A Web-Based Prototyping Platform for Smart Textiles with a Novel Quadtree Sampling Approach

Sarah Theresa Schütz

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2018

Declaration

Contents

Declaration iii

Acknowledgement vii

Abstract viii

Kurzfassung ix

1 Introduction 1

1.1 Problem Statement 1

1.2 Research Objective 1

1.3 Contributions 2

1.3.1 Framework for Rapid Prototyping 2

1.3.2 A Novel Sampling Approach 2

1.4 Thesis Structure 3

2 State of the Art & Related Work 4

2.1 Smart Textiles 4

2.1.1 Fundamentals 4

2.1.2 Projects Using Smart Textiles 4

2.1.3 Large-Scale Textile Sensors 6

2.2 Sensor Monitoring using Web Technologies 7

2.2.1 Monitoring and Controlling Home Appliance 7

2.2.2 Sensor Data as a Service 7

2.2.3 Custom Sensor Designs 7

3 Application Design 9

3.1 Objectives 9

3.2 Scope of Functions 10

3.2.1 Data Filtering 10

3.2.2 Gesture Recognition 10

3.2.3 Triggering Actions 10

3.2.4 Extended Functionality 11

3.3 Architecture 11

3.3.1 Application Setup 11

3.3.2 Data Processing 12

4 Implementation Details **14**
4.1 Technology Stack 14
4.2 Serial Data Input 15
4.2.1 Reading Data from Serial Port 15
4.2.2 Structuring Serial Data 16
4.3 Filtering 18
4.3.1 Filter Types 19
4.3.2 Implementation Structure 25
4.4 Gesture Recognition 27
4.4.1 Heuristic Approach 27
4.4.2 SVM Classifiers 28
4.4.3 Data Capturing 29
4.5 Triggering Actions 29
4.6 Communication 30
4.6.1 Web API 30
4.6.2 Streaming of Raw Data and Recognized Gestures 32

5 User Interface **33**
5.1 Device Configuration 34
5.2 Filtering 34
5.3 Gesture Recognition 34
5.4 SVM Classifiers 36
5.5 Action Triggers 37

6 A Dynamic Quadtree Approach for Reading Sensor Data **39**
6.1 Motivation 39
6.2 Quadtrees 40
6.2.1 Definition 40
6.2.2 Region Quadtree 40
6.2.3 Quadtrees in Different Research Areas 40
6.3 Method Description 41
6.4 Tree Structure 42
6.4.1 Tree Composition 42
6.4.2 Conditions to Expand a Node 42
6.5 Serialization 43
6.6 Reading Sensor Values and Finding the Optimal Threshold 44
6.6.1 Reading of Sensor Values 44
6.6.2 Hardware Requirements 46
6.6.3 Choosing the Optimum Threshold 46

7 Evaluation **51**
7.1 QuadTree Approach 51
7.1.1 Apparatus 51
7.1.2 Dynamic Resolution Sampling 51
7.1.3 Sampling Speed 58
7.1.4 Data Length 58
7.1.5 Use Cases 59

7.2 Web-Based Prototyping Approach 66
7.2.1 Custom Visualizations 66
7.2.2 Expert Interview 70

8 Discussion & Limitations **75**
8.1 Web-Based Prototyping Approach 75
8.1.1 Custom Visualizations 75
8.1.2 Expert Interview 76
8.1.3 Limitations 76
8.2 QuadTree Approach 77
8.2.1 Dynamic Resolutions 77
8.2.2 Sampling using Quadtree Structure 77
8.2.3 Limitations 78

9 Conclusions and Future Work **79**

A Source Code of Quadtree Sampling **81**

B DVD Contents **93**
B.1 PDF 93
B.2 Source Code 93

References **94**
Literature 94
Online sources 97

Acknowledgement

I would like to thank my advisor Dr. Michael Haller for helping me find a research topic and his support during the whole process, whenever I had questions about my research or writing. Furthermore, I want to thank him for taking the time to read my thesis several times and providing me with valuable feedback, which helped me get the most out of my work, for which I am very grateful. A very special gratitude goes out to Patrick Parzer, whoes ideas were decisive to the work described in this thesis. I also thank him for all his support during the whole process of my research and various discussions which inspired me to try new approaches and played a viable role in the success of my work. I am very thankful to Anita Vogl for helping me build my initial prototype, introducing me to the fabrication of smart textiles and supporting me especially during the incipient time of my research, but also later whenever I needed it. I would also like to acknowledge Dr. Christian Rendl for his brilliant ideas and lots of help during the work on my project and finally for kindly offering his time and expertise in an interview for the evaluation of my work. Particular thanks goes as well to Dr. Kathrin Probst, Dr. Florian Perteneder, Joanne Leong, and Sebastian Gassler for their kind support and valuable feedback during brain storming sessions, which were deciding to the outcome of my work. I am grateful to the Media Interaction Lab for providing me with a nice work environment and all the equipment I needed as well as lots of know how in the field of human computer interaction and smart fabrics. And finally, last but not least, I want to say thank you to all my lovely family and friends for their moral and emotional support whenever necessary. I am so grateful to have all of them in my life.

Abstract

Over the last years, valuable research was conducted in the area of smart pressure-sensitive textiles. Many different prototypes for various research topics have been developed. Although these prototypes are often fundamentally different concerning the used technology or the intended use case, they still share a lot of common functionality. However, every prototype has its own custom application and therefore similar software for the same functionality is reimplemented multiple times by different people. This thesis introduces a generic framework for rapid prototyping with different pressure sensors. The main focus lies on sensors based on smart fabrics. The architecture of the implemented, web-based and therefore platform independent system, is designed in a flexible, module-based style, to enable developers to easily adapt and extend the application to their specific needs. Additionally, a novel firmware approach for the sampling of the sensor data of smart textiles is introduced. Due to the use of a quadtree structure the sampling performance can be increased significantly and the length of the transmitted data can be reduced. The sampling approach is described, implemented and evaluated elaborately on different use cases.

Kurzfassung

In den letzten Jahren konnte die Forschung an intelligenten, druckempfindlichen Textilien wertvolle Ergebnisse erzielen. Dafür wurden Prototypen mit den verschiedensten Technologien und für diverse Anwendungsgebiete entwickelt. Trotz technischer Unterschiede zwischen den einzelnen Prototypen, gibt es zahlreiche Gemeinsamkeiten und Funktionalitäten, die in der Entwicklung und der Datenverarbeitung immer wieder benötigt werden. Daher kommt es vor, dass bei der Entwicklung von Software für intelligente Textilien, die gleichen Funktionen mehrmals von verschiedene Forschungsteams umgesetzt werden. Diese Arbeit beschäftigt sich mit der Ausarbeitung und Implementierung eines allgemeinen Frameworks, welches Entwicklern eine Grundlage für die rasche Entwicklung von anwendungsbezogener Software bietet. Das entwickelte System unterstützt Forscher, möglichst schnell mit dem Testen verschiedener Anwendungen für ihre druckempfindlichen Sensoren beginnen zu können, ohne viel Aufwand in Softwareentwicklung investieren zu müssen. Bei der Entwicklung der Softwarearchitektur wurde auf einen flexiblen, Modul-basierten Aufbau geachtet, um Entwicklern die Änderung oder das Hinzufügen von erweiterten Funktionalitäten zu erleichtern. Außerdem wurde eine neuartige Methode zur Abtastung der Sensorwerte für die Software direkt am Mikrocontroller von intelligenten Textilien entwickelt. Durch die Verwendung einer Quadtree-Struktur kann die Geschwindigkeit des Abtastens erhöht und die übertragene Datenlänge reduziert werden. Die Abtastmethode wurde ausführlich beschrieben, umgesetzt, getestet und die Anwendung des Verfahrens auf verschiedene Anwendungsfälle wurde evaluiert.

Chapter 1

Introduction

Research in the field of human computer interaction is often focused on developing new techniques for user input. Since initial research in the late 90s [25], smart textiles have been an active subject of research in this area until now [9, 21–23]. Many different prototypes for different use cases have been developed. This thesis focuses on developing a common software for smart textiles, including a novel sampling approach to increase performance for large-scale textile sensors.

1.1 Problem Statement

There are many different projects using smart fabrics for different use cases including but not limited to the interaction with mobile or IoT devices [10, 23, 31], posture or joint monitoring [7, 26, 29] and tracking of vital signs [14, 26]. Many projects implement a user interface to visualize information [7] and provide the functionality to edit different configurations to the user [23, 26, 29]. However, although similar functionality is needed by many research projects, like the need for a user interface or functionality like data sampling, signal processing or gesture recognition, no research has been found attempting to provide a common ground for smart textile applications. Therefore, individual software has to be written for every research project and the same functionality is implemented multiple times by different research groups.

1.2 Research Objective

Schneegass and Amft write about "Current and Future Challenges for Smart Textiles" in [25]. In their opinion, one major objective regarding research on smart fabrics represents the development of a common technology for the fabrication of smart textiles, which can then be customized with different software. While their book aims to present a first step to developing a universal smart fabric, this thesis likewise focuses on the creation of a common basis for smart fabric technology. However, on the contrary to the approach of Schneegass and Amft, the focus lies on the software behind smart textiles. The goal of the research is to build a universal software to be used with a variety of different smart textile prototypes, providing base functionality for important repetitive tasks like data sampling for the software running on the prototype's microcontroller as well as functionality for signal processing or gesture recognition. The research aims at addressing smart textile researchers, developers and user experience designers alike, by considering the different needs of different groups of users.

1.3 Contributions

The contributions of this thesis can be divided in two main parts. First, a web-based framework is developed which provides a common solution to repetitively needed problems, like signal processing and gesture recognition. Second, the software running on the microcontroller of smart textile prototypes has been considered and a novel sampling approach was developed.

1.3.1 Framework for Rapid Prototyping

A framework was developed implementing an extensible architecture and providing generic implementations for common tasks when developing smart textile prototypes. Thus, enabling researchers to build custom software for their prototypes on top of the provided framework is saving them valuable time, because lots of the functionality they need is already implemented and they can start prototyping different use cases quickly, while focusing on the interaction design instead of the software development. Summarized, the main advantages for user interactions researches, developers and user interaction designers are:

1. Basic functionality for signal processing and gesture recognition as well as for triggering different actions using REST API calls is provided. Custom code can always be added to extend the given functionality.
2. The generic user interface enables user experience designers or researches without programming skills to prototype different interactions and use cases for given textile sensors and provides a basic data visualization. For specific needs, custom visualizations can be added.

1.3.2 A Novel Sampling Approach

Furthermore, the software on the microcontroller of smart textiles is addressed. Due to research on large textile prototypes [3, 14, 27] and on using industrial fabrications methods for smart textiles which enable the fabrication of large textile sensors with a high resolution [5], alternatives to sampling each single intersection of conductive lines individually were researched and are presented in the course of this thesis. The following list contains the major points of research that were conducted in this matter:

1. The possibility to combine multiple sensors together and therefore sample different resolutions of a specified textile is discussed and the results of this methods are evaluated.
2. The approach of using a quadtree structure to improve sampling performance and decrease data length is presented. The performance and the limitations of this approach are evaluated.

3. Various use cases are evaluated in matters of performance and the length of the transmitted data using the proposed quadtree method. The results are compared to the standard sampling approach of sampling all single sensors sequentially.

1.4 Thesis Structure

This thesis starts with summarizing the fundamentals and important recent projects in the field of smart textiles. Furthermore, an overview of different research projects about the use of web technologies for sensor monitoring is as well given in Chapter 2. Chapter 3 introduces the basic idea and the goals for the developed application. All needed functionality and the architecture of the framework are also discussed in this chapter. The next chapter illustrates details about the implementation of the described system. The main focus of this chapter is to provide insights of the implementation and facilitate the customization of the project for developers. In Chapter 5 the user interface of the application is shown and described. Chapter 6 introduces a new approach of sampling the sensor data using a quadtree method to increase sampling performance and decrease the amount of data that needs to be transmitted to the application. The subsequent chapter presents the results of both the prototyping framework and the quadtree approach and evaluates different aspects of the implementations. These evaluations as well as possible limitations are then discussed in Chapter 8. The work concludes with a final résumé of the achieved contributions and some interesting remarks about future work.

Chapter 2

State of the Art & Related Work

This chapter outlines the current state of the art research on smart textiles and presents an overview about related work in this area. First, the fundamentals of smart textiles are described. Additionally, major recently developed projects are mentioned. Furthermore, projects from other domains which are using similar technologies for data monitoring are discussed.

2.1 Smart Textiles

This section presents a brief review of the fundamentals about smart textiles. The basic theory behind smart textiles and possible techniques for an industrial production are illustrated. Furthermore, different research projects in this area are outlined to establish the context for the research project described in this thesis.

2.1.1 Fundamentals

As stated in [25] by Schneegass and Amft, smart textiles are not fundamentally different to standard textiles with the difference that additional functionality is added, e.g. gesture recognition or posture monitoring. They also mention that, in comparison to different wearable technologies, conductive yarns sewn or woven into fabric are unobtrusive and do not change the appearance of the clothes or other textiles, and therefore a private usage of the functionality of smart textiles is possible.

The textile industry provides a variety of different textiles suitable for the creation of smart fabrics, depending on the properties of the used raw material, smart fabrics can be created using standard textile production methods like knitting or weaving, thus enabeling large-area textile sensors with high accuracy [5].

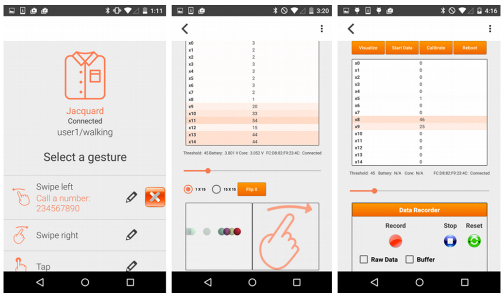
2.1.2 Projects Using Smart Textiles

Some important recent smart textile projects are listed and shortly discussed to provide an overview of current developments in the field. The goal is to show that a lot of different prototypes were developed in recent years and providing a common solution for rapid prototyping is a major contribution to the field of smart textile research.

(a)

(b)

(c)

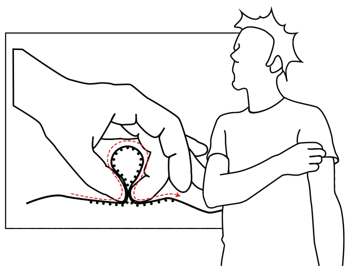


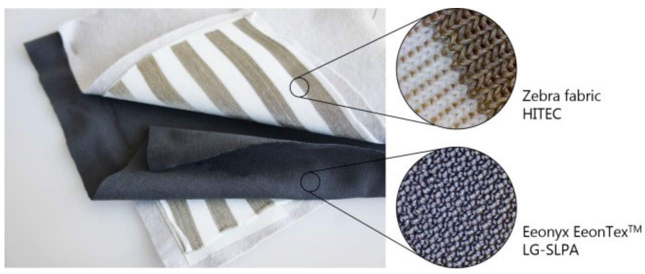
(d)

Figure 2.1: The yarns (a) developed in *Project Jacquard* can be used for weaving smart textiles (b) and were for example woven into the sleeve of a jacket (c). The developed mobile application is shown in (d). Images are taken from [23].

One of the most known projects concerning smart textiles is *Project Jacquard* by Google described in [23], where conductive yarns (see Figure 2.1 (a)) were developed and for example woven into the sleeve of a jacket, as shown in Figure 2.1 . The jacket is connected to a smartphone, were basic gestures, like swipe, hold and tap, are detected and actions, like dialing a certain number, picking up a call or ordering an Uber, are triggered. Furthermore, the possibility of using smart textiles to control IoT devices was explored in a user study that resulted in mainly positive feedback. The for the project developed application (see Figure 2.1 (d)), provides functionality like data-logging, the possibility of mapping specified gestures to the available functionality and a visualization of the data. Similar functionality is provided by the user interface of the developed system described in this thesis. However, in addition to mapping specified gestures to defined actions, the provided framework also makes it possible to train new gestures and define custom actions. Thus, more possibilities are provided for the user. Another difference between the two approaches is the target group, while the mobile application of *Project Jacquard* appears to be targeted at the end-user, the user interface described in this thesis is developed mostly for research purposes.

Another project working with lines of conductive material that are integrated in textiles is *Pinstripe* [10]. *Pinstripe* provides the user the possibility to use their clothing for continuous input which the user can then use for example to scroll through a menu or playlist on a mobile phone by pinching a piece of fabric together to create a fold and rolling it back and forth to continuously change the input value (see Figure 2.2 (a)). The granularity of the input change correlates to the size of the fold. Similar to the


(a)



(b)

Figure 2.2: The use of the *Pinstripe* prototype (a) is illustrated and the composition of the *Flextiles* sensor (b) is shown. The illustration in (a) is taken from [10] and the image in (b) is taken from [21].

provided implementation, the data produced by the performed fold was structured in a matrix and transferred to a computer via a serial connection, where signal processing was done to remove outliers. The described framework already provides functionality like reading the data from the serial connection and data filtering, and therefore would have speeded up the development of the prototype.

Further work with interactive textiles was done by the *Media Interaction Lab*[1]. They developed *FlexTiles*, a flexible pressure-sensitive input sensor was implemented using 3 layers of fabric (see Figure 2.2 (b)) presented in [21]. This sensor was then used in a project to improve sensations in prosthetic limbs [13] and for a smart sleeve with implemented gesture recognition with a combination of machine learning and a heuristic approach [22], which inspired the distinction between machine learning and heuristic approaches for the gesture recognition. Furthermore, this work was essential for the development of the described system. The prototypes used for testing the system were mostly created using the *FlexTiles* technology.

2.1.3 Large-Scale Textile Sensors

While the previously described textile prototypes mostly focus on sensors specifically adapted for different parts of the body and provide rather small input areas for specific use cases, this sections describes several projects implementing large, potentially scalable textile sensors.

Cheng et al. from the *German Research Center for Artificial Intelligence (DFKI)* for example created a pressure-sensitive carpet in [33], to be placed in front of specific furniture to track user interactions and identify the person currently using the furniture. Their underlying assumption is, that interactions with furniture can be tracked and the position of the used doors or drawers correlates to different functionality or diverse furniture content, and thus they enable to track exact actions of different people. While the current prototype is a 60×60 cm big smart textile with a resolution of 32×32 sensors, they are now working on further increasing the size and resolution.

Sundholm et al., from the same institution, further evaluated the use of smart textile-based resistive pressure sensors for the recognition of floor-based workout exercises in

[1]MI-Lab at the University of Applied Sciences Upper Austria in Hagenberg (http://mi-lab.org/)

a gym environment in [27]. They prototyped a mat with a physical size of 80 × 80 cm and a resolution of 80 × 80 sensors. Due to the use of three analog-to-digital converters with 24 bits respectively, they are able to sample the whole sensor area in 25 ms.

Another use of smart textile sensors is discreet sleep monitoring, which was researched by Li et al. in [14]. They developed a smart mat with 32 × 32 sensors which they used to analyze the distribution of pressure and to monitor the respiratory rate while sleeping.

These projects indicate, that there are various use cases for large-scale textile prototypes. This research as well as research on high resolution textile sensors inspired this research on speeding up the sampling of large smart textile sensors which led to the development of the proposed quadtree sampling approach.

2.2 Sensor Monitoring using Web Technologies

The idea of using web technologies to monitor different sensors is not new. This section describes different research projects where various data is monitored or sensor configurations are changed using web-based technologies.

2.2.1 Monitoring and Controlling Home Appliance

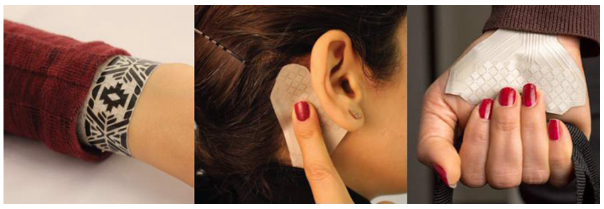
Wang et al. in [28] designed and implemented *AnyControl*, a system based on IOT and web technologies, to monitor and configure different home appliances with the goal of enhancing the user experiences of using multiple devices. *AnyControl* enables users to remotely monitor data about their homes, like room temperature or humidity, or define different tasks for controlling various appliances and trigger them automatically according to different conditions of their home environment. The implemented architecture is very similar to the system architecture described in this thesis. Sensors deliver environmental data to a *Raspberry Pi*, which is running a *Python* program implementing data processing and a web server, offering a web API to the users. The processed data is transmitted to a mobile application using a wireless WebSocket connection. A web-based *Android* application provides functionality to monitor the configured sensors, directly control different home appliances and define triggers to automate different tasks.

2.2.2 Sensor Data as a Service

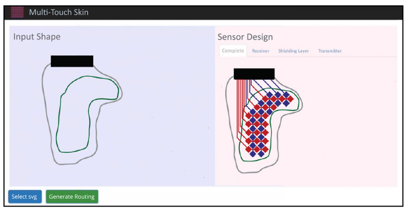
Lee et al. used a *Sensor Data as a Service* (SDS) approach to build a RESTful web service in [16], with the goal to share solar and water data as well as environmental information from the *Nevada Nexus* project between researchers. The data was collected using *Arduino* boards, sent to a web server using *Representational State Transfer* (REST) calls and persisted to a database. A user interface provides access to the data, offers search and analysis functionality and visualizes the sensor data.

2.2.3 Custom Sensor Designs

In [18] Nittala et al. developed a capacitive body-worn sensor, which is flexible and can track multi-touch input on different parts of the body (see Figure 2.3 (a)). The sensor is designed to be worn directly on the skin and the layout can have custom non-rectangular



(a)



(b)

Figure 2.3: The flexible sensor can be placed on various body parts (a). A web application helps designers to create sensor layouts (b). The images are taken from [18].

shapes to fit different body parts. The sensor data is transmitted from an *Raspberry Pi Zero*, via a wireless WebSocket connection, where their data processing pipeline is implemented. While no indication about the technology of their data processing pipeline is given, they additionally created a web interface (see Figure 2.3 (b)) to assist with custom sensor designs, where the designer can upload an SVG file describing the form and size sensor of the desired sensor and the application automatically constructs the sensor layout.

Chapter 3

Application Design

The purpose of this chapter is to outline the goals and the defined range of functions that is supported, and to discuss the architecture of the developed application. More details about the concrete implementation of the single parts of the described architecture follow in Chapter 4.

3.1 Objectives

The goal of the developed framework is to make prototyping with smart textiles easier for researchers and developers as well as for user experience designers. To keep the barriers for working with the framework to a minimum, four key principles have been defined which were considered in the development of the prototyping application.

Simplicity: The first major goal is simplicity. The resulting application and architecture should be easy to understand and use. Users are supposed to have basic programming skills. They should also have a basic understanding of how machine learning works and what a learning model is. However the developer or researcher should not have to have any preliminary knowledge about smart fabrics or hardware in general. The data from the smart fabric prototype should be provided for the user. Working with the data should be possible without understanding the background behind the data sampling of the device.

Universality: The developed system should work with any pressure-sensitive prototype without any specific use case in mind and should allow for rapid prototyping of different use cases. Therefore, the recognized gestures and triggered actions must be variable and easy to configure.

Expandability: An important key requirement is expandability. Not only the possible gestures, filters and triggered actions should be editable, also the learning models and training algorithms as well as provided heuristics should be easily extensible for developers. Basic filters, learning functionality and the infrastructure to add heuristics are provided. Developers can choose from the provided features or extend the system with custom algorithms.

Platform Independence: Another major objective is platform independence. Developers should not be forced to work with a specific operating system or programming language. Therefore, the developed system is web-based and is not based on a specific operating system. The implementation is built using *Node.js*. However, the target application, where the smart fabric prototype should be used in, does not rely on a specific programming language, but should only be able to connect to a WebSocket server to receive the sensor data as well as information about recognized gestures.

3.2 Scope of Functions

The implemented software should collect data from any connected textile prototype and empower a researcher, developer or user experience designer to rapidly prototype different gestures and dispatch action when specific gestures are performed. If the connected prototype complies with the necessary data structure (see Section 4.2), no programming should be necessary prior to prototyping. Filtering the raw data, defining new gestures and adding requests that are triggered by specified gestures should all be configurable via the user interface.

3.2.1 Data Filtering

Sometimes it can be useful to apply different filters on the raw data prior to the gesture recognition algorithm. Filters can for example eliminate errors in the sensor data caused by issues in the hardware implementation or they can be used to transform the data into another range which might improve the gesture recognition or is necessary because the values are mapped to request parameters for the triggered action. Predefined filters are provided which can be applied on the data via the user interface. The ordering of the filters is of importance since many filters are not commutative. Therefore, it is necessary to provide a simple solution for reordering filters once they are added.

3.2.2 Gesture Recognition

To recognize gestures that are drawn on or performed with the smart textile, a machine learning algorithm is provided. Defining gestures, capturing data for a specific gesture and training the machine learning model should be possible via the user interface. Programming should only be necessary for algorithmic gesture recognition or customized learning models.

3.2.3 Triggering Actions

Once a gesture is recognized it should be attached to a specific action that is performed, like for example answering a phone call, turning off the light or switching to the next song while listening to music. This actions should be configurable via the user interface and also the linking between the recognized gesture and the according action should be manageable without any programming.

3.2.4 Extended Functionality

No programming skills are required to prototype simple demonstrators. Nevertheless, the functionality of the system can be extended by adding custom implementations. Developers should be able to adapt the application to a different input data structure, add custom filters, define recognized gestures based on algorithms instead of machine learning or implement different learning models. All parts of the application should be exchangeable separately without effecting other functionality. For example, adding custom filters should not impact the gesture recognition functionality.

3.3 Architecture

A module-based application structure is necessary to provide the described functionality and to ensure that the application is easily extendable and can be adjusted for various use cases. The different modules and their functionality in the processing pipeline are described in Section 3.3.2. For every module at least one implementation is provided. Detailed information about the implementations is discussed in Chapter 4.

3.3.1 Application Setup

The application is split into three parts: a **backend application**, a **web server** and a **client application** with the graphical user interface (UI). The backend application is the core of the developed application. It handles the data analyzing and gesture recognition and triggers requests to public APIs. The UI on the other hand, is not required for the application to work, but provides a more convenient way to use the application and furthermore enables more users to benefit from the developed tool for prototyping, like for example user experience designers, which do not have a lot of programming experience. The web server represents the connection between the backend and the frontend and handles the communication between them. Some functionality of the backend and the client application relies on the same information like for example existing filter types and applied filters. To guarantee that both parts of the application act based on the same data, shared information is stored in a database which can be accessed from both the backend (directly) and the client application (via the provided API from the web server) and is the single source of truth. The basic setup is shown in Figure 3.1. When the server receives a request from the client, it either reads from the database and sends back the result or the database is updated according to the request. If entities (for example filters, triggers or gestures) are added or deleted from the database, the server also communicates these changes to the backend application. The backend application on the other hand sends the analyzed raw data and recognized gestures to the server which are forwarded to the client and visualized in the user interface. When the backend application is re-started all existing entities are read from the database and used for initialization before the application starts reading and analyzing data from the textile sensor.

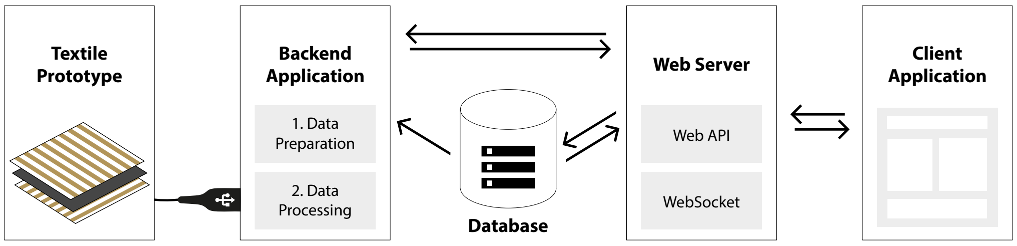


Figure 3.1: The application consists of a backend and a client part and a web server which handles the communication between them. Additionally, data is stored in a database to ensure consistency.

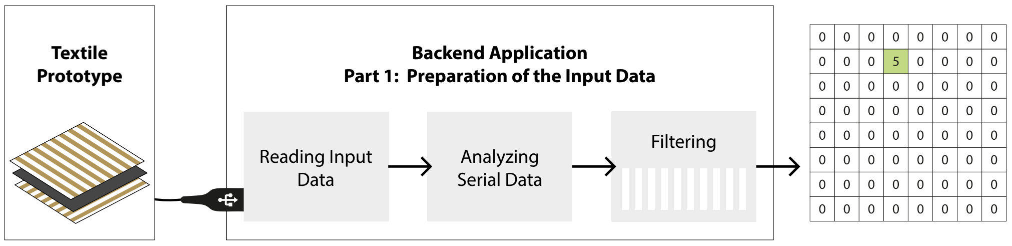


Figure 3.2: Data from the smart textile sensor is transmitted via a serial port. The first steps are to read the data from the serial port, analyze the input stream and transform the data to flattened arrays and apply filtering.

3.3.2 Data Processing

The processing of the sensor data starts with reading the transmitted data from the smart textile sensor and transforming it into a structure that the application understands. This procedure is divided in three different modules, which are illustrated in Figure 3.2. The first module handles the connection to the hardware of the prototype. It receives the raw sensor data and forwards it to the analyzing module. The input data from the textile sensor does not always have the necessary structure for the application. For example, when data is read via a USB serial port, the data is transmitted via an array buffer and not all the data is transmitted at once. The application however expects to get one array containing all data from one state of the sensor. Therefore, the second module handles the analysis of the input stream and transforms the data into the required structure. When the current state of all sensors is completed, the transformed data can be passed to the next module. Before the data is ready to be visualized and used for the different gesture recognition algorithms, filtering is applied on the data to handle errors in the sensor data or achieve different effects that improve the gesture recognition.

Once the data is complete and filtered, it is passed to the data capturing module, which stores the data to a *CSV* file. The stored data can later be used to train a machine learning model. Furthermore, the data is also forwarded to the gesture recognition algorithms to detect gestures performed on the smart textile. This part of the processing pipeline is shown in Figure 3.3. If a gesture is detected by one of the algorithms,

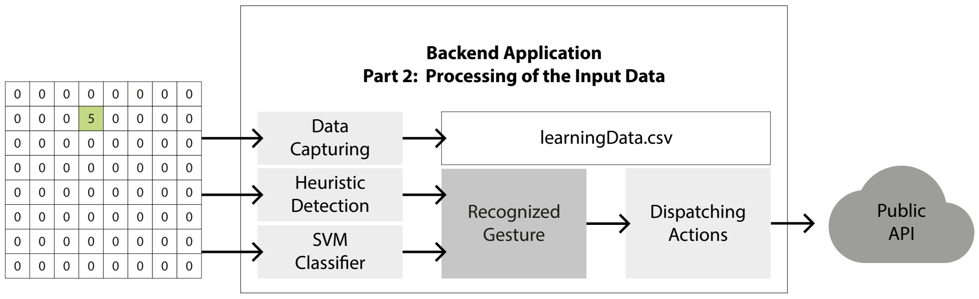


Figure 3.3: The input data is sent to the data capturing module and to the gesture recognition algorithms. If a gesture is detected and an action trigger for this gesture is defined, the corresponding HTTP request is dispatched.

