Interactive Web Video – Modern User Experience for Viewers and Editors

Paul Lonauer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2018

Declaration

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

Hagenberg, June 26, 2018

Paul Lonauer

Contents

Abstract

Developments in the modern web environment have facilitated the delivery of interactive web video. However, the field of interactive video has seen less efforts to improve existing systems in recent times. Moreover, the majority of the improvements were focused on the presentation side of the content.

With little effort being put into improving the video editor experience, they often lack the appropriate, user-friendly tools to add interactivity to their web videos. With better tools to work with, editors will be more inclined to produce new interactive video content, drawing more attention to the topic in general.

This thesis aims to describe the technical foundations of interactive video in general, before delving into the state of the art of interactive web video. Alongside this thesis, an interactive web video system was implemented, which combines positive influences of existing web video players and interactive video authoring tools. On one hand, this system provides editors with a modern way of authoring interactive web videos. On the other hand, the system also contains a state-of-the-art video player, capable of presenting the interactive content to viewers.

The developed system represents a basis for further research and might also draw some additional attention to the field of interactive web video. User experience tests have been conducted to confirm the correct implementation of the video player and the video authoring system. The tests show that the proposed system of using JSON to define interactible elements works for editors without programming experience, but is not yet the ideal solution, since the JSON format can be too pedantic at times. Concerning viewer experience, the tests show that the implemented video player functionality, including the interactive features, are well understood and utilized.

Kurzfassung

Die Entwicklungen im Bereich des modernen Webs haben die Verbreitung von interaktiven Videos erleichtert, jedoch hat das Feld der interaktiven Videos in der letzten Zeit nur wenige Neuerungen erfahren. Außerdem waren ein Großteil der Verbesserungen auf die Darstellung der Inhalte fokussiert.

Da nur wenig Aufwand betrieben wird, die Benutzerfreundlichkeit für Redakteure zu verbessern, fehlt es ihnen oft an den passenden, benutzerfreundlichen Werkzeugen um Interaktivität in ihre Videos zu bringen. Mit verbesserten Werkzeugen werden Redaktuere eher dazu geneigt sein, mehr interaktive Inhalte zu produzieren, was wiederum mehr Aufmerksamkeit auf das Thema lenkt.

Diese Arbeit zielt darauf ab, die technischen Grundlagen von interaktiven Videos im Allgemeinen zu erklären, bevor sie näher auf den Stand der Technik im Bezug auf interaktive Web-Videos eingeht. Im Rahmen der Arbeit wurde auch ein interaktives Web-Video System entwickelt, das die positiven Aspekte von bereits bestehenden Web-Video Playern und interaktiven Video-Autorensystemen kombiniert. Einerseits bietet das System den Redakteuren einen modernen Weg Videos mit Interaktivität zu versehen. Andererseits beinhaltet das System auch einen Video Player, der dem Stand der Technik entspricht und dem Zuschauer auch die interaktiven Inhalte präsentieren kann.

Das entwickelte System stellt eine Basis für weitere Forschung dar und lenkt potentiell auch zusätzliche Aufmerksamkeit auf das Gebiet der interaktiven Videos. Die Benutzerfreundlichkeitsstudien wurden durchgeführt, um die korrekte Implementierung des Video Players und des Autorensystems zu gewährleisten. Diese Studien haben gezeigt, dass das vorgeschlagene JSON System für die Definition von interaktiven Elementen zwar selbst für Redakteure ohne Programmiererfahrung funktioniert, jedoch noch keine ideale Lösung darstellt, da das JSON Format teilweise zu pedantisch ist. Bezüglich der Benutzerfreundlichkeit für Zuschauer wurde festgestellt, dass die implementierte Video Player Funktionalität, sowie die interaktiven Features, gut verstanden und angenommen werden.

Chapter 1

Introduction

Web video content is a rapidly evolving field in the modern web environment. With *YouTube*, *Netflix*, *Twitch* and many others being on the forefront, video content and its display is constantly being refined to provide users with the best experience possible. In order to stay at the top, companies will want to experiment new and interesting features to attract more viewers. One such feature would be the implementation of interactive video concepts. *YouTube*, for instance, has tried this in the past with their annotation system, but the system was discontinued on May 2, 2017, and replaced by the new, but more restrictive *YouTube* end screen and info cards [30].

Interactive video has been a topic of interest since the very early days of video content. The first cinematic adaptation of the interactive video concept was presented in 1967 and featured a voting system which the audience could use to influence the trajectory of the film at key moments [11]. Even before the era of video production, interactive stories were a matter of widespread interest. So-called 'gamebooks' can be dated back to 1930, when the concept of influencing one's own reading experience through choices birthed an entirely new genre of books [21]. Nowadays, interactive videos are mostly used for marketing purposes.

While developments in the modern web environment have facilitated the delivery of interactive web video, there are still a few issues to be solved until both the viewer and the editor can have a satisfying experience. While a lot of time has been invested into making the potential interaction methods useful and understandable, many interactive video solutions have foregone concerns about the user experience for video editors. The biggest issue here is, that the developers of those solutions expect their users to be developers as well, which is often not the case.

1.1 Overview

Chapter 2 aims to provide enough insight into the field of interactive video to comprehend the information contained in the following chapters. For this reason, commonly used terms are defined for the scope of this thesis. The Chapter continues with a glimpse into the history of interactive video, to better understand how the developments in the field came to be. Lastly, conventional technology behind interactive video and its predecessors is explained.

Chapter 3 is going to discuss the current state of interactive video technology. It will elaborate on some of the concepts presented in the previous chapter, their developments and modern day implementations. The Chapter will continue by discussing similar software to the proposed video player, since the technology behind those implementations and the design decisions they incorporate are of interest. The Chapter will conclude with a number of fields of application of interactive video technology.

Chapter 4 is intended to display and explain the video player, which was implemented with the knowledge from the previous chapters, from a technical design point of view. To start off, the requirements of the video player are discussed in order to provide a better understanding for some of the decisions explained later on. To introduce the architecture of the video player from a logical point of view, the most relevant design patterns for the conception of the player are presented afterwards. Finally, the important design decisions that make up the core architecture of the piece of software are discussed and potential alternatives are mentioned.

Chapter 5 is meant to explain the concrete, inner workings of the video player. The first Section is going to focus on a the core component of the piece of software and explains how different requirements are implemented within it. Later, the discussion of the actual implementation of the player progresses through the source code one component at a time, while introducing important auxiliary modules afterwards. Many times during the chapter, relevant parts of the implementation are illustrated using code snippets, which show the corresponding JavaScript code.

Chapter 6 focuses on the usability tests, which were conducted in order to ensure that the primary goal of this thesis has been met. This goal was the implementation of an interactive video player alongside corresponding authoring mechanisms in such a way that the user experience of both viewers and editors is increased compared to the current state of the art. The first two sections are aimed at describing the group of participants, the testing procedure, and the results of the editor and the viewer usability test respectively. The third Section displays the opportunities for further improvement for the video players, which have been deduced from the results of the usability tests.

Chapter 2

Technical Foundations

This Chapter aims to provide enough insight into the field of interactive video to comprehend the information contained in the following chapters. For this reason, commonly used terms are defined for the scope of this thesis. The Chapter continues with a glimpse into the history of interactive video, to better understand how the developments in the field came to be. Lastly, conventional technology behind interactive video and its predecessors is explained.

2.1 Definition of Terms

2.1.1 Interactive Video

According to the definition of the *Cambridge Online Dictionary*, *interaction* describes 'an occasion when two or more people or things communicate with or react to each other' [26]. While this definition held up fine in the early days of interactive video, where videos were shown in a movie theater and the only means of viewer interaction was actually part of an interaction concept, the definition is not quite fitting for modern, online video content. This is due to the fact, that when following the *Cambridge* definition, almost all web video content is interactive simply because of its pause functionality. While it is a very basic reaction of the video player, it is still enough to qualify it as interactive. Thus, the term interactive video has to be defined more precisely in order to be of any use in the scope of this paper. A better definition includes the separation of conventional interaction elements such as play/pause buttons, as well as a progress bar with seeking functionality. Common means of interaction that qualify video content for the interactive tag include:

- clickable subtitles,
- clickable video overlays,
- means of changing the video view (e.g. a button to toggle CGI[1] effects within a video),
- processing of camera, microphone or similar input data with effects on the video.

[1]Computer-generated imagery

2.1.2 Viewer

The viewer of a video is, simply put, the consumer of the video content. This definition fits the context of interactive video as well. Additionally, the viewer is also the person interacting with the video player and he is the main target of all interaction concepts. While it is naturally possible for multiple individuals to consume a video at the same time, interactive videos are not commonly suited to be interacted with by several people at once anymore. In the past the input of multiple viewers was accumulated to determine a single course of interaction for the video, effectively making the video still controlled by one input.

2.1.3 Editor

The editor of a video is usually the individual making the creative decisions about the arrangements of certain shots to create a complete video. While this might fit interactive videos as well, this thesis requires a special definition of editors. Since the video player implementation is specifically targeted at the *ORF*, the Austrian broadcasting corporation, the term describes people not actually involved in the technical part of the video production. *ORF* editors investigate issues, interview people, write stories and finally, add means of interaction to videos. Due to this circumstance, editors are going to have little technical knowledge, meaning that they need an easy way to define those means of interaction.

2.2 History of Interactive Video

2.2.1 Gamebooks

Gamebooks are books which allow the reader to influence the progression of the plot via his own decisions. After reading through a section of the book, the reader is usually presented with a choice of narrative branches. Each choice contains a page number the reader has to turn to if he chooses that path [20]. While not technically falling into the specification of video content, gamebooks have most certainly influenced the way interactive video has developed. After all, most movies are cinematic adaptions of books, so the step from interactive books to interactive movies was an inevitable one. gamebooks can be seen as early as 1930, when Doris Webster's and Mary Alden Hopkins's book *Consider the Consequences* [21] gave birth to the genre. Another relevant addition to the genre are the *Choose Your Own Adventure* books, where the reader takes up the role of the protagonist in various fantasy scenarios.

2.2.2 Interactive Cinema

Just like gamebooks, interactive movies feature a branching story line which the viewer is given control over at key moments. In the case of a showing within a cinema, interactive movies provide viewers with voting devices connected to an in-house system. Using those devices, the audience would cast their vote to determine the path of the story that is going to appeal to the majority of them. The interactive movie experience *Kinoautomat* [11] by Radúz Činčera, was first shown in 1967 and is commonly considered

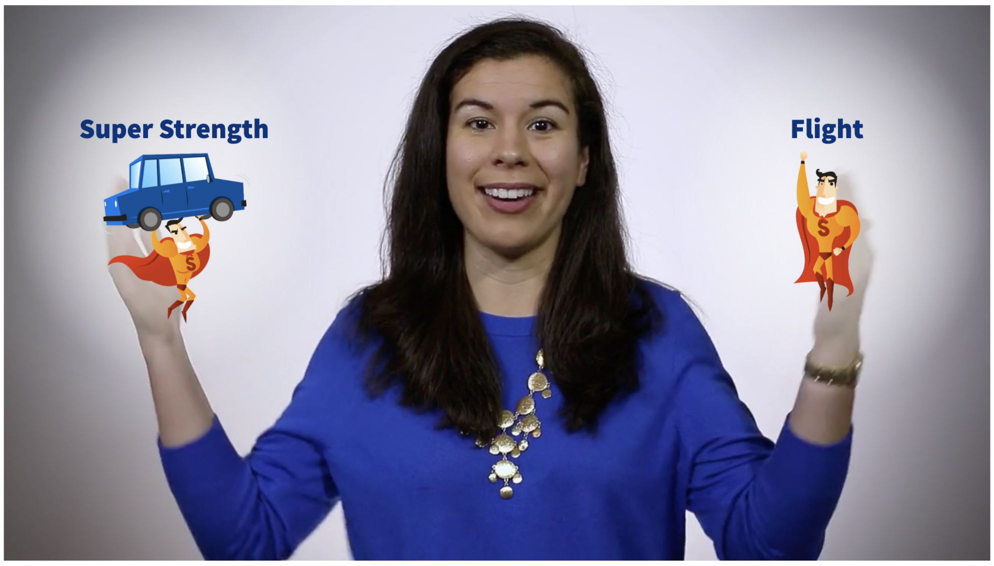


Figure 2.1: A sample survey question about super powers found on *SnapApp* [27].

the first cinematic adaptation of the interactive video concept. Note that voting system of *Kinoautomat* was not yet automated, but instead operated by a manager appearing on stage to ask the viewer about their choice.

2.2.3 Interactive Web Video

Nowadays, interactive videos are mostly used for marketing purposes in the web. Online marketing in the form of videos can take up varying forms and might not directly promote a single product, as long as they create awareness of the brand. Those varying forms include:

- videos with branching story lines,
- videos with included surveys (see Figure 2.1),
- interactive video guides.

2.2.4 Video Games

In a broader sense, video games can also be considered a form of interactive video. Although this topic quite exceeds the scope of this thesis, there are a few games that float elegantly between being a movie and being a game. For example, many of *Telltale's* games [29], as well as *Square Enix's Life is Strange* [28] are renowned for letting their players make major decision within the plot. In addition to that, there are many sections the players get to explore with complete control over the game characters. Still, some among the gaming community do not consider those 'experiences' real games, since they lack meaningful gameplay features apart from navigating the free play segments and reacting to *Quick Time Events*[2]. Nevertheless, these games are relevant as a way of bridging the gap between traditional games and non-interactive video content.

2.3 Technology behind Interactive Video

2.3.1 Gamebooks

While the technology behind books is not exactly noteworthy, gamebooks have made clever use of printing conventions, namely page numbers, to create the immersive, interactive experiences they are famous for [20]. A gamebook traditionally lets the user decide how the story is going to unfold on many occasions throughout the book. For example, a character from the book might have an encounter with a black bear. The bottom of the page might then read:

- To play dead, turn to page 18.
- To fight the bear, turn to page 27.
- To run from the bear, turn to page 36.

Naturally, having the story scattered around within the book has downsides. The book can no longer be read comprehensively from the front to the back, so potential readers have to be informed about the process before they start reading. From a programmer's or mathematician's point of view gamebooks are also interesting, because their non-linear paths can be displayed as directed graphs[3] [24].

2.3.2 Interactive Cinema

Before the invention of relevant technology for creating interactive video experiences, the interactivity had to be provided manually. Moderators had to appear on stage to poll the audience for their choice of in-movie actions and projector operators had to swap out the movie reels accordingly. The invention of *LaserDisc* [5, 6] revolutionized this workflow significantly. *LaserDisc* is a home video format and the first commercial optical disc storage medium, but in contrast to movie reels or VHS tapes, *LaserDiscs* allow for random access of the video material, while the others would require tedious rewinding and fast-forwarding to get to specific points. This meant, that chapters and other video segments could be arranged in a way similar to gamebook pages. *LaserDisc* players connected to a joystick could then use its input to react to the viewers actions. Despite being very relevant for the progression of interactive video, *LaserDiscs* have not been able to gain a major foothold within the home entertainment industry. They were quickly replaced by DVDs and DVDi[4] technology. This step removed the need for additional equipment like joysticks and further promoted the interactive genre through the inclusion of DVD playing function in gaming consoles such as the *Playstation 2*.

[2] *Quick Time Events* are sections of video sequences which demand quick reactions from the player. Depending on the player's success, the sequence can go different ways.

[3] A set of nodes or vertices, connected by lines.

[4] DVD interactive

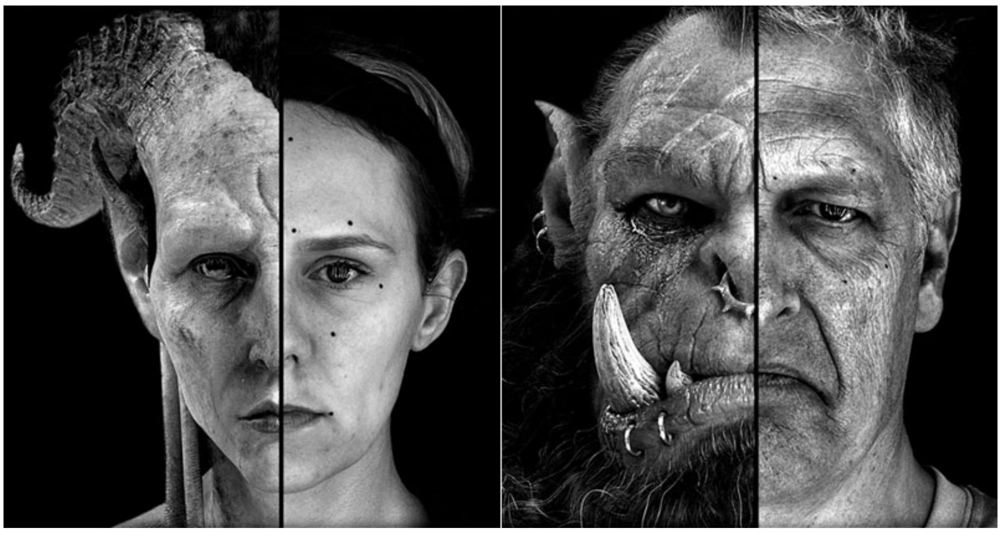


Figure 2.2: Split-face portraits of *Warcraft* [25] actors and their CGI counterparts.

