Tampere University of Applied Sciences

Prototyping Touchless User Interface for Interacting with a Website

Jungsoo Moon

BACHELOR'S THESIS
December 2019

Degree Programme in Media and Arts

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Media and Arts

MOON, JUNGSOO:
Prototyping Touchless User Interface for Interacting with a Website

Bachelor's thesis 42 pages
December 2019

The objective of this thesis was to implement a touchless user interface that runs in the web browser for interacting with a website, using the input received through running face tracking and facial expression recognition on a video stream from a webcam.

For this, a high fidelity prototype is developed as a web application, aiming to perform the following operations on the demo website created as a part of the prototype: point and click, scrolling, and paging. For point and click operations, two different concepts are presented: the point-pointer and the area-pointer, which enables performing the operations on the point and the area, respectively.

The prototype that performs the aimed operations is successfully implemented, showing that it is feasible to create a touchless user interface that runs in the web browser. In this thesis, techniques used for implementing the prototype are documented, as well as limitations found during the development process.

Key words: touchless user interface, prototyping, face tracking, facial expression recognition, web browser, web development

CONTENTS

1 INTRODUCTION

1.1 Motivation

While touchless user interface is growing popular as a new way of interacting with a computer, its usage often entails acquiring specialized software/hardware. Motivated by this circumstance, this thesis examines how the touchless user interface that runs in the web browser can be implemented, only using a webcam, which is built-in input device on most laptops and smartphones nowadays.

1.2 Scope and goal

Touchless interaction is an emerging interaction technique in Human-Computer Interaction (HCI) field that studies the design of computer technology, focusing on the interaction between humans and computers [1]. This thesis studies the practical implementation of touchless interaction on the web. The web has a unique interface of links that enable changing displayed content and accessible links. The position of the interactive elements is not restricted to specific areas such as a menu bar but spread throughout the document [2]. In order to enable interaction in this particular environment, a touchless user interface is prototyped to perform the following operations:

- Point and click
- Scrolling up and down
- Paging back and forward

The prototype is implemented as a web application and made to perform the aimed operations on a demo website created as a part of the prototype. For this, web technologies such as HTML, CSS, and JavaScript are used along with Web API provided by the web browser. Touchless input for controlling the prototype is received by running face tracking and facial expression recognition on the video stream from the webcam, using a JavaScript library called face-api.js [3]. Face-api.js is built on top of TensorFlow.js [4], a JavaScript-based machine learning framework that helps to solve complex problems, such as computer vision and speech recognition, in the web browser.

The thesis focuses on demonstrating concepts and techniques used for prototyping a touchless user interface that can perform the pursued operations on the demo website, after a brief introduction to touchless interaction. Parts related to a framework technology used for developing the prototype (React [5]) are kept to minimal to make the thesis concise. To check the entire code of the prototype, visit a Git repository at https://github.com/dalgrang/touchless-ui-for-a-website.

2 Touchless interaction

2.1 An emerging method for achieving human-computer interaction

Touchless interaction is rising as a new interaction paradigm with the advancement in voice recognition and gesture recognition that enable us to communicate with computers using voice or gesture [6]. Interaction can be categorized as touchless if it happens without physical contact between the human and the computer. Therefore, devices like a Wii that requires using a wireless controller for interaction is not touchless [7].

Increasing demand for touchless interaction is well reflected in market research reports. A report from Fortune Business Insights anticipated that the global speech and voice recognition market to be worth USD 28.3 Billion by 2026, growing at a compound annual growth rate (CAGR) of 19.8% between 2018 and 2026 [8]. The gesture recognition market is anticipated to grow at CARG of 29.63% between 2017 and 2022, reaching USD 18.98 Billion by 2022, according to a report from MarketsandMarkets [9].

2.2 Characteristics of touchless interaction

By contrasting characteristics of touchless interaction with that of touch interaction, properties of touchless interaction can be examined clearly. Table 1 below, which is constructed by O'Hara et al., exhibits some contrast points of touch and touchless interaction [10].

Touch	Touchless
co-proximate with surface	distant from surface
transfer of matter	no transfer of matter
pressure on surface	no pressure on surface
momentum of object	no momentum
attrition and wear of surface	no attrition or wear
movement constrained by surface	freedom of movement
haptic feedback	no haptic feedback

Table 1. Contrasting characteristics of touch vs. touchless interaction

First, touchless interaction happens at a distance from a surface of a system, while touch interaction requires the system to be in reach for touching. Depends on the sensing technology used for touchless interaction, the distance from the surface can be varied. Second, there is no transfer of matter to or from the system with touchless interaction, while a transfer of matter is necessitated with touch interaction due to contact required for interaction. Third, it is improbable to cause damage to the system with touchless interaction since no pressure or momentum is applied to the surface of the system. Contrarily, damage can occur to the system with touch interaction because of its nature that requires touching the surface of the system, applying pressure and momentum. Fourth, with touchless interaction, movement is not constrained by the surface of the system, unlike movement with touch interaction. Finally, touchless interaction doesn’t give haptic feedback, while touch interaction provides one when the contact is made with the surface of the system. Consequently, a lack of haptic feedback in touchless interaction reduces resources to refine manipulations for interacting with the system [10].

2.3 Benefits of touchless interaction

According to de la Barré et al., circumstances that touchless interaction is favorable over touch-based interaction are as follows [7]:

- In places where a high hygienic condition is required, such as operating rooms for surgery, using touchless interaction for managing medical devices can save time and resources compared to using touch interaction that necessitates sterilization after each surgical operation.
- In locations such as public spaces, where vandalism could take place, using touchless interaction for interactive installations can prevent harm by positioning input and out devices at a distance.
- In environments such as a classroom, where interactive systems have big displays at a distance for shared use by a number of people, touchless interaction can be useful.
- When there is a very short time to search, hold, and understand input devices, touchless interaction comes handy compared to touch interaction.

In addition, the user could possibly find a touchless interaction more interesting and enjoyable to use than a touch interaction even if it is more challenging to use and more inclined to error. While reasons for this haven't been researched enough so far, one hypothesis is that feeling of gaining magical power that comes when controlling something remotely without needing to physically touch it plays as the major factor [7].

2.4 Applications of touchless interaction

Each technology has suitable applications and markets according to its ecological niche. Thus, it is incorrect to predict that new technologies such as Microsoft Kinect that enable touchless interaction would put an end to the keyboard and mouse, replacing touch interaction [2]. Touch and touchless interaction coexist and compensate each other, thriving in applications that are suitable for each. Below, examples of touchless interaction that found its application in automotive, consumer, and healthcare sectors, are introduced.

FIGURE 1. Amazon Echo that receives voice input from the user [11]

In the consumer sector, touchless interaction is adapted to a Smart home where advanced automation systems are integrated to enable managing heating, lighting, and electronic devices remotely. In particular, smart speakers that function as a hub of the Smart home, such as Amazon Echo and Google Home, are achieving commercial success in this area [6]. Figure 1 illustrates Amazon Echo that controls other connected smart devices with the voice command. It is integrated with Amazon's virtual assistant Alexa, which reacts to the wake word 'Alexa'. The user can send the voice command to the device by speaking naturally after the wake word, to perform various operations [12].

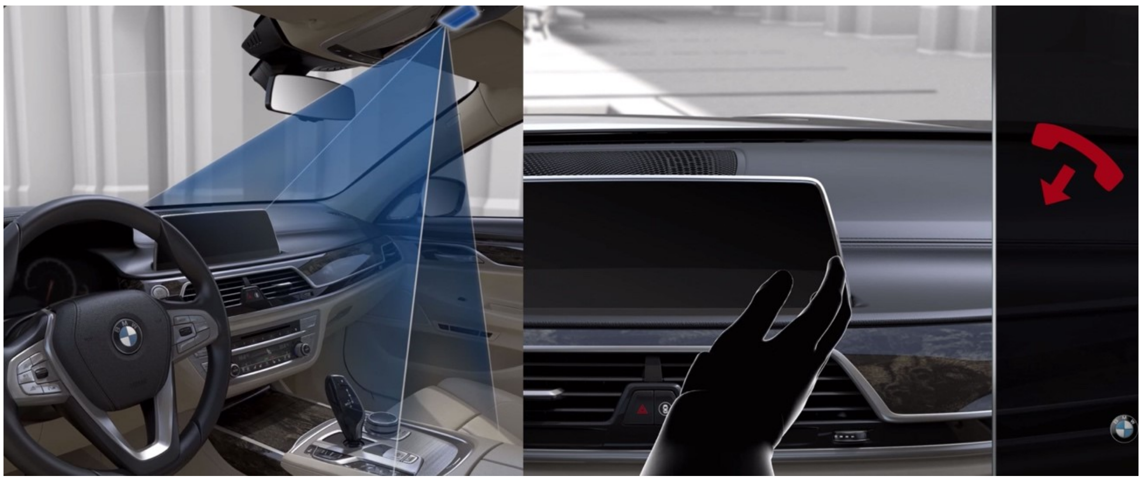


FIGURE 2. BMW gesture control system that allows operating various function with hand gestures [13]

In the automotive sector, touchless interaction is being adapted as it makes utilizing a built-in system easier and decreases the probability of accidents caused by the distraction of the driver by reducing the need for taking eyes away from the road [14]. For example, BMW gesture control system that uses hand gestures for performing various functions such as accepting/rejecting an incoming call and increasing/decreasing volume. Hand gestures of the driver are detected by a camera in the roof lining that scans an area in front of the dashboard, as shown on the left side of Figure 2. On the right side of the figure, a hand gesture (swiping the hand towards the passenger side, across the control display) that rejects an incoming call is illustrated [13].

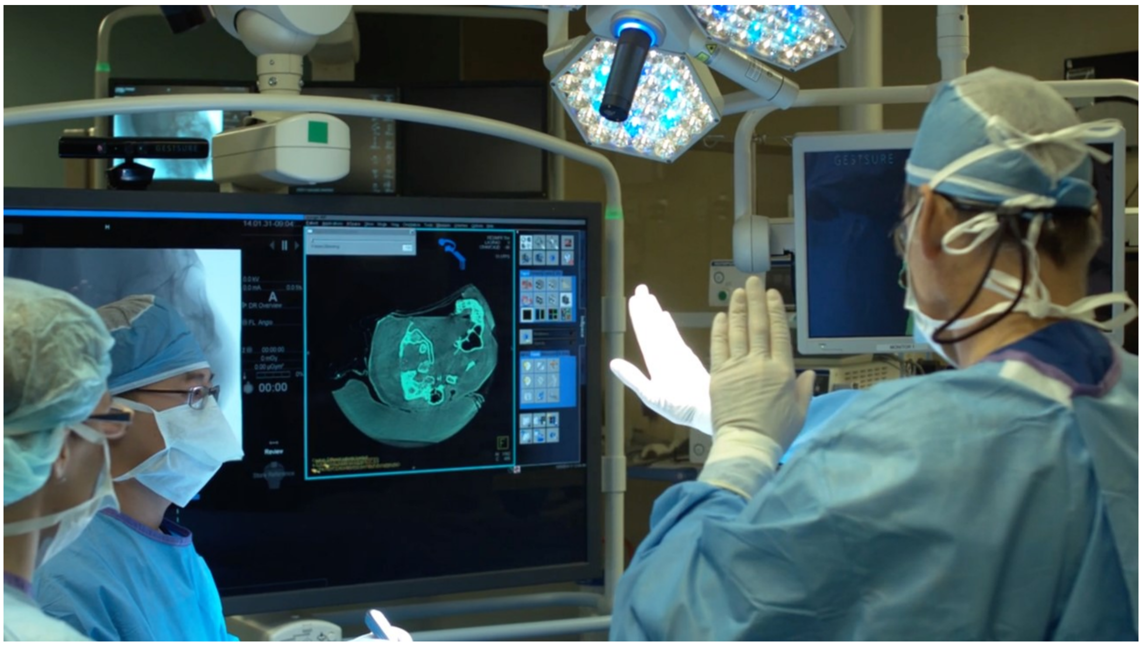


FIGURE 3. Image manipulation with touchless hand gestures using GestSure system [15]

In the healthcare sector, such as hospitals, an interest in touchless interaction is growing as it offers an effective solution for keeping the environment sterilized by removing the need for touching the systems [16]. An example of it is GestSure system that utilizes Microsoft Kinect for enabling surgeons to navigate MRI and CT scans with touchless hand movements [17]. Figure 3 shows a surgeon manipulating a medical image in the operating room without breaking sterility by using touchless interaction.

3 Prototyping a touchless user interface for interacting with a website

3.1 Concept

The prototype is built as a single-page application using React, which is a JavaScript library for building user interfaces, to be able to dynamically update the content of the demo website, without reloading the entire application. By doing so, the prototype application is made to keep functioning without interruption, when the user navigates between pages of the demo website. To interacts with the content of the demo website, the touchless user interface sits on top of the demo website, as illustrated in Figure 4.