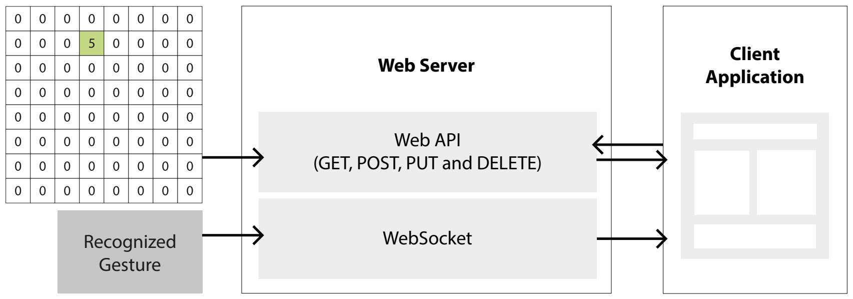


Figure 3.4: The sensor data and recognized gestures are sent to the user interface and visualized there. For the transmission the WebSocket connection handled by the web server is used. The user interface also communicates via the web API provided by the same server to interact with the backend application (see also Figure 3.1).

a HTTP request to a public or custom API can be dispatched, if such an action has been configured for the specific gesture. This allows for rapidly prototyping all use cases that can be managed via HTTP requests. Examples include using different gestures to handle music (for example *Spotify*[1]) and turning on the lights by using smart light bulbs, e.g. *Philips Hue*.[2]

The sensor data as well as recognized gestures are all sent to the client application via the WebSocket connection handled by the web server, as shown in Figure 3.4. The same web server also provides a web API which is used by the client to make changes to the processing pipeline of the backend application, like for example to add different filters or train new gestures, which will be recognized by the SVM classifier.

[1]https://www.spotify.com/at/
[2]https://www2.meethue.com/en-us

Chapter 4

Implementation Details

This chapter provides more insights about the implementation. The sequence of the discussed topics represents roughly the order they are executed in. First, the technology that was used to implement the idea is mentioned. Second, the requirements on the hardware prototypes and the reading of the serial data stream are discussed. After that, there is some information given about filtering and gesture recognition. The end of this chapter describes ways to communicate with the application and how data is sent to and received from the user interface. While aiming to summarize the whole implementation, a special focus has been given to details, that are important for others to work with, which are explained more elaborately.

4.1 Technology Stack

The server-side application was developed using *Node.js*. The client relies on *JavaScript*, HTML and CSS. This decision should facilitate the use of the software on different platforms and the communication between the client and the server application. The used technology stack of the implementation can be split into two parts. Although the client application and the server-side implementation are based on the same technology, they are separate applications and can be developed independently.

The user interface (UI) is built as a single page application using the *React*[1] library. Due to the use of the JSX[2] syntax extension and *ECMAScript 6* (ES6) language features, which are not supported by all browsers yet, the source code has to be pre-processed and transformed into standard *ECMAScript 5* (ES5) [36, 37]. Thus, *Babel*[3] is used to compile JSX and ES6. It is configured as part of a *webpack*[4] setup that also handles script bundling and hot reloading during development. The basic project setup was done using the *create-react-app*[5] tool. Some additions have already been made. For example the use of the CSS extension language SASS (*Syntactically Awesome Style Sheets*)[6] has been configured.

[1]https://reactjs.org/

[2]https://facebook.github.io/jsx/

[3]https://babeljs.io/

[4]https://webpack.js.org/

[5]https://github.com/facebook/create-react-app

[6]http://sass-lang.com/

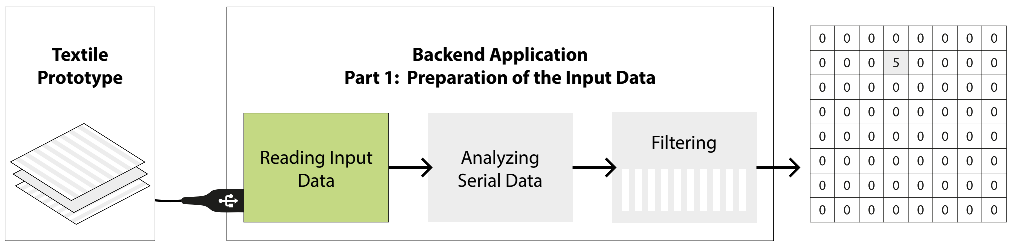


Figure 4.1: The first module handles reading the input data from the serial port.

The backend of the implementation is realized in *Node.js*. The main functionality of the backend is to read out the sensor data from the USB serial port, analyze the data and apply filters and gesture recognition algorithms. A HTTP server handles the communication with the user interface via REST API calls and a WebSocket connection. Shared information between the frontend and backend application, e.g. which filters are applied or which gestures are trained, is stored in a *MongoDB*[7] database for which *mongoose*[8] is used for object modeling. For the web server the *Express*[9] framework is used, together with the *body-parser*[10] middleware for parsing the JSON body of the requests. More details about the REST API and the WebSocket communication are discussed in Section 4.6.

The backend and the client application both use *yarn*[11] as the package manager of choice. Like *npm*[12] it relies on the large *npm* software registry. However *yarn* is used over *npm* because it is faster, a lockfile is generated, which guarantees that the very same version of all dependencies will be installed across multiple machines, and it has a flat dependency tree which avoids duplicate dependencies [39].

4.2 Serial Data Input

Reading and analyzing the raw sensor data from the connected device describes the first step of the application. First, the data has to be read from the input device. After that, the data needs to be transformed into a structure the application can work with.

4.2.1 Reading Data from Serial Port

As shown in Figure 4.1, the first module of the backend application handles reading the input data from the connected device. The implementation expects the textile prototype to be connected via an USB port. There are different *Node.js* libraries for reading serial input from a COM-port. A fast and easy solution is using the *Serial Port JSON Server (SPJS)*[13]. This tool starts a WebSocket server and distributes the sensor data

[7]https://www.mongodb.com/
[8]http://mongoosejs.com/
[9]https://expressjs.com/
[10]https://www.npmjs.com/package/body-parser
[11]https://yarnpkg.com/lang/en/
[12]https://www.npmjs.com/
[13]https://github.com/johnlauer/serial-port-json-server

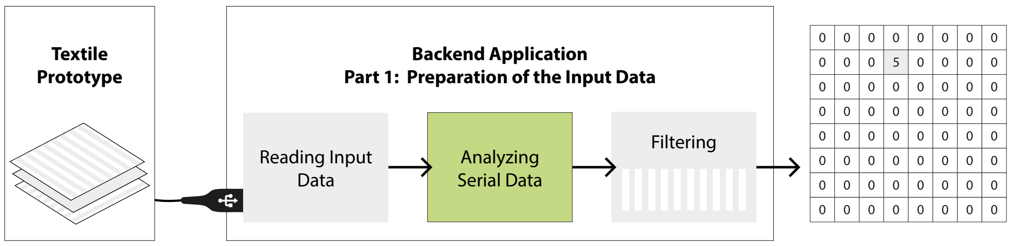


Figure 4.2: After the input data is read from the connected prototype, the data has to be transformed in the required data structure.

to any open connection. However, it is not sufficient if the data needs to be altered by the backend application before it is distributed to the client application. Therefore, a custom implementation of the described module is necessary. In the provided framework the module, handling the reading of the input data, correlates to the class `SerialInputReader`, which uses the *serialport*[14] library to read from the USB port. `SerialInputReader` extends the `EventEmitter` class. Every time new data is read from the serial port the `incoming-data`-event is emitted with the corresponding data.

4.2.2 Structuring Serial Data

The implemented application is designed to work with arrays of sensor values, that contain all values for the current state of the sensor at once. The data from the serial port does not always contain a complete iteration of all sensor values, but is returned in form of an array buffer, where the number of the read bytes can differ depending on the timing of the start of the connection and the performance of the USB port and the connected device. Therefore, before the data can be passed to the filter pipeline, it has to be analyzed. This step is handled by the second module (see Figure 4.2). The single data chunks have to be connected to an array containing all values for one state of the connected sensor in the right order.

Analyzing the Serial Data Stream

To combine and structure the raw data in the correct order the class `DataAnalyzer` implements the second module of the backend application (see Figure 4.2). The implementation depends on a specific structure of the input data. First, 6 header bytes describing the data transmission are expected, followed be the values of all single sensors. The exact data structure is described in detail in the next section.

Depending on this structure the `analyze` method of the `DataAnalyzer` class transforms the incoming serial input stream into arrays containing one complete state of the connected sensor, respectively. Therefore, the first step is to watch for the start of a new message and use the header data to initialize all required information. Subsequently, the sensor information is read and stored. Once the incoming data of the current state is complete, an event is sent containing the transformed data. The basic algorithm is defined in Algorithm 4.1.

[14]https://www.npmjs.com/package/serialport

Algorithm 4.1: The algorithm analyzes the input data from a serial port and stores the data in a global list *sensorValues*. Once one entire state of the sensor is received, an event with the resulting values is emitted.

```
global variables
    $isInitialized \leftarrow false$
    $index \leftarrow 0$
    $sensorValues \leftarrow ()$
    $rows$, $cols$, $dataLength$                                ▷ undefined
end global variables

ANALYZE($data$)
Analyzes and transforms the input data into one complete state of all sensors.
    if $isInitialized$ then
        HANDLEDATA($data$)
    else
        INITIALIZE($data$)
    end if
end

HANDLEDATA($data$)
    if $index = 0$ then                    ▷ start of the sensor data transmission
        remove the first 6 entries from the $data$ array (header bytes)
    end if
    for $i \leftarrow 0, \ldots,$ length of $data - 1$ do
        $sensorValues[index] \leftarrow data[i]$
        $index \leftarrow index + 1$
    end for
    if length of $sensorValues = dataLength$ then    ▷ $sensorValues$ are complete
        EMITEVENT($sensorValues$)
        $index \leftarrow 0$
    end if
end

INITIALIZE($data$)
    for $i \leftarrow 0, \ldots,$ length of $data - 1$ do
        if $data$ contains start byte 0xDF at index $i$ then
            $rows \leftarrow data[i + 1]$
            $cols \leftarrow data[i + 2]$
            $dataLength \leftarrow rows \cdot cols$
            $isInitialized \leftarrow true$
            $restData \leftarrow$ all values from $data$ starting from index $i$
            ANALYZE($restData$)
        end if
    end for
end
```

The implementation of the algorithm must take into account that the start byte `0xDF` can also appear as part of the sensor values. Therefore, it is not sufficient to assume a new transmission is starting when `0xDF` is received. Furthermore, it must be considered that the header bytes do not have to be located at the beginning of the read part of the input stream and can even be received in multiple parts.

If a user wants to use a prototype with a different structured data stream, the `DataAnalyzer` can be exchanged by a custom class implementing a method called `analyze` which receives an incoming data from the stream and emits a `data-array` event with the current values when a complete state of the sensor input is received.

Raw Data Structure

To comply with the developed application, the hardware has to send the data in a specified way. All sensor values are sent sampled with 8 bits, which means every sensor is represented by 1 byte. At the beginning of each transmission a 6 byte long header is sent, containing a defined start value and information about the sensor size, the used encoding and the length of the following data. The first byte always has the value `0xDF`. It is used to mark the start of the transmission header. The second and third byte contain the resolution of the used sensor by specifying the number of rows and cols. Byte number four is an indicator for the encoding used for the data and byte number five and six store the length of the submitted data. An example header is shown in Figure 4.3.

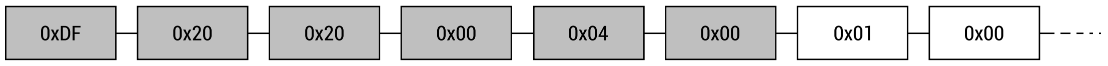
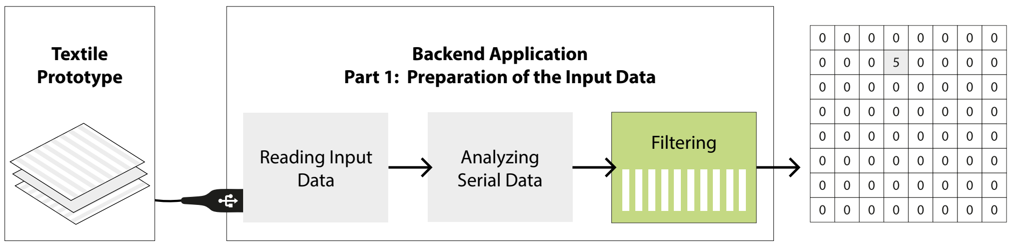


Figure 4.3: Every data transmission starts with six bytes of header information. The connected hardware in this example has a resolution of 32 rows and 32 columns, no encoding algorithm is used on the raw data and there are 1024 bytes following containing the data.

4.3 Filtering

To account for errors and flickering in the raw sensor data, the next module is data filtering, as shown in Figure 4.4. Different filters are provided which can be applied as a preprocessing step on the sensor data prior to data capturing and gesture recognition. Multiple filters can be combined to achieve the desired result. Since most filter combinations are non-commutative, the order of the applied filters is of importance and has to be considered carefully.

In this section, first, different filter implementations are explained and formalized. In the end, the implementation structure of the filter pipeline and the possibility to implement new filters are discussed.



0	0	0	0	0	0	0	0
0	0	0	5	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 4.4: Once the analysis of the input data is complete, filtering is applied on the data to account for errors in the raw data.

4.3.1 Filter Types

Ten different filters have been formalized and implemented. The input data D for every filter are the sensor values $D(i)$, where $i \in [0, N-1]$ and N is the product of the sensor resolution in x and y direction. D can either be the raw sensor data from the serial data stream discussed in Section 4.2 or result from an already applied filter. Each filter iterates over all values $D(i)$ and transforms them according to the specific implementation of the filter. Then the transformed values $D'(i)$ are returned, which are then used as input for the next filter or directly as input for the data capturing and gesture recognition as well as for the data visualization.

Scale Filter

The *scale filter* (see Figure 4.5) is a linear filter which applies the scale factor s on every value $D(i)$ of the input D and can be formalized as

$$D'(i) \leftarrow D(i) \cdot s. \tag{4.1}$$

The scale factor s is not constant, but can be specified by the user when adding the filter. *Scale filters* can be used to amplify or decrease the signal intensity. In case they are used to amplify a signal they will most likely be used in combination with another filter that will filter out noise in the sensor data to prevent increasing the noise as well.

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

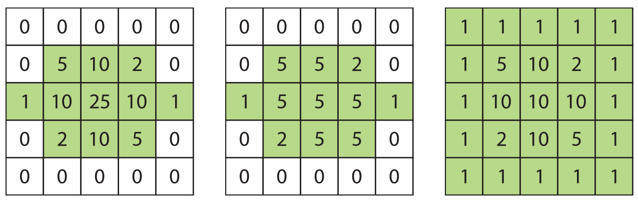
0	0	0	0	0
0	10	20	4	0
2	20	50	20	2
0	4	20	10	0
0	0	0	0	0

(b) D'

0	0	0	0	0
0	50	100	20	0
10	100	250	100	10
0	20	100	50	0
0	0	0	0	0

(c) D'

Figure 4.5: The *scale filter* is used to multiply all values with a constant factor s. The input data (a), the filter result where $s = 2$ (b) and the result of the filter where $s = 10$ (c) are shown.



0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

0	0	0	0	0
0	5	5	2	0
1	5	5	5	1
0	2	5	5	0
0	0	0	0	0

(b) D'

1	1	1	1	1
1	5	10	2	1
1	10	10	10	1
1	2	10	5	1
1	1	1	1	1

(c) D'

Figure 4.6: The *clamping filter* limits the range of the sensor values. The input data (a) and the results of the filter with $r_{\text{min}} = 0$ and $r_{\text{max}} = 5$ (b) and $r_{\text{min}} = 1$ and $r_{\text{max}} = 10$ (c), respectively, are shown.

Clamping Filter

To restrain the input values D to a specified range $[r_{\text{min}}, r_{\text{max}}]$ the *clamping filter* can be used. With an active *clamping filter* (see Figure 4.6) all values D between r_{min} and r_{max} (including r_{min} and r_{max}) remain unchanged while values below r_{min} become r_{min} and values above r_{max} become r_{max}, i.e.,

$$D'(i) \leftarrow \begin{cases} r_{\text{min}} & \text{if } D(i) < r_{\text{min}}, \\ D(i) & \text{if } r_{\text{min}} \leq D(i) \leq r_{\text{max}}, \\ r_{\text{max}} & \text{if } D(i) > r_{\text{max}}. \end{cases} \tag{4.2}$$

The *clamping filter* is useful to set an upper bound on the sensor values or to eliminate negative values (resulting from other filters).

Band-Pass Filter

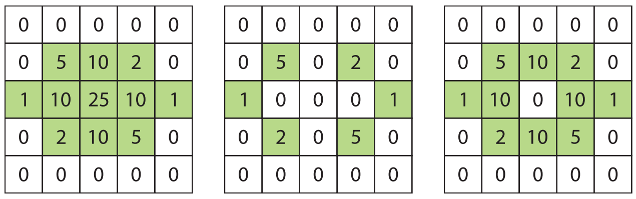
Similar to the *clamping filter*, the *band-pass filter* (see Figure 4.7) also works with a range $[r_{\text{min}}, r_{\text{max}}]$ and keeps values in this range (again including r_{min} and r_{max}) unaffected. Values outside the specified range however are cleared and default to zero, i.e. (analog of Equation 4.2),

$$D'(i) \leftarrow \begin{cases} 0 & \text{if } D(i) < r_{\text{min}} \text{ or } D(i) > r_{\text{max}}, \\ D(i) & \text{if } r_{\text{min}} \leq D(i) \leq r_{\text{max}}. \end{cases} \tag{4.3}$$

Highest Peak Filter

The *highest peak filter* (see Figure 4.8) sets all signal values $D(i)$ to zero except for the highest occurring value. There can be be multiple occurrences of the same value if multiple sensors share the same highest value:

$$D'(i) \leftarrow \begin{cases} 0 & \text{if } D(i) < \max(D), \\ D(i) & \text{if } D(i) = \max(D). \end{cases} \tag{4.4}$$



0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

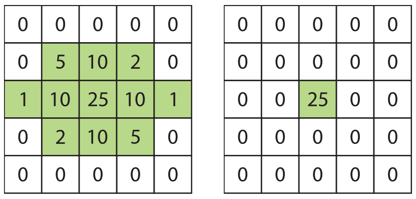
0	0	0	0	0
0	5	0	2	0
1	0	0	0	1
0	2	0	5	0
0	0	0	0	0

(b) D'

0	0	0	0	0
0	5	10	2	0
1	10	0	10	1
0	2	10	5	0
0	0	0	0	0

(c) D'

Figure 4.7: This figure shows the results of the *band-pass filter*. The input data is shown in (a). The results for $r_{\text{min}} = 0$ and $r_{\text{max}} = 5$ as well as $r_{\text{min}} = 1$ and $r_{\text{max}} = 10$ are shown in (b) and (c), respectively.



0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

0	0	0	0	0
0	0	0	0	0
0	0	25	0	0
0	0	0	0	0
0	0	0	0	0

(b) D'

Figure 4.8: The *highest peak filter* only keeps the highest value and sets all other values to 0. The input data (a) and the result (b) are shown.

Rotation Filter

Like explained in Section 4.2, the alignment of a single value $D(i)$ of the sensor data matrix D is specified by the order the sensor values $D(i)$ are sent in by the micro-controller of the device. The *rotation filter* performs a 90 degree clockwise rotation on the sensor data (see Figure 4.9). This can be useful in case the sensor values are aligned differently than someone would expect. Reasons therefore could be advantages for the hardware implementation or later using a prototype in a different orientation than initially planned.

The rotation of the sensor data D is performed in three steps. The first step is to expand the one-dimensional input data to a two-dimensional matrix M by using the width w and height h specified, when creating the filter. If w and h are not specified, the default value is 32 respectively, because this is the standard sensor size of the devices used for developing this filter. With the given width and height of the resulting matrix, all values $M(i, j)$ with $i, j \in \mathbb{N}$ can be calculated as

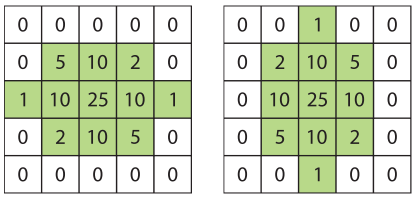
$$M(i, j) \leftarrow D((j \cdot w) + i), \tag{4.5}$$

where $i = 0, \ldots, w - 1$ and $j = 0, \ldots, h - 1$. The next step is to rotate the matrix M by 90 degrees to get the rotated matrix

$$M'(i, j) \leftarrow M(j, (w - i - 1)). \tag{4.6}$$

Finally, the 2-dimensional data is flattened to one dimension, that is,

$$D'((j \cdot w') + i)) \leftarrow M'(i, j), \tag{4.7}$$



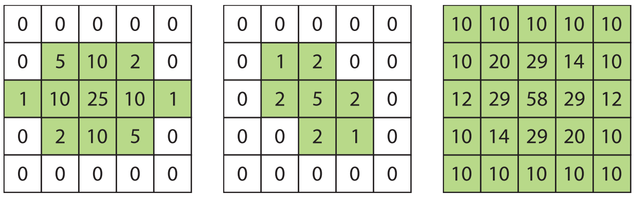
0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

0	0	1	0	0
0	2	10	5	0
0	10	25	10	0
0	5	10	2	0
0	0	1	0	0

(b) D'

Figure 4.9: The result (b) of the *rotation filter* is a 90 degree clockwise rotation of the input data (a).



0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D

0	0	0	0	0
0	1	2	0	0
0	2	5	2	0
0	0	2	1	0
0	0	0	0	0

(b) D'

10	10	10	10	10
10	20	29	14	10
12	29	58	29	12
10	14	29	20	10
10	10	10	10	10

(c) D'

Figure 4.10: The *remap filter* maps the input values (a) from the given range r_{min} to r_{max} to a new range r'_{min} to r'_{max}. The results of two examples are shown in (b) (where $r_{\text{min}} = 0$, $r_{\text{max}} = 255$, $r'_{\text{min}} = 0$ and $r'_{\text{max}} = 50$) and (c) (where $r_{\text{min}} = 0$, $r_{\text{max}} = 100$, $r'_{\text{min}} = 10$ and $r'_{\text{max}} = 200$).

where $w' = h$, due to the rotation of the matrix in the previous step. While the *rotation filter* always rotates 90 degrees, 180 and 270 degree rotations can be achieved by applying the filter multiple times. Since the *rotation filter* is the only filter implementation that is changing the location of d in the input data D, the positioning of the *rotation filter* in the filter pipeline is of no importance.

Remap Filter

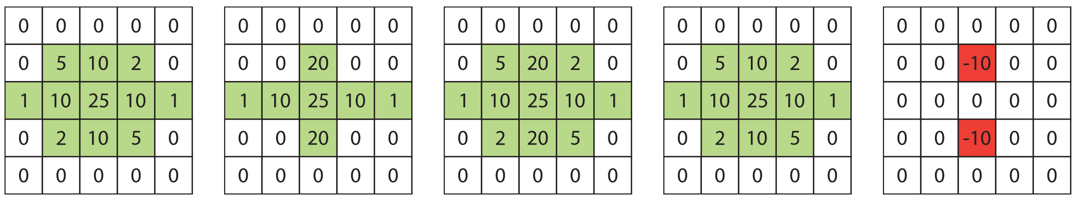
If the sensor data D should be mapped to another range the *remap filter* (see Figure 4.10) can be used. It takes all sensors $D(i)$ with an original range $r = [r_{\text{min}}, r_{\text{max}}]$ and maps them to a specified range $r' = [r'_{\text{min}}, r'_{\text{max}}]$ using linear interpolation, i.e.,

$$D'(i) \leftarrow \frac{(D(i) - r_{\text{min}}) \cdot (r'_{\text{max}} - r'_{\text{min}})}{(r_{\text{max}} - r_{\text{min}})} + r'_{\text{min}}. \tag{4.8}$$

Before the interpolation, the filter automatically sets all values below r_{min} or above r_{max} to r_{min} and r_{max}, respectively, to ensure a fixed lower and upper bound of the input values. Therefore it is not necessary to use the *remap filter* in combination with a *clamping filter*.

Individual Offset Filter

All previously discussed filters only depend on the current input data. The *individual offset filter* (see Figure 4.11) is different. When it is added by the user or on server



(a) D_1

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(b) D_2

0	0	0	0	0
0	0	20	0	0
1	10	25	10	1
0	0	20	0	0
0	0	0	0	0

(c) $O_{1\ldots n}$

0	0	0	0	0
0	5	20	2	0
1	10	25	10	1
0	2	20	5	0
0	0	0	0	0

(d) D

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(e) D'

0	0	0	0	0
0	0	-10	0	0
0	0	0	0	0
0	0	-10	0	0
0	0	0	0	0

Figure 4.11: This figure shows an example of the *individual offset filter* where $n = 2$. The first two data inputs (a) and (b) that are received by the filter are used to calculate the offset values (c). The result for the input data (d) after the initialization process is shown in (e).

start-up, the first n times that the filter receives data, this data D is used to initialize the filter's offset values O. $O(i)$ for the current initialization step $m = 1, \ldots, n$ is defined to contain the maximum value of $D_m(i)$ and the offset values from the previous initialization step $O_{m-1}(i)$ which means after time n the maximum value of all values $D(i)$ used for initialization is

$$O_m(i) \leftarrow \begin{cases} D_m(i) & \text{if } m = 1, \\ \max(D_m(i), O_{m-1}(i)) & \text{if } 1 < m \leq n. \end{cases} \tag{4.9}$$

This offset data O is then subtracted from the *individual offset filter*'s input data D to generate the filtered data

$$D'(i) \leftarrow D(i) - O(i). \tag{4.10}$$

As the filter can result in negative values $D'(i)$, it should be combined with a *clamping filter* or a *band-pass filter* with $r_{\text{min}} = 0$. The *individual offset filter* is a good choice to account for errors in the sensor data that are constant over time and most likely caused by hardware problems like shortcuts in the used prototype.

Adaptive Offset Filter

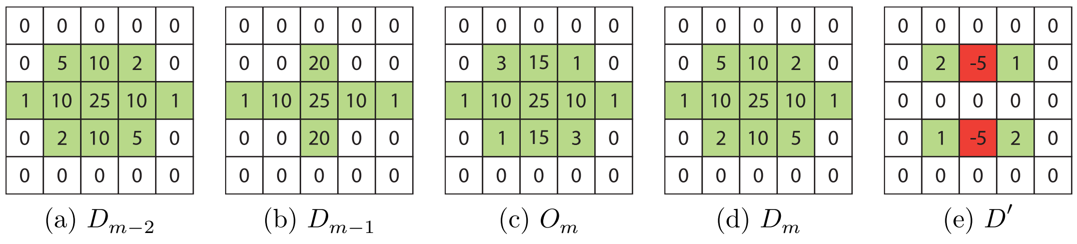
Similar to the *individual offset filter*, the *adaptive offset filter* (see Figure 4.12) also calculates offset values $O(i)$ and subtracts them from the input data values $D(i)$. The difference is, that O is not calculated just once when initializing the filter, but the filter keeps a history of the last n data frames and uses the average of this frames as an offset

$$O_m(i) \leftarrow \frac{1}{n} \cdot \sum_{m=1}^{n} D_m(i). \tag{4.11}$$

Like in the *individual offset filter* before, O is then subtracted from D to get the resulting values

$$D'(i) \leftarrow D(i) - O(i), \tag{4.12}$$

which are returned by the filter.


(a) D_{m-2}

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(b) D_{m-1}

0	0	0	0	0
0	0	20	0	0
1	10	25	10	1
0	0	20	0	0
0	0	0	0	0

(c) O_m

0	0	0	0	0
0	3	15	1	0
1	10	25	10	1
0	1	15	3	0
0	0	0	0	0

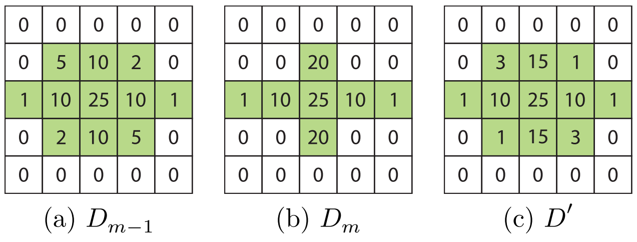
(d) D_m

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(e) D'

0	0	0	0	0
0	2	-5	1	0
0	0	0	0	0
0	1	-5	2	0
0	0	0	0	0

Figure 4.12: This is the equivalent example of Figure 4.11 for the *adaptive offset filter*. Instead of the first two frames, the last two inputs (a) and (b) are used to calculate the offset values (c). Another difference is, that the average value is used to calculate the offset values instead of the maximum value. Again (e) shows the result for the current input data (d).


(a) D_{m-1}

0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(b) D_m

0	0	0	0	0
0	0	20	0	0
1	10	25	10	1
0	0	20	0	0
0	0	0	0	0

(c) D'

0	0	0	0	0
0	3	15	1	0
1	10	25	10	1
0	1	15	3	0
0	0	0	0	0

Figure 4.13: The *sliding window filter* returns the average of the last n data frames (in this example $n = 2$). The result of the data inputs, which are shown in (a) and (b), is shown in (c).

Sliding Window Filter

The *sliding window filter* (see Figure 4.13) uses the same underlying principal as the *adaptive offset filter*. The last n input frames are stored and averaged as well. The difference is that the average of all stored frames is directly returned as the result of the filter, i.e.,

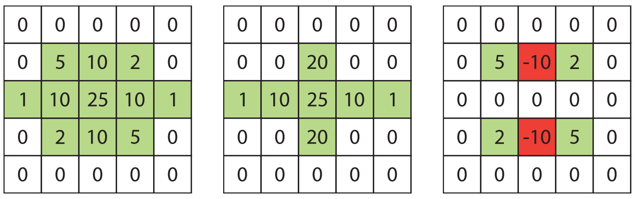
$$D'(i) \leftarrow \frac{1}{n} \cdot \sum_{m=1}^{n} D_m(i). \tag{4.13}$$

Delta Filter

Another filter, which stores data from the previous state as well, is the *delta filter* (see Figure 4.14), which always returns the difference of the current data values $D_m(i)$ and the previous input values $D_{m-1}(i)$ by subtracting D_m from D_{m-1},

$$D'(i) \leftarrow D_{m-1}(i) - D_m(i). \tag{4.14}$$

Due to the subtraction, negative values are possible in the resulting data D' of the *delta filter*.