2.3.3 How Video Games Differ from Videos

From an outside point of view, an in-game cutscene might not differ much from a movie scene, but from a more informed perspective, those two boast a fundamental difference. A modern day video game cutscene is created using the game engine to let the 3D models of the game's characters perform preset actions within the game's environment. The scenes are rendered dynamically on the player's machine. In contrast to that, movie scenes are pre-rendered [3]. The usage of 3D models for movie scenes depends on the its type, since animated or CGI using movies are utilizing 3D models and/or environments as well.

A good example for the effects of CGI can be seen in Figure 2.2. These different styles of video production have developed alongside each other to cover the needs of two varying markets. The game approach provides adaptability, seamlessness, and low memory consumption. Game cutscenes can easily be adjusted to minor changes within the environment, don't differ much from the imagery the player sees during gameplay and do not use much space, since the cutscene assets are recycled. On the other hand, the movie approach provides higher quality, low processing cost, and stability. Since movies are not required to be rendered in real time, the can opt for a much higher quality than game cutscenes. At the same time, the require almost no processing power on the viewer side, since the video player only needs to read and display the video data. Furthermore, movies can be expected to look the same on every device. Video data can only be interpreted in one way, whereas game engines are more error-prone. In conclusion, it can be seen that games lean themselves more towards interactivity, while movies prefer accessibility for a broader audience. Since the field of interactive video is a combination of the two realms, it has to be able to successfully merge their interests together to produce satisfactory results.

Chapter 3

State of the Art

This Chapter is going to discuss the current state of interactive video technology. It will elaborate on some of the concepts presented in the previous chapter, their developments and modern day implementations. The Chapter will continue by discussing similar software to the proposed video player, since the technology behind those implementations and the design decisions they incorporate are of interest. The Chapter will conclude with a number of fields of application of interactive video technology.

3.1 Current State of Interactive Video

Gamebooks, as they have been discussed in the previous chapter, still exist and have a decent customer base. Incidentally, the concept has been combined with the video gaming sector to produce the Japan-influenced genre of visual novels [1]. The dialogue text in visual novels is sometimes voice acted and all scenes are supported by background and character visuals, although the characters are not usually animated as one would expect from a video game.

Moving on to interactive cinema, it can be observed that no attempts to implement the concept have reached any large scale popularity. This is likely due to a combination of the lack of individual decision making and the added financial burden that would be put onto cinema owners.

Interactive web video is the field that has garnered the most interest over the years, since interactive video today is mostly used for marketing purposes within the web environment. Web technology offers a multitude of possible implementations for interactive video players, ranging from more traditional *Flash*[1]-based approaches to more modern HTML5/JavaScript alternatives. These approaches are further supported by cutting-edge technology developments in the field of modern web, such as the *React* JavaScript library. Many organizations have implemented a form of interactive web video players and authoring tools, but no official standard has been established yet.

[1]Adobe Flash Player, a (web) video player released in 1996.

3.2 Similar Software

3.2.1 H5P Interactive Video

H5P is an abbreviation of *HTML5 Package* and is an open-source content collaboration framework. The developers of *H5P* aim to enable anyone to create and share interactive media using HTML5 [16]. While the *HTML5 Package* contains many interesting features, the interactive video part of it is the most relevant here. Its website states that:

> Videos may be enriched with interactivities like explanations, extra pictures, tables, Fill in the Blank and multiple choice questions. Quiz questions support adaptivity, meaning that you can jump to another part of the video based on the user's input. Interactive summaries can be added at the end of the video. Interactive videos are created and edited using the H5P authoring tool in a standard web browser.

Technology

H5P is a framework based on PHP, which means that it requires PHP to be installed on the server that hosts the website. The package is available as a plugin for popular CMSs[2] such as *Drupal*, *WordPress*, and *Moodle*, so the editors working with *H5P* only need to interact with the framework via a comprehensible user interface. From the perspective of the user, the package produces an HTML5 video element and uses JavaScript to manipulate it. This guarantees that all HTML5 compatible browsers will produce a valid video player with only minor differences.

Design Decisions

H5P decided to provide its content editors with a simple video authoring tool, a screenshot of which can be seen in Figure 3.1. Paired with an extensive interactive video tutorial for authors [15], the tool provides its users with many options for interaction within their videos. The developers have opted to represent many functions with icons in order to ensure a more compact interface while at the same time avoiding possible translation issues.

The authoring tool displays the video to the editor as it would be presented to the viewer, recreating a sort of WYSIWYG[3] usability concept. Furthermore, the tool allows the editors to use drag and drop functionality for drawing and positioning the interactible elements. This makes sure that even editors with very little experience in the field of video authoring can produce meaningful results with this tool, as long as they have at least some experience with watching interactive videos, as well as a little knowledge about the tool's features.

The *H5P* video player, see Figure 3.2, looks like a conventional video player at the first glance, but its features quickly reveal themselves to the viewers. First of all, the player provides its users with a bookmark menu, which can be toggled by clicking the

[2]Content Management Systems
[3]What You See Is What You Get

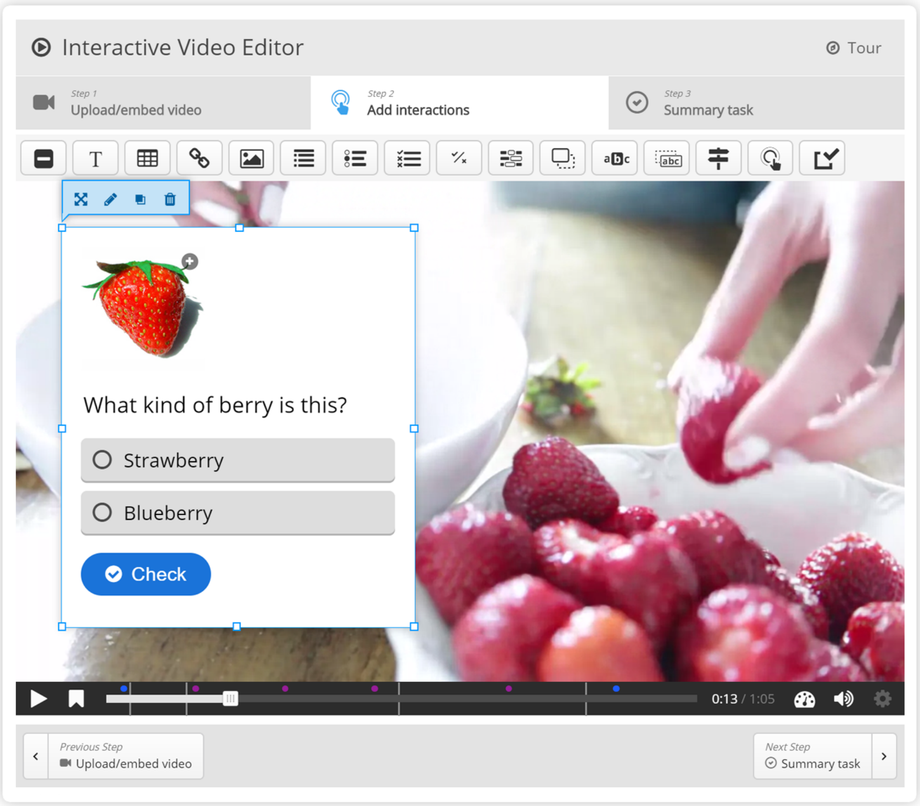


Figure 3.1: A screenshot of the *H5P* video authoring tool [16].

bookmark button on the bottom left. Bookmarks are listed chronologically, make the users jump to the corresponding section of the video when clicked, and are also indicated by vertical white lines along the video progress bar. On the progress bar the users can also find blue and purple markers for additional information, as well as tasks they can fulfill. Those tasks and bits of information are displayed on the screen like the 'Serving tips', and 'Summary' in Figure 3.2. When clicked, they trigger a popup containing their respective content. The remaining control elements of the player include a mandatory play/pause button, a playback rate menu, a sound toggle button, a video quality menu, and a fullscreen toggle button. After a few minutes of testing the player, a few usability issues have been identified.

- The player can only be paused by clicking the pause button. Pressing the space bar or clicking the video does nothing.
- The player can only be muted or unmuted. There is no precise volume control option.
- When trying to pause the video for the next popup elements, the viewer might accidentally unpause the video, since the player pauses automatically for some popups.

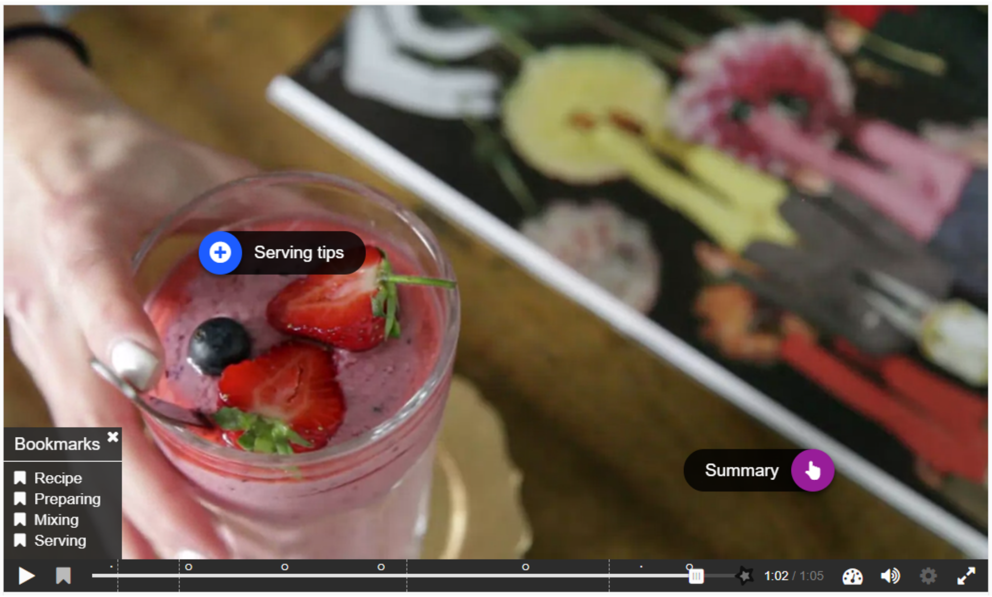


Figure 3.2: A screenshot of the *H5P* video player [16].

3.2.2 YouTube

When talking about web video content, one can not forego mentioning the *YouTube* video empire. The formerly independent, but now *Google*-owned company has been a huge influence on the web video market since its founding in 2005 [19]. While their content is not primarily focused on interactivity, they have provided users with the means for creating interactive video experiences.

Annotations

YouTube's first spin on interactive video was the implementation of video annotations. Through an online video editing tool, users were given the opportunity to add clickable boxes to their videos. These boxed could lead the viewer to another section of the same video, a different video, or a *YouTube* channel. This technology led to the creation of many interactive videos on the platform. Unfortunately, the system was not implemented for mobile devices and never received updates to make it more mobile friendly. Thus, the annotation system was discontinued on May 2, 2017 [30]. This means that annotations can no longer be added or edited, only deleted. Existing annotations still show when using a desktop computer. The system was replaced by the implementation of end screens and info cards.

Technology

The *YouTube* video player itself is proprietary, so it cannot be used or edited freely. In order to include the player in a website, developers can make use of the *YouTube Player API*, which offers an iframe embedding service. The modern video player is based on a combination of JavaScript and HTML5, so at its core, the player uses a HTML5 video

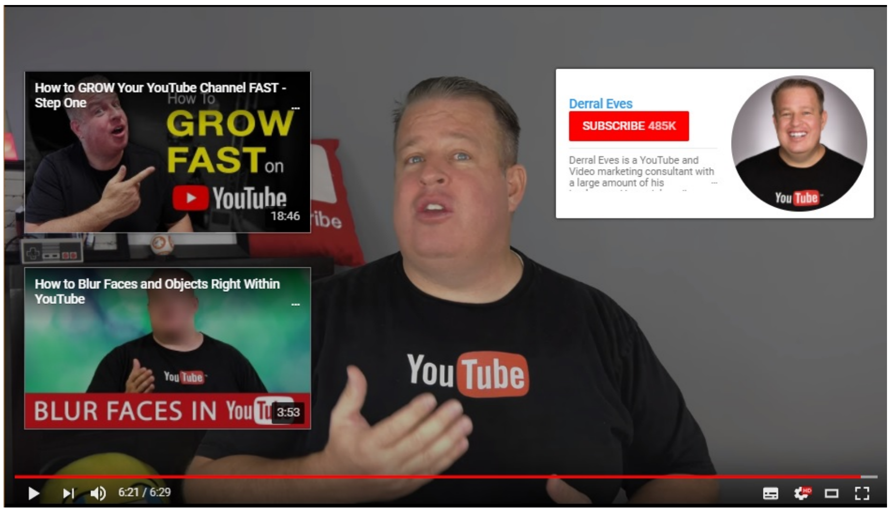

How to GROW Your YouTube Channel FAST - Step One
How To GROW FAST on YouTube
18:46
How to Blur Faces and Objects Right Within YouTube
BLUR FACES IN YouTube
3:53
Derral Eves
SUBSCRIBE 485K
Derral Eves is a YouTube and Video marketing consultant with a large amount of his
6:21 / 6:29

Figure 3.3: A *YouTube* video end screen [12].

element to display the video content. The overlays for the end screen and info card features are also based on HTML5.

Design Decisions

The *YouTube* editor studio offers its content creators a sophisticated video authoring tool with many features. The features of interest here are the end screen and info card tools. As with the *H5P* tool, the video can be played back to enable precise time-wise placement of the interactive elements. Info cards can be placed at any time of the video, while end screen elements can only appear in the last 20 seconds of the video and have to be at least 20 seconds from the start of the video. Both tools offer a few different elements to choose from. End screen elements can promote a different video, a *YouTube* playlist, a channel, or a *YouTube* approved website. In addition to that, info cards can also contain a poll. Apart from those predefined elements, the *YouTube* content creators cannot define any elements themselves. Info cards are restricted to appear in the top right corner of the video, while end screen elements can appear anywhere in the video area, as can be seen in Figure 3.3. Since info cards only appear in the top right corner, their informational content is quite restricted. This is alleviated by the fact that, once clicked, the info cards open up a more detailed menu on the right. Clicked end screen elements immediately redirect the user to their content.

3.3 Fields of Application

3.3.1 Educational Entertainment

The *BBC*[4] *Research and Development* team has produced a new way of expanding peoples culinary expertise. The project called *Cook-Along Kitchen Experience* or, in

[4]The British Broadcasting Corporation (BBC) is a British public service broadcaster.

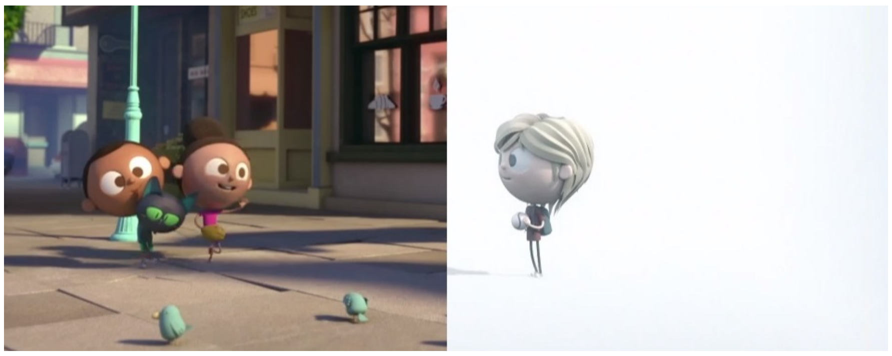


Figure 3.4: A comparison of two opposite *Give A Fuller Life* video states.

short, *CAKE* [2] became publicly available in 2016 and represents a prime example of interactive video content. At the beginning, the application offers various recipes to choose from. Then the user is asked a few basic questions about his kitchen environment and the number of guests he wants to cook for. With this information the application builds a directed graph of video guide snippets. The user is then shown the snippets one by one and is given a chance to review and replicate the cooking steps at his own pace. After each step, the guide pauses and waits for confirmation that the trainee has completed the instructions. This way, even the most helpless cooking apprentices are able to create a delicious dish.

3.3.2 Cognitive Rehabilitation

In their paper [7], Martinez-Moreno et al. have elaborated on the use case for interactive video concerning cognitive rehabilitation for neurologically impaired patients. They planned to have patients deal with activities of the daily life via an interactive video environment, allowing them to navigate those situations at their own pace, while being closely monitored by the therapists conducting the experiment. The team mentions that the means of interaction with the video have to be optimized to resemble the real life scenario and to provide a satisfying user experience.

3.3.3 Interactive Experiences

In their video, titled *Give A Fuller Life* [18], the organization *Mended Little Hearts* presents users with a short animated clip in the day of a child protagonist with a serious heart defect. Depending on how much the users are willing to donate to *Mended Little Hearts*, more details, such as music, environment and additional characters, are added to the video. In Figure 3.4 we can see the different video states in *Give A Fuller Life*. The right half of the Figure displays the video when the user does not want to donate at all, whereas the left half displays the video when the user is planning to donate 20$.

Chapter 4

Technical Design

With the knowledge from the previous chapters, an implementation of an interactive video player was produced. A screenshot of the resulting video player can be seen in Figure 4.1. This Chapter is intended to display and explain the implemented video player from a technical design point of view, while the next Chapter is going to concern itself with a less theoretical approach.

To start off, the requirements of the video player are discussed in order to provide a better understanding for some of the decisions explained later on. To introduce the architecture of the video player from a logical point of view, the most relevant design patterns for the conception of the player are presented afterwards. Finally, the important design decisions that make up the core architecture of the piece of software are discussed and potential alternatives are mentioned.

