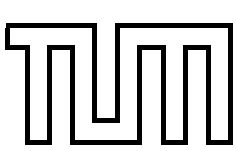
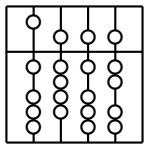




Tangible Information Displays

Florian Echtler

TECHNISCHE UNIVERSITÄT MÜNCHEN

Institut für Informatik, Lehrstuhl I16

Tangible Information Displays

Florian Echtler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Rüdiger Westermann
Prüfer der Dissertation:	1. Univ.-Prof. Gudrun J. Klinker, Ph.D.
	2. Univ.-Prof. Dr. Andreas Butz, Ludwig-Maximilians-Universität München

Die Dissertation wurde am 8.7.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.11.2009 angenommen.

To my parents, without whom this thesis would never have started.
To Andrea, without whom it would have ended in the lunatic asylum.

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist es, eine generische Softwarearchitektur für Multi-Touch und Multi-User Interfaces vorzustellen.

In den letzten Jahren hat sich die Forschung im Bereich neuartiger Benutzerschnittstellen stetig intensiviert - insbesondere Multi-Touch und Multi-User Interfaces finden immer mehr Beachtung. Ein Grund hierfür ist die zunehmende Verfügbarkeit praktisch einsetzbarer, erschwinglicher Eingabegeräte.

Aufgrund dieser gestiegenen Verbreitung wurde in wenigen Jahren eine beachtliche Anzahl verschiedenster Anwendungen für diese neuartigen Eingabegeräte entwickelt. Vom Standpunkt eines Softwareentwicklers aus betrachtet weisen diese Anwendungen jedoch einige Nachteile auf. Beispielsweise sind die meisten dieser Systeme monolithisch und machen es daher schwierig, ihren Quellcode wiederzuverwenden. Auch müssen gewisse Kernfunktionen wie z.B. Gestenerkennung immer wieder neu implementiert werden. Zuletzt sind diese Anwendungen häufig auf eine bestimmte Art von Eingabehardware zugeschnitten und können nicht ohne weiteres mit einem anderen Gerät benutzt werden.

Um diesen Einschränkungen entgegenzuwirken, wurde in dieser Arbeit eine Softwarearchitektur entworfen, die es ermöglichen soll, beliebige interaktive Anwendungen zu modellieren. Auch wurde als Teil dieser Architektur eine formale Beschreibung für Gesten entwickelt.

Eine Referenzimplementierung für diese Architektur ist *libTISCH*. Ein Entwickler, der libTISCH benutzt, soll in der Regel nicht mehr Zeit für die Entwicklung einer neuartigen interaktiven Anwendung benötigen als für ein konventionelles grafisches Benutzerinterface. Dasselbe gilt für die Integration neuer Sensorhardware - bestehende Anwendungen sollen ohne weitere Modifikation verwendbar sein, sofern ein passender Treiber zur Verfügung steht. Um die Eignung von libTISCH für diese Aufgaben zu untersuchen, wurden mehrere Anwendungen entwickelt und auf verschiedener Sensorhardware getestet. Die erzielten Ergebnisse belegen die angestrebte Funktionalität im Hinblick auf Anwendungsentwicklung und Hardwareintegration.

Abstract

The goal of this thesis is to provide a generic architecture and software framework for graphical multi-touch and multi-user interfaces.

In recent years, research in novel types of computer-human interaction has increased considerably. Particularly multi-touch and multi-user interfaces have received a lot of interest, partly due to the availability of robust and affordable sensor hardware. This trend has been accelerated by the emergence of commercial products which have made these interaction concepts available to a wide user base in a surprisingly short timeframe.

Although a considerable amount of useful applications has already been written based on these new modalities, they share some deficiencies from a developer's point of view. Even when source code is available, most of these applications are written in a monolithic fashion, making reuse of code difficult. Furthermore, they duplicate large amounts of core functionality such as gesture recognition and are often locked to a single type of input hardware.

To address this lack of reusability and portability, a layered architecture is presented in this thesis to describe an interactive application in a generalised fashion. As part of this architecture, a formal description of gestures will also be specified.

A reference implementation of this architecture, *libTISCH*, is presented. When using this framework, a developer should not require more time for creating a novel user interface than for a conventional one. The same applies to integration of new types of input hardware - existing software should "just work" after a suitable adapter has been provided. A number of example applications have been created with libTISCH and tested on various input sensors. The results show the suitability of libTISCH for the intended tasks regarding software development and hardware integration.

Acknowledgements

Writing a thesis thankfully requires no blood, but a significant amount of sweat and at least some tears (metaphorically speaking). Therefore, I would like to thank all those people who have helped me during these nearly four years (i.e., 3 years, 5 months and one really nasty week).

First of all, many, many thanks go to my advisor, Prof. Gudrun Klinker, for limitless support, advice and help in pursuing this thesis. More encouragement towards and freedom in choosing a research area can hardly be imagined. I would also like to thank my second advisor, Prof. Andreas Butz, for his thought-provoking comments and many invaluable last-minute tips.

Special thanks go to my colleagues Manuel Huber and Marcus Tönnis who were always ready to discuss and help with any electronic or mechanic issues, particularly the really esoteric ones. I am also much obliged to Peter Keitler, Patrick Maier, Simon Nestler, Daniel Pustka, Michael Schlegel and Björn Schwerdtfeger for being an amazing team to work with.

Let me express my gratitude towards all those students who have contributed to this thesis: Andreas Dippon, Nikolas Dörfler, Thomas Pototschnig, Martin Weinand, Franziskus Karsunke and Amir Beshay. I am also indebted to all those who sacrificed their time to read draft versions of this thesis and point out some of the innumerable mistakes: Gudrun Klinker, Andreas Butz, Marcus Tönnis, Heike Kreitmaier, Matthias Rahlf, Andreas Dippon, Andrea Echtler and especially Chris Hodges.

Finally, I would like to thank those people in my immediate vicinity who had to endure a rich variety of "thesis moods", but nevertheless helped me in ways I could not have imagined: Sylvia & Ernst Echtler, Carsten Dlugosch and many others. Last but not least, I would like to thank Andrea Echtler for her unending patience and subtle ways in steering me back towards a finished thesis with my sanity still largely intact.

This thesis was supported by the *Bayerische Forschungsstiftung* within the scope of the "TrackFrame" project and by the *Europäische Forschungsgesellschaft für Blechverarbeitung* through the "Kopiertreiben" project.

This thesis has entirely been created with open-source[1] software such as LaTeX, Evince, `make`, GIMP and Inkscape. The sole exception is figure 4.4, which was drawn using the Eagle freeware[2] edition.

All pictures were created by the author unless noted otherwise. British spelling is used throughout this document.

[1] free-as-in-speech
[2] free-as-in-beer

Contents

1 Introduction **1**

1.1 Motivation . . . 2
1.2 Challenges . . . 4
1.3 Related Areas of Research . . . 5
1.3.1 Computer-Human Interaction . . . 6
1.3.2 Input Sensor Hardware . . . 7
1.3.3 Software Architectures for Interactive Systems . . . 8
1.4 Document Structure . . . 9

2 Related Work **11**

2.1 Computer-Human Interaction . . . 11
2.1.1 Interaction Metaphors . . . 12
2.1.2 Multiple Orientations . . . 12
2.1.3 Applications . . . 13
2.2 Input Sensor Hardware . . . 14
2.2.1 Mechanical Sensors . . . 14
2.2.2 Electrical Sensors . . . 15
2.2.3 Acoustic Sensors . . . 19
2.2.4 Optical Sensors . . . 20
2.2.5 Sensor Capabilities . . . 26
2.3 Software Architectures for Interactive Systems . . . 27
2.3.1 Layered Architectures . . . 27
2.3.2 Windowing Systems . . . 27
2.3.3 Widget Sets and Toolkits . . . 28
2.3.4 Toolkits and Frameworks for Novel Input Devices . . . 28
2.3.5 Gesture Recognisers . . . 29

3 A Layered Architecture for Interaction **31**
3.1 Fundamentals . . . 31
3.1.1 Concepts . . . 31
3.1.2 Architecture Design . . . 36
3.2 Transport of Motion Data . . . 38
3.2.1 Design Considerations . . . 38
3.2.2 Location Transport Protocol . . . 39
3.3 A Formal Specification of Gestures . . . 40
3.3.1 Widgets and Event Handling . . . 41
3.3.2 Abstract Description of Gestures . . . 42
3.3.3 Gesture Description Protocol . . . 50

4 Sensor Hardware **57**
4.1 Fundamental Techniques . . . 57
4.1.1 Synchronised Active Illumination . . . 57
4.1.2 Interleaving Disjoint Light Sources . . . 64
4.1.3 Using LEDs as Sensors . . . 65
4.2 Interactive Surfaces . . . 67
4.2.1 TISCH . . . 67
4.2.2 MiniTISCH . . . 72
4.2.3 SiViT . . . 75
4.2.4 FlatTouch . . . 76
4.2.5 LCD with IR-LED Sensor . . . 81
4.2.6 Visible-light Display & Sensing . . . 82
4.3 Commercial Systems . . . 84
4.3.1 Free-Air Handtracking . . . 84
4.3.2 iPhone . . . 85
4.4 Sensor Capabilities . . . 85

5 The libTISCH Middleware **87**
5.1 Design Considerations . . . 87
5.1.1 Interoperability and Network Transparency . . . 87
5.1.2 Speed-Accuracy Tradeoff . . . 88
5.2 Hardware Abstraction Layer . . . 88
5.2.1 Adapters for Existing HAL Software . . . 88
5.2.2 Native Hardware Drivers . . . 91
5.2.3 Camera-Based Tracking: `touchd` . . . 92
5.3 Transformation Layer . . . 98
5.3.1 Removal of Lens Distortion . . . 99

5.3.2 Perspective Correction . . . 100
5.3.3 Online Transformation Process . . . 100
5.4 Gesture Recognition Layer . . . 101
5.4.1 Gesture Matching Algorithm . . . 101
5.4.2 Default Gestures . . . 104
5.4.3 Performance . . . 106
5.5 Widget Layer . . . 106
5.5.1 OpenGL-based Widgets . . . 107
5.5.2 Widget Bindings for Other Languages . . . 110
5.5.3 Class Diagram . . . 111

6 Applications 115
6.1 Interfacing with Legacy Applications . . . 115
6.1.1 Pointer Control Interface . . . 115
6.1.2 Gestures for Mouse Emulation . . . 116
6.1.3 Discussion . . . 117
6.2 Casual Entertainment . . . 118
6.2.1 Picture Browser . . . 118
6.2.2 Sudoku . . . 120
6.2.3 Virtual Roaches . . . 121
6.2.4 Tangible Instruments . . . 123
6.3 Interaction with Mobile Devices . . . 124
6.3.1 Detecting Phones on a Tabletop Display . . . 125
6.3.2 Joining Casual Games . . . 128
6.3.3 Evaluation . . . 129
6.4 Collaborative Applications . . . 130
6.4.1 Virtual Chemistry . . . 130
6.4.2 Interactive Whiteboard . . . 130
6.4.3 Virtual Patient . . . 131

7 Conclusion 135
7.1 Discussion . . . 135
7.2 Outlook & Future Work . . . 137
7.3 Summary . . . 139

A Appendix 141
A.1 libTISCH Configuration Files . . . 141
A.1.1 Calibration File . . . 141
A.1.2 `touchd` Parameter File . . . 142

A.2 Firmware for the ATtiny13 LED Controller 143
A.3 A Minimal X3D Renderer 145
A.4 MPX Compatibility Patch for FreeGLUT 146
A.5 GLUT-Compatible Wrapper for the iPhone 146
A.6 libTISCH Class Reference 147
A.6.1 `libtools` 148
A.6.2 `libsimplecv` 151
A.6.3 `libsimplegl` 155
A.6.4 `libgestures` 158
A.6.5 `libwidgets` 161

B Glossary **169**

C Bibliography **175**

List of Figures

2.1 Resistive touchscreen 16
2.2 Capacitive touchscreen 17
2.3 Projected capacitive sensing 18
2.4 FTIR-based touchscreen 22

3.1 Overview of the four architecture layers 37
3.2 Relationship between regions, gestures and features 43
3.3 Overlapping widgets capturing input events 44
3.4 Desynchronisation of widgets and regions 45
3.5 Protocol flow 51

4.1 Active illumination modes 59
4.2 Continuous vs. synchronised illumination 60
4.3 LED pulse capacity diagram 61
4.4 LED control circuit 62
4.5 Two consecutive images taken with an HDR camera 65
4.6 LED modes of operation 66
4.7 Overview of TISCH 67
4.8 Point light sources 69
4.9 Reflection of incident light 69
4.10 Overhead light source 70
4.11 Light source for diffuse illumination 71
4.12 Objects captured by diffuse illumination 72
4.13 MiniTISCH 73
4.14 Siemens Virtual Touchscreen 76
4.15 Inverted FTIR 77
4.16 FlatTouch 79
4.17 Touching the surface with three fingers 79
4.18 LCD with IR-LED touch sensor 81

4.19 Schematics of LED-based display and sensor matrices 82
4.20 Simultaneous display and sensing with LED matrices 83
4.21 Commercial interaction devices 84

5.1 `touchd` modules 93
5.2 Processing stages for a sample FTIR image 95
5.3 Peak detection 97
5.4 Blob tracking 97
5.5 Contact-shadow correlation 98
5.6 Gesture matching algorithm 102
5.7 process_input(packet p) 102
5.8 process_gestures() 103
5.9 Widget examples 110
5.10 libTISCH interpretation/widget layer class diagram 114

6.1 Picture browser 118
6.2 Picture annotations 119
6.3 Sudoku game 121
6.4 Hunting virtual roaches hiding under a book 122
6.5 Playing a chord on the virtual piano 123
6.6 Using Beatring with plain wooden blocks 124
6.7 Using Beatring with fiducial markers 125
6.8 Bluetooth name/location assignment 127
6.9 Interaction between mobile and tabletop Sudoku 129
6.10 Colour selection process 131
6.11 Using the virtual whiteboard 132
6.12 Interaction with the virtual patient 132

A.1 LED control state machine 144

Chapter 1

Introduction

This thesis aims at providing a generic approach to implementing multi-touch, multi-user and tangible interfaces. Since some years, especially the term "multi-touch" has appeared in computer science with increasing frequency. More recently, it has even made the jump to mass media. But what are these concepts which we keep hearing about?

For decades, interaction between computers and their users has happened mostly through two devices: keyboard and mouse. While the occasional joystick or touchscreen has been used in gaming or sales applications, mouse and keyboard have remained a core part of the vast majority of computer systems. They have also helped to define the look of the *user interface (UI)* itself: almost all software which is in use nowadays adheres to the *windows, icons, menus, pointer (WIMP)* paradigm with its windows, text fields, buttons and similar items.

Although this concept has been a spectacular success so far, there have been attempts at other, radically different approaches to *graphical user interfaces (GUIs)*. One such approach is the concept of multi-touch, or more generally, multi-point input. All previously used input devices and interfaces are designed to react to one single input location which is pinpointed by the mouse cursor. When several of these input points can appear in the same context at once, the number of dimensions in which the user's actions can be measured suddenly multiplies.

User interfaces which are based on these new concepts are often labelled *natural user interfaces (NUIs)* based on the rationale that human interaction with the natural world is also not limited to a single point. Rather, humans use all their fingers, hands and feet to interact with real-world objects. It

is this property which suggests that such interfaces may be easier and more intuitive to use.

A variety of options exists to implement such interfaces. Multi-touch sensors extend the well-known touchscreen concept to allow several fingers to be used simultaneously. Tangible interfaces provide physical handles for virtual objects, also allowing several of them to be used at once. Hybrid solutions combine these novel concepts with established input technologies, e.g., by enabling several mice and pointers to be used in parallel.

Although research on these topics has already been conducted for over 20 years, why are we only now seeing such a surge in attention towards them? Partly, this is because the results from these decades of research are now finding their way into commercial products and therefore into the hands of everyday users; partly, this is due to surprising recent discoveries which have significantly lowered the price of crucial sensor hardware components, thereby giving an increasing number of hobbyists and researchers access to such devices.

Whatever the reasons are, the increased interest in these interaction concepts has naturally led to the development of a wide range of new applications. Although many of them are extensions of existing software, some of these applications break existing paradigms to offer completely novel ways of interaction.

1.1 Motivation

As mentioned above, the goal of this thesis is to provide a generic approach to implementing software based on these novel interaction concepts. But is such a generic solution really necessary? Before discussing this question, let us first consider some scenarios which might present themselves in this context.

New Input Device Imagine a company which wants to bring its newly-developed multi-touch input device to market. The device by itself is not an attractive product, it needs support for applications to become one. Consequently, this company has two choices: either provide an attractive application out-of-the-box or provide a driver which allows existing applications to be used with the device. Unfortunately, both choices are far from perfect: in the first case, the company has to branch out into application development, which is probably not its primary area of expertise. In the second case, existing software may not work on the

device as expected, as this new device will likely differ subtly from those used to develop the applications.

Web-Based Information Booth Suppose a touchscreen-based information booth is to be installed. As the touchscreen offers multi-touch capabilities, some eye-catching games are planned. The main task of browsing information should however be done through a website. But how should this double functionality be implemented? It would be a time-consuming task to modify an existing browser engine to accept multi-touch input. On the other hand, reducing all multi-touch data to mouse pointer movements would severely limit the usability of the additional games.

Bringing Multi-Touch Software to Market Consider a startup company which has an amazing idea for an application with a multi-touch user interface. As they have no hardware-related expertise, they are planning to solely rely on distributing their software. But which hardware should they target? As they cannot afford to lock their program to a single type of device, their best bet is to create an internal hardware abstraction layer. However, this requires them to update their entire installation base when a new input device should be supported. Moreover, if the application makes use of gestures (which is likely), then their product needs to contain a significant amount of code for their recognition.

Prototyping of Multi-Touch Applications Think of a researcher who tries to develop a novel, multitouch-based application for a certain task. As the most suitable user interface for this task is not yet known, a number of prototypes will have to be created. It is very likely that the researcher will still be using a conventional desktop GUI for this task. In order to test any feature, it is necessary to switch to a multitouch-enabled interface. This may be tedious in the case of many small, quick tests. In addition, even an application based on completely new paradigms is likely to contain some well-known user interface objects such as buttons, sliders etc. Moreover, some novel user interface concepts such as movable tiles have already emerged as being applicable to a wide range of tasks. Would the researcher in question have to rebuild some or all of these components from scratch in order to experiment with them, this task would likely consume a significant amount of the work schedule.

After considering these scenarios, it is apparent that some general difficulties exist. Summarised very briefly, these are

- the sketchy support for existing, mouse based software,
- the lack of a common denominator for accessing various kinds of sensor hardware
- and the need to re-implement core components such as GUI elements or gesture recognition time and again.

We can now return to the question whether a generic approach to implementing such interfaces is really needed. Would it solve the described problems? The answer is yes, although only if one important restriction is made. As development in the field of these novel interfaces is continuing at an amazing speed, any generic solution presented now can only serve as a starting point. As new sensor devices and interaction methods emerge in the future, the approach introduced in this thesis will have to be re-evaluated. Nevertheless, even a generic solution which is only as complete as it is possible in the current context will still offer significant benefits.

1.2 Challenges

Naturally, the decision to pursue such a generic approach to novel user interfaces raises other questions. What are the challenges in designing such a solution? Is there only one strategy?

First, we will try to identify the most crucial requirements which this solution needs to fulfil. Such a generic approach should

- hide hardware differences from the application,
- allow easy integration of new types of input devices,
- provide a fall-back path for existing mouse-based applications,
- provide generic routines for common tasks such as gesture recognition,
- and offer an abstracted high-level view for the application developer.

Some of these requirements, particularly the first three, are rather straightforward and have to some extent already been implemented as standalone concepts. However, the last two requirements turn out to be more challenging. What high-level view is appropriate for the developer? What is a gesture?

With respect to the last question, it becomes apparent that there is no formal definition of a gesture. The Merriam-Webster Dictionary defines *gesture* as "the use of motions of the limbs or body as a means of expression". By this broad definition, even using a mouse is already a gesture. Consequently,

one challenge is to find a means of specifying gestures in a formal and therefore computer-readable way which nevertheless does not constrain the range of gestures that can be described.

When thinking about high-level interfaces for developers, a number of cues can be taken from existing *application programming interfaces (APIs)*. Modular, event-driven systems for creating user interfaces exist in abundance, thereby suggesting a certain suitability for the task. However, it is not immediately apparent whether these designs that have been developed for conventional GUIs are also well suited for these novel concepts. For example, it seems likely that a single logical event queue will not be sufficient to deal with several simultaneous streams of input data as they appear on a multi-touch device. Moreover, the same motion event can have quite different meaning depending on the context in which it originated. Another challenge is therefore to find a way of delivering these events to the correct recipients and interpreting them within the correct context.

Finally, we will consider the question of alternate strategies. In this thesis, we will follow the path of taking established methodologies and adapting or extending them to meet the requirements and challenges described above. But radically different concepts are also possible. One example has been presented by Wilson et al. [124]. In this approach, input data is directly converted to forces in a physics simulator, thereby allowing natural interaction without explicitly specified gestures. While meeting some of the requirements presented earlier, this method nevertheless fails to cover the entire spectrum of interaction and can therefore also be only considered a starting point.

1.3 Related Areas of Research

After having presented the motivation and main challenges for this thesis, we will now look at the larger context in which it is embedded. This thesis strongly relates to three overarching fields of research which span several disciplines such as electrical engineering, cognitive psychology and of course computer science. We will now briefly summarise the historical and current context provided by each of the following fields.

1.3.1 Computer-Human Interaction

The first area is *computer-human interaction (CHI)*, often also called *human-computer interaction (HCI)*. Novel methods of input need new ways of interacting with them, as a simple translation of existing mouse-based concepts is often insufficient. Therefore, some important design decisions for this thesis need to be based on results from this field of research.

CHI seems to be a relatively new field of research within the larger context of computer science. However, its beginnings can be traced back to the 1960s when the first graphical user interfaces were developed at Stanford ARC and later, at Xerox PARC [129]. Until recently, the main body of work has been focused on usability of common desktop interfaces. Widely known examples include, e.g., Fitts's Law [38]. It was published already in 1954 and deals with the tradeoff between speed and accuracy which is involved when moving towards a target.

Even though the topic of multi-touch seems to have emerged only in the last years, research in this area has started at least 25 years ago. One notable example is *VIDEOPLACE* by Myron W. Krueger [73], which showcased a wealth of interaction techniques that still seem revolutionary even in today's context.

However, only now that such interfaces are available to a wider audience, a significant amount of CHI research has been guided in this direction. One interesting topic in this context is orientation. While a normal desktop workplace consists of a monitor which usually never changes its orientation with respect to the user, this is quite different for many types of recently adopted interfaces. A tabletop display, for example, might be viewed by several users from quite different directions - some standing behind the table, some off to one side. The same applies to a mobile device which might be flipped over by the original user to show the content to a friend standing opposite. This fact has to be taken into account when designing software for such novel devices.

Another subject which needs to be considered is that of input focus. Most existing interface designs have been developed with a single pointer in mind. Therefore, most toolkits and widget sets are not equipped to deal with several simultaneous input points, even if they are mapped back to mouse events. While this might still work for unrelated windows on the same screen, two or more input foci within the same window are likely to cause malfunctions of various severity.

Along with the increasing amount of interest in multi-touch, a wealth of new gestures has also been proposed. These include the now-ubiquitous two-

finger scaling and rotation gestures as well as more esoteric ones [32]. When aiming for a generic method of dealing with gestures, this body of work can serve as a test case. Ideally, a generic approach should be able to represent every imaginable gesture.

1.3.2 Input Sensor Hardware

The second area, which is only marginally related to computer science, is input sensors. In order to interact with any computer program, the actions of the user have to be conveyed to the machine in some way. The most well-known and widely used devices for this task are keyboards and mice. Unfortunately, they are limited to a single person and a single point of interaction by design and are therefore not the primary choice for the types of interfaces which this thesis deals with.

Historically, the first computer input devices were simply arrays of switches, either operated manually or through a punch card. During the beginnings of computer science in the late 1940s and 1950s, these arrays directly modified the internal state of the machine. However, the switch to alphabetical input was soon made with keyboards which are little different from those in use today [78]. Later, in 1968, Douglas Engelbart [30] introduced the mouse, which was preceded by the invention of the trackball in 1952 [13]. Again, the basic functionality of these devices has changed little since.

Touch sensors, on the other hand, have appeared independent of computing applications. Surprisingly, they have already been used for electronic musical instruments during the 1960s [75]. Touchscreens, which combine touch sensors with a display, were developed during the late 1960s and early 1970s [11]. The first multi-touch system appeared in 1982 [81], yet this type of input sensor remained a niche application until recently.

Nowadays, available input sensor technologies can roughly be grouped into three categories: electrical, optical and mechanical. Well-known examples include laptop touchpads as electrical sensors, the Nintendo *Wiimote* [87] as an optical sensor and ball mice as mechanical sensors.

However, these examples all have in common that they are designed to report a single point of interaction, controlled by a single person, back to the machine. While the possibility exists to connect several such devices to a single machine and have them controlled by several users [57], this is a purely software-based solution and will be discussed elsewhere. Here, we will only

consider hardware which has explicitly been designed to track several points of interaction simultaneously.

Especially among hobbyists, optical multi-touch sensors are very popular as they can be constructed with little effort from commonly available hardware. On the other hand, commercial products focus mainly on electrical sensors, in particular capacitive ones. This type of sensors is usually less sensitive to environmental influences and allows better integration with thin display screens.

During the course of this thesis, several existing sensor designs have been extended or combined with each other in order to provide more data about the users' actions or to increase robustness.

1.3.3 Software Architectures for Interactive Systems

The third and last area is design of software architectures for interactive systems. For the common software developer, the software presented in this thesis should make it easy to create a new multi-touch GUI, ideally as easy as it would be for a normal GUI. Therefore, existing design methodologies with a large user base should be considered.

So far, there seems to be little work with respect to designing software architectures for novel kinds of input devices. Of course, in computer science, the entire branch of software engineering focuses on the construction of architectures in general and therefore provides generic guidelines. This branch has developed since the late 1960s and has its roots in the so-called "software crisis" of the 1960s to 1980s. During that time, it became apparent that prior structural planning was unavoidable to manage the huge growth in computer capacities and code volume.

One example which has been employed in this thesis is the widespread practice of splitting a larger task into self-contained subtasks, each of which is then implemented in a so-called layer. Together, these layers and the interfaces between them form a stack in which every single layer can be swapped for a different implementation without the need to change any other layer, as long as the interfaces stay the same. This technique has been employed in many network stacks, for example [60]. As most of the processing needed in our case can be broken down into four subtasks (data acquisition, transformation, interpretation and presentation), the resulting framework is split into four distinct layers with well-defined interfaces between them.

One important point which also needs to be considered in this context is

support for the existing, mouse-based infrastructure. The new design should therefore offer hooks to provide mouse-compatible input data to the underlying operating system.

From a developer's point of view, existing design concepts should be taken into account. One extremely widespread and well-known such concept is that of widgets. Widgets are small, self-contained building blocks of user interfaces, e.g., a button, a scrollbar or a slider. The development of this concept can be traced back to the Alto system [129] by Xerox PARC and to MIT's Project Athena [111]. Extending these accepted practices to multi-touch or multi-user systems will ease the transition for developers accustomed to common GUI libraries.

1.4 Document Structure

Following this short survey of the existing research within the context of this thesis, we will now give a brief structural overview of this document.

Chapter 2 - Related Work takes a detailed view at related fields of research and the existing work within them.

Chapter 3 - A Layered Architecture for Interaction gives an overview about the architecture which has been designed to subsume the various aspects of novel interaction devices into a coherent system.

Chapter 4 - Sensor Hardware describes the various types of input hardware which have been employed to retrieve data about the users' actions.

Chapter 5 - The libTISCH Middleware details the software framework which was developed to aid application development on the input devices described previously.

Chapter 6 - Applications describes several end-user applications which have been constructed based on this framework.

Chapter 7 - Conclusion provides a discussion of the presented work and future research directions.

Summary

In this chapter, the topic and motivation for this thesis was presented along with some example scenarios. The larger context provided by existing fields of research was also examined. We shall now proceed to review the related work within these fields.

Chapter 2

Related Work

This chapter will provide a short survey of the existing work within the three fields mentioned previously. A large body of work has emerged in the last few years, and even dedicated conferences such as "Tabletops and Interactive Surfaces" have already been held. However, most related publications are still distributed over a range of larger conferences such as CHI and UIST, both touching relevant areas. Whether the current surge in research and media attention that has focused on multi-touch or multi-user interfaces is the start of a paradigm shift remains to be seen.

2.1 Computer-Human Interaction

The first main area of research which this thesis relates to is computer-human interaction (CHI). This field itself has connections to diverse other disciplines, such as cognitive psychology and statistics, and is the most abstract one of the three related fields of research. In many cases, CHI-related research does not even consider implementation details such as sensor hardware or software libraries, but instead focuses solely on the question of usability, e.g. by examining paper-based mockups of a novel interface. A centrepiece of CHI-related research is the so-called *user study*, which aims to generate a reliable assessment whether a certain interface or interaction method is really offering benefits compared to existing system. To this end, a number of test subjects use the interface in question while their interactions and opinions are recorded and later analysed with statistical tools.

2.1.1 Interaction Metaphors

The concepts which are used in interacting with common graphical applications have reached such a large user base that they might already be considered common cultural knowledge, at least in technologically oriented societies. However, the same does not yet apply to multi-touch or tangible interaction concepts.

While the commercial success of the *iPhone* has started to promote some device-specific interaction metaphors, they are a long way from being as well-known as their mouse-based predecessors. Unfortunately, the application of such gestures may be hampered by patent issues [64]. While it is questionable whether specific movements of fingers can be patented, commercially-oriented adopters of such techniques may be deterred by the prospect of expensive litigation.

Another area where several simple gestures have been widely adopted is touchpad-based interaction. Many of these input devices now allow the user to use gestures such as dragging two fingers for scrolling and tapping with three fingers to simulate a right-button click.

Although new interaction metaphors are constantly being introduced, some very basic multi-touch gestures seem to have emerged across a wide range of devices and applications. These gestures are used for basic spatial transformations such as moving, scaling or rotating objects. To a certain degree, they mimic the movements which would be applied to real-world objects, e.g., rotating a sheet of paper by pushing with two fingers in opposite directions.

However, an universally accepted library of gestures will likely not be available in the foreseeable future. In this context, the work of Epps et al. [31] should be considered, in which the most intuitive use of hands and fingers for a variety of common tasks was evaluated without even using a computer. Participants were given paper mockups of user interfaces and instructed to perform certain tasks using their hands without further information as to which gestures should be used.

2.1.2 Multiple Orientations

Conventional, mouse-based user interfaces share one common property which may not be obvious at first sight. All GUI elements, particularly text, are oriented along one primary axis which points towards the user. When considering horizontal interfaces where many users can be interacting with the device from various directions, this single axis of orientation is insufficient. To sup-

port such a scenario, the system should be able to rotate every GUI element (or at least groups of them) independently of the others. One example for a system which extends the existing Java GUI framework to support this kind of interaction is *DiamondSpin* [102].

2.1.3 Applications

When looking at the available applications which have so far been built with these novel concepts and interfaces, it quickly becomes apparent that this is an area which still has room for improvement. While a large amount of smaller applets, demos and the like does exist, few of these have so far made the jump to actual, day-to-day use as applications. Of course, this may in part be due to a "chicken-and-egg" problem: without convincing software, people will be reluctant to acquire the hardware needed to run potential applications. At the same time, developers will be reluctant to target these novel platforms when the number of potential users is still tiny. This highlights the importance of hardware-independent applications, as these will be able to reach a larger number of potential users. Particularly support for using legacy input devices such as multiple mice might also provide a significant benefit.

When reviewing the currently existing applications, it seems that tools which allow one or more users to manipulate pictures using multi-touch input are probably the most prevalent type of software at the moment. Examples include the so-called *interface currents* by Hinrichs et al. [51] or the *PhotoHelix* by Hilliges et al. which combines multi-touch input with a tangible control device [49].

Another widely used scenario for multi-touch input is control of a virtual map display, such as in [39, 114, 7]. More advanced map applications, e.g. including additional interaction with mobile devices, also exist [101].

While the two previous categories of applications are mostly of a rather casual nature, one "serious" application area where novel user interfaces seem to show a lot of potential is music. The best-known example in this context is the *reacTable* [69] which has already been used by professional musicians. One of the few commercial multi-touch systems, the *Lemur* [63], is also intended to be used for controlling electronic music devices.

Perhaps surprisingly, games do not account for a large percentage of applications for such novel interfaces. Although many demos take the form of simple reaction games [98], only few novel games have been created [116] so far. Another approach which also has been explored only rarely until now, is to use existing games, e.g., Warcraft 3 with novel input devices [114].

Regardless of their use as day-to-day applications, a wide variety of interfaces using novel interaction devices has already been presented. When regarding this body of work, it becomes apparent that any generic approach will have to be able to encompass a sizable number of different modes of interaction.

2.2 Input Sensor Hardware

As described in the previous chapter, input sensors have evolved for at least 50 years. However, since the introduction of the mouse, most other types of input devices have been consigned to specialised niche applications. Input sensors can be broadly categorised as direct or indirect [103]. A well-known example for direct sensors are touchscreens, which directly record the position of the user's finger. On the other hand, indirect sensors such as mice or the *Wiimote* usually record the motion of the device itself which is then translated into cursor motion. Note that the distinction is not always intuitive at first sight - a touchpad is still an indirect input device, as the data is not manipulated directly at the touched location, but instead through a cursor.

From an abstract point of view, the field of input devices has a significant overlap with tracking and positioning devices in general. Therefore, several such devices will also be introduced where appropriate. Special emphasis will be placed on the support of multiple interaction points.

2.2.1 Mechanical Sensors

For the sake of completeness, mechanical sensors should also be considered. They have been the first kind of dedicated input sensors [13] and have been employed in joysticks, ball mice [30] or trackballs for decades. While rolling over the surface beneath the mouse, the ball drives two perpendicular rotary encoders which translate the movement along each axis into "ticks" which are arbitrary, mouse-dependent units. Although most mice adhere to a common connection standard such as the *Universal Serial Bus (USB)*, the number of ticks per unit length varies widely, resulting in potentially different user experiences when changing devices.[1]

Mechanical mice also highlight one of the big drawbacks of mechanical sensors: they are susceptible to accumulation of dirt, which degrades performance

[1] These per-device differences even apply to modern optical mice.

and necessitates regular cleaning. Therefore, mechanical sensors have largely vanished from commonly used input devices.

However, high-end mechanical sensors are able to provide the highest available tracking accuracy, though at significant cost. They are employed in high-precision measurement devices, e.g. by Faro [36], where the additional cost of dirt-proof encapsulation is insignificant compared to the overall price. While this device has not primarily been designed to be an interaction device, it nevertheless is shipped with a mouse driver that allows use of common applications with the measurement arm.

2.2.2 Electrical Sensors

Electrical sensors are a very common type of input device. For example, most laptop touchpads fall into this category. These sensors try to measure changes in the electric or magnetic field which are triggered by the presence of, e.g., a finger or a special kind of pen. Other well-known hardware devices in this category include graphics tablets or touchscreens.

Resistive Sensing

Resistive sensors are the cheapest and most common kind of electronic input devices and are usually found on small touchscreens, e.g. in mobile devices. This sensor consists of two transparent, conductive layers which are held apart by a grid of elastic, transparent spacers (see figure 2.1). Usually, these layers consist of a transparent plastic film that has been coated with *indium tin oxide (ITO)*. Two opposing sides of the upper layer are equipped with contacts between which a potential difference is applied. The other two sides of the lower layer also carry contacts, which are connected to voltage sensors. When pressure is applied to the screen, the upper and lower layer make contact and a voltage divider network is formed. The two voltage measurements obtained at the lower layer can now be used to calculate the x and y coordinates of the contact point when the total voltage as well as the resistance of the layers is known.

This is also the reason why this type of screen does not support detection of multiple contact points. Any two measurements which result from two or more contacts could equally result from a single contact point at the centre of gravity of the other points. By a trick, it is nevertheless possible to infer the positions for two simultaneous contacts as shown in [7]. When the second finger is put down, the reported position immediately jumps from the first

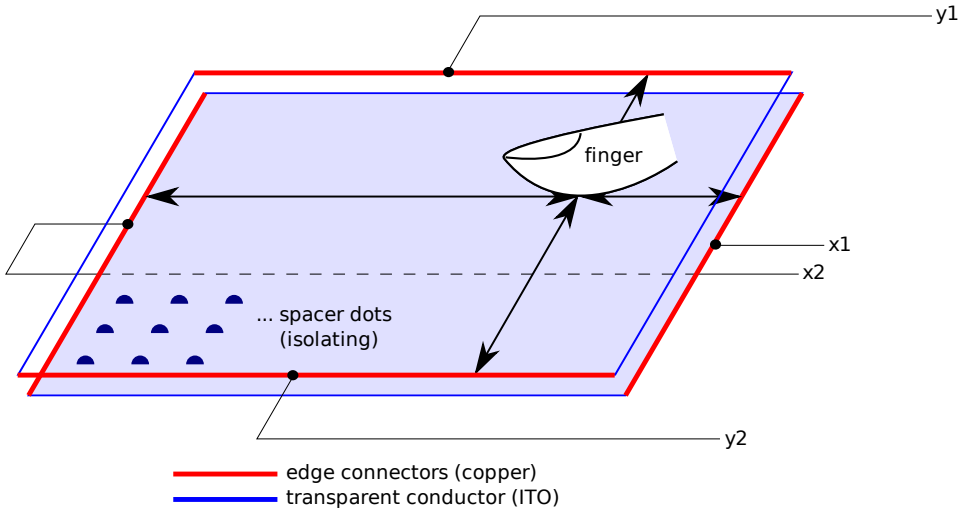


Figure 2.1: Resistive touchscreen

position to the centre point. As this jump is too fast to be caused by real movement, the conclusion can be drawn that a second finger has been placed on the surface. Moreover, the jump vector is roughly half of the distance to the second point, whose position can now be calculated, too. From this point on, the relative scaling and rotation of the two points can be inferred as long as only one of both is moved at a time.

While this is a quick way to generate events similar to true dual-touch, it is nevertheless only a temporary solution and fails to produce sensible results for more than two input points. Another drawback of resistive touch sensors is that due to the flexible upper layer and the small air gap beneath, they can be damaged by sharp objects or wear out due to repeated pressure on a single point. They are therefore not well suited for frequently used or public applications. Moreover, the sensor does not supply any pressure information; the contact data is binary. An advantage of this kind of sensor is that it can be operated equally well with gloved hands, pens or any other object.

Capacitive Sensing

Capacitive sensing is a term which encompasses several quite different technologies. In its simplest form, called surface capacitive sensing, it works similar to the resistive sensing method mentioned above. Several electrodes (usually four) are attached to the corners of a non-conductive surface or coating which

acts as a dielectric. When a conductive object comes into contact with the surface, it creates one "virtual" capacitor together with each of the four electrodes (see figure 2.2). Together with additional circuits at the electrodes, each of these capacitors forms part of an oscillator. As the capacitance is proportional to the distance to the electrodes, the resulting frequency of the oscillator changes with the distance between contact point and electrode. This frequency can easily be measured by the sensor controller and is then used to infer the exact location of the contact point.

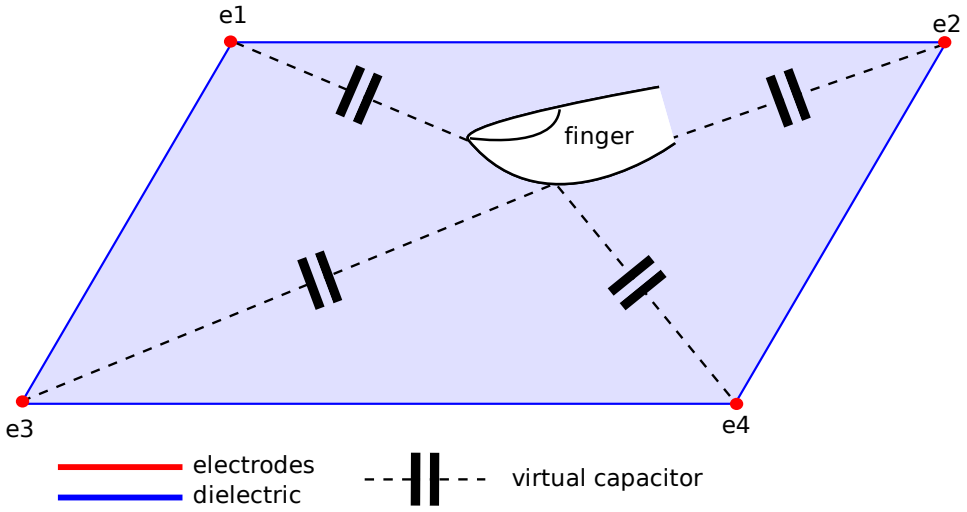


Figure 2.2: Capacitive touchscreen

Again, this method is unable to reliably detect two or more points, as further contacts severely disturb the capacitance. Depending on the specific implementation, the "jump detection" method mentioned above could again be employed to support up to one moving and one stationary contact simultaneously.