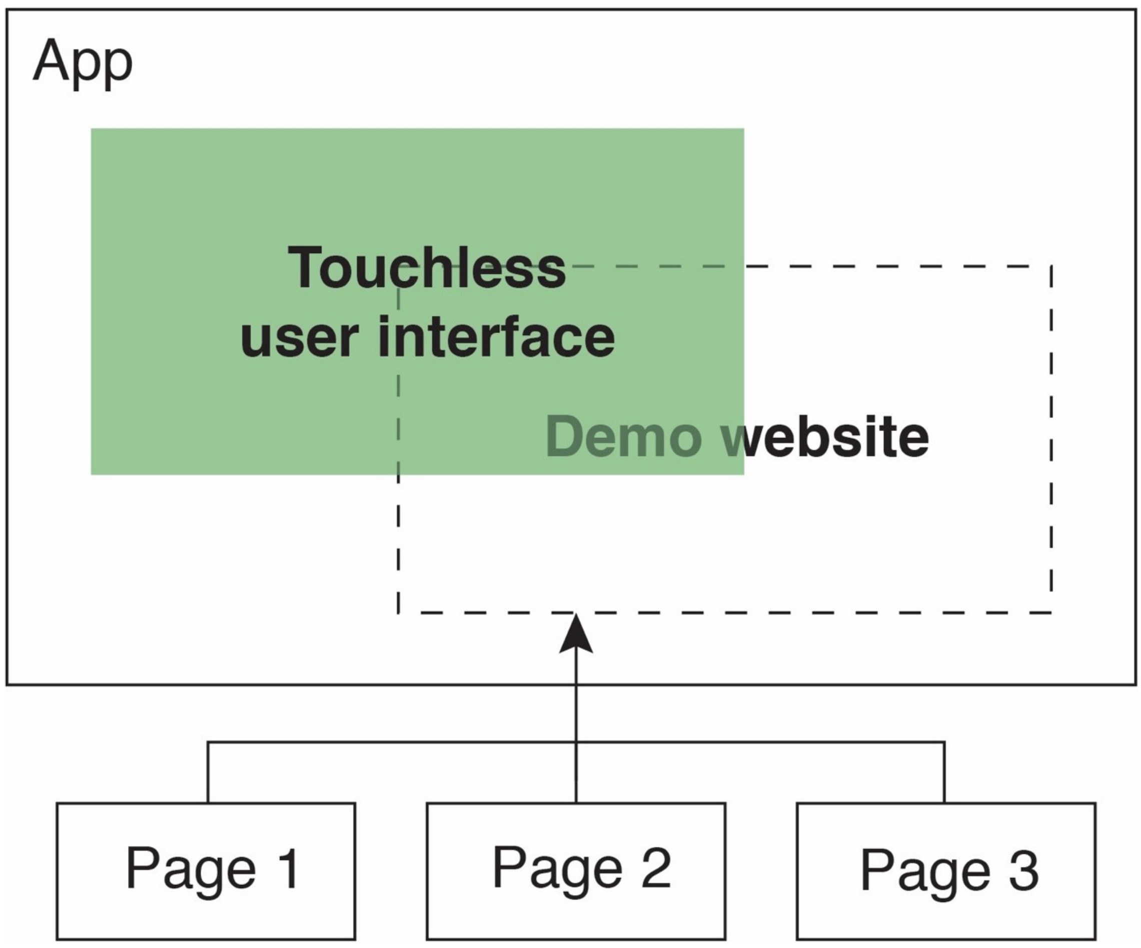


FIGURE 4. Structure of the prototype

The touchless user interface is designed to be controlled using input from the face. The user’s face is captured on a video stream from the webcam, and by running face tracking and facial expression recognition on the video stream using face-api.js, the position of the face and facial expression are received. The position of the face is used to enable pointing operation, and facial expression is used to trigger click operation.

For performing point and click operations, two different concepts are implemented. The first concept is to use a point-pointer that is similar to a traditional mouse cursor for the operations. The point-pointer moves in the browser’s viewport, according to the position of the face, and when it hovers over a clickable element of the website, click operation can be triggered by making a facial expression corresponding to an emoji appeared on it, as illustrated in Figure 5.

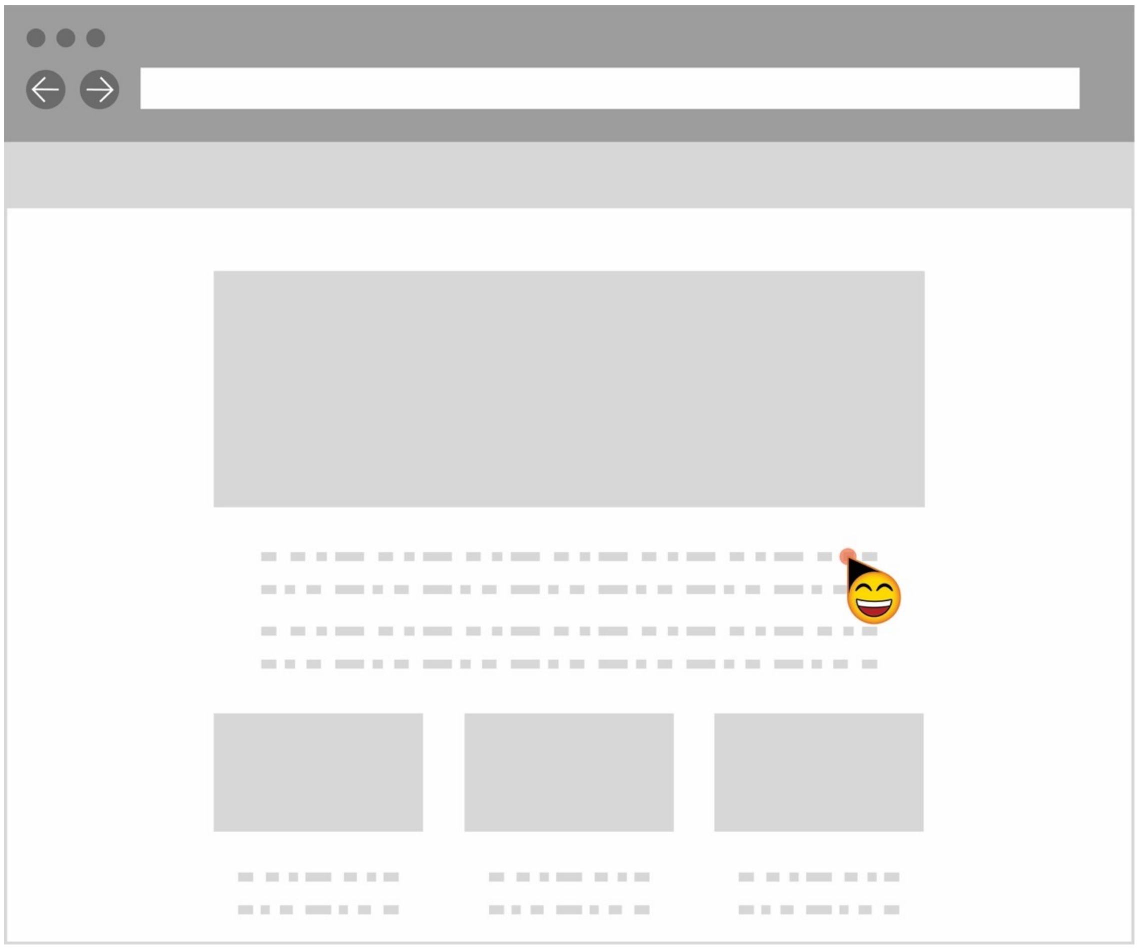


FIGURE 5. Pointing a clickable element with the point-pointer

The second concept is to use an area-pointer which points an area in the 3x3 grid that covers the entire viewport for the operations. The area-pointer points an area according to the position of the face, and detects clickable elements beneath the pointed area. Clicking a specific one among the multiple elements in the pointed area is achieved by assigning different facial expressions for triggering click operation on each element. The facial expression that triggers click operation is indicated on each element with a corresponding emoji, as illustrated in Figure 6.

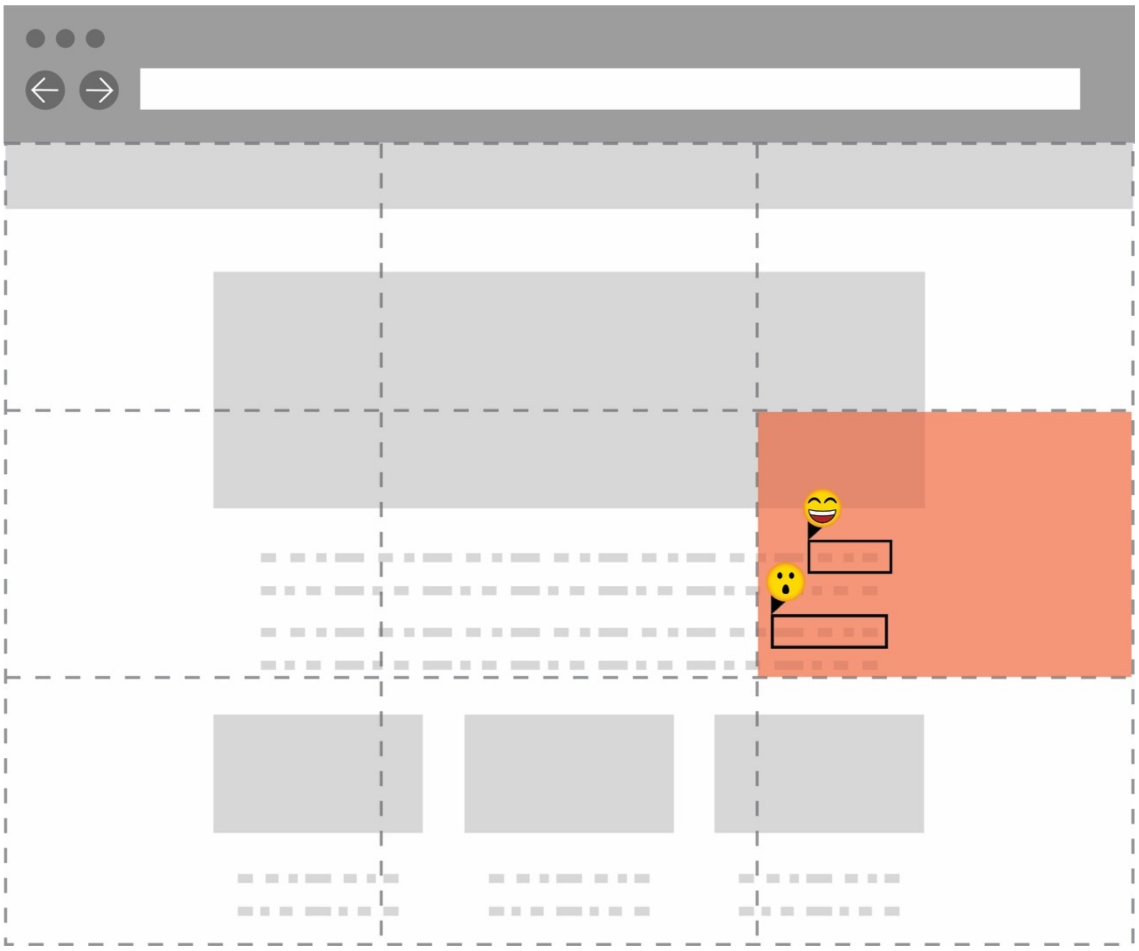


FIGURE 6. The area-pointer detecting multiple clickable elements in the pointed area

Scrolling operations are performed by clicking buttons that are made to appear at the top (for scrolling up) or at the bottom (for scrolling down) of the viewport, when the point-pointer and the area-pointer are pointing upward or downward from the boundaries that are defined for each. When the point-pointer and the area-pointer are pointing leftward or rightward from the defined boundaries, buttons for triggering paging operations are made to appear at the left (for paging backward) or at the right (for paging forward) of the viewport.

3.2 Input for controlling the prototype

3.2.1 Getting a video stream from the webcam

In the web browser, accessing the video stream from the webcam is achieved by using the `MediaDevices.getUserMedia()` method provided by Web API. The method asks the user permission for accessing a media input such as a webcam or a microphone, and once the user accepts permission, it returns the `MediaStream` object that contains the input from the requested media device [18]. For the prototype, a video stream with a low resolution (320 x 240 pixels) is received from the webcam for fast performance.

It is found that the received video stream using the method is not mirrored, resulting in its left and right direction to be the other way around from the user's. Thus, the position information received by tracking face on the video stream needs to be handled accordingly to represent the user's left and right correctly.

