Implementation of UI support for Touch Enabled Media Platforms

Christoffer Joakim Törnroos

Bachelor’s Thesis in Information Technology

Vasa 2021

EXAMENSARBETE

Författare:	Christoffer Joakim Törnroos
Utbildning och ort:	El- och automationsteknik, Vasa
Inriktningsalternativ:	Informationsteknik
Handledare:	Jan Berglund, Novia University of Applied Sciences
	Staffan Granholm, Hibox Systems Oy

Titel: Implementation av pekskärmsstöd för interaktiva mediaplattformar

---

Datum 12.2.2021 Sidantal 55 Bilagor: 1

Abstrakt

Detta examensarbete behandlar implementationen av pekskärmsstöd för Hibox Systems interaktiva media- och informationssystem. Implementationens syfte är att systemet ska kunna användas på pekskärmsbaserade plattformar. Systemet från sin början är utvecklat för TV-apparater och TV-boxar och har därför inget stöd för att användas på en pekskärm. I samband med utvecklingen av pekskärmsstöd ska även ett systemanpassat sätt att använda animationer implementeras för att förbättra användningen av systemet på en pekskärm.

Implementationerna sker i systemets webbapplikation, dlx, som körs på slutanvändarens enhet. Webbapplikationen är utvecklad i JavaScript, CSS och HTML och därav kommer implementationen göras i dessa språk med hjälp av några bibliotek som utökar funktionaliteten i JavaScript men också bidrar med utökad funktionalitet för användning av pekskärmar.

Slutprodukten av arbetet blev en ordentlig bas att använda sig av när funktionaliteten av pekskärmsstödet skulle implementeras i alla diverse applikationer och menyer. I dessa applikationer och menyer kunde även animationer implementeras med hjälp av den nya animationskomponenten.

Språk: engelska Nyckelord: JavaScript, CSS, HTML, pekskärm, animationer

---

BACHELOR'S THESIS

Author:	Christoffer Joakim Törnroos
Degree Programme:	Electrical Engineering and Automation
Specialization:	Information Technology
Supervisor(s):	Jan Berglund, Novia University of Applied Sciences
	Staffan Granholm, Hibox Systems Oy

Title: Implementation of UI support for Touch Enabled Media Platforms

Date February 16, 2021 Number of pages 55 Appendices 1

Abstract

This thesis looks into the implementation of touch support for Hibox Systems interactive media and information system. The purpose of the implementation is that the system can be used on touch enabled media platforms, such as a tablet. From the beginning, the system was only intended to be used on TVs and STBs and that is why there is no support whatsoever for use on a touch enabled platform. In addition to the implementation of touch support, a way of handling and running animations should be implemented as well to improve the use-flow of the UI.

The implementation will be done in dlx; the frontend web application that is run on the end-user devices using JavaScript, CSS, and HTML. This task also includes evaluating several different libraries that will be used to extend JavaScript functionality and also extend touch support functionality.

The results of this task were a proper implementation to lean on when developing for touch enabled platforms allowing for an easy and structured use when in need of touch support when developing new applications or user interaction functionality. Moreover, there is also a user-friendly animation component ready to use when developing new features.

Language: English Key words: JavaScript, CSS, HTML, touch support, animations

Glossary

JavaScript

A lightweight programming language most well-known to use as a scripting language for the web.

Hyper Text Markup Language

A tag-based language that builds the structure of a website or application.

Cascading Style Sheets

Used to style webpages written in HTML.

Document Object Model

Defines the logical structure of a document, generally in web development the HTML structure that is created when the application has been loaded by a browser.

Hyper Text Transfer Protocol

A protocol used widely on the world wide web to fetch resources such as HTML documents.

ECMAScript

ECMAScript is the specification of JavaScript released in yearly cycles. The latest standard is ECMAScript 2020.

npm

An online repository for Node.js projects and command-line utilities for interacting with those projects.

jQuery

A legacy library that extends and simplifies the functionality of JavaScript.

zepto

A minimalist library that can be used instead of jQuery to improve load time.

Hammer.JS

An open-source library that extends and simplifies touch support functionality.

TouchSwipe

An open-source jQuery/Zepto plugin that extends detection of swipes, pinches and drags.

zepto.touch

An open-source Zepto module that extends the use of the Zepto library for simplifying the touch support functionality.

Abbreviations

JS JavaScript

HTML Hyper Text Markup Language

CSS Cascading Style Sheet

DOM Document Object Model

HTTP Hyper Text Transfer Protocol

ES5 ECMAScript 5 (2009)

ES6 ECMAScript 6 (2015)

STB Set-Top Box

UI User Interface

Hibox System

dlx

The frontend webapp used on devices.

hbx

A shared library used among several webapps.

hiboxadmin

The web-based administration tool.

portlet

The building block in a menu structure which usually has a function tied to it.

ClickHandler

The main component in this thesis that handles all pointer related events.

Table of Contents

1 Introduction 1
1.1 Employer 1
1.2 Software 1
1.3 User Interface 1
1.4 Task 2
2 Tools and libraries 4
2.1 HTML 4
2.2 CSS 4
2.3 JS 4
2.4 jQuery 5
2.4.1 Zepto 5
2.5 Hammer.JS 5
3 The principles behind touch support in JavaScript 6
3.1 Events 6
3.1.1 Event handlers 6
3.1.2 Event objects 6
3.1.3 Event propagation & target 7
3.2 Mechanisms 7
3.2.1 Zepto 7
3.2.2 Native JavaScript 8
3.2.3 HTML event handler attributes 8
4 Development 9
4.1 Devices 9
4.1.1 Device profiles 10
4.1.2 The mobile platform 11
4.2 Implementation of touch support 12
4.2.1 The Frontend: dlx 12
4.2.2 Evaluation of extended touch libraries 14
4.2.3 Evaluation of zepto.touch 15
4.2.4 Evalutation of TouchSwipe 16
4.2.5 Evaluation of Hammer.JS – The chosen one 17
4.2.6 The ClickHandler principles 20
4.2.7 The ClickHandler – swipes 23
4.2.8 The ClickHandler – edge swipes 24
4.2.9 The ClickHandler – touch scrolling 26
4.2.10 The ClickHandler – an application use case 28

4.3 Implementation of animations 34
4.3.1 The CSS 34
4.3.2 The plan & why 35
4.3.3 The component 35
5 Conclusion 40
6 References 41
7 Table of figures 43
8 Table of Code Examples 44

1 Introduction

1.1 Employer

Hibox Systems Oy Ab is a global provider and developer of advanced information and entertainment solutions. Hibox designs and delivers TV solutions for telecom operators, media companies, hotels, hospitals and cruise ships. Hibox has a wide knowledge when it comes to television solutions. Founded back in 2005 when a TV-based information and entertainment solution was requested by a local hotel. Since then, the market has thrived, and the solutions have advanced in the same pace as the knowledge. [1]

1.2 Software

Smartroom, the software, which consists of a backend, frontend, and an administration tool, is intended from the beginning to deliver information and entertainment to a television. During the years it has grown bigger with implementations of new features and extended support of TV devices from suppliers such as Philips, Samsung, and LG.

The software consists of a user interface (dlx), a centre, and an administration web application (hiboxadmin). These can be deployed to the cloud or on a local server in the facility. The TVs are connected to the centre and assigned to the room they are located in. Rooms and floors are part of a network that is configured in the administration web application. Once connected, information and entertainment can be delivered and configured to each network or per TV basis.

1.3 User Interface

There are different types of TVs aimed at specific use-cases – consumer, hospitality, and signage. Consumer TVs are the ones you have at home, signage TVs offers solutions to display information in a hospital for example and hospitality TVs offers solutions for delivering information and entertainment in a hotel for example. Since television manufacturers offer browser-based solutions for hospitality TVs, the user interface *dlx* is built using HTML, CSS, and JS, very much alike a standard web application. It is highly customizable and user friendly. A few years ago, a browser on a TV differed a lot from a browser on a computer regarding support for new CSS and JS features. That is starting to pan out. Modern hospitality TVs support a lot of new CSS and JS features, almost everything

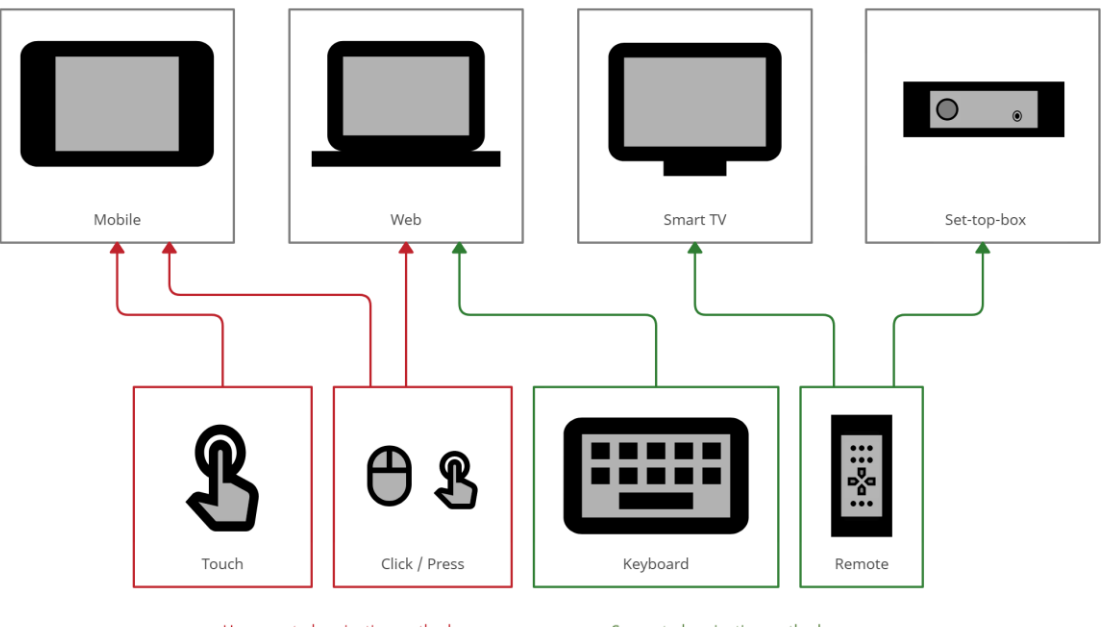
that a computer browser does. However, the processing power of a TV is limited and that is why you need to think about what you use to deliver a good end-user experience.

When developing a UI, interaction and navigation for the end-user is crucial. That is why Hibox Systems have UX experts, to make it as user-friendly as possible. TV software is special, since you only have a remote-control with a set of buttons to work with - which are not even standardized. For each TV model, the same software is used but the remote-control is different from the other. The remote-control buttons need to be mapped to a mapping tree in the software. This way, each specific model that uses their own specific remote-control can be used to navigate the user interface *dlx*.

1.4 Task

The system described earlier is developed with device types using some sort of remote in mind, the system cannot be used on a touch-enabled platform such as a tablet. The mobile platform is already implemented but cannot be used since there is no way for the users to navigate in the UI.

Figure 1. Supported and unsupported navigation methods.



Besides navigating the UI on TV using a remote, using a keyboard in a PC browser also works for development purposes. The keyboard is mapped as a standard remote.

My task is to implement touch support for touch enabled media platforms which means implementing the necessary parts of touch support required for dlx to work on a touch enabled platform. In addition to that, a system adapted way to create and control animations needs to be implemented to improve the user-experience in the UI when using touch or click input.