However, there is a second type of capacitive sensing, usually called projected capacitive technology (see figure 2.3). In these systems, a grid of conductors is spread over the sensing surface. These can be very thin wires or printed traces of transparent, conductive materials such as ITO. As the horizontal conductors are separated from the vertical ones by an insulating layer, each of the crossings forms a tiny capacitor. By consecutively charging each of these capacitors through one of the vertical conductors and then measur-

ing the capacitance at each of the horizontal ones, a capacity "image" of the surface can be generated. When a conductive object approaches or touches the surface, the capacitance of the closest elements changes, thereby indicating contact (highlighted in green).

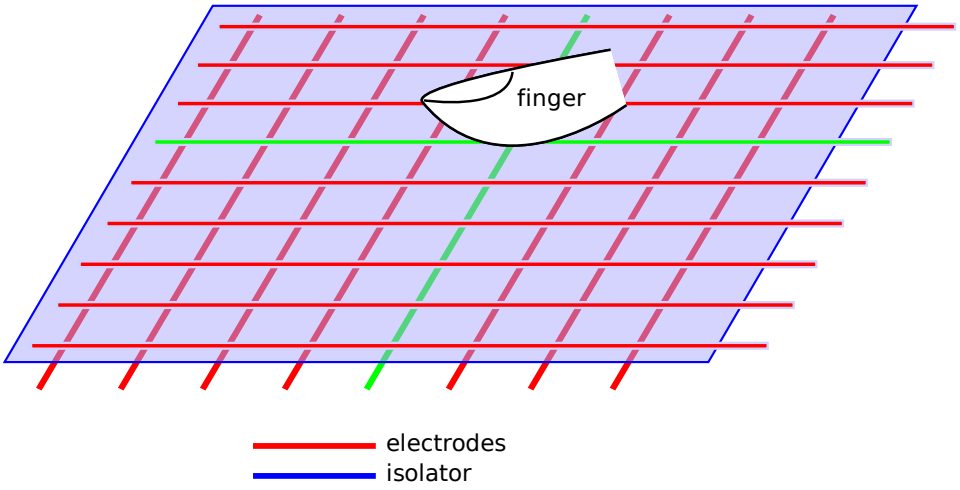


Figure 2.3: Projected capacitive sensing

Note that there are two possible methods of driving this type of sensor. In the first case, each single capacitance is measured separately, resulting in one measurement per crossing. Touch points can be calculated with methods very similar to those applied to images. This method is employed, e.g., by the iPhone [64]. In the second case, a grid of m rows and n columns of conductors will produce $m + n$ distinctive measurements, each one indicating the total capacitance along a conductor. This has repercussions on the available data. Image-based methods will not work here, as additional heuristics are needed to translate the resulting row and column maxima into actual touch locations.

Capacitive Coupling

Slightly different approaches are followed by two commercial products. The first one is *SmartSkin* [98] from Sony. It also employs a matrix of horizontal and vertical conductors which are separated by an insulator. However, the capacitances are only marginally relevant for measuring touch points. Instead, a high-frequency signal is sent through the vertical conductors and received through the horizontal ones. When the user touches the surface, the body

acts as ground and drains the signal. This loss of intensity can be measured. As only one wire at a time is used to transmit the signal, an unambiguous mapping of touch locations to sensor locations is possible. The system is also able to sense objects on the surface if they are conductive and being touched by the user.

The second system is *DiamondTouch* [15] from Mitsubishi Research. Its name results from the distinct rhomboid shape which the conductors employ to cover the entire surface with little overlap. One distinguishing property of DiamondTouch is that the user itself acts as transmitter. Through a special seat cushion or foot mat, the user is connected to the system and transfers a high-frequency signal to the conductors in the surface upon contact. This concept has the significant advantage that by using different signals and conductive mats, it is possible to identify the user to whom a certain contact point belongs. However, there is also one drawback: when one user creates two or more contacts, an unambiguous identification is not possible, as row and column signals are measured simultaneously. Therefore, only an axis-aligned bounding box containing the set of possible contact points from a single user can be measured directly. Any further differentiation depends on point tracking and additional heuristics.

2.2.3 Acoustic Sensors

A third, completely different type of input sensor is based on acoustics. Instead of augmenting the interaction surface with conductors, microphones are employed to pick up specific sounds created or altered by the user. One advantage of this kind of sensor is that many existing surfaces can be easily converted into input devices. This is especially interesting for applications where a high degree of robustness is desired.

Surface Acoustic Wave

This type of acoustic sensor also contains an active component. Piezo-electric emitters send high-frequency sound pulses into the material which are reflected at the borders and finally arrive at a receiver. When a person touches the surface, part of the wave energy is absorbed by the finger. By analysing the received signal, the touch location can be inferred. An example for a commercial product which employs this technology in vandalism-proof touchscreens is the *SecureTouch* [28] system by EloTouch.

Acoustic Pulse Recognition

The other type of system which we are going to investigate relies on sounds which are actively created by the user. This means that simply touching the surface will not be registered by the sensor. Rather, the user needs to knock on or scratch over the surface to generate input data. The generated sounds can be analysed in two different ways. The first approach, called time difference of arrival, takes data from two or more microphones and correlates the signals arriving at each one of them [92, 130]. By analysing the time difference between pairs of signals, a set of hyperbola can be calculated which ideally intersect in the point where the sound originated. One drawback of this approach is that reflections and multi-path propagation inside the material tend to add a high margin of error to the signal correlation and therefore also to the location estimation.

A slightly different method which partly solves the problem mentioned above is to pre-record a number of sounds by knocking at known locations on the surface [93]. This approach is called *location template matching*. It only requires a single microphone, and sounds generated by the user are then compared to the library of stored sounds. The closest match is assumed to have been recorded close to the actual location, thereby providing an estimate on the position. The accuracy of this method mainly depends on the quality of the calibration process.

A variant of this method does not attempt to determine the exact location of the interaction, but rather tries to classify the shape which the user is tracing on the surface. To this end, the distinct sounds created by different paths are pre-recorded and then compared to the signal at runtime [46].

2.2.4 Optical Sensors

A wealth of multitouch-capable sensors is based on optical principles, likely because they are for the most part easier to manufacture than electrical systems and more precise than acoustic sensors. Unless noted otherwise, all of these systems operate in the *infrared (IR)* spectrum which is invisible to the human eye.

Occlusion Sensors

A very simple approach to sensing contacts is to surround the area in question with a bezel carrying an array of light emitters and sensors. These are usually

arranged in pairs opposite each other and, with suitable optics, can exactly detect when an object interrupts the specific light beam. Again, this suffers from similar drawbacks as some electrical sensors mentioned earlier, as only an axis-aligned bounding box can be measured directly. An example for this kind of system is the commercial *CarrolTouch* [27] system from EloTouch.

A similar, slightly more complex approach is followed by the *SmartBoard* [107] systems from SMART. Instead of separate light barriers which result in a binary value each, the emitter is a continuous light strip which is viewed by line cameras situated in the edges. Depending on the number of cameras and the processing unit, two or more contact points can be detected simultaneously. From an abstract point of view, every camera projects a number of "occlusion rays" across the surface which intersect in a contact point. Yet again, this system has limitations when sensing several contact points, as these quickly lead to ambiguities and occlusion problems.

Frustrated Total Internal Reflection

One very popular method for multi-touch sensing is that of *frustrated total internal reflection (FTIR)*. Originally, this method has been patented for fingerprint sensing in 1965 [121]. However, in 2005, Jeff Han [45] adapted this principle for use in a touchscreen by basically just scaling it up. The basic principle of operation is as follows (see also figure 2.4). Infrared *light-emitting diodes (LEDs)* are placed on the rim of a sheet of, e.g., acrylic glass and radiate into the material. As long as the surface is untouched, the infrared light is reflected internally similar to an optical fibre due to the large difference in refractive index to the surrounding air. However, if a dense object such as human skin touches the surface, the total reflection is interrupted and the contact area is illuminated. The bright spot which subsequently appears on the projection surface can then be recorded by the back-mounted camera and delivered to a computer for processing.

As this type of sensor is very easy to build, even from off-the-shelf components, a large number of systems based on this principle have been constructed. Consequently, a significant amount of improvements and modifications have been published. One notable example deals with the drawback that it is uncomfortable for the user to drag fingers over plain acrylic glass. The "NUI Group" community [88] has shown that by adding a thin layer of elastic silicone on top, the user experience can be improved dramatically. Details on various materials can be found in the technical report by Schöning et al. [100] It is also worth noting that only very few materials other than human skin are

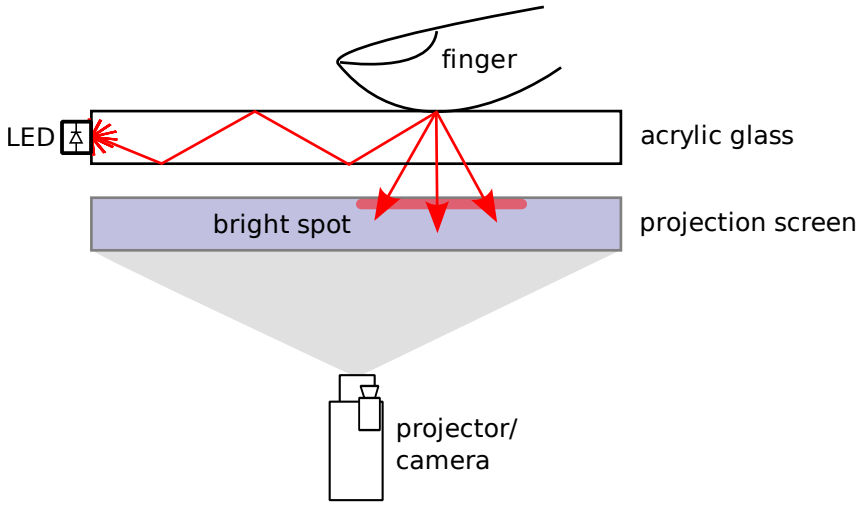


Figure 2.4: FTIR-based touchscreen

reliably able to trigger the FTIR effect. Depending on the application, this fact can turn out to be an advantage as well as a drawback.

While highly popular, this method requires significant space behind the projection screen, often resulting in bulky setups. A modified approach which applies the FTIR concept to an *liquid crystal display (LCD)* screen is called *FlatIR* by Hofer et al. [53]. This method puts an FTIR surface in front of the LCD panel. As these panels are mostly transparent to infrared light, the reflections from surface contacts can be detected behind the display. To save space, these emissions are not captured by a camera, but instead by an array of photodiodes which has been inserted behind the backlight. While effective, this concept requires major re-engineering of the entire display.

"Backscatter" Detection

Another variant of optical multi-touch sensing can be achieved by directing infrared light onto the interactive surface from behind. This method is often called *diffuse illumination (DI)*. As an approximation, it can be assumed that roughly 50 % of this light is directly reflected back towards the camera, while the other half radiates outward. When an object now approaches the surface, it starts reflecting some of the light shining out of the surface. Therefore, the closer an object is to the surface, the brighter it appears. Again, the display

surface acts as a diffuser, so the object will also quickly get out of focus with increasing distance. One difference between this method and FTIR is that almost any kind of object can be sensed instead of only human skin. Moreover, objects can already be detected before they directly touch the surface. As a consequence, it becomes difficult to unambiguously differentiate between objects that are actually in contact with the surface and those which are only hovering above. Additionally, the image processing steps which are needed for touch detection are slightly more complex, as a high-pass filter has to be employed to separate distant, out-of-focus objects from those which are very close to the surface. Some well-known setups which utilise this type of sensing include *Surface* [82] and *reacTable* [69].

One disadvantage which this method has in common with most other camera-based methods is that a significant amount of space behind the display is needed, even in combination with a fish-eye lens. A system which heavily modifies this method to counter this drawback is *ThinSight* [61]. As mentioned above, LCD screens usually permit transmission of infrared light. ThinSight takes advantage of this fact by placing an array of integrated IR distance sensors behind an LCD. These sensors continue to work through the display and sense the approach of any kind of object through reflected infrared radiation, similar to the DI method. In contrast, the space requirements of the entire setup are reduced significantly. Unfortunately, a significant amount of modifications to the display system are necessary.

A variant of backscatter detection is "liquid displacement sensing" [50]. In this method, a glass plate is covered with a thin sheet of white latex which acts as top-projection screen. The space between glass and latex is filled with ink and lit from the backside. A camera viewing the sheet from below normally only sees a black ink surface. However, when an object is pressed on the latex sheet, the ink is displaced and the white latex becomes visible through the glass plate. This concept works with fingers as well as other objects and even allows a certain degree of pressure sensitivity.

Assisted Hand Tracking

In this category, we will now consider hardware which tracks the user's hands and possibly also fingers by means of arbitrary devices which are either worn on or held in the hand. These devices include such diverse categories as fiducial markers, infrared emitters or video game controllers.

At first glance, the following systems do not seem to have anything to do with multi-touch sensing. Nevertheless, the data produced by these trackers

is easily mappable onto the general concept of multi-touch. In popular media, this type of interaction is often cited, as it prominently appears in the movie "Minority Report" [109]. One commercial example is the hand-tracking system produced by ART [1]. While this company mainly manufactures high-precision tracking systems which deliver motion data in six degrees of freedom, they also offer an extension which enables these systems to track two hands and three fingers on each hand simultaneously in three *degrees of freedom (DOF)* (six in the case of the hands). Two special gloves equipped with infrared emitters have to be worn. From the 3D positions of the various emitters detected by the stereo cameras, the entire hand posture can then be calculated. If a person wearing these gloves stands in front of a large screen, the projection of the 3D hand and finger positions into the screen plane can be interpreted as multi-touch data.

Another widely known approach to delivering hand and finger tracking data are so-called *data gloves*, which usually employ stretch sensors embedded in a glove above the fingers [37]. When coupled with any 3-DOF or 6-DOF tracking system, these gloves are also able to deliver accurate hand and finger positions.

A less capable, but also less expensive variant of this technology can be created by using the Nintendo Wiimote as a sensing device. The Wiimote contains a high-performance IR sensor which reports the position and size of the four brightest spots in the sensor's field of view with 100 Hz. As shown by J. Lee [76] and L. Vlaming [117], this can be employed for low-cost hand tracking. Again, special gloves are needed which either carry a battery and IR LEDs or retro-reflective tape on two fingers each (usually thumb and index finger). In the second case, the Wiimote has to be placed near an infrared illuminator which provides light for the reflectors. While the reported data is only two-dimensional, the mapping onto a screen plane would discard the third dimension anyway. To reduce errors, the sensor viewing direction should therefore be oriented roughly perpendicular to the screen.

Finally, this category also includes dedicated hand-held tracking devices such as the Wiimote in its originally intended mode of usage or so-called *flight-sticks* which are best described as a joystick handle that can be freely moved through space.

Free-Hand Tracking

While the previously described methods for tracking hands in 3D space are easy to set up, they nevertheless require the user to wear or carry additional

equipment which may be distracting. Some researchers have focused on setups which are able to track bare hands without markers. One such system is *Touchlight* [122] by A. Wilson, which uses a stereo camera to track objects approaching the interaction surface. To allow the cameras a clear view, a special type of projection screen is used. It contains a holographic element which is transparent or opaque depending on the view angle. From the point of view of the projector, the screen is opaque and therefore acts as a diffuser and projection surface, whereas for the cameras, it appears transparent.

Another system which has also been presented by Wilson is *PlayAnywhere* [123]. Here, a short-range projector and camera with wide-angle lens have been set up so that they can be put on any non-reflective surface and use the area in front of the device as projection screen and sensing surface. An infrared light source illuminates the area, and objects create shadows extending backwards from the device on the surface. When an object like a finger comes closer to the surface, it occludes a larger part of its own shadow, thereby changing the shape of the shadow that is visible to the camera. When touching the surface, the shadow has a distinct peak as opposed to a rounded end when the finger is at a distance. This can be recognised by the system and translated into a touch event. Moreover, blank sheets of paper can also be recognised and tracked in order to be used as miniature projection screens.

A slightly different approach which explicitly focuses on tracking fingertips is *Brightshadow* [99] by J. Rekimoto. Here, the system builds two or more "light walls" created by directed IR emitters which are synchronised with the camera. For two light walls, the camera takes three consecutive frames. In the first frame, both light walls are off, while during the second and third frame, one light wall each is active. This allows easy background subtraction and recognition of objects such as fingers which intersect the light wall(s). From this data, a 3D position of the fingertip can be calculated.

Hybrid Approaches

An interesting combination of several previously mentioned approaches is *SecondLight* [62] by S. Izadi. The basis of the system is an FTIR-based tabletop display. However, the projection surface is not an ordinary diffuser, but rather a switchable one, often called "privacy glass". This material operates on the same principle as liquid crystal displays and can be electrically switched between diffuse and transparent states. Both states are alternated 60 times per second. In the diffuse state, the surface acts as screen for a synchronised projector, while in the transparent state, a synchronised camera is able to see

through the surface and reliably recognise hands and objects above. In an additional step, a second projector is added which projects through the surface in the transparent state and is therefore able to display additional images on small hand-held projection screens, provided they can also be tracked by the camera.

2.2.5 Sensor Capabilities

Sensor Technology	multi-point	user identification	direct touch	hover state	arbitrary objects	fiducial markers
Resistive	-	-	+	-	+	-
Capacitive (standard)	-	-	+	o	-	-
Capacitive (projected)	+	-	+	o	-	-
Capacitive coupling	+	+	+	o	-	-
Acoustic	o	-	+	-	-	-
FTIR	+	-	+	-	-	-
Backscatter	+	-	+	+	+	+
Hand Tracking (assisted)	+	+	-	+	o	o
Hand Tracking (free-hand)	+	-	-	o	o	o

Table 2.1: Sensor Technology Overview

To provide a short overview of the numerous sensing methods presented above, table 2.2.5 quickly summarises the main features and the degree to which they are supported (+ full support, o rudimentary support/dependent on implementation, - no support).

2.3 Software Architectures for Interactive Systems

As mentioned before, there seems to be little work so far which focuses on building software architectures or frameworks for novel types of interactive systems. However, such approaches are abundant in other disciplines of computer science, e.g. in ubiquitous computing as shown in the survey by Endres et al. [29]. Generally, such a *middleware* will greatly ease standard tasks for developers when designed well.

To get an overview of the context as well as of the specific work which has already been done regarding frameworks for novel input devices, we will first look at the general principles involved in software design followed by a brief survey of the existing application-specific systems.

2.3.1 Layered Architectures

One of the most basic principles on which many larger software systems are built is that of a stacked or layered architecture. In this design, the whole task is split into a series of several smaller, self-contained subtasks. These subtasks or layers are sorted into a stack, with the least specific tasks at the bottom and the most specific ones at the top. Each layer in this stack should only communicate with those directly above and below by means of well-defined interfaces. This yields the additional benefit that one layer usually can be swapped with an alternative implementation without changing the rest of the stack. One widely known example for this kind of design is the ISO/OSI network model [60]. Many operating systems can also be viewed as such a stack, with hardware drivers and the kernel at the bottom, libraries and daemons in the middle and end-user applications at the top end of the stack.

2.3.2 Windowing Systems

When considering a common GUI-based system in its entirety, a special role is filled by the windowing system. Well-known examples include *Xorg*[2] [131], *Quartz* [42] or the Windows GUI system [83]. Its most basic task is to mediate between the hardware on one side and the end-user application on the other side. Most windowing systems are more than a simple hardware abstraction

[2] the current reference implementation of the X11 Window System

layer, as they provide a canvas system which allows painting of graphics primitives and pixmaps as well as the ability to arrange several partly overlapping drawing surfaces, i.e. GUI windows. Additionally, a message passing interface is usually also provided, which is employed to direct user input such as mouse movement or keystrokes to the correct window. Moreover, the windowing system is responsible for displaying and controlling the mouse pointer and sometimes the input focus. This is important when considering multi-pointer interaction, as all examples mentioned above only provide a single mouse pointer for the entire screen. So far, only an experimental extension to Xorg, called *Multi-Pointer X (MPX)* [57], offers the ability to control a virtually unlimited number of mouse pointers, thereby enabling simultaneous control of several legacy applications or of a single multi-pointer aware program. The next version of Microsoft Windows is also expected to contain similar functionality, although details are still sketchy.

2.3.3 Widget Sets and Toolkits

Especially in the realm of graphical user interface design, the concept of the widget is commonly used. The term derives from "window gadget" and was coined in 1988 during *Project Athena* [111] at MIT, even though similar concepts were already used in the *Xerox Alto* [129]. A widget is a small, self-contained building block of a GUI. By combining several different types of widgets, a complex user interface can be rapidly assembled. Common widgets include buttons, sliders, text entry boxes and many more. As a consistent "look-and-feel" as well as easy interoperability between individual widgets is generally desired, widgets are mostly grouped together in large collections, called toolkits, which share a common, often configurable, look and implementation details. Examples for widely used toolkits include *Qt* [97], *GTK+* [112], *Swing* [110], *Aqua* [2] or the Windows User Interface API [84].

2.3.4 Toolkits and Frameworks for Novel Input Devices

All these toolkits mentioned in the previous section have started their development at a time when multi-touch or multi-pointer input were largely unknown concepts. These systems have therefore not been designed to deal with several simultaneous input points, even if the windowing system should already offer support for this functionality. For this reason, extensions for or redesigns of existing toolkits which focus on supporting multiple input points have started to appear. Due to its success as one of the first commercially

available multi-touch input devices, a significant portion of these libraries are based on the DiamondTouch hardware platform. Examples include *DiamondSpin* [102], which modifies the widely used Java API to allow free rotation of GUI elements as well as multi-user input, or *DTFlash* [33], which aims to achieve the same for Flash development. Recently, a .NET-based toolkit [14] also has been published.

On the other hand, many systems which are based on optical tracking currently employ the *Tuio* [68] protocol which is based on the OSC Specification [126] to deliver tracking data to applications. While these are not GUI frameworks, they nevertheless form the basis for many applications. One well-known example is *reacTIVision* [69], which focuses on tracking tangible objects on a rear-illuminated surface. Another system which is extensively used in the hobbyist community is *touchlib* [89] and its successor *CCV* [90]. The main goal of these systems is to provide an easily accessible platform for multi-touch tracking, e.g. with a simple webcam.

Another library which goes beyond the object or blob tracking provided by the previously mentioned ones is *libavg* [118]. libavg offers a Python scripting interface which can be directly employed to create interactive graphical applications. A tool whose main focus is visual programming for audio and video installations is *vvvv* [119], though it has also been adapted to multi-touch development. One other important development platform is *Processing* [41] and others based thereon [79]. Processing aims to provide a Java-based development environment which hides most of the language's complexity from the user. Its simplicity has resulted in high popularity among users which do not have previous programming experience, such as graphical designers.

While some of these software packages already provide reusable components to some degree, their adaptability to varying conditions such as different hardware devices or programming environments is limited. Although support for camera-based tracking and touch detection is already quite broad, a full GUI toolkit similar to those for common mouse-based interfaces does not yet seem to exist.

2.3.5 Gesture Recognisers

A different aspect of higher-level interaction support is provided by software which tries to recognise gestures in the input stream as opposed to simply reacting to touch/release events. Several approaches based on DiamondTouch have been presented by Wu et al. [128, 127]. A common aspect of these systems is that gesture recognition still is performed inside the application itself.

However, there are approaches to separate the recognition of gestures from the end-user part of the application [48, 26]. With the exception of *Sparsh-UI* [44], these systems are not yet beyond the design stage. Sparsh-UI follows a layered approach with a separate gesture server that is able to recognise some standard gestures for rotation, scaling etc. independently of the end-user application.

Looking at these systems, it is apparent that even those which follow approaches at generalisation in one context do not extend this generalisation to other contexts. For example, most of the generic gesture recognition software which exists is still focused on a single hardware device. Moreover, no coherent formalism has so far been presented which could provide a basis to model interaction concepts.

Summary

A review of the related work which was presented in this chapter shows that most research still is focused on creating novel kinds of input devices. While many diverse applications have already been written for these devices, they often do not evolve beyond the status of tech demos. Moreover, the topics of reusability and generalisation of the underlying concepts seem to have received almost no attention up to now.

Chapter 3

A Layered Architecture for Interaction

In this chapter, the system architecture which has been developed to create a coherent model of novel interaction devices shall be described. As mentioned previously, a large body of work on multi-touch interfaces, tangible interfaces and other novel types of interaction has been created during the last ten years. Almost all of the systems which have been developed so far are monolithic and have been designed for a single type of input hardware, thereby limiting reusability of code and concepts.

To address this limitation, a layered architecture has been designed which has the goal to subsume all these various means of interaction into one single, coherent formalism [22].

3.1 Fundamentals

In this section, we shall first take a look at the concepts which our architecture is based on in order to reach its goal. We will then go on to describe the architecture itself and briefly present its separate layers.

3.1.1 Concepts

Capturing Movement Data

From an abstract point of view, the starting point in any kind of computer-human interaction is the intent of the user. Assuming familiarity with the

interface in question, the user will then proceed to execute an action that causes the intended result.

When looking at user interfaces throughout the history of computing, the user's actions can at the most generic level be divided into two types:

Text entry. Text-based interfaces were the first ones that allowed users to truly interact with the computer through a sequence of commands and responses (*command line interface (CLI)*). They are still common today, though use of the command line has receded and often been replaced by keyboard shortcuts. For the purpose of our classification, it does not matter whether the text itself is entered through a keyboard, through a speech recognition interface or by any other means.

Movement. All other actions of the user can be very broadly classified as movement.[1] We can further subdivide this category as follows:

Mediated movement. In this case, the user movements are recorded by a physical object which is directly or indirectly connected to the computer. The simplest example for such a physical object is, of course, the mouse. However, all those types of interaction which are usually described as "tangible" can also be put into this category.

Free movement. This case describes those interactions where the user does not have to touch and move a specific object, but can instead move freely through space. While additional equipment such as special gloves may still have to be worn in some cases, the movements themselves are unconstrained. A popular and well-known example for this kind of interaction appears in the movie "Minority Report" [109].

On-surface movement. Finally, this case is a hybrid of the previous two. Here, the user does touch an object such as a screen. However, it remains fixed and the movements of the user relative to the surface of this object are recorded. The most common example for this type of interaction are touchscreens, but also touchpads.

In the context of this thesis, the main focus will be on interactions of the second type – movement. While text-based modes of interaction shall be

[1] Although technically, text entry also requires the user to perform finger or lip movements, we will not consider them here.

considered where appropriate, the presented architecture has not primarily been designed to deal with them. Therefore, the most basic step which this architecture has to perform is to locate and identify various types of objects and track their movements through space. It is also necessary that this position data can be related to the display coordinate system later in some way.

When reviewing the various types of movement which are considered here, it can be concluded that the system should conceptually distinguish between at least three different entities:

- fingers
- hands
- objects

In this case, "objects" serves as a catch-all term for anything which does not fall into the first two categories, e.g., tangible UI elements. The question why differentiation between these entities is necessary in the first place is easily answered, as very different semantics may be attached to, e.g., finger movement as opposed to object movement. From now on, we will refer to these entities collectively as "input objects".

When considering the kinds of data which can be gathered about the motion of these entities, the following list of data atoms emerges:

Position. Definitely the most basic and most important type of data is the spatial position of the entity in question. Depending on the sensor hardware, this can be a two- or three-dimensional vector relative to an arbitrary, hardware-dependent reference coordinate system.

Point-of-Interest/Peak. Depending on the object in question, the position may not be the same as the point-of-interest. One example is a hand with outstretched index finger. While the overall position would likely be described by the centre of the palm, the point-of-interest would rather be located at the peak of the index finger. In some contexts, this location is also called the *hot spot*.

Orientation. While not all types of sensors (e.g., touch sensors) are able to deliver this information, it is nevertheless important especially for tangible UI elements. However, this information is also meaningful for fingers and hands when available.

Contact Size/Pressure. These two properties are mostly relevant for touch sensors. Unfortunately, they are difficult to separate, as many sensors

detect an increase in pressure as an increase in size. Additionally, this does not take into account that different persons may have different-sized fingers in the first place, so this data should only be used if the sensor in question is actually able to deliver true pressure measurements.

Shape. Especially when using camera-based sensors, the shape of an object can sometimes also be determined. This information can be valuable when using arbitrary objects as tangible UI elements.

Identifier (ID). When using objects that are tagged with fiducial markers, every one of these markers usually exhibits a unique identifier which may also serve to deliver additional meaning. The same applies to hardware such as the DiamondTouch which is able to identify the user whom a certain contact belongs to. In addition, temporary identifiers should be assigned to untagged objects and stay with them as long as they move within the sensors' range.

Parent. In some cases, a parent-child relationship exists between two kinds of objects, e.g., between fingers and hands. The ability to express this relationship should be provided through an additional "parent id" which relates to the identifier of the parent object.

The architecture design should be able to process all of these measurements about the previously mentioned types of objects, if the sensor hardware is able to provide them.

Alignment of Motion Data

While three-dimensional user interfaces exist, the vast majority of currently existing systems is based on a two-dimensional output device such as an LCD or a projection screen. Graphics are displayed on such a screen based on pixel coordinates.

The previously described set of data on the motion of various entities used for interaction is generally delivered in the coordinate system of the sensor hardware itself. For a camera-based system, this might be image coordinates, whereas for an electrical field-based sensor, the coordinates will probably refer to conductor rows and columns. If several sensors are used, the data may even be described in one of several overlapping coordinate systems.

From these considerations, a second basic concept for the architecture design can be deduced. It is necessary to translate all previously measured data

into a common reference frame. This process is often referred to as *registration*. The single reference coordinate system which will be available in any scenario is the screen coordinate system. Note that while the output device in its entirety may be composed of several disjoint sub-displays, we will not consider this case here, as a large body of work on merging such multi-display setups into a single, rectangular virtual display exists (e.g., [35]). The same applies to the special case of non-planar displays [10].

To allow easier processing in the following steps, the data should be transformed into pixel coordinates. It is the responsibility of this alignment process to appropriately scale the coordinates if the screen resolution changes.

Recognition of User Intent

Assuming that the previously described steps have been performed, all relevant data about the user's (or users') motions are now available within a single, screen-aligned frame of reference. The next logical step now is to assign meaning to these movements. In a slightly more narrow context, this process is often also called gesture recognition.[2]

We have now arrived at an especially crucial point. For this translation step, the motions to be recognised have to be described to the system in a machine-readable way. Some existing systems present a catalogue of predefined gestures from which the ones to be recognised can be selected. However, this approach is inflexible when considering varying hardware platforms with different input capabilities. Therefore, this concept is probably the most complicated one which the proposed architecture has to provide and will be described in detail in section 3.3.

Visual Feedback

The fourth and last step is to close the feedback loop with the user by generating graphical output. The displayed content should react to the user's intentions which have been interpreted by the previous processing step. This goal can be achieved in a variety of ways, e.g., by re-using the graphical components from an existing GUI toolkit or through drawing libraries such as OpenGL.

[2] Although this implies a focus on gestures only as we use them in everyday conversation, this term will be used for the broader task of translating motion into semantic entities throughout the rest of the document. From here on, the term "gesture" will therefore describe any interaction triggered through the user's movement.

One additional subtask that is part of this concept is to deal with the fact that the previously described gestures are not guaranteed to be valid over the entire interaction area. Rather, it is to be expected that each type of GUI element is sensitive to one set of gestures. Therefore, the areas covered by specific GUI objects and the associated gestures have to be communicated to the gesture recognition task described previously.

3.1.2 Architecture Design

Each one of the four previously described conceptual steps can be mapped to one "slice" in a layered architecture which is presented in figure 3.1. The data flowing through the layers from bottom to top decreases in quantity while at the same time gaining in information density and semantic content.

The four layers are modelled on the concepts which have been identified in the previous section, and are ordered from bottom to top as follows:

Hardware Abstraction Layer (HAL). This layer performs the task of retrieving motion data about the user's actions. As different hardware may have very different capabilities, it translates the data into the seven atoms mentioned in section 3.1.1.

Transformation Layer. The transformation layer acts as a filter that transforms the motion data received from the HAL into the screen coordinate space of the display. The steps necessary to achieve this task may be a combination of such operations as homogeneous transformation, radial undistortion or a simple vector translation.

Interpretation Layer. Probably the most difficult task, the interpretation of motion data, is accomplished by this layer. In addition to the screen-aligned motion data, it also receives region and gesture definitions by the layer above. Moreover, as different types of hardware may need to use different methods of interaction, a capability description for the specific kind of sensor in use can also be loaded.

Widget Layer. Finally, the user-visible output and therefore the final part of the feedback loop between user and interface is formed by the widget layer. Basically, any graphics library or toolkit can be used to form this final part of the stack.

Between the layers, two types of communication protocols are employed. The first one is used to transport motion data from the HAL through the

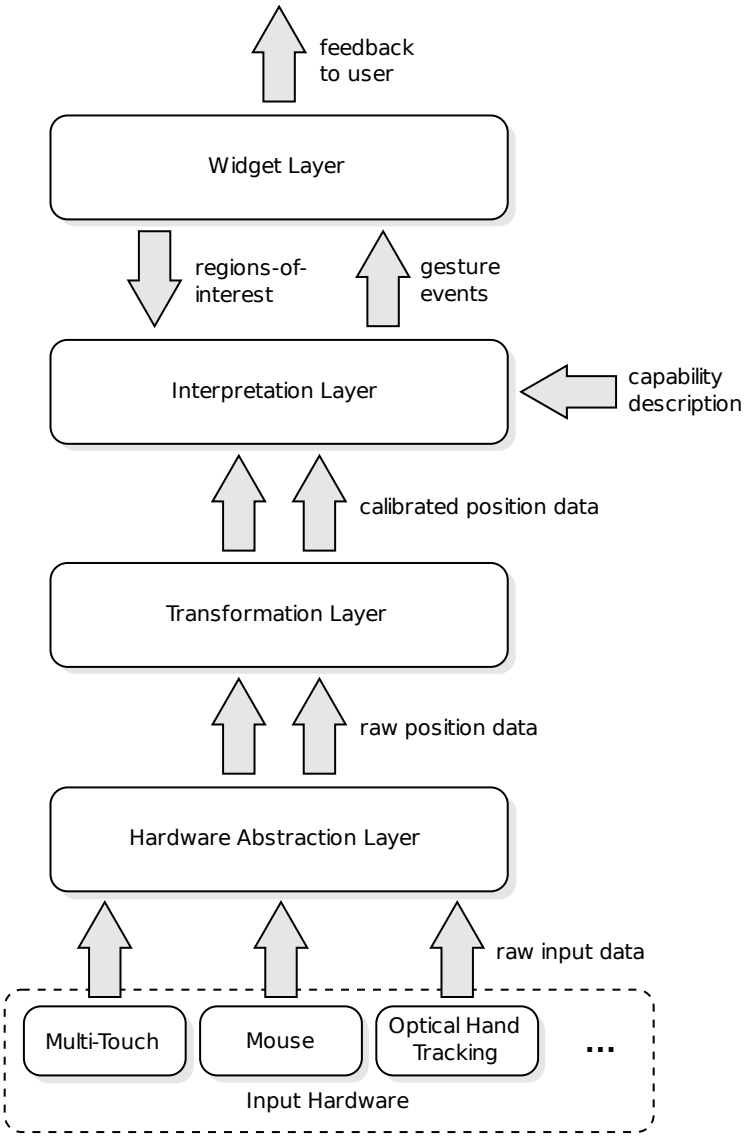


Figure 3.1: Overview of the four architecture layers

transformation layer to the interpretation layer, while the second one is used between interpretation and widget layer in both directions to describe regions and gestures as well as deliver recognised events. Both protocols will be described in detail in the following sections.

But why is it necessary at all to strictly separate these tasks into inde-

pendent layers? The reason is twofold. First, a clear and formal definition of abstract entities is required in order to communicate between these layers. While a monolithic solution would perhaps not require these clear definitions and could therefore possibly be implemented more quickly, the long-term goal of improving reusability will benefit from clear conceptual separation. The second reason is that interoperability will be easier to achieve when the layers are largely independent from each other. For example, a new widget layer can be created without any restrictions as to which programming language or graphics engine should be used for the task as long as it adheres to the protocol specifications given below.

3.2 Transport of Motion Data

3.2.1 Design Considerations

Between the three lower layers of the presented architecture, motion data of the various types described in section 3.1.1 has to be delivered. Between the two lowest layers, this motion data is still in sensor coordinates. After it has passed through the transformation layer, it has been converted to reference coordinates. However, the same protocol can be employed regardless of the reference frame used, as long as the two protocol streams can be addressed individually.

One aspect which has to be kept in mind is that of bandwidth. For example, while the exact shape of an object may be useful in some applications such as recognition of arbitrarily shaped tangible objects, describing it accurately will require a significant amount of data. Even when only the outline of each object is delivered, this expands to a list with hundreds of positions even for small objects. Moreover, non-optical sensors may only be able to detect a very rough outline that does not contain much information about the true shape of the object in question. For these reasons, the shape of the object is best described by an approximation. There are several possibilities for such an approximate description. The most straightforward one is a sequence of lines along roughly linear parts of the shape. A simpler approach, however one with the added benefit of also giving an estimate on the object's rotation, is the equivalent ellipse. This is usually given by two or three direction vectors which are orthogonal with respect to each other and describe the axes of the ellipse.

A large part of the data atoms required here are already present in the Tuio protocol [68]. However, two crucial aspects are not available: the point-of-

interest and the parent-child relationship. Moreover, Tuio is a binary protocol which is slightly more efficient, but not human-readable and therefore more difficult to debug. It also requires an external parser for the underlying OSC protocol. On the other hand, Tuio has already been extensively used in existing systems. Based on these considerations, the *Location Transport Protocol (LTP)* was designed as a clear-text protocol which is compatible with Tuio plus additional data fields. To gain interoperability with the existing body of software, an adapter was written which converts between LTP and Tuio in both directions.

3.2.2 Location Transport Protocol

LTP is composed of short, self-contained messages. Each message describes a sensor reading for a particular object. In order to synchronise sender and receiver, a special "frame" message can also be sent which indicates that a new sensor reading has been taken, e.g., a camera image. This message is then followed by an arbitrary number of location messages. Therefore, the specification of LTP in *extended Backus-Naur form (EBNF)*[3] notation is as follows:

```
      <message>  ::=  <frame_msg> | <location_msg>
    <frame_msg>  ::=  'frame' <int:framenum>
<location_msg>   ::=  <objecttype> <vector:position>
                      <double:size> <int:id>
                      <int:parent_id> <vector:peak>
                      <vector:axis1> <vector:axis2>

       <vector>  ::=  <double:x> <double:y>
   <objecttype>  ::=  'finger' | 'hand' | 'object' |
                      'blob' | 'other'
```

The object type can be one of five identifiers:

`'finger'` for objects that can unambiguously be identified as fingers, e.g. contacts on an FTIR surface,

[3] We will assume the non-terminal prefix symbols `<int>`, `<double>`, `<string>` etc. to be already defined as they are in most programming languages. To enhance clarity, they can be followed by an additional descriptor, separated by a colon.

`'hand'` for motion sensors which can reliably detect and identify the user's hands,

`'object'` for clearly identifiable tangible UI elements, e.g., those tagged with fiducial markers,

`'blob'` for objects detected by systems which can only reliably capture an outline, e.g., a shadow and

`'other'` for any other kind of entity.

Depending on the sensor as well as the object type, the rotation of the object relative to the reference coordinate system can sometimes not be uniquely determined. Therefore, the more general representation of an equivalent ellipse will be delivered, which is composed of two axes that are by definition perpendicular to each other. Should the sensor-object combination support unambiguous determination of the current rotation, for example in a camera-based setup with fiducial markers, then the first axis will represent the direction of the object's local x axis and the second axis that of the local y axis. Otherwise, the two axes will span an ellipse approximating the object's shape and orientation.

An example LTP data stream therefore looks as follows (each line corresponds to one message):

```
frame 58
hand    524.19 271.58  397  52  0  536.05 300.98  4.84 1.88  0.62 -1.61
finger  528.71 294.36   64  15 52  534.25 300.90  1.51 1.18  0.39 -0.50
frame 59
hand    524.37 272.07  388  52  0  536.05 300.98  4.64 1.73  0.57 -1.54
finger  528.87 294.30   62  15 52  532.32 302.63  1.22 0.86  0.28 -0.40
frame 60
hand    524.07 271.51  398  52  0  536.05 300.98  4.92 1.83  0.60 -1.64
frame 61
frame 62
```

Note that after frame 59 the "finger" object with ID 15 vanishes, and shortly after its parent "hand" object with ID 52. In frames 61 and 62, no objects have been detected.

3.3 A Formal Specification of Gestures

When traversing the layer stack from bottom to top, we have now arrived at the last and most crucial interface, which is between the interpretation and widget

layers. As mentioned above, communication at this point is bidirectional. First, the widget layer needs to specify screen regions and the gestures which are to be recognised within them. Afterwards, the interpretation layer will notify the widget layer when one of the previously specified gestures has been triggered by the user.

3.3.1 Widgets and Event Handling

Before discussing the specification of gestures resp. events, we will briefly examine how widgets and events are handled in common mouse-based toolkits. Here, every widget which is part of the user interface corresponds to a *window*. While this term is mostly applied only to top-level application windows, every tiny widget is associated with a window ID. In this context, a window is simply a rectangular, axis-aligned area in screen coordinates which is able to receive events and which can be nested within another window. Due to this parent-child relationship between windows, they are usually stored in a tree.

