognizes the tags out of the box and provides an easy API to access the tag information: its identification number and position and orientation on the surface.

In order to test and demonstrate the capabilities of our glove, we've implemented a finger painting application. We consciously picked this application domain, because it enabled us to depict the concepts we wanted to present very nicely. With the hand part information provided by our glove we could assign separate functions for separate hand parts. Each finger could paint in its own color and thickness. Each knuckle was an eraser with the same thickness as its corresponding finger. A panning function was assigned to the palm tags.

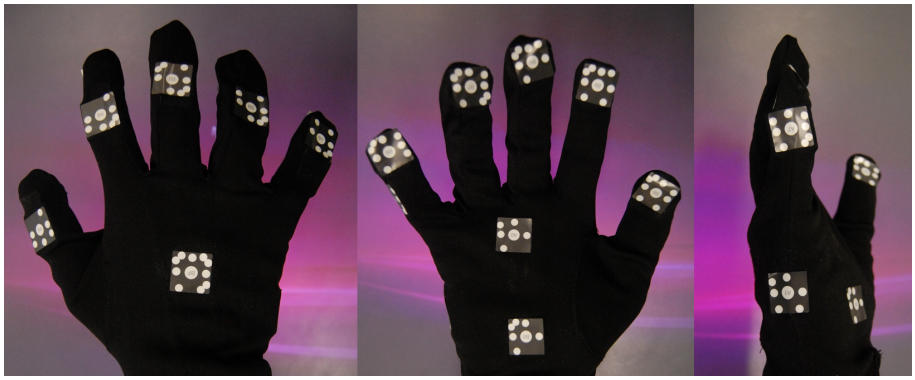

Figure 1.1: The Fiduciary Glove.

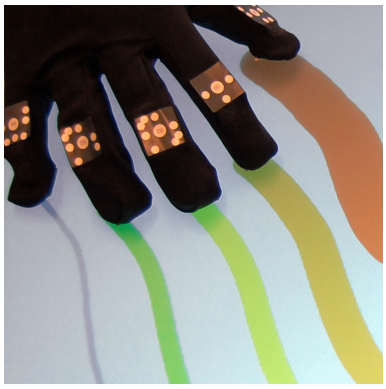

Figure 1.2: Separate functions for separate hand parts.

Postures were recognized by tracking distances and angles between certain tags touching the surface (cf. fist posture in Figure 1.3). Our finger painting application assigned a clearing function to the fist posture. If a user placed down a fist, the canvas was cleared. Because the distances and angles between touching tags were continuously tracked, fluent transitions between different postures could be recognized. Gesture recognition was done by tracking changes in the position of a tag (or possibly a set of tags) over time. With the knowledge of which hand part performed the gesture we could restrict the recognition to only particular fingers or parts of the hand. In our application, a swipe with the back of the hand saved the canvas to an image file.

Because each glove could be assigned to a certain person, true multi-user scenarios were possible. This mechanism was used in the painting application, if two or more people were participating in the painting activity. If someone wanted to clear the canvas, a consensus was needed. Everyone had to place down a fist on the surface to clear the canvas.

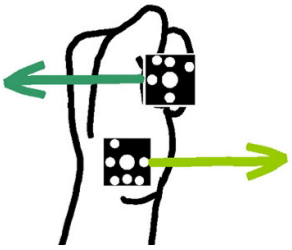

Figure 1.3: Fist posture.

1.3 Background

For the development of the Fiduciary Glove Toolkit, we looked additionally at related work in the areas of toolkits, toolkit design as well as gestures and postures.

1.3.1 Toolkits

Applications for the Fiduciary Glove are naturally touch-based. Toolkits supporting touch-input are various: Several commercial products ship with their own toolkits. The Microsoft Surface [2] comes with the Surface SDK [3], a multi-touch library including marker support. The incorporation of Windows Presentation Foundation (WPF) and XNA facilitate rapid prototyping. Smart Technologies offers the Smart SDK [6] for their products, e.g. the SmartTable [7]. The DiamondSpin SDK [47] was developed at MERL to support the development on the DiamondTouch table [17], especially for around-the-table interaction. In contrast to their commercial pendants, numerous toolkits are rooted in the research and open-source community. Hansen et al. [23] for example gave an overview over PyMT and tell of their experience when using it for HCI research. PyMT [5] is an open-source python library for multi-touch. Diaz-Marino et al. [16] presented a toolkit for the DiamondTouch hardware which simplifies the retrieval of events from this device.

Even though, we could not identify a toolkit that supports the programming for a glove, we could find research in the area of toolkits for tangibles that are tracked on digital surfaces. The reacTIVision [28] is a computer-vision based framework which enables the tracking of objects tagged with fiduciary markers. Papier-Mache uses RFID as well as computer-vision to identify and track objects [31].

1.3.2 Toolkit Design

In 'Framework Design Guidelines' [14], Cwalina and Abrams talk about their experiences during their work on the .Net Framework. It is a rich source of unmistakable guidelines about what to do and what not to do when designing a framework or API. In 'What Makes APIs Hard to Learn? Answers from Developers' [44] Robillard presents the results of a survey answered by 80 developers, 'cutting across different job titles, seniority levels and technology use'. 'What Makes APIs Difficult to Use' by Minhar Fahim Zibran [56] is a good summary of design guidelines from several other papers. By 'Mapping the Space of API Design Decisions' [49], Stylos and Myers put design recommendations from experienced API designers into context. In another paper [40] Myers introduces the concept of low threshold and high ceiling, an important design strategy for APIs. 'Measuring API Usability' [13] contains several API design guidelines suggested by Steven Clarke.

1.3.3 Gestures and Postures

Because related work does not always clearly differentiate between the two terms 'gesture' and 'posture', I present related work in a joint section. Early work has been done 1990 by Henry et al. who presented Artkit [25], a toolkit capable of recognizing and interpreting gestures. Not much later, Landay and Myers explained how to extend an existing user interface toolkit to support gesture recognition [34]. Rubine suggests an example-based definition of gestures. He presents GRANDMA, a toolkit that allows the effortless integration of gestural input [45]. The system recognizes single-stroke gestures after having been trained with samples. In [53] Wobbrock presents an easy recognition algorithm for gestures that does not necessarily require training.

In collaboration with Balakrishnan, M. Wu introduced whole hand gestural interaction techniques for multi-user tabletop displays [54]. The Gesture Toolkit developed by Khandkar and Maurer introduces a gesture recognition engine [30] as well as a definition language for custom gestures [29]. The Gesture and Activity Recognition Toolkit (GART) is a toolkit that enables rapid

development of applications with gesture input [36]. Malik et al. utilize gestural input to interact with a large, wall-mounted surface [38].

1.4 Problem Statement

This work deals with two issues: the lack of methods for reliable hand part, hand and user identification; and the absence of tools that support developers in this domain.

1.4.1 Hand part, Hand and User Identification

Even though approaches for the identification of hand parts, hands and users through various methods exist, they all lack either completeness or reliability. While one approach is able to identify users but not hands and hand parts, others can more or less identify hand parts but not users (cf. 1.1 Context). Especially hand part identification often demands prerequisites, like multiple touch points, or touches that do not move. In order to identify hand parts, hand and user all at the same time, developers have to resort to multiple approaches and cope with unreliable solutions.

1.4.2 Tool Support

If developers decide to implement a custom tabletop application that employs hand parts, hands and users there are no tools that could support them. The lack of libraries and utilities results in a very tedious and time consuming development. The main reason is that you have to handle many low-level details when there are no tools availabe for certain tasks. Hand parts, hands and users have to be recognized manually which often involves several fine-grained steps. In our painting application (cf. Fiduciary-Tagged Gloves), for example, a hand part is identified in the following fashion: Each time a contact event happens you have to check if the contact is a tag, then if it is a byte tag. If so, the identification number of the tag is retrieved and checked against the identifier of the hand part of the gloves. Only then the hand part information can be accessed.

Also, integrating custom postures into the applications is very difficult. For a posture recognition, many low-level details have to be dealt with. The relations between all touching hand parts have to be continuously monitored. If hand parts belong to the same hand or user, the angles and distances between them have to be analyzed. Both values have to be calculated manually. Then, the angles and distances have to be compared to existing postures in order to recognize a posture. If a posture recognition engine and definition language for postures are not implemented, postures have to be hard-coded into the application.

If developers want to use gestures, they need to create an own gesture recognition engine or adjust an existing engine such as the $1 recognizer [53]. In order to design or integrate a gesture recognition engine properly, developers have to learn and understand the low-level mechanisms and where to make adjustments so that the engine fits their needs.

Dealing with low level input systems leads to complex code. It makes it hard to keep an overview over the application. In order to identify a touching hand part more than 20 if cases are needed in our application. A lot of these if-statements reoccur in other events that are raised when a touch changed its position or orientation, or when a touch disappears. Duplicate code leads to higher complexity and makes the navigation through the code wearisome. Another disadvantage of duplicates is that the code is harder to maintain. Changes have to be made at multiple code locations.

1.5 Goals and Methods

In order to address the aforementioned problems, the main goal of this research is to provide very easy access to hand parts and at least easy access to gestures and postures. Additionally, mechanisms for defining new gloves, postures as well as gestures must also be provided. Testing

already defined gloves, postures and gestures must also be included. A secondary goal is that the setup and configuration are easy enough so that they don't pose obstacles to designers when starting with applications for the Fiduciary Glove.

Therefore I will offer a toolkit. An API will serve as basis and allow easy access to hand parts, hands, users gestures and postures. It will hide implementation details to reduce complexity for the developer. Utilities will complement the toolkit and help to define and test gloves, postures and gestures. An extensive documentation, accompanied by code examples, reaching from small code snippets to complete sample applications, will provide an easy start for even average skilled programmers and a gradual learning curve. The toolkit makes also complex scenarios possible and meets the needs of expert programmers. To ensure an easy installation and setup, I will provide an installation wizard that fully integrates the toolkit into the development environment.

For the design of the toolkit, I will follow design guidelines and strategies derived from related work.

1.6 Contributions

The main contributions of this research are:

1. An inexpensive and reliable approach for the identification of hand parts, hands and users on an interactive tabletop - the Fiduciary Glove.

2. A toolkit that supports developers in their efforts in programming applications for the Fiduciary Glove. This includes an API that provides easy access to hand parts, hands, users, postures and gestures. This API will be self-documenting to a great extent. In any case, an elaborate documentation will assist developers in learning the API. The toolkit also features utilities that help developers to define, adjust and test custom gloves, postures and gestures.

1.7 Overview

The following paragraphs will provide you with a quick outlook on the remaining chapters of this thesis:

Chapter 2 - Related Work

This chapter gives an introduction into the research field of this thesis. In the first section of this chapter I give a detailed overview over approaches for the identification of hand parts, hand and users. Then, toolkits and APIs in the area of touch input and tangibles are introduced. The next section presents prior work about design guidelines for toolkits, APIs and frameworks. Furthermore, this chapter includes related work about recognition engines and definition languages for postures and gestures.

Chapter 3 - Technical Background

The technical foundations for the Fiduciary Glove and the Fiduciary Glove Toolkit are illustrated in this chapter. This includes a quick introduction into the Microsoft Surface hardware and the Surface SDK. At the end of the chapter, the $1 recognizer by Wobbrock is described. It serves as gesture recognition engine in our toolkit.

Chapter 4 - Fiduciary Glove

Basis and motivation for this thesis is the Fiduciary Glove. The first part of this chapter is concerned with the glove itself and the design iterations. Then, the development of a painting application for the glove and the involved issues are described.

Chapter 5 - Fiduciary Glove Toolkit

The main chapter of this thesis deals with the development of the Fiduciary Glove Toolkit. This chapter begins with the definition of the term toolkit. Afterwards toolkit qualities and guidelines for the design of toolkits and APIs are presented. In the following section the design of the actual Fiduciary Glove Toolkit is illustrated. This includes a description of the API, utilities and widgets. Then, the documentation of the toolkit is highlighted. The remainder of this chapter deals with the seamless integration of the toolkit into the development environment.

Chapter 6 - Sample Applications

Chapter 6 describes the development of sample applications using the Fiduciary Glove Toolkit. A reimplementation of the painting application from Chapter 4, using all features of the toolkit, is presented in the first part of this chapter. A comparison of the old and new version serves as a first benchmark for the toolkit. The second sample application is posture trainer that illustrates rich usage of postures.

Chapter 7 - Conclusion

The last chapter begins with a suggestion of future work and a summary of contributions of this research. Finally, a conclusion is drawn with hindsight on the work presented in the previous chapters.

2 Related Work

In this chapter I present related work that constitutes the academic basis of this thesis. The literature includes work about hand part, hand and user identification, glove-based tracking, toolkits, toolkit design and gestures and postures.

2.1 Hand Part, Hand and User Identification

In related work various approaches have been presented that deal with the recognition of particular hand parts, mostly the fingers or the distinction between the hands of one person or between multiple people.

2.1.1 Computer Vision Approaches

Already in 1985, Kruger introduced VideoPlace [33]. In what he called an artificial reality, the silhouette of a person was tracked by a video camera. The positions of both hands were then deduced from the silhouettes and used to determine certain hand postures. These postures, together with hand movement in space, triggered changes in the appearance and behaviour of several onscreen objects.

Later methods are able to recognize not only hands but fingers. [35], for example, suggested a set of algorithms with which the images of a video camera can be processed to improve finger recognition. First the foreground is extracted. This is done by analyzing the difference between a background image that was taken before the hand reached into view and the current video camera image (cf. Figure 2.1). The resulting similarity map is then transformed to a binary image by applying a threshold. Through shape filtering the binary image fingertips are then detected.

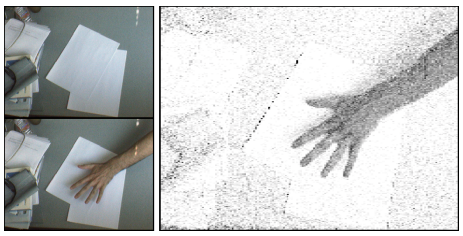

Figure 2.1: Similarity map [35].

The Visual Panel [55] developed by Zhang et al. detects a single fingertip over a white rectangular object (cf. Figure 2.2). This object, in the paper a simple piece of cardboard, can then be used as surface for touch-input. It can act similar to a touchpad by mapping each corner of the object to the corresponding corner of a display. Since a touch is recognized based on an image recorded from above the object, the system is not able to distinguish a real touch from a finger hovering over the object.

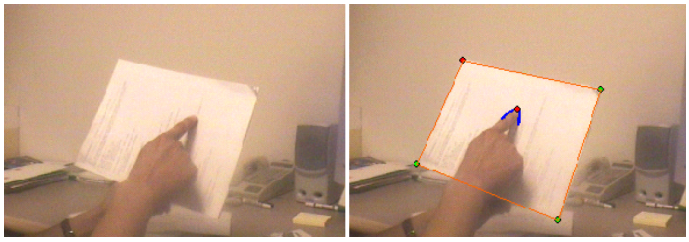

Figure 2.2: Visual Panel [55].

Similarly the EnhancedDesk [41] is able to detect fingertips with a single infrared camera mounted above an interactive rear-projected tabletop (cf. Figure 2.3). In contrast to the Visual Panel, multiple fingers can be identified through template matching. The system is also capable of recognizing symbolic gestures performed with one or two fingertips. The gestures were trained with 80 samples each.

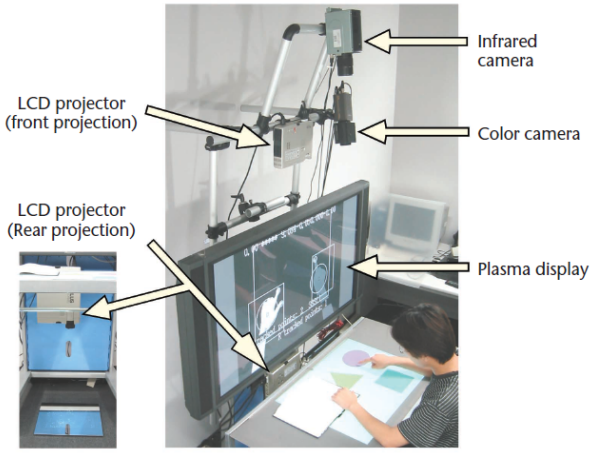

Figure 2.3: EnhancedDesk [41].

Schlattman et al. [46] described a hand tracking system with can track a hand in '6+4' degrees of freedom. This means that it can not only identify which of four predefined static hand gestures the hand is forming, but also its translation and rotation. For the hand recognition, the system calculates the visual hull of a hand from images from at least three cameras (cf. Figure 2.4). Since the four gestures were selected so that they have jutting hand parts, the finger tips can be retrieved with simply feature algorithms.

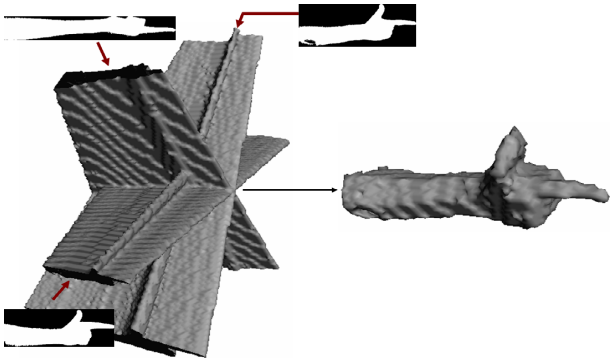

Figure 2.4: Visual hull of a hand [46].

The Microsoft Surface [2] visually infers fingertips from blobs in the raw image data received from the built-in infrared cameras. While the position is accurately recognized, the orientation is subject to a little jitter. This jitter is highly dependent on the angle in which the finger touches the surface. The Microsoft Surface is described in detail in Chapter 3.1.

Dang et. al [15] analyze groups of touch blobs to distinguish if finger touches belong to the same hand. Therefor they calculate the distances and angles between the blobs and match them with possible finger arrangements of human hands. According to the authors, the system is not as reliable as it could be due to the flexibility of the thumb. It was the main reason for false recognition during the evaluation phase.

2.1.2 Specialized Hardware

Besides computer vision, specialized hardware can be used to identify hand parts, hand and users. The MERL DiamondTouch surface [17] identifies which touch came from which person through capacitive coupling. Beneath the screen lies a grid of antennas (cf. Figure 2.6). Each antenna transmits a unique signal. When a person now touches the surface, she capacitively couples with the antennas beneath the touch position. The antenna signals are transmitted through the body to a chair or mat the person sit or stands on. Depending over which chair or mat the system receives the signal from the antennas, it can assign the touch to the person in contact with it (cf. Figure 2.5). Because the antennas are not transparent, the screen of the DiamondTouch has to be top-projected.

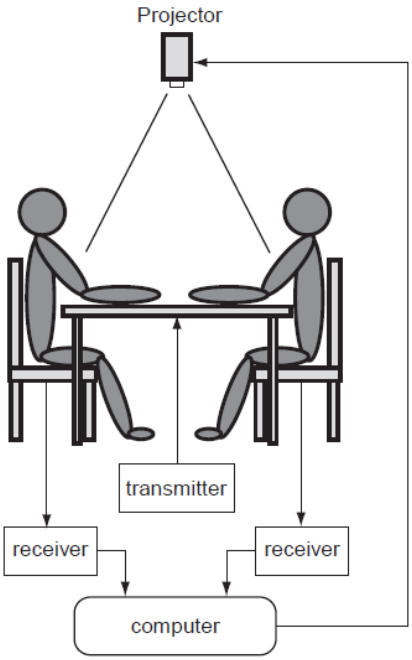

Figure 2.5: DiamondTouch capacitive coupling [17].

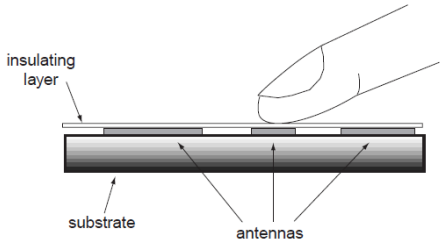

Figure 2.6: Antennas beneath the top-projected surface of the DiamondTouch [17].

M. Wu and Balakrishnan [54] used the shape of the DiamondTouch signals to recognize fingers and postures (cf. Figure 2.7). Therefor samples of the signal shape of fingers and postures were recorded and then matched with the current signal shapes.

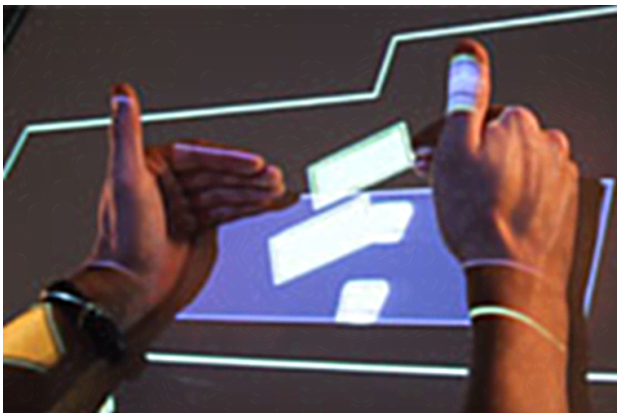

Figure 2.7: Corner-shaped postures for shape definition [54].

SmartSkin by Rekimoto [42] is based on a sensor architecture. The sensor consists of a grid of transmitters and receivers. The transmitter and sender electrodes run orthogonally to each other (cf. Figure 2.8). Each transmitter sends a signal with an unique frequency and voltage. This is picked up by the receivers at the intersections. When a conductive object is close to an intersection, it capacitively couples to the electrodes which drains a certain amount of the transmitted signal strength. Consequently, the same happens with the received signal strength. Therewith a hand can be recognized, and if the grid is dense enough even fingers. Rekimoto et al. used this capability to detect postures with multiple fingers.

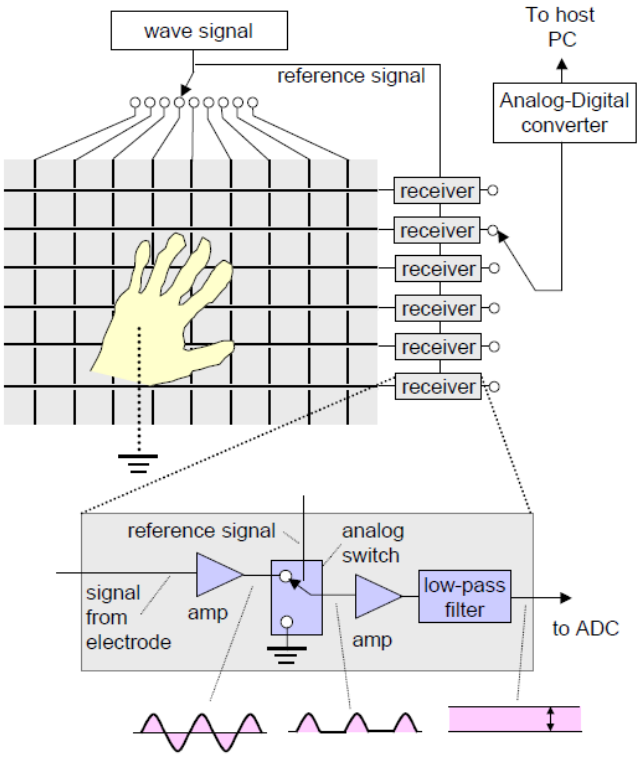

Figure 2.8: SmartSkin sensor grid [42].

Benko's approach to derive which hand part caused a touch was to attach muscle sensors to the forearm (cf. Figure 2.9). After recording the signals for touches with certain hand parts these samples built the basis for the later recognition that estimates finger identity and touch pressure [9].

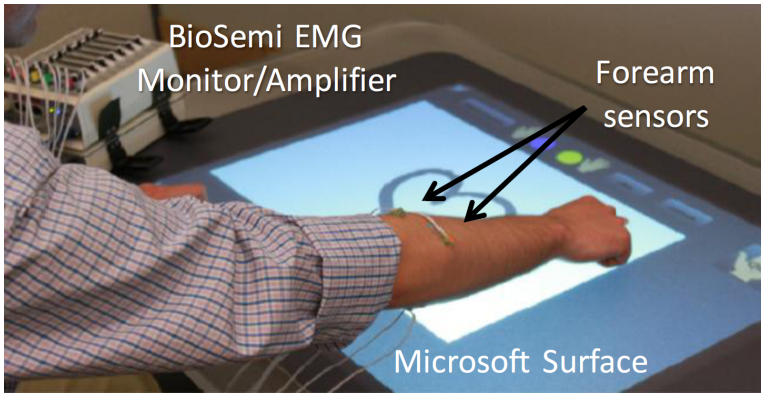

Figure 2.9: Electromyography sensors on forearm [9].

The RidgePad introduced by Holz and Baudisch [26] is based on a fingerprint scanner. The system can extract roll, pitch, yaw and user ID from a touch. A major limitation of this device is time complexity. The used fingerprint scanner requires a noticeable time before it can transmit its image. The subsequent code sequence for fingerprint-comparison takes 200-300ms per comparison with existing fingerprints in the database.

2.2 Glove-Based Tracking

Glove-based tracking is well known in virtual reality environments. Sturman [48] and Malik [37], for example, both survey ways to track gloves, e.g., by optical tracking, markers, silhouette analysis, magnetic tracking, optical fibres.

More recent methods include vision-based recognition of colored gloves to discriminate parts of the hand [51]. The hand part identification is simplified by the unique layout of colored patches on the glove (cf. Figure 2.10). The patches also obviate an issue of other visual approches: Those cannot reliably tell if the front or the back of the hand faces the camera. This can lead to false interpretations of hand parts, e.g. the thumb is recognized as pinkie, or a left hand as a right hand.

Figure 2.10: Color glove [51].

Glove-based tracking that uses fiduciary tags is also known in augmented reality environments, where it is used primarily as a way to calculate gestures made in 3D-space, and as a way for a person to handle virtual objects (e.g., Buchmann [11], Figure 2.11).

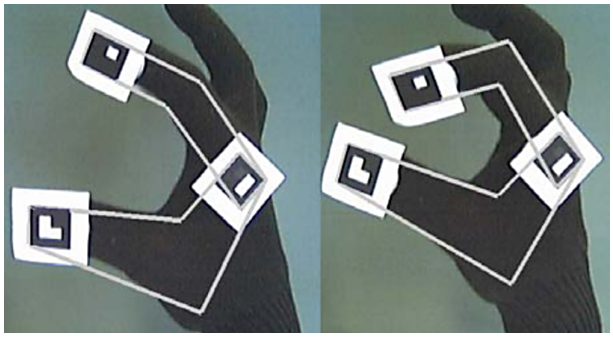

Figure 2.11: FingARtips glove [11].

2.3 Toolkits

Toolkits that handle touch-input are available for most commercial interactive surfaces. The Surface SDK ships with the Microsoft Surface [2]. It serves as basis for our toolkit. Therefore, this toolkit will be illustrated in Chapter 3 which covers the technical background of this thesis. The Smart SDK [6] is provided by Smart Technologies. It enables developers to program multi-touch applications with up to 120 touches for their products, e.g. the SmartTable [7]. It includes prebuilt multi-touch capable interface elements, such as textregions, a rich documentation and code examples. It also provides project templates for the IDE to simplify the creation of a new application.

The DiamondSpin SDK [47] was developed at MERL to support the development on the DiamondTouch table [17], especially for around-the-table interaction. Items onscreen are automatically reoriented towards the outside of the DiamondTouch table. Through this automatism developers don't have to worry about low-level implementation details like the translation from polar to Cartesian coordinates. Rapid prototyping was facilitated by the SDK's capability of reusing and extending existing Java Swing components.

Diaz-Marino et al. [16] introduced a toolkit for the DiamondTouch hardware which supports the development of multi-touch and multi-user applications. The API is exposed through types and their properties, methods and events. The toolkit provides high-level events and incorporates gestures such as finger brushing to extend the standard capabilities of the DiamondTouch.

Hansen et al. [23] illustrates the usage of PyMT for HCI research. PyMT [5] is an open-source python library for multi-touch. It's based on Phyton which does not need to compile and is plattform independent. PyMT enables therefore flexible and rapid prototyping, without porting. It contains a set of reusable widgets that speeds up the development process, especially for average skilled programmers.

Even though, we could not identify a toolkit that supports the programming for a glove, we could find research in the area of toolkits for tangibles that are tracked on digital surfaces. The reacTIVision [28] is a computer-vision based framework which enables the tracking of objects tagged with fiduciary markers. Papier-Mâché uses RFID as well as computer-vision to identify and track objects [31].

2.4 Toolkit Design

The majority of the design decisions for the Fiduciary Glove Toolkit are based on the guidelines suggested by Cwalina and Abrams. Their book 'Framework Design Guidelines' [14] presents conventions, idioms, and patterns for reusable .Net Libraries. Its focus on .Net libraries demands its consideration for our toolkit which is based on the Surface SDK. The book chapters cover naming, type design and usage guidelines amongst other things. Its most valuable guidelines are later described in detail in Chapter 5.2.

Stylos et al. [50] examined the implications of method placement on API learnability with a study. Their first finding is that the study subjects had a common programming strategy. According to the authors, the programmers first tried to find an appropriate starting class. Therefor they browsed the class list in the documentation, then looked up details of the potential candidate in the documentation. Hence, a good naming scheme for types is very important. After an appropriate class was found the programmers browsed properties and methods by using the code completion. Stylos implies that important methods should therefore be in the class from which the method is most likely called. He illustrates this with an example of the send method in a mailing application (cf. Code Example 2.1): from a purely technological point of view, the send method would be placed in a class representing the mail server, of course. Since developers will rather start with the message class than the server-class, Stylos suggests to place the send method in the message class instead of the server class. The advantage is that programmers don't have to locate the server class in order to send the message. On the contrary, the parameter in the send method of the message

class gives a hint to it. Referencing other needed classes in at least one of the methods of the start class helps programmers to find these and use their methods. Even though this might result in a method placement that is logically incorrect, it helps to improve the usability of an API. Also, if programmers do not find an expected method, they might question their correct starting class choice.

```
MailMessage mailMessage = new MailMessage();
// the class mailServer has to be found by consultating the ↩
   documentation
mailServer.send(mailMessage);
// parameter type gives an implicit hint which class to use.
mailMessage.send(mailServer);
```

Code Example 2.1: Method placement comparison.