3.2.2 Getting the position of the face for point operation

When the face is detected from the video stream, face-api.js returns a bounding box (a blue-lined rectangle in Figure 7) that contains the detected face, and this bounding box provides its coordinates relative to the video stream. While the center position of the bounding box (a red circle in Figure 7) can be used as input for point operation, it is found that it doesn't track the face sufficiently when the face is turned left/right or tilted up/down.

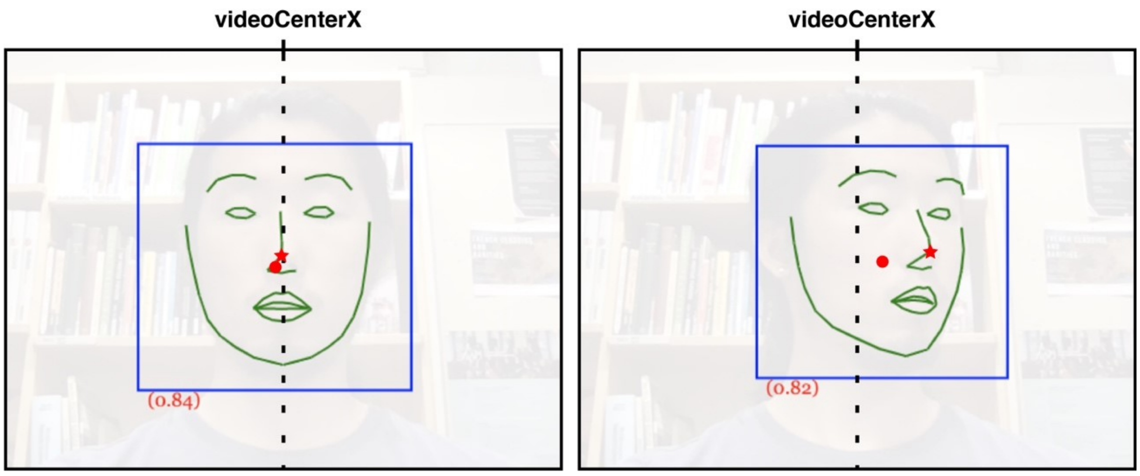


FIGURE 7. The bounding box and facial landmarks tracking the face that is facing front and right

Thus, face detection is made further to get facial landmarks (green lines in Figure 7) of the detected face, and the position of the nose tip (a red star in Figure 7) is received from it. Figure 7 shows how the center of the bounding box and the position of the nose tip track the face that is facing front and right. On the left side of the figure where the face is directly facing the webcam, both are tracking the face similarly. However, on the right side of the figure where the face is turned right, the position of nose tip tracks the face better compared to the center of the bounding box. To move the center of the bounding box to the same position as the nose tip, the user has to move the entire upper body to shift the position of the head. Therefore, the position of the nose tip that better represents turning/tilting of the face is chosen as input data for pointing operation. In Figure 8, facial landmarks of the detected face returned by face-api.js are illustrated in detail.

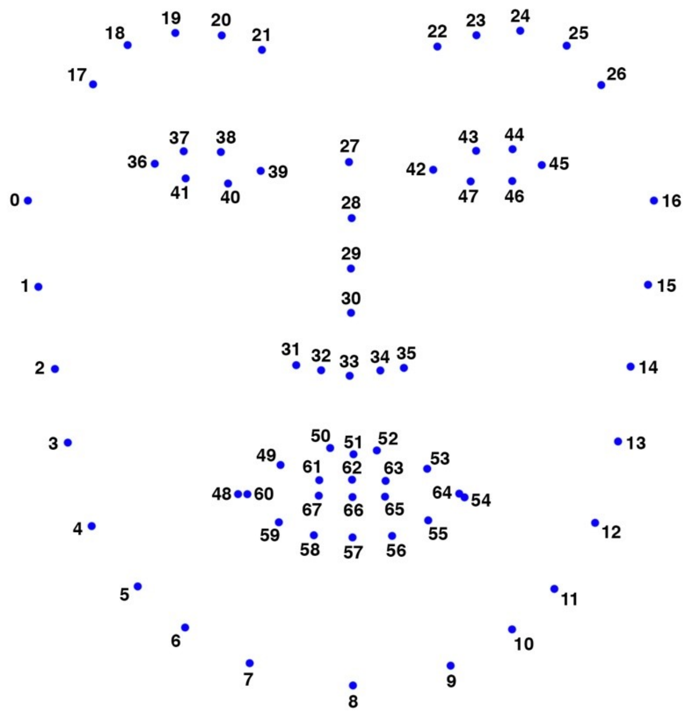
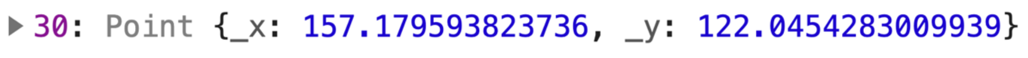


FIGURE 8. 68 facial landmarks where point number 30 indicates the nose tip

The facial landmarks are returned as an array of 68 points, and coordinates of the nose tip can be accessed by referring to the index number 30 in the array. The coordinates are relative to the video stream on which the face detection runs. For instance, with the video stream of 320 x 240 pixels, X and Y coordinates of the nose tip range from 0 to 320 and 0 to 240, respectively. An example of the returned X and Y coordinates of the nose tip, when the face detection runs on the video stream of 320 x 240 pixels, is shown in Figure 9.

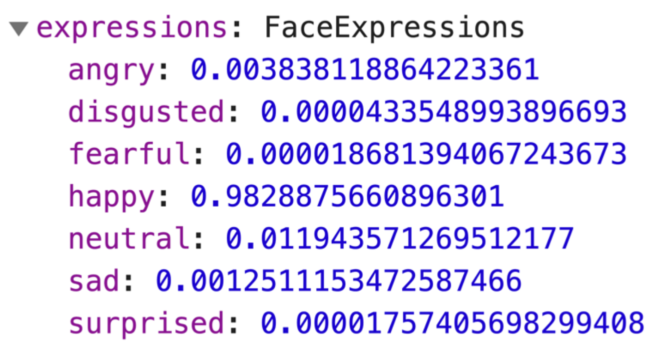


```
▶ 30: Point {_x: 157.179593823736, _y: 122.0454283009939}
```

FIGURE 9. An example of the X and Y coordinates of the nose tip

3.2.3 Getting signal for click operation from the facial expression

With face-api.js, the following seven facial expressions can be detected from the face: angry, disgusted, fearful, happy, neutral, sad, and surprised. An example of a returned JavaScript object that contains the result of facial expressions recognition is shown in Figure 10.



```
▼ expressions: FaceExpressions
    angry: 0.003838118864223361
    disgusted: 0.0000433548993896693
    fearful: 0.0000186813940672436 73
    happy: 0.9828875660896301
    neutral: 0.011943571269512177
    sad: 0.0012511153472587466
    surprised: 0.000017574056982994 08
```

FIGURE 10. A result of facial expressions recognition when the face is making a happy facial expression

In the object, each facial expression is represented as a 'key: value' pair, where the key is a string representation of the facial expression, and the value is a numeric value that indicates a probability of the facial expression being made by the face. The probability value ranges between 0-1, where 0 indicates impossibility, and 1 indicates certainty. In this prototype, facial expression with a probability above 0.85 is considered as an expression that the user is intentionally making.

For the point-pointer (described further in section 3.3) that points a single element at a time, ‘happy’ facial expression is used to trigger click operation. Figure 11 shows a code snippet that checks the probability of happy facial expression to determine if the user wants to trigger click operation on the pointed element. The `expressions` object contains the result of facial expression recognition, as shown in Figure 10.

```
let probability = expressions['happy'];

if ( probability > 0.85 ) {
    // Trigger click operation on the pointed element
}
```

FIGURE 11. A code snippet checking if the user is making ‘happy’ expression to trigger click operation

For the area-pointer (described further in section 3.4) that points an area, every facial expression, except the ‘neutral’ expression, is used to trigger click operation since multiple clickable elements can be detected in the pointed area. The ‘neutral’ expression is excluded since it is considered as a default expression that the user is making without intention for performing any operation. A code snippet in Figure 12 checks the probability of every facial expression provided by face-api.js (except ‘neutral’ expression) to trigger click operation on a specific element among multiple elements in the pointed area.

```
let detected = null;
let probability = 0;

Object.keys(expressions).forEach(function(expression) {
    if ( expression !== 'neutral' && expressions[expression] > probability ) {
        probability = expressions[expression];
        detected = expression;
    }
});

if ( probability > 0.85 ) {
    if ( detected === 'happy') {
        // Trigger click operation on the element #1 in the pointed area
    } else if ( detected === 'surprised') {
        // Trigger click operation on the element #2 in the pointed area
    } else if ( detected === 'sad') {
        // Trigger click operation on the element #3 in the pointed area
    } else if ( detected === 'angry') {
        // Trigger click operation on the element #4 in the pointed area
    } else if ( detected === 'disgusted') {
        // Trigger click operation on the element #5 in the pointed area
    } else if ( detected === 'fearful') {
        // Trigger click operation on the element #6 in the pointed area
    }
}
```

FIGURE 12. A code snippet checking what facial expression the user is making to trigger click operation on a specific element in the pointed area

By using JavaScript `Object.key()` method on the object `expressions` (shown in Figure 10), an array consists of names of the facial expression is returned, and by running the `forEach()` method on the array, probability of each facial expression in the `expressions` is checked. A facial expression with the highest probability is determined at the end of the `forEach()` execution, and its name and probability are assigned to variables `detected` and `probability`, respectively. Then, when the probability of detected facial expression is above 0.85, click operation is triggered on the specific element according to the facial expression of the user.

3.2.4 Preventing unwanted multiple clicks

It is found that triggering click operation by checking the probability of facial expression as described in section 3.2.3 causes undesirable multiple clicks since the facial expression is made and fade out gradually over time, unlike instant action of pressing and releasing a button on the mouse. For example, even if the user begins to stop making a facial expression after achieving click operation, click operation could be still triggered continuously until the expression fades out enough so that the probability of it decreases below 0.85. To solve the issue, conditional statements (in code snippets shown in Figures 11 and 12) that were only checking if the probability of the detected facial expression is above 0.85, are adjusted to check the value of a variable `isClicked` as well, as shown in Figure 13.

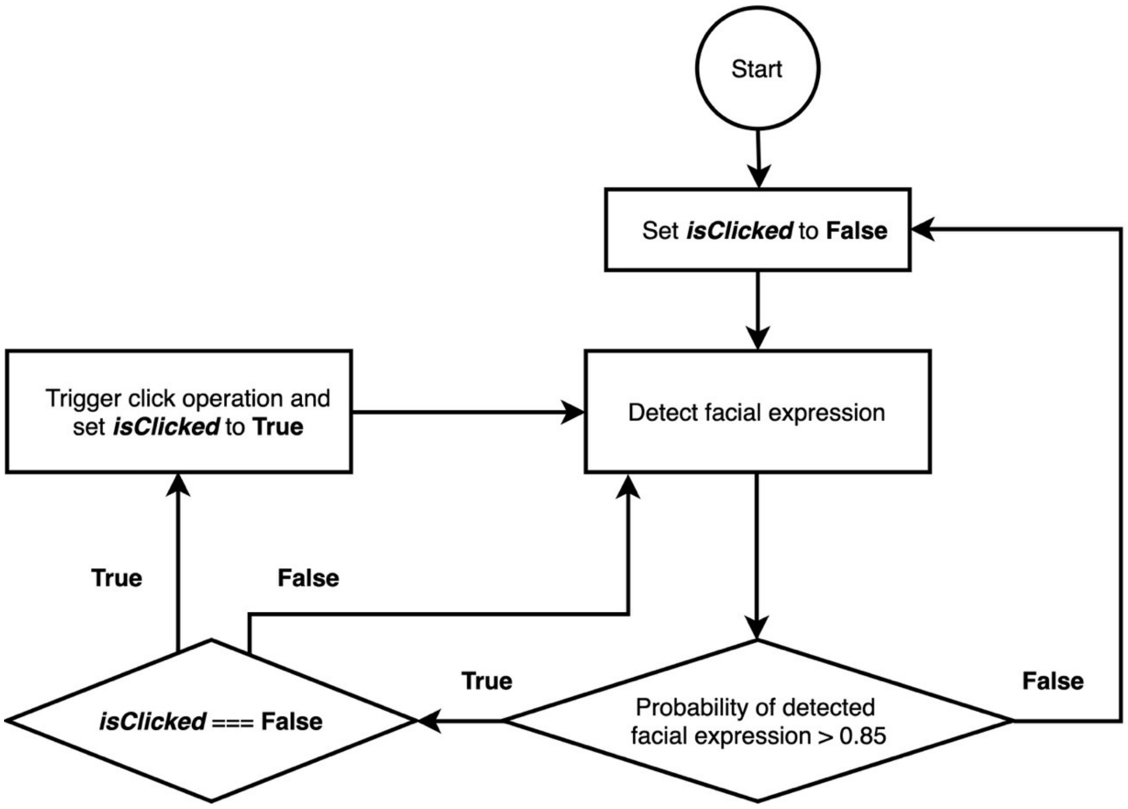

Start
Set *isClicked* to **False**
Trigger click operation and set *isClicked* to **True**
Detect facial expression
True
False
isClicked === **False**
True
Probability of detected facial expression > 0.85
False

FIGURE 13. Conditional logic for preventing multiple clicks

The variable `isClicked` is set to `False` initially, allowing to trigger click operation when a probability of detected facial expression goes above 0.85. When click operation is fired by the facial expression, `isClicked` is set to `True`, to prevent triggering extra clicks even if the probability of the detected facial expression is still above 0.85. When the probability drops below 0.85, indicating that the facial expression that triggered the latest click operation is faded out, `isClicked` is set back to `False`, enabling to trigger click operation when the user newly makes a facial expression with the probability higher than 0.85.