Should a new mouse event occur at a specific location, then this tree is traversed starting from the root window which usually spans the entire screen. Every window is checked whether it contains the event's location and whether its filters match the event's type. If both conditions are met, the check is repeated for the children of this window until the most deeply nested window is found which matches this event. The event is then delivered to the corresponding handler of this window. This process is called *event capture*.

However, there are occasions where this window will not handle the event. One such occasion is, e.g., a round button. Events which are located inside the rectangular window, but outside the circular button area itself should have been delivered to the parent instead. In this case, the button's event handler will reject the event, thereby triggering a process called *event bubbling*. The event will now be successively delivered to all parent windows, starting with the direct parent, until one of them accepts and handles the event. Should the event reach the root of the tree without having been accepted by any window, it is discarded.

When we now compare this approach to our previously presented architecture, one fundamental difference is apparent. Instead of one single class of event, we are dealing with two semantically different kinds of events.

The first class is comprised of *input events* which describe raw location data generated by the sensor hardware. These events are in fact quite similar to common mouse events. However, if we were to deliver these events directly

to the widgets, no interpretation of gestures would have happened yet. The widget resp. the application frontend would have to analyse the raw motion data itself.

In the interpretation layer, these input events are therefore transformed into a second event class, the *gesture events* which are then delivered to the widgets. The existence of these two different event classes will influence some parts of the design which will be discussed in the following section.

Before we arrive at this discussion, the following question should be asked: why are existing concepts such as widgets and events used here instead of a radically new concept? This question is easily answered, as these existing concepts have the significant advantage of being known to a vast number of developers. As the primary goal of this approach is to make it easier for developers to build an interface based on novel input devices, building on established and widely known practices is a reasonable choice.

3.3.2 Abstract Description of Gestures

As no artificial restrictions should be imposed on the developer as to which gestures are available, a generic and broadly applicable way of specifying them has to be found. To this end, the three abstract concepts of *regions*, *gestures* and *features* shall now be introduced. Their relationships are shown in figure 3.2.

From an abstract point of view, regions are on-screen polygons which correspond to widgets. A region can contain an arbitrary number of gestures which are only valid within the context of this region. Gestures can be shared between regions and are then valid in all containing regions. A gesture itself is composed of one or more features. Features are simple, atomic properties of the input objects and their motions which can in turn also be shared between gestures. Each of these features can be filtered through setting boundary conditions. Should all features of one gesture match their respective boundary conditions, then the gesture itself is triggered and delivered to its containing region.

Regions

The primary task of regions is to assign input events to their corresponding widgets. As it is the case with any regular GUI, a gesture-based interface can also be assumed to be divided into nested areas. In a mouse-based UI, these

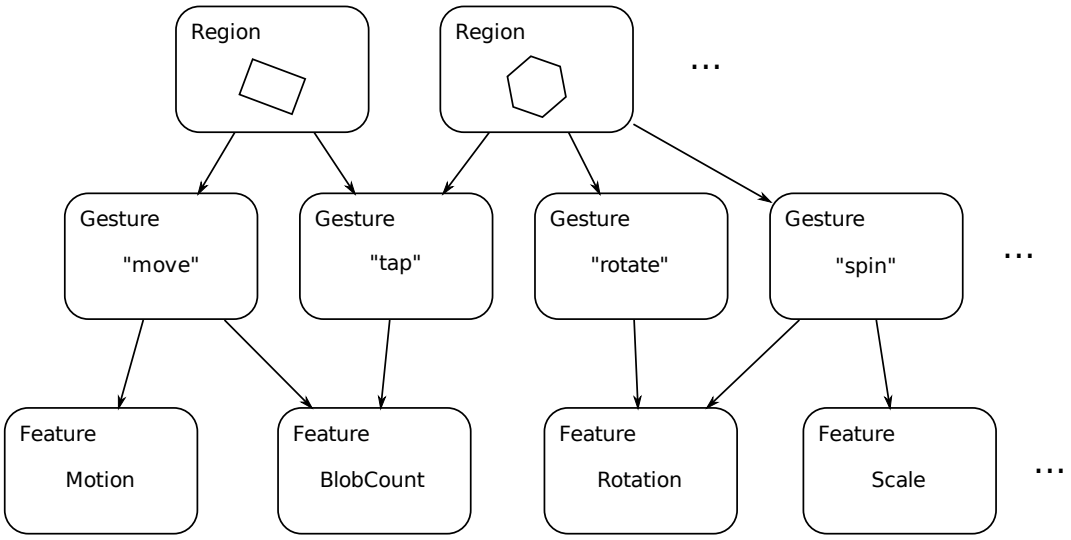
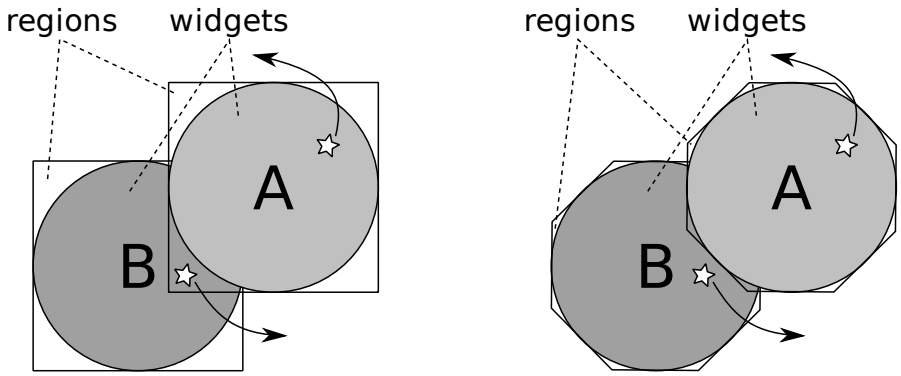


Figure 3.2: Relationship between regions, gestures and features

areas are called *windows* as described above. When moving to the presented, more general approach to user interfaces, this concept needs to be extended. For example, the fixed orientation and axis alignment is insufficient when considering table-top interfaces, e.g., a round coffee table.

Therefore, a region is defined as an area in screen coordinates which has a unique identifier and is described by a closed, non-intersecting polygon. Regions are managed in an ordered list, with the first region in the list being the topmost region on screen. This means that lower regions can be totally or partially obscured by those on top.

But why do regions need arbitrary shapes? Wouldn't a simple rectangle still be sufficient? The answer to these questions is more complicated than it seems at first glance. Consider two overlapping widgets as shown in figure 3.3(a). In a standard toolkit, the input event which was erroneously captured by widget A could simply be "bubbled" back to widget B. However, in the presented architecture, the input events are converted to gesture events before being delivered to the widgets. The two input events would merge into one gesture event which cannot be split back into the original input events. Where should this single event now be directed to? The solution is therefore to ensure that input events are always assigned to the correct widget in the first place. The most straightforward way to achieve this goal is to allow regions of arbitrary shape which can closely match the shape of the corresponding widget as shown



(a) with rectangular regions (b) with arbitrary regions

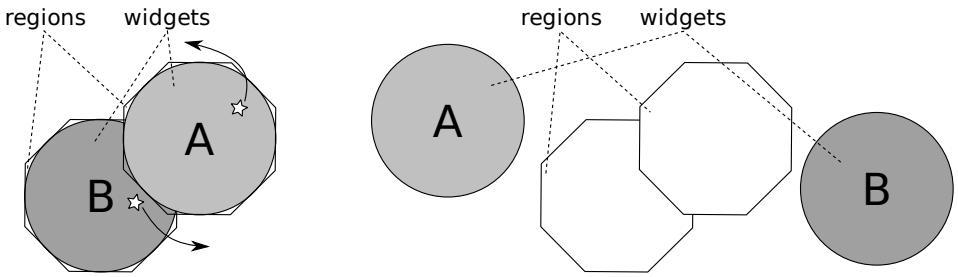
Figure 3.3: Overlapping widgets capturing input events

in figure 3.3(b).

Besides having arbitrary shape, regions can also further select input events based on their object type. This behaviour is realised through a number of flags, one for each of the object types mentioned above. When one of these flags is set, the region is sensitive to input events from this object type. If the flag is cleared, the region is transparent to this type of input event.

In addition to these object type flags, a region can also be flagged as *volatile* to describe that its location or shape may change without user interaction. Why is it necessary to explicitly mark such regions? In this context, it is important to consider that every region actually has two representations: one in the widget layer which describes the current state of the widget's graphical representation, and one in the interpretation layer which represents the state of the widget when it was last transmitted. These two representations are not necessarily synchronised (see figure 3.4). Keeping them synchronous even while the user is modifying the widget would generate significant amounts of traffic between the two layers. Depending on the communications channel, this may lead to dropouts and other undesirable behaviour. Therefore, some cases such as "self-modifying" regions may require special handling to update the secondary representation of the region at the correct moment. Such behaviour is sometimes called *lazy updates*. Other aspects of this issue will be discussed below.

At runtime, the input events described in the previous section are checked



(a) before movement (b) after movement

Figure 3.4: Desynchronisation of widgets and regions

against all regions, starting from the first one. When the location falls inside the region and the flag for the corresponding object type is set, this input event is captured by the region and stored for subsequent conversion into gesture events. Otherwise, regions further down are checked until a match is found. When no match occurs, the input event is finally discarded.

Gestures

The core element of this formalism are *gestures*. An arbitrary number of gestures can be attached to every region. These gestures can either be created from scratch or taken from a list of predefined default gestures.

At runtime, these gestures can then be triggered by the input events which are captured in the containing region. Should the conditions for one or more specific gestures match, an event describing the gesture is delivered to the containing region and therefore to the widget whose outline is described by the region. A gesture is composed of a name, a number of flags and one or more features which will be detailed in the next section. The name can either be an arbitrary descriptor chosen by the developer for custom gestures, or one of a list of predefined "common" gesture names. In the latter case, no features need to be specified, as these are part of the existing definition.

Additionally, two flags can be set to further differentiate the behaviour of the gesture. When the gesture is marked as *one-shot*, then it will only be sent once for a specific set of input objects. For example, consider a "tap" gesture which is to be triggered when the user touches a region. The corresponding event should only be delivered once after the first input event has occurred,

not subsequently while the user continues to touch the region. In this case, setting the one-shot flag will ensure the desired behaviour.

The gesture can also be flagged as *sticky*. In this case, input events which started the gesture will continue to be sent to the originating region, even if they move outside of the region's boundaries. The region can be said to capture the *identifiers* of these input events, not only their locations. This prevents gestures from stopping abruptly when the respective location events leave the area where they started. This is a consequence of the previously described dual, asynchronous representations of regions. Without this "stickiness", the region would have to be updated continually after every single event, thereby creating excessive protocol traffic.

Finally, the gesture can have the *default* flag set. Should such a gesture be received, its name and features will be added to the list of standard gestures which can be accessed using only their name. This allows applications to register their own custom gestures for reuse among several widgets or to overwrite the definitions of the standard gestures given below.

Currently, 5 predefined standard gestures are available. These gestures and their semantics are as follows:

tap - triggered once when a new input object appears within the region

release - triggered once when all input objects have left the region

move - sent continuously while the user moves the region

rotate - sent continuously while the user rotates the region

scale - sent continuously while the user scales the region

Note that the actual features which comprise these gestures are not given here. The reason is that these features may differ significantly depending on the sensor. For example, on a camera-based touchscreen, rotation can be achieved by turning a single finger, whereas a capacitive sensor will require at least two fingers rotating relative to each other. However, this is irrelevant for the semantics of the resulting gesture - the intention of the user stays the same. Therefore, the composition of these default gestures can be redefined dynamically depending on the hardware used.

Features

As mentioned previously, gestures themselves are composed of *features*. Every feature is a single, atomic property of all input events that have been captured by a region. Examples for such properties are the average motion vector or the total number of input objects. A feature can appear in one of two variants: as a *feature template* when it is sent to the interpretation layer and as a *feature match* when it is later sent back to the widget layer. Both variants never appear as standalone entities, but only as components of a gesture.

By sending a gesture composed of one or more feature templates, the widget layer specifies what properties the motion data must have in order to trigger this gesture. When these conditions are later met, the actual values of these properties are sent back within the gesture as feature matches.

A feature is described by a name, type flags, optional boundary values and a result value. The name describes the specific kind of feature, i.e., which class is responsible for handling the feature calculation. The type flags are similar to those already described for regions. For every type of input object, one flag is present. If this flag is set, then input events of this type are incorporated into the feature calculation. While the flag settings on a region provide a first high-level filter that determines which input events are captured at all, the flags on each feature provide a more fine-granular control over which events are actually used for calculating this specific feature.

Depending on the class of feature, one or more boundary values can be given in a feature template that limit the value which the feature itself is allowed to take. For example, a feature with a single numerical result can have a lower and an upper boundary value. Note that the boundary values always have the same type as the result value itself. After the value of a feature has been calculated, it is checked against the boundary values if they are present. Should the value of the feature fall within the specified range, the feature template changes into a feature match which has a valid result value.

Features can be divided into two groups: single-match and multi-match. Single-match features have a single result value for the entire region, such as the average motion vector. Multi-match features, on the other hand, can have several result values, usually up to one result per object inside the region. Why is this distinction necessary? As an example, consider a hypothetical user interface which should display a tile that can be moved by the user when touched and dragged. Additionally, every single touch location on the tile should be

highlighted to provide additional visual feedback. For the motion information, a gesture that contains a single-match feature providing the average motion vector is sufficient. The individual motion vectors are not needed. However, for displaying the touch locations, the individual coordinates have to be delivered. The respective gesture has to contain a multi-match feature representing the object locations. Should this region be moved with, e.g. three fingers, then every movement will trigger one motion event and three location events.

Conceptually, both types of features are used in exactly the same way; the only difference is that a gesture which is composed of multi-match features can be triggered several times within a single frame of sensor data. Note that while mixing single- and multi-match features within a single gesture is possible, it is unlikely to have the desired effect, as only one single result will be produced.

We will now briefly describe the currently available features.

Single-Match Features:

ObjectCount This feature counts the number of input events within the current region. E.g., if the appropriate filters for finger objects are set and the user touches the region with two fingers, this feature will have a result value of 2. A lower and upper boundary value can be set.

Motion This feature simply averages all motion data which has passed the filters and gives a relative motion vector as its result. Two boundary vectors can be specified which describe an inner and outer bounding box for the result vector. This can be used, e.g., to select only motions within a certain speed range.

Rotation In this feature, the relative rotation of the input events with respect to their starting position is calculated. This feature itself is a superclass of two different kinds of sub-features. The first subfeature, *MultiObjectRotation*, can only generate meaningful results with two or more input objects and extracts the average relative rotation with respect to the centroid of all event locations. The second subfeature, *RelativeAxisRotation*, requires only one input object, but needs a sensor which is able to capture at least the axes of the equivalent ellipse of the object. The average relative rotation of the major axes of all input objects is extracted. In both cases, the result value is a relative rotation in radians which can again be constrained by two boundary values that form lower and upper limit.

Scale Similar to *Rotation*, this feature calculates the relative change in size of the bounding box and has the corresponding scaling factor as a result. This feature also has two optional boundary values which serve as lower and upper limit.

Multi-Match Features:

ObjectID The results of this feature are the IDs of all input objects within the region that have passed the filters. Two boundary values can again be specified to constrain the results to a smaller subset of IDs, e.g., to filter for specific tangible objects with previously known IDs.

ObjectParent This feature is similar to *ObjectID*, but returns the *parent ID* of each input object instead of the object IDs themselves. To receive both IDs for all objects, this feature can be paired with *ObjectID* in a single gesture.

ObjectPos The results of this feature are the positions vectors of all input objects. This feature currently does not have any additional boundary conditions.

ObjectDim This feature has a special result type called *dimensions*. This is similar to the shape descriptor used in LTP and gives an approximation for the outline and orientation of an object through its equivalent ellipse. Two optional *dimension* objects can be given as boundaries, specifying upper and lower limits for each component of the shape descriptor. This filter allows to select, e.g., only blobs of a certain size and height/width ration.

ObjectGroup This feature generates a match for each subset of input objects which can be grouped together in a circle of a specified radius. The result is a vector containing the centroid of one group. Two boundary values can be given, with the first component describing the minimum number of objects and the second component determining the radius of the circle.

Examples

To give a better understanding of how these concepts work, the decomposition of some gestures into features shall now be discussed. The five standard gestures mentioned previously can easily be mapped to a single feature each, e.g., the “release” gesture consists of an *ObjectCount* feature with both lower

and upper boundary set to zero. As the *one-shot* flag of the gesture is also set, this results in a single event as soon as the object count (e.g., finger contacts inside the region) reaches zero.

Another important mapping is that of the "move", "rotate" and "scale" gestures which contain a single *Motion*, *Rotation* and *Scale* feature, respectively. Note that a freely movable widget which uses all three gestures will behave exactly as expected, even though the raw motion data is split into three different entities. Consider, for example, rotating such a widget by keeping one finger fixed at one corner and moving the opposing corner with a second finger. In this case, the widget rotates around the fixed finger, thereby seemingly contradicting the definition of the *Rotation* feature which delivers rotation data relative to the centroid of the input events. However, as the centroid of the input events itself also moves, the resulting motion events will modify the widget's location constantly to reflect the expected behaviour.

While a large number of interactions can already be modelled through single features and carefully selected boundaries, combining several different features significantly extends the coverage of the "gesture space". For example, a user interface might provide a special gesture which is only triggered when the users quickly swipes five fingers across the screen. This can easily be described by the combination of an *ObjectCount* feature with a lower boundary of five and a *Motion* feature with a lower boundary equal to the desired minimum speed.

3.3.3 Gesture Description Protocol

Control Flow

In this section, the formal specification of the *Gesture Description Protocol (GDP)* which is used between the two topmost layers shall now be detailed. From a high-level point of view, the protocol flow follows the left side of the diagram shown in figure 3.5. To provide a temporal context, relevant LTP messages are also shown on the right side.

As mentioned previously, the top-level object used in this protocol is the *region* which describes a screen area along with associated gestures. The lifecycle of a region consists of five distinct stages which are also shown in figure 3.5.

Registration. When a new widget is added to the user interface, the widget layer registers the corresponding region with the interpretation layer. A unique, non-zero numerical identifier is sent along which will later be used by both layers to refer to this region.

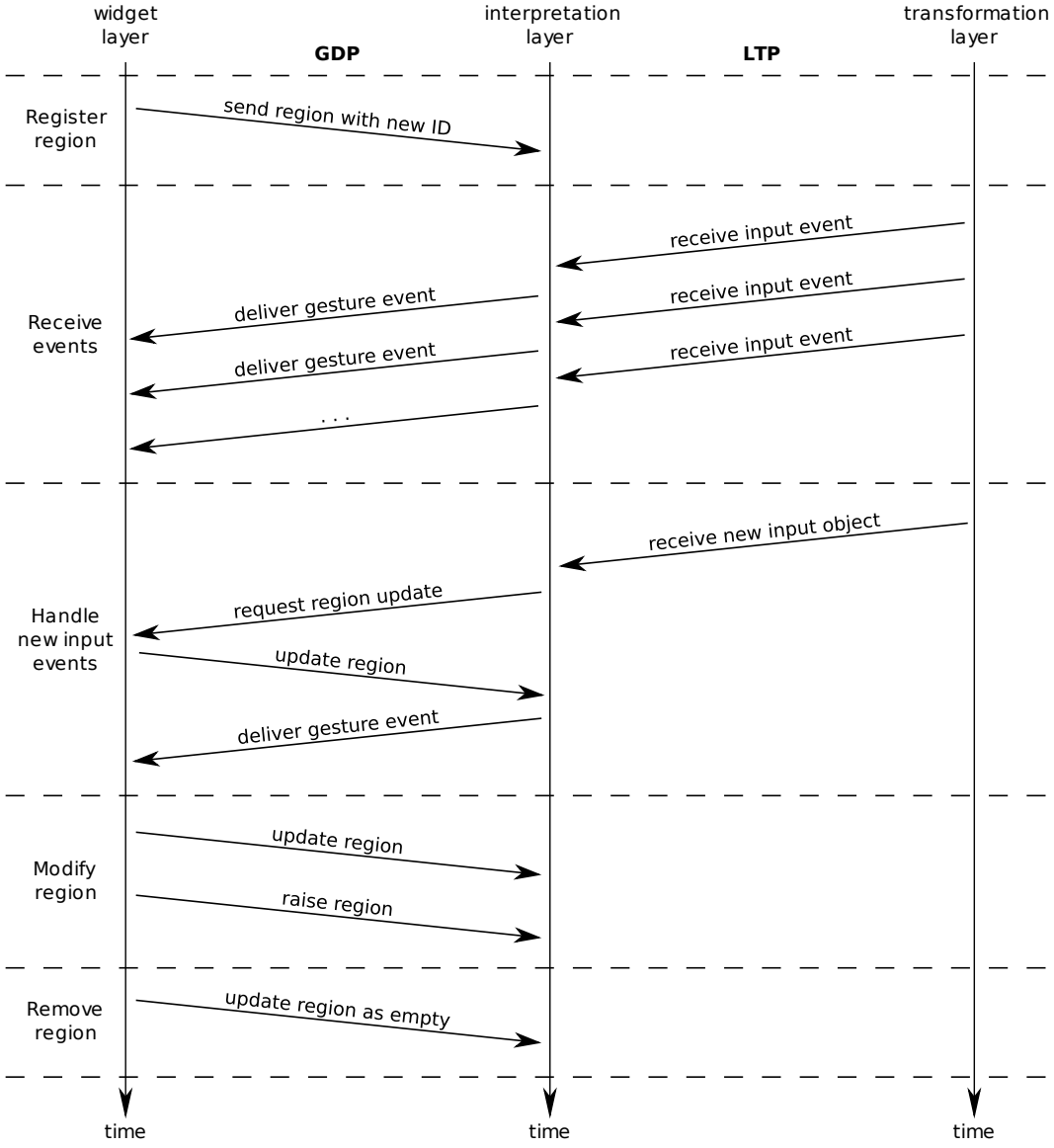


Figure 3.5: Protocol flow

Receiving continuous events. When input events arrive which fall within the region and trigger a gesture, this event will be delivered to the widget layer along with the identifier of the region. Depending on the type of gesture, this event may be sent only once or continuously as long as the relevant conditions are fulfilled. Note that if the gesture is flagged

as *sticky*, events will continue to be sent even if the input objects leave the original boundaries of the region. This is especially important when considering, e.g., movable widgets or generally widgets whose region may change due to user interaction. Without use of the sticky flag, the region would have to be updated after every event, thereby leading to an additional round trip between the two layers for every single event which would significantly increase lag. When using the sticky flag, however, even very fast user movements will continue to be delivered to the correct region without the need to send an updated region outline.

Reacting to new input data. While the previously described behaviour is necessary to provide a smooth user experience, special handling is necessary when a new input ID arrives. When a region is modified through gestures as described in the previous step, the graphical representation will already have adapted to the new conditions, while the region which has been transferred to the interpretation layer still has the old position and shape - the two layers are out of sync. This is not a problem as long as no new input events arrive or only input events whose ID has already been associated with a sticky gesture. However, if an input event with a new, unassociated ID appears, the current locations of certain regions need to be retrieved. Otherwise, the input event might miss the region which it is destined for, as the stored position differs from the graphical representation. To counter this problem, the interpretation layer sends a message requesting updates for specific regions to the widget layer. An update is requested for those regions which are flagged as *volatile* and for those regions which contain one or more sticky gestures, as the two representations for these kinds of regions may be out of sync.

Modifying a region. This step already appeared as part of the previous one, but can also be triggered by the widget layer on its own. It is very similar to the registration step with the only difference being that a previously sent identifier is used again. The internal representation of this region within the interpretation layer is then overwritten. This is useful, e.g., to change the behaviour of a widget by registering a different set of gestures. A variant of this step is to leave the region itself unchanged, but *raise* it to the top of the region stack. Of course, this behaviour needs to be mirrored by the graphical representation in the widget layer.

Removing a region. Again, this step is similar to the previous one. When a widget is removed, its corresponding region can be unregistered with the

interpretation layer by updating the region to have no boundary points. This empty region will not match any input events and can therefore be removed altogether.

Protocol Specification

For describing the protocol itself, we will start with the smallest entity, the *feature*, and construct the larger entities consecutively.

```
        <feature>  ::=  <template> | <match>
       <template>  ::=  <feature_header> <type:boundary>*
          <match>  ::=  <feature_header>

<feature_header>  ::=  <string:class> <int:has_result>
                       <int:flags> <type:result>
                       <int:boundary_count>
```

The `class` string specifies which type of feature is transmitted here. It may contain one of the descriptors listed in section 3.3.2. Note that this also implicitly defines which data type is used for the result and boundaries. The next item, `has_result`, describes whether this feature is a *template* (has_result = 0), i.e. a specification sent by the widget layer, or a *match* (has_result = 1), i.e. contains a result determined by the interpretation layer. After the result marker, a bitfield of flags specifies which types of input objects this feature will react to. This is followed by the `result` itself, which has a type that is determined by the feature class. If a template is transmitted, the result must still be present, but will be discarded. The next item is `boundary_count`, which specifies how many boundaries will now be transmitted. Finally, this is followed by as many `boundary` values as determined by boundary_count. However, this is only the case for templates - matches never contain any boundary values, as they have no meaning in this context. Therefore, matches and templates share the same header up to the boundary list.

The next entity to be considered is the *gesture*, which is composed of features. The specification is therefore quite straightforward:

```
<gesture>  ::=  <string:name> <int:flags>
                <int:feature_count> <feature>*
```

The `name` of the gesture can be freely chosen. However, if it matches a predefined name such as one of those described in section 3.3.2, then the feature count can be set to zero and the rest of the gesture specification will be read from the pool of predefined gestures. In any other case, the `flags` field first defines the behaviour of the gesture. It is composed of three flags: *sticky*, *one-shot* and *default*, whose meaning has already been described previously. Particularly if the *default* flag has been set, the gesture will also be added to the pool of predefined gestures.

Finally, the `feature_count` field specifies how many feature specifications will follow. It is important to note that a gesture will always be sent back to the containing region and that its specification is only valid in the context of that region. Therefore, different gestures with the same name can exist in different regions.

A gesture can be transferred in both directions, again either as template or as match. The differentiation happens solely by transmitting the features themselves either as templates or as matches. Note that mixing template and match features within a single gesture will result in undefined behaviour.

Finally, we will now specify the *region* descriptor which is slightly more complex, as a region consists of its own boundary as well as of a list of gestures.

```
<region>  ::=  <string:id> <int:flags>
               <int:point_count> <vector>*
               <int:gesture_count> <gesture>*
```

The first element of every region is a unique `id` which can be freely chosen by the application. The same identifier will be later used to deliver gestures to this region or to update its description. This identifier should follow the usual C language conventions, i.e. contain no spaces or special characters. The following `flags` parameter determines which types of input objects this region is sensitive to, as well as whether the region is marked as *volatile*, i.e. may change without user input. Afterwards, the region itself is specified as a list of `point_count` vectors which describe a polygon. The last point in the list is assumed to be connected to the first one, thereby always resulting in a closed polygon. Specifying a self-intersecting polygon is not supported and may result in undefined behaviour. Finally, the gestures for this region are specified in the same manner, as a list with `gesture_count` elements.

Note that when a region is registered for the first time, it is inserted at the top of the region list and is therefore the first one to be checked for matches

with input data. Consequently, regions which are registered later may partially or completely cover the previous region. Depending on the users' interactions, it may later be necessary to raise certain obscured regions to the top again. This can be achieved by sending the corresponding message followed by the identifier of a region which will be moved to the top of the region list.

After all building blocks of the protocol have been described, the global protocol specification shall now be given. This includes messages from the widget layer (client) to the interpretation layer (server) as well as in the opposite direction. Usually, one gesture recognition task will continuously run on an interactive surface while several different client applications sequentially connect to this server.

```
<client_message>  ::=  <region_msg> | <raise_msg> | <quit_msg>
    <region_msg>  ::=  'region' <region>
     <raise_msg>  ::=  'raise' <string:id>
      <quit_msg>  ::=  'bye'

<server_message>  ::=  <update> | <event>
       <update>   ::=  'update' <string:id>
        <event>   ::=  'gesture' <string:id> <gesture>
```

The client mainly sends two messages with the purpose to modify the server's copy of the currently active regions. For creating a new region or modifying an existing one, the string `'region'` is prepended to the region descriptor itself to uniquely identify the kind of message. The second type of message consists of the string `'raise'` followed by a region identifier. Should a region with this identifier have been registered at the server, it becomes the topmost one. The client can also send a message (`'bye'`) indicating its termination, which will cause the server to erase all regions and wait for a new client connection.

On the other hand, the server can send two different types of messages. The `update` message requests an update of the region with the specified identifier from the client, while the `event` message delivers a recognised gesture to one of the client's regions which is also named in the message.

Note that while this protocol does not provide explicit support for multiple clients, this can nevertheless be easily achieved by, e.g., utilising several separate protocol channels such as network streams.

Summary

In this chapter, the fundamental design of our interaction architecture has been presented. The concepts used for modelling interaction have been presented as well as the four layers derived from these concepts. Moreover, a formal specification for gestures has been introduced as well as the communication protocols which are used between the layers.

Chapter 4

Sensor Hardware

In this chapter, the various types of input hardware which have been developed or extended during the course of this thesis will be discussed. First, some fundamental techniques which are applicable to several different setups will be presented. In the following sections, the hardware systems themselves shall be discussed.

4.1 Fundamental Techniques

As mentioned previously, an input device requires some sort of physical sensor which gathers data on the users' actions. When reviewing the related work, it becomes apparent that currently optical sensors, i.e. cameras, are the most common type used for input devices. As cameras have a multitude of other applications, especially in industrial settings, there are some techniques regarding their application which have been extensively used in such industrial vision scenarios. However, these methods have rarely been published in a scientific context. We will therefore review them here instead of in the section on related work.

4.1.1 Synchronised Active Illumination

A significant problem in many machine vision setups is stray light, i.e. light which reaches the camera but does not originate from the objects which the system is designed to detect. Depending on the intensity, this can lead to false-positive detection by creating ghost objects in the image or to false negatives by obscuring the objects themselves. The two main sources of stray light are

the sun and ceiling lamps. While direct sunlight is not relevant in most cases, even ambient daylight on cloudy days may prove to be problematic.

From an abstract point of view, the solution to this problem is to provide illumination for the relevant objects which is continuously more intensive than the rest of the scene, i.e. has a high contrast with respect to the background. However, this proves to be quite difficult at times, as many environments have highly variable lighting intensity. The most obvious approach is to use high-intensity light sources to illuminate the scene while at the same time adjusting the camera sensitivity accordingly. As visible light of high brightness may be distracting or even dangerous to the human eye, light sources in the near-infrared spectrum of 800 nm - 900 nm wavelength are usually used in such a scenario. Unfortunately, almost all sources of stray light mentioned above have very broad emission spectra which include the near-infrared range. Therefore, high-intensity lighting can only alleviate, but not solve the problem altogether. Moreover, there is an upper limit to the brightness which can be achieved with external lighting, as infrared light of very high intensity can still cause damage to the human eye.

A more subtle approach to this problem is that of modulated light [105]. In most cases, the light source which is used in the system to illuminate the relevant objects is composed of LEDs. These have several advantages, such as a clearly defined emission spectrum, high durability and low waste heat generation. Another property of LEDs is that they can be operated in a pulsed mode. While the easiest and most straightforward method is to power them with a constant current according to their specification, there is also the option of applying high currents for short periods of time and letting the LEDs cool off afterwards. By repeating this process fast enough, a seemingly constant light output of equal intensity to the continuous operating mode can be achieved.

It is important to note that during the pulses, the current as well as the intensity of the emitted light can be approximately one order of magnitude higher than during normal operation. This fact can now be exploited to increase overall contrast by modifying the camera exposure time.

In figure 4.1, three possible variants of illuminating a scene are presented. Ambient light intensity is shown in blue, while the intensity of the active light sources is shown in red. The curves are not to scale.

In the first case, which has been described above, the active illumination is powered constantly. While it is usually brighter than the ambient light, the total contrast is still low. As the camera integrates all light hitting the sensor

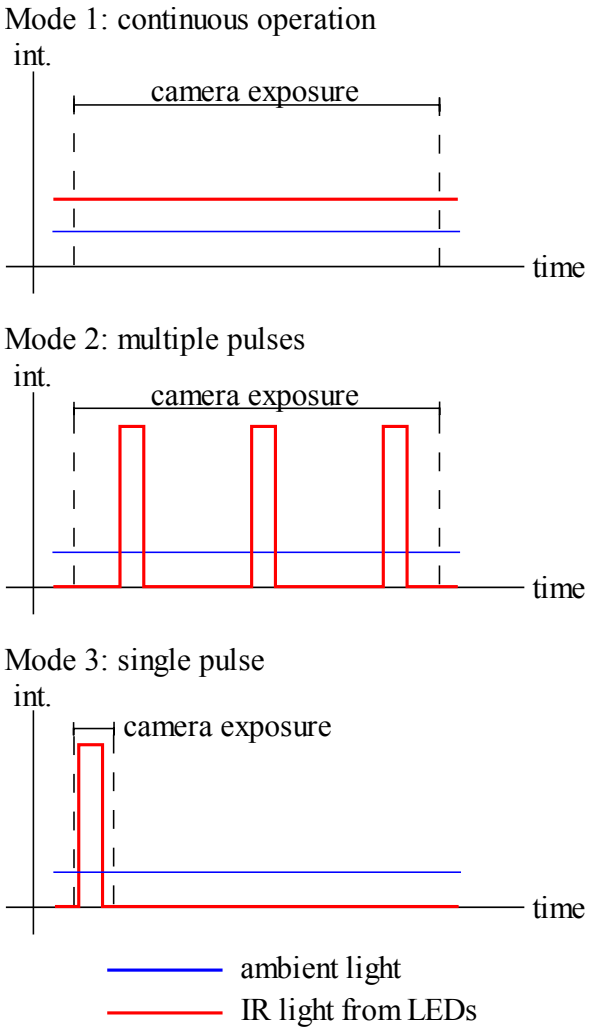


Figure 4.1: Active illumination modes

during the exposure time, the contrast is the relative difference between the integrals of the respective light intensity over time.

In the second case, the active light source is now operating in pulsed mode. The peak intensity is significantly higher than in the first case. However, due to the unavoidable cool-down phase in between the pulses, the integral over the entire exposure time does not change significantly relative to the first case. Therefore, the overall contrast is still not noticeably better.

The third case now circumvents this problem by applying two modifica-

tions [25]. First, the pulsed light source is now synchronised to the camera so that a pulse occurs at the start of exposure of every frame. Second, the camera exposure time is set to the same duration as a pulse, thereby integrating light only during this short period. However, as the active light source is up to one order of magnitude brighter at this point, the contrast also increases dramatically. Special care must be taken to avoid other sources of errors such as reflections, as the very high relative brightness from the active light sources can easily flood the camera sensor, making correct object detection difficult.

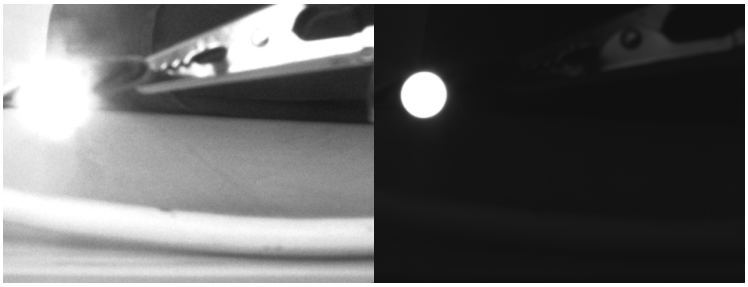


Figure 4.2: Continuous vs. synchronised illumination

Figure 4.2 illustrates the effectiveness of this method. Both sides of the image show the same scene, an LED which is viewed head-on by the camera. On the left side, the LED is powered continuously. The brightness value of the LED itself is 255 which is also the maximum. The average brightness value of the background is approximately 160. Assuming that the LED itself is the foreground object to be detected, this results in a contrast of $1 : 1.6$. On the right side, the LED is pulsed and synchronised to the camera. The foreground brightness is still 255, while the background brightness has dropped to approximately 20, resulting in a contrast of $1 : 12.8$. Although the exact contrast ratio can not be determined from this example, as the illuminated pixels are saturated in both cases, it can safely be concluded that the contrast can be increased at least by a factor of eight ($r = 12.8/1.6 = 8.0$).

When implementing such a setup, an important additional aspect needs to be considered, as the maximum achievable framerate is influenced by the cool-down duration for the LEDs. As it is desirable to maximise LED current as well as pulse duration to achieve the best contrast possible, the cool-down time also increases. However, the summed duration of pulse and cool-down phase must not exceed the time needed for acquisition of one frame, i.e. the inverse of the framerate. To balance these two opposing factors, the datasheet

for the specific type of LED which is used has to be consulted.

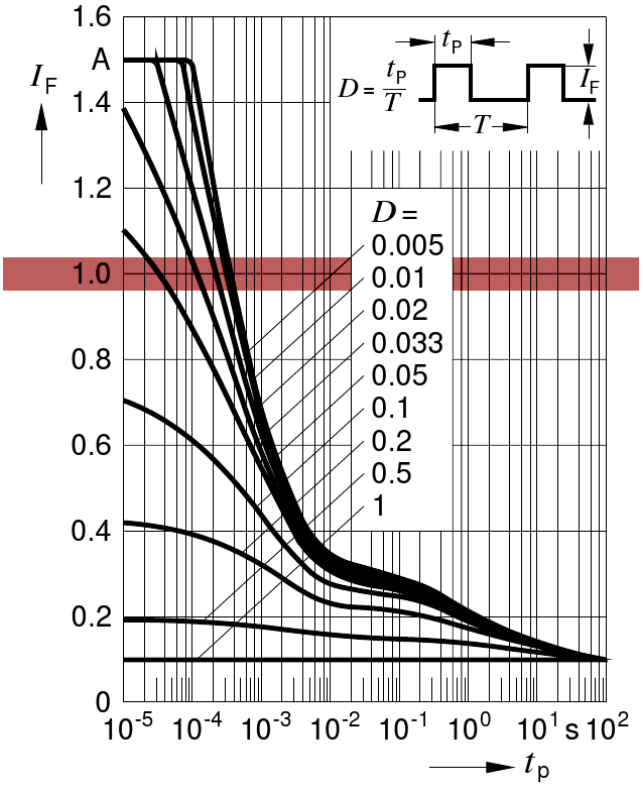


Figure 4.3: LED pulse capacity diagram (image from [91])

In figure 4.3, an excerpt from the datasheet [91] for Osram *SFH4250* LEDs is shown. Each curve represents one possible ratio between pulse and cool-down duration, usually called duty cycle. In many setups, a pulse current of 1 Ampere is used, which is also highlighted in the diagram. Assume a frame rate of $f = 60\,Hz$. Therefore, one full pulse/cool-down cycle must have a duration of $D_{max} = \frac{1}{f} = 16.67\,ms$. Now, the total cycle duration has to be calculated based on the duty cycle for each curve and the allowed pulse duration at a current of 1 A. For example, at a duty cycle of 3.3 %, the pulse duration is approximately $t_P = 120\,\mu s$ for a total cycle duration of $D = 3.6\,ms$. At a ratio of 1 % with a pulse duration of $t_P = 250\,\mu s$, the total duration already rises to $D = 25\,ms > D_{max}$, which is too long. Therefore, a duty cycle of 2 % is the appropriate choice with a pulse duration of $t_P = 200\,\mu s$ and a total duration

of $D = 10\,ms$, which still offers a comfortable safety margin.

When selecting a camera for use in such a setup, it is important to choose a model which is able to provide such short exposure times on the order of 100 microseconds. Moreover, the camera needs to use a global shutter which captures all image pixels at the same time, as synchronisation would otherwise not be possible. A synchronisation output is also necessary.

As an experiment, a common webcam (Logitech *Quickcam 5000*) was modified with such an output. On the main controller chip, an output pin carrying a line-sync signal was identified using an oscilloscope. This signal was filtered using an *LM1881* sync separator *integrated circuit (IC)* to obtain a frame-sync signal. Unfortunately, even extensive tuning of the LED timing relative to the frame signal did not yield satisfactory results, as this webcam uses a rolling shutter which sequentially captures each image line. It is therefore safe to conclude that common webcams can not be used in conjunction with synchronised illumination, as they usually lack the required global shutter and precise exposure controls as well as synchronisation connectors.

Instead, an industrial-grade camera should be selected, preferably one which can be controlled according to the IIDC standard [58] due to its broad software support. A well-suited and relatively affordable example is the *Firefly MV* from Point Grey Research [94].