Robillard conducted a survey that was answered by 80 developers with different skill and experience levels [44]. The survey focused on learning strategies and obstacles faced while learning a new API. In matters of learning strategies, most of the developers stated that they learned APIs by reading documentation (78 percent), followed by using coding examples (55 percent). Some answered that they experimented with APIs (34 percent), read articles (30 percent) or asked colleagues (29 percent). Only few named books or code tracing through a debugger as strategies for learning an API. From the questions regarding obstacles Robillard derived five major categories, namely: 'Resources', 'Structure', 'Background', 'Technical environment' and 'Process'. The last three categories represent mainly personal or specific issues causing obstacles with an API and will be not elaborated on therefore. Insufficient or inadequate examples, incomplete or inadequately presented documentation, and missing reference on how to use the API for a specific task were mostly mentioned in the category 'Resources'. Thus, we put a major emphasis on creating extensive and understandable examples and documentation for our own API. It is important to avoid a mismatch between the example's purpose and the user's goal. One solution according to Robillard is to provide different types of examples, so that a developer can move forward to a more complex type if the simple type does not fit the needs anymore. The following three types of examples are suggested: code snippets, tutorials and complete applications. Snippets are small code fragments showing how to access the basic API functionality. Tutorials are longer and describe how to implement a more complex scenario with the API, step by step and maybe in prose. Complete applications are code exmaples of how to use several aspects of the API in one application. As reasons for using examples some experts quoted that examples provided 'best practices' of API use, informed the design of code that used the API, provided validation about an API design and confirmed developers hypotheses about how specific goals can be accomplished. Issues with the API's structural design and the API's testing, debugging and runtime behaviour constitute the 'Structure' category. Robillard describes how the API's high level design would often explain puzzling behaviour, but is not reflected in the low-level documentation. The API behaves according to the design, but other developers can't interpret and understand this behaviour on basis of the given documentation. Another reason for puzzling behaviour originates in inconsistent design, which makes it impossible to predict the behaviour by comparing similar elements or scenarios. We added remarks to our documentation that describe where and how the described element fits into the high level design.

Zibran summarized design guidelines for API design [56]. His findings are concentrated to five characteristics that indicate API usability: An API is usable if it is:

- easy to learn,
- easy to remember,
- easy to write client code,

- easy to interpret client code and
- difficult to misuse.

In the following section Zibran describes several factors that influence these five categories. Complexity influences all categories. If an API is very complex, it's harder to learn, harder to remember, harder to write own code, harder to read code from others and easier to misuse. Naming plays a significant role for the usability of an API. Names should be self-documenting in the best case. Consistency is key: Same names should mean the same thing, different names different things, and naming conventions should be followed. What Zibran names 'Ignorance of Caller's Perspective' means that an API should be designed from the perspective of the caller. Replacing Boolean parameters with values from an enumeration for example make the code instantly more understandable, since the values carry a semantic meaning. If the caller only sees 'true' or 'false' he has to resort to the documentation to find out what the value actually means. Even though an API should be usable with the least consultation of documentation - according to Zibran - an API without good documentation and code examples is typically difficult to use. Both have to be complete, clear and errorless. Inconsistency and ignoring conventions make an API extremely difficult to learn, use and read. Also, conceptual incorrectness leads to difficulties in learning and using an API. For example, using a list when a set is needed is conceptually incorrect, because lists allow duplicates while sets don't. This can cause problems in certain situations when duplicates are syntactically allowed but conceptually not right. Inappropriate method parameters and return types can hinder the learning process. Zibran suggests following Bloch's guideline of usually using not more than three parameters. In addition, the ordering of parameters across similar methods should be consistent. Returning an empty collection or object is preferable to returning simply null. Parameterized constructors should be optional, not mandatory. A default constructor without parameters should always be provided if possible, because this is naturally expected by many programmers. For the same reason Zibran discourages from using the factory pattern, also a guideline suggested by Ellis, Stylos and Myers [18]. It is more complicated than simple constructors and its advantages don't outweigh the loss of accessibility. Choosing proper data types will lower the work load for developers. Avoiding strings if a more fitting type exists. Error handling and exceptions both prevent the user from misusing the API. Error messages should include all the information available to describe the problem and at best a solution for repair or recovery. Exceptions should only be thrown for unexpected outcomes and not be overused, because it makes the API more difficult to use. Programmers have to handle exceptions on their own, and should not be bothered with it for no reason. An API with high usability should not have 'leftovers for client code'. This means that all necessary functionality is included. Zibran states: 'If the API can implement a functionality that the user needs, it should not be left for the user to implement'. In order to give more flexibility to the users of a function, you should use an interface as parameter type instead of an actual class implementation. Through this mechanism the user can decide whether to use a given implementation or use an own class.

'API Design Matters' [24] illustrates how important good design is for the usability of an API. The author Michi Henning admits that this is a hard task and opportunities for doing it wrong are plenty. The cost of poor API design is large and inefficient code. Beyond this, poor APIs are harder to understand and to work with. While coding against a good API is fluent and stressless, poor API design takes up more time to get started with and more lines of code in the end. Also, the code complexity grows. This entails higher testing effort and increased likelihood for bugs.

In 'Toolkit and Interface Creativity' Greenberg [21] states that toolkit design should be iterative just like the design of a normal application. When developing the toolkit and first applications with it side by side, application ideas that prove useful can be integrated into the toolkit right away. All applications can then use the newly available functionality of the toolkit. Also, if a feature of the toolkit turns out to have a suboptimal design it is detected early and can be refined instantly.

By 'Mapping the Space of API Design Decisions' [49], Stylos and Myers put design recommendations from experienced API designers, comparative studies and informal online discussions into context. As a first step the authors identified three stakeholders of an API: API designers, API users and product consumers. API designers develop the API itself. 'Some of their goals are: to maximize the adoption of an API, to minimize the support costs, to minimize development costs (this is perhaps less important since it is a one-time cost), and to be able to release the API in a timely fashion.'. API users develop own applications using the API. Their goals are to use an API that allows for quick, stressless and error-proof programming. Product consumers are indirectly affected by API design because they will be the users of the final product that was built with the help of the API. The goals of consumers are to have a product that serves its purpose, follows consistencies and has no bugs. Then, Stylos and Myers describe desired qualitites for APIs. At the highest level they identified two qualities: usability and power. Usability involves learnability, productivity, error-prevention, simplicity, consistency and matching mental models. Expressiveness, extensibility, evolvability, performance and robustness constitute the quality power. The impact on which of the stakeholders these aspects of usability and power have is depicted in Figure 2.12.

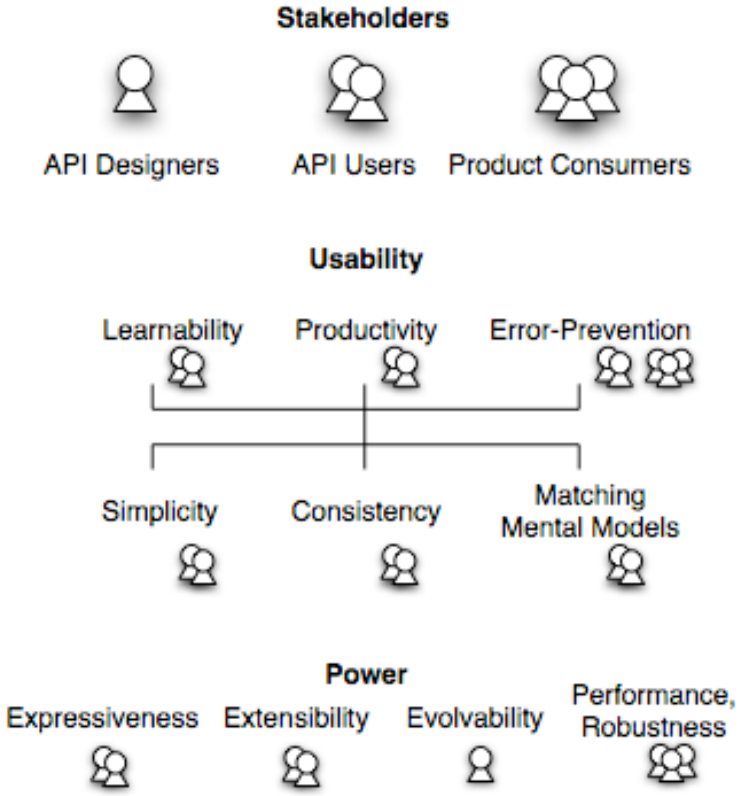

Figure 2.12: Qualities of an API and affected stakeholders [49].

In the next section, Stylos et al. map out the space of API design decisions. Therfor, they group design decision into two categories that were identified after several iterations. The first category is class design decisions versus structural design decisions. The second category puts specific programming language features and architectural features into opposition. Then, this categorized space is filled in with the topics of the API design recommendations gathered from API design experts (cf. Figure 2.13). This map of the space of API design decisions shall help to get an overview over the amount of API decision that there are and a better understanding of the field of API usability.

In the very beginning of 'How to Design a Good API and Why it Matters' [10] Joshua Bloch points out the importance of an API from two perspectives. The first is when an API is good.

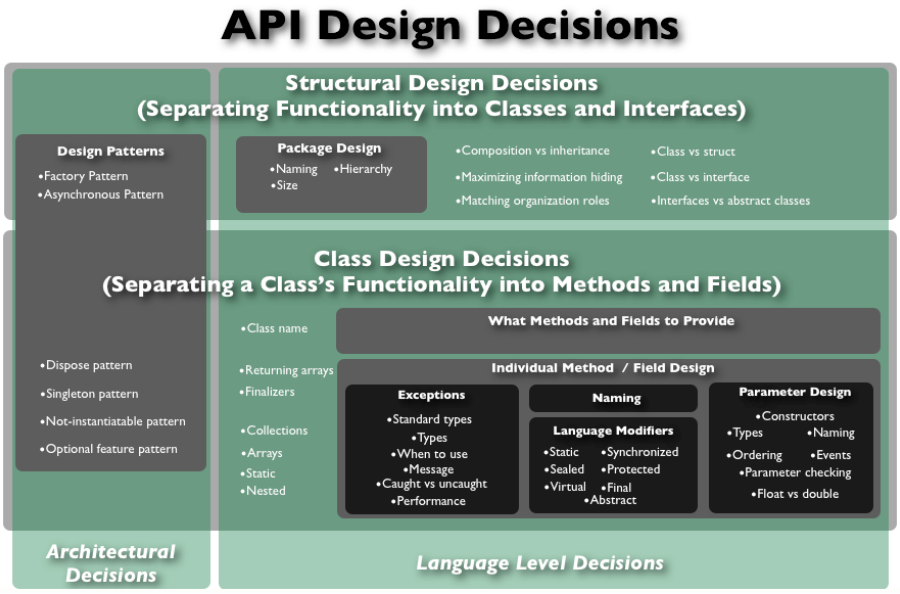

Figure 2.13: Map of the space of API design decissions [49].

An API can be 'among [the] company's greatest assets', when it is well designed. The second perspective is when something along the way went wrong, when the API is not usable. Not only the development costs rise when an API is not well-designed, but also the support costs. Issues with APIs are tried to be solved in dialogue with the developers of the API. For Bloch a good API is defined by the following characteristics. It is:

- easy to learn,
- easy to use (even without documentation) and
- hard to misuse.

Moreover is produced code easy to read and maintain. The API should be powerful enough to meet the requirements and be easy to extend. And last but not least it is mandatory that the design of the API fits the audience. Bloch presents general principles that help to design an API with these characteristics. '[An] API Should Do One Thing and Do it Well' is the first of the principles and means that the functionality of an API should focus on one thing. A bad sign is if functionality is hard to name. The second principle is: 'API Should Be As Small As Possible But No Smaller'. Even though an API should be complete, there is no room for unnecessary functions. When you are not sure, if functionality should be provided by the API, leave it out. You can always add it later, but never remove it. If something exists, people will use it. Consequently if you remove the functionality in a newer version of the API, the programs that used it will crash. As a next principle Bloch argues for minimizing the accessibility of everything. Public fields except for constants should be avoided in public classes. The more information is hidden, the more independently modules can be handled. Like in most other papers about API design Bloch attaches great importance to naming. Names should be self-explanatory, documentation should not be necessary to understand the meaning of whatever is named. Consistency plays a great role here - 'same word means same thing'. This is not only meant for the API itself, but also across other APIs on the platform. An indicator for a good naming scheme is when the code reads almost like prose. Even though naming should work without it, documentation is an obligatory element of every API. Every class, interface, method, constructor, parameter and exception has to be documented according to Bloch. Documentation helps you to understand the context and

concept better. When designing an API, the usability-to-performance ratio has to be considered. Decisions for convenience functions can lead to bad performance. The author mentions making a type mutable as one example for a bad design decision that limits the performance of the API. Yet, usability comes first. You should not change the API just because of slight performance issues. In the next sections Bloch elaborates on these principles. In terms of class design he argues for minimizing mutability as aforementioned. Classes should only be mutable if there is a good reason. If a class has to be mutable, you have to keep the state-space small and well-defined. Subclasses should be implemented only where a 'is-a' relationship exists. If a subclass is inevitable, it has to be documented how methods use one another. Regarding method design the API designer should think of convenience functions a developer could need besides the core functionality. Bloch calls this boilerplate code. This type of code is typically generated through copy-and-paste, annoying and error-prone. Therefore it has to be reduced to a minimum. Helper-functions like writing an XML file to an output stream should be implemented in the API if they will consistently be needed by API users. Also, methods should never violate the principle of least astonishment. Comment especially on extraordinary behaviour of a function for that reason. If errors occur, let the API code fail fast. If possible, handle errors at compile time - through generics or static typing for example. At runtime throw an exception after the first incorrect method call and as the name implies to indicate a exceptional condition. Each method should be failure-atomic. This will let developers narrow down the source of failure very quickly. Next, methods should be overloaded carefully. Indistinct overloading is to avoid. If two versions are ambiguous you should rethink the overload. API developers should also consider using a different method instead of an overloading. In methods with multiple parameters, especially when overloaded, use consistent parameter ordering and limit the parameter count to three or less. For parameters, interface types are preferable over classes for the sake of flexibility. Parameter types should also be as specific as possible. This simple principle shifts the error from runtime to compile time. Bypass strings if a better type exists. Strings are tedious to work with while other types can be handled effortlessly. Double is preferable to float parameters or return values because the loss of performance is minimal, the precision gain however is significant. Return values that require extraordinary processing should be replaced by empty values of the same type as the typical return type. Returning an empty collection instead of null makes a check for null unnecessary.

'Measuring API Usability' [13] contains several API design guidelines suggested by Steven Clarke. He claims that a user-centered design approach is one of the best ways to a usable API. Therefore you need to understand your user first. In the case of an API this is the developer. Clarke uses the cognitive dimensions framework to describe how API and delevoper characteristics have an impact on each other and influence the usage of the API. The twelve dimensions that affect the way developers work with an API and expect it to work are:

1. Abstraction level: The minimum and maximum level of abstraction provided by the API versus the abstraction level extremes actually used by the targeted developers.
2. Learning style: The learning style required by the API versus the learning style of the targeted developers.
3. Working framework: How much of the APIs concept has to be understood so that effient working is possible versus is understood by the targeted developers.
4. Workstep unit: How big a workstep in the API is versus how big of a workstep can be handled by the targeted developers.
5. Progressive evaluation: To which extent can partially completed code be executed to obtain feedback on code behavior.
6. Premature commitment: How many decisions have to be made by the developer and what the consequences of those decisions are.

7. Penetrability: To what extent the API supports exploration, analysis and understanding of its components.
8. API elaboration: How much of the API has to be adapted to meet the needs of a targeted developer.
9. API viscosity: How difficult it is to make changes in the API.
10. Consistency: How much of the usage of the API can be inferred, once a part of it is already learned.
11. Role expressiveness: To which extent components match with their roles in the program as whole.
12. Domain correspondence: To which extent components map to the domain.

Clarke then described three personas to profile and characterize different developer groups: The opportunistic, the pragmatic, and the systematic developers. Programmers of the first type, with opportunistic stereotypical behaviour, appreciate convenience features that raise productivity. Having full control over their code is only a secondary to them, since the cost for this control is higher development effort. Hence predefined controls and black-box components are very important to this type. A lack of these would keep them from using an API. Pragmatic programmers in contrast tend to look for a balanced control-productivity-ratio. They value productivity features. At the same time they accept a longer development time when certain situations call for custom solutions. If control is mandatory they deal with low level problems. Systematic programmers constitute the most defensive type of programmer. Their focus lies on full control over the application. Hence, they do not use an API before having a clear understanding of the technology behind and do most of their work in code.

With the help of the twelve dimensions, an API analysis can be compared to the persona of an targeted developer group. This way the usability of an API for a developer group can be measured. An exemplary comparison is illustrated in Figure 2.14.

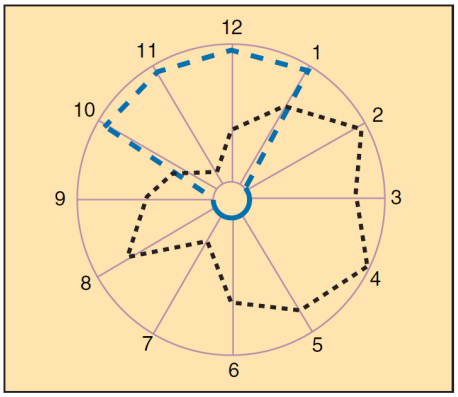

Figure 2.14: API analysis and developer profile comparison [13].

Myers introduces the concept of 'low threshold' and 'high ceiling' in [40]. Low threshold means that it is possible for beginners to get started with a system, and high ceiling means that experts can achieve their more complicated goals. [43] adds attribute to these two the concept of 'wide walls'. Wide walls means that a wide range of exploration is provided, that a great variety of applications is possible and facilitates through the tool.

Butler et al. [12] and Johnson [27] illustrate various methods of documenting frameworks. They suggest to offer different types of documentation to assist many types of developers. Their

suggestions include framework overview, examples, cookbooks and recipes. The overview helps developers to understand the context and purpose of the toolkit. Examples are applications that are developed with the toolkit and especially designed for documentation. Cookbooks and recipes should use the example applications as concrete reference for their discussion and illustration. Cross-referencing between recipes is helpful and facilitates the exploration of the toolkit functionality.

In [20] Gould motivates his work with a list of different aspects of usability. The most important ones in terms of toolkits are system performance, system functions, outreach materials, and installation. System performance includes system reliability and robustness, which basically means that if a system is unavailable, it is also not usable. System functions means that all required functions should be provided. Outreach materials are online help and reading materials, i.e. documentation. The installation should not pose an obstacle for the user, otherwise the system is not usable. Then he describes four requirements for usable systems: early focus on users, empirical measurements, iterative design and integrated design. Early focus on users accompanies empirical measurements. Users should be involved early in the development process and should continuously give feedback. With this feedback the system can be iteratively refined. Integrated design means that all aspects of usability as described above evolve concurrently. Therefor, the development should be conducted or supervised by one person that coordinates the usability.

2.5 Postures and Gestures

Some related work about postures and gestures has already been mentioned in the other sections of this chapter since several authors introduced their methos for hand part identification or toolkits together with concepts about postures or gestures.

Freeman et al. [19] present ShadowGuides which can be used to teach gestures. When a hand touches the surface, the raw image as captured by the interactive surface is shown as shadow with a little offset next to the hand. This gives visual feedback of how the device perceives the current hand posture. ShadowGuides consists of two parts: a registration pose guide, that teaches users how to perform so-called registration hand poses that serve as a starting point for gestures. The second part are the user shadow annotations. The user shadow annotations guide a user through a gesture. Once a registration pose is performed the user shadow annotations for all possible gesture that have this pose as starting point are displayed. Color-grading helps to distiguish the annotations of the different gestures. User shadow annotations are based on a single-point dynamic guides introduced by Bau [8]. A single-point dynamic guide shows paths for gesture completion only for the gestures that can be performed in the current state; they are continuously updated. User shadow annotation extend this with arrows, shape deformation keyframes and dynamic markers. Arrows are used for multi-finger gestures and should indicate the direction in which the gesture has to be performed; it helps additionally to differentiate them from single-finger gestures. Shape deformation keyframes show how the shape of the gesture changes over time. Dynamic markers highlight changes in a posture that go beyond simple movement or shape changes. When, for example, a new finger has to be added to the hand pose at a certain time, a dynamic marker is displayed at an appropriate time and position.

With the Artkit [25] by Henry et al. developers could already integrate gestures in user interfaces in 1990. Henry used so-called sensitive areas to enable gesture recognition. These areas were invisible rectangles that accepted gestural input and handled the gesture alphabet. Multiple of these sensitve areas could also handle different alphabets, allowing for some kind of gesture encapsulation in objects.

Landay and Myers offer an approach for extending an existing user interface toolkit to support gesture recognition [34]: Gesture interactors. In order to integrate gestures in a interface, a developer can create a gesture interactor. It has to be specified which event causes the interactor to start and where this event must happen on screen. Then, a gesture recognition engine and a

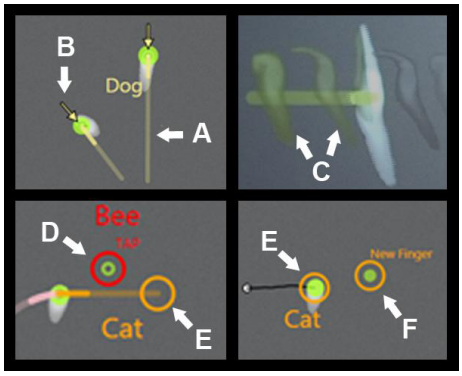

Figure 2.15: A: Dynamic guide, B: Arrow, C: Shape deformation key frames, D,E,F: Dynamic markers. [19].

corresponding callback method are set. This method is called when the gesture recognition engine returns its result. When a gesture is performed, an event is raised and the gesture passed to the gesture recognition engine. In the callback method, the developer can decide how to handle the gesture result.

The GRANDMA toolkit [45] by Rubine enables the easy integration of gestural input to custom applications. The system recognizes single-stroke gestures after a training period. It lets the developer define the effect of each gesture. The attributes of a gesture are fully accessible which enables the implementation of complex scenarios based on the characteristics of the gesture and not only the gesture name.

M. Wu and Balakrishnan [54] used single-hand as well as two-handed postures in their system. As single-hand postures they introduced: a flat hand, a vertical and horizontal hand (both with only the side of the hand touching the surface), and a tilted horizontal hand. Two-handed postures were: two vertical hands and two corner-shaped hands. Especially the two-handed postures were not simply assigned to certain functions but also enriched these with its characteristics. For instance, the posture with two corner-shaped hands copies the spanned area defined by the two corner shapes.

Khandkar et al. present GestureToolkit, which includes a gesture recognition engine as well as a definition language for gestures [29]. It is device independent since the touch-input provider can be exchanged. It also comes with a hardware simulator that simulates touch-input for testing applications on the development machine. The abstraction level of the definition language hides low-level details.

3 Technical Background

This chapter illustrates the technical background for both the Fiduciary Glove and the Fiduciary Glove Toolkit. This includes description of the Microsoft Surface hardware as well as the Surface SDK and the fiduciary markers that ship with the hardware. The end of the chapter covers the gesture recognition engine we adopted for our toolkit.

3.1 Microsoft Surface

For the development of the first application for our glove and also the Fiduciary Glove Toolkit, we used a Microsoft Surface 1.0 developer unit (cf. Figure 3.1). It is an interactive tabletop surface based on diffuse rear-illumination.

On top of its housing sits an acrylic screen. A diffuser prevents ambient light from penetrating the surface and at the same time enables the multi-touch capabilities of the device. Inside, a projector, an infrared light source and five infrared cameras are aimed at this screen. The projector is a normal DLP (Digital Light Processing) rear-projector and has a resolution of 1024 by 768 pixels. The infrared light source consists of multiple LEDs and emits infrared light with a wavelength of 850 nanometer toward the screen. If nothing touches the screen, the infrared light is scattered by the diffuser and only little light reflects back. However, when something IR-reflective, such as skin, touches the screen, more light is reflected at this position (cf. Figure 3.2). The five infrared cameras pick up this reflection and therefore recognize touches. Their resolution of 1280 by 960 pixels is high enough for the identification and tracking of multiple fiduciary markers simultaneously touching the screen.

In order to improve the recognition, various filters are applied to the received image. The Microsoft Surface includes an application which shows a stream of the processed raw images. The raw image is then processed and depending on the form factor of the touch, a finger, a blob or a tag are recognized.

3.2 Surface SDK

First of all, the Surface SDK includes a library, enabling the rapid prototyping for the Microsoft Surface. It can be used for application based on Windows Presentation Foundation (WPF) or XNA. From a pure programmer's point of view, writing a Surface application does not differ too much from programming a conventional WPF desktop application. The main difference is that Surface windows and components handle contact events in addition to the traditional input events

Figure 3.1: Microsoft Surface 1.0. [http://www.microsoft.com/presspass/presskits/surfacecomputing/gallery.mspx#0]

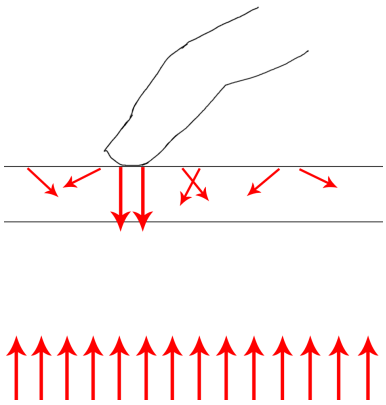

Figure 3.2: Diffuse rear-illumination.

Contact event type	Description
ContactChanged	Occurs when a contact over an element changes its attributes, including position and orientation.
ContactDown	Occurs when a contact over an element is placed on the Microsoft Surface screen.
ContactEnter	Occurs when a contact enters an element's boundaries or is placed on the area inside the boundaries.
ContactHoldGesture	Occurs when the Microsoft Surface software recognizes a press-and-hold gesture.
ContactLeave	Occurs when a contact leaves an element's boundaries or leaves the area over the element.
ContactTapGesture	Occurs when the Microsoft Surface software recognizes a tap gesture.
ContactUp	Occurs when a contact over an element leaves the Microsoft Surface screen.

Table 3.1: Contact events.

like the down event for mouse buttons and keys. The different contact event types are listed in Table 3.1.

The arguments of a contact event include information about the identifier, contact type, position and orientation. Furthermore, they provide a contact bounding rectangle, a contact ellipse that represents the contact area on the surface and a timestamp that indicates when the event happened. If the contact was a tag, the tag type and tag value can also be accessed.

The Surface SDK provides several tools that support the development.

The Surface Simulator that can be used to test and debug applications on other machines than the Microsoft Surface. This allows developers to work at their own desktop most of the time and use the actual device only when inevitable.

The Input Visualizer depicts input information that the Microsoft Surfaces receives and processes. For instance, if you touch the surface with a finger it displays a semicircle with the closed end at the fingertip (cf. Figure 3.3). An arrow emphasizes the recognized orientation of the finger. A label shows the identification number for the finger. As long as the device does not lose track of the touch, the identification number stays associated with the touch and keeps the same value. When the finger is lifted or moves too fast, the tracking is lost. If the same finger touches the surface again, a new identification number will be assigned. This is the striking feature of fiduciary markers. Additionally to an identification number they have a tag value that never changes, no matter how often they are lifted from and placed down on the surface again. Also the ori-

entation of a marker is recognized more accurately because it is not calculated from a blob but based on certain characteristics of the tag (cf. section Fiduciary Tags below, Figure 3.4, source: [http://msdn.microsoft.com/en-us/library/ee804813%28v=surface.10%29.aspx])

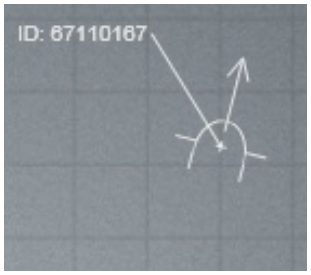

Figure 3.3: Input Visualizer (finger).

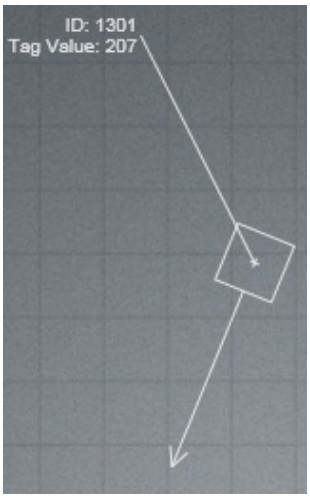

Figure 3.4: Input Visualizer (tag).

Surface Stress is an application for stability and robustness tests. In random mode, it generates multiple contacts of fingers, fiduciary markers (both types) and blobs in a random fashion. In parameterized mode, you can define the following stress factors manually: contact types used, active area on screen, relative density of contacts, maximum contact velocity and length of contact movement. The stress test stops in both modes after a predefined duration.

The Identity Tag Printing Tool serves as a tool to create custom identity tags with a regular printer as the name already suggests.

3.3 Fiduciary Tags

We picked tags of the type shown in Figure 3.5 for three reasons. First, they come with the Microsoft Surface which recognizes them out of the box. Second, the Surface SDK provides a very convenient interface to access the following information provided by these tags: The position and orientation on the surface as well as an eight bit tag value. The third criterion that let us choose this type of tag was its performance. With performance we mean the size of the tag as well as the recognition speed. Even though the Microsoft Surface is also able to recognize more complex identity tags (Figure 3.6), they are not appropriate for our purpose. They are too big to fit on a fingertip and the recognition is too slow for tracking moving objects. However, a fast hand part tracking is mandatory since we want to support gestures like a swipe.