0	0	0	0	0
0	5	10	2	0
1	10	25	10	1
0	2	10	5	0
0	0	0	0	0

(a) D_{m-1}

0	0	0	0	0
0	0	20	0	0
1	10	25	10	1
0	0	20	0	0
0	0	0	0	0

(b) D_m

0	0	0	0	0
0	5	-10	2	0
0	0	0	0	0
0	2	-10	5	0
0	0	0	0	0

(c) D'

Figure 4.14: The *delta filter* always returns the difference of the last two data frames (a) and (b). The result is shown in (c).

4.3.2 Implementation Structure

All described filters are implemented as single classes. They inherit from a base class `Filter`, which ensures that the methods `getName` and `filter` are overwritten in the subclasses and provides access to the filter's id which has to be set at time of creation:

```
class Filter {

    constructor(id) {
        this.id = id;
    }

    getId() {
        return this.id;
    }

    /** Implementation required */
    static getName() {
        throw new Error('getName() method not implemented');
    }

    /** Implementation required */
    filter(data, rows, cols) {
        throw new Error('filter() method not implemented');
    }

    reset() {}
}
```

The parameters `rows` and `cols` of the `filter` method are not used for all filters. They are for example important for the *rotation filter*, because the dimensions of the sensor are needed to convert the one-dimensional data array into a two-dimensional matrix with the correct dimensions.

Filter Composites

The composite pattern has been used to ensure that different filters can be added and combined dynamically to facilitate maximum flexibility. Developers can add single filters or build a whole filter pipeline. This was achieved by implementing another filter, additionally to the ten previously discussed filter types. This filter is called *filter composite* and behaves like a regular filter, but internally stores a list of multiple filters.

By calling `appendFilter(filter, options = {})` with either an object from a class derived from `Filter` or a `string` containing the name of a filter type, a filter is added to the collection. When the `filter` method of the *filter composite* is called, the `filter` method of each filter from the list is called, respectively. `removeFilter(filterId)` can be used to remove a filter from the collection again and `moveFilter(startIndex, endIndex)` enables the user to move the filter to a different position in the list which can be necessary because not all filters are commutative and so the order in which the filters are called is important. *Filter composite* also implements the `reset()`-method, which again iterates over all stored filters and resets each of them, respectively.

Filter Factory

Filters can either be created manually or by using the *filter factory*. The *filter factory* allows to create a filter via a string containing the type of the filter. Additionally, options can be forwarded to the filter if necessary:

```
const BandPassFilter = require('./filter/BandPassFilter');
const ClampingFilter = require('./filter/ClampingFilter');
const IndividualOffsetFilter = require('./filter/IndividualOffsetFilter');
...

class FilterFactory {

    getFilter(type, options = {}) {

        switch(type.toLowerCase()) {

            case BandPassFilter.getName().toLowerCase(): {
                return new BandPassFilter(options.id, options.min, options.max);
            }
            case ClampingFilter.getName().toLowerCase(): {
                return new ClampingFilter(options.id, options.min, options.max);
            }
            case IndividualOffsetFilter.getName().toLowerCase(): {
                return new IndividualOffsetFilter(options.id,
                options.initializationSteps);
            }
            ...

            default: {
                return null;
            }

        }
    }
}
```

This is especially useful to enable the user to add new filters from the user interface where the class definition of the various filter implementations is not known.

Adding New Filters

Due to the chosen structure, it is possible to implement new filters and extend the possibilities the application can be used for. Every new filter should extend the discussed

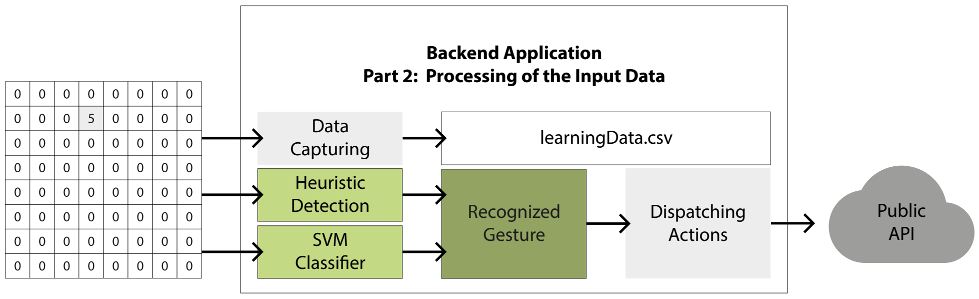


Figure 4.15: Two types of gesture recognition algorithms are performed on the input data once it is analyzed and filtered.

base class `Filter` and at least overwrite the mandatory classes `getName` and `filter`. While `getName` is only supposed to return a string with the filter's name (used for the *filter factory*), the `filter` method gets the current input data D, which can be either the raw sensor data or the result of a previous filter, so that the user can perform custom logic on D before returning the transformed data D' to be passed to the next filter or the data capturing and gesture recognition algorithms.

4.4 Gesture Recognition

Once the input data has been analyzed and filtered, the data is ready to be processed by different modules of the backend application. One task that is performed on the final input data, is gesture recognition, as shown in Figure 4.15. Different approaches have been implemented for the gesture recognition. The implemented application provides two different ways how gesture recognition can be accomplished which are described in this section. Furthermore, information about capturing data, which is for example necessary for a variety of machine learning algorithms, is given.

4.4.1 Heuristic Approach

One simple but not very flexible approach for gesture recognition is to implement algorithmic rules for specified gestures. For some gestures and use cases this might be the way to go. An advantage of this approach is that additionally to the recognized gesture properties like the position, length or direction of the gesture can be calculated and used for various purposes. In the discussed implementation, a `HeuristicDetector` class was implemented, where multiple custom heuristic rules can be added. A heuristic is in this context a class implementing a method called `analyze` which implements a rule to detect one or multiple different gestures. Every time new data is received, all added heuristics are tested. If a gesture is detected, this gesture is returned with all data that was implemented by the specific rule, which means when implementing a heuristic, the developer has the opportunity to add custom properties to the gesture object. Additionally to the gesture data the type *heuristic* is set to be able to distinguish between recognized gestures from the heuristic approach and gestures detected by the machine learning algorithm. A very simple heuristic could be implemented as follows:

```
const THRESHOLD = 2;

class SimpleTouch {
    analyze(data) {
        // iterate data and look for elevated values
        for(let i = 0; i < data.length; i++) {
            if(data[i] > THRESHOLD) {
                return { name: 'Touch', force: data[i] };
            }
        }
        return undefined;
    }
    reset() { }
}
const simpleTouchInstance = new SimpleTouch();
module.exports = simpleTouchInstance;
```

After the `SimpleTouch` gesture is added to the `HeuristicDetector`, the gesture is recognized anytime the smart fabric is touched anywhere. While this is a very simple example, heuristics can be a powerful tool to implement the recognition of multi-tap and slide gestures. Another use case would be for example to use heuristics to analyze the direction of a movement and combine this direction with the result of the SVM classification, like it was done in the *SmartSleeve* project [22].

4.4.2 SVM Classifiers

A common approach is to use a machine learning algorithm for the classification of different gestures. For the implementation the *Support Vector Machines* (SVM) algorithm is used for the gesture recognition as the input features are the raw sensor values. Therefore, there can be a high number of features and SVM is able to handle classification problems with many features [12]. The *npm* registry offers a library called *libsvm-js*[15], which is a port of the c++ library *libsvm* from Chang and Lin [38]. This library is used to perform the SVM classification.

Only if no pre-trained serialized SVM model exists, a new model is created. Before the model can be used to classify gestures it has to be trained. For the training labeled training data is necessary. Once the model is trained, it is serialized and stored in the file `/app/data/recources/svmModelData.txt` and ready to classify the input data with the sensor values into the trained gestures. Also a probability value of the correctness of the classified gesture is given which can be used for example to decide if the value is high enough that a corresponding action should be triggered. If the application is restarted at a later time the serialized model is used and there is no need to train a new classifier again except if gestures should be added or deleted.

Because the features used for the machine learning approach are the raw sensor values and there is no feature extraction prior to the learning algorithm, the recognized gestures are not location invariant. One way this could be implemented, is implementing a blob tracking algorithm and using the blob properties as features for the SVM classifier. This approach was used and tested in the *SmartSleeve* project [22].

[15]https://www.npmjs.com/package/libsvm-js

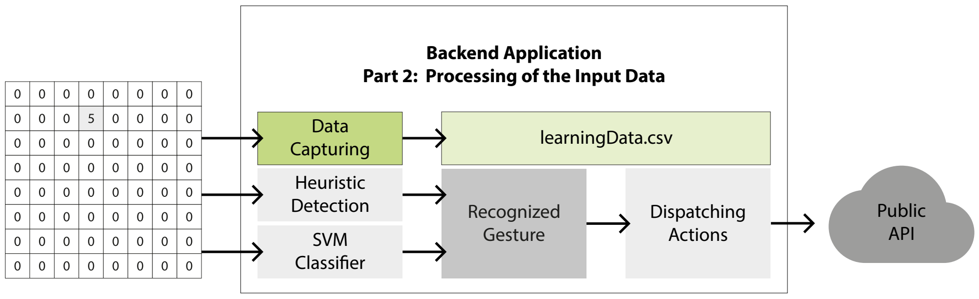


Figure 4.16: The prepared input data is passed to a data capturing module which stores the captured data to a specified CSV file.

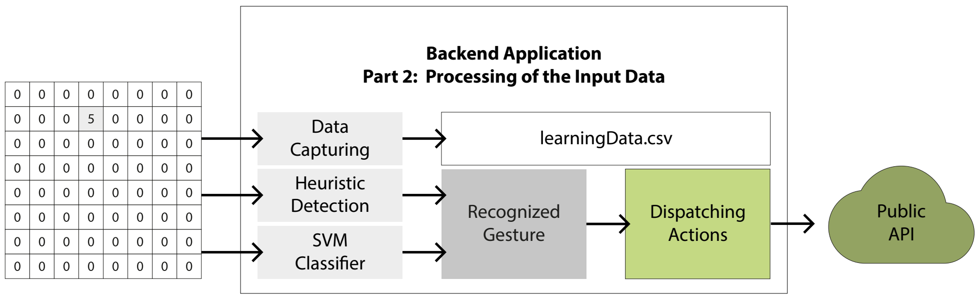


Figure 4.17: When a gesture is recognized, the configured action is dispatched which triggers a HTTP request to a public or custom API.

4.4.3 Data Capturing

Also a system to capture labeled training data for the machine learning algorithm was implemented. Every frame of incoming data is also passed to the `DataCapturer` after the raw data was analyzed and filtered (see Figure 4.16). If the instance variable `isCapturing` is set to true, every incoming dataset is stored in an array. When the capturing is finished, all stored data frames are saved in a CSV file. The methods `start(gestureName)` and `stop()` are provided to handle the beginning and ending of the capturing process. Also the possibility to just capture a single frame was implemented with the `captureSingleFrame(gestureName)` method. The `gestureName` passed when starting the data capturing is used as label for the captured data which is used as feature input.

4.5 Triggering Actions

Once prototyping different use cases, the recognition of a specific gesture should most likely result in a specified action that is triggered, as shown in Figure 4.17. Actions are for example answering a phone call, controlling a music player or switching the lights on or off. For this reason, the `ActionDispatcher` class was implemented. The `ActionDispatcher` contains a list of all defined triggers. Every trigger has the following format:

```
{
    _id: ...,                  // MongoDB ObjectId
    request: {
        method: ...,           // HTTP method to use (GET, POST etc.)
        url: ...,              // URL for the request
        data: {                // parameters that should be sent
            ...                //   with the request (key-value pairs)
        }
    },
    labels: [ ... ],           // gesture labels that should trigger the  request
    name: ...
}
```

Whenever a gesture is recognized, the *labels* array of all defined triggers is checked for an occurrence of the recognized gesture. If the gesture label is contained, the request of the respective trigger is made using the *axios*[16] library.

4.6 Communication

There are two types of communication between the backend and the client. As shown in Figure 4.18 they are both handled by the same web server. On application startup the local HTTP based web server is started on port 9000 using the *Express*[17] framework. This server provides a web API for various interactions between the front- and backend application and is also used to start a WebSocket server.

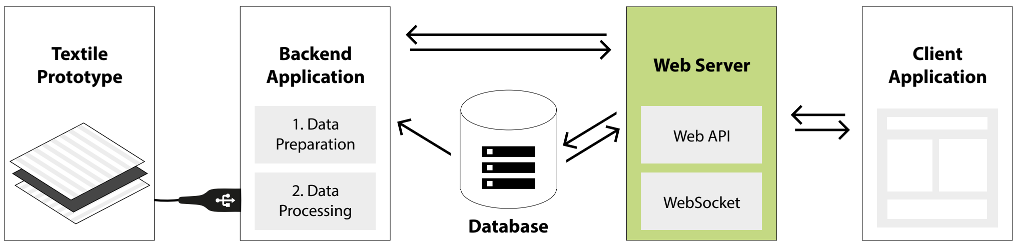


Figure 4.18: The web server handles the communication between the client and the backend application.

4.6.1 Web API

The web API was developed so the backend application can be managed from a user interface. While not every detail of the backend can be changed via the web API the most important tasks and actions that are reasonable to be executed via an user interface are supported. If a user feels limited by the functional range of the provided web API it can easily be extended.

One main functionality of the web API is handling which filters are applied on the data, defining which gestures will be recognized and which actions will be triggered when a specific gesture is identified. These actions depend heavily on create, read, update and

[16]https://www.npmjs.com/package/axios
[17]https://expressjs.com/

Table 4.1: This table provides an overview about the provides REST API calls to create, read, update and delete filters, gestures and action triggers.

Method	Filters	Gestures	Triggers	Description
`GET` `POST`	/filter	/gesture	/trigger	read all elements create a new item

`GET`
`PUT`
`DELETE`
	/filter/:id	/gesture/:id	/trigger/:id	read element by id
				update specified element
				delete item

Table 4.2: The API provides endpoints to get a list of all filter types, reset and reorder filters and retrain the SVM classifier. Additionally, the data capturing can be triggered and stopped. Also information about the serial port connections of the device can be handled using the provided endpoints.

Method	Route	Description
`GET`	/filter/types	list all filter types
`PUT`	/filter/reset	reset the filter pipeline
`PUT`	/filter/reorder	change the ordering of the filter in the filter pipeline with start and end index
`POST`	/gesture/train	train the SVM classifier
`GET`	/data/ports	list all serial port connections of the current device (used for debugging)
`POST`	/data/device/:productId	change product id of hardware device
	/data/capture/start/:gestureId	start capturing data
	/data/capture/stop	stop data capturing
	/data/capture/:gestureId	capture a single frame

delete (CRUD) actions of the particular entity and were therefore implemented using a REST architectural style (see Table 4.1). In addition to this RESTful API there are a view more specific endpoints available for the filter and the gesture entity, respectively. They allow to retrieve a list of all existing filter types, to reset and reorder filters and to train a new SVM classifier or retrain the existing SVM model. Another important task available via the web API is controlling the data capturing. Existing endpoints enable the user to start and stop capturing data for the machine learning algorithm used for the gesture recognition. Additionally, there is the possibility to retrieve information about all devices currently connected to the used computer. This feature is useful for debugging if no data can be retrieved and is therefore part of the data controller. Also a route for changing the product id of the hardware, which is used to identify the correct COM port for receiving data input, exists. The exact routes are summarized in Table 4.2.

4.6.2 Streaming of Raw Data and Recognized Gestures

The WebSocket server is implemented using the *ws*[18] library and can also be accessed on port 9000. Since the *WebSocket Protocol* is used instead of the *Hypertext Transfer Protocol* (HTTP) this must be declared in the URL by using `ws://` as a substitute for `http://`. The server only allows clients to connect and receive messages and does not listen to any messages sent by the client. Three different messages types are sent over the WebSocket connection: the raw sensor data, gestures recognized by implemented heuristic functions and gestures identified by the SVM classifier. While the messages are structured differently, they all have a specified type property so they can be easily distinguished by the client application.

Sensor Data Message

Messages of type `raw` are sent when there is new data from the serial input, which has already been analyzed and filtered. The message has the following format:

```
{
    type: 'raw',
    timestamp: ...,                        // JavaScript Date object
    serialData: [...],
}
```

Message for a Gesture Recognized by a Heuristic

If there is an implemented heuristic which detects a specific gesture, the following message is sent including the name of the gesture and different custom properties:

```
{
    type: 'heuristic',
    name: ...,
    gesture: {
        ...                                // properties of the gesture
    },
}
```

The properties that are sent with the gesture are defined by the particular implementation of the heuristic function, respectively.

SVM classified gestures

When a gesture is recognized by the SVM algorithm, the following message is sent which gives information about the recognized gesture:

```
{
    type: 'gesture',
    name: ...,
    label: ...,
    probability: ...,                      // predicted probability in %
}
```

The message specifies the type `gesture` and informs about the probability that the gesture was classified correctly.

[18]https://www.npmjs.com/package/ws

Chapter 5

User Interface

For the most important configurations of the backend application a graphical web-based user interface has been designed and implemented. Thus, also novice users with little or no programming experience are able to prototype simple use cases, train gestures and trigger actions when they are recognized by using the predefined interactions with the backend application.

The user interface (UI) is divided in five components: *Device Configuration*, *Filtering*, *Gesture Recognition*, SVM *Classifiers* and *Action Triggers*. They are all shown in the order they should be configured in (see Figure 5.1).

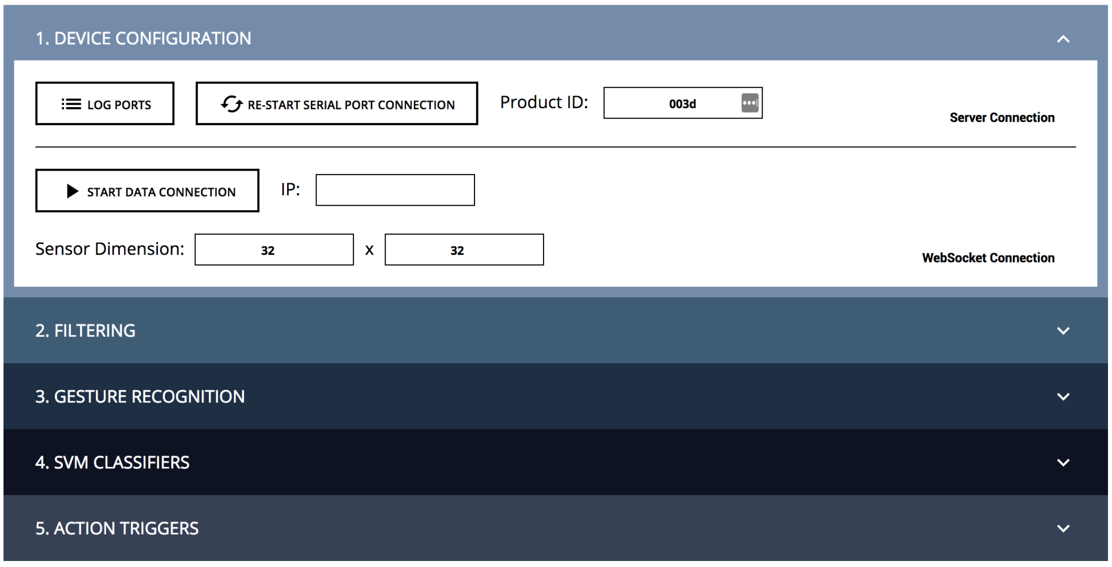


Figure 5.1: The UI contains five steps. The first section contains basic settings concerning the configuration of the hardware device.

5.1 Device Configuration

The first step is the configuration of settings concerning the hardware prototype, which is represented by the first section of the user interface shown in Figure 5.1. The application needs to know how to identify the right USB port for reading the input data from a smart fabric device. It is built to identify the correct port by scanning all available serial ports for a specified device id which can be configured in the first section of the UI. If the device id is not known to the user, there is the possibility to log all available ports to the developer console of the browser to identify the correct port manually and look for the device id there.

Furthermore, the IP address for the WebSocket connection can be configured in case the UI is not opened on the same device that the server is running on. The connection to the web socket server can be started and stopped. Only if the connection is open, raw data for the visualization and information about recognized gestures is transmitted. For the transmitted data the number of rows and cols can be configured which impacts the visualization discussed in the next section.

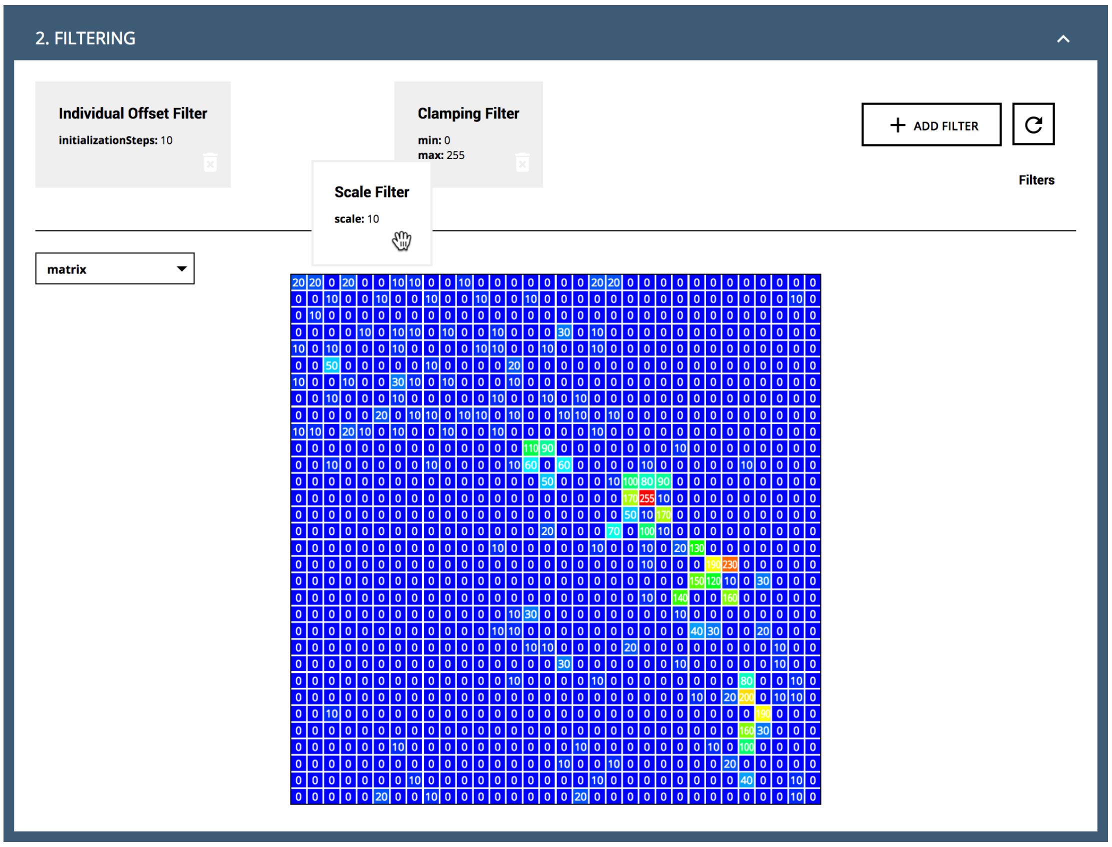
5.2 Filtering

After the device has been configured, the next action is to define the filtering in the second section of the user interface (see Figure 5.2). By using the ADD FILTER button various different filters (see Section 4.3.1) can be added via a dialog that is shown in Figure 5.3. The information about which options are necessary for a certain filter type and which input fields have to be displayed, is stored in the database. Therefore, if a developer implements a custom filter type which should be available in the user interface, a data base entry has to be made describing the interface for the dialog shown in Figure 5.3. Once multiple filters are added, the order of the filters can be changed via a simple drag and drop gesture. This is crucial, because some filters are not commutative and therefore having a simple solution to change the order of the applied filters is fundamental.

The second important part of the filtering section is the visualization of the data. If there is an open web socket connection, incoming data is displayed in matrix form on a canvas element. The number of rows and columns of the visualization depends on the settings configured in the *Device Configuration* section of the UI. The color of the single entries of the data matrix depends on their value. The values 0 to 255 are interpolated linearly between the hue values 240 (which corresponds to blue) and 0 (red) using the HSL (Hue, Saturation, Lightness) color model.

5.3 Gesture Recognition

In the third section of the user interface recognized gestures specified via an algorithmic description of a heuristic are shown (see Figure 5.4). The gray area on the left contains the name of the recognized gesture. On the right side more information about the recognized gesture, like directions or strength of the applied force, is shown. Which information is shown for a specific gesture is defined when implementing the heuristic.



Individual Offset Filter
initializationSteps: 10

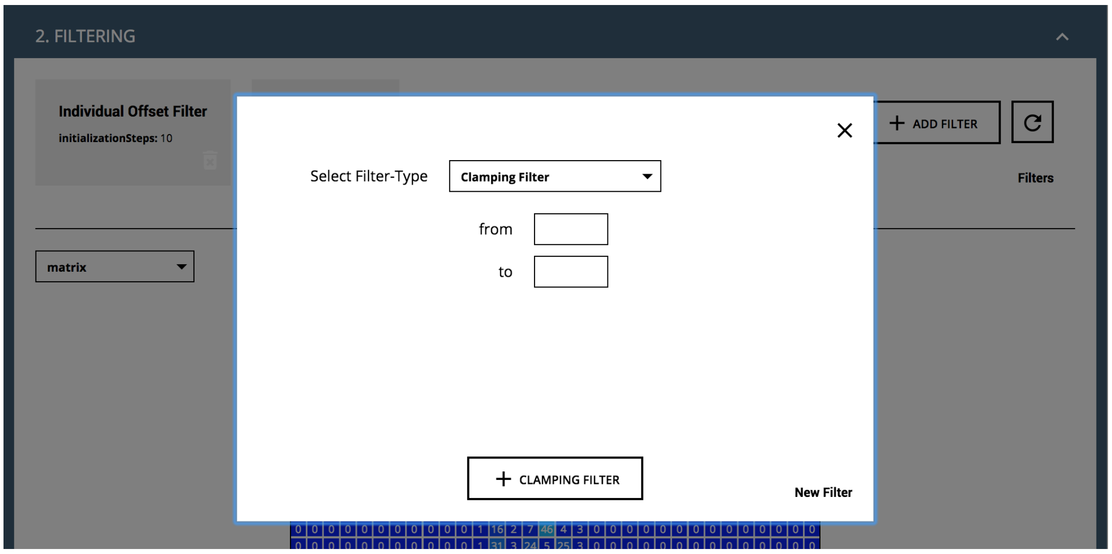
Clamping Filter
min: 0
max: 255

Scale Filter
scale: 10

matrix

20	20	0	20	0	0	10	10	0	0	10	0	0	0	0	0	0	0	20	20	0	0	0	0	0	0	0	0	0	0	0	0
0	0	10	0	0	10	0	0	10	0	0	10	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	10	0	10	10	0	10	0	0	10	0	0	0	30	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	10	0	0	0	10	0	0	0	0	10	10	0	0	10	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	50	0	0	0	0	0	10	0	0	0	0	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	10	0	0	30	10	0	10	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	10	0	0	0	10	0	0	0	0	0	10	0	0	10	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	20	0	10	10	0	10	10	0	10	0	0	10	10	0	10	0	0	0	0	0	0	0	0	0	0	0	0
10	10	0	20	10	0	10	0	0	10	0	0	10	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	90	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0
0	0	10	0	0	0	0	0	10	0	0	0	0	10	60	0	60	0	0	0	0	10	0	0	0	0	0	10	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	10	100	80	90	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	170	255	10	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	10	170	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20	0	0	0	70	0	100	10	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	10	0	0	10	0	20	130	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	190	230	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	150	120	10	0	30	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	140	0	0	160	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	10	30	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	40	30	0	0	20	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	10	0	0	0	0	20	0	0	0	0	0	0	0	0	10	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30	0	0	0	0	0	0	10	0	0	0	0	0	10	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	10	0	0	0	0	0	0	0	0	80	0	0	10	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	20	200	0	10	10	0
0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	190	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	160	30	0	0	0
0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	10	0	100	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	10	0	0	0	0	0	0	20	0	0	0	0	0
0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	40	0	0	10	0
0	0	0	0	0	20	0	0	10	0	0	0	0	0	0	0	0	20	0	0	0	0	0	0	0	0	0	0	0	0	10	0

Figure 5.2: The second section allows to define new filters, delete existing filters or reorder the filters contained in the filter pipeline with a simple drag and drop mechanism. It also contains the visualization of the sensor data.



Individual Offset Filter
initializationSteps: 10

matrix

Select Filter-Type: Clamping Filter

from

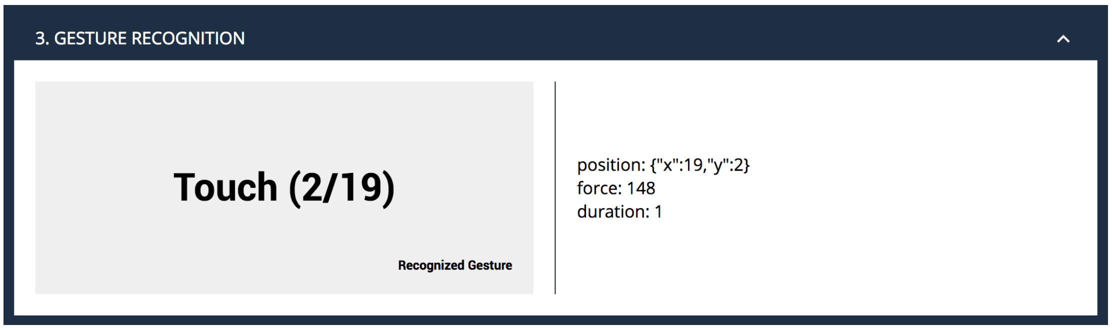
to

+ CLAMPING FILTER

New Filter

0	0	0	0	0	0	0	0	0	0	0	1	16	2	7	46	4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	31	3	24	5	25	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.3: A modal containing a form allows to add new filters to the filter pipeline. The form fields change according to the selected filter type.



position: {"x":19,"y":2}
force: 148
duration: 1

Figure 5.4: A touch gesture was recognized with a value of 148.