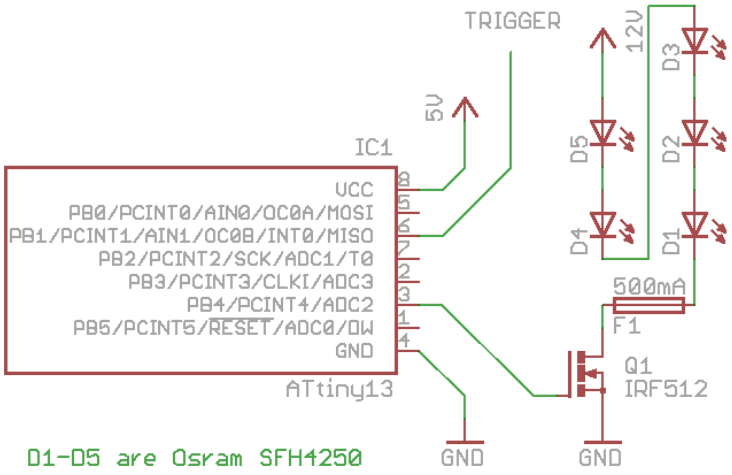


Figure 4.4: LED control circuit

A practical issue which also has to be addressed is how to synchronise the LEDs to the camera. While most industrial cameras have a highly configurable synchronisation output which offers a start-of-frame signal, this output is unsuitable for driving high-powered LEDs directly. Therefore, an additional power switching element has to be added. As this power switch will be interfaced to a digital output, a logic-level *field effect transistor (FET)* is an appropriate choice. One suitable example is the *IRF512* [34]. While it is entirely possible to simply connect the FET to the camera output and use it to directly switch the LEDs, this approach is somewhat error-prone. For example, it is not possible to predict what state the synchronisation output will have while the camera is not capturing images - this is entirely dependent on the camera model. Should the output be continuously active in this case, then the LEDs will also continuously be powered at high current without the necessary cool-down pauses. This will quickly lead to overheating and permanently damage the light source.

In figure 4.4, an extended version of the LED control circuit is shown. While the FET mentioned previously is still used to switch the LEDs, it now is not directly connected to the camera anymore. Instead, a microcontroller has been added as an additional control instance which prevents the LEDs from being powered continuously regardless of the camera output. As an additional safety measure, a fuse has been inserted in series with the LEDs. The rating of this fuse should be selected so that it will immediately blow at the current which flows through the LEDs. While this may sound counterintuitive at first, the fuse will not trigger during normal operation, as the pulses of few hundreds of microseconds are too short to actually trip the fuse. However, should all other safety measures fail, the continuous current will blow the fuse before the LEDs can suffer permanent damage. For reasons of convenience, it is advisable to select a so-called polyfuse which will reset itself after the voltage supply has been turned off [52].

The control IC shown in the above circuit is an *ATtiny13* microcontroller [3]. While this IC adds some complexity to the setup, it is nevertheless necessary due to its real-time capabilities which are usually not provided by off-the-shelf hard- and software. When a trigger pulse arrives from the camera, the LEDs should be activated with as little delay as possible. After the calculated pulse duration, they should be disabled again regardless of any external signals. Moreover, the controller should not allow reactivation before the cool-down period has passed. As these requirements are difficult to realise with non-programmable logic such as a *555* timer [115], a microcontroller is the appropriate choice. While this IC requires a programming device to load its

firmware, this step is only necessary once during setup. An example firmware for this type of controller is described in appendix A.2.

4.1.2 Interleaving Disjoint Light Sources

A second technique which can be employed in the context of infrared sensing is that of interleaving several spatially disjoint light sources. Depending on the setup, different types of visual information may be captured best by different types of illumination. However, the approach of combining several conceptually different light sources to illuminate the setup poses a problem, as the light sources are likely to interfere with each other. Light from source A may very well appear as unwanted stray light when trying to capture light from source B.

One approach to solving this problem is to separate the light optically. For example, infrared LEDs with different emission spectra could be used for the different light sources. Narrow-band filters in front of each camera could specifically select the light which one camera is sensitive to. However, this solution has two drawbacks. First, optical narrow-band filters are difficult to create and usually do not achieve acceptable transmission ratios. Second, this approach requires one single camera for each light source, thereby increasing cost as well as complexity. This approach is therefore only suitable when several cameras are required from the start.

A different, more flexible solution is to interleave the light sources over two or more camera frames. If a synchronisation circuit like the one described in the previous section is available, this approach requires almost no additional hardware. When only one single light source is active during a given camera frame, no interference is possible. Moreover, this solution is usable with only a single camera as opposed to the previous one. When several cameras are used, care should be taken to interleave the camera frames themselves over time, i.e. only one camera is capturing an image at any given moment. Otherwise, interference would again occur. This method has only one drawback when using a single camera, as the effective framerate is reduced in this case and results from the raw framerate divided by the number of disjoint light sources.

When using the circuit described above, it is sufficient to replicate only the part consisting of FET, fuse and LEDs for each light source and connect it to one microcontroller output pin each. By counting the input pulses and thereby the camera frames in the controller firmware, it is possible to enable one from n sets of LEDs for every n-th frame.

An extended variant of this method is available when using a camera which is capable of *high dynamic range (HDR)* imaging. Such cameras can store two or more sets of settings such as exposure time, gain value etc. and automatically switch between them after a frame has been captured. This results in two or more consecutive images which have been captured with completely different settings. When using only one active light source, an example for this method would be to use odd frames with very short exposure times to specifically capture the synchronised illumination, e.g. from an FTIR screen, and to use even frames with normal exposure times to capture an image of the scene as it appears to the naked eye. The odd frames can then be used to detect touch events while the even frames can be used for tracking of fiducial markers. When the camera offers a sufficiently high framerate, the HDR mode can therefore be employed to support two completely different modes of tracking with a single camera. Two consecutive raw images which have been taken with a Point Grey Dragonfly2 in HDR mode are shown in figure 4.5. For the image on the left side, the camera parameters are tuned to reception of light from pulsed infrared LEDs as described above. The LEDs themselves are clearly visible around the display's rim. For the right-hand image, the camera parameters are adjusted to their default settings to capture an image using the ambient light in the room.

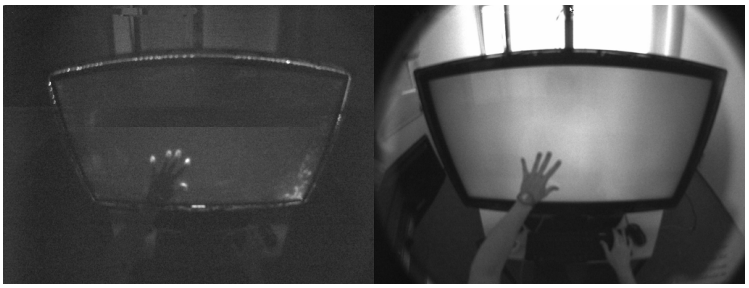


Figure 4.5: Two consecutive images taken with an HDR camera

4.1.3 Using LEDs as Sensors

In all methods described up to this point, LEDs have exclusively been used as light sources, i.e. emitters. However, LEDs offer the additional functionality of also being able to function as light *sensors*. The reason for this behaviour is that from a purely semiconductor-centric viewpoint, an LED does not differ

from a photodiode which is usually employed as a discrete light sensor. While the photodiode is optimised for reception and the LED for emission of light, both types of component can fulfil both roles. Although this capability has been described as early as 1973 [85], it is not widely known.

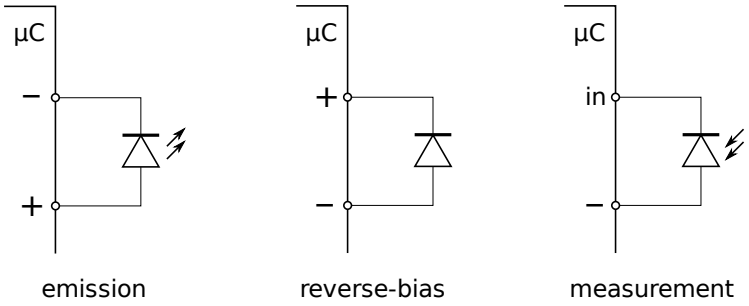


Figure 4.6: LED modes of operation

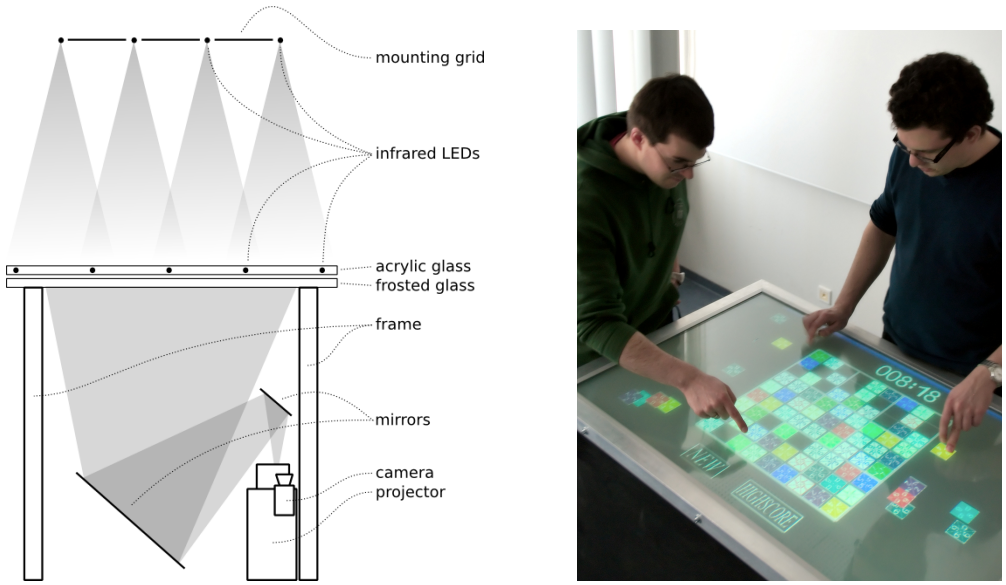
One big advantage of this method is that the same LED can now be used alternately as sensor and emitter. Usually, an LED is connected in conducting direction to a current source which causes it to emit light. In order to sense light instead, two possible methods exist. The first one [95] is to connect the LED to an analog measurement device such as an analog input pin on a microcontroller and directly measure the photocurrent. While this is the most precise method, it quickly requires additional circuit components such as a dedicated analog-digital conversion IC for larger numbers of LEDs and therefore may be quite complex to implement. The second method [16] is to just connect the LED between two digital input-output pins. This allows to drive the LED normally for emitting light, but also to *reverse-bias* it. Under reverse voltage, the LED behaves as a tiny capacitor and stores a small charge, but does not conduct current. When the supply pin is now switched to input mode, this virtual capacitance discharges through the input pin (see figure 4.6). The amount of incident light has an approximately linear influence on the discharge speed, and therefore also on the time until the voltage at the input drops below the trigger threshold. Through appropriate wiring, it is even possible to use off-the-shelf LED matrices alternately as emitter array or as sensor array [56].

4.2 Interactive Surfaces

In this section, we shall now look at the hardware implementations of the previously mentioned concepts, particularly those which are aimed towards flat horizontal displays. These setups are usually called tabletop interfaces.

4.2.1 TISCH

The central hardware device which this thesis was started upon is called *Tangible Interactive Surface for Collaboration between Humans (TISCH)*.



(a) TISCH hardware diagram

(b) TISCH with two users

Figure 4.7: Overview of TISCH

The hardware configuration of TISCH is shown in figure 4.7(a). The core element is a frosted-glass table from *Ikea* which acts as projection surface and main structural element. The table is mounted to a frame constructed from *Isel* aluminium profiles. On top of the frosted-glass table, a second sheet made of 8 mm acrylic glass has been placed. On its rim, this outer surface carries 70 Osram *SFH4250* infrared LEDs which enable it to function as an FTIR-based touch detector (see also section 2.2.4). Two mirrors underneath the table provide the necessary optical path length which the projector as well as

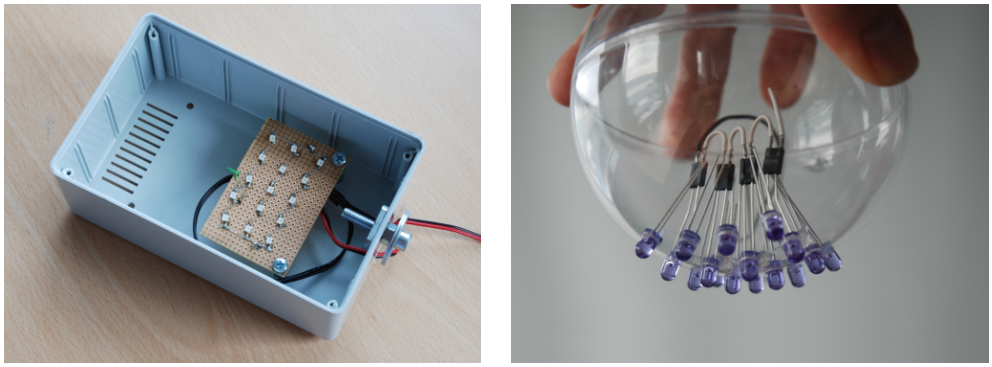
the infrared camera require to view the entire surface. The projector (InFocus *LP290*) displays a maximum of 1024x768 pixels, while the camera (Point Grey *Firefly MV*) has a sensor size of 720x576 pixels. The interactive surface covers an area of approximately 1.1 m x 0.7 m and is situated at a height of about 0.9 m. These values result in a sensor resolution of about 15 DPI and in a display resolution of about 25 DPI. It can be comfortably operated by one or more persons standing beside the table. An example is shown in figure 4.7(b). As the interior offers additional space next to the mirrors, the computer as well as the power supply and control electronics can be placed here, creating a completely self-contained system which is protected by side panels. Moreover, the entire frame has been mounted on four wheels, thereby enabling the entire setup to be easily moved to another location.

Shadow Tracker

An additional feature of TISCH besides the FTIR-based multi-touch sensor is the shadow tracker [21]. On the ceiling above the device, a second infrared light source has been mounted which uniformly illuminates the table surface. Any opaque object on the surface will now cast a clearly defined shadow on the projection screen which can also be detected by the same camera. This second light source is interleaved with the FTIR sensor as described in section 4.1.2. Both sets of LEDs are also pulsed through a control circuit to increase contrast. This is especially important in the lab environment in which TISCH is situated, as it exhibits highly variable lighting conditions.

One application of this setup is the differentiation between contacts from fingers which belong to the same hand. As these contact points appear below the same shadow in most cases, they can now be grouped together. Moreover, this method now enables the system to identify objects on the surface which do not trigger a response from the FTIR sensor, such as tangible interface elements or mobile devices.

The design of the secondary light source proved to contain some unexpected challenges. At first glance, the obvious choice for creating clearly defined shadows would have been a point light source. Two options were evaluated, both of which are shown in figure 4.8. The first one is composed of 15 SMD LEDs mounted on a regular circuit board, while the second one is composed of 16 5 mm LEDs arranged in a radial pattern on a spherical surface.



(a) with SMD LEDs (b) with 5 mm LEDs

Figure 4.8: Point light sources

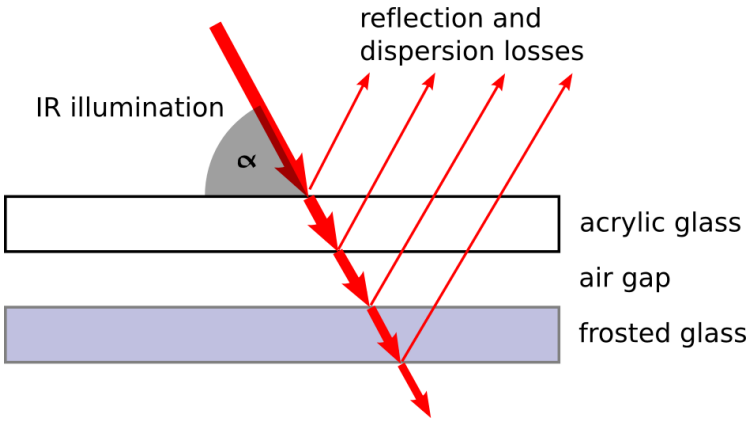


Figure 4.9: Reflection of incident light

Surprisingly, both of these light sources illuminated only a small circular region directly beneath the light source itself. While the desired clear shadows were visible in this region, the lack of illumination in the outer areas proved this setup to be unsuitable. The reason for this effect is that light from outside has to pass a total of four material-air interfaces (see figure 4.9). If each layer reflects only 15% of incoming light (a conservative assumption), the total intensity arriving at the camera already drops to approximately $(1 - 0.15)^4 \approx 52\%$ of emitted light. The reflected percentage increases with decreasing angle of incidence according to Fresnel's equations (see [43]). Below the critical angle

of approximately 41°, the light transmission even drops to zero. This is simply because at this angle, total reflection starts to occur and all light is captured in the upper plate.

Figure 4.10: Overhead light source (LEDs highlighted)

Due to this effect, it was necessary to redesign the overhead light source in such a way that the incident rays hit the table surface at an angle close to 90°. Therefore, the third and final iteration of this light source now consists of an array of 28 narrow-beam infrared LEDs (Osram *SFH485*) that are arranged in a regular grid of 4x7 elements at a distance of approximately 25 cm each (shown in figure 4.10. As the grid is suspended from the ceiling at a distance to the table surface of about 1.5 m, the light cones overlap slightly on the table surface, thereby creating an almost parallel field of light. While the illumination is now sufficiently uniform, the system still has two drawbacks. First, the ceiling-mounted component requires the interactive table itself to be placed at a fixed spot, rendering it immobile for as long as the shadow tracker is supposed to be used. Second, persons bending over the table are likely to also generate shadows with their heads and upper bodies, which are usually not supposed to be tracked.

"Diffuse Illumination" Modification

These two aforementioned drawbacks suggest a modification to the setup. By moving the secondary light source *below* the table surface, both of these disadvantages can be remedied. This method is often called "diffuse illumination" (see also section 2.2.4). As the light grid setup from the previous approach would intersect the projector's light path, a slightly different design is needed. Two spotlights consisting of 16 LEDs each (Osram *SFH485*) have been placed within the table on opposing sides, thereby uniformly illuminating the projection surface from below. Similar to the first approach, these LEDs are also interleaved with the touch-detecting LEDs in the acrylic surface plate to avoid interference.

An additional feature of this approach is that fiducial markers which have been placed face-down on the surface are now visible to the camera and can be tracked. One drawback of this approach, however, is that certain dark materials absorb enough infrared light to remain invisible to the camera. Nevertheless, the advantages with respect to the shadow-tracking approach outweigh this shortcoming. An image of one light source is shown in figure 4.11.

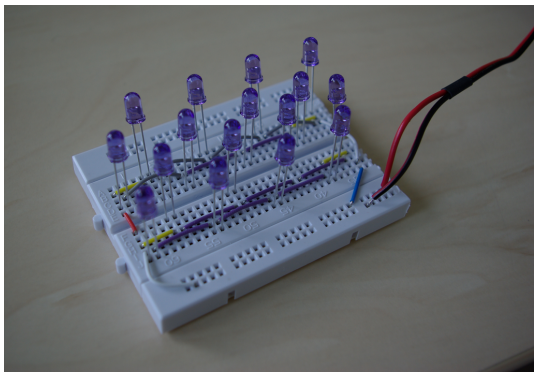


Figure 4.11: Light source for diffuse illumination

While this light source is quite similar to the point light source discussed above, the reflection properties of the surface differ significantly depending on whether the light comes from above or from below. Although this type of light source was unsuitable for shadow tracking, it can easily be used in this application.

In figure 4.12, the thresholded (top) and denoised (bottom) images from two objects on the surface are shown. The user's hand is clearly visible, also

the fiducial marker. Note that this marker with a physical size of 6 x 6 cm is already below the threshold where the code can be reliably detected. This is due to diffusion caused by the projection surface. The marker in its entirety can still be tracked.

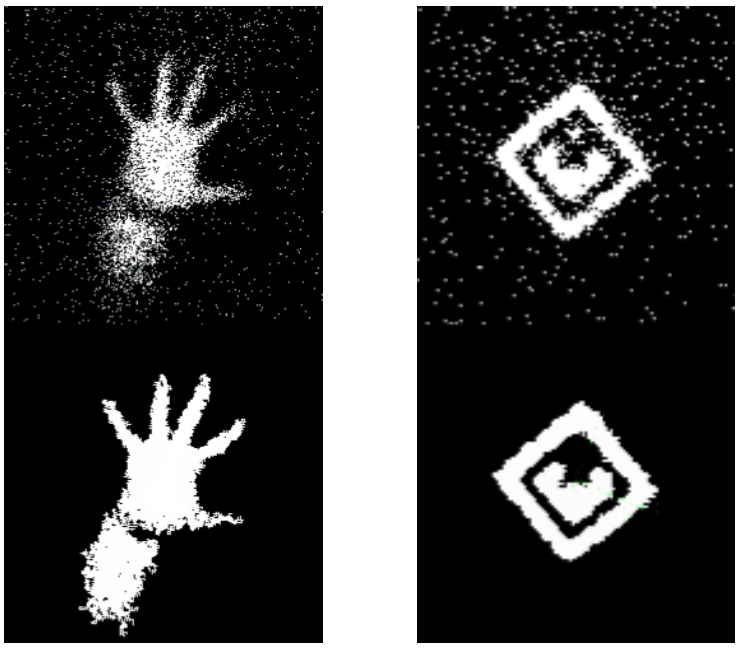


Figure 4.12: Objects captured by diffuse illumination before and after noise filtering

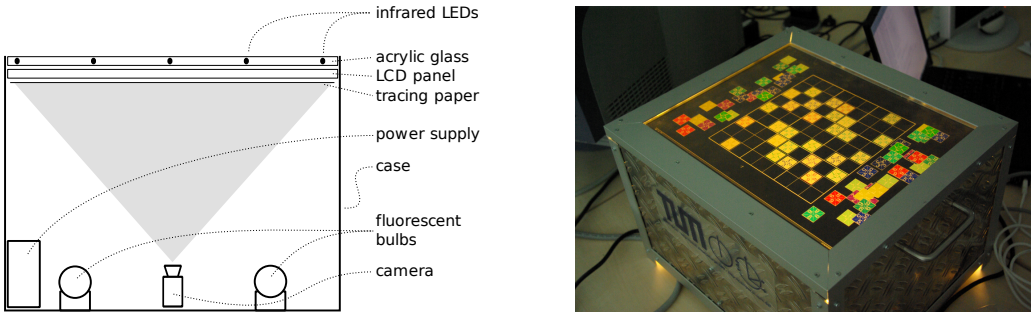
4.2.2 MiniTISCH

While the previous, projection-based approach has various advantages such as image size and brightness, it may not be the ideal solution for some applications. When requirements such as high resolution or a small, compact device are given, LCD screens often are superior to projectors. In the current context of interactive surfaces, this choice of display type raises the question of detecting touches on such a screen.

As it turns out, LCDs are mostly transparent to infrared light. The "light valves" which compose red, green and blue pixels on this kind of display are equipped with coloured filters that allow only one primary colour to pass

through. However, blocking non-visible wavelengths such as infrared or ultraviolet light would be unnecessary or even detrimental from an engineering perspective, as the increase in absorbed energy might decrease the panel's lifetime. Therefore, regardless of its display state, an LCD allows a significant portion of infrared light to pass through, much like the projection surface from the previous section.

The fact that LCD panels do not block IR light can now easily be exploited to convert such a panel to a touchscreen in a way which is very similar to the projector-based approach. An FTIR surface consisting of a 5 mm acrylic plate with 40 Osram *SFH4250* LEDs is placed in front of the panel whose infrared emissions are again captured by a rear-mounted camera. This comes at the cost of added depth of the entire setup, as the camera requires a minimum distance from the front to be able to capture the entire surface. One additional challenge in this context is the backlight. In most cases, the backlight is a thick plate of acrylic glass behind the screen which has a white, opaque diffuser on the rear and two or more cold-cathode fluorescent lights at the sides. Especially due to the diffuser, the backlight is not transparent to infrared light. As the backlight is a crucial component without which an LCD cannot function properly, a replacement has to be found.



(a) Schematic view

(b) Displaying a Sudoku game

Figure 4.13: MiniTISCH

One possible solution would be replacing the diffuser with a material which is opaque to visible light, but transparent to infrared light. While such materials exist, they are very expensive and usually not available in a form which is easily applicable in this context, such as a thin film. A different, easier solution is therefore to simply move the backlight backwards so that it is at the

same distance from the screen as the camera and does not block the infrared light from the FTIR surface. In the current setup, the original backlight was replaced by four energy-saving light bulbs which are arranged symmetrically around the camera. The diffuser was replaced by a sheet of tracing paper which evenly distributes light from the bulbs while still allowing the FTIR response to be visible. The acrylic distribution plate was removed altogether. In figure 4.13, the entire setup is shown.

As the energy-saving bulbs still emit a significant amount of infrared light, a synchronisation/pulsing circuit again needs to be employed to achieve sufficient contrast. Even though the camera itself is equipped with an infrared band-pass filter, the backlight would otherwise still outshine the response from the FTIR screen. Another aspect which needs to be carefully considered is the choice of LCD screen for this approach. Many such displays contain a controller circuit which is directly attached to the panel itself through a flexible flat cable that runs along the entire width of the display. In a normal assembly, this circuit is folded around the backlight and rests on the rear side of the display. To give the camera an unobstructed view of the panel's rear, however, this circuit has to be moved out of the way. As this is not easily possible with all LCD models, some experimentation may be needed to find a suitable device. In this setup, a BenQ *FP757* 17 inch monitor was disassembled to retrieve the panel and controller circuit.[1] This screen offers a maximum of 1280x1024 pixels, therefore providing a display resolution of about 95 DPI. A Point Grey *Firefly MV* with a sensor size of 720x576 pixels was again used as camera, resulting in a sensor resolution of approximately 50 DPI.

With this setup, it seems also possible at first glance to add a second internal light source for diffuse illumination. Due to the portable nature of the device, a shadow-tracking approach with an external infrared emitter is impractical. However, an evaluation with two different secondary light sources has shown that the reflective properties of the combined LCD screen and tracing paper make detection of reflected light from objects on the outer side of the display surface almost impossible. In both variants (15 wide-angle LEDs or 16 narrow-angle LEDs), the vast majority of the light was immediately scattered back towards the camera, outshining any additional reflection from external objects.

[1] An internet community which focuses on constructing video projectors from LCD screens also offers information on which models are easy to disassemble and "unfold" [80].

One method to counter these problems would be to switch to a camera which offers HDR functionality. This would allow to alternate between odd frames with active FTIR light source for touch detection and even frames with long exposure times for shadow tracking with ambient light. Unfortunately, the currently mounted camera does not offer this option.

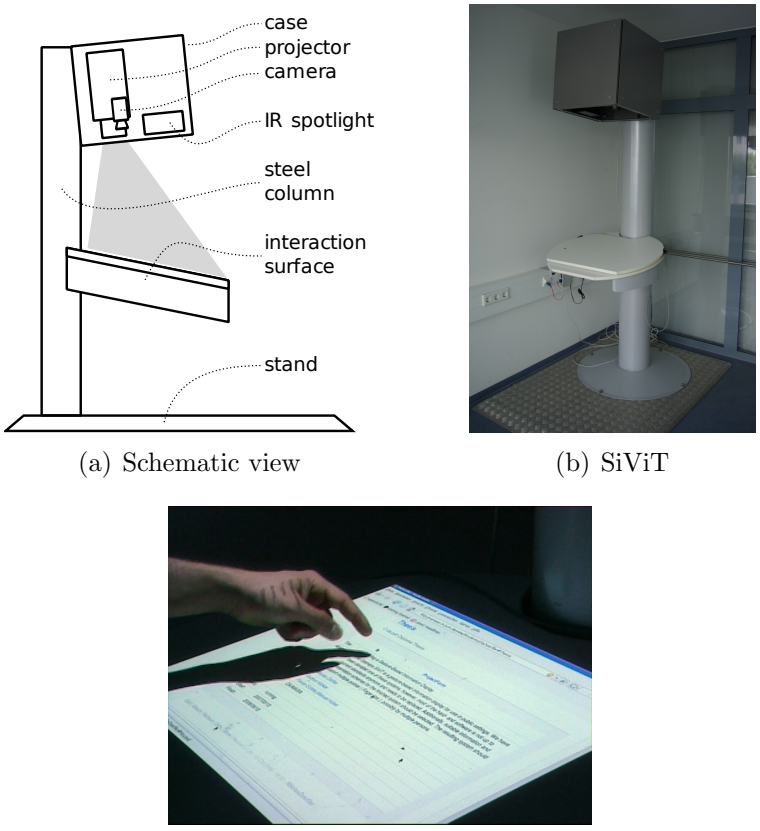
4.2.3 SiViT

The Siemens Virtual Touchscreen (SiViT) is another kind of input device which seems quite different from the previously described ones at first glance. This commercial system was designed to be deployed in public settings like train stations where it may be subjected to dust, vibrations, vandalism and other harsh environmental conditions. Therefore, its main structural element is a massive steel column at the top of which a metal box containing all electronic components is mounted. Below, a table made from particle board acts as projection and interaction surface. As all sensitive devices are out of reach of the user, the setup is quite robust.

The device compartment contains a projector, computer, infrared camera and two high-powered infrared spotlights. The camera as well as the projector are focused on the table. As its white surface reflects a large amount of infrared light, most objects above the table such as a user's hand appear darker than the background. In the original setup, the user could control the mouse pointer of a standard Windows installation by using a single outstretched finger. A click could be triggered by keeping the hand still for a short while. However, this system was primarily designed to use a single pointer with legacy applications.

We have updated the system, which was originally built around 1999 [106], with a recent computer and software [18]. In particular, we added a small hardware modification to allow more natural interaction while not compromising the system's ruggedness. Two piezo-electric microphones were added underneath the projection surface to detect sounds, thereby allowing the user to activate objects by tapping or knocking them as on a touchscreen. To keep the additional hardware's complexity small, only two microphones were used which can be directly connected to a standard computer's audio input. However, this introduces the drawback that the location of a sound on the surface can not be located exactly, but only up to a hyperbolic curve. Therefore, it is necessary to correlate this information with the optical tracking to achieve an accurate estimation of an interaction.

One drawback of the current setup is that even when taking the optical tracking data into account, the accuracy still is only approximately 15 - 20 cm.



(a) Schematic view

(b) SiViT

(c) Using a web browser

Figure 4.14: Siemens Virtual Touchscreen

One reason for this low precision may be the composition of the particle board surface from compressed wood shavings. As the internal structure is highly random, reflections and multi-path propagation of sound waves within the material are unpredictable. While the raw sound data is processed by the *generalized cross correlation with phase transformation (GCCPHAT)* algorithm which is designed to counter such effects, a more homogeneous material might still offer significant improvement.

4.2.4 FlatTouch

As described earlier, projector-based multi-touch interfaces usually require significant space behind the display. While sensor solutions for flatscreens exist,

they are usually either quite expensive or require significant re-engineering of the display panel. But there are other options worth exploring. In this approach, an FTIR sensing surface is put *in front* of a regular, unmodified flatscreen display. An infrared camera which views the screen from the same side as the user is attached to the display in an off-centre position to keep interference with the users' actions to a minimum. By synchronising the FTIR light source with the camera, a sufficiently high contrast can be achieved to capture infrared light shining *through* the user's finger upon surface contact. Such a system in use can be seen in figure 4.16(b).

Theory of Operation

When changing to a more abstract point of view for a moment, it becomes apparent that a finger touching an FTIR surface is simply lit from below. However, a human finger is far from being completely opaque, as a short experiment with a flashlight can easily show. In particular, it allows partial transmission of the red and infrared parts of the light spectrum. This property is sometimes being used in heart rate sensors which are clipped to the finger tip and measure the transmission intensity of infrared light. Due to this effect, some of the infrared light hitting the finger through the FTIR surface will also radiate *outwards*. Moreover, a fraction of the light reflected downwards will be reflected a second time on the display surface, thereby creating a halo of infrared light around the contact point. These effects are illustrated in figure 4.15.

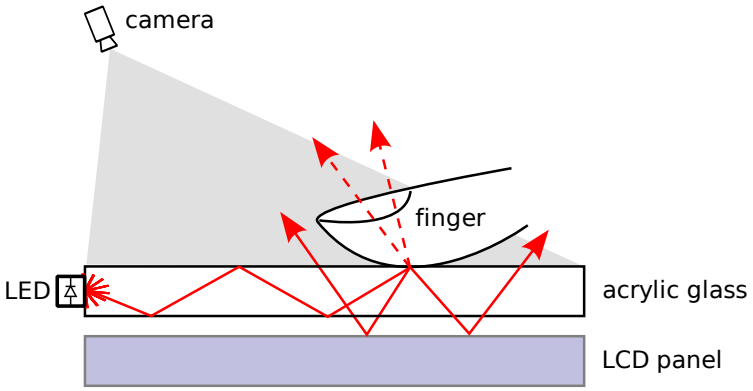


Figure 4.15: Inverted FTIR

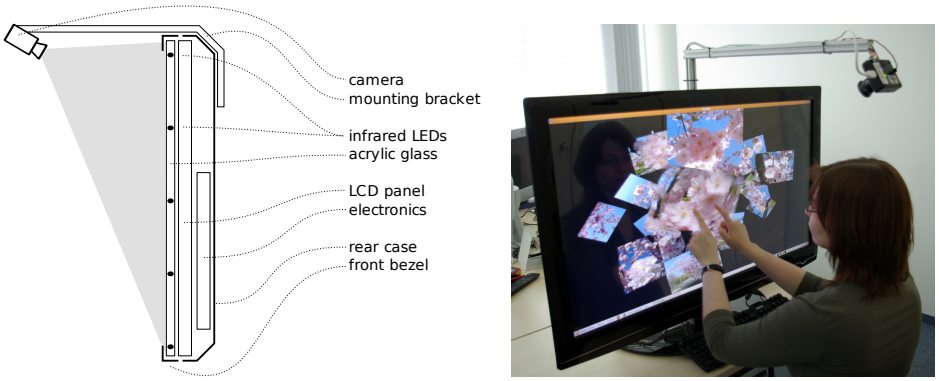
These various paths emit light towards the user which can now be sensed by an infrared camera placed on the front side of the display. As the intensity of the light reflected towards the outside is lower than that reflected towards the display, special care must be taken to achieve a high contrast with respect to stray light such as the infrared emissions from the LCD backlight.

Hardware Setup

For evaluation of this approach, a 42 inch LCD television screen from LG was chosen. This display supports full *high definition (HD)* resolution of 1920 x 1080 pixels and offers a maximum brightness of 500 cd/m^2, which is appropriate for most indoor conditions. After temporarily removing the front bezel, a 5 mm acrylic plate was inserted in front of the panel. The plate is held in place by the reattached front cover and also provides additional protection for the LCD panel. On the rim of this acrylic plate, 150 high-powered infrared LEDs (Osram *SFH4650*) with an emission wavelength of $850nm$ have been attached with instant glue. These LEDs are organised in 30 groups of 5 diodes each. Within each group, the LEDs are connected in series, whereas the groups themselves are wired in parallel to a 12V power supply through a switching circuit. This circuit pulses the LEDs in sync with the camera's shutter as described in section 4.1.1. The camera used in this setup is a Point Grey *Dragonfly 2* with a resolution of 1024 x 768 pixels running at a frame rate of 30 Hz. The camera is equipped with an infrared low-pass filter to block all interference from visible light, particularly from the screen content. A diagram of the entire setup is shown in figure 4.16(a).

The correct placement of the camera is of particular importance. By using a fish-eye lens with a focal length of 2.5 mm, the camera can be moved close to the screen while still being able to view the entire surface. However, care must be taken not to position the camera in such a way that the user is hindered. For a vertical display, attaching the camera to a short beam above the user's head is the most suitable solution. In case of a horizontal display, the best position for the camera would likely be centred above the surface to avoid blocking one side of the display. However, this may require attaching the camera to the ceiling, as the mounting beam might otherwise still interfere with the user's movement.

A sample raw image from the camera is shown in figure 4.17. The halos around the three fingers touching the surface are clearly visible, also the increased brightness of the fingertips themselves. Although this image is similar



(a) Schematic view (b) FlatTouch in use

Figure 4.16: FlatTouch

to that taken from the backside of an FTIR surface, there are some important differences. In FTIR setups, the bright blobs resulting from surface contacts usually reach maximum intensity near their centre while gradually getting darker towards the border of the contact area. With this setup, the brightness distribution is less clearly defined. In most cases, a contact results in two bright stripes to the left and right of the finger with a slightly darker area (the finger itself) in-between.

Figure 4.17: Touching the surface with three fingers

When applying the usual image processing steps such as background sub-

traction, thresholding, denoising and connected-component analysis to such an image, this will usually result in the two bright stripes besides the finger being detected as blobs which move and rotate in parallel. This data can directly be delivered as input events and will produce the expected results in most cases despite generating two blobs per contact point. As turning the finger rotates the blobs with respect to each other, even single-fingered rotation is possible up to a certain point. Depending on the camera location and angle, however, this may at some point lead to occlusion of one blob by the finger itself.

An important observation in this context was that the backlight of the LCD panel may interfere with the touch detection under some circumstances, as the fluorescent tubes which are usually employed emit significant amounts of infrared light. While the backlight does reduce the contrast between background image and touching fingers, this does not pose a problem as the LED pulsing method still produces sufficiently high contrast.

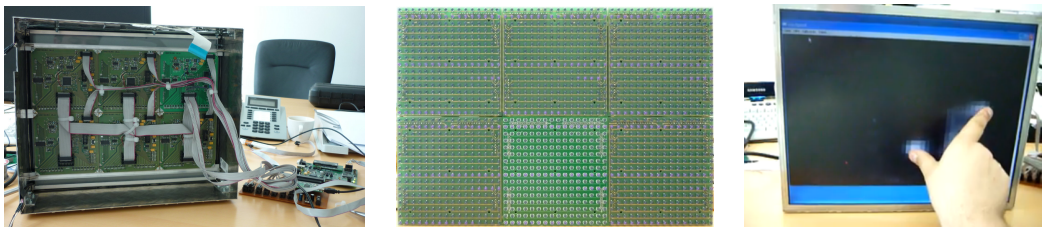
However, some LCDs offer the ability to regulate the brightness of the backlight. This is achieved through pulse-width modulation of the backlight voltage. In the case of the LG television which we are using, this modulation is done at a frequency of 200 Hz. Unfortunately, as the camera is running at a frame rate of 30 Hz, this causes intense pulsing and flickering of the LCD background in the camera image. An easy solution to this problem is to turn the LCD brightness up to maximum intensity, thereby effectively turning the modulation off. This results in a constant, even background brightness. Note that LCD panels are mostly transparent to infrared light regardless of the displayed image. Therefore, changing content of the on-screen image does not influence the background brightness.

One aspect which has not yet been studied are long-term effects of the modification on the hardware. The acrylic front plate acts as a heat insulator, thereby increasing the temperature of the panel assembly during operation. This may reduce the lifetime of the panel itself or the backlight. While this sensing method is highly suitable for a table-based setup where users are able to place their feet beneath the table, some reports suggest that running LCD screens horizontally for longer periods of time may also lead to premature failure of the device, maybe also due to decreased cooling efficiency. Adding active cooling might be a possible approach to mitigate these effects.

4.2.5 LCD with IR-LED Sensor

A different approach to multi-touch sensing on flatscreens is to remove the camera altogether and instead scale the sensor across the entire area of the display while simultaneously moving it closer. One such setup is ThinSight [61], which uses an array of commercial distance sensors to achieve this result. However, these sensors are expensive and required in large numbers, resulting in prohibitive cost for the entire setup.

A slightly different method exploits the previously mentioned capabilities of LEDs to replace the expensive distance sensors [95]. An appropriately wired array of infrared LEDs can be placed behind the backlight and detect touches in a similar manner to the various diffuse illumination approaches, yet without the need for a full optical system with its distance requirements. In fact, the resulting array is functionally identical to a large-scale *charge-coupled device (CCD)* sensor with low resolution, in this case 48x32 pixels.



(a) Rear view of sensor array (b) Front view of sensor array (c) Live sensor data

Figure 4.18: LCD with IR-LED touch sensor (images from [95])

As the "camera pixels" can be accessed directly in such a circuit, a different approach can be used to differentiate between light from the LEDs and environment light. By modulating the emitting LEDs with a known frequency, e.g. 10 kHz, and adding an appropriate band-pass filter to the sensing circuit, a clear separation of emitted light and background noise can be achieved.

One additional aspect which must be taken care of is that a matrix of LEDs is usually accessed through row and column conductors. However, as some LEDs must always be active while others are used as sensors, a simple matrix circuit is insufficient. Otherwise, the active LEDs would influence the readings taken from the sensing LEDs. Therefore, each column of LEDs is accessed through two conductors that are wired to even or odd LEDs only, thereby effectively creating two matrices which are interleaved column-wise

(see also variant 2 in figure 4.19).

4.2.6 Visible-light Display & Sensing

In the previous setup, infrared LEDs were used. However, nothing prevents the same method from working also with visible-light LEDs, thereby offering the potential to use such an LED field as display and touch sensor simultaneously [17]. One way of implementing this method was already presented in [56], however, this approach was only designed to allow alternating use of the two functions.

When aiming towards simultaneous display and touch sensing, some problems arise. The most fundamental limitation is that sensing is only possible for those LEDs which are adjacent to one or more currently emitting LEDs, as otherwise no reflected light from a touching object would be present. While it might be possible to rely entirely on occlusion of environment light for sensing, this approach is highly unpredictable, as the intensity of suitable wavelengths will vary randomly. It is important to note that not all types of LEDs are equally well suited for such setups. In particular, infrared and visible "hyper-red" LEDs seem to be most sensitive to their own wavelengths, making them suitable for sensing applications.