Every tag has at least four IR-reflective circles. The bigger circle in the center locates the tag on the Microsoft Surface screen. The three circles located at the left, right, and bottom of the tag are the so-called guide circles. They determine the tag orientation. Additionally each tag contains from zero to eight data bits that define the tag value. The highest order bit (bit 7) is at the 1 o'clock (cf. Figure 3.7). Less significant bits are read clockwise in a descendent order. An IR-reflective circle at one of these positions encodes a 1 for the corresponding bit, the absence of a circle a 0. If there are no data bit circles, the encoded value is zero. Exemplary tag values can be seen in Figure 3.8

One disadvantage of byte tags is that they store only eight bits of data (one byte), so there are 256 unique tag values. Identity tags store 128 bits of data in two sets of 64 bits - one serving as

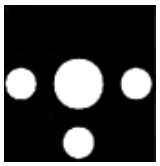

Figure 3.5: Byte tag.

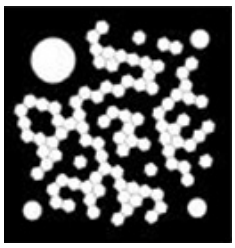

Figure 3.6: Identity tag.

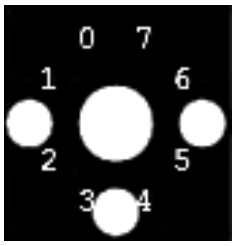

Figure 3.7: Data bit positions.

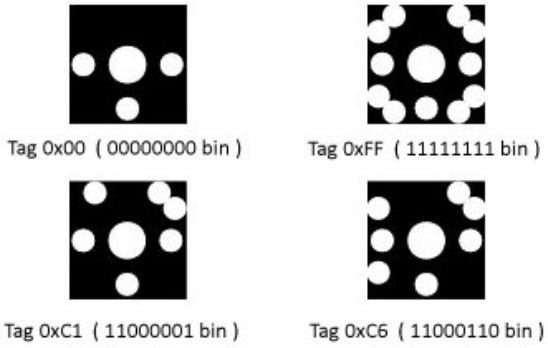

Figure 3.8: Tag examples.

series number, the other one as tag value. Accordingly, using this tag type would offer more tags and therefore more gloves. The limitation of 256 distinct tags restricts us to 17 Gloves at most. This amount of gloves is still enough for common collaboration tasks with a maximum of five participants. Even if each of them wears two gloves, we do not need more than ten gloves, so that more than 100 tags are left for other tangibles.

In order to be recognized properly, Microsoft recommends that the tags are perfectly flat. In the best case, they should be attached to a solid, flat object.

3.4 Gesture Recognizer

We incorporated the $1 recognizer by Wobbrock as basis for gesture recognition. It recognizes single stroke gestures only. This means that only gestures that consist of one continuous path can be loaded into and recognized by this engine.

In order to create a new gesture, a filename and a point array have to be passed to the save method of the engine. The filename has to include the path to the file. Such a point array can be sampled while a touch point traces a single stroke on the Microsoft Surface, for example. Each time the touch changes its position, its coordinates - together with a timestamp - are saved into the array. When the path for the gesture is finished, this array is normalized by the gesture recognition engine. Therefore it samples the point array down to 64 values. After that, the radian between the centroid point and the first point of the down-sampled array is calculated. Then, all points in the array are rotated by the calculated radian about their centroid. Following the engine scales the bounding rectangle of the new points to a standardized size. Afterwards, the points are translated to new positions so that they fit in the scaled bounding rectangle. The last step of the normalization is that all points are translated again so that their centroid lies in the origin point. The normalized point array is then saved to an XML file with the given filename.

Using the timestamps of the first and last entry in the point array, the gesture duration is calculated. This information together with the current date and time, as well as the number of points in the original array is stored as attributes of the root element of an XML file. Then the normalized array is run through. For each of its points a new child element 'Point' is created for the root element 'Gesture'. The coordinates and timestamp of the point were stored as attributes of the corresponding 'Point' element (cf. Code Example 3.1).

```
<!DOCTYPE GESTURE [

  <!ELEMENT GESTURE (POINT+)>
  <!ELEMENT POINT>

  <!ATTLIST GESTURE NAME CDATA #REQUIRED>
  <!ATTLIST GESTURE NUMPTS CDATA #REQUIRED>
  <!ATTLIST GESTURE MILLSECONDS CDATA #REQUIRED>
  <!ATTLIST GESTURE DATE CDATA #REQUIRED>
  <!ATTLIST GESTURE TIMEOFDAY CDATA #REQUIRED>
  <!ATTLIST POINT X CDATA #REQUIRED>
  <!ATTLIST POINT Y CDATA #REQUIRED>
  <!ATTLIST POINT T CDATA #REQUIRED>

]>
```

Code Example 3.1: Gesture file DTD.

Once at least one gesture has been saved, the existing gestures can be loaded into the recognition engine. This is done by passing a directory string to the method for loading gestures. The recognition engine then loads all gestures complying with the structure in Code Example 3.1. The loaded gestures are then ready to be recognized by the engine.

The first step for the gesture recognition is similar to saving a new gesture. The recognition method of the engine demands a point array as input parameter. It then samples it down and normalizes it in the same fashion as described for the save method. The normalized point array is then compared to the arrays of the loaded gestures. For every comparison a recognition confidence between 0 (no match) to 1 (perfect match) is determined. The closer the input point array resembled the gesture point array, the higher the recognition confidence. After the comparison the gesture name, recognition confidence as well as the distance and angle difference between the compared arrays are stored as an entry in a list. When all loaded gestures have been compared to the normalized input point array, this list is ordered by confidence. The ordered list is the return value of the recognition method. The first entry in the list is the best match and can be processed as recognition result.

The recognition accuracy with only one template sample for a gesture is 97 percent. Nevertheless, more samples can be created to improve accuracy and enable the recognition of mirrored gestures, e.g. two samples for a circle, one drawn clockwise, the other one counterclockwise. The limitations of the recognizer are given by its algorithms. They cause rotation, scale, and position invariance of defined gestures which facilitates point comparison and improves the variation tolerance. At the same time, the invariances make it impossible to define gestures with a certain position, size or orientation.

A C# library [1] allows the easy access of the described features of this recognition engine - saving new gestures, loading existing gestures and recognizing gestures.

4 The Fiduciary Glove

This chapter covers the concept of the Fiduciary Glove, two design iterations and the first application that was built for it before the Fiduciary Glove Toolkit existed. In the end the issues during the development of the application are discussed.

4.1 Building the First Fiduciary Glove

An ordinary glove is the basis for the Fiduciary Glove. The glove we used for our prototype cost less than ten dollar. The material it's made of is similar to suede and therefore almost completely IR-absorbent. Fiduciary byte tags (cf. Chapter 3.3) are stuck onto 15 key hand parts of this glove. These tags are recognizable by the Microsoft Surface and allow for a reliable hand part identification. Through their association with hand parts, gloves and users can be identified, too.

In the following I will explain which hand parts we picked and why. It is not surprising that the fingertips were chosen, being the standard hand part for touch input. A tag is attached to each fingertip with the tag orientation facing the same direction as the fingertip itself (cf. Figure 4.1).

Two parts on the hand's inside were chosen as key hand parts for the following reasons: The palm is necessary to distinguish between a flat hand and a situation where only all fingers are touching the surface. Additionally the palm is naturally used for smearing and similar gestures. As second important hand part on the inside of the hand we identified the heel of the hand, close to the wrist. This hand part would allow to perform grasps with the fingertips, for example. If the distance between the fingers and the hand wrist decreases, a grasp can be identified. Also, we use this part of the hand to push things away from us in real life. This could be used for communicating rejection, for example. Both tag orientations face in the same direction as the hand (cf. Figure 4.1).

Figure 4.1: Hand parts (front).

Also the backs of the fingers - the knuckles - were picked as hand parts. Each knuckle is a natural counterpart of the fingertip in the same finger. Hence, opposing functions can be mapped in a natural fashion to fingertip and knuckle. This concept seemed intruiging and led to the decision to include the knuckle to the key hand parts. A tag is stuck onto a knuckle in way that its orientation faces away from the hand (cf. Figure 4.2).

The center of the back of the hand was first of all picked to have a counterpart for the palm hand part. These two tags have a similar relation as fingertip and knuckle. The back of the hand is

also needed to differentiate between a flat hand that lies with its back on the surface and only all knuckles touching the surface.

Figure 4.2: Hand parts (back).

The last two tags are stuck to the outer side of the pinkie and the side of the hand (cf. Figure 4.3). These hand parts are important to recreate natural postures with the side on the hand, like a straight hand or a fist. The hand parts on the side of the hand make several interesting interaction techniques possible. Changing from the palm hand parts to the two hand parts on the side of the hand could be used as a opening gesture, for example.

Figure 4.3: Hand parts (side).

Because the Microsoft Surface only recognizes flat tags, we could not simply print the tag onto the glove fabric itself. For the first prototype, we glued the tags first on small squares of plastic and then on the glove for this reason.

4.2 Design Iterations

The original glove design with the tags outside left exposed edges which catch on things (cf. Figure 4.4). In the next design iteration we glued the squares of plastic to the inside of the glove,

where they lay underneath the part of the glove to be tagged. We then stuck the fiduciary onto the glove itself, atop these flat surfaces. The result was a reasonably comfortable and very lightweight glove where edges do not catch (cf. Figure 4.5).

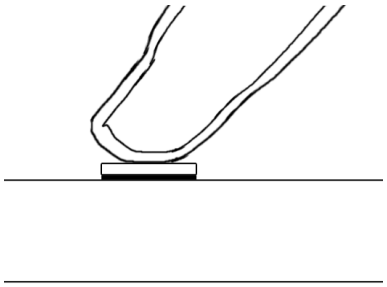

Figure 4.4: Prototype design with plastic outside.

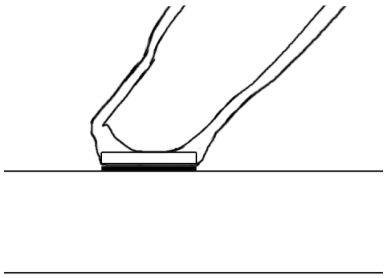

Figure 4.5: First design iteration with plastic inside.

For a new version of the glove we experimented with transparent tags that were sponsored by the Microsoft Surface product group (cf. Figure 4.6). For these kind of tags, the negative version of the tag pattern is printed with transparent IR-absorbent ink on an IR-reflective, transparent sheet. Transparent tags have an obvious advantage over opaque tags: You can see through them. Thus, they could be attached on top on human readable labels, for instance. Our first transparent glove has no labels beneath the transparent tags, because we wanted to explore, if transparent tags improved the impression that your touching the surface with hand parts and not tags.

Figure 4.6: Fiduciary Glove with transparent tags.

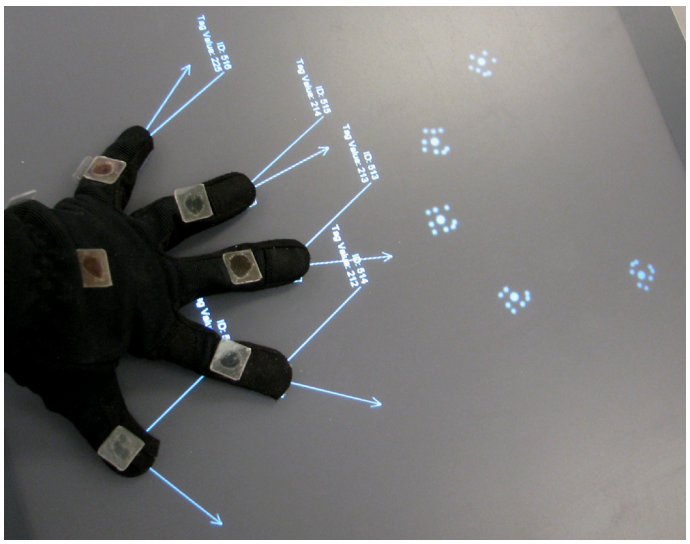

Figure 4.7: Raw image as seen by the Microsoft Surface.

While the Microsoft Surface still recognizes only the fiduciary markers, a human sees meaningful icons. Figure 4.7 shows a Surface application that displays a mirrored raw image the Microsoft Surface receives. Even though the tag pattern is invisible to the human eye, the device recognizes them just as clear as the standard tags. For the Fiduciary Glove with transparent tags we decided to glue transparent acrylic on the outside of the glove again and the tags on top, since it protects the tags better. This aspect was more important than having gloves where edges don't catch because Microsoft could provide us only with a limited set of transparent tags.

4.3 First Application for the Fiduciary Glove

The first application we implemented for the fiduciary glove was a painting application. We decided that this application domain would demonstrate the expressive interaction techniques the Fiduciary Glove affords very visually. Although the Surface SDK already provides rich event arguments each time a tag contact is processed (cf. Chapter 3.2), it became quickly clear that mapping the tag to its corresponding hand part manually on every event was too cumbersome. For this reason we wrote XML files containing this mapping. The mapping was done by associating the tag value of each of the 15 tags on the glove to a hand part type and a hand part position. Next to the mapping, these so-called glove configuration files contained a glove identifier and whether it is a right or left hand glove (cf. Code Example 4.1).

```
<Glove Hand="Left">
    <Tag id="47">
        <Position>Thumb</Position>
        <Type>Finger</Type>
    </Tag>
    <Tag id="25">
        <Position>Index</Position>
        <Type>Finger</Type>
    </Tag>
   [...]
</Glove>
```

Code Example 4.1: Glove configuration file.

Defining a glove in an external file seemed reasonable, since this would also allow that the same mapping was used in multiple applications. This design decision was supported by the fact that the tags on the gloves rarely changed. The glove configuration files were then read by a simple parser which created a glove object from the given information. A glove object holds all the information the corresponding glove configuration file provided and could optionally be associated with a user.

Whenever the Microsoft Surface recognized a touch and raised a contact related event we checked if the source of the touch was a byte tag; and if so, whether one of our glove objects contained a tag whose tag value was the same as the one that caused the event. A positive match meant that a hand part of one of our gloves touched the surface. Hand part type and position could now be retrieved and processed as needed (cf. Code Example 4.2). It's obvious that even with the mapping the access of hand part information still happened low-level. Nevertheless, the newly gained knowledge about which hand part touched the surface offered expressive interaction techniques, which are: separate functions for separate hand parts, the overloading of postures and gestures with hand part information, as well as the reliable user identification.

4.3.1 Separate Functions for Separate Hand Parts

Knowing both hand part type and position, it was an easy task to assign separate functions to separate hand parts. Each of the fingers could paint in its own color and thickness.

The thickness was automatically predetermined by the hand position. The thumb painted with the broadest stroke, the pinkie with the thinnest stroke. The other fingers had stroke thicknesses according to their position between thumb and pinkie.

The colors, in contrast, could be freely assigned to the fingers. In an early version of the painting application this happened through touching a color on a palette on a side of the screen (cf. Figure 4.8). A disadvantage of this solution was that only fixed color value could be picked.

Therefore we introduced a more flexible tool in a later version of the application. Colors were now assigned with a tagged tangible object - the color selection cube (cf. Figure 4.9). Depending

```
private void ContactDown(object sender, ContactEventArgs e){
  if (e.Contact.IsTagRecognized && e.Contact.Tag.Type == TagType.Byte↩
      ){
    byte currentContact = e.Contact.Tag.Byte.Value;
    string tagId = ByteToHexadecimalString(currentContact);
    Tag tag = null;
    foreach (Glove glove in this.gloves){
      if (glove.containsTag(tagId)){
        tag = glove.getTag(tagId);
        break;
      }
    }
    if (tag != null){
      string tagPosition = tag.getPosition();
      string tagType = tag.getTagType();
      if (tagType == "Finger"){
        // START PAINTING
      }
    }
  }
}
```

Code Example 4.2: Hand part access (without Fiduciary Glove Toolkit).

on which side of it was placed down on the surface, a ring control for picking hue, saturation or luminance was displayed at its position. The center of the ring was filled with the currently picked color. In the same color, an ellipse was drawn to the position of the nearest finger on the surface. A line from the ring center to this ellipse should emphasize the association between finger and control. A new color could be picked by rotating the cube. At the same time, a semitransparent white line rotated along the ring. The color beneath the line indicated the current selection. When the wished color was found, the cube had to be lifted from the surface to permanently assign the color to the finger. If the associated finger was lifted first, the color assignment was cancelled for this finger. Then, the color selection cube connected to the finger that was the second closest. By incorporating the HSL-color model through our cube practically any color could be mixed. On a higher level this meant that functions of separate hand parts - in this case the fingers - could be altered during runtime.

Also other hand parts than the fingers got their own functions. The backs of the fingers - the knuckles - could erase in the same thickness as the corresponding finger. Accordingly, the knuckle of the thumb was the thickest eraser.

4.3.2 Postures

In the pre-toolkit version of the painting application, hand postures were recognized by monitoring either the distance or angle between two or more particular tags. Postures are defined by a subset of glove tags within a certain range of distances and angle to each other. Consider two tags attached to the side of the hand (Figure 4.10). When the side of the hand is in a straight orientation, the angle between the two tags is close to 0 degree (Figure 4.10a). The angle changes to around 90 degree with an L-shaped posture, to 120 degree with a curved C-shaped posture, and around 180 degree with a fist (as shown in Figure 4.10 b-d). Thus the sole comparison of the relative orientation angle between these two particular tags at the side of the hand allows reliable identification of four different postures. In the painting application a fist posture could be used to clear the canvas as long as the person who placed down the fist on the surface was the sole artist of the painting.

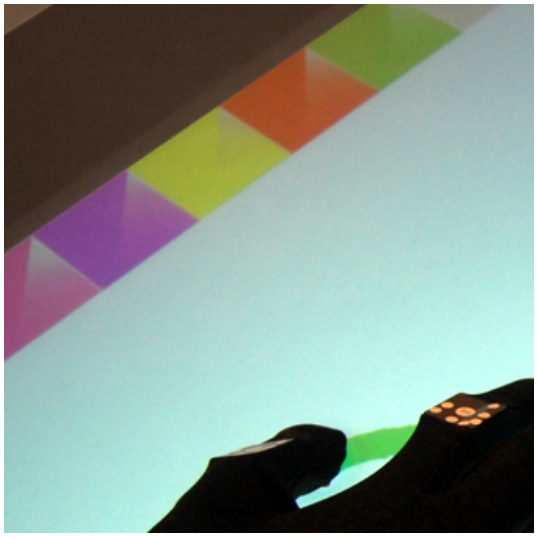

Figure 4.8: Color palette.

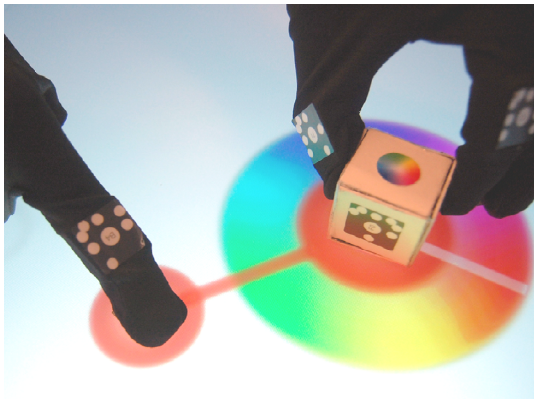

Figure 4.9: Color selection cube.

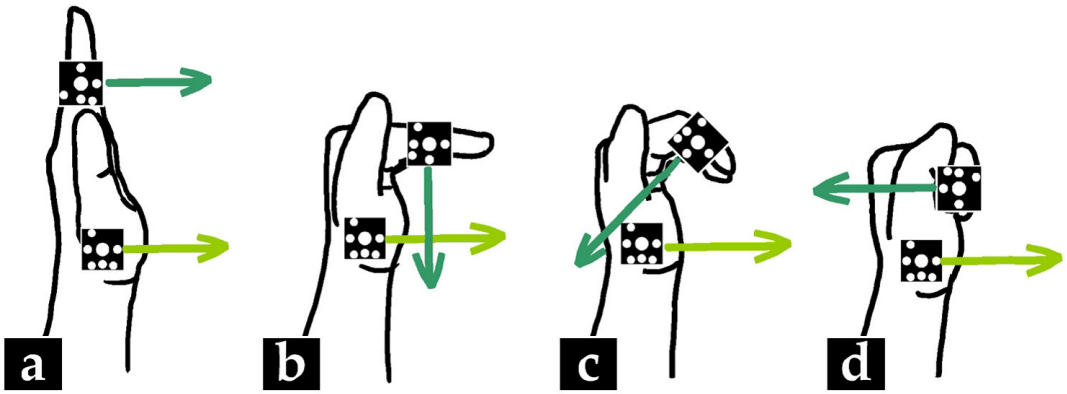

Figure 4.10: Straight hand, L-shape, C-shape, fist.

Our second approach interpreted the distance (instead of orientation) between specific tags to differentiate between hand postures. Consider, for example, tags attached to the fingertips, palm, and wrist of the glove around the user's hand. If the hand is pressed flat against the surface all of these tags are detected simultaneously. When spreading the fingers the distance between the fingers increase, and when bending the fingers - like when grabbing an object - the distance

between the fingers and the palm decrease and thus these postures can be discriminated. This posture type was not featured in the finger painting application however.

Transitions between postures could also be recognized. As a person changes postures while touching the surface, the changes in distance and angle between the tags are also continuous. Thus we could easily track the fluent transitions between these postures. This capability is also essential for gesture recognition.

By wearing a right and left glove a user could perform advanced postures with multiple gloves. If both gloves performed the L-shaped posture (cf. Figure 4.10b, Figure 4.11 right side) with the thumb and index finger, a rectangle was spanned between the center points of the two postures. The rectangle defined the area of the canvas that should be copied once the advanced posture for pasting had been performed. This posture is identical to the posture for copying, with the slight, but important difference that this time the L-shaped posture is performed with the thumb and pinkie finger. Since gloves could be programmatically assigned to a specific user, interference by touches of other users was not possible. The multi-glove posture for copying and pasting was only recognized and processed if performed by two gloves belonging to the same user.

4.3.3 Gestures

Gestures were recognized by tracking changes of the position of a hand part over time. A C# library by Wobbrock (cf. Chapter 3.4) provided the gesture recognition engine. The standard version of this library did not meet our requirements in the following point: it did not recognize which hand part or tag performed a gesture, of course. Therefore, gestures could not be restricted to hand parts.

To address this issue, the original recognition method was extended. It now accepted a hand part as second parameter beside the point array. The recognition method recognized gestures based on the given point array as usual, but added the hand part to each of the recognition results. This way the hand part was associated with the recognized gesture and we could perform different actions, when the same gesture was performed by different hand parts.

In order to be able to recognize gestures, the recognition engine needed predefined gestures. Thus, we had to create gesture files that could be loaded into the engine. Even though the library provided a method to save a point array as a gesture in XML file format, no application existed to create and save new gestures. In order to address this problem, we integrated a mode in our painting application in which you could do this. In this mode only one stroke at a time could be painted. When a new stroke was started, the last stroke was deleted. After pressing a save button that was only visible in the gesture definition mode, a standard file dialog was opened, in which you could choose the destination and name for the gesture file. After confirming the dialog, the last painted stroke and the specified filename were used as input for the save method of the gesture recognition engine. The gesture was saved to the file and loaded into the recognition engine right away.

4.3.4 Multi-User Scenarios

Multi-user scenarios were feasible, because each glove could be assigned to a specific user in code. Our application could reliably tell which tag mapped to which hand part, which hand part belonged to which glove and in the end by which user the glove was owned. This enabled true multi-user scenarios without interference. Two and more people could therefore work simultaneously on the same surface.

They could not only paint in their own colors, they could also perform gestures and postures independently from the others. Our application leveraged this for a clearing function. While a user could clear the canvas with a fist posture as long as he was the sole artist of the painting, in a multi-user scenario this action would impact other people. In this case a consensus was needed

(cf. multi-user coordination policies in [10]). Therefor all people involved in the painting had to place down their fists for clearing the canvas.

Even multi-glove postures could be performed without worrying about interference. If two people were accidently performing a multi-glove posture with their gloves, it was not mistakenly recognized but ignored. Our application knew exactly, that the two gloves forming the posture were owned by different people. In Figure 4.11 you can see one user painting with his glove in the upper left area of the screen, while another user is performing the copying posture with both gloves at the right side of the screen.

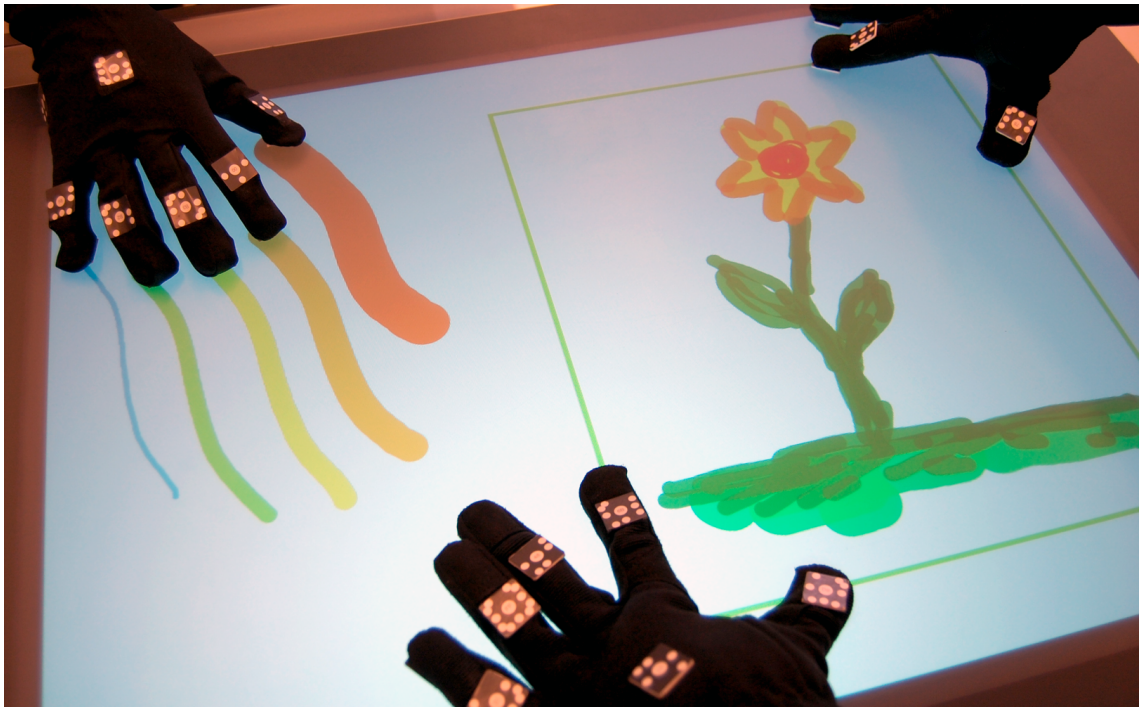

Figure 4.11: Multiple gloves.

4.3.5 Issues

Even though it was possible to design and implement an application for the Fiduciary Glove, it was a cumbersome and time consuming job.

Low-level Details

The main problem during the development of the painting application was that many things had to be dealt with low-level.

This started already at the very beginning when mapping a tag to a hand part. If we had not created the glove configuration files as aforementioned, functions would have been assigned to tags and not hand parts. This would have meant that the painting function had to be assigned to each of the tags on the fingers separately. Even with our tag to hand part mapping through the glove configuration files, the whole process was still tedious. While creating the first glove configuration files manually, we realized that it was strenuous and error prone. If one of the mapped tag values was wrong, it led to confusing behavior, of course. For instance, if the tag value of a hand part existed in another glove configuration file, the hand part could be misinterpreted. This way, a finger could act as a knuckle, and would erase instead of paint.

Very low-level was also the hand part identification. Even though we already introduced a higher abstraction level through the glove configuration files, retrieving hand part information from

contact event arguments was still required many low-level steps. First, our application checked if the contact was a tag and then its tag type. If it was a byte tag, the tags byte value was queried. This value was then translated to its string representation, e.g. 0x255 to "FF". This was necessary since the glove configuration files were designed in a way so that human could manually create them. The hexadecimal string of the tag byte value was printed on each tag and therefore easy to copy as tag identifier in the glove configuration files. Next, the application checked if one of the gloves contained a tag with the determined tag identifier. If that was the case, the hand part position and type could be retrieved.

The posture recognition happened on a very low level, too. The angles and distances between touching tags of the same glove or user had to be continuously calculated and analyzed. If the values fell in a certain range, the particular posture was recognized. Since no posture recognition engine or definition language for postures existed, the postures were hard-coded into the application.

As already mentioned above, we used the $1 recognizer as recognition engine. In order to integrate it into our application we had to learn and understand how it worked and where we had to make adjustments so it would fit. The recognizer accepts a point array, samples the array down to a certain number of points and matches it with the point arrays of predefined gestures. This array had to be filled for each tag on every event. In order to be able to tell which tag and therefore hand part caused a gesture we had to customize the recognizer at this point. We changed the recognition method so that it would also accept a hand part next to the point array. This hand part was then returned together with the gesture recognition result. Even though we were now able to tell which hand part performed which gesture, we had to deal with it low-level.

4.3.6 Complex Code

A side effect of the mass of low-level details in the application code was that it got very complex very soon. It was hard to keep an overview of it. When handling a touch event, often more than 20 if cases were needed to sift through all possible states (cf. hand part identification in section 'Low-level Details'). A lot of these if-statements reoccurred in other events that were raised when a touch changed its position or orientation, or when a touch disappeared. Duplicate code made the navigation through the code wearisome. These duplicates also lead to code that was hard to maintain. If duplicate code had to be changed, the change had to happen at multiple code locations.

Another source of unnecessary complexity was the fact that a specific hand part was defined by two separate values: Hand part position and hand part type. Because of this division two values had to be checked to identify a certain hand part. A single hand part identifier would require only one check and lead to cleaner and more readable code.