3.3 Point and click using the point-pointer

3.3.1 Creating the point-pointer

The point-pointer that moves according to the position of the user's face is designed to be visually distinguishable from the operating system's pointer in order to avoid confusion, as shown in Figure 14. When its pointy end hovers on the clickable elements, a smiling face emoji representing happy facial expression that triggers click operation is made to appear on its round body. Also, it is sized bigger than the operating system's pointer to be noticed easily.

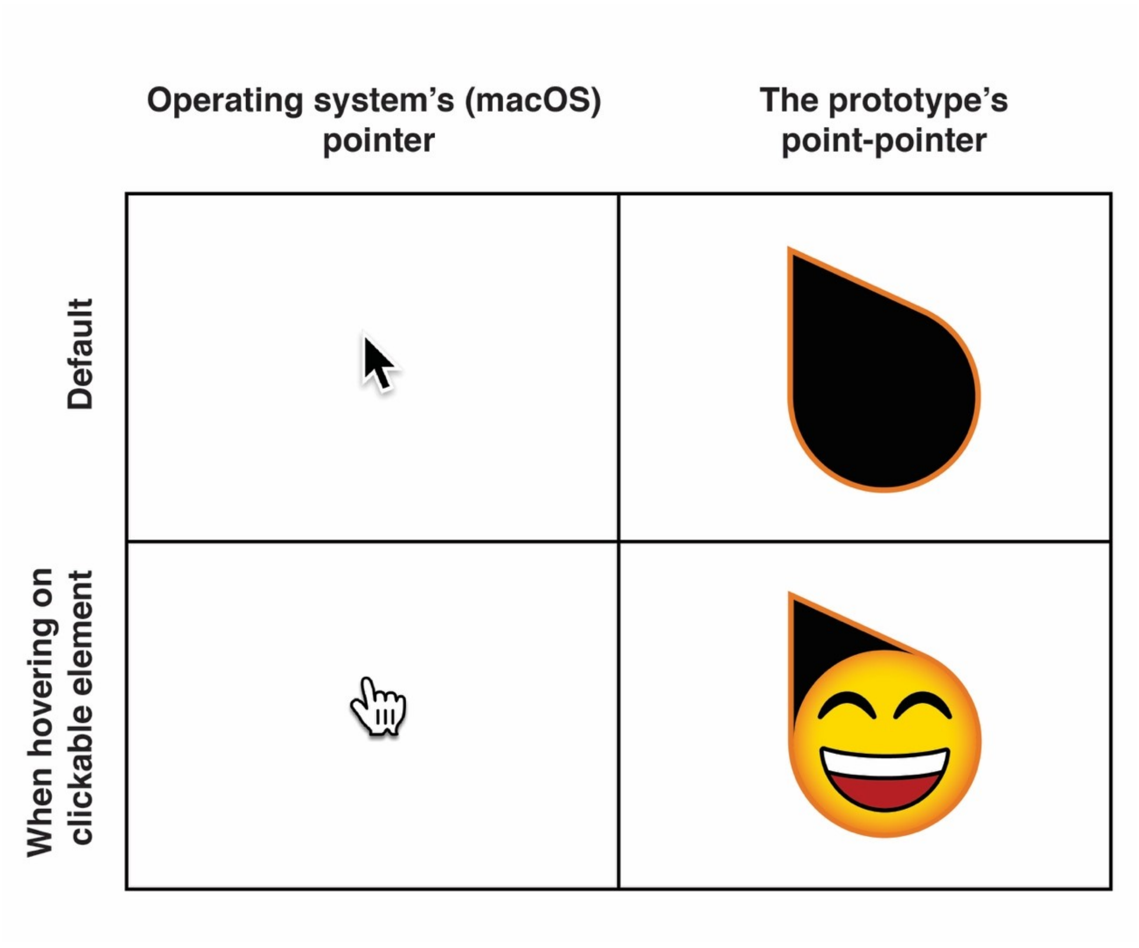

Operating system's (macOS) pointer
The prototype's point-pointer
Default
When hovering on clickable element

FIGURE 14. Operating system's pointer and face-controlled point-pointer

3.3.2 Mapping the position of the face to the viewport

As described in section 3.2.2, coordinates received by tracking the position of the nose tip is relative to the dimensions of the video stream on which the face detection is run, not to the dimensions of the viewport. Using a video stream that has the same dimensions as the viewport is not a feasible option since the dimension of the video stream that can be provided by the webcam is limited while the dimension of the viewport can vary significantly depends on how the user sizes the web browser. Thus, the coordinate of the nose tip in the video stream is processed with a formula shown in Figure 15, to be mapped relative to the viewport regardless of dimensions of the video stream and the viewport.

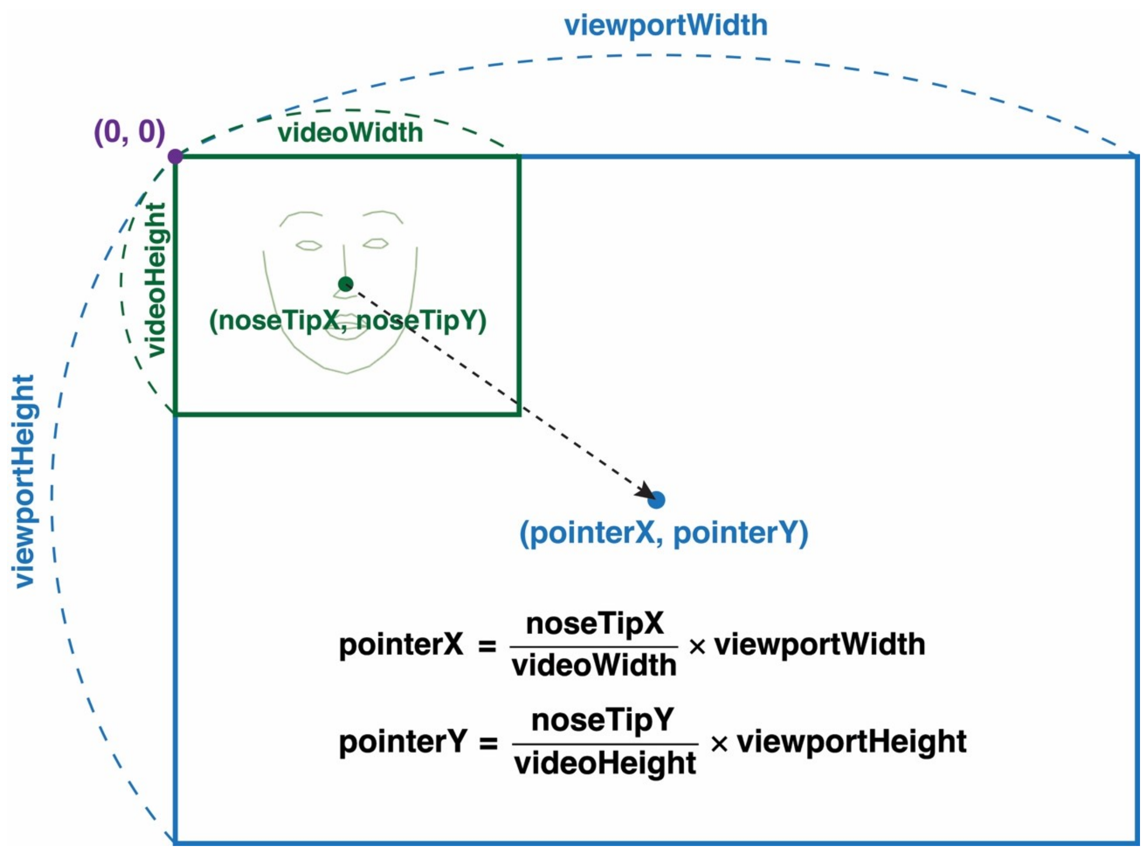

viewportWidth
(0, 0)
videoWidth
videoHeight
(noseTipX, noseTipY)
viewportHeight
(pointerX, pointerY)
pointerX = noseTipX / videoWidth × viewportWidth
pointerY = noseTipY / videoHeight × viewportHeight

FIGURE 15. Mapping the position of the face to the viewport

The width and the height of the viewport are received by accessing the `Window.innerWidth` and the `Window.innerHeight` property that are provided the Web API, and they are updated accordingly when the viewport is resized by the user to maintain the mapping. In addition, to minimize the distance that the user's face needs to move the point-pointer to the desired point in the viewport, a multiplier value `sensitivity` is added to the formula, as shown in Figure 16. The `sensitivity` is a positive number that is greater than or equal to 1. The higher the `sensitivity`, the less movement is required from the user to move the pointer to the desired point.

$$\text{pointerX} = \left(\frac{\text{noseTipX}}{\text{videoWidth}} \times \text{viewportWidth} \times \text{sensitivity}\right) - (\text{sensitivity} - 1) \times \frac{\text{viewportWidth}}{2}$$

$$\text{pointerY} = \left(\frac{\text{noseTipY}}{\text{videoHeight}} \times \text{viewportHeight} \times \text{sensitivity}\right) - (\text{sensitivity} - 1) \times \frac{\text{viewportHeight}}{2}$$

FIGURE 16. A formula accommodating a multiplier value `sensitivity`

3.3.3 Moving the point-pointer

Moving the point-pointer in the viewport is achieved by using CSS. First, the `position` property of the point-pointer is set to `fixed` so that it is positioned relative to the viewport. Then the point-pointer's `z-index` property is set to a higher number than that of the elements in the demo website to make it always visible in the viewport. Finally, the point-pointer is moved by assigning the value of the `pointerX` and the `pointerY` (described in section 3.3.2) to its CSS `top` and `right` properties, respectively. The `pointerY` value is assigned to the `right` property instead of the `left` property, since its left and right directions are the other way around from the user's, as explained in section 3.2.1. Also, the point-pointer's CSS `transition` property is set to `top 300ms, right 300ms;`, in order to move it smoothly without jittering.

With the `sensitivity` value (described in Figure 16), the pointer can move beyond the viewport area, which is not the desired result. Thus, to prevent the point-pointer from moving beyond the border of the viewport area, the value of the `pointerX` and the `pointerY` is limited to stay in a range from 0 to the `Window.innerWidth` and a range from 0 to the `Window.innerHeight`, respectively.

3.3.4 Detecting an element with the point-pointer

Coordinates depicted by a pointy end of the point-pointer is used to detect the element on the website. The coordinates of the pointy end are received by using the `Element.getBoundingClientRect()` method of the Web API. The method returns a `DOMRect` object that represents a rectangle containing the element on which it is called. By accessing the properties of the `DOMRect` object, the size of the element and its position relative to the viewport can be obtained [19]. In Figure 17, calling the method on the point-pointer to get the coordinates of its pointy end is illustrated. By accessing the `left` and the `top` properties of the returned `DOMRect` object (shown as a rectangle with a purple border), the coordinates of the pointy end (shown as a green circle) are received.