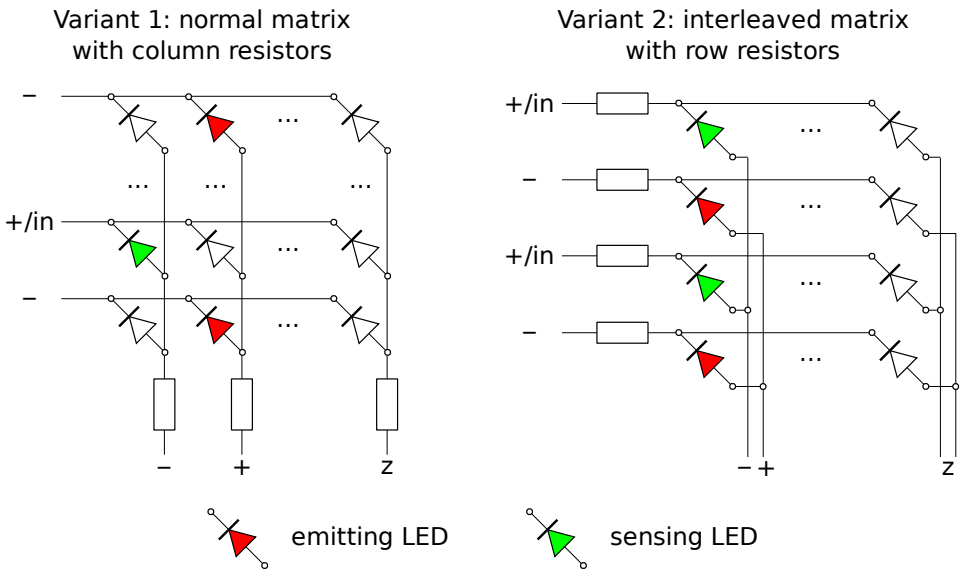
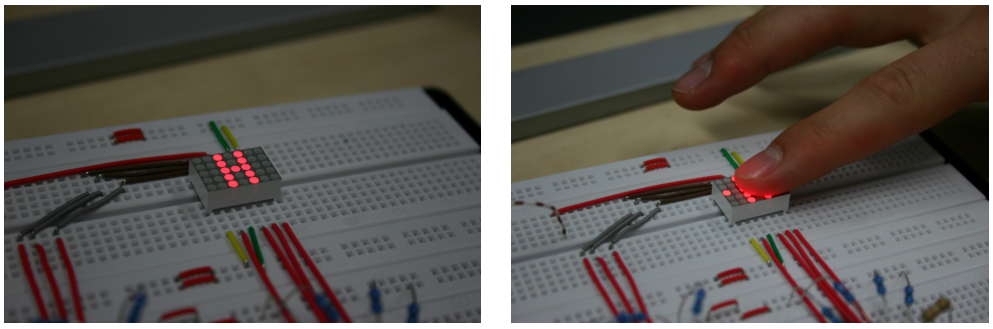


Figure 4.19: Schematics of LED-based display and sensor matrices

A second problem which needs to be considered is that the LEDs which are driven as emitters at any given moment must not influence those which are used as sensors at the same time. One way of avoiding this problem is to insert current-limiting resistors into the column conductor.[2] However, this approach has the drawback that with increasing number of emitting LEDs in one column, the intensity drops rapidly. Moving the current-limiting resistors to the row lines fixes the problem of varying intensity. Unfortunately, in an ordinary matrix with a single set of row and column conductors, this causes leakage currents to occur which render the measurements invalid. An approach to solving both problems is therefore to keep the current-limiting resistors in the row lines, but split the matrix internally by providing two alternating sets of column lines as described in the previous section. This allows to have all even LEDs in one column emit light with constant intensity while at the same time sensing reflected light using the odd LEDs. The drawback of this approach is that commercial LED matrices which only have a single set of column lines are unsuitable. Moreover, it has proven difficult to find suitable discrete LEDs which are sensitive enough to their own emitted wavelength. The two variants are illustrated in figure 4.19. A setup which uses variant 1 is shown in operation in figure 4.20.



(a) LED matrix displaying a letter (b) Sensing presence of a finger

Figure 4.20: Simultaneous display and sensing with LED matrices (images from [17])

[2]For consistency, we will assume LED matrices where the column lines are connected to the individual anodes.

4.3 Commercial Systems

While all previously described setups have been constructed during the course of this thesis, some commercial systems have also been employed or repurposed as multi-point input devices to test the suitability of the presented architecture beyond custom hardware. They shall briefly be mentioned here for completeness' sake.

(a) ART hand tracker (b) Wiimote (c) iPhone

Figure 4.21: Commercial interaction devices

4.3.1 Free-Air Handtracking

One widely-known example for multi-finger interaction in popular media is the movie "Minority Report", in which the protagonist uses special gloves to control a vertical large-scale display with gestures in mid-air. Several commercial hardware devices are available which can be used to realise this kind of interaction.

ART Handtracking

A commercial free-air handtracking system is available from Advanced Realtime Tracking [1]. It consists of an active IR marker which is attached to the back of the hand and three finger markers which are connected to the hand marker through short wires. In conjunction with the ART tracking cameras, a 6-DOF pose of the hand and the three fingertips can be extracted. Various interaction modalities for this system can be envisioned, such as triggering an interaction by pinching two fingers together or by touching a virtual plane in mid-air.

Wiimote

One drawback of the previously mentioned systems is its high cost, particularly due to the tracking cameras. One other solution which is easily available is to use the widely popular Nintendo Wiimote, either as a replacement for a tracking camera or as an interaction device itself. In the first mode, the user also has to wear gloves, either with reflectors or with active IR emitters at the fingertips. In the second mode, at least two IR emitters (the "sensor bar") have to be present in a fixed location relative to the screen.

4.3.2 iPhone

Finally, the commercially very successful iPhone from Apple has also been used in this thesis. Its capacitive sensor (see also section 2.2.2) allows to sense several simultaneous finger contact points, although not their orientation. A "hover state", while probably supported by the hardware, is not made available through the API.

4.4 Sensor Capabilities

The various purpose-built and commercial devices which have been presented in the previous sections offer a wide range of capabilities which have to be integrated into the framework. The following matrix describes the degree of support (+ full support, o rudimentary support, - no support) for the various available features. Although it is a software-based solution, the capabilities of a multi-mouse setup shall be compared here, too.

Multi-User Several persons can use the system simultaneously.

Multi-Touch Several interaction points simultaneously are possible.

Direct Direct touch means that the user interacts directly with the display as opposed to a proxy object.

Hover Interaction without touching is possible.

Tangible Objects Arbitrary objects on the surface can be tracked and used for interaction.

Fiducial Markers Objects can carry tags, markers or similar which uniquely identify them. [3]

	Multi-User	Multi-Touch	Direct	Hover	Tangible Objects	Fiducial Markers
TISCH	+	+	+	o	+	o
MiniTISCH	+	+	+	o	+	o
SiViT	+	o	+	+	+	- (+)[3]
FlatTouch	+	+	+	+	+	- (+)[3]
LEDTouch	+	+	+	o	o	-
(Multi-)Mouse	+	-	-	+	-	-
iPhone	-	+	+	-	-	-
ART Handtrack	+	o	-	+	-	+
Wiimote	+	o	-	+	-	-

Table 4.1: Device Capabilities

Summary

In this chapter, various fundamental techniques for developing and improving multi-touch sensors were presented along with their benefits and drawbacks. Moreover, a number of devices which have been constructed or enhanced during the course of this thesis were discussed, along with some commercial systems which were also used.

[3] Support for this feature may depend on which side the fiducials are on.

Chapter 5

The libTISCH Middleware

Now that the architecture as well as the various available input devices have been discussed, we shall examine the core software implementation of the presented framework. The library, called libTISCH, is licensed under the *Lesser General Public License (LGPL)* [40] and can be downloaded from the open-source repository site *Sourceforge* [20]. The library was primarily developed on Linux, but supports the three major platforms Linux, MacOS X and Windows. One design goal was to keep the number of external dependencies as low as possible; the only core dependencies are therefore the C++ *Standard Template Library (STL)* and an *OpenGL Utility Toolkit (GLUT)*-compatible library such as FreeGLUT [4]. The implementation closely follows the design of the architecture and is therefore split into four separate layers.

5.1 Design Considerations

The previously described architecture does not specify any implementation details. Therefore, a wide range of implementations are possible. For the libTISCH reference design, several high-level design decisions thus had to be made which shall be discussed here.

5.1.1 Interoperability and Network Transparency

With respect to the layers, no specific type of transport technology between them has been specified. The libTISCH implementation uses the *User Datagram Protocol (UDP)* across all three interfaces, ensuring the highest degree of interoperability with alternative implementations as well as network trans-

parency. While using a network protocol invariably introduces an additional source of latency when compared to other inter-process communication methods such as local sockets or shared memory, our evaluation has shown that this additional latency remains within acceptable limits (see section 5.4.3).

5.1.2 Speed-Accuracy Tradeoff

In most cases where a tradeoff between speed and accuracy had to be made, the decision was made for speed. The rationale behind this decision is that an interface designed for direct or indirect manual interaction should be designed as resilient as possible, i.e. not require high-precision interaction from the user such as hitting tiny buttons which are smaller than a finger.

5.2 Hardware Abstraction Layer

As mentioned previously, the lowest layer has the task of converting raw motion data into a common format describing the movement of various object types over the interactive surface. This can either be done by converting data from existing software packages or by directly reading sensor data from the hardware.

5.2.1 Adapters for Existing HAL Software

There is a wide range of existing software which fulfils the role of hardware abstraction in some way. Examples include the X11 windowing system resp. its Xorg reference implementation, the Ubitrack tracking framework or the reacTIVision library. Here, the adapters which translate events from existing software into libTISCH's own LTP format are presented.

Tuio Converters

The most basic type of HAL implementation is the Tuio [68] converter. The OSC-based Tuio format has gained some popularity in interactive surface applications. Several widely used software packages such as touchlib [89] or reacTIVision [67] communicate through this protocol. A converter is therefore desirable to improve interoperability between libTISCH and other software in this field. While Tuio lacks some features which have prompted the adaption of a different protocol within libTISCH (LTP, see also section 3.2.2), it is nevertheless very similar. Therefore, the two protocols can easily be converted

into each other. The same authors who devised the Tuio protocol also provide a small OSC parsing library, oscpack [6]. Both converters are based on this library.

Ubitrack Adapter

The Ubitrack library [55] has been developed with a strong focus on industrial-grade tracking with six degrees of freedom. Although the connection to interactive systems may not be apparent at first glance, many of the features of Ubitrack, such as reliable tracking of fiducial markers, can be highly valuable in this context. Therefore, an Ubitrack adapter has been included in libTISCH. The only requirement with respect to the tracking data is that it should be delivered in a coordinate system whose x-y plane is roughly parallel to the screen on which the interactive application is displayed. This coordinate system does not have to be registered exactly relative to the screen; a rough alignment is sufficient. The reason for this rough pre-alignment is that in one mode of operation, the Ubitrack adapter uses a virtual plane parallel to the x-y plane for detection of emulated touches. The exact registration between tracking coordinates and screen coordinates is later performed by the transformation layer.

The Ubitrack adapter has three basic modes of operation. In the first and most simple mode, the adapter accepts 3-dimensional position data or 6-dimensional pose data for a list of objects. This data is converted into uncalibrated 2-dimensional screen coordinates by projecting the data into the x-y plane and is then sent as input events of type *object*. When the input data consists of poses with orientation, the orientation vector is also projected into the x-y plane and used as orientation for the generated input event. This mode is suitable for tracking, e.g., tangible objects with fiducial markers on a surface.

The second mode is slightly more complex. Here, the adapter switches to a touch emulation mode. A tracked object's data is converted into events of type *blob* (generic tracked entity) as long as a certain configurable distance to the screen plane is kept. Should the object's distance fall below this threshold, an additional *finger* object with the same position and orientation appears, thereby emulating a touch. This mode can be used in applications where, e.g., each hand of the user can be tracked individually. While the hand is held away from the screen, a "hover" mode of interaction can be used which changes to an emulated "touch" state when a virtual plane above the screen is crossed.

Finally, in the third mode of operation, input data has to be sent as pairs of tracked objects. While these objects are separated by a distance above a configurable threshold, the position of their centroid is sent as a *blob* object with the orientation determined by the vector between the two objects. When the distance falls below the threshold, an additional *finger* object with the same size and orientation is added. This mode is suitable for applications where at least two fingers resp. fingertips of the user can be tracked individually. "Hover" interaction is possible while holding the fingers apart, whereas "touch" interaction is emulated by pinching the fingers together in mid-air.

Generic Mouse Adapter

While developing a multitouch-enabled application, it may often be convenient to quickly test a certain aspect of the interface without access to a full multi-touch interface. For this reason, a mouse adapter has been included with libTISCH. While it can provide only a single spot of interaction, this is nevertheless often sufficient for quick testing. This adapter is available on all three platforms, as the mouse data can be retrieved through the cross-platform GLUT library. However, this results in the limitation that the mouse adapter is only available when the included OpenGL-based graphics layer is used.

This adapter translates the mouse pointer into an elliptic *blob* object, thereby impersonating a generic object on the surface. By using the mouse wheel, the equivalent ellipse of the blob can be rotated, giving access to additional interaction modalities. When the primary mouse button is pressed, an additional *finger* object of same size and shape appears, simulating a surface contact. A registration step is not needed in this case, as GLUT always delivers event coordinates relative to the window within which the application is running.

MPX Adapter

While the previously described mouse adapter is mainly a tool for quick testing, its MPX extension offers additional options. As mentioned earlier, the multi-pointer X server MPX is an extension of the widely used X11 windowing system which allows an arbitrary number of mice and mouse pointers to be used simultaneously. By providing a patch to the FreeGLUT library, it is now possible to differentiate between the various mouse events. As the blob emulation works in the same manner as described above, several persons can now use several mice simultaneously to interact with a libTISCH-based appli-

cation. Again, no registration is necessary; however, it is mandatory to use libTISCH's own widget layer to access this feature, as the top-level window itself has to be registered with the X server for reception of multi-mouse events. The modifications to FreeGLUT are described in detail in appendix A.4.

iPhone Adapter

Although it may not seem obvious at first sight, there is almost no difference between the MPX-based HAL implementation and the one for the iPhone. Data delivered from the iPhone's touchscreen is almost isomorphic to that provided by MPX, with the exception that the iPhone is physically unable to provide hover data or blob orientation. Although GLUT is not available for the iPhone, a compatibility wrapper has been provided inside libTISCH which allows the various interfaces of the iPhone API to appear as a GLUT implementation. Details on this wrapper are provided in appendix A.5.

5.2.2 Native Hardware Drivers

While the hardware devices presented in the next section can also be used, e.g., as mouse drivers or connected to one of the existing libraries, their popularity suggests that it would be beneficial for users of libTISCH to be able to use them "out of the box" without the need for additional large software packages.

Wiimote Connector: `wiimoted`

The Wiimote is a very popular interaction device, partly due to its low price of approximately €40. In its original mode of operation, the user holds the device in one hand and points it towards the so-called "sensor bar", which in fact consists just of two infrared LED clusters at a distance of 30 cm. The sensor itself is situated inside the Wiimote itself and consists of a high-speed infrared camera with integrated image processing functionality. The camera directly delivers the image coordinates and sizes of the four brightest blobs to the device, which then sends this data to a computer through a Bluetooth link.

As the Wiimote itself contains all necessary sensor hardware, it can be used as a standalone infrared camera. This offers another mode of interaction where the device itself is stationary and is used to track infrared-reflective or active markers on the user's fingertips. This method has been presented by

Vlaming et al. [117]. In both cases, the driver connects to the Wiimote using the `wiiuse` library [74] which is available for Linux and Windows.

The libTISCH Wiimote driver [9] has the primary task of translating image blobs delivered by the Wiimote into LTP packets. In the first mode of operation, the Wiimote is held by the user. The driver takes the two brightest spots in the image and uses their centre point as the primary location at which a *blob* object is sent. The orientation is determined by the vector between those two points. When the user pushes one of the buttons, a *finger* object is added, simulating a touch. This mode is similar to one of the Ubitrack adapter modes. The "sensor bar" should be situated below the display, roughly parallel to its lower edge.

In the second mode of operation, the Wiimote itself is stationary on top or next to the display and facing the user, who wears gloves which are equipped either with reflectors or IR emitters on each thumb and forefinger. When holding the fingers apart, two spots are tracked for each hand, treated as in the previous mode and sent as a *blob* object at the average location. When the user pinches the fingers together, the spots merge into a single one, prompting a touch emulation event and adding a *finger* object at the same location.

DiamondTouch Adapter

Another device which has enjoyed great popularity in the field of multi-touch interfaces is the DiamondTouch [15], also because it was the first commercially available device. While the device does not deliver true multi-touch data, but rather a list of row and column intensities, it is on the other hand able to reliably differentiate between several users. The adapter will try to match the row and column intensities, thereby creating an estimate of where the actual contact spots are. These can be directly sent as finger blobs, with the added feature that the parent ID for these blobs can be set according to the user to which they belong. Note that due to the lack of a real DiamondTouch interface, this adapter has not yet been evaluated on real hardware and must therefore be considered non-functional until further testing has been conducted.

5.2.3 Camera-Based Tracking: `touchd`

In the context of this thesis, the most important kind of HAL implementation is the optical tracking layer. It offers tracking and analysis of image blobs for such diverse input modalities as FTIR, shadow tracking or diffuse illumination.

An overview of the data flow is given in figure 5.1. The implementation consists of three major sub-components, which shall now be discussed in detail.

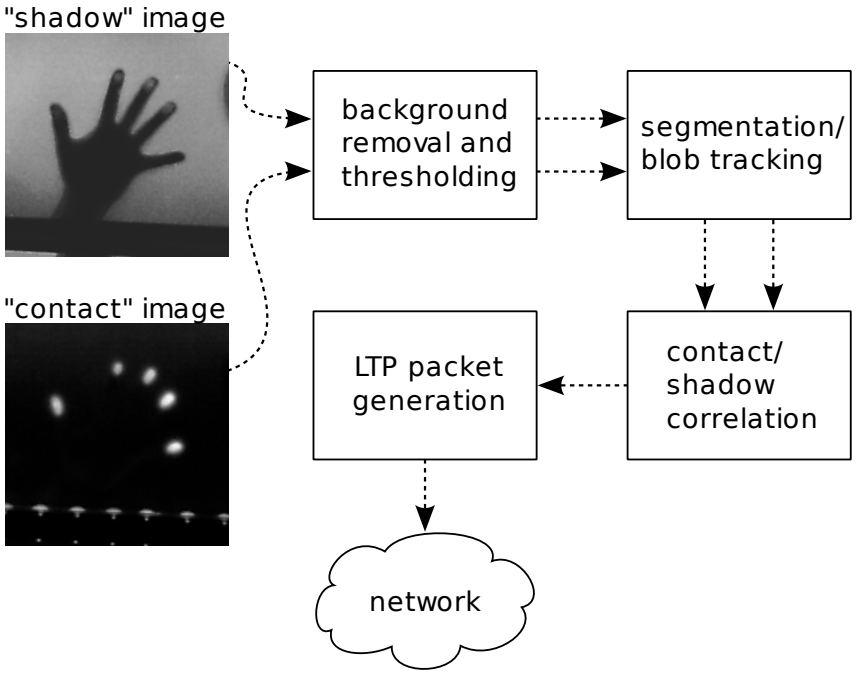


Figure 5.1: `touchd` modules

Image Acquisition

`touchd` uses an abstract class `ImageSource` to retrieve raw image data from the camera. This class mainly offers a method to retrieve the image data itself, and a set of methods to adjust camera hardware parameters such as frame rate, sensor gain, shutter speed and brightness control. There are currently three implementations of the `ImageSource` class:

`V4LImageSource` Available on Linux. This image source uses the Video4Linux API, version 2, and offers full support for all camera parameters if the underlying hardware driver also supports them.

`DCImageSource` Available on Linux and MacOS X. This image source uses `libdc1394` [19] to access cameras which comply to the 1394 Digital Camera specification (mostly Firewire-based cameras). All parameters are supported.

`DirectShowImageSource` Available on Windows. This image source uses the DirectShow API to access any camera which has a generic DirectShow driver available. Parameter setting is currently unsupported.

Image Processing

To extract meaningful data from the raw images, they have to be subjected to several processing steps. From an abstract point of view, the goal of this processing pipeline is to separate regions of interest from the background. While the background is mostly uniform in all current applications, its brightness may rapidly change due to varying external lighting conditions, such as sunlight or fluorescent ceiling lights.

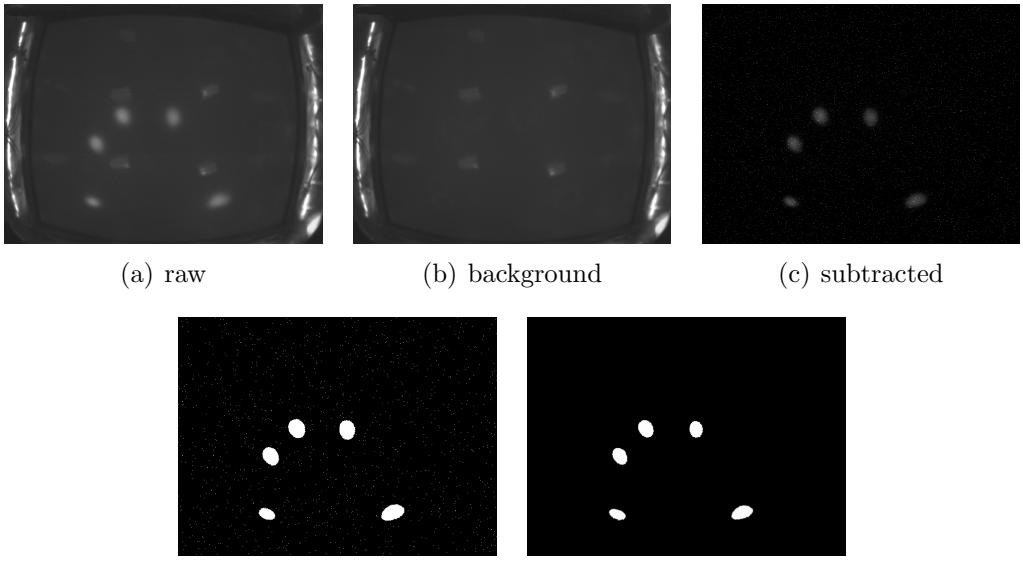
Therefore, the first step is generation of a difference image between the raw camera data and a dynamically adapted background image. This background image is initialised to black upon startup. After every frame, the pixel values of the background image are updated with a moving average between the current background value b and the current raw image value r according to $b_t = (1 - a)b_{t-1} + ar_t$, with the experimentally determined value $a = \frac{1}{512}$. a is a fraction of a power of two as this can be implemented with fast fixed-point integer arithmetic.

After background subtraction, the next step is generation of a binary image by thresholding of the difference image. Due to the background subtraction step, the threshold can be chosen as the difference between the overall intensity of the background image and the raw image. Even though this difference may be quite small, it is nevertheless sufficient, as varying lighting conditions have already been compensated by the dynamic background.

While the resulting binary image now mostly contains regions of interest, small amounts of noise are usually still present. These can effectively be removed through an erosion step which counts the number of bright neighbour pixels for each bright pixel and removes this pixel if it has less than, e.g., 8 neighbours.

Sample images from each of these five stages are shown in figure 5.2.

These steps were first implemented in pure C++. However, the performance impact was noticeable and occasionally too large. Therefore, a second, alternative implementation in MMX assembler is now available which provides a significant performance boost on systems which use the `gcc` compiler. While popular image processing libraries such as OpenCV [59] offer the same functionality, the algorithms used here are simple enough to be implemented with-



(a) raw (b) background (c) subtracted

(d) thresholded (e) denoised

Figure 5.2: Processing stages for a sample FTIR image

out the need for an additional external dependency. Moreover, while OpenCV provides accelerated versions of those functions, too, they are not available under an open-source license but have to be paid for instead.

As mentioned previously, some hardware setups support the alternating acquisition of two images with different active illumination. This also means that for each type of illumination, a separate background image needs to be stored. Therefore, the entire image processing sequence has been encapsulated in a class `Pipeline` which also stores a separate background image and threshold parameters. Consequently, it is important to reliably determine which image (e.g, odd- or even-numbered) belongs to which light source. Two options for this decision exist. When using a camera based on the IIDC standard, this camera may be able to embed information such as the state of its output pins into the first few pixels of the image. Should such a feature not be available, the total brightness of the images can be used for differentiation, as it usually differs significantly depending on the switched lighting conditions. An additional option which is currently not implemented but could be easily added is based on an indicator LED. A suitably placed LED which is only active and visible in, e.g., odd frames could be employed to differentiate the images.

Image Analysis and Correlation

After the image has been converted into a binary representation, the most important step is now to analyse the properties of the foreground objects. The first operation is therefore to perform so-called connected component analysis, which combines coherent areas of pixels into a single object which is usually referred to as *blob*. For each blob, the so-called moments of first and second order [54] are calculated which can be used to describe the primary location, orientation and shape of the blob in terms of an approximated ellipse which has its centre point located at the centroid of the blob. The long and short axis of this ellipse are usually called major and minor axis of the blob. While this information is quite sufficient to describe, e.g., a finger contact spot, it lacks some aspects which are necessary for describing more complex blobs such as the outline of a user's arm.

Therefore, the next step after calculation of the moments is to locate the peak of the blob, should one exist. For the arm outline, this peak should correspond to the tip of an outstretched finger or the tip of a flat palm. To this end, the major axis of the blob is scanned outward from the centroid. Perpendicular to this axis, consecutive scanlines are searched for pixels belonging to the blob. When a scanline without such pixels has been found, the previous scanline contains the outermost pixel of the blob in one direction. This process is done twice for both directions of the major axis, resulting in two candidates for the blob's peak (see also figure 5.3). The final result is determined by a two-step process: if the blob has already been tracked in the previous frame, the peak closer to the previous one is used. Otherwise, the peak which is more distant from the image border is taken, following the rationale that a blob such as a hand appearing from the side will not have its point-of-interest located at the very border, but rather towards the image centre.

While all blobs have now been identified and processed for one single image, it is highly desirable to also track the blobs across images, i.e. assign a persistent identifier to each blob that does not change even if the blob moves or changes its shape. Therefore, the motion of each blob has to be estimated based on the blob positions in the previous frame. While more sophisticated solutions to this task such as Kalman filters [66] exist, the following simple algorithm has proven to work reliably. First, blob speed and centroid position from the previous frame are used to generate an estimated current position for each old blob. Next, the new blobs within a configurable radius around this estimated position are evaluated. The blob whose centroid is closest to the estimated new location is tagged with the old identifier and removed from the

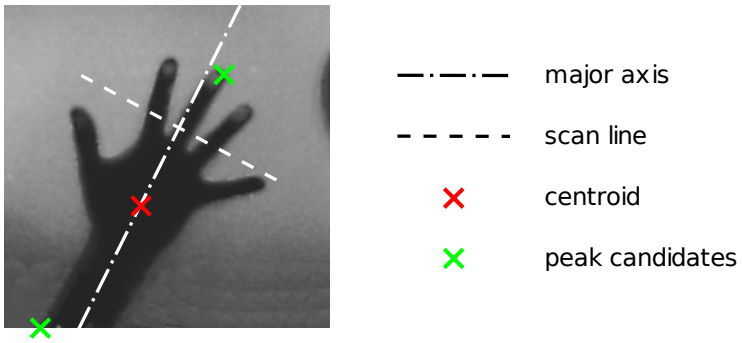


Figure 5.3: Peak detection

search list. If no new blobs have been found within the search distance, the identifier is removed. This process is repeated until all old identifiers have been reassigned or removed. The remaining new blobs are tagged with previously unused identifiers. This process is also illustrated in figure 5.4.

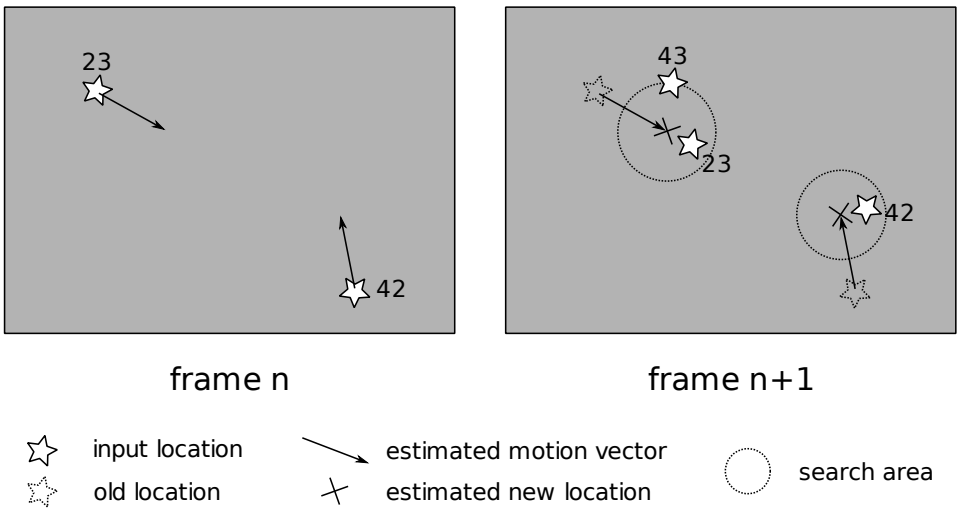


Figure 5.4: Blob tracking

The following step which only needs to be performed when two disjoint illumination sources such as an FTIR screen and a shadow tracker are used is correlation of contact and shadow blobs. At the centroid of each contact blob, the shadow image is examined. Should a shadow blob cover this location, its

identifier is set as parent ID for the respective contact blob, thereby allowing differentiation between contact spots belonging to different hands. This process is illustrated in figure 5.5.

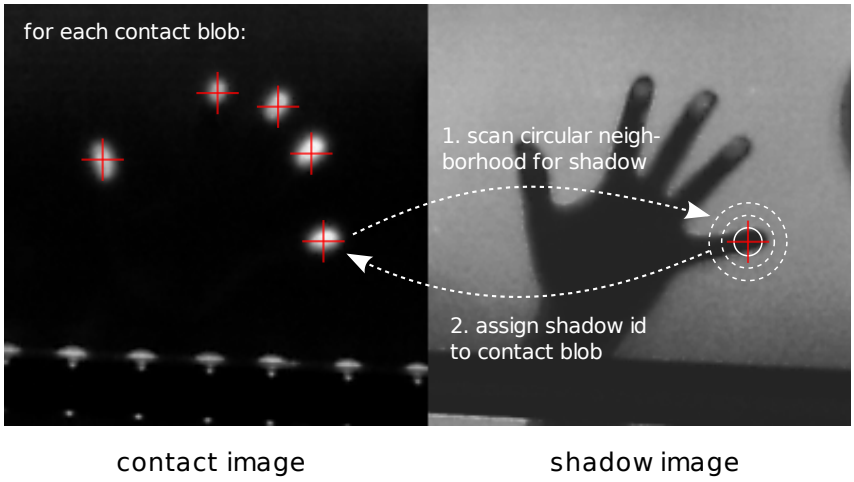


Figure 5.5: Contact-shadow correlation

After this final step, the blob data can now be transmitted to the following layer in LTP format. As mentioned above, libTISCH uses UDP as transport mechanism. The raw position data is therefore now sent to port 31408 on the local host by default. A parameter file which can be used to fine-tune the various image processing and tracking steps is described in appendix A.1.2.

5.3 Transformation Layer

As the data sent by the hardware abstraction layer is still described in terms of sensor coordinates, it is now necessary to transform this location data into screen coordinates so they can be easily correlated with the visible elements of the user interface. This task is fulfilled by the transformation layer in two major steps. The necessary parameters are stored in a file named `.tisch.calib`, which is located in the user's home directory or, should this directory not exist, in the `/tmp` directory. The format of this file is described in appendix A.1.1.

5.3.1 Removal of Lens Distortion

The first task, which is only applicable to camera-based interfaces, is the removal of lens distortion. Especially when using a wide-angle or fisheye lens, the original image may be heavily influenced by so-called barrel distortion. Usually, this step is first performed on the entire original image. However, this is a processor-intensive task which may degrade performance of the image processing, especially at high frame rates. Fortunately, when the foreground objects can be assumed to be small relative to the image size, it is sufficient to process the image as-is and perform the undistortion step only on the object positions. This is usually the case in this application, particularly for finger contact spots and tangible objects.

The undistortion process relies on the assumption that an ideal camera would linearly map world coordinates to image coordinates according to a function $r = f(x) = ax$, with x being the distance between world point and camera baseline and r being the distance between related pixel and image centre. When a lens is used, it tends to warp the mapping into a more complex function which is usually approximated as $r = ax + bx^3$ [70].

To revert this warping, the coordinates of an object now have to be scaled according to their distance from the image centre. While the image centre is not automatically equal to the optical centre, we have found this assumption to be sufficiently accurate within this context, as the difference is usually only on the scale of few (1-2) pixels.

The two parameters a, b from this equation have to be estimated once for a given combination of camera and lens. As the lens distorts straight lines into curves, the estimation can be done based on lines in the image which are known to be straight. In many computer vision systems, the estimation is therefore done based on an image of a chequerboard pattern. However, in some interactive setups such as the MiniTISCH, it can be next to impossible to reliably position such a pattern in front of the camera without obstruction by the screen. Disassembling the entire system to take an image of the pattern is usually also beyond question. Therefore, a quick-and-dirty solution to this problem is to adjust the parameters interactively using visual feedback. The `distort` utility displays the camera image and allows the user to unwarp it by adjusting the parameters. The keys `1` and `2` modify the parameter a while `3` and `4` change b. The current parameters are displayed on-screen and can be transferred to the calibration file by pressing `s`. Although this ad hoc process is less accurate than an automated parameter estimation, it has nevertheless proven to be sufficient for this application.

5.3.2 Perspective Correction

After the transformation from world coordinates to image coordinates now has been linearised, the second step is to determine the perspective transformation from sensor to screen coordinates. This step is also applicable for non-camera sensors and accounts for such factors as the sensor being off-centre or even upside down, the scaling being different etc. As the number of parameters is too high to be estimated manually in this case, a calibration procedure has to be performed. Luckily, this step does not require line patterns, but instead can be reliably calculated using point correspondences. The calibration tool sequentially displays four crosshair markings in the corners of the display and prompts the user to touch (or activate) each one in turn. Samples are taken continuously, and if the jitter of the active position falls below a certain threshold, an average over the last 30 samples is taken and stored together with the screen coordinates of the current crosshair.

After the minimum of four correspondences has been captured, the correspondences are stored in two matrices, forming an equation system $a_n = Xb_n$. By applying a singular value decomposition as described in the well-known book by Hartley and Zisserman [47], the third matrix X representing the transformation between the two coordinate systems can be estimated. The singular-value decomposition algorithm used here is based on the popular Template Numerical Toolkit [96]. The resulting matrix is also stored in the calibration file.

5.3.3 Online Transformation Process

When a suitable calibration file has been generated through `distort` and `calibtool`, the calibration daemon `calibd` can be started to continuously convert raw data packets from the hardware abstraction layer into calibrated data which is represented in screen coordinates. By default, this daemon listens on port 31408 and sends the transformed packets to port 31409[1]. These packets are now ready to be processed by the gesture recognition layer. Should the screen resolution change, the calibration process has to be repeated. To quickly access this feature, the daemon can be sent the *hangup* signal on Unix systems, which will cause it to quit and start the calibration tool instead. After the recalibration has finished, the daemon will be automatically restarted.

[1] 0x7AB1 in hexadecimal notation

5.4 Gesture Recognition Layer

One of the most central components of libTISCH is the generic gesture recogniser. As described previously, this layer receives abstract region and gesture definitions as well as input data and tries to match gesture descriptions to the incoming movements. When a match has been found, the abstract gesture template is converted into a concrete instance and sent back to the corresponding region.

Note that due to the use of UDP as communications protocol, it is entirely possible to send updates for default gesture definitions from one client while another client is already active. This allows to redefine the default gestures on-the-fly even without explicit support by the primary client.

5.4.1 Gesture Matching Algorithm

The `gestured` process consists of two threads. The first thread listens for region and gesture descriptors on UDP port 31410 and updates an internal list containing `Region` objects and their associated `Gesture` objects. This list is properly locked prior to any modification to avoid race conditions with the main thread, which uses this list to retrieve the gesture templates. Additionally, every region object inside this process contains an associated "input state" object which maintains the state and history of all input objects that have fallen inside the region. When regions are received from a client process, the source address and source UDP port of the received packets are stored. Recognised gestures are sent back to this stored address and port, thereby automatically establishing a two-way communications channel.

The main thread listens on UDP port 31409 for input data packets which already have been transformed into screen coordinates. The variables and instruction sequence used by the main thread are given in figure 5.6.

The three main variables are the list of regions (which is shared with the update thread as mentioned above), a dictionary which describes the IDs of those input objects which are currently attached to a sticky region and a flag which is set when new, previously unseen IDs have arrived.

After an input data packet has been received, it is first checked whether it contains position information or a frame marker.

Should it contain position data, the algorithm checks whether this identifier has been received for the first time. If this is the case, a flag is set to indicate that one or more new identifiers have appeared. Next, the list of currently active sticky regions is searched for this identifier. If a match is found, the

```
vector <Region> regions
map <id,Region> stickies
bool newid

while true do
    receive packet p
    if type of p is "input data" then
        process_input p
    if type of p is "frame marker" then
        process_gestures
```

Figure 5.6: Gesture matching algorithm

blob is immediately appended to the input state of this region. Otherwise, all regions starting from the topmost one are checked whether they have the appropriate flag for this specific type of input data set and whether the position is inside the region. Should both conditions match, the input data object is appended to the region's current input state. Otherwise, when no region with the correct flag set contains the input position, it is discarded. This process is shown in figure 5.7.

```
begin
    if id of p is new then
        newid := true
    if stickies contains id of p then
        get matching region s from stickies
        append p to s
    else
        foreach region r from regions do
            if r contains position of p then
                if r is flagged with type of p then
                    append p to r
                    break
end
```

Figure 5.7: process_input(packet p)

In the case that the packet contains a frame marker, the actual gesture recognition can now be performed, as at this point the entire input data for

this frame has been received. First, should the flag for new identifiers have been set, all currently active sticky regions have to be updated from the widget layer. The rationale behind this step is that the graphical representation of an active sticky region may have changed regarding its shape and position without notifying the gesture recognition layer. While existing identifiers which have properly been assigned to a sticky region will be sent to that region regardless of its position, this does not hold for newly appearing input data. Should a new input spot be created, it may therefore not lie within the currently stored region, even if it appears that way to the user. Therefore, the two layers have to be re-synchronised at this point.

```
begin
    if newid == true then
        foreach region r from stickies do
            request update for r
        foreach region r from regions do
            if r has flag "volatile" then
                request update for r
        clear stickies
    foreach region r from regions do
        foreach gesture g from r do
            foreach feature f from g do
                update f with each inputdata i from r
            if all features match then
                flag gesture as match
        foreach match m from r do
            if m has flag "one-shot" then
                if inputdata i for r has not changed then
                    continue
            if m has flag "sticky" then
                foreach inputdata i from r do
                    add id of i and r to stickies
            transmit m to id of region r
    newid := false
end
```

Figure 5.8: process_gestures()

Now all regions can be separately checked for gesture matches. In a preparation step, first all features contained within the gesture templates are loaded with the stored input data from the region. As the input data will not change anymore within this frame, the features need to be loaded only once. After this process, the matching gestures for the current region can now be processed. Note that a single gesture template may result in several matches depending on the features used. For each match, the *one-shot* flag is checked first. Should the set of input identifiers for this region have changed with respect to the previous frame, the gesture is transmitted. In the next frame, it will not be sent again unless the currently active input identifiers change. Gestures for which the *sticky* flag has been set also need additional handling. Should such a gesture have matched, the input identifiers which are currently active for this region are added to the list of sticky IDs along with a pointer to the region itself. This ensures that further actions from these identifiers will be transmitted to this region regardless of synchronisation issues. Finally, the gesture is transmitted back to the widget layer with the region's ID. The entire gesture matching algorithm is shown in figure 5.8.

5.4.2 Default Gestures

A central feature of the protocol described in section 3.3 is the ability to specify gestures by name only and retrieve the specific features from a pool of default gestures. One set of such default gestures is loaded by the `gestured` process at startup which consists of the following definitions:

Gesture Name: *move*

Flags: none

Feature: *Motion*

Flags: `0xFF` (i.e. all object types)

Boundaries: none

Gesture Name: *rotate*

Flags: none

Feature: *MultiBlobRotation*

Flags: `0xFF` (i.e. all object types)

Boundaries: none

Gesture Name: *scale*

Flags: none

Feature: *MultiBlobScale*

Flags: `0xFF` (i.e. all object types)

Boundaries: none

Gesture Name: *tap*

Flags: *one-shot*

Feature: *ObjectID*

Flags: `0xFF` (i.e. all object types)

Boundaries: none

Feature: *ObjectPos*

Flags: `0xFF` (i.e. all object types)

Boundaries: none

Gesture Name: *release*

Flags: *one-shot*

Feature: *ObjectCount*

Flags: `0xFF` (i.e. all object types)

Boundaries: lower boundary: 0, upper boundary: 0

The definitions of the first three gestures are straightforward, as each one of them can be mapped to a single feature without boundaries. Note, however, that the *MultiObjectRotation* feature in the definition of the "rotate" gesture could be replaced by the *RelativeAxisRotation* feature when a sensor is present that is able to capture the rotation of a single object. As both features are derived from an abstract *Rotation* feature, the redefinition will be transparent to any application.