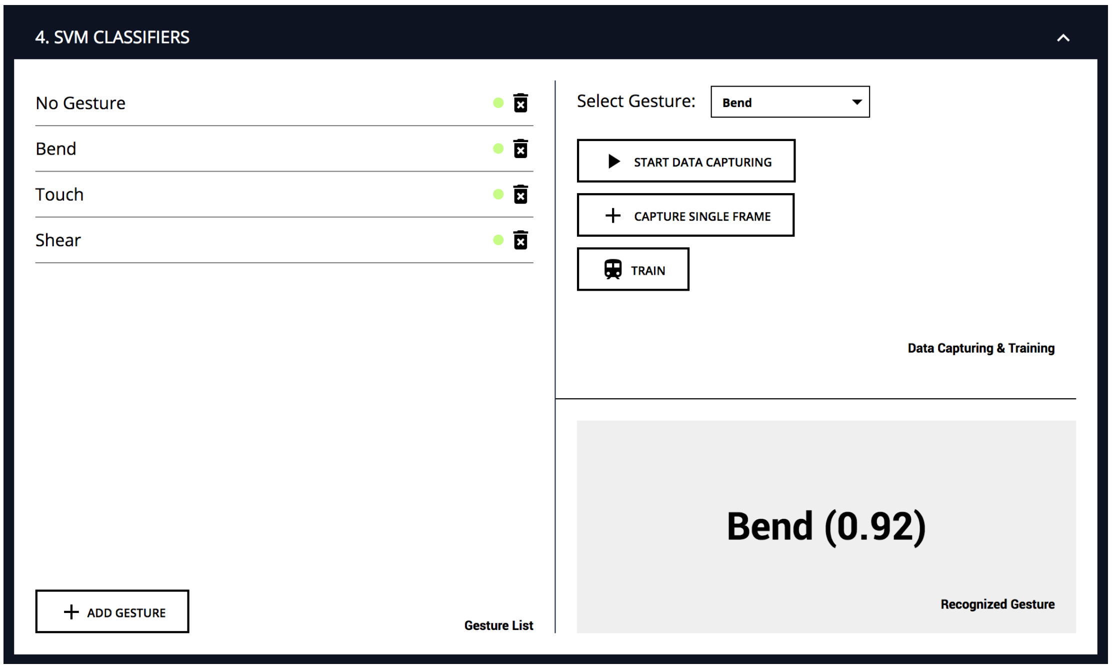
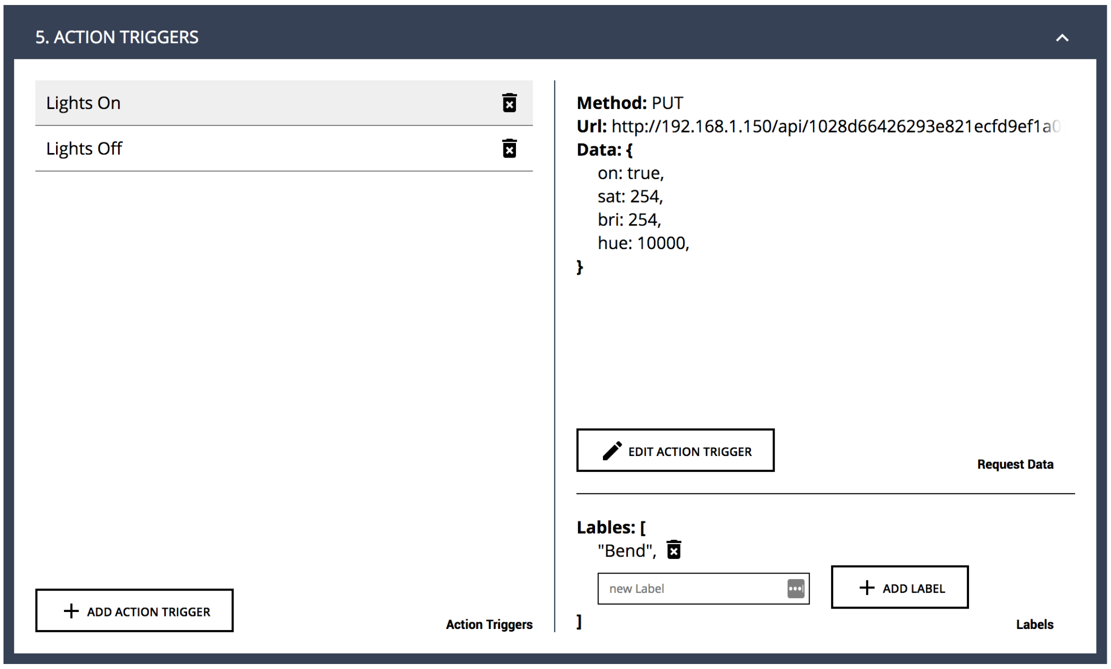


Figure 5.5: Section 4 is concerned with the definition of gestures and training of the machine learning algorithm. Gestures recognized by the SVM classifier are shown here.

5.4 SVM Classifiers

Gestures recognized be the machine learning algorithm can be configured and visualized in Section 4 of the UI (see Figure 5.5). The left side contains a list of all stored gestures (loaded from the database). Again it is possible to add new gestures by opening a dialog comparable to the ADD FILTER dialog shown in Figure 5.3 or delete a gesture by clicking on the trash can icon next to the gesture name. The small green or grey circle next to the gesture indicates if the gesture has already been trained. If the circle is green the SVM classifier is ready to recognize the gesture.

The right side again contains a grey area which shows the name of the last recognized gesture, like in the previous section. Furthermore, it contains functionality to capture



Method: PUT
Url: http://192.168.1.150/api/1028d66426293e821ecfd9ef1a
Data: {
    on: true,
    sat: 254,
    bri: 254,
    hue: 10000,
}

Lables: [
    "Bend",
]

Figure 5.6: Section 5 is concerned with the definition of action triggers.

data and train the machine learning model. To capture data, the user first has to select one of the defined gestures to specify for which gesture the data should be captured. Then the gesture needs to be performed on the prototype and the button START DATA CAPTURING is clicked. Only after clicking STOP DATA CAPTURING the gesture which is performed on the smart textile should be released. If only one single frame of data is needed, there is also the possibility to capture a single frame by using the corresponding button. When all the necessary data has been captured, the TRAIN button can be used to re-train the SVM classifier. After the training process, ideally all defined gestures should be marked with a green circle. If this is not the case the most likely reason therefore is, that there is was no data captured for some of the gestures and they therefore could not be trained.

5.5 Action Triggers

The last section of the user interface (see Figure 5.6) is for defining which actions should be triggered once a specified gesture has been recognized by either a programmed heuristic or the machine learning model. The section is structured similar to the one for specifying the gestures for the SVM classifier. On the left side there is a list of all defined triggers and the possibility to add new triggers or delete them. As described in Section 4.5, an action can be an arbitrary HTTP request to any public or local API. When an action trigger is selected from the list, all the information about the trigger (the HTTP method, the URL and the data for the JSON body of the request) is listed on the right side. In the right bottom area the user has the possibility to link the specified action to a gesture, that should trigger the action when it is recognized.

For specifying the JSON body that will be sent with a triggered request, a special syntax has been developed to empower the user to not only use static values in the request body, but also to dynamically map information from the corresponding recognized gesture or predefined values from the database to specified entries in the request body. The developed syntax allows to specify placeholders for various values of interest. When the action is triggered the strings are parsed and replaced with the desired values.

Gesture Parameters: When a developer implements a custom algorithm to recognize gestures, custom properties can be specified that are stored in the object of the recognized gesture. In the recognized *Touch* gesture from the example shown in Figure 5.4 there are three custom properties defined: *position*, *force* and *duration*. These parameters of the recognized gestures can be accessed in the request body specification by using the `$` symbol, e.g. `$position$`, `$force$` or `$duration$`.

Variables: Another possibility is to map properties of the request body to predefined values stored in the database. This can be accomplished with the use of the `%` symbol, e.g. `%brightness%` or `%hue%`, to accessed values for *brightness* and *hue* that are stored in the list of trigger variables in the database.

Iteration: For some use cases it might be necessary to iterate through a list of predefined values. An example could be, that five colors are defined and every time a specified gesture is performed the light should switch to the next defined color. For this use case an additional syntax was developed: `%colors[%colorIndex%++]%`. Using this syntax a defined array of multiple `colors` and the current index `colorIndex` are read from the database. The value of `colors` at index `colorIndex` is then used for the request body. Additionally, the value for `colorIndex` is updated so the next time the request is triggered, the next value will be used. The direction, which is used for the iteration is specified by using `++` or `--`. When the end or the beginning of the array is reached, the iteration will start again with the first or last entry, respectively.

Chapter 6

A Dynamic Quadtree Approach for Reading Sensor Data

In addition to the developed modular system for handling and interpreting sensor data coming from smart fabrics and the user interface for facilitating rapid prototyping, also a new method how the sensor data can be sampled more efficiently was developed and evaluated. This chapter describes the basic idea and the motivation for this approach. The evaluation is discussed in the subsequent chapter. The approach provides an alternative, possibly faster method to read large-scale textile sensors. However, this idea does not have to be implemented in order to be able to use the developed prototyping framework with custom prototypes.

6.1 Motivation

The proposed approach is motivated by research on industrially woven or knitted textiles, which can be produced at large-scale and with high resolution. If the sensor data of large textiles is read by iterating over all intersections of conducive yarns (which are the measurable points of the fabric), the time complexity is $O(n^2)$. Therefore, it can be concluded that for large-scale or high resolution textile sensors, the performance of sampling all sensors individually, rapidly decreases with increasing size and resolution of the sensor. Table 6.1 shows an example of the sampling performance of smart fabrics with different amounts of sensors. The example is based on an industrial manufactured textile sensor with a resolution of 32×32 sensible values. The table shows, that for large-scale and high resolution prototypes, the time needed for sampling all sensor values is not sufficient. Furthermore, the number of bytes that need to be transmitted to the web platform is enormous for big sensors. The amount of transmitted data can be decreased by using a lossless compression algorithm like run-length encoding. However, data compression would not decrease the time of the sampling process. Therefore, an optimized way of sampling the data, which in the best case also results in less data that needs to be transmitted, is needed. One idea to achieve this is using a quadtree structure, a technique often used for various purposes in different fields including image processing [2, 4, 11, 17], computer graphics [6, 15, 19, 20, 32] and robotics [8, 30].

Table 6.1: For large prototypes with high resolutions the time for sampling all sensors individually and the number of transmitted bytes increases rapidly.

# of Sensors	Sample Time	Method	Frames/Second	Bytes
1	$\approx 28\,\mu$s	measured	35,714	7
100	2.8 ms	calculated	357	106
$1,000$	28 ms	calculated	36	1,006
$1,024$	27.54 ms	measured	36	1,030
$10,000$	280 ms	calculated	3.6	10,006
$100,000$	2.8 s	calculated	0.4	100,006

6.2 Quadtrees

Before getting into more detail about the proposed quadtree approach, this section establishes the basic theory about quadtrees and give a short overview about the use of quadtrees in various research areas.

6.2.1 Definition

Samet gives a good overview about quadtrees in [24]. He defines quadtrees as "hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space" and distinguishes between two types of quadtrees: *region quadtrees* and *point quadtrees*. A comprehensible description of region and point quadtrees can also be found in [1] by *Aluru*.

6.2.2 Region Quadtree

The implemented approach for reading smart textile sensors is categorized as a *region quadtree*, which is therefore described briefly by using the example given by *Samet* in [24]. The focus of the implemented region quadtree lies on organizing the content of the given area. An example of a region quadtree structure is displayed in Figure 6.1. The raw data, which in the example of *Samet* is a binary array describing a black and white image, is shown in Figure 6.1 (a). In the implementation for the smart fabric, this represents the sampled data from the textile sensor. The data is divided in four equally sized parts. If the thereby created regions do not consist only of 1s or 0s, the region is again divided in four quarters. This procedure is continued until all created regions consist entirely of one value. The result is shown in Figure 6.1 (b) and the composed tree structure is shown in Figure 6.1 (c).

6.2.3 Quadtrees in Different Research Areas

To establish some context, some examples for the use of quadtrees in different fields are outlined. The examples provide an idea about typical use cases for quadtrees but do not attempt to illustrate a complete list of all applications where quadtrees might be used.

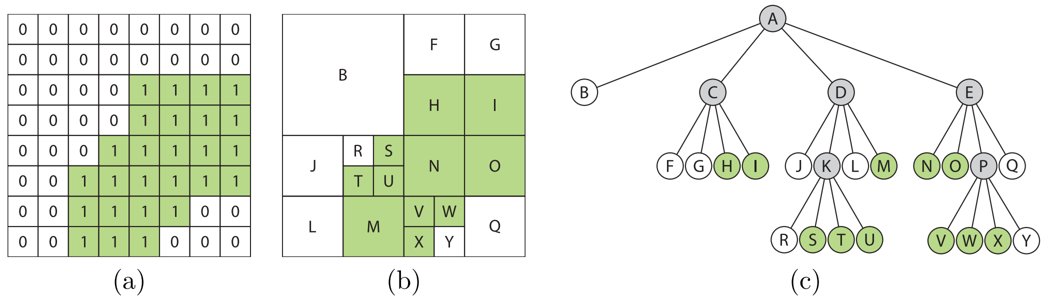


Figure 6.1: This figure shows an example for a *region quadtree* which was re-illustrated from [24]. The raw data (a) and the structured data (b) as well as the corresponding tree structure (c) are shown.

Computer Graphics: One common use case of quadtrees in the field of computer graphics is for example the rendering of terrains in different levels-of-detail (LOD) depending on the current view [15, 19, 20]. More recent research on the topic [6, 32] shows, that the use of quadtrees in terrain LOD algorithms is still not outdated.

Robotics: Another use case for quadtrees in the area of robotics is pathfinding [8, 30]. In pathfinding, quadtree structures can be used to describe the world and its obstacles in a more compact way, because areas without any obstruction can be combined and treated as a single entry [8].

Image Processing: Also in image processing quadtrees play a viable role. Already in the 1980s, quadtrees have been used for shape analysis and shape matching [2]. Use cases from recent years include image segmentation and image comparison [4, 11, 17]. Image segmentation can for example be useful for image annotation [4] or as a preprocessing step to image processing to specify regions of particular interest [17]. An example for image comparison is the work from Kirichek and Kurai, who use quadtrees to segment two similar images and then calculated a third images based on the differences [11].

6.3 Method Description

The described novel sampling technique focuses on improving the sensor sampling speed of smart fabrics prototypes and reducing the length of the transmitted data at the same time, by using the well established quadtree algorithm in a new context. The basic idea is to use a region quadtree structure to divide the textile sensor area into regions, instead of sampling all single sensor values individually. An example is shown in Figure 6.2. Instead of sampling all the values (see Figure 6.2 (a)), the sensor is divided in four equally big areas which are sampled as one single sensor, respectively (see Figure 6.2 (b)). If there are values above a certain configurable threshold t, the corresponding area is again divided in four smaller parts (see Figure 6.2 (c)) and this step is repeated until the maximum resolution is reached (see Figure 6.2 (d)). All regions with a value below t are not sampled in more detail. The subsequent sections first describe how the tree structure is built and discuss the conditions for sampling a specific region with more

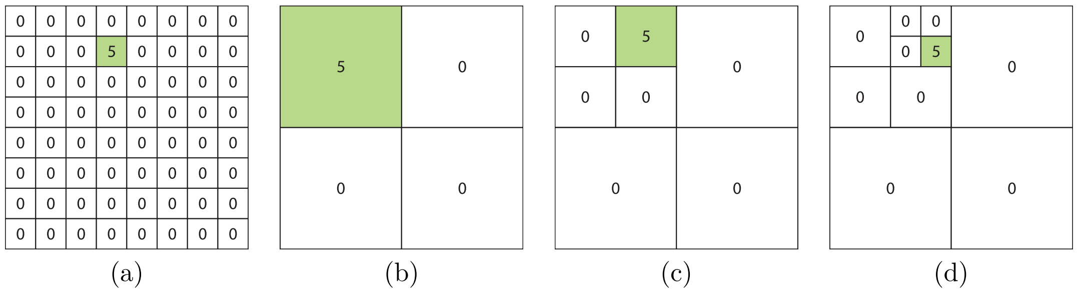


Figure 6.2: Instead of sampling all single sensors (a), the textile is divided in 4 parts (b) and only areas with a value above the threshold t are sampled in more detail (c). This step is repeated until the maximum resolution is reached (d).

detail. Second, the serialization of the tree is outlined and the resulting data structure is illustrated. Finally, reading different sensor values, sampling sensor regions and finding an appropriate threshold t is explained in detail.

6.4 Tree Structure

The focus of this section is the algorithm used to build the quadtree structure and the conditions for deciding if a node should be further extended or remain a leaf node. The serialization of the quadtree, once it is complete, is discussed in the next section.

6.4.1 Tree Composition

The first step to build the quadtree, is to sample the value for the whole sensor area and use this value to initialize the root node of the tree structure. Next, the conditions for expanding the current node and sampling the sensor region with more detail are checked, and if necessary the node is expanded. Expanding a node means that the area corresponding to the particular node is divided in 4 parts of equal size. Four child nodes are added to the current node which correspond to one of these four new regions, respectively. This step is repeated recursively until the tree structure is completed, like shown in Figure 6.2. The approach is described algorithmically in Algorithm 6.1.

6.4.2 Conditions to Expand a Node

If a current node should remain a leaf node or needs to be expanded, to closer examine the corresponding sensor region, depends on two conditions. First, it is important to check if the sensor region j associated with the node is bigger than one single sensor. If the j consists of only one sensor, the maximum depth of the tree is reached and the corresponding node cannot be expanded further. If only this condition is applied when generating the quadtree structure, the tree would already be built correctly. However, all nodes would be expanded as often as possible until the maximum depth is reached, which would result in sampling the whole sensor in as much detail as possible. This is no improvement over sampling all sensors and would even worsen performance. Thus, a second condition is necessary.

Algorithm 6.1: This algorithm describes how the quadtree is built.

```
1:  global variables
2:      $data$, value of the current node
3:      $children$, child nodes of current node
4:  end global variables

5:  Expand($P$)
    Samples the value for the current node and checks if the node must be expanded to
    sample the values in more detail based on the condition $P$.
6:      $data \leftarrow$ SampleSensorRegion( )        ▷ read the value for the current node
7:      $children \leftarrow ( )$
8:      if $P$ is valid then
9:          for $i \leftarrow 0, \ldots, 3$ do                  ▷ add 4 child nodes
10:             $child \leftarrow \langle nil, ( ) \rangle$
11:             $children \leftarrow children \cup (child)$
12:             Expand(P)                              ▷ expand the child node
13:         end for
14:     end if
15: end
```

The second condition does not check if it is possible to divide a sensor region j, but checks if the expansion of a node is necessary. The goal is to distinguish between areas where no pressure is applied and regions or single sensors with touch input or deformation, because these active areas should be sampled with maximum resolution in order not to loose information. This is done by applying a specified threshold t on the sensor value $D(j)$. In case the sampled value $D(j)$ for the current node is above t there is activity in the current sensor region j and the sensor should be sampled with a higher resolution. If $D(j) < t$, there is no detailed sampling necessary because no pressure is applied anywhere on the corresponding region.

6.5 Serialization

When the sampling of the smart textile sensor is finished and the quadtree is complete, the values of all leaf nodes are serialized before the data is transmitted from the microcontroller of the smart textile prototype to the connected computer. The serialization is done starting at the root node and traversing the tree in depth-first order. The depth and the value of all leaf nodes of the tree are saved to a data array that is transmitted. Additional the number of nodes added to the transmission data is counted to later calculate the length of the data transmission. The algorithm used for the serialization is shown in Algorithm 6.2. After the serialization of all nodes is complete, the 6 header bytes are added at the beginning of the data array (see Section 4.2).

The described serialization results in a data structure where every value is submitted in 2 bytes. The first bytes signifies the depth of the value in the quadtree structure and the second byte specifies the value itself. In case all sensor have to be sampled with the maximum depth, the resulting data that needs to be transmitted, is twice as long

Algorithm 6.2: The quadtree is traversed and the length and the value of all leaf nodes are stored in a list.

```
 1: global variables
 2:     $nodeCounter \leftarrow 0$
 3:     $data \leftarrow (\,)$
 4: end global variables
 5: RetrieveNodeValue($node$)
    Retrieve the depth and the data value of the given node.
 6:     if $node$ is a leaf node then
 7:         $nodeCounter \leftarrow nodeCounter + 1$
 8:         $data \leftarrow data \cup$ (depth of $node$)
 9:         $data \leftarrow data \cup$ (value of $node$)
10:     else
11:         for all $childNode \in$ children of $node$ do
12:             RetrieveNodeValue($childNode$)
13:         end for
14:     end if
15: end
```

using the quadtree structure than with the regular sampling of all single sensors. On the other hand, in case only few sensors differ from 0, less data needs to be transmitted because big regions with the same value can be combined. An example of an 8×8 sensor, the corresponding tree structure and the transmitted data with and without using the proposed quadtree approach, is shown in Figure 6.3.

6.6 Reading Sensor Values and Finding the Optimal Threshold

This sections gives a detailed description on how the sampling of the single sensors works and how multiple sensors can be combined to a sensor region j. Furthermore, two different approaches for evaluating the optimum threshold t are discussed.

6.6.1 Reading of Sensor Values

Every sensible point of the smart textile has to be the crosspoint of two lines of conductive yarns. To read a specific sensor value of the prototype, one of the corresponding lines has to be connected to reference voltage and the other line has to be connected to the analog digital converter (ADC), while all other sensor lines are connected to ground potential instead, as shown in Figure 6.4 (a). The same method can also be used to read a region of multiple sensors at once by connecting multiple conductive lines to reference voltage and their counterpart to the ADC (see Figure 6.4 (c)).

The `analogRead()` method from the Arduino API is used to sample the value $D(i)$ of a single sensor i or $D(j)$ for the currently configured sensor region j. It takes the reference voltage of 3.3 Volts of the Arduino Due board and divides it by the number of possible values which, with the default 10-bit analog to digital converter, is $2^{10} = 1024$

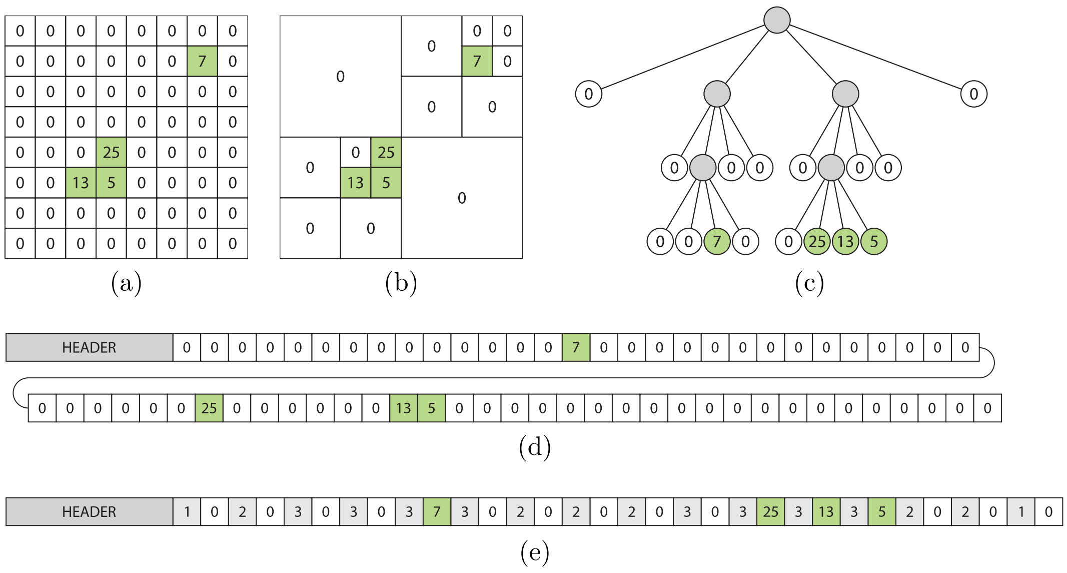


Figure 6.3: An example of possible sensor data (a) is given and the corresponding quadtree structure is shown in (b) and (c). The serialized data, that is transmitted to the connected computer and used by the described prototyping application, is demonstrated and compared between the transmission of the raw sensor data (d) and the transmission resulting from the quadtree algorithm (e).

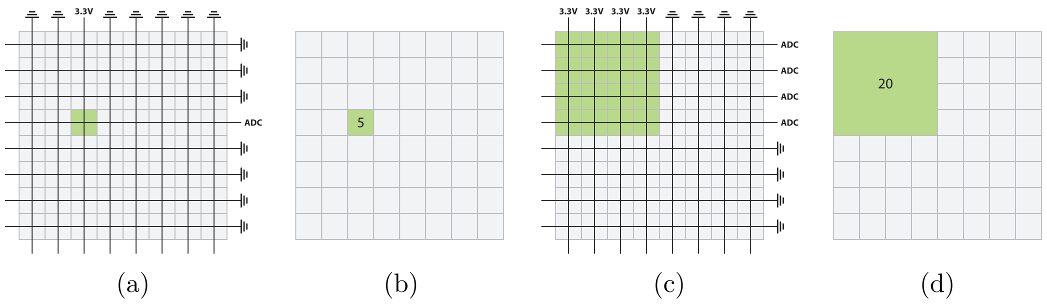


Figure 6.4: By connecting one (a) or multiple (c) conductive lines to reference voltage and the corresponding lines on the other side of the crosspoint to the ADC, while all other lines are connected to ground potential, one single sensor (b) or a sensor region (d) can be read.

[34, 35]. The resulting values range from 0 to 1023. For the implemented example the method `analogReadResolution()` is utilized to use an 8-bit analog to digital converter instead. Therefore, the results are values between 0 and 255, where a difference of 1 between two values corresponds to a change in voltage of 0.0129 Volts (12.9 mV). Thus, it can be concluded that the sampled value $D(i)$ or $D(j)$ is a direct indicator for the sampled voltage

$$U_v = D(i) \cdot 12.9\,\text{mV}, \tag{6.1}$$

which can be applied for $D(j)$ as well. However, the measured voltage is not actually

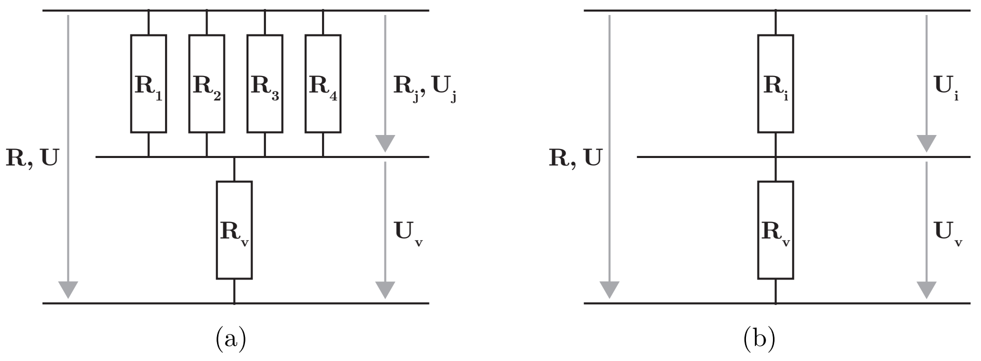


Figure 6.5: While all resistors (crosspoints of conductive yarns) of a specified sensor region $R_j = (R_1, R_2, \ldots, R_n)$ are connected in parallel, R_j is connected serially to a pull-down resistor $R_v = 5.65\,\mathrm{k\Omega}$ (a). The equivalent for a single sensor is shown in (b).

the voltage at the specified crosspoint(s), but the voltage at a pull-down resistor R_v, which is connected serially to the sensor values like shown in Figure 6.5.

For a single resistor (see Figure 6.5 (b)) this means if R_i is high (which is the default state if the sensor is not touched) the voltage U_i is also high which results in U_v being low because

$$U_v = U - U_i, \tag{6.2}$$

where U does not change. Therefore, the value $D(i)$ returned by the `analogRead()` method is also low, which is the expected behaviour when no sensor is touched. If pressure is applied on a crosspoint of two conductive lines, the corresponding resistor R_i decreases, which results in a higher voltage U_v, which means a higher value $D(i)$. The same logic can be applied on sensor regions j and the corresponding value $D(j)$.

6.6.2 Hardware Requirements

To make it possible to either read single sensors or sensor regions of different sizes, it is necessary to have a hardware that supports reading dynamic resolutions of the sensor. One suggested approach is to use analog matrix switches (ADG1438BRUZ) to control the single conductive lines. One analog matrix switch has 8 analog switches which means it can connect 8 conductive lines respectively or multiple lines at once. Multiple matrix switches can be daisy-chained to control bigger matrices. It is possible to connect a single line to either reference voltage or ground potential. Therefore, two matrix switches are needed for every 8 lines, one that handles the reference voltage and one for ground potential. An example hardware, how this could be implemented for a 32×32 sensor is shown in Figure 6.6. This setup allows to dynamically change the read resolution of the sensor, because the software can define how many crosspoints should be read at once. This is an important prerequisite for building the quadtree.

6.6.3 Choosing the Optimum Threshold

According to the initial sensor values when no pressure is applied, the threshold t must be specified. The threshold t needs to be chosen carefully since it is the only criteria

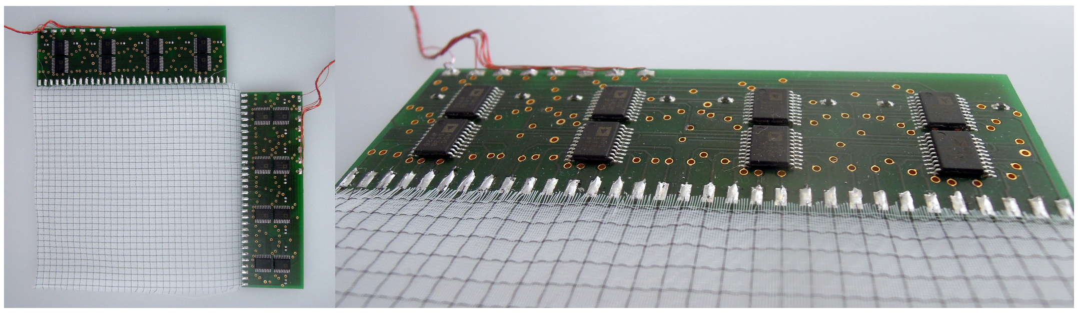
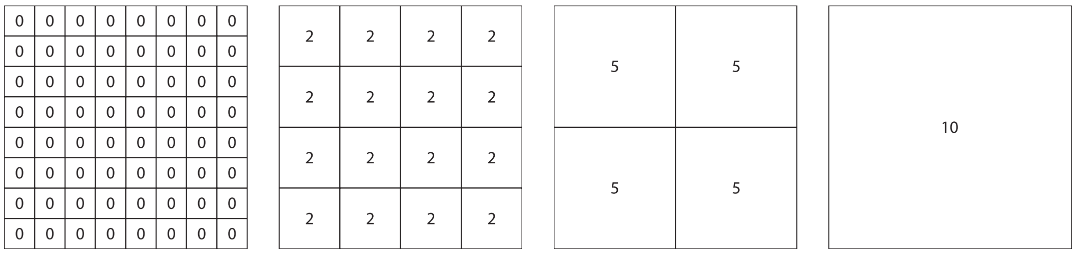


Figure 6.6: This is a prototype with an industrial woven smart fabric sensor with 32×32 conductive yarns (black) which are connected to two boards with 8 analog matrix switches, respectively.