2 Tools and libraries

2.1 HTML

HTML or Hyper Text Markup Language is a tag-based language that defines the structure of a webpage. Everything that is visually seen on a webpage are HTML elements, such as headers, paragraphs, pictures, and videos. These elements can have classes, ids and attributes assigned so that they can be uniquely or collectively identified by JS and CSS. It is a sort of document that is transferred over the internet with HTTP.

The HTML can be manipulated by JS and styled by CSS. The JS code can either be placed in another file and be imported in the HTML file or be put directly into a script tag in the HTML file. The CSS works by the same principle. [2]

2.2 CSS

CSS or Cascading Style Sheets is a language that defines the style of a structured document, for example HTML. With CSS everything regarding the visual aspect of a webpage can be programmed. Using unique IDs or collective classes, CSS can be applied to several elements with the same line of code or individually to a single element. The colour of a header, the background colour of a webpage, how close to each other all the elements are or defining animations. CSS is a very powerful styling language, if there is an idea, there is a way. [3]

2.3 JS

JS or JavaScript is a lightweight scripting language that is most often used to making a webpage interactive. Animations, button clicks, colour changes, touch gestures and mouse clicks are examples of what JS can handle and manipulate. JS can also be used when developing server-side applications using Node.JS.

JS on a webpage is a client-language, meaning that the code is executed in the user’s browser. It is also visible to the user in the browser’s development tools. However, each browser implements their own JS runtime and each browser differ from the other.

JS is a prototype-based language that supports both object-oriented and functional programming styles. Its standardisation is called ECMAScript. Modern browsers support the first revision ES5 (ECMAScript 2009) and older browsers support at least the original

JavaScript ES3 (ECMAScript 1999). ES6 (ECMAScript 2015) is the second revision and was released back in 2015. [4]

2.4 jQuery

jQuery is a JavaScript library with the intention to extend and simplify the functionality of JavaScript. Tasks that require several lines of code in native JS are wrapped into functions that you can call with a single line of code using jQuery. This makes it easier and faster to use. This library is widely used all around the world in web development. [5]

2.4.1 Zepto

Zepto is a minimalist JavaScript library that can be used instead of jQuery due to its jQuery compatible API. It is less than a third of the size of jQuery and loads much faster. Zepto is split into several modules. The default Zepto library is 5 specific modules including the core, event handling, ajax, web forms handling and support for Internet Explorer. The remaining 12 modules are optional and can be added according to the needs by doing a custom distribution of the library using npm. [6]

2.5 Hammer.JS

Hammer.JS is an open-source library that can recognize touch-gestures such as swipes, pans, and long press. It is a very small library and easy to use when in need for extended touch functionality features. [7]

This is one library among others that serve the same purpose. You can read more about why this library was chosen in chapter *Evaluation of extended touch libraries.*

3 The principles behind touch support in JavaScript

A click, press, swipe, drag – you name it, are user interaction events performed using a mouse or a finger. A general name for the interaction events is pointer events. There are also events that are fired when the webpage has finished loading or a form has been submitted. Almost everything that happens on a webpage is an event. The event is an action that can be responded to if desired. I am going to focus on the events caused by interacting, the pointer events.

3.1 Events

3.1.1 Event handlers

Code Example 1. Defining an event handler on an element using Zepto.

```
1.  $( element ).on( 'click', function( eventObject ) {
2.      doSomething( eventObject );
3.  } );
```

In Code Example 1 above, the event handler is marked in red. This is the function that is executed when the element listened on fires a click event. With the event handler you define what is going to happen when the configured event occurs.

3.1.2 Event objects

Code Example 2. The event object parameter.

```
4.  $( element ).on( 'click', function( eventObject ) {
5.      doSomething( eventObject );
6.  } );
```

The event object is marked in red. The object contains data about the event. In this case, a click.

In Appendix 1 an example is shown of how an event object could look like. There is a lot of data about the event that can be used for various things. That is why it is often sent as a parameter to the event handler.

3.1.3 Event propagation & target

When working with events in JavaScript, event propagation is a must to understand, at least to some degree. Event propagation consists of two pieces, capturing and bubbling. To showcase bubbling, which is relevant in this project, we need a HTML structure, a DOM tree.

Code Example 3 A simple HTML structure.

```
1.  <div>
2.   <a href="...">
3.    <img src="..."></img>
4.   </a>
5.  </div>
```

In this DOM tree, when clicking the image element, the click event also fires on all its ancestors; the link element, the div element and all the way up to the window object, where it is terminated. This is the bubbling part of propagation.

The target is the image element since the click originated from that element. The target is saved in the event object.

3.2 Mechanisms

There are several different mechanisms that can be used when adding event listeners.

3.2.1 Zepto

Zepto uses the jQuery API syntax which means that adding an event listener using Zepto is the same code as adding an event listener using jQuery. It is flexible and has multiple ways of adding an event listener. Since there are many alternatives, there are several methods that have been deprecated and others that are considered bad practice. These methods also support adding event listeners to several elements using the same line of code but separating the elements using a comma.

Code Example 4. Zepto method used for adding an event listener.

```
1.  $( element ).on( event, childSelector, data, function, map )
```

Code Example 5. Zepto method used for removing an event listener.

```
1.  $( element ).off( event, selector, function, map )
```

3.2.2 Native JavaScript

There are native JavaScript methods that serve the same purpose as the jQuery methods above. These methods are part of a DOM element which means that every separate element which you want an event listener on, you need a new line of code. The element cannot be separated like in jQuery. The event listeners cannot be triggered manually either using code when using this method.

Code Example 6. Native JS method used for adding an event listener.

```
1.  element.addEventListener( event, function, [ options ] )
```

Code Example 7. Native JS method for removing an event listener.

```
1.  element.removeEventListener( event, function, [ options ] )
```

3.2.3 HTML event handler attributes

From back in the day when neither jQuery nor native JavaScript methods were available to attach event handlers to elements, inline event handlers were used. Today these are considered bad practice since they are inefficient in the long run. But they are easy to use, that is why we may still see them being used to this day.

Inline event handlers are attached using HTML attributes. The event handler is attached to an event-type attribute like this.

Code Example 8. HTML inline event handlers.

```
1.  <button onclick="doSomething()">Click me</button>
2.  <button onmouseover="doSomethingElse()">Hover me</button>
```

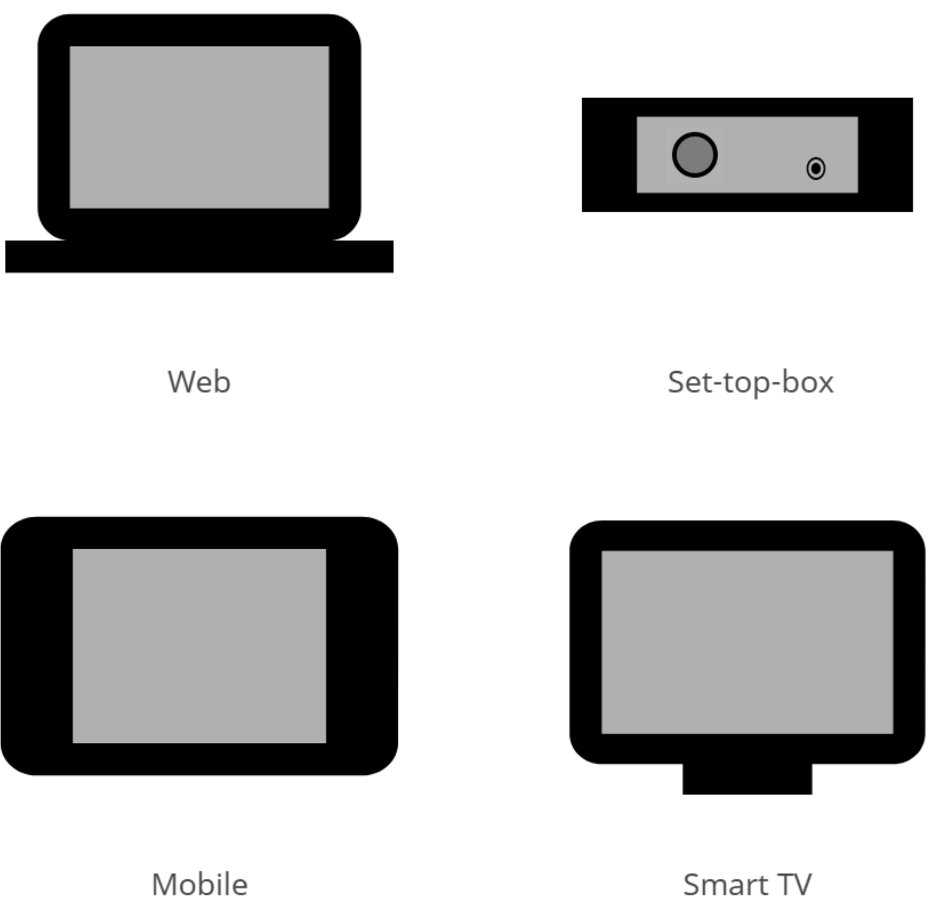
Easy to use but in the long run, if there are about tens or even hundreds of buttons, it would be a nightmare to maintain since they are scattered in the HTML structure.

4 Development

Developing support for touch enabled platforms is a broad area to fill on a large system with the main device-type being a TV. Firstly, code behind the functionality needs to be implemented. Secondly, animations are required to make the user experience fluent and understandable. These animations must be integrated into the system as a component since they must cooperate with touch gestures such as swipes and opening and closing of applications and other instances.

4.1 Devices

Figure 2. Supported device types.



These days there are four primarily used device types (see Figure 2); computers, set-top boxes, tablets, or smart TVs. These four device types have one thing in common, browser-based solutions that support managed web applications to be configured and run. The device types are very different in what they support and how they are used, that is why there is a way in the Hibox System to differentiate them, called device profiles.

4.1.1 Device profiles

Figure 3. System device profiles.