The definition of the last two gestures is slightly more complex. The "tap" gesture has the *one-shot* flag set and will therefore be only triggered once when a new input ID appears within the containing region. The *ObjectID* and *ObjectPos* features will deliver the ID and position of this new input ID. Note that as both are multi-match features, this gesture can be triggered multiple times in a row, e.g., if several new input IDs appear simultaneously.

The last default gesture, "release" is also flagged as *one-shot* and contains an *ObjectCount* feature with both boundaries set to zero. This gesture will therefore be triggered once when the number of objects within the containing region reaches zero.

5.4.3 Performance

One crucial aspect is the performance and latency impact which is caused by the separately implemented gesture recogniser. In any kind of interactive application, avoiding latency is of high importance, as it can significantly degrade the user experience. To test the latency introduced by the gesture recogniser, it was used in conjunction with the mouse-based frontend (see also section 5.2.1). When one of the mouse buttons is pressed, a timestamp is taken inside the frontend. The mouse adapter starts sending simulated input data packets to the gesture daemon at this point, which processes them and sends gesture events back to the frontend. After an event has been received which was triggered by the button press, a second timestamp is taken. The difference between the two values is the total latency added through two UDP transmissions and the gesture processing step in-between.

A test with 100 samples resulted in an average latency of approximately 3.43 ms with a standard deviation of 1.99 ms. The high variation results from the granularity of the internal timer of the mouse data transmitter. However, even with this ad hoc measurement, it is safe to conclude that the additional latency introduced through the separate gesture recogniser is within acceptable limits for an interactive system. These tests were conducted on a dual-core machine with an Intel *Core2* processor running at 2.0 GHz under Linux 2.6.29.

5.5 Widget Layer

The final part of libTISCH is the widget layer which displays graphical objects that react to gestures recognised by the previous layer. This layer has to accomplish two subtasks besides the most basic one of drawing the widgets. First, it needs to determine the screen coordinates and therefore the regions covered by the various widgets and relate them to the gesture recogniser, updating them when necessary. Second, it needs to react to recognised gestures, updating the widgets' state accordingly.

5.5.1 OpenGL-based Widgets

The reference widget layer is based on OpenGL and C++ for the best cross-platform compatibility possible. OpenGL offers various options for maintaining a stack of transformations, thereby making it easy to render a tree of nested objects. One difficulty when using OpenGL is determining the screen coordinates of graphical objects, as the original coordinates are usually modified by several transformation matrices such as the modelview and projection matrix. A possible way to solve this problem is the so-called *feedback mode* in which transformed vertices are not drawn into the framebuffer, but instead written to a separate storage array from which their screen coordinates can then be retrieved. However, this approach suffers from the drawback that feedback mode is often only poorly supported by many OpenGL implementations. Moreover, it can cause a significant number of graphic context switches which usually result in a severe performance hit. Therefore, a faster and more reliable solution to this problem is to retrieve the current modelview and projection matrix when needed and explicitly multiply the vertices with these matrices in the userspace part of the library. Additionally, not all vertices which are needed to draw the widget are also necessary to define its outline and thereby the region which should be registered. Each widget should consequently implement two different methods, called `outline()` and `draw()` to handle these separate tasks. These methods are defined in the abstract base class `Widget` from which every concrete widget implementation should be derived.

The most basic widget which does not have a graphical representation of its own is the `Container` which just groups other widgets. In this context of the parent-child relationship between container and sub-widgets, an important task is to determine the absolute rotation of the widget itself with respect to the screen coordinates, as all movement and location data within the delivered features is in screen coordinates. Especially if the widget is inside a container which itself has been rotated relative to the screen, then all vector data inside received features first has to be converted into the parent container's coordinate system. To this end, the two unit vectors $(1.0, 0.0)$ and $(0.0, 1.0)$ from the widget's local coordinate system are also projected into screen coordinates, resulting in two transformed vectors from which the absolute rotation as well as scale can be determined. A widget can then hand a received vector to its parent widget to transform this vector into its coordinate system. Afterwards, this vector can be used to transform the widget's own pose, thereby correctly applying the transformation within the parent's coordinate system.

The following widgets are available:

`Widget` The abstract base class from which all other widgets are derived. A new widget should implement the virtual methods `action()` (handles received gestures), `outline()` (defines the region in local coordinates) and `draw()` (paints the widget on-screen). Additionally, the member object `region` should be filled with the desired gestures.

`Button` This rectangular widget accepts the two most basic gesture events, "tap" and "release". Two virtual methods of the same name are called accordingly. Additionally, its look can be customised with an arbitrary texture.

`Tile` The tile widget is derived from the `Button` class and adds handling for the remaining three simple gesture events "move", "scale" and "rotate". An instance of `Tile` can therefore be transformed by the user without having to add further code.

`Dial` The dial widget displays a round knob which can be rotated by the user to adjust its value. This can either be done through the default "rotate" gesture or by moving a single interaction point tangential to the widget's rim. The current percentage is displayed on top of the dial. To avoid capturing events outside the circular widget, the `Dial` uses an octagonal region to better approximate the true widget shape.

`Textbox` This widget displays a text field as it is commonly used in many desktop interfaces. However, while a standard text box can be filled using the keyboard, this has some issues, especially in the context of novel user interfaces. A normal keyboard may be unavailable, or several users might want to enter text simultaneously. Therefore, this widget expands to reveal a small on-screen keyboard when tapped. After text entry, a second tap on the text itself will shrink the widget back to its original size, hiding the keys.

`Container` This widget is derived from `Tile`, but offers the additional methods `add()`, `raise()` and `remove()` in order to manage child widgets. Should this widget be transformed, either in code or through the standard gestures inherited from `Tile`, then all sub-widgets will be transformed accordingly.

`MasterContainer` This class is of special importance. It is derived from `Container` and handles the additional task of registering the regions of all directly or indirectly contained widgets with the gesture recognition layer.

Moreover, it receives gesture events and delivers them back to the correct widget. For this reason, only one MasterContainer can currently exist per application.

`Checkbox` The checkbox widget is derived from `Button` and acts as a toggle switch. Upon receiving a tap, it will alternate between a checked state which displays a cross and an unchecked, empty state.

`Slider` The slider widget, just like `Dial`, allows the user to adjust a numerical value. Here, this can be done by linearly moving the sliding part along its "tracks" as opposed to rotating the widget.

`Label` This completely passive widget has been added for convenience. Even completely novel user interfaces will very likely require text labels. Note that as the label widget doesn't register a region, it is completely transparent to any input objects and will not receive any gestures.

An additional important class which is not a widget itself is `Window`. This class wraps a GLUT window, which is itself an abstraction from the various low-layer OpenGL implementations, and therefore offers a platform-independent way of creating an OpenGL context. `Window` is also derived from `MasterContainer` in order to automatically register all widgets within this context with the gesture recognition layer. Currently, only one `Window` object can therefore be opened within one process. However, this is only a minor limitation, as most applications built on this framework can be expected to run in fullscreen mode, thereby requiring only a single window from the start. Elements like popup menus do not require top-level windows, but can instead be created from the framework's own widgets.

In figure 5.9, most of the standard widgets are shown in an example application. Although this sample shows only the default textures, all widgets can be textured individually. Note that the textbox widget is currently in its expanded state, showing the on-screen keyboard.

To demonstrate the ease with which an application can be built with libTISCH, consider the example given in listing 5.1. This example provides a basic picture browser in which every picture reacts to the previously described default gestures for motion, rotation and scaling.

The main steps in this application are creation of a new `Window` object, creation of a texture for each image passed on the command line and finally the creation of one `Tile` object for each texture with randomised position and

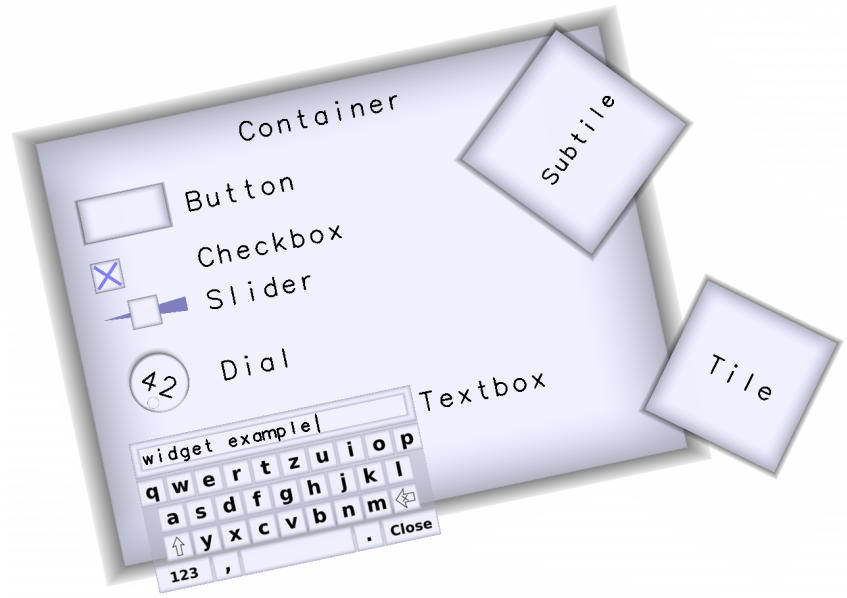


Figure 5.9: Widget examples

angle. Note that although the window size and the range for widget positions are predetermined, the window will correctly handle resizing, maximisation etc. Immediately after allocation, each tile is added to the window (which is also a `Container`). Finally, all widgets are registered with the gesture recognition layer and the main loop is started.

5.5.2 Widget Bindings for Other Languages

As the reference implementation is written in C++, the resulting native libraries offer the possibility of being bound to other popular languages such as Python or Java. This process is vastly simplified through the use of the *Simplified Wrapper and Interface Generator (SWIG)* [5]. By describing the classes to be wrapped in a meta language, a wrapper library and classes for a variety of languages can be generated. All widget classes are wrapped, including `MasterContainer`. As the creation and management of OpenGL contexts may differ widely between languages, the `Window` class is not part of the wrapper. Therefore, when using the wrapper library in a different environment, care should be taken to create and activate an OpenGL context before any of the drawing or registration functions are called. One reference wrapper which

is already included with libTISCH is that for Java. After using *JOGL* [65] to create an OpenGL context, a `MasterContainer` object can be instantiated and widgets added to it. The popular graphical development environment *Processing* [41] which is based on Java can thus now also be used to develop libTISCH-based applications. An example on how to use libTISCH's widgets within an OpenGL context provided by Processing is given in listing 5.2.

5.5.3 Class Diagram

To provide a concluding overview of the most important classes and their relationships, particularly for the interpretation and widget layers, a *Unified Modeling Language (UML)* inheritance diagram is given in figure 5.10.

Summary

In this chapter, the reference implementation libTISCH of the previously designed architecture has been described. libTISCH offers support for a variety of input hardware devices and frees the developer from the task of reimplementing gesture recognition algorithms again and again. Moreover, it even allows to swap the definitions of gestures depending on the hardware used. A generic widget layer based on OpenGL is also part of libTISCH. It uses the gesture recogniser to react to user actions and provides various ready-made common user interface elements.

```
#include <tisch.h>
#include <Window.h>
#include <Tile.h>

int main( int argc, char* argv[] ) {

  // create a new window,
  Window* win = new Window( 640, 480, "PicBrowse" );

  // initialize random number generator
  srandom(45890);

  // read all pictures specified on command line
  for (int i = 1; i < argc; i++) {

    // load image into new texture
    RGBATexture* tex = new RGBATexture( argv[i] );

    // create a tile widget with random position
    Tile* tile = new Tile(
      tex->width()/10, tex->height()/10, // size
      rand()*600-300, rand()*450-225,    // position
      rand()*360,                        // orientation
      tex                                // texture
    );

    // add to window
    win->add( tile );
  }

  // perform initial registration and start main loop
  win->update();
  win->run();
}
```

Listing 5.1: libTISCH Picture Browser Example

```
import javax.media.opengl.*;
import processing.opengl.*;
import libtisch.*;

MasterContainer master;
int first = 1;

void setup() {

  size(640, 480, OPENGL); // create window
  Loader ltload = new Loader(this); // load wrapper library

  RGBATexture tex = new RGBATexture( "color9.png" );
  Tile widget = new Tile( 50,50, 50,50, 0.0, tex );

  master = new MasterContainer(640,480);
  master.add(widget);
}

void draw() {

  PGraphicsOpenGL pgl = (PGraphicsOpenGL) g;
  GL gl = pgl.beginGL(); // get OpenGL context

  // reset OpenGL matrices
  ortho( 0, 640, 0, 480, -1000000, 1000000 );
  background(128);
  resetMatrix();
  gl.glMatrixMode(GL.GL_MODELVIEW);
  gl.glLoadIdentity();

  // perform first update
  if (first != 0) { master.update(); first = 0; }

  master.draw(); // redraw master container
  pgl.endGL();
}
```

Listing 5.2: libTISCH Processing Wrapper

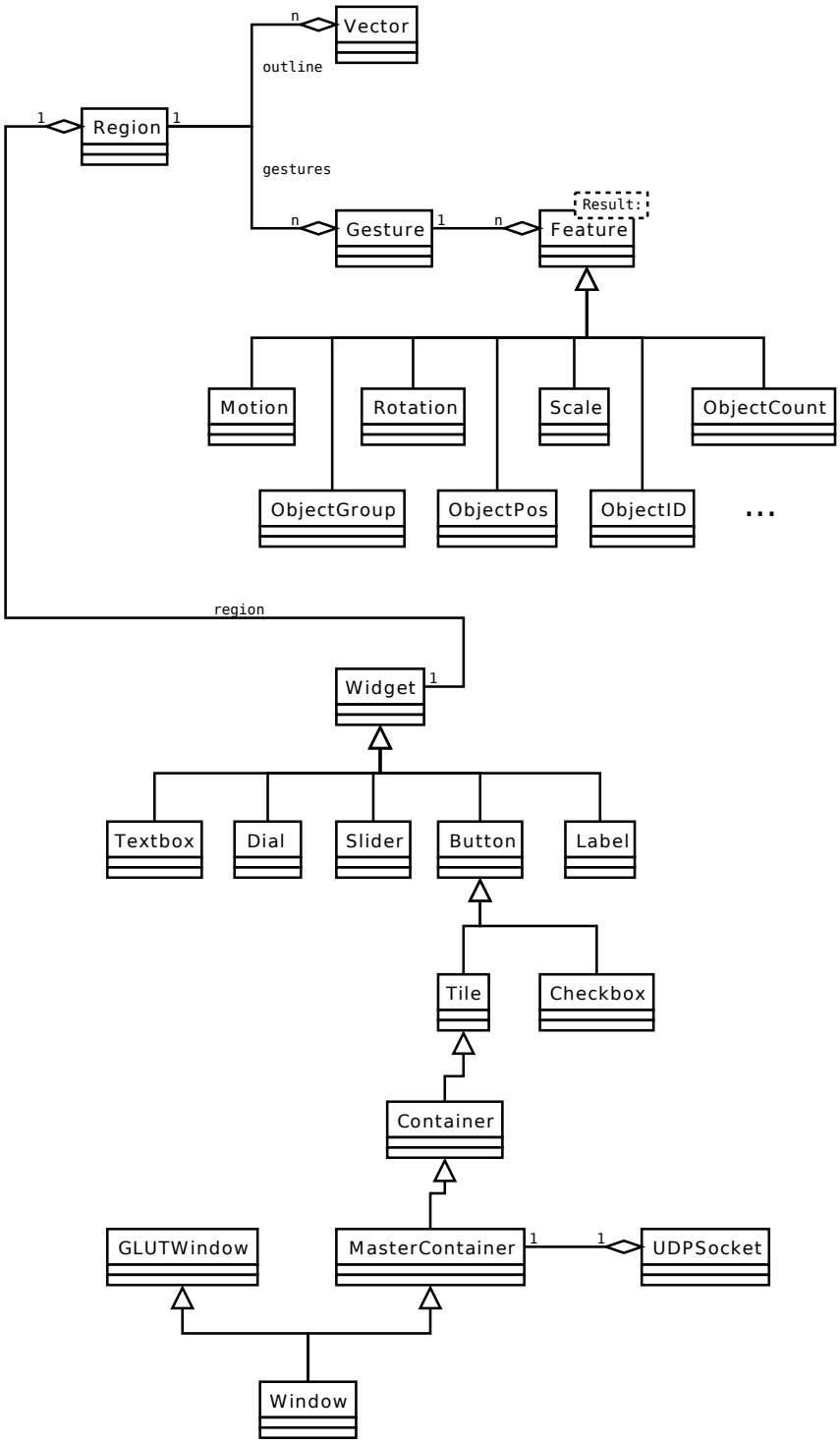


Figure 5.10: libTISCH interpretation/widget layer class diagram

Chapter 6

Applications

In this chapter, various real-world applications of libTISCH and the suitability of libTISCH for their implementation shall be reviewed. While some of the applications presented here require special capabilities which can currently only be provided by one specific input device from those available, most of these programs can indeed be run on different sensor and display hardware without modification.

6.1 Interfacing with Legacy Applications

When using a direct-touch input device, it is often still necessary to switch to desktop input devices such as keyboard and mouse for accessing standard desktop applications, e.g. a web browser. For quickly looking up some information or for operating an information booth such as the SiViT (see section 4.2.3), it is desirable to use such software without the need for legacy input devices.

6.1.1 Pointer Control Interface

The easiest way to support such interaction between standard applications and novel input hardware is to control the mouse pointer (or, in the case of MPX, pointers) using input data from the hardware abstraction layer. For a normal mouse-based application, no difference in the received events will be perceptible.[1]

[1] When using MPX, applications may sometimes react unexpectedly when more than one mouse pointer is within their window at once.

There are two basic methods to control the mouse pointer from within a userspace program. The first, portable method uses a Java application to send mouse events to the operating system [120]. This is supported by the Java class `Robot` which is available on all supported platforms. However, the Java privileges have to be set correctly to allow this class to actually access the device. Moreover, this approach is limited to a single mouse pointer, as most operating systems so far do not support multiple pointers. Should several instances of input data exist, the mouse pointer will remain locked to the one which was detected first.

Therefore, the second method which is only available on X11-based systems uses the XTest extension to control the pointer(s). Should an MPX-capable X server with a suitable version of XTest be installed, this extension allows to control each pointer independently of the others. Otherwise, the behaviour is similar to the Java method.

6.1.2 Gestures for Mouse Emulation

Depending on the available features of the input device, various gestures can be employed to control the pointer. The first distinction is whether the device is able to differentiate between a "hover" and a "touch" state. The "hover" state corresponds to just moving the mouse, while the "touch" state corresponds to clicking and/or dragging the mouse. In the default configuration, "blob" input events are used to control the pointer's motion, while "finger" events which are associated to the blob trigger a click with the primary mouse button.

When this distinction is not supported by the hardware, two options are available. The first and most straightforward option is to execute any emulated mouse movement in conjunction with an emulated button press. Should the hardware be unable to deliver "blob" events (or equivalent data), each "finger" event will trigger a movement with pressed button. However, emulating mouse movement with a constantly pressed button will invariable have some adverse effects on the applications, such as involuntarily selecting any text the mouse is moved over. A different option which does not exhibit such effects is to translate touch and movement with a single finger into pointer movement only. Button press events are not triggered until a second finger is put down.

To support the more complex interactions which are possible with mice such as scrolling or context menus, the number of additional "finger" objects is evaluated. Tapping and dragging with one additional finger causes scroll events to be sent, whereas tapping with two additional fingers triggers a secondary-button click. While these modes of interaction may seem arbitrary at first,

they are well-known to many laptop users, as recent touchpads offer similar multi-finger behaviour.

6.1.3 Discussion

To test the real-world behaviour of the MPX-based mouse emulator, it was used in conjunction with the Wiimote as well as the iPhone. While the iPhone itself does not allow any mouse-based interaction, it can nevertheless serve as a multitouch-capable, wireless touchpad for a common desktop interface. As libTISCH applications are inherently network-transparent, such a setup can be easily provided. A standard web browser (Firefox) was started on a common laptop, but controlled through the mouse emulator.

Although no quantitative evaluation was performed, using a standard mouse-based application with novel input devices yielded an important insight. As most common user interfaces take advantage of the high precision which is available through the mouse, using them with a less precise device such as the Wiimote can be challenging. As the Wiimote connector requires the user to push one of the device's buttons to trigger interaction (see section 5.2.2), this motion can cause the cursor to jump away from the originally targeted location, sometimes causing a short drag-and-drop event which is mostly undesired. Moreover, the cursor showed noticeable jitter due to small, involuntary hand movements which were amplified by the distance between Wiimote and infrared light source.

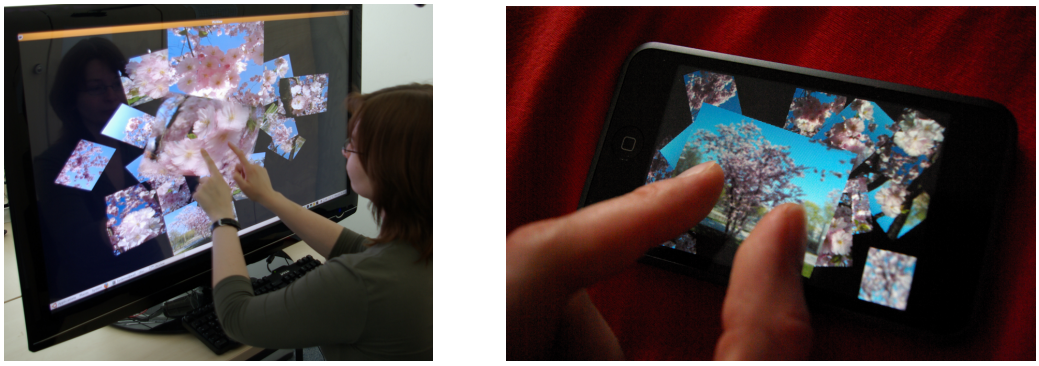
With the iPhone, a surprisingly accurate control of the pointer was possible. Due to lack of a hover state, the second option described above was used in which a click had to be triggered through a second finger, while scrolling was done through dragging with three fingers. Due to the high accuracy of the iPhone's touch sensor, a user experience similar to a standard touchpad was possible. One noticeable drawback in this setup was due to the resolution and size differences between the two screens. The fingers frequently had to be lifted off and placed at the other end of the iPhone screen in order to traverse the entire laptop screen with the cursor. While this effect could be mitigated by adding a scaling factor to the mouse emulator, this would cause the setup to lose some precision, thereby causing other unwanted effects as with the Wiimote.

6.2 Casual Entertainment

As mentioned earlier, one of the most prevalent class of applications for multi-touch and multi-user interfaces is casual entertainment software. To showcase the abilities of libTISCH, several such applications have been created.

6.2.1 Picture Browser

Probably *the* most common application for novel interfaces is conceptually quite simple: the picture browser. Usually, a number of pictures are scattered randomly across the entire interactive surface and can be manipulated by users with the common move/scale/rotate gestures. It has already been shown in section 5.5.1 that such an interface can be easily implemented with libTISCH by using *Tile* widgets. In figure 6.1, the same application is shown running on two conceptually quite different hardware devices, thereby showcasing the abstraction capabilities of libTISCH with respect to input as well as output.



(a) on FlatTouch (b) on iPhone

Figure 6.1: Picture browser

A small modification to this example serves to underline the ease with which the functionality of libTISCH applications can be extended. Instead of *Tile* widgets, their child class *Container* can also be used. Through the container functionality, it is now possible to add, e.g., an annotation field to each picture by adding a single line of code compared to listing 5.1. This code snipped is shown in listing 6.1.

This small change results in the user interface shown in figure 6.2. Of course, additional code would be needed to reliably save and restore these

```
...

// create a container widget with image texture
Container* cont = new Container( ..., tmp );

// add textbox with size and position relative to image size
cont->add( new Textbox(
  tmp->width()/20, tmp->height()/200, // size
                0, tmp->height()/20   // position
) );

// add to window
win->add( cont );

...
```

Listing 6.1: Adding annotation fields

annotations; however, such functionality is beyond the scope of libTISCH.

Figure 6.2: Picture annotations

6.2.2 Sudoku

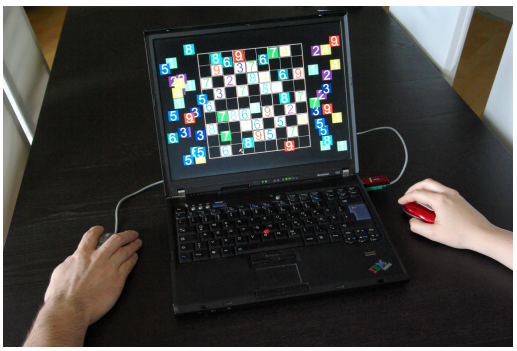
Maybe the most common type of interactive software is games. A very popular pen-and-paper game is Sudoku, in which the player has to fill a usually rectangular grid with numbers so that every number occurs only once in each row, column and sub-grid. Many variants of Sudoku exist, using larger grids, different symbols, non-rectangular grids and more. Interestingly, although Sudoku originally is a single-player game, it can very well be played collaboratively by several persons. Also due to its complexity, it has on several occasions been used as a test case for user interfaces [72]. Therefore, Sudoku provides a well-suited example for a complex multi-user interface and has therefore already appeared at various locations throughout this thesis.

An example of the interface is shown in figure 6.3(a). Tiles with white background represent fixed numbers which can not be changed by the user, while tiles with coloured background are movable and can be slotted into the fields of the grid. The interface was inspired by the browser game JigSawDoku [77]. On several occasions such as the faculty's "Open House" day, casual users were observed playing the game collaboratively (see figure 6.3(c)). Notable observations in this context were that up to eight persons could collaborate effectively and solve several puzzles. Users tended to divide the remaining numbers among them so that one person was responsible for a specific number.

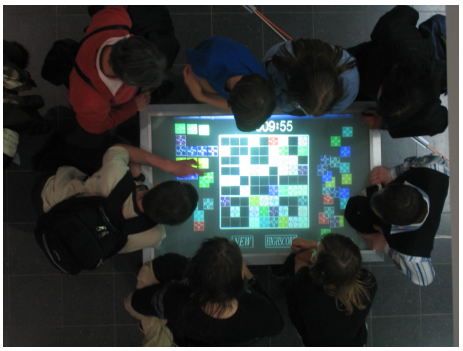
As the image clearly shows, users distribute themselves evenly around the table. Therefore, no assumptions regarding the viewing direction can be made for this setup. However, the numbers on the game tiles of course have one single preferred viewing direction. The first iteration of this game featured tiles which carried every number four times for the four primary directions. Unfortunately, this arrangement created a significant amount of visual clutter when the grid started to fill up, thereby confusing users. The most straightforward solution is to allow the user to turn the tile's content so it faces in the most comfortable viewing direction. As it would be inconvenient to require every single tile to be rotated individually, all tiles which share the same number always rotate synchronously. This approach also takes advantage of the fact that users usually do not mix numbers between them, i.e. only one user is likely to look for a specific number at one time. An example where a single orientation is sufficient is shown in figure 6.3(b). Here, two users collaborate at a standard laptop computer using two mice in parallel.

The implementation of this game is very straightforward. All tiles are instances of *Tile* which have their "scale" handler disabled. Moreover, they feature a custom "rotate" handler which does not rotate the entire tile, but

(a) Closeup with single player



(b) Playing Sudoku with multiple mice



(c) Collaborative Sudoku playing

Figure 6.3: Sudoku game

only its content and propagates this change to all tiles which share the same number. Finally, for the fixed tiles, the "move" handler has also been disabled.

Another important feature of this system is that the game logic is handled by a separate server process. This allows collaboration on the same game between several disjoint devices.

6.2.3 Virtual Roaches

While this game seems to fall squarely into the "nonsense" category at first sight[2], it nevertheless serves to illustrate an important capability of libTISCH, namely the use of arbitrary tangible objects. In this application, any trackable

[2] Although any practical use of this application seems hard to find, it has been suggested to use it as a tool for therapy of people who suffer from phobia of insects. Running this

object placed on the interactive surface will attract virtual roaches which hide beneath the object as illustrated in figure 6.4. The user can then remove the object to reveal the roaches which can be squashed by touching them. This application can currently only be run on the TISCH device due to the simultaneous requirement of arbitrary object tracker and touchscreen.

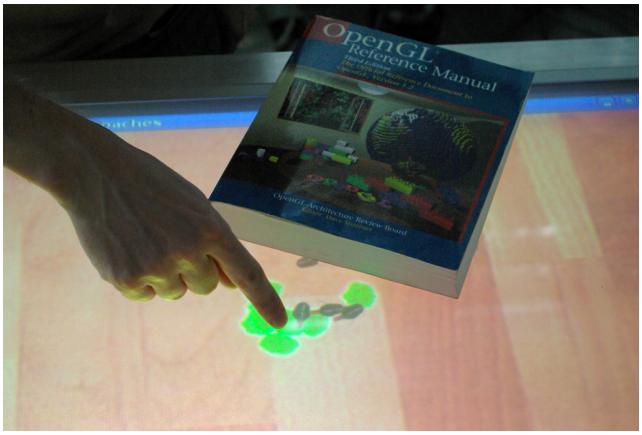


Figure 6.4: Hunting virtual roaches hiding under a book

In this application, two different kinds of custom widgets are used. The first widget covers the entire screen and is responsible for displaying the background as well as registering for "shadow" events which indicate appearance, movement or removal of an object on the surface. This information is propagated to the roach widgets which change their movement accordingly. These objects are derived from the `Button` widget class and therefore react to "tap" events.

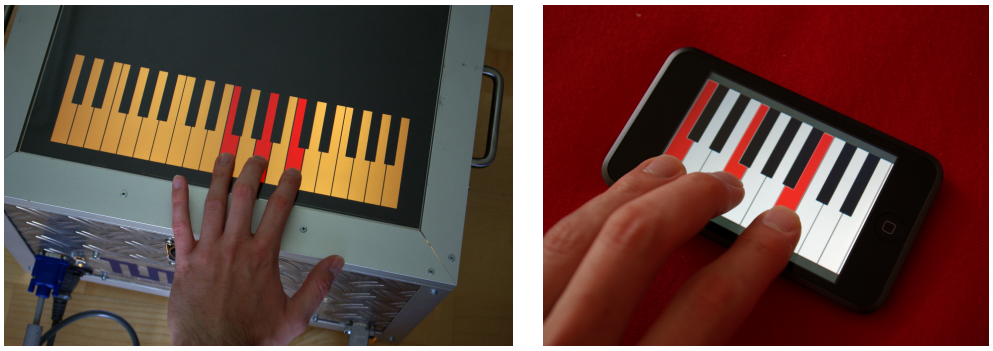
While the program is running, the roach widgets move on their own without user interaction. This highlights the important issue of synchronisation between the widget layer and the gesture recogniser, as the roaches will quickly move beyond their originally registered position. The gesture recognition layer requests a region update when a new input identifier has been received so that it can be checked against the true positions of the regions. However, only volatile regions or those with sticky gestures will be updated. As the roach widgets do not contain any sticky gestures, they have therefore to be flagged

application on a front-projected system such as the SiViT would cause the virtual roaches to appear on top of any object such as a hand and might help the user to gradually cope with the resulting panic.

as volatile.

6.2.4 Tangible Instruments

Another very accessible application for tabletop systems are virtual musical instruments. A virtual piano, for example, is possibly the most intuitive multi-touch application imaginable, as almost any person will instantly recognise a piano keyboard and its inherent ability to react to several keystrokes at once (see figure 6.5).



(a) on MiniTISCH (b) on iPhone

Figure 6.5: Playing a chord on the virtual piano

Regarding the implementation, this application is quite simple, as it only consists of black and white button widgets which are layered on top of each other. When the "tap" method of one button is called, an appropriate MIDI event is sent to the sequencer interface. This can either be a hardware device connected through a MIDI interface or a software synthesiser which generates sound through the computer's audio interface. In terms of sound latency, the hardware synthesiser is usually the better choice as software implementations often require a significant amount of processing time, especially when generating multiple concurrent sounds. Note that the iPhone lacks a software synthesiser and can therefore only be used in conjunction with an external MIDI device.

A different, more complex music application is *Beatring*. It was inspired by BeatBearing [8], an application in which steel bearing balls can be placed in slots above a flat-panel display. On this display, a bar is shown which

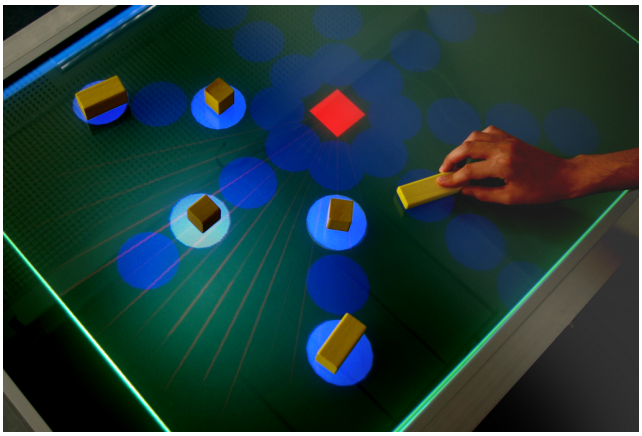


Figure 6.6: Using Beatring with plain wooden blocks

repeatedly scans from left to right below the slots. When the bar passes beneath a slot containing a steel sphere, a sound depending on the row is triggered. This is very similar to the drum sequencer interfaces found in many professional music applications.

Beatring extends this concept by bending the linear rows into a circle which is scanned by a virtual clock hand. A dial in the centre of the circle can be turned using the usual multi-touch gestures to slow down or speed up the rotation speed and therefore the playback speed of the drum sequence. Instead of the electrical connections used in BeatBearing, this application relies on the shadow tracker from TISCH to allow any opaque object to serve as a trigger for sounds as shown in figure 6.6.

Another mode of operation is possible with a top-projected surface or regular LCD screen. Fiducial markers can be tracked through Ubitrack using a top-mounted camera, thus enabling the same application to be used without requiring a touch-sensitive surface or a shadow tracker. The position data is delivered through the Ubitrack adapter described in section 5.2.1. This setup is shown in figure 6.7.

6.3 Interaction with Mobile Devices

Nowadays, one can reasonably expect a person who is interacting with a tabletop display to also carry a mobile phone. As most mobile phones are capable computers themselves with advanced features such as 3D graphics, wireless

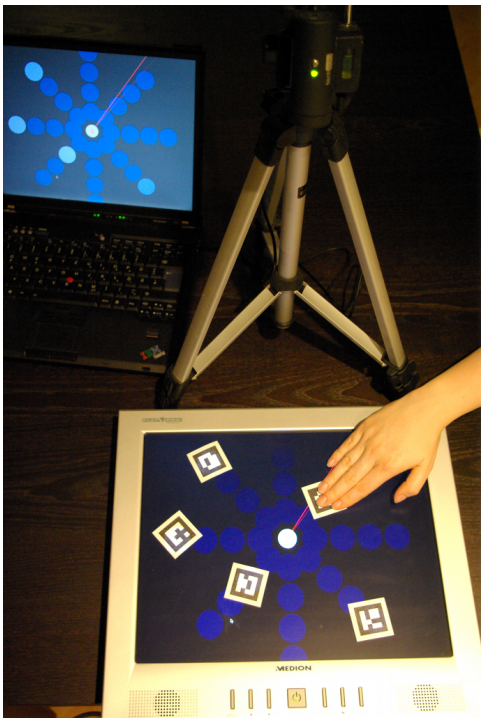


Figure 6.7: Using Beatring with fiducial markers

network connectivity and the ability to run custom software, a rich new area of interaction possibilities results from the combination of these two kinds of interfaces [24].

6.3.1 Detecting Phones on a Tabletop Display

Prior to exploring new kinds of interaction, the presence and location of a mobile device (phone, organiser or similar) on the surface needs to be determined. Some approaches require the device to be equipped either with specialised software [125] or with a fiducial marker [82] beforehand. However, these methods are unsuitable for any kind of casual interaction in which passersby may spontaneously want to couple their mobile device with the surface. Therefore, a method to detect unmodified devices is desirable [23].

The first component of the approach presented here is the previously described shadow tracker. This method is able to reliably detect the presence of almost any kind of opaque object on the surface. However, a decision whether

the object actually is a mobile device is not yet possible. Therefore, a Bluetooth proximity detector is added as second component. As proximity sensing is performed via the *Received Signal Strength Indicator (RSSI)*, this information should be available with low latency. Therefore, a Broadcom USB adapter has been chosen which supports the "inquiry with RSSI" feature that was introduced in version 2.1 of the Bluetooth Core Specification [12]. This enables the adapter to continuously run inquiry scans while at the same time delivering RSSI data on all discoverable devices within radio range. One scan cycle takes about 1 second as opposed to older Bluetooth devices that do not have this feature. These adapters have to issue a time-consuming connection request (up to 11 seconds in the worst case) for every single RSSI measurement. Moreover, this connection requires pairing the mobile device and the adapter for which the user has to enter a password. In contrast, the presented approach is able to function without explicit user interaction.

Of course, this very fact raises some questions regarding security and privacy. One might argue that persons who allow a personal Bluetooth-enabled device to be discoverable know what they are doing. However, most people are unaware of the implications. For example, certain phones are vulnerable to a wide range of Bluetooth-based attacks when they are in discoverable mode [108]. It might therefore be advisable to remind the user through a message on the interactive surface that the Bluetooth transceiver should be turned off after use. Care should also be taken not to compromise the users' privacy by generating a log of detected devices. Although the software does store a list of device addresses, this list is not timestamped and never saved to disk and should therefore be unproblematic regarding privacy.

The tracking software is composed of two threads. The first one continuously collects RSSI data from Bluetooth devices within reception range, while the second one receives and processes notifications from the gesture recognition layer. The gesture specification is designed to differentiate between shadows that are really cast by mobile phones and those cast by other objects, such as the users' hands.

One obvious and easily applied criterion is blob size. There are upper and lower bounds on the surface area which a mobile phone covers, as it is usually roughly pocket-sized. For this setup, these bounds have experimentally been determined to be at 2000 and 10000 square pixels, respectively. These values depend on camera resolution and size of the surface. In this case, a camera with a resolution of 720 x 576 pixels is viewing a surface area of 1.15 x 0.75

m, resulting in a covered area of approximately 2 mm^2 per pixel. While the area range may seem large, it was chosen to account for, e.g., the difference between open and closed clamshell phones. In practice, these values have proven to be sufficient to include every phone which was placed on the surface while filtering out other objects such as a user's arm. The second criterion for the gesture specification is blob motion. Before a phone can be reliably recognised, it should remain motionless on the table for one second, as this is approximately the duration of one inquiry scan cycle. The specification is therefore composed of two features, `BlobDimensions` and `BlobMotion` with appropriate boundaries. When both features match, a "phone" gesture (or rather, event in this case) is delivered by the recognition layer, meaning that the corresponding shadow is a candidate for being from a mobile phone.

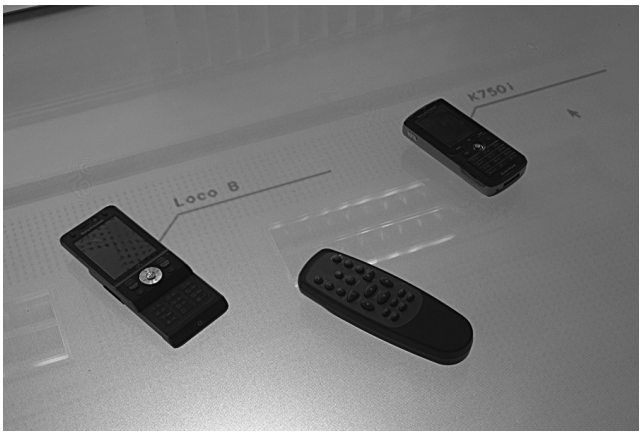


Figure 6.8: Bluetooth name/location assignment

The next step is to correlate these candidates with the proximity data from the Bluetooth thread. The RSSI measurements are usually returned in dBm.[3] The values typically range between -40 dBm for close proximity and -90 dBm at the limits of reception range. Obviously, these values are dependent on the mobile phone as well as the Bluetooth adapter in use. As our adapter is mounted at a distance of approximately 80 cm below the tabletop, a phone lying on the surface generates RSSI values of about -60 dBm. Therefore, a proximity threshold of -65 dBm is used to determine whether a phone is on

[3]Even though this value might not reflect the true received signal power, but rather some internal measure, we can accept this measurement as-is, as we are currently relying on an experimentally determined threshold to decide between the inside- and outside-range cases.

or near the table surface. Although the distance to a mobile phone which is carried in the pocket of a person standing beside the table is about the same, the RSSI values for such phones are significantly lower. This is due to the non-uniform reception pattern of the dipole antenna which is used in almost all Bluetooth dongles. Such an antenna usually exhibits several distinct lobes with high reception sensitivity. In this setup, the antenna is oriented so that the main lobe points straight upwards, thereby favouring phones located on the surface and not those beside the table.