(a) 8×8 (b) 4×4 (c) 2×2 (d) 1×1

Figure 6.7: The sampled value increases with the number of combined sensors.

influencing the performance of the quadtree. As discussed in Section 6.6.1 the sampled sensor value is 0 when no pressure is applied on the smart textile. Applying pressure on the fabric results in higher values in the corresponding area. Therefore, a suitable value for t would be 0 or close to 0, depending on how much noise should be tolerated before considering a value active input. However, considering the single resistors $R_i \in \{R_1, R_2, \ldots, R_n\}$ of a sensor region R_j are all connected in parallel, R_j is smaller than the smallest resistor R_i because

$$\frac{1}{R_j} = \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_n}. \tag{6.3}$$

Therefore, if multiple sensors are read at once, it is possible that their value $D(j) > 0$, even if all sensors of the region would be 0 if they were sampled individually. An example is shown in Figure 6.7. While all single sensors have a value of 0, the values increase with the size of the sampled region. This is a problem because a fixed threshold $t = 0$ does not work anymore. Thus, another solution is needed to decide if the resolution of the region should be increased. One possibility is to approximate the values of the single sensors of a specified region and to apply t on the single sensor value instead. Another solution would be a dynamic threshold that is calculated based on the depth of the sampled sensor region. Both options are discussed, implemented and evaluated.

Approximation of Single Sensor Values

The first approach is to approximate the values of all single sensors. First, the total resistance R_j of the specified sensor region j has to be determined. According to Ohm's law the relation between the voltage R, the current I and a resistor R is

$$U = I \cdot R. \tag{6.4}$$

This means that

$$I = \frac{U}{R}, \tag{6.5}$$

where U has the constant value of 3.3 V. The total resistance R equals the sum of R_j and R_v because they are connected serially, i.e. $R = R_j + R_v$. Thus, it can be concluded that

$$I = \frac{3.3}{R_j + R_v}. \tag{6.6}$$

With this information Ohm's law can be applied again, i.e.

$$U_j = \frac{3.3}{R_j + R_v} \cdot R_j, \tag{6.7}$$

which means

$$R_j = \frac{U_j \cdot R_v}{3.3 - U_j}, \tag{6.8}$$

where U_j can be calculated from the read sensor value $D(j)$ and $R_v = 5.65\,\mathrm{k\Omega}$. Once R_j is known the next step is to calculate one single resistor R_i under the assumption all resistors R_i of the current sensor region are the same. While this is not necessarily true, it is the best possible approximation since no further information about the ratio between the resistors is given. Nevertheless, this is a valid solution because if the values derived by this calculation are close to 0 (defined by the threshold t), there is no force input in this region and small differences in the resistors do not matter. If the resulting values are above t, the discussed approach samples the corresponding region with a higher resolution, which means the values are not directly used. Therefore, averaging the values is valid and one single resistor

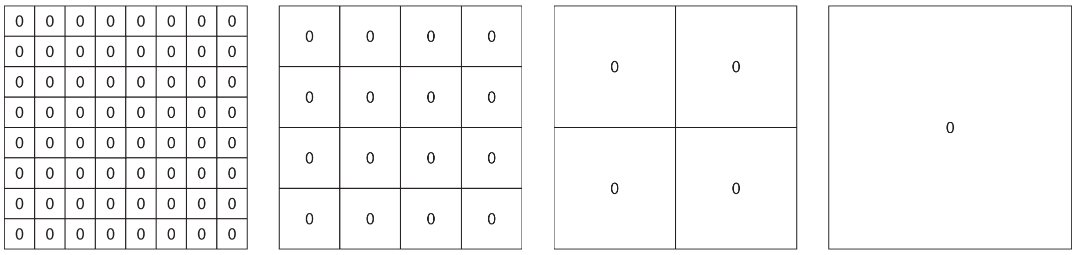
$$R_i = N \cdot R_j, \tag{6.9}$$

according to the behaviour of resistors connected in parallel which is formalized in Equation 6.3, where N is the number of sensible crosspoints R_i in the selected sensor region. The last step is to divide the total voltage U of 3.3 V between the two resistors R_i and $R_v = 5.65\,\mathrm{k\Omega}$ which are connected serially (see Figure 6.5 (b)) to calculate

$$U_v = U \cdot \frac{R_v}{R_i + R_v}, \tag{6.10}$$

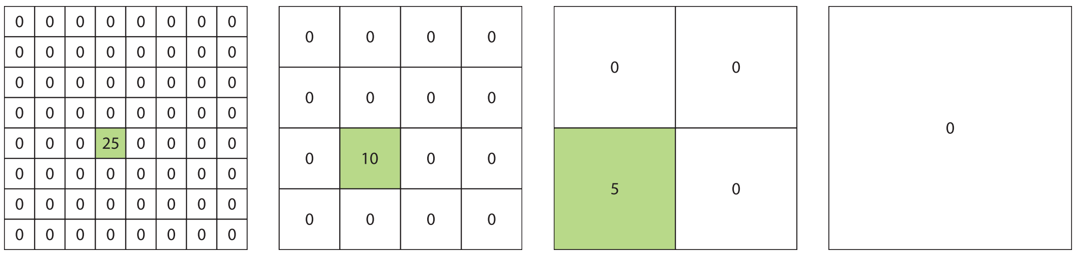
which is needed to calculate the new sensor value

$$D(i) = \frac{U_v}{12.9\,\mathrm{mV}}. \tag{6.11}$$



(a) 8×8 (b) 4×4 (c) 2×2 (d) 1×1

Figure 6.8: If the values of the single sensors are estimated and used as the value for the whole region, the result for a sensor where no pressue is applied is always 0 independet of the size of the sampled region.



(a) 8×8 (b) 4×4 (c) 2×2 (d) 1×1

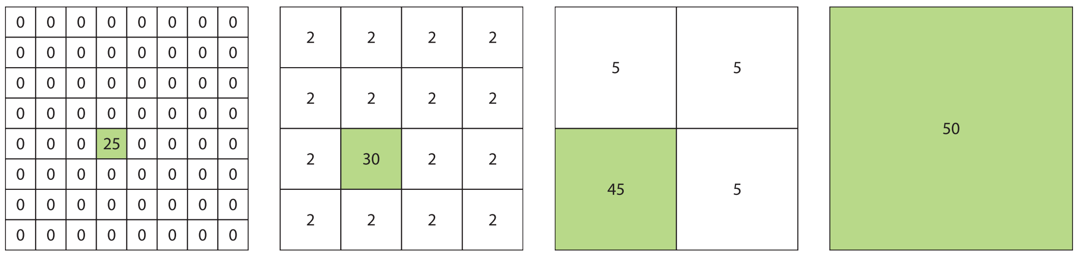
Figure 6.9: With increasing size of the sampled regions, pressure on selected sensors (a) has less input on the final value (b) and (c) until the pressure is not measurable anymore (d).

This means that the specified sensor region with a value $D(j)$ consists of n sensors $D(i)$ where $D(j) > D(i)$. The single sensor value $D(i)$ can now be used to apply the threshold t. The expected result is shown in the example in Figure 6.8.

However, the assumption that all resistors in the specified region are equal could be a serious problem for this approach, because this results in an average value R_i. The problem with averaging the values is, that pressure on one single sensor might be eliminated with increasing size of the region, as shown in the example in Figure 6.9. This behaviour is evaluated in Chapter 7.

Dynamic Thresholding

Another approach is to work with the original values $D(j)$ from the sensor region and adjust the applied threshold instead. When the resistance R_i of one crosspoint between conductive lines without any pressure applied is known, the value for a specified region can be calculated assuming no pressure is applied on the whole region j. The amount of sensors in j can be calculated according to the depth d that is currently evaluated. R_j can then be calculated using Equation 6.3. When the resistance R_j for the whole region is known, the corresponding sensor value $D(j)$ can be calculated using Equation 6.10 and 6.11, where R_i and $D(i)$ are substituted by R_j and $D(j)$, respectively. The result is the sensor value for a given region where no pressure is applied. Alternatively, this



(a) 8×8 (b) 4×4 (c) 2×2 (d) 1×1

Figure 6.10: Without estimating the single sensor values, applied pressure is measurable in all resolutions.

value can also be evaluated empirically. The initial value of a region (see Figure 6.7) can be compared to the measured value $D(j)$ of the current region (see Figure 6.10) and used as a threshold to decide if a given region has to be sampled in more detail. If a region is not sampled with maximum detail, there is no pressure applied on the region and the value 0 can be returned instead of the sampled value. Otherwise the sampled sensor value is returned. This approach has also been tested. The results are described in Chapter 7.

Chapter 7

Evaluation

This chapter describes the results that were accomplished and evaluates the proposed quadtree approach for reading sensor data as well as the developed prototyping framework. First, the evaluation of the quadtree approach is discussed. The results of the prototyping platform are presented in the subsequent section.

7.1 QuadTree Approach

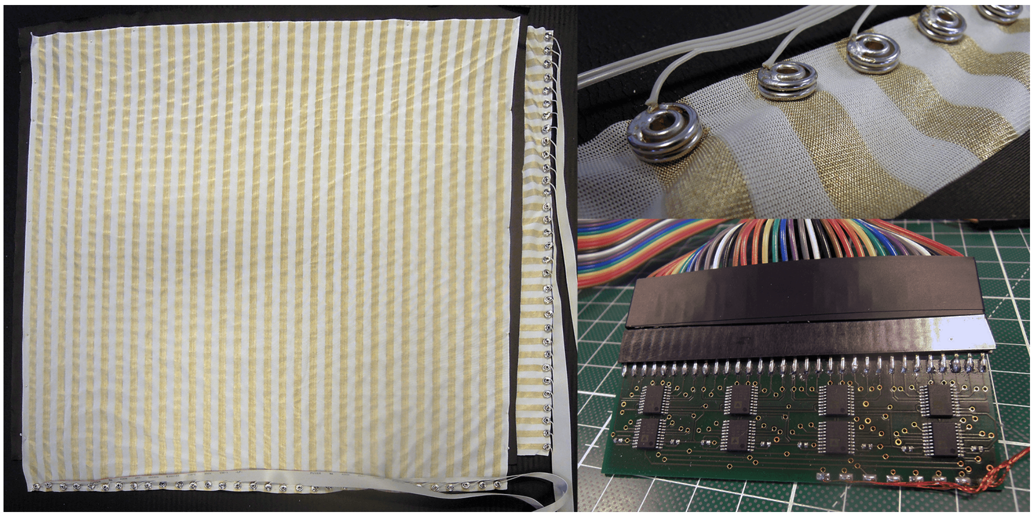
The proposed quadtree method, for speeding up the sampling of the sensor data and possibly reducing the transmitted data at the same time, has been tested. This section focuses on the results of this approach. First, the outcome of sampling the sensor with different resolutions is discussed. Second, the maximum number of active sensors, which still results in a speed up or less transmitted data using the quadtree method, is evaluated, respectively. Finally, the quadtree approach is tested on potential use cases.

7.1.1 Apparatus

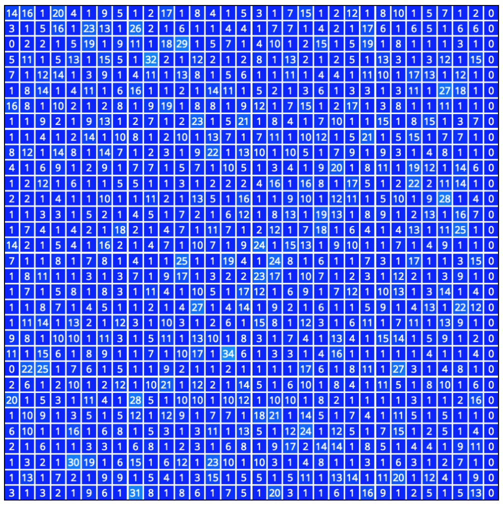
The quadtree approach was developed due to current research on industrially woven smart textiles which can be produced with high resolutions. One example of woven textile sensor is shown in Figure 6.6. However, at the current state of the textile development, the sensor data from the woven textile has too much noise to appropriately apply and evaluate the proposed approach (see Figure 7.1 (b)). Therefore, a new prototype has been implemented for the evaluations. The new prototype (see Figure 7.1) has a resolution of 32×32 sensors and is build out of two perpendicular layers of Zebra Fabric with one pressure sensitive layer in between, like in [21]. For the sampling of the sensor data, the same analog matrix switches were used, as described in Section 6.6.2. Therefore, the sampling speed is comparable to the speed of the woven textile. However, the resulting sensor data is much smoother (see Figure 7.1 (c)) and therefore provides an excellent basis for the evaluation of the quadtree sampling approach.

7.1.2 Dynamic Resolution Sampling

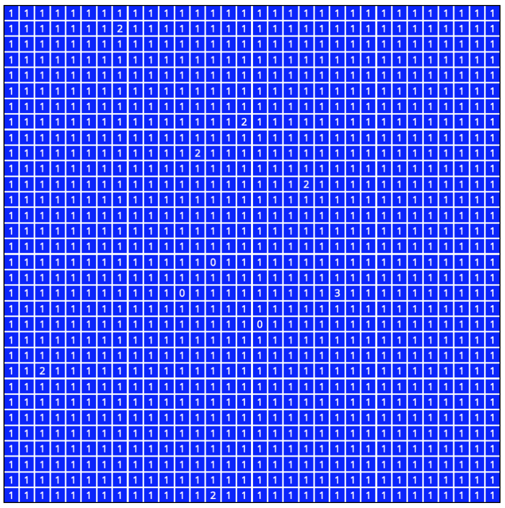
The use of the analog matrix switches facilitates sampling a given textile with different resolutions. This means for example, every $2 \times 2 = 4$ or $4 \times 4 = 16$ adjacent sensors of



(a)



(b)



(c)

Figure 7.1: For the evaluation of the quadtree sampling approach, a new prototype (a) with a resolution of 32 × 32 sensors was built using zebra fabric, because the sensor data from the industrially woven textile (see Figure 6.6) has too much noise (b) for a proper evaluation. The sensor data from the new prototype (c) is much smoother.

a 32 × 32 sensor are sampled at once, like if they were one single sensor. The result is a 16 × 16 or 8 × 8 matrix, respectively. Due to the fact that less values need to be sampled, the sampling time decreases and less data needs to be transmitted. The differences in sampling speed and data length for a 32 × 32 sensor are summarized in Table 7.1.

Dynamic Resolution without Averaging

As described in Section 6.4.2, the value of a sampled region increases according to the number of sensors. The results of sampling a 32 × 32 sensor with different resolutions

Table 7.1: This table summarizes the time needed for the sampling of the sensor data and the length of the produced data for different resolutions of a 32×32 sensor.

Resolution	**Sampling Time**	**Transmitted Data**	**Figure**
32×32	32 ms	1030 bytes	Figure 7.2 (a)
16×16	8 ms	262 bytes	Figure 7.2 (b)
8×8	2 ms	70 bytes	Figure 7.2 (c)
4×4	585 μs	22 bytes	Figure 7.2 (d)
2×2	174 μs	10 bytes	Figure 7.2 (e)
1×1	57 μs	7 bytes	Figure 7.2 (f)

is shown in Figure 7.2. The raw values $D(i)$ of the 32×32 sensor are all less or equal to 2 (see Figure 7.2 (a)). This value increases when sampling the sensor with lower resolutions. If sensor regions j of $2 \times 2 = 4$ sensors are sampled as one to get a resulting matrix of 16×16 values, the highest value $D(j)$ is still 2 (see Figure 7.2 (b)). However, when sampling $4 \times 4 = 16$, $8 \times 8 = 64$ or $16 \times 16 = 265$ at a time, the resulting matrices of 8×8 (see Figure 7.2 (c)), 4×4 (see Figure 7.2 (d)) or 2×2 (see Figure 7.2 (e)) values, already contain values $D(j)$ up to 4, 6 and 14, respectively. When all 32×32 sensors are sampled like one single sensor, the resulting value is 36 (see Figure 7.2 (f)).

Dynamic Resolution with Averaging

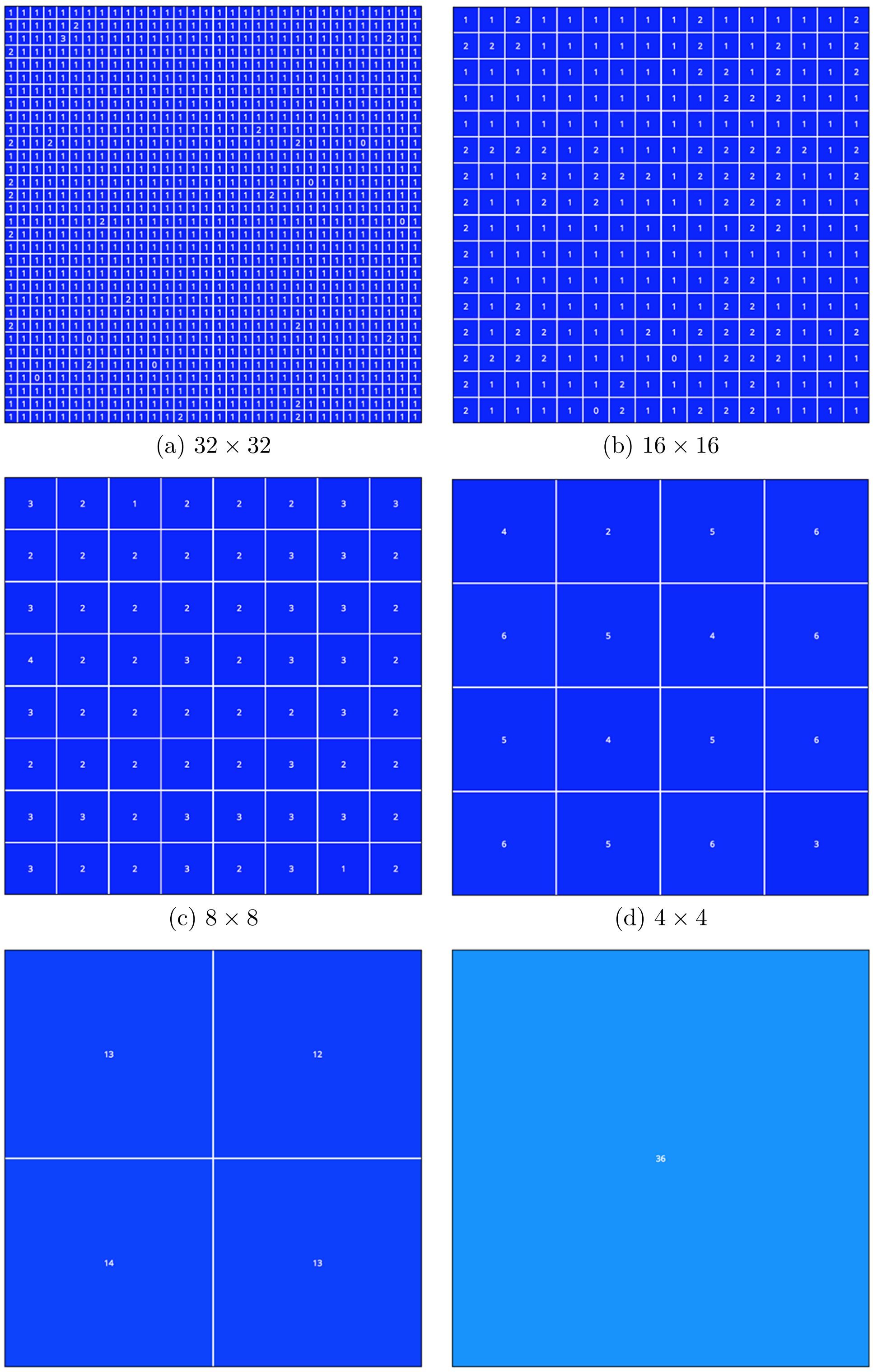
One possible solution to handle this, is estimating the single sensor values $D(i)$ of a sensor region j, like described in Section 6.6.3. The result is shown in Figure 7.3. In comparison to the results discussed in the previous section, the value $D(j)$ for a specified sensor region extending over multiple sensors is always 0 (see Figure 7.3 (b-f)), which is the expected behaviour since no pressure was applied anywhere on the textile.

Dynamic Resolution with Averaging and Pressure

However, as expected, estimating the single sensor values also has a downside. Due to the fact that no information about the single sensor values $D(i)$ of a sensor region j is known, the only way of estimating the single values $D(i)$ is averaging them. Calculating the average has the drawback, that for big sensor regions, pressure on single sensors does not make a difference (see Figure 7.4). While on the original resolution of 32×32 input sensors, applied pressure is clearly distinguishable (see Figure 7.4 (a)), the sensor value $D(j)$ for the region j where the pressure is applied, decreases with lower resolutions (see Figure 7.4 (b-d)). When 16×16 (see Figure 7.4 (e)) or 32×32 (see Figure 7.4 (f)) sensors are sampled together, applied pressure is not noticeable anymore.

Dynamic Resolution without Averaging and Pressure

Without the single value estimation, the applied pressure is noticeable in all resolutions (see Figure 7.5). Therefore, this approach has been used for the quadtree implementation. A dynamic threshold, according to the current depth level of a sensor region, was applied to account for the increasing values $D(j)$, even when no pressure is applied.



(a) 32×32 (b) 16×16 (c) 8×8 (d) 4×4 (e) 2×2 (f) 1×1

Figure 7.2: Different resolutions of a 32×32 sensor with no pressure applied. The values $D(j)$ increase with the number of sensors that are sampled at once.

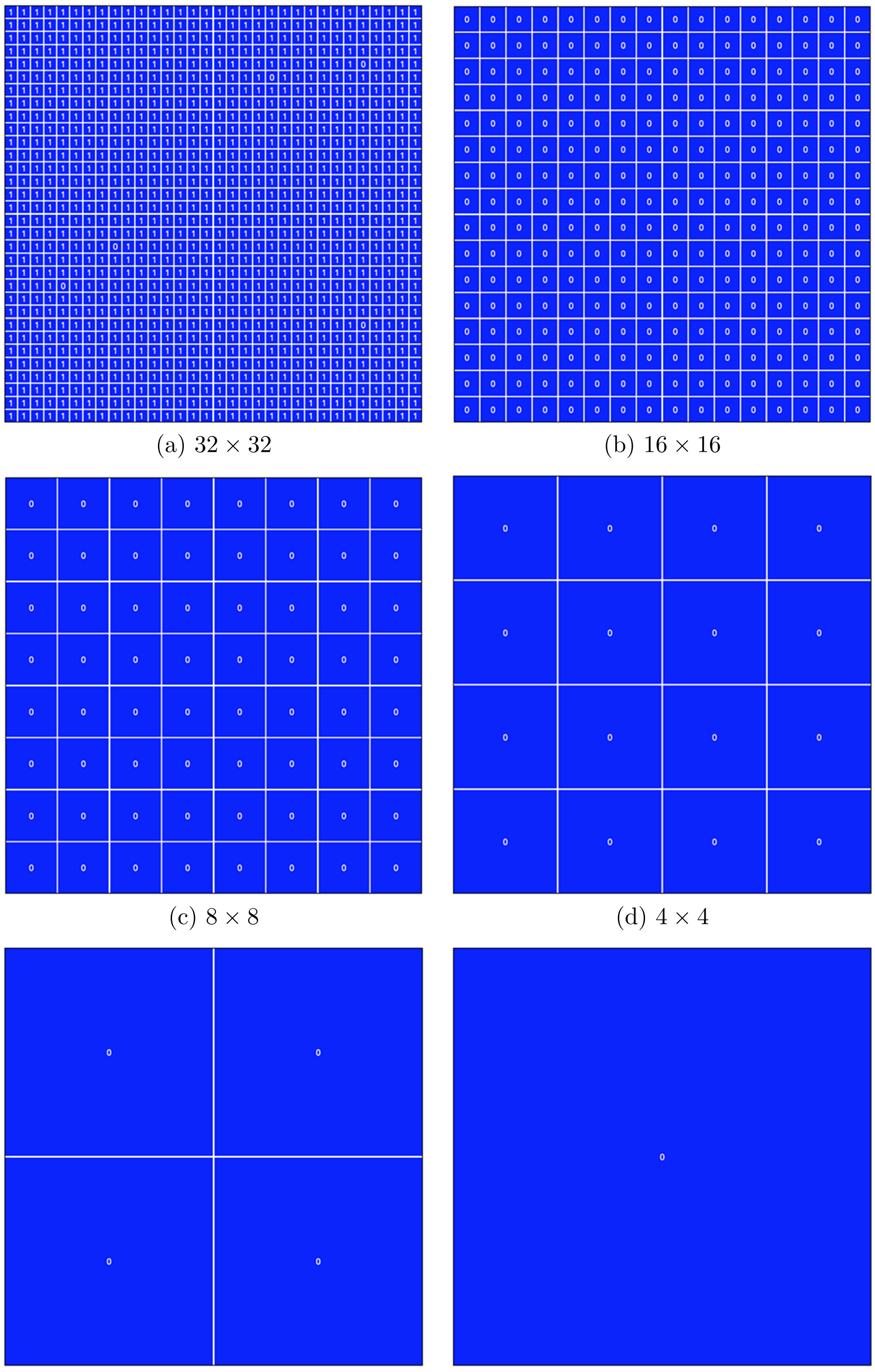
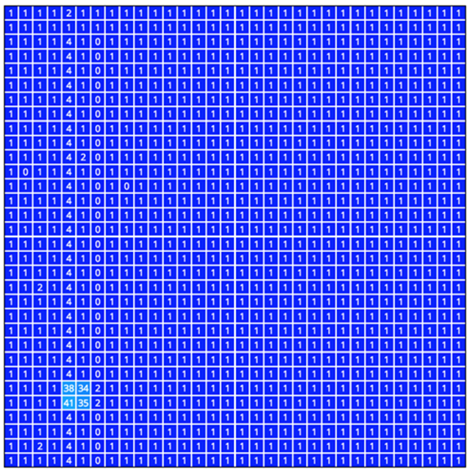
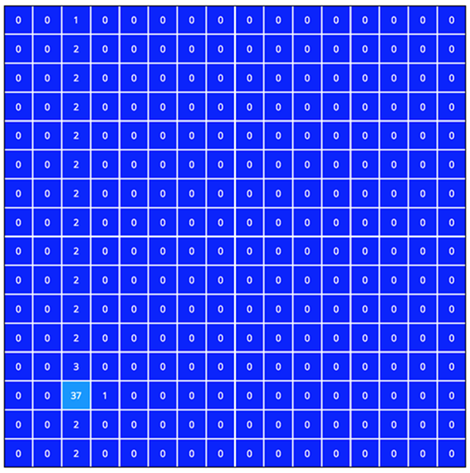
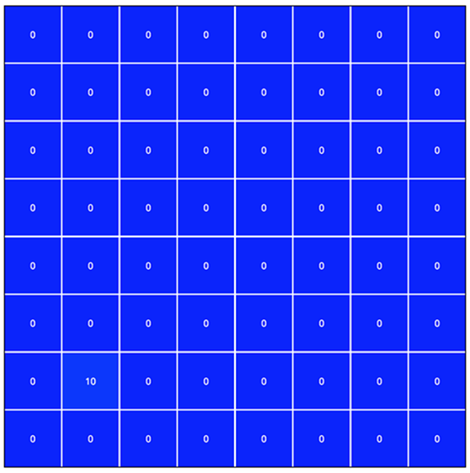
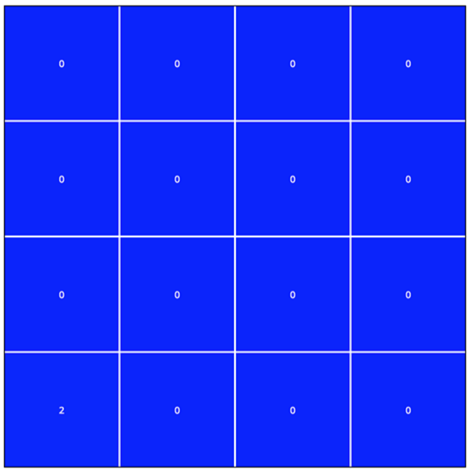


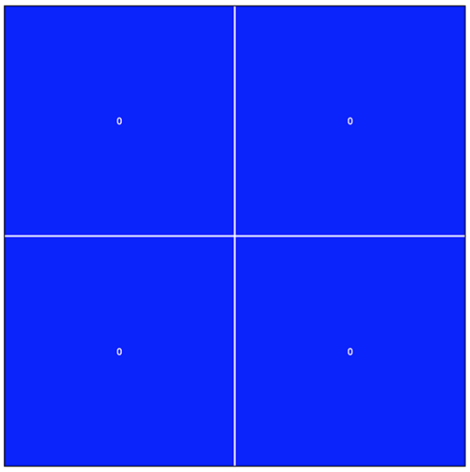
Figure 7.3: The sampling of different resolutions with approximating the single sensors $D(i)$ of each region of a 32×32 sensor, results in $D(j) = 0$ when no pressure is applied.


(a) 32 × 32


(b) 16 × 16


(c) 8 × 8


(d) 4 × 4


(e) 2 × 2

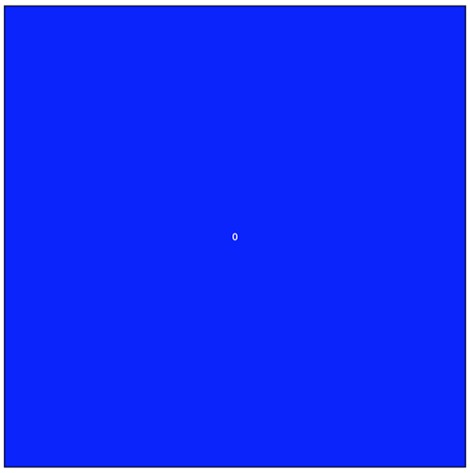
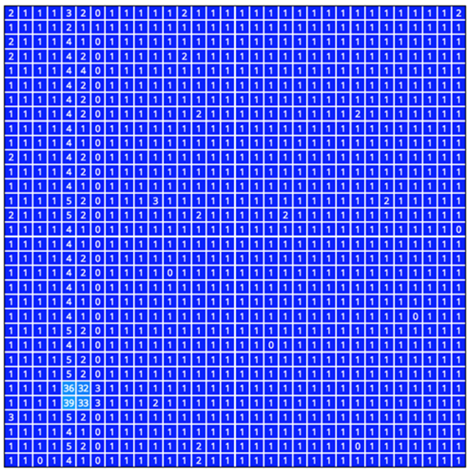
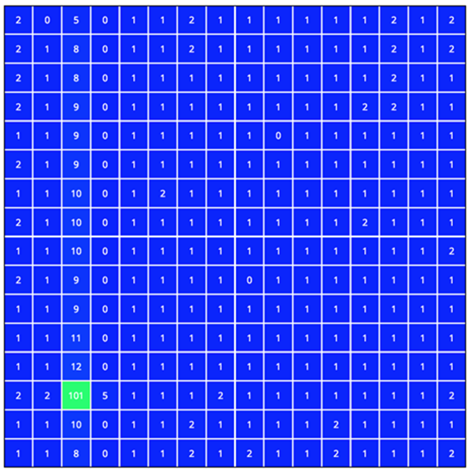

(f) 1 × 1

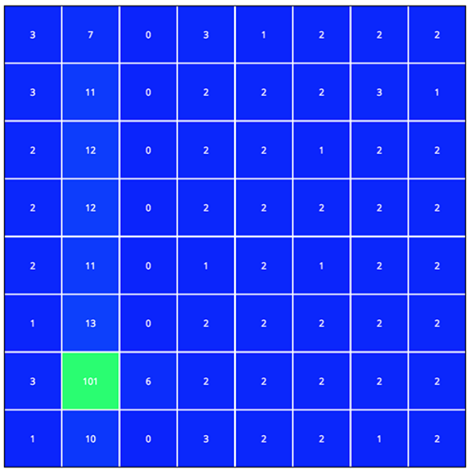
Figure 7.4: Due to the averaging of the sensor values, pressure on single sensors does not make a difference for big sensor regions.