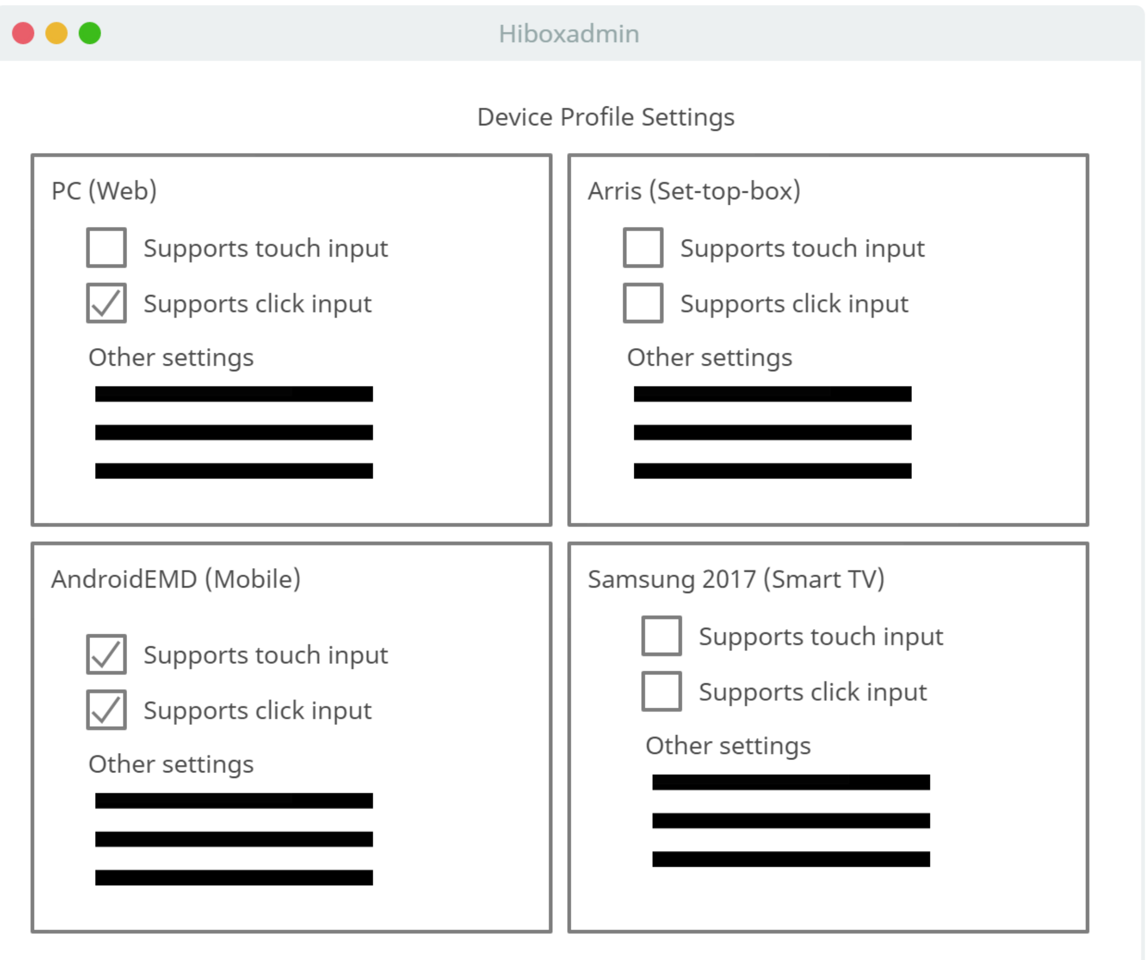
Hiboxadmin

List of device profiles

Amino STB	Arris STB
AndroidTV	AndroidEMD
LG	LG-legacy-tv
Samsung2013 (Orsay)	Samsung2017 (Tizen)
Philips2014	PhilipsSignageTouch

To easily manage and separate different device types in the system, each device is assigned to a device profile. Device profiles (see Figure 3) consist of shared device settings for all devices under that profile. For example, which video codecs and media transport methods the device supports and whether the device supports click or touch inputs (see Figure 4).

Figure 4. Touch & click support are device profile settings.



These settings in Figure 4 are set manually in the device profile for the intended device type depending on what specifications the device has. There can be several device models that use the same device profile and each one of those device models has specific features and different specifications of what they support. The device profile settings are a general set of settings that applies to the device models in the device profile. The device settings on the other hand are device-specific settings that apply to a specific device.

4.1.2 The mobile platform

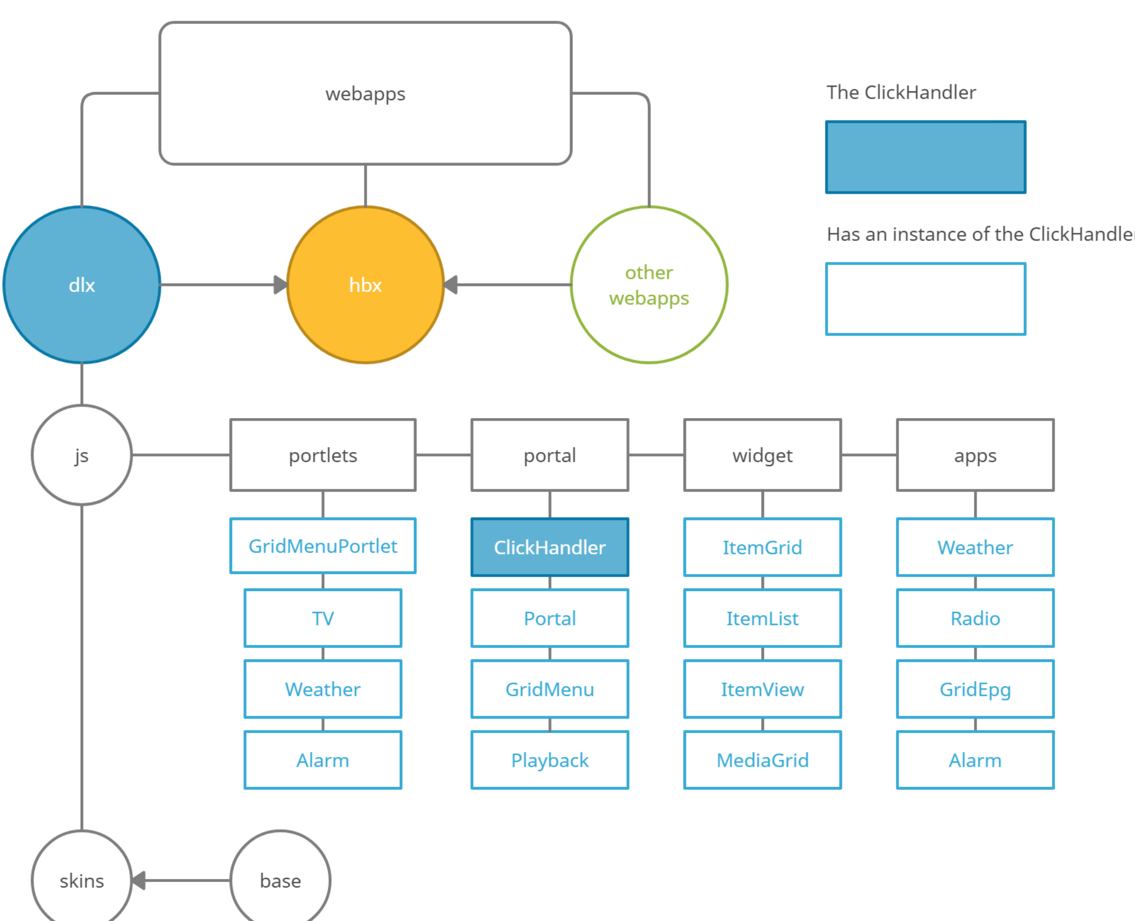
AndroidEMD stands for Android Enterprise Managed Device, which means that the device is under control by a management software. The management software can control every device that is assigned to the software, individually or in bulk. These devices are also locked and have configured ruleset applied which decides what you can and cannot do on the device. This is mandatory when there are hundreds of devices that need configurations and are used by people that do not own the devices in a hotel for example.

4.2 Implementation of touch support

4.2.1 The Frontend: dlx

The frontend web application that is running on all televisions, tablets and set-top-boxes is called dlx. The relevant components of dlx and how it is structured can be seen in Figure 5 below. It is a structured webapp circled around components. The main component is *Portal*. The portal creates the menu, the menu creates the portlets, the portlets are based on other components depending on the function of the portlet. This makes it easy to maintain since everything is split up. All components are also re-usable since they are independent components which only need to be created and used inside other components. Clicks and touches are needed system-wide and in Figure 5 below, you can see that the ClickHandler, the component that handles as the click and touch input, is created inside almost every component.

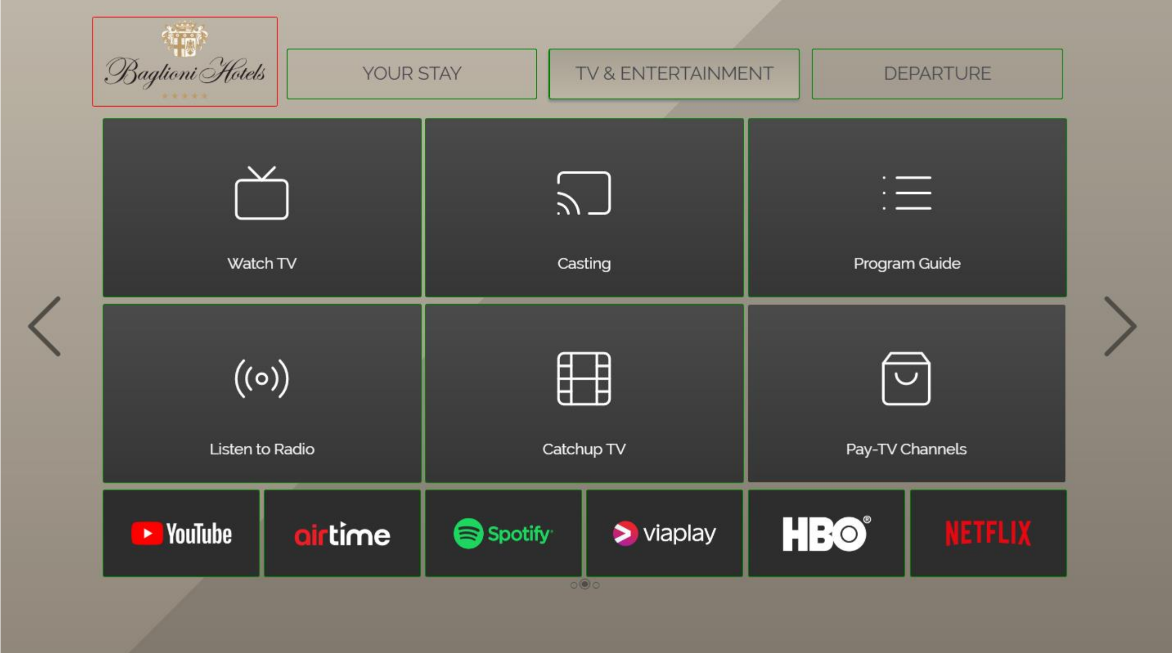
Figure 5. A diagram showing how the ClickHandler is created in almost every other component and presents a part of the structure of dlx and hbx.



Portlets are entities the end-user can interact with in a menu. They are the building blocks of the menu. They can also be entities that only show information to the end-user. In Figure 6 all individual items visible are portlets. The portlets marked with a green border are GridMenu portlets. They are part of a complex menu called GridMenu. The portlet marked with a red border is a logo portlet. It is the type of portlet that shows information to the end-user. The whole window is the portal and everything you can see in the figure is configured in hiboxadmin.

Portlets are often built by not using any components since they only serve the purpose of being an item in a menu displaying information. The portlets are part of an infopage in hiboxadmin and are configured on a node. The node can be configured to do something on execution (OK press on the remote or a click). For example, launch the radio application or go to TV mode. They can also be configured to launch native applications such as YouTube, Airtime, Spotify and so on as can be seen in Figure 6. These applications are not part of dlx. They are applications that are installed natively on the TV or the tablet as apps downloaded from an app store.