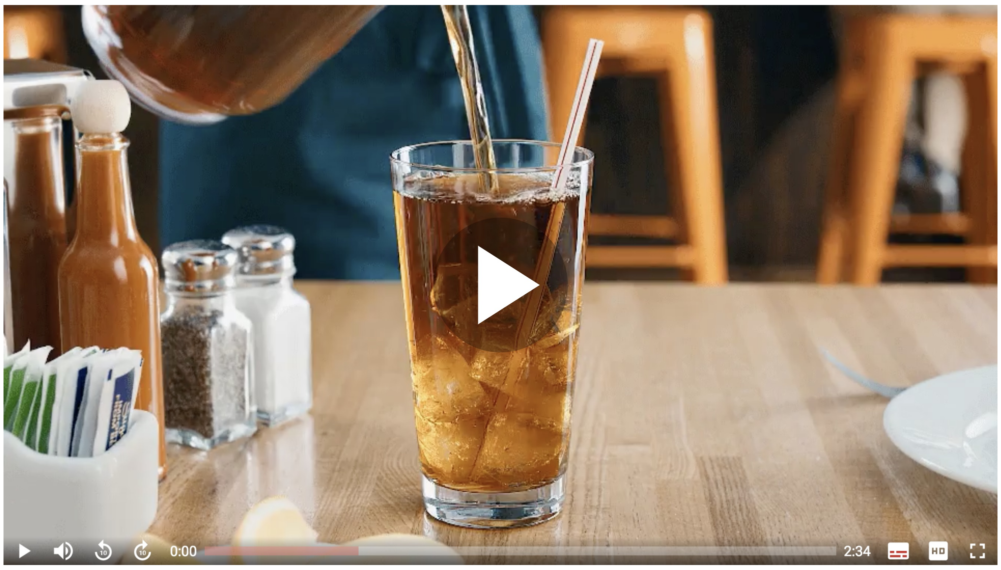


Figure 4.1: A screenshot of the implemented video player's poster screen.

4.1 Video Player Requirements

Since a crude interactive video player with little functionality can be developed within minutes, some requirements for the implementation had to be set up beforehand to guarantee a desirable outcome. These requirements align themselves to common practices concerning modern video players, as well as ideas provided by the editors. They affect the look and feel of the video player on the frontend side as much as the inner workings of the backend components. Consequently, an intended workflow for the video player in a production environment has been developed alongside the video player requirements.

4.1.1 Intended Workflow

For the video player to be any benefit for the editors, they have to be able to create and manage the interactive video content themselves, without any help from a developer. Concerning videos, this is taken care of by the content management system, so there is no difference to a conventional video player. Ideally, the CMS should support the means of interaction of the video player as well. This way, the editors would simply have another few text boxes to fill out and the interactive part of the video would be done. Unfortunately, the means of interaction have to be addressed separately, since no CMS is going to support such a custom made functionality automatically. To achieve the best possible editor experience, the process of adding interactivity to video content should be kept as straightforward as possible, while making use of workflow patterns the editors already apply in other fields. More on the topic of keeping the authoring experience simple can be found in Section 4.3.5.

4.1.2 Deduced Requirements

Composing common practices found within other popular video players along with the suggestions from the editors, a few requirements for the interactive video player could be established. The technical design decisions described in the following sections have proven to be beneficial for fulfilling those requirements:

- clean, but appealing control elements, look and feel,
- buttons for rewinding and fast-forwarding the video,
- buttons for toggling high definition, subtitles and fullscreen,
- moving preview images with a play button and optional text,
- support for all modern browsers,
- simple authoring of interactive elements.

4.2 Design Patterns

A pattern is a schematic solution to a commonly occurring problem. As such, design patterns are schematic solutions to common software design problems. The patterns are an essential part of a structured approach to software development, as they bridge the gap between programming paradigms and concrete algorithms [4]. As any piece of well designed software, this video player implementation also makes use of a few design patterns. The most notable and defining patterns are going to be discussed within this section.

4.2.1 Composite Pattern

The composite pattern is a structural design pattern, which states that a group of objects is treated the same way as a single instance of those objects would be treated. John Vlissides et al. [4] describe the basic intent of the pattern as follows:

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

With the video player being based on the *React* JavaScript library[1], the pattern was almost unavoidable to begin with. Similar to plain HTML, *React* components are treated as composites and can thus contain any number of additional components within them. The implementation makes use of this pattern to split certain functionality off to reusable, lower-level components, such as the button component. The player component provides its buttons with the necessary data for them to function and the buttons handle their repetitive but necessary setup process themselves.

4.2.2 State Pattern

The state pattern is a behavioural design pattern describing the change of software behaviour based on an internal state [4]. React components are naturally built on the concept of keeping a component state object to track the current state of the component and also to determine how the component is supposed to be displayed at a given time. Since any state update triggers a redrawing of the component, the state pattern becomes an integral part of the *React* component lifecycle.

Expanding further on the concept of states, the *Redux*[2] JavaScript library provides a predictable state container, while also changing the state handling rules slightly. A *React* application using *Redux* no longer has a state container for each component, but one state container for the entire application. *Redux* users refer to this as the principle of a 'single source of truth' [9], additionally stating that:

> This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle.

Although the prospect seems interesting for the video player, it does not currently utilize the *Redux* library. Once the application expands to have more components and more deeply nested sub-components, the implementation of the library can once again be considered.

[1] https://reactjs.org/
[2] https://redux.js.org/

4.2.3 Observer Pattern

The observer pattern is another behavioural design pattern. It describes the concept of an *m to n* relationship between an object of interest and its dependents, called observers, wherein the dependents are automatically notified and updated once a change within the state of the object occurs [4]. The pattern is implemented within the video player through the use of JavaScript event listeners. Usually, those listeners consist of a single function, which is called once a specific event occurs. The listeners can be attached to various HTML elements, such as buttons and video elements. Those elements keep track of their attached listeners and take care of executing the functions.

React does not interfere with basic JavaScript event handling, but adds some more events in connection with the *React* component lifecycle, which components can react to as their developers see fit. The video player implementation keeps track of many events which are the result of user interaction, such as clicks on the video player and resizing of the browser, but it also observes events which might not necessarily be connected to a user's input. Most notably, these events include information about the video playback progress, as well as a perpetually repeating function maintaining a healthy state of the player.

4.3 Design Decisions and Alternatives

At many points during a software development project, developers and designers find themselves at a virtual crossroads. Decisions for one or another approach, language, framework, or design have to made. To make an informed decision about such matters, many alternatives to the proposed options have to be considered. This Section presents some of the more important decision which have been made during the video player development.

4.3.1 Building the Player from Scratch

Very early in the implementation process the decision to build the player from scratch has been made. This meant that the starting point of the video player was a very basic video element (see Figure 4.2) and all the required features had to be designed, created and added to this basis manually. A diagram depicting the architecture of the resulting video player can be seen in Figure 4.3. This decision meant opposing the utilization of video player frameworks such as

- *H5P*[3] (see Figure 3.2),
- *JWPlayer*[4],
- *Flowplayer*[5].

Adapting one of those preassembled video players would have greatly increased development speed at first, since a lot of the functionality would have been implemented already, but such a decision would also have severely limited the freedom of choice when it comes to the visual representation of the player, as well as its basic interaction design.

[3]https://h5p.org/
[4]https://www.jwplayer.com/
[5]https://flowplayer.com/

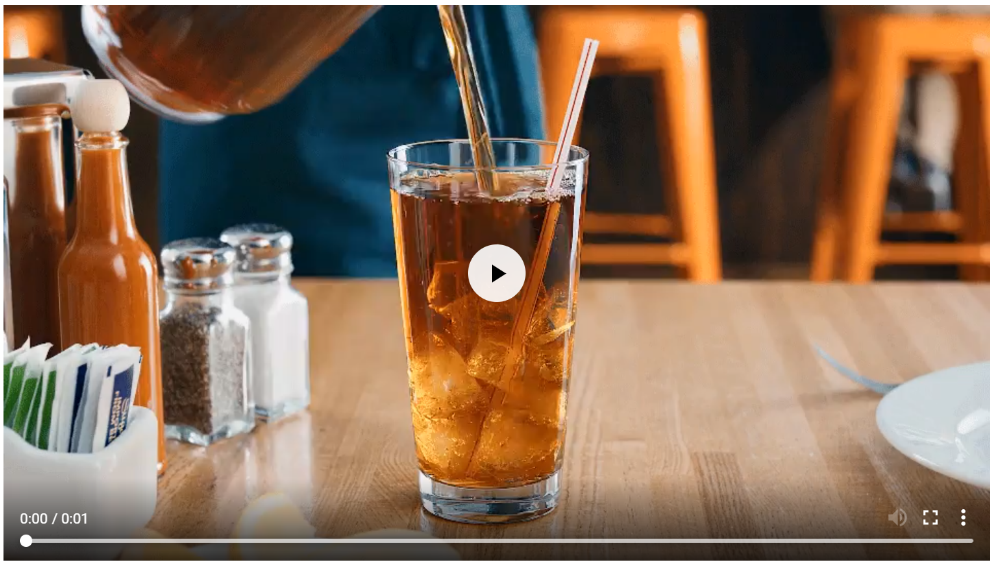


Figure 4.2: A screenshot of the basic HTML5 video element with control bar.

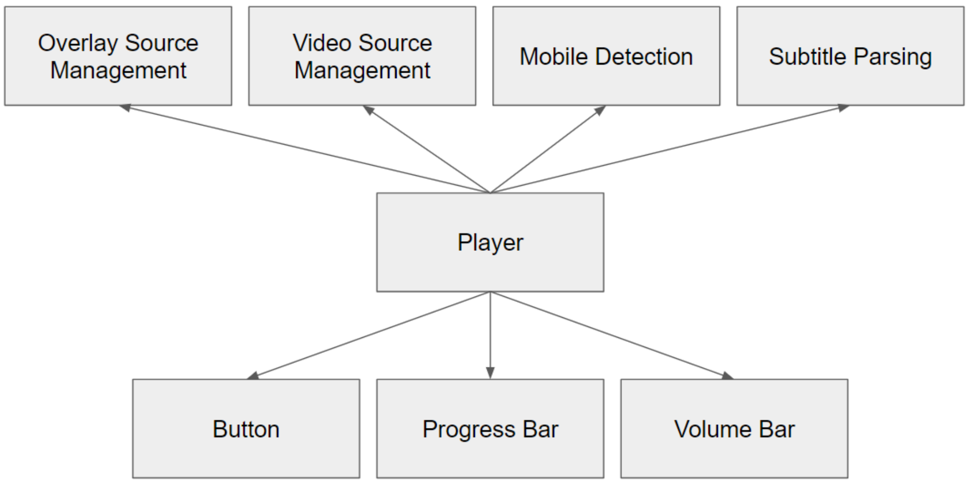


Figure 4.3: A diagram depicting the architecture of the video player.

Additionally, using a framework with interactive video components built in would also hamper the adaptability of those systems.

As it was too early to tell whether one video player's restrictions, the conclusion to start building without a video framework was the only justifiable one. In retrospect the decision still seems sound, since the additional time spent developing the conventional video player functionality has led to a more profound understanding of the underlying structures, which might not have been possible using a complex, prefabricated framework.

4.3.2 Using Plain HTML5

While it was clear that no prefabricated video player frameworks would aid with the development of the video player, one question remained: 'Is the HTML5 video element the best choice for a modern web video player?' A dig into the history of web multimedia reveals that ever since the HTML5 video/audio element dethroned the previous de facto standard of using *Adobe Flash* in 2007 [10], no relevant contenders for web multimedia playback could gain any significant traction. The answer for the remaining choice between *Flash* and HTML5 was obvious, since using *Flash* as a basis for the video player would have been a step back of more than ten years of modern web development. *Flash* can nowadays only be considered as a fallback for ancient browsers without support for HTML5 video. Ultimately, this feature was not implemented, since the additional benefit would hardly be worth the trouble of implementing such a fallback.

While all modern browsers support the HTML5 video element, they are not entirely agreeing on how to deal with the HTML5 progress element, a vital part of conventional video players. From a simplified point of view, the progress element consists of two or possibly three bars laid on top of each other. The bars are intended to visualize the video's current time, the amount of buffered video content and the duration of the video and thus only differ in color and width. Even with such seemingly simple requirements, browsers expect different handling for the styling of their progress elements. Consequently, the video player contains a custom implementation of a progress bar specifically adapted to the needs of the project. This goes to show that not all HTML5 elements are equally supported across browsers, even though they might seem to be at first glance.

4.3.3 Overlays

As mentioned in Section 2.2, the interaction part of interactive web video can take up many varying forms. This implementation provides support for video overlays, which can either link to a website or to a different video source. In the case of a linked website, the destination page will open within a new browser tab. In the case of a different video, the video player will proceed to change its source attribute to the desired video, resulting in an almost seamless transition between the two videos. Thus, the video player provides support for the following means of viewer interaction:

- acquiring additional information about a topic by clicking a link to an external website,
- continuing to consume video content by clicking on recommended videos at the end of one video,
- making choices about video progression, possibly resulting in branching story lines or an interactive video guide,
- switching through multiple views of the same video via overlays.

Since elements within a browser are displayed in layers, the overlays were placed in a layer in front of the video element in order to provide the feeling of a truly interactive video player. Had they been placed below or above the video element, chances are that the video player would merely be seen as a conventional video player with some interactive elements on the side. The implementation deliberately avoided adding additional

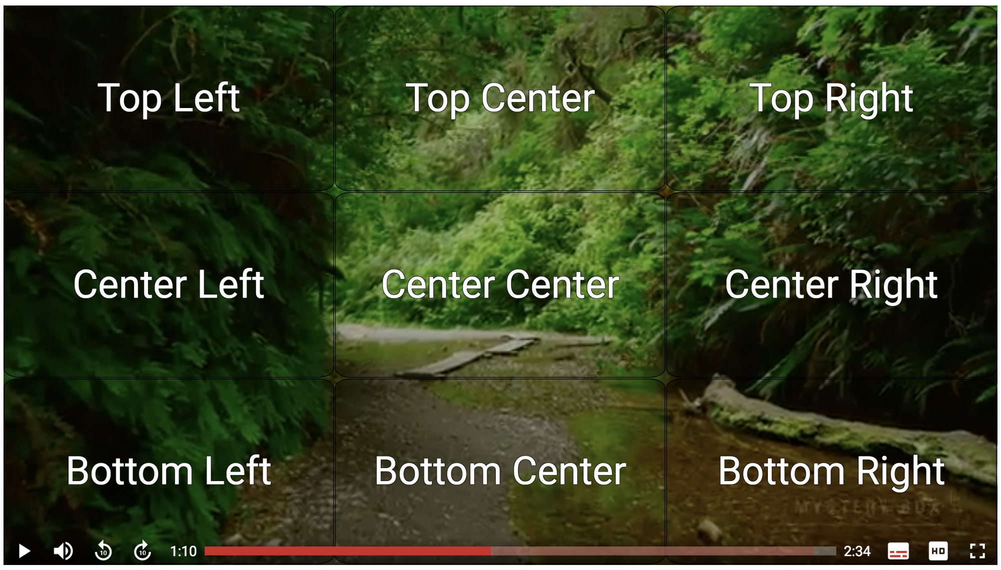


Figure 4.4: A screenshot of the implemented video player's overlay test screen.

content like polls or blocks of information to the overlays. While those means of interaction might seem like a reasonable way of adding interaction to video content, those elements would likely have cluttered the video player area too much for the video content to make sense anymore. Polls and information are better suited to be displayed above or below the video area, or moved to another, possibly overlay-linked, page entirely.

4.3.4 Overlay Positioning

Since making the editors write HTML code was out of the question because of its complexity and degree of possible failure, overlay authoring was designed in a very specific way to make it as easy as possible for non-tech-savvy content editors to envision and realize their own overlays. An integral part of this simplistic design is the positioning of the overlay elements. The video area was divided into a grid of nine sections and each section was given a language-based identifier to allow for intuitive positioning of overlays. The screenshot in Figure 4.4 shows the nine overlay sections with their identifiers written onto them.

While this system makes it very easy for beginners to produce crude, interactive overlays, it is not particularly suited for intermediate or advanced users. It does not provide much of a learning curve, since it only allows those nine fixed positions. Users with more experience in the field of web design might desire a more complex, but also more adaptable system of overlay positioning. Such a system could make use of percentage measurements in order to allow for more customizable elements. Since the overlay authoring is not expected to be handled by tech-savvy people, though, such a functionality was not added to the overlay toolkit initially, but may well be added in the future if necessary.

```
{
    "start": "0:35",
    "end": "1:05",
    "overlays": [
        {
            "text": "Website Overlay",
            "link": "http://localhost:41882/index.html",
            "position": "top right"
        },
        {
            "text": "Video Overlay",
            "video": "http://localhost:41882/video.mp4",
            "position": "bottom left"
        }
    ]
}
```

Listing 1: An overlay collection containing both available overlay types.

4.3.5 Using JSON for Overlays

A few file formats could be considered for the definition of the overlays. One desirable trait of those formats is that the editors have already worked with it and can produce it reliably. The most notable contenders in this case are subtitle formats such as SRT[6] and VTT[7]. Unfortunately, those two formats do not lend themselves towards overlay definition at all and would have to be butchered quite a bit to make them fit the purpose, so no known file formats were suitable. One step further up the complexity ladder are XML and JSON, which already fulfill the requirements to be an overlay format. While XML is not necessarily any worse than JSON, it is not any simpler to read or write, whereas it is easier to work with JSON within a JavaScript environment, so the choice here was clear. The goal was then to create a notation that is comprehensible while not restricting the authoring of overlays any more than the positioning does already. Examples of the overlay notation can be seen in Listing 1, where one set of overlays is defined, containing a website-linking overlay as well as a video-linking one. A diagram describing the underlying notation can be seen in Figure 4.5.