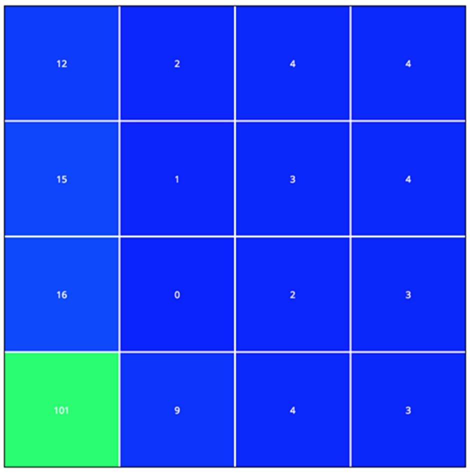
(a) 32×32



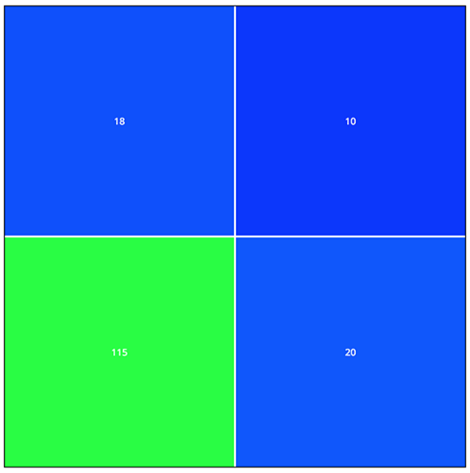
(b) 16×16



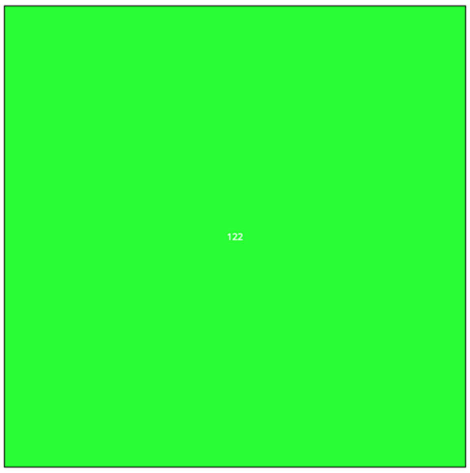
(c) 8×8



(d) 4×4



(e) 2×2



(f) 1×1

Figure 7.5: Without estimating the single values $D(i)$ of a sensore region j, applied pressure is measurable in all resolutions.

7.1.3 Sampling Speed

The goal of the quadtree approach was to increase the sampling time of the sensor values and decrease the amount of transmitted data at the same time. First, this section evaluates the performance of the quadtree regarding sampling time.

Before the quadtree was implemented, the sampling of the sensor values was implemented in a procedural style using `C` as a programming language. For the implementation of the tree, an object orientation programming language was preferred and therefore `C++` was used for the implementation of the quadtree approach. Before the quadtree was implemented, the sampling of all sensor values was re-implemented using object orientation to evaluate the speed difference between the procedural and the object oriented code. The differences are summarized in the following table:

	`C` (procedural)	`C++` (object oriented)
1 Single Sensor	28 μs	32 μs
32×32 Sensor	28 ms	31 ms

The performance of the quadtree approach depends on the number of sensors that have a value above 0. These sensors are henceforth be called active sensors, because these are the sensors of interest where pressure is applied to. The relationship between the number of active sensors and the sampling time is shown in Figure 7.6. However, the sampling time of the quadtree approach not only depends on the number of active sensors, but also on their location. If all active sensors are close together the performance of the quadtree sampling approach is much better than with the same number of active sensors spread across the whole sensor area. Therefore, the sampling time of the quadtree approach with a specified number of active sensors has been measured in a best case and a worst case approach. The best case approach is that all active sensors are close together and as little sensor regions as possible need to sampled in more detail. With this approach the quadtree outperforms the standard sampling method until the number of 280 active sensors is reached. If more than 280 sensors are active, the quadtree approach is slower than the original sampling technique. In the worst case, all active sensors are as far away from each other as possible, which means that multiple regions of the quadtree need to be sampled with great detail. In this case the performance of the quadtree approach is better up to a number of active sensors of 35. If 36 sensors are active, in the worst case scenario, the quadtree approach is already slower.

7.1.4 Data Length

The same evaluation as with the speed of the quadtree approach was also conducted according to the length of the transmitted data. The results are presented in Figure 7.7. If all sensors need to be sampled with the maximum depth of the quadtree, twice as much data needs to be transmitted (excluding the 6 header bytes). In the best case scenario the quadtree approach results in less transmitted data until a number of 508 active sensors. However, if the active sensors are arranged in the worst possible case, the point where the quadtree stops being an improvement and actually results in more data, that needs to be transmitted, is between 85 and 86 active sensors.

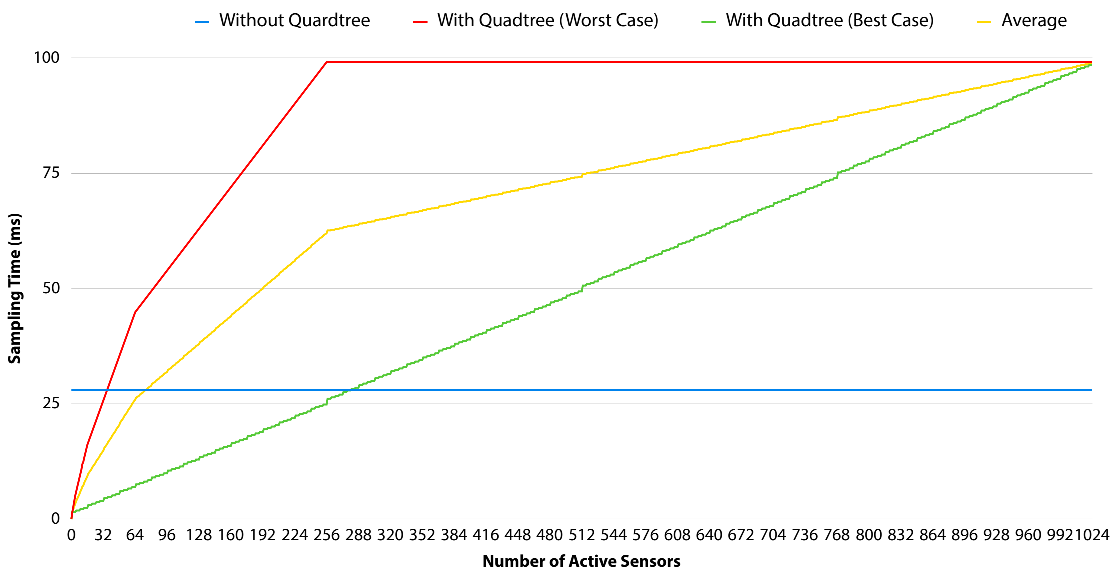


Figure 7.6: The performance of the quadtree approach depends on the number of active sensors. Without using a quadtree, the sampling of a 32×32 sensor always takes 28 ms.

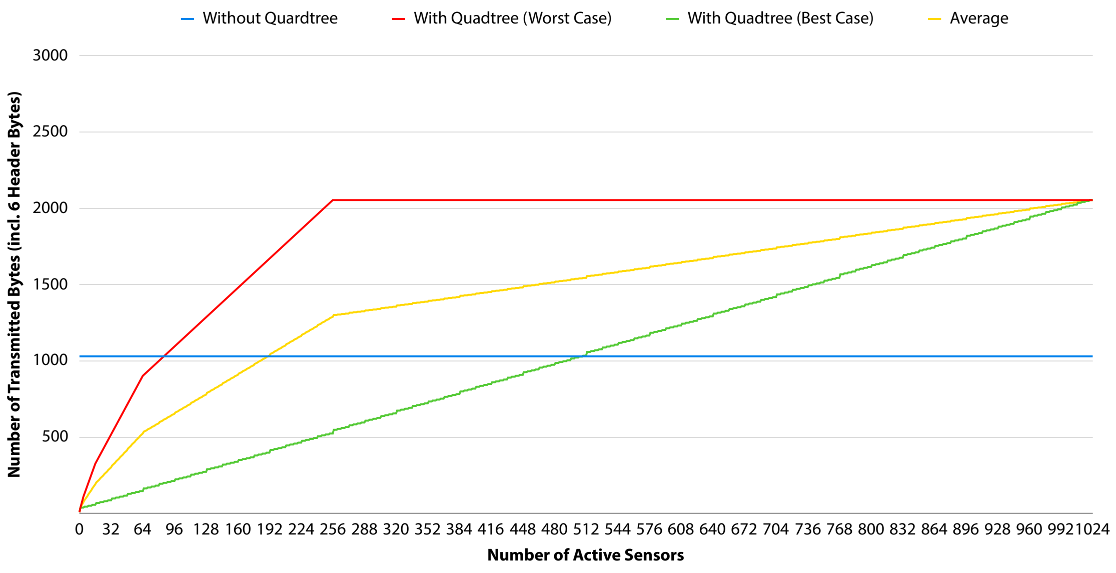


Figure 7.7: The transmitted quadtree data varies according to the number of active sensors. Without the quadtree method 1030 bytes are transmitted constantly.

7.1.5 Use Cases

While Figures 7.6 and 7.7 show the performance of the quadtree in two theoretical scenarios, several typical use cases have also been tested to evaluate the performance. The results of this use cases are described in this section. Most use cases were performed on a woven textile senor (with filtering applied to smooth the input data). The resulting pressure image was replicated on the prototype with the zebra fabric to test the quadtree

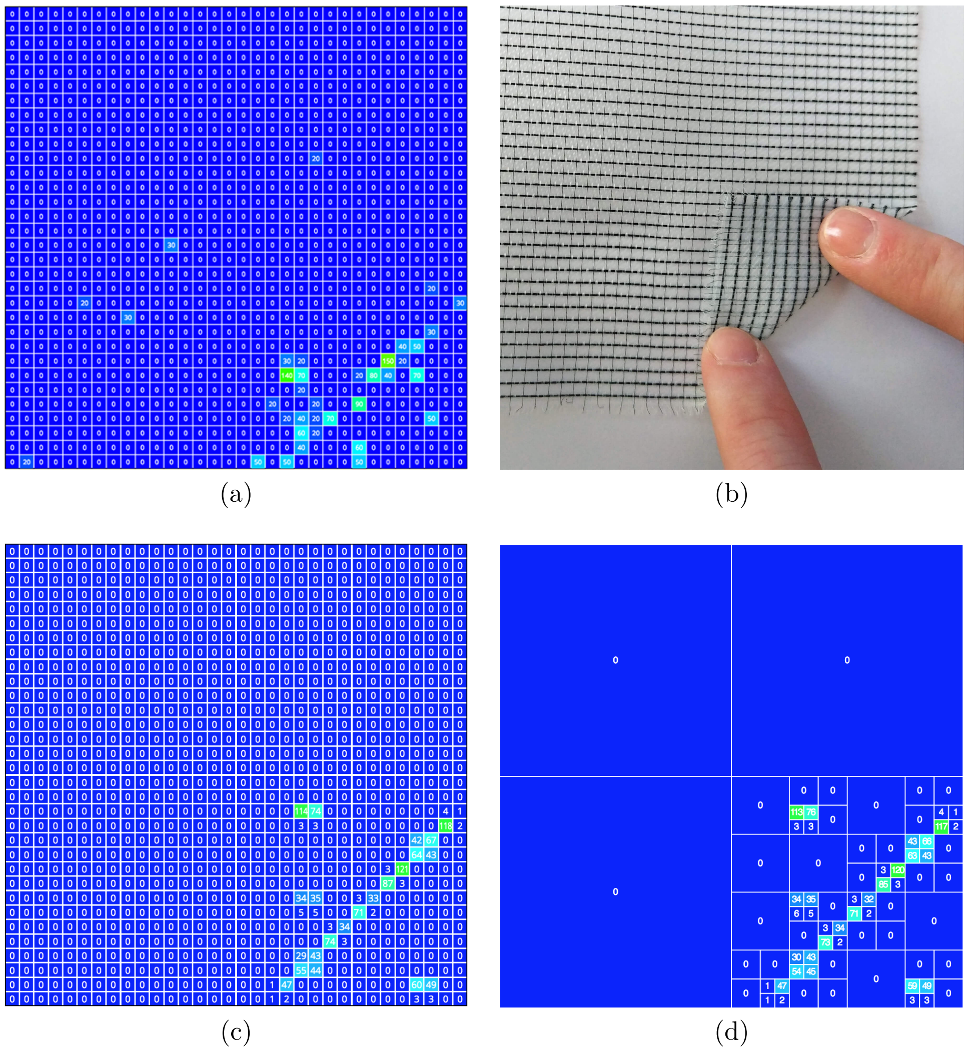


Figure 7.8: A fold gesture (b) was performed with the woven smart textile prototype, which resulted in the pressure pattern shown in (a). The sensor data was replicated on the new prototype (c) and resulted in the quadtree structure shown in (d).

approach and measure the sampling speed. The sensor values, the performed gesture and the results of the quadtree approach are shown, respectively. Additionally, two use cases for big textile sensors have been tested on the zebra fabric directly.

Fold

As shown in Figure 7.8, the fold gesture expands diagonally over one quarter of the textile sensor. The replicated pressure pattern has 40 active sensors. However, only one quarter of the sensor needs to be sampled in more detail. Therefore, the quadtree approach is 268 % faster than the standard approach, because big parts of the sensor can be combined to large sensor regions. The transmitted data is 158 bytes long, in comparison to 1030 bytes without the quadtree method.

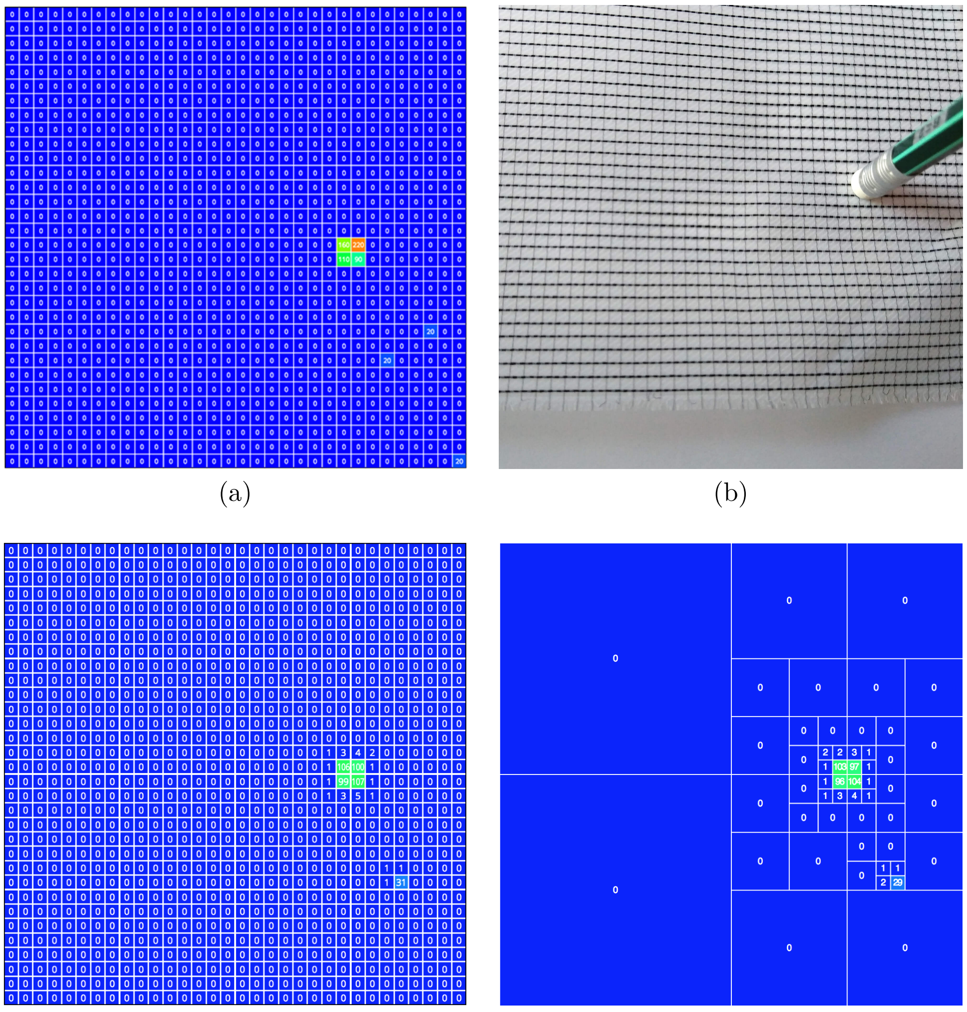

(a) (b) (c) (d)

Figure 7.9: Pressure applied with the back of a pen on the woven smart textile prototype (b) results in the sensor data shown in (a). The replication on the new prototype and the resulting quadtree structure are shown in (c) and (d), respectively.

Pen Input

Pressure with the back end of a pen resulted in the fewest active sensors from all tested use cases (see Figure 7.9). While 4 sensors were actively pressed with the pen, some neighboring sensors were affected too, as well as one outlier that was randomly picked to simulate minimal noise in the sensor data. The total sum of active sensors was 20. However, the input gesture is located between two quarters. Nevertheless, the quadtree method results in a speed-up of 437 % and only 110 bytes need to be transmitted.

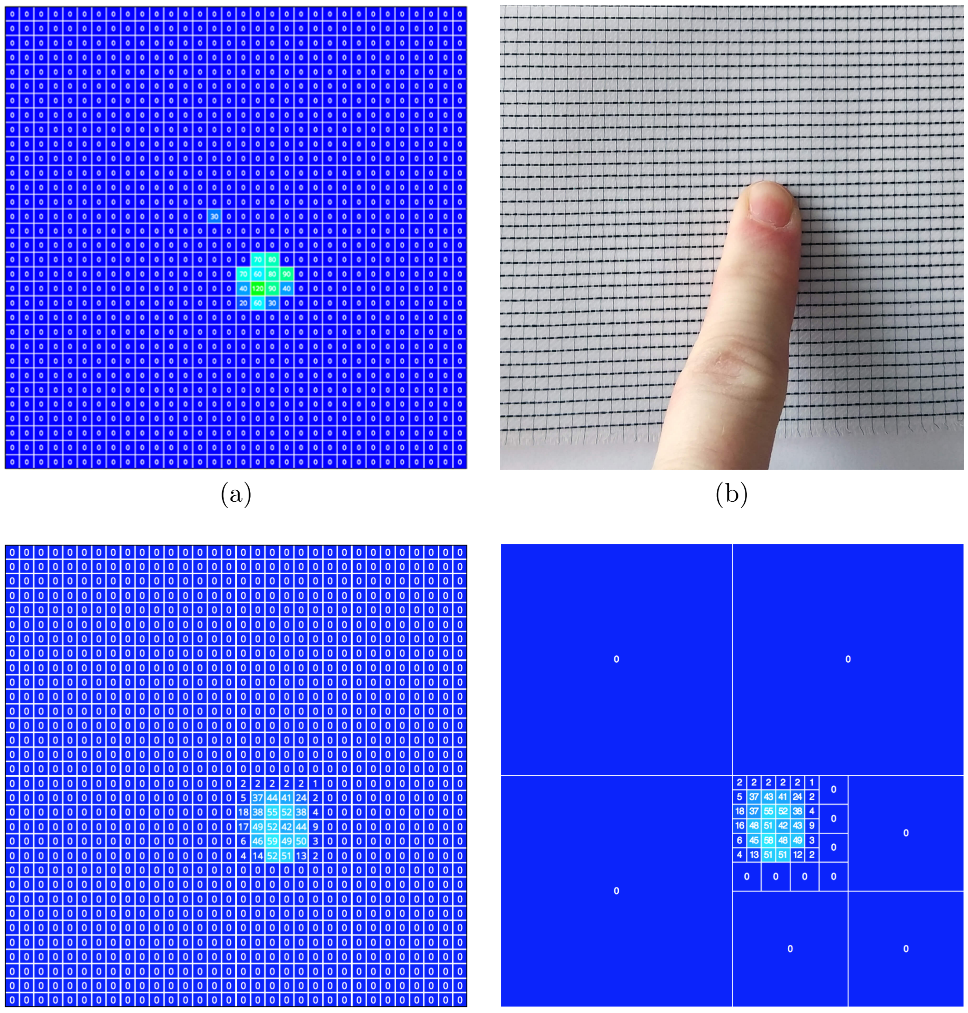

(a) (b) (c) (d)

Figure 7.10: This figure shows the pressure pattern (a) of touching the smart textile with 1 finger (b). Again the sensor data was replicated on the prototype made out of zebra fabric (c) and the corresponding quadtree structure (d) is shown.

Touch Input with One Finger

Furthermore, touch gestures with different fingers were examined. First, only one finger was used to touch the smart textile (see Figure 7.10). The result were 36 active sensors. Although this means more active sensor compared to the pen input, the sampling was even 474 % faster using the quadtree approach. This can be explained due to the fact that all active sensors from the finger input lie in one quarter of the sensor area (see Figure 7.10 (c)). Thus, 3 quarters do not have to be sampled in more detail, in comparison to the example with the pen input, where only two quarters remained entirely untouched.

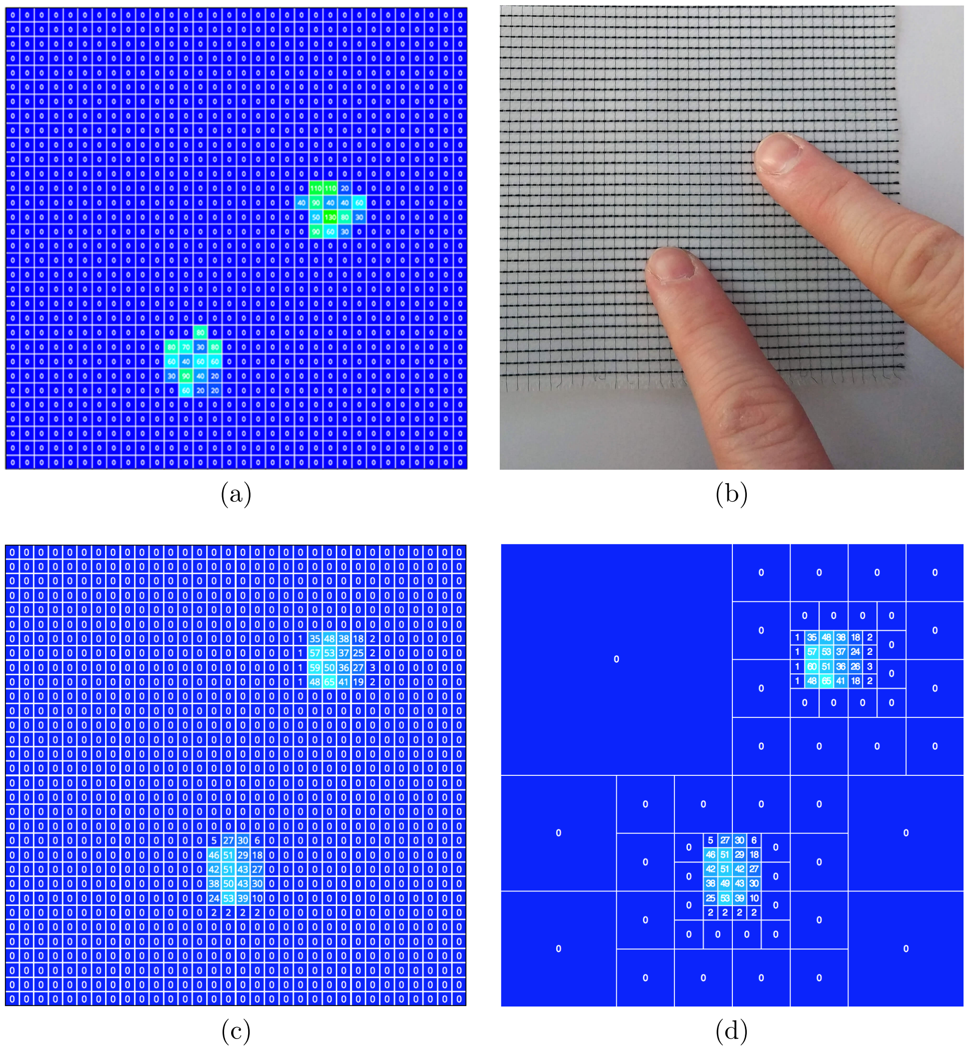


Figure 7.11: When the woven smart textile is touched with 2 fingers (b), two active areas are recognizable in the pressure pattern (a). The replication on the zebra fabric and the resulting quadtree structure are shown in (c) and (d), respectively.

Touch Input with Multiple Fingers

As expected, the number of active sensors increased with the number of fingers used to touch the textile sensor. Two fingers for example resulted in 48 active sensors (see Figure 7.11). However, the quadtree approach still performed 188 % faster and needed only 200 bytes to be transmitted, instead of 1030 bytes (including header files). If four fingers are used to apply pressure on the textile (92 active sensors), the quadtree approach still outperforms the standard sampling technique by 85 %, although the fingers are spread across the sensor and no quarter of the smart textile remains completely untouched anymore (see Figure 7.12).

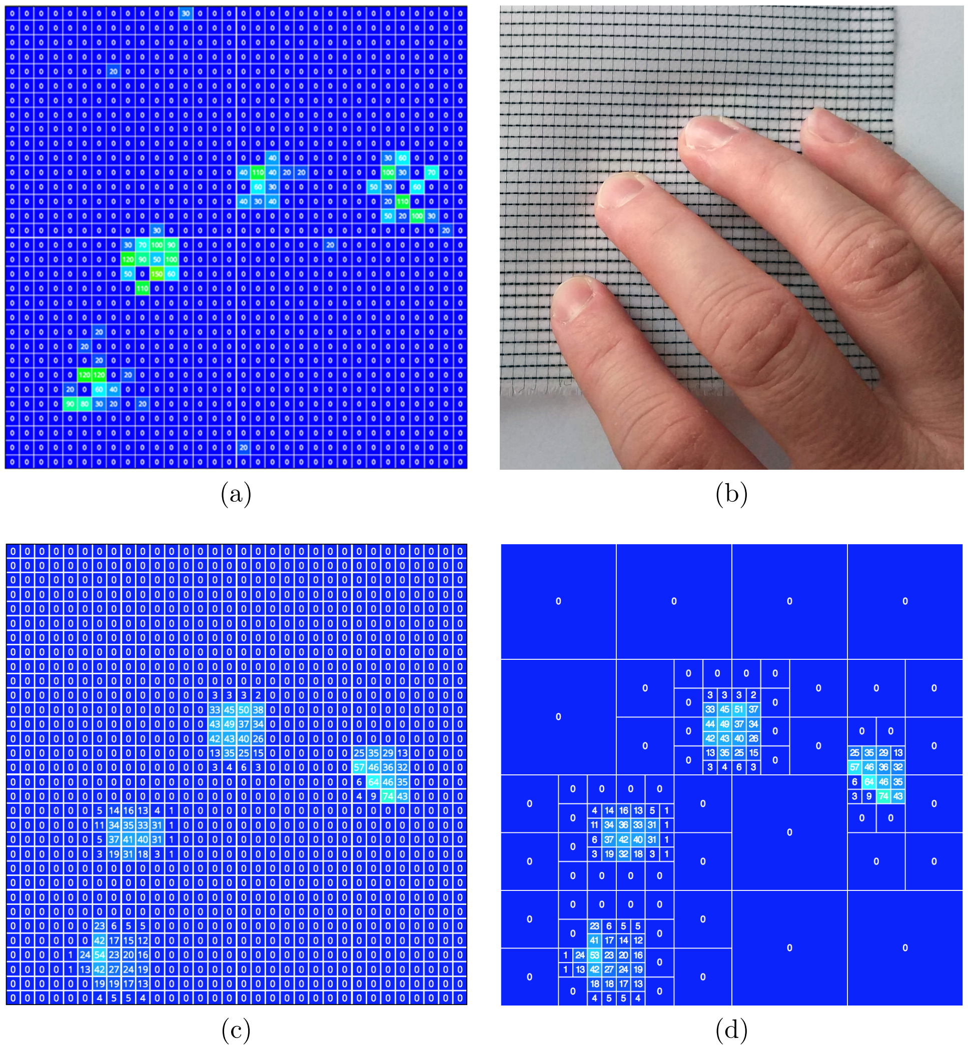


Figure 7.12: The pressure pattern for touching the smart textile with 4 fingers (b) is shown in (a). The pattern was then replicated on the prototype made out of zebra fabric (c). The corresponding quadtree structure is shown in (d).

Shear

With 108 active sensors, the shear gesture (see Figure 7.13) was the biggest gesture tested on the woven textile sensor. The sampling of the sensor using the quadtree approach took 14 ms in comparison to the sampling time of 28 ms without the use of a quadtree. Therefore, the sampling was still 97 % faster. Also the length of the transmitted data was 922 bytes shorter in comparison to sampling all sensors individually.