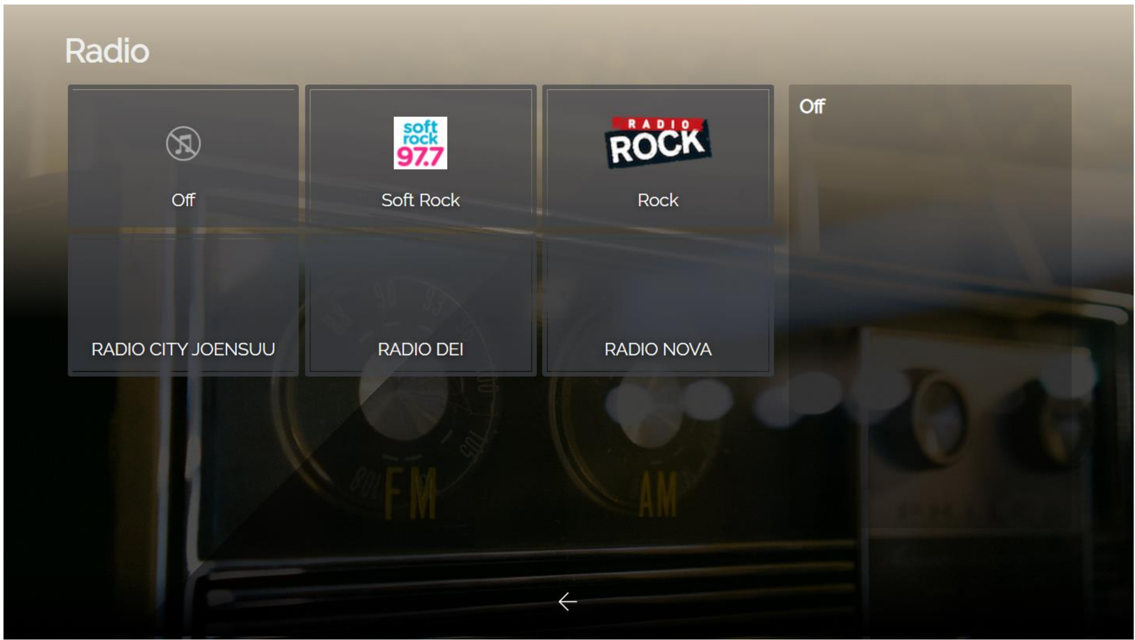
Figure 6. A screenshot of dlx marking menu portlets with a green border and other portlets with a red border.



Apps are applications that deliver a purpose or service, for example listening to radio, checking the program guide, browsing recordings, or buying pay-tv channels. They are also configured in hiboxadmin. Apps are opened by focusing and pressing OK on a portlet using a remote. When the implementation of touch support is completed, there will also be

support for opening applications by clicking the portlets. The apps are built using several other components, called widgets, for example a radio application displaying several radio stations as separate elements, a widget (a component) called *ItemGrid* is used to display the radio stations in a grid. The touch support can then be implemented in *ItemGrid* instead of in the radio app. This way, the *ItemGrid* and the touch support for the widget can be reused in another app.

Figure 7. The radio in dlx is an app.



4.2.2 Evaluation of extended touch libraries

Using a library that extends and simplifies the use of touch events such as swipes and long press is a good idea when developing touch support for a system that initially does not support touch events. There are lots of libraries that offer extended touch support for easy use, that is why a test of what seems to be the best candidates is needed to evaluate which one suits Hibox System's needs the best, both long-term and short-term. Documentation and user-friendliness of the library is also having a weight when deciding which one to use. Documentation is important as it minimizes the workload for the developer implementing and maintaining the code. User friendliness is partly associated with how good the documentation is, but also how the library fits in with the already existing system and how it feels using it. Those are the two main aspects that helps me to decide which library to choose.

4.2.3 Evaluation of zepto.touch

As stated before, the frontend, dlx, already uses a minimized jQuery library called Zepto. Zepto is divided into modules so that the user can choose which modules they want depending on their needs.

Figure 8. The list of available Zepto modules. **[8]**

module	default	description
zepto	✓	Core module; contains most methods
event	✓	Event handling via `on()` & `off()`
ajax	✓	XMLHttpRequest and JSONP functionality
form	✓	Serialize & submit web forms
ie	✓	Add support for Internet Explorer 10+ on desktop and Windows Phone 8.
detect		Provides `$.os` and `$.browser` information
fx		The `animate()` method
fx_methods		Animated `show`, `hide`, `toggle`, and `fade*()` methods.
assets		Experimental support for cleaning up iOS memory after removing image elements from the DOM.
data		A full-blown `data()` method, capable of storing arbitrary objects in memory.
deferred		Provides `$.Deferred` promises API. Depends on the "callbacks" module. When included, `$.ajax()` supports a promise interface for chaining callbacks.
callbacks		Provides `$.Callbacks` for use in "deferred" module.
selector		Experimental jQuery CSS extensions support for functionality such as `$('div:first')` and `el.is(':visible')`.
touch		Fires tap– and swipe–related events on touch devices. This works with both \`touch\` (iOS, Android) and \`pointer\` events (Windows Phone).
gesture		Fires pinch gesture events on touch devices
stack		Provides `andSelf` & `end()` chaining methods
ios3		String.prototype.trim and Array.prototype.reduce methods (if they are missing) for compatibility with iOS 3.x.

Zepto Touch is a module for extending the touch functionality with events such as swipes, taps, single taps, and long taps. I tested this library first since we already used Zepto. Node Package Manager (npm) is used to build a custom Zepto build.

Since dlx already is using a custom build with the modules *zepto, event, form, fx, fx_methods and selector* I only must add the module *touch* to the list of modules.

Code Example 9. Defining Zepto modules for a customized library.

```
1.  >$ MODULES="zepto event form fx fx_methods selector touch" npm run-script dist
```

This will build a new minified Zepto library with all the included modules. Next up is to include the newly customized Zepto library to dlx.

Code Example 10. Including the Zepto library into the dlx webapp.

```
1.  <script type="text/javascript" src="js/zepto.min.js"></script>
```

Code Example 11. Adding a swipe left event listener using Zepto.

```
1.  $( element ).on('swipeLeft', function (eventObject) {
2.        callback();
3.  });
```

It is simple to use for basic touch events. Adding an event listener which executes a callback upon a left swipe.

There is also the possibility to add options to these events such as how many fingers are needed to execute the event, the default threshold, excluded elements and specific thresholds for tap events.

Code Example 12. Options for Zepto Touch.

```
1.  $.fn.touch.defaults = {
2.      fingers: 1,
3.      threshold: 75,
4.      longTapThreshold: 500,
5.      doubleTapThreshold: 200,
6.      excludedElements: 'label, button, input, select, textarea, .noTouch',
7.      pageScroll: true,
8.      swipeMove: null
9.  };
```

These are all great and needed features, but zepto.touch has problems. It is lacking documentation and it is triggering the callback registered in the handler twice for every single event for some odd reason. A debug is necessary to fix this, but I am moving on to the next library as I do not think it is worth the effort to debug a library if I have other alternatives. [9]

4.2.4 Evalutation of TouchSwipe

This library is a jQuery plugin. Since the frontend uses zepto, it needs to be a zepto plugin. I found a zepto port of the plugin on GitHub and decided to try it out. It has all the features needed such as swipes, pinches, and single/double taps. It is used as a Zepto plugin which means that classes and methods of this plugin are added to the zepto instance. It is included into the project after zepto has been included (see Code Example 13).

Code Example 13. Adding zepto TouchSwipe to dlx.

```
1. <script type="text/javascript" src="js/zepto.min.js"></script>
2. <script type="text/javascript" src="js/zepto.touchSwipe.min.js"></script>
```

Code Example 14. Adding a swipe event listener using TouchSwipe *swipe* method.

```
1. $( element ).swipe( {
2.   swipe:function(event, direction, distance, duration, fingerCount, fingerData) {
3.      callback();
4.   },
5.   threshold: 0,
6.   fingers: 'all'
7. });
```

To add an event listener, TouchSwipe uses its own method, *.swipe()* (see Code Example 14). Options are defined directly in the argument object. The library is easy to use and has good documentation. The problem is that TouchSwipe stored its instance in a data annotation in the DOM element that the event was registered on. This does not work as intended in our software and TouchSwipe cannot fetch the instance from the DOM in the end. This causes the event handler to malfunction and for that reason it is not an alternative anymore. I am not sure whether this is due to the port of the plugin from jQuery to zepto or if it is a configuration issue. I decided to move on to Hammer.JS. [10]

4.2.5 Evaluation of Hammer.JS – The chosen one

Hammer.JS is described as an open-source multi-gesture library that adds support for touch-gestures to a web application. This library offers a lot of easy-to-use functionality in a structured way. It is not a plugin like the other two libraries are. It does not depend on any other library at all. It also offers a very good documentation including API documentation, examples, demos and a changelog.

By default, when creating a new instance of Hammer.JS, it adds a set of tap, double tap, press, horizontal pan, and swipe to be recognized with that instance. Pinch and rotate needs to be enabled manually. There is also a set of configurable options for every set of recognizers.

Figure 9. The list of available configurable options for the swipe recognizer.

Option	Default	Description
event	swipe	Name of the event.
pointers	1	Required pointers.
threshold	10	Minimal distance required before recognizing.
direction	DIRECTION_ALL	Direction of the panning.
velocity	0.3	Minimal velocity required before recognizing, unit is in px per ms.

What makes this library incredibly easy to use and understand is the structure. In the example below, the Hammer.JS instance is created for an element into a variable. The instance can then be configured to set the options in the desired way. In this case, a swipe is configured to only work in the vertical or horizontal direction depending on what the variable *direction* is set to. With this library, the event is registered on the Hammer.JS instance instead of directly on the HTML element. The HTML element is a part of the instance.

In Code Example 15, *event.preventDefault()* is called right after the callback was executed. This is to ensure that the browsers native handling of touch gestures is disabled, so that they do not interfere with the configured touch gesture.

Code Example 15. Adding a swipe event listener using the Hammer.JS library.

```
1.  var swipeHandler = new hbx.global.Hammer( element );
2.  swipeHandler.get( 'swipe' ).set( {  direction: Hammer.DIRECTION_VERTICAL } );
3.  swipeHandler.on( 'swipe' + direction, function( event ) {
4.      callback();
5.      event.preventDefault();
6.  } );
7.
8.  this.swipeHandlers.push( swipeHandler );
9.
```

In the ending section of Code Example 15, all the Hammer.JS instances are stored in an instance variable so that they can be removed when needed and thus the event listeners are removed. (see Code Example 16).

Code Example 16. Removing all Hammer.JS swipe event listeners by destroying its handlers.

```
1.  unregisterSwipeEvents : function( element ) {
2.    if ( this.isTouchEnabled() ) {
3.     this.swipeHandlers = _js.reject( this.swipeHandlers, function( handler ) {
4.     var has = handler.element === element;
5.       if ( has ) {
6.          handler.destroy();
7.       }
8.       return has;
9.     }, this );
10. }
```

In addition to preventing default browser actions on touch events, CSS also has a property of letting the browser know the touch-intent of the application called *touch-action*. This is used in Hammer.JS as default when creating a Hammer.JS instance. The CSS property is added to the element with a value depending on what gesture is configured.

Figure 10. A screenshot of the style which is directly set on an element in a Hammer.JS instance.

```
element.style {
   left: 0px;
   touch-action: pan-y;
   user-select: none;
   -webkit-user-drag: none;
   -webkit-tap-highlight-color: rgba(0, 0, 0, 0);
}
```

According to the table in Figure 11 below, the element above in Figure 10 has a horizontal swipe event registered since the *touch-action* has the value *pan-y*. This means that the Hammer.JS instance has been configured to only detect horizontal swipes.

Figure 11. A list presenting which value given in the touch-action CSS property that work with which gesture.

