Utilizing User-Drawn Input for Web Prototyping

Elisabeth Grömer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 16, 2017

Elisabeth Grömer

Contents

Abstract

Web developers and designers often face the challenge to communicate their ideas to others, such as colleagues or customers. Frequently, they use rough pen and paper drawings to present their concepts. These drawings are afterwards the initial point for developing prototypes which are again shown to each involved party. A combination of both steps would probably expedite the design and development process.

This thesis take a look at a possible solution for this problem, including several methods to recognize drawn input and converting it into code. The Web Components technology is considered because of its ability to preserve the component based character of web development, even if this peculiarity is not familiar to the user. Furthermore, the tool developed in the course of this thesis is evaluated to verify whether this approach is feasible or not.

Kurzfassung

Webentwickler und -designer stehen oft vor der Herausforderung, ihre Ideen für andere verständlich darzulegen, wie zum Beispiel für Kollegen oder Kunden. Häufig verwenden sie grobe Zeichen auf Papier um ihre Konzepte zu präsentieren. Diese Zeichnungen sind später der Ausgangspunkt um Prototypen dieser zu entwickeln, welche anschließend wiederum präsentiert werden. Eine Kombination dieser beiden Schritte könnte die Entwicklungsgeschwindigkeit einer Website erhöht werden.

Diese Arbeit befasst sich mit einer möglichen Lösung dieses Problems, einschließlich verschiedener Methoden um gezeichnete Eingaben in funktionierenden Code umzuwandeln. Die Web Components-Technologie wird wegen ihrer Fähigkeit zur Erhaltung des komponentenbasierten Charakters der Webentwicklung näher betrachtet. Außerdem wird das Tool, welches im Zusammenhang mit der Arbeit entwickelt wurde, evaluiert um die Umsetzbarkeit der Herangehensweise darzulegen.

Chapter 1

Introduction

These days, creating prototypes while developing something on the Web is common. These prototyping process can be used in almost each step of development, for single items as well as a full website in its entirety. Frequently, visualizations, rough designs or sketches are produced during meetings, which can reveal how something should look. Additionally, these first drafts get afterwards converted into code, to demonstrate their functionality. This prototypes are then presented to a team, a customer or someone else, who can now envision how the resulting product would appear. However, this two-step process is often time-consuming, especially when multiple stakeholders are involved.

This process of preparing a design and creating a functional prototype afterwards can be time consuming, especially if there is a range of functionalities. A fast, project independent and easy to use prototyping tool should combine the steps of creating a first graphical draft with the integration of its functionality.

Over the last few years many different prototyping tools for web prototyping have been developed. Mostly, they focus on the design aspect of web projects, requiring the use of any type of graphical design software beforehand to create first drafts of a possible layout. These drafts are then used to create first prototypes of a website, using one complete image for every function the developer intends to implement.

1.1 Goal

To find a possible remedy for the disadvantages of current prototyping processes this thesis is concerned with the following question:

> How is the usage of user-drawn gestures feasible for web prototyping and can it enhance the speed of prototyping?

The goal is to identify a way of how user-drawn input is feasible for web prototyping, to circumvent multiple steps of creating a design and adding functionality. Therefore it should be possible to combine the tasks of design and applying function. Additionally, the speed of creating a prototype shall be measured to evaluate the usability and effectiveness of using drawn input. Furthermore the accuracy of the chosen gestures for each element should be evaluated, to identify potential issues.

1.2 Structure

This thesis is structured in six chapters, including theoretical and applied sections. Reading the thesis from the start to the end is recommended, however it is also possible to focus on single chapters. Chapter 2 clarifies the terminology used in the thesis, to ensure that all used terms are comprehensible to every reader and no misunderstandings are originated. In Chapter 3 the theoretical classification of recognition algorithms is presented, including the basic classifications and multiple common technologies. Chapter 4 emphasizes the theoretical background of the thesis. Technologies for achieving the aim of this thesis are discussed and compared. Chapter 5 is concerned with the practical implementation of the thesis project. It describes all made technical decisions needed for development and furthermore provides insights into implementation details. Chapter 6 contains the evaluation of the thesis project. It exposes the performance of all developed gestures and an analysis of the tool itself. Additionally it emphasizes the results corresponding with the thesis' goal. Concluding, Chapter 7 gives a summary concerning the project, its outcome and an outlook to future work.

Chapter 2

Terminology

Multiple terms used in this thesis could be misleading due to their different usage in distinct fields. Therefore, the most important or varying terms are explained to avoid misunderstandings. Additionally, each reader should be able to understand the terms in the way they are used in this thesis, even if they are not familiar with the topic. Furthermore, some terms have a distinct meaning in association with the field of gesture recognition and web prototyping.

2.1 Web Application

A *web application* is present in a web browser. It consists of HTML and CSS for markup and styling in combination with JavaScript to provide functionality and behavior. Additionally, all JavaScript based libraries and frameworks can be included for additional features. The Web Components technology as well as the recognition algorithm uses JavaScript to enable the gestures and the implementation of custom elements.

2.2 Recognition

The *recognition* is conducted after a user has finished drawing a gesture and started the process to convert it into an element. During the recognition process, the points of a drawn gesture are interpreted and used to identify the corresponding element.

2.3 Gestures

The term *gesture* has multiple definitions, depending on their field of application. In this context, the term is defined as shapes which are drawn by using a mouse, pen or a finger. It is used as an abbreviation for *user-drawn* *input*. To enable the recognition, the user has to draw a figure, shape or symbol, which is named a gesture.

2.4 Web Components

Web Components are a combination of different technologies which are frequently used together. Therefore mostly the term Web Components is used to refer to all of them. Once exclusively individual parts of the technology are mentioned, the term is not used, to differentiate between all and only single components. In Chapter 4 all parts of the Web Components technology are described in detail.

2.5 Prototype, Mockup, Wireframe

The three terms *prototype, mockup* and *wireframe* are frequently used synonymously on the Web. However, all three terms sound considerable similar by their definition, but they differ substantial in their purpose and usage.

Prototype

A *prototype* simulates user interaction with the resulting product. It allows the user to experience the content and interactions with everything implemented in the prototype. Additionally, these main interactions should already interact similar to the final product.

Mockup

A *mockup* is a middle to high fidelity, static design representation [27]. Frequently, mockups are design drafts or the final design itself. It is used to display the structure of information and visualizes the content. The main purpose of mockups is to demonstrate the visual element of the project, not the functional aspects.

Wireframe

A *wireframe* is a low fidelity representation of a design [27]. They display the conceptual layout of the website and how a user can interact with anything located on this website.

2.6 Context Menu

The *context menu* is a technology implemented in most programs and thereby also in web browsers. It is usually opened by right-clicking on any area in the browser window. Furthermore the point where the click is performed specifies the content of the context menu. In web development, the default menu can also be substituted by a particular, customized menu.

Chapter 3

Recognition

The area of application for recognition algorithms ranges from handling abstract gestures, over complete sketches to complex handwritten texts. Various approaches are available to serve these tasks, differing in their advantages, disadvantages and most important their particular purposes. All of these approaches have underlying mathematical calculations, which determine the complexity as well as the time and space requirements of the algorithms. Furthermore, they specify their possible usage, due to hardware restrictions which may occur concerning the mentioned costs.

Classification of algorithms is an intricate problem, concerning the circumstance that most available technologies are not mutually exclusive. Therefore, multiple approaches are frequently combined to enhance the resulting algorithm. Moreover, a single technology may not be sufficient to fulfill all required tasks of one problem statement. For that reason, it is sometimes difficult to classify an algorithm in one distinct category. Instead, the identification of a set of technologies is possible, which consists of multiple approaches to fulfill one purpose jointly.

Nevertheless, the classification of algorithms is still suitable to distinguish between the main approaches underlying each algorithm, especially to identify the most appropriate for one specific area of application.

3.1 Classification of Algorithms

Various approaches are available to recognize shapes. Most of them can be assigned to at least one category of recognizers, some to more than one, especially when they combine different technologies to improve the recognition results or need to fulfill highly specific tasks which could not be achieved by a single approach. The available algorithms can be divided in three categories, which are mentioned by Hammond and Paulson [4] as follows.

3.1.1 Motion-based Recognition

Motion-based recognition concerns itself primarily with the path of the stroke [4]. More precisely, the characteristics of a drawn shape are more important than the actual appearance of this shape. Among others, these characteristics include properties such as the stroke order, the stroke direction, an possible arc angle. Therefore, to enable the recognition process, these gestures often need to be drawn in a specific way, influencing the natural drawing style of the user. However, some algorithms revoke these constraints, to enable the drawing of these shapes without a long training period.

Furthermore, some algorithms need a particular training, to be able to recognize different gestures. The training sets required to perform this training can be significant, depending on the specific algorithm. However, these algorithms are known for achieving good results, frequently corresponding with the amount of training data.

3.1.2 Appearance-based Algorithms

These algorithms are described by Hammond and Paulson as follows [4]:

> *Appearance-based algorithms focus primarily on what a sketched shape looks like; the timing and ordering of points are usually ignored, with the strokes translated into their bitmap counterparts before performing recognition.*

To identify shapes, template matching approaches are used frequently. Template matching is also used by motion-based algorithms. However they utilize a more basic technology to calculate the matching template for a shape, considering each single stroke. The appearance-based algorithms use a more advanced manifestation to identify the shape by its whole appearance, not by the single strokes. The main disadvantage of these appearance-based algorithms is that they cannot identify shapes which are drawn differently than their template. Shapes which can be drawn in variations, for example rotated by a value significantly deviating from the template requires a additional template for each possible rotation. Therefore, template sets can grow fast, which increases the storage and time costs for all gestures and their recognition.

3.1.3 Geometric-based Recognizers

Geometric-based recognizers attempt to use geometric formulas to describe primitives [4]. This approach is only usable on some basic primitives, which can unambiguously be described by geometric formulas, such as lines, curves, ellipses or combinations of these basic shapes. The available shapes are depending on the chosen algorithm, since each of them support a different set of primitives.

3.2 Common Recognition Approaches

Within the three main classifications, various algorithms were developed in the last years. Many of them can be related to one or more of the above mentioned categories. Furthermore, multiple algorithms can be divided in some subcategories based upon their basic functionality and their underlying technology. The most common approaches used in gesture recognition are in [11] categorized as follows.

3.2.1 Feature-based Classifiers

Feature-based algorithms define each gesture by a specific set of features. These features are mathematical expressions of a shapes peculiarities. To recognize an input gesture, all features of the complete feature-set are calculated for the gesture and the results are compared to the features of a template gesture. One of the main representatives of these algorithms is the Rubine recognizer GRANDMA [8], which defines 13 features including:

- the length and the angle of the bounding box diagonal,
- the distance between the first and the last point,
- the total gesture length,
- the duration of the gesture.

These features are defined for each recognition algorithm separately, however following Rubine [8], they should fulfill the criteria that:

> *A small change in the input should result in a correspondingly small change in each feature.*

Another recognizer using a feature-based approach developed by Cho [3] focuses on the recognition and segmentation of handwritten Korean scripts. It uses nine features to describe each gesture and has to deal with segmentation difficulties in case of distinguishing between single symbols. Frequently, the segmentation of single letters is done by measuring the empty space between them. However, some Korean symbols include spaces as part of their appearance, therefore this method could not be used alone. Hence, the algorithm was adapted to distinguish between symbols using the time the user needs after finishing one symbol until the start of the next one.