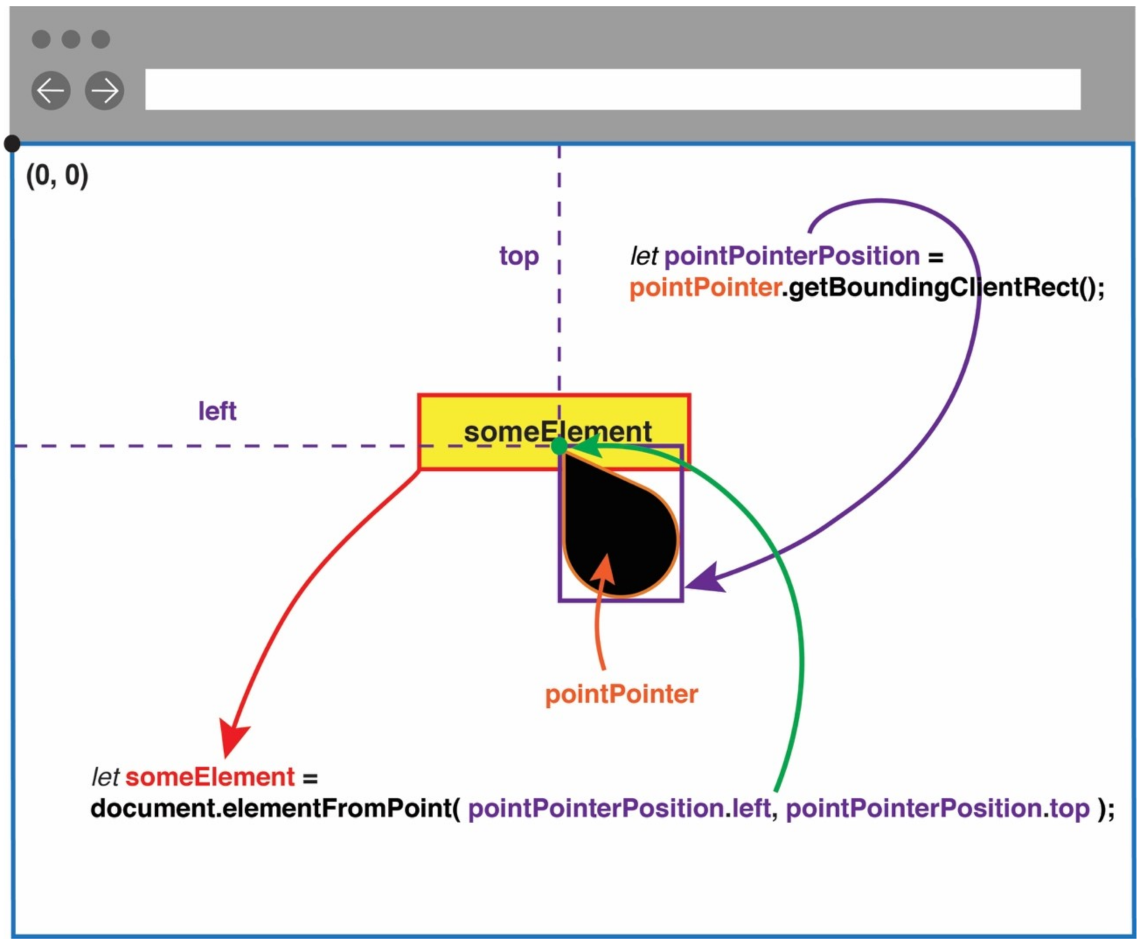
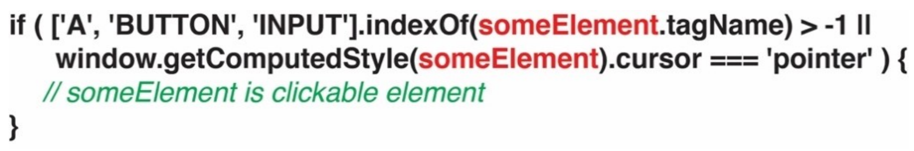

(0, 0)
top
left
let pointPointerPosition =
pointPointer.getBoundingClientRect();
someElement
pointPointer
let someElement =
document.elementFromPoint(pointPointerPosition.left, pointPointerPosition.top);

FIGURE 17. Detecting an element at coordinates depicted by the pointy end of the point-pointer

The received coordinates are passed as parameters of the `Document.elementFromPoint()` method of Web API. The method returns the `Element` object that is located at the specified coordinates relative to the viewport [20]. The point-pointer's CSS `pointer-events` property is set to `none` so that an element beneath the point-pointer is returned, not the point-pointer itself. The returned element by the method is indicated as a rectangle with a red border in Figure 17.

3.3.5 Checking if the pointed element is clickable

The detected element with the point-pointer is examined if it is clickable. The element is considered as a clickable if its HTML tag is one that reacts to the click operation, such as `<a>`, `<button>` and `<input>`, and it can be checked by accessing the element's `tagName` property which returns the HTML tag name of the element [21]. However, this approach for determining the element's clickability doesn’t work if a click event handler is added by the `addEventListener()` method to the element with an HTML tag that doesn’t react to the click operation, such as `<div>`, `<span>`, `<p>`, and so on. Unfortunately, while some browsers’ developer tool provides a method, such as `getEventListeners()` of Chrome DevTools Console [22], that returns event listeners registered on the specified element on their console, the Web API doesn’t provide a native method for it at the time of writing. Thus, as a workaround, the element’s CSS `cursor` property is checked if it is set to `pointer`, which commonly indicates that the element is clickable. Checking the `cursor` property of the element is achieved by using the `getComputedStyle()` method of the Web API. A code snippet that checks `tagName` and CSS `cursor` property of the detected element (a variable `someElement` in Figure 17) is shown in Figure 18.

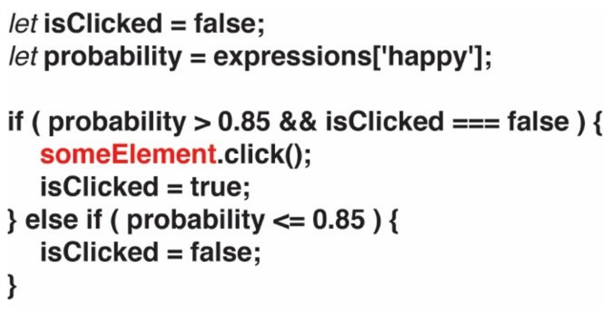


```
if ( ['A', 'BUTTON', 'INPUT'].indexOf(someElement.tagName) > -1 ||
    window.getComputedStyle(someElement).cursor === 'pointer' ) {
  // someElement is clickable element
}
```

FIGURE 18. A code snippet checking if a detected element is clickable

3.3.6 Clicking the pointed element

Once `someElement` is turned out to be a clickable element after checking as shown in Figure 18, the user can perform click operation on the element by making ‘happy’ facial expression, as described in section 3.2.3. Figure 19 shows a code snippet that triggers click operation on `someElement` with the Web API’s `click()` method. Note that the value of `isClicked` is also checked in the condition of the if statement to prevent unwanted multiple clicks as described in section 3.2.4, along with the probability of a happy facial expression.



```
let isClicked = false;
let probability = expressions['happy'];

if ( probability > 0.85 && isClicked === false ) {
   someElement.click();
   isClicked = true;
} else if ( probability <= 0.85 ) {
   isClicked = false;
}
```

FIGURE 19. A code snippet triggering click operation on the pointed element with ‘happy’ facial expression

3.4 Point and click using the area-pointer

3.4.1 Creating the area-pointer

The area-pointer is implemented to be able to point an area in the 3x3 grid that covers the browser’s viewport. It is created by nesting nine `<div>` elements inside of a container `<div>`. The container `<div>` is made to cover the entire viewport by setting its CSS properties as followings: `position: fixed; top: 0; left: 0; width: 100%; height: 100%;`. Its `z-index` property is set to the higher value than that of elements in the demo website so that it is always positioned above the deme website. The nine `<div>` elements inside of the container are positioned in 3 rows and 3 columns to form a 3x3 grid by setting their CSS properties as followings: `float: left; width: 33.333%; height: 33.333%;`. The border of the 3x3 grid is made visible to visually indicate each area, while its opacity is set low to minimize obstructing visibility of the demo website beneath it. In Figure 20, the area-pointer is shown along with `<div>` elements that structure it. Note that the position of each area in the 3x3 grid is named in its class attribute.

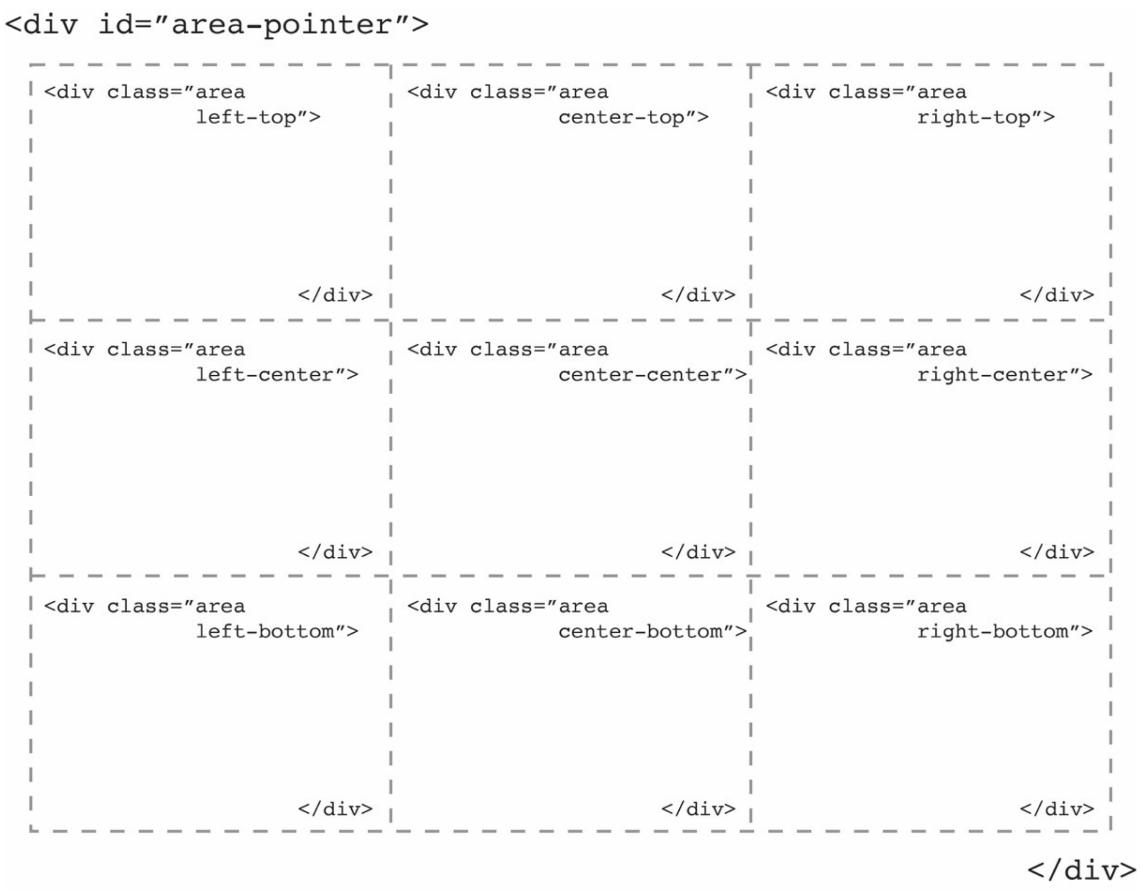


FIGURE 20. The area-pointer

3.4.2 Selecting an area according to the position of the face

The position of the face is separated into nine positions according to where the nose tip of the detected face is positioned in relation to the center position of the video stream to make it corresponds to the areas in the 3x3 grid. Boundaries that separate the position of the face is illustrated as dashed rectangles in the video stream in Figure 21. They are set compactly around the center of the video stream so that the position of the face can be easily switched with a slight movement of the user’s face. If the detected nose tip is positioned outside of the defined boundaries, buttons that trigger scrolling or paging operation are made appear, depends on where the nose tip is outside of the defined boundaries. (described later in section 3.5)