Gesture	Least restrictive touch-action value
press	auto
tap	auto
multitap	manipulation
vertical pan/swipe	pan-x
horizontal pan/swipe	pan-y
rotate	pan-x pan-y
pinch	pan-x pan-y

These *touch-action* properties tell the browser how to handle touch events and in case something goes wrong, its fallback is JS, for example to *event.preventDefault()*. It is

configured automatically when a Hammer instance is created. If a horizontal swipe is configured, *touch-action: pan-y* is set, if a vertical swipe is set, *touch-action: pan-x* is set.

The previously highlighted features in the library, plus that everything worked out-of-the-box without tweaking the code for compatibility with the Hibox Systems frontend, made me choose this library. It fulfilled the needs and secured the long-term need of extended touch-support.

4.2.6 The ClickHandler principles

The core behind this implementation is the ClickHandler. In Figure 5 you can see that the ClickHandler is created in several components. This is to serve one purpose; to register and handle click & touch events in that specific component. The ClickHandler itself is a 400-liner component that handles the actual events registered.

First things first, the portal which is the core of dlx instantiates the ClickHandler. Portal is the component that builds the initial DOM and instantiate all the required components that are necessary for dlx to work on a device.

Code Example 17. Creation of a new ClickHandler.

```
1.  this.clickHandler = new DLX.portal.ClickHandler( {
2.              isClickEnabled : DLX.env.deviceProfile.clickEventsSupported,
3.              isTouchEnabled : DLX.env.deviceProfile.touchEventsSupported,
4.              portal : this
5.          } );
```

When the ClickHandler is created in portal, it is already stated whether the device supports touch and click inputs or not. These device profile settings become instance variables on initialization that regulate which events that can be registered.

Code Example 18. ClickHandler methods regulating the use of click and touch input.

```
1.  _init : function( options ) {
2.      this.clickEnabled = _js.result( options, 'isClickEnabled', false );
3.      this.touchEnabled  = _js.result( options, 'isTouchEnabled', false );
4.  },
5.
6.  isEnabled : function() {
7.      return ( this.clickEnabled || this.touchEnabled );
8.  },
9.
10. isClickEnabled : function() {
11.     return this.clickEnabled;
12. },
13.
14. isTouchEnabled : function() {
15.     return this.touchEnabled;
16. },
```

The methods *isEnabled*, *isClickEnabled* and *isTouchEnabled* can be used when the ClickHandler's methods for registering events are called to regulate the event registering. If a device does not support click, it is unnecessary to register a click event at all.

Code Example 19. ClickHandler public methods for registering a click event on an element.

```
1.  registerClickEvent : function( elementOrFunction, callback, context ) {
2.      return this._registerEvent( 'click', elementOrFunction, callback, context );
3.  },
```

In Code Example 19 the public method to register a click event is presented. This is the method that other components use when registering a click event on an element.

Code Example 20. ClickHandler private method to register pointer events.

```
1.  _registerEvent: function( type, elementOrFunction, callback, context ) {
2.      if ( this.isEnabled() ) {
3.          var element = elementOrFunction;
4.          if ( _js.isFunction( element ) ) {
5.              element = element();
6.          }
7.  
8.          $( element ).on( type, function( event ) {
9.              callback.call( context, event );
10. 
11.             event.stopPropagation();
12.             event.stopImmediatePropagation();
13.             event.preventDefault();
14.         } );
15.     }
16. 
17.     return this;
18. },
```

In the private *_registerEvent* method in Code Example 20, *isEnabled* is called to check if either of touch or click is supported and if that is the case, register the event. When registering a click event, it does not mean that a click event is only supported on click-enabled platforms such as a browser on a PC. A click event is also executed when pressing a touch screen.

Freeing memory is also an important part when working with events. Events are stored in the memory of the client and if not handled correctly, they can easily cause a memory leak which can cause the app to be unresponsive or even crash. To prevent that, a method that unregisters specified events on an element is needed. This method will be used when the element is not needed anymore but is not removed from the DOM. If the element is destroyed and removed from the DOM, the events are unregistered natively by the browser.

Code Example 21. ClickHandler public method for unregistering click events.

```
1.  unregisterClickEvents : function( element ) {
2.       if ( this.isEnabled() && element ) {
3.           $( element ).off( 'click' );
4.       }
5.       return this;
6.   },
```

The method *unregisterClickEvents* uses zepto method *.off()* to remove all event listeners.

Every method in the ClickHandler returns the ClickHandler itself, namely *this.* It returns itself because *this* is according to Kyle Simpson in You Don't Know JS a "*binding that is made when a function is invoked, and what it references is determined entirely by the call-site where the function is called*" [11], which is the ClickHandler in this case. The object *this* is returned because of the comfort chaining methods.

Code Example 22. An example of how to chain ClickHandler methods.

```
1.  var that = this;
2.  this.portal.getClickHandler()
3.      .registerClickEvent( element, function() {
4.          // to do on click
5.      }, this )
6.      .registerMouseEnterEvent( element, function() {
7.          // to do on mouse enter
8.      }, this )
9.      .registerMouseLeaveEvent( element, function() {
10.         // to do on mouse leave
11.     }, this )
12.     .registerSwipeLeftEvent( element, function() {
13.         // to do on swipe left
14.     }, this );
```

Every method called in Code Example 22, returns the ClickHandler, which in return contains the next method. That is why it is chainable.

4.2.7 The ClickHandler – swipes

Ordinary swipes are registered using Hammer.JS. Ordinary swipes include right, left, up and down swipes. They are registered by calling a public method in the ClickHandler which in return calls a private method in the ClickHandler.

Code Example 23. ClickHandler public methods to register a swipe event in a specified direction.

```
1.  registerSwipeLeftEvent : function( elementOrFunction, callback, context ) {
2.      return this._registerSwipeEvent( 'left', elementOrFunction, callback, context );
3.  },
4.
5.  registerSwipeRightEvent : function( elementOrFunction, callback, context ) {
6.      return this._registerSwipeEvent( 'right', elementOrFunction, callback, context );
7.  },
8.
9.  registerSwipeUpEvent : function( elementOrFunction, callback, context ) {
10.     return this._registerSwipeEvent( 'up', elementOrFunction, callback, context );
11. },
12.
13. registerSwipeDownEvent : function( elementOrFunction, callback, context ) {
14.     return this._registerSwipeEvent( 'down', elementOrFunction, callback, context );
15. },
```

The private method contains the actual code executing the callback fired by the swipe.

Code Example 24. ClickHandler private method to register swipe events.

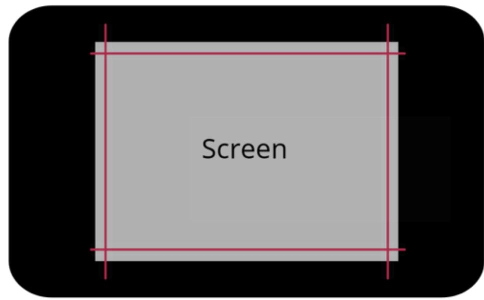
```
1.  _registerSwipeEvent: function( direction, elementOrFunction, callback ) {
2.     if ( this.isTouchEnabled() ) {
3.          var element = elementOrFunction;
4.          if ( _js.isFunction( element ) ) {
5.              element = element();
6.          }
7.
8.          var swipeHandler = new hbx.global.Hammer( element );
9.
10.    swipeHandler.get( 'swipe' ).set( direction === 'down' || direction === 'up' ?
11.    {  direction: Hammer.DIRECTION_VERTICAL, threshold: 75 } : { direction: Hammer.
    DIRECTION_HORIZONTAL, threshold: 75 } );
12.
13.       swipeHandler.on( 'swipe' + direction, function( event ) {
14.           callback();
15.           event.preventDefault();
16.       } );
17.
18.       this.swipeHandlers.push( swipeHandler );
19.    }
20.
21.     return this;
22. },
```

The parameters sent are the direction of the swipe, the element of a function returning an element, and a callback. Since a swipe is a touch-enabled platform feature only, the event is only registered if touch is supported on the device. By defining the direction in each public method, the private method can determine which direction the swipe is meant for to execute that specific callback.

4.2.8 The ClickHandler – edge swipes

Edge swipes are like ordinary swipes except that the starting point of the swipe is defined to an area of 50 pixels around the edges. This swipe is used for various purposes, for example when navigating back to the previous mode, for example from the TV mode to the menu mode.

Figure 12. The area of 50 pixels is between the edge of the screen and the red lines which is the area where the user can start an edge swipe.



The swipe mechanism works exactly in the same way as an ordinary swipe except that the starting position decides whether the callback should be executed or not.

Code Example 25. ClickHandler method returning the starting position of a swipe.

```
1.  _getStartPosition: function( event ) {
2.      var delta_x = event.deltaX;
3.      var delta_y = event.deltaY;
4.      var final_x = event.srcEvent.pageX || event.srcEvent.screenX || 0;
5.      var final_y = event.srcEvent.pageY || event.srcEvent.screenY || 0;
6.
7.      return {
8.          x: final_x - delta_x,
9.          y: final_y - delta_y
10.     };
11. },
```

The touch-end coordinate of a touch event is acquired and stored in the event object's *event.srcEvent.pageX* on the x-axis and *event.srcEvent.pageY* on the y-axis. Those data properties contain the data where the user let go of his finger from the screen when swiping. The method *_getStartPosition* determines in this case if the swipe event occurred on the edge-area between pixel 0 and 50 by subtracting the touch-end point coordinate with the distance the user swiped on the screen. What remains is where the user started the swipe.

Code Example 26. How the swipe start position is used.

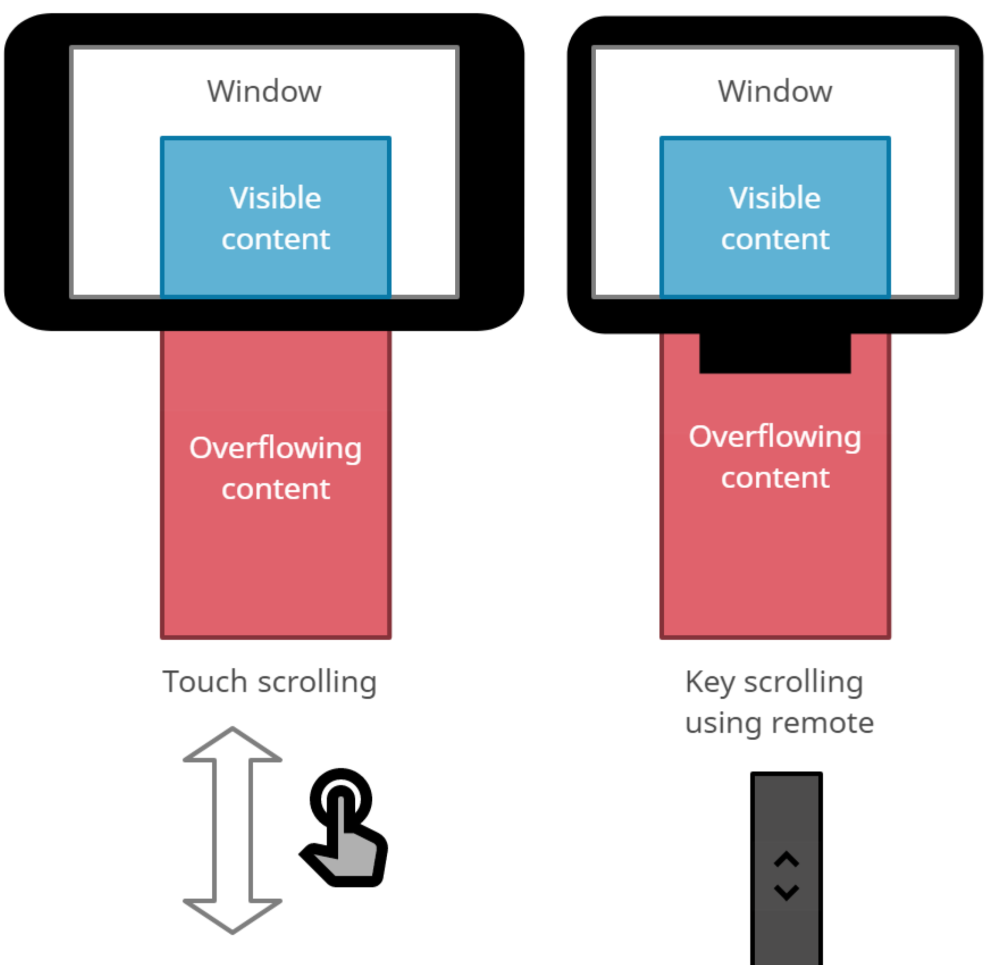
```
1.  var startingLimit = 0;
2.  var endingLimit = 50;
3.
4.  swipeHandler.on( 'swipe' + direction, function( event ) {
5.      var position = that._getStartPosition( event );
6.
7.      if ( position.x >= startingLimit && position.x <= endingLimit ) {
8.          callback();
9.          event.preventDefault();
10.     }
11. } );
```