3.2.2 Hidden Markov Model

Using a Hidden Markov Model, each gesture is described through states. These states are not visible, rather they emit features for each state. Thereby

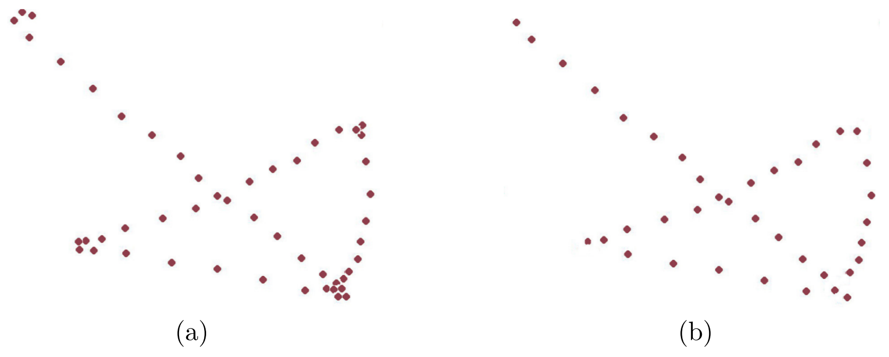


Figure 3.1: Improvement of gesture before recognition. (a) shows the raw gesture, (b) visualizes the revised gesture. Published in [1].

gestures can be recognized, such as by the algorithm of Anderson, Bailey and Skubic [1] which defines four steps to include a HMM in a gesture recognizer:

1. preprocessing,
2. features,
3. HMM parameters,
4. optional post-processing for removing false alarm or additional recognition steps.

The recognition algorithm by Anderson, Bailey and Skubic [1] includes in the preprocessing some basic steps which are used by numerous algorithms, such as removing jitter and errors. Additionally, the gestures are improved by removing points which are nearly identical and resampling the points to a fixed distance (see Figure 3.1). The features and HMM parameters are defined corresponding to the used set of gestures, similar to the feature-based approach. Post-processing steps are optional and again frequently implemented corresponding to the gesture set, because they highly depend on the gestures and their selected features.

3.2.3 Neural Networks

Neural networks are commonly used for gesture recognition purposes. A neural network is described by Shiffman [26] as a network of individual, simple nodes, which read an input, process it, and generate an output. The network process the information collectively in parallel through the nodes. As predicated by Pittman [7]:

Neural networks are not programmed, they are trained on real examples.

A neural network is not a static system, but rather it is able to learn while it is active. By adjusting internal weights for operations, this learning process is achieved. Therefore, outcomes are evaluated and depending on this evaluation, the network adjusts its weights until the optimal outcome is obtained. The learning of a neural network can be achieved by several strategies, described by [26]:

Supervised Learning involves another party, which can evaluate the outcome of the system and provide feedback to the network whether this outcome is correct or incorrect, as well as the proper solution to initiate the learning process.

Unsupervised Learning is used if the correct answers are not known. Therefore, the network needs to identify the relationships between multiple inputs, rather than obtaining a response concerning the correctness by another party.

Reinforcement Learning uses observations to learn. The network makes a decision, obtains a specific outcome and observes what happens next. If this observation yields an unfavourable outcome, the network will adjust the weights of the decision it made at the beginning. This adjustment of weights can finally result in a different decision.

Neural networks need a large set of data and multiple training iterations to obtain proper weights to make decisions. This circumstance may be a disadvantage due to the effort necessary to construct such a network, however their results are commonly very accurate, correlating with the amount of training data.

In the domain of recognition and more specific recognizing handwritten text, Pittman [7] uses the neural network approach. Streams of coordinates which may contain interesting areas are fed to the network, which uses these data to identify the input gesture.

3.2.4 Dynamic Programming

Dynamic programming is accomplished by dividing the larger problem statement into smaller portions, to start solving the small parts and work up until the whole problem is solved. According to Borgohain [14] it is divided into four steps:

1. Split the problem into overlapping sub-problems.
2. Solve each sub-problem recursively.
3. Combine the solutions to sub-problems into a solution for the given problem.
4. Do not compute the answer to the same problem more than once, rather store it in a memory, to recall it if required.

Dynamic programming is used by Myers and Rabiner [6] to:

> *. . . find the best concatenation of reference patterns to match a given test pattern by first determining the optimal reference pat-*

> *tern to match any portion of the test pattern and then attempting to find the optimal way in which to concatenate these pieces.*

3.2.5 Ad-hoc Heuristic Recognizers

Ad-hoc heuristics are explained by Wobbrock, Wilson and Li as follows [11]:

> *By "ad-hoc" we mean recognizers that use heuristics specifically tuned to a predefined set of gestures.*

A possible implementation is the recognizer for handwritten Korean scripts [3]. This recognizer requires specific heuristic methods to cover all peculiarities. These heuristics are specifically designed for this purpose and are likely not usable for other tasks without modifications.

Chapter 4

State of the Art

To fulfill the goal of this thesis and the corresponding project, various different technologies needed to be considered, to select the most appropriate. Various prototyping, mockup and wireframing tools are currently available, therefore it was essential to gain an overview over already existing tools, to identify their approaches and features.

The Web Components technology was considered for the implementation, to ensure that the tool remains expandable. Utilizing it for this purpose is possible via its native implementation or current frameworks and libraries. To identify the most appropriate solution, multiple frameworks and libraries are compared to the native implementation, to identify advantages and disadvantages.

Additionally, the recognition of drawn-input is a core element of the thesis. Therefore it is essential to identify a suitable technology to fulfill this purpose. Different recognition approaches and algorithms are compared to identify the most adequate.

4.1 Tools

Many different prototyping, mockup and wireframing tools are available on the Web. These three terms are often used synonymical and therefore also the non functional types can contain multiple capabilities to include functionality. Although the main focus of this thesis is on Web Prototyping, also mockup an wireframing tools are considered, because they frequently focus not only on these purposes, but provide additional prototyping services.

They currently available tools use various approaches concerning their functionality. Several tools focus either on the design or the functional aspect differentiated by their purpose of wireframing, mockup or prototyping. The pure design tools contain mostly no functional prototyping aspects, while the most functional tools have no or only limited possibilities of altering or creating a design. Four currently available tools which combine both aspects at least partially are studied in more detail.

Balsamiq

Balsamiq[1] is a wireframing tool available since 2008, focusing on usability. Some features are drag and drop creation of mockups, click-through prototypes, exporting to png or pdf and the usage of keyboard shortcuts. It is based on *Adobe Flash Player*[2]. Exporting to HTML code is possible using third party plugins.

InVision

InVision[3] is a web-based system focusing on design and includes basic prototyping options. First drafts have to be created in another program and uploaded to the website. Afterwards, some fundamental functions can be added, for example linking to other pages. In this scenario, the user has to create a draft for every link and target page, indicating every possibility.

JUSTINMIND

With this tool, creating and adding functionality is combined. It uses a drag and drop approach inside a graphical editor. It is necessary to install a program locally to work with this tool[4].

Marvel

Marvel[5] provides the functionality to upload drafts or to create them directly. Afterwards, it is possible to link between different designs, specify gestures to invoke these linking and define transitions.

4.2 Web Components

Web Components are a combination of various technologies. They are a part of the browser, therefore no external library or framework is needed. They can achieve everything that is possible with HTML, CSS and JavaScript. Existing Web Components can be used without writing their code again, simply by adding an import statement to an HTML page [24]. However, the implementation status of the various technologies is different for each browsers.

[1]https://balsamiq.com/
[2]http://get.adobe.com/de/flashplayer/about/
[3]https://www.invisionapp.com/
[4]https://www.justinmind.com/
[5]https://marvelapp.com/

Natively, Web Components consist of four technologies, as described by the W3C and the Mozilla Developers Network:

HTML Templates: A mechanism for holding client-side content that is not to be rendered when a page is loaded but may be subsequently be instantiated during runtime using JavaScript [23]. It is utilized to enhance the usage of client-side templates, a functionality commonly achieved by using CSS or JavaScript workarounds. These workarounds have some disadvantages, such as immediate fetching of resources or the availability of exploitable vulnerabilities.

Shadow DOM: Describes a method of establishing and maintaining functional boundaries between DOM subtrees and how these subtrees interact with each other within a document tree [18]. All markup and CSS inside a shadow DOM is scoped to its host element, therefore all styles applied to elements inside cannot affect elements on the outside.

Custom Elements: Define and implement new types of DOM elements in a document [16]. The custom elements technology allows the user to define completely new HTML elements or to extend basic elements. Additionally, a custom element is frequently used to combine all functionality of a new element into one tag.

HTML Imports: A way to include and reuse HTML documents in other HTML documents [17]. An imported document can itself contain numerous import statements. To avoid multiple imports of the same file, resources already fetched are not loaded again. Imported files are loaded automatically. However the HTML markup of an imported file is not rendered automatically. JavaScript is necessary to start the rendering process of the imported code.

Web Components can be created using plain HTML, CSS and JavaScript or with different already existing technologies, such as two of the most popular and largest systems *Polymer* and *X-Tag* or some other libraries. Additionally to the main functionality, they offer a set of polyfills to work in browsers, not supporting Web Components natively.

Polymer

Polymer[6] is a library for creating and using Web Components developed by Google. It also contains a large set of predefined components which can be used by simply importing them. Additionally, it has polyfills for browsers which do not natively support any Web Component technology yet. Each *Polymer* element contains its own DOM, called the local DOM. Within this area, all components, styling or functionality of a component can be encapsulated. The implementation of the local DOM in different browsers

[6]https://www.polymer-project.org/

is described by the Polymer Team as follows [12]:

> *Polymer supports multiple local DOM implementations. On browsers that support shadow DOM, shadow DOM may be used to create local DOM. On other browsers, Polymer provides local DOM via a custom implementation called shady DOM which is inspired by shadow DOM.*

For that reason, *Polymer* can be used over multiple browsers, even if they are not supporting the shadow DOM feature of the Web Components technology. Additionally, other polyfills from webcomponents.org[7] are used to fulfill the tasks of the not natively supported technologies.

X-Tag

X-Tag[8] is an open source library for component development. To operate, it needs at least the custom component technology. For browsers not supporting this technology, it uses the same polyfills as *Polymer* to work.

In contrast to *Polymer*, *X-Tag* only depends on the Custom Elements technology and gives the developer the right to opt in for Shadow DOM [25]. The two remaining technologies are not covered by *X-Tag*.

Bosonic

Bosonic[9] is a set of tools to facilitate the development of Web Components. As described by the *Bosonic* developers [15]:

> *Built on top of the web components polyfill library*[10]*, Bosonic provides a very thin layer of syntactic sugar that eases the burden of working with Web Components primitives.*

It includes a set of predefined Web Components, to ensure an easy starting point for developing own custom elements. Furthermore, it offers support for older browser version, which do not natively support the Web Components technologies as consequence of the polyfill library which it is build upon.

SkateJS

SkateJS[11] is a library built on top of the *W3C web component specs*[12] that enables you to write functional and performant web components with a very

[7]http://webcomponents.org
[8]https://x-tag.github.io/
[9]https://bosonic.github.io/
[10]https://www.webcomponents.org/polyfills/
[11]https://github.com/skatejs/skatejs
[12]https://github.com/w3c/webcomponents

small footprint. To operate correctly, it requires the functionality of Custom Elements and Shadow DOM v1. It does not need any other libraries to be used However, if the before mentioned technologies are not supported, a library which provides polyfills is recommended.

Slim.js

Slim.js[13] is a lightweight, opensource Web Component library. It is build upon ES2015 classes, which enables providing new capabilities such as data binding. It uses all Web Components technologies and needs an inclusion of a polyfill library to operate in browsers which do not support Web Components natively.