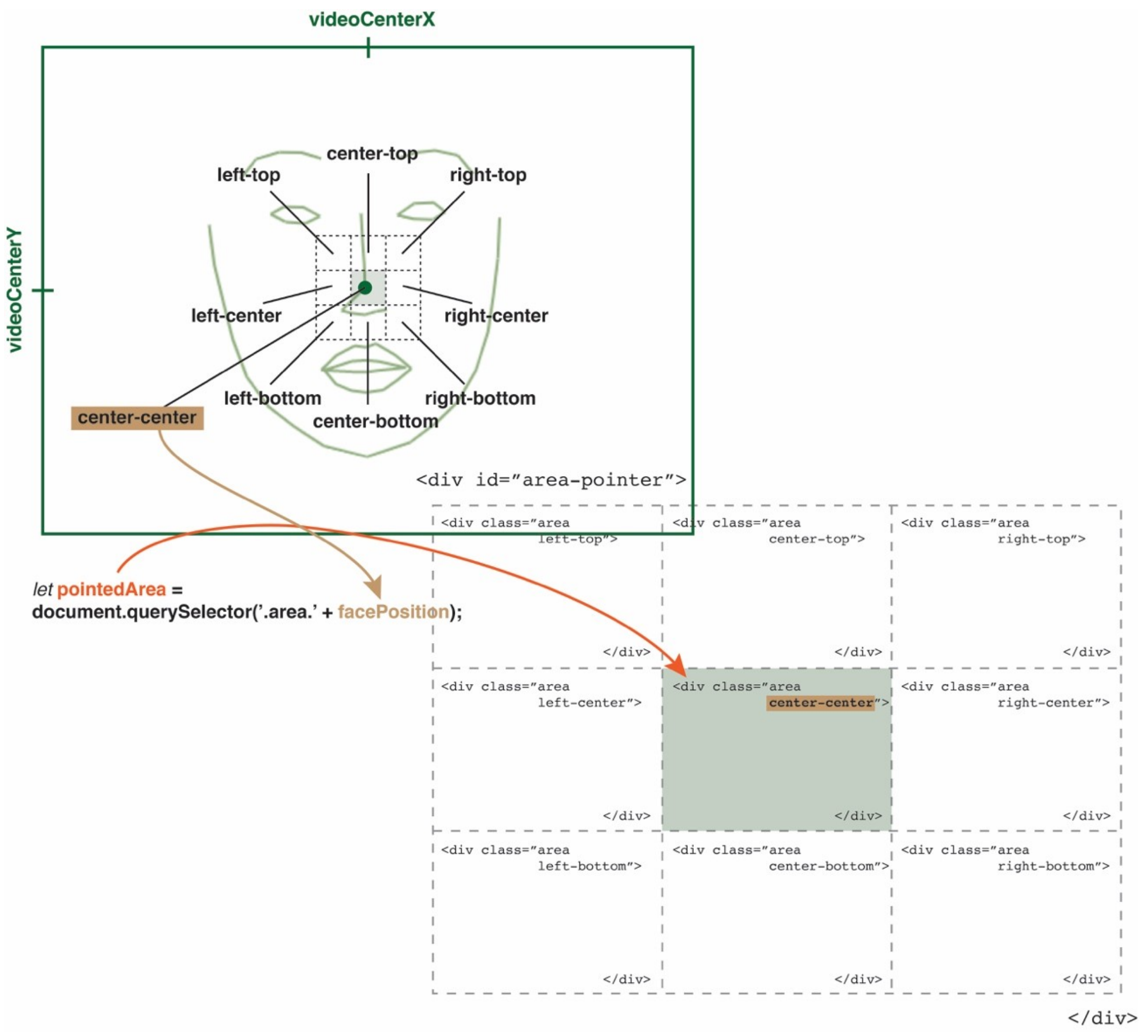
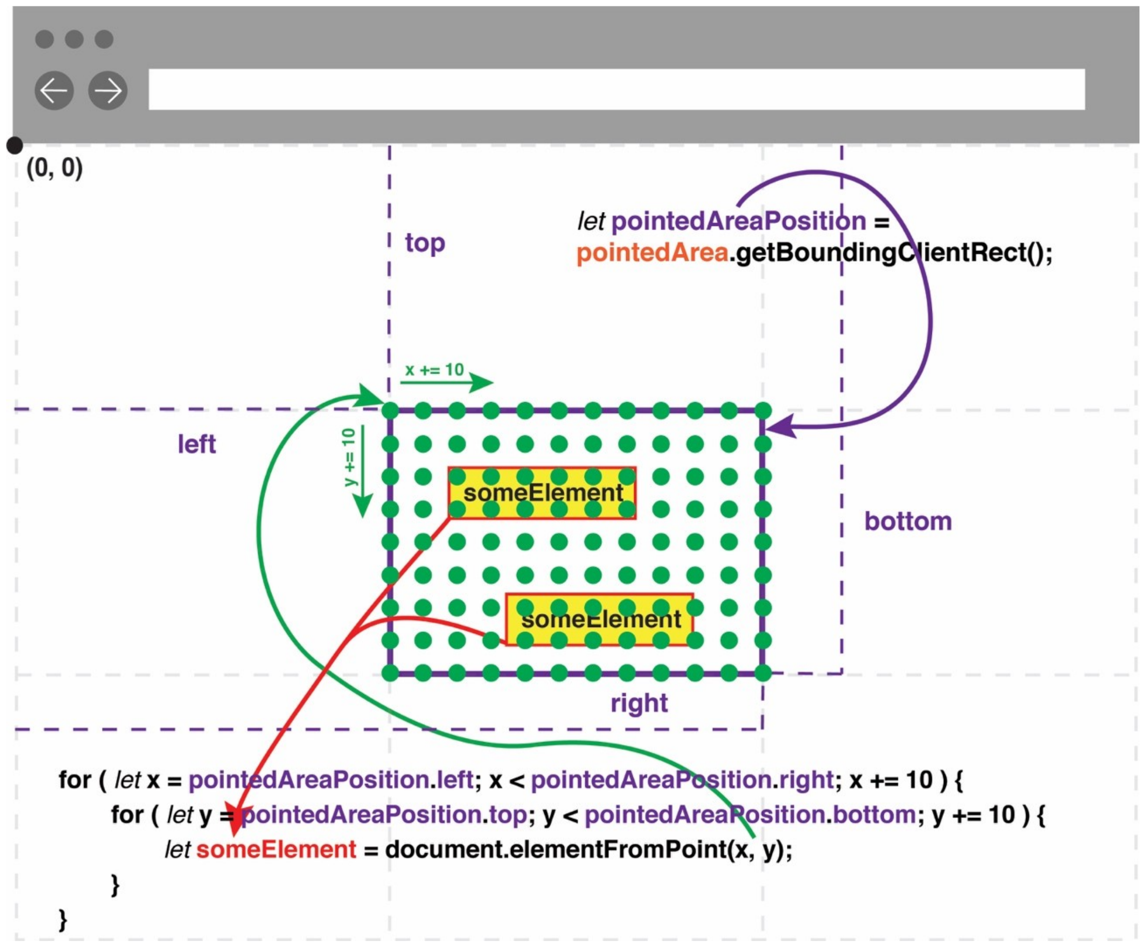

videoCenterX
videoCenterY
center-top
left-top
right-top
left-center
right-center
left-bottom
right-bottom
center-bottom
center-center
let pointedArea =
document.querySelector('.area.' + facePosition);
<div id="area-pointer">
<div class="area left-top"> </div>
<div class="area center-top"> </div>
<div class="area right-top"> </div>
<div class="area left-center"> </div>
<div class="area center-center"> </div>
<div class="area right-center"> </div>
<div class="area left-bottom"> </div>
<div class="area center-bottom"> </div>
<div class="area right-bottom"> </div>
</div>

FIGURE 21. Selecting an area in the area-pointer according to the position of the face

Once the position of the face is recognized by the defined boundaries, an area in the 3x3 grid that corresponds to the position of the face is selected, by passing the position of the face in conjunction with '.area.' as a parameter to the `Document.querySelector()` method, as shown in Figure 21. The selected area is highlighted with CSS `background-color` property so that the user can identify which area in the 3x3 grid is pointed. The opacity of the `background-color` is set to a low value to make the demo website beneath it visible.

3.4.3 Detecting elements in the pointed area

Elements of the demo website beneath the pointed area are detected by executing the same method used for the point-pointer to get an element at a point (described in section 3.3.4), throughout the pointed area using nested for-loop, as illustrated in Figure 22.



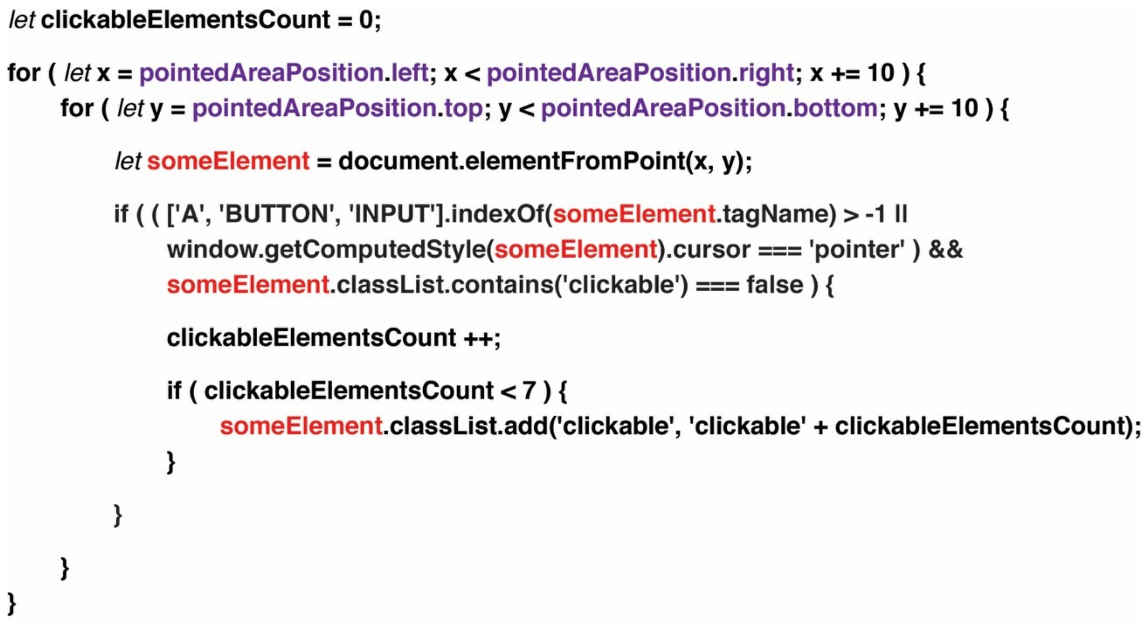
```
let pointedAreaPosition =
pointedArea.getBoundingClientRect();
```

```
for ( let x = pointedAreaPosition.left; x < pointedAreaPosition.right; x += 10 ) {
      for ( let y = pointedAreaPosition.top; y < pointedAreaPosition.bottom; y += 10 ) {
            let someElement = document.elementFromPoint(x, y);
      }
}
```

FIGURE 22. Detecting elements in the pointed area

The area-pointer's CSS `pointer-events` property is set to `none` so that elements beneath the pointed area is returned from the `document.elementFromPoint()` method, not the area-pointer itself. The x and y loop variables in the nested for-loop are incremented by 10 at the end of each iteration to point the coordinates on every tenth pixel (shown as green circles in Figure 22) in the pointed area. This way, the faster performance is achieved compared to checking every coordinate in the area, and at the same time, the area is still checked densely enough to not miss elements beneath it.

The detected elements in the pointed area are examined if they are clickable by checking the elements' `tagName` and CSS `cursor` property, in the same way as described in section 3.3.5. Then, to the elements that are identified as clickable, two classes, ‘clickable’ and ‘clickable’ + `clickableElementsCount`, are added to indicate them as clickable elements, as shown in Figure 23.



```
let clickableElementsCount = 0;

for ( let x = pointedAreaPosition.left; x < pointedAreaPosition.right; x += 10 ) {
    for ( let y = pointedAreaPosition.top; y < pointedAreaPosition.bottom; y += 10 ) {

        let someElement = document.elementFromPoint(x, y);

        if ( ( ['A', 'BUTTON', 'INPUT'].indexOf(someElement.tagName) > -1 ll
            window.getComputedStyle(someElement).cursor === 'pointer' ) &&
            someElement.classList.contains('clickable') === false ) {

            clickableElementsCount ++;

            if ( clickableElementsCount < 7 ) {
                someElement.classList.add('clickable', 'clickable' + clickableElementsCount);
            }

        }

    }
}
```