Finally, the list of phone candidates and Bluetooth devices can be compared. In an ideal case, there is one unassigned candidate and one newly detected device in range, which makes the assignment trivial. In this case, the optical characteristics (size and length of major and minor axis) of the candidate are also stored along with the Bluetooth data. This can be used for later identification of devices if ambiguities arise. For example, a candidate blob can appear without suitable Bluetooth devices in range. This can occur when the discoverable mode of a phone has a fixed timeout or Bluetooth is turned off completely. In this case, the blob features are compared with the list of previously recognised devices. The best-fitting match according to a squared-error measure is then used to match the blob with Bluetooth data. Also, if several blobs and Bluetooth devices appear simultaneously, the blob features are compared with previous matches to resolve this ambiguity. Therefore, this method is currently unable to differentiate between several previously unknown objects which are placed on the surface in a short timeframe (< 1 sec.). A view of various devices on the surface which have been annotated with their Bluetooth names where available is shown in figure 6.8.

6.3.2 Joining Casual Games

Due to its collaborative nature, a Sudoku game was again chosen as a test case for interaction between an interactive surface and mobile devices. The Sudoku implementation consequently features a second mode in which users can choose between playing directly on the tabletop interface or on their own mobile device. On one side of the tabletop, a "join area" is located in which users can place their mobile device in order to join the game (see figure 6.9). Instructions for activating the Bluetooth transceiver and placing the device are displayed inside the join area. Once a device has been located as described above, download of the mobile Sudoku client to the device is initiated. Users can now pick up their device in order to authorise the file transfer. After the download has finished, most mobile devices will automatically prompt the user

whether the received software should be executed. After the client has been started, it will immediately connect to the game server and join the running game.

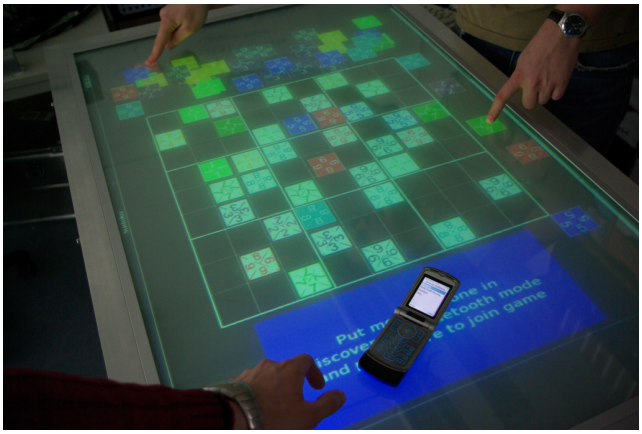


Figure 6.9: Interaction between mobile and tabletop Sudoku

6.3.3 Evaluation

To test our casual connection method in a real-world scenario, we first evaluated the pairing process between mobile devices and the tabletop system through an expert review with four participants. The reviewers were given the task to join the running Sudoku game with their mobile phone by following the instructions displayed on the tabletop system.

All four persons agreed that the main drawback was the still noticeable delay between putting down the device and starting the download. While the optical tracker is able to recognise the device almost immediately, the list of visible Bluetooth devices is often cached internally by the transceiver. This caching process may add a significant delay until the download is initiated. The duration after which cached entries expire is usually not adjustable externally and depends on the transceiver being used. Testing several transceivers with respect to their scan cycle duration and cache expiry time is therefore advisable.

A valuable suggestion by one of the reviewers was to display visual feedback as soon as the optical tracking detects a potential mobile device. To provide

such feedback, a pulsing coloured circle will now appear below the device, thereby informing the user that the joining process has been started.

6.4 Collaborative Applications

While the previously mentioned demos and games are well suited to illustrate the potential of novel interaction concepts, their practical potential is limited. We will therefore now consider some applications which are geared towards real-world use.

6.4.1 Virtual Chemistry

Another natural application for multi-touch is interaction with 3D content. The user is able to employ more intuitive metaphors for grasping, moving and rotating a 3D object on the screen than with a mouse-based method such as ArcBall [104]. While these metaphors are still far from perfect as the user can only touch a flat surface, the metaphors for swiping along an axis to rotate the object in that direction or for grabbing it with two or more fingers and rotating it along the view direction seem natural enough.

In one application [71], a custom molecule viewer is used for displaying a 3D view of chemical reactions. This viewer is based on Ubitrack and receives 3D rotation and position information from arbitrary tracking devices in order to adjust the viewpoint. To provide multi-touch input to this application, a second type of Ubitrack adapter is currently being developed which receives movement and rotation gestures from an arbitrary multi-touch input device. These gestures are then translated into a persistent 6D pose which is passed on to the Ubitrack library and delivered to the molecule viewer.

6.4.2 Interactive Whiteboard

One of the most simple, yet practical uses of an interactive surface is to convert it into a virtual whiteboard. Several users can collaborate to sketch diagrams, discuss ideas and get a digital copy of their results immediately afterwards.

One problem which quickly comes to mind when considering several users is that of colour selection. Usually, such quick sketches are easier to understand when drawn in different, easily discernible colours. On a real whiteboard, this task is solved by having several pens available from which every user can choose. Therefore, the virtual whiteboard needs a means to differentiate

between users, as otherwise only a single global colour selection would be possible. While this may be sufficient in some cases, it is nevertheless better not to restrict the user arbitrarily.

Obviously, this differentiation between users should be supported by the hardware. Applicable devices would be, e.g., the Wiimote, the DiamondTouch surface and TISCH in conjunction with the shadow tracker. While the latter is, strictly speaking, not able to differentiate between users, but between hands, this does not pose a problem here. In order to select a colour for a specific user, two variants are possible. The first one is to display one or more persistent colour palettes at a fixed location on the screen, preferably near the edges. Every user can then independently assign a colour to the own hand and subsequently paint with this colour by touching the screen. The second variant is to assign a specific gesture to display an individual colour selector at the hand's position. Use of this method on the TISCH is illustrated in figure 6.10 with a simple five-finger gesture.

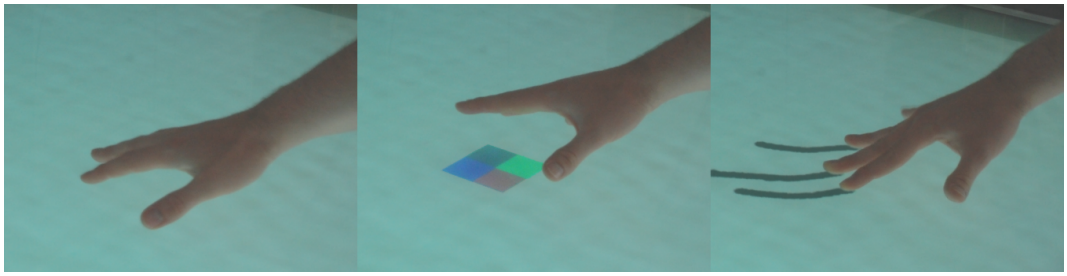
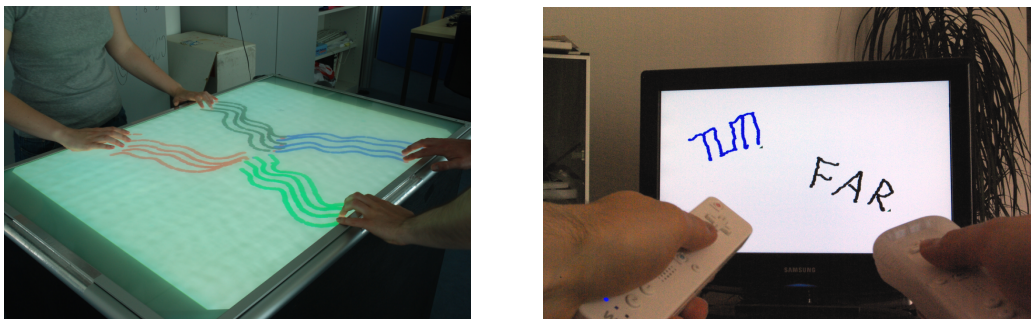


Figure 6.10: Colour selection process

As mentioned earlier, the Wiimote does also allow to differentiate between several users simply by giving every user a separate device. Therefore, this whiteboard application can also be collaboratively used with several Wiimotes. This case, as well as usage on the TISCH, are shown in figure 6.11.

6.4.3 Virtual Patient

Personnel from emergency response services such as ambulance crews, fire brigade etc. often need to be trained in the process of so-called *triage*. Should an unforeseen event such as a natural disaster, large-scale accident or similar occur, these first responders arriving at the scene are required to assess the severity of the various injuries as quickly as possible. This is necessary to ensure the best utilisation of strained resources such as ambulances, paramedics,



(a) on TISCH (b) with Wiimotes

Figure 6.11: Using the virtual whiteboard

emergency physicians and similar. This assessment process should be completed in under 90 seconds per casualty and is usually guided by a flowchart. As the persons conducting the triage may be under considerable physical and mental stress, it is important to train this process repeatedly to achieve the best possible performance, even under adverse conditions. However, these trainings are a highly complex and costly process, as they usually involve a number of mimes who have to be made up with realistic-looking injuries and who have to be trained themselves in reacting appropriately to the paramedics.

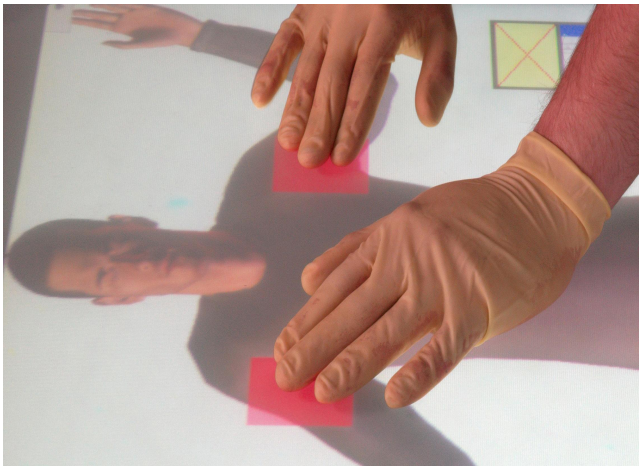


Figure 6.12: Interaction with the virtual patient (image from [86])

To provide a less involved alternative to this kind of training, a virtual pa-

tient application has been developed [86]. This application randomly displays one of several virtual injured persons. As the virtual patient should be as close to life-size as possible, this application is designed to run on the TISCH tabletop interface only. Paramedics can interact naturally with the virtual patient, e.g. by touching a wrist to check the patient's pulse which is then displayed in a small window above the wrist. This feature also highlights an improvement with respect to existing triage training software. While these existing programs are used with a common desktop computer, the multi-touch interface requires users to stand and occasionally use both hands for a task, such as propping the patient up (see figure 6.12). It is therefore closer to reality than a pure desktop-based interface. For example, should the paramedic be holding a clipboard or similar device, it has to be put down prior to certain actions.

In an evaluation performed by Nestler et al. [86] with paramedics from the fire department in Munich, it was shown that the accuracy of the training results when using the virtual patient application were comparable to those obtained with mimes. Due to the significant reduction in complexity regarding the training exercises, the virtual patient seems to provide a viable alternative to common training procedures.

Summary

In this chapter, various games and applications which have been built based on the libTISCH framework have been reviewed. Where applicable, these programs have been tested with different input hardware devices or been evaluated in user studies or expert reviews.

Chapter 7

Conclusion

In this chapter, the architecture and concepts introduced in this thesis shall be discussed. Also, a look at future extensions of the presented implementation shall be taken.

7.1 Discussion

The contributions offered by this thesis are threefold. First, an architecture for describing a wide variety of interactive systems, especially those based upon novel input devices, has been presented. Second, as a core part of this architecture, a generic and extensible protocol for the specification of gestures was developed. Finally, a reference implementation of this architecture, libTISCH, has been presented and tested in a range of scenarios. Each of these contributions shall now be discussed shortly.

Architecture Design

The overall experience currently supports the design of the presented interaction architecture. This architecture has proved to be generic enough to subsume a wide variety of applications, gestures and hardware devices into one coherent system. Extensibility is provided through a number of clearly documented interfaces, making it easy to add new widgets or support for new input hardware.

Where applicable, this architecture follows established design practices from existing window-management systems and toolkits. Other solutions have envisioned completely different approaches to novel interaction devices such as using a physics simulation instead. However, the integration of existing

concepts such as events and widgets will provide an easy migration path for developers which are used to these established methods.

Using the Gesture Description Protocol

A point of critique made by several developers using the library pertained to the gesture description protocol. In some cases, the mapping from real-world gestures to abstract GDP entities is not obvious due to the high number of configurable parameters offered by the specification. While this complexity is unavoidable to provide the high degree of flexibility required by some applications, it may be confusing for developers.

Partly, this problem can be solved by providing appropriate default gestures for common use cases and assigning them to suitable widgets. Of course, these default gestures should be usable for a wide range of hardware devices. Therefore, it may be necessary to create several sets of default gestures for conceptually different sensors and load them according to the hardware which is currently in use.

libTISCH Performance

An important aspect of any library aimed at developing interactive software is performance. The layered design of the underlying architecture does not mandate a specific means of communication. To achieve best platform and language independence, UDP was used in the reference implementation. Tests have shown that each UDP link can be expected to add an average of 1-2 ms of latency to the overall system delay when running on the same host. This results in an expected total latency of not more than 6 ms that is caused by the use of three UDP links between the layers.[1]

However, should even this small increase be unacceptable due to other factors, two or more separate layers could be integrated into one single, multi-threaded application and communicate internally. Another option would be to link the layers via shared memory, Unix pipes or other, local-only means of inter-process communication.

[1]Note that this does not take hardware delays into account which inevitably occur when linking separate hosts.

7.2 Outlook & Future Work

While this thesis has presented an important step towards a more generic approach to novel modes of interaction, there are still aspects which have not been explored in-depth.

Cross-Device Gesture Standardization

Some basic gestures such as "move" or "rotate" will likely be part of a large number of applications. However, depending on the input device, the movements used to trigger these events may be quite different. To avoid burdening the end-application developer with specifying gestures over and over again, a basic set of gestures should be available regardless of the hardware device used. The responsibility for specifying those gestures then shifts to the person who implements the hardware abstraction layer for a specific input device.

Consistency of Interaction Metaphors

However, this approach immediately highlights another problem. How can the end user know which movements actually are needed to trigger a specific kind of interaction? If these movements are dependent on the specific hardware device in use, either a learning phase specific to each device is unavoidable, or the interface has to be designed in such a way that the user is subtly guided to the correct movements for each device. Graphical affordances may be used to this end. Another possibility is to design the basic gestures in such a way that they appear identical or at least similar to the user, despite the different underlying implementation.

Analysis of Application Areas

When viewed from a less technical perspective, it becomes apparent that one big drawback of multi-touch and other novel kinds of interfaces in general is the still present lack of applications. A wider analysis of other scientific and industrial disciplines, especially those not directly related to computer science, should be conducted in order to identify areas where such user interfaces can provide a benefit over existing standard interfaces.

Extension of Widget Library

When looking at standard desktop toolkits like Swing, the developer is offered a huge variety of widgets. In comparison, the rudimentary widget set in libTISCH offers only the most basic functionality and relies on the developers to derive more specialised widgets on their own. The previously mentioned analysis of new application areas may give insights into which other types of widgets should be included in a standard set, thereby easing the deployment of such user interfaces in other disciplines.

Non-Graphical User Feedback

Another topic which has not yet been considered is that of non-graphical feedback such as sound, vibration etc. The Wiimote, for example, offers vibration, sound generation and four bright LEDs which can deliver additional signals to the user. Future touchscreens may deliver localised vibration or deformation for haptic feedback. Support for these features would necessitate an additional feedback channel from the gesture recognition layer to the hardware abstraction layer as well as an extension of the gesture formalism to described the type of feedback to be delivered.

Multi-Sensor Support

Although libTISCH supports a variety of sensors, merging their data at runtime is currently only supported for some special cases such as the combined shadow/FTIR tracker. A generic sensor fusion approach is probably beyond the scope of libTISCH. However, such functionality is already offered by the Ubitrack library. The existing Ubitrack adapter offers an easy way to explore complex multi-sensor setups such as, e.g., combining fiducial markers with a touchscreen and hand-held interaction devices.

3D Interaction

Finally, an extension which may not be apparent at first sight is the support of fully three-dimensional interaction, for example in virtual or augmented reality environments. While the implementation presented in the context of this thesis has been focused on two-dimensional interfaces, the architecture as well as the protocol specifications do not pose any restrictions in this regard. For example, a region in 3D could be described by a sequence of triangles, thereby composing a polyhedron. While free-hand interaction using the Wiimote has already been

implemented within this thesis, this is still inherently a two-dimensional mode, as the data is projected into the screen plane prior to processing. However, all protocols fully support the transport of three-dimensional data. While most of the existing features would have to be extended or replaced in order to support 3D interaction, the basic gesture matching algorithm would also work in this context.

7.3 Summary

In this thesis, a new approach to interactive systems has been presented. An abstract architecture has been developed which can be used to formally describe the various components of a wide variety of user interfaces, from the hardware up to the graphical representation of the UI. Special emphasis has been given to a formal specification of gestures. In principle, this formalism adheres to widely used design methods such as widgets and events, but extends them where necessary. An important extension is the separation of events into *input events* which describe motion and *gesture events* which describe meaning.

A reference implementation of this architecture, libTISCH, has been developed. libTISCH was used to create several different kinds of applications which have been run on various sensor devices. While a significant amount of work remains to be done, this architecture and library are an important step of bringing novel kinds of interfaces such as multi-touch systems closer to the mainstream.

Appendix A

In this chapter, some of the more technical details of certain important implementation aspects shall be described. Moreover, a class reference for libTISCH is given in order to help developers using this library.

A.1 libTISCH Configuration Files

A.1.1 Calibration File

The calibration file is stored under the name `.tisch.calib`, either in the user's home directory or in /tmp. It contains two major calibration elements: the perspective transformation matrix from sensor to screen coordinates and the radial undistortion parameters. The format of the file is as follows:

```
m1 m2 m3
m4 m5 m6
m7 m8 m9
c0 c1 c2 c3
tx ty tz
sy sy sz
```

The parameters `m1 - m9` describe the homography which translates sensor coordinates into screen coordinates. The bottom row `m7 m8 m9` should be equal to $(0, 0, 1)$ for a perfect calibration. Small deviations may occur due to numerical inaccuracies of the singular value decomposition which is used to calculate the mapping. For example, in one real-world calibration, this last row of the matrix has the values $(-0.000182321, 0.0000219296, 1.0)$.

The parameters `c0 - c3, tx ty tz, sx sy sz` describe the radial undistortion. `c0 - c3` are the coefficients for the undistortion equation mentioned

in section 5.3.1. This equation takes the general form $r = ax + bx^3$ for almost all kinds of lenses. Therefore, the coefficients `c0, c2` are usually zero. As this function can only be applied to normalised pixel coordinates in the range $[-1, 1]$, the image coordinates have to be mapped to this range before undistortion is applied. This can be achieved through the translation `tx ty tz` and scale `sx sy sz` parameters. Note that the centre of the normalised coordinates does not necessarily map to the centre of the image. Instead, it should map to the centre of distortion which may be located off-axis. After undistortion, the mapping is applied in reverse to retrieve the final, undistorted image pixel coordinates.

A.1.2 `touchd` Parameter File

As various camera and tracking parameters may require fine-tuning depending on the exact hardware used, the `touchd` blob tracking daemon can also be configured through a plain-text file. This file is named `.tisch.touchd` and is stored in the same location as the calibration file.

This file consists of one or more blocks, each of which describes one tracking modality. Several tracking modalities may be specified which are cycled through after every incoming camera image. One block has the following format:

```
name
threshold invert bgfactor
minsize maxsize id factor radius scanpeak colour1 colour2
width height fps camtype gain exposure brightness flash
```

The `name` parameter is a string which specifies the tracking mode described in this block, e.g. "`finger`". This is also the identifier which is later used for transmitting blobs detected by this tracking mode. All other parameters are numerical unless noted otherwise.

As described in section 5.2.3, the base threshold is calculated as the absolute difference between the average intensities of background image and foreground image. The `threshold` value is added to this difference to obtain the final value. The `invert` parameter selects whether the background image should be subtracted from the foreground image or vice versa. In the first case (`invert == 0`), objects which are brighter than the background will be detected, e.g. for FTIR tracking. In the second case (`invert == 1`), darker objects will be detected, e.g. for shadow tracking. The `bgfactor` parameter determines

which percentage of the foreground image intensity should be integrated into the background. Currently, only fractions of powers of two are supported to avoid generic floating-point calculations. A fraction like $\frac{1}{512}$ can be realised through fast integer shift operations.

The next set of parameters controls the blob tracking algorithm. `minsize` and `maxsize` specify the minimum and maximum number of pixels which a blob must contain. Otherwise, it is discarded. `id` determines the first number which will be assigned as a new identifier. This number will be continually increased while the software is running. `radius` specifies the radius in pixels around the estimated new blob location in which a corresponding blob is searched. `scanpeak` and `factor` determine whether a peak should be determined for each blob (`scanpeak > 0`) and by which factor the axes of the equivalent ellipse should by scaled to obtain the search axes. The `colour1` and `colour2` parameters are only relevant for the visualisation and specify the colour in which the blob centroid and the motion vector shall be displayed. Both values are RGB triples.

The final set of parameters in each block determines the camera settings. `width`, `height` and `fps` specify the image size and frame rate. `camtype` determines which camera driver should be used (1: Video4Linux, 2: Firewire IIDC, 3: DirectShow). The following three settings, `gain`, `exposure` and `brightness` set these three camera parameters. Note that the range and values may differ widely between camera models. The final parameter, `flash`, is only valid when a flash controller such as the ATtiny13 mentioned below is connected which supports different modes of operation such as alternating light sources. The `flash` parameter is passed to the `FlashControl` class which configures the flash controller accordingly. The exact meaning of this parameter depends on the firmware used in the control circuit.

A.2 Firmware for the ATtiny13 LED Controller

The LED pulsing method described in section 4.1.1 benefits from using a microcontroller to activate the LEDs. Such a device offers higher flexibility and reliability than possible when, e.g., controlling the LEDs directly through the camera. For example, strict timing requirements such as observing the necessary cool-down delays between two pulses would otherwise not be possible.

To this end, a custom firmware for the ATtiny13 microcontroller was written. This firmware contains a primitive state machine with 7 states:

0 wait for trigger signal, LEDs off

1 check if odd or even cycle

2/3 turn corresponding LED group on

4 flash active: start timer, wait $300\mu s$ (pulse duration)

5 timer expired, turn all LEDs off

6 cooldown: start timer, wait $19.7ms$ (cooldown phase)

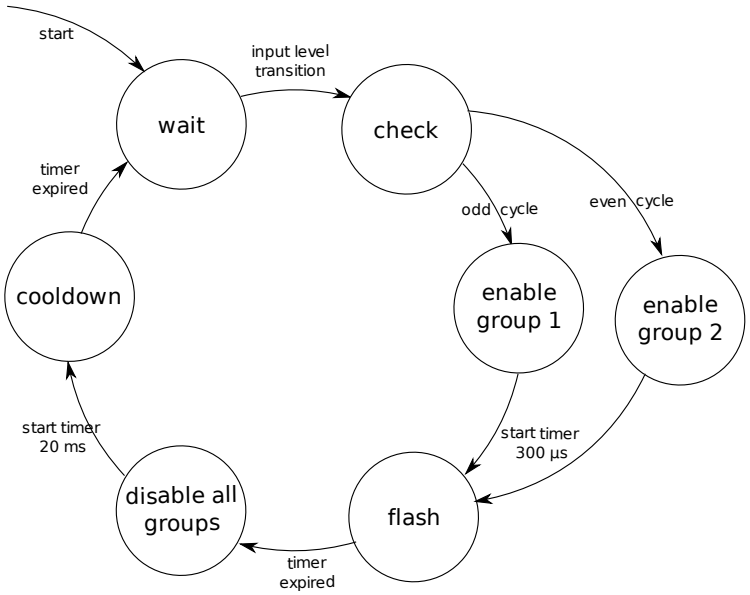


Figure A.1: LED control state machine

After state 6 has ended, the state machine returns to the default state 0. The state graph therefore is cyclic. State transitions can be triggered by two different interrupts, either by a logical transition from 1 to 0 on the input pin or by an one-shot timer interrupt when the timer has been started by the previous transition. The presented state machine is designed to control two LED groups, but can be easily modified to support more. Note, however, that for n LED groups, the effective framerate will be reduced by a factor of $\frac{1}{n}$.

A.3 A Minimal X3D Renderer

This concept of a lightweight rendering engine for the X3D standard has first been developed for the Ubitrack library. Due to its small size, it was also included in libTISCH to provide a fast, easily accessible method to render 3D content.

The core idea behind this concept relies on the fact that X3D is an *Extended Markup Language (XML)* dialect. The inherent tree structure of XML is used to describe a scene graph, which is often also a tree or at least a directed acyclic graph. Most XML parsing libraries support the *Simple API for XML (SAX)* programming model, which triggers event callbacks upon entry and exit of a node during parsing. When viewing the XML tree as a scene graph, then this sequence of events is nearly identical to that which occurs during scene graph traversal. Therefore, a separate scene graph data structure and traversal code can be replaced by the XML representation and parser callbacks which call appropriate OpenGL commands depending on the current node type.

Of course, this approach requires an XML parsing library in the first place. Libraries which fully support the entire XML standard can reach significant code size on their own, thereby rendering the entire concept largely useless. Fortunately, non-validating libraries can still be kept quite small. One example which this renderer is based on is TinyXML [113]. It has the additional benefit of using the C++ STL as its only dependency. By heavily using many STL data structures, the library can remain very small (about 90k of compiled code size).

However, while this concept is able to significantly reduce source code size, an unoptimised implementation suffers from bad performance. The reason is that the XML document tree and particularly all node attributes like colour information, vertex coordinates etc. have to be re-parsed for every rendering pass. Fortunately, OpenGL offers the ability to compile so-called *display lists* at runtime, which are sequences of commands that are stored in the GL server and can later be called through a simple identifier. This feature provides significant optimisation potential: each node is parsed regularly when it is visited for the first time, generating the appropriate OpenGL commands. At the same time, a display list is compiled and stored in a hash table with the node's identifier as index. In all subsequent rendering passes, parsing can be aborted for every node which has an entry in the hash table. Instead, the stored display list can be called which delivers the same commands.

A.4 MPX Compatibility Patch for FreeGLUT

When implementing cross-platform 3D graphics, OpenGL is by far the most common choice. As OpenGL itself does not deal with context allocation, event handling and similar issues, the GL Utility Toolkit (GLUT) library is usually employed for these tasks. Currently, the most widely used GLUT implementation is FreeGLUT [4].

The GLUT programming model allows to register callbacks for various events such as key presses, mouse movement etc. As this quasi-standard is over ten years old, the signature of these callbacks is naturally geared towards a single pointer. All mouse-related callbacks simply deliver a single x-y coordinate pair. When considering extensions such as MPX, this is insufficient. Should, for example, two pointers move at the same time, the application will likely receive motion events with coordinate sets that alternate between the two different positions.

To deal with this limitation, a patch to FreeGLUT was developed which adds multi-pointer aware versions of the four most basic mouse callbacks which deal with entry/exit, motion, button and passive motion events. The functions for setting these callbacks are:

```
void glutXExtensionEntryFunc( void (*callback)( int, int ) )
void glutXExtensionButtonFunc( void (*callback)( int, int, int, int, int ) )
void glutXExtensionMotionFunc( void (*callback)( int, int, int ) )
void glutXExtensionPassiveFunc( void (*callback)( int, int, int ) )
```

When the FreeGLUT library has been built with support for these features, then the macro `GLUT_HAS_MPX` will be defined to indicate their presence. The signature of each the callback functions themselves now has an additional `int` parameter. When the callback is triggered, this parameter denotes the pointer from which the event originated. By maintaining a hash table with the pointer IDs as keys, the event can thus easily be assigned to the correct set of coordinates.

It is currently unclear whether Windows 7 will offer similar functionality. If this is the case, then this API could be extended to provide access to multi-pointer input data under Windows, too.

A.5 GLUT-Compatible Wrapper for the iPhone

By using OpenGL, even such exotic platforms as the iPhone can be integrated into the framework with little effort. The programming environment which

Apple provides for the iPhone and iPod touch does offer an graphic library based on OpenGL ES (embedded systems). Unfortunately, an equivalent for GLUT does not exist. Functions such as context creation, callback handling etc. have to be performed through the ObjectiveC-based UIKit library.

However, as ObjectiveC is binary-compatible to C, it is possible to create a wrapper module which offers the most important GLUT functions in C, but which is internally implemented in ObjectiveC to access the UIKit interface. Additionally, this offers the opportunity to re-use the extension callbacks which were previously introduced for MPX. Semantically, the touch events from the iPhone do not differ from the MPX extension pointer events, with the sole exception that the iPhone hardware is not able to deliver "hover" events which trigger the passive motion callback from GLUT. This callback is therefore non-functional. All other extended callbacks work as expected, with a finger ID being delivered instead of a pointer ID.

While this wrapper cannot provide the entire functionality of GLUT, it offers the basic features which are needed to get an OpenGL context and receive input events. One important feature which should be supported but has not yet been implemented is rendering of geometric primitives such as cubes, spheres or text strings. Of course, some of the advanced functions of GLUT such as support for multiple windows are not usable on the iPhone. However, as libTISCH is also geared towards a single top-level window, this does not pose a problem.

A.6 libTISCH Class Reference

libTISCH is divided into five core libraries. Each of them shall be described briefly, followed by a list of the contained classes and their methods. This section is designed to help developers build applications based on libTISCH. Unless noted otherwise, all classes are available on all three supported operating systems (Linux, MacOS X and Windows).

Note that all classes which need to be transferred between layers at some point, including, e.g. `Vector` or `Region`, define stream input/output operators (`<<` and `>>`). This allows easy serialisation and unserialisation through any `std::iostream` object.

A.6.1 libtools

This library provides a collection of various low-end helper classes and templates. Some of these have been inspired by classes from the Boost library. While simply integrating Boost would have been a possibility, prior experience has shown that it is still a significant source of compile-time errors due to slight changes between library releases. Therefore, the classes `Functor`, `SmartPtr`, `Thread` and `Vector` have been re-written from scratch.

`Vector`

Description This class provides a simple encapsulation of a 3-vector along with a number of common vector operations mapped to C++ operators. The class is templated with the component type, allowing for various kinds of vectors. The default component type is `double`.

Definition

```
template <typename Type> class _Vector
typedef _Vector<double> Vector
```

Constructor `_Vector( Type x = 0, Type y = 0, Type z = 0 )`

Methods

The arithmetic operators `+ - *` can be used as expected. `vec1 * vec2` results in the dot product, while `vec * number` results in a scaled vector. The operation `vec1 & vec2` results in the cross product.
The member variables `x,y,z` provide direct access to the components.
`double length()` returns the length of the vector.
`void normalize()` scales the vector to a length of 1.
`void rotate( double angle )` rotates the vector around the specified angle in the x-y plane.

`BasicBlob`

Description This class describes a generic input object through properties such as size, position, orientation, identifier etc. All member variables are public. The name results from the computer vision term for connected components which are often used as input objects. However, entities such as tangible objects can also be described with `BasicBlob`.

Definition `class BasicBlob`

Constructor `BasicBlob()`

Member Variables

`int size` Size of the object in arbitrary units, e.g. pixels for vision-based blobs.

`int id` Unique identifier for the object.

`int pid` Unique identifier of the parent object when available, 0 otherwise.

`Vector pos` Position of the object.

`Vector speed` Current (estimated) movement vector.

`Vector peak` Peak (e.g. fingertip) of the object when available.

`Vector axis1,axis2` Major and minor axis of the equivalent ellipse.

`UDPSocket`

Description This class is an object-oriented wrapper around the widely used *Sockets* interface for network programming. `UDPSocket` is derived from the STL class `std::iostream`. Therefore, the default stream input and output operators `>> <<` can be used to transmit and receive objects as UDP packets.

Definition `class UDPSocket: public std::iostream`

Constructor `UDPSocket( const char* address, int port, struct timeval* to = 0 )` The `address` parameter specifies the local address on which this socket should listen. This can be a hostname, an numerical address or a constant such as `INADDR_ANY` for all available interfaces. The `port` parameter specifies the UDP port which the socket should listen at. When this parameter is set to 0, an arbitrary unused port will be chosen by the operating system. The optional timeout parameter `to` can point to a `struct timeval` object that specifies how long the socket should wait for new data to arrive before aborting the operation. When this parameter is unset, the socket will wait indefinitely.

Methods

`void target( const char* address, int port )` specifies the address and port where packets from the socket should be sent to. The address and port specification is the same as in the constructor.

`in_addr_t source( int* port )` returns the address and port where the last received packet came from. The address is returned as a 32-bit integer.

`void flush()` discards all buffer contents and clears all error flags, thereby resetting the socket state.

`Thread`

Description This class provides a simple object-oriented wrapper around an execution thread. To create a new thread, a subclass of `Thread` has to be created which implements its own `run()` method. For convenience, the class also provides access to an internal mutex object for thread synchronisation.

Definition `class Thread`

Constructor `Thread();`

Methods

`virtual void* run()` has to be overwritten in a derived class and provides the inner thread loop.

`void start()` executes the `run()` method in a new thread.

`void lock()` acquires a lock of the thread's internal mutex. If another thread already has locked the mutex, then the call will block until the mutex is released again.

`void release()` releases the internal mutex, allowing other threads to lock it.

`SmartPtr`

Description As libTISCH relies heavily on the STL, there are occasions where pointers have to be stored in STL containers. When using normal pointers, this can easily lead to memory leaks at runtime. It is therefore advisable to use a reference-counting pointer in these cases. The `SmartPtr` class provides such a pointer which offers the usual pointer semantics. Additionally, it will automatically delete the object which it points to when all references have been deleted.

Definition `template <class X> class SmartPtr`

Constructor `SmartPtr( X* p = 0 )`

Methods

All pointer operators `* = ->` can be used as with normal pointers.

`X* get()` will explicitly return the internally stored pointer.

`Functor`

Description A functor is a "canned function call". A reference to a normal function is stored along with a list of parameters. At a later point in time,

these parameters are passed to the function and the call is executed by the functor object.

Definition `template <typename Result> class Functor`

Constructor `Functor` is a special case, as it should in general not be created through its constructor. Rather, the polymorphic function `bind` can be used to create a functor, e.g. as follows: `Functor<void> func = bind( glColor3f, 1.0, 1.0, 0.0 );`

Methods
`operator() ()` - calls the stored function with the stored parameters, e.g. `func();`

A.6.2 `libsimplecv`

This library provides a set of basic image management and computer vision functions. While all of these functions could have been provided by an external library such as OpenCV, this approach would have had two significant drawbacks. Besides adding an additional dependency along with unavoidable problems such as version conflicts, the open-source version of OpenCV is written in pure C. While an accelerated implementation is available, this version is not available under a free license, but has to be paid for. Therefore, libsimplecv offers accelerated versions in MMX assembler for the most performance-critical functions. The helper classes `ColorLUT`, `YUYVImage` and `YUV420Image` shall not be described in detail, as they are only used internally by the image sources to convert raw camera image data into the common luminance or RGB colour spaces.

`Image`

Description The core element of libsimplecv is the abstract `Image` class. Three concrete implementations exist for 8-bit greylevel images (`IntensityImage`), 16-bit greylevel images (`ShortImage`) and 24-bit RGB colour images (`RGBImage`).

Definition `class Image`

Constructor `Image( int width, int height, double bpp )` creates a new image with dimensions `width * height` and `bpp` bytes per pixel (fractions are possible).

Methods

In all following methods, target images should always have the same size as the current image itself unless noted otherwise.

`unsigned char getPixel( int x, int y, int channel = 0 )` retrieves a single pixel value.
`void setPixel( int x, int y, unsigned char value, int channel = 0 )` sets a single pixel value.
`void clear( int value = 0 )` sets all bytes in the image to the same value.

The following functions are only available for ***`RGBImage`****:*
`void getChannel( int channel, IntensityImage& target )` extracts a single colour channel into `target`.
`void getIntensity( IntensityImage& target )` converts the RGB data into intensity data in `target`.
`void combine( const IntensityImage& red, const IntensityImage& green, const IntensityImage& blue )` creates an RGB image from three separate channel intensity images.

The following functions are only available for ***`ShortImage`****:*
`void update( const IntensityImage& img, const IntensityImage& mask )` calculates the new pixel intensity values as a weighted average between the current intensity and the intensity in `img`. `mask` can be used to exclude image regions from being updated.
`void subtract( const IntensityImage& source, IntensityImage& target, int invert )` subtracts the local intensity values from the source image. Intensity values from source are temporarily extended to 16 bit. The result is written into `target`. When `invert` is true, the source image is subtracted from the local image instead.
`void convert( IntensityImage& img )` converts the local 16-bit intensity values into 8-bit values and stores them in `img`.

The following functions are only available for ***`IntensityImage`****:*
`void sobel( IntensityImage& target )`

`void sobel()` apply a horizontal and a vertical Sobel operator to the image, thereby highlighting edges.
`void adaptive_threshold( int radius, int bias, IntensityImage& target )` apply an adaptive thresholding operator to the image.
`int threshold( unsigned char value, IntensityImage& target )`
`int threshold( unsigned char value )` apply a standard global threshold to the image.
`void invert( IntensityImage& target )`
`void invert( )` subtract the image values from 255 and store the result.
`int histogram( int hg[] )` calculates an intensity histogram of the image.
`int intensity()` calculates the average image intensity.
`long long int integrate( Point start, Vector& centroid, Vector& axis1, Vector& axis2, unsigned char oldcol = 255, unsigned char newcol = 0 )` performs a connected-component analysis. The seed point is located at `start` while `oldcol` and `newcol` specify the colour which should be searched for and the replacement colour. Centroid, major and minor axis are stored in the respective parameters. The return value is the blob size in pixels.
`void undistort( Vector scale, Vector delta, double coeff[4], IntensityImage& target )` performs a generic radial undistortion on the image, based on the coefficient array.
`void despeckle( IntensityImage& target, unsigned char threshold = 8 )` applies either an erosion or dilatation operator to the image. A pixel is set to full intensity if at least `threshold` neighbour pixels also have full intensity.
`IntensityImage& operator-=( const IntensityImage& arg )`
`void subtract( const IntensityImage& i1, const IntensityImage& i2 )` subtracts two images from each other. Negative result values are clamped to 0.

`ImageSource`

Description `ImageSource` is an abstract class which is used to encapsulate various kinds of camera interfaces in its derived classes `DirectShowImageSource`, `V4LImageSource` and `DCImageSource`.

Definition `class ImageSource`

Constructor

`V4LImageSource( const std::string& path, int width, int height, int fps = 30, int debug = 0 )`
`DCImageSource( int width, int height, int fps = 0, int num = 0, int verbose = 0 )`
`DirectShowImageSource( int width, int height, const char* videodev = "", int verbose = 0 )`
These constructors create concrete implementations of image sources. The parameters `width`, `height`, `fps` and `verbose` all have the same meaning. The selection of a specific camera is done either through a device path for Video4Linux, a device number for Firewire/IIDC or a device name for DirectShow.

Methods

`int acquire()` instructs the camera to buffer the most recent image.
`void release()` releases the previously buffered image. A call to `acquire` should always be followed by a call to `release`.
`void start()` starts the camera's image acquisition process.
`void stop ()` stops the camera.
`void getImage( IntensityImage& target )` loads the buffered image into an intensity image, possibly converting it in the process.
`void getImage( RGBImage& target )` loads the buffered image into an RGB image, possibly converting it in the process.
`void setFPS( int fps )`
`void setGain( int gain )`
`void setExposure( int exp )`
`void setShutter( int speed )`
`void setBrightness( int bright )` set various camera parameters. These may not be implemented in all image sources. Particularly the DirectShow implementation is currently lacking support for camera parameters.
`void printInfo( int feature = 0 )` displays either information about the selected feature or about the camera state in general.

`Line`

Description This class offers an abstract representation of the Bresenham line algorithm. It can be used to perform an arbitrary operation for all pixels in an image on a line between two points.

Definition `class Line`

Constructor `Line( Point p1, Point p2 )` creates a line between two points with integer coordinates.

Methods

`void foreach( int x, int y )` This pure virtual method will be called once for every point on the line. Derived classes need to overload this function with their own implementation.

`void follow()` executes `foreach` for all points.

`Circle`

Description Similar to `Line`, this class represents the Bresenham circle algorithm and allows arbitrary operations on image pixels which are situated on a circle.

Definition `class Circle`

Constructor `Circle( Point c, int r )` creates a circle with centre point `c` and radius `r`.

Methods

`void foreach( int x, int y )` This pure virtual method will be called once for every point on the circle. Derived classes need to overload this function with their own implementation.

`void follow()` executes `foreach` for all points.

A.6.3 `libsimplegl`

The `libsimplegl` library provides a number of convenience classes in order to access OpenGL graphics functions independently from the operating system. Although lots of this functionality is already provided by GLUT, there are some subtle differences regarding available OpenGL features which are hidden through this library. Additionally, it provides the lightweight X3D rendering engine described in section A.3.