Tailor Seat

Additionally, two use cases for bigger sensors were tested on the new prototype made out of zebra fabric directly. The first tested use case was sitting on the textile sensor

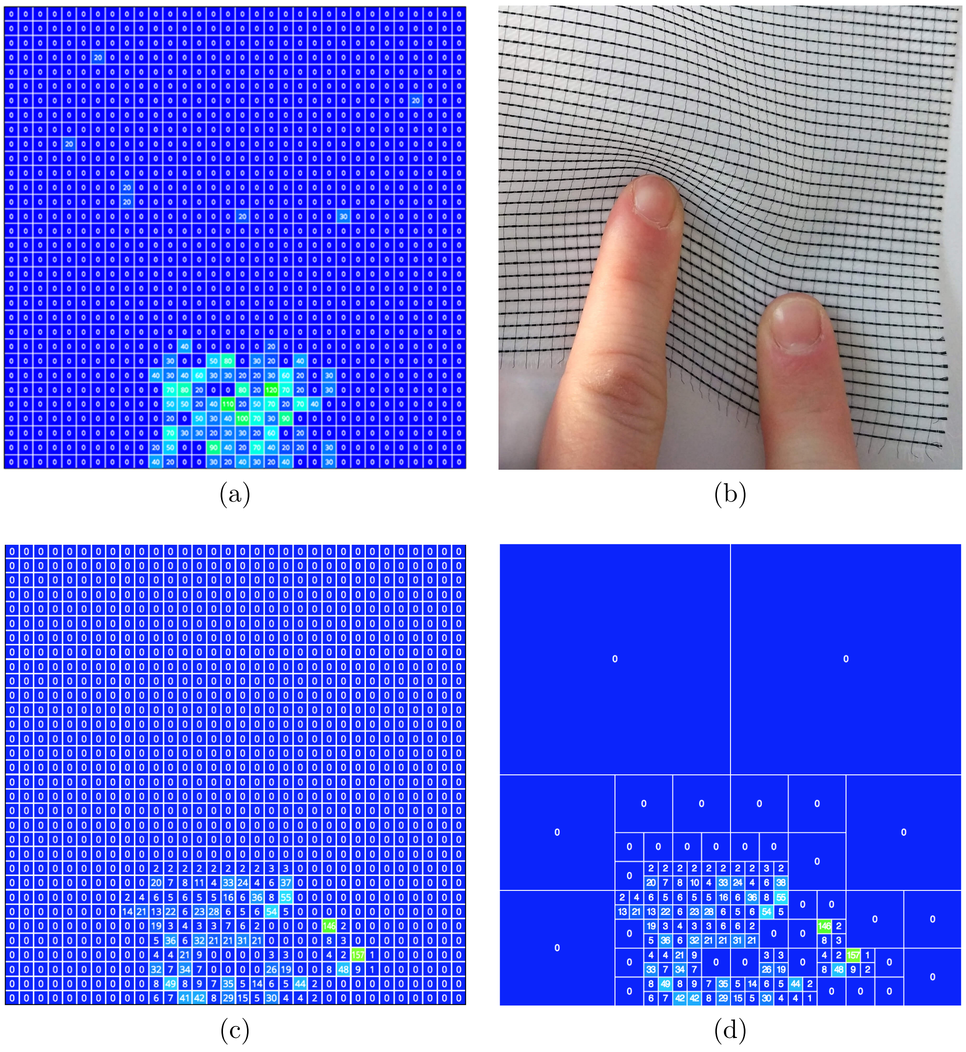


Figure 7.13: The pressure pattern (a) of a performed shear gesture (b) was replicated on the new prototype (c) and resulted in the quadtree structure shown in (d).

in tailor style (see Figure 7.14). The sitting position activated a total number of 168 sensors, which is the maximum number of active sensors during the evaluation of all different use cases. The result of the speed evaluation shows, that the quadtree approach still performs 24 % faster in comparison to the previous simple sampling method. The length of the transmitted data decreased from 1030 bytes with the standard approach to 458 bytes using the quadtree method.

Stand

Furthermore, standing on the textile sensor was tested (see Figure 7.15). The result were 88 active sensors with a sample time of 14 ms, which corresponds to a speedup of 105 %. The length of the transmitted data could be reduced form 1030 bytes to 266 bytes (including header bytes).

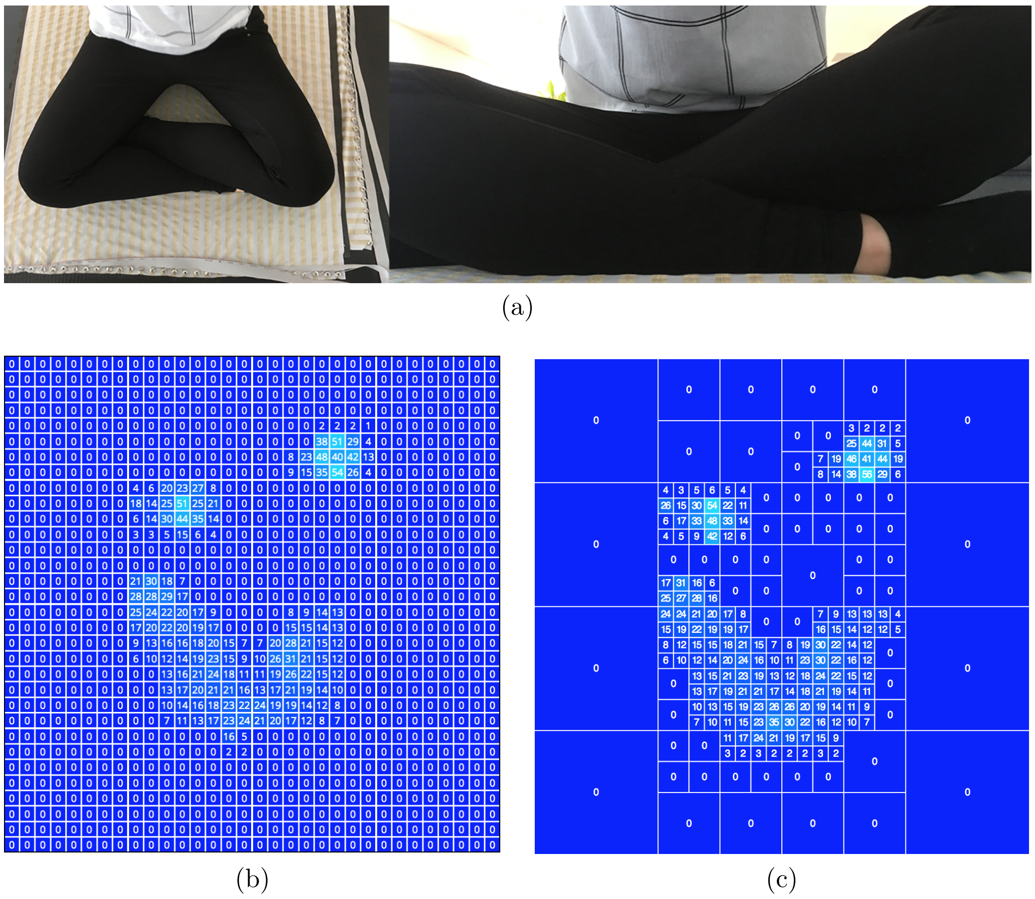


Figure 7.14: Sitting on the smart textile in tailor style (a) was tested directly on the new prototype made out of zebra fabric. The resulting pressure pattern and the quadtree structur are shown in (b) and (c), respectively.

7.2 Web-Based Prototyping Approach

Two different methods were used to evaluate the result of the developed framework. One technique was to implement a new, completely different data visualization which is needed to perform a study in a different project. The goal was to evaluate the difficulty of adding custom visualizations and the possibilities for different visualizations for specific use cases. Second, an interview with an expert in the area of sensing technologies was conducted to evaluate how he could benefit from the application design and find out about his thoughts about working with the framework and problems he experienced.

7.2.1 Custom Visualizations

The standard sensor visualization is a matrix with all input values of the sensor, which are colored corresponding to their value. An example for a 32×32 sensor is shown in Figure 5.2. For different use cases, there might be the need to add custom visualizations. Therefore, a custom visualization for a given concept was prototyped to evaluated

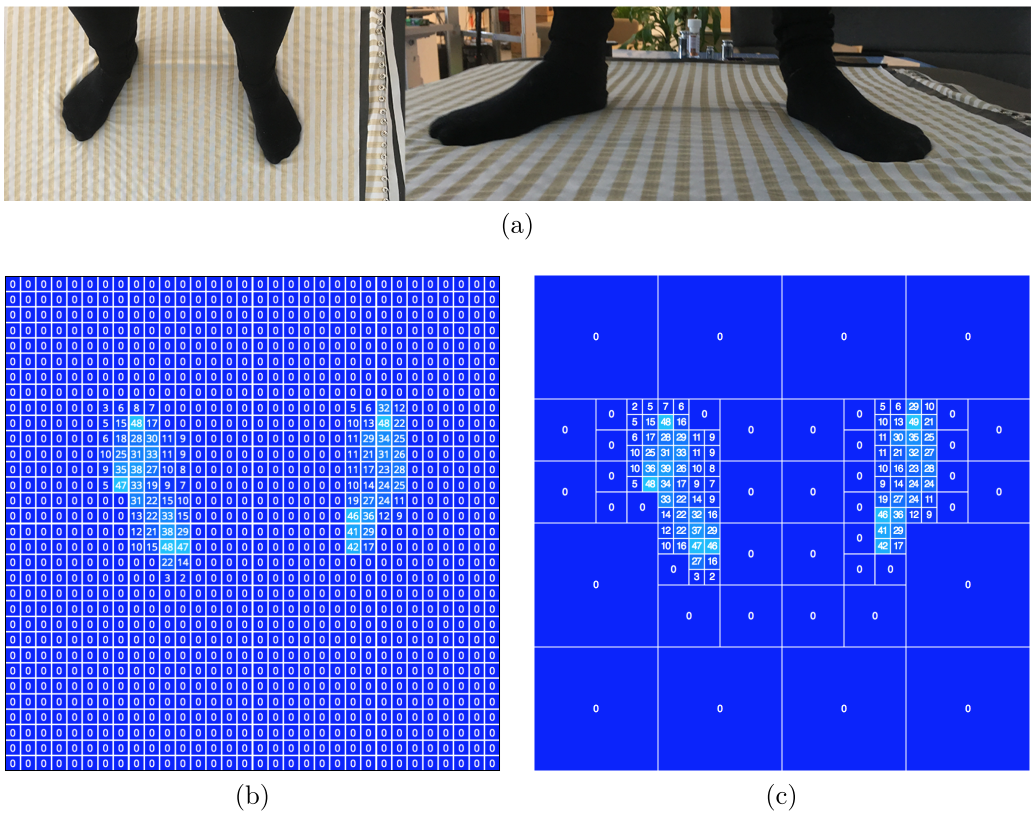


Figure 7.15: Standing on the smart textile prototype (a) was also evaluated. The resulting sensor data (b) and the corresponding quadtree structure (c) are shown.

if custom visualizations are possible and what options a developer has for the implementation.

Use Case

The developed visualization was created for an already existing prototype. The prototype consists of the left handle of a bike covered with a smart textile, which contains 4 sensor columns. Additionally to the prototype, a study that should be conducted was already drafted, but the concept needed to be implemented. The goal of the study is to find out, if people are able to apply different levels of pressure with selected fingers while riding a bicycle or motorcycle. The necessary visualization was implemented for evaluation purposes. The study itself is conducted in a separate project.

Seven different pressure levels were defined corresponding to seven predefined colors. A graphical visualization of a hand should be the center of the visualization which indicates the pressure applied for each finger, by coloring the finger with the appropriate color, respectively. The thumb was excluded from the study. Based on this setting, different configured tasks should be completed by the subject. Each exercise defines a specific color (i.e. pressure level) that should be reached and a selected finger which should be used to apply the pressure. Once the level is reached, the pressure needs to be hold steady for a given number of seconds. The time remaining is visualized by a clock animation.

Implementation

For the implementation a separate *React* component named `HandVisualization` was created which contains the developed visualization. Therefore, the new visualization can simply be switched with the default visualization by rendering the new component instead. To switch between different visualizations in the user interface, a select field exists, which defines which visualization to show.

Data Input: The information about the current sensor values is already available in the global data storage of the client application. Therefore, the new visualization can access the data and only implement custom logic if necessary. For example, in the developed visualization the highest value of each column was used as the value for the corresponding finger.

Hand Visualization: The visualization of the hand was implemented using a SVG image that was illustrated in *Adobe Illustrator* and then exported to SVG. While the SVG file could be included using the HTML `img` element, a better solution is to create a new *React* component, copy and paste the source code of the create SVG file into that component and then include this component instead. The advantage of this solution is, that all parts of the created SVG visualization can be accessed and styled using *Cascading Style Sheets* (CSS). This is for example necessary to dynamically change the colors of the single fingers.

Study Settings: Different test sequences need to be defined for different subjects. The easiest way to achieve this is creating a standard *JavaScript* file which exports the sequences of configurations for different settings. This file is then imported by the visualization. The following code illustrates how the configuration could look like:

```
export default {
    Setting1: [
        { fingerNumber: 3, matchNumber: 2, seconds: 1 },
        { fingerNumber: 1, matchNumber: 6, seconds: 3 },
        { fingerNumber: 1, matchNumber: 2, seconds: 2 },
    ],
    Setting2: [
        ...
    ]
};
```

Visualization Flow: Starting the study and handling the flow of the different study configurations is done using standard *JavaScript*. The concept of *React* makes it easy to listen to changes in any properties of the visualization and react accordingly. Switching to the next configuration when a task is completed successfully is done using the `setTimeout()` method, once the correct pressure level is reached. When the timeout is complete, the visualization switches to the next configuration. If the required pressure level cannot be held, `clearTimeout()` is used to stop the timer.

Timer Animation: The timer animation is implemented using CSS animations, which are a powerful tool for quick and simple animations. If more complex animations are needed, SVG animations can be used which can be exported from different animation applications and then embeded and triggered via *JavaScript*.

Adding the Custom Visualization

The file `client/src/components/Visualization/types.js` defines which visualizations exist. Once the new visualization is implemented as a single *React* component `HandVisualization`, the only step necessary to include the visualization to the user interface of the developed framework, is to import the new component in this file and configure the name and the needed properties of the new visualization according to the example of the matrix visualization:

```
import React from 'react';
import MatrixVisualization from './MatrixVisualization';
import HandVisualization from './HandVisualization';

const types = [
    { name: 'matrix', component: <MatrixVisualization width={500} height={500} /> },
    { name: 'hand', component: <HandVisualization /> },
];

export default types;
```

After including the component to the types configuration, the visualization is automatically added to the view selection and can already be used. It is recommended to store all visualizations in a separate folder located at `client/src/components/Visualization`, to store them at one location and keep the import paths simple. However, this is not necessary as long as the path in the `import` statement is set correctly.

Result

The resulting visualization is shown in Figure 7.16. The four fingers, from the index finger to the little finger are colored according to their pressure level. The available colors are shown on the left side of the visualization from bottom (low pressure) to top (high pressure). On the left side of the visualization a select field allows to choose a study setting and start a study process (see Figure 7.17 (b)). Once the study is started (see Figure 7.17 (a)), only the finger which should be used to perform the current task is colored. One of the color fields on the left is slightly enlarged and has a black frame. This color should be matched by applying the appropriate pressure. The enlarged color field without border signifies the current pressure level as well as the color of the highlighted finger. The text on the right side of the visualization states the task that should be accomplished and contains how long the defined pressure level should be held. As soon as the correct pressure level is reached, a timer animation shows the remaining time until the task is complete (see Figure 7.17 (c)). If the pressure level cannot be held for the given number of seconds, the timer is reset and the countdown starts again when the

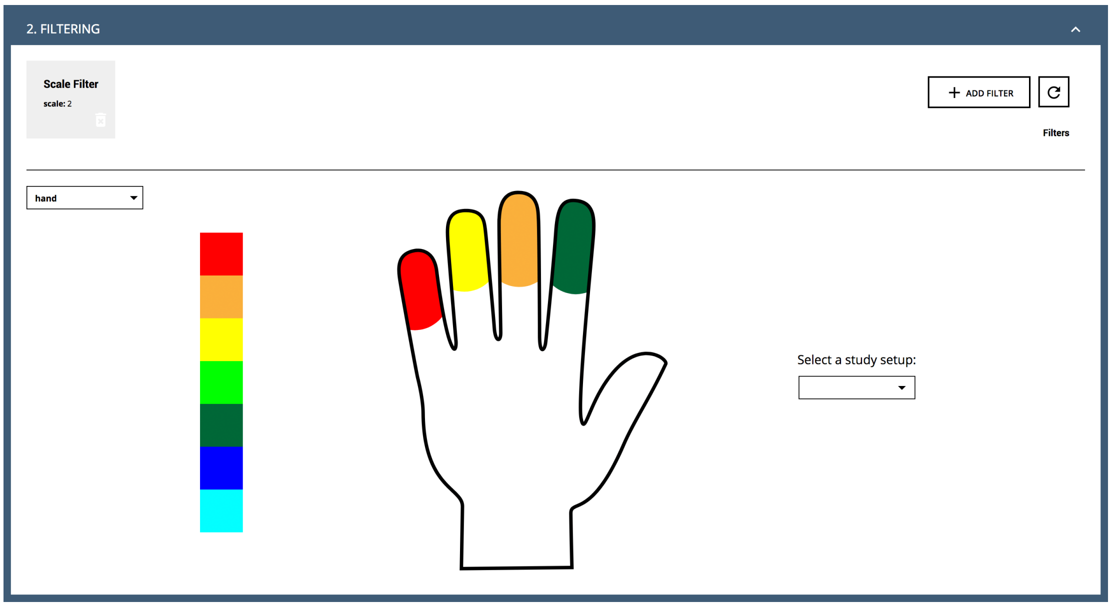


Figure 7.16: A new visualization was developed to visualize the pressure applied with different fingers, respectively.

correct pressure level is found again. Once the timer is complete (see Figure 7.17 (d)), the visualization automatically switches to the next study configuration that should be accomplished. Figure 7.17 (e) shows the screen indicating the end of the study. If the reset button is clicked, the visualization returns to the start screen, which is shown in Figure 7.16.

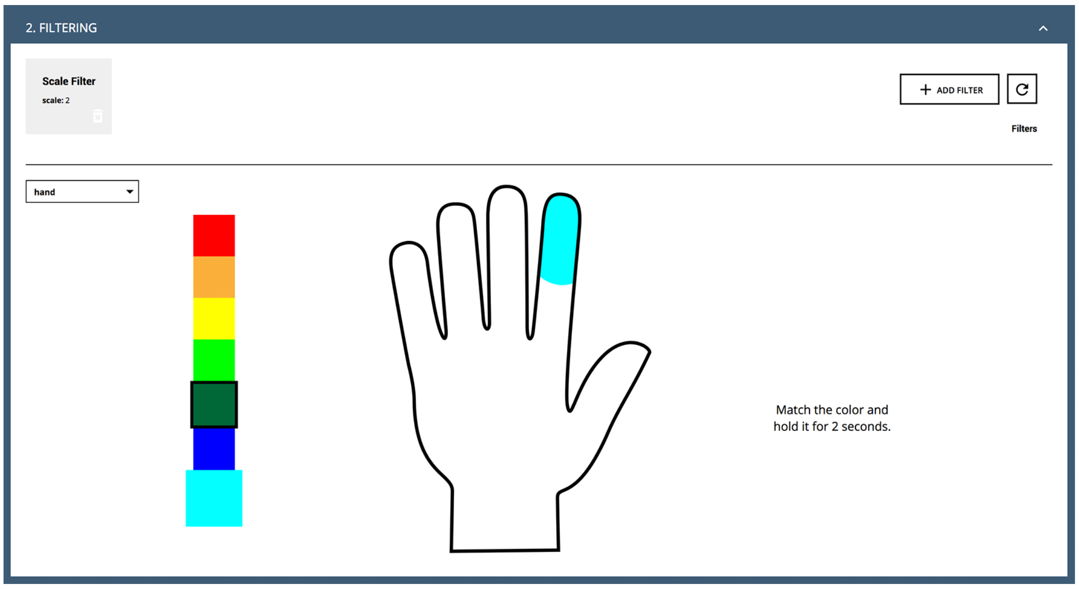
7.2.2 Expert Interview

To evaluate the developed prototyping tool also from the perspective of a researcher from the field, an interview with an expert in the field of human computer interaction and sensing technologies was conducted. The main points of interests that were evaluated in the interview are, if the framework is simple to understand and which improvements could be made to further enhance and simplify the prototyping experience.

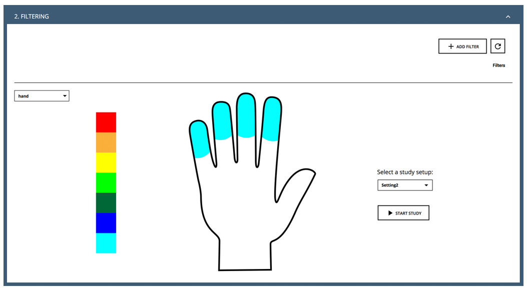
About the Expert

The interviewed expert is a researcher and postdoc student at the *Media Interaction Lab*[1] of the *University of Applied Sciences Upper Austria* located in Hagenberg. He has several years of experience with sensing technologies and worked with sensor foils, i.e. piezoelectric sensors. For the signal processing and the visualization of the signal data he works mostly with C#. Therefore, he has little experience with modern web development, although he is currently working on an mobile app prototype, where *React*

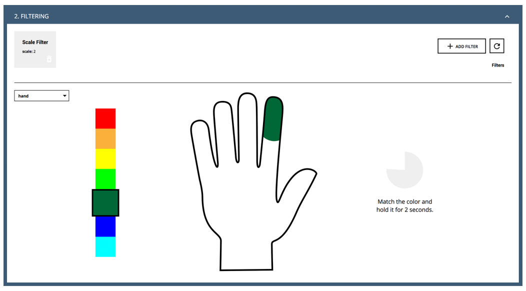
[1]http://mi-lab.org/



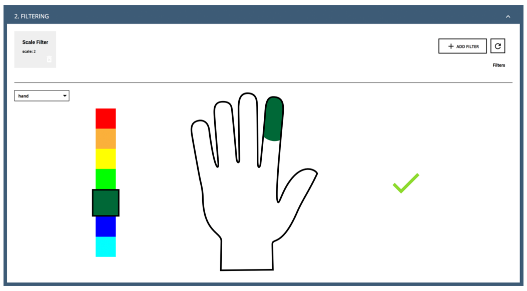
(a)



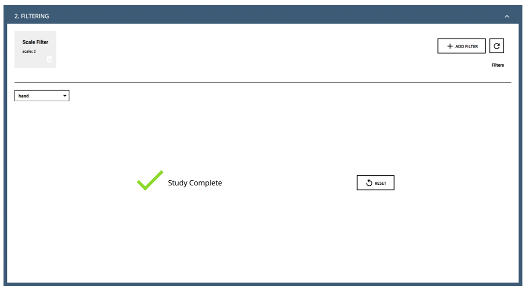
(b)



(c)



(d)



(e)

Figure 7.17: When a study setup is chosen and started (b), different pressure levels have to be reached with selected fingers (a), respectively. Once the correct pressure level is reached, it needs to be held steady for the given number of seconds (c). When the task is complete (d), the visualization switches to the next task automatically until all tasks are done and the study is completed (e).

Native[2] is used. Thus, he is familiar with the basics of the *React*[3] library, which is used for the development of the user interface of the described application.

Use Case

For the publication of a resistive yarn he and his colleques implemented several textile sensor prototypes to demonstrate different use cases of the developed yarn. For the prototyping, the developed framework described in this thesis was used for the data processing, the visualization and the triggering of the wanted actions. Although the user interface was used to prototype, most of the changes and additions were made to the backend application. The UI was hardly touch, however some small features like custom buttons were added.

In the backend application several different filters and two simple algorithmic gesture descriptions were added: one simple touch heuristic to recognize different tap and slide gestures and a heuristic returning the current position of the finger when moving it in a circle on the smart textile. Furthermore, blob detection has been implemented and the features of the detected blob have been used for the SVM classification instead of the raw sensor data.

Data Structure

The prototypes they developed for the publication were constructed in different ways. While for some prototypes single lines of the resistive yarn were hand-stitched, other prototypes used bigger woven textile sensors. However, for all prototypes the data structure was the same. The raw data was always a matrix of input values.

Experience with the Prototyping Framework

The participant mentioned that for him it was easy to start working with the framework and he didn't have difficulties understanding the application structure. The structure of the backend application is easy to comprehend because the single modules, like custom heuristics or SVM models, can be edited by the developer without touching the rest of the application code. However, he imagines customizing the UI might be a little bit tricky at first, if the developer has never used *React* or a similar frontend framework before. He liked the concept of triggering API request when specified gestures are recognized, because it is easy to connect to existing APIs, like the *Philips HUE* lights for example.

Ideas for Improvement

Working with the framework also revealed several aspects regarding work efficiency and the user experience for the developer, which can be improved.

Adding Custom Filters: If a new custom filter is developed, there are multiple steps necessary to include the filter in the application so it can be used by the backend application and configured from the user interface:

[2]https://facebook.github.io/react-native/
[3]https://reactjs.org/

1. The filter class has to be implemented, where the behaviour of the filter is defined.
2. Then the filter has to be added to the `FilterFactory` (see Section 4.3.2).
3. To make the filter usable from the UI, a database entry has to be made, to define the input fields for the necessary parameters.

Therefore, it can be tricky for the developer to implement a new filter and get it to work. This process could be simplified, so that fewer steps are necessary to implement a custom filter.

Settings for Multiple Prototypes: The framework is currently based on the work with one prototype. All settings of the prototype, for example the defined filters or learned gestures, are stored in a database. However, there is currently no way to switch to another prototype without overriding the settings for the last prototype. While multiple databases for the different prototypes can be used as a workaround, this is not an optimal solution and there is room for improvement.

About the User Interface

The collapsible sections allow to focus on one particular section and save screen space by closing sections, that are not needed at the moment. The training of different gestures for the SVM model via the user interface worked fine. Nevertheless, also for the UI there are some aspects that can be improved.

Data Visualization: The visualization of the data only shows the data after the filtering pipeline, like it is used for the gesture recognition. However, for the developer it can be interesting to also see the raw data next to the filtered data, so they can be easily compared. This would for example help to find out if there are errors in the original sensor signal or if a filter is not working correctly, when there is a problem with the data.

Gesture Visualization: The user interface always shows the currently recognized gesture or a placeholder if no gesture is recognized. This results in the phenomenon that short touch gestures for example are only visualized briefly before they disappear again and there is no possibility to access the gesture and the transmitted properties again. Therefore, a systematic way of showing the last recognized gestures would improve the user interface.

Manage Filters: Once a filter is added to the filter pipeline, it would be helpful for developers to be able to activate and deactivate a filter without removing the filter from the filter pipeline and later adding the filter again, because this can be quite cumbersome. Also the possibility to edit filters would improve the user experience.

Sensor Settings: The UI provides functionality to start and stop the WebSocket connection and to adjust the size of the visualization. This settings could be extended, to provide more configuration of the prototyping device for example to really change the size of the sensor area that is read and transmitted from the fabric and filtered afterwards.

Keyboard Shortcuts: While gathering data for the training of the SVM model, interacting with the user interface and performing the gesture on the smart textile at the same time can be difficult. Therefore, it would help to implement simple keyboard shortcuts for the most important functionality, especially related to data capturing.

Advantages and Disadvantages of being Web-Based

In comparison to a C# application where the application can be built to a `.exe` file, the setup of the application is more complex because all the dependencies have to be installed and the database setup with the basic filter types has to be established. An advantage on the other hand the is that the application is platform independent. While the framework was developed on *MacOS High Sierra*, the participant installed the system on a desktop machine running *Windows 10*. Furthermore, due to the use of state of the art web development technologies, people with experience in web development will have little difficulty understanding the application and adding additional features, while they might have more trouble using C# or C++. As for the use of REST API calls to trigger specific actions, the framework is future-proof and extendible because REST APIs are widely used.

Conclusion

Although there are some details that can be improved and don't have the optimal user experience yet, the interviewed expert liked the implemented application. The overall working experience was positive, although not always as efficient as it could be.

Chapter 8

Discussion & Limitations

This chapter focuses on the interpretation and discussion of the results described in Chapter 7. First, the focus lies on the web-based framework for rapid prototyping. Afterwords, the evaluation of the proposed quadtree approach is discussed.

8.1 Web-Based Prototyping Approach

The implementation of the web-based prototyping framework was evaluated by adding a custom visualization and conducting an expert interview to find out about the experience of another developer who used the framework to implement real use cases. In this section the results of this evaluation are discussed and the limitations of the framework are summarized.

8.1.1 Custom Visualizations

A custom data visualization for conducting a study on applying different pressure levels on a bike handle with different fingers respectively, was implemented to evaluate how easy a developer can implement custom visualizations and which possibilities exist for visualizing the data. As described in Section 7.2.1, new visualizations can be easily added to the framework. The developed visualization only has to be included and configured in one file. The complexity of developing a custom visualization depends on the concept of the wanted visualization. Since the visualization is a part of the web user interface, developers can make use of all tools used for modern web development. The *React* framework facilitates building components with a lot of user interaction, although it can also complicate working with the framework if no prior experience with *React* or similar frontend libraries or frameworks is given.

The setup for the frontend development already enables the use of the CSS extension language SASS[1] which facilitates the styling of custom visualizations. Exporting SVG files of vector illustrations and directly copying the SVG into a *React* component allows to access and change all components of the SVG using CSS and *JavaScript*. Another possibility to achieve great data visualizations is to use a library like *D3.js*[2] to implement

[1]https://sass-lang.com/
[2]https://d3js.org/

dynamically generated SVG code with endless possibilities. Due to the rapid evolution of modern web development and the meanwhile good browser support of modern CSS features, there are hardly any design limits and new visualization can be added quite fast, depending on the experience of the developer in modern web development.

However, the implementation of the distinct visualization for a specific use case showed, that the location of the visualization in the user interface might not be optimal. If only the sensor data is shown, it makes sense to arrange the visualization as part of the filtering section because the applied filters have an immediate affect on the visualization and the visualization can be used to check if the applied filters deliver the correct result. In case of the hand visualization which was implemented to conduct a user study, the visualization should be better placed in a separate section.

8.1.2 Expert Interview

The expert interview proofed that the concept of the prototyping framework works and the developed application can be used by developers with different prototypes for different use cases. However, currently there are still some limitations which should be further studied and will be improved in future work, such as simplifying the addition of new filters and filter management in general, extended functionality for the user interface as well as the visualization of unfiltered data.

8.1.3 Limitations

While the evaluation of the prototyping framework generated mostly positive results, there are still some limitations which are to be studied and improved in future work.

User Interface: The user interface is currently limited to show one visualization at a time, it is not designed to show multiple different visualizations of the data simultaneously. Although it is not hard for developers to change this fact, it is at this point in time not part of the concept. Furthermore, only the filtered data is transmitted to the user interface and therefore there is no possibility to also visualize the raw data before the filter pipeline is applied.

User Management: Currently a big limitation is also that no system for user management and multiple projects per user is implemented. Therefore, one instance of the framework is only able to handle the configuration of a single prototype.

Shared Resources: A nice addition to the framework would be a concept where researchers can share resources developed for the framework. For example, in the expert interview the creation of custom filters was mentioned. The implementations of these filters could also be helpful for research projects of other developers or research groups. Therefore, a way of shearing resources between different people and projects would be an improvement.

8.2 QuadTree Approach

The developed quadtree method was evaluated as well in Chapter 7. This section first discusses the results of sampling a sensor with different resolutions. Second, the results of the quadtree performance are illustrated. Finally, the main limitations of this approach are listed.