All code was in the same namespace except for the gesture recognizer. Distinct functionalities were not properly separated. The resulting complexity in the main namespace made it hard to find a feature quickly.

All these issues lead to this decision: If we wanted to explore the space of the expressive interaction the Fiduciary Glove affords, we had to design an environment that improves development efficiency. A rapid prototyping toolkit would enable us (and other developers) to iterate over ideas and adjust applications on the run. This toolkit is described in the following chapter - the Fiduciary Glove Toolkit.

5 The Fiduciary Glove Toolkit

In this chapter the core of this thesis is introduced - the Fiduciary Glove Toolkit. The chapter includes a set of general toolkit qualities and concrete guidelines for toolkit design, largely based on the work by Cwalina and Abrams [14] and consolidated with concepts from other related work. Subsequently, a quick walkthrough gives an impression on how to use the Fiduciary Glove Toolkit presented. Then, the utilities that are included in the toolkit are described. The next section elaborates on the Fiduciary Glove API. After this, the toolkit documentation is illustrated. The chapter ends with a description of its integration into the development environment.

Before qualities and guidelines for toolkits are described, the MSN encarta entry [4] shall give an impression of how the term *toolkit* is originally understood:

> **toolkit** / ˈtuːlkɪt/ n. 1 a set of tools, especially for a specific type of work, kept in a special box or bag. 2 a collection of information, resources, and advice for a specific subject area or activity

Figure 5.1: Dictionary entry for toolkit.

Therefore, the Fiduciary Glove Toolkit qualifies as a toolkit even in its classical meaning: It offers an Application Programming Interface that enables the access of information about a specific subject area, namely information about hand parts, hands, users and gestures and postures. Secondly, it includes a set of tools for a specific type of work: the creation and testing of glove configurations, gestures and postures.

5.1 Toolkit Qualities

In this section general qualities of a toolkit are listed. They are derived from related work presented in Chapter 2.

5.1.1 Simplicity

Simplicity is a key quality of a well-designed toolkit. It demands basic features that can even be used by beginners. Keeping it simple also means the functionality of a toolkit should focus on one thing; it should be minimal [24]. If an toolkit gets too complex it is harder to learn, to use and to read (cf. 'API Should Do One Thing and Do it Well' in [10]). If you are not sure about a feature, leave it out. It will help to reduce complexity. You can always add it later, but never remove it. People will use the feature and if you removed it, their programs will crash. Providing plain overloads of constructors and methods with no or few parameters simplifies the access of the API. Van Gurp and Bosch state that components should be small and simple [22]. Large components have a lot of features but the reuseability is limited. Small components on the other hand can be used in various situations; due to the low complexity they are also easier to understand.

5.1.2 Accessiblity

Well-designed toolkits are accessible and easy to learn [10]. In order to cater to a broad range of developers with different skill sets, requirements and programming styles, you have to identify and understand the potential users of your toolkit. Personas can help you with this. The Visual Studio usability group around Steven Clarke identified three personas to characterize different developer groups: The opportunistic, the pragmatic, and the systematic developers, as mentioned in Chapter

2.4. Opportunistic programmers appreciate convenience features that raise productivity. Having full control over their code is only secondary to them, since the cost for this is higher development effort. Hence, predefined controls and black-box components are very important to this type. A lack of these would keep them from using a toolkit. Pragmatic programmers in contrast tend to look for a balanced control-productivity-ratio. They value productivity features. At the same time they accept a longer development time when certain situations call for custom solutions. If control is mandatory they deal with low level problems. Systematic programmers constitute the most defensive type of programmer. They do not make assumptions about the functionality of a library, but test it first. They desire full control, even at the cost of productivity or ease of coding. They explore code so they understand how everything works and fits together. Also, they like to get into detail and adjust or replace components. In order to ease the access for each of these personas it is important to design the API in a way so that it has a low threshold and a high ceiling as described by Myers et al. [40]. Low threshold, or low barrier to entry [14], means that the basic functionality of the API is very easy to use, which is closely realted to aspects of the toolkit quality 'Simplicity'. Having quick successes with these features will motivate programmers to use the toolkit. Consequently, the initialization has to be as simple as possible since this is the first step the developer has to take. Do not require users to conduct an extensive initialization and instantiation of several types that need to be linked together before they are able to run their program for the first time. Cwalina also mentions convenience features as crucial elements to lower the barrier of entry. Providing good defaults for properties and parameters can lower the workload for basic scenarios. A high ceiling in contrast means that complicated, more advanced scenarios can be implemented. Extensibility and customization capabilities play important roles here. Developers should be able to extend the functionality of the API by creating custom components, for example.

5.1.3 Linear Learning Curve

In order to pitch the more sophisticated features of your API to designers at the beginner or intermediate level, it's mandatory that the API has a gradual learning curve as depicted in Figure 5.2. If the learning step to a more complex feature is not big, the motivation and ease of trying it out is far higher than otherwise. Also, the learning by doing effect is much stronger if the learning curve is gradual, because the process of programming and learning is a continuous flow.

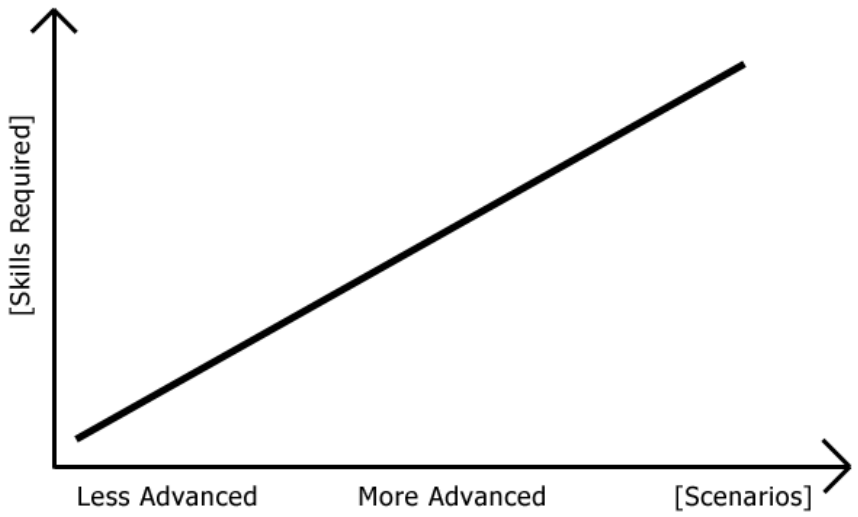

Figure 5.2: Linear learning curve [14].

5.1.4 Completeness

Although a well-designed toolkit should be simple and minimal, it should at the same time offer all functionality required by the targeted developer. Michi Henning states in 'API Design Matters' [24] that an API must provide sufficient functionality for programmers to achieve their goals. Additionally, if the API can implement a functionality that the user needs, it should not be left for the user to implement [56]. If it turns out that a function is constantly demanded by developers it should be integrated, which leads over to the next quality: Iterative Design.

5.1.5 Iterative Design

Bloch [10], Gould [20] and Greenberg [21] argue that an iterative design is a prerequisite for a successful toolkit. Developing the toolkit and first applications concurrently helps to identify issues early and prevents 'nasty surprises' [10]. On the one hand new features of the toolkit can be tested right away in the applications and evaluated. On the other hand useful ideas from the applications can be built into the toolkit and are then accessible for other applications. Since the usability of the toolkit is constantly evaluated while it is developed, it is very easy to react to design flaws and make adjustments.

5.1.6 Integrated Design

A good toolkit follows an integrated design. When talking about integrated design Gould [20] means that all aspects of usability evolve in parallel and under one person. This person coordinates the usability throughout all aspects like system performance, system functions and installation. Integrated design is similary understood by Cwalina et al. [14] who state that the toolkit should seamlessly fit into the programming ecosystem. The toolkit should fit into other development tools, programming languages and application models. IDE integration, tool support, and installer are mandatory parts for this integration.

5.1.7 Performance and Robustness

As every usable system, a good toolkit is performant and robust [49].

5.1.8 Proven Design Basis

Good toolkits have a proven design basis. They build on existing concepts. Novel concepts should only be introduced with the highest caution. Chris Sells states in [14]: 'Please don't innovate in library design. Make the API to your library as boring as possible. You want the functionality to be interesting, not the API.'

5.1.9 Evolvability

A good toolkit is designed so that it evolves well. When facing design tradeoffs, you should consider their impact on the evolvability of the toolkit.

5.1.10 Error-preventive Design

The design of a good toolkit prevents errors and is hard to misuse [10]; error-prevention is part of the usability of a toolkit [49].

5.1.11 Consistency

Consistency is a recurrent theme in toolkit design papers. Productivity and usability of a toolkit are highly dependent on this factor. Inconsistencies hamper transfer of knowledge. When a method behaves differently in similar situations it confuses the programmer, who will then have to find the reason for this [cf. 'Principle of Least Astonishment' in [10]). Inconsistencies also occur among multiple toolkits and APIs. Following common conventions helps to transfer knowledge from one toolkit or API to another.

5.2 Toolkit Design Guidelines

The guidelines and strategies described in the following served as foundation for the Fiduciary Glove Toolkit. They are derived from related work introduced in Chapters 2.3 and 2.4.

5.2.1 Naming Guidelines

Naming is an important factor for an API with self revealing functionality. The guidelines in this chapter should be followed strictly to leverage this opportunity.

Capitalization Conventions

PascalCasing is used for all identifiers except for parameters. PascalCasing capitalizes the first letter of each word in the name. If the identifier consists of a single word, it has just an initial capital. Parameter names are written in camelCasing. This means that the first character is lower-case for the first word and upper-case for the following words in the name. Acronyms are exceptionally handled. If they consist only of two letters, both are capitalized. If they consist of more letters, only the first is written in upper-case. Acronyms in the beginning of camel-cased identifiers are not capitalized at all. An overview of the identifier casing can be seen in Figure 5.3.

Identifier	Casing	Example
Namespace	Pascal	`namespace FiduciaryGlove {…}`
Type	Pascal	`public class Glove {…}`
Interface	Pascal	`public interface IGestureRecognizer {…}`

Method	Pascal	`public class Glove`
		`{`
		`  private void SetupHandparts();`
		`}`

Property	Pascal	`public class Glove`
		`{`
		`  public string Identifier;`
		`}`

Event	Pascal	`public class GloveWindow`
		`{`
		`  public event EventHandler HandpartDown;`
		`}`

Field	Pascal	`public class GloveWindow`
		`{`
		`  public static readonly string GloveRepository =`
		`  RepositoryRoot + "Gloves";`
		`}`

Enum value	Pascal	`public enum Identifiers { Thumb, … }`

Parameter	Camel	`public class GloveWindow`
		`{`
		`  public void AddGlove(Glove glove) {…}`
		`}`

Figure 5.3: Identifier casing [14].

General Naming Conventions

Identifier names should be readable. For instance, RecognizedPosture reads easier than PostureRecognized. For this reason, underscores, hyphens, other special characters and Hungarian notation are to avoid. Also, readability should be favored over brevity. Comprehensibility is the most important aspect when choosing a public name in an API. Abbreviations as well as acronyms must not be used since you cannot be sure that these are well-known or still understandable in the future. Semantic names should be preferred over language specific keywords, e.g. GetLength is better than GetInt.

Names of Assemblies and DLLs

Names for an assembly DLLs should follow the pattern (cf. Code Example 5.1):

```
<Company>.<Component>.dll
```

Code Example 5.1: Assembly DLL naming scheme [14].

Names of Namespaces

Names of namespaces are specified by the following template (cf. Code Example 5.2):

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

Code Example 5.2: Namespace naming scheme [14].

The company prefix will result in unique namespace names. Namespaces from other companies will most certainly not have namespaces with the same name. All components of a namespace name are pascal-cased. No type in a namespace should have the same name as the namespace, because types have to be fully qualified for many compilers.

Names of Classes and Interfaces

Classes should be named with nouns. The names have to be pascal-cased and without a prefix like "C". Derived classes can end with the name of the base class. This implicitly illustrates the relationship between the two. Interfaces should normally be named with adjectives and have a prefix 'I' that highlights it as an interface. When a class exists that constitutes a standard implementation of the interface, its name should be the same as the interface name, yet without the 'I' prefix.

If a type is derived from a type in the .Net Framework, the following guidelines have to be followed: In case the base type is `System.Attribute`, add the suffix 'Attribute'. If you derived from `System.Delegate`, add the suffix 'EventHandler' to names of delegates that are used in events. If the derived delegate is not used as event handler, then add the suffix 'Callback'. On no account add the suffix 'Delegate'. If the type derived from `System.EventArgs`, add the suffix 'EventArgs'. Analogue for a derivation from `System.Exception` and `System.IO.Stream`: add the suffix 'Exception' and 'Stream', respectively. If a type implements `IDictionary` add the suffix 'Dictionary', if it implements `IEnumerable`, `ICollection` or `IList` then add the suffix 'Collection'.

Names of Type Members

Method names should be verbs or verb phrases, because they represent actions. Property names are nouns or adjectives, and should not match the name of 'Get' methods. For instance, if a method 'GetGloveModel' exists, there should not be a property with the name 'GloveModel'.

Collection properties should rather have a plural name than a name ending with 'Collection' or 'List'. Boolean properties have to be named with an affirmative phrase, e.g. 'IsActive' instead of 'IsntActive' or 'IsInactive'. Just like methods, events should be named with a verb or verb phrase. Event handlers have to end with 'EventHandler'. The two parameters in event handlers are always named with 'sender' and 'e'. Field names are pascal-cased and consist of a noun or adjective. Refrain from adding a prefix to field names, e.g. 's_' for a static field.

Naming parameters

Parameters are always camel-cased. They should be as descriptive and meaningful as possible, so the caller knows exactly what to insert. For this reason, do not use abbreviations and numeric indices.

5.2.2 Type Design Guidelines

Types and Namespaces

Namespaces should organize types in functionality groups. Too many namespaces and deep hierarchies should be avoided. Types for basic scenarios and those for complex tasks should be in different namespaces.

Choosing between Class and Interface

Classes are preferable over interfaces, since class-based APIs evolve easier than interface-based APIs. The reason behind this is that members can be added to classes without problems. Adding a member to an interface breaks existing code, however. Use abstract classes instead of interfaces for the decoupling of implementation and contract. Abstract classes can be extended and implement common functionality for subclasses. Classes cannot inherit from multiple classes but they can implement multiple interfaces. In this case, use interfaces when you target multiple inheritance.

Abstract Class Design

For abstract classes, a protected constructor should be added. It allows the base class to do its own initialization when a subtype is created. Also, at least one concrete type that inherits from the abstract class should be provided. The implementation unveils unnoticed design flaws in the design of the abstract class, and serves as code example for the usage of the abstract class.

Interface Design

For the same reason you should provide an inheriting type for an abstract class, you should also provide at least one implementation of the interface. It supports the validation and understanding of the interface.

Enumeration Design

Enumerations should be used to strongly type properties, parameters and return values that draw their value from a set of values. Replace associated static constants with an enumeration. Enumerations should not model open sets. Never include reserved entries in an enumeration. New values can always be added later, reservation slots are not necessary. Also, enumerations should contain more than one value. In any case, provide a value of zero which is assigned to the overlying default value.

5.2.3 Member Design Guidelines

Property Design

Get-Only properties should be used, if you want to prevent the caller from changing the property value. Set-Only properties should never be provided. In this case, create a method instead. A property should be preferred over a method when its purpose it to simply access a value. Methods should be used when the operation is a conversion when it has conceivable side effects or when the operation returns either a different value on each call or an array. Provide a good default value for each property.

Constructor Design

A simple default constructor with no or only few parameters has to be provided. If parameters are needed, they have to be primitives or enumeration values. A default constructor increases the usability of an API, because developers can choose what properties they set and in which order these are set. Nevertheless, provide other constructors with parameters that serve as shortcuts for setting properties. In this case, use the same name for the parameter as for the property. The only difference should be the casing, of course. A constructor should not only do much more than instantiation and setting properties. The processing of constructors has to be efficient.

Method Design

Developers should consider placing methods in classes where they are easy to find, even if this means that they are conceptually in the wrong place.

Event Design

`System.EventHandler<T>` should be favored over creating a new delegate as event handler. Create a class that inherits from `EventArgs` as event arguments. Using `EventArgs` directly should be avoided since you lose the capability to extend the event arguments. With custom event arguments you can later add data if needed. To raise events, a protected virtual method has to be created. This enables subclasses to handle the corresponding event with an override. The method name is the event name with 'On' as prefix. It demands one parameter 'e' that is typed as the event argument class (cf. Code Example 5.3).

```
public event EventHandler<GloveTouchEventArgs> HandpartDown;

protected virtual void OnHandpartDown(GloveTouchEventArgs e){
  EventHandler<GloveTouchEventArgs> handler = HandpartDown;
  if (handler != null){
    handler(this, e);
  }
}
```

Code Example 5.3: Event handler.

Field Design

Instance fields should not be public or protected but private. Properties can provide access to these fields. Constant fields are used for constants that never change.

Parameter Design

A parameter should have the least derived type that still provides the needed functionality. Never use reserved parameters. Methods can be extended with overloads if more parameters are required in the future. When overriding a method, name the same parameter identical. Favor enumerations over Boolean values where it improves readability. Especially, if two or more parameters are Boolean, use enumerations instead. They make the purpose of each value much clearer. If you do not know for sure that there will never be more than two values, use enumerations. Booleans can be used for two-state values. If an enumeration is used as parameter, insert a check for validity. It is possible to pass an integer even if there is no high-level value for it in the enumeration. Out and ref parameters should be avoided. Users of the API need to be knowledgeable about pointers and have to understand how value and reference types differ. The number of parameters should not exceed three. Also, the order of parameters should stay consistent in member overloads [10]. The same parameter should also be in the same position in all overloads (cf. Code Example 5.4).

```
public class Example{
  public Example();
  public Example(string name);
  public Example(string name, int count);
}
```

Code Example 5.4: Member overload parameters.

Never use overloads that have different semantics, but similarly typed parameters at the same position. If this is the case, split the overloads into two methods with different names.

5.2.4 Designing for Extensibility

Unsealed classes provide inexpensive extensibility because they allow for inheritance. Protected members can be used when advanced customization should be provided. They provide extensibility through inheritance without complicating the public interface. Protected members in unsealed classes should be treated as public for documentation, security and compatibility analysis, since any developer can inherit from the unsealed class and access the protected member.

5.2.5 Exceptions

Exception throwing

Exceptions should be used for failure reporting, and for this purpose only. Vice versa, only exceptions should be used to report failure. Do not use return values for this, since they can be ignored while exceptions cannot. Do document exceptions that are thrown in public members because of contract violations.

Choosing the Right Type of Exception to Throw

For usage errors, existing exceptions should be thrown. These errors have to be communicated to the API user and should not be handled programmatically. For program errors, create a custom exception if the errors are handled differently than in any other existing exception. Otherwise, use an existing exception. In any case, catch the exception and handle it programmatically. If an exception has to be thrown, chose the most specific exception that is appropriate. The error messages have to be grammatically correct, each sentence has to end with a period, question marks and exclamation points are to avoid. Then, the messages can be presented to end users without extra processing. Catching general exceptions swallows errors and important information is lost. Catch specific exceptions instead.

Using Standard Exception Types

The following rules for standard exception types should not be infringed:

- Never throw or catch `System.Exception` or `System.SystemException`.
- Also, refrain from throwing or deriving from `System.ApplicationException`.
- `InvalidOperationException` is thrown when an operation is not appropriate in the current object state.
- `ArgumentException` is thrown when bad arguments are passed to a member and none of its subtypes - `ArgumentNullException` and `ArgumentOutOfRangeException` - is applicable.
- When null is passed as argument and not valid, an `ArgumentNullException` is thrown.
- `ArgumentOutOfRangeException` is thrown when the passed argument is used for a collection and does not fall into the range of the collection.
- `NullReferenceException`, `IndexOutOfRangeException` and `AccessViolationException` should never be thrown by APIs; they are reserved for the execution engine to communicate a bug.
- Throwing `StackOverflowExceptions` is prohibited. It should only be thrown by the runtime environment.
- Never catch `StackOverflowExceptions`, since stack overflows have an immense impact on robustness of an application and cause major problems.
- `OutOfMemoryExceptions` are to be thrown only by the runtime environment.

Designing Custom Exceptions

When designing custom exceptions, always derive from `System.Exception` or another base exception. The name of a custom exception has to end with 'Exception'.

5.2.6 Usage Guidelines

Arrays

Arrays should be replaced with collections where possible. Never set array fields as read-only, because elements of the array can be changed nevertheless. Only for byte values, arrays are preferable over collections.

Collections

Use only strongly typed collections. Avoid `ArrayList` or `List<T>` in public APIs, because they do not offer notifications like Collection<T>. Also, refrain from using Hashtable or `Dictionary<TKey, TValue>` in public APIs. Their usage is acceptable for internal implementations. For public APIs, use IDictionary or a custom implementation of it. Use neither `IEnumerator<T>`, `IEnumerator`, or any implementation of these two types. Never implement IEnumrable<T> and `IEnumerator<T>` on an identical type. This means that a type should either be a collection or an enumerator, never both. If a collection is used as parameter, use the most unspecific type of collection, mostly IEnumerable<T>. If a property is a collection, do not provide a setter. If a completely new collection is needed, let the user clear the collection and

add the new items manually. If a collection should be read-only, use `ReadOnlyCollection<T>` instead of `Collection<T>`. You can add 'ReadOnly' as prefix to highlight this special relation. For future extensibility, consider to use a custom subclass of a generic base collection. This way you can add helper methods later. When writing a custom collection implement `IEnumerable<T>` or `ICollection<T>` were applicable. Add an appropriate suffix to the custom collection: 'Dictionary' for an `IDictionary<TKey, TValue>` implementation and 'Collection' for an `IEnumerable<T>` implementation. If the data structure of the implementation is a special case, like a stack or a linked list, name it after the structure. Never return null values from methods or properties that normally return a collection. Always return an empty collection instead. Code using the returned value can handle an empty collection in the same way as a normal collection. If the return value is null, you need another mechanism - a check for null.

Uri

Always use `System.Uri` if you need to model a URI or URL. Additionally, you can add overloads that take a string instead of an `Uri` for the most commonly used methods. Do not overload each and every method, since a `Uri` is still better than a string. If you receive a string that is meant to be an URI, store it as `Uri`. On the contrary, URIs should never be stored as strings.

System.Xml Usage

`XMLReader`, `XMLWriter` or `IXPathNavigable` should be used to represent XML data, do not use `XMLNode` or `XMLDocument` for this purpose. For input and output parameters use `XMLReader`, a subtype of `XMLNode` or `IXPathNavigable`.

5.2.7 Comments

Comments should only be used when describing something not obvious. Single-line comments are to favor over multi-line comments. Comments should only be placed at the end of a line if they are very short. Normally, they should be in the line above the section that is commented on.

5.2.8 Common Design Patterns

Factories

Normally constructors are to favor over factories because they are simplier and more convenient in their usage. Factories on the other hand have turned out be harder to understand and therefore harder to use (e.g. [32] and [18]). Developers expect an API to follow the Create-Set-Call pattern which demands a default or simple constructor for instantiating a type as described in the previous guidelines.

Factories can be a better choice after all, if you need a lot of control during the creation of instances. Factories should also be used for conversions (cf. Code Example 5.5).

```
DateTime.d = DateTime.Parse("10/10/1999");
```

Code Example 5.5: Factory usage [14].

Factory operations should be implemented as methods rather than properties. Also, created instances are returned as method return values and not out parameters. If no better name is available, name a factory after the name of the type that is created with an added 'Create' prefix.

Component-orientated Design

For APIs, a component-oriented design is desirable and expected by most developers. The difference to object-oriented design is that it focuses on interchangeable code modules rather than the relationships between objects. These modules work independently and developers do not need to know about the internal mechanisms. Such an API is exposed as types with properties, methods and events. Therefore, its usage follows the Create-Set-Call pattern: First, default or simple constructors initialize objects, then the necessary properties are set and in the end, methods are called. An advantage of this pattern is that the order in which the properties are set can be freely chosen by the user of the API. This allows for intermediary validation, for example.

Dependency Properties

Dependency properties store their value in a property store instead of a field. They should be used if WPF features like data binding or animations are needed for a type. When a type implements a dependency property it should inherit from `DependencyObject`. Types that inherit from `DependencyObject` provide an efficient property store. For each dependency property, a regular property should be added that accesses the dependency property (cf. Figure 5.6). The name of the `DependencyProperty` should be the same as the name of the regular property but with the suffix 'Property'. Do not set a default value for a dependeny property in code, since this overrides the value that was set in the Visual Designer. Neither validation checks nor change notifications should be implemented in the corresponding regular property of a dependency property. Dependency properties offer mechanisms for both cases. Validation is done via passing a validation callback to the `Register` method of the dependeny property. Change notifications are a built-in feature of dependency properties. They are used by supplying a change notification callback in the `FrameworkPropertyData` (cf. Figure 5.6).

```
//Property for accessing the DependencyProperty
public Glove RightGlove{
  get{
    return (Glove)this.GetValue(RightGloveProperty);
  }
  set{
    this.SetValue(RightGloveProperty, value);
  }
}

//DependencyProperty
public static readonly DependencyProperty RightGloveProperty = ↩
    DependencyProperty.Register("RightGlove", typeof(Glove), typeof(↩
    User), new FrameworkPropertyMetadata(null, new ↩
    PropertyChangedCallback(OnRightGloveChanged)));

//Change notification callback
private static void OnRightGloveChanged(DependencyObject obj, ↩
    DependencyPropertyChangedEventArgs args){
  User user = (User)obj;
  Glove glove = (Glove)args.NewValue;
  if (glove.User != user){
    glove.User = user;
  }
}
```

Code Example 5.6: Dependency property.

5.3 Walkthrough

Before the utilities and API of the Fiduciary Glove Toolkit are described, a short walkthrough will guide you through an exemplary course of developing an application with the toolkit. In this example the background color changes depending on which hand part or posture is placed down on the surface.

5.3.1 Glove Configuration

The first step when programming an application with the Fiduciary Glove Toolkit is to provide a glove configuration so that the API knows which tag belongs to which hand part. The toolkit comes with a utility that simplifies the creation of such a glove configuration dramatically - the Glove Configurator. This utility shows a hand from three perspectives. Tags are associated with hand parts by placing the tag on the corresponding hand part indicator on screen as you can see in Figure 5.4).

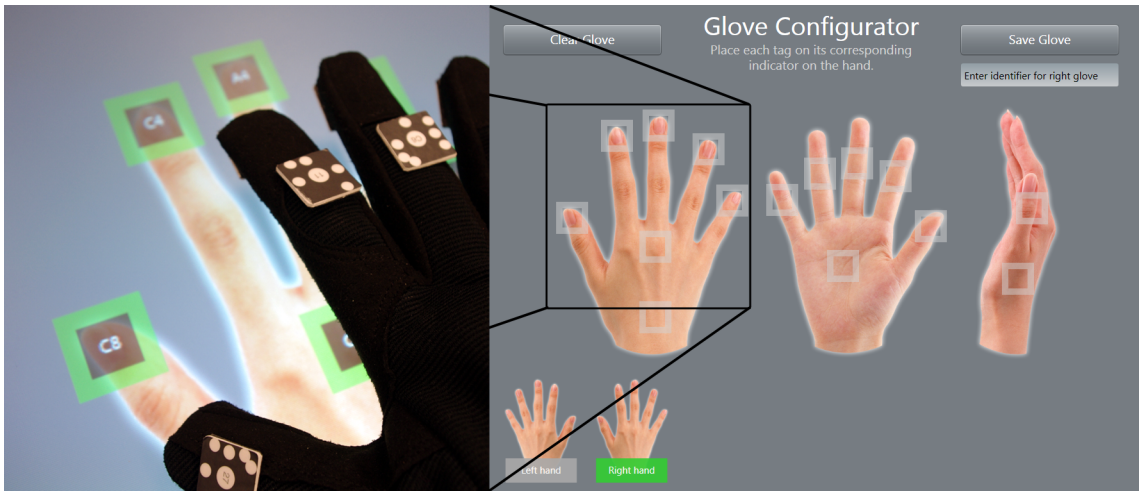

Figure 5.4: Mapping tags to hand parts.

When all hand parts are associated with tags, the glove configuration can be named and saved to a file. This glove configuration is then loaded by the API, and the hand parts of the corresponding glove are recognized.

5.3.2 Posture Creation

The second step is to create a set of postures. The Fiduciary Glove Toolkit also features a utility for this purpose: The Posture Configurator. In order to configure a new posture simply place down the posture on the surface. Connections are drawn between the hand parts that form the posture to give a visual feedback of how the posture will be saved. After adjusting and naming the posture, it can be saved. For our application, let us assume that a fist and a straight hand posture have been created with this tool (cf. Figures 5.5 and 5.6).

5.3.3 Application Setup

Now that glove configurations and postures have been defined, a new project has to be created in Visual Studio. The Fiduciary Glove Toolkit includes a project template (cf. Figure 5.7 which relieves the developer of having to start from scratch.