`GLUTWrapper`

Description `GLUTWrapper` is not a class, but rather a collection of defines and helper routines which hide most remaining differences between various implementations of GLUT. It also provides some helper functions which subsume various repetitive tasks.

Methods

`template <class T> void glutPaintArrays( int num, T* vertices, T* texcoord = 0, T* normals = 0, GLenum mode = GL_TRIANGLE_STRIP, GLenum where = GL_TEXTURE0 )` This function draws one single vertex array with `num` triples of type `T`. Optionally, a 2D texture coordinate array and a 3D normal array of the same type can also be specified.

`GLUTWindow`

Description This class provides an object-oriented wrapper around a top-level window with associated OpenGL context. GLUT event callbacks are translated into virtual method calls, thereby allowing easy access in derived classes.

Definition `class GLUTWindow`

Constructor `GLUTWindow( int w, int h, const std::string& title )` The `w/h` parameters determine the dimensions of the window, whereas the `title` parameter specifies a titlebar text.

Methods

`void clear( float red = 0.0, float green = 0.0, float blue = 0.0, float alpha = 0.0 )` sets the clear colour and empties the colour and depth buffers.
`void print( const std::string& text, int x, int y )` displays a string at the specified position using bitmap fonts.
`void mode2D()` sets an orthogonal projection matrix and various other parameters for using OpenGL as a 2D rendering engine with blending.
`void show( const RGBImage& img, int x, int y )`
`void show( const ShortImage& img, int x, int y )`
`void show( const IntensityImage& img, int x, int y )` These functions display a raw, unscaled image at the specified window coordinates.
`void swap()` exchanges the front and back buffers and should usually be called at the end of the `display` callback.
`void run()` executes the GLUT main loop and will not return. It should therefore be called at the end of the `main()` routine.
`void idle()`
`void display()`
`void reshape( int w, int h )`

`void keyboard( int key, int x, int y )` are the window-global virtual callback functions which are identical to the GLUT callbacks with the same names.
`void mouse( int num, int button, int state, int x, int y )`
`void passive( int num, int x, int y )`
`void motion( int num, int x, int y )`
`void entry( int num, int state )` are the pointer specific virtual callbacks. These also correspond to their GLUT-specific counterparts with the exception that the all have an additional first parameter `num` which contains the ID of the pointer that triggered the callback.

`PicoPNG`

Description This class contains a minimal *Portable Network Graphics (PNG)* parser which can be used to read any RGBA PNG image file into memory. It is based on the picoPNG parser by Lode Vandevenne.

Definition `class PNGImage`

Constructor `PNGImage( const std::string& file)` opens the specified filename as PNG and allocates a suitable memory block for the raw image data.

Methods
`unsigned char* data()` returns a pointer to the data block.
`int width(), int height()` return the dimensions of the image.

`Texture`

Description This class provides a generic wrapper for an OpenGL texture. It is templated with the four parameters. Several widely used default combinations of parameters are provided.

Definition
```
template < int TEXTURE_TARGET, int TEXTURE_FORMAT,
int DATA_FORMAT, int DATA_TYPE > class Texture
typedef Texture < DEFAULT_TEXTURE_TARGET, GL_RGBA8, GL_RGBA,
GL_UNSIGNED_BYTE > RGBATexture
```

Constructors
`Texture( GLint w, GLint h, GLenum filter = GL_LINEAR, GLenum mode = GL_REPLACE )` creates an empty texture with dimensions `w * h`, filtering mode `filter` and texture mode `mode`.

`Texture( const char* pngfile, GLenum filter = GL_LINEAR, GLenum mode = GL_REPLACE )` creates a texture directly from a PNG file named in `pngfile`.

Methods

`void bind( GLenum where = GL_TEXTURE0 )` binds the texture to the specified texture unit.
`void release()` detaches the texture from its current texture unit.
`int width(), int height()` return the texture dimensions in units which are suitable for `glTexCoord*` calls. E.g., when `TEXTURE_TARGET` equals `GL_TEXTURE_2D`, then width and height will be 1.0.
`void load( const GLvoid* data, GLenum data_format, GLenum data_type )`
`void load( const char* pngfile )`
`void load( const IntensityImage* img )`
`void load( const RGBImage* img )` offers various methods to load data from memory or from a file into the texture.
`void read( GLvoid* data )` writes the texture contents to the specified memory location.

`X3DRender`

Description This class contains the lightweight X3D rendering engine described earlier.

Definition `class X3DRender: public TiXmlVisitor`

Constructor `X3DRender()`

Methods This class has no public methods of its own. Instead, a `TiXmlDocument` should be created, e.g. as `TiXmlDocument* doc = new TiXmlDocument("foo.x3d"); doc->LoadFile();`. Afterwards, e.g. in the `display` callback of a `GLUTWindow` object, the renderer can be instructed to traverse the document by calling `doc->Accept( &x3drender );`.

A.6.4 `libgestures`

This library provides the core functionality for the generic gesture recognition engine described in section 5.4. In particular, these are the `Region`, `Gesture` and `Feature` classes.

`Region`

Description This class is derived from `std::vector< Vector >` which describes the outline of the on-screen region as a closed polygon. The last element in the vector is implicitly connected to the first one.

Definition `class Region: public std::vector<Vector>`

Constructor `Region( int flags = (1<<INPUT_TYPE_COUNT)-1 )` The `flags` parameter is a bitmask which can contain any of the following flags: `1 << input_type` specifies that this region should be sensitive to the given type of input object, whereas the predefined flag `REGION_FLAGS_VOLATILE` tells the system that this region may move without explicit user interaction.

Methods

`int contains( Vector v )` determines whether the parameter `v` is inside the region or not and returns 0 or 1 accordingly.

`int flags(), void flags( int f )` can be used to retrieve and set the region's flags.

`std::vector<Gesture> gestures` allows access to the gestures which are currently active for this region.

`Gesture`

Description This class encapsulates the concept of a gesture event. As a gesture is composed of features, it is derived from a `std::vector` of pointers to `FeatureBase` objects (see also below). Due to the polymorphic nature of `Feature` objects, only pointers can be stored in the container. To avoid any potential memory leaks, these pointers are encapsulated in the `SmartPtr` class described earlier.

Definition `class Gesture:`
`public std::vector< SmartPtr<FeatureBase> >`

Constructor `Gesture( std::string name, int flags = 0 )` creates a gesture with descriptor `name`. The `flags` parameter determines special behaviour of the gesture: `GESTURE_FLAGS_STICKY` describes a gesture which will continue to capture input events, even if they have moved outside of the original region, `GESTURE_FLAGS_ONESHOT` describes a gesture which will only be triggered once for a specific set of input object identifiers, and `GESTURE_FLAGS_DEFAULT` determines that this gesture should be

added to the default set of gestures which are stored in the interpretation layer.

Methods
`std::string& name()` returns the descriptor of this gesture.
`int flags()` returns the gesture's flags.

`Feature`

Description The feature class is the most basic building block for the formalisms used in libTISCH. `Feature` is derived from a pure virtual class `FeatureBase`. Pointers to this base class are in turn stored in `Gesture` objects. Additionally, all features are templated with a parameter `Value` that determines the basic data type which is used as result and boundary values.

Definition `template< class Value > class Feature: public FeatureBase`

Constructor `Feature( int tf = (1<<INPUT_TYPE_COUNT)-1 )` Similar to `Regions`, the constructor takes a `flags` parameter which describes the types of input objects which the feature will react to. No other flags besides the input types are available.

Methods
`Value result()` returns the current result of this feature.
`void bounds( std::vector<Value>& bnd )` sets the boundary array of this feature to `bnd`.
`int next()` This function needs to be overloaded in derived features which are able to provide more than one single result. When called, the next available result should be selected and 1 returned. Otherwise, 0 should be returned if no more results are available. `void load( InputState& state )` This function needs to be overloaded in all derived features. The `InputState` object contains the current state and history of all input objects within the containing region. From this data and the boundary array, the `load` function should generate one or more result values and store them internally.
`const char* name()` This function also has to be overloaded in all derived features and should return a constant string specifying the feature's name.

`Factory`

Description The `Factory` class is responsible for generating feature objects by name. As a single, global factory object exists which can be accessed through the global function `g_factory()`, it is usually not necessary to create other factories. However, it is important in this context to always add the macro `RegisterFeature( FeatureClassName );` once to every source file which defines a new feature class. This macro ensures that the new feature type can be constructed by the global factory object.

A.6.5 `libwidgets`

The last library, `libwidgets`, offers a high-level programming interface for rapidly building gesture-based user interfaces. These widgets are rendered using OpenGL graphics to provide speed and flexibility. The classes are designed to be easily extensible into new types of widgets.

`Widget`

Description This class is the virtual base class for all other widgets. It provides the basic functionality of defining an outline in screen coordinates, drawing a texture and registering the widget with the interpretation layer.

Definition `class Widget`

Constructor `Widget( int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int regflags = (1<<INPUT_TYPE_COUNT)-1 )` The parameters `w, h, x, y, angle` control the initial size, position and orientation of the widget. Note that the origin of the reference coordinate system is in the centre of the parent widget or window. The `tex` parameter can optionally point to a texture object which is used to paint the widget, whereas the `regflags` parameter can be used to specify custom flags for the internal `Region` object.

Methods

`void glOutline2d( double ox, double oy )` This function should only be used in the `outline` function.

`void outline()` This virtual function should be overloaded in a derived widget when a non-rectangular outline is desired. To this end, three or more calls to `glOutline2d` should be made which specify points on the bounding polygon in widget-local coordinates. These points should be as close as possible to the graphical representation of the widget.

`void enter()` This function modifies the OpenGL matrix stack so that the local coordinate system is equal to that of the widget itself. A call to this function must always correspond to a later call to `leave`.
`void leave()` This function reverts the effects of a previous `enter()` call.
`void texture( RGBATexture* tex )` updates the widget's texture pointer.
`void paint()` draws the default representation of the widget, which is a rectangle containing the default texture given in the constructor.
`void draw()` This virtual function can be overloaded in derived classes for more complex drawing routines. Its default implementation consists of the three method calls `enter(); paint(); leave();`. Any alternative implementation should also contain two corresponding `enter()` and `leave` calls.
`void action( Gesture* gst )` This pure virtual function must be overloaded in all derived widgets. It is called when a gesture event for the widget's region has been received from the interpretation layer.

`Textbox`

Description The textbox class is directly derived from `Widget`. It provides a text entry area and additionally a pop-up keyboard which appears directly below the text. To show the keyboard, the text area should be tapped once. A second tap will hide the keyboard again. Although it would be possible to implement every single key as a widget, this would unnecessarily increase network traffic. As the ordered layout of the keys provides an easy way to map tap locations to keys, the keyboard is implemented as a single widget.

Definition `class Textbox: public Widget`

Constructor `Textbox( int w, int h0, int h1, int x = 0, int y = 0, double angle = 0, RGBATexture* tex = 0 )` creates a new textbox widget. All parameters are equal to those in the base widget with the exception of the two height parameters `h0,h1`. As the widget can have two states, `h0` specifies the height in retracted state whereas `h1` specifies the expanded height. Note that the ratio between these two heights should also be reflected in the texture, as the lower part $(h1 - h0)$ will be hidden in retracted state. When no texture is specified, then the widget will load a default texture from `"Textbox.png"`.

Methods
`std::string get()` retrieves a copy of the current text field contents.
`void set( std::string text )` changes the text field contents.

`Slider`

Description The slider widget has the purpose of adjusting a numerical value by linearly moving an object.

Definition `class Slider: public Widget`

Constructor `Slider( int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0 )` creates a new slider widget with the same parameters as the base widget. When no texture is specified, the default texture will be loaded from `"Container.png"`.

Methods

`double get()` returns the current position of the slider as a fraction between 0 and 1.

`void set( double val )` sets the slider's current position. Values outside the $[0, 1]$ range will be clipped.

`Dial`

Description The dial widget has the same purpose as a slider, which is to allow modification of a numerical value. However, this task is accomplished through rotation of a circular object. Additionally, the dial can optionally display its current value as a number on top.

Definition `class Dial: public Widget`

Constructor `Dial( int d, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0 )` As the dial is inherently circular, its dimensions can be specified through a single parameter `d` determining the diameter. All other parameters have the usual semantics. A default texture will be loaded from `"Dial.png"` when the `tex` parameter is 0.

Methods

`double get()` returns the current position of the dial as an angle between 0 and 2π.

`void set( double val )` sets the dial's current position. Values outside the $[0, 2\pi]$ range will be clipped.

`Label`

Description This completely passive class can be used to display a static text label.

Definition `class Label: public Widget`

Constructor `Label( int w, int h, std::string& text, int x = 0, int y = 0, double angle = 0.0 )` creates a label widget with the usual parameters and the initial label contents `text`.

Methods
`void set( std::string& label )` changes the displayed text.

`Button`

Description The button is one of the most basic components of many interfaces. This widget reacts to two gestures, "tap" and "release" and triggers two callbacks accordingly which can be overloaded in derived widgets.

Definition `class Button: public Widget`

Constructor `Button( int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0 )` creates a new button with the usual parameters. The default texture is `"Container.png"`.

Methods
`void tap( Vector pos, int id )` is called when a tap event occurs. The `pos` parameter specifies the location in screen coordinates, while the `id` parameter contains the identifier of the input object which triggered the event.
`void release()` is called when all input objects have been removed from the region.

`Checkbox`

Description A specialised variant of the button is the checkbox. Tapping toggles between two states, checked and cleared. When checked, a cross is displayed on top of the widget to indicate its state.

Definition `class Checkbox: public Button`

Constructor `Checkbox( int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0 )` This constructor is identical to that of the `Button` class.

Methods
`int get()` returns 0 when unchecked, 1 when checked.
`void set( int state )` sets the state of the checkbox.

`Tile`

Description The tile widget is also a very useful object, especially for common multi-touch and multi-user applications like picture browsing. A tile automatically reacts to "move", "rotate" and "scale" gestures. It is also possible to selectively disable some gestures when they should not be used in a specific UI.

Definition `class Tile: public Button`

Constructor `Tile( int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF )` The tile constructor is almost identical to those described earlier with the exception of the `mode` parameter. This parameter can be set to any combination of the flags `TISCH_TILE_MOVE`, `TISCH_TILE_ROTATE` and `TISCH_TILE_SCALE`. When one of the flags is unset, then the tile will not react to the corresponding gesture.

Methods `Tile` does not define any methods beyond those inherited from `Button` and `Widget`.

`Container`

Description The container widget can be used to group other widgets together. It is derived from `Tile` to take advantage of the movement capabilities which have already been implemented there.

Definition `class Container: public Tile`

Constructor `Container( int w, int h, int x, int y, double angle = 0.0, RGBATexture* tex = 0, int mode = 0 )` constructs a new `Container` with the same parameters as the `Tile` constructor. However, the default `mode` is 0, thereby disabling all movement.

Methods

`void add( Widget* w )` adds an existing widget to the container. From this point on, the widget will be transformed together with the container.
`void raise( Widget* w = 0 )` pushes the specified widget to the top of the container. Any other widget inside the container may be partially or fully obscured. When the default parameter of 0 is passed, then the container itself will be raised with respect to its sibling widgets.
`void remove( Widget* w )` detaches the specified widget from the container again.

`MasterContainer`

Description A master container exists to handle a special case: one top-level container should exist per application which handles issues such as communication with the interpretation layer and delivering incoming events to the correct widgets. Usually, developers only need to deal with master containers when integrating libTISCH into new windowing environments or programming languages.

Definition `class MasterContainer: public Container`

Constructor `MasterContainer( int w, int h )` Due to its designation as a top-level container which directly corresponds to a classic window on operating system level, the `MasterContainer` constructor lacks many of the previously used parameters. Only width and height need to be specified.

Methods

`void update( Widget* target = 0 )` instructs the `MasterContainer` to send an update for the specified widget to the interpretation layer. When the `target` parameter is 0, then all widgets shall be updated.

`void adjust( int w, int h )` When the containing window changes its size, then the `MasterContainer` should be notified of this change by calling the `adjust` method with the new dimensions as parameters.

`Window`

Description The window class is derived from two separate classes: `GLUTWindow` and `MasterContainer`. The first component handles issues related to creating and managing a top-level window with an appropriate OpenGL context, while the second component handles communication with the interpretation layer as mentioned above.

Definition `class Window:`
`public GLUTWindow, public MasterContainer`

Constructor `Window( int w, int h, std::string title, int use_mouse = 0 )` creates a new top-level window with dimensions `w * h` and titlebar text `title`. Should `use_mouse` be set to 1, then the window will automatically generate LTP packets based on mouse pointer actions within the window. Note that when this parameter is active at the same

time as another LTP source such as `touchd`, unspecified and very likely erratic behaviour will result.

Methods `Window` defines no additional methods. In most cases, developers will use the methods inherited from `Container` to add widgets to the window.

Glossary

API
: application programming interface - General term for a code interface through which a software library can be controlled/accessed.

CCD
: charge-coupled device - A widely used type of camera sensor.

CHI
: computer-human interaction - Discipline of computer science as well as a catch-all term for any interaction between computers and their users. Also, the name of a prestigious conference in this field. See also *HCI*.

CLI
: command line interface - Text-based method of interacting with a computer which is based on commands and responses.

DI
: diffuse illumination - Sensor technology for multi-touch and tangible interfaces.

DOF
: degrees of freedom - Metric used to describe trackers, e.g., 3DOF for position trackers in 3D space or 6DOF for position and rotation trackers.

DPI
: dots per inch - Metric for resolution of input and output devices.

EBNF
: extended Backus-Naur form - Means of describing a formal language based on a context-free grammar.

FET
field effect transistor - Semiconductor element which regulates electrical current based on input voltage.

FTIR
frustrated total internal reflection - Sensor technology for multi-touch interfaces.

GCCPHAT
generalized cross correlation with phase transformation - Signal processing algorithm designed to cope with reverberations and echoes in the signal.

GDP
Gesture Description Protocol - Communications protocol presented in this thesis which is used to define gestures.

GLUT
OpenGL Utility Toolkit - Helper library for cross-platform access to OpenGL.

GUI
graphical user interface - General term for a user interface which does not rely on text alone.

HAL
hardware abstraction layer - General term for a software which subsumes various types of hardware into one representation.

HCI
human-computer interaction - Discipline of computer science as well as a catch-all term for any interaction between computers and their users. See also *CHI*.

HD
high definition - Standard for high-resolution displays; "Full HD" is equivalent to a resolution of 1920 x 1080 pixels.

HDR
high dynamic range - General term for any method which enables an image sensor to capture data outside its standard dynamic range.

IC
integrated circuit - General term for any complex electronic semiconductor component, e.g., a processor.

IIDC
Instrumentation and Industrial Control Digital Camera - Specification for accessing and controlling industrial-grade cameras.

IR
infrared - Part of the light spectrum between wavelengths of 700 to 1400 nm (near-infrared); mostly invisible to the human eye. Different from thermal radiation (far-infrared) in the wavelength range of 3 to 5 μm.

ITO
indium tin oxide - Conductive substance which is transparent in thin films; often used in touchscreens.

LCD
liquid crystal display - Widely used technology for flatscreen displays.

LED
light-emitting diode - Semiconductor element which emits monochromatic light when a voltage is applied.

LGPL
Lesser General Public License - Free, open-source software license by the Free Software Foundation.

LTP
Location Transport Protocol - Communications protocol presented in this thesis which delivers abstracted tracking data about the user's motion.

MIDI
Music Instruments Digital Interface - Connection standard and protocol used between musical instruments, computers and auxiliary devices. See also *OSC*.

MMX
MultiMedia Extensions - Intel processor extension to allow simultaneous processing of several independent data items.

MPX
Multi-Pointer X - Extension to the standard X server that allows for multiple mouse pointers and pointing devices.

NUI
natural user interface - General term for a number of novel types of user interfaces; also the name of an internet community focused on these devices.

OSC
Open Sound Control - Communications network protocol used mainly between software sequencers. See also *MIDI*.

PNG
Portable Network Graphics - Commonly used image file format with lossless compression and transparency support.

RSSI
Received Signal Strength Indicator - Metric for reception strength of radiowaves, usually given in decibel (dBm).

SAX
Simple API for XML - Quasi-standard for accessing XML parsers. See also *XML*.

SMD
surface-mounted device - Class of electronic circuit component which is soldered on the surface of the circuit board instead of on the backside.

STL
Standard Template Library - C++ library which is part of the C++ standard and provides generic classes such as vectors, lists etc.

SWIG
Simplified Wrapper and Interface Generator - Tool to automatically generate C/C++ library bindings for a variety of other programming languages.

TISCH
Tangible Interactive Surface for Collaboration between Humans - Multitouch tabletop interface used in this thesis.

UDP
User Datagram Protocol - Connectionless datagram protocol from the Internet suite of protocols.

UI
user interface - General term for any means of communication between computers and users.

UIST
User Interface Systems and Technology - Prestigious conference focusing on technical aspects of user interfaces.

UML
Unified Modeling Language - Formal language for describing software systems.

USB
Universal Serial Bus - Hardware standard and protocol for connecting peripheral devices to a computer.

WIMP
windows, icons, menus, pointer - Interaction concepts on which many current user interfaces are modeled.

XML
Extended Markup Language - Meta-language concept which can be used to describe arbitrary, nested data structures.

Bibliography

[1] Advanced Realtime Tracking. ARTrack. http://www.ar-tracking.de/, accessed 2009-05-13.

[2] Apple Corporation. User Experience Technologies: Aqua. http://developer.apple.com/documentation/MacOSX/Conceptual/OSX_Technology_Overview/UserExperience/UserExperience.html, accessed 2009-05-13.

[3] Atmel Semiconductors. ATTiny13 Datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc2535.pdf, 2003 (accessed 2009-07-06).

[4] S. Baker et al. FreeGLUT. http://freeglut.sourceforge.net/, accessed 2009-05-14.

[5] D. Beazley. SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org/, accessed 2009-05-14.

[6] R. Bencina. oscpack library. http://www.audiomulch.com/~rossb/code/oscpack/, accessed 2009-05-14.

[7] H. Benko, A. Wilson, and P. Baudisch. Precise selection techniques for multi-touch screens. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 1263–1272, New York, NY, USA, 2006. ACM Press.

[8] P. Bennett and S. O'Modhrain. The BeatBearing: a tangible rhythm sequencer. In *Proceedings of NordiCHI 2008: 5th Nordic Conference on Computer-Human Interaction (electronic proceedings)*, 2008.

[9] A. Beshay. Hand tracking with the Wiimote. Master's thesis, Technische Universität München, Department of Computer Science, July 2009.

[10] O. Bimber. *Spatial Augmented Reality*. A K Peters, 2005.

[11] D. Bitzer. Touch Screen for PLATO IV system, 1964.

[12] Bluetooth SIG. Core specification 2.1 + EDR. `http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/`, 2007 (accessed 2009-07-06).

[13] T. Cranston, F. Longstaff, and K. Taylor. DATAR: Digital Automated Tracking and Resolving, 1952.

[14] R. Diaz-Marino, E. Tse, and S. Greenberg. Programming for multiple touches and multiple users: A toolkit for the DiamondTouch hardware. In *UIST '03: Companion proceedings of the 16th annual ACM symposium on User interface software and technology*, 2003.

[15] P. Dietz and D. Leigh. DiamondTouch: a multi-user touch technology. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 219–226, 2001.

[16] P. Dietz, W. Yerazunis, and D. Leigh. Very low-cost sensing and communication using bidirectional leds. *UbiComp 2003: Ubiquitous Computing*, pages 175–191, 2003.

[17] A. Dippon. Gleichzeitige Ansteuerung einer LED-Matrix als Display und Sensor. Master's thesis, Technische Universität München, Department of Computer Science, July 2009.

[18] N. Dörfler. Building a gesture-based information terminal. Master's thesis, Technische Universität München, Department of Computer Science, Oct. 2008.

[19] D. Douxchamps. libDC1394. `http://damien.douxchamps.net/ieee1394/libdc1394/`, accessed 2009-05-14.

[20] F. Echtler. libTISCH: Library for Tangible Interactive Surfaces for Collaboration between Humans. `http://tisch.sourceforge.net/`, accessed 2009-05-14.

[21] F. Echtler, M. Huber, and G. Klinker. Shadow tracking on multi-touch tables. In *AVI '08: Proceedings of the working conference on Advanced Visual Interfaces*, pages 388–391, 2008.

[22] F. Echtler and G. Klinker. A multitouch software architecture. In *Proceedings of NordiCHI 2008*, pages 463–466, Oct. 2008.

[23] F. Echtler and G. Klinker. Tracking mobile phones on interactive tabletops. In *MEIS '08: Proceedings of the Workshop on Mobile and Embedded Interactive Systems*, 2008.

[24] F. Echtler, S. Nestler, A. Dippon, and G. Klinker. Supporting casual interactions between board games on public tabletop displays and mobile devices. *Personal and Ubiquitous Computing*, 13(to appear), 2009.

[25] F. Echtler, T. Sielhorst, M. Huber, and G. Klinker. A Short Guide to Modulated Light. In *TEI '09: Proceedings of the conference on tangible and embedded interaction*, pages 393–396, Feb. 2009.

[26] J. Elias, W. Westerman, and M. Haggerty. Multi-touch gesture dictionary. United States Patent 20070177803, 2007.

[27] Elo TouchSystems. Carroltouch. `http://www.elotouch.com/Products/Touchscreens/CarrollTouch/`, accessed 2009-05-13.

[28] Elo TouchSystems. Securetouch. `http://www.elotouch.com/Products/Touchscreens/SecureTouch/`, accessed 2009-05-13.

[29] C. Endres, A. Butz, and A. MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems*, 1(1):41–80, 2005.

[30] D. C. Engelbart et al. SRI-ARC. A technical session presentation at the Fall Joint Computer Conference in San Francisco, 1968.

[31] J. Epps, S. Lichman, and M. Wu. A study of hand shape use in tabletop gesture interaction. In *CHI '06: Extended abstracts of the SIGCHI conference on Human Factors in computing systems*, pages 748–753, 2006.

[32] A. Esenther and K. Ryall. Fluid DTMouse: better mouse support for touch-based interactions. In *AVI '06: Proceedings of the working conference on Advanced Visual Interfaces*, pages 112–115, 2006.

[33] A. Esenther and K. Wittenburg. Multi-user multi-touch games on DiamondTouch with the DTFlash toolkit. In *INTETAIN '05: Proceedings of the International Conference on Intelligent Technologies for Interactive Entertainment*, 2005.

[34] Fairchild Semiconductor. IRF512 Datasheet. `http://www.datasheetcatalog.org/datasheets/166/283672_DS.pdf`, accessed 2009-05-14.

[35] R. Faith et al. Distributed Multihead X. `http://dmx.sourceforge.net/`, accessed 2009-05-13.

[36] Faro UK. Measuring arms. `http://measuring-arms.faro.com/`, accessed 2009-05-13.

[37] Fifth Dimension Technologies. 5DT Data Glove 14 Ultra. `http://www.5dt.com/products/pdataglove14.html`, accessed 2009-06-28.

[38] P. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381–391, 1954.

[39] C. Forlines and C. Shen. Dtlens: multi-user tabletop spatial data exploration. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 119–122, New York, NY, USA, 2005. ACM.

[40] Free Software Foundation. GNU Lesser General Public License, Version 3. `http://www.gnu.org/licenses/lgpl-3.0-standalone.html`, 2007 (accessed 2009-05-14).

[41] B. Fry and C. Reas. Processing. `http://processing.org/`, accessed 2009-05-13.

[42] D. Gelphman and B. Laden. *Programming with Quartz: 2D and PDF Graphics in Mac OS X*. Morgan Kaufmann, 2006.

[43] C. Gerthsen and D. Meschede. *Gehrtsen Physik*. Springer, 2005.

[44] S. Gilbert et al. SparshUI Toolkit. `http://code.google.com/p/sparsh-ui/`, accessed 2009-07-06.

[45] J. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118, 2005.

[46] C. Harrison and S. E. Hudson. Scratch input: creating large, inexpensive, unpowered and mobile finger input surfaces. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 205–208, New York, NY, USA, 2008. ACM.

[47] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[48] X. Heng, S. Lao, H. Lee, and A. Smeaton. A touch interaction model for tabletops and PDAs. In *PPD '08. Workshop on designing multi-touch interaction techniques for coupled public and private displays*, 2008.

[49] O. Hilliges, D. Baur, and A. Butz. Photohelix: Browsing, Sorting and Sharing Digital Photo Collections. In *To appear in Proceedings of the 2nd IEEE Tabletop Workshop, Newport, RI, USA*, Oct. 2007.

[50] O. Hilliges, D. Kim, and I. Izadi. Creating Malleable Interactive Surfaces using Liquid Displacement Sensing. In *Proceedings of the 3rd IEEE Tabletop and Interactive Surfaces*, Oct. 2008.

[51] U. Hinrichs, S. Carpendale, and S. D. Scott. Interface currents: supporting fluent face-to-face collaboration. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 142, New York, NY, USA, 2005. ACM.

[52] S. Hodges. Private communication, 2008.

[53] R. Hofer, D. Naeff, and A. Kunz. FLATIR: FTIR multi-touch detection on a discrete distributed sensor array. In *TEI '09: Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, pages 317–322, New York, NY, USA, 2009. ACM.

[54] M.-K. Hu. Visual pattern recognition by moment invariants. *Information Theory, IEEE Transactions on*, 8(2):179–187, 1962.

[55] M. Huber, D. Pustka, P. Keitler, F. Echtler, and G. Klinker. A System Architecture for Ubiquitous Tracking Environments. In *Proceedings of the 6th International Symposium on Mixed and Augmented Reality (ISMAR)*, Nov. 2007.

[56] S. E. Hudson. Using light emitting diode arrays as touch-sensitive input and output devices. In *UIST '04: Proceedings of the 17th annual ACM*

symposium on User interface software and technology, pages 287–290, New York, NY, USA, 2004. ACM.

[57] P. Hutterer. MPX - The Multi-Pointer X server. `http://who-t.blogspot.com/`, accessed 2009-07-05.

[58] IIDC Working Group. IIDC Camera Specification 1.31. `http://damien.douxchamps.net/ieee1394/libdc1394/iidc/IIDC_1.31.pdf`, accessed 2009-05-13.

[59] Intel Corporation. Open Computer Vision Library. `http://sourceforge.net/projects/opencvlibrary/`, accessed 2009-05-14.

[60] International Organization for Standardization. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994 - `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip`, 1994 (accessed 2009-05-13).

[61] S. Izadi, S. Hodges, A. Butler, A. Rrustemi, and B. Buxton. ThinSight: integrated optical multi-touch sensing through thin form-factor displays. In *EDT '07: Proceedings of the 2007 workshop on Emerging displays technologies*, page 6, 2007.

[62] S. Izadi, S. Hodges, S. Taylor, D. Rosenfeld, N. Villar, A. Butler, and J. Westhues. Going beyond the display: A surface technology with an electronically switchable diffuser. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '08)*, 2008.

[63] JazzMutant. Lemur. `http://www.jazzmutant.com/lemur_overview.php`, accessed 2009-05-20.

[64] S. Jobs et al. Touch screen device, method, and graphical user interface for determining commands by applying heuristics. U.S. Patent 7,479,949, 2008.

[65] JOGL Project. JOGL: Java binding for the OpenGL API. `https://jogl.dev.java.net/`, accessed 2009-05-14.

[66] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

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

[68] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A protocol for table-top tangible user interfaces. In *Proceedings of Gesture Workshop 2005*, 2005.

[69] M. Kaltenbrunner, S. Jordà, G. Geiger, and M. Alonso. The reacTable: A Collaborative Musical Instrument. In *WETICE '06: Proceedings of the Workshop on Tangible Interaction in Collaborative Environments (TICE) at the 15th International IEEE Workshop on Enabling Technologies*, 2006.

[70] J. Kannala and S. Brandt. A generic camera calibration method for fish-eye lenses. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1*, pages 10–13, Washington, DC, USA, 2004. IEEE Computer Society.

[71] F. Karsunke. Controlling 3D objects by using a multitouch surface with gesture recognition. Master's thesis, Technische Universität München, Department of Computer Science, July 2009.

[72] G. Klinker. Sudokuvis - how to explore relationships of mutual exclusion. In *Advances in Visual Computing, Fourth International Symposium, ISVC 2008 Las Vegas, USA, December 1-3*, volume 5359(2) of *Lecture Notes in Computer Science*, Berlin, 2008. Springer.

[73] W. Krueger, Myron, T. Gionfriddo, and K. Hinrichsen. VIDEOPLACE - an artificial reality. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'85)*, pages 35–40, 1985.

[74] M. Laforest. WiiUse library. `http://www.wiiuse.net/`, accessed 2009-05-14.

[75] H. LeCaine et al. Printed circuit keyboard. `http://www.hughlecaine.com/en/prcirkb.html`, 1962 (accessed 2009-05-13).

[76] J. Lee. Tracking your fingers with the Wiimote. `http://johnnylee.net/projects/wii/`, accessed 2009-05-13.

[77] R. Lee and G. Greenspan. JigSawDoku. `http://www.jigsawdoku.com/`, accessed 2009-05-14.

[78] Librascope. LGP-30. `http://en.wikipedia.org/wiki/Librascope_LGP-30`, 1956 (accessed 2009-07-04).

[79] H.-H. Lin and T.-W. Chang. A camera-based multi-touch interface builder for designers. In *Human-Computer Interaction. HCI Applications and Services*, 2007.

[80] Lumen Labs. LCD Monitor Database. `http://www.baseportal.com/cgi-bin/baseportal.pl?htx=/Lumenlab/main`, accessed 2009-05-14.

[81] N. Mehta. A flexible machine interface. Master's thesis, Department of Electrical Engineering, University of Toronto, 1982.

[82] Microsoft Corporation. Surface. `http://www.microsoft.com/surface/`, 2008 (accessed 2009-05-13).

[83] Microsoft Corporation. Desktop Window Manager. `http://msdn.microsoft.com/en-us/library/aa969540(VS.85).aspx`, accessed 2009-05-13.

[84] Microsoft Corporation. Windows User Interface. `http://msdn.microsoft.com/en-us/library/aa383743.aspx`, accessed 2009-05-13.

[85] F. M. Mims, III. *Led Circuits and Projects*, pages 60–61, 76–77, 122–123. Howard W. Sams, New York, USA, 1973.

[86] S. Nestler, M. Huber, F. Echtler, A. Dollinger, and G. Klinker. Development and evaluation of a virtual reality patient simulation (VRPS). In *The 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, Plzen, Czech Republic, Februar 2009.

[87] Nintendo. Wii Controllers. `http://www.nintendo.com/wii/what/controllers`, accessed 2009-06-23.

[88] NUI Group Community. What is a compliant surface? `http://wiki.nuigroup.com/Compliant_surface`, accessed 2009-05-13.

[89] NUI Group Community. Touchlib. `http://www.nuigroup.com/touchlib/`, accessed 2009-06-28.

[90] NUI Group Community. Community Core Vision. `http://ccv.nuigroup.com/`, accessed 2009-07-06.

[91] OSRAM Opto Semiconductors. SFH 4250 Datasheet. `http://www.osram-os.com/`, 2005 (accessed 2009-07-06).

[92] J. A. Paradiso, C. K. Leo, N. Checka, and K. Hsiao. Passive acoustic knock tracking for interactive windows. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 732–733, New York, NY, USA, 2002. ACM.

[93] D. T. Pham, M. Al-Kutubi, M. Yang, Z. Wang, and Z. Ji. Pattern Matching for Tangible Acoustic Interfaces. In *IPROMS 2006: 2nd Conference on Intelligent Production Machines and Systems*, 2006.

[94] Point Grey Corporation. `http://www.ptgrey.com/`, accessed 2009-05-13.

[95] T. Pototschnig. Development of a multitouch sensor for LCD screens. Master's thesis, Technische Universität München, Department of Computer Science, May 2009.

[96] R. Pozo. TNT: Template Numerical Toolkit. `http://math.nist.gov/tnt/`, accessed 2009-05-14.

[97] Qt Software. Qt Toolkit. `http://www.qtsoftware.com/products`, accessed 2009-05-13.

[98] J. Rekimoto. SmartSkin: an infrastructure for freehand manipulation on interactive surfaces. In *CHI '02: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 113–120, 2002.

[99] J. Rekimoto. Brightshadow: shadow sensing with synchronous illuminations for robust gesture recognition. In *CHI '08: Extended abstracts of the SIGCHI conference on Human Factors in computing systems*, pages 2769–2774, 2008.

[100] J. Schöning, P. Brandl, F. Daiber, F. Echtler, O. Hilliges, J. Hook, M. Löchtefeld, N. Motamedi, L. Muller, P. Olivier, T. Roth, and U. von Zadow. Multi-touch surfaces: A technical guide. Techreport, Technische Universität München, 2008.

[101] J. Schöning, M. Rohs, and A. Krüger. Spatial authentication on large interactive multi-touch surfaces. In *IEEE Tabetop 2008: Adjunct Proceedings of IEEE Tabletops and Interactive Surfaces*, October 2008.

[102] C. Shen, F. Vernier, C. Forlines, and M. Ringel. DiamondSpin: an extensible toolkit for around-the-table interaction. In *CHI '04: Proceedings of the Conference on Human Factors in Computing Systems*, pages 167–174, 2004.

[103] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, 2005.

[104] K. Shoemake. *Arcball rotation control*, pages 175–192. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[105] T. Sielhorst. Private communication, 2008.

[106] Siemens. History Site. `http://w4.siemens.de/archiv/de/geschichte/zeitleiste/chronik_3.html`, accessed 2009-06-23.

[107] Smart Technologies. SMART Board. `http://www.smarttech.com/SmartBoard`, accessed 2009-05-13.

[108] A. Solon, M. Callaghan, J. Harkin, and T. McGinnity. Case Study on the Bluetooth Vulnerabilities in Mobile Devices. *IJCSNS*, 6(4):125, 2006.

[109] S. Spielberg. Minority Report. `http://www.imdb.com/title/tt0181689/`, 2002 (accessed 2009-05-13).

[110] Sun Corporation. Creating a GUI with JFC/Swing. `http://java.sun.com/docs/books/tutorial/uiswing/index.html`, accessed 2009-05-13.

[111] R. R. Swick and M. S. Ackerman. The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire. In *USENIX Winter*, pages 221–228, 1988.

[112] The GTK+ Project. GTK+ Toolkit. `http://www.gtk.org/`, accessed 2009-05-13.

[113] L. Thomason. TinyXML. `http://www.grinninglizard.com/tinyxml/`, accessed 2009-06-18.

[114] E. Tse, C. Shen, S. Greenberg, and C. Forlines. Enabling interaction with single user applications through speech and gestures on a multi-user tabletop. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 336–343, New York, NY, USA, 2006. ACM.

[115] T. van Roon. 555 timer tutorial. `http://www.uoguelph.ca/~antoon/gadgets/555/555.html`, 2004 (accessed 2009-07-06).

[116] R. Viladomat. Reactable Role Gaming. `http://www.youtube.com/watch?v=QflrIK-m4Ts`, 2009 (accessed 2009-07-04).

[117] L. Vlaming, J. Smit, and T. Isenberg. Presenting using two-handed interaction in open space. In *Proceedings of the 3rd Annual IEEE Workshop on Horizontal Interactive Human-Computer Systems (Tabletop 2008)*, pages 31–34, 2008.

[118] U. von Zadow. libAVG. `http://www.libavg.de/`, accessed 2009-07-06.

[119] vvvv Group. vvvv: a multipurpose toolkit. `http://vvvv.org/`, accessed 2009-06-28.

[120] M. Weinand. Development and Evaluation of Ergonomic Gesture-Based Menus. Master's thesis, Technische Universität München, Department of Computer Science, May 2009.

[121] W. White. Method for optical comparison of skin friction-ridge patterns. U.S. Patent 3,200,701, 1965.

[122] A. Wilson. TouchLight: an imaging touch screen and display for gesture-based interaction. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 69–76, 2004.

[123] A. Wilson. PlayAnywhere: a compact interactive tabletop projection-vision system. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 83–92, 2005.

[124] A. Wilson, S. Izadi, O. Hilliges, A. Garcia-Mendoza, and D. Kirk. Bringing physics to the surface. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 67–76, New York, NY, USA, 2008. ACM.

[125] A. Wilson and R. Sarin. Bluetable: connecting wireless mobile devices on interactive surfaces using vision-based handshaking. In *GI '07: Proceedings of Graphics Interface 2007*, pages 119–125, New York, NY, USA, 2007. ACM.

[126] M. Wright. The Open Sound Control 1.0 Specification. `http://opensoundcontrol.org/spec-1_0`, 2002 (accessed 2009-07-06).

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

[128] M. Wu, C. Shen, K. Ryall, C. Forlines, and R. Balakrishnan. Gesture registration, relaxation, and reuse for multi-point direct-touch surfaces. In *TableTop '06: Proceedings of the first IEEE international workshop on horizontal interactive human-computer systems*, pages 185–192, 2006.

[129] Xerox PARC. Alto, 1973.

[130] L. Xiao, T. Collins, and Y. Sun. Acoustic source localization for human computer interaction. In *SPPRA'06: Proceedings of the 24th IASTED international conference on Signal processing, pattern recognition, and applications*, pages 9–14, Anaheim, CA, USA, 2006. ACTA Press.

[131] X.Org Foundation. The X Window System. `http://www.x.org/`, 2004 (accessed 2009-05-13).