Another option for overlay authoring is a WYSIWYG tool. Such a tool could display the video content while offering to create overlays at the click of a button. This would simplify the process of retrieving the correct timestamps, setting the correct overlay position and producing valid JSON code. The code would then be produced by the tool in the background, so the editors would not have to bother themselves with correct syntax. As with the advanced overlay positioning option, the WYSIWYG option was not implemented initially, but can be added to the video player at a later point if the creation of overlays in JSON turns out to be too cumbersome.

[6]https://matroska.org/technical/specs/subtitles/srt.html
[7]https://developer.mozilla.org/en-US/docs/Web/API/WebVTT_API

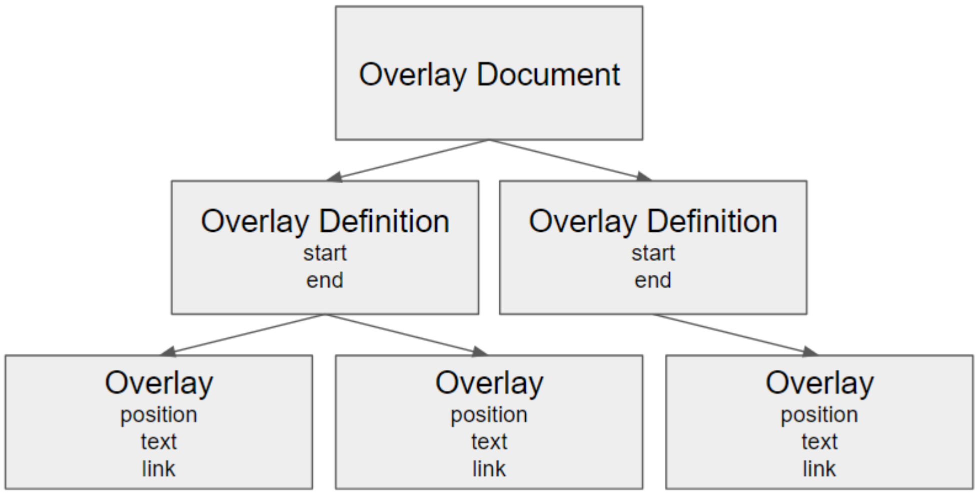


Figure 4.5: A diagram explaining the JSON notation.

	Chrome	*Firefox*	*IE*	*Safari*	*iOS Safari*
VP8 and Vorbis in WebM	6.0	4.0	9.0	3.1	No
VP9 and Opus in WebM	29.0	28.0	?	?	?
Streaming WebM via MSE	?	42.0	?	?	?
Theora and Vorbis in Ogg	Yes	3.5	No	No	No
H.264 and MP3 in MP4	Yes	Yes	9.0	Yes	Yes
H.264 and AAC in MP4	Yes	Yes	9.0	3.1	3.1
FLAC in MP4	62.0	51	?	?	?

Table 4.1: A Table describing the current media format support [22].

4.3.6 Video Formats and Streaming

The HTML5 video element comes with support for a lot of video formats from the get-go, although those supported formats vary from browser to browser. The current media format support according to *Mozilla Development Network* [22] can be seen in Table 4.1.

Based on this media support situation, only the MP4 format can be used on its own somewhat safely. For other formats developers should provide a fallback format, so browsers have more options to choose from. Note that since browsers are continually being updated, this Table might be out of date soon due to developments in the field of modern web media.

On the subject of modern web media, the video player also supports video streaming as opposed to progressively downloading video content. This means instead of downloading a monolithic video file from the server, the video player connects to the server via specialized streaming protocols and only downloads specific parts of the video. Apart from using up less bandwidth by only downloading content once it is needed, this is an improvement in the department of security, since the video file is not stored on the user's machine. At the server side, streaming relieves a lot of unnecessary network traffic. A lot of video content is downloaded from the server while not actually being consumed, so reducing that traffic is a major boon.

Streaming video requires different file formats from regular video content, the most popular of which are *HLS*[8] and *MPEG-DASH*[9]. Since such streaming file formats are quite new technology, not many browsers support them out of the box yet. *Caniuse.com* reports that *MPEG-DASH* is only supported by *Microsoft Edge* and *HLS* only by *Microsoft Edge*, *Safari* and some mobile browsers [13]. Nevertheless, *HLS* support can be added to any browser except *iOS Safari* by including libraries using the *Media Source Extension API*[10]. Therefore, *HLS* is the streaming format of choice for the video player. When provided with video content that needs to be streamed, the video player first determines whether the browser supports *HLS* out of the box. If it does not, it lets the *HLS.js* library[11] handle the case from there on. If it does, no further action is required to ensure the video is played back correctly.

[8]https://developer.apple.com/streaming/
[9]https://www.encoding.com/mpeg-dash/
[10]https://developer.mozilla.org/en-US/docs/Web/API/Media_Source_Extensions_API
[11]https://github.com/video-dev/hls.js/

Chapter 5

Implementation

As mentioned in Chapter 4, a video player was implemented using the foundations from previous chapters. Having discussed the technical design aspects of the piece of software, the actual implementation remains of interest. This Chapter is meant to explain the concrete, inner workings of the video player, as opposed to the more theoretical approach of Chapter 4.

The first Section of the Chapter is going to focus on a the core component of the piece of software and explains how different requirements are implemented within it. Later, the discussion of the actual implementation of the player progresses through the source code one component at a time, while introducing important auxiliary modules afterwards. Many times during the chapter, relevant parts of the implementation are illustrated using code snippets, which show the corresponding JavaScript code.

5.1 The Core Component - Player

From a simplistic point of view, *React* components allow developers to define their own HTML elements, which they can subsequently use to build web applications. At the core of such applications there is always one all-encompassing component, which the *React* framework then injects into one or more concrete HTML elements within the web page. For this video player implementation, the core component is the player component. It represents the centerpiece of the application and composes all other components and modules alongside some basic HTML elements to make up a fully functioning interactive video player application. This Section aims to give detailed information about the inner workings of the player component through the presentation and discussion of relevant code snippets.

5.1.1 Video Elements

While conventional video players settle for presenting their viewers with a static poster image while the video is loading or simply not started yet, the video player takes the video preview functionality one step further. By instating not one but two HTML5 video elements, the video player can present the viewer with a short, looping poster video in order to gather his interest. To make it clear that the user still has to give his permission before the video is going to start playing, an obvious play button is placed in the middle of the poster video. For informational purposes the poster video can also be decorated with a title and subtitle, providing the user with hints about the video content. The two video elements are placed on top of each other via the use of CSS positioning, while the element playing the poster video is naturally placed in the front layer. Once the user interacts with the video player in a way that alters the current time of the video, the foreground video element disappears, revealing the more important background video element. Those kinds of interactions include the following:

- clicking the play button on the control bar,
- clicking the play button in the middle of the foreground video element,
- clicking the foreground video element itself,
- clicking the fast-forward button,
- clicking or generally interacting with the progress bar,
- pressing the space button while the video player is focused.

Since this elaborate video poster system requires the users to have a sufficiently capable internet connection, which cannot always be assumed, the video player can still be provided with a regular poster image to display instead. In this case, the poster image will be displayed as long as the poster video data has not been loaded sufficiently. Once the poster video content is loaded, the video will begin to play on its own, providing an enhanced video preview experience. A simplified version of the Player components' `render` method can be seen in Listing 2. This shows how the video player combines the two video elements, as well as its other components, to provide all the necessary functionality.

5.1.2 Control Bar

Despite not being the cornerstone of interactivity for video players, no video player wishing to provide viewers with a satisfying user experience can go without some type of control elements. The implementation bundles all of its control elements into one control bar. The bar is located at the bottom of the player and features a colour gradient going from transparent to black towards the lower edge of the player. HTML-wise, the control bar is a simple `div` element with CSS flexbox attributes containing control buttons as well as the progress bar of the video. Figure 5.1 shows the control bar with the following elements from left to right:

- play/pause button,
- volume button,
- rewind button,
- fast-forward button,
- current time of the video,
- progress bar,
- total time of the video,
- subtitle button,
- high definition button,
- fullscreen button.

```
render() {
  return (
    <figure className='container'>
      <div className='overlayLoading'></div>
      <video className='base-player' poster={this.state.preview}></video>
      <div className='overlayContainer'>{this.state.overlays}</div>
      <div className='subtitles'>{this.state.subtitleText}</div>
      <video className='loop-player' poster={this.state.preview}></video>
      <div className='previewInfo'>
        <div>{this.props.title}</div>
        <div>{this.props.editor}</div>
      </div>
      <div className='overlayPlay'></div>
      <div className='controls'>
        <Button onClick={this.togglePlay}></Button>
        <Button onClick={this.toggleMute}></Button>
        <VolumeBar className='volume'/>
        <Button onClick={this.rewind}></Button>
        <Button onClick={this.fastforward}></Button>
        <span className='timeText right'>{this.getCurrentTimeText()}</span>
        <ProgressBar className='video'/>
        <span className='timeText left'>{this.getDurationText()}</span>
        <Button onClick={this.toggleSubtitles}></Button>
        <Button onClick={this.toggleHd}></Button>
        <Button onClick={this.toggleFullscreen}></Button>
      </div>
    </figure>
  );
}
```

Listing 2: A simplified version of how the video player combines all its components.

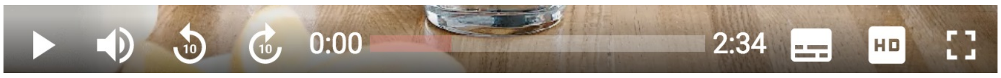


Figure 5.1: A screenshot of the video player's control bar.

Should the user hover over the volume button with his mouse, the volume bar is going appear for a few seconds. It can be used to fine-tune the video volume to a desired value, instead of having to choose between muted and unmuted states. While on mobile, volume control is completely handed over to the device's operating system, so there is no need for the volume button at all. The rewind and fast-forward buttons provide the users with reliable control over the video progress, without having to resort to precise clicks or touches on the progress bar. This proves to be especially useful within a mobile context. Concerning mobile users, the height of the control bar and subsequently the size of the buttons, is also a matter of interest. While within typical desktop screen widths, the control bar has a more subtle height and leaves a lot of space for the video content to shine, but once the application notices a mobile-style screen width, the control bar

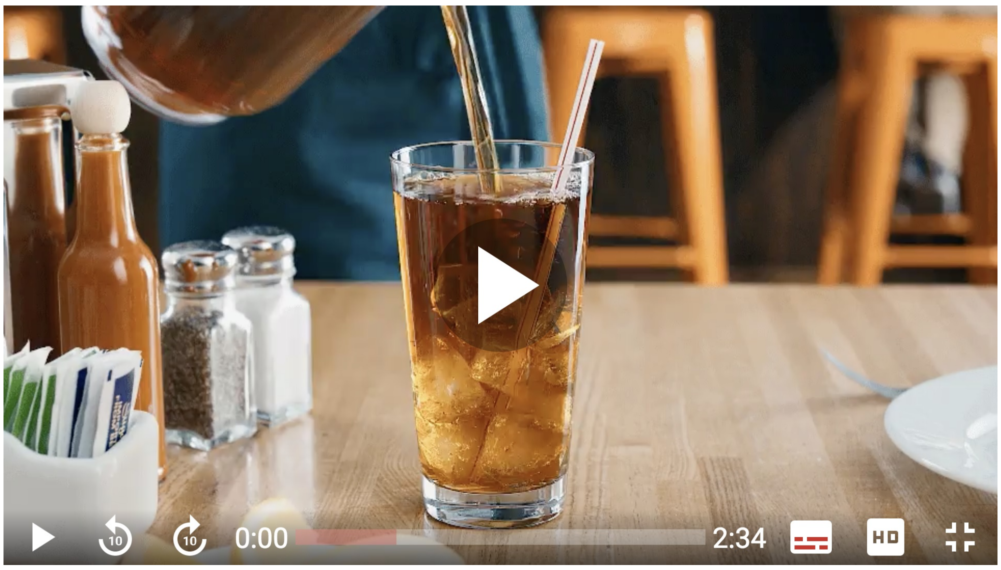


Figure 5.2: A screenshot of the video player's mobile screen.

```
getControlHeight = () => {
    if (window.matchMedia('(min-width: 1400px)').matches)
        return this.state.playerHeight * .05;
    else if (window.matchMedia('(min-width: 1000px)').matches)
        return this.state.playerHeight * .0625;
    else if (window.matchMedia('(min-width: 800px)').matches)
        return this.state.playerHeight * .075;
    else
        return this.state.playerHeight * .1;
}
```

Listing 3: How the player determines the control bar height according to its own height.

increases its height, so the buttons are much more accessible to accommodate for the less precise touch systems. The video player's mobile screen can be observed in Figure 5.2 and compared to the desktop screen in Figure 4.1.

The code responsible for the adaptation of the control bar height can be observed in Listing 3. It utilizes the `matchMedia` function from the basic JavaScript window object to determine the width of the browser window. It then uses this information to choose the correct multiplier ranging from 0.1 to 0.05 for the control bar height calculation. Since the video player is fixed to an aspect ratio of 16:9, the width of the browser window directly influences the required height of the control bar.

5.1.3 Overlays

The cornerstone of interaction within this video player implementation are video overlays. A glimpse of how those overlays are displayed can be seen in Figure 4.4. Since overlay positioning and their definition in general has already been discussed in Section

4.3.4, this Section here is going to focus more on the technical implementation.

Starting with the basic definition of overlays, it becomes self-evident that these elements are required to be displayed in a layer in front of the video content, but inside of the rectangle in which the video content is shown. Consequently, a container `div` element is produced, scaled to the video player's size positioned in a layer in front of it. This container element is subsequently populated with all overlays defined in the JSON file. Overlays which are currently not applicable to the video because the video state does not presently fulfill their requirements are provided with a `display: none` CSS style. It was necessary to create all overlays while keeping some hidden, since the overlays can be defined with a background image attached to them. If the overlays were to be created the moment they became relevant, the image might load too slowly, resulting in a halfway loaded background image being displayed. A predictive system was considered, spawning the overlay elements a few seconds in advance, allowing their images to load properly, but the idea was ultimately discontinued, since viewers can also use the progress bar to quickly advance through the video, making the video state unpredictable. More information about how the overlays are parsed from their JSON definition file and ultimately arrive at the video player can be found in the corresponding part of Section 5.3.1.

The overlays themselves are not outfitted with an event listener for clicks, since this would potentially create a large number of listeners which would slow down the application substantially. Instead, the overlay container is equipped with a single event listener and overlays are provided with ID attributes. Once an overlay is clicked, it can easily be determined exactly which overlay the user wanted to interact with by referring to the event parameters. The relevant parts of the event handler that is called for each overlay click can be seen in Listing 4. After selecting the correct overlay, the handler determines which actions execute, but only if the overlay data signals that the overlay contains an interaction.

5.2 Auxiliary Components

Although the player component handles a lot of the video player functionality itself, a few use cases have warranted being split off into auxiliary components because of their complexity. These components can generally exist on their own, but their functionality has been heavily specialized to fit the needs of the application. As such, they take a few attributes for their setup and subsequently function without further interference. This Section focuses on explaining where the components assist the video player workflow and how they do so.

5.2.1 Button

To consistently provide button elements with event listeners and to reduce clutter within the player component's render function, while making the definition of buttons more convenient and readable, the button component was implemented. Similar to conventional button elements, the `onClick` attribute of the component can be provided with a function, which is called once the button registers a click event. In contrast to conventional buttons, the component does not only apply this event listener to clicks. In

```
overlayClick = (event) => {
    if (this.state.mobile || event.button === 0) {
        // Determine which overlay was clicked
        const i = event.target.dataset.i;
        const j = event.target.dataset.j;
        const overlay = this.state.overlays[i][j];
        if (overlay.functional) {
            // Offers a link?
            if (overlay.link !== undefined) {
                if (!this.state.paused) this.togglePlay();
                window.open(overlay.link, '_blank');
            }

            // Offers a video?
            if (overlay.videoSource !== undefined) {
                this.setState({
                    subtitleSource: overlay.subtitleSource,
                    videoSource: overlay.videoSource
                }, () => { this.loadVideo(this.state.hd); });
            }
        }
    }
}
```

Listing 4: Determining what to do with an overlay click event.

order to support mobile devices more responsively, the event handler is also added to the `touchstart` event. This avoids the pause between the registration of the touch and the actual click event being sent. Mobile devices wait before issuing a click event, since they want to give the user more time to differentiate between click and drag actions, but since both of those actions should trigger the video player buttons, the pause can safely be avoided most of the time. More information about how the event listeners are attached can be fount in Listing 5. In the Listing it can be seen that the given event handler is further wrapped into `event.preventDefault` and `event.stopPropagation` commands to make sure that the triggering event is handled correctly and does not propagate to pause the video, for example.

If the developer does not want to skip the pause on mobile, he can add the `noTouch` attribute to the button. The video player uses this functionality to ensure its fullscreen button works properly. Due to a special restriction from the fullscreen API[1], the web fullscreen functionality cannot be triggered by touch events that are meant to scroll around the page. Thus, the video player should ignore the touch event in this case and wait for the device to determine that the user definitely wanted to click the fullscreen button.