The method *_getStartPosition* returns an object containing the starting position values *x* and *y* of the swipe and if the swipe started on the edge, the callback will be executed. If the value is over 50, the swipe started beyond the 50 pixel edge area and the callback is ignored. This only works for left and right edge swipe as the condition only checks the x-axis but could be improved by checking the y-axis as well for up and down swipes from the vertical edges.

4.2.9 The ClickHandler – touch scrolling

Scrolling content using the mouse wheel is something that everyone is used to. Scrolling content using a finger on a touch screen is also a prerequisite type of thing when interacting with a touch screen. Usually, content that hotels provide for their guests does not fit on the window area. The guests need the functionality to scroll the content, in most cases only vertically but in some cases also horizontally. This is not something that is enabled by default since the TV's scroll content by altering the *offsetTop* value by pressing the up and down buttons on a remote control. On a touch enabled platform, there is a smarter way to add scrolling functionality.

Figure 13. Scrolling using a remote on a TV is already supported while scrolling using touch input is not.



Adding support for scrolling using touch input is easy as it is natively supported by the browser. It is the same functionality as if you were scrolling a webpage on your phone.

Code Example 27. ClickHandler public methods that enables touch scrolling in a specific direction.

```
1.  enableHorizontalTouchScrolling : function( element ) {
2.      return this._enableTouchScrolling( 'horizontal', element );
3.  },
4.  enableVerticalTouchScrolling : function( element ) {
5.      return this._enableTouchScrolling( 'vertical', element );
6.  },
```

Code Example 28. ClickHandler private methods that enables touch scrolling.

```
1.  _enableTouchScrolling: function( direction, elementOrFunction ) {
2.     if ( this.isTouchEnabled() ) {
3.         var element = elementOrFunction;
4.         if ( _js.isFunction( element ) ) {
5.            element = element();
6.         }
7.
8.          switch ( direction ) {
9.              case 'horizontal' : {
10.                 $( element ).css( 'overflow-x', 'scroll' );
11.                 break;
12.             }
13.             case 'vertical' : {
14.                 $( element ).css( 'overflow-y', 'scroll' );
15.                 break;
16.             }
17.             case 'all' : {
18.                 $( element )
19.                     .css( 'overflow-y', 'scroll' )
20.                     .css( 'overflow-x', 'scroll' );
21.                 break;
22.             }
23.         }
24.
25.         $( element ).addClass( 'touch-scrolling-enabled' );
26.     }
27.
28.     return this;
29. },
```

The code above adds the CSS property *overflow-x / overflow-y* with the value *scroll* to the given element that allows the user to scroll overflowing content either on the vertical or horizontal axis. [12]

This property also adds a native scrollbar to the right side of the element. In a customized system a scrollbar is not needed nor wanted. The scrollbar can be removed by adding a *display: none* to the scrollbar's own webkit CSS selector. [13]

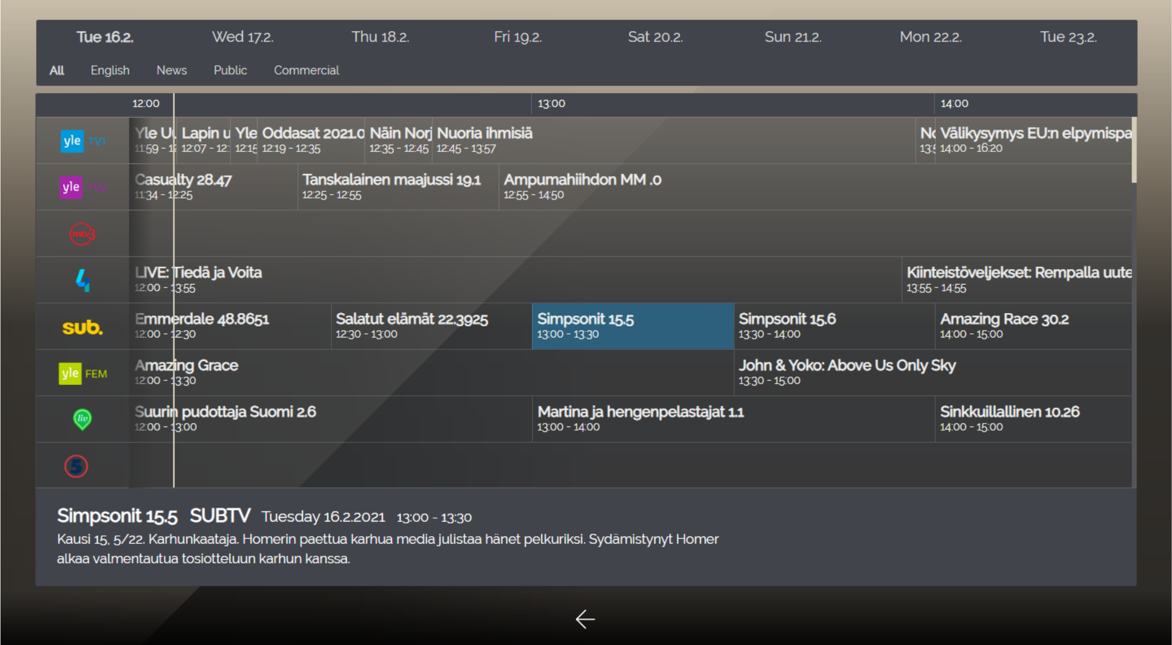
Code Example 29. How to hide the native scrollbar using CSS.

```
1.  ::-webkit-scrollbar {
2.      display: none;
3.  }
```

4.2.10 The ClickHandler – an application use case

The ClickHandler was intended to be easy to use and implement into other applications. One of the first applications that gets touch support implemented is the program guide, the *GridEpg*. The GridEpg is a 1600 lines of code application and is advanced in the way that it contains a lot of information about upcoming tv programs. A normal app is about 300-600 lines of code.

Figure 14. The GridEpg application.



The reason I chose to present this application was that it showcases the use of *event.target* and event propagation as I talked about earlier in chapter *3.1.3 Event propagation & target*. The GridEpg is a view of several channel rows that contains tv programs on that channel in that time. All these programs have unique identifiers which are used to identify the program when navigating and viewing info about that exact program. This is of course only when using a TV remote. Touch support cannot be implemented in the same way or integrated to use the same method as the TV remote does since a click is an entirely different navigation method. This is where *event.target* shines. But first, a presentation of how the *GridEpg* works and how the touch support is implemented from the start.

The core of touch support, the ClickHandler is already created in portal and the *GridEpg* fetches it from there. The app also creates variables to later regulate how the navigation works depending on which input method is enabled, click or touch.

Code Example 30. The ClickHandler is fetched from portal and variables are created to check if click or touch is enabled.

```
1. this.clickHandler = DLX.env.portal.getClickHandler();
2. this.touchEnabled = this.clickHandler.isTouchEnabled();
3. this.clickEnabled = this.clickHandler.isClickEnabled();
4. this.clickOrTouchEnabled = this.clickHandler.isEnabled();
```

Code Example 31. An example of how swipes to every direction is implemented into the GridEpg.

```
1. if ( this.clickOrTouchEnabled ) {
2.     this.clickHandler
3.         .registerSwipeLeftEvent( this.programList.elements.list, function() {
4.             // method to scroll the list to the right
5.         }, this )
6.         .registerSwipeRightEvent( this.programList.elements.list, function() {
7.             // method to scroll the list to the left
8.         }, this )
9.         .registerSwipeUpEvent( this.programList.elements.list, function() {
10.            // method to scroll the list down
11.        }, this )
12.        .registerSwipeDownEvent( this.programList.elements.list, function() {
13.            // method to scroll the list up
14.        }, this );
```

The interesting events in *GridEpg* is the click and mouseover event. These are the events that takes use of *event.target* to know which program is clicked and they both propagate. Each row has a unique identifier and each program in that row has a unique identifier. The program is identified on row basis. The row element is a parent of the program element. That is why it is important that propagation is supported by the events used here. Every time a click happens on a program element, it propagates upwards towards the window where it is terminated. That means that the event will propagate all the way past the channel row element to fetch the channel identifier.

Figure 15. The programs are identified per channel row basis with identifiers.

Figure 16. The HTML of a program element.

```
<div id="epglisting_row_4" class="epgprogrambg gridepglistingrow">
  ::before
  <div id="filler_4" class="epgfiller epgprogram epgprogramselected" style="left
   </div>
  <div id="program_30992-1613469600000" class="epgprogram 0" style="left:0px; wi
  <div id="program_30992-1613471400000" class="epgprogram 1" style="left:330px;
  <div id="program_30992-1613473200000" class="epgprogram 2 epgprogramselected"
    <span class="programtitle" style="left:0px">Simpsonit 15.5</span>
    <span class="programtime" style="left:0px">13:00 - 13:30</span>
  </div>
```

In Figure 16, the program HTML element class contains a number. That number is the identifier of that program on that channel row. The channel row identifier is found in the id of the row *epglisting_row_<identifier>* marked in Figure 16.

Code Example 32. How a click event is registered in GridEpg.

```
1.  .registerClickEvent( this.programList.elements.list, function( event ) {
2.      var $clickedProgram = $( event.target ).closest( '.epgprogram' );
3.      var $clickedChannel = $( event.target ).closest( '.channelname' );
4.
5.      if ( $clickedProgram.length ) {
6.          this._selectProgramByPointer( $clickedProgram );
7.      }
8.      else if ( $clickedChannel.length ) {
9.          this._selectChannelByPointer( $clickedChannel );
10.     }
11. }, this )
```

Often, handlers are registered on each element, in this case that would be the program element but that would not be efficient since there are lots of program elements. Instead, the handler is registered on the parent element (the program list) and each click origin is stored in the event object's property *event.target*. Then the zepto method *.closest()* is used to fetch the closest element to *event.target* with the class name *.epgprogram* for the program and *.channelname* for the channel row. Those elements contain unique identifiers in their class which are needed to determine which program or channel that is clicked. They are sent as parameters to the methods doing the actual selection and execution.