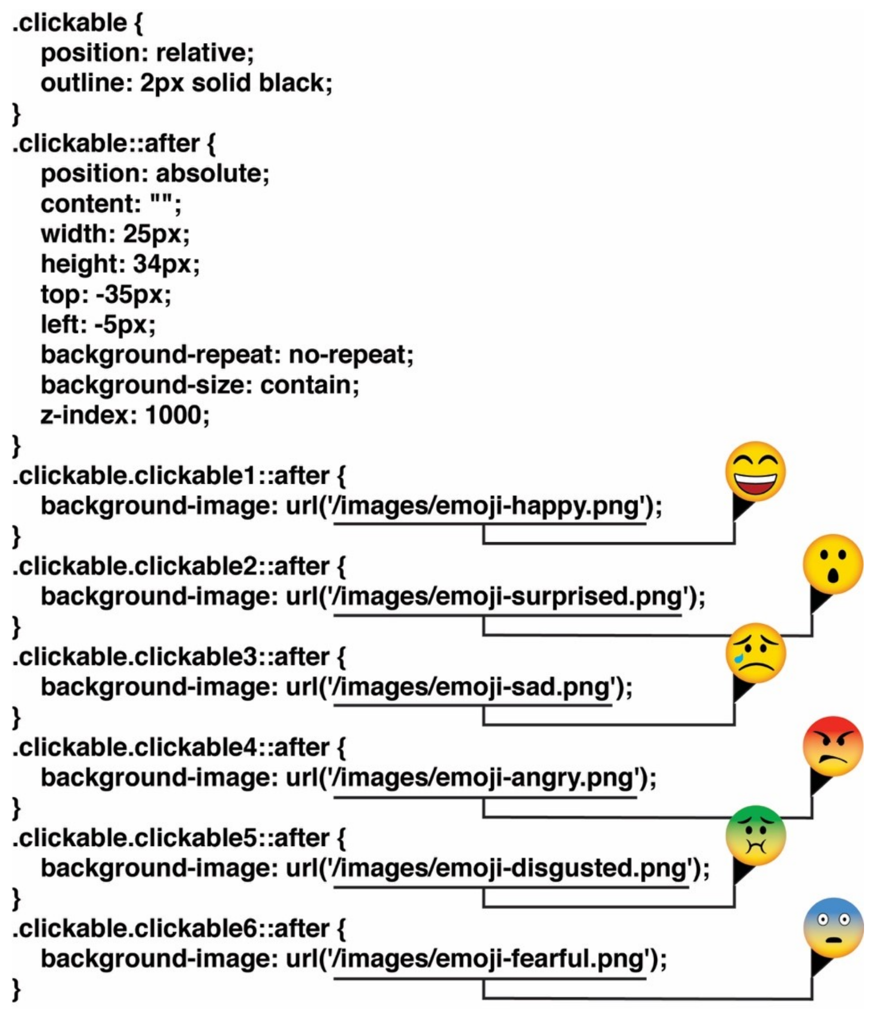
FIGURE 23. A code snippet that adds two indicator classes to each element identified as clickable

The first class, ‘clickable’, is added to the element as an indicator that the element is already detected as a clickable element. As illustrated in Figure 22 (multiple green circles on the elements), the same element can be detected multiple times while checking points in the pointed area, and by examining whether the element already has the class ‘clickable’, indicating the same element multiple times as a clickable element is prevented. With the second class, ‘clickable’ + `clickableElementsCount`, each clickable element is distinguished from another. A variable `clickableElementsCount` provides unique numeric value to each clickable element as it increments by 1 when the clickable element is detected. A total number of clickable elements that can be detected at once in the pointed area is limited to six to correspond to the number of facial expressions that trigger click operation on the area-pointer.

Whenever the pointed area is changed, or the page is scrolled, clickable elements are detected newly. Before the new detection is made, the two added classes on the previously detected clickable elements are removed, so that they can be appropriately added to the newly detected clickable elements.

3.4.4 Indicating click-triggering facial expression with emojis

Emojis corresponding to the facial expression that triggers click operation is displayed on each detected element, using CSS `::after` selector, as shown in Figure 24.



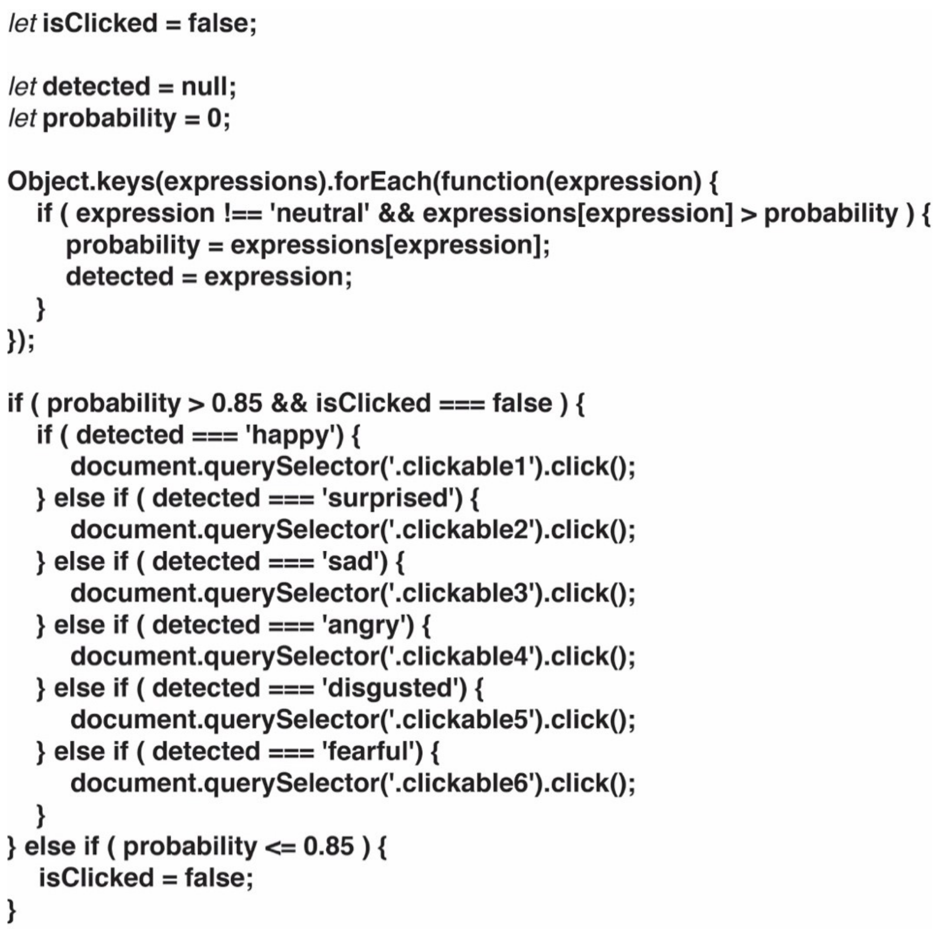
```
.clickable {
   position: relative;
   outline: 2px solid black;
}
.clickable::after {
   position: absolute;
   content: "";
   width: 25px;
   height: 34px;
   top: -35px;
   left: -5px;
   background-repeat: no-repeat;
   background-size: contain;
   z-index: 1000;
}
.clickable.clickable1::after {
   background-image: url('/images/emoji-happy.png');
}
.clickable.clickable2::after {
   background-image: url('/images/emoji-surprised.png');
}
.clickable.clickable3::after {
   background-image: url('/images/emoji-sad.png');
}
.clickable.clickable4::after {
   background-image: url('/images/emoji-angry.png');
}
.clickable.clickable5::after {
   background-image: url('/images/emoji-disgusted.png');
}
.clickable.clickable6::after {
   background-image: url('/images/emoji-fearful.png');
}
```

FIGURE 24. Displaying emojis on clickable elements using CSS `::after` selector

The detected elements (elements with ‘clickable’ class) are positioned by setting their CSS `position` property to `relative` so that emojis added to their pseudo-element created with CSS `::after` can be positioned relative to them. In addition, the detected elements are highlighted with CSS `outline` property. While the element’s `border` property could be used for highlighting purpose, the `outline` is used instead since it doesn't affect the content of the element, unlike the `border` that takes up space from the content [23].

3.4.5 Clicking the element in the pointed area

In order to trigger click operation on the element in the pointed area, the facial expression of the user is checked as described in Figure 12 in section 3.2.3. Also, a value of variable `isClicked` is checked to prevent unwanted multiple clicks as described in section 3.2.4. The detected elements are selected with their unique class name ('clickable1', 'clickable2', and so on), and click operation is triggered on the element that has an emoji corresponding to the user's facial expression, as shown in Figure 25.



```
let isClicked = false;

let detected = null;
let probability = 0;

Object.keys(expressions).forEach(function(expression) {
   if ( expression !== 'neutral' && expressions[expression] > probability ) {
      probability = expressions[expression];
      detected = expression;
   }
});

if ( probability > 0.85 && isClicked === false ) {
   if ( detected === 'happy') {
      document.querySelector('.clickable1').click();
   } else if ( detected === 'surprised') {
      document.querySelector('.clickable2').click();
   } else if ( detected === 'sad') {
      document.querySelector('.clickable3').click();
   } else if ( detected === 'angry') {
      document.querySelector('.clickable4').click();
   } else if ( detected === 'disgusted') {
      document.querySelector('.clickable5').click();
   } else if ( detected === 'fearful') {
      document.querySelector('.clickable6').click();
   }
} else if ( probability <= 0.85 ) {
   isClicked = false;
}
```

FIGURE 25. A code snippet triggering click operation on the element in the pointed area according to the user's facial expression

3.5 Scrolling and paging

3.5.1 Scrolling up and down

When the point-pointer reaches the top or bottom border of the browser's viewport, or when the area-pointer receives the position of the face that is located upward or downward, outside of the boundaries that are defined to separate the position of the face (described in section 3.4.2), it is considered that the user intends to perform scroll operation.

The buttons that trigger scrolling up and down operations are made to appear only when those operations are possible. Whether the document can be scrolled up is determined by checking the value of `Window.pageYOffset` property that shows how much the document is scrolled vertically [24]. When its value is bigger than 0, indicating the document can be scrolled up, a button that can trigger scrolling up operation is made visible, as shown on the left side of Figure 26. Whether the document can be scrolled down is determined by comparing the sum of the `Window.pageYOffset` and the `Window.innerHeight` (the height of the viewport) with the `document.documentElement.scrollHeight` that indicates the total height of the document. When the sum is smaller than the total height of the document, indicating the document can be scrolled down, a button that can trigger scrolling down operation is made visible, as shown on the right side of Figure 26.

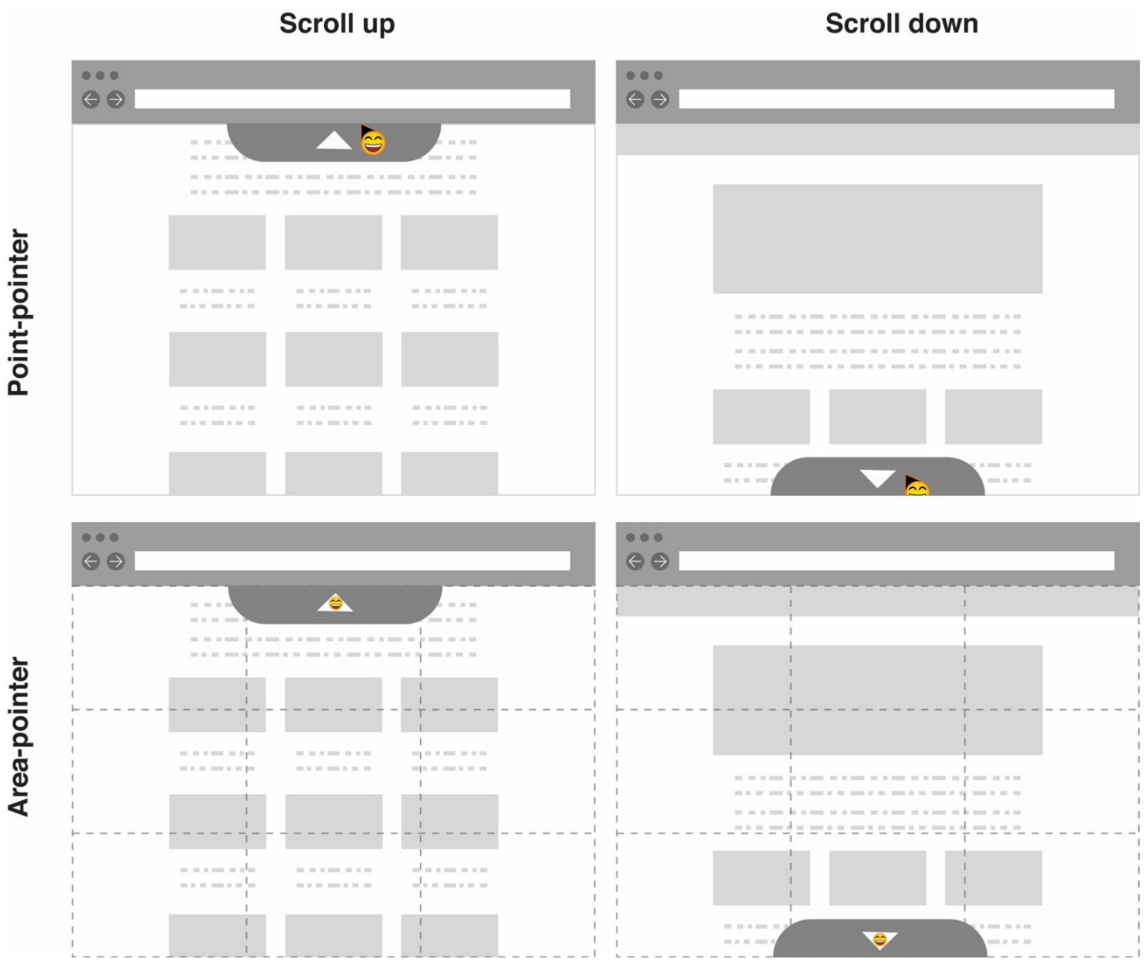


FIGURE 26. Buttons for scrolling operation

For the area-pointer that utilizes multiple facial expressions to trigger click operation, a ‘happy’ facial expression is chosen for triggering click operation on the scroll buttons, and it is indicated by a corresponding emoji on the buttons.