Button components also have to switch between active and inactive states in order to properly convey their functionality. The play button, for example, has to transform into a pause symbol, once the video is started. For this, the button component uses the

[1] https://developer.mozilla.org/en-US/docs/Web/API/Fullscreen_API

```
componentDidMount() {
    const wrappedHandler = (event) => {
        event.preventDefault();
        event.stopPropagation();
        this.props.onClick(event);
    };

    if (this.props.onMouseMove !== undefined)
        this.ref.addEventListener('mousemove', this.props.onMove);
    if (this.props.onClick !== undefined) {
        this.ref.addEventListener('click', wrappedHandler);
        if (this.props.noTouch === undefined)
            this.ref.addEventListener('touchstart', wrappedHandler);
    }
}
```

Listing 5: Adding event listeners within the button component.

attributes `active`, `idActive` and `idInactive`. The ID attributes determine the ID the button is going to receive depending on its active state. This ID can subsequently be used by the CSS to set the correct background image for the button. `active` requires a boolean value that determines whether the button is currently active, thus influencing the ID of the button.

5.2.2 ProgressBar and VolumeBar

As stated in Section 4.3.2, the video player contains a custom implementation of the HTML5 progress element. This is due to the fact, that many browsers treat the HTML5 differently in some way, since the specification of the element does not seem to be restrictive enough. For the video player, the progress element just needed to provide two to three separately stylable bars to display the video state, so it was more efficient to just write a custom component, instead of dealing with every browser's quirks. Thus, the progress and volume bar components were created. The video player's progress bar is permanently visible on the control bar, while the volume bar only shows up when the user hovers over the mute button. The volume bar also grows vertically instead of horizontally like the progress bar.

Apart from showing the user his current progress of watching the video, the progress bar of a video also provides the functionality of seeking. Seeking describes the act of clicking on the progress bar to quickly jump from one point of the video to another. To do this correctly, the video player has to determine the position of the user's click in relation to the width of the progress bar. To see how the video player handles click events within the progress bar, refer to Listing 6. Here it can be seen that the initial click activates a seeking mode for the player, while subsequent `mousemove` events trigger a function which sets the current time of the video, or, in the case of the volume bar, the volume of the player, to the position of the user's click.

```
progressClick = (event) => {
    this.focusContainer();
    this.setState({ progressMousedown: true });
    this.setProgressToClientX.call(this, event.clientX);
}

mouseMove = (event) => {
    if (this.state.progressMousedown)
        this.setProgressToClientX.call(this, event.clientX);
    if (this.state.volumeMousedown)
        this.setVolumeToMouse.call(this, event);
}

setProgressToClientX = (clientX) => {
    const rect = this.progressBar.getBoundingClientRect();
    const pos = (clientX - rect.left) / rect.width;
    this.player.currentTime = pos * this.player.duration;
    this.progressBar.set('progress', pos * this.player.duration);
}
```

Listing 6: How clicking and moving along the progress bar affects the video.

5.3 Auxiliary Modules

Much like auxiliary components, the modules discussed in this Section support the video player in one way or another. Unlike components though, modules do not represent React components or classes. Some of these modules only contain a single function which could not be fit into one the components above for reasons of keeping the code orderly and logically separating parts of code.

5.3.1 Overlay Source Management

Being the most extensive of these four modules, the overlay source management has a lot of tasks to cover. For starters, it deals with a custom notation for subtitles and overlays, while also providing support for subtitles in SRT format. Furthermore, it also translates the custom notation into a reliable format, since the JSON notation provides editors with some leeway concerning their time formats, for example. The overlay management also offers the player several methods to retrieve the correct subtitle or overlay for a given time within the video. All this is done while caching the requested files, so they do not have to be loaded and parsed anew for every request sent to the overlay manager.

Upon mounting the player component issues a command to the overlay manager to retrieve and parse the overlays from a given URL. As part of this, the overlay manager converts the overlay positions and video URLs within the JSON file to a more strict format, so the player component has less trouble through having to react to editor input. Since parsing can be a rather expensive operation in terms of CPU strain, the result of this request is stored within the player's state for further use. The player component then builds the HTML structure for the overlays itself, since from a conceptional point of view, the module has no business dealing with HTML code.

```
getSources = (videoId) => {
    if (cache[videoId] !== undefined) {
        return new Promise((resolve, reject) => {
            resolve(cache[videoId]);
        });
    } else return axios.get(`${apiUrl}${videoId}`).then(bits => {
        return Promise.all([
            axios.get(bits.data[0].sources.q4a.loadBalancerUrl)
                .then(q4a => extractHLS(q4a.data)),
            axios.get(bits.data[0].sources.q8c.loadBalancerUrl)
                .then(q8c => extractHLS(q8c.data))
        ]).then(results => {
            const videoSources = {
                sdvideo: results[0],
                hdvideo: results[1],
                preview: bits.data[0].sources.q8c.preview
            }
            cache[videoId] = videoSources;
            return videoSources;
        });
    });
};
```

Listing 7: How a video ID is transformed into a streaming URL.

Periodically, the player component then checks back with the overlay manager to receive an array of indices, indicating which overlays to display. Alongside the call for the indices, the overlay manager is also asked to provide the player component with the text that should currently be displayed as a subtitle for the video. Since subtitles and overlays largely use the same notation, most of the code for element lookups and other tasks can be reused.

5.3.2 Video Source Management

Similar to the overlay source management, the video source management makes sure the player component ends up with the correct video files. For normal video URLs, not much has to be done to guarantee that, but editors can also choose to direct the player towards video by using their video ID within a specialized filehandler API. In order to do so, the editor does not provide any video URL and only provides the video player with an ID. As can be seen in Listing 7, the video source manager then calls upon the filehandler API to receive several video URLs for different increments of video quality. After choosing two appropriate video quality options, the video source manager consults the API again to receive streaming URLs for those options. From the list of URLs corresponding to available streaming protocols, the manager picks the *HLS* compatible one and hands it back to the player component. The video player then proceeds to stream the video content either directly or through the use of an *HLS* enabling library. In case the player component is going to request the video source data for the same video ID again, the video source manager stores the gathered data within a cache object. This enables the

```
    1
    00:00:00,000 --> 00:00:05,000
    The quick brown fox jumps over the lazy dog
```

```
{
    "start": 0,
    "end": 5,
    "text": "The quick brown fox jumps over the lazy dog"
}
```

Listing 8: An SRT and a JSON subtitle.

manager to deal with future requests more quickly, as it does not have to contact the filehandler API another two times to collect the required data.

5.3.3 Mobile Detection

Since a large amount of modern web traffic comes from mobile devices, competent web developers make sure their web pages and applications look presentable and perform adequately on both mobile and desktop platforms. In order to adapt the behaviour of an application depending on whether the visitor is using a mobile or desktop device, the application first has to determine which one of the two it is. Mobile device detection is usually done through analysis of the browser's user agent string, which contains information about the browser version and operating system. The video player uses an adaption of the very lightweight *DetectMobileBrowsers*[2] script to match the user agent string with every mobile device string available. This makes the method of discovering mobile users almost failure proof, assuming that the module has access to an unmodified user agent string and that the module is updated regularly.

5.3.4 Subtitle Parsing

Although the video player comes with its own subtitle notation, editors might still prefer to use more conventional subtitle formats. In order to satisfy their needs, a parsing mechanism for such formats had to be devised. The format of choice for the time being was SRT, but any other format would have served the purpose just as well and would not have caused any more problems during the implementation. The incorporation of such files from an editor's point of view is very simple, since he only has to direct the video player to the SRT file, instead of the JSON one. As long as the URL given to the application ends in either `.json` or `.srt`, the overlay source manager knows how to respond correctly. A side-by-side comparison between the two formats can be found in Listing 8.

Ideally, the video player does not have to adapt to different source types itself, so the overlay source manager makes use of an SRT parsing module to convert the provided SRT file to the JSON notation the video player already understands. Referring to Listing 9, it can be seen that the SRT content arrives as plain text. After setting up a few

[2]http://detectmobilebrowsers.com/

```
parseSrt = (data = '') => {
    const subs = [];
    let idx = 0;
    let lastLine = 0;
    let time, text, sub;
    let endIdx = lastNonEmptyLine(lines) + 1;
    const lines = data.split(/(?:\r\n|\r|\n)/gm);

    for (let i = 0; i < endIdx; i++) {
        lastLine = i;
        try {
            sub = {};
            text = [];
            i = nextNonEmptyLine(lines, i);
            sub.id = parseInt(lines[i++], 10);
            time = lines[i++].split(/[\t ]*-->[\t ]*/);
            sub.start = toSeconds(time[0]);
            idx = time[1].indexOf(' ');
            if (idx !== -1) time[1] = time[1].substr(0, idx);
            sub.end = toSeconds(time[1]);
            while (i < endIdx && !!lines[i].trim())
                text.push(lines[i++]);
            subs.push(sub);
        } catch(error) {
            return {
                line: lastLine,
                error: 'Subtitle conversion error.'
            }
        }
    }
    return subs;
}
```

Listing 9: How an SRT subtitle file is parsed to fit the JSON notation.

variables, this text is split into lines, which are subsequently processed. The parsing process relies on the assumption that the file adheres to the official SRT specification, but for the unusual case that it does not, the processing step of each line is wrapped in a try-catch block. Should the procedure fail at any point, a failure notification is returned which contains the index of the last processed line. When the whole operation is done, the result adheres to the JSON subtitle notation of the video player so it does not have to worry about file formats.

Chapter 6

Evaluation

The primary objective of this thesis was the implementation of an interactive video player alongside corresponding authoring mechanisms in such a way that the user experience of both viewers and editors is increased or at least provided at the level of the current state of the art. To ensure that this goal has been met, usability tests with members of the appropriate user groups were necessary. This Chapter contains three sections on this matter. The first two of those are aimed at describing the group of participants, the testing procedure, and the results of the editor and the viewer usability test respectively. The third Section displays the opportunities for further improvement for the video players, which have been deduced from the results of the usability tests.

6.1 Editor Experience

This Section is centered around the user experience test for editors using the authoring part of the implemented video player. Afterwards, the testing procedure will be examined in great detail, while clarifying the purpose the individual parts of the test. Any limitations the testing procedure might have had are also of interest. In the end, the results of the test will be presented and discussed.

6.1.1 Participants

In their article, Nielsen et al. [8] claimed that to find all of an interfaces usability flaws, tests had to be conducted with at least 15 participants. A corresponding graph depicting the relationship between the number of participants and the percentage of usability issues found can be seen in Figure 6.1.

They furthermore mention though, that with enough time and budget to test so many people, the testing effort should be divided into three separate tests [14]. Between those tests, the interface should undergo another iteration in order to fix the found flaws immediately. The next test should then theoretically find some undiscovered issues from the previous test, as well as any issues that might have been produced as part of the iteration. Accordingly, the viewer experience test was conducted with five participants to ensure a good balance between the time spent testing and the number of usability issues identified.

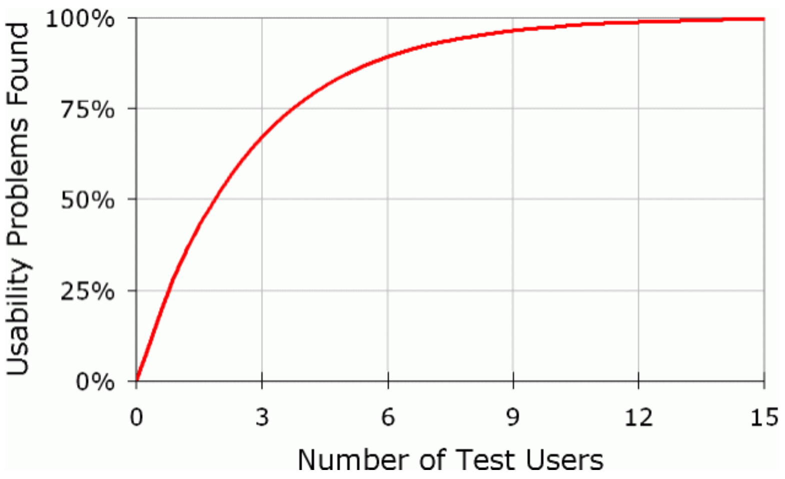


Figure 6.1: A graph depicting the relationship between the number of participants and the percentage of usability issues found [14].

Contrary to a conventional usability analysis like the viewer experience test, this test placed a substantial restriction on the pool of participants, since the authoring procedure had to be tested by users who would actually have to include the workflow into their work routine in the end. Following Nielsen's advice concerning participant numbers, five editors could be recruited for the test despite those restrictions.

During the test, some demographic data about the participants, as well as data about their PC usage, was gathered in order to enable the drawing of conclusions based on it. The information that was collected was the following:

- hours of daily PC usage,
- programming experience,
- preferred operating system,
- age.

Starting with the hours of daily PC usage, it can be observed that the average number of hours is significantly higher than the average of viewer experience test participants. This is due to the fact that the editors are required to use a PC for their work, which adds another few hours each day to the number of hours they spend in front of a PC at home. A graph depicting the daily amount of hours the editors spend using a PC can be seen in Figure 6.2. PC usage was expected to be essential for the completion of the tasks, since many of the necessary processes, like copying, pasting, saving, and refreshing web pages, translate well from day to day PC usage.

Another crucial part for the tasks given to the participants was their previous programming experience. With reasonable insight into the inner workings of computer software, it was expected that participants would have no problems. As apparent in Figure 6.3, the participating editors only stated having either 'no experience' or 'only experience with HTML and CSS', which is unfortunate, but also expected, as programming skills are not required for their occupation. On the other hand, this circumstance makes sure that the system is ready for use without any educational requirements.

The third potentially important factor for the test was the participants' preferred

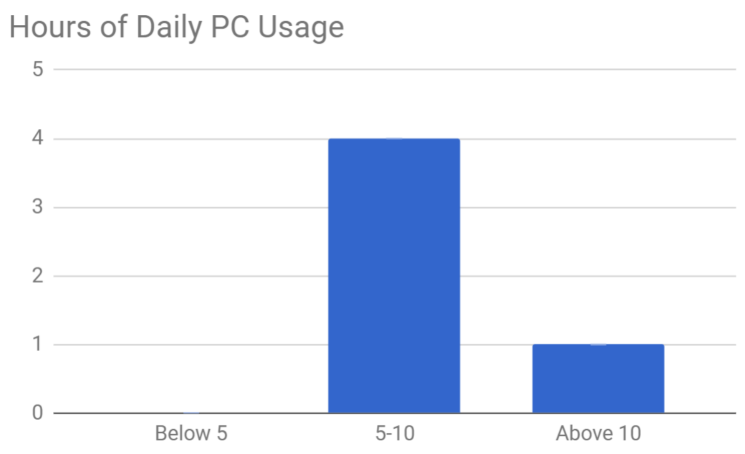


Figure 6.2: A graph depicting the daily amount of hours the editors spend using a PC.

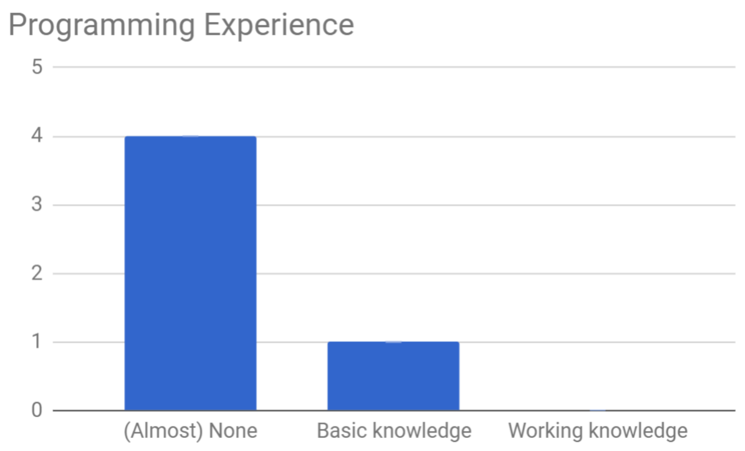


Figure 6.3: A graph depicting the participants' programming experience.

operating system. In an ideal testing environment, this would not have been an issue, but due to the way the test was set up, users with preferences for *Apple's* operating system, *Mac OS*, had an advantage over others. This ties into the necessary processes mentioned while discussing PC usage. Since the keyboard shortcuts for those commands are different on *Windows* and *Mac* devices, users without *Mac OS* experience were expected to be more hesitant when it comes to executing them. The preference distribution graph can be seen in Figure 6.4.

The last piece of data that should be mentioned is the age distribution, which can be seen in Figure 6.5. Although no assumptions concerning a correlation between age and testing performance have been made, the data was collected in order to find a correlation if there is one.

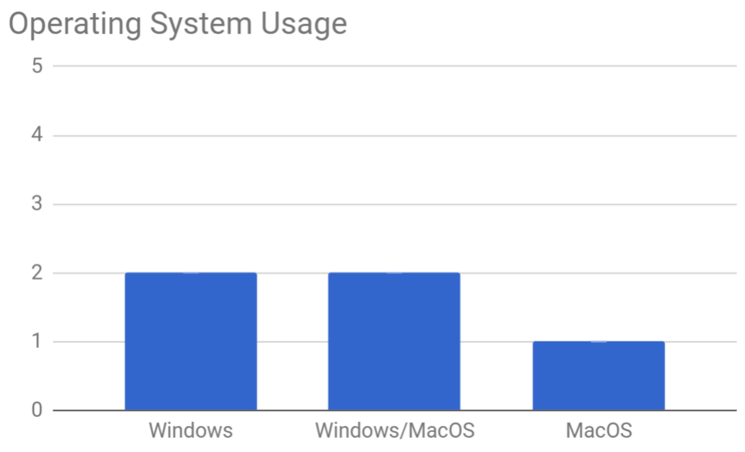


Figure 6.4: A graph depicting the participants' operating system preferences.