Code Example 33. The method selecting the program based on channel and program identifiers.

```
1.  _selectProgramByPointer: function( $program ) {
2.      var row = this._getRow( $program );
3.      var index = this._getIndex( $program );
4.      if ( row >= 0 && index >= 0 ) {
5.          if ( this.touchEnabled ) {
6.              if ( !this.programClicked ) {
7.                  this._selectProgram( row, index );
8.                  this.programClicked = true;
9.                  return;
10.             }
11.             else {
12.                 if ( this.selected.index === index && this.selected.row === row )
    {
13.                     this._pLexecuteSelected( undefined, true );
14.                     this.programClicked = false;
15.                 }
16.                 else {
17.                     this._selectProgram( row, index );
18.                     this.programClicked = true;
19.                     return;
20.                 }
21.                 return;
22.             }
23.         }
24.         else {
25.             this._pLexecuteSelected( undefined, true );
26.             this.clickOrTouchEnabled = true;
27.         }
28.     }
```

There are two different ways of selecting a program that depends on whether touch is enabled at all. If touch is enabled, a program needs to be clicked twice to see the full information about the program since the minified version of the information can be seen in the same view when selecting the program. If click is enabled and touch is not, the mouse hover event is also registered since the user uses a mouse. The mouseover event then handles the actual selection, and the click handles the execution to open the full information view. There is no hover when using only touch and that is why the touch needs to handle the selection and execution in two different presses as you can see in Code Example 33 above. The hover mechanism can be read about later in this chapter.

In Code Example 33 above, a program is selected by using the parameter *$program* sent containing the program element clicked. The channel row and program identifiers are fetched using the methods *_getIndex* and *_getRow*.

Code Example 34. The methods parsing the program element for channel and program identifiers.

```
1.  _getIndex: function( element ) {
2.      if ( element.hasClass( 'epgprogram' ) && !element.hasClass( 'epgfiller' ) ) {

3.          return parseInt( element.prop( 'class' ).replace( /[^0-9]/g,'' ) );
4.      }
5.  },
6.
7.  _getRow: function( element ) {
8.      if ( ( element.hasClass( 'epgprogram' ) || element.hasClass( 'epglistingrow' )
    ) && !element.hasClass( 'epgfiller' ) ) {
9.          return parseInt( element.parent().prop( 'id' ).replace( /[^0-
    9]/g,'' ) );
10.     }
11. },
```

The element is parsed for numbers which in this case is the identifiers. The program identifier can be found in the program elements class. The channel identifier can be found in the program elements parent id, the channel row listing. They are returned as integers and can be used to select the program if they are positive integers.

Code Example 35. The method selecting the program.

```
1.  _selectProgram: function( rowIndex, programIndex ) {
2.      this._hideSelection();
3.      this.programList.setSelection( rowIndex, true );
4.      this.selected = this._fetchProgram( programIndex );
5.      this._showSelection();
6.      this._showInfoButton();
7.  },
```

The program is selected by first selecting the correct channel row by using the channel row identifier (*rowIndex*). Then the program is fetched using a method by sending the program identifier to the method for comparison with a program list.

Code Example 36. The method fetching the program from an EPG data list.

```
1.  _fetchProgram: function( programIndex ) {
2.      var currentRow = this.programList.getSelectedIndex();
3.      var channel = this.channels[ currentRow ];
4.      if ( !channel ) {
5.          return null;
6.      }
7.      var programs = this.visiblePrograms[ channel.id ];
8.
9.      if ( programs && programs.length ) {
10.         if ( programIndex >= 0 ) {
11.             return found = { row: currentRow, index: programIndex, program: progra
    ms[ programIndex ] };
12.         }
13.     }
14. },
```

In Code Example 36 above, the program identifier passed as a parameter is used to find the correct program in an EPG data list. The returned program object is then stored inside

an instance variable called *this.selected* which is used by the method in Code Example 37. This method shows more information about the selected program**.** below to show info about that specific program.

Code Example 37. This method shows more information about the selected program.

```
1.  _pLexecuteSelected: function( key, click ) {
2.              var program = this.selected.program;
3.              if ( program && !program.isFiller && this.selected.type !== 'filler' )
    {
4.                  this._showList( false ); // hide the list of programs
5.                  this._showInfo( program );
6.              }
7.  }
```

The mouse hover mechanism selects the program when click is enabled but touch is not. It works almost the same as a click, but it only selects a program. It takes use of the *_selectProgram* method in the same way.

Code Example 38. How a mouseover event is registered in GridEpg for selecting a program.

```
1.  .registerMouseOverEvent( this.programList.elements.list, function( event ) {
2.      var row = this._getRow( $( event.target ).closest( '.epgprogram' ) );
3.      var index = this._getIndex( $( event.target ).closest( '.epgprogram' ) );
4.      if ( row >= 0 && index >= 0 ) {
5.         this._selectProgram( row, index );
6.      }
7.  }, this );
```

Hovering with the mouse over a program element in the program list fetches the channel and program identifier by bubbling upwards through the elements containing those identifiers.

This kind of solution for application touch support is not the ordinary. Often, the touch support is implemented in the application’s components used but, in this case, the *GridEpg* does not take use of components since it is a relic of past time.

4.3 Implementation of animations

Animations almost goes hand in hand with touch inputs. Static transitions when swiping is not satisfying nor informative to the user's eyes. The most important part is the informative aspect the animations present. For example, when an element is swiped up from the bottom of the screen for displaying a channel list for example, that element should be transitioned from the bottom to the top of the screen in a linear way at the same velocity as the swipe performed. This gives the feeling that it "follows" your interactions with the element. Same thing applies when the user wants to exit the channel list – it should transition from the top to the bottom at the same velocity as the swipe. If there were no transition nor animation, the element would pop up on the screen immediately upon swipe – this gives no information how to exit the channel list. Animations are essentially a cluster of visual information and cues for our minds of how to use a system.

4.3.1 The CSS

First off, since animating using JavaScript can often perform poorly, CSS will be used to animate the elements. CSS animations consists of two components; a style and a set of keyframes that tells what the start and end states are. The CSS animations can be defined in a separate CSS file and added to the elements that needs animations.

Code Example 39. A set of keyframes in CSS that defines the states of a specified animation.

```
1.  @keyframes slide-in-bottom {
2.      0% {
3.          opacity: 0;
4.          -webkit-transform: translateY(1000px);
5.          transform: translateY(1000px);
6.      }
7.
8.      100% {
9.          opacity: 1;
10.         -webkit-transform: translateY(0);
11.         transform: translateY(0);
12.     }
13. }
14.
15. @keyframes slide-out-bottom {
16.     0% {
17.         opacity: 1;
18.         -webkit-transform: translateY(0);
19.         transform: translateY(0);
20.     }
21.
22.     100% {
23.         opacity: 0;
24.         -webkit-transform: translateY(1000px);
25.         transform: translateY(1000px);
26.     }
27. }
```

These keyframes are the defined animations that can be added to an element using the CSS properties *animation-name* which is the name of the keyframe animation *and animation-duration* which is over what period the animation shall be executed.

Code Example 40. An example of how to add an animation to an element containing specified selector classes.

```
1.  .animations-enabled .app-messages-view.messages-view-animate.final,
2.  .animations-enabled .app-weather.weather-animate.final {
3.      animation-name: slide-out-bottom;
4.      animation-duration: .5s;
5.  }
```

4.3.2 The plan & why

To handle the animations on the JavaScript side of things, a component is needed. The component will be used in several other components that are supposed to be animated, for example in the on-screen-display, applications, and the menu. There are two reasons why this is needed.

First, because the system runs on several different device types. The animations are resource-heavy features that should only be used on devices that can withstand the drawing. Old television and STB:s does not have that capacity.

Second, how the system handles the visibility of elements. Since it is unnecessary to always remove and create new elements depending on the user's interaction, the elements are hidden when not in use using the CSS property *visibility*. That CSS property does not support animation frames since it does not re-paint the elements, an example of a CSS property that does re-paint the element is *display*. [14]

4.3.3 The component

This component handles the actual animations. The animations themselves are run by the browser and defined by CSS. It is based on a "two-step" animation concept; the first animation *initial* can always be run but the second animation *final* depends on the success of *initial*. For example, when opening an application, the *initial* animation is run upon the opening and when closing the application, the *final* animation is run. But upon creation of an element in a menu for example, the *initial* animation is run but the *final* animation is not needed in any way since the menu element are never "closed".

Code Example 41. Animation component method that is executed when the component is created.

```
1.  _init: function( container, animationClass ) {
2.      this.$container = $( container ) || {};
3.      this.animationClass = animationClass || 'unspecified-animation';
4.
5.      this.renderSpeed = DLX.env.device.getRenderSpeed() === 'fast';
6.      this.isClickOrTouchEnabled = DLX.env.portal.isClickOrTouchEnabled();
7.      this.animationsSupported = this.renderSpeed && this.isClickOrTouchEnabled;
8.
9.      if ( this.isAnimationsSupported() ) {
10.         this._checkAnimated();
11.     }
12.     else {
13.         this.disableAnimations();
14.     }
15. },
```

When initiating the component, we are checking the render speed since animations are resource heavy. Render speeds are defined manually per device platform, some are slow, and some are fast. Animations require the device to be fast. Animations are only run on click or touch supported devices also, since TV's does not need animations in the same way as a touch enabled platform does.

If these two conditions are met, we check if the element is animated with the *checkAnimated()* method.

Code Example 42. Animation component private method that checks if the specified element is animated or not.

```
1.  _checkAnimated: function() {
2.      this.enableAnimations();
3.      this._runAnimation( 'initial' );
4.      var cssProps = hbx.global.getComputedStyle( this.$container[0] );
5.  if ( cssProps.animationName !== 'none' && cssProps.animationDuration !== '0s' ) {

6.      this.isAnimated = true;
7.  }
8.      this._endAnimation( 'initial' );
9.  },
```

The purpose of this method is to check if there are any actual animations defined by CSS on the element. If there are, the instance variable *this.isAnimated* is set to true. The instance variable is checked later when running the actual animation. If it is true, the *initial* animation can proceed, if it is false, the animation is not run at all since there is no animation defined.

Code Example 43. Animation component public method that runs the *initial* animation.

```
1.  runInitial: function( reset ) {
2.      if ( this.$container.length && this.isAnimated ) {
3.          var that = this;
4.          hbx.global.requestAnimationFrame( function() {
5.              that._runAnimation( 'initial' );
6.              that._setVisibility( true );
7.          } );
8.      }
9.  },
```

If the container (the element) exists and is animated, *requestAnimationFrame* is called and passed a callback as parameter that runs the animation. The *requestAnimationFrame* method is part of the *window* object which in the Hibox System is stored in *hbx.global*. From Mozilla Web Docs [15]:

> *The* ***window.requestAnimationFrame()*** *method tells the browser that you wish to perform an animation and requests that the browser calls a specified function to update an animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint.*

Code Example 44. Animation component private method that runs the animation by adding a class to the element.

```
1.  _runAnimation: function( when ) {
2.      this.$container.addClass( this.animationClass ).addClass( when );
3.      this[ when ] = true;
4.  },
```

The callback *_runAnimation()* run by *requestAnimationFrame* adds a class to the element which means that the animation defined by CSS is run on that element.