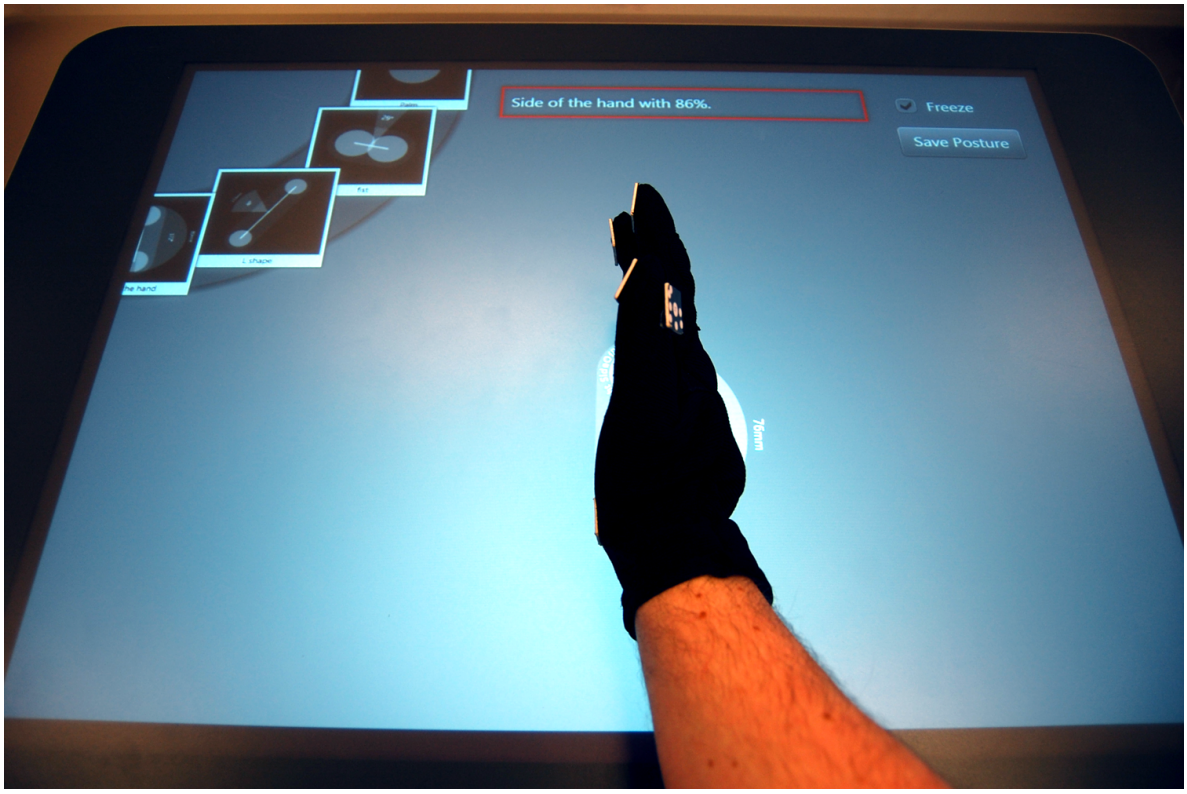

Figure 5.5: Straight hand posture.

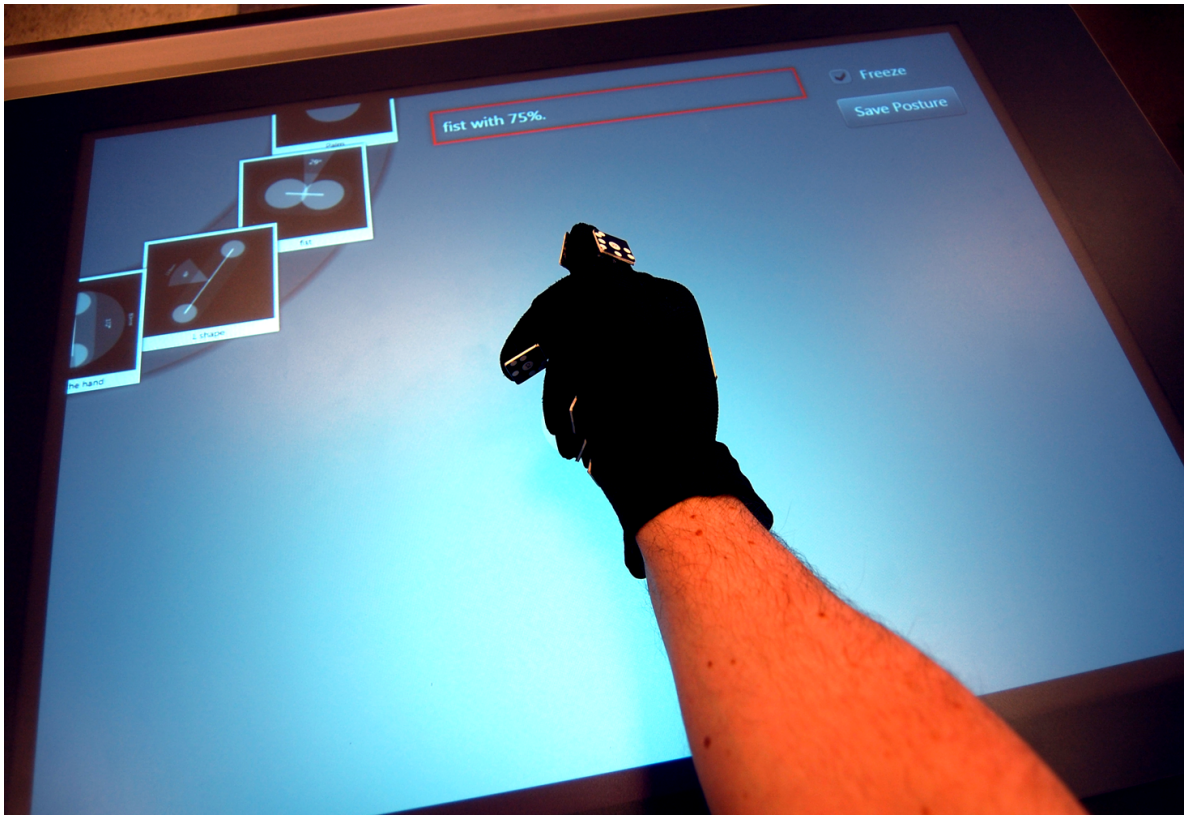

Figure 5.6: Fist posture.

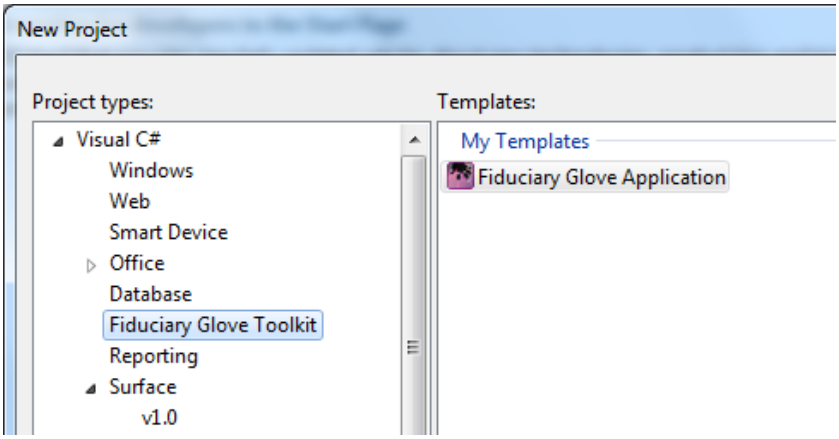

Figure 5.7: Fiduciary Glove Application template.

The template sets up a basic code framework from which the developer can start from (cf. Code Example 5.7). Comments explain the purpose of each line. We comment out the lines for the gesture recognizer, since we do not need it for our example. The template also includes all references to required libraries.

```
public partial class Window1 : GloveWindow{
  public Window1(){
    InitializeComponent();

    //Creates a glove for every glove model in the glove repository
    this.LoadGloves();

    //Alternatively you can also instantiate glove by glove;
    //Glove glove = new Glove(this, "Glove1");

    //Creates posture recognizer and loads all postures from the ↩
       posture repository into the recognition engine
    this.PostureRecognizer = new PostureRecognizer();
    this.PostureRecognizer.LoadPostures(PostureRepository);

    //Creates gesture recognizer and loads all gesture from the ↩
       gesture repository into the recognition engine
    this.GestureRecognizer = new GestureRecognizer();
    this.GestureRecognizer.LoadGestures(GestureRepository);

    this.HandpartDown += new EventHandler<GloveTouchEventArgs>(↩
       Window1_HandpartDown);
  }

  void Window1_HandpartDown(object sender, GloveTouchEventArgs e){
    Console.WriteLine(e.Glove.Hand + " " + e.Handpart.Identifier);
  }
}
```

Code Example 5.7: Template code.

5.3.4 Hand Part Event Subscription

The project template already contains a subscription for the HandpartDown event and a callback method (cf. Code Example 5.7). We change the body of the callback method so that it changes the canvas background to blue if the hand part is a thumb and to red if it is the knuckle of the thumb (cf. Figure 5.8).

```
void Window1_HandpartDown(object sender, GloveTouchEventArgs e){
  if(e.Handpart.Identifier == Handpart.Identifiers.Thumb){
    this.Container.Background = Brushes.Blue;
  } else if(e.Handpart.Identifier == Handpart.Identifiers.↩
     ThumbKnuckle){
    this.Container.Background = Brushes.Red;
  }
}
```

Code Example 5.8: HandPartDown callback method.

5.3.5 Posture Event Subscription

In order to change the background color by placing down one of the defined postures, we have to subscribe to another event: PostureDown (cf. Code Example 5.9). This event is received every time the posture recognizer in Code Example 5.7 recognizes that a posture has been placed down on the surface. Also, a callback method has to be added to our code that handles the event. The background is changed to green if the recognized posture was a straight hand; and to yellow if a fist posture was recognized.

```
[...]
 this.HandpartDown += new EventHandler<GloveTouchEventArgs>(↩
    Window1_HandpartDown);
 this.PostureDown += new EventHandler<GlovePostureEventArgs>(↩
    Window1_PostureDown);
}

void Window1_PostureDown(object sender, GlovePostureEventArgs e){
  if(e.RecognizedPosture.Identifier == "StraightHand"){
    this.Container.Background = Brushes.Green;
  } else if(e.RecognizedPosture.Identifier == "Fist"){
    this.Container.Background = Brushes.Yellow;
  }
}

void Window1_HandpartDown(object sender, GloveTouchEventArgs e){
[...]
```

Code Example 5.9: PostureDown event subscription.

This is where the example ends. An application that reacts to certain hand parts and to postures was implemented in just a few steps with the help of the utilities, the template and the API the Fiduciary Glove Toolkit provides.

5.4 Utilities

Besides the two utilities that were mentioned in the walkthrough the Fiduciary Glove Toolkit also features the Gesture Configurator. All three utilities - the Glove, Gesture and Posture Configurators - will be described in the following sections.

5.4.1 Glove Configurator

The Glove Configurator supports the creation of custom glove configurations. A first version was already implemented for the painting application we presented in Chapter 4.3. This early prototype already featured most of the functionality of the final version. The adjustments for the toolkit version are decribed in the end of the section about this utility.

Tag to Hand Part Mapping

The center area of the screen displays three views of the hand; one of the front, one of the back and one of the side of the hand as you can see in Figure 5.8. Rectangular, yellow frames indicated parts of the hand that could be associated with a tag on a glove. When a tag touched one of these indicators, the corresponding hand part got associated with this tag. The association got highlighted by a semitransparent, dark fill of the rectangle and a label on top saying the tags byte value (cf. Figure 5.9).

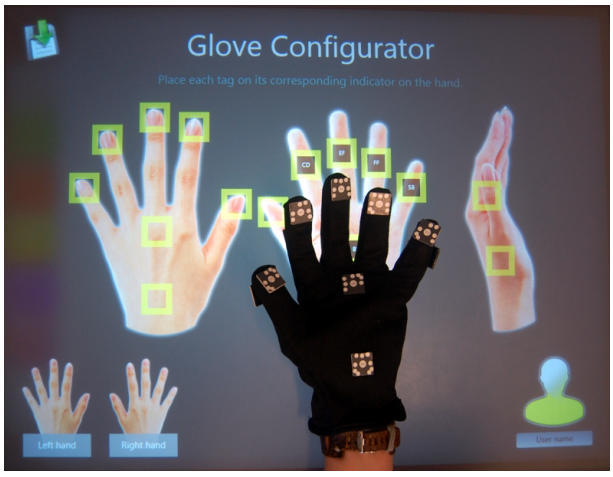

Figure 5.8: Glove configurator prototype.

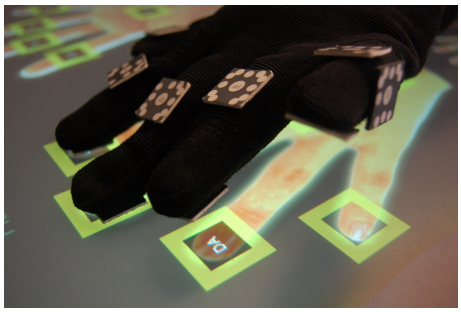

Figure 5.9: Associated indicators.

Hand Selection

In the lower left corner of the screen were a left and a right hand depicted. With these controls a user could switch the view from a left to a right hand and vice versa. The three views of the hand in the center of the screen smoothly transitioned accordingly. The switch from one view to the other did not affect the assignment of tags to hand parts. For instance, if you assigned some tags on a view for a left hand glove, and switched to the view for a right hand glove, the tag assignment for the left glove was kept. If you switched back again to the view for the left glove you could pick up where you left off.

Saving a Glove Configuration

A save button was positioned in the upper left corner. When pressed, it opened a file dialog. You could then pick a destination and a filename for the glove configuration file. Then, an XML-file was created with the following structure: The root was a 'Glove' element. An attribute 'Hand' indicated whether this was a glove for the right or the left hand. An optional 'User' attribute associated a username permanently to the glove. Inside the 'Glove' element, sub-elements were created for every tag associated with an indicator. The byte value of the tag was stored in the 'ID' attribute of this element. Then, an element for both the position on the hand and the hand part type

was created inside the 'Tag' element. The possible values for hand part position and type can be taken from Table 5.1.

Hand part type	Hand part positions
Finger	Thumb, Index, Middle, Ring, Pinkie
Knuckle	Thumb, Index, Middle, Ring, Pinkie
Palm	Palm, Wrist
Back	Back
Side	Hand, Pinkie

Table 5.1: Hand part type and positions.

In the lower right corner a person icon and a label were shown. The label could be used to enter a user name. If a user name was set, this user name would be permanently associated with glove and saved to the optional attribute as mentioned above.

Refined Design

The Glove Configurator has been refined for the toolkit because of some bad design decisions in the initial version: One of the most striking faults was the possibility to assign a user permanently to a glove. The same glove could not be used by different users in different applications or scenarios. A user is now linked to a glove in code, independently for each application. Therefore, the optional attribute in the XML file as well as the control in the lower right corner have been removed in the Glove Configurator version for the toolkit.

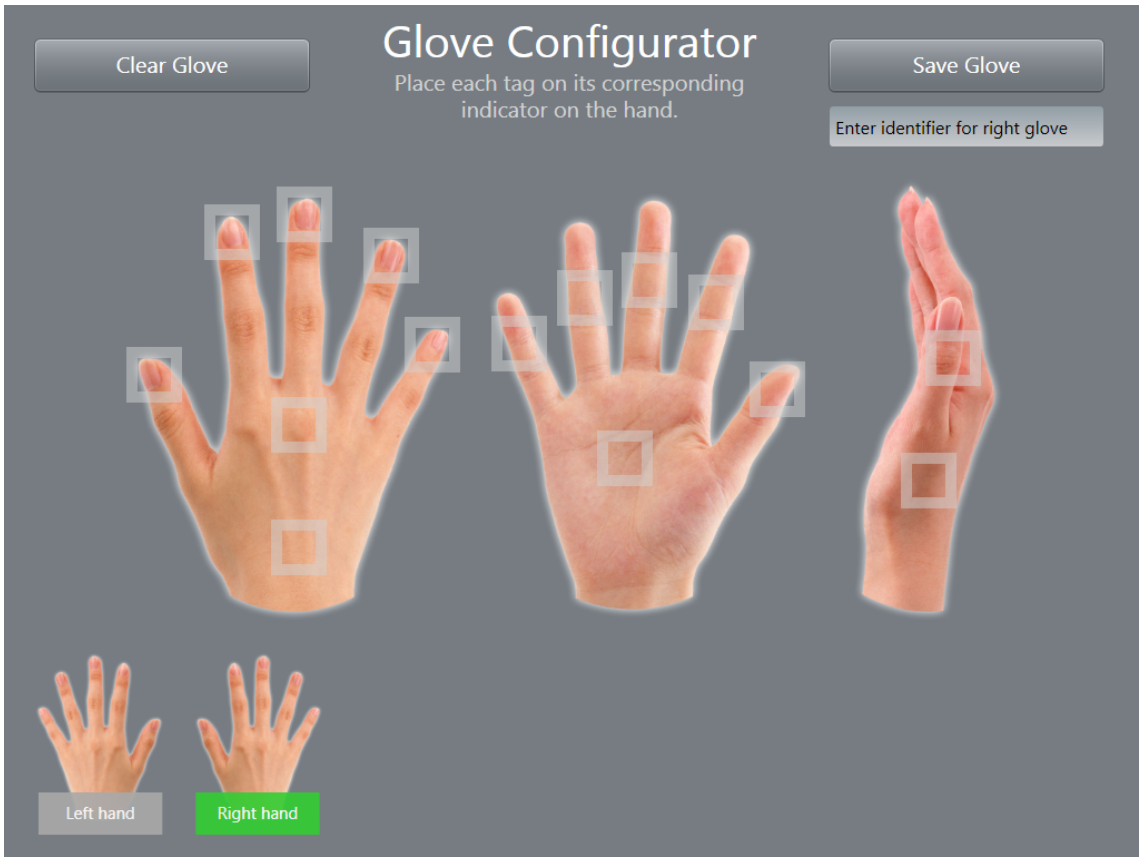

Figure 5.10: Glove Configurator.

The save button can now be found in the upper right corner (cf. Figure 5.10). The position

change was done to achieve a consistent look-and-fell throughout all utilities; in the other two utilities, the Posture Configurator and the Gesture Configurator, the save button is also in the upper right corner. Before you can save a glove configuration you have to enter a glove identifier in the new label below the save button. In the old version of the Glove Configurator this identifier was determined by the filename that was defined in a file dialog. We removed the file dialog in the new version since it does not suit the tabletop environment. File dialogs are simply geared to traditional desktop applications. Now, new glove configuration files as described in Chapter 6 are stored in a central repository. The filename is set by the given glove identifier. Another fault was the absence of a clearing function. Once a tag was associated with a hand part there was no way to remove it again. You could override the old value by touching the same indicator with a different tag, but clearing the value was simply not possible. In the new Glove Configurator a clear button is added in the upper left corner of the screen, the former position of the save button. By pressing it, all indicators in the current view are cleared. Also, the rectangular indicators are more subtle. The frame has now a semitransparent white color. This improves the impression that you are mapping your glove to a hand and not tags to indicators. The active state visual has changed just a little bit. If an indicator is associated with a tag, the frame color is then tinted in lime green. The fill of a set indicator is the same as in the old version - a semitransparent dark gray with a label on top saying the tags byte value (cf. Figure 5.11).

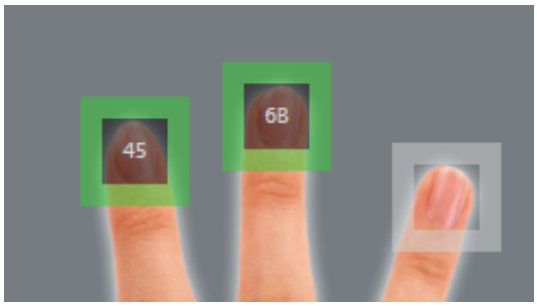

Figure 5.11: Hand part indicators.

5.4.2 Gesture Configurator

The Gesture Configurator facilitates the creation, training and testing of gestures. Each gesture has a gesture name and is defined by a set of samples. A sample is basically a point array that was sampled while tracing a path. They are stored as XML files in a gesture repository that is created in the course of the installation of the Fiduciary Glove Toolkit.

Defining a Gesture

The first step to creating, training or testing a gesture is tracing a continuous, single stroke path, e.g. a line or circle, with any hand part. It does not matter which hand part traces the path because in this utility the focus is on testing and defining the raw version of a gesture and not the version that got enriched with the hand part information. Even though this means that paths could also be traced with a normal touch, we decided that gesture paths always have to be traced with a hand part. The reason behind this decision is that the tracking of tags is slower than the tracking of normal touches. A gesture defined by a normal touch could not be performed with a hand part if the touch moved faster during the tracing than a tag could be tracked. Whenever the hand part position changes while tracing a path, its position is stored to a point array together with a timestamp. This array will later be used by the gesture recognition engine.

In order to give immediate visual feedback, the traced path is drawn as a lime green path. With this visual representation, a user of the utility can get an instant impression of how the raw

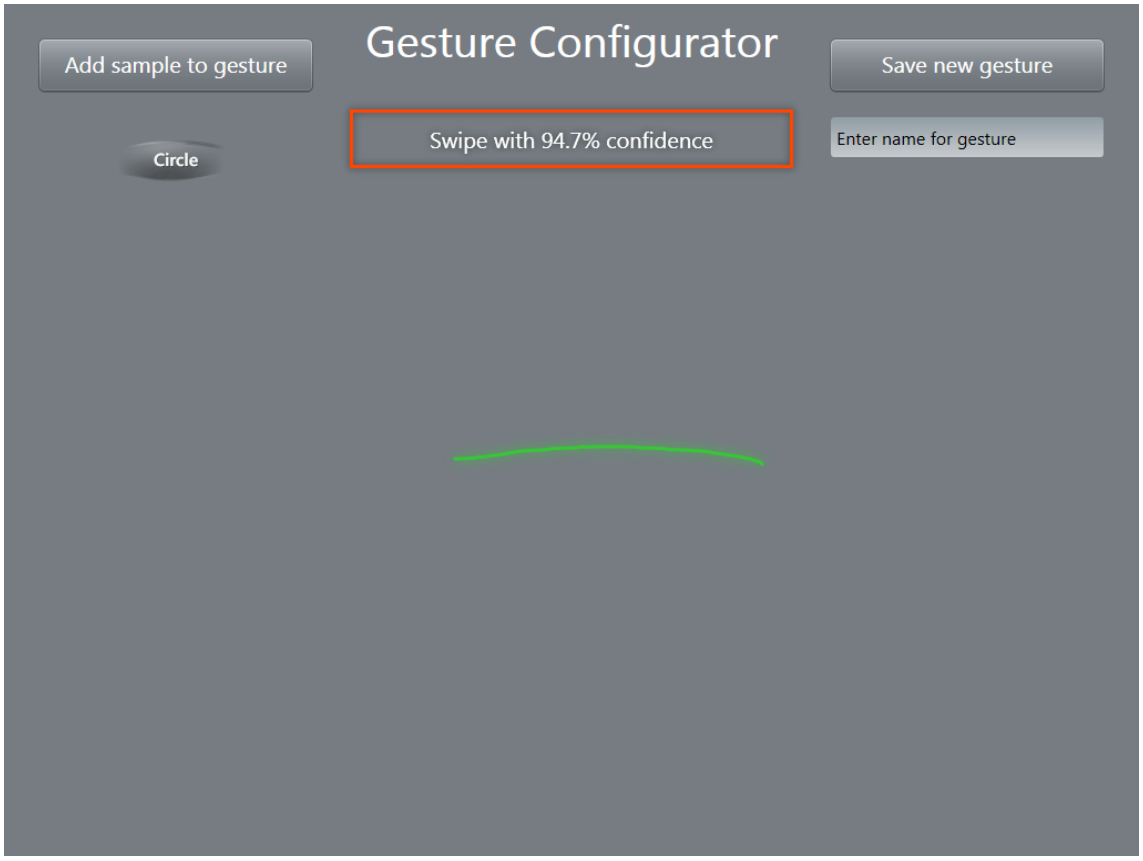

Figure 5.12: Gesture Configurator.

path data was recorded by the Microsoft Surface. The user can also recognize when the Microsoft Surface lost track of the hand part, for example because the hand part moved too fast. In this case, the path ends at the position of the tracking loss.

Gesture Recognition

When the path is finished - either on purpose by lifting the tracing hand part or accidently as aforementioned - a glow around the drawn path indicates that the path is now processed by the recognition engine that is a built-in feature of the Fiduciary Glove API. Therefore, the sampled point array is passed to the gesture recognition engine. The engine compares the array to the samples of existing gestures and returns a recognition result. The name and the confidence of the best matching gesture are then displayed in the framed label below the 'Gesture Configurator' title.

If the user's intention is to define a new gesture, the recognition result is an important indicator. If the recognition confidence is high, i.e. 80 percent and above, it means that the same, or at least a very similar gesture already exists. In order to caution against possible interferences with existing gestures the frame of the recognition result label is tinted in orange for a recognition confidence of 80 to 90 percent, and in deep red for a confidence of over 90 percent (cf. Figure 5.12 top, Figure 5.13).

In case of a high recognition confidence, defining a new gesture with the traced path does not make much sense. It will most certainly interfere with the already existing gesture. If the recognized gesture name fits the look of the traced path, the user should use the displayed gesture name to access the gesture in an application. Alternatively, the user can still save the gesture and load only this gesture into the gesture recognition engine later to handle the interference of the

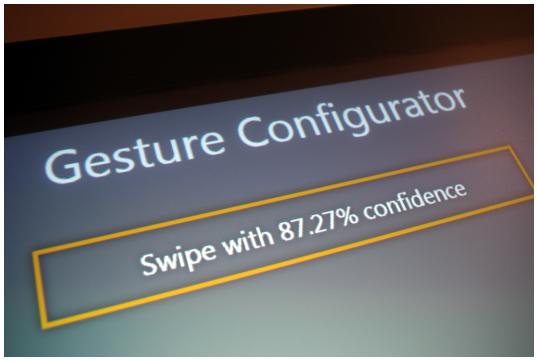

Figure 5.13: Gesture recognition result.

two.

Saving a Gesture

For saving a gesture, you have to enter a gesture name in the label below the save button in the upper right corner. Only then the saving button will react. The Gesture Configurator will then check if a gesture with the same identifier already exists. If the gesture name is not taken, a new gesture with this name is created and the path is used as a first sample for the gesture. If a gesture with the same name already exists, the Gesture Configurator adds the path as a new sample for the gesture with the given name.

Adding a Sample to an Exisiting Gesture

The traced path can also be added to the samples of a gesture directly. Therefore you choose the name of the gesture you want to add a sample for from the collapsed list of all gestures below the button in the upper left corner. After selecting the appropriate gesture and pressing the button above, the path is saved as a new sample for this gesture. Saving a new sample makes a lot of sense, when the right gesture for the traced path exists, but another gesture was recognized instead. Adding the path data as a new sample for the right gesture will lead to better recognition results.

5.4.3 Posture Configurator

The Posture Configurator lets you create, update and test custom postures. Postures are defined by the distance and angle between hand parts and gloves.

Defining a Posture

Whenever a hand part touches the surface, an ellipse is drawn beneath it - the hand part blob. A circular label around this hand part blob displays the hand part identifier and whether this hand part is on a glove for a right or left hand (cf. Figure 5.15). The label is rotated so that its center intersects with the orientation vector of the hand part. The hand part blob orientation is set to the orientation of the hand part. If multiple hand parts touch the surface, connections are drawn between the blobs of the hand parts as long as they belong to the same glove and are the same hand part type. An additional constraint is that connections are only drawn between nearest neighbors amongst touching hand parts of the same hand part type. For instance, if the thumb, the index finger, the pinkie and the palm of one glove touch the surface, connections are drawn from thumb to index finger and from index finger to pinkie. A connection to the sole palm hand part does not exist, since the palm belongs to a different hand part type. At the center of each connection the

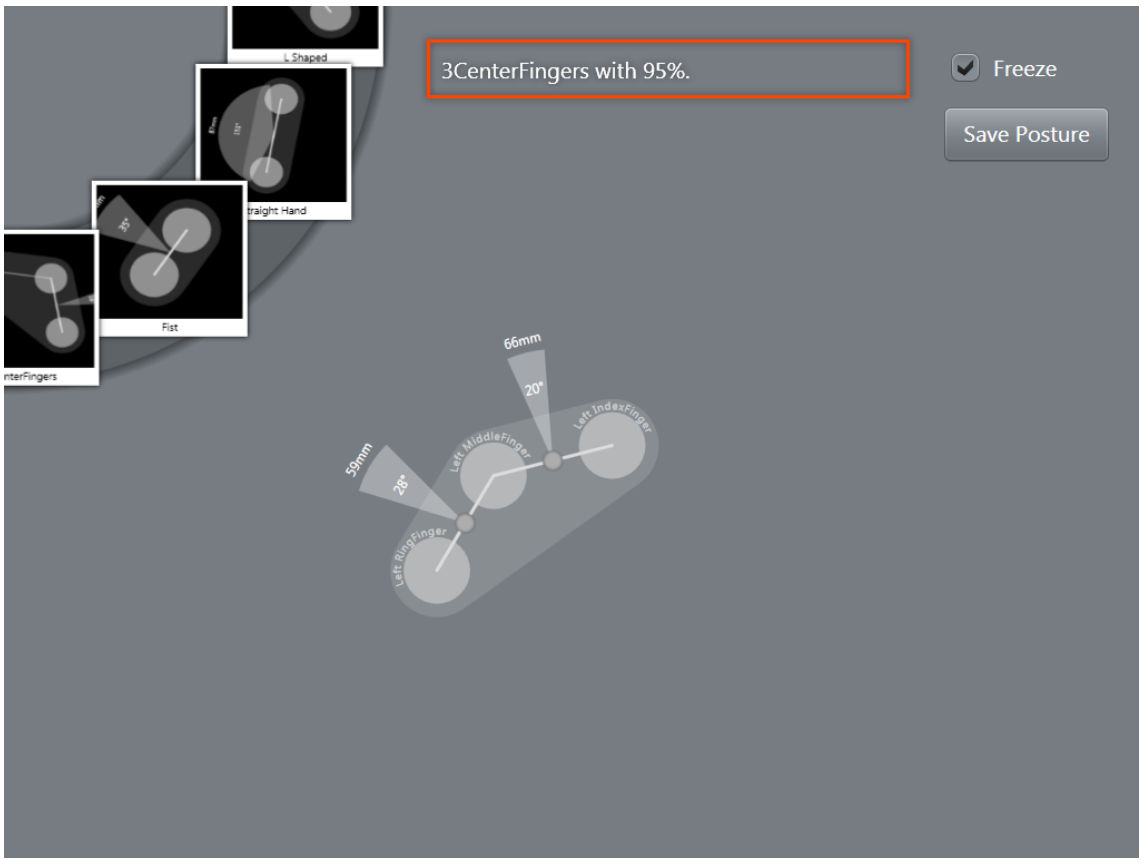

Figure 5.14: Posture Configurator.

center point of a pie segment is positioned. The spanned angle of this segment is the orientation difference of the two connected hand part blobs. Additionally, a label in the center of the pie segment displays this angle in degrees. Just outside the arc of the pie segment, a label shows the distance between the hand parts in millimeters.