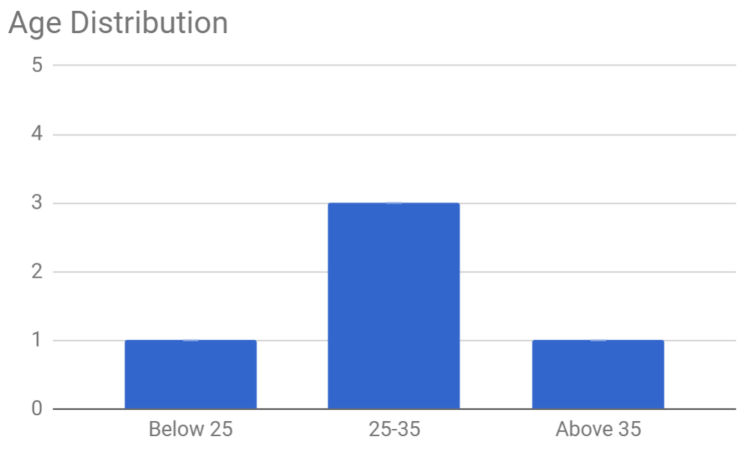


Figure 6.5: A graph depicting the age distribution of the editor experience test.

6.1.2 Procedure

The basic purpose of the editor user experience test was to determine whether the intended method of authoring overlay content for videos is compatible with the editors. As discussed in Section 4.3.5, the process consists of editors modifying a JSON overlay file while adhering to a custom JSON notation. This was the cause for some concern though, as most of the editors had never once written or seen JSON before and the notation was new to them as well. Consequently, a quick, one-page guide for the notation was designed where they could see a few sample overlays with a textual description of each line of code. The hierarchy of the elements found in the JSON file was also depicted similar to Figure 4.5. Lastly, the quick guide contained a few tips and facts about the overlay file, like how the `position` element is composed, that the overlays have to chronologically sorted, and the difference between defining a `link` and a `video`. The

```json
{
  "start": "0:32",
  "end": "0:40",
  "overlays": [
    {
      "position": "top right",
      "text": "In Memoriam: Sephen Hawking",
      "link": "http://tv.orf.at/highlights/orf1/
       ↪  180315_in_memoriam_stephen_hawking100.html"
    }
  ]
}
```

Listing 10: The JSON code the editors had to modify in the first scenario.

Time	3:24 - 3:30
Position	top right
Text	Wikipedia: Athetose
Link	https://de.wikipedia.org/wiki/Athetose

Table 6.1: The overlay data for the second scenario.

editors were presented with the JSON quick guide before their test started in order to properly prepare them for their tasks.

The usability test was split into three scenarios, each of which covered a specific use case of the overlay authoring system. The scenarios were set up with a believable story to them, so the editors could identify themselves more easily with their given task. The first scenario, for example, mentioned that the editor had watched the most recent video and had discovered a typo in one of the overlay texts. The corresponding task for the editor was to locate and fix the typo within the overlay file. By starting the test off with a simple textual replacement task, the editors were given time to accommodate themselves with the test setup, the saving and browser refreshing processes, as well as the JSON file. This was meant to ensure that any issues appearing during the completion of the following tasks can be attributed to problems of the authoring system. The code the editors had to modify can be seen in Listing 10. The typo was located within the `text` attribute value 'In Memoriam: Sephen Hawking'.

The second scenario had the editors create their first overlay. They were given the data for the overlay in a table format and copying an existing overlay to avoid issues was suggested. This scenario was meant to determine whether the editors could easily produce new overlay content on their own, assuming they had basic knowledge about the overlay notation and some existing overlay content to base their creations off of. The given overlay data can be seen in Table 6.1 and the code the editors ideally produced can be seen in Listing 11.

The third scenario had the editors create another overlay. Again, they were given the data for the overlay in a table format. This time, the overlay offered additional video content instead of a link to a website. Since editors were expected to experience some

```json
{
  "start": "3:24",
  "end": "3:30",
  "overlays": [
    {
      "position": "top right",
      "text": "Wikipedia: Athetose",
      "link": "https://de.wikipedia.org/wiki/Athetose"
    }
  ]
}
```

Listing 11: The JSON code the editors had to produce in the second scenario.

```json
{
  "start": "8:12",
  "end": "8:21",
  "overlays": [
    {
      "position": "center right",
      "text": "Selbstversuch: Sprechen mit dem Sprachcomputer",
      "link": "http://cdn.einser.info/videos/als/
       ↪  0-T119-MEINS-SELBSTVERSUCH-PC_H264-720p_BT-V2.mp4"
    }
  ]
}
```

Listing 12: The JSON code the editors had to produce in the third scenario.

issues during the second scenario, this scenario was added to find out whether those issues can be resolved by adding a bit of routine. The code the editors ideally produced can be seen in Listing 12.

After each scenario, meaning when the participants indicated that they were done with their task, they were asked the following questions:

- How did you feel during the completion of this task?
- How hard was this task for you?
- Are you satisfied with the result you achieved?

These questions were meant to assess the difficulty of the individual tasks and to find out how much the acquired routine helps with the transition from the second to the third scenario, which are objectively almost equally challenging. The question about their satisfaction with their result was asked to make sure the participants were able to complete the task as they saw fit, not being held back by any issues with the authoring system. The participants were asked another set of questions after they were done with all three scenarios:

- Do you feel that you are able to work with the JSON notation?
- Do you see how these overlays can enhance current video content?

- Do you think that users are going to utilize the overlays?
- Do you think that other editors are going to have similar success with the notation?
- How would you improve the system?
- In an ideal world, how would you like to create overlays?
- Would an overlay tool, which writes JSON for you, improve your experience?

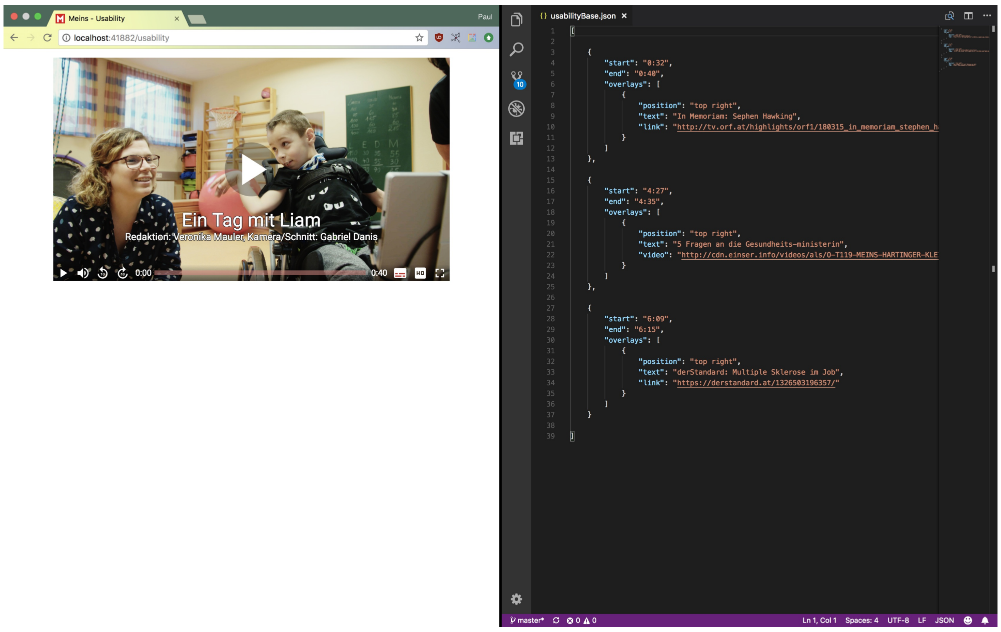
These questions were asked to get a feeling for the editors general satisfaction with the implementation of the overlay system. It was also important to gauge whether the editors would feel safe if they had to work with the authoring system in the future or explain it to someone else from the same department. The last three questions were aimed at collecting ideas for improvements to the system, as well as a possible implementation of a more CMS-like authoring tool, which would take the writing of JSON out of the equation.

The test was setup on a *MacBook Pro* hosting a local web server. The screen the editors were presented with during their test can be seen in Figure 6.6. The left side of the screen contains *Google Chrome*[1], which was used to view the hosted website since it proved to be the most reliable browser for local testing and general development of the video player. The JSON document was opened on the right side of the screen in *Visual Studio Code*[2], although any text editor with syntax highlighting for JSON would have done a similarly good job. The setup on a *MacBook* posed a limitation for the test, since not all of the editors were proficient at handling the Mac operating system. Ideally, the test would have been set up on two separate devices in order to be able to offer the test on both *Windows* and *Mac OS*. To find a compromise between ideal testing and organizational convenience, the decision has been made to keep the test on one device, but to include the editors preferred operating system in the questionnaire to be able to relate problems to this information. The server had to be set up locally in order to enable a quick feedback loop of editing the JSON document, saving, reloading the website in the browser, and observing the changes.

6.1.3 Results and Discussion

The results of the test were generally satisfactory, since no tests had to be interrupted for reasons of failure or stress and no participant left the test feeling unsatisfied with their performance or the overlay system. Because the system was outfitted with appropriate safety mechanisms, the software infrastructure handled the malformed JSON files the editors have produced during the process well. The notation itself was not a factor of complication for the tasks, since the attribute names and allowed values were very straightforward. JSON itself was more of a problem, considering it can be quite pedantic when it comes to correct placement of brackets, quotation marks, colons, and commas. Many times the editors did not fully copy a reference overlay for editing and ended up with a couple of syntax errors due to missing brackets. Although it took them some time to fix those errors, they were very patient and able to fix them on their own with the help of Visual Studio Code highlighting the faulty areas.

[1] https://www.google.com/chrome/
[2] https://code.visualstudio.com/


Meins - Usability
localhost:41882/usability
Ein Tag mit Liam
Redaktion: Veronika Mauler, Kamera/Schnitt: Gabriel Danis
usabilityBase.json
master*
Ln 1, Col 1 Spaces: 4 UTF-8 LF JSON

```json
[

    {
        "start": "0:32",
        "end": "0:40",
        "overlays": [
            {
                "position": "top right",
                "text": "In Memoriam: Sephen Hawking",
                "link": "http://tv.orf.at/highlights/orf1/180315_in_memoriam_stephen_h
            }
        ]
    },

    {
        "start": "4:27",
        "end": "4:35",
        "overlays": [
            {
                "position": "top right",
                "text": "5 Fragen an die Gesundheits-ministerin",
                "video": "http://cdn.einser.info/videos/als/O-T119-MEINS-HARTINGER-KLE
            }
        ]
    },

    {
        "start": "6:09",
        "end": "6:15",
        "overlays": [
            {
                "position": "top right",
                "text": "derStandard: Multiple Sklerose im Job",
                "link": "https://derstandard.at/1326503196357/"
            }
        ]
    }

]
```

Figure 6.6: The screen the editors were presented with during their test.

Another issue the editors encountered, but were not fully aware of, was the differentiation between the `link` and the `video` attribute. Since most of them copied the same reference overlay for both the second and third scenario, it contained a `link` for both of them. This was not perfectly correct for their task in the third scenario though, since this one expected the editor to change the attribute name to `video`. Coincidentally, this did not result in particularly poor overlay behaviour due to the fact that when a modern browser is instructed to open a video URL in a new tab, it automatically wraps the video content into a basic HTML5 video element. The resulting browser view can be seen in Figure 6.7.

A quick solution to the problem would be to scrap the distinction between links and videos in the notation, so the editors would not even have to consider adjusting their behaviour depending on the type of overlay they want to create. By checking the URL for file name extensions or alternatively analyzing the response type, this work could be done by the overlay parsing module instead. The last, minor issue one editor had with the JSON notation was an insecurity whether she had to write an American English 'center' or a British English 'centre' for the overlay position value to work. Although the correct value was quickly looked up, this issue could have been prevented by simply allowing both versions for the position attribute.

Moving on to the set of questions the editors have been asked after the first scenario, it can be seen that the general purpose of the scenario has been met. Editors had to adjust themselves to the environment they had been put in and the tasks they were given. Most of them mentioned that, although their task was very clear to them, they first had to learn how to make changes to the overlays by modifying the JSON, saving, and reloading the browser window. Some of the participants claimed to be overwhelmed by the interface at first glance, since they had never seen any code before.

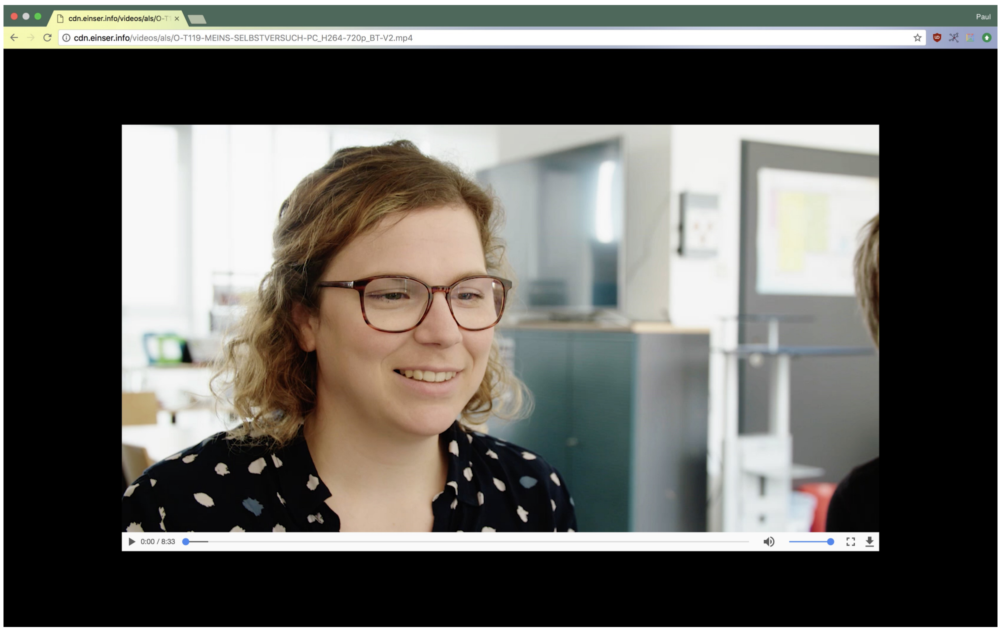

cdn.einser.info/videos/als/O-T
Paul
cdn.einser.info/videos/als/O-T119-MEINS-SELBSTVERSUCH-PC_H264-720p_BT-V2.mp4
0:00 / 8:33

Figure 6.7: How a modern browser reacts to a video URL in a new tab.

The second set of questions shows a lot less confusion about the interface and general processes required to manipulate the overlays. That said, there were a few problems that occurred during the copying of the reference overlays, which made it harder than necessary to complete the task. Nevertheless, more than half of the participants answered that the task was not particularly hard for them. When asked why they felt this way, one of them stated: 'We are editors. Copying, pasting, and modifying is an integral part of what we do.'

The third set of questions was supposed to show a positive trend concerning the perceived task difficulty, but since the reports from the second scenario were already mostly positive, there was little trend to be observed. Nonetheless, the two editors, who expressed having difficulties during the second test, showed much more confidence in their ability to solve the problem and stated that the routine helped them immensely. As mentioned above, this was the scenario in which the participants had to produce a video overlay instead of a link overlay, a fact that four of them did not realize. Despite producing a slightly inappropriate overlay, all of the participants reported being satisfied with the result they achieved.

Looking at the questions posed after the completion of the test, a few answers and statistics are notable. For instance, four out of five editors felt like they had a decent grasp of the overlay authoring system after the test, as long as they have reference overlays to copy from. All of the participants were able to see the immediate benefit the implementation of overlays brings for their current video content and they were positive the viewers are going to click on the elements if they seek the additional information they provide. Most editors expected their colleagues to do similarly well on the test as well as with the authoring system in general, independent from how well they did. When asked about their ideas for improving the system, four editors stated they would prefer a more CMS-like way of overlay definition, avoiding having to write well-formed JSON code.

During the test, two suggestions for improving the viewer's user experience have been made by the editors. The first one proposes being able to open overlay links and videos in a new, unfocused tab by left-clicking the overlay while holding the CTRL (*Windows*) or the Command (*Mac OS*) key, or by using the middle mouse button. Since link overlays already open in a new tab, this is only a truly noticeable improvement for the use case of video overlays. There it can be useful if the viewer does not want to stop his current video, but still wants to watch additional content later on. Generally, this change provides the users with more reliable video player behaviour, considering the next suggestion, because the second one recommends a visual distinction between link and video overlays. From a user experience standpoint, having no distinction between those two options is a major oversight, since the viewer currently has no way of knowing what a click on the overlay is going to do exactly, if it is not clearly defined by the overlay text.

At the very end of the test, participants were presented with a *System Usability Scale*[3] questionnaire. The *U.S. Department of Health & Human Services* [17] describes this type of questionnaire as follows:

> The *System Usability Scale (SUS)* provides a "quick and dirty", reliable tool for measuring the usability. It consists of a ten item questionnaire with five response options for respondents; from Strongly agree to Strongly disagree. Originally created by John Brooke in 1986, it allows you to evaluate a wide variety of products and services, including hardware, software, mobile devices, websites and applications.