8.2.1 Dynamic Resolutions

If a sensor is sampled with different resolutions, the results of the two approaches described in Section 6.4.2 were as expected. If multiple sensors are combined to one bigger region r, the value $D(j)$ for this region is above 0 even if all single sensors have a value $D(i) = 0$. When the described expression for estimation the single values $D(i)$ of a sensor region r is applied, the value of a region where no pressure is applied is 0. However, the downside of this approach is, as expected, that for big sensor regions applied pressure on single sensors does not make any difference and the value $D(j)$ is still 0. The size of the region, where pressure on single sensors doesn't have any influence anymore depends on the amount of pressure which is applied. In the evaluated example medium pressure was applied on 4 neighbouring sensors. The impact of this pressure was noticeable until a region size of 8×8 sensors. Therefore, the approach of estimation the single sensor values is not feasible and using a dynamic threshold is the better alternative.

8.2.2 Sampling using Quadtree Structure

The evaluation of sampling a 32×32 sensor with different amounts of active sensors revealed, that in the worst case scenario, where all active sensors are as far spread as possible, the performance of the quadtree approach is only better up to a number of 35 active sensors. If all active sensors are close together the quadtree approach outperforms the standard sampling technique up to 280 active sensors. In total the tested prototype has a number of 1024 single sensors. The same evaluation was made for the length of the transmitted data. Up to a number of 85 active sensors in the worst case scenario and 508 active sensors in an optimal case, the quadtree approach produces less data. These results might not seem very promising because the number of active sensors in the worst case scenario is very low and even in the best case scenario less then 50 % of all sensors can be active, for the quadtree sampling approach to perform better in terms of speed and data length. However, testing various possible use cases showed, the quadtree sampling method outperforms the standard sampling technique in both sampling speed and data length in all tested cases. The increase in sampling speed ranges from 24 % up to 474 %. The maximum transmitted data length is 458 bytes, which equals 44 % of the original 1030 bytes. This is due to the fact, that when performing real use cases, even if sitting on the textile sensor, a high number of sensors remain untouched. The maximum number of active sensors that was reached during the evaluation was 168. Therefore, it can be concluded, even though the number of active sensors must be rather low for the quadtree approach to succeed, the developed sampling method is still a success because in real use cases this is typically the case.

8.2.3 Limitations

Also the developed sampling approach using a quadtree structure is subject to various limitations which need to be considered for the implementation.

Sensor Size: At the time being, the implemented quadtree sampling approach can only handle quadratic sensors were the length of one side is a power of two. However, this is due to the fact, that no edge handling for other sizes was implemented and does not constitute a general limitation for the proposed quadtree sampling approach.

Region Size: The value $D(j)$ for a specified sensor region j increases with the number of sensors in j. Therefore, for regions from a certain size of n of sensors, the sampled value $D(j)$ is always be the maximum value 255. Thus, regions with and without pressure can no longer be distinguished. The number of sensors varies according to exact specifications of the used textile and can either be calculated or evaluated empirically. The thresholds have to be set accordingly in a way that regions where the maximum value is reached are always sampled in more detail up to a depth where regions with and without applied pressure are still discriminable. Thus, this behaviour is not fatal for the quadtree sampling approach, however, it decreases it's performance for very large sensors.

Signal Noise: The proposed sampling approach is only applicable for textiles with a high signal to noise ratio and little noise in the signal. If the sensors produce too much noise, all sensors are sampled with maximum detail which results in slowing performance and increasing the length of the transmitted data instead of being an improvement.

Chapter 9

Conclusions and Future Work

An easy to use extensible framework for rapid prototyping was developed which is tailored to the needs of working with smart textiles. This thesis provides a detailed description of the concept, architecture and implementation of the application. The major advantage for researchers and developers in the field of smart textiles is, that once they developed their prototype they can use the generic software and start prototyping use cases without writing custom software. Basic functionality for signal processing, gesture recognition and triggering of different actions once a specified gesture is recognized, is provided out of the box. Due to the flexible module-based architecture, custom features can easily be implemented.

Additionally, a novel approach for sampling the sensor data of a given textile sensor was developed. The approach uses a quadtree structure to reduce the number of sensors that need to be sampled. Only regions where pressure is applied are sampled in full detail. This technique is a major improvement over the default approach of sampling all sensors in every tested use case. Using the proposed quadtree technique, the sampling of large textile sensors is significantly faster and less data needs to be transmitted. An overview of the achieved results is summarized in Table 9.1.

Table 9.1: The evaluation of different uses cases shows significant performance improvements due to the implemented quadtree approach. Furthermore, less data needs to be transmitted. This table provides an overview of all tested use cases and summarizes the number of sensors with applied pressure, the increase in sampling speed and the decrease of the data that needs to be transmitted.

Use Case	**Active Sensors**	**Speedup**	**Data Length**	**Figure**
Fold	40	268 %	158 bytes	Figure 7.8
Pen Input	20	437 %	110 bytes	Figure 7.9
Touch (1 Finger)	36	474 %	104 bytes	Figure 7.10
Touch (2 Fingers)	48	188 %	200 bytes	Figure 7.11
Touch (4 Fingers)	92	85 %	308 bytes	Figure 7.12
Shear	108	97 %	290 bytes	Figure 7.13
Tailor Seat	168	24 %	458 bytes	Figure 7.14
Stand	88	105 %	266 bytes	Figure 7.15

In future work it would be interesting to evaluate the WebSocket performance in terms of speed. Since the streamed sensor data is fundamentally similar to video data, the use of the WebRTC protocol for the streaming of the sensor data to the client should be implemented and the performance could be compared to the WebSocket connection. Furthermore, a usability study of the user interface can be conducted, to increase usability of the UI and the user interface can be extended to account for showing multiple visualizations simultaneously.

Additionally, the use of the developed prototyping application on an IoT device can be interesting to evaluate. The goal thereby is to install the backend application including the web server directly on the IoT device. Therefore, no further computer is necessary. Due to the platform independence of the developed system this should be possible without major problems. However, the reading of the input data has to be updated since the sensor data is no longer transmitted via a serial connection.

Appendix A

Source Code of Quadtree Sampling

SensorQuadTree.ino:

```
#include <SPI.h> // library for the communication with SPI devices
#include "src/SensorReader/QuadTreeReader.h"
#include "src/DataSource/DefaultSource.h"
#include "src/Util/MuxBoardConfigurator.h"

namespace std {
    void __throw_length_error( char const* e )
    {
        Serial.println("Length Error :");
        Serial.println(e);
        while(1);
    }
}

// set up the speed, data order and data mode
// maximum speed: 10,5 MHz
// data order: least-significant bit first
// data mode: data sampled at falling edge
SPISettings settings(10500000, LSBFIRST, SPI_MODE1);

// pin for reading speed
const int timer = 11;

// resulting character array that is transmitted to an application
char* carray;

// set slave pins for the 2 PCBs
// one board is used for power and one board is used for reading
// which one does which job is not of importance
const int slaveSelectPin1 = 4; // left board
const int slaveSelectPin2 = 10; // top board

// analog pin used to read the data of the configured sensor (combination)
const int analogInPin = A0;

DataSource* dataSource = new DefaultSource(analogInPin);
MuxBoardConfigurator* muxBoardConfigurator = new MuxBoardConfigurator(settings,
    slaveSelectPin1, slaveSelectPin2);
const int rows = 32;
const int cols = 32;

SensorReader* sensorReader = new QuadTreeReader(dataSource, muxBoardConfigurator,
    cols, rows);

void setup() {
    // Both select pins are set to OUTPUT
    // because they are only used to decide which of the 2 boards is currently active
    // HIGH means they are not listening to signals
    pinMode(slaveSelectPin1, OUTPUT);
    pinMode(slaveSelectPin2, OUTPUT);
    digitalWrite(slaveSelectPin1, HIGH);
    digitalWrite(slaveSelectPin2, HIGH);

    pinMode(timer, OUTPUT);
    digitalWrite(timer, HIGH);

    // Serial Port Initialization
    Serial.begin(115200);

    // initialize SPI
    SPI.begin();
}

void loop() {
    digitalWrite(timer, LOW);
    carray = sensorReader->read();
    digitalWrite(timer, HIGH);

    // send array data with all values to Serial Port
    Serial.write(carray, sensorReader->getDataLength() + 6);

    delay(10);
}
```

SensorReader.h:

```
#ifndef SENSOR_READER_H
#define SENSOR_READER_H

/**
 * INTERFACE for reading data from sensor
 */
class SensorReader {

  public:
       virtual char* read() = 0;
       virtual unsigned short getDataLength() = 0;
};

#endif
```

MuxSensorReader.h:

```
#ifndef MUX_SENSOR_READER_H
#define MUX_SENSOR_READER_H
#include "SensorReader.h"
#include "../DataSource/DataSource.h"
#include "../Util/MuxBoardConfigurator.h"

/**
 * INTERFACE for reading data from sensor with mux boards
 */
class MuxSensorReader : public SensorReader {

    protected:
        DataSource* dataSource;
        MuxBoardConfigurator* configurator;

    public:
        MuxSensorReader(DataSource* dataSource, MuxBoardConfigurator* configurator);
};

#endif
```

MuxSensorReader.cpp:

```
#include "MuxSensorReader.h"

MuxSensorReader::MuxSensorReader(DataSource* dataSource, MuxBoardConfigurator*
     configurator) {
    this->dataSource = dataSource;
    this->configurator = configurator;
}
```

QuadTreeReader.h:

```
#ifndef QUADTREE_READER_H
#define QUADTREE_READER_H
#include <vector>
#include "MuxSensorReader.h"
#include "../DataSource/DataSource.h"
#include "../Util/MuxBoardConfigurator.h"
#include "../QuadTree/Node.h"
#include "../Util/Bounds.h"
#include "../QuadTree/Predicates/Predicate.h"

class QuadTreeReader : public MuxSensorReader {

    private:
        std::vector<char> carray;
        char sensorValue;
        char rows;
        char cols;
        unsigned short leaveNodeCounter;
        Node* rootNode;
        Predicate* predicate;
        void setDataHeader();
        int calculateMuxValue(int value);
        void readTreeAndFillArray();
        void addNodeValue(Node* node);

    public:
        QuadTreeReader(DataSource* dataSource, MuxBoardConfigurator* configurator,
    char cols, char rows);
        ~QuadTreeReader();
        virtual char* read();
        virtual unsigned short getDataLength();
};

#endif
```

QuadTreeReader.cpp:

```
#include "QuadTreeReader.h"
#include "../QuadTree/Node.h"
#include "../QuadTree/Predicates/DefaultQuadTreePredicate.h"

QuadTreeReader::QuadTreeReader(DataSource* dataSource, MuxBoardConfigurator*
    configurator, char cols, char rows) : MuxSensorReader(dataSource, configurator)
    {
    this->rows = rows;
    this->cols = cols;
    this->leaveNodeCounter = 0;
    this->predicate = new DefaultQuadTreePredicate(15);
}

QuadTreeReader::~QuadTreeReader() {
    delete predicate;
}

void QuadTreeReader::setDataHeader() {
    // write Header for the array transmission

    unsigned short dataSize = getDataLength();
    char dataSizeByte1 = (dataSize & 0xFF00) >> 8;
    char dataSizeByte2 = dataSize & 0x00FF;

    char header[6] = {0xDF, rows, cols, 0x01, dataSizeByte1, dataSizeByte2};
    carray.insert(carray.begin(), header, header+6);
}

char* QuadTreeReader::read() {

    // build QuadTree
    rootNode = new Node(new Bounds(0, 0, cols, rows), 0);
    rootNode->expand(dataSource, configurator, predicate);

    // read QuadTree
    carray.clear();
    leaveNodeCounter = 0;
    addNodeValue(rootNode);
    setDataHeader();
    delete rootNode;
    return (&carray[0]);
}

void QuadTreeReader::addNodeValue(Node* node) {
    if(node->isLeaf()) {
        leaveNodeCounter++;
        unsigned short depth = node->depth;
        carray.push_back(depth);
        if(depth == 5) {
            carray.push_back(node->data);
        } else {
            carray.push_back(0);
        }
    } else {
        for(Node* child : node->children) {
            addNodeValue(child);
        }
    }
}

unsigned short QuadTreeReader::getDataLength() {
    // node data + depth of all nodes = 2 Bytes per node
    return leaveNodeCounter * 2;
};
```

Node.h:

```
#ifndef NODE_H
#define NODE_H
#include <vector>
#include "../DataSource/DataSource.h"
#include "../Util/MuxBoardConfigurator.h"
#include "../Util/Bounds.h"

class Predicate; // forward declaration because of circular references

class Node {

    private:
        void fill(DataSource* dataSource, MuxBoardConfigurator* configurator);
        int calculateMuxValue(int value);

    public:
        std::vector<Node*> children;
        char data;
        unsigned short depth;
        Bounds* bounds;
        Node(Bounds* bounds, char depth);
        ~Node();
        bool isLeaf() { return children.empty(); };
        void expand(DataSource* dataSource, MuxBoardConfigurator* configurator,
    Predicate* predicate);
};

#endif
```

Node.cpp:

```
#include "Node.h"
#include "Predicates/Predicate.h"

Node::Node(Bounds* bounds, char depth) {
    this->bounds = bounds;
    this->depth = depth;
}

Node::~Node() {
    delete bounds;

    for (std::vector<Node*>::iterator it = children.begin(); it != children.end();
    ++it)
    {
        delete (*it);
    }
    children.clear();
}

void Node::fill(DataSource* dataSource, MuxBoardConfigurator* configurator) {

    int muxPowerValue = calculateMuxValue(bounds->endX - bounds->startX);
    unsigned int muxPower = muxPowerValue << bounds->startX;
    configurator->configureMuxBoard(muxPower, configurator->slaveSelectPin1);

    int muxReadValue = calculateMuxValue(bounds->endY - bounds->startY);
    unsigned int muxRead = muxReadValue << bounds->startY;
    configurator->configureMuxBoard(muxRead, configurator->slaveSelectPin2);

    this->data = dataSource->read();
}

void Node::expand(DataSource* dataSource, MuxBoardConfigurator* configurator,
    Predicate* predicate) {

    // read value for Node
    fill(dataSource, configurator);

    // check if Node shoud be split
    if(predicate->test(this)) {

        for(int i = 0; i < 4; i++) {
            Bounds* currentBound = bounds->getQuarter(i);
            Node* child = new Node(currentBound, depth + 1);
            children.push_back(child);
            child->expand(dataSource, configurator, predicate);
        }
    }
}

int Node::calculateMuxValue(int value) {
    const int base = 2;
    int returnValue = 1;
    for(int step = 0; step < value; step++) {
        returnValue *= 2;
    }
    returnValue -= 1;
    return returnValue;
}
```

Predicate.h:

```
#ifndef PREDICATE_H
#define PREDICATE_H
#include "../Node.h"

class Predicate {

    public:
       virtual bool test(Node* node) = 0;

};

#endif
```

DefaultQuadTreePredicate.h:

```
#ifndef DEFAULT_QUADTREE_PREDICATE_H
#define DEFAULT_QUADTREE_PREDICATE_H
#include "Predicate.h"

class DefaultQuadTreePredicate : public Predicate {

    private:
        char threshold;
        bool isSingleSensor(Node* node);
        bool isBelowThreshold(Node* node);

    public:
        DefaultQuadTreePredicate(char threshold);
        bool test(Node* node);
        char getValueForDepth(Node* node);

};

#endif
```

DefaultQuadTreePredicate.cpp:

```
#include "DefaultQuadTreePredicate.h"

DefaultQuadTreePredicate::DefaultQuadTreePredicate(char threshold) {
    this->threshold = threshold;
}

bool DefaultQuadTreePredicate::test(Node* node) {
    return !isSingleSensor(node) && !isBelowThreshold(node);
}

bool DefaultQuadTreePredicate::isSingleSensor(Node* node) {
    return node->bounds->isSingleEntry();
}

bool DefaultQuadTreePredicate::isBelowThreshold(Node* node) {
    return node->data < (getValueForDepth(node) + threshold);
}

char DefaultQuadTreePredicate::getValueForDepth(Node* node) {
    switch(node->depth) {
        case 0: return 40;
        case 1: return 13;
        case 2: return 5;
        case 3: return 3;
        case 4: return 2;
        case 5: return 0;
    }
}
```

DataSource.h:

```
#ifndef DATA_SOURCE_H
#define DATA_SOURCE_H

/**
 * INTERFACE for accessing sensor data
 */
class DataSource {

    public:
        virtual char read() = 0;
};

#endif
```

DefaultSource.h:

```
#ifndef DEFAULT_SOURCE_H
#define DEFAULT_SOURCE_H
#include "DataSource.h"
```

```

class DefaultSource : public DataSource {

    private:
        int analogInPin;

    public:
        DefaultSource(int analogInPin);
        char read();

};

#endif
```

DefaultSource.cpp:

```
#include "DefaultSource.h"
#include <spi.h>

DefaultSource::DefaultSource(int analogInPin) {
    this->analogInPin = analogInPin;
}

char DefaultSource::read() {
    analogReadResolution(8);
    return analogRead(analogInPin);
}
```

Bounds.h:

```
#ifndef BOUNDS_H
#define BOUNDS_H
#include <spi.h>

class Bounds {

    public:
        unsigned short startX;
        unsigned short startY;
        unsigned short endX;
        unsigned short endY;

        Bounds();
        Bounds(unsigned short startX, unsigned short startY, unsigned short endX,
    unsigned short endY);
        bool isSingleEntry();
        void setBounds(unsigned short startX, unsigned short startY, unsigned short
    endX, unsigned short endY);
        Bounds* getQuarter(char index);
};

#endif
```

Bounds.cpp:

```cpp
#include "Bounds.h"

Bounds::Bounds() {
    this->startX = 0;
    this->startY = 0;
    this->endX = 1;
    this->endY = 1;
}

Bounds::Bounds(unsigned short startX, unsigned short startY, unsigned short endX,
     unsigned short endY) {
    this->startX = startX;
    this->startY = startY;
    this->endX = endX;
    this->endY = endY;
}

bool Bounds::isSingleEntry() {
    return (endX - startX) == 1 && (endY - startY) == 1;
}

void Bounds::setBounds(unsigned short startX, unsigned short startY, unsigned short
     endX, unsigned short endY) {
    this->startX = startX;
    this->startY = startY;
    this->endX = endX;
    this->endY = endY;
}

Bounds* Bounds::getQuarter(char index) {
    const int width = (endX - startX) / 2;
    const int height = (endY - startY) / 2;
    switch(index) {
        case 0:
            return new Bounds(startX, startY, startX + width, startY + height);
        case 1:
            return new Bounds(startX + width, startY, endX, startY + height);
        case 2:
            return new Bounds(startX, startY + height, startX + width, endY);
        case 3:
            return new Bounds(startX + width, startY + height, endX, endY);
    }
}
```

MuxBoardConfigurator.h:

```cpp
#ifndef MUX_BOARD_CONF_H
#define MUX_BOARD_CONF_H
#include <spi.h>

```

```
class MuxBoardConfigurator {

    private:
        char maskingBytes(char value);
        SPISettings settings;

    public:
        MuxBoardConfigurator(SPISettings settings, int slaveSelectPin1, int
    slaveSelectPin2);
        void configureMuxBoard(unsigned int value, int slavePin);
        int slaveSelectPin1;
        int slaveSelectPin2;
};

#endif
```

MuxBoardConfigurator.cpp:

```
#include "MuxBoardConfigurator.h"

MuxBoardConfigurator::MuxBoardConfigurator(SPISettings settings, int slaveSelectPin1
    , int slaveSelectPin2) {
    this->settings = settings;
    this->slaveSelectPin1 = slaveSelectPin1;
    this->slaveSelectPin2 = slaveSelectPin2;
}

void MuxBoardConfigurator::configureMuxBoard(unsigned int value, int slavePin) {
    char mux1 = maskingBytes(value & 0x000000FF);
    char mux2 = maskingBytes((value & 0x0000FF00) >> 8);
    char mux3 = maskingBytes((value & 0x00FF0000) >> 16);
    char mux4 = maskingBytes((value & 0xFF000000) >> 24);
    char imux1 = ~mux1;
    char imux2 = ~mux2;
    char imux3 = ~mux3;
    char imux4 = ~mux4;

    SPI.beginTransaction(this->settings);
    // start listenin with specified board (slavePin)
    digitalWrite(slavePin, LOW);

    // send mux configurations
    SPI.transfer(imux4);
    SPI.transfer(mux4);
    SPI.transfer(imux3);
    SPI.transfer(mux3);
    SPI.transfer(imux2);
    SPI.transfer(mux2);
    SPI.transfer(imux1);
    SPI.transfer(mux1);

    // stop listenin with specified board (slavePin)
    digitalWrite(slavePin, HIGH);
    SPI.endTransaction();
}
```

```

/**
* map to correct mux input on hardware level
*/
char MuxBoardConfigurator::maskingBytes(char value) {
    char maskedValue = 0x00;

    if ((value & 0x01) == 0x01) {
        maskedValue = maskedValue | 0x08; }
    if ((value & 0x02) == 0x02) {
        maskedValue = maskedValue | 0x04; }
    if ((value & 0x04) == 0x04) {
        maskedValue = maskedValue | 0x02; }
    if ((value & 0x08) == 0x08) {
        maskedValue = maskedValue | 0x01; }
    if ((value & 0x10) == 0x10) {
        maskedValue = maskedValue | 0x10; }
    if ((value & 0x20) == 0x20) {
        maskedValue = maskedValue | 0x20; }
    if ((value & 0x40) == 0x40) {
        maskedValue = maskedValue | 0x40; }
    if ((value & 0x80) == 0x80) {
        maskedValue = maskedValue | 0x80; }

    return maskedValue;
}
```

Appendix B

DVD Contents

Format: DVD+RW, Single Layer, 4.7 GB

B.1 PDF

Path: /

Schuetz2018.pdf	Thesis

B.2 Source Code

Path: /ProjectSourceCode

Framework.zip	Source Code of Web-Based Prototyping Framework
Quadtree.zip	Source Code of Quadtree Sampling Approach

References

Literature

[1] S. Aluru. “Quadtrees and Octrees.” In: *Handbook of Data Structures and Applications*. Ed. by D. P. Mehta and S. Sahni. Chapman & Hall/CRC, 2005. Chap. 19 (cit. on p. 40).

[2] B. B. Chaudhuri. “Applications of Quadtree, Octree, and Binary Tree Decomposition Techniques to Shape Analysis and Pattern Recognition”. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7.6 (1985), pp. 652–661 (cit. on pp. 39, 41).

[3] J. Cheng et al. “Recognizing Subtle User Activities and Person Identity with Cheap Resistive Pressure Sensing Carpet”. In: *Proceedings of the International Conference on Intelligent Environments – IE ’14*. 2014, pp. 148–153 (cit. on p. 2).

[4] G. R. Conde Márquez, H. J. Escalante, and L. E. Sucar. “Simplified Quadtree Image Segmentation for Image Annotation”. In: *Proceedings of the Automatic Image Annotation and Retrieval Workshop – AIAR ’10*. 2011, pp. 24–34 (cit. on pp. 39, 41).

[5] K. Gönner et al. “Precision Fabric Production in Industry”. In: *Smart Textiles*. Ed. by S. Schneegass and O. Amft. Springer International Publishing, 2017. Chap. 2, pp. 17–30 (cit. on pp. 2, 4).

[6] X. Guo and J. Liu. “Real-time Rendering of Large-scale Terrain Based on OpenCL”. In: *Proceedings of the International Conference on Information and Automation – ICIA ’16*. 2016, pp. 1227–1231 (cit. on pp. 39, 41).

[7] J. Haladjian et al. “A Smart Textile Sleeve for Rehabilitation of Knee Injuries”. In: *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the International Symposium on Wearable Computers – UbiComp/ISWC ’17*. 2017, pp. 49–52 (cit. on p. 1).

[8] J. Hirt et al. “Using Quadtrees for Realtime Pathfinding in Indoor Environments”. In: *Proceedings of the International Conference on Research and Education in Robotics – EUROBOT ’10*. 2011, pp. 72–78 (cit. on pp. 39, 41).

[9] P. Holleis et al. “Evaluating capacitive touch input on clothes”. In: *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services – MobileHCI ’08*. 2008, pp. 81–90 (cit. on p. 1).

[10] T. Karrer et al. “Pinstripe”. In: *Proceedings of the International Conference on Human Factors in Computing Systems – CHI ’11*. 2011, pp. 1313–1322 (cit. on pp. 1, 5, 6).

[11] G. Kirichek and V. Kurai. “Implementation quadtree method for comparison of images”. In: *Proceedings of the International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering – TCSET ’18*. 2018, pp. 129–132 (cit. on pp. 39, 41).

[12] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas. “Supervised Machine Learning: A Review of Classification Techniques”. In: *Emerging Artificial Intelligence Applications in Computer Engineering*. Ed. by J. Breuker et al. IOS Press, 2007. Chap. 1, pp. 3–24 (cit. on p. 28).

[13] J. Leong et al. “proCover: Sensory Augmentation of Prosthetic Limbs Using Smart Textile Covers”. In: *Proceedings of the Annual Symposium on User Interface Software and Technology – UIST ’16*. 2016, pp. 335–346 (cit. on p. 6).

[14] W. Li et al. “Smart Mat System with Pressure Sensor Array for Unobtrusive Sleep Monitoring”. In: *Proceedings of the International Conference of the IEEE Engineering in Medicine and Biology Society – EMBC ’17*. 2017, pp. 177–180 (cit. on pp. 1, 2, 7).

[15] P. Lindstrom et al. “Real-Time, Continuous Level of Detail Rendering of Height Fields”. In: *Proceedings of the International Conference on Computer Graphics and Interactive Techniques – SIGGRAPH ’96*. 1996, pp. 109–118 (cit. on pp. 39, 41).

[16] K. Ma and R. Sun. “Introducing Websocket-Based Real-Time Monitoring System for Remote Intelligent Buildings”. *International Journal of Distributed Sensor Networks* 9.12 (2013) (cit. on p. 7).

[17] Z. F. Muhsin et al. “Improved Quadtree Image Segmentation Approach to Region Information”. *The Imaging Science Journal* 62.1 (2014), pp. 56–62 (cit. on pp. 39, 41).

[18] A. S. Nittala et al. “Multi-Touch Skin: A Thin and Flexible Multi-Touch Sensor for On-Skin Input”. In: *Proceedings of the International Conference on Human Factors in Computing Systems – CHI ’18*. 2018, 33:1–33:12 (cit. on pp. 7, 8).

[19] R. Pajarola. “Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation”. In: *Proceedings of the Conference on Visualization – VIS ’98*. 1998, pp. 19–26 (cit. on pp. 39, 41).

[20] R. Pajarola. *Overview of Quadtree-based Terrain Triangulation and Visualization*. Tech. rep. 02-01. University of California, Irvine: Department of Information & Computer Science, 2002 (cit. on pp. 39, 41).

[21] P. Parzer et al. “FlexTiles: A Flexible, Stretchable, Formable, Pressure-Sensitive, Tactile Input Sensor”. In: *Proceedings of the International Conference Extended Abstracts on Human Factors in Computing Systems – CHI EA ’16*. 2016, pp. 3754–3757 (cit. on pp. 1, 6, 51).

[22] P. Parzer et al. “SmartSleeve: Real-time Sensing of Surface and Deformation Gestures on Flexible, Interactive Textiles, using a Hybrid Gesture Detection Pipeline”. In: *Proceedings of the Annual Symposium on User Interface Software and Technology – UIST ’17*. 2017, pp. 565–577 (cit. on pp. 1, 6, 28).

[23] I. Poupyrev et al. “Project Jacquard: Interactive Digital Textiles at Scale”. In: *Proceedings of the International Conference on Human Factors in Computing Systems – CHI ’16*. 2016, pp. 4216–4227 (cit. on pp. 1, 5).

[24] H. Samet. “The Quadtree and Related Hierarchical Data Structures”. *ACM Computing Surveys* 16.2 (1984), pp. 187–260 (cit. on pp. 40, 41).

[25] S. Schneegass and O. Amft. “Introduction to Smart Textiles”. In: *Smart Textiles*. Ed. by S. Schneegass and O. Amft. Springer International Publishing, 2017. Chap. 1, pp. 1–15 (cit. on pp. 1, 4).

[26] S. Schneegass et al. “SimpleSkin: Towards Multipurpose Smart Garments”. In: *Proceedings of the International Symposium on Wearable Computers – ISWC ’15*. 2015, pp. 241–244 (cit. on p. 1).

[27] M. Sundholm et al. “Smart-mat: Recognizing and Counting Gym Exercises with Low-cost Resistive Pressure Sensing Matrix”. In: *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing – UbiComp ’14*. 2014, pp. 373–382 (cit. on pp. 2, 7).

[28] D. Wang et al. “AnyControl - IoT based Home Appliances Monitoring and Controlling”. In: *Proceedings of the International Computer Software and Applications Conference – COMPSAC ’15*. 2015, pp. 487–492 (cit. on p. 7).

[29] Q. Wang et al. “Zishi: A Smart Garment for Posture Monitoring”. In: *Proceedings of the International Conference Extended Abstracts on Human Factors in Computing Systems - CHI EA ’16*. 2016, pp. 3792–3795 (cit. on p. 1).

[30] A. Yahja et al. “Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments”. In: *Proceedings of the International Conference on Robotics and Automation – ICRA ’98*. 1998, pp. 650–655 (cit. on pp. 39, 41).

[31] S. H. Yoon et al. “TIMMi: Finger-worn Textile Input Device with Multimodal Sensing in Mobile Interaction”. In: *Proceedings of the International Conference on Tangible, Embedded, and Embodied Interaction – TEI ’14*. 2015, pp. 269–272 (cit. on p. 1).

[32] Z. Zhang and N. Zhang. “A LOD Algorithm Based on Out-of-Core for Large Scale Terrain Rendering”. In: *Proceedings of the International Conference on Mechatronic Sciences, Electric Engineering and Computer – MEC ’13*. 2013, pp. 2168–2171 (cit. on pp. 39, 41).

[33] B. Zhou and P. Lukowicz. “Textile Pressure Force Mapping”. In: *Smart Textiles*. Ed. by S. Schneegass and O. Amft. Springer International Publishing, 2017. Chap. 3, pp. 31–47 (cit. on p. 6).

Online sources

[34] *Arduino Reference: analogRead()*. URL: https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/ (visited on 04/19/2018) (cit. on p. 45).

[35] *Arduino Reference: analogReference()*. URL: https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/ (visited on 04/19/2018) (cit. on p. 45).

[36] *Can I Use: ES6 classes*. URL: https://caniuse.com/#search=es6 (visited on 04/04/2018) (cit. on p. 14).

[37] *Draft: JSX Specification*. 2014. URL: https://facebook.github.io/jsx/ (visited on 04/04/2018) (cit. on p. 14).

[38] *libsvm-js*. URL: https://www.npmjs.com/package/libsvm-js (visited on 04/06/2018) (cit. on p. 28).

[39] *Yarn Package Manager*. URL: https://yarnpkg.com/lang/en/ (visited on 04/04/2018) (cit. on p. 15).

Check Final Print Size

— Check final print size! —

width = 100mm
height = 50mm

— Remove this page after printing! —