Adjusting Distance and Angle Tolerance

In the center of each connection a handle is positioned. After pressing this handle, its color turns green and dashed paths representing the tolerance values for distance and angle as well as handles to changes these are unveiled. Parallel to the connection, two buttons for ignoring the distance or angle are shown (cf. Figure 5.16). The default distance tolerance is five millimeters. The connection line between the two hand parts is extended by half the distance tolerance to both sides. These extensions are drawn as white dashed lines. At the outer end of these lines, the adjustment handles are attached. They are colored in red.

When a touch enters one of the handles, both get activated and turn green (cf. Figure 5.17). A new distance tolerance is set by moving the touch that entered the handle. The touch does not have to stay on the handle or even in line with the connection. The distance tolerance and extension lines on both sides are adjusted according to the distance between the touch and the center point of the hand part ellipse on the side of the entered handle. When lifting the touch the new distance tolerance is permanently set and the handles turn red again (cf. Figure 5.18).

The default angle tolerance is five degrees. The pie segment representing the angle between the two hand parts is extended to both sides by half the angle tolerance. The extended segment is not filled but features a white dashed line as border. At the two corners of the arc, red adjustment handles are attached.

Figure 5.15: Hand part connection.

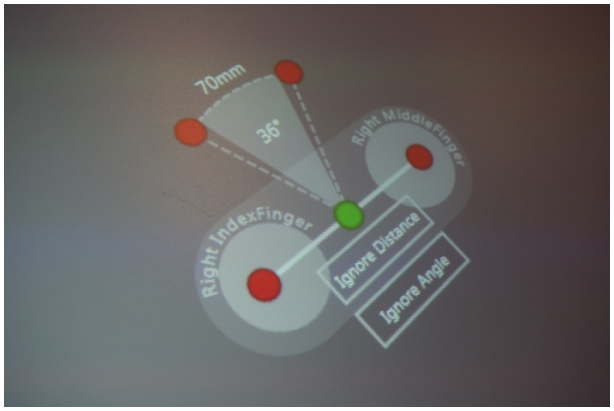

Figure 5.16: Distance and angle tolerance.

The angle tolerance handles are activated in the same fashion as the handles for the distance tolerance. By entering either one of them, both handles turn green (cf. Figure 5.19). Depending on the distance between the touch and the center point of the pie segment arc, the angel tolerance and the angle of the dashed pie segment are adjusted.

Ignoring Distance and Angle

The distance and angle between two connected hand part blobs can be ignored by pressing the corresponding button parallel to the connection line. When ignoring the distance, the color of the connection line changes from white to a semitransparent gray. Ignoring the angle hides the pie segment. Ignored factors will not be considered during the recognition. For instance, if you only care about the hand parts forming a posture, but not about the distance and angle between them, you ignore both distance and angle between their blobs. The posture is always recognized whenever the touching hand parts are the same as before, the distances and angles can be different.

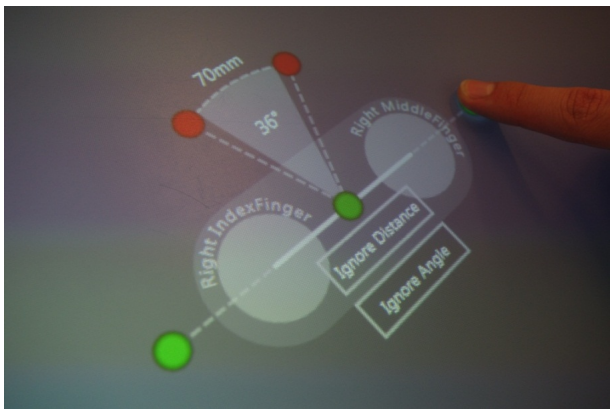

Figure 5.17: Adjusting the distance tolerance.

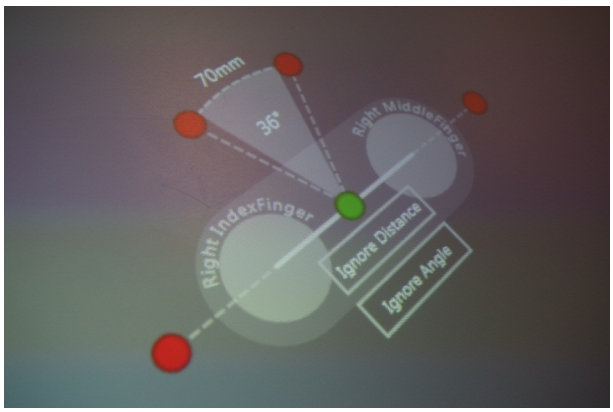

Figure 5.18: Changed distance tolerance.

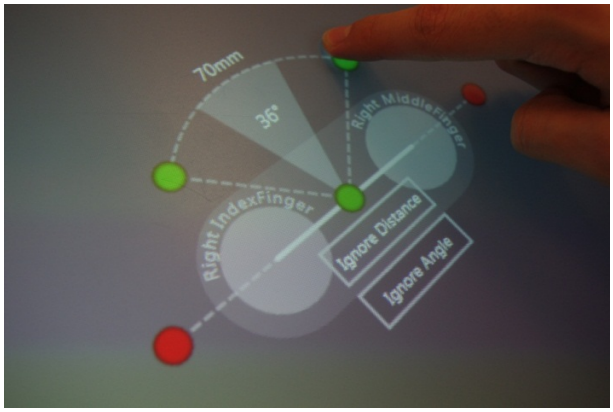

Figure 5.19: Anjusting the angle tolerance.

Glove Blobs

In order to emphasize that hand parts belong to the same glove a gray blob is drawn beneath all hand parts of a glove as you can see in Figure 5.20.

The balance point of the blob serves as glove blob position (cf. black circle in Figure 5.21). The orientation for the glove blob is calculated in the following way: First the orientation of each hand part type is calculated. If only one hand part blob is drawn for a certain hand part type then the hand part type orientation corresponds to the orientation of the sole hand part blob. If more hand part blobs of the same hand part type exist, then the slopes of the connections are calculated.

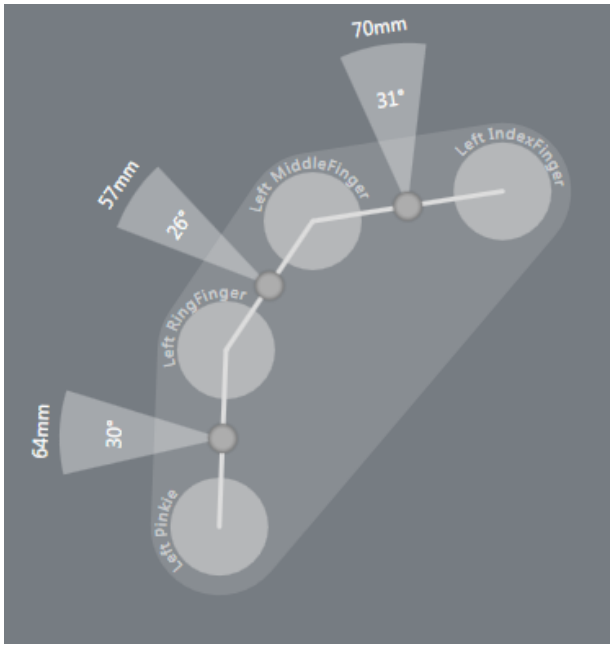

Figure 5.20: Glove blob.

The angle of the perpendicular of each slope is set as orientation of the corresponding connection. The mean of all connection orientations defines the orientation of the hand part type. Analogically, the glove blob orientation is the mean of the hand part type orientations (cf. arrow in Figure 5.21).

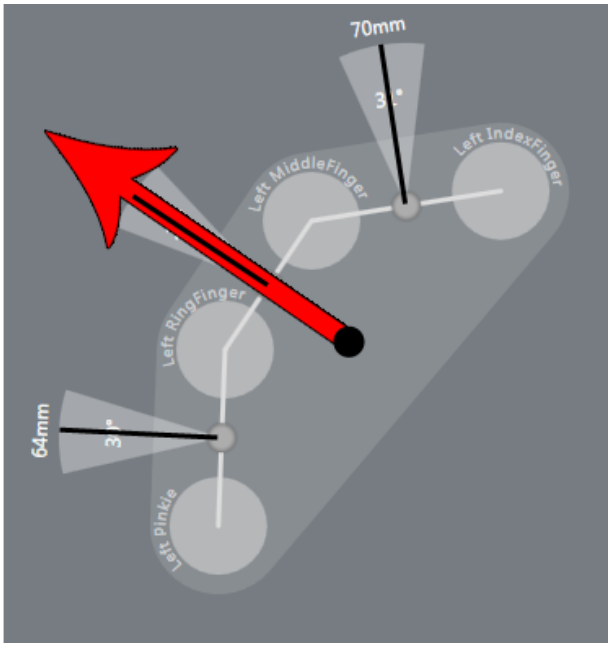

Figure 5.21: Glove blob orientation.

In an older version of the Posture Configurator, the glove blob orientation was the mean of the orientations of the hand part blobs and not the mean of the orientation of the connections. None of the two options had clear advantages over the other. For some postures the first version calculated a better overall orientation for the whole glove blob, for others the second one did. Therefore, we visualized the alternatives in the same view, as you can see in Figure 5.22. The green arrow pointing in the one o'clock direction visualized the version where the orientation was

calculated by the mean of the hand part blob orientations, the red arrow pointing in the two o'clock direction the other version. Then, informal interviews with some of the members of the lab this project was developed at were conducted. Most of the experts argued for the second version, because they found that through the visualization of the connections the red arrow mapped better to the subjective perception of the glove blob orientation. Therefore, we picked this orientation alternative. The same was applied for the the orientation of a glove. The orientation of a glove is now calculated by the orientation mean of the fictive connections between hand parts that touch the surface.

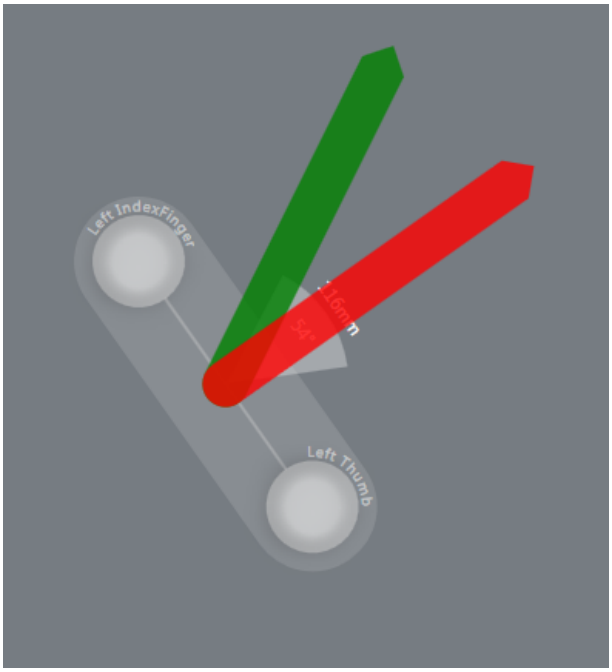

Figure 5.22: Glove blob orientation alternatives.

Multiple Glove Blobs

When multiple glove blobs are drawn, the nearest neighbor is searched for each glove blob. A connection line is added from the balance point of each glove blob to its nearest neighbor. These glove connections look just like the hand part connections: First of all, a line is drawn between the balance points of the glove blobs (cf. Figure 5.23). A pie segment is attached to the center point of this line and represents the angle between the glove blobs which is the orientation difference of the gloves. Also, a handle for accessing the adjustment mode is positioned there. When pressing this handle, tolerance handles and ignore-buttons appear. They work identically as their counterparts for hand part connections. The defined tolerance for distance or angle specifies how much the distance or angle between the gloves may differ from the original posture. If the distance or angle is ignored, the particular factor is not considered during the recognition.

Posture Recognition

Whenever a blob or connection changes, it passes a list of all glove blobs to an adjusted version of posture recognizer which is provided by the Fiduciary Glove API. This version accepts a list of glove blobs instead of hand parts. This allows for a posture recognition without hand parts touching the surface, e.g. during freeze mode. The posture recognizer compares the glove blob configuration to existing postures and returns a list of recognized postures, each entry containing

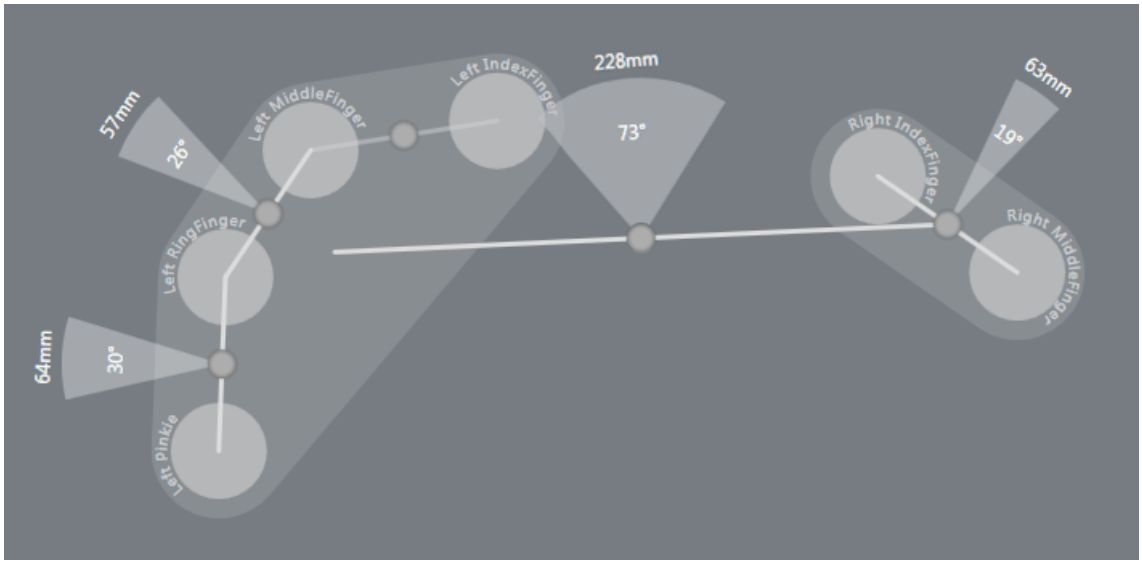

Figure 5.23: Glove blob connection.

the posture name and the recognition confidence. The posture name and confidence of the best match are displayed in a label at the top of the screen (cf. Figure 5.24). The frame of the label has an orange color when the recognition confidence is between 80 and 90 percent, a red color for values over 90 percent.

Figure 5.24: Posture recognition result.

Freeze Function

In the normal mode, the glove blob and hand part ellipses and connections are only shown when the hand parts are in contact with the surface (cf. Figure 5.15). The visuals disappear when the glove or respectively hand part are lifted from the surface. In order to have an unobstructed view on the visual representation, a freeze mode was integrated. After checking the freeze mode checkbox in the upper right corner, the current visual representations of gloves and hand parts do not disappear when the gloves and hand parts leave the surface. As a consequence of this, we had to design the aforementioned blobs so that we could permanently access the information provided by touching hand parts. The last known position and orientation is stored for the corresponding blobs when the hand part leaves the surface. With this easy mechanism, the postures can be adjusted and saved, even if all hand parts have left the surface. Another advantage of the freeze mode is that the

handles can be controlled much easier because there are no actual hand parts blocking the space anymore. Also, postures with multiple gloves can be created by a single person now.

Browsing Existing Postures

With the ring control in the upper left corner (cf. Figure 5.25), you can browse through existing postures. For each posture the ring control shows a screenshot that was taken when the posture was saved. At the bottom of the white border around the screenshot the posture name is displayed. The postures are ordered by the number of hand parts used to form the posture. By rotating the ring control, postures that are currently not visible can be moved into view. If all postures do not fit on the ring control, they are dynamically exchanged off-screen. Depending on which posture representations are shown in the visible quarter of the ring, items on the opposite side of the ring are switched.

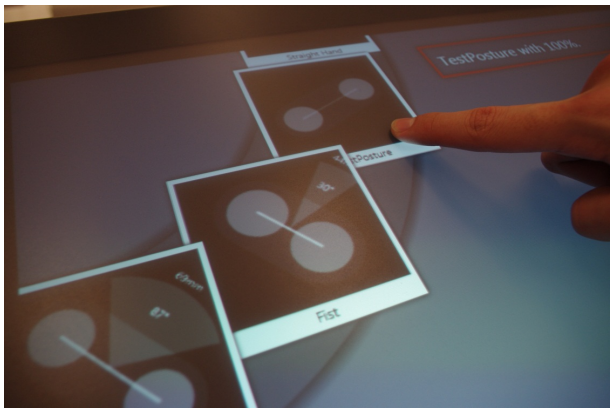

Figure 5.25: Ring control.

Saving a Posture

For saving the defined posture, the save button has to be pressed. Thereafter, a dark overlay appears which contains a list view of all glove blobs (cf. Figure 5.26). Each entry consists of a screenshot of the glove blob, a textbox for the name of the single-glove posture the glove blobs forms, and two checkboxes. Below the screenshot, the glove identifier and the hand of the glove that formed the blob are displayed. The default text in the textbox requests the user to enter a name for the defined single glove posture. The first checkbox specifies if this posture should be saved as hand independent. Hand independence means that the posture will be recognized, no matter if the performing glove is a right or left hand glove. If this option is not checked, the posture is only recognized if performed with a glove for the same hand as the glove that defined the posture. The second checkbox specifies if the posture should be saved. By default, the hand independence checkbox is unchecked, the save checkbox checked.

If the glove blob formed an existing single glove posture, the label displays the name of the recognized posture, and the save checkbox is unchecked. If the existing posture should be overwritten with the new version, this checkbox has to be checked manually.

If more than one glove blob exists, another entry for a multi-glove posture is added. Its screenshot has an orange border instead of a white border to set it visually apart from the single -glove postures. The screenshot shows an overview over all glove blobs forming the multi-glove posture. Because multiple gloves are involved, a checkbox for hand independence would not make much sense in this case. Therefore, only a save checkbox exists for this entry. Once every list entry is configured as desired, the save button at the bottom of the overlay has to be pressed to save all

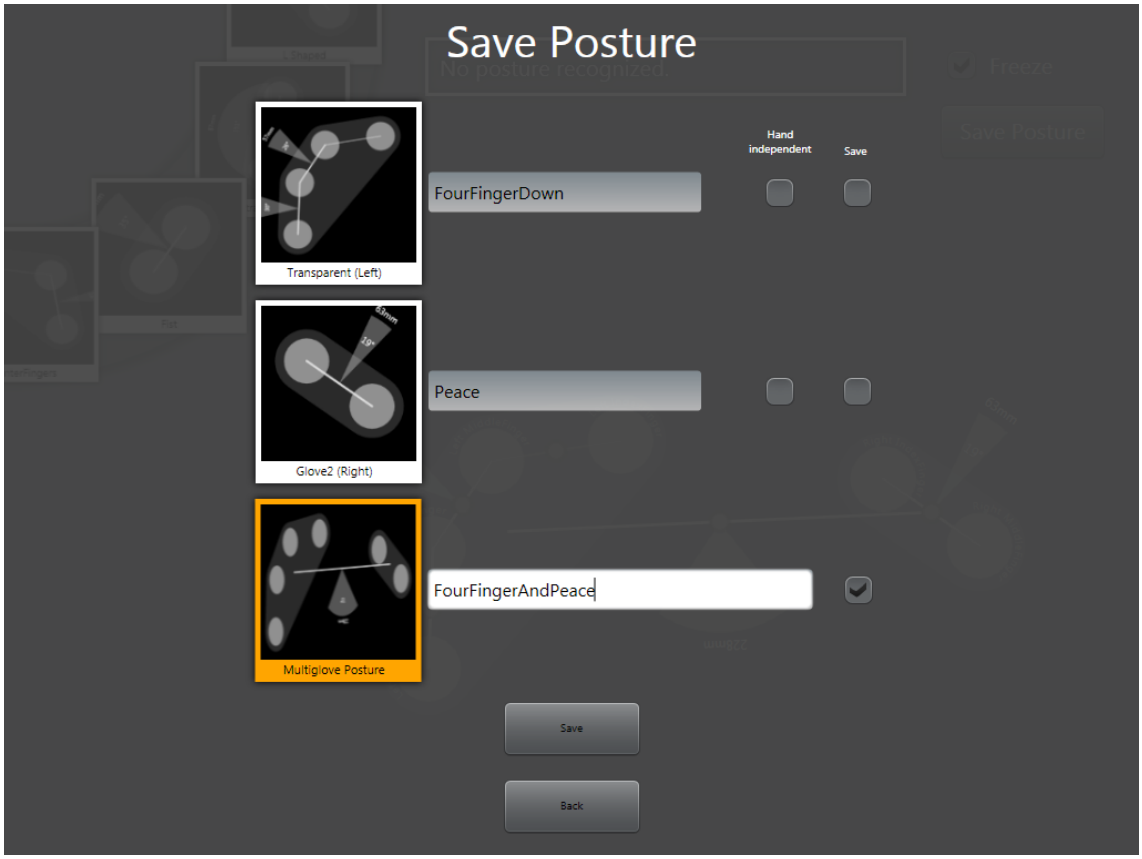

Figure 5.26: Posture saving screen.

gestures with a checked save checkbox. The gestures are then saved to posture configuration files in XML format.

The back button on the bottom of the screen cancels the save process, hides the overlay and returns to the normal view.

5.4.4 Implementation

All utilities of the Fiduciary Glove Toolkit are based on C#.NET using WPF 3.5 and especially the Surface SDK. Many controls are Surface controls. These controls are touch-capable counterparts of traditional C# or WPF controls. They react only to touches and not tags. This prevents the user form accidently touching them with a hand part on a glove. We leveraged this restriction as an implicit confirmation mechanism. The user has to take off a glove and consciously touch the control with a bare finger. Surface textboxes offer another advantage over normal textboxes. When a Surface textbox is touched, a software keyboard is shown on screen that allows entering text into this label. A hardware keyboard is not needed and the user's focus can stay on screen.The Gesture and Posture Configurator use the API of the Fiduciary Glove Toolkit.In the following section, unique implementation details for each utility are described.

Glove Configurator

- The Glove Configurator is a `SurfaceWindow` and receives touch events.
- Important controls in the Glove Configurator are the rectangular indicators. They are `SurfaceUserControls` and are therefore able to handle contact events raised by the Surface software. Whenever a tag touches one of these indicators, the control stores the byte

value of the tag and changes its appearance as described earlier. This happens independently from other indicators.

- The controls for switching the view from a left to a right hand and back are also `SurfaceUserControls`.

Gesture Configurator

- The Gesture Configurator is a `GloveWindow` which is provided by the Fiduciary Glove API. This enables the Gesture Configurator to receive hand part, gesture and posture related events.
- The collapsed list in the upper left corner is a `SurfaceSelectorControl`. This control is not included in the standard Surface SDK. Fortunately it can be integrated into an application by adding the SurfaceSelectorControl.dll as a reference. This file is provided with the Newsreader Application for the Microsoft Surface.
- For the gesture recognition, the Gesture Configurator uses the gesture recognizer provided with our API.

Posture Configurator

- The Posture Configurator is a `GloveWindow`.
- The ring control in the upper left corner of the screen is a custom control based on a `SurfaceUserControl`.
- Also the handles used to adjust tolerances or ignore angle or distance of a hand part or glove connection are `SurfaceUserControls`.
- Each checkbox in the Posture Configurator is a `SurfaceCheckbox`.
- For the posture recognition, the Posture Configurator uses a modified version of posture recognizer provided with our API. The reason is that the built-in posture recognizer uses a list of touching hand parts for its recognition. For the Posture Configurator, however, it is important that postures are recognized based on glove blobs and not hand parts. In this way, the currently displayed glove blobs can be checked against possible interference with existing postures when the freeze mode is active and no hand parts are in contact with the surface. Hence, the posture recognition in the Posture Configurator takes a list of glove blobs instead of hand parts as input parameter.

5.5 The Fiduciary Glove API

Figure 5.27 illustrates the basic class architecture of the Fiduciary Glove API. The depicted elements and dependencies will be described in detail in the following sections.

5.5.1 GloveWindow

The basis of every application using the Fiduciary Glove API is the `GloveWindow`. It inherits from SurfaceWindow and is therefore capable of receiving contact events inside the window. Because SurfaceWindows are fullscreen by default, the `GloveWindow` basically gathers the events from all contacts, no matter where they happen on screen. Each of these contact events is analyzed by the `GloveWindow`. Depending on the result of this analysis hand part, gesture or posture events are raised which is elaborated on in the following sections.

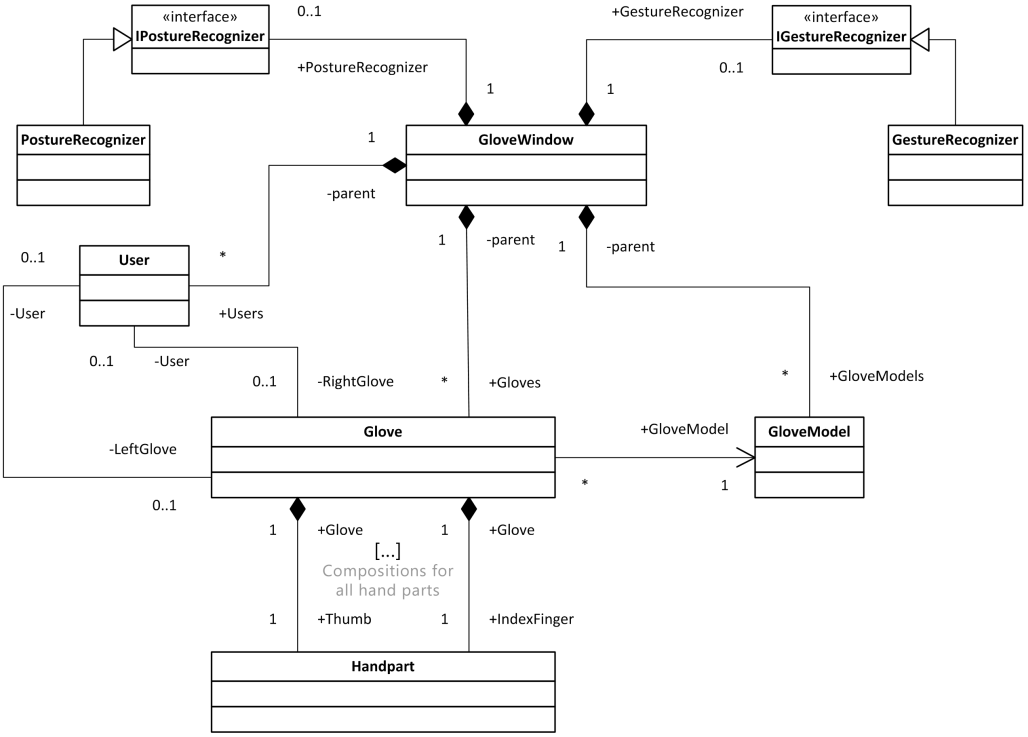

Figure 5.27: Class diagram.

5.5.2 Gloves and Hand Parts

This section covers how information about gloves and hand parts is made available to developers. Also, the identifier and type of each hand part as well as available hand part events are listed.

Glove Configuration Files

The source for hand part information are glove configurations. A glove configuration specifies the tag to hand part mapping for a specific Fiduciary Glove. These configurations can be created with the Glove Configurator (cf. Chapter 5.4.1). They are saved to glove configuration files in XML format. The XML root is a 'Glove' element. In its attribute 'Identifier' the glove identifier is stored. This way, the identifier does not have to be extracted from the filename, but can be retrieved during the XML parsing process. The 'Hand' attribute specifies if the gloves is for a right or left hand. In contrast to the glove configuration files used for the first application for the Fiduciary Glove (cf. Chapter 4.3) the new version in the API is built from a hand part perspective. For each hand part an element with the name of the hand part is created, e.g. a 'MiddleFinger' element for the middle finger. The content of this element is the hexadecimal value of the tag, which can be read off from the center circle on a tag. If no tag is mapped to that hand part, the content of the hand part element stays empty.

Gloves

The glove configurations are then loaded by the Fiduciary Glove API. For each configuration an instance of `GloveModel` is created. A `GloveModel` object has an 'Identifier' property and a 'Hand' property. Another 15 properties model the tag to hand part mapping of the current glove configuration. The `Handpart` properties are named after the particular hand part with a concatenated 'Tag', e.g. 'IndexFingerTag'. All properties are set with the corresponding values provided by the glove configuration file. A `GloveModel` basically stores only the tag to hand

Hand part	Identifier	HandpartType
Thumb	Thumb	Finger
Index finger	IndexFinger	Finger
Middle finger	MiddleFinger	Finger
Ring finger	RingFinger	Finger
Pinkie	Pinkie	Finger
Back of the thumb	ThumbKnuckle	Knuckle
Back of the index finger	IndexKnuckle	Knuckle
Back of the middle finger	MiddleKnuckle	Knuckle
Back of the ring finger	RingKnuckle	Knuckle
Back of the pinkie	PinkieKnuckle	Knuckle
Palm	Palm	Palm
Heel of the hand	PalmWrist	Palm
Back of the hand	BackOfHand	BackOfHand
Outer side of the pinkie	SideOfPinkie	SideOfHand
Side of the hand	SideOfHand	SideOfHand

Table 5.2: Hand part identifiers and types.

part mapping. To be able to access not only tag values of hand parts, but high-level hand part objects, instances of the type `Glove` have to be created. Each `Glove` is associated with up to 15 `Handpart` instances via 15 properties. The properties are named after the corresponding hand part, e.g. 'RingFinger'. This design demands many properties, but allows for a very easy access of hand parts. For instance, if the `Handpart` instance for the thumb needs to be accessed, the 'Thumb' property is used (cf. Code Example 5.10).

```
Handpart thumb = glove1.Thumb;
Handpart indexFinger = glove1.IndexFinger;
```

Code Example 5.10: Accessing hand parts of a glove.