Code Example 45. Animation component private method that ends the animation by removing a class from the element.

```
1.  _endAnimation: function( when ) {
2.      this.$container.removeClass( this.animationClass ).removeClass( when );
3.      this[ when ] = false;
4.  },
```

Its equivalent *_endAnimation()* removes the animation class and ends the animation.

After the *initial* animation has been run, the *final* animation can be run.

Code Example 46. Animation component public method that runs the *final* animation.

```
1.  runFinal: function() {
2.      if ( this.$container.length && this.isAnimated && this.initial ) {
3.          var that = this;
4.          this._registerAnimationEndEvent();
5.          hbx.global.requestAnimationFrame( function() {
6.              that._endAnimation( 'initial' );
7.              that._runAnimation( 'final' );
8.          } );
9.      }
10. },
```

When the *final* animation is run, the *initial* animation is ended, the class is removed, and the *final* animation class is added instead.

Since the element (for example an application) is visible and the element is going to be hidden and the *final* animation is going to be run, the code needs to know when to hide the element. To solve that, an event is registered to keep track of when the animation has ended, and when it has ended, the element is hidden, and the animation is ended. To free up memory, the event is also unregistered. If this event was not used, the element would hide the moment the user tells the application to close.

Code Example 47. Animation component private method that registers an animationend event handler.

```
1.  _registerAnimationEndEvent: function() {
2.      if ( this.$container ) {
3.          var that = this;
4.          this.$container
5.              .on( 'webkitAnimationEnd mozAnimationEnd MSAnimationEnd oanimationend
    animationend', function() {
6.                  if ( ( that.final && !that.initial ) ) {
7.                      that._endAnimation( 'final' );
8.                      that._setVisibility( false );
9.                      that._unregisterAnimationEndEvent();
10.                 }
11.             } );
12.     }
13. },
```

With a two-step animation concept, with a unique class added to the element depending on which animation it is, *initial* or *final*, different animations can be defined for those classes in CSS.

Code Example 48. An example of how to define different animations to the same element by using classes.

```
1.  .animations-enabled .app-messages-view.messages-view-animate.initial,
2.  .animations-enabled .app-weather.weather-animate.initial {
3.      animation-name: slide-in-bottom;
4.      animation-duration: .5s;
5.  }
6.
7.  .animations-enabled .app-messages-view.messages-view-animate.final,
8.  .animations-enabled .app-weather.weather-animate.final {
9.      animation-name: slide-out-bottom;
10.     animation-duration: .5s;
11. }
12.
```

5 Conclusion

Implementing touch support system-wide is not that hard in the sense of writing code. It is hard in the sense of having the same piece of software work at the same time on a TV and a tablet. Every piece of code written needs to be supported by both types. There is a lot of logic involved and a lot of different scenarios which were not presented in this thesis to test when developing. It has been helpful that the software has been built in a smart way from the beginning so that it is easy to implement new features and new components.

Finding the right way to implement animations was interesting. Since it is easy to just animate elements using only CSS, that would have been the way to go if there was only one platform in mind. But when there are several platforms, with several different performance and support aspects, an animations component was mandatory to get full control over the animations over several platforms.

The base of the touch support is finished. That base can be used when adding touch support to applications, menus, portlets, widgets, and to the portal itself. It has everything needed but can be extended with other gestures that may be needed in the future. As of now, there are lots of applications that lack touch support. These applications can now get touch support easily by using the ClickHandler component.

6 References

[1] "About | Hibox Systems," [Online]. Available: https://www.hibox.tv/company.shtml. [Accessed 31 12 2020].

[2] "HTML: MDN Web Docs," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. [Accessed 31 12 2021].

[3] "CSS: MDN Web Docs," [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/CSS. [Accessed 31 12 2020].

[4] "JavaScript: MDN Web Docs," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed 12 31 2020].

[5] "jQuery Introduction: w3schools.com," [Online]. Available: https://www.w3schools.com/jquery/jquery_intro.asp. [Accessed 31 12 2020].

[6] "ZeptoJS," [Online]. Available: https://zeptojs.com/. [Accessed 25 01 2021].

[7] "Getting started: Hammer.js," [Online]. Available: https://hammerjs.github.io/getting-started/. [Accessed 25 01 2021].

[8] "Modules: ZeptoJS," [Online]. Available: https://zeptojs.com/#modules. [Accessed 25 01 2021].

[9] "npm: zepto.touch," [Online]. Available: https://www.npmjs.com/package/zepto.touch. [Accessed 25 01 2021].

[10] "GitHub: TouchSwipe-Zepto-Plugin," [Online]. Available: https://github.com/huanz/TouchSwipe-Zepto-Plugin. [Accessed 25 01 2021].

[11] K. Simpson, "GitHub | You-Dont-Know-JS - Chapter 1," 27 08 2019. [Online]. Available: https://github.com/getify/You-Dont-Know-JS/tree/1st-ed. [Accessed 18 03 2021].

[12] "MDN Web Docs: overflow-y," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/overflow-y. [Accessed 02 02 2021].

[13] "MDN Web Docs: ::-webkit-scrollbar," [Online]. Available: https://developer.mozilla.org/en-us/docs/Web/CSS/::-webkit-scrollbar. [Accessed 02 02 2021].

[14] "Elliance: CSS Keyframe Animations: display vs. visibility," [Online]. Available: https://aha.elliance.com/2016/04/01/css-keyframe-animations-display-vs-visibility/. [Accessed 02 02 2021].

[15] “MDN Web Docs: Window.requestAnimationFrame(),” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed 02 02 2021].

[16] "Introduction to events: MDN Web Docs," [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events. [Accessed 31 12 2020].

[17] “jQuery API Documentation,” [Online]. Available: https://api.jquery.com/category/events/. [Accessed 31 12 2020].

7 Table of figures

Figure 1. Supported and unsupported navigation methods. 2
Figure 2. Supported device types. 9
Figure 3. System device profiles. 10
Figure 4. Touch & click support are device profile settings. 11
Figure 9. A diagram showing how the ClickHandler is created in almost every other component and presents a part of the structure of dlx and hbx. 12
Figure 10. A screenshot of dlx marking menu portlets with a green border and other portlets with a red border. 13
Figure 11. The radio in dlx is an app. 13
Figure 5. The list of available Zepto modules. [8] 15
Figure 6. The list of available configurable options for the swipe recognizer. 18
Figure 7. A screenshot of the style which is directly set on an element in a Hammer.JS instance. 19
Figure 8. A list presenting which value given in the touch-action CSS property that work with which gesture. 19
Figure 12. The area of 50 pixels is between the edge of the screen and the red lines which is the area where the user can start an edge swipe. 24
Figure 13. Scrolling using a remote on a TV is already supported while scrolling using touch input is not. 26
Figure 14. The GridEpg application. 28
Figure 15. The programs are identified per channel row basis with identifiers. 29
Figure 16. The HTML of a program element. 30

8 Table of Code Examples

Code Example 1. Defining an event handler on an element using Zepto. 6
Code Example 2. The event object parameter. 6
Code Example 3 A simple HTML structure. 7
Code Example 4. Zepto method used for adding an event listener. 7
Code Example 5. Zepto method used for removing an event listener. 7
Code Example 6. Native JS method used for adding an event listener. 8
Code Example 7. Native JS method for removing an event listener. 8
Code Example 8. HTML inline event handlers. 8
Code Example 9. Defining Zepto modules for a customized library. 15
Code Example 10. Including the Zepto library into the dlx webapp. 16
Code Example 11. Adding a swipe left event listener using Zepto. 16
Code Example 12. Options for Zepto Touch. 16
Code Example 13. Adding zepto TouchSwipe to dlx. 17
Code Example 14. Adding a swipe event listener using TouchSwipe *swipe* method. 17
Code Example 15. Adding a swipe event listener using the Hammer.JS library. 18
Code Example 16. Removing all Hammer.JS swipe event listeners by destroying its handlers. 19
Code Example 17. Creation of a new ClickHandler. 20
Code Example 18. ClickHandler methods regulating the use of click and touch input. 20
Code Example 19. ClickHandler public methods for registering a click event on an element. 21
Code Example 20. ClickHandler private method to register pointer events. 21
Code Example 21. ClickHandler public method for unregistering click events. 22
Code Example 22. An example of how to chain ClickHandler methods. 22
Code Example 23. ClickHandler public methods to register a swipe event in a specified direction. 23
Code Example 24. ClickHandler private method to register swipe events. 23
Code Example 25. ClickHandler method returning the starting position of a swipe. 24
Code Example 26. How the swipe start position is used. 25
Code Example 27. ClickHandler public methods that enables touch scrolling in a specific direction. 27
Code Example 28. ClickHandler private methods that enables touch scrolling. 27
Code Example 29. How to hide the native scrollbar using CSS. 27

Code Example 30. The ClickHandler is fetched from portal and variables are created to check if click or touch is enabled. 29
Code Example 31. An example of how swipes to every direction is implemented into the GridEpg. 29
Code Example 32. How a click event is registered in GridEpg. 30
Code Example 33. The method selecting the program based on channel and program identifiers. 31
Code Example 34. The methods parsing the program element for channel and program identifiers. 32
Code Example 35. The method selecting the program. 32
Code Example 36. The method fetching the program from an EPG data list. 32
Code Example 37. This method shows more information about the selected program. 33
Code Example 38. How a mouseover event is registered in GridEpg for selecting a program. 33
Code Example 39. A set of keyframes in CSS that defines the states of a specified animation. 34
Code Example 40. An example of how to add an animation to an element containing specified selector classes. 35
Code Example 41. Animation component method that is executed when the component is created. 36
Code Example 42. Animation component private method that checks if the specified element is animated or not. 36
Code Example 43. Animation component public method that runs the *initial* animation. . 37
Code Example 44. Animation component private method that runs the animation by adding a class to the element. 37
Code Example 45. Animation component private method that ends the animation by removing a class from the element. 37
Code Example 46. Animation component public method that runs the *final* animation..... 38
Code Example 47. Animation component private method that registers an animationend event handler. 38
Code Example 48. An example of how to define different animations to the same element by using classes. 39

Appendix 1

```
1.  MouseEvent {
2.      altKey: false
3.      bubbles: true
4.      button: 0
5.      buttons: 0
6.      cancelBubble: false
7.      cancelable: true
8.      clientX: 799
9.      clientY: 631
10.     currentTarget: div#app
11.     data: undefined
12.     defaultPrevented: false
13.     detail: 1
14.     eventPhase: 3
15.     fromElement: null
16.     isDefaultPrevented: ƒ ()
17.     isImmediatePropagationStopped: ƒ ()
18.     isPropagationStopped: ƒ ()
19.     isTrusted: true
20.     layerX: 799
21.     layerY: 631
22.     movementX: 0
23.     movementY: 0
24.     offsetX: 800
25.     offsetY: 632
26.     pageX: 799
27.     pageY: 631
28.     path: (4) [div#app, html, document, Window]
29.     preventDefault: ƒ ()
30.     relatedTarget: null
31.     returnValue: true
32.     screenX: 2453
33.     screenY: 524
34.     shiftKey: false
35.     stopImmediatePropagation: ƒ ()
36.     stopPropagation: ƒ ()
37.     target: div#app
38.     timeStamp: 168305.60499999957
39.     type: "click"
40.     view: Window {0: Window, 1: Window, window: Window, self: Window, document: docum
    ent, name: "", location: Location, …}
41.     which: 1
42.     x: 799
43.     y: 631
```