4.3 Recognition

Utilizing drawn input for web prototyping is a field of ongoing research. The various concepts range from recognizing only basic geometrical symbols up to complete sketches. Most of them are suitable for several cases, while some are exceedingly specialized for one task.

To retain the recognition process as fast and easy as possible, the algorithm needs to fulfill some preconditions:

- usable without long training period,
- recognition without huge delay between drawing and detection start,
- fast recognition of drawn gesture,
- easy expandable to add new gestures,
- few restraints regarding possible gestures,
- usable with mouse, pen and finger.

Among all available algorithms, not every type is usable in the web prototyping context. Concerning the circumstance that most HTML elements are represented by rectangles, a complete sketch would mostly consist of rectangles. Algorithms frequently depend on rather different shapes to distinguish between them, therefore it would be intricate if all shapes are rectangles. For that reason, an approach where only single gestures are recognized is chosen.

4.3.1 The Dollar Family

The Dollar Family algorithms focus on interface prototyping. However, their approaches are usable in other areas as well, because their underlying idea is just to recognize one or multiple strokes. All of them follow a template

[13]http://slimjs.com/

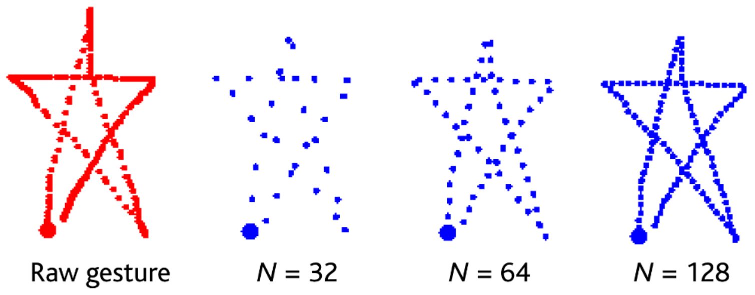


Figure 4.1: A drawn gesture resampled to 32, 64 and 128 points [11].

matching approach, whereby the extension of gestures can be realized. Additionally, they do not need a large set of training samples or a long time for familiarization. The family consists of different algorithms, which would all be suitable for web prototyping, however, they differ in their possible gestures and recognition speed. Additionally, they support any possible input method, as far as the whole drawn path can be obtained through this method. This is an advantage, because as stated by Huawei, Xiangshi and Shumin [5]:

> *The complexity of a stroke gesture may have an impact on the difference between finger and pen gestures.*

For that reasons, the members of the Dollar Family are considered in detail, to select the most suitable for recognizing drawn HTML elements.

$1 Recognizer

The $1 Recognizer is the first of the family, developed for recognizing unistroke gestures. To identify the drawn gesture, a drawn candidate gesture is compared to all available template gestures to find the best match. The basic algorithm contains as described in [11] four steps:

1. *Resample the Point Path* The points received through any input method are resampled to 64 points per path, as visualized in Figure 4.1.
2. *Rotate* The resampled gesture is rotated once, based on the Indicative Angle. This angle is formed between the centroid and the first point of the gesture. Thereby, the gesture is rotated until this angle is 0° (see Figure 4.2).
3. *Scale and Translate* The gesture is scaled non-uniformly to a reference square.
4. *Find the Optimal Angle for the Best Score* The gesture is again rotated and compared to the template gestures to find the best score.

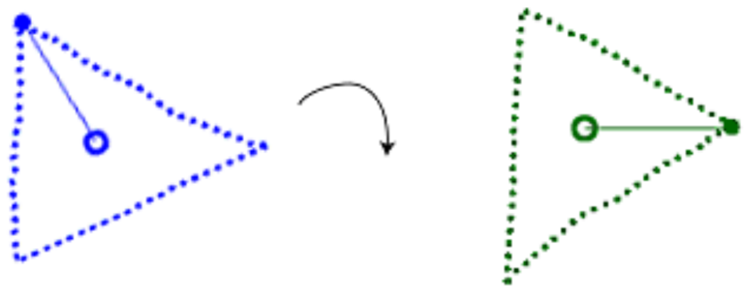


Figure 4.2: A gesture rotate until its Indicative Angle is 0° [11].

The limitations of the $1 Recognizer are that it cannot distinguish between gestures which are only different in their orientation, location or size.

$N Recognizer

The $N Multistroke Recognizer is another member of the Dollar Family and an extension to $1. In [2] it is described as:

> *...a lightweight, concise multistroke recognizer that uses only simple geometry and trigonometry.*

The algorithm extends on the $1 Recognizer by enabling the usage of multistroke templates. These patterns can be drawn by the user either by any number on strokes or by their unistroke representation. Nevertheless, internally, all multistroke templates are considered as unistroke gestures by connecting the single lines. One main difference of $N is that it generates all possible stroke orders and directions from one multistroke gesture. Therefore, the computing effort for one gesture rises with the number of strokes. As can be seen in Figure 4.3 a `X` already has eight different permutations. Regarding these generation of permutations, the main limitation of the recognizer is space and time. More gestures increase the processing load, as well as do the number of strokes. The algorithm reaches its computational limit at gestures with more than six possible strokes.

$P Recognizer

$P is the latest member of the Dollar Family. It is again a multistroke recognizer, but enhances the $N Recognizer concerning recognition speed and required memory. As presented by [10]:

> *Gestures are seen as unordered sets, or what we call clouds, grouping points together.*

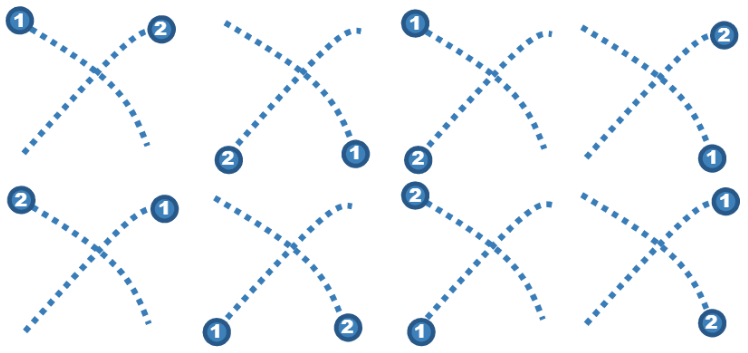


Figure 4.3: The eight different possibilities how the gesture can be drawn [2].

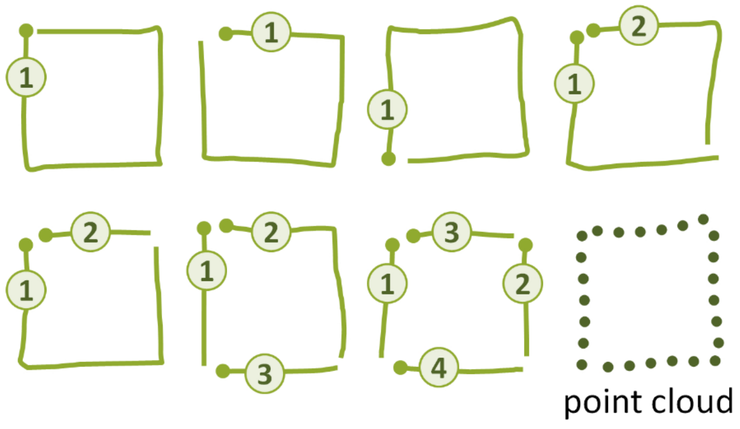


Figure 4.4: The eight different possibilities how the gesture can be drawn [10].

Therefore, the stroke order, direction or number of strokes become irrelevant, because all gestures are defined by multiple points which do not need to be in any order (see Figure 4.4). For that reason, the $P Recognizer outperforms the $N Recognizer concerning its execution and memory costs, because it does not need to compute every possible stroke order and direction for one gesture.

The recognition method of one gesture operates similar to the other two algorithms, by comparing one candidate gesture to all stored template gestures. The result is calculated using the Nearest-Neighbor approach. This technology compares all points of the resampled and processed candidate point cloud to all points of a template gesture. The template with the smallest distance of the points is delivered as the result [2].

4.3.2 Penny Pincher

Penny Pincher is built upon the Dollar Family algorithms. The algorithm is described as follows [9]:

> *Designed to do the absolute minimum amount of work possible to match candidate gestures with templates so that more templates can be evaluated within the same amount of time as other recognizers.*

Penny Pincher is mostly used in games, where gestures are significantly inconsistent and inaccurate. Additionally, in games, only limited computational resources are available to the recognizer. As $N, the algorithm concatenate gesture strokes together to one unistroke gesture.

The algorithm uses a different approach for detecting gestures. In contrast to the Dollar algorithms, Penny Pincher avoids rotating, scaling and translating the gesture [9]. To compensate this, more templates for one gesture need to be defined beforehand and loaded while the recognition algorithm is active.

Penny Pincher performs great if gestures are direction sensitive. However, the main limitation is that it needs multiple gestures for one template to match different possible rotations and stroke orders of a gesture.

Chapter 5

Implementation

This thesis is about the possibility of utilizing user-drawn input for web prototyping. To emphasize the possibility and to demonstrate one opportunity of how this could be achieved, a web prototyping tool was developed (Figure 5.1), which uses drawn input for creating HTML code. Some developmental decision needed to be made beforehand, such as choosing an appropriate recognition algorithm and a suitable web technology for this purpose. Additionally, the given set of HTML elements needed to be modified to perform proper as a set of prototyping elements. To develop this tool, four main steps needed to be accomplished:

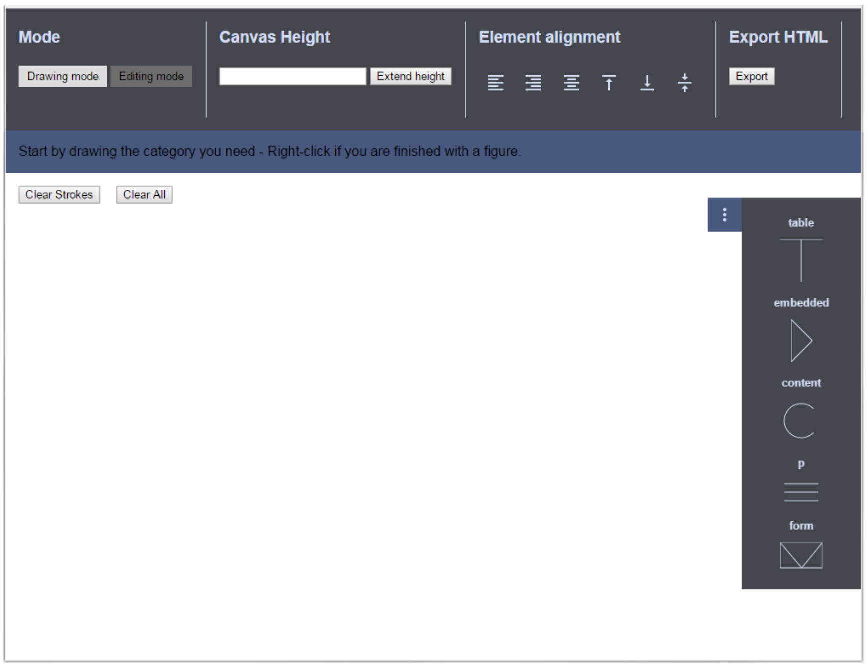
1. categorization and selection of elements,
2. combining chosen elements with fitting gestures,
3. implementation of recognition technology and components,
4. implementation of user-friendly interaction methods.

5.1 Preconditions

The prototyping tool requires a few preconditions to operate on a system.

Node.js[1] needs to be installed on the system first, including the package manager *npm*[2]. *Node.js* and *npm* are responsible for downloading and managing the additionally needed packages, frameworks and libraries. Furthermore, a web server implementation is needed for exporting HTML files. This server is build with *Node.js*, which utilizes the modules *express*[3], *body-parser*[4] and *fs*[5] for realizing the server and exporting purposes.

[2]https://www.npmjs.com/
[3]http://expressjs.com/
[4]https://github.com/expressjs/body-parser
[5]https://nodejs.org/api/fs.html



Start by drawing the category you need - Right-click if you are finished with a figure.

Figure 5.1: The interface of the prototyping tool as developed during the thesis project.

Polymer[6] is utilized as Web Component framework. To install *Polymer* itself and all its component dependencies, *npm* and *bower*[7] are used.

5.2 Selection of Elements

The early phases of creating a new web project are commonly characterized by obtaining first impressions regarding the basic look and functionality of single parts of the future project. To outline these aspects, prototypes are frequently used as soon as possible. These prototypes are often rough, to emphasize only the first important characteristics concerning the look and functionality. Concerning this, not every HTML element is needed to create this kind of prototypes.

Hence, only 23 HTML elements were used for recognition and as options the user can choose from. These elements were selected due to their effect in a web project. These effects can be semantic or non-semantic, which means that an element can either tell the browser and developer about its content or not. It is not possible to distinguish between semantic and non-semantic elements only by their output in the browser, because some appearances can be achieved by using either a semantic or a non-semantic element. For

[7]https://bower.io/

example, the element `<strong>` which renders the text in bold and also indicates that the text is important, compared to the `<b>` element, which again renders the text in bold but has no special semantic meaning. This classification in semantic and non-semantic elements is a general declaration which is used in this thesis for comprehension, while other, more specific subdivisions may exist.

semantic elements

Semantic elements are used to add meaning to HTML code, which can be recognized by the developer, the browser or some other tools, such as screen readers. These elements can have a visual representation to emphasize their special meaning or not. An example for a semantic element without a visual representation would be the `<header>` element. It defines a head-area for the document or a section, to structure the parts of a website.

non-semantic elements

Non-semantic elements add no special meaning to their content. They are frequently used for developing purposes such as grouping and styling of elements. Non-semantic elements can also have a visual representation. One example of a non-semantic element would be the `<b>` which renders the contained text in bold but adds, unlike `<strong>`, no special meaning. Another example is the `<div>` element, which is commonly used for structuring but has no visual representation or any meaning.

To use HTML elements for web prototyping, not all elements were used as explained before. Visual representations which can be achieved using either semantic or non-semantic elements were implemented only once for recognition. Mostly, their non-semantic element is used, because in the first design and prototyping phases the semantic aspect is often not the main concern.

5.3 Categorization

To receive a relevant set of HTML elements the lists on w3schools [28], Mozilla Developer Network (MDN) [22] and W3C Web Education Community Group [20] were compared and combined. Mostly, these three sources contain the same elements, but they differ in their categorization, concerning their names and divisions. The categories used for recognition are primarily taken from the W3C division of elements while some elements are sorted in a different way. The resulting categories and their elements are:

table: The first category is similar to the W3C, MDN and w3schools category but contains only the `<td>` element which can be drawn to add new columns to a table. Some other elements of a table are added automatically, such as `<tbody>` or `<th>`.

embedded: This name is taken from the W3C categorization. It contains for example `<img>`, `<iframe>` and `<video>` which are mostly the same elements as in the original W3C category.

content: This category is again taken from W3C. Four of the contained elements are the same as in the original category, two are added to it for simplification.

p: This element is the only one with its own category. Inside, no other recognition of elements is possible, however text can be formatted using the context menu. Four options are available: `<a>`, `<i>`, `<u>` and `<b>`. These elements have no gesture representation because they refer to a marked text and can not drawn before any text is entered.

form: The form is used by all three categorizations. Six HTML elements such as `<input>` and `<textarea>` are available for the gesture recognition and some options for these elements via the context menu.

This categorization of HTML elements was used to keep the recognition process as fast as possible and mainly to achieve a small set of possible gestures for the users. Due to this, at most six different possible gestures for one category are possible, which can be easily recognized and drawn by the user.

5.4 Chosen Gestures

Each HTML element used is represented by one gesture. These gestures are drawn by the user to add new elements. Therefore it is important that the user can draw them easily and fast. Gestures with multiple strokes are more time consuming than gestures which can be drawn with one stroke, so one-stroke-gestures are preferred. Furthermore, it is necessary that the recognizer can also identify the right element if it was drawn fast and maybe quite inaccurate. To achieve this, the gestures need to differ from each other as much as possible, without adding too many complex details. Additionally, each gesture needs to be meaningful to the user, to ensure that they are recognizable and usable without long training times.

Considering all these needs, the gestures were defined to either simulate the resulting element, be a letter of the elements name or consist of already known shapes or symbols which can be connected to the element via an intuitive relation. Through the categorization of elements some gestures can be used multiple times. Especially for gestures which look similar to their corresponding element this fact is an advantage. Without this categorization each element would need a completely different gesture, which would impair their simplicity. Many elements need to define some specific values, such as

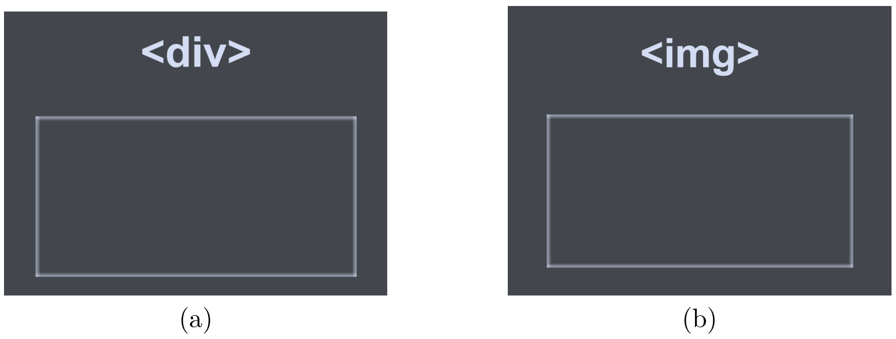


Figure 5.2: Two HTML elements represented by drawing a rectangle. `<div>` (a) and `<img>` (b).

their size, through the drawn shape, which are saved during the recognition process and used to create the element. Without utilizing the gesture to define the size already while drawing, some additional complexity would be necessary to obtain this information. For example the `<div>` and the `<img>` element are both represented by drawing a rectangle (see Figure 5.2).

The size of the drawn shape is used for determining the size of the HTML element. If the categorization is absent, at least one of these elements would need another gesture. For instance the `<img>` could be a rectangle containing another shape. This would also be understandable for many developers but it would add complexity to the simple, one-stroke drawable, shape which should be avoided as much as possible.

For some gestures, such additional complexity cannot be circumvented, because categories may contain similar looking elements. Therefore, it was necessary to take shapes or symbols for extending the basic shape which are already familiar to many developers, to facilitate the usage. One of these elements is the HTML5 `<video>`, whose rendered shape is a rectangle too, so the preferred gesture would be a rectangle, such as for the `<img>` element in the same category. This element needs again a defined size, so a completely different gesture would be not ideal. Hence the basic-shape of this gesture is a rectangle, to obtain the size, containing another shape to distinguish between `<video>` and `<img>`. The play-button is chosen as contained shape, because it is already well-known by developers through its prevalent usage.

Many elements in web development are represented by rectangles. However, not all of them need to be actually drawn as rectangle, because regularly, their content determine their final appearance, for example by defining the size through the amount of contained text. For these elements it is usually not necessary to define a exact size beforehand, only their position is vital. They can easily be represented by more abstract gestures for example the `<p>` element whose gesture is a shape with three lines among themselves to constitute written text, comparable to the text box symbol of text editing programs.

5.5 Recognition Algorithm

The possibility of utilizing user-drawn input for web prototyping depends reasonably on the chosen recognition algorithm. Two main techniques are common for this purpose: recognizing complete sketches at once and recognizing single gestures. The recognition of complete sketches is able to compute larger drawings with multiple elements while single gesture recognition focuses on direct identification of one specific gesture. The complete sketch recognition approach would feel more like creating a design prototype with current design tools. However it can hardly be used for web prototyping because most HTML elements are represented by the same geometrical shape, the rectangle. Additionally the component based characteristic of web development would be neglected as everything is interpreted at the same time. On the other hand the single gesture recognition has the drawback that the recognition needs to be triggered after every component. This preserves the HTML component character and ensures that it can be distinguished between different similar-looking elements, but it requires a short training period at the beginning. Nevertheless, single gesture recognition is a possible technique for web prototyping. Thereby elements can be represented by different, well-known symbols or shapes to simplify the usage without conflicts for similar elements.

Deeming these considerations, the $P Recognizer was implemented for utilizing drawn-input for web prototyping. The $P Recognizer avoids storage complexity of $N by representing gestures as *clouds of Points* [10] which are independent in stroke order and direction. Therefore the storage of large gesture-sets is managed superiorly. The $P Recognizer was chosen because it performs similar to the $1 Recognizer on unistrokes and is superior to the $N Recognizer on multistrokes [10].

5.5.1 Recognition Categories

The recognition algorithm can be used without modifications for recognizing drawn HTML elements. However, since the algorithm matches the candidate point cloud to the point cloud of each template [10] to find the best fitting result, the original algorithm would be slowed down by the amount of different templates. To ensure that the recognition process is not decelerated by the number of different templates, the category system is applied. By dividing all available gestures in different categories, the algorithm has to check at most eight templates for one category.

Furthermore without any categorization, all templates would need to be unique, which would add more complexity to the gestures as intended. As described in detail later, the elements should be as easy to remember and draw as possible and should fulfill some requirements due to familiarity for the user. Therefore it would be more appropriate if different HTML elements could be represented by the same gesture. With the original, unmodified recognizer this is not possible, because one gesture can only be linked with one resulting element. To achieve the connection of one gesture with multiple elements, the category system of HTML elements was utilized once more.

The first implementation step was to divide all templates into different category point clouds, by creating one point cloud for every category, similar to the three categories shown here:

```
var NumPointClouds = 6;
var NumPointCloudsEmbedded = 8;
var NumPointCloudsForm = 7;

this.PointClouds = new Array(NumPointClouds);
this.PointClouds = new Array(NumPointCloudsEmbedded);
this.PointClouds = new Array(NumPointCloudsForm);
```

Due to this, it was possible that the same pattern is linked to multiple elements in different categories. For example, the `<img>` and `<div>` tag use the same template:

```
this.PointCloudsEmbedded[0] = new PointCloud("img", new Array(
  new Point(50,50,1), new Point(150,50,1),
  new Point(150,51,2), new Point(150,100,2),
  new Point(149,100,3), new Point(50,100,3),
  new Point(50,99,4), new Point(50,51,4)
));
this.PointCloudsContent[0] = new PointCloud("div", new Array(
  new Point(50,50,1), new Point(150,50,1),
  new Point(150,51,2), new Point(150,100,2),
  new Point(149,100,3), new Point(50,100,3),
  new Point(50,99,4), new Point(50,51,4)
));
```

Each point of the cloud is created by the already implemented constructor:

```
function Point(x, y, id){
  this.X = x;
  this.Y = y;
  this.ID = id;
}
```

The `id` of every point references the stroke to whom the point belongs to. This `id` can be the same for each point, which would ensure that the gesture is explicitly defined by a unistroke template. This could accelerate the recognition process if it is the only unistroke template in one category and if all other templates are considerably different. However, if this conditions are not met, using multistroke templates results in better recognition success.

To work correctly with multiple category point clouds, the main recognition function needed to be altered as well. At first, the function needs to distinguish between recognizing a category or an element in a specific category. If the current function call is evoked to identify the category, the recognition can immediately be started by iterating over all possibilities in the category point cloud. If the category is already set, the new function to recognize the element in a category is accessed. This function chooses the correct point cloud depending on the current category and starts again the iteration through the templates. This modified function is, despite its main purpose to recognize the element, also responsible for acquiring additional gesture parameters.

5.5.2 Recognizing Element Parameters

As suggested by Vatavu, Anthony and Woobrock, due to the representation of gestures as unordered set of points, the $P Recognizer cannot tell the direction, starting and endpoint of a drawn gesture [10]. Therefore it is not possible to obtain the drawn position or size with the initial recognizer.

The result found through the recognition should also contain the size and position of the drawn element, represented through their coordinates: `minX`, `maxX`, `minY`, `maxY`. For this purpose the size is complemented to the original constructor:

```
function Result(name, score, minX, maxX, minY, maxY){
  this.Name = name;
  this.Score = score;
  this.Size = {
    "minX":minX,
    "maxX":maxX,
    "minY":minY,
    "maxY":maxY
  };
}
```

To obtain these coordinates, all points of a drawn shape are saved and afterwards used to find the minimum and maximum values. Therefore the current ECMAScript 2015 `...` Spread-Operator [21] is used to keep the code as short and clean as possible.

```
Math.min(...xCoords);
Math.max(...yCoords);
```

The coordinates are tracked during the drawing process, using the current mouse, pen, or finger position of the user. The coordinates received during the drawing process need to be transformed for displaying on the canvas because of their different positions to the viewport. Mouse, pen or touch coordinates are perceived as an absolute value in the viewport, originating at the top left corner of the browser window. The values which are displayed on the canvas are relative to the canvas position, which means the origin is at the top left corner of the canvas. Due to these different origins, the drawn points need to be transformed for displaying on the canvas.

Through subtracting the absolute canvas position from the input coordinates, the position on the canvas is received. These resulting coordinates are afterwards used for displaying the recently drawn point.

```
function getMousePos(canvas, evt) {
    var rect = canvas.getBoundingClientRect();
    return {
        x: evt.x - rect.left,
        y: evt.y - rect.top
    };
}
```

5.6 Component Development

Every used HTML element is represented by one specific Web Component. These components have some basic functionality in common, however many components needed to be designed differently in order to fit the needs of the corresponding HTML element.

For developing Web Components, the *Polymer Framework* is used. Its basic structure is the same for every component created, including:

- `<dom-module>`,
- `<template>`,
- `<script>`.

5.6.1 The Dom-Module

To specify the DOM for a component, the `<dom-module>` is used. The `id` attribute must match the `is` property of the element. The name of this attributes is the filename too, to identify the component.

```
<dom-module id="my-div">

  Polymer({
    is: 'my-div',

  });

</dom-module>
```

The components obtained with the basic *Polymer* installation, are already following the convention of the W3C Working Draft for custom elements, which require the following for the validity of a name [19]:

- *They start with a lowercase ASCII letter, ensuring that the HTML parser will treat them as tags instead of as text.*
- *They do not contain any uppercase ASCII letters, ensuring that the user agent can always treat HTML elements ASCII-case-insensitively.*

- *They contain a hyphen, used for namespacing and to ensure forward compatibility (since no elements will be added to HTML, SVG, or MathML with hyphen-containing local names in the future).*

Polymer suggests to prefix custom components with "`my-...`" to clearly differentiate between self-implemented and already available elements. This standard name is maintained for the custom components, combined with the name of the native HTML element. Through this denotation, the component is accessible in the same way as all already existing elements.

```
<my-div></my-div>
```

5.6.2 The Template

The `<template>` block contains basic HTML elements, which are present in the local DOM of the component.

Every functional item in the `body` of a HTML page can also be integrated in a *Polymer* Web Component, such as HTML, CSS and JavaScript. Additionally, it is possible to import other files to use, for example JavaScript libraries, specific CSS files or even other Web Components in this template. A component can be build as detailed as necessary. The main advantage of Web Components is that the component can be used multiple times, without rewriting the same code fragment repeatedly. Therefore, the DOM of a website stays clearer and simpler, because the main part of each component is hidden in the Shady DOM, and needs to be written only once. Furthermore, one component is usually usable in multiple projects without extensive adjustments since elements, styling and behavior can be combined in a single file. For that reasons it is for example possible to write all code for an image-gallery inside one component and include it on different positions on a website. The following code presents a template for a `<div>` element:

```
<template>
  <style include="shared-styles"> ... </style>
    <div class="container" on-click="select" on-contextmenu="showCtx"></
    div>
    <div class="editingArea"></div>
    <nav class="context-menu">
      <ul class="context-menu__items"></ul>
    </nav>
</template>
```

For extending the native HTML elements to be suitable for the prototyping purpose, the template area includes at least the basic HTML element which corresponds to the Web Component. This basic element is later used for exporting valid code which could be used for further development. The listeners for `on-click` and `on-contextmenu` are used by every component to access the basic functionality for selection and the custom context menu. Additionally, for many components the two-way databinding of *Polymer* is utilized, especially for elements where the value can directly be edited via the `contenteditable` attribute. The appendix `::input` is needed to declare the element as input accessor for the referenced property.

```
<a text-content="{{link::input}}" contenteditable></a>
```

Specific styles which are only used in one type of Web Components can be defined directly in the template, to keep all parts of a single component separated from each other.

```
<style include="shared-styles">
  :host {
      display: block;
      border: 1px solid #bfbebe;
  }
  div{
    width: 100%;
    height: 100%;
  }
</style>
```

All styles in this area are applied to every instance of this specific element. General styles of HTML elements which would need to be written multiple times in every component they are used, can be defined in general CSS files, such as `shared-styles.css`. These files can afterwards be included in each component where they are needed. The root element of a component can be accessed for styling with `:host`.

5.6.3 The Script

The script contains the specific properties and functions of each component. It is responsible for handling all events which are component specific or need to access hidden component parts, which are inaccessible from the outside. For example, every modification on the Shady DOM needs to be handled through these functions.

```
properties: {
  options: {
    type: Array,
    value: [{ oName: 'editSize', oText: 'Edit' }, { oName: 'editDiv',
    oText: 'Edit content' }]
  },
  divName: String
},
attached: function(){
  storeElement(this);
},
select: function(){}
```

The properties of a component can be defined at different times in the lifecycle. First, setting a property for one instance of a component type can be done at the creation of the element with:

```
<my-div divName="div1"></my-div>
```

Second, it is possible to set a default value for properties directly in the code. These settings are present at all instances of the element. This circumstance is used for standardizing the creation of the context menu. The property `options` contains all elements which should be rendered in the menu.

```
properties: {
  options: {
    type: Array,
    value: [{ oName: 'editSize', oText: 'Edit' }, { oName: 'editDiv',
    oText: 'Edit content' }]
  }
}
```

Each Polymer element has a fixed lifecycle defined as follows [13]:

created: Called when the element has been created, but before property values are set and local DOM is initialized.

ready: Called after property values are set and local DOM is initialized.

attached: Called after the element is attached to the document.

detached: Called after the element is detached from the document.

attributeChanged: Called when one of the element's attributes is changed.

The function attached is used to create basic options and functions after the components basic structure is created and attached to the DOM.

```
attached: function(){
  storeElement(this);
}
```

The basic functions for displaying the context menu, handling resizing, moving and deleting, as well as grouping are not implemented in the component itself, because they are used by every component and would therefore be repeated multiple times. Beside these basic functions, many components need to handle specific tasks defined by their underlying HTML element, for example an `<img>` tag which must be able to specify an image file to show. Another more advanced example is the `<p>` element, which can hold different other elements, such as italic or bold text. The corresponding `<my-p>` Web Component needs to fulfill these requirements, too.

5.7 Application Functionality

To utilize the user-drawn input, the recognition functionality combined with the developed components are wrapped in one application. This application is responsible for managing and generating the elements, handling user input and providing accessibility and helper methods.

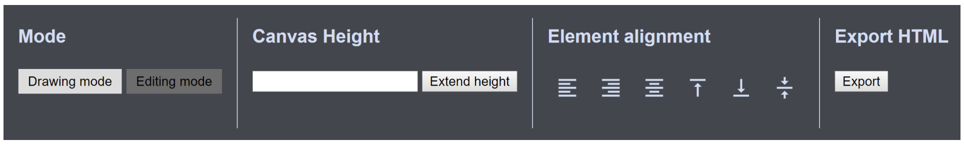


Figure 5.3: The main menu, containing functionality to enhance the usability.

5.7.1 Basic Application Structure

The main application is a Web Component in itself and consists of multiple areas, which fulfill different purposes.

Canvas: The canvas is the most important part of the application, since it is responsible for recognizing the drawn input. The Web Components generated from this input are afterwards stored for further usage.

Main menu: The main menu contains the functionality to switch between drawing and editing mode, extend the canvas height, align elements or export the code.

Component menu: Various components require the usage of a distinct component menu.

Helper menu: The helper menu indicates all possible components which can be recognized.

Drawing vs. Editing

To ensure that the system is able to distinguish between a users drawing purpose from a single click on an object, some considerations were necessary. To simplify this process, the drawing and editing modes were established, together with the functionality to switch between them. Changing between drawing and editing mode is possible via the buttons located in the main menu of the application or by pressing the keys *D* and *E* on the keyboard.

While the system is in drawing mode, each interaction with the canvas is classified as a drawing attempt, therefore the canvas will visualize them. Editing an element is not possible currently, because the canvas is located above all elements, to ensure that the drawing process is not blocked by single elements, which are already recognized and inserted. Switching to the editing mode disables all recognition possibilities. Drawing on the canvas is no longer possible, while the functionality to select elements is activated. Additionally, using the right mouse button will no longer start the recognition process, otherwise it will open the context menu. On the canvas this action will open the browser specific context menu, while right clicking on an element will open its custom context menu.

Alignment

Moving elements can be achieved in two ways, either by using the moving functionality of each component or by the alignment menu. This menu can be utilized if elements should be aligned relative to the canvas. It is a functionality to enhance the usage comfort of the application and to facilitate the alignment of single elements. By using the menu buttons, each element can be aligned in six ways: left, right, centered horizontally, top, bottom and centered vertically. All alignment options react on resizing of the canvas, therefore the alignment position will stay the same, independent of the canvas size.

Key Bindings

Key bindings are used to fulfill various purposes to raise the usability of the tool. Multiple functionalities are accessible via the applications interface and in addition via Key Bindings. The available bindings are:

- **Esc** to leave a category or an element,
- **E** to switch to editing mode,
- **D** for switching to drawing mode,
- **Ctrl+C** for copying one or multiple elements,
- **Ctrl+V** to paste the copied elements,
- **Backspace** to delete all selected elements.

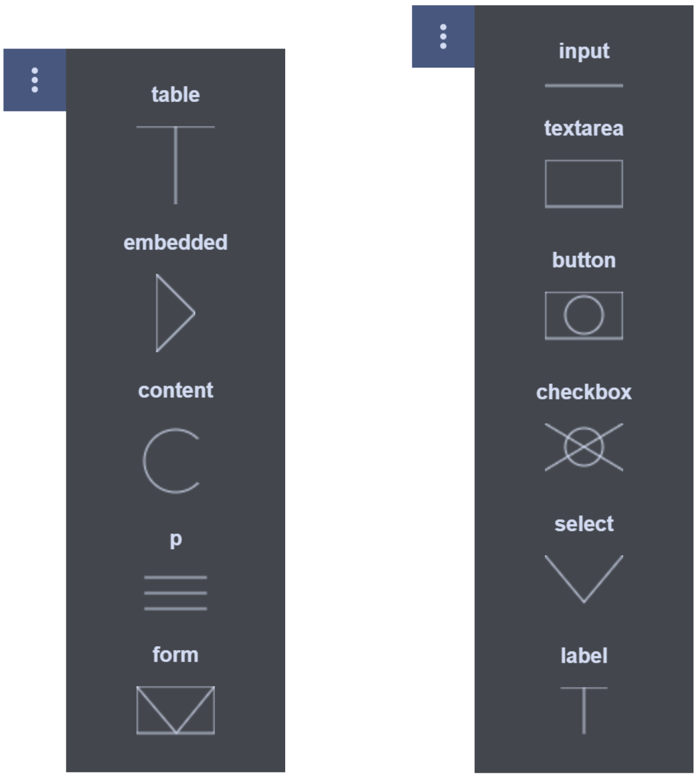
These key bindings are also available using a virtual keyboard, if no physical keyboard is available.

5.7.2 Drawing Helpers

All HTML elements are divided into five categories. These categories are not familiar to every user, therefore it is necessary to support them by providing an overview over the possible gestures in each category. This helper menu is once more a container, holding multiple Web Components, one component for each helper. When drawing an element, the user has to select the category first. Therefore, the helper contains all possible category gestures (see Figure 5.4 (a)). After drawing and recognizing one of the available opportunities, the included components in the helper menu are altered, to coincide the currently active element category, as visualized in Figure 5.4 (b).

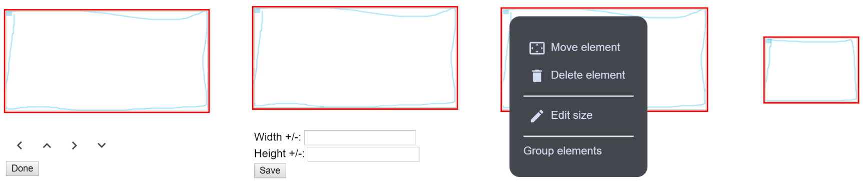
5.7.3 Basic Component Functions

Several functions are used by all, or at least by most components. To avoid repetitive code for these types of functionality, they are not embedded in each component but rather relocated in a separate file. These functions



(a) (b)

Figure 5.4: Two helper menus, containing all possible gestures for recognition. Helper with categories (a) and helper with form elements (b).



(a) (b) (c)

Figure 5.5: The functions for moving(a), scaling(b) and grouping(c) elements.

fulfill certain basic behavior, such as moving, scaling and grouping which is available for each HTML element (see Figure 5.5).

Moving

After drawing and recognizing an element, it may be displaced due to an inaccuracy occurred while drawing. To correct this, the user has the possibility to move the object. The moving function can be accessed through the context menu (see Figure 5.6).

The component can be moved either by the buttons, the arrow keys or dragging. Using the arrow keys or the buttons moves the element by one pixel. By pressing the *Ctrl key* simultaneously, the element is moved by ten pixels per step.

Scaling

The size of an element can be defined by either using a new absolute value or by altering the current one. To trigger an immediate response, the alteration of the latest size is selected for this function. Depending on the object, width, height or both can be modified by addition and subtraction of pixel values.

Deleting

Deleting an element is an essential functionality for every prototyping tool. In the prototyping context the deletion need to be treated carefully. To remove an element, it must be deleted from any possible group and additionally the parent container needs to be notified to remove the object. Afterwards the component can be destroyed.

Grouping

Grouping elements is valuable for binding elements together, in order to alter them simultaneously. Grouping elements is possible as soon as two or more components are selected. After grouping, the same methods as for single elements are available for the whole group, preserving the functionality to alter single elements in the group.

5.7.4 Context Menu

The context menu is available by right clicking a component, granting access to the basic functions such as moving and deleting, and various component specific faunctions. The basic menu visualized in Figure 5.6 is present for each component.

Furthermore, various elements need special functions to fulfill their specific requirements. The `<p>` element, for example, needs a possibility to modify the contained text, in order to format it bold, italic or underlined, which can be seen in Figure 5.7.

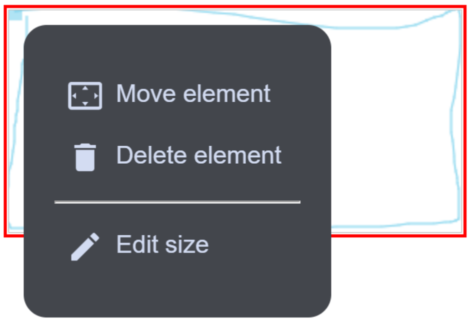


Figure 5.6: The context menu of a `<img>` element, where options for moving, scaling and deleting are present.

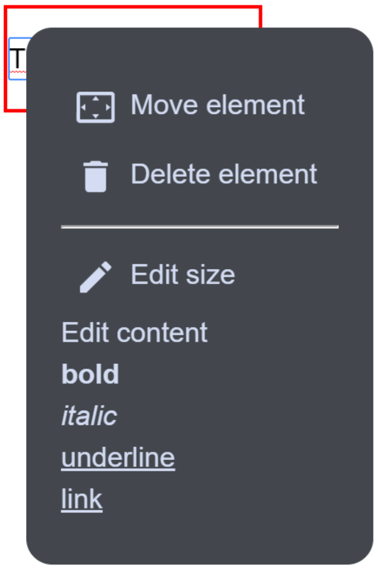


Figure 5.7: The context menu of a `<p>` element, containing the specific menu options.

5.8 Export HTML

For finalizing the prototyping workflow by saving, the tool has to meet several preconditions, including *Node.js*, *Bower* and *Polymer* in particular. However, the prototype developed with this tool is considered a starting point for further development and improvements, even if the tools requirements cannot be used or installed on every environment. To ensure that the results of the first prototyping phase can be used where they are required, an export function was mandatory.

This function allows the user to export and save the drawn prototype into plain HTML and CSS. By iterating over all existing elements, they are transformed into their corresponding native HTML element. Within this iteration, the information stored in a component needs to be extracted and rewritten to work with the native element.

```
var removeThis = [];
$.each(ht, function(i, e){
  if(e.abc){
    $.each(ht, function(j, p){
          if(e.abc == p){
              p.childElem.push(e);
              removeThis.push(e);
          }
        });
  }
});
```

The parent-child structure of all elements is rebuild within this function to simplify the following HTML generation. Inside the array, each child node is attached to its parent and afterwards removed from the array as distinct component, in order to be referenced only through the parent.

The element specific properties such as the source for the image are combined with the new code, for preserving the functionality. Furthermore, each element receives a unique class name, to ensure the conjunction of the HTML element and the corresponding CSS styles.

```
function buildImg(elem){
  var elemCode =
    "<img src='"+elem.source+"' class='img-"+bodyCount.img+"' />";
  var elemCss =
    ".img-"+bodyCount.img+"{"+cssDefault(elem)+"}";
  bodyCount.img++;
    return {html: elemCode, css: elemCss}
}
```

Additionally, the specific styling for each Web Component is saved in a separate CSS file and imported into the HTML, to retain the two code types separated.

```
function cssDefault(elem, mLeft, mTop){
  return "margin-left:"+elem.style.marginLeft+"; margin-top:"+elem.style
     .marginTop+"; position: absolute; width:"+elem.style.width+"; height
     :"+elem.style.height;
}
```

The basic CSS contains the position and size of the component. These values are set as absolute in the window because objects may overlap. Elements without a distinct width or height are treated separately to avoid redundant code.

After finalizing the HTML generation, the complete HTML code is send to a *Node.js* server for processing the received data and exporting it to valid files. Involving a server is mandatory, because file saving is not feasible using exclusively JavaScript for safety reasons.

```
$.ajax({
  type: 'POST',
  data: JSON.stringify(data),
    contentType: 'application/json',
    url: 'http://localhost:3000/endpoint',
    success: function(data) {
      window.open(data.redirect, '_blank');
    },
    async: false
});
```

Immediately after the files are completed, a new tab showing the result is opened to ensure that the generation was successful. The received files are usable on any system, capable of processing HTML5 and CSS.

Chapter 6

Evaluation

The developed tool needed to be evaluated, to verify whether the defined goal was achieved. Web developers were asked to examine the tool to obtain feedback from users with different previous knowledge and programming style. They had to replicate a given design and to validate the accuracy of each gesture. Additionally, the testers were asked to mention each difference, advantage and disadvantage of both tools. The results of both parts were later used to address the research question.

6.1 Accuracy of Gestures

To ensure that the chosen gestures can be distinguished by the recognition algorithm, the test users hat to draw each gesture 20 times to evaluate the error rate. Additionally, advantages and disadvantages of single gestures were identified by the users to improve those with less distinct results or difficulties while drawing.

Calculating the average of all recognitions during the evaluation, the average recognition rate for all elements is 89.13% while the average value for all categories is 89% and for elements located inside categories 89.17%. An category results overview is given in Table 6.1.

6.1.1 Table

The *table* category including the category gesture and all elements inside, has a recognition rate of 95.5% which is the highest success rate over all categories. However, the category gesture itself achieved only 91%, placing itself on rank three over all these gestures. Inside this category, all recognitions were successfully, regarding the circumstance that only one gesture is present. Therefore, incorrect recognitions were not possible, except if the input was insufficient.

Table 6.1: Average recognition results per category.

category	*avg. category*	*avg. elements in category*
table	91%	100%
embedded	93%	89.4%
content	84%	83%
p	85%	-
form	92%	93.3%

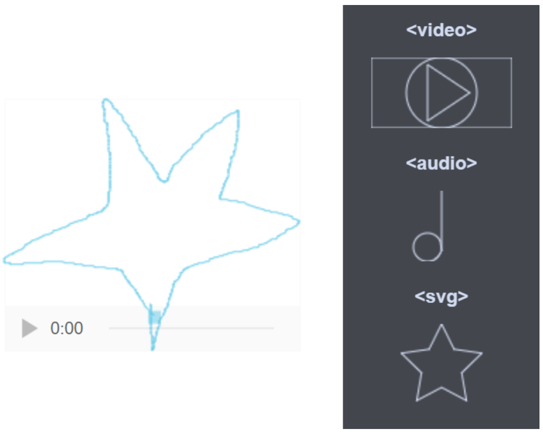


Figure 6.1: The `<svg>` gesture miscrecognized as `<video>` gesture, due to the rotation.

6.1.2 Embedded

Better results were accomplished by the *embedded* category gesture. It was successfully recognized in 93% of all test attempts. Concerning the elements included in this category, distinct results were achieved. The `<img>` had a recognition rate of 100%. However, some gestures included led to a faulty recognition. 96% of the attempts for `<audio>` were successful, which results in an error rate of 4%. The recognitions of `<video>` and `<iframe>` performed slightly worse with 95% and 89%.

The `<svg>` element achieved the least reliable results with a recognition rate of 67% and therefore an error rate of 33%. As a consequence, this gesture may need some adaptations to be easier distinguishable from other elements. This gesture was recognized successfully when it was drawn similar to the template gesture. Drawing it rotated or even upside down resulted in a failed recognition attempt (see Figure 6.1).

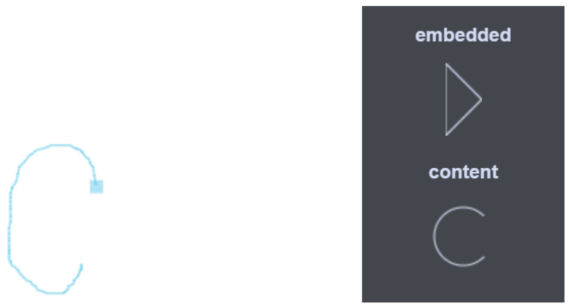


Figure 6.2: A content gesture incorrectly recognized as embedded gesture through its shape.

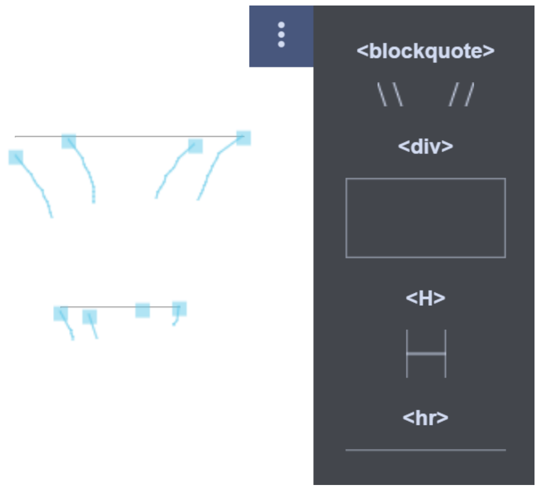


Figure 6.3: `<blockquote>` gestures, recognized as `<hr>`.

6.1.3 Content

The *content* category yielded the majority of wrong recognitions concerning the category gesture itself. Provided the drawn gesture looked similar to Figure 6.2, it was recognized as embedded, instead of content. This results from the circumstance that the start and end point of the gesture are considerable close to each other, which is interpreted by the recognition algorithm as one point respectively as tip of the embedded gesture. The elements in the content category had considerable different results, regarding their recognition rate. The first element in this category is the `<blockquote>`, obtaining insufficient results with an error rate of 43%. It was frequently recognized as `<hr>`, especially when drawn lines were short or highly curved (Figure 6.3). The `<div>` had superior results, similar to the `<img>` element in the embedded category. These two elements share the same gesture, therefore this conformable result was expected. Furthermore, a success rate of 100% was

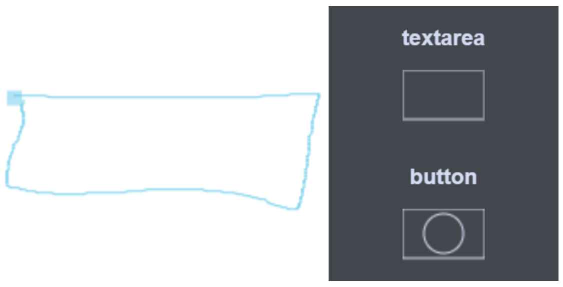


Figure 6.4: The gesture to create a `<textarea>`, incorrect recognized as `<button>`.

achieved by the `<hr>` element.

The error rate for `<H>` elements is above average with 25%. In contrast, the `<a>` has an error rate of 4% which is beneath the average in this category and the total average.

The `<list>` achieved distinct outcomes correlating to the test users. The recognition rate ranges from 60% up to 85%, which results in an average of 70%. The recognition failed frequently if both strokes of the gesture were drawn the same length.

6.1.4 P Category

The category *p* achieved a recognition rate of 85% which is slightly more successful than the content category. However the categories table, embedded and form performed superior.

6.1.5 Form

A result of 92% correct recognitions was achieved by the *form* category. Furthermore, within all categories containing more than one additional elements, this category achieved the best recognition result for its containing element, with an average success rate of 93.3%. The `<input>` element achieved a rate of successful recognitions of 100%. The gesture associated with this element is equal to the `<hr>`. Therefore similar values were expected.

The aspect ratio of a drawn rectangle led to different recognition results for the `<textarea>` element. A vertical or nearly quadratic rectangle was successfully recognized as `<textarea>` by the tool, while a horizontally drawn rectangle implies a `<button>` to the algorithm (Figure 6.4). Therefore elongated gestures were frequently not recognized as a `<textarea>`.

The `<button>` gesture is equivalent to the `<textarea>`. `<button>` elements were wrongly recognized as `<textarea>` and vice versa. Nevertheless the `<button>` had an error rate of 14% which is much higher as for the `<textarea>`.

`<checkbox>` and `<label>` achieved a success rate of 90% and 89%, which is higher than the `<button>` element. However, all remaining elements of the form category obtained a higher recognition rate. The `<select>` was the second most successful gesture in this category with 98% successful recognitions.

6.2 Usability for Prototyping

To demonstrate that the usage of user-drawn gestures is feasible for web prototyping, the time creating a prototype was compared between the developed gesture recognition tool and another well-known software, which uses a drag and drop approach. Therefore each user had to replicate a given design while the time was measured using a stopwatch. The reproduction was made using the developed tool first and afterwards with the mockup software *Balsamiq*.

One main difference between both is that the recognition tool focuses on basic HTML elements, while *Balsamiq* uses basic elements such as images combined with more complex components such as breadcrumbs. Therefore *Balsamiq* does not support every HTML element which is available in the developed tool but instead offer more advanced elements. However, the suitability for web prototyping was evaluated with basic HTML elements and components, so this difference between both tools was no significant. Additionally, using *Balsamiq* it is only possible to implement some basic functionality, such as linking between pages and implementing images. HTML functionality which is usable with the own approach, such as selecting checkboxes or displaying the dropdown function of select boxes is not possible.

6.2.1 Design Template

To compare the required time for creating a small prototype, a basic design template was essential (Figure 6.5). This design, which was later replicated by the test users, contains different HTML elements, especially those available in both editors.

6.2.2 Time and Feature Evaluation

All test users were asked to replicate the given design as fast and accurate as possible. The time of this process was measured using a stopwatch. Furthermore, each occurring difficulty and all considerable advantages and disadvantages should be mentioned.

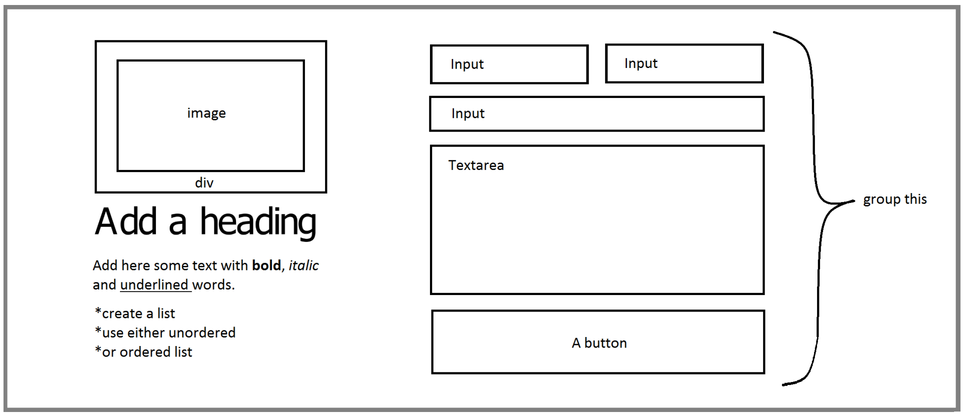


Figure 6.5: The basic design template, containing different plain HTML elements.

Drawn Input

The drawing approach of the gesture recognition tool required a trial period at the beginning for the test users. This time was not recorded to keep the resulting time comparable to the other tool, which uses a more familiar technique for creating elements.

The average time to replicate the given design was 06:50 minutes. The specific results for single users ranges between 04:27 minutes and 07:58. Therefore, some users required almost twice as much time than others to create the design using the drawing tool. One prototype resulting of this test can be seen in Figure 6.6.

To many users, the difference of how elements are represented and created to traditional editors was difficult. The tool uses one gesture for one HTML element, which results in one drawing for each desired element, with only a few exceptions. For some elements, for example the `<h1>`, this results in users trying to insert text with the `<p>` element and format this text to become a heading. In various text editors this is a valid practice, however when thinking in HTML elements, these two are different tags, which should not be included in each other. In contrast, `<i>`, `<b>` and `<u>` are usable in `<p>` elements, therefore they are not represented by a separate gesture, but rather accessible via editing the `<p>` element.

The category system chosen for the elements was assessed as unfamiliar by most users. This categorization required a few attempts until the users were able to remember the elements located in each category. The categorization itself was mentioned as useful, because only a few gestures where possible in each category and therefore mistakes while drawing were minimized.

One feature of the tool has turned out to be time-consuming for some

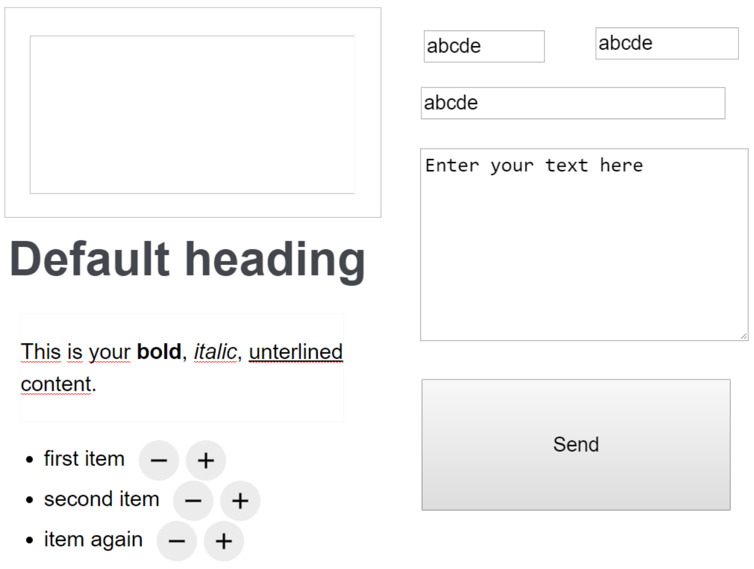


Figure 6.6: Given design replicated using the drawing input approach.

users. Switching between drawing and editing mode lead to some difficulties for several test users. For them, it was unfamiliar to use the buttons or the key bindings to switch between the two modes. Therefore, some time was lost for trying to select an element while the drawing mode was still active. However, for other users this system was comprehensible after the first few tries. For them, it was perspicuous that drawing and editing could not be achieved in one step, because already recognized element could impede the drawing of new gestures.

The main advantage for most users is, that elements appear in the same size as they were drawn before, and not in a default element size which is given by the system. Thereby, creating rough prototypes is speed up, however drawing accurate elements need some practice. Additionally they mentioned that the drawing process feels nearly as natural as creating first sketches with pen and paper. Furthermore the exporting function was appreciated by the test users, because it offers a possibility for using the prototype later, without rebuilding it completely.

Drag and Drop

For comparing the drawing to the drag and drop approach, *Balsamiq* was used, a mockup software with the main focus on usability prototyping. In contrast to gesture recognition, drag and drop for creating components is more familiar. Nevertheless all test user had no experience using *Balsamiq*, to ensure that the measured time is not biased by previously gained experience.

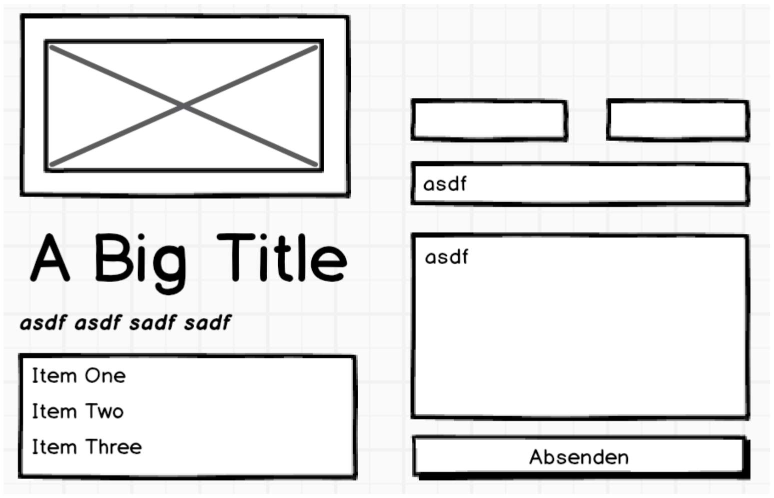


Figure 6.7: The replicated mockup, created with *Balsamiq*.

While replicating the given design, all elements could be found in the *Balsamiq* UI, except the `<div>`, regarding the circumstance that the software does not use HTML elements for developing prototypes. Therefore, the *Rectangle/Canvas/Panel* element was used as substitution. One resulting outcome can be seen in Figure 6.7. The time was measured too, starting once the blank mockup area was loaded. The average amount of time to replicate the design was 06:39 minutes.

The main advantage mentioned by the test users is that the categories of elements are easier to understand, because they divide all elements by their topic. For example, the category *Media*, includes the *Image* or the *Button* included in *Forms*. However, various elements belong to multiple categories so they are included in all these categories. This circumstance can simplify the location of elements but increase also the number of elements in one category. Additionally, the drag and drop approach was more familiar to the test users, therefore no time was needed to familiarize with the input method before starting the replication.

The most significant disadvantage users revealed is the absence of a native HTML export functionality. An additional third party plugin is needed to enable this. Furthermore, the number of available elements in each category and their order was criticized. Additionally, the elements options box was seen as too large and placed too near to the elements, which can disturb the user.

6.3 Results

After evaluating all elements, 17 of 23 elements achieved a recognition rate of more than 85%, while 3 were between 85% and 75% and only 3 gestures were below 75%. This result reveals that the majority of gestures is feasible for web prototyping. However, at least the gestures below 85% need some revision, to be easier drawable for the user and recognizable for the system. Additionally, all gestures which had difficulties regarding different aspect ratios need either an additional template, a reworked gesture or maybe an alteration of the recognition algorithm. Various gestures, such as the `<video>` element had a higher recognition rate, however they were criticized by the users due to their drawing complexity. These gestures should be simplified if possible. Simple gestures worked out really well, therefore they could be reused as often as possible, to ensure that the drawing process is as fast and easy as possible, without loosing information during the drawing process. Furthermore, the distinctions between the drawing style of multiple users lead to some false recognitions. This differences should be utilized to improve the templates stored for each gesture, to fit the needs of multiple users. The category system was unfamiliar to most test users, although these categories are even used officially (see Chapter 4). However they are rarely used deliberately in web development, therefore they are not considered by most users. Nevertheless, the system should be maintained, because these categories simplify the creation of unique gestures, speed up the recognition process and ensure that the user is not overwhelmed by too many gestures at once.

The evaluation of the time needed to replicate a given template revealed that the drag and drop approach was slightly faster. It took an average of 06:39 minutes to replicate the design using *Balsamiq* which utilizes the drag and drop technique and 06:50 minutes to replicate it using the drawing input. That means that the average time needed was 11 seconds lower using *Balsamiq*. However, the single results between various users were different for each method. For some users the drawing approach accelerated their prototyping speed, while for some user the speed was slowed down. Users which had already some experience with drawing on a device obtained better results than users which had little or no experience with this technology. Most users were more familiar with drag and drop as input method, therefore this test users achieved a valid result faster.

Comparing the results of both approaches revealed that using drawn input is feasible for web prototyping, although current, familiar technologies are marginally faster. The concept of using drawn gestures as input method may not be familiar to many users yet, but while creating elements with the tool most users got used to it and became faster. However, the drawing approach may not be appropriate for every user. It depends on personal preferences whether it can be time saving or would increase the time needed.

Chapter 7

Conclusion

While developing a website, many developers and designers need to show their ideas to other team members or a customer. Often, these first ideas should be shown graphically, to ensure that each party has the same understanding of the result. These ideas are often created during meetings with the relevant stakeholders, where sketching these ideas must be done fast. Therefore, mostly pen and paper are used to create them. Afterwards, these ideas are used to create a first prototype, where the functionality of these sketches can be demonstrated to all involved parties.

A combination of these steps would facilitate the everyday life of developers and designers. Therefore the utilization of user-drawn input for generating functional, valid code was a possible consideration. As stated in the thesis, user-drawn input is feasible for prototyping. The gesture approach which is used for the thesis project is appropriate for recognizing different HTML elements, due to their similar, rectangular design. Also, to achieve the ability to add new components which are needed in specific areas, the Web Component technology was utilized.

Drawing on a mobile device to create a prototype may be a unfamiliar technique for many people. However, after getting familiar with it, it may become a fun and fast way of developing. It can speed up the prototyping process and also lower the number of revision steps, because functionality can easily be demonstrated after the first design or composition draft. Therefore the involved people do not need to wait until this design is rewritten into code and then be seen that the functionality is not as desired and something else should be tried.

It was proven that user-drawn input is feasible for web prototyping and that it can enhance the speed of prototyping. However, at least the time enhancement can only be demonstrated for selected users, a general time saving over all users can not be stated. This could be attributed to differences in the familiarity with different input methods and the disposition to get familiar with new technologies.

Following the evaluation of the tool developed for this thesis, it was revealed that there is still some space for improvements. For example, a possibility to define custom gestures for elements should be included, to fit the drawing style of different users as much as possible. Additionally, the creation of custom categories could be a potential extension because this could speed up the prototyping process significantly.

Appendix A

Contents of the CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Thesis

Pfad: /

Groemer_Elisabeth_2017.pdf	Master thesis (entire document)

A.2 Online Sources

Pfad: /online

*.pdf	Copies of the downloaded online sources

A.3 Images

Pfad: /images

*.png	Rendered Images

A.4 Source Code

Pfad: /project

editor.zip	Source code of the developed tool

A.5 Miscellaneous

Pfad: /misc

evaluation.xlsx	Evaluation numbers and calculations

References

Literature

[1] Derek Anderson, Craig Bailey, and Marjorie Skubic. “Hidden Markov Model Symbol Recognition for Sketch-Based Interfaces”. In: *Proceedings of the AAAI 2004 Fall Symposium, Workshop on Making Pen-Based Interaction Intelligent and Natural.* (Menlo Park, CA). AAAI Press, 2004, pp. 15–21 (cit. on p. 9).

[2] Lisa Anthony and Jacob O. Wobbrock. “A lightweight multistroke recognizer for user interface prototypes”. In: *Proceedings of Graphics Interface (GI 1́0).* Toronto, Ontario: Canadian Information Processing Society, May 2010, pp. 245–252 (cit. on pp. 18, 19).

[3] Mi Gyung Cho. “A new gesture recognition algorithm and segmentation method of Korean scripts for gesture-allowed ink editor”. *Information Sciences: An International Journal Archive* 176 (9 May 2006), pp. 1290–1303 (cit. on pp. 8, 11).

[4] Tracy Hammond and Brandon Paulson. “Recognizing sketched multistroke primitives”. *ACM Transactions on Interactive Intelligent Systems* 1 (1 Oct. 2011), 4:1–4:34 (cit. on pp. 6, 7).

[5] Tu Huawei, Ren Xiangshi, and Zhai Shumin. “Differences and Similarities between Finger and Pen Stroke Gestureson Stationary and Mobile devices”. *ACM Transactions on Computer-Human Interaction* 22 (August 2015), 22:1–22:39 (cit. on p. 17).

[6] C. S. Myers and L. R. Rabiner. “A comparative study of several dynamic time-warping algorithms for connected word recognition.” In: *The Bell System Technical Journal.* Vol. 60. Alcatel-Lucent, Sept. 1981, pp. 1389–1409 (cit. on p. 10).

[7] James A. Pittman. “Recognizing handwritten text”. In: *Proceedings of the CHI ’91 SIGCHI Conference on Human Factors in Computing Systems.* (New Orleans, Louisiana, USA). 1991, pp. 271–275 (cit. on pp. 9, 10).

[8] Dean Rubine. "Specifying gestures by example" (4 July 1991), pp. 329–337 (cit. on p. 8).

[9] Eugene M. Taranta, Andrés N. Vargas, and Joseph J. LaViola. "Streamlined and accurate gesture recognition with Penny Pincher". *Computers and Graphics* 55 (2016), pp. 130–142 (cit. on p. 20).

[10] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. "Gestures as point clouds: A $P recognizer for user interface prototypes". In: *Proceedings of the ACM International Conference on Multimodal Interfaces (ICMI 1́2).* (Santa Monica, California). New York: ACM Press, Oct. 2012, pp. 273–280 (cit. on pp. 18, 19, 26, 28).

[11] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. "Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes". In: *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 0́7).* (Newport, Rhode Island). New York: ACM Press, Oct. 2007, pp. 159–168 (cit. on pp. 8, 11, 17, 18).

Online sources

[12] The Polymer Project Authors. *Local DOM Basics and API.* 2016. URL: https://www.polymer-project.org/1.0/docs/devguide/local-dom (cit. on p. 15).

[13] The Polymer Project Authors. *Polymer Element Lifecycle.* 2016. URL: https://www.polymer-project.org/1.0/docs/devguide/registering-elements (cit. on p. 32).

[14] Nilutpal Borgohain. *Dynamic programming for beginners. Part 1.* URL: https://www.hackerearth.com/practice/notes/dynamic-programming-for-beginners-part-1/ (visited on 07/05/2017) (cit. on p. 10).

[15] *Bosonic - A practical collection of everyday Web Components.* URL: https://bosonic.github.io/index.html (visited on 04/14/2017) (cit. on p. 15).

[16] W3C Editor's Draft. *Custom Elements.* Feb. 2017. URL: http://w3c.github.io/webcomponents/spec/custom/ (cit. on p. 14).

[17] W3C Editor's Draft. *HTML Imports.* Feb. 2017. URL: http://w3c.github.io/webcomponents/spec/imports/ (cit. on p. 14).

[18] W3C Editor's Draft. *Shadow DOM.* Feb. 2017. URL: http://w3c.github.io/webcomponents/spec/shadow/ (cit. on p. 14).

[19] W3C Working Draft. *Custom Elements.* Oct. 2016. URL: https://www.w3.org/TR/custom-elements/#valid-custom-element-name (cit. on p. 29).

[20] W3C Web Education Community Group. *HTML/Elements*. Nov. 2011. URL: https://www.w3.org/community/webed/wiki/HTML/Elements (cit. on p. 23).

[21] Mozilla Developer Network. *ECMAScript 2015 Spread Operator*. May 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator (cit. on p. 28).

[22] Mozilla Developer Network. *HTML element reference*. Jan. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element (cit. on p. 23).

[23] Mozilla Developer Network. *HTML Template*. May 2017. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template (cit. on p. 14).

[24] Mozilla Developer Network. *Web Components*. Jan. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components (cit. on p. 13).

[25] *Polymer vs. X-Tag*. July 2014. URL: https://pascalprecht.github.io/2014/07/21/polymer-vs-x-tag-here-is-the-difference/ (visited on 05/28/2017) (cit. on p. 15).

[26] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. URL: http://natureofcode.com/book/chapter-10-neural-networks/ (visited on 06/14/2017) (cit. on pp. 9, 10).

[27] Marcin Treder. *Wireframing, Prototyping, Mockuping – What's the Difference?* Sept. 2016. URL: https://designmodo.com/wireframing-prototyping-mockuping/ (visited on 05/15/2017) (cit. on p. 4).

[28] w3schools.com. *HTML Element Reference*. June 2017. URL: https://www.w3schools.com/TAgs/ref_byfunc.asp (cit. on p. 23).