The tag value for a hand part can be retrieved over the associated `GloveModel`. A `GloveModel` can be associated with multiple `Glove` instances; they act as proxies. Each time a tag contact event is raised by the Microsoft Surface, the `GloveModels` of all `Glove` instances are searched for the tag. Therefor the tag to hand part mappings in each `GloveModel` are compared to the tag value. In case of a match, an event is raised for all `Glove` instances associated with this `GloveModel`.

Hand Parts

When instantiating a new `Glove` for a `GloveModel`, it is checked which of the mapping properties in the `GloveModel` are set. If a property has a value, a `Handpart` instance is created and assigned to the corresponding property in the `Glove` instance. For instance, if the 'IndexFingerTag' property of the `GloveModel` is set, the 'IndexFinger' property of the `Glove` is set with a new instance of the type `Handpart`. The `Handpart` type contains an 'Identifier' property which is set with the handpart identifier, of course. This way, the hand part can be identified without knowing the corresponding `Glove` object. Depending on the assigned identifier, the 'HandpartType' property is set automatically. The corresponding hand part identifier and hand part type for each hand part are shown in Table 5.2.

Each `Handpart` also includes two properties 'Position' and 'Orientation'. Both are instantly updated when the corresponding tag changes its position or orientation on screen. The Boolean property 'IsDown' indicates whether the hand part (or the tag respectively) is currently in contact with the surface or not.

Hand part event	Description
HandpartChanged	Occurs when a hand part over an element changes its attributes, including position and orientation.
HandpartDown	Occurs when a hand part over an element is placed on the Microsoft Surface screen.
HandpartEnter	Occurs when a hand part enters an element's boundaries or is placed on the area inside the boundaries.
HandpartHoldGesture	Occurs when the Microsoft Surface software recognizes a press-and-hold gesture by a hand part.
HandpartLeave	Occurs when a hand part leaves an element's boundaries or leaves the area over the element.
HandpartTapGesture	Occurs when the Microsoft Surface software recognizes a tap gesture by a hand part.
HandpartUp	Occurs when a hand part over an element leaves the Microsoft Surface screen.

Table 5.3: Hand part events.

Hand Part Events

A hand part event exists for every contact event in the Surface SDK (cf. Table 3.1) as you can see in Table 5.3. The corresponding hand part event is raised when the `GloveWindow` receives a contact event of a tag that was mapped to a hand part in one of the `GloveModel` instances. These events can be received on different levels (cf. Code Example 5.11). If you subscribe to an event with a `GloveWindow`, the event is received for any hand part on any glove. If you subscribe to an event with a `Glove`, the event is only received if caused by one of its hand parts. If a `Handpart` subscribes to an event, the event is received only if the hand part caused it itself.

```
public partial class Window1 : GloveWindow
{
  public Window1(){
    Glove glove1 = new Glove(this, "Glove1");
    Glove glove2 = new Glove(this, "SecondGlove");

    // Hand part event subscription with GloveWindow. The event is ↩
       received for both gloves.
    this.HandpartDown += new EventHandler<GloveTouchEventArgs>(↩
       Window1_HandpartDown);

    //Hand part event subscription with Glove. The event is only ↩
       received if caused by hand parts on glove1.
    glove1.HandpartDown += new EventHandler<GloveTouchEventArgs>(↩
       glove1_HandpartDown);

    //Hand part event subscription with Handpart. The event is only ↩
       received if caused by index finger on glove1.
    glove1.IndexFinger.HandpartDown += new EventHandler<↩
       GloveTouchEventArgs>(glove1IndexFinger_HandpartDown);
  }
}
```

Code Example 5.11: Hand part event subscription.

Hand Part Type Events

You can also subscribe only to events caused by a certain hand part type. In order to do so, simply exchange the 'Handpart' in 'HandpartDown' or any other hand part event with the desired hand part type. For example a 'FingerDown' event is only received if a finger caused the contact on the surface. You cannot subscribe to these kind of events with a `Handpart`, since it is already logically associated with a certain hand part type (cf. Code Example 5.12.

```
public partial class Window1 : GloveWindow
{
  public Window1(){
    Glove glove1 = new Glove(this, "Glove1");
    Glove glove2 = new Glove(this, "SecondGlove");

    // Finger event subscription with GloveWindow.
    // The event is received for both gloves.
    this.FingerDown += new EventHandler<GloveTouchEventArgs>(↩
        Window1_FingerDown);

    // Back of hand event subscription with Glove.
    // The event is only received if caused by the back of glove1.
    glove1.BackOfHandLeave += new EventHandler<GloveTouchEventArgs>(↩
        glove1_BackOfHandLeave);
  }
}
```

Code Example 5.12: Hand part type event subscription.

GloveTouchEventArgs Usage in Event Callback

The event arguments all hand part events provide are GloveTouchEventArgs. These arguments most importantly contain the `Handpart` that caused the event. Also, the corresponding `Glove` is exposed. It could be accessed via the `Handpart`, but since it is often used in common scenarios providing a shortcut seemed reasonable. For performance reasons, the tag identifier is directly accessible. This value is already provided by the contact event arguments, retrieving it from the `Glove`'s corresponding `GloveModel` would only cost performance. In order to offer a complete argument set for experts, the original contact event arguments are provided, too.

These arguments can be used in the event callbacks to assign separate functions to separate hand parts (cf. Code Example 5.13). Functions can also be assigned to hand parts that meet certain criteria (cf. Code Example 5.13, last if-statement).

```
void Window1_HandpartDown(object sender, GloveTouchEventArgs e){
   if(e.Handpart.Identifier = Handpart.Identifiers.Thumb){
     // Function for thumb.
   }
   if(e.Handpart.Identifier = Handpart.Identifiers.IndexFinger){
     // Function for index finger.
   }
   if(e.Handpart.Position.X > 100.0){
     // Function for any handpart whose x-coordinate is greater than ↩
         100.0.
   }
}
```

Code Example 5.13: Hand part event callback.

5.5.3 Users

A `Glove` instance can be assigned to a specific person. A person is represented by an instance of the type `User`. The name of the user is set with the property 'Identifier'. The 'RightGlove' and 'LeftGlove' properties have to be set with `Glove` instances. Hence, a `User` can receive events for both gloves.

User Subscription to Events

If a `User` subscribes to a hand part or hand part type event, the event is received if caused by an appropriate hand part on any of his two gloves (cf. Code Example 5.14).

```
public partial class Window1 :  GloveWindow
{
  public Window1(){
    Glove glove1 = new Glove(this, "Glove1");
    Glove glove2 = new Glove(this, "SecondGlove");
    Glove glove3 = new Glove(this, "G3");

    User tim = new User(this, "Tim");

    tim.RightGlove = glove1;
    tim.LeftGlove = glove2;

    // Side of hand event subscription with User.
    // The event is received if caused by glove1 or glove2.
    tim.SideOfHandChanged += new EventHandler<GloveTouchEventArgs>(↩
        tim_SideOfHandChanged);
  }
}
```

Code Example 5.14: User event subscription.

5.5.4 Gestures

The Fiduciary Glove API features a modified version of the $1 recognizer by Wobbrock which was described in Chapter 3.4. The reason for the modification was that the standard version of the provided C# library [1] did not recognize which hand part performed a gesture, of course. Hence, the original recognition method has been extended, so that it now accepts a hand part as second parameter next to the point array. When a hand part leaves the surface, not only its point array is passed to the recognition method, but also the hand part itself. The recognition method then does its job as usual, but adds the hand part to each of the recognition results. This way the hand part is associated with the recognized gesture. With this solution, different actions can be taken depending on which hand part performed the gesture.

Instantiating a Gesture Recognizer

In order to receive gesture events, the 'GestureRecognizer' property of the `GloveWindow` has to be set with an implementation of the interface `IGestureRecognizer`. By typing the property with an interface, it is very easy for developers to exchange the standard gesture recognizer implementation in the toolkit with their own, custom gesture recognizer. As name of the standard implementation the most obvious was picked: 'GestureRecognizer'. This helps average skilled programmers to quickly find this appropriate implementation for the interface. When the gesture recognizer is instantiated, it has to load the gestures it is supposed to recognize. The `LoadGestures` method

takes an `Uri` that specifies in which folder the gestures are. In Code Example 5.15 the `Uri` is retrieved from the class variable 'GestureRepository', which points to the gesture repository folder that is created when installing the Fiduciary Glove Toolkit.

```
public partial class Window1 : GloveWindow
{
  public Window1(){
    this.GestureRecognizer = new GestureRecognizer();

    // Loads gestures from the GestureRepository Uri.
    this.GestureRecognizer.LoadGestures(GestureRepository);
  }
}
```

Code Example 5.15: Instantiating a Gesture Recognizer.

Subscribing to Gesture Events

Now that the gesture recognizer is set up, a list of points is sampled for every hand part in contact with the surface. Whenever the position of a hand part receives an update, it is added to the list together with a timestamp. When a hand part leaves the surface, the point array and hand part are used as parameters for the recognition method of the gesture recognizer. The point array is then compared to the postures that were loaded by the recognizer beforehand (described in detail in Chapter 3.4). After all postures are run through, the recognition method bundles the results in event arguments of the custom type `GloveGestureEventArgs`. When the method returns a GestureRecognized event is raised. This event can be subscribed to by a `GloveWindow`, `Glove`, `User`, and a `Handpart` (cf. Code Example 5.16).

```
public partial class Window1 : GloveWindow
{
  public Window1(){
     Glove glove1 = new Glove(this, "Glove1");
     Glove glove2 = new Glove(this, "SecondGlove");

     // GestureRecognized event subscription with GloveWindow.
     // The event is received for both gloves.
     this.GestureRecognized += new EventHandler<GloveGestureEventArgs↩
        >(Window1_GestureRecognized);

     // GestureRecognized event subscription with Glove.
     // The event is received only if performed by one of the hand ↩
        parts on this glove.
     glove1.GestureRecognized += new EventHandler<↩
        GloveGestureEventArgs>(glove1_GestureRecognized);
  }
}
```

Code Example 5.16: Gesture event subscription.

GloveGestureEventArgs Usage in Event Callback

The `GloveGestureEventArgs` contain the `Handpart` and a list with recognition results, each consisting of a gesture name and the corresponding recognition confidence. The list is ordered by recognition confidence. The best result from this list is exposed as separate value in the event

arguments. The event callback method receives these arguments and can be used to assign a function to a gesture and even restrict it to certain hand parts (cf. Code Example 5.17).

```
void Window1_GestureRecognized(object sender, GloveGestureEventArgs e↩
    ){
   if(e.RecognizedGesture.Identifier == "Swipe" && e.Handpart.↩
       Identifer == Handpart.Identifiers.IndexKnuckle){
     // Function for swipe with knuckle of index finger.
   }
}
```

Code Example 5.17: Gesture event callback.

5.5.5 Postures

Since neither a posture definition language nor a recognition engine could be found that met our demands, the Fiduciary Glove API features a custom solution for posture definition and recognition.

Posture Configuration Files

In contrast to the gesture recognition where the gesture definition language came with the C# library [1], a posture definition language had yet to be formulated. The definition language that is currently used in the Fiduciary Glove API follows an XML structure. A major premise for the design has been that it was easy and readable enough so that postures could be defined manually. The structure of a posture configuation file follows the DTD in Code Example 5.18.

```
<!DOCTYPE POSTURE [
  <!ELEMENT POSTURE (GLOVE|POSTUREPAIR+)>
  <!ELEMENT GLOVE (HANDPART|HANDPARTPAIR+)>
  <!ELEMENT HANDPART (#PCDATA)>

  <!ATTLIST POSTURE IDENTIFIER CDATA #REQUIRED>
  <!ATTLIST GLOVE HAND CDATA #OPTIONAL>
  <!ATTLIST POSTUREPAIR POSTURE1 CDATA #REQUIRED>
  <!ATTLIST POSTUREPAIR POSTURE2 CDATA #REQUIRED>
  <!ATTLIST POSTUREPAIR DISTANCE CDATA #OPTIONAL>
  <!ATTLIST POSTUREPAIR DISTANCETOLERANCE CDATA #OPTIONAL>
  <!ATTLIST POSTUREPAIR ANGLE CDATA #OPTIONAL>
  <!ATTLIST POSTUREPAIR ANGLETOLERANCE CDATA #OPTIONAL>
  <!ATTLIST HANDPARTPAIR HANDPART1 CDATA #REQUIRED>
  <!ATTLIST HANDPARTPAIR HANDPART2 CDATA #REQUIRED>
  <!ATTLIST HANDPARTPAIR DISTANCE CDATA #OPTIONAL>
  <!ATTLIST HANDPARTPAIR DISTANCETOLERANCE CDATA #OPTIONAL>
  <!ATTLIST HANDPARTPAIR ANGLE CDATA #OPTIONAL>
  <!ATTLIST HANDPARTPAIR ANGLETOLERANCE CDATA #OPTIONAL>
]>
```

Code Example 5.18: Posture configuration file DTD.

For both single-glove and multi-glove posture files, the root of the XML is a 'Posture' element. The sole attribute of this element is an identifier - the name of the posture. If the posture is a single-glove posture, the only child of the root element is a 'Glove' element. If the hand independence checkbox for this single-glove posture was not set, then the element has also an attribute 'Hand' that contains either the value 'Left' or 'Right'. If the single-glove posture consists

of a single hand part, then one child element 'Handpart' is added. The content of this element is the identifier of the hand part. If the single-glove posture consists of multiple hand parts, 'HandpartPair' child elements are added for every hand part connection inside the glove blob. The 'HandpartPair' element contains up to six attributes. Two attributes are mandatory: 'Handpart1' and 'Handpart2', they store the connected hand parts. The other four attributes are optional. Two attributes for angle and distance, and two for the angle and distance tolerance. If angle or distance is ignored, the corresponding attribute and its respective tolerance attribute are omitted. If the posture is a multi-glove posture, then the root contains 'PosturePair' child elements. Similar to 'HandpartPair' elements, each of these elements features between two and six attributes. The two required attributes, 'Posture1' and 'Posture2', store the single-glove postures of two of the gloves that formed the multi-glove posture. The other four attributes store angle and distance as well as the two corresponding tolerance values for the connection between the gloves. We decided to define a multi-glove posture as a conglomerate of postures for the following reasons: it reduces redundancy. Multi-glove postures are assemblies of single-glove postures. The single-glove postures can also be used as independent postures. And vice versa, if a single-glove posture already exists, it does not have to be completely modeled in a multi-glove posture file. A reference in the posture attributes of a 'PosturePair' element is enough. Additionally, it makes it easier to read the multi-glove posture files. You can clearly see how the posture is composed. Posture configuration files can be conveniently created with the Posture Configurator described in Chapter 5.4.3.

Instantiating a Posture Recognizer

After all desired postures are defined and saved to posture configuration files, the 'PostureRecognizer' property of the `GloveWindow` has to be set. For the same reason as for the gesture recognizer, it is typed with an interface: `IPostureRecognizer`. Expert developers can exchange and extend the posture recognition engine. Beginners just use the standard implementation of the interface that is included in the API: `PostureRecognizer` (cf. 5.19). After the instantiation of a posture recognizer, the postures that are to recognize have to be loaded into the recognition engine. This is done by calling the `LoadPostures` method of the gesture recognizer. The method demands a `Uri` to a folder which contains posture configuration files.

```
public partial class Window1 : GloveWindow
{
  public Window1(){
    this.PostureRecognizer = new PostureRecognizer();

    // Loads gestures from the PostureRepository Uri.
    this.PostureRecognizer.LoadPostures(PostureRepository);
  }
}
```

Code Example 5.19: Instantiating a Posture Recognizer.

Subscribing to Posture Events

Subsequently, `GloveWindow`, `Glove`, and `User` can subscribe to the posture events listed in Table 5.4:

GlovePostureEventArgs Usage in Event Callback

Each posture event affords `GlovePostureEventArgs`. The structure of these event arguments is similar to the `GloveGestureEventArgs` to advance consistency. They offer access to a list of

Posture event	Description
PostureChanged	Occurs when the attributes of the current posture change. This includes the position or orientation.
PostureDown	Occurs when a posture is recognized.
PostureUp	Occurs when a posture disappears. This means that the posture has been lifted from the surface or a different posture than the last has been recognized.

Table 5.4: Posture events.

```
public partial class Window1 : GloveWindow
{
  public Window1(){
     // PostureRecognized event subscription with GloveWindow.
     // The event is received for both gloves.
     this.PostureDown += new EventHandler<GlovePostureEventArgs>(↩
        Window1_PostureDown);
  }
}
```

Code Example 5.20: Posture event subscription.

recognition results. Each of these results consists of a posture name and a recognition confidence. The list is ordered by recognition confidence. The arguments also contain the entry with the highest confidence as separate value; it serves most commonly as actual recognition result. In contrast to the `GloveGestureEventArgs`, the posture event arguments do not include a `Handpart` or a list of `Handparts` that caused the event. Since postures are already defined for certain hand parts, this information can be retrieved through the recognized posture if needed. Each callback method of a posture event receives these arguments. The confidence value can be used as threshold for taking action on posture events. In Code Example 5.21, the fist posture triggers the corresponding function only when recognized with more than 80 percent confidence.

```
void Window1_PostureDown(object sender, GlovePostureEventArgs e){
   if(e.RecognizedPosture.Identifier == "Fist" && e.RecognizedPosture↩
      .Confidence > 0.8){
     // Function for a fist posture if it was recognized with more ↩
        than 80 percent confidence.
   }
}
```

Code Example 5.21: Posture event callback.

5.5.6 GloveButton Widget

Widgets allow a black-box usage of the toolkit functionality. Developers can use a widget without knowing how the funcitonality is achieved in the backbround. One widget is currently included in the Fiduciary Glove API. It is a button which can be restricted to certain hand parts, so that for example only the thumb can click it. As most components in the Fiduciary Glove Toolkit, the creation of a `GloveButton` follows the Create-Set-Call pattern described in Chapter 5.2; Accordingly, a `GloveButton` instance can be created with a default constructor; the text on the button is set afterwards with the string property 'Text'. Alernatively, a constructor with a single parameter

for setting the text on the button can be used for the instantiation which is provided to meet the needs of programmers with a different programming style.

```
GloveButton button = new GloveButton();
button.Text = "Press me!");

// Shortcut constructor
GloveButton buttonWithText = new GloveButton("Press me!");
```

Code Example 5.22: GloveButton instantiation.

Subscribing to GloveButton Events

A `GloveButton` can subscribe to two events: `ButtonDown` and `ButtonUp`. The first is raised when the button is pressed down, the other one when its released. When any of these events is raised, `GloveButtonEventArgs` are passed to the event handler. The corresponding callback method can retrieve the `Handpart` as well as `Glove` from these event arguments.

```
button.ButtonDown += new EventHandler<GloveButtonEventArgs>(↩
    button_ButtonDown);
```

Code Example 5.23: GloveButton event subscription.

Configuring the GloveButton Functionality

The functionality of a `GloveButton` can be configured with one property or a set of methods. As its name already suggests, the 'HandpartsNeededToPressDown' property specifies the amount of hand parts needed to press the button down. For instance, if the value is set to three, this number of hand parts have to be placed down on the button to press it down. The `RestrictToHandpart` and `RestrictToHandparts` methods enable a developer to restrict the functionality of the button to certain hand parts. However, this does not mean that the `GloveButton` is restricted to a specific `Handpart` on a `Glove` (e.g. `glove1.Thumb`) , but rather the general hand part (e.g. a thumb on a left hand). In order to combine a hand part identifier with the left or right hand information, the type `HandpartModel` can be used. It is basically a general representation of a hand part without association to a specific glove (cf. Code Example 5.24).

```
GloveButton button = new GloveButton("Only for left thumbs!");
HandpartModel leftThumb = new HandpartModel(Handpart.Identifiers.↩
    Thumb,  GloveModel.LeftOrRightHand.Left);
button.RestrictToHandpart(leftThumb);
```

Code Example 5.24: Restricting the GloveButton functionality.

5.6 Documentation

Even though self-descriptive naming reduces the amount of documentation consultations, good and complete documentation is still mandatory for an API. It helps developers to understand aspects they are not sure about and simplifies the feature exploration. The Fiduciary Glove Toolkit includes the following types of documentation.

5.6.1 Code Documentation

All publicly exposed types and members of the API are documented with a descriptive text. For each parameter of a constructor or method, its purpose is specified. For the documentation of a code block, like a method for example, XML tags are inserted in special comment fields in the source code directly before the code block. Where applicable, documentation conventions are adopted from the Surface SDK and the .Net Framework.

5.6.2 Code Snippets

In order to provide means to learn only a single feature of the API, developers can resort to a set of code snippets. Each one of these code fragments covers only a very specific aspect of the API. Once an appropriate snippet is found, developers can copy and paste it into their own code where they need the snippet's functionality. It should integrate with no or minor adjustments.

5.6.3 Tutorials

Tutorials are step-by-step descriptions of complex scenarios using several features of the API. They explicitly illustrate how multiple features can be integrated and how they fit together. Before each major step, comments anticipate its purpose.

5.6.4 Sample Applications

Also, whole sample applications are provided with the Fiduciary Glove Toolkit. Developers can run the source code and even use it as a basis for their applications. Also, two of the utilities of the Fiduciary Glove Toolkit make use of the API and can therefore serve as examples.

5.6.5 Utility Manuals

A collection of manuals explains the purpose and usage of all utilities provided with the Fiduciary Glove Toolkit.

5.6.6 Build-a-Glove Guide

In order to work perfectly, tags should be stuck onto the glove at an exact position and orientation. A guide specifies the layout a new Fiduciary Glove should conform to. In addition the guide recommends glove and tag materials.

5.7 IDE Integration

Next to good and extensive documentation, a seamless IDE integration helps developers to get started and get to know the functionality of the Fiduciary Glove Toolkit. An install wizard sets the scene for the developer. First, it creates the folder structure. This includes folders for the utilities and the configuration files for gloves, postures and gestures.

5.7.1 Fiduciary Glove Application Template

New applications for the Fiduciary Glove can be created effortlessly using the application template that comes with the toolkit. The Fiduciary Glove Application Template can be picked directly from within Visual Studio, just like any other preinstalled template. A custom template icon showing the Fiduciary Glove makes it easily distinguishable from other custom templates. A template is basically a code skeleton with just the bare essentials needed to get going. The Fiduciary Glove Application template inherits from `GloveWindow`. It creates a `Glove` instance for every

`GloveModel` defined by a glove configuration file. Events are therefore received whenever any of the gloves touches the surface. An exemplary event is inserted at the end of the constructor to illustrate how to access basic hand part information. When opening a template, all needed references to external libraries as the Fiduciary Glove API are already included.

5.7.2 Visual Designer Integration

Since `Glove`, `User` and `GloveButton` derive from `UserControl` (cf. subchapters Gloves and Users), they can be integrated into the toolbox of the Visual Designer. This makes them very easy to use. The controls can be easily added to projects via drag-and-drop. Also, their properties can be set in the Visual Designer, and can even be bound to data through the usage of corresponding DependencyProperties. Further convenience features like automated event handler creation speed up the development process, particularly for average skilled programmers.

6 Sample Applications

This chapter illustrates two sample applications that were developed with the help of the Fiduciary Glove Toolkit. The first serves as example for a complex application that uses all features of the Fiduciary Glove Toolkit. Since it is a reimplementation of an existing application, it gives a first impression of how the toolkit can raise productivity. The second sample application focuses on the usage of postures, and can serve as simple means to evaluate postures playfully.

6.1 Finger Painting Application

The old version of the finger painting application (cf. Chapter 4.3) was implemented before the idea for the Fiduciary Glove Toolkit even existed. The new version will reimplement most of the features of the old version with the help of the Fiduciary Glove Toolkit.

6.1.1 Glove, Gesture and Posture Creation with Utilities

For reimplementing the finger painting application with the help of the toolkit, the needed gloves, gestures and postures had to be created first. While this had happened in the old version manually and in code, it is now conveniently done with the utilities that come with the toolkit. For each glove we created a glove configuration file with the Glove Configurator. The fist posture which is used for clearing the canvas was created with the Posture Configurator; and the Gesture Configurator allowed us to create the swipe posture, and even train it for better recognition.

6.1.2 Creating a New Project

For the old version, we basically had to start from scratch. As basis served a project template the Surface SDK provided, which did not include one line of code about hand parts, postures and gestures, of course. For our reimplementation we used the Fiduciary Glove Application Template that is installed with the toolkit. It sets up basic code that helps developers to get an idea of how hand parts, gestures and postures are integrated into a custom application. Also, required references are already added.

6.1.3 Implementation with the Fiduciary Glove API

The retrieval of hand part, gesture and posture information had been a cumbersome and complicated task without the Fiduciary Glove API. With it, accessing hand parts, gestures and postures was rather easy. The first step was to load the gloves. For each glove configuration a glove object was initialized. Each of these glove instances was assigned to a user. This was done by setting the 'User' property of a glove. Then, recognizers for gestures and postures had to be assigned and the defined gestures and postures loaded. The code that was required for this was already included in the template.

Reimplementing Separate Functions for Separate Handparts

Now that the basic setup had been done, the first real feature could be implemented - separate functions for separate hand parts. As before, each finger paints in its own color and thickness. Therefore, the `GloveWindow` subscribed to the `FingerDown`, `FingerChanged`, and `FingerLeave` events. In the `FingerDown` event a path with the finger's color and thickness is started at the position of the finger. In the `FingerChanged` event, this path is continued; and in the `FingerLeave` event the path is closed and merged with the canvas. The erasing function is assigned to the knuckles in a similar fashion; Therefor, the `KnuckleDown`, `KnuckleChanged` and `KnuckleLeave` events are subscribed to. The erasing is done similarly to the painting function, but in this case, the paths

of all knuckles have the same color - the background color of the `GloveWindow`. The choice of finger and knuckle events restricted the painting function to fingers and the erasing function to knuckles. So far, no other hand part had a function assiged to it.

Reimplementing the Gesture Function

The swipe with the back of the hand saves the canvas. In the old version it was implemented in the following fashion: The points for each touching tag had to be stored manually on each position change, and when the tag left the surface passed to the gesture recognizer. Additionally, the gesture recognizer had to be adjusted to be able to tell which hand part performed the gesture. In the new version the `GloveWindow` simply subscribed to a `GestureRecognized` event which is raised when a gesture is recognized. In the corresponding callback method, it is checked whether or not the posture was a swipe and had been recognized with a certain confidence and performed with the back of the hand. If so, the canvas is flattened and saved to a png file in the application folder.

Reimplementing the Posture Function

The fist posture clears the canvas. If more than one person participated in the painting acitivity, all have to place down a fist on the surface for clearing.

While postures were hard-coded in the old version and therefore difficult to maintain and extend, we could access them effortlessly with the Fiduciary Glove Toolkit. Because the application should react when the fist posture is placed down on the surface, the `GloveWindow` subscribed to the `PostureDown` event. The callback method checks if the recognized posture was a fist. If so, it determines the user of the glove that performed the posture. Then, a Boolean value is set for the user in a dictionary that specifies that the user is currently performing a fist posture. With the dictionary could be determined if all people participating in the painting had their fist on the surface at the current moment. If so, the canvas clearing function was called.

6.1.4 Omissions in the New Version

Before the old and new version are compared, omissions of the new version are illustrated. Neither the palette nor the color selection cube were reimplemented. The palette would have been implemented in the same way as before, so it would not prove much. The color selection cube would have been a nice addition, but our toolkit does not support the integration of tangible objects. This is one of the areas of future work though. Another feature that has not been implemented is the copy and paste function with multiple gloves. This is due to the fact that I had to return from Calgary and could not get access to a Microsoft Surface in Germany.

6.2 Posture Trainer

The second application that was programmed with toolkit support was the Posture Trainer. Its purpose is to train single-glove postures in a playful and competitive way. After setting the points that are needed to win and number of players on the start screen (cf. Figure 6.1), the view changes to the game screen. One after another, people register as players by placing down a hand part of their glove (cf. Figure 6.2). After all players are set, the game starts. A posture is randomly picked from the posture repostory. On all sides of the screen the name of the posture is displayed (cf. Figure 6.3). A countdown timer in the center of the screen shows the time that is left to perform the posture. The player who performs the right posture first gets a point (cf. Figures 6.4 and 6.5). Either then, or when the countdown is over, the next random posture name is displayed. When a player reaches the point limit that was set on the start screen, the game is over and the player wins.