Once the button for the scroll operation is clicked, the `Window.scrollBy()` method is triggered to scroll the document. The method’s first and second parameters indicate the amount of the horizontal and the vertical pixel value to be scrolled, respectively [25]. Thus, a positive number is passed as the second parameter of the method to scroll down the document, while a negative number is passed instead to scroll up the document.

3.5.2 Paging backward and forward

When the point-pointer reaches the left or right border of the browser’s viewport, or when the area-pointer receives the position of the face that is located leftward or rightward, outside of the boundaries that are defined to separate the position of the face (described in section 3.4.2), it is considered that the user intends to perform paging operation.

The buttons that trigger paging operations are made to appear only when the current page has a previous page or a next page to go. In order to determine whether the current page has a previous or a next page, the pages visited by the user is tracked by examining a `location` object that is provided by React Router, which is a popular routing library for React. The `location` object contains the properties regarding the current page's location, including the `key` property, which has a unique string value presenting the current page [26]. When the user navigates between pages, the `location.key` value of the page is added in order of occurrence to an array that is created when the application is initiated. If the user pages backward or forward, adding the `key` of the page to the array is prevented since the same `key` is already existing in the array. When the current page's `key` value is located other than at the beginning of the array, it is determined that the current page has the previous page, and a button for paging backward is made to appear as shown in the left side of Figure 27. When the current page's `key` value is located other than at the end of the array, it is determined that the current page has the next page, and a button for paging forward is made to appear as shown in the right side of Figure 27.

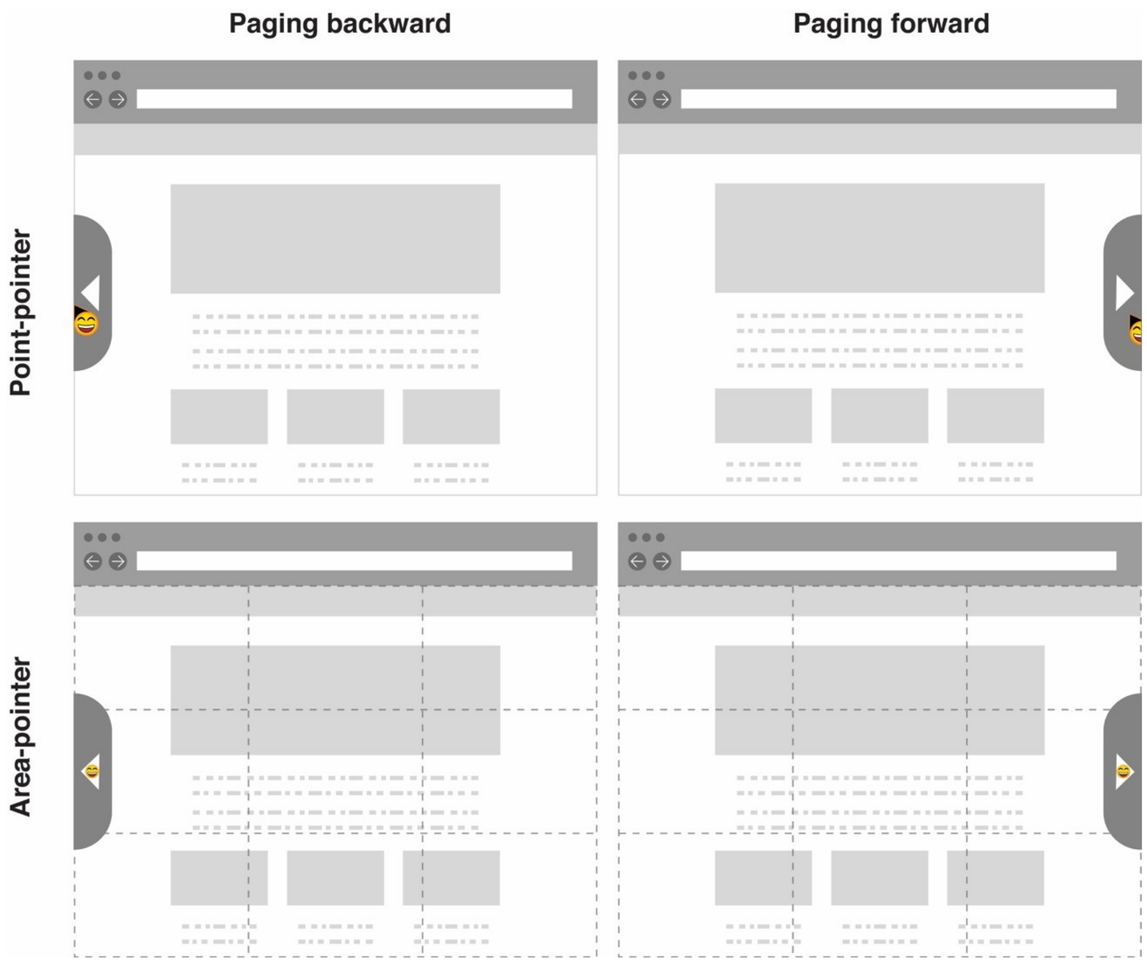


FIGURE 27. Buttons for paging operation

Once the button for paging operation is clicked, the page is moved backward or forward using the History interface of Web API. When the paging back button is clicked, the `Window.history.back()` method is triggered to move one page backward from the current page, and when the paging forward button is clicked, the `Window.history.forward()` method is triggered to move one page forward from the current page.

4 Discussion

4.1 Results

The prototype of touchless user interfaces for interacting with a website is successfully developed, and methods used for it are demonstrated in the thesis. For performing point and click operation, two different concepts, the point-pointer and the area-pointer, are implemented. The point-pointer provided a familiar way to interact with the website by imitating a traditional mouse cursor. On the other hand, the-area pointer provided a rather experimental way for interacting with the website by that pointing an area instead of a point. Scrolling and paging operations are triggered when the buttons that are designated for the operations are clicked by the point-pointer or the area-pointer. While the operations that were set as the goal of the prototype are made functional, certain limitations were found in the process of development, as described below.

4.2 Limitations

4.2.1 Lack of a native method for checking event listeners of the element

Since Web API doesn’t provide a native method for checking event listeners of the element, as described in section 3.3.5, the element’s CSS `cursor` property is examined (assuming its value to be `pointer` if the element is clickable) instead, to detect the element that is made to react to click event by the the `addEventListener()` method. While this workaround works in most cases, it could cause issues in case the `cursor` property of the clickable element is set to something else other than `pointer`, not following the convention.

4.2.2 Limited number of detectable facial expressions

As a total number of facial expressions that can trigger click operation is limited to six (every facial expression detectable using face-api.js, except the ‘neutral’ expression), a total number of elements that the area-pointer can detect at once in the pointed area is restricted to six, as a consequence. A workaround for this limitation could be to optimize the layout of the website to avoid cramming more than six clickable elements in an area in the 3x3 grid.

4.3 Development suggestions

While the prototype for this thesis is implemented to be controlled by input from the user’s face using face tracking and facial expression recognition, other types of input could also be integrated by utilizing other techniques such as hand gesture recognition or voice recognition, to develop the prototype further. For example, voice recognition could be useful when implementing a typing feature that is lacking in the prototype. Also, the prototype could be developed as a web browser extension so that it can interact with any other websites in the web browser, not only with the demo website that is created as a part of the prototype.

REFERENCES

[1] Interaction Design Foundation. Human-Computer Interaction (HCI). Read: 20.07.2019. https://www.interaction-design.org/literature/topics/human-computer-interaction

[2] D. Wigdor et al. Brave NUI World: Designing Natural User Interfaces for Touch and Gesture. Elsevier (2011). Read: 20.07.2019.

[3] Vincent Mühler. face-api.js. Read 02.08.2019. https://github.com/justadudewhohacks/face-api.js/

[4] Google Brain Team. TensorFlow.js. Read 02.08.2019. https://www.tensorflow.org/js

[5] Facebook. React. Read 02.08.2019. https://github.com/facebook/react

[6] Katerina Kralik. The Future of Touchless Technologies: Voice or Gesture (2018). Read: 05.08.2019. https://program-ace.com/blog/the-future-of-touchless-technologies-voice-or-gesture/

[7] Barré, René de la et al. “Touchless Interaction-Novel Chances and Challenges.” HCI (2009). Read: 01.09.2019. https://www.researchgate.net/publication/215876112_Touchless_Interaction_-_Novel_Chances_and_Challenges

[8] Fortune Business Insights. Speech and Voice Recognition Market Size, Share & Industry Analysis, By Component (Solution, Services), By Technology (Voice Recognition, Speech Recognition), By Deployment (On-Premises, Cloud), By End-User (Healthcare, IT and Telecommunications, Automotive, BFSI, Government, Legal, Retail, Travel and Hospitality and Others) and Regional Forecast, 2019 - 2026 (2019). Read: 26.11.2019.

https://www.fortunebusinessinsights.com/industry-reports/speech-and-voice-recognition-market-101382

[9] MarketsandMarkets. Gesture Recognition and Touchless Sensing Market by Technology (Touch-based and Touchless), Product (Sanitary Equipment, Touch-less Biometric), Industry, and Geography - Global Forecast to 2022 (2019). Read: 18.11.2019.
https://www.marketsandmarkets.com/Market-Reports/touchless-sensing-gesturing-market-369.html

[10] O'Hara et al. On the Naturalness of Touchless: Putting the "Interaction" Back into NUI. ACM Transactions on Computer-Human Interaction (2013).
Read: 23.09.2019.
https://www.researchgate.net/publication/242070747_On_the_Naturalness_of_Touchless_Putting_the_Interaction_Back_into_NUI

[11] Amazon. Amazon Echo Plus: Creating a Smart Home Device Group ECHO (2017). Accessed: 20.11.2019.
https://www.youtube.com/watch?v=UH8I8vKMJ5E

[12] Amazon. Set Up Your Amazon Echo (1st Generation). Read: 15.11.2019.
https://www.amazon.com/gp/help/customer/display.html?nodeId=201601770

[13] BMW USA. Gesture Controls | BMW Genius How-To (2015).
Accessed: 22.11.2019.
https://www.youtube.com/watch?v=wqvAPskg_k0&feature=emb_title

[14] Parkaj Singh. Automotive Gesture Recognition—The Next Level in Road Safety (2019). Read: 22.11.2019.
https://www.electronicdesign.com/automotive/automotive-gesture-recognition-next-level-road-safety

[15] GestSure. A showcase video embedded in GestSure website.
Accessed: 17.11.2019.
https://www.gestsure.com/

[16] Cronin, Seán & Doherty, Gavin. Touchless computer interfaces in hospitals: A review. Health Informatics Journal 25(4):146045821774834 (2018).
Read: 19.11.2019.
https://www.researchgate.net/publication/323107258_Touchless_computer_interfaces_in_hospitals_A_review

[17] Chakraborty, I., Paul, T. Touchless Interaction: Communication with Speech and Gesture. User Experience Magazine, 14(1) (2014).
Read: 17.11.2019.
https://uxpamagazine.org/touchless-interaction/

[18] MDN Web Docs. MediaDevices.getUserMedia(). Read 15.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

[19] MDN Web Docs. Element.getBoundingClientRect(). Read 15.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect

[20] MDN Web Docs. DocumentOrShadowRoot.elementFromPoint().
Read 15.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/DocumentOrShadowRoot/elementFromPoint

[21] MDN Web Docs. Element.tagName. Read 19.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/Element/tagName

[22] Google Developers. getEventListeners(object). Read 20.08.2019.
https://developers.google.com/web/tools/chrome-devtools/console/utilities#geteventlisteners

[23] MDN Web Docs. outline. Read 12.10.2019.
https://developer.mozilla.org/en-US/docs/Web/CSS/outline

[24] MDN Web Docs. Window.pageYOffset. Read 15.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/Window/pageYOffset

[25] MDN Web Docs. Window.scrollBy(). Read 20.08.2019.
https://developer.mozilla.org/en-US/docs/Web/API/Window/scrollBy

[26] React Training. location. Read 26.08.2019.
https://github.com/ReactTraining/react-router/blob/master/packages/react-router/docs/api/location.md