To be precise, the ten questions the *SUS* questionnaire contains are the following:

- I think that I would like to use this system frequently.
- I found the system unnecessarily complex.
- I thought the system was easy to use.
- I think that I would need the support of a technical person to be able to use this system.
- I found the various functions in this system were well integrated.
- I thought there was too much inconsistency in this system.
- I would imagine that most people would learn to use this system very quickly.
- I found the system very cumbersome to use.
- I felt very confident using the system.
- I needed to learn a lot of things before I could get going with this system.

The results of the questions showed that although the editors would not completely agree, that people are going to get used to the system very quickly, they were satisfied with the complexity of it and would not need the assistance of a technical person to be able to use the system. They did not think the system was unnecessarily complex or inconsistent. Only one participant chose something other than 'strongly disagree' at having to have learned a lot before being able to use the system. Overall, the authoring

[3] https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

system reached an average score of 77.5 according to the *SUS* grading scheme, with standard deviation of around 15 points. The median score was 77.5 as well.

6.2 Viewer Experience

This Section is centered around the user experience test for viewers using the implemented video player. As in the previous section, it will start by describing the participants and the target audience. Afterwards, the testing procedure and its limitations will be examined. In the end, the results of the test will be presented and discussed.

6.2.1 Participants

Unlike the editor experience test presented in Section 6.1, the viewer experience test placed much lighter restrictions on the pool of participants. Therefore, the participants of the test could be picked much more freely. Just like in the previous test, the video player was tested with five participants, as proposed by Nielsen [14], in order to find a suitable balance between testing effort and number of issues found. Again, some demographic data about the participants, as well as about their PC usage was gathered, although this test left out the participant's previous programming experience, as this was not relevant for their tasks in this test. This left the data with the following:

- hours of daily PC usage,
- preferred operating system,
- age.

Compared to the editor experience test, the first notable difference here is the number of hours spent in front of a PC per day. The graph depicting the amount of hours spent by viewers can be seen in Figure 6.8. The average amount of hours per person decreased by about three hours. This can be related to the fact that viewer experience test participants did not have to be an editor or have to work in a similar field. Consequently, those people might spend a lot less time in front of a PC, since they do not have to use one for their day job. In order to capture the point of view of both experienced and non-experienced PC users, the participants were chosen in such a way that three of them had an IT-dependent occupation and two of them did not, which is reflected in Figure 6.8 showing two participants in the lower bracket and three participants in the middle bracket.

Coincidentally, the distribution of operating systems of both participant groups is the same. This should not be a matter of concern though, since all three defined groups of operating system users are well populated, as can be seen in Figure 6.9. Operating system preferences were less important for the viewer experience test, since the video player usage does not require any keyboard shortcuts at all. Because the test setup was the same in both testing scenarios though, the data was still recorded in order to find any correlations between operating system preferences and video player test performance in the subsequent analysis.

As in the editor experience test, the participant's age was the last piece of potentially relevant data for the analysis. Similarly, no assumptions about correlations between age and test performance had been made, since actual PC usage was expected to have more of an impact on the performance, regardless of age. Nevertheless, the data was recorded

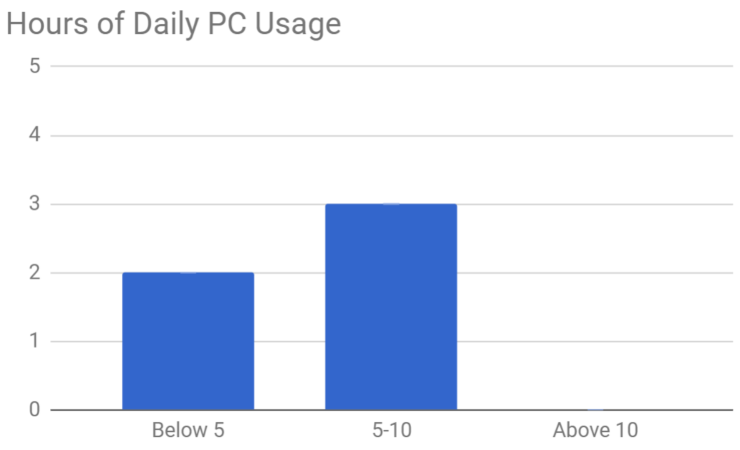


Figure 6.8: A graph depicting the daily amount of hours the viewers spend using a PC.

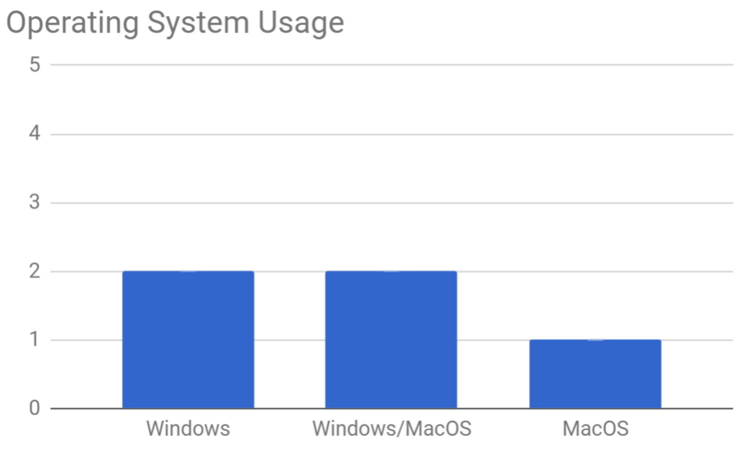


Figure 6.9: A graph depicting the participants' operating system preferences.

to make sure that possible correlations can be discovered.

6.2.2 Procedure

The primary objective of the viewer experience test was to determine whether the video player interface was similar enough to popular video players for viewers to recognize and utilize all or most of its features. Secondary objectives of the viewer experience test included:

- determining the usefulness of adaptive video quality settings,
- testing whether the overlays are interpreted correctly,
- checking for any unknown button icons within the control bar.

Unlike in the editor experience test, the participants of the viewer experience test had

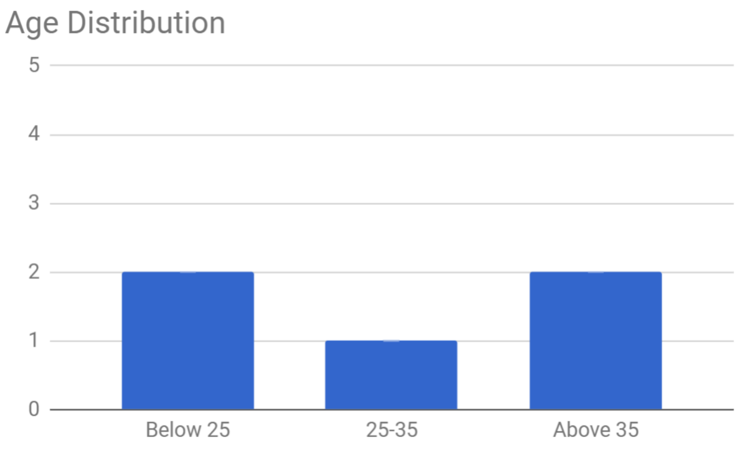


Figure 6.10: A graph depicting the age distribution of the viewer experience test.

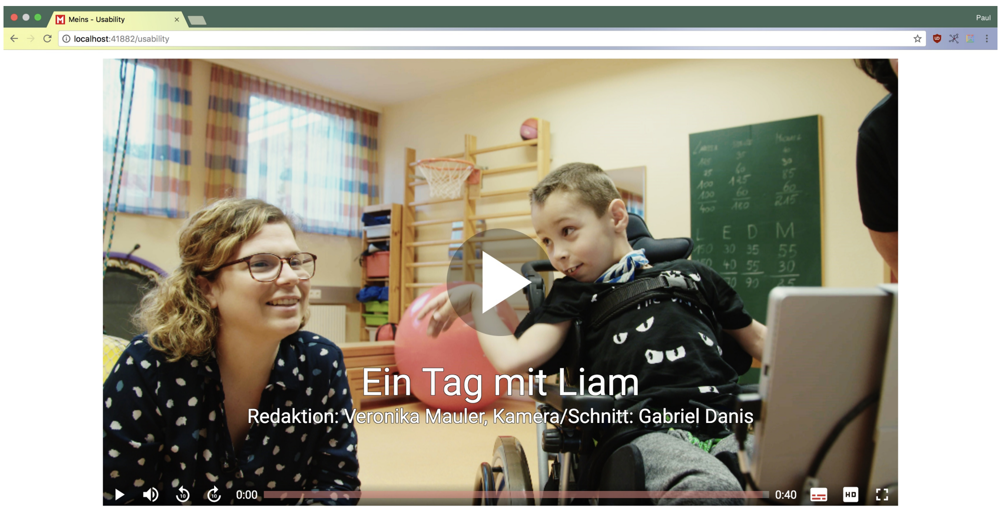

Meins - Usability
localhost:41882/usability
Paul
Ein Tag mit Liam
Redaktion: Veronika Mauler, Kamera/Schnitt: Gabriel Danis
0:00
0:40

Figure 6.11: The screen the viewers were presented with at the start of their test.

no preparation material to ease them into the matter, because it was important to gauge the first impression the video player makes on the viewers. Additionally, viewers were not expected to need any kind of introduction to operating a video player. Thus, the participants were presented the screen found in Figure 6.11 at the start.

The viewer experience test was split into four parts, each of which covered one of the goals mentioned above. In contrast to the other test, the parts were not exactly tasks the participants had to complete in order to be done with the part, but rather scenarios they had to immerse themselves in to answer questions about their opinions and behaviours afterwards. For example, the first task had the viewers watch 40 seconds of a recent video in order to familiarize themselves with the video player interface and its functionality. If the participants had any previous experience with video players, they were also encouraged to look for similarities and differences between their known video player and the one in front of them. After this test, they were asked whether the video player had fulfilled all their expectations and which expectations they had in the first place. In order to collect ideas for further improvement of the viewer side of the video player, participants were also asked whether they missed any kind of functionality within it.

The second scenario presented the setting that the viewer was not satisfied with the video quality the video player displayed currently. They were asked how they would react in such a situation and could either describe their approach verbally, or act it out with the support of the video player in front of them. This part was designed to determine whether viewers would know what to do if they wanted to watch the video in a higher quality, and whether they wanted the video player to automatically adapt to their network situation. Depending on the participants' answers, the questions they were asked included:

- Would you expect the video player to automatically adjust the video quality if your bandwidth allowed for it?
- Are there cases where you would not want this to happen, especially considering mobile data plans?
- Would you like to have more precise control over the quality of the video, like YouTube's options ranging from 144p to 4K?

The third scenario directed the viewers towards the overlay around the end of the sample video. Once the overlay was visible on their screen, just like in Figure 6.12, they were asked to describe what the rectangular, labeled shape constituted to them. If the participants stated that the element seemed clickable to them, they were asked what they expected to happen once they did so. They were also asked if they thought that presenting additional information to a topic in such a way was a good idea and whether they personally would click on such overlays during a video.

The fourth scenario asked the participants to name or describe every single element found on the control bar on the bottom side of the video player. This was done to determine whether the viewers would understand each of the many buttons and would understand when to use them. After describing the elements, the participants were asked whether they had any trouble with one of them. If they did, they were asked for the reason why they had trouble and possibly also a suggestion for a solution. Then, the participants were asked about their behaviour when encountering an unknown button to determine whether the video player could bank on the explorative minds of viewers when including lesser known icons and features. Since there were no questions to be asked to conclude the viewer experience test, it was finished after the collection of the demographic data from the participant.

6.2.3 Results and Discussion

Again, no tests had to be interrupted for any reason and no participant left the test feeling unsatisfied with the video player, so the general result of the test was a satis-

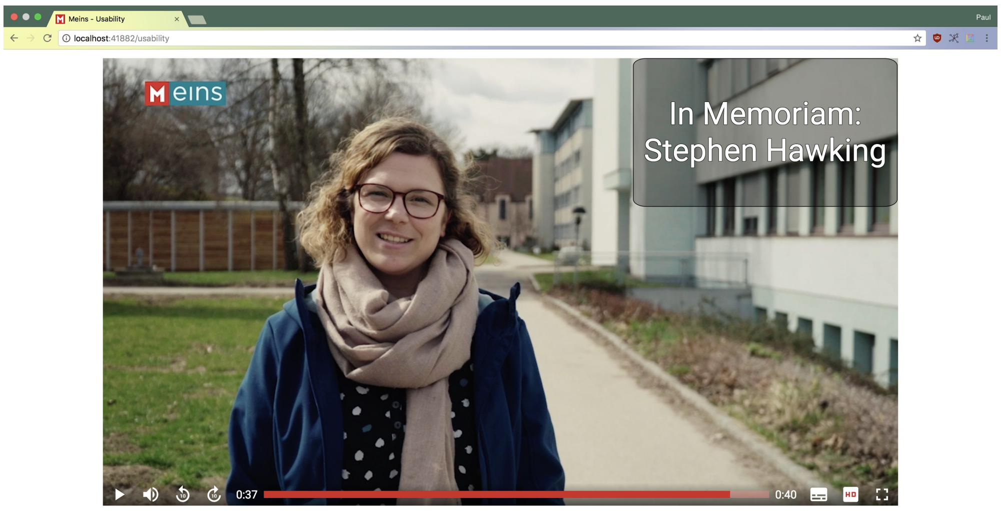


Figure 6.12: The screen showing the overlay to the viewers during their test.

factory one. The application handled the user interaction perfectly, so no user had to experience any part of the system failing. Overall, the participants described the video player as similar to its competitors from the likes of *YouTube* and *Netflix*, partly also because of its red colour theme. The theme was not chosen with the intention of creating similarity, though, but to align with company colours. The first set of questions revealed three cases in which the viewers would expect the video player to act differently. The first of those was the case of double clicking on the video screen. Here, the viewers would likely expect the video player to go into fullscreen mode, as many modern video players do. From a code perspective, this is a very simple piece of functionality to add, since it only requires an additional event listener to wait for a double click event, which then leads to a call of the fullscreen method.

The second case was one viewer missing the functionality of being able to quickly rewind or fast-forward the video using the arrow keys on the keyboard. Again, many current video players offer this functionality, so it makes sense to include it here as well. Because the video player is already listening to keyboard events in order to provide the play/pause functionality on a press of the space bar, this change does not require more than an `IF` statement and a method call.

The last case found through the first set of questions was the lack of a preview image attached to the progress bar. This is a rather complicated issue to tackle correctly, since the video player can currently handle complete video sources and streaming ones, which would likely have to be treated differently. A crude option would be to attach another, smaller video element to the progress bar, and having it display the video at the desired time stamp. This solution is concerning in at least one aspect, which is download efficiency. Once a part of the video is downloaded for one of the video elements, meaning the original video player and the small thumbnail player, it should not have to be downloaded again if the other video element needs to display it. It is unclear how different browsers go about caching video data, so this implementation might put additional stress on the viewers' internet connection.

The second set of questions was targeted at the means of video quality adjustment. Four out of five participants would correctly aim for the HD button on the control bar in case the video quality was not high enough. It might not be very common for people who might not concern themselves with video quality to look for an HD button nowadays, since many modern video players already have an adaptive quality option enabled. Therefore, a bad video quality might cause a poor user experience, because the issue is never resolved by the user.

Four participants answered that they want the video player to adjust the video quality automatically according to the available bandwidth. An opt-out possibility for the adaptive quality should be provided as well, though. This would be akin to *Twitch's* video player, which offers preset video options ranging from '160p' to '1080p', as well as an 'Auto' option to let the video player decide the best quality for the viewer. In theory, choosing the video quality automatically consists of determining the bandwidth of the viewer and choosing the quality option accordingly depending on how much data the video uses per second.

None of the participants had any concern with mobile data being used for higher quality videos, but one of them pointed out that the video player should remember whether a viewer has already opted out of the adaptive quality. This should be done via cookies, which are not much of a developmental issue, but rather a legal one nowadays. Lastly, the participants stand three to two against the implementation of multiple quality options instead of simply having HD and SD. This question was posed in order to gather data about the theory that viewers only use either the best available option or the worst. Some flaws can be seen with this assumption, though, so the divided answers of the participants are understandable. The theory has two requirements:

- The best option is acceptable for conventional bandwidths and does not heavily strain mobile data plans, so anything above 1080p is unsuitable.
- The worst option is acceptable for viewing at desktop screens, so anything below 360p is unsuitable.

As long as those two requirements are met, the current HD/SD system should be fine for the video player. The button design might still have to be adjusted to fit the adaptive quality option, though.

The third set of questions was targeted at video overlays and how the viewers perceived them. All of the participants interpreted the overlay as clickable, although one of them noted that it looked like an advertisement. That participant also suggested putting the overlays near the end of the video, as to not interrupt the video flow for the viewer. Overlays being interpreted as advertisements is definitely a concern, but one that might be fixed by sensible overlay usage by the editors. This includes:

- not cluttering the screen with multiple overlays,
- not showing an overlay for too long,
- not having an overlay for every element of the video,
- having a concise, descriptive text for all overlays.

With those factors in mind, overlays should not be interpreted as an annoyance by the viewers. If this still fails, adding an overlay-disabling button to the control bar should be considered.

The last set of questions was aimed at confirming viewer recognition of the control bar elements. Except for one mistake, all elements throughout all the tests have been recognized successfully. The one exception was caused by the subtitle button. It is unlikely that the recognition issue here can be solved by replacing the icon of the button, since it is already leaning on icons from *YouTube's* and *Netflix's* video players. A good solution likely consists of common icons, which are already present, combined with tooltips explaining a button when the viewer hovers over it. Four out of five participants were positive that tooltips would help viewers understand what possibly unknown buttons do without having to try them out. Two of them advised against overusing the tooltip feature, though, since tooltips on commonly understood buttons are going irritate viewers more than they would educate. Lastly, only three participants stated that they would click on an unknown button before knowing what it does, which paints a gloomy picture for the strategy of banking on the viewers' explorative minds.

6.3 Opportunities for Further Improvement

To conclude the evaluation chapter, all issues that have been discovered should be recapped alongside their proposed solutions. This Section is going to be divided into four parts, describing the individual changes in ascending order according to the estimated effort the change is going to require.

6.3.1 Minor Changes

Minor changes are issues of the video player that should be fixable within less than an hour each. These changes include:

- activating fullscreen when the viewer double clicks on the video player,
- binding rewind and fast-forward functionality to the arrow keys,
- detecting the URL extensions of overlay links,
- storing volume and video quality preferences in cookies,
- adding tooltips to possibly unknown buttons,
- implementing a better distinction between link overlays and video overlays.

Going down the list from top to bottom, we can see that the first two fixes are very easy to implement. Fullscreen, rewind, and fast-forward functionality already exists and only has to be bound to additional user actions. Effectively, it is like adding an event listener to the correct user event and only calling the corresponding player component method within it.

Detecting the URL extensions of overlay links should be done in order to relieve the work load of editors creating overlays. They should not have to think about whether they are linking to a video or to a website, when the overlay system can easily do that for them. This change involves retrieving the ending of the given URL, matching it to a few video file extensions and making the system react accordingly, so it is a very manageable task.

Storing volume and video quality preferences in cookies is important in order to let the viewers customize their video player experience better. If the video player remembers the users' preferences, they no longer have to experience a bad start to a video because of inappropriate standard settings. This task involves setting up a preference data structure, which is automatically stored in a cookie when users make a change to their preferences. The cookie preferences have to be loaded into the video player when it is loaded, but not more often than that. If not already present on the website, this task might also include setting up a cookie disclaimer in order to adhere to legal restrictions.

Adding tooltips to possibly unknown buttons might appear to be less work than the cookie task, but anything that is going to be displayed to the viewer should be done with extra care for mobile and desktop experiences at different resolutions and orientations. In theory, a simple `div` element placed above the button and filled with a suitable, concise description should suffice. In practice, developers have to account for font scaling, edges of the video player and possible line breaks within the description as well. Additionally, having text within the description is going to open the player up to localization problems.

Implementing a better distinction between link overlays and video overlays can be very hard or very easy, depending on the finesse that is going to be put into the final solution. A crude fix would just add an obligatory text like 'Link:' or 'Video:' to the start of the overlay text, so users can clearly see what they are going to receive when they click on it. As before, having text within the video player can open it up to localization problems, although 'Link:' and 'Video:' do not pose a threat in an English/German environment. A more refined option could shape the overlays a little differently, or display a thumbnail of a video in the background of the overlay. Ultimately, a great part of this task is finding a suitable solution to the problem.

6.3.2 Adaptive Quality

Adaptive quality has become a staple of modern web video players, so it should certainly not be missing from this one. The previous Section outline the theory about an adaptive quality setting, which consists of determining the bandwidth of the viewer and choosing the quality option accordingly depending on how much data the video uses per second.

In practice, determining the viewer's bandwidth likely puts additional strain on the internet connection, because such a speed test requires data as well. To elaborate, such a speed test usually takes a file of a known size on a reliable, possibly external file server, and sends a request for that file from the user's web browser, while simultaneously starting a timer. Once the file is fully downloaded, the file size of it is divided by the amount of seconds it took to download it. This results in an approximation of the user's bandwidth. Naturally, the downloaded file is useless to the user, so it is simply disposed of and the data has been transferred without much benefit. Fortunately, there is a *Network Information API*, which can provide developers with information about a user's internet connection without the need for cumbersome speed tests. Unfortunately, the *Network Information API* is currently only supported by *Chrome*, mobile *Firefox*, mobile *Opera*, and *Android Webview* [23], so fallback options have to be implemented regardless.

6.3.3 Progress Bar Preview

The issue of the progress bar preview has been discussed a bit in the previous section, but will be explained in further detail here. The problem is that modern video players offer a preview image to users when they hover over the progress bar. This is a feature which is not easy to provide in an elegant way. The crude proposal made above suggests adding another, smaller video element to the progress bar, which loads the video at the desired time, pauses and displays the image. While this would probably work fine, it could create additional strain on the viewers' internet connection, since web browsers might not realize the two video elements are loading the same video and that they should recycle the video data for both of them.

Another option would be to create an array of thumbnails for each video, so a fairly accurate thumbnail can be displayed to the user. This would decrease loading times for clients, but require video processing time and resources on the video server. The video player would thus lose the functionality of displaying any video fed to it. Alternatively, viewers might be satisfied with seeing the timestamp they are going to arrive at, when they click the progress bar at a certain position. This would be akin to the minor fix of adding tooltips to buttons, although the tooltip should move along the progress bar to follow the cursor.

6.3.4 Authoring Tool

Although the editors handled the JSON notation reasonably well in their user experience test, they agreed that they would rather work with an authoring tool. Such an authoring tool would lift the responsibility of writing JSON from the editors, while providing them with a more intuitive way of producing overlays. A first draft of how the interface of such a tool could look like can be seen in Figure 6.13. Such a tool does come at a price, though, since it combines a lot of features to provide its functionality. These are:

- video input and display,
- overlay input and parsing,
- JSON export,
- timelines.

Fortunately, most of the features are rather straightforward to implement. A first version of video input and display can be extracted from the video player, but will have to be extended with some functionality to create, delete, and fill overlays with a click and keyboard input. Hover effects will be helpful in letting the editors know what clicks are going to do at all times. Ideally, the file is loaded from the file system, because this minimizes the possibility of availability and buffering issues.

Overlay input and parsing is important because editors will regularly want to provide an existing overlay file to start the process off. The system can almost completely be copied from the video player, although the parsed data will have to be stored in a modifiable data store, so changes can be made during the authoring process.

JSON export is a cornerstone of the authoring tool, without which it would not be usable. If the overlay data structure is built correctly in background while the editors continually update the overlay specification, the step is not necessarily complicated. It boils down to directing the editor's browser to an URL which makes it download the

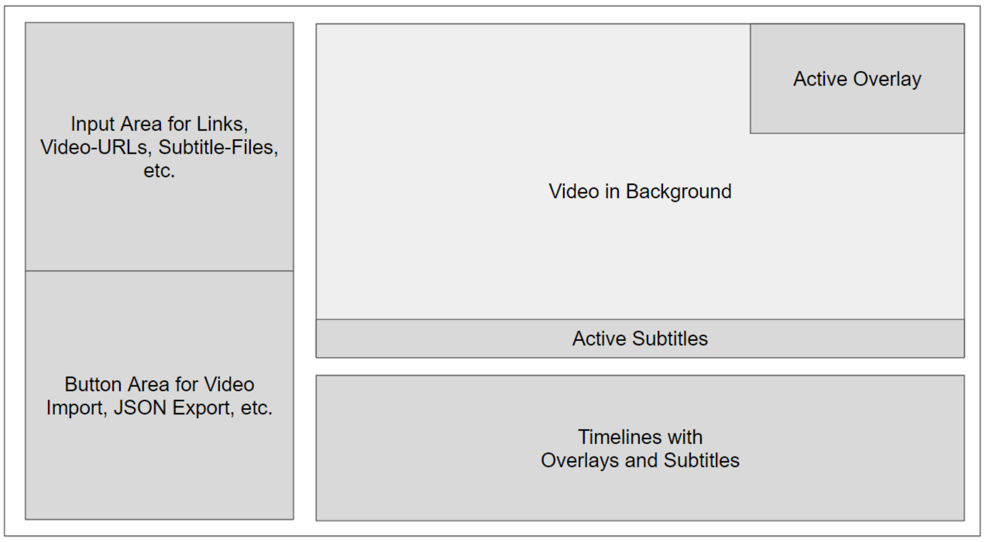

Input Area for Links, Video-URLs, Subtitle-Files, etc.
Button Area for Video Import, JSON Export, etc.
Active Overlay
Video in Background
Active Subtitles
Timelines with Overlays and Subtitles

Figure 6.13: A first draft of the authoring tool interface.

content of the data structure in JSON format. The editors then only have to store the file on a server, make it publicly available and provide the video player with the URL.

Timelines might be the most complex area of the authoring tool. They should show the editors at one glance which parts of the video contain subtitles or overlays and how many of them there are. Therefore, several timelines should be stacked on top of each other to be able to better distinguish multiple overlays. Colour coding can also help increase distinguishability, so those options might have to be used in tandem to display the overlay situation in the worst case of having nine overlays at once.

Overall, the authoring tool might take a while to produce and should be rigorously tested for usability issues as early as possible. In the end, it will surely be worth the effort, as it provides a much more reliable and enjoyable overlay production experience for editors. This will likely lead to more overlays being produced, which means a more fleshed out video experience for the viewers. In a way, both parties profit from an improved authoring system.

Chapter 7

Conclusion

Web video content is very well supported by modern browsers. Not only have successful companies like *YouTube* and *Netflix* been able to build very sophisticated video players, development has been greatly simplified by the introduction of HTML5 and its video element. With the right set of tools and programming experience, video player development is merely a matter of time one is willing to invest. To a large extent, this can be attributed to the standards provided by the *W3C*[1], which developers can rely on when developing for multiple browsers.

Talking about modern web standards, the *React* framework has also helped tremendously in building the video player with a modular architecture in mind. Its components can easily be switched out or even deactivated, like the subtitle button when there are no subtitles. Through adding additional functionality to the core Player component via other components and modules, the application was able to grow in a controlled and maintainable manner.

Concerning user experience, a lot of insight could be gathered from the current state of the art. While the general user experience of modern web video players is very good, many interactive video solutions have foregone concerns about the user experience for video editors. The issue here is, that the developers of those solutions expect their users to be developers as well. So a less programming-reliant way of defining interactible elements was implemented and tested. Despite the tests showing that the implementation was a success, using JSON seemed like a less than ideal solution. Concerning viewer experience, the tests have shown that users understand and like new methods of interaction within a video, as long as they are not bombarded with additional content they might have no interest in. The only issues with the video player itself were related to features the viewers have come to expect from other websites.

7.1 Outlook

As described in Chapter 6, the video player is still missing a few features that should be added before it can be used in production. Additionally, once the features are added and any issues have been fixed, the video player and the corresponding authoring system should be tested once again. After all, the user experience tests were reduced in group

[1]https://www.w3.org/

size in order to be able to conduct more of them. Despite the persisting issues, the *ORF* is pushing to have the player on one of its pages in the near future. The code will be given over to the *ORF*, so it will likely not become open source. Regardless, there is hope that the video player is going to be maintained well by some of their talented developers.

With the recent rise of 4K monitors, web video content providers are facing more difficulties as their viewers demand higher quality videos to utilize their screens to the fullest extent. This, combined with the lack of significant interactive web video news, leads to the belief that web video is not currently planning to go down the interactive path.

On the other hand, many gaming platforms are offering interactive video experiences recently. *Late Shift*[2] is one of many interactive movies offered on the game distribution platform *Steam*. Perhaps producers have found interactive movies to be much more akin to games than they are to movies, and are adjusting their distribution channels accordingly.

[2] http://lateshift-movie.com/

References

Literature

[1] Dani Cavallaro. *Anime and the visual novel: narrative structure, design and play at the crossroads of animation and computer games.* McFarland, 2009, pp. 7–8 (cit. on p. 8).

[2] Jasmine Cox et al. “Object-Based Production: A Personalised Interactive Cooking Application”. In: *Adjunct Publication of the 2017 ACM International Conference on Interactive Experiences for TV and Online Video.* ACM. 2017, pp. 79–80 (cit. on p. 13).

[3] James Dargie. “Modeling techniques: movies vs. games”. *ACM SIGGRAPH Computer Graphics* 41.2 (2007), pp. 2–3 (cit. on p. 7).

[4] Erich Gamma et al. “Design patterns: Abstraction and reuse of object-oriented design”. In: *Proceedings of the European Conference on Object-Oriented Programming.* Springer. 1993, pp. 406–431 (cit. on pp. 15–17).

[5] David P Gregg. *Disc-Shaped Member.* US Patent 4,893,297. Jan. 1990 (cit. on p. 6).

[6] David Paul Gregg and Keith O Johnson. *Video signal transducer having servo controlled flexible fiber optic track centering.* US Patent 3,530,258. Sept. 1970 (cit. on p. 6).

[8] Jakob Nielsen and Thomas K. Landauer. “A Mathematical Model of the Finding of Usability Problems”. In: *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems.* CHI ’93. Amsterdam, The Netherlands: ACM, 1993, pp. 206–213 (cit. on p. 35).

Online sources

[9] Dan Abramov and Andrew Clark. *Redux - Three Principles.* [Online; accessed 23-April-2018]. 2018. URL: https://redux.js.org/introduction/three-principles (cit. on p. 16).

[10] Kevin Calhoun et al. *Standardized multimedia elements in HTML5.* 2007. URL: https://www.w3.org/2007/08/video/positions/Apple.pdf (cit. on p. 19).

[11] Alena Činčerová. *Kinoautomat.* [Online; accessed 03-January-2018]. 2010. URL: http://www.kinoautomat.cz/index.htm?lang=gbr (cit. on pp. 1, 4).

[12] Derral Eves. *How To Use YouTube End Screen Editor for Your Videos.* [Online; accessed 09-April-2018]. 2016. URL: https://www.youtube.com/watch?v=Q4dYiyCg6q8 (cit. on p. 12).

[13] Fyrd. *Can I use... - Browser support tables for modern web technologies.* [Online; accessed 28-April-2018]. 2018. URL: https://caniuse.com (cit. on p. 23).

[14] Nielsen Norman Group. *Why You Only Need to Test with 5 Users.* [Online; accessed 20-May-2018]. 2000. URL: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (cit. on pp. 35, 36, 45).

[15] H5P. *H5P Tutorial.* [Online; accessed 02-April-2018]. 2018. URL: https://h5p.org/tutorial-interactive-video (cit. on p. 9).

[16] H5P. *H5P Website.* [Online; accessed 02-April-2018]. 2018. URL: https://h5p.org (cit. on pp. 9–11).

[17] U.S. Department of Health & Human Services. *System Usability Scale (SUS).* [Online; accessed 27-May-2018]. 2018. URL: https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html (cit. on p. 44).

[18] Mended Little Hearts. *Give A Fuller Life.* [Online; accessed 04-January-2018]. 2017. URL: http://www.giveafullerlife.com/ (cit. on p. 13).

[19] University of Illinois at Urbana-Champaign. *YouTube History.* [Online; accessed 05-April-2018]. 2009. URL: https://web.archive.org/web/20090111223210/http://www.cs.uiuc.edu/news/articles.php?id=2006Feb3-126 (cit. on p. 11).

[20] Demian Katz. *Gamebook FAQ.* [Online; accessed 12-March-2018]. 2018. URL: https://gamebooks.org/FAQs (cit. on pp. 4, 6).

[21] Rubenstein Library. *Consider the Consequences.* [Online; accessed 03-January-2018]. 2017. URL: https://rubensteinlibrary.tumblr.com/post/165617259862/consider-the-consequences-by-doris-webster-and (cit. on pp. 1, 4).

[7] Jose María Martínez Moreno et al. *Assessing a cognitive rehabilitation environment based on interactive video and eye-tracking technologies.* 2015. URL: http://oa.upm.es/41803/1/INVE_MEM_2015_222344.pdf (cit. on p. 13).

[22] Mozilla and individual contributors. *Media formats for HTML audio and video.* [Online; accessed 28-April-2018]. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats (cit. on p. 22).

[23] Mozilla and individual contributors. *Network Information API.* [Online; accessed 08-June-2018]. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API (cit. on p. 52).

[24] Numberphile. *400 and Fighting Fantasy.* [Online; accessed 19-March-2018]. 2012. URL: http://www.numberphile.com/videos/400_fighting_fantasy.html (cit. on p. 6).

[25] Universal Pictures. *Warcraft.* [Online; accessed 20-March-2018]. 2016. URL: https://www.uphe.com/movies/warcraft (cit. on p. 7).

[26] Cambridge University Press. *Definition of Interaction in the Cambridge Dictionary.* [Online; accessed 12-March-2018]. 2018. URL: https://dictionary.cambridge.org/dictionary/english/interaction (cit. on p. 3).

[27] SnapApp. *SnapApp Showcase*. [Online; accessed 05-January-2018]. 2017. URL: https://www.snapapp.com/platform/interactive-content-types/interactive-video (cit. on p. 5).

[28] INC. SQUARE ENIX. *Life is Strange*. [Online; accessed 12-March-2018]. 2018. URL: https://www.lifeisstrange.com/en-us/games/life-is-strange (cit. on p. 5).

[29] Incorporated Telltale. *Telltale Games*. [Online; accessed 12-March-2018]. 2018. URL: https://telltale.com (cit. on p. 5).

[30] YouTube. *YouTube Annotations*. [Online; accessed 05-April-2018]. 2018. URL: https://support.google.com/youtube/answer/7342737?visit_id=1-636585357341677209-1833817765&p=keep_fans_engaged&rd=1&hl=en (cit. on pp. 1, 11).