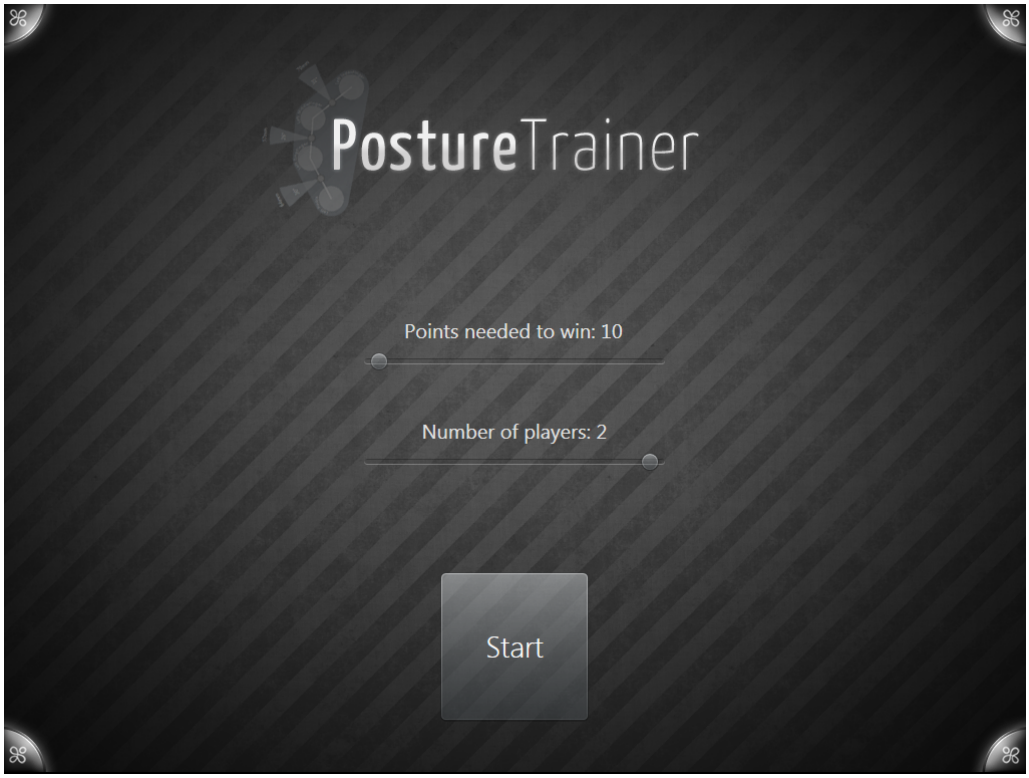

Figure 6.1: PostureTrainer start screen.

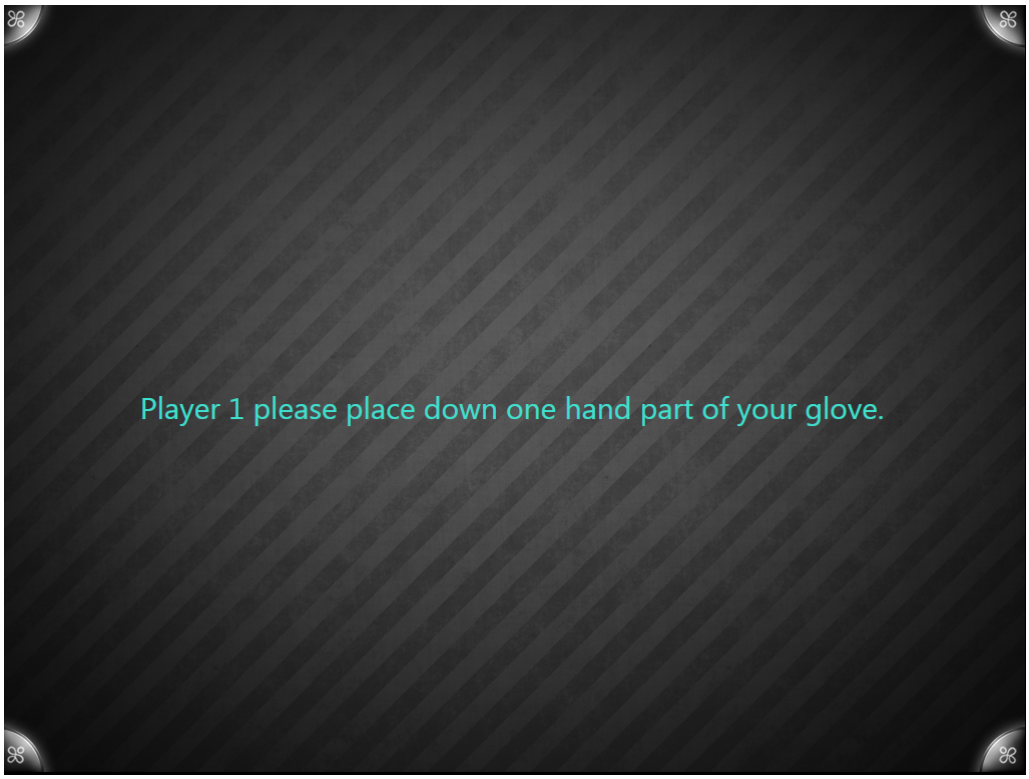

Figure 6.2: Player registration prompt.

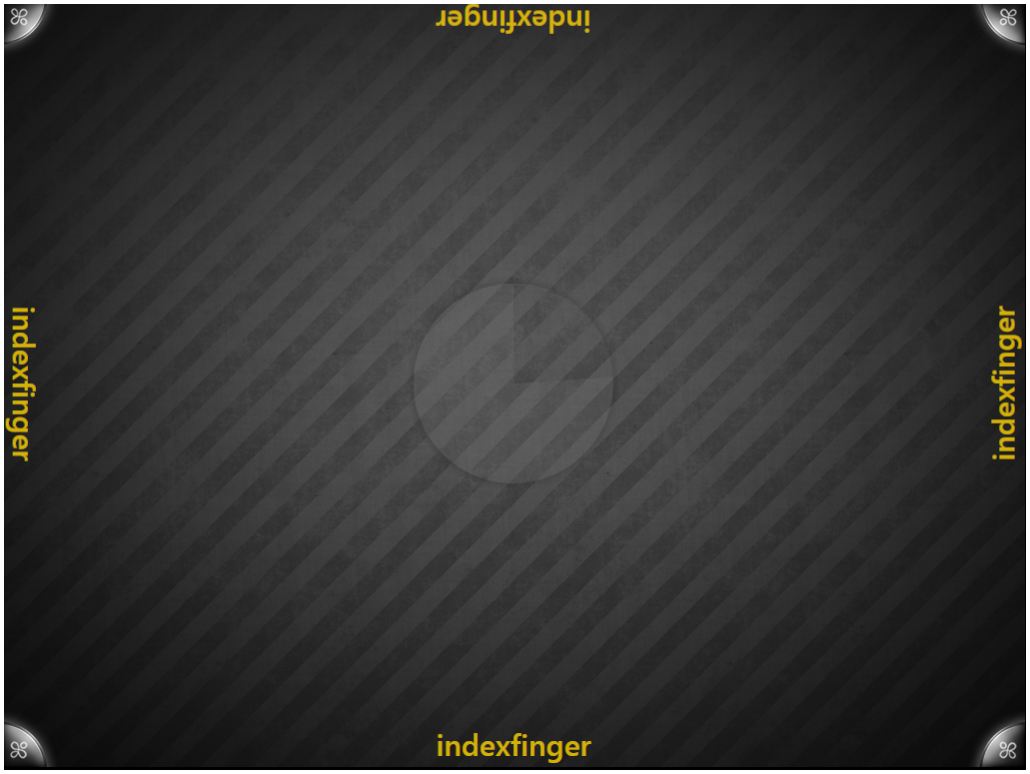

Figure 6.3: PostureTrainer game screen.

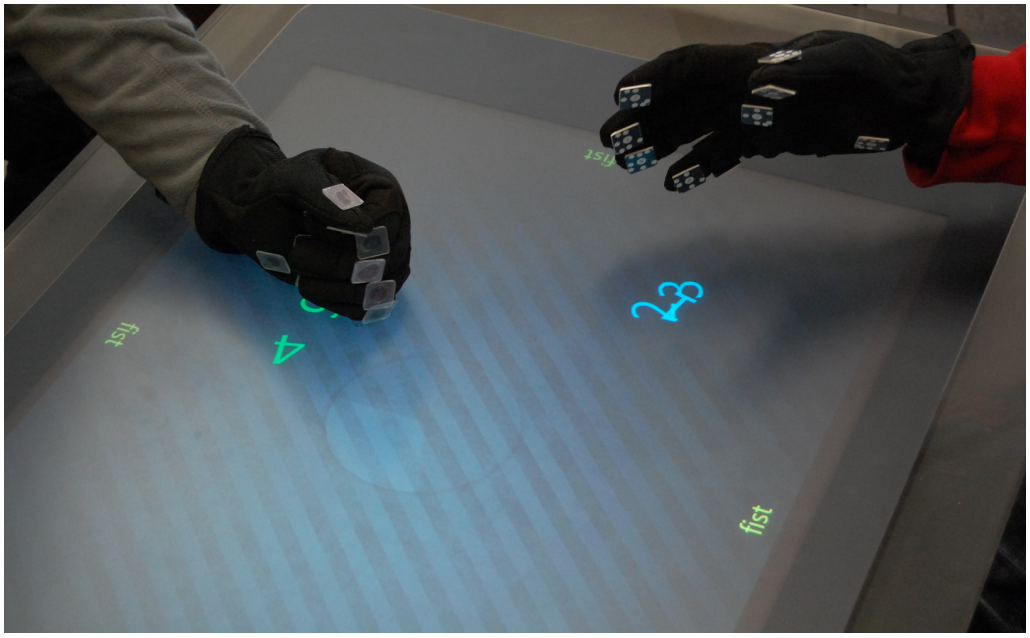

Figure 6.4: Players trying to perform the right gesture.

Figure 6.5: Player 1 (right) scores by performing the right posture.

6.2.1 Creation of Postures

A rich set of postures is crucial for this application. Therefore, several single-glove posture were created with the Posture Configurator. The built-in posture recognition helped to create postures that are distinct enough to allow for a reliable recognition.

6.2.2 Creating a New Project

This project was created with the Fiduciary Glove Application template. Unnecessary parts like the gesture recognition code and the hand part event subscription had been removed.

6.2.3 Implementation with the Fiduciary Glove Toolkit

The start screen was implemented solely with Surface components, which is still possible since the Fiduciary Glove API is built on top of the Surface SDK. When starting the game, two events from the Fiduciary Glove API are subscribed to: The `HandpartDown` event for the player registration and the `PostureDown` event for the actual gameplay. In the beginning the game screen prompts player after player to register by placing down one handpart of the glove they play with. This is where the `HandpartDown` event is raised. In the callback method the glove of the handpart was determined and associated to the `User` instance that represents the player that was currently prompted to register. After all players had registered, the `GloveWindow` unsubscribed from the `HandpartDown` event, and subscribed to the `PostureDown` event instead. Subscribing with the `GloveWindow` meant that `PostureDown` events were received for all gloves. In the callback method, the name of the recognized posture was checked against the posture name currently displayed at the sides of the screen. If the names matched, a point was awarded to the user of the glove that performed the posture. At the same time, additional information provided by the posture event arguments were leveraged to give the players visual feedback about the current points of each player (cf. Code Example 6.1).

```
private void GameWindow_PostureDown(object sender, ↩
    GlovePostureEventArgs e)
{
  if (e.RecognizedPosture.Identifier.Equals(this.WantedPosture.↩
      Identifier))
  {
     // Get scoring player
     User scoringPlayer = e.Glove.User;

     // Add a point to player's score
     PlayerPoints[scoringPlayer] += 1;

     // Check if player has won, save time to log, etc.
     [...]
  }
}
```

Code Example 6.1: Posture Trainer callback method.

From the posture event arguments could also be retrieved where on screen the posture happened. At this position and in the orientation of the posture, the updated total points of the scoring player were drawn in the player color (cf. Figure 6.5). Subsequently the old posture name was replaced with the name of a new posture that was randomly picked from the posture pool. When the points of a player reached the preset point limit, the game stopped, and the `GloveWindow` also unsubscribed from the `PostureDown` event, so no posture events would be received anymore. In an overlay statistics were presented. The points of each player are shown in a bar chart in the centre of the screen. By pressing one of two buttons the players could then choose to play again or go back to the start screen.

6.2.4 Usage as Posture Evaluation Utility

After implemententing the Posture Trainer it became clear that it could be also used for quick and rough evaluations of postures. If a certain posture could seldomly - or never - be performed by the players in the time frame of the countdown, the usability of this postures has to be questioned. For this reason, a log function was integrated. Every time a posture was successfully performed, the posture name and the needed time is recorded. If the posture couldn't be performed before the countdown was over, the posture name and a flag for not completion is marked. Next to the total and average time of all successful performances, the number of successful and failed performances as well as the time of the quickest and slowest performance is stored to a log file for each posture that was trained in the current session. The log files of several session give an impression of which postures can be performed very well and which can not.

6.2.5 Limitations

A prerequisite for the Posture Trainer to work is that the posture names are rather self-descriptive, because the application does not offer any other visual representation (e.g. a photo) than the posture name. Even though the Posture Configurator stores a screenshot for every posture created with it, we refrained from using them. The screenshots do not show the actual posture, but the glove blobs that the Posture Configurator draws as visual representation and control interface. Outside this utility, the screenshots don't carry much meaning, since the glove blobs can't be understood without the context.

7 Conclusion

The Fiduciary Glove Toolkit enables developers to integrate expressive interaction with all parts of the hand, enriched with gestures and postures in custom applications. The Fiduciary Glove was the basis therefor. Chapter 1 gave an overview over the research project of this thesis. Then related work and the technological background for the Fiduciary Glove as well as the toolkit were investigated. In Chapter 4 the Fiduciary Glove and the first application leveraging its capabilities were described. Chapter 5 started with a definition of the term toolkit and a set of design guidelines and strategies for API and toolkit design that had been derived from related work. Then, a short walkthrough guided through the development of a small application example using the Fiduciary Glove Toolkit. Subsequently, the utilities and API of the toolkit were introduced. Sections about documentation and IDE integration complemented this chapter. In Chapter 6 two sample application that have been developed with the Fiduciary Glove Toolkit were highlighted.

This final chapter presents areas of future work as well as the contributions of this thesis. The thesis is concluded by closing words.

7.1 Future Work

The major goal of instrumenting developers with a tool for integrating hand parts, gestures and postures in their own applications, has been achieved. However, the presented work also opens up various paths for future work.

7.1.1 Exploration of Expressive Interaction

Now that rapid prototyping is possible with the Fiduciary Glove Toolkit, the expressive interaction the Fiduciary Glove affords can be explored. There are several questions to be answered: How and how much can we leverage the knowledge about which hand parts caused which touch? Or which hand part performed which gesture or posture? Basically: how far can we go with overloading interaction with hand part information? Mapping the space of expressive interaction the Fiduciary Glove affords could result in a topology of hand parts, gestures and postures that work and also work together.

7.1.2 Toolkit Extension

Even though the Fiduciary Glove Toolkit already implemented many of the features we had in mind, there is still room for improvement.

Various Input Providers

So far, our toolkit receives its input only from the Surface SDK . For the future, the toolkit should be extended so that other hardware and software can operate as input providers. This capability would prepare the Fiduciary Glove Toolkit especially for future approaches that don't require a glove for the hand part identification. The existing applications that have been explored and refined by then could be instantly used on these devices.

Custom Gloves and Handparts

In the current version, the toolkit does not support different glove sizes very well. While the basic hand part recognition works just as well for gloves with different sizes, the Posture Configurator and posture recognition engine work only with a standard size glove. Postures defined with a glove can only be recognized when performed with a glove of the same size, since the distances and angles between hand parts are crucial for the recognition. This problem could be partly solved

by redesigning the GLove Configurator. A mechanism for resizing the depicted hands would allow to define gloves of various sizes; the resize factor could be stored in the glove configuration file and used to recognize postures that were defined by a different sized glove.

Also, the hand parts that can be accessed via the toolkit are fixed. Developers can only use the 15 key hand parts we've identified. In the future it should be possible for developers to attach fiduciary tags on other hand parts, or even parts of the body by themselves. In the best case, the access of these new parts should be as convenient as it is for the standard hand parts. In an early version, we've tried to incorporate this feature by using placeholders in the enumeration for hand part identifiers. After reading in one of the API design guidelines that placeholders have to be avoided, we refrained from this decision and postponed it for a later version of the Fiduciary Glove Toolkit.

Widget Development

The GloveButton demonstrates nicely how features of the Fiduciary Glove Toolkit can be provided as a black box. It hides all unnecessary details from the developer, but can still be customized easily. It exposes its capabilities through properties, that can change its behaviour and appearance in various ways. This flexibility makes it a powerful, yet handy tool to enrich custom applications quickly with hand part information. Unfortunately there hasn't been enough time for the development of more widgets. The first step will be to port existing controls, like sliders and text boxes, to the Fiduciary Glove Toolkit.

Tangible Object Integration

In the first version of the finger painting application, we illustrated how the Fiduciary Glove could interact with a tangible object - the color selection cube (cf. Chapter 4.3. Since the Fiduciary Glove API is based on the Surface SDK, it is possible to integrate tangibles in the same way - via the tag on the tangible. In my opinion, this is not optimal. Ironically, the tangible object is not tangible in code - it is detached from the real life object and represented by a tag. The Fiduciary Glove Toolkit could provide mechansims to integrate custom tangibles and then expose them on a higher level, similar to the Fiduciary Glove.

Combining Gestures and Postures

In order to fully leverage the possibilities that arise from the integration of gestures and postures, the Fiduciary Glove Toolkit should be extended so that it supports the combined usage of both. Whereas other approaches suggest separate multi-finger gestures, we argue that the same result can be achieved with a combination of gestures and postures. A swipe with four fingers for example could be assembled from a swipe gesture and a four finger posture.

7.1.3 Toolkit Usability Evaluation

Unfortunately, time did not permit to conduct an extensive usability evaluation of the Fiduciary Glove Toolkit besides several code and concept walkthroughs with my supervisors and the rough benchmark through the reimplementation of the finger painting appication. An evaluation of API and utilities could include a comparison of developer expectations and the actual features, which could reveal mismatches between the mental model and implemented concept. Also, the performance of different developer types could be analysed and compared. This would supply helpful feedback on how to make the toolkit as accessible as possible for different kinds of developers and how to find an optimal compromise in controversial aspects.

7.2 Contributions

Next I will summarize the main contributions of this thesis project. The first section illustrates the contributions that were achieved through the Fiduciary Glove. The second section covers the contributions through the Fiduciary Glove Toolkit.

7.2.1 Fiduciary Glove

With the Fiduciary Glove I've presented a method for the identification of hand parts, hands (or gloves, respectively) and users. It can be built from an ordinary glove and a set of fiduciary markers. Its reliability distinguishes this approach from others. Knowing exactly and for sure which hand part caused which touch allows for expressive touch interaction. Since hand parts, gloves and users are associated, the identification of gloves and users is just as reliable. Although all this requires people to wear gloves, it allows to explore meaningful interfaces well before research solves the problem of accurate barehand touch identification.

7.2.2 Fiduciary Glove Toolkit

The Fiduciary Glove Toolkit instruments developers with tools to fully leverage the capabilities of the Fiduciary Glove. An API allow an easy and at the same time powerful development of applications that incorporate rich interaction with various hand parts. Multi-user scenarios are just as feasible as the integration of gestures and postures. The required gesture and posture recognition engines are already built-in. Events provide an easy and familiar way to assign functions to hand parts, gestures and postures. The API is complemented with a set of three utilities: Glove, Gesture and Posture Configurator. The Glove Configurator allows developers to easily prepare a glove to be accessed via the API. With a comprehendible interface, glove configuration files can be created intuitively. The second utility, the Gesture Configurator, lets developers define, train and test gestures. Gestures become visible and tangible through a path that is drawn while the gesture is traced. A parallel gesture recognition prevents developers to define duplicate or intefering gestures. The Posture Configurator poses a utility that makes an erstwhile complex task rather simple. A posture is defined by simply placing it down on the surface. The posture being defined is concurrently analysed by the posture recognizer and matched against existing postures. The recognition result helps to identify possible interferences quickly.

An extensive documentation helps developers to get started with the API and the utilities. It ranges from small code snippets over step-by-step tutorials to whole sample applications. Manuals for the utilities and a guide illustrating how to build a Fiduciary Glove complement the documentation.

7.3 Closing Words

I strongly believe that future approaches will allow the expressive interaction with all parts of the hand without the Fiduciary Glove. Until then, I invite the readers of this thesis to try out the Fiduciary Glove Toolkit by themselves and leverage the human hand's potential as expressive input device in their own applications. You have the appropriate tools at your fingertips:

http://grouplab.cpsc.ucalgary.ca/cookbook/

References

[1] $1 Recognizer C# library, http://depts.washington.edu/aimgroup/proj/dollar/dollar.zip.

[2] Microsoft Surface 1.0, http://www.microsoft.com/surface.

[3] Microsoft Surface SDK, http://www.microsoft.com/downloads/en/details.aspx?familyid=3db8987b-47c8-46ca-aafb-9c3b36f43bcc&displaylang=en.

[4] MSN Encarta, http://encarta.msn.com/dictionary_561534184/toolkit.html.

[5] PyMT, http://pymt.eu.

[6] Smart SDK, https://smarttech.com/us/support/browse+support/download+software.

[7] Smart Table, http://smarttech.com/table.

[8] O. Bau and W. E. Mackay. OctoPocus: a dynamic guide for learning gesture-based command sets. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 37–46, 2008.

[9] H. Benko, T. S. Saponas, D. Morris, and D. Tan. Enhancing input on and above the interactive surface with muscle sensing. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 93–100, New York, NY, USA, 2009. ACM.

[10] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[11] V. Buchmann, S. Violich, M. Billinghurst, and A. Cockburn. FingARtips: gesture based direct manipulation in augmented reality. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, page 221, 2004.

[12] G. Butler and P. Denomee. Documenting frameworks to assist application developers. *Object-Oriented Application Frameworks. M. Fayad, D. Schmidt, R. Johnson (eds.) John Wiley and Sons, NY*, 1999.

[13] S. Clarke. Measuring API usability. *DOCTOR DOBBS JOURNAL*, 29(5):1–5, 2004.

[14] K. Cwalina and B. Abrams. *Framework design guidelines*. Addison-Wesley, 2005.

[15] C. T. Dang, M. Straub, and E. Andre. Hand distinction for multi-touch tabletop interaction. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces - ITS '09*, page 101, Banff, Alberta, Canada, 2009.

[16] R. A. Diaz-Marino, E. Tse, and S. Greenberg. Programming for multiple touches and multiple users: A toolkit for the DiamondTouch hardware. In *UIST 2003 Conference Companion*, 2003.

[17] P. Dietz and D. Leigh. DiamondTouch: a multi-user touch technology. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, page 226, 2001.

[18] B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[19] D. Freeman, H. Benko, M. R. Morris, and D. Wigdor. Shadowguides: visualizations for in-situ learning of multi-touch and whole-hand gestures. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 165–172, New York, NY, USA, 2009. ACM.

[20] J. D. Gould. How to design usable systems. In R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg, editors, *Human-computer interaction*, pages 93–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[21] S. Greenberg. Toolkits and interface creativity. *Multimedia Tools and Applications*, 32:139–159, 2007.

[22] J. V. Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software: Practice and Experience*, 31(3):277–300, 2001.

[23] T. Hansen, C. Denter, and M. Virbel. Using the PyMT toolkit for HCI Research.

[24] M. Henning. API design matters. *Queue*, 5(4):36, 2007.

[25] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, UIST '90, pages 112–122, New York, NY, USA, 1990. ACM.

[26] C. Holz and P. Baudisch. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 581–590, New York, NY, USA, 2010. ACM.

[27] R. E. Johnson. Documenting frameworks using patterns. In *conference proceedings on Object-oriented programming systems, languages, and applications*, page 76, 1992.

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

[29] S. Khandkar and F. Maurer. A Domain Specific Language to Define Gestures for Multi-Touch Applications.

[30] S. Khandkar, S. Sohan, J. Sillito, and F. Maurer. Tool support for testing complex multi-touch gestures. In *ACM International Conference on Interactive Tabletops and Surfaces*, pages 59–68. ACM, 2010.

[31] S. R. Klemmer, J. Li, J. Lin, and J. A. Landay. Papier-Mache: toolkit support for tangible input. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 406, 2004.

[32] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 199–206, 2004.

[33] M. Krueger, T. Gionfriddo, and K. Hinrichsen. VIDEOPLACE - an artificial reality. *ACM SIGCHI Bulletin*, 16(4):35–40, 1985.

[34] J. A. Landay and B. A. Myers. Extending an existing user interface toolkit to support gesture recognition. In *INTERACT '93 and CHI '93 conference companion on Human factors in computing systems*, CHI '93, pages 91–92, New York, NY, USA, 1993. ACM.

[35] J. Letessier and F. Berard. Visual tracking of bare fingers for interactive surfaces. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 119–122, 2004.

[36] K. Lyons, H. Brashear, T. Westeyn, J. Kim, and T. Starner. Gart: The gesture and activity recognition toolkit. In *Proceedings of the 12th international conference on Human-computer interaction: intelligent multimodal interaction environments*, pages 718–727. Springer-Verlag, 2007.

[37] S. Malik. *An Exploration of Multi-finger Interaction on Multi-touch Surfaces. University of Toronto*. PhD thesis, 2007.

[38] S. Malik, A. Ranjan, and R. Balakrishnan. Interacting with large displays from a distance with vision-tracked multi-finger gestural input. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 43–52. ACM, 2005.

[39] N. Marquardt, J. Kiemer, and S. Greenberg. What caused that touch?: expressive interaction with a surface through fiduciary-tagged gloves. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 139–142, New York, NY, USA, 2010. ACM.

[40] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, March 2000.

[41] K. Oka, Y. Sato, and H. Koike. Real-time fingertip tracking and gesture recognition. *IEEE Computer Graphics and Applications*, 22:64–71, 2002.

[42] J. Rekimoto. Smartskin: an infrastructure for freehand manipulation on interactive surfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, pages 113–120, New York, NY, USA, 2002. ACM.

[43] M. Resnik, B. Myers, K. Nakakoji, B. Shneiderman, R. Pausch, and M. Eisenberg. http://www.cs.umd.edu/hcil/cst/papers/designprinciples.htm.

[44] M. P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, pages 27–34, 2009.

[45] D. Rubine. Specifying gestures by example. *ACM SIGGRAPH computer graphics*, 25(4):329–337, 1991.

[46] M. Schlattmann and R. Klein. Simultaneous 4 gestures 6 dof real-time two-hand tracking without any markers. In *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, VRST '07, pages 39–42, New York, NY, USA, 2007. ACM.

[47] C. Shen, F. D. Vernier, C. Forlines, and M. Ringel. Diamondspin: an extensible toolkit for around-the-table interaction. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 167–174, New York, NY, USA, 2004. ACM.

[48] D. J. Sturman and D. Zeltzer. A Survey of Glove-based Input. *IEEE Comput. Graph. Appl.*, 14(1):30–39, 1994.

[49] J. Stylos and B. A. Myers. Mapping the space of API design decisions. 2007.

[50] J. Stylos and B. A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112, 2008.

[51] R. Y. Wang and J. Popović. Real-time hand-tracking with a color glove. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 63:1–63:8, New York, NY, USA, 2009. ACM.

[52] J. O. Wobbrock, M. R. Morris, and A. D. Wilson. User-defined gestures for surface computing. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 1083–1092, New York, NY, USA, 2009. ACM.

[53] J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, page 168, 2007.

[54] M. Wu and R. Balakrishnan. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. In *Proceedings of the 16th annual ACM symposium on User interface software and technology*, UIST '03, pages 193–202, New York, NY, USA, 2003. ACM.

[55] Z. Zhang, Y. Wu, Y. Shan, and S. Shafer. Visual panel: virtual mouse, keyboard and 3d controller with an ordinary piece of paper. In *Proceedings of the 2001 workshop on Perceptive user interfaces*, PUI '01, pages 1–8, New York, NY, USA, 2001. ACM.

[56] M. F. Zibran. What makes APIs difficult to use? *IJCSNS*, 8(4):255, 2008.

List of Figures

List of Code Examples

List of Tables

A Glossary

Abbreviation	Description
API	Application Programming Interface
DLP	Digital Light Processing
DTD	Document Type Definition
IR	Infrared
LED	Light Emitting Diode
SDK	Software Development Kit
WPF	Windows Presentation Foundation
XML	eXtensible Markup Language
XNA	Not acronymed, game programming technology

B DVD Content

This is a list of folders and files on the DVD attached to every copy of this thesis.

- **Documentation/**: code snippets, tutorials and utility manuals
  - **Fiduciary Glove API Documentation/**: interactive API documentation
- **Literature/**: pdf files of related work
- **Media/**: media that has been produced for the thesis
  - **Photos/**: photos of the Fiduciary Glove, the utilities and applications
  - **Screenshots/**: screenshots of the utilities and applications
  - **Videos/**: videos of applications
- **Presentations/**: thesis project presentations
  - **Exit Talk Calgary 122110/**: exit talk at the University of Calgary, 12/21/2010
  - **Presentation Munich 020811/**: thesis presentation at the LMU Munich, 02/08/2011
- **Software/**: software that has been produced for the thesis
  - **Fiduciary Glove Application Project Template/**: Visual Studio project template for an application for the Fiduciary Glove
  - **Fiduciary Glove Toolkit API/**: API of the Fiduciary Glove Toolkit
  - **Fiduciary Glove Toolkit Installer/**: installer of the Fiduciary Glove Toolkit
  - **Sample Applications/**: sample applications produced for the thesis
    - **Finger Painting Application/**: finger painting application developed with the toolkit (cf. Chapter 6.1)
    - **Finger Painting Application without Toolkit/**: finger painting application developed with the toolkit (cf. Chapter 4.3)
    - **PostureTrainer/**: posture training application developed with the toolkit (cf. Chapter 6.2)
  - **Utilities/**: utilities of the Fiduciary Glove Toolkit
    - **GestureConfigurator/**: Gesture Configurator from Chapter 5.4.2
    - **GloveConfigurator/**: Glove Configurator from Chapter 5.4.1
    - **PostureConfigurator/**: Posture Configurator from Chapter 5.4.3
- **Thesis/**: latex sources and images of the thesis document
- **DiplomaThesis_JohannesKiemer.pdf**: complete thesis